Data Structure Design for GPU Based Heterogeneous Systems

Jens Breitbart
Research Group Programming Languages / Methodologies – Universität Kassel
jbreitbart@uni-kassel.de

ABSTRACT

This paper reports on our experience with data structure design for systems having both multiple CPU cores and a programmable graphics card. We integrate our data structures into the game-like application OpenSteerDemo and compare our data structures on two pc-systems. One System has a relative fast single core CPU and slower GPU, whereas the other one uses a high-end GPU with a slower multi core CPU. We design two grid based data structures for effectively solving the k-nearest neighbor problem. The static grid uses grid cells of uniform size, whereas the dynamic grid does not rely on given grid cells, but creates them at runtime. The static grid is designed for fast data structure creation, whereas the dynamic grid is designed to provide high GPU simulation performance. The high performance is achieved by taking advantage of the GPU memory system at the cost of a more complex construction algorithm. Our experiments show that with a slower CPU the algorithm for creating the dynamic grid becomes the bottleneck and no overall performance increase is possible compared to the static grid. This also holds true when the simulation is run with a faster CPU and a slower GPU, even though the break-even point is different. We experimented with data structure creation on the GPU, but the performance of the static grid is not feasible. The dynamic grid cannot be created on the GPU due to the lack of recursive function support. We provide a dynamic grid creation algorithm, which uses multiple CPU cores. This algorithm is slower than its sequential counterpart due to the parallelization overhead.

KEYWORDS: GPGPU, k-nearest neighbor, games, OpenMP, CUDA

1. INTRODUCTION

Computer games are one of the most compute intense applications for end user and their demand in processing power increases with every generation. Their current performance needs can hardly be satisfied with the slow increase in single core performance. There are two hardware development trends, which provide end-user systems with an additional increase in computing resources. Additional CPU core are added to a single chip, so the overall performance of the CPU is increasing at a high rate, even though when single core performance cannot be increased. Furthermore the programmability of the last generation of graphics processing units (GPUs) has reached a level at which they can be programmed without requiring any knowledge of graphics APIs like e.g. OpenGL and can thereby easily be used as an additional computing resource. The additional processing power for both multiple CPU cores and programmable GPUs cannot be achieve by recompiling the application, but require a change in both software architecture and algorithm design.

This paper reports on our experience with the modification of a game-like application called *OpenSteerDemo* to use both multiple CPU cores and a programmable GPU. We outline problems and solutions that occur during development. OpenSteerDemo is written in C++ and is the demo application of the OpenSteer steering library. *Steering* refers to life-like navigation of autonomous characters, so-called *agents* used for instance in computer games [1]. Each agent follows a so called *steering behavior*, which is solely based on the local environment of the agent. OpenSteerDemo is used to simulate different kinds of steering behaviors; each of them is implemented in a separate plugin. The software design of OpenSteerDemo and its plugins is similar to that of games. It runs a main loop calculating the steering behaviors and drawing the agents to the screen. We worked with a plugin called *Boids*, which simulates flocking. Agents in this scenario compute their seven nearest neighbors' and decide based on the neighbors' positions where to move next.

The original plugin relies on OpenMP [2] to support multiple CPU cores [3]. OpenMP is a thread based programming system using pragmas to allow easy work

distribution among threads. We continue to use OpenMP for our work as the pragmas can be easily integrated into existing applications and only require small changes of the existing code.

We first develop a plugin utilizing the GPU for the simulation and then design two spatial data structures used at the GPU to increase performance. Both data structures partition the world into cuboidal parts (*cells*) and store which agents are within a cell. This information is used to speed up the neighbor search, which is the most time-consuming part of the simulation. The first data structure – called *static grid* – relies on given grid cell with a fixed size, whereas the so called *dynamic grid* creates cells at runtime based on the current position of the agents. The static grid requires less CPU processing power than the dynamic grid, but provides slower simulation performance on the GPU. Choosing the best performing data structure depends on both the available CPU/GPU processing power and the number of simulated agents. Data structure creation becomes the performance bottleneck for all our test systems when reaching a certain number of agents. We also experimented with data structure creation with multiple CPU core or the GPU, but the results were not practical usable.

We program the GPU with NVIDIAs CUDA [4] even though it only supports NVIDIA GPUs. The underlying concept of CUDA strongly resembles that of OpenCL [5], which is expected to support GPUs from different vendors, so our solution could easily be implemented with OpenCL as well. We cannot use OpenCL, as compilers are not available to public at the time of writing.

The paper is organized as follows. First, Section 2 gives a brief overview of the used programming systems and describes the differences between GPUs and CPUs. The next Section (Section 3) gives an introduction to the architecture of OpenSteerDemo and the Boids plugin. Section 4 describes the parallelization approach of the existing multi core plugin and shows that a similar approach can be used with CUDA as well. The main part (Section 5) explains the design of spatial data structures. Section 6 gives a final performance overview of all developed plugins. Section 7 discusses related work, while Section 8 summarizes the paper and gives a brief outline of possible future work.

2. PROGRAMMING SYSTEM

We used OpenMP throughout our work to support multiple CPU cores. OpenMP is a programming system designed for shared memory architectures and uses threads. The OpenMP thread creation relies on a fixed fork-join-structure. Work that should be executed in parallel must be embedded within a parallel region. At the start of a parallel region a number of threads, which execute whatever code is embedded within the region, is created. At the end of the parallel region all created threads must be joined. Using a parallel region thereby imposes some overhead. Access to variables shared by multiple threads must be synchronized. OpenMP 3.0, which was released in May 2008, added support for non-regular task parallelism with the so called task construct. When a thread reaches a task construct it may decide to directly execute the embedded code or may skip this code and put the task into a work queue, which is being worked at by all threads [2]. A detailed overview of OpenMP can be found in Chapman et al. [6].

GPUs are not designed to be used for sequential computations and consist of hundreds of processors for which one cannot provide reasonable performance. This concept is difference to that of CPUs and necessitates new programming systems. During our work we used NVIDIAs CUDA, which provides the power of the GPU in the C programming language. The CUDA information presented in the rest of this section is based on [7] if not explicitly stated otherwise.

NVIDIAs CUDA is a general-purpose programming system only available for NVIDIA GPUs and was first publicly released in the end of 2007. By using CUDA, the GPU (called *device*) is exposed to the CPU (called *host*) as a co-processor with its own memory. The device executes a function (called *kernel*) in the SPMD model, which means that a user-configured number of threads run the same program on different data. Threads executing a kernel must be organized within so called *thread blocks*, which may consist of up to 512 threads; multiple thread blocks are organized in a *grid*, which may consist of up to 2^{32} thread blocks. One thread block is always scheduled onto one so called *multiprocessor* of the device. One multiprocessor consists of 8 processors. The number of multiprocessors of a device depends on hardware used. The current maximum of multiprocessors on a single device is 30. Furthermore thread blocks are important for algorithm design, as only threads within a thread block may be synchronized and synchronization of threads within different thread blocks is not possible. NVIDIA suggests having at least 64 threads in one thread block and up to multiple thousands of thread blocks – and thereby more threads than the device has processors – to achieve high performance at the device.

In contrast to main memory used by the CPU, its GPU counterpart – called *global memory* – is not cached and accessing it costs an order of magnitude more than most

calculations. For example, 32 threads require 400 - 600 clock cycles for a read from global memory to complete, whereas an addition executed by the same amount of threads takes only 4 clock cycles. Due to the high cost for reading data from global memory, the device offers multiple ways to circumvent this overhead. The device uses an efficient thread scheduler that uses the massive parallelism approach of the device to hide the latency by removing threads that issued a global memory read from its processor and scheduling a thread that is not waiting for data. This is one of the reasons why the device requires more threads than there are processors available to achieve good performance. Another way of reducing global memory accesses is by using a special kind of memory called *shared memory*. Shared memory is fast memory located on the multiprocessors of the device itself and is shared by all threads of a thread block. Accessing shared memory cost about 4 clock cycles for 32 threads and may be used as a developer managed cache. Global memory usage cannot be circumvented, since this is the only kind of memory, which can be accessed by both host and device. Data that is stored in main memory must be copied from main memory to global memory by a CUDA memcopy like function call, if it should be accessed by the device. Results of a kernel must be stored in global memory and the CPU must issue a memcopy from global memory to main memory to use them. All transfers done by CUDA memcopy functions are DMA transfers and have a rather high cost of initialization and a rather low cost for transferring the data itself. See figure 1 for an overview of the CUDA memory model. We use only registers, shared memory and global memory in our implementations.

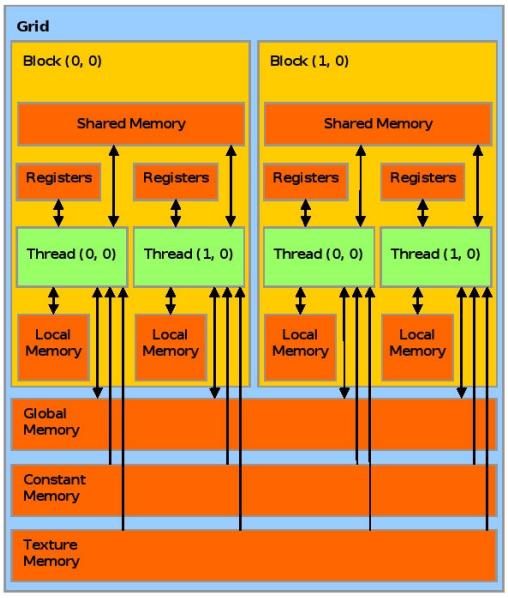

Figure 1. CUDA Memory Model [7]

We use CuPP [8] to ease the integration of CUDA into OpenSteerDemo. CuPP is a framework we explicitly designed to ease the integration of CUDA into C++ applications. It provides techniques freeing the developer from manually transferring data from main memory to global memory and vice versa. We use a STL vector like data structure provided by CuPP that makes the data stored automatically available at both host and device. The CuPP vector monitors if e.g. the device changes the data and then automatically updates the host data as soon as it is accessed. For example, if a CuPP vector is filled with data by the host and then only used by the device, only one memory transfer transferring the initial data to global memory will be issued. We use the CuPP vector for all data accessed by both host and device, if not explicitly stated otherwise.

CuPP furthermore provides a technique called *type transformations*, which allows the developer to use two data representations for the same data on host and device. CuPP transforms the data from one representation into the other, when transferring from one memory domain into the other. The transformation is done by the CPU. We call the type used at the CPU *hosttype*, whereas the type used at the GPU *devicetype*. The type transformations are used in section 5.1 to provide the CPU with a data representation that can be created effectively, whereas the devicetype allows fast transfer to global memory. A detailed description of CuPP can be found in [9].

3. OpenSteerDemo ARCHITECTURE

OpenSteer [10] is a C++ open-source library written by Reynolds in 2002. It provides simple steering behaviors and a basic agent implementation. OpenSteerDemo is the demo application of OpenSteer. The Boids plugin is a plugin for OpenSteerDemo, which simulates flocking [1] in a three dimensional world. All agents in the Boids plugin can move freely across a finite spherical world. If one agent leaves the world at one side, it is put back into the world at the diametric opposite of its original position. The calculation to determine where the agent wants to move next is only based on its current state – e.g. its speed – and its seven nearest neighbors.

The following architecture of the Boids plugin was developed by Knafla and Leopold [3]. The simulation done by the Boids plugin can be divided in two stages. First the new state of all agents is calculated (called *update stage*) and then drawn to the screen (called *draw stage*). The update stage itself is again divided into two substages. The first substage is called *simulation substage* and includes the steering calculations and the search to identify the 7 nearest neighbor agents. The algorithm used

to find the 7 nearest neighbors of one agent is a fairly simple *O(n)* algorithm, which searches through all agents and returns the 7 nearest ones. The results of the simulation substage are vectors representing the direction and speed every agent wants to move. These vectors are used in the next substage called *modification substage* to update the position of every agent. The draw stage is executed after the modification substage and draws the new agent positions to screen. The design of OpenSteerDemo itself is similar to the ones of games. It runs a main loop executing first the update stage and then the draw stage. The main loop is part of the OpenGL Utility Toolkit (known as *GLUT*) and the stages are functions, which are called by GLUT.

4. PARALLEL BOIDS PLUGIN

Exposing the parallelism of the calculation of the Boids plugin to use multi core CPUs was already done by Knafla and Leopold [3]. The implementation uses OpenMP and splits all agents equally among the threads. A thread calculates both the simulation and modification substage for the agents associated with. In the simulation substage the agent's position are read, whereas in the modification substage the positions are changed, so both substages must not be carried out in parallel. Barrier synchronization is used to prevent this. Knafla and Leopold demonstrate that their parallelization approach works well on multi core systems and provides an almost linear speedup regarding the update stage. The speedup of the overall application is not linear as the draw stage is still executed sequentially.

The first version we developed to incorporate the GPU is based on the multi core plugin developed by Knafla and Leopold. We replaced the original used STL C++ vectors with CuPP vectors to free us from the need to manually transfer data from main memory to global memory or vice versa. Analysis of the memory transfers done by CuPP shows that data is only transferred when it is must be transferred – meaning when the data stored on the device or the host is out of date. A detailed description of how CuPP achieves this functionality can be found in [9]. The code running at the GPU is mostly just a copy and paste work of the original OpenSteer code, except for the modifications outlined next.

Our parallelization approach at the device is similar to what Knafla and Leopold proposed for multi core CPUs and only differs in detail. Instead of having one thread calculate multiple agents, we use a separate thread for each agent and thereby can provide the device with a high number of threads. We use the GPU to calculate the complete update stage and use the CPU only for the draw stage. By using this approach, we only need to transfer the initial data to global memory at the beginning of the simulation and do not need to update the data at the GPU. The only data that must be transferred in every simulation step is a matrix representing the position and orientation of the agents, as this is used by the draw stage.

Synchronization of all threads within a kernel is not possible, so we must use two kernels, one for each substage. There are no data dependencies between the agents in one substage, so we do not need to guarantee any order of how the threads are put into the thread blocks. During the neighbor search all threads must access the position of all agents. We use shared memory to cache position data. We load chunks of position data from global memory into shared memory, have all threads searches for neighbors in them and then continue with the next chunk. A detail description of this technique and technical details regarding the implementation can be found in [11]. We refer to this plugin as the *basic* plugin. The basic plugin can simulate about four times the number of simulation step per second compared to the OpenMP based one; it is possible to simulate about 10240 agents at 24 frames per second (*fps*).

5. NEIGHBOR SEARCH WITH SPATIAL DATASTRUCTURES

The basic plugin does not use an efficient algorithm for the neighbor search as every agents needs to look at all other agents to find its neighbors. We now describe a spatial data structure called *grid* that we use to speed up neighbor search.

A grid subdivides the world into small areas, which we call *cells*. Agents are assigned to cells based on their current position, so one cell contains all the agents that are within its range. A grid can be used to improve the neighbor search performance, as one agent does not need to look at all other agents to find its neighbors, but only at the agents stored in the cells within its search radius. The search inside a cell is done with the brute force approach described before.

5.1. Static Grid

We refer to our grid implementations shown in this section as *static grid*. The term static was chosen to distinct this solution to the dynamic grid demonstrated in the next section and indicates the way cells are created. The static grid subdivides the world in cubic cells all of them the same size. The number of cells cannot be changed after a grid has been created and is identical for each dimension, so the overall shape of the static grid is a

cube as well. The dynamic grid on the other hand creates cells with different sizes dynamically.

We provide for two different implementation of the static grid – one creates the grid at the CPU and transfers it to global memory, whereas the other one directly uses the GPU to create the grid and thereby does not need to transfer any data to global memory.

We use the CuPP type transformations to work with two different data representations for the CPU created static grid. The creation of the grid is done before the simulation substage is executed and redone for every simulation step – meaning we never update the grid, but clear and refill it with new data in the next simulation step. We choose this way for simplicity, however we do not expect updating an existing grid to be more efficient than creating a new one.

The hosttype of the static grid is an aggregation of multiple C++ STL vectors, each vector represents a cell. Cells store the agent indexes of the agents within the range of a cell. All cell vectors are stored in another vector, so the grid itself is a vector of vectors storing agent references. The benefit of this approach is that adding elements to the grid is a O(1) operation. To add an element we must calculate the index of the cell vector and append the element. Appending an element to a C++ STL vector is guaranteed to be a O(1) operation, when no memory reallocation is done. To prevent unneeded memory reallocations, we clear the used C++ STL vectors instead of creating new ones. C++ vectors never free memory already allocated so after the agents are distributed equally throughout the world, the cell vectors hardly need to grow beyond their current size.

Based on our previous experience described in [9] and that memory transfers to global memory are DMA transfers, we expect transferring one large memory block to be preferred over transferring multiple smaller memory blocks. We designed the devicetype to consist of only two independent memory blocks. One memory block contains the data of the cell vectors ordered by their index (called *data memory block*) and the other one (called *index memory block*) contains the indexes to find the cell vectors within the first memory block.

Transferring the hosttype to global memory would require one memory transfer per cell, whereas the devicetype requires two memory transfers to transfer all data. Transforming the hosttype into the devicetype is a *O(n)* operation, as we have to copy all *n* agent-references stored in the hosttype into a new continuous memory block. Creating the index memory block is a *O(k)* operation, with k being the number of grid cells. Creating the devicetype is therefore a *O(n+k)* operation.

Our GPU constructed static grid only uses the devicetype. The creation itself is split into three distinct steps. Each step is implemented in a separate kernel to guarantee synchronization between the steps. The first two kernels are used to build up the index structure, whereas the last kernel fills the data memory block. We describe the three steps of our algorithm next.

Count The count kernel counts, the number of agents, which must be stored within each grid cell, and saves the results within the index memory block. The count kernel uses one thread per grid cell. The threads are distributed among multiple thread blocks. Each thread looks at all agents and counts the number of agents within its grid cell boundaries. We use shared memory as a cache for agent data. The results are written to the index memory block.

Scan The scan kernel calculates the start position of each cell within the data memory block by issuing an exclusive scan on the index memory block. Scan is also known as parallel all-prefix-sums done on an array. Scan uses a binary associate operator Δ with the identity I and an array of n elements as input

$$[a_0, a_1, ..., a_{n-1}]$$

and returns the array

$$[I, a_0, (a_0 \Delta a_1), ..., (a_0 \Delta a_1 \Delta ... \Delta a_{n-2})]$$

as a result. Our implementation is based on one provided by NVIDIA, which is discussed in [12]. Our kernel executes the all-prefix-sums on the index data using addition as the binary operator, so the index data at position 0 contains a 0, position 1 contains the number of agents to be stored in the 0th cell, and position 2 contains the number of agents to be stored in both the 1st and the 0th cell and so on.

Fill The fill kernel fills the data memory block with the references to the agents. We use one thread per grid cell to store agent references in the data memory block. All threads scan through all agent positions and write the agents' index to the data memory block, if the agent is within grid cell of the current thread. The position, to which the agent references should be written in the data memory block, is based on the values stored in the index data structure and the number of agents already belonging to the cell. We use shared memory as a cache.

Executing these 3 steps after one another creates the device type of the static grid on the device. The benefit of this solution is that there is no need to transfer any data to global memory for the simulation – except for the first simulation step, at which we transfer the initial data of the agents to the device. We only transfer the data required to issue the draw calls back to main memory.

We can use shared memory as a cache in the first GPU based plugin, because all agents simulated by one thread block look at the same agents to find their neighbors. The plugins using the static grid cannot use shared memory as a cache for agent position data, as agents of a thread block are not guaranteed to have any common data requirements. Agents are put in thread blocks without any order, so the agents located in one grid cell are distributed throughout the CUDA grid. Reading from global memory is one of the most expensive operations on the device so we expect this to reduce performance of the static grid based implementation, however the profiling tools currently available do not allow us to explicitly measure the time required in chosen code regions.

Introducing the static grid into our application increases the performance of the simulation by a factor up to 35 compared to the plugin using no spatial data structure. Creating the grid on the device is slower than creating it on the CPU and transferring it to global memory, so using the GPU to create the static grid does not increase performance. Achieving good performance with the static grid requires finding an appropriate number of cells to be used by the simulated scenario. If a *wrong* number of cells is used, the performance may be reduced by up to 90%. A detailed overview of the performance can be found at section 6.

5.2. Dynamic Grid

The plugin using the static grid cannot use shared memory to cache global memory accesses, because the agents within one thread block have no common data requirements. We now propose a new mapping scheme to solve this issue and thereby increase performance in some scenarios. We continue to use one thread per agent, but map a group of agents close together to one thread block. A group of agents close together must look at roughly the same agents to find their neighbors, so we can use shared memory to store chunks of agent position data in shared memory. We use this mapping scheme for the simulation kernel, but continue to use the old scheme in the modification substage as there would be no benefit from using the new scheme.

Combining the new mapping scheme with the static grid is complex. We could try to map one grid cell to one thread block, but the number of agents in one grid cell varies between 0 and n – with n being the number of agents currently simulated. This variant causes two problems.

- The number of threads per thread block is fixed and limited to a maximum of 512. If we want to simulate more than 512 agents with one thread block, we must simulate multiple agents per thread, which is possible but requires a complete redesign of the kernel.
- One grid cell could contain all agents. If this would be the case, the whole simulation is executed by one multiprocessor, which leads to a poor work balance at the device.

We solve these problems by introducing a new data structure called *dynamic grid*.

In contrast to the static grid discussed in the last section, the dynamic grid relies not on given grid cells, but creates them on the fly. A grid cell of the dynamic grid occupies a cuboidal part of the world. All grid cells can differ in size, but have a common maximum number of agents within its borders. We call the maximum number of agents in one grid cell *max* throughout the rest of this section. In our case, *max* is identical to the number of threads per thread block used to execute the simulation kernel. This restriction is required, as we map one grid cell to one thread block. All thread blocks simulating less than *max* agents, have idle threads.

Despite allowing the usage of shared memory on the device, the dynamic grid also automatically adopts to the simulated scenario, so there is no more need to manually choose the number of grid cells.

The internal data structure of the dynamic grid consists of two vectors. One vector – called *data vector* – stores tuples of agent positions and agent reference for all agents. The second vector – called *cell vector* – stores the dimension of the grid cell, its position and which agents are within the cell. The algorithm to create the dynamic grid guarantees that the agents of one cell are stored continuously within the data vector, so we only need to store the first and the last agent within the cell to identify all agents of the cell. The algorithm to create the dynamic grid is split into 2 steps.

- Fill the data vector.
- Recursively partition the data vector in a way that agents stored next to each other are close together in the simulated world. A partition with $\leq max$ agents is a cell.

In the first step the data vector is filled in an unordered fashion with pairs consisting of both the agent position and a reference to the agent itself.

The second step in our algorithm is similar to Quicksort. It recursively subdivides and partitions the data vector. The partitioning of the agents is done by one of three dimensions of the simulation. The algorithm to choose the dimension is based on practical experiments. The dimension is chosen at runtime by first calculating the distance from the center point of all agents in the current partition to the border of the space covered by the partition. Afterwards we partition alongside the dimension with the minimal distance to the border. The algorithm stops to subdivide a partition as soon as the number of agents is $\leq max$.

Creating the dynamic grid is done on the host, because the device does not support recursive functions. However, we can use multiple CPU cores to construct a dynamic grid in parallel. The parallel algorithm uses OpenMP tasks. Each recursive subdivision of a partition is a task, until the size of the partition reaches a certain threshold. We stop at a certain threshold to prevent the overhead generated by the OpenMP task construct for small tasks. Synchronization is only required to ensure that all tasks are completed and when a partition contains $\leq max$ agents. At this point we must take care that not multiple threads add a cell into the cell vector at the same time.

The performance of the dynamic grid strongly depends on the used system. The dynamic grid requires more CPU processing power and less at the GPU. On a system with a rather slow CPU and a fast GPU the performance is decreased compared to the performance of the static grid, whereas on a system with a faster CPU and a slower GPU the performance is increased. The next section gives more details of the performance on both kinds of systems.

Table 1. System Specification

	System I	*System II*
CPU	AMD Athlon 64 3700+ (2,4 GHz)	2 x AMD Opteron 270 (2 x 2 x 2 GHz)
GPU	GeForce 8800 GTS (640 MB)	GeForce GTX 280 (1 GB)

6. PERFORMANCE

The specifications of the systems used to benchmark our plugins can be found in Table 1. System I uses a faster single core CPU and a slower GPU compared to System II, which uses two dual core CPUs and one of the fastest GPUs currently available.

When simulating 2^{17} agents on System I the static grid provides about 14 simulation steps per second, whereas the basic plugin can only simulate 0.4 fps per second. Experimenting with the static grid at System II shows that for up to 2^{15} agents the performance of System II is superior to that of System I; however with more agents the creation of the grid becomes more time consuming. System I provides about 1.4 times the performance of System II when simulating 2^{16} agents.

Creating the static grid at the GPU is no feasible option for System I as both the count and fill kernel require more time than creating the grid at the CPU and transferring it to GPU memory. System II provides better performance, but both kernels are not faster than creating the data structure at the CPU and copy it to GPU memory. The GeForce GTX 280 of System II provides additional functionality like atomic operations, which may be used to design a faster algorithm at the cost of lost compatibility.

The dynamic grid was designed to reduce the runtime of the kernel at the GPU at the cost of a high CPU utilization. The overall performance of the dynamic grid plugin is better than that of the static grid plugin for up to about 2^{15} on System I. Experiments with the parallel creation of the grid show that this is not practical for the amount of agents that can be simulated in real time. On System II it takes about 0.08 seconds to create a dynamic grid for 40960 agents in one simulation step with one thread. Running the code with 4 threads doubles the time required to construct the grid. The performance lost is resulted from the overhead of both the OpenMP task construct and creating and joining the threads for every simulation step. We cannot prevent the reoccurring creating and joining of the threads, as the simulation is implemented in functions that are repeatedly called by GLUT.

7. RELATED WORK

PSCrowd by Reynolds [13] simulates 15.000 agents in a similar scenario to the Boids plugin at the Playstation 3 (*PS3*). He uses the PowerPC processor of the PS3 to construct the spatial data structure and the Synergistic Processor Units (SPUs) to execute the calculation for all agents. PSCrowd uses a technique called SkipThink, which only simulates a fraction of the agents in one simulation step, but still provides a reasonable overall result. In contrast to Reynolds we work on a different platform and concentrate on data structure design instead of the overall implementation of a crowd simulation. Lauterbach et al. [14] have developed two algorithms to

construct bounding box based algorithms on modern GPUs or other many core architectures. The performance of their implementation with CUDA is similar to that of CPU based implementation. Lauterbach et al. say that the performance of their algorithm should increase, as soon as GPUs offer higher flexibility e.g. recursive function calls or better synchronization primitives. The work of Lauterbach et al. is focused on ray tracing and relies on heuristics that may not work well in our scenario.

8. CONCLUSION / FUTURE WORK

In this paper we show our experience of how to take the full benefit of current end user systems with a focus on how to include programmable GPUs. Our work with different data structures shows that data structure design should not necessarily be designed for maximum performance when it is used, but also that data structure creation itself may easily become the performance bottleneck. This experience by itself is not novel, but if the GPU is used for calculations the break even point may come sooner than expected, especially with current high-end GPU providing almost one teraflop of processing power. Using the GPU for data structure creation is not a good option when the first generation of CUDA capable GPUs should be supported. We expect our results to be valid for OpenCL as well, as it strongly reassembles the programming model of CUDA. However, as OpenCL is supposed to support a wide range of different hardware, the performance and break even point of all implemented may vary. Furthermore upcoming hardware, like Intel's Larrabee [15], which is expected to provide reasonable sequential performance, may be used to construct the data structure.

Future work on OpenSteerDemo could try to expose the functionality of the latest generation of GPUs to create the data structures and thereby free the CPU for other calculations. Furthermore experiments with a more flexible multi core programming system could possible be used to effectively create the data structure in parallel. It may also be useful to exploit the parallelism of being able to use the GPU and the CPU at the same time.

ACKNOWLEDGEMENTS

The author is grateful to Claudia Fohry for some several hints on presentation. The author also thanks NVIDIA for providing the graphics card used in System II.

REFERENCES

[1] C. W. Reynolds, "Steering behaviors for autonomous characters," Game Developer Conference, 1999, pp. 763-782.

[2] *OpenMP Application Program Interface,* 2008, version 3.0.

[3] B. Knafla and C. Leopold, "Parallelizing a real-time steering simulation for computer games with OpenMP," PARCO, IOS Press, 2007, pp. 219-226.

[4] *CUDA website,* 2009. Available: http://www.nvidia.com/cuda

[5] *OpenCL Specification,* 2009, version 1.0.

[6] B. Chapman, G. Jost, and R. v. d. Pas, USING OpenMP: PORTABLE SHARED MEMORY PARALLEL PROGRAMMING (SCIENTIFIC AND ENGINEERING COMPUTATION), The MIT Press, 2007.

[7] *NVIDIA CUDA compute unified device architecture programming guide version 2.1,* NVIDIA Corporation, 2008.

[8] *CuPP website,* 2009. Available: http://www.plm.eecs.uni-kassel.de/plm/index.php?id=cupp

[9] J. Breitbart, "CuPP – A framework for easy CUDA integration in C++ applications," presented at the 2009 IEEE Int. Parallel & Distributed Processing Symposium, Rom.

[10] *OpenSteer website,* 2009. Available: http://opensteer.sourceforge.net

[11] J. Breitbart, "A framework for easy CUDA integration in C++ applications," Diplome thesis, Universität Kassel, Kassel, Germany, 2008.

[12] M. Harris, S. Sengupta, and J.D. Owens, "Parallel prefix sum (scan) with CUDA," in GPU GEMS 3, H. Nguyen, Ed. Addison Wesley, 2007, ch. 39, pp. 851-876.

[13] C. W. Reynolds, "Big Fast Crowds on PS3," SIGGRAPH Symp. On Videogames, 2006.

[14] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH construction on GPUs," Europgraphics, 2009.

[15] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A Many-Core x86 Architecture for Visual Computing", SIGGRAPH, 2008