Using common graphics hardware for multi-agent traffic simulation with CUDA

David Strippgen and Kai Nagel
Transport Systems Planning and Transport Telematics (VSP)
TU Berlin, Salzufer 17-19, Sekr. SG 12, 10587 Berlin, Germany
{strippgen,nagel}@vsp.tu-berlin.de

ABSTRACT

Today's graphics processing units (GPU) have tremendous resources when it comes to raw computing power. The simulation of large groups of agents in transport simulation has a huge demand of computation time. Therefore it seems reasonable to try to harvest this computing power for traffic simulation. Unfortunately simulating a network of traffic is inherently connected with random memory access. This is not a domain that the SIMD (single instruction, multiple data) architecture of GPUs is known to work well with. In this paper the authors will try to achieve a speedup by computing multi-agent traffic simulations on the graphics device using NVIDIAs CUDA framework.

Categories and Subject Descriptors

I.6.8 [**SIMULATION AND MODELING**]: Types of Simulation—*Parallel*; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

1. INTRODUCTION

Over the last decade, the graphic cards found in common home PCs have evolved from mere display devices over 3D rendering devices to today's generally programmable multicore devices. There has been some research on harvesting the computational power of GPUs[19] [2] [17] [6] based on OpenGL and DirectX. But it proved to be rather cumbersome to express general algorithms in terms of textures and 3D operations. Also, the absence of any other data primitive than float numbers has been a drawback for many possible applications. Lately relevant graphics device companies (NVIDIA and ATI/AMD) have come up with frameworks (SDKs) to program GPUs for general problems. These SDKs are named FireStream[1] and CUDA[7]. Nowadays in the presence of these SDKs the premises have changed rather dramatically. It has become feasible to take a given –CPU-based– algorithm and convert it for GPU execution rather straight away.

SIMUTools 2009 Rome, Italy

Problem domains like fluid simulation or molecular dynamics, where a small amount of code has to be run on a huge amount of independent data, successfully adopted the GPU as their computational needs intrinsically fit into the SIMD architecture of the GPU.

Simulation of networks –or, as in this paper, of traffic networks– have in common that they move entities over a randomly connected network. This highly depends on the use of random access memory and dynamic data structures. Thus, the architecture of GPUs is seemingly not the best fit for their demand. Nevertheless the GPU still has remarkably fast access to the main memory, albeit without a big cache. On modern CPUs sophisticated caching techniques help to speed up random memory accesses. The CUDA framework tries to mask the lack of caching by computing other threads, while one thread has to wait for a global memory access.

Therefore it might be possible to benefit from the multicore architecture and the fast memory, though it might not carry the high yield that other domains can achieve.

In the further paper the NVIDIA Framework CUDA is used for implementing a traffic simulation. The outline of this paper is as follows: After presenting the related work in the next section we will recall some fundamental facts about the hardware used as well as the queue traffic simulation that will be implemented in section 3. In the fourth section we will describe the various different data structures and variations of the queue simulations algorithm we benchnmarked. The results of the benchmarking will be presented in section 5 and a conclusion will be given in the last section of this paper.

2. RELATED WORK

The MATSIM[22] framework has been used thoroughly in large scale traffic simulations. It is known to deliver plausible results in terms of every day traffic. The MATSIM framework uses genetic algorithms to simulate typical weekday traffic. It starts with an initial demand, computed from several data sources, e.g. census data, questionnaires. This initial demand consists of complete activity chains for all agents for the whole day. This demand is then executed with a mobility simulation, summing up the experienced travel times delays and times of activity for every agent. After each iteration, new plans are being calculated, based on the results of the simulation run. Therefore a fitness function, which determines how the experienced travel/activity times should be rated, is defined and genetic algorithms generate new mutations of the executed plans with respect to this fitness function. The new plans will be executed in the sim-

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5666
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5666

GeForce 8600 GT	
Num. cores	32
Clock rate	1.18GHz
Mem bandwidth	7638 MB/s
GeForce GTX280	
Num. cores	240
Clock rate	1.3GHz
Mem bandwidth	120 GB/s

Table 1: Technical data of GPUs used

ulation again. This process converges to a Nash equilibrium. The range of iterations necessary for the system to move towards a Nash equilibrium can vary from a few iterations up to several hundreds. Therefore reducing the execution time of the mobility simulation is of great interest. There are different implementations of the mobility simulation in the MATSIM framework. The most advanced is the java-based implementation of the queue simulation algorithm[3]. Although it has been tried to implement a multi-core version of the queue simulation on a beowulf-cluster the results implied, that the ethernet network latencies –even on an Gigabit network– make it difficult to gain a decent speedup by adding more clusters. A solution was the use of special Myrinet network hardware, but the overall cost of such a cluster is high[4]. We therefore concentrated on optimizing the single-CPU version of the queue simulation in recent years. To use cheap commodity hardware to speedup the simulation on a single computer would be of great benefit. In this paper several GPU based version of this queue simulation algorithm will be presented to gain a relevant speedup on a single computer system.

The GPGPU toolkits have been widely adopted by researchers and industry alike. Todays GPUs are used in many fields, for example in molecular dynamics[18], gas and fluid dynamics[16], astro-physics[10] [11], for coupled map lattices[14] genetic programming[23], graph algorithms[12] as well as DNA sequencing[24] or even database queries[15].

Most of these examples bear in common, that they involve very computation intensive operations and are known as being highly adaptable to a SIMD architecture. GPUs are optimized for SIMD operations, as most of the traditional duties in the field of rasterization of 3D images can efficiently be computed in that way.

We retrieved only three papers dealing with multi-agent simulation and GPU computation alike. Two papers were either restricted to the ant model of multi-agent simulation[8], and therefore nor concerned with network topologies or just benchmarking the GPU with multi-agent games like "game of life"[21]. One more research group we found has released a preliminary paper on field based vehicular simulation with GPUs[20]. This paper does not present any results though. The topic of network simulation on GPU is shortly discussed in another report on benchmarking GPU applications, it implements the MITSIM[9] algorithm on a GPU[5]. No other paper of our knowledge deal with network based multi-agent simulation at this time.

3. PRELIMINARIES

In this section we will shortly describe, which GPU architecture we will use as well as how the CUDA framework is structured. Furthermore we will summarize the queue simulation algorithm and the necessary data structures.

3.1 The NVIDIA GPU and CUDA framework

NVIDIA GPUs can be found in roughly 70 million PCs and notebooks around the world. The recent G80 series of NVIDIA GPUs had up to 128 cores and 1GB of memory. The newer G200 series reached a computational peak of 1 TFlop (single-precision floating point operations per second). In this paper one G80 series GPU, namely the Geforce 8600GT and one GPU of NVIDIAS latest G200 series (GTX280) will be used. Table 1 gives an overview of the number of cores and the clock rate of either RAM and GPU for these models as well as their typical memory bandwidth. The older G80 GPU is assembled on a passive-cooled graphics card, the memory bandwidth is notably lower than that of average G80 cards. The latest version of the CUDA framework (Version 2.0)[7] under Windows XP is used to implement our algorithms. The CUDA framework is an extension to the C language that enables us to write code for CPU and GPU in the same file. It therefore is rather easy to program in for any experienced C/C++ programmer. It basically adds the keywords `__device__`, `__global__`, `__host__` as method decorators to indicate whether the methods are run on the host CPU or the GPU and from where they could be called. Additionally it adds a syntax for describing with how many parallel threads a method should be started. Methods declared *global* are so called "kernels". These kernels could be called from the "host" (the PC/CPU the graphics device is running in) and run on the GPU. These kernels run simultaneously on different data sets in multiple threads of execution.

As we only have a limited number of "real" hardware processors, threads are joined to thread blocks, which again are bundled to a grid of thread blocks. This distinction is necessary as a block of threads (with recently up to 512 threads) is sharing a set of registers as well as a rather small on-chip memory area. Threads running in the same block can be synchronized with each other, whilst threads amongst block of threads cannot be synchronized. Threads within a block can also access a small amount of additional local shared memory. Though only up to 512 threads can make up one block of threads the grid of blocks can run thousands or even trillions (the actual upper limit of individually indexable threads being $65536 \times 65536 \times 512$) of threads in parallel.

The architecture of the G80 series consists of 16 multiprocessors with 8 thread processors each, summing up to 128 processors that can execute kernels in parallel. Blocks of threads are run in warps of 16 threads. Reads and writes to global memory can be done in random access, i.e. the SDK offers gather and scatter operations.

Nevertheless, as there is no effective caching in place, this access gets expensive when it does not obey a rather strict regime of SIMD execution. To avoid this, a block of threads will be suspended when an out of order memory access is demanded. While this block of threads waits for the memory access to be done, other blocks could do some work. A rather high number of threads is needed to mask these out of order memory accesses. It is important to maintain a high rate of memory accesses that are indexed by the actual thread ID, as these –so called coalesced– accesses could be handled in an optimized fashion causing an order of 10 less latency.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5666
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5666

3.2 The queue simulation

The queue simulation algorithm uses a graph for representing the traffic network. The streets are represented by links in the graph and the junctions by the nodes of the graph. This network is filled with agents. Every agent has a predefined plan for the whole day. This plan contains a succession of activities each with a route to travel from one activities location to the others.

3.2.1 Data structures for the queue simulation

For implementing the queue simulation we basically need dynamic fifo (first-in first-out) queues. As it is not possible to allocate memory on the fly while executing kernels, we must find a way to safely allocate some upper limit of memory for all data structures. Fortunately it is possible to make reasonable assumptions about the biggest size the queues can have.

One queue holds all vehicles traveling along the link. This queue is limited by the maximum number of vehicles the link has space for, often calculated as

$$space_{link} = length * lanes/carsize$$

A second queue is needed to hold vehicles, which are ready to leave a link in the actual timestep. The maximum size of this queue is given by the flow capacity of a link in the given timestep. For every link the flow capacity is the maximum number of vehicles that can travel the link in a certain timestep. The timestep used in our simulation is usually 1 second.

$$size_{buffer} = capacity_{flow} * \Delta t_{timestep}/\Delta t_{flowperiod}$$

Therefore the maximum size of both queues is known and can be allocated before a simulation is run.

3.2.2 The simulation loop

The queue simulation of traffic is based on a rather simple algorithm. Above defined buffers are used for storing vehicles, that are moving on the streets. Each link holds vehicles, that travel along, in a queue. When a vehicle enters the links queue, the minimum time duration is calculated that the vehicle has to spend on the link using the given maximum speed allowed on the actual link.

$$time_{link} = maxSpeed_{link} * length_{link}$$

Vehicles stay in this queue until they have traveled the link, i.e. this above time is spend. They are ready to leave the link when two more conditions apply. First, no more vehicles can leave the link than the link has flow capacity for this timestep. To assert this condition the second the buffer for outgoing vehicles with its confined space is used. Vehicles are being moved from the link to this buffer only if there is still room in this buffer. Second, a vehicle in this buffer can leave if there is space left in the destination links queue.

```
void sim() {
  while (time != end) simstep();
}

void simstep() {
  time++
  for all links: moveLink()
  for all nodes: moveNode()
}

void moveLink() {
 depart = get departure time of first veh in queue
 while ( depart < now && buffer.hasSpace())
 {
   move veh to buffer
   remove veh from link
   depart = get departure time of next veh in
       queue
 }
}

void moveNode() {
 for all incoming links buffers:
 {
  while (buffer is not empty)
  {
   dest = destination link of first veh
   if( dest.hasSpace())
   {
     move veh on top to destination link
   } else {
     // if first veh cannot leave, none can
     break and return;
   }
  }
 }
}
```

Listing 1: Pseudo code for transport simulation

The movement code of the queue simulation is drafted in pseudo code in listing. 1. This code does not handle insertion and removing of vehicles on their source and destination links. This is handled in an additional step in an extra kernel execution.

As we can see from the code, each simulation step consists of two large loops. Each `moveLink()` call is independent from each other, as it only accesses the link and the buffer of a link. The whole loop could easily be executed in parallel. As we have a few thousands to hundred thousands of links in an typical simulation, this also yields a sufficiently high number of threads to mask the necessary out of order memory accesses.

The calls to `moveNode()` are completely independent too. Although the buffers insert the vehicles into different queues of the vehicles destination links, these links are all only connected with this one node, so all nodes can be run in parallel without mutually competing for link spaces. Using the nodes to distribute the outgoing vehicles gives us distinct control over the priorities these links have at a certain node. It is therefore possible to prioritize e.g. the main road. The drawback of this in terms of parallel execution is that we have this double nested loop in the `moveNode()` method, leading to a highly serial execution path. This causes some additional uncoalesced read/write operations.

A second version of the algorithm has been implemented, where all link buffers run in parallel and deliver their vehicles concurrently. This apparently leads to race conditions in respect to the free spaces in a destination links queue. Fortunately the CUDA framework has functionality to access and change global memory in an atomic operation. A get-and-increment atomic operation is used to "occupy" a linkspace for one vehicle in a safe manner. Therefore it is guaranteed that no vehicle will be overridden and lost.

This atomic operation is more expensive than a regular memory access, but resolving the double nested loop will hopefully lead to a higher degree of parallel execution. This variant of the original algorithm has been benchmarked too for every data structure used further on.

To run our code on the GPU we declare the `moveLink()` and `moveNode()` methods as kernels and call them with an

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5666
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5666

appropriate number of threads. The overall number of thread is of course the number of links respectively nodes in the network, so that every link/node is run in a separate thread.

As been said, the CUDA framework is basically the C language with some concepts for kernel definition and execution added. It is rather straightforward to implement a naive version of the traffic simulation. With the basic algorithms above, the queue simulation is easily implemented on a CPU. It is parallelized by declaring some portions of it – namely the `moveLink()` and `moveNode()` methods– to run it on the GPU. This is done by declaring the methods as "kernels" and changing the calling code to reflect the number of threads to run in parallel.

In the further paper this implementation will be used to run the simulation. Only the underlying data structures of the buffer will be changed to achieve speedups.

3.2.3 Handling of activities

In our simulation every agent has a structured plan for the whole day. This plan consists of activities and routes between these activities. Obviously, for a traffic simulation the routes between the activities are the interesting bits. Therefore executing an activity is done by having the agent "wait" for the end of the activity somewhere outside the traffic simulation.

Each agent starts and ends with an activity called "home". In between these two activities he can do numerous other activities, e.g. "work", "school", "leisure", "shopping". This daily plans data structure is held in a big array for all agents and there is an additional administrative array holding pointers to the beginning and end as well as to the actual position within the plan for each agent. Each activity in a plan has a defined departure time. Each route is a sequence of links the agent has to travel. Each time the agent passes a link the position pointer of the agent's plan is increased. This also is done, when the agent leaves an activity.

When an agent reaches an activity, he/she is removed from the traffic simulation as the attended activity is outside of the traffic simulations scope. When the departure time for an activity is reached, the agent is inserted into the traffic simulation again. A separate kernel is responsible for taking care of this process. At every timestep this kernel runs over the plans of all agents in parallel and checks whether the plans position of this agent points to an activity and if so, if a particular agent needs to be inserted into the traffic simulation again (i.e. he is attending an activity **and** activity's endtime is reached or passed).

This is probably not the most efficient implementation of the insertion process and will most likely not scale well with increased agent count. There are several ways to alleviate this problem in future versions of the simulation, but as these optimizations would be most likely GPU specific, this is beyond the scope of this paper. The question this paper wants to answer is, if it is possible to gain a speedup by using GPU hardware and mostly CPU oriented algorithms.

4. VARIOUS IMPLEMENTATIONS

4.1 Data structures

When programming in Java or C++ there are libraries of several dynamic data structures in place one can rely on when it comes to implementing buffers. In the CUDA framework, these data structures are absent. Therefore all dynamic data structures have to be implemented manually. Namely this is the fifo buffer needed for the queue simulation. As there is no way to allocate/deallocate memory in a kernel, all memory has to be previously allocated by using the maximum sizes mentioned above.

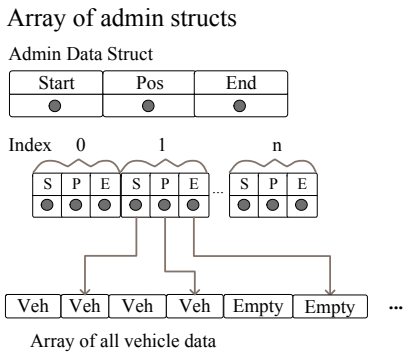

Figure 1: Administration structure for vehicle data on link/in buffer

The dynamic queues on the GPU will be implemented by using two distinct blocks of data. One big unstructured piece of memory stores the actual vehicle data. Another array of administrative data points into this big array of vehicle data. It holds the start, end and insertion positions of the respective link or buffer. Similar data structures have been utilized for graph search algorithms[12] before.

4.1.1 Array of structs

In Fig. 1 you find the typical data structure for link administration. For every link it contains information about the starting position of this links data in the big vehicle array as well as the end position and the insertion point for the next vehicle. The position member tells us how many of the queues slots are actually filled with cars right now. If *pos* == *start*, as it is to the beginning of the simulation, there is no car in the queue, otherwise all cells from *start* to *pos* are filled with car data. The integer values *start*, *pos* and *end* refer to indices inside of the big chunk of unsorted car data.

If we add a vehicle to the link or buffer we insert it at the position pointed to by the *pos* member of the struct and increase this member by one to point to the next free space. If we, on the other hand, want to remove one car from the start of the queue, we have to move all remaining cars one position nearer to the start and decrement the *pos* pointer to point to the now empty slot.

In the case of not using the `moveNode()` method an atomic operation is necesary to ensure that no cars get lost is the increase of the *pos* pointer, when a vehicle is moved from the buffer to the links queue. CUDA provides a method to receive the content of a global memory position and increase it in one atomic operation. This content point to a unique memory position within the buffer, which is reserved for the thread that issued the atomic operation. If the received position is before the *end* position of the particular link, the vehicle can move there, otherwise the destination link is

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5666
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5666

full. No other buffer can index this position anymore as the position pointer is already increased by one.

Performance results in table 2 (marked as AOS respectively AOSNODES) show that this implementation of the administrative data structure bears some drawbacks. Two neighboring threads have to access memory with an offset of the size of the struct in fig. 1. This leads to uncoalesced memory accesses. This comes with a high performance penalty. To ease the uncoalesced memory accesses it is necessary to align the data accessed by two neighboring threads by 8, 16, 32 or 64 bytes. This can be achieved by a rather simple transition of the data structure.

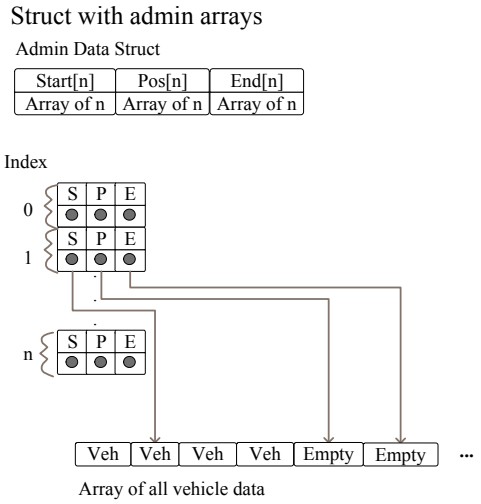

Figure 2: SoA layout for the administrative structure

4.1.2 Struct of arrays

Instead of using an array filled with structs of the above layout, this struct is changed to hold pointers to arrays of a simple data type, i.e. an integer value in this case. This will –in terms of memory access– give lead to a memory layout that better aligned with thread indices. Two adjacent kernels will have adjacent memory accesses, separated by 4 byte integers, which enables coalesced memory accesses, the fastest way to access the global memory from a thread block. In terms of the implementation it is also a simple optimization step, as one only "shifts" the index to the right. This has been done to all administrative memory layouts and the allocation code has to be slightly adopted. The changes layout of the data structure is illustrated by fig. 2.This Struct of Arrays (SOA) layout has been suggested from the CUDA team[13].

The actual struct is to change from the form in fig.3(a) to that in fig.3(b). Likewise the implementation needs changes. A former expression

```
int size = array[index].pos - array[index].start;
```

will change to

```
int size = array.pos[index] - array.start[index];
```

This transition could be done in a nearly mechanical way. It was applied to the buffer and the link's and agent's administrative structs. Results of this simple data structure "optimization" could be found in table2 as SOA and SOANODES.

```
QueueAdmin{
  int pos;
  int start;
  int end;
}
```

(a) AoS before

```
QueueAdminSoA{
  int pos[];
  int start[];
  int end[];
}
```

(b) SoA after

Figure 3: Translation of the struct from AoS to SoA.

4.1.3 Ring buffers

Another drawback of the above implementation is the need to shift remaining vehicles in the buffer whenever a vehicle is removed. This leads to performance penalties in a congested situation, when only a small amount of vehicles is allowed to leave a link and all remaining vehicles must be moved forward each timestep. This causes uncoalesced memory accesses, which should be avoided. A common way to avoid this is the use of ring buffers known from file I/O implementations. A possible ring buffer data structure is given by the following struct, as illustrated by fig. 4.

```
QueueAdminRing{
  int start[];
  int len[];
  int ep[];
  int count[];
}
```

The ring buffer comes with some extra overhead in administrative data and effort. The ring buffer implementation above has one *start* pointer, that is pointing into the vehicles block to indicate this links first memory position as before. The other members of the struct are relative to the start position. The *len* member gives us the maximum size of this buffer. So *start + len* point behind the last element of this buffer. The *ep* (extraction point) member indicates, where the first element of the actual queue resides. The *count* member is also relative to the extraction point member and indicates how many units are in the queue right now. Vehicles are removed from a memory position calculated by

$$pos_{top} = start + ep$$

.

Up to *count* vehicles can be removed from there, calculating the next position as

$$next_{top} = start + (ep + i) \mod len$$

where i runs from *0..count -1*. We can insert up to *len - count* vehicles at the insertion position calculated as

$$pos_{insert} = start + (ep + count) \mod len$$

Insertion and removal of vehicles does not need to move any existing vehicles anymore, enhancing situations where the AOS data structure was not performing well. The size of our administrative data structure is increased by one integer which could result in performance losses in the uncongested timesteps of the simulation.

Results show that the ring buffer implementation clearly outperforms the AOS version in all simulation runs, gaining even more advantage with higher network load.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5666
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5666

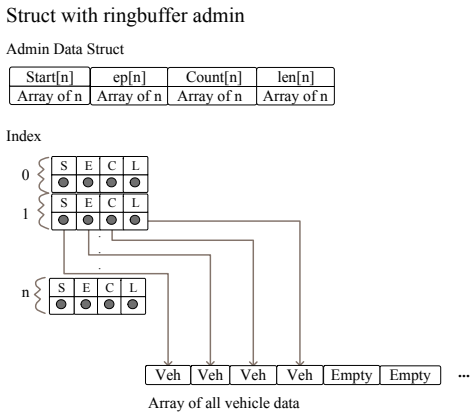

Figure 4: Administration structure implemented as a ring buffer

4.2 Algorithms

4.2.1 Node versus buffer movement

One particular non-parallel part of the simulation algorithm described above is the double-nested loop in the `moveNode()` code. By stepping through the incoming links in predefined sequence of choice, one has the opportunity to regulate the traffic at a finer level if there is demand for it, e.g. one could have streets with higher capacity move first. Dropping this behavior in favor of letting the buffers freely compete for a space on the vehicles destination link, a kernel can be run over all link buffers in parallel. It might be beneficial to the performance if the outer loop of `moveNode()` is removed and the kernel is run over all buffers in parallel.

```
void moveBuffer() {
 while (buffer is not empty)
 {
  ATOMIC ( get pointer to a unique free space on
       destination link)
  if (pointer is VALID)
  {
    move veh on top to destination link
  } else {
     // if first veh cannot leave, none can
     break and return;
  }
 }
}
```

Listing 2: Pseudo code for concurrent buffer movement

Listing 2 describes the new method `moveBuffer()`. One change to the code is necessary in terms of memory access. Before, the nodes controlled access to their outgoing links and there was no race condition for the links free spaces, as every link is only connected to exactly one node that could possibly insert vehicles into the link. When running in parallel over all buffers, two buffers of incoming links of the same node could try to insert their first vehicle onto an outgoing link at the same time. To race condition could be resolved with the ability of CUDA to issue atomic operations to global memory. In this case an atomic fetch-and-increment operation secures a link space position for an outgoing vehicle and increases the pointer to free space to another place in one atomic instruction, guaranteeing the returned pointer to be unique. This implementation was run additionally to the `moveNode()` runs. The regular movement code with the `moveNode()` method could be found in the results with the post-fix NODES, i.e. AOSNODES, SOANODES, RINGNODES. The runs with `moveBuffer()` do not have a post-fix, i.e. AOS, SOA, RING. The results will show, that the extra speedup provided by dropping the explicit control over the nodes/junctions scheduling algorithm is minor, therefore using the more elaborate `moveNode()` algorithm is to be prefered.

4.2.2 Separate vehicle movement

One additional code mutation was implemented dealing with the actual movement of vehicles within the `moveNode()` respectively `moveBuffer()` code. This code variant was only implemented for the best performing data structure RING. The (uncoalesced) movement of vehicle data within the inner loop was replaced with a simple integer write into a new index array. In a separate kernel, the actual movement of the vehicles was computed. This movement could then be performed in a more coalesced manner. Only the index writing was out-of-order. The performance improvements of this where not as big as expected, although this variant turned out to be the fastest on the GeForce 9800 GT card, the improvement was small and could not be reproduced on the GTX280.

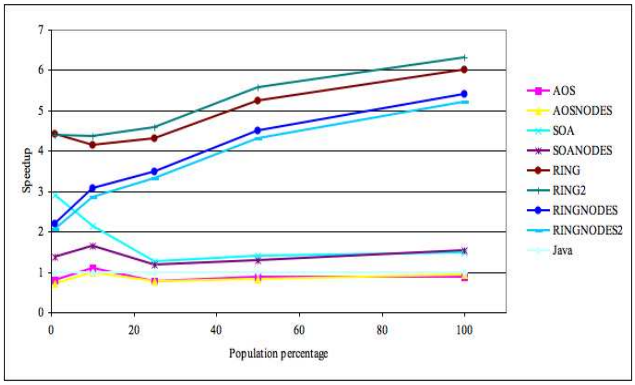

Figure 5: GeForce 8600GT speedup relative to java implementation

5. PERFORMANCE RESULTS

Several samples of an existing simulation run of the traffic in the Zurich area were chosen to benchmark the performance of the data structures. Samples of approximately 850k agents (100%), 425k agents (50%), 212k agents (25%), 85k agents (10%) down to a mere 8.500 agents (1%) were used. The network consists of about 37k links and 24k nodes.

First, all of these samples were run on our highly optimized Java version of the MATSIM mobility simulation. As the two implementations, the GPU approach on the one hand and the optimized java implementation on the other, differ in their algorithms rather significantly the measured performance is not easily compared. The java version does

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5666
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5666

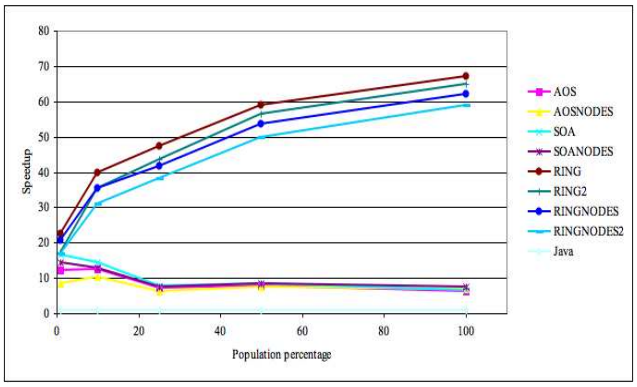

Figure 6: GTX 280 speedup relative to java implementation

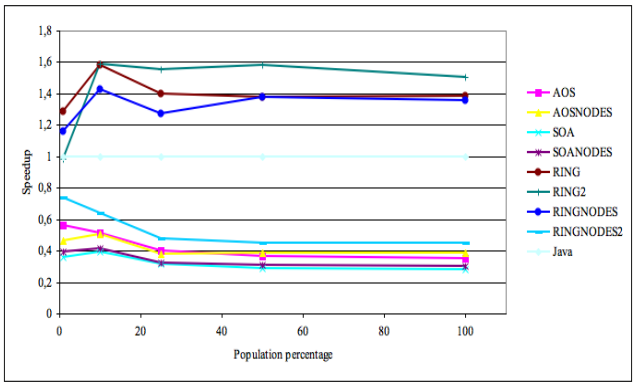

Figure 7: Single CPU speedup relative to java implementation

additional operations (e.g. creating events) that are not implemented in the GPU version. On the other hand this version is capable of disabling links with no traffic, therefore being highly inexpensive when there is no or little traffic in the network. Despite these differences the comparison still gives a rough idea of the performance improvements of a GPU implementation.

The simulation was run with the differently sized agent samples on the CPU and the two GPUs. The Java and the CPU versions were run on a Intel Pentium Dual Core 2.2 GHz, the GPU version on a desktop computer with a GeForce 8600 GT and one with a GTX200. The technical data of these cards could be found in table 1. The performance results can be found in fig. 5 for the older GeForce 8600 GT and in fig. 6 for the GTX 280. To get a better understanding of which part of the speedup must be attributed to not using dynamic data structures and the Java language we implemented a CPU version of the CUDA C-like code, which was executed on a single core at 2.2Ghz. As we can see from fig. 7 the CPU version was slower than the Java version for the simpler implementations, but a little bit faster for the RING variations. It is also interesting to notice the difference in caching schemes between CPU and GPU which results in the SoA approach to be even a little slower than the AoS implementation on a CPU. This is in stark contrast to the 8600 GT results in fig. 5, where the SoA implementation was significantly faster on all runs.

On the GPU the different samples were run for the three given data structures (array of structs (AOS), struct of arrays (SOA) and ring buffer (RING)) in two code variants. One with the original `moveNode()` code that serializes the traffic on the nodes for finer control over the streets priorities, marked with NODES, and one where all buffers compete for linkspace with no post-fix. Two additional runs (RING2, RINGNODES2) were benchmarked with the separated vehicle movement. The speedups in these diagrams are relative to the java versions runtime.

As we can see from the results on the GeForce 8600GT our naive implementation can at least compete with the java version. Using a data structure more suitable for our needs will brings it up to a speedup of 6 over the java version. Changing the movement code for the vehicles improves the performance not necessarily, though.

On the newer GTX280 all runs were faster than the java version. The GTX280 has 7.5 times the number of processors of the GeForce 8600GT, therefore it should give us an performance improvement of this magnitude as well. It apparently does so for the data structures AoS and SoA, as the speedup is around 8 for these data structures. With the more sophisticated ring buffer, it reaches a speedup of over 60. This must be attributed to the improved memory management of the GTX280 and the much higher memory bandwidth this card offers. This is a very promising result, as it implies that our implementation will automatically profit from coming hardware improvements.

6. CONCLUSION AND OUTLOOK

As one can see from the above results these simple "optimization" leads to a performance gain around the factor 6 from an unsuitable data structure to the ring buffer implementation. A speedup of up to 67 times compared against out highly optimized java version was achieved. This speedup was achieved by using proper data structures adopting an algorithm developed for CPU usage. The GTX280 GPU could simulate up to 16000 seconds within one second of realtime with an relevant population sample (10%). Some more peak realtime speedups could be found in table 2.

Nevertheless the code is far from being optimized. Several well known parallel algorithms like prefix scans or double buffered techniques could be used, to speed up the simulation code. This implementation uses the simplest possible way to activate the agents, i.e. to look at every agent in every timestep and check if it needs activation. This seems a certain candidate for further optimization. It apparently does not scale very well with increasing agent count. Profiler runs with the actual implementation indicate that this part of the program gets the dominant factor with increasing agent count. Still it should feasible to "sort" the agents into some buckets, regarding their planned departure time and then only inspect that one bucket for every timestep, that hold the agents with a departure scheduled for this timeslot. Not having truly dynamic data structures might be problematic, as one would have to reserve space for all agents in every bucket, to make sure we can handle every thinkable constellation of departure. On the other hand, one might

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5666
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5666

Impl	10%k	25%	50%	100%
Java opt.	462	308	181	108
GPU 8600 GT	1419	1077	816	585
GPU GTX280	16.383	12.892	9695	6699

Table 2: Speedup against realtime for different number of agents

fill smaller buckets and mark them with an timstep, having to run over some smaller buckets with the same timestep instead of one large. To investigate further improvements will be part of our future research.

7. REFERENCES

[1] ATI FireStream www page. Firestream: DAAMIT GPGPU framework, accessed 08/2008.

[2] B. Bustos, O. Deussen, S. Hiller, and D. Keim. A graphics hardware accelerated algorithm for nearest neighbor search, 2006.

[3] N. Cetin. Large-scale parallel graph-based simulations. Master's thesis, Swiss Federal Institute of Technology (ETH) Zürich, Switzerland, 2005.

[4] N. Cetin, A. Burri, and K. Nagel. Parallel queue model approach to traffic microsimulations. In *In Proceedings of Swiss Transportation Research Conference*, 2002.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, not published pre-print, 2008.

[6] R. D. Chiara, U. Erra, V. Scarano, and M. Tatafiore. Massive simulation using GPU of a distributed behavioral model of a flock with obstacle avoidance. In B. Girod, M. A. Magnor, and H.-P. Seidel, editors, *VMV*, pages 233–240. Aka GmbH, 2004.

[7] CUDA www page. CUDA: NVIDIA GPGPU framework, accessed 08/2008.

[8] R. M. D'Souza, M. Lysenko, and K. Rahmani. Sugarscape on steroids: simulating over a million agents at interactive rates. In *Proceedings of Agent2007 conference*, Chicago, IL., 2007.

[9] DYNAMIT/MITSIM www page. http://mit.edu/its, accessed 2008.

[10] E. Elsen, V. Vishal, M. Houston, V. Pande, P. Hanrahan, and E. Darve. N-body simulations on GPUs, Jun 2007.

[11] T. Hamada and T. Iitaka. The chamomile scheme: An optimized algorithm for n-body simulations on programmable graphics processing units, Mar 2007.

[12] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In S. Aluru, M. Parashar, R. Badrinath, and V. K. Prasanna, editors, *HiPC*, volume 4873 of *Lecture Notes in Computer Science*, pages 197–208. Springer, 2007.

[13] M. Harris. CUDA workshop pre-ISC2008, dresden, Juni 2008.

[14] M. J. Harris, G. Coombe, T. Scheuermann, and A. Lastra. Physically-based visual simulation on graphics hardware. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 221, New York, NY, USA, 2005. ACM.

[15] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 511–524, New York, NY, USA, 2008. ACM.

[16] K. Hegeman, N. A. Carr, and G. S. P. Miller. Particle-based fluid simulation on the GPU. In V. N. Alexandrov, D. G. van Albada, Peter, and J. Dongarra, editors, *International Conference on Computational Science (4)*, volume 3994 of *Lecture Notes in Computer Science*, pages 228–235, 2006.

[17] J. Krüger and R. Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Transactions on Graphics*, 22:908–916, 2003.

[18] W. Liu, B. Schmidt, G. Voss, and W. Müller-Wittig. Molecular dynamics simulations on commodity GPUs with CUDA. pages 185–196. 2007.

[19] P. Micikevicius. General parallel computation on commodity graphics hardware: Case study with the all-pairs shortest paths problem. In H. R. Arabnia, editor, *PDPTA*, pages 1359–1365. CSREA Press, 2004.

[20] K. S. Perumalla. Efficient execution on GPUs of field-based vehicular mobility models. *PADS*, 0:154, 2008.

[21] K. S. Perumalla and B. G. Aaby. Data parallel execution challenges and runtime performance of agent simulations on GPUs. In H. Rajaei, G. A. Wainer, and M. J. Chinni, editors, *SpringSim*, pages 116–123. SCS/ACM, 2008.

[22] B. Raney and K. Nagel. An improved framework for large-scale multi-agent simulations of travel behaviour. In P. Rietveld, B. Jourquin, and K. Westin, editors, *Towards better performing European Transportation Systems*, page 42. Routledge, London, 2006.

[23] D. Robilliard, V. Marion-Poty, and C. Fonlupt. Population parallel gp on the G80 GPU. In *EuroGP*, pages 98–109, 2008.

[24] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 8(1), 2007.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5666
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5666