Kd-tree Based N-Body Simulations with Volume-Mass Heuristic on the GPU

Klaus Kofler*, Dominik Steinhauser †, Biagio Cosenza *, Ivan Grasso *, Sabine Schindler † and Thomas Fahringer *
* *Institute of Computer Science, DPS, University of Innsbruck, Austria*
Email: [klaus|cosenza|grasso|tf]@dps.uibk.ac.at
† *Institute of Astro- and Particle Physics, University of Innsbruck, Austria*
Email: [Dominik.Steinhauser|Sabine.Schindler]@uibk.ac.at

***Abstract*—N-body simulations represent an important class of numerical simulations in order to study a wide range of physical phenomena for which researchers demand fast and accurate implementations. Due to the computational complexity, simple brute-force methods to solve the long-distance interaction between bodies can only be used for small-scale simulations. Smarter approaches utilize neighbor lists, tree methods or other hierarchical data structures to reduce the complexity of the force calculations. However, such data structures have complex building algorithms which hamper their parallelization for GPUs.**

In this paper, we introduce a novel method to effectively parallelize N-body simulations for GPU architectures. Our method is based on an efficient, three-phase, parallel Kd-tree building algorithm and a novel volume-mass heuristic to reduce the simulation time and increase accuracy.

Experiments demonstrate that our approach is the fastest monopole implementation with an accuracy that is comparable with state of the art implementations (GADGET-2). In particular, we are able to reach a simulation speed of up to 3 Mparticles/s on a single GPU for the force calculation, while still having a relative force error below 0.4% for 99% of the particles. We also show competitive performance with existing GPU implementations, while our competitor shows worse accuracy behavior as well as a higher energy error during time integration.

***Keywords*-N-body, GPGPU, Kd-tree;**

I. Introduction

In history, understanding the motion of celestial objects under their mutual gravitational attraction, motivated to search for a solution to the N-body problem. Newtonian attraction forces between each pair of bodies lead to their acceleration and hence collective motion. The goal is to predict future positions and velocities for all bodies (often called particles in this context) starting from a given initial state. The corresponding differential equations to this initial value problem can be solved analytically only for $N \leq 3$. Larger simulations can only be calculated numerically.

The challenge in N-body codes is to find those mutual gravitational attraction forces. The simplest way to evaluate the force acting on a single particle is by summing the contribution from all the other particles, called *direct summation* approach. However, this rather brute-force method is of order $\mathcal{O}(N^2)$ and is hence only a viable option for small problems.

Particle-mesh codes, as described in [1] are more efficient than direct summation. However, close particle interactions are not well modeled. Hybrid approaches such as P^3M [2] overcome this problem.

Tree codes: When calculating the gravitational force contribution of a reasonably distant group of close bodies on a single particle, this group can be approximated with a single, more massive proxy-body. To add information about the distribution of the particles inside this proxy-body, higher order moments of the gravitational potential's multipole expansion of the particle group can be used. This observation is exploited in the so-called tree code approach to the N-body problem [3]. Tree codes make use of space-partitioning data structures to recursively divide up the simulation domain in sub-volumes. While Barnes&Hut [3] used octrees, we adopt Kd-trees in our work: Each volume is split in two sub-volumes according to a splitting plane. The leaves of the tree contain just one single particle. Thus, for each node the potential in terms of a multipole expansion is calculated. When calculating the force contribution for a single particle, a tree traversal is done. In case a tree node, representing a part of the simulation domain, is reasonably remote from this particle, the approximate potential in this node can be used to calculate the force contribution of all particles contained within this node. Hence, the subtree of this node does not need to be considered anymore. The *cell opening criterion* [3] defines whether the tree walk can be stopped at the current node, using this node as proxy body, or the descent is continued further in the tree to calculate the force more accurately.

GPU: Being a very computational intensive application, N-body simulations have been historically interesting for high performance computing. Recently, even Graphics Processing Units (GPUs) have been exploited for this task. However, while *direct summation* approaches are quite easy to be run on GPUs, more advanced hierarchical methods are very challenging. The use of intricate data structures, their traversal and in particular building them is a task which makes it hard to exploit the massive parallelism offered

by such hardware. Three factors are critical for N-body simulations based on hierarchical methods on GPUs: 1) to run the whole algorithm on the GPU in order to avoid expensive communication bottlenecks due to CPU calculation of intermediate steps; 2) a fast traversal algorithm for the hierarchical data structure; 3) a fast building algorithm for such a data structure.

Accuracy: For all hierarchical methods, the accuracy of simulations also plays an important role: clustering the particles aggressively may lead to impressive performance improvement, but the obtained results lack accuracy. Not just the traversal, but also the layout of the hierarchical data structure affects accuracy.

Our contribution: In this paper, we introduce a novel method to accurately and efficiently calculate the gravitational forces, the computationally most expensive part of N-body simulations, on the GPU. Our contributions are:

- a novel hierarchical method for calculating gravitational forces based on a Kd-tree on a GPU;
- a probabilistic approach based on volume-mass heuristic (VMH) to efficiently group particles in a Kd-tree and drastically improve the accuracy and efficiency of the Kd-tree traversal;
- a parallel and accurate approach to efficiently build the Kd-tree on the GPU: by using a three phase building algorithm, we maximize the thread utilization of the GPU during the building in both top- and bottom-part of the Kd-tree;

Outline: Our N-body method consists of three parts: the tree building, the tree traversal, and the time integration. A parallel implementation of the Kd-tree building algorithm for the GPU is presented in Section III. Section IV introduces our probabilistic approach called volume-mass heuristic (VMH). The tree traversal and the force calculation are discussed in Section V, the time integration in Section VI. Performance and accuracy are presented and discussed in Section VII. Section VIII concludes the paper.

II. Previous Work

N-body simulations: Historically, many researchers from both computer science and astrophysics have developed N-body simulations on supercomputers. Warren and Salmon [4] designed one of the first parallel implementation of the Barnes&Hut algorithm. The authors of [5] propose a parallel implementation of an N-body code using a Kd-tree structure.

A very widespread code used in astrophysics, mainly for cosmological simulations and simulations on galaxy scales, is the treePM code GADGET [6], [7]. This code implements a combination of a particle-mesh and a Barnes&Hut tree code, massively parallelized for distributed memory machines using MPI.

Nyland et al. [8] implemented a *direct summation*, brute-force technique. They improved their code by means of loop unrolling and by manually prefetching a certain number p of body descriptions on the GPU. Elsen et. al [9] created a similar solution using the BrookGPU programming language [10].

The Gravity Pipe (GRAPE) [11] designated a very efficient hardware implementation of Newtonian pair-wise force calculations between particles in a self-gravitating N-body system.

Instead of direct summation, smarter approaches to attack the N-body problem use hierarchical algorithms and three dimensional space partitioning strategies. Hamada et al. [12] implemented a parallel, hierarchical N-body simulation which efficiently calculates the $\mathcal{O}(N \log N)$ tree code and $\mathcal{O}(N)$ fast multipole method (FMM) on multiple GPUs. Hamada and Nitadori reached 190 TFlops [13] on DEGIMA, a cluster of 576 GPUs interconnected by InfiniBand, using their tree code.

The parallel cosmological simulator ChaNGa [14] is a hierarchical N-body gravity solver written in CHARM++, which has been run on the NCSA Lincoln GPU cluster.

2HOT [15] is a N-body simulation code based on a parallel hashed octree algorithm. It is designed to run efficiently on up to 256k processors with an efficiency of 0.86. 2HOT also includes a CUDA, as well as an OpenCL version for the gravitational interaction functions.

Bédorf et al. [16] implemented Bonsai[1], a sparse octree gravitational N-body code that runs entirely on the GPU. In contrast to most other tree codes, Bonsai traverses the tree breadth-first to calculate the force acting on each particle.

Probabilistic Heuristics for Hierarchy Building: An approach to improve the traversal time of hierarchical data structures is to use a probabilistic heuristic while building it: the higher the probability of a node to be accessed, the higher it will be placed in the hierarchy. Similar heuristics have been largely used in the context of ray tracing, known as SAH (Surface Area Heuristics). Wald et al. [17] introduced an algorithm to build SAH-based Kd-trees in $\mathcal{O}(N \log N)$. Other approaches using SAH have been used for BVH (bounding volume hierarchy) and other hierarchical approaches [18]. Zhou et al. [19], in particular, presented an algorithm for constructing Kd-trees on GPUs. They achieve real-time performance by exploiting the streaming architecture of modern GPUs at all stages of Kd-tree construction. They develop a special strategy for large nodes at upper tree levels to further exploit the fine-grained parallelism of GPUs. Our work applies a similar method to build a tree for N-body simulations.

III. Parallel Kd-tree building

Our Kd-tree building algorithm is designed to perform well on modern GPUs. This means that it has to expose a large amount of parallel operations. To exploit the full

[1]Version 2, http://castle.strw.leidenuniv.nl/software/bonsai-gpu-tree-code.html

potential of a modern GPU, several thousands of threads must run concurrently at the same time. The program was implemented in OpenCL. Our highly parallel implementation is inspired by the algorithm presented in [19] and consists of three phases:

- Large node phase
- Small node phase
- Kd-tree output phase

The large node phase takes place at the beginning of the tree construction, where only few nodes, containing many particles, exist. To increase the degree of parallelism, both, inter- and intra-node parallelism are exploited during this phase. In the small node phase, many nodes are handled at a time. Therefore, it is better to avoid the additional synchronization overhead introduced by the intra-node parallelism and rely on inter node parallelism only in this phase.

A pseudo code representation of our implementation is shown in Algorithm 1. It shows, that the implementation is split into four loops. The first loop represents the so called large node phase, the second the small node phase while the last two perform the up pass and the down pass to sort the tree nodes in depth first ordering. All iterations of these four main loops in our implementation have to be executed sequentially. Therefore, these loops cannot be used to exploit parallelism. However, as explained in the following paragraphs, there are several possibilities for parallelization inside those loops.

Algorithm 1 Kd-tree construction.

```
function BUILDKDTREE(particles:list)
    nodelist ← new list()
    activelist ← new list()
    nextlist ← new list()
    smalllist ← new list()
    rootnode ← new node(particles)
    rootnode.offset ← 0
    nodelist.add(rootnode)
    activelist.add(rootnode)
    while !activelist.empty() do                    ▷ large node step
        PROCESSLARGENODES(nodelist, activelist, nextlist,
smalllist, particles)
        activelist ← nextlist
    end while
    activelist ← smalllist
    while !activelist.empty() do                    ▷ small node step
        PROCESSSMALLNODES(nodelist, activelist, nextlist, par-
ticles)
        activelist ← nextlist
    end while
    treeHeight ← max level of all nodes in nodelist
    for level ← treeHeight to 0 do
        UPPASS(nodelist, particles, level)
    end for
    tree ← new list()
    for level ← 0 to treeHeight do
        DOWNPASS(nodelist, tree, level)
    end for
end function
```

Large node phase: In the large node phase, all large nodes are split in two child nodes in the middle of their longest dimension. Their particles are distributed to the children depending on their position. This step is repeated until no more large nodes are left. A large node is defined as a node containing at least 256 particles. In this phase, the inter node parallelism is maximized, e.g. by reductions in local memory and parallel prefix scans which are both known to perform well on GPUs [20]. While the reductions in local memory are used to accelerate the bounding box calculation, parallel prefix scans are needed to calculate the position of particles in the particle array in parallel after a node is split. The application of the afore mentioned techniques introduces several global synchronizations due to data dependencies. However, the overhead introduced by additional synchronization is outweighed by the increase of parallelism. Furthermore, in this phase the node splitting decision is designed not to be affected by the number of particles inside the node in order to scale to bigger data-sets. Algorithm 2 depicts the large node phase implementation. It is composed of six parallel loops, each of which is implemented as a separate OpenCL kernel function.

Distributing the particles of a parent node to its two child nodes, is the most time consuming part of the large node phase, since it requires rearranging of the particles of the parent node. When building a Kd-tree, this can be done only after selecting the splitting point, since the particles have to be partitioned according to the splitting point along the splitting dimension. The particles don't have to be sorted, but all particles belonging to the left child have to be at the beginning of the parent's node particle sub-array, while all the particles belonging to the right child have to be moved to the end of that array. This can be done in linear time in each timestep. When executing our implementation on a CPU, one OpenCL thread is started for each active node which assigns the particles to the child nodes in a sequential loop. This approach works well for CPUs. However, it does not expose enough parallelism to reach a good performance on GPUs, since there are not many active nodes in this phase. Therefore, we use a parallel prefix scan to determine for each particle its index in the particle list of the left and right child, respectively. Using the result of the prefix scan, the particles can be inserted into the particle lists of the two child nodes in parallel.

Small node phase: When no more large nodes are left, the program enters the small node phase. In this phase we aim at reducing the synchronization overhead (it needs only one synchronization at the end of each iteration) by starting only one single thread per active node. Increasing the parallelism any further would not improve the performance, since the number of active nodes is very high in most iterations. In order to improve the quality of the tree, this phase uses a splitting strategy based on VMH as described

Algorithm 2 Large Node Phase

```
function PROCESSLARGENODES(nodelist:list, activelist:list, nextlist:list, smalllist:list, particles:list)
    chunklist ← new list()
    ▷ group particles to chunks
    for all n in activelist in parallel do
        Group all particles in n into fix sized chunks and store them in chunklist
    end for
    ▷ calculate per-chunk bounding box
    for all c in chunklist in parallel do
        Compute bounding box for each chunk c
    end for
    ▷ calculate per-node bounding box
    for all n in activelist in parallel do
        Compute bounding box for each node n using the bounding boxes of the chunks
    end for
    ▷ split large nodes
    for all n in activelist in parallel do
        set n.splittingPoint to the spatial median along the longest dimension
        Split node n at n.splittingPoint
        Store generated child nodes in nextlist
    end for
    nodelist.add(nextlist)          ▷ add all new nodes to nodelist
    ▷ sort particles to children
    for all n in activelist in parallel do
        for all p in n.particles do
            if p.pos[splitDim] < n.splittingPoint then
                n.leftChild.particles.append(p)
            else
                n.rightChild.particles.append(p)
            end if
        end for
    end for
    ▷ small node filtering
    for all n in nextlist in parallel do
        if n is small node then
            smalllist.add(n)
            nextlist.remove(n)
        end if
    end for
end function
```

Algorithm 3 Small Node Phase

```
function PROCESSSMALLNODES(nodelist:list, activelist:list, nextlist:list, particles:list)
    ▷ split small nodes
    for all n in activelist in parallel do
        ▷ calculate VMH
        VMH ← ∞
        for all potential splitting points sp of node n do
            VMH_sp ← CALCVMH(n, sp)
            if VMH > VMH_sp then
                VMH ← VMH_sp
                n.splittingPoint ← sp
            end if
        end for
        Split node n at n.splittingPoint
        Store generated child nodes in nextlist
        nodelist.add(nextlist)      ▷ add all new nodes to nodelist
        ▷ sort particles to child nodes
        for all p in n.particles do
            if p.pos[splitDim] < n.splittingPoint then
                n.leftChild.particles.append(p)
            else
                n.rightChild.particles.append(p)
            end if
        end for
        ▷ Leaf node filtering
        for all n in nextlist do
            if n.particles.size = 1 then
                nextlist.remove(n)
            end if
        end for
    end for
end function
```

in Section IV. In our environment, each particle inside a node introduces one splitting candidate (along the node's longest dimension). The VMH cost has to be evaluated on each splitting candidate, which makes this strategy infeasible for large nodes. After the split, the particles of the parent node are assigned to the two child nodes, depending on their position. Each node is split according to the candidate giving minimal VMH, until the leaf nodes, containing only one single particle, are reached. The implementation of this phase is shown in Algorithm 3. It is composed of one parallel loop which is mapped to an OpenCL kernel function. Due to the typically high number of active nodes during this phase, the splitting of nodes into chunks is not necessary. Starting one OpenCL thread for each active node is sufficient to keep all processing elements of a GPU busy.

Kd-tree output phase: During the first two phases, new nodes are added to the nodelist in the same order as they are created, which means, they are not sorted. In order to enable a efficient tree walk, the nodes are ordered in a depth-first manner, which is done in the last phase of our tree construction. To sort the nodes of the Kd-tree two passes have to be performed: First a bottom-up pass which calculates the center of mass and the mass of each node (which corresponds to the proxy-body in the node or the potential's monopole moment) as well as the size of the subtree underneath it. The size of the subtree is important in order to calculate the actual position of a node in the final tree. The implementation of this pass is described in Algorithm 4. The second pass is building the final tree top down. The root is written at the beginning of the array. For each node at position i, the left child will be written to position $i+1$ and the right child to position $i+1+sizeof(leftChild)$. Sorting the nodes in that way, a linear traversal of the node array is equal to a depth-first traversal of the tree. A detailed description of this pass can be found in Algorithm 5.

Algorithm 4 Up pass

```
function UPPASS(nodelist:list, particles:list, position:int)
    for all n in nodelist at level position in parallel do
        if n.isLeaf then
            n.size ← 1
            n.mass ← n.particles[0].mass
            n.centerOfMass ← n.particles[0].pos
            n.l ← 0
        else
            n.size ← n.leftChild.size + n.rightChild.size + 1
            n.mass ← n.leftChild.mass + n.rightChild.mass
            n.centerOfMass ← (n.leftChild.centerOfMass
·n.leftChild.mass + n.rightChild.centerOfMass
·n.rightChild.mass) / n.mass
            n.l ← maximum side length of n.boundingBox
        end if
    end for
end function
```

Algorithm 5 Down pass

```
function DOWNPASS(nodelist:list, tree:list, position:int)
    for all n in nodelist at level position in parallel do
        if !n.isLeaf then
            n.leftChild.offset ← n.offset + 1
            n.rightChild.offset ← n.offset + 1 + n.leftChild.size
        end if
        tree[n.offset] ← n
    end for
end function
```

IV. Volume-Mass Heuristic (VMH)

When analyzing the requirements of an optimal Kd-tree for an N-body simulation, we determined that they are very similar to the requirements for ray-tracing. In both cases,it is not really important that the tree is balanced, but the average path length of the walks through the trees are minimized. However, there is a slight difference between those two applications. In ray-tracing each ray walks from the root to a leaf, deciding at each node n if it should advance to the left or the right child of n. Therefore, the SAH heuristic tries to even the probability of taking the left or right path at each node. In contrast to that, for the N-body simulation, the path of each particle is highly divergent, since it always advances to both children of a node, unless in the cases when no further descent is needed on that node (i.e. the cell opening criterion is not fulfilled). Therefore we want a tree where, on each node the probability to stop the decent at the left child is equal to the probability to stop it on the right child.

Due to this similarity of requirements on the tree, we used a variation of the SAH to determine the splitting point of the nodes in our tree (at least for the small nodes, see Section III). In our case, the heuristic is ported to 3D and the surface area is replaced by the mass of the corresponding node. This leads to the following equation:

$$VMH(x) = V_l(x) \cdot M_l(x) + V_r(x) \cdot M_r(x)$$

where $V_l(x)$ and $M_l(x)$ correspond to the volume and mass of the potential left child of the node, splitting the node at an axis aligned plane crossing the parent node at position x in the splitting dimension. $V_r(x)$ and $M_r(x)$ denote the volume and mass of the node's right child when split at position x. $VMH(x)$ is the volume-mass heuristic cost for the split position x. The cost is evaluated at numerous split position candidates for each node. The node is split at the candidate which minimizes the VMH cost.

V. Force calculation with Kd-trees

As described in the previous section, trees can be used to efficiently reduce the computational effort to solve the N-body problem numerically. In our implementation we are using a Kd-tree.

Gravitational force calculation: The main idea of tree algorithms is to reduce the computational cost of the force computation on a single particle by using a hierarchical multipole expansion. All particles in the simulation domain are hierarchically grouped into cells, the tree nodes for which the multipole expansion is calculated. For the force contribution of distant particles, a larger grouping of particles, namely a node in a higher level of the tree, can be used. Using the cell opening criterion, it can be decided whether a group of particles can be used or the tree needs to be traversed further. This approach allows to compute the force on a single particle with approximately $\log N$ interactions.

Unlike other implementations which are using quadrupole (e.g. Bonsai [16]) or even higher moments (e.g. Gasoline [21]), we follow the approach of GADGET-2 and only use monopole moments with the advantage that less memory is required, as just the total mass in a node and the center-of-mass coordinates need to be stored. Furthermore, the computational effort is lower while constructing the tree as higher moments do not need to be calculated and the monopole moments can be calculated conveniently during tree construction (see Section III). However, using monopole moments lowers the force accuracy. Still, the force accuracy can be controlled by the cell opening criterion e.g. by using a smaller cell opening angle. Depending obviously on the problem to be solved and on the implementation, opening more cells is still a small trade-off compared to computing higher order moments during tree construction.

Cell opening criterion: In our implementation, as we are using monopole moments for tree nodes, we apply the same strategy used in GADGET-2 [7], and use their *optimal* cell opening criterion. A cell (i.e. a node in the Kd-tree) is accepted if

$$\frac{GM}{r^2}\left(\frac{l}{r}\right)^2 \leq \alpha|\mathbf{a}|$$

evaluates to true, with G being the gravitational constant, M the total mass in the node, r the distance of the particle under consideration to the center-of-mass of the node, and l the largest side-length of the axis aligned bounding box around all nodes inside the corresponding node. Finally, $\mathbf{a}$ is the acceleration of the particle from the last timestep and α a *tolerance parameter*, used to control the force accuracy. However, in some cases this criterion is fulfilled also if the actual particle is located within a considered node which would lead to large force errors. To prevent against this, we additionally require the particle to lie sufficiently outside the bounding box of a node to be accepted. For more details we refer to [7].

A. Parallel force calculation using a Kd-tree

After the tree has been built, the actual force on each particle can be calculated by walking through the tree in a depth first manner. For each particle an OpenCL thread is started, walking the tree beginning from the root. On each node, the cell opening criterion is evaluated. If it is fulfilled, the walk will advance to both child nodes of the current node. If not, the force acting on the particle is calculated, using the current node as a proxy for all particles within the node. The pseudocode for our tree walk is shown in Algorithm 6. Although the tree walk is highly divergent, it can be implemented as a single loop, due to the depth-first ordering of the nodes.

Algorithm 6 Force calculation for each particle using the previously constructed Kd-tree

```
function TREEWALK(particles:list, tree:list)
    for all p in particles in parallel do
        for currentNode ← 0 to tree.size do
            n ← tree[currentNode]
            if n.isLeaf or !OPENCELL(p, n) then
                p.force ← p.force+ CALCFORCE(p, n)
                ▷ skip entire subtree of current node
                currentNode ← currentNode + n.size
            else
                ▷ continue depth-first walk
                currentNode ← currentNode + 1
            end if
        end for
    end for
end function
```

VI. Time Integration

To carry out full N-body simulations, we implement a time-centered leapfrog integration scheme (e.g. [3], [22]) with constant timesteps. Positions of the particles are advanced at full timesteps (drift) while new velocities are calculated at halfsteps (kick),

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{v}_{i+\frac{1}{2}}\,\Delta t$$
$$\mathbf{v}_{i+\frac{1}{2}} = \mathbf{v}_{i-\frac{1}{2}} + \mathbf{a}_i\,\Delta t$$

with x_i being the position and v_i the velocity of a particle at time i and Δt being the timestep. At each full timestep, the acceleration a_i is calculated using the Kd-tree implementation presented in the previous sections. Dynamic tree updates are used to prevent rebuilding the tree in each timestep: after calculating the new positions of the particles, the center of mass and bounding box of each tree node are updated. This update is performed by propagating the updated positions/bounding boxes bottom up the Kd-tree in a single pass. The tree is rebuilt when the computational cost (measured in numbers of interactions per particle) exceeds the initial value (when the tree was rebuilt the last time) by 20 %. Initially, $v_{i-\frac{1}{2}}$ is calculated by kicking the system of particles by half a timestep.

Being of $\mathcal{O}(n)$, the time needed for the time integration is negligible with respect to the tree building and force calculation.

VII. Results and Evaluation

In this Section we evaluate the result of our implementation in comparison to the state of the art simulation codes in terms of accuracy and execution speed.

A. Accuracy

Numerical issues in N-body simulations include force accuracy, time integration accuracy and dynamic range (mass resolution). We want to test our code with respect to the force accuracy provided by the approximation of the Kd-tree, as well as the conservation of energy.

When evaluating the accuracy of the force calculation, the mean squared error is not an ideal metric, since bodies with a high accuracy can compensate high errors of other bodies. This does not reflect well the quality of the solution. Especially hierarchical methods using a data structure which is inappropriate for the problem are very accurate on some bodies while others show a high error. Therefore, the 99 percentile gives more information about the quality of the solution, since it gives an upper limit for the error on almost all individual particles.

We compare our results to the widely spread GADGET-2 code, also because we use the same monopole and cell opening criterion, and to Bonsai, the state of the art N-body code for GPUs.

For our tests we are using a particle distribution according to a Hernquist density profile [23], an analytical model to describe dark-matter halos, spherical galaxies and bulges. For accuracy evaluation, we use 250,000 particles with a total mass of $1.14 \times 10^{12} \mathrm{M}_{\odot}$.

In collisionless systems (as the above cited applications of the Hernquist profile are), approximate calculations of the

force are sufficient as long as the force errors are random and small enough. On the other hand, the gravitational field of N point masses can be calculated exactly by direct summation. GADGET-2 provides a convenient functionality to calculate the forces by direct summation ($\mathbf{a}_{\text{direct}}$) besides the calculation via its octree. We are using this output as a reference and calculate the relative force errors of all particles from our implementation (GPUKdTree), GADGET-2 and Bonsai according to

$$\frac{\delta a}{a} = \frac{|\mathbf{a}_{\text{direct}} - \mathbf{a}_{\text{GPUKdTree/Bonsai/GADGET2}}|}{|\mathbf{a}_{\text{direct}}|}$$

To allow accuracy comparison between the different codes, we set the softening to zero as our implementation and GADGET-2 are using a spline-kernel softening and Bonsai is using Plummer softening (see [24] for more details about gravitational softening in N-body codes).
We note that for the cell opening criterion of GADGET-2, the acceleration of each particle of the previous timestep is needed. As this information is not available for the first force calculation, GADGET-2 uses the standard Barnes&Hut opening criterion to calculate the force acting on the particles. However, this force is then just used for the optimal opening criterion and the forces used in the first timestep are recalculated. In our implementation, we start with zero acceleration for all particles which effectively opens all cells for all particles, leading to direct summation of all forces. These accelerations agree with the ones computed in GADGET-2 using direct summation, and are used in the cell opening criterion of the actual tree walk. In Figure 1 the fraction of particles with a relative force error larger than indicated by the curve are shown for different values of the tolerance parameter α.

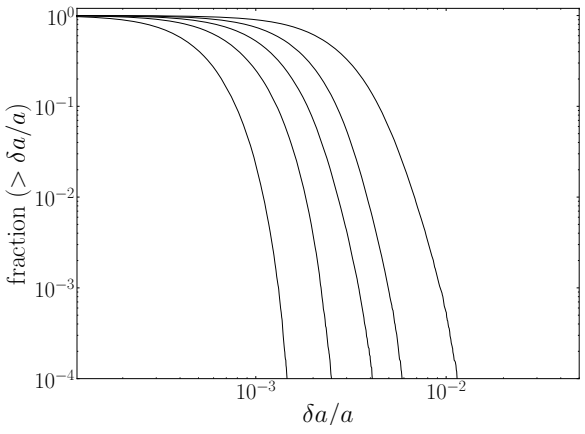

Figure 1. Relative force error for different values of $\alpha = 0.0001, 0.00025, 0.0005, 0.0025, 0.001$ in the cell opening criterion. The graph shows the fraction of particles having a relative force error larger than the indicated value.

As it is difficult to compare the two different cell opening criteria and tree topologies, we investigate at which cost a certain accuracy is achieved. Bonsai uses a slightly modified Barnes&Hut criterion $d > \frac{l}{\Theta} + \delta$ with Θ being a parameter to control the accuracy. For details we refer to [16]. In Figure 2 we show the mean number of interactions per particle needed to get a smaller relative force error for 99% of the particles as indicated by the points value along the y-axis. For all tested parameters, GADGET-2 needs less interactions than Bonsai to reach a comparable 99 percentile, although Bonsai is calculating quadrupole moments in each interaction. Also GPUKdTree needs less interactions to achieve the same accuracy as Bonsai. For low accuracy settings, our approach is even more efficient than GADGET-2.

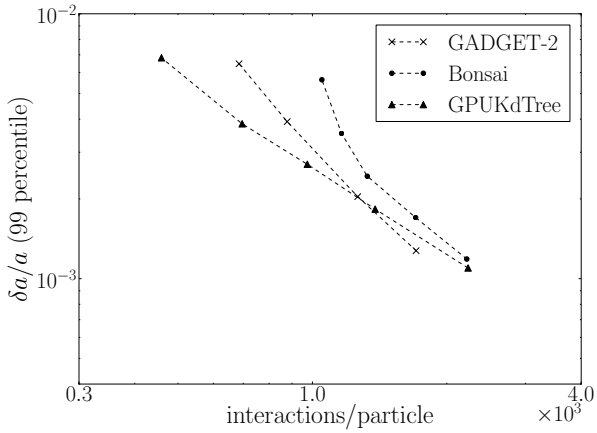

Figure 2. Mean number of interactions per particle needed to achieve a certain accuracy for 99% of the particles in the simulation domain. The different data points again correspond to runs with different accuracy parameters $\alpha = 0.005, 0.0025, 0.001, 0.0005$ for GADGET-2, $\alpha = 0.0025, 0.001, 0.0005, 0.00025, 0.0001$ for GPUKdTree as well as $\Theta = 0.6, 0.7, 0.8, 0.9, 1.0$ for Bonsai.

We also want to compare the accuracy of all three codes when calculating the same amount of interactions. We chose a value of 1000 interactions/particle for our test problem and adjusted the values for α and Θ accordingly. As can be seen in Figure 3, our implementation performs slightly better than GADGET-2. The results of Bonsai however, show a much higher scatter in relative force errors.

Finally, as another measure for the quality of our code, we observe and compare the energy conservation. In Figure 4 we plot the relative energy error

$$\delta E = \frac{E_0 - E_t}{E_0}$$

with E_0 being the total energy (kinetic plus potential energy of the particle distribution) at the beginning of the simulation and E_t the total energy at simulation time t. For all three codes we chose the same configuration as for the accuracy comparison in Figure 3. For GPUKdTree and Bonsai we chose a fixed timestep of 0.003 Myr. In GADGET-2 we set

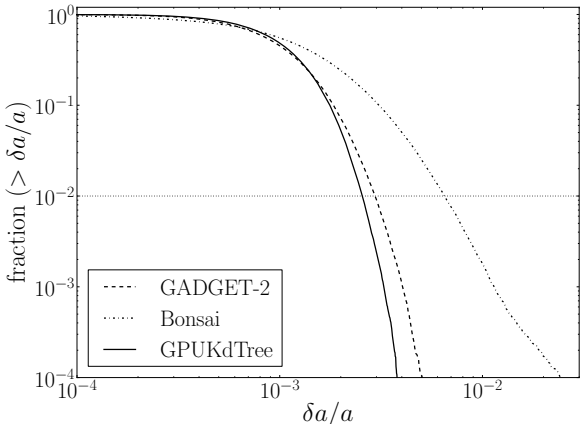

Figure 3. Relative force errors for the three different codes used. For each code α and Θ was chosen that the mean number of interactions per particle is 1000. The dotted line marks the error at the 99 percentile.

this value as the maximum allowed timestep in order to prevent the usage of the individual timestepping (differently sized timestep for each particle depending on the current acceleration acting on the particle) for a fair comparison between all codes. The results show that our GPUKdTree implementation provides a small energy error throughout the whole simulation, comparable to GADGET-2. Bonsai however shows a somewhat higher but at the same time also more constant error. Both, GPUKdTree and GADGET-2 show more scatter in the error with some spikes in the distribution having a higher maximum error than Bonsai.

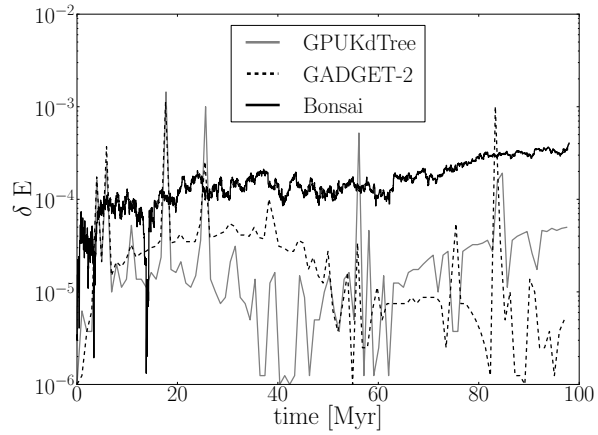

Figure 4. The relative energy error $\delta E = \frac{E_0 - E_t}{E_0}$ throughout the simulation is shown.

B. Performance

We evaluate the performance of our implementation on various CPUs and GPUs from different vendors. Our OpenCL implementation was designed to run on any device that supports OpenCL. For performance reasons we use a dedicated algorithm to sort bodies during the large node phase for GPUs and CPUs. NVIDIA GPUs could not run our OpenCL code correctly, giving wrong results without any error message. However, since we used LibWater [25] to implement our program, it could easily be ported to CUDA without any changes in our code. The CUDA version works flawlessly on the NVIDIA GPUs.

To evaluate the performance, we are using datasets containing different number of particles, all using a Hernquist density profile as used in our accuracy experiments described in the previous paragraph. The dataset containing two million particles could not be run on the AMD Radeon HD5870 due to its limitation of the maximal buffer size. We also compare the performance of our implementation with the one achieved with GADGET-2 [7] and Bonsai [16]. GADGET-2 contains no implementation for GPUs and can only be executed on CPUs. For our experiments we use the same dual socket Intel Xeon X5650 system with a total of twelve cores that we used to evaluate the performance of our implementation on CPUs. Bonsai is implemented in CUDA and is therefore limited to NVIDIA GPUs. Furthermore, the version of Bonsai which is available online did not work on our Tesla k20c GPU. On this hardware, the program crashed due to a CUDA driver error. Hence, we could evaluate Bonsai's performance only on a NVIDIA GeForce GTX480.

For a fair comparison, we set the accuracy parameters for each implementation to achieve an error below 0.4% for 99% of the particles. This results in an α of 0.001 and 0.0025 for GPUKdTree and GADGET-2, respectively. For Bonsai, Θ is set to 1.0.

Tree building: Table I shows the time needed for tree building on different hardware with different data sizes. The numbers show, that our tree building algorithm fits the GPU architecture quite well. All GPUs show a speedup between 3.3 and 10.4 over the tested CPU. It is noteworthy, that the NVIDIA GPUs are more effective for smaller datasets, while the AMD GPUs scale better with the problem size. The relative bad performance of the AMD GPUs on small problem sizes is related to the very high number of kernels that have to be called during the tree building (see Section III) in correlation with their high kernel invocation overhead [26]. The simulation with 2 million particles could not be run on the AMD Radeon HD5870 due to its restriction to the maximum size of a single data structure. It is interesting to note, that the NVIDIA GeForce GTX480 shows almost the same performance as the much newer NVIDIA Tesla k20c, although the latter one has a much higher peak performance (1.3 vs. 3.5 TFLOPs).

The times given for GADGET-2 and Bonsai include the sorting of the particles and the building of the actual tree, since they can construct the tree only on pre-sorted particles. Building the octree used in both GADGET-2 and Bonsai is much faster than building the Kd-tree used in our approach.

Table I
TREE BUILDING TIMES IN MS

N. Particles	250k	500k	1M	2M
Xeon X5650	881	1795	3640	7278
GeForce GTX480	158	290	595	1202
Tesla k20c	167	293	586	1195
Radeon HD5870	262	381	675	-
Radeon HD7950	152	219	380	698
GADGET-2 (X5650)	50	90	180	370
Bonsai (GTX480)	24	43	83	167

Table II
FORCE CALCULATION USING A PREVIOUSLY CONSTRUCTED TREE TIMES IN MS

N. Particles	250k	500k	1M	2M
Xeon X5650	456	966	1996	4145
GeForce GTX480	236	476	934	1844
Tesla k20c	204	405	801	1588
Radeon HD5870	155	287	572	-
Radeon HD7950	85	169	332	651
GADGET-2 (X5650)	909	1940	4160	8580
Bonsai (GTX480)	40	81	163	325

The main reason for this is the rearranging of the particles. To build an octree, the domain is decomposed using a Peano-Hilbert curve [6]. At the beginning of the tree building, the particles are sorted according to this domain composition. By doing so, the particles do not have to be rearranged during the rest of the tree building. When building a Kd-tree, on the other hand, the particles have to be rearranged in each iteration of the tree building step, which takes a significant amount of time.

Tree Walk: As explained in Section V, the force on each particle is calculated by walking through the previously built tree. The performance for the tree walk is given in Table II. Also, the tree walk is faster on all tested GPUs than on the tested CPU. The speedup varies between 1.9 and 6.3 depending on the GPU and data size. The AMD GPUs are suited better for the tree walk than both NVIDIA GPUs. Even the old AMD Radeon HD5870 is able to outperform both NVIDIA GPUs. During the tree walk, the large kernel invocation overhead of the AMD GPUs plays a minor role, since the tree walk of all particles consists of only one single kernel call. Using a AMD Radeon HD7950, our implementation can reach a throughput of 3 Mparticles/s.

The measurements clearly show, that our implementation is much faster than GADGET-2, mainly due to the efficient use of the massively parallel GPU architecture. Also, using the same CPU, the tree walk of our implementation is approximately twice as fast as in GADGET-2. However, GADGET-2 lacks a shared-memory implementation and is handicapped by overhead due to the MPI library in these tests. Bonsai shows a very high performance in our test case. However, this high performance comes at the cost of worse error distribution, as shown in Figure 3, and worse energy conservation, as depicted in Figure 4.

VIII. CONCLUSION

We have observed, that octrees for N-body simulations can be built very fast, on both GPUs and CPUs, when the particles are pre-sorted according to a Peano-Hilbert curve. Constructing a Kd-tree takes more time, mainly due to the rearranging of the particles in every timestep. By using GPUs, the Kd-tree construction can be accelerated up to 10 fold over the execution time on CPUs. The tree building time of GPUKdTree scales linearly with the number of particles in the simulation.

Our implementation of the tree walk is noticeably faster than the one of GADGET-2, when using the same hardware. It also shows better scalability than GADGET-2 with increasing problem sizes. Even more, executing the treewalk of our implementation on GPUs gives another speedup of up to 6 times over the CPU. We achieve a throughput of up to 3 Mparticles/s which is the highest performance reached on an AMD GPU that we are aware of. However, Bonsai achieves an even higher performance using NVIDIA GPUs. This shows, that Bonsai's breadth-first tree walk fits the GPU architecture better than our implementation, performing a depth-first walk. However, Bonsai also shows a much higher scatter in relative force errors. That leads to a few particles with a quite high error.

Comparing the octree and Kd-tree structure in terms of accuracy, our observations show, that using a Kd-tree with a good heuristic (VMH), a moderate accuracy can be reached with very few interactions. Using a Kd-tree instead of an octree does not have a negative effect on the relative force error distribution, as our results show a slightly better behavior than the ones obtained with GADGET-2. The validity of GPUKdTree is also underlined by our energy conservation tests.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper. This work has been partially funded by Interreg4 and FWF as part of the EN-ACT project (P7030-015-030) and doctoral school - Computational Interdisciplinary Modelling FWF DK-plus (W1227). The authors acknowledge the UniInfrastrukturprogramm of the BMWF Forschungsprojekt Konsortium Hochleistungsrechnen.

REFERENCES

[1] P. E. Kyziropoulos, C. K. Filelis-Papadopoulos, and G. A. Gravvanis, "N-body simulation based on the particle mesh

method using multigrid schemes," in *FedCSIS*, 2013, pp. 471–478.

[2] J. Harnois-Déraps, U.-L. Pen, I. T. Iliev, H. Merz, J. D. Emberson, and V. Desjacques, "High-performance P^3M N-body code: $CUBEP^3M$," *MNRAS*, vol. 436, pp. 540–559, Nov. 2013.

[3] J. Barnes and P. Hut, "A hierarchical $O(N \log N)$ force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, Dec. 1986.

[4] M. S. Warren and J. K. Salmon, "Astrophysical n-body simulations using hierarchical tree data structures," in *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '92. Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, pp. 570–576. [Online]. Available: http://dl.acm.org/citation.cfm?id=147877.148090

[5] J. G. Stadel, "Cosmological N-body simulations and their analysis," Ph.D. dissertation, University Of Washington, 2001.

[6] V. Springel, N. Yoshida, and S. D. White, "GADGET: a code for collisionless and gasdynamical cosmological simulations," *New Astronomy*, vol. 6, no. 2, pp. 79–117, Apr. 2001. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S1384107601000422

[7] V. Springel, "The cosmological simulation code gadget-2," *Monthly Notices of the Royal Astronomical Society*, vol. 364, no. 4, pp. 1105–1134, Dec. 2005. [Online]. Available: http://doi.wiley.com/10.1111/j.1365-2966.2005.09655.x

[8] L. Nyland, M. Harris, and J. Prins, "Fast n-body simulation with cuda," in *GPU Gems 3*, H. Nguyen, Ed., 2007, ch. 31.

[9] E. Elsen, M. Houston, V. Vishal, E. Darve, P. Hanrahan, and V. S. Pande, "Poster reception - n-body simulation on gpus," in *SC*, 2006, p. 188.

[10] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for gpus: stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 777–786, Aug. 2004. [Online]. Available: http://doi.acm.org/10.1145/1015706.1015800

[11] P. Hut, J. M. Arnold, J. Makino, S. L. McMillan, and T. L. Sterling, "Grape-6: A petaflops prototype," in *proceedings of the 1997 Petaflops Algorithms Workshop (PAL'97)*, 1997.

[12] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji, "42 tflops hierarchical n-body simulations on gpus with applications in both astrophysics and turbulence," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 62:1–62:12. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654123

[13] T. Hamada and K. Nitadori, "190 tflops astrophysical n-body simulation on a cluster of gpus," in *SC*, 2010, pp. 1–9.

[14] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kalé, and T. R. Quinn, "Scaling hierarchical n-body simulations on gpu clusters," in *SC*, 2010, pp. 1–11.

[15] M. S. Warren, "2hot: An improved parallel hashed oct-tree n-body algorithm for cosmological simulation," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 72:1–72:12. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503220

[16] J. Bédorf, E. Gaburov, and S. P. Zwart, "A sparse octree gravitational n-body code that runs entirely on the gpu processor," *J. Comput. Physics*, vol. 231, no. 7, pp. 2825–2839, 2012.

[17] I. Wald and V. Havran, "On building fast kd-trees for ray tracing, and on doing that in O(N log N)," in *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 2006, pp. 61–69.

[18] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley, "State of the art in ray tracing animated scenes," in *STAR Proceedings of Eurographics 2007*, D. Schmalstieg and J. Bittner, Eds. The Eurographics Association, Sep. 2007, pp. 89–116.

[19] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," in *ACM SIGGRAPH Asia 2008 papers*, ser. SIGGRAPH Asia '08. New York, NY, USA: ACM, 2008, pp. 126:1–126:11. [Online]. Available: http://doi.acm.org/10.1145/1457515.1409079

[20] H. Nguyen, *GPU Gems 3*, 1st ed. Addison-Wesley Professional, 2007.

[21] J. W. Wadsley, J. Stadel, and T. Quinn, "Gasoline: a flexible, parallel implementation of TreeSPH," *New astronomy*, vol. 9, pp. 137–158, Feb. 2004.

[22] T. Quinn, N. Katz, J. Stadel, and G. Lake, "Time stepping N-body simulations," *ArXiv Astrophysics e-prints*, Oct. 1997.

[23] L. Hernquist, "An analytical model for spherical galaxies and bulges," *The Astrophysical Journal*, vol. 356, p. 359, Jun. 1990. [Online]. Available: http://adsabs.harvard.edu/doi/10.1086/168845

[24] W. Dehnen, "Towards optimal softening in three-dimensional N-body codes - I. Minimizing the force error," *MNRAS*, vol. 324, pp. 273–291, Jun. 2001.

[25] I. Grasso, S. Pellegrini, B. Cosenza, and T. Fahringer, "Libwater: heterogeneous distributed computing made easy," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 161–172. [Online]. Available: http://doi.acm.org/10.1145/2464996.2465008

[26] P. Thoman, K. Kofler, H. Studt, J. Thomson, and T. Fahringer, "Automatic OpenCL device characterization: guiding optimized kernel design," in *Euro-Par*, 2011, pp. 438–452.