A Hybrid Parallel Barnes-Hut Algorithm for GPU and Multicore Architectures

Hannes Hannak, Hendrik Hochstetter and Wolfgang Blochinger

```
@InProceedings{Hannak2013,
      author="Hannak, Hannes and Hochstetter, Hendrik and Blochinger, Wolfgang",
      editor="Wolf, Felix and Mohr, Bernd and an Mey, Dieter",
      title="A Hybrid Parallel Barnes-Hut Algorithm for GPU and Multicore
             Architectures",
      booktitle="Euro-Par 2013 Parallel Processing",
      year="2013",
      publisher="Springer Berlin Heidelberg",
      address="Berlin, Heidelberg",
      pages="559--570",
      isbn="978-3-642-40047-6",
      doi="10.1007/978-3-642-40047-6_57",
      url=" https://doi.org/10.1007/978-3-642-40047-6_57"
}
```

A hybrid parallel Barnes-Hut algorithm for GPU and multicore architectures

Hannes Hannak[1]*, Hendrik Hochstetter[2], and Wolfgang Blochinger[1]

[1] Institute of Parallel and
Distributed Systems, University of Stuttgart
`<first.last>@ipvs.uni-stuttgart.de`

[2] Computer Graphics and Multimedia Systems Group, University of Siegen
`hochstetter@nt.uni-siegen.de`

Abstract. Using the Barnes-Hut algorithm as an example we deal with the design of parallel algorithms that are able to exploit multicore CPUs and GPUs conjointly. Specifically, we demonstrate how to modularize a parallel application according to specific aspects of parallel execution. This allows for a flexible assignment of individual modules to the two parallel architectures based on their actual performance characteristics. Furthermore, we discuss a hybrid module for the most time consuming part of the algorithm that utilizes CPU and GPU simultaneously employing on a novel load balancing heuristic. Our experimental evaluation shows that our method greatly increases overall efficiency by allowing to deploy an optimal configuration of modules for each individual computer system.

1 Introduction

In recent years GPU based parallel computing has attained considerable interest. However, most algorithm designs presented so far exclusively exploit the GPU for executing parallel tasks while the (potentially many) cores of the CPU are running idle.

In this paper, we discuss a modularized parallel application which enables a flexible assignment of individual parts of the algorithm for execution on the GPU, the CPU cores, or also on both platforms. For our studies we have chosen the Barnes-Hut algorithm as an example which computes the evolution of a large set of particles based on the forces individual particles exert on each other. In contrast to existing work (e.g. [6, 7]) our approach specifically takes into account that depending on the actual capabilities of the hosts' CPU cores and GPU, different assignments of computational parts of the Barnes-Hut algorithm may result in the most efficient solution.

Achieving such a degree of flexibility is especially important when executing parallel applications in highly heterogeneous environments. A typical example

* Hannes Hannak is supported by a grant of the Landesgraduiertenförderung Baden-Württemberg

of this kind of scenarios are Desktop Grids (e.g. [1, 13]), which combine the computing power provided by volunteers into a global computing grid and are known to aggregate a huge variety of hardware resources [9]. Also, Cloud Computing instances exhibit a similar scenario: Even though the user is guaranteed certain minimal hardware limits, the actual configurations and capabilities can vary considerably.

2 Preliminaries

2.1 The Barnes-Hut algorithm

N-Body methods simulate the dynamics occuring in a set of N particles based on the forces (e.g. gravity) the particles exert on each other. Simulations proceed in discrete timesteps, each resulting in new particle positions. As exact approaches consider the forces between each pair of particles, $\mathcal{O}(N^2)$ calculations are necessary per timestep. To treat larger numbers of particles, approximation methods have been proposed that considerably decrease the quadratic complexity [14].

One well known hierarchical approach to the N-body problem is the Barnes-Hut algorithm [2] which exhibits $\mathcal{O}(N \log N)$ time complexity. It employs a tree data structure to approximate forces acting on the individual particles. The leaf nodes of the *Barnes-Hut Tree* hold the positions and masses of single particles whereas each inner node represents particle equivalents summarizing the positions and masses of all particles of the subtree rooted at that node. Thus inner nodes act as pseudo particles, which correspond to a certain area of the simulation space.

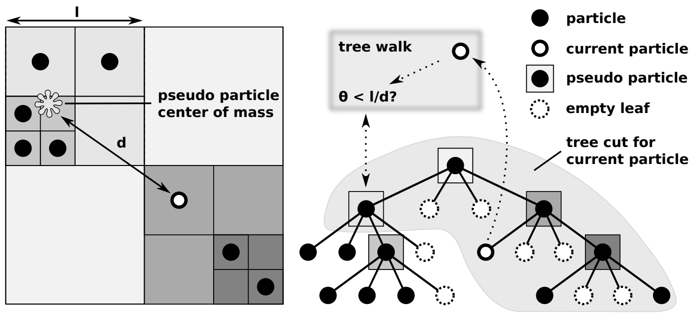


Fig. 1. Simulation space and corresponding Barnes-Hut tree (same colors denote respective pseudo particles). For clarity we show a two-dimensional space, however, the same principles hold in 3D, too. We also illustrate the parameters of the opening criterion applied for steering the tree walk for one particle and the resulting tree cut.

The computations of a timestep consist of the following consecutive steps:

S1) Tree Building: The simulation space is recursively divided into equally sized subspaces, until each of them contains at most one particle (cf. Fig. 1). These subspaces define the leaf nodes of the Barnes-Hut tree which store the position and the mass of the contained particle.
S2) Computing Pseudo Particles: For each inner node in the tree, the pseudo particle data have to be computed. The mass is calculated as the sum of the masses of the child nodes and the position as the center of mass.
S3) Force Evaluation: Forces are computed for each particle separately by traversing the tree, starting with the root node (*tree walk*). To every pseudo-particle an *opening criterion* is applied to decide whether it can be used for force evaluation or if the node must be expanded. The opening criterion is given by $\theta < l/d$, where θ is a chosen approximation factor, l the length of the subspace the node represents and d the distance of the node's position to the current particle (cf. Fig. 1). Thus, choosing a smaller value for θ decreases the simulation's approximation error but increases running time, as more interactions have to be computed. If the opening criterion applies to an inner node, the algorithm executes recursively on all child nodes and adds up the individual forces. Otherwise, either the pseudo particle data is used to compute forces or, in case of a leaf node, the particle data. The set of interaction partners define a particle's *tree cut* as that part of the tree that is needed for the particle's force evaluation.
S4) Particle Update: Once the total force on each particle is known, the algorithm calculates the new positions and velocities of the particles (by applying Newton's laws). These serve as input for the next timestep.

2.2 GPGPU computing

In contrast to CPUs, GPUs consist of a hierarchical combination of many primitive processor cores. On the nVidia platform, e.g., each *Streaming Multiprocessor (SM)* comprises a set of simple *Streaming Processors (SP)* which share a common program counter and control unit. All SPs within an SM execute instructions synchronously and thus follow the SIMD architecture model, whereas different SMs operate independently and follow the MIMD model.

We use nVidia GPUs and the CUDA SDK to speed up the Barnes-Hut algorithm. nVidia GPUs offer hardware support for the scheduling of threads. When running kernels, the scheduler selects small groups of threads, called *warps*, for execution on SMs. If single threads of a warp follow different program paths, all paths have to be followed one after another as each SM only possesses one control unit. So, to achieve high efficiency with the computational resources of GPUs, one must avoid thread divergence as if-then-else constructs have to be serialized in the SIMD model. Because of this limitation, GPUs are best suited to compute regularly structured problems.

To attenuate this constraint, modern GPUs introduce *warp vote functions* to allow all threads of a warp to evaluate conditions jointly. For a boolean variable `var`, the warp vote function `__all(var)` returns true in each thread of the warp if `var` is true in every thread. If `var` is false in only one thread, false is returned in all threads of the warp. Thus, warp vote functions in conditional expressions ensure coherent program paths within warps and avoid thread divergence which can greatly increase efficiency.

3 A Barnes-Hut method for hybrid architectures

To efficiently use the computing power provided by the CPUs and GPUs of different hosts, we designed a modular method that employs the most suitable platform for each of the steps of the Barnes-Hut algorithm.

We first describe the structure and organisation of the simulation data in Section 3.1 followed by the characteristics of the individual steps that lead to our modularization in Section 3.2. As force evaluation is the most expensive step, we afterwards focus on the implementation of this module. In Section 3.3 we lay out how we use the GPU for force evaluation. To exploit the maximum available computational power, we designed an additional module that runs on both platforms in parallel (*hybrid mode*) which we discuss in Section 3.4. Furthermore, we invented a new load balancing mechanism to minimize processor idle time. We present details and advantages of this method in Section 3.5.

3.1 Data structures and data organisation

As described in Section 2.1, the Barnes-Hut algorithm can be divided into a sequence of distinct steps each of which depends on the outcome of preceding ones but can be solved as an independent entity. In our approach *modules* encapsulate the functionality of these steps. Depending on its implementation, each module can be executed in parallel on either one or both platforms. The definition of explicit interfaces between successive modules allows us to run the algorithm with an arbitrary assignment of modules to execution platforms. The best configuration for a specific host can be determined by initially performing a benchmarking run evaluating all possible combinations.

In principle, we replicate particle and tree data in the memories of the two platforms to minimize costly data transfers between CPU and GPU memory during computations. However, if data that is needed by the subsequent module has been changed on one platform, e.g., by updating particle positions or tree building (cf. Tab. 1), it must be transferred if the execution crosses platform borders. As the resulting memory latencies reduce efficiency, we minimize idle time by overlapping data transfers with the module execution.

To facilitate modularity and to speed up data transfers and computations, we introduce two further steps/modules between tree building (S1) and computing pseudo particles (S2):

S1a) Tree Linearization: To allow for an efficient transfer between platforms and to speed up tree traversals, we transform the tree data structure into a set of linear arrays: The *tree array* reflects the recursive structure of the tree, the *next* and *more arrays* allow for a stack-free tree traversal and the *particle indirection array* allows particles to be accessed according to their spatial position.
S1b) Particle Sorting: Spatial proximity within particles can be exploited in a

number of cases during the computation to speed up execution and memory access. As particles close to each other need to interact frequently, keeping them in close memory positions increases cache efficiency. Thus, we sort particles through the particle indirection array using three dimensional space filling curves.

3.2 Modularization

Each of the steps described above exhibits different characteristics, espcially with respect to data-dependencies. In the following paragraphs, we give a detailed account of the characteristics that determined the parallelization strategy for each module. Table 1 provides an overview.

Table 1. Modules with interfaces and platforms (C = CPU, G = GPU, H = Hybrid.)

	Requires	**Provides**	**Platform**
S1	particle data	unsorted tree	C
S1a	unsorted tree	next/more, particle indirection, tree arrays	C
S1b	particle indirection array, particle data	sorted particle data	C, G
S2	tree array, sorted particle data	pseudo particles	C, G
S3	next/more, pseudo particles, sorted particle data	forces	C, G, H
S4	forces, sorted particle data	particle data	C, G

The most time consuming steps of the Barnes-Hut algorithm are tree building (S1) and force evaluation (S3) which both have a time complexity of $\mathcal{O}(N \log N)$. All four remaining steps are of linear complexity. Especially particle sorting (S1b) can be done in linear time as one can determine the actual spatial order of particles from the tree structure in one traversal.

Concerning data access patterns, tree building (S1), tree linearization (S1a) and particle sorting (S1b) are the most complex problems. Tree building is realized by inserting particles into a dynamically changing tree. The way in which the tree is arranged in memory thus depends on the order the particles are processed in. This usually leads to highly irregular access patterns. During tree linearization (S1a), the tree data structures are rearranged in the order of a depth first traversal to speed up succeeding steps. Computing pseudo particles (S2) and force evaluation (S3) benefit from tree linearization (S1a) and particle sorting (S1b) in that they have less irregular access patterns traversing the tree than the steps before. Particle sorting (S1b) allows for regular access patterns on particle data in later steps but has highly irregular access patterns, itself.

In addition to irregular access patterns, tree building (S1), linearization (S1a), and computing pseudo particles (S2) suffer from data dependencies. During tree building (S1), particles are inserted in parallel into a shared tree. Changes to the tree have to be made atomic through locking mechanisms so that no inconsistent states of the tree may arise. This limits the amount of parallelism that can be achieved. Particle sorting (S1b), force evaluation (S3) and particle update (S4), in contrast, do not involve any data dependencies and thus are trivially parallelizable.

Our partition of the Barnes-Hut algorithm into a sequence of unique steps was guided by the characteristics described above. They are summarized in Table 2. Although force evaluation dominates the running time in sequential implementations with 97 %, all steps have to be parallelized. If only force evaluation was parallelized, following Amdahl's law a maximum speedup of 33.3 could be achieved, no matter how many processors were employed. However, implementing different parallelization approaches for each step can be easily accomplished through our modular design.

Table 2. Properties of the modules of our implementation of the Barnes-Hut algorithm.

	Access patterns	**Data dependencies**	**Time complexity**	**Sequential CPU running time (in %)**
S1	highly irregular	yes	$\mathcal{O}(N \log N)$	≈ 2
S1a	highly irregular	yes	$\mathcal{O}(N)$	< 1
S1b	highly irregular	no	$\mathcal{O}(N)$	< 1
S2	irregular	yes	$\mathcal{O}(N)$	< 1
S3	irregular	no	$\mathcal{O}(N \log N)$	≈ 97
S4	regular	no	$\mathcal{O}(N)$	< 1

As GPUs usually use relatively slow, high-bandwidth memory, regular access patterns are crucial to achieve good performance. If irregular access patterns occur, compute time is wasted waiting for memory accesses. In contrast, on CPUs, more sophisticated memory hierarchies with larger and more cache stages are employed which can better hide latencies if irregular patterns occur. We thus expect GPUs to perform best in particle update (S4) and force evaluation (S3) and less well in steps with more irregular access patterns and data dependencies.

Due to the combination of highly irregular access patterns and data dependencies, we expect tree building (S1) and linearization (S1a) to be best suited for an execution on CPUs. The additional steps of tree linearization (S1a) and particle sorting (S1b) provide tree and particle data in an order that allows for more regular memory access patterns, so that the most time consuming step, the force evaluation (S3), can be parallelized efficiently on both CPU and GPU.

3.3 GPU based force evaluation

As motivated above, GPUs are well suited to perform the force evaluation by executing one thread per particle. Because we sort particles (S1b), each warp only processes nearby particles. The force evaluation of nearby particles leads to very similar tree cuts, so we thereby greatly reduce thread divergence.

To completely avoid thread divergence, interaction lists can be employed in GPU-based force evaluation [7, 4]. This approach groups nearby particles together. A tree walk computes an interaction list for each group that stores all particles and pseudo-particles that contribute to the groups' forces. Forces acting on the individual particles of a group are computed completely synchronously, using the interaction list. In contrast, like [6], we use warp vote functions to achieve a similar effect. During the tree walk, the opening criterion is evaluated using `__all`. Approximations thus are only acceptable if all threads of a warp agree. Else the threads continue the tree walk with the children of the current node. So, all threads of a warp access the same node data and don't diverge. Evaluating the opening criterion for each particle separately, would cause parts of the tree to be traversed to different depths. Using `__all`, all threads of a warp follow the deepest path necessary for any one of the warp's particles, thereby increasing accuracy.

Although more interactions are computed using warp vote functions, the reduced amount of memory accesses and the absence of thread divergence greatly improve performance. Additionally, latency hiding through the scheduling hardware can work more effectively with warp vote functions than interaction lists as the memory bound tree traversal and the compute bound evaluation of interactions are not artificially separated but interleaved.

3.4 Hybrid force evaluation

Our modularized approach allows us to combine CPU- and GPU-based implementations of the force evaluation to yield a hybrid module that utilizes both platforms in parallel. To balance the workload among CPU and GPU, each processor evaluates forces for a subset of particles. The size of the subset depends on the processor's computational power.

We model the CPU as a set of equally powerful processors. Because of its hardware-based thread scheduling we can't control the GPU's load balancing. We thus represent the GPU as a single processor. For the same reason, it is difficult to predict GPU running times solely from the problem structure. Thus, we exploit the spatio-temporal stability of N-Body problems. We measure running times of CPU and GPU computations and adjust the GPU's computational power p_{GPU} dynamically in each timestep. That way, we are able to capture the ratio between the computing power of GPU and CPU and to distribute the load, accordingly.

If computations on the GPU (t_{GPU}) took less time than on the CPU (t_{CPU}), p_{GPU} is increased to $p_{\mathrm{GPU}}\left(c_{\mathrm{inc}}\frac{t_{\mathrm{GPU}}}{t_{\mathrm{CPU}}} + (1 - c_{\mathrm{inc}})\right)$ so that the GPU evaluates more forces. If on the other hand the CPU took less time than the GPU, p_{GPU} is decreased to $p_{\mathrm{GPU}}\left(c_{\mathrm{dec}}\frac{t_{\mathrm{GPU}}}{t_{\mathrm{CPU}}} + (1 - c_{\mathrm{dec}})\right)$. Else, p_{GPU} remains unchanged. The constants c_{inc}, c_{dec} and the initial value of p_{GPU} are precomputed in a benchmark run. c_{inc} and c_{dec} determine how fast p_{GPU} is increased or decreased. They are chosen in such a way that no work is stolen from the faster platform due to fluctuations in t_{GPU} or t_{CPU} that may be caused by the different ways in which CPUs and GPUs operate, the operating system, or the system's user.

3.5 A novel dynamic load balancing scheme for force evaluation

To improve the hybrid force evaluation's load balancing, the varying costs to evaluate forces acting on different particles must be considered, instead of assuming a uniform cost per particle. For particles in dense areas of the simulation space more interactions have to be computed than for particles in sparse areas.

Due to the spatio-temporal stability of N-body problems, particles move only slightly between timesteps. Thus, the number of interactions also changes slowly and provides a good measure for the computational cost to evaluate forces.

A common approach to load balancing is counting the number of interactions for each particle and using the data to distribute work in subsequent timesteps [14]. As GPUs use hardware-based thread scheduling, this approach isn't very well suited for our hybrid module. When counting interactions on the GPU, more resources are required and less threads can be resident in SMs, which strongly impairs the scheduling mechanism and the GPU's performance.

We thus introduce a novel heuristic approach to predict the number of interactions. It uses only data gathered during one traversal of the tree and does not introduce any overhead into the force evaluation itself. Our method is based on the tree level l of each particle x which turned out to be a good measure for the particle density in the region surrounding that particle (cf. Fig. 2). If l is

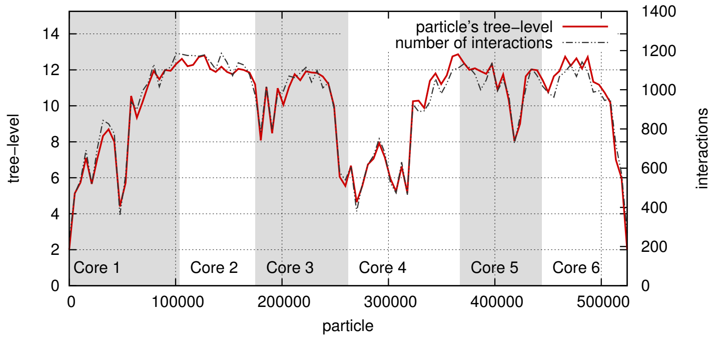


Fig. 2. Comparison between our tree-level heuristic and the number of interactions for one timestep during force evaluation using $\theta = 0.75$. Columns depict the resulting particle distribution among six CPU cores (grey and white).

large, many particles are located close to x and many interactions have to be considered to evaluate the force acting on x. The tree level thus provides a good heuristic to approximate the computational cost for a given particle.

To incorporate our heuristic into the force evaluation, we write a *cost array* containing the prefix sum of the tree level (and thus cost) of each particle. The last element of this array holds the total cost C to evaluate all forces. Each processor i, which may be a CPU core or the GPU, has a computing power p_i that determines the amount of work assigned to it. The system's total computing power is $p_{\text{total}} = \sum_i p_i$. During simulations, each processor i computes forces for a contiguous subset of particles, the lower and upper indices of which are the same as the indices of $C\frac{\sum_{k=0}^{i-1} p_k}{p_{\text{total}}}$ and $C\frac{\sum_{k=0}^{i} p_k}{p_{\text{total}}}$, respectively, in the cost array which can be determined through binary search. Fig. 2 shows a particle distribution resulting from our load balancing scheme. Note how, e.g., Core 1 is assigned a larger particle subset than Core 2, as for most particles assigned to Core 1 less interactions have to be computed.

4 Performance evaluation

To evaluate our approach we performed test runs on a variety of different hardware (see Table 3). For each run we used two colliding galaxies of equal mass as input. Each galaxy consisted of a stellar disk surrounded by a dark matter halo following the Springel-Hernquist model. We used Starscream [5] to create the galaxies and place them on a parabolic orbit.

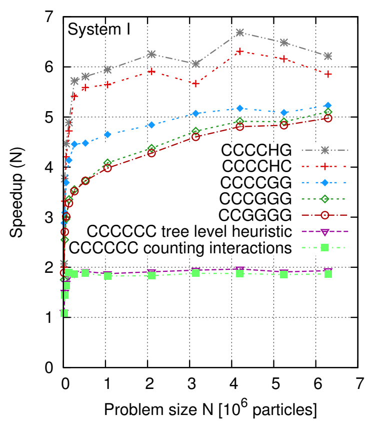
Table 3. Configuration of the node types used for the experimental evaluation.

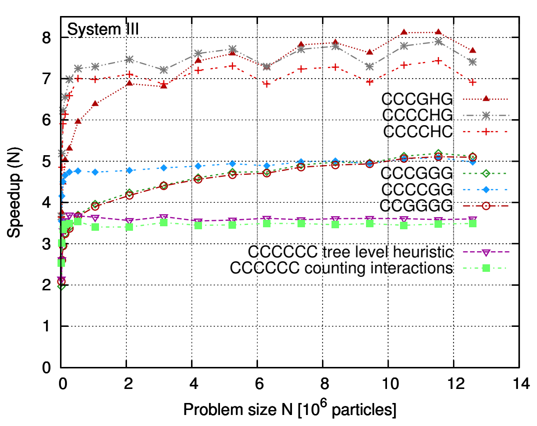
Type	CPU (Cores / Threads / Frequency)	RAM	GPU	VRAM
I	Intel Core i7 M620 (2 / 4 / 2,67 GHz)	8 GB	nVidia GeForce NVS 3100M	256 MB
II	AMD Phenom II X6 1055T (6 / 6 / 2.8 GHz)	4 GB	nVidia GeForce GTX 660 Ti	2 GB
III	Intel Core i7-2670QM (4 / 8 / 2.2 GHz)	4 GB	nVidia GeForce GTX 570M	1.5 GB

We denote the configuration of platforms that execute the simulation steps as strings of six characters, representing the the six modules. Each module is either executed on the CPU (C), the GPU (G), or in hybrid mode (H) on CPU and GPU in parallel. Speedups are based on the sequential CPU running time on the respective test system. Figure 3 shows the speedups obtained using different combinations of modules on our test systems for varying problem sizes.

In purely CPU-based parallel configurations (CCCCCC), we achieved speedups of 1.8 (type I), about 5 (type II), and 3.7 (type III). On all three systems, the speedups of our heuristic load balancing scheme could compete with those of load balancing based on interaction counting. In the sequential configuration CCC-CCC, we measured an overhead of 2 % just to count interactions. For GPU-based force evaluation this introduced an overhead of more than 30 %.

To find the most efficient module combination, we ran all possible combinations on each system. On type I, the best combination turned out to be CCCCHG which obtained speedups of 6 and above. The most efficient combination for type II, CCGGGG, achieved speedups close to 60 for the largest problem sizes. This discrepancy can be explained by the different ratio of GPU to CPU computing power on the test systems. On type II the GPU outperforms the CPU, whereas on type I both have similar peak performance. On type III, the best module combination depended on the problem size. For problem sizes of below $6 \cdot 10^6$ particles, CCCCHG yielded the highest speedups whereas for larger problems CCCGHG was fastest. We attribute this observation to the fact that GPUs only reach their maximum performance if all computational resources are fully utilized. As the GPU on type III is highly capable, this was only achieved for problems with a higher number of particles. As expected, we obtained different





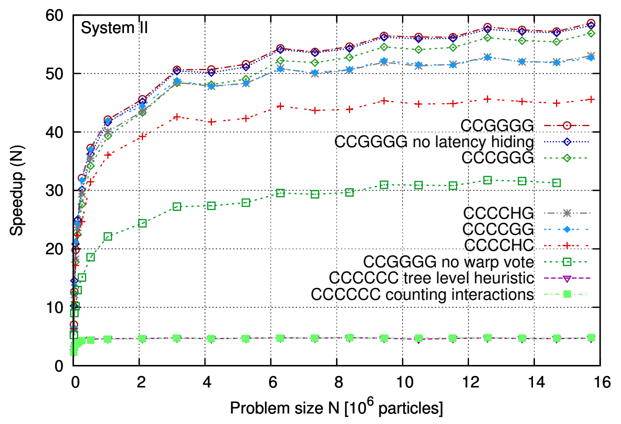


Fig. 3. Speedup of different parallel module combinations compared to purely sequential CPU implementations. Timings averaged over 10 time steps using $\theta = 0.75$.

results among our test systems, and as such a variety of systems is common in, e.g., Desktop Grids, the results prove our idea of a flexible modular design.

The modern GPU on type II supports the warp vote function `__all`, which decreased the time of the force evaluation by over 50 %. Using the hybrid force evaluation module (CCCCHG) on type I, we measured a speedup of over 20 % compared to configuration CCCCGG. On type III, CCCGHG was over 50 % faster than CCCGGG. This shows that combining the computational power of CPU and GPU is an effective way to increase the performance of simulations.

Memory latencies constituted only about 1 % of the total running time on type II in configuration CCGGGG. Through our latency hiding approach, we reduced latencies by 50 % to below 0.5 %. On type I in configuration CCCCHG, latencies were reduced by over 66 % to less than 0.25 % of the total time.

5 Related work

In most works, only the computationally most expensive part, the force evaluation, is computed on the GPU, whereas the CPU computes all other steps of the Barnes-Hut algorithm [12, 7]. More recent works run the whole simulation on GPUs [6, 4]. In contrast, our code is completely modularized to allow arbitrary combinations of CPU- and GPU-based implementations of the individual steps.

The Barnes-Hut tree is an irregular structure that does not easily fit the parallel SIMD model of GPU programming. Construction and traversal of trees on GPUs are topics of ongoing research. Two popular approaches to parallelize construction of trees on GPUs have been proposed in literature: Tree construction can either be accomplished by inserting particles in parallel into a dynamically changing shared tree using locks to prevent race-conditions [6] or the tree can be constructed level by level, which requires particles to be sorted [4].

GPU-based force evaluation can be realized executing one thread per particle. As recursive functions are not supported on older GPU generations, tree traversals have to be done iteratively. Stack-based tree traversals were used in [7, 4, 6]. Tree traversals using next and more arrays were first employed in [11] and later were used on the GPU [12], as in our work. To prevent thread divergence in SMs, force evaluation can be modified to use interaction lists, as first described in [3]. They were used in GPU-based force evaluation in [7, 4]. Instead, like [6], we use warp vote functions to prevent thread divergence.

Our modular design offers opportunities for latency hiding every time execution moves to another platform. As most works employ non-modular designs, little information can be found on this topic. We are aware of only one work discussing latency hiding between GPU and CPU computations. In [10] the CPU determines interaction lists and the evaluation of forces is offloaded to the GPU. However, interaction lists are computed piecewise, so that completed interaction lists can be sent to the GPU while the CPU computation continues.

6 Conclusion

In this paper we introduced a modularized parallelization of the Barnes-Hut algorithm. By defining interfaces between modules and carefully choosing data structures we facilitate efficient module implementations for CPU and GPU that allow a flexible dynamic assignment to platforms. Through the design of hybrid modules that combine the computing power of CPU and GPU, we fully utilize all available computational resources.

Our test results prove that for different host systems very different combinations of GPU- and CPU-based modules yield the best overall performance, and that the best combination depends highly on the underlying hardware. Hence, by incorporating our modular design, we are able to improve an existing implementation of the Barnes-Hut algorithm for Desktop Grids [8]. The flexibility and adaptability our modular multi-platform approach offers, render it an ideal model for the design of future algorithms for heterogeneous environments.

References

1. D. P. Anderson. Boinc: A system for public-resource computing and storage. In *5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, 2004.
2. J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(6096):446–449, Dec. 1986.
3. J. E. Barnes. A modified tree code: Don't laugh; it runs. *Journal of Computational Physics*, 87(1):161–170, 1990.
4. J. Bédorf, E. Gaburov, and S. P. Zwart. A sparse octree gravitational n-body code that runs entirely on the GPU processor. *Journal of Computational Physics*, 231(7):2825–2839, April 2012.
5. J. J. Billings. Starscream – An open source galaxy modeling and simulation tool. `http://jayjaybillings.org/starscream/`. (accessed in July, 2012).
6. M. Burtscher and K. Pingali. An efficient CUDA implementation of the tree-based Barnes Hut n-body algorithm. In W.-m. W. Hwu, editor, *GPU Computing Gems Emerald Edition*, chapter 6, pages 75–92. Morgan Kaufmann Publishers Inc., 2011.
7. E. Gaburov, J. Bédorf, and S. P. Zwart. Gravitational tree-code on graphics processing units: Implementation in CUDA. *Procedia CS*, 1(1):1119–1127, 2010.
8. H. Hannak, W. Blochinger, and S. Trieflinger. A desktop grid enabled parallel barnes-hut algorithm. In *Proc. of the 31st IEEE International Performance Computing and Communications Conference (IPCCC 2012)*, Austin, Texas, USA, 2012. IEEE Computer Society.
9. E. Heien, D. Kondo, and D. Anderson. A correlated resource model of internet end hosts. *Parallel and Distributed Systems, IEEE Transactions on*, 23(6):977 –984, june 2012.
10. P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kalé, and T. R. Quinn. Scaling hierarchical n-body simulations on GPU clusters. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, page 1–11, 2010.
11. J. Makino. Vectorization of a treecode. *Journal of Computational Physics*, 87(1):148–160, Mar. 1990.
12. N. Nakasato. Implementation of a parallel tree method on a GPU. *Journal of Computational Science*, 3(3):132 – 141, 2012.
13. S. Schulz, W. Blochinger, M. Held, and C. Dangelmayr. Cohesion - a microkernel based desktop grid platform for irregular task-parallel applications. *Future Generation Computer Systems - The International Journal of Grid Computing: Theory, Methods and Applications*, 24(5):354–370, 2008.
14. J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy. Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity. *Journal of Parallel and Distributed Computing*, 27(2):118–141, June 1995.