Spatial Data Structures, Sorting and GPU Parallelism for Situated-agent Simulation and Visualisation

A.V. Husselmann and K.A. Hawick
Computer Science, Institute for Information and Mathematical Sciences, Massey University, North Shore 102-904, Auckland, New Zealand
email: { a.v.husselmann, k.a.hawick }@massey.ac.nz
Tel: +64 9 414 0800 Fax: +64 9 441 8181
March 2012

ABSTRACT
Spatial data partitioning techniques are important for obtaining fast and efficient simulations of N-Body particle and spatial agent based models where they considerably reduce redundant entity interaction computation times. Highly parallel techniques based on concurrent threading can be deployed to further speed up such simulations. We study the use of GPU accelerators and highly data parallel techniques which require more complex organisation of spatial datastructures and also sorting techniques to make best use of GPU capabilities. We report on a multiple-GPU (mGPU) solution to grid-boxing for accelerating interaction-based models. Our system is able to both simulate and also graphically render in excess of $10^5 \sim 10^6$ agents on desktop hardware in interactive-time.

KEY WORDS
grid-boxing; sorting; GPU; thread concurrency; data parallelism

1 Introduction

Simulation problems involving interacting spatially located entities such as N-Body particle models [1–4] or spatial agent-based models often require considerable computational power to support simulation of adequately large scale systems. Large size is important to expose and investigate complex and emergent phenomena that often only appear on logarithmic length and systems size scales. To reduce the $O(N^2)$ computational complexity of such interacting systems, spatial partitioning methods are often used. Depending upon how effective the scheme used is, the computational complexity can be reduced down to $O(N \log N)$. Furthermore, a good scheme will also support - and not impede - the introduction of parallelism into the computation so that an appropriate parallel computer architecture [5] can be used to further reduce compute time per simulation step. As a vehicle, we use the Boids model originally by Reynolds [6, 7].

Spatial partitioning [8–10] has been employed as far back as the earliest attempts of Andrew Appel [11] who produced the first multipole partitioning method, with many subsequent developments [12]. The N-Body particle simulation has served as a simple but effective testing platform for improving the performance of particle simulations and generally interaction-based models which rely on particle-particle interactions [13]. The vast majority of particle simulations find themselves suffering an inherent performance problem and a severe lack of system size scaling. This is mostly due to particle interaction with every other particle inducing an $O(N^2)$ complexity. Even for a conceptually simple in-
interaction such as a nearest neighbour (NN) interaction, it is still necessary, in the simplest cases, to perform calculations (such as Euclidean distance) for every combination of two particles. Collision detection is one such problem.

This problem is mitigated by using methods such as Barnes-Hut [14, 15] treecodes, Fast Multipole methods [16], and tree organisational methods [17, 18] including Octrees [19–26], K-D trees [27, 28], Kautz trees [29], Adaptive Refined Trees [30], and more, all of which fall under the category of spatial partitioning. Not all of these algorithms would necessarily be suitable for any one particle simulation. Some of these algorithms are suitable for simulations which frequently have high density in one or several spatial locations. Some perform best with uniformly distributed particles, and some attempt to systematically treat several particles in the same light as a point mass to reduce computation in exchange for losing a small amount of precision. This technique is often well-suited to applications in astrophysics. There also exists algorithms which sacrifice precision for performance, with techniques as simple as updating only a certain number of agents per simulation time step, which dramatically increases performance but the loss of precision is so great it will no longer reflect the original simulation code. Choosing which one of these algorithms or methods to use in a simulation therefore ultimately depends on the purpose of the simulation.

This work contributes towards efficiency in the simulation of large multi-agent systems which do not require high precision in very small clusters. To accomplish such a task in a single-threaded environment, one would normally employ standard techniques such as Octrees, or K-d trees which are relatively simple to implement in single-threading. K-d trees offer some parallelisation opportunities with some authors reporting piece-wise GPU datastructure construction algorithms which are of $O(N \log^2 N)$ and also $O(N \log N)$ complexity [27,28].

The former is normally the case when constructing a K-d tree by performing an $O(N \log N)$ sort for every node in the tree to obtain the median. The latter is the case when using a linear median-finding algorithm, which is not trivial to implement on GPU, nor well-suited on GPU either.

One space partitioning algorithm responds very well to parallelism: grid-boxing [8] (sometimes known as uniform grid space partitioning). The NVIDIA CUDA SDK comes with a particle simulator which makes use of this technique [31]. Our single-GPU implementation of grid-boxing is heavily based on this.

Graphical Processing Units (GPUs) [32] lend themselves particularly well to accelerating simulations especially after the advent of NVidia’s CUDA. CUDA is a powerful addition to C++ which allows the programmer to write code in C or C++ containing sections of special syntax which NVidia’s special compiler in the CUDA SDK compiles into GPU instructions before handing the rest of the program code to the system’s built in C or C++ compiler.

Figures 3(a) and 3(b) show examples of K-d trees,
which are very popular for accelerating ray tracing programs [27, 28]. Since many aspects of ray tracing are inherently parallel, K-d trees have also received a large amount of research effort in order to offload as much as possible computation from the CPU to GPU. For this reason, GPU K-d trees are of great interest, and numerous implementations already exist [33–35]. Each of these implementations have different ratios of CPU code to GPU code. Significant parallelism can be achieved by building the tree level-by-level instead of node-by-node, but even with this advantage, it is difficult to retain most computation on the GPU.

Our article is structured as follows. In Section 2 we present the method by which we construct the grid-boxing datastructure, and also the method by which we use the result to accelerate body-body interactions with multiple-GPUs (mGPU) and also single-GPU in a standard simulation of the Boids model as propounded by Reynolds. Following this, in Section 3 we discuss the results we obtain and the approximate scaling we observe from this algorithm, from low numbers of agents (16384) to excessive numbers (1,000,000+). In Section 4 we discuss our results and implementation, and how it compares to other techniques. Section 5 provides conclusions and future work that we may pursue in this area.

2 Method

We use NVidia’s CUDA as the parallelisation platform. With this choice comes several restrictions, and perhaps the most prominent of these comes with the memory architecture. A CUDA device’s memory architecture is divided into several kinds: constant, global, shared, and texture. Each of these have different scopes and access penalties. Global memory takes by far the longest at about 200 cycles, followed by faster constant memory and very fast shared memory. The scope of these range from application to block level, and also kernel level. A CUDA block is conceptually a 1, 2 or 3D grid of CUDA threads, whose dimensions are arbitrarily defined by the user. For optimal results, it is beneficial to adjust these parameters to suit the program. This is especially important to allow latency hiding when reading global memory (the slowest).

Due to this architecture, it is impractical to use pointers and other simple data structures which require them on GPU. Efficient practical algorithms for generating trees on GPUs generally make use of hash tables, or space-filling curves. Morton ordering (also known as Z-ordering or N-ordering) is a very popular space-filling curve for the purpose of ensuring spatial locality [36]. Treecodes on GPU make use of Morton codes, which serve the purpose of encoding tree nodes for storing in hashtables [20, 34]. We use this method to increase coalesced global memory reads. The CUDA architecture generally reads fixed-size blocks of memory, and caches this for a short time when a thread requires it. It is for this reason that scattered reads from many threads earns a very large time penalty.

Algorithm 1 Single-GPU implementation of grid-boxing in boids.

```plaintext
Allocate and initialise a vec4 array as velocity
Allocate and initialise a vec4 array as position
Allocate a uint array as hashes
Allocate a uint array as indices
Allocate a uint array as gridBoxStart
Allocate a uint array as gridBoxEnd
Copy simulation parameters to device

//For n frames.
for i ← 0 to n OR NOT exit_condition do
    i ← i + 1
    //Calculate hashes for all boids.
    for j ← 0 to NUM_BOIDS do
        hashes[j] = calculate_hash(position[j])
    end for
    Sort by hash key (hashes,indices)
    Populate gridBoxStart and gridBoxEnd
    Scatter write boids
    Perform Boid kernel
    copyVectorsToDevice()
    drawBoids()
    swapDeviceBuffers()
end for
```

Algorithm 1 presents the single-GPU version of grid-boxing. All actions in this algorithm are performed in parallel, except for the loop containing the exit condition. This loop is simply used to advance to the next frame.

The very first kernel launched in this algorithm is the hash calculation kernel. This kernel has a simple task, and that is to populate the hashes and indices arrays. Algorithm 2 contains a pseudocode version of this kernel. Once this kernel has completed, the next step is to sort by the hash key. This step is accomplished using Thrust [37]. Thrust makes use of a parallel merge sort for our purposes. In appropriate situations, it uses an aggressively optimised GPU Radix sort written by...
lates hashes for all boids.

The first modification we made was the addition of host

datastructure construction time and all other subsequent

computation time required by this kernel dwarfs that of the

agent interaction kernel. In larger systems, the com-

puter across more than one GPU - the agent-

order to achieve this, we only parallelise one part of

GPUs in order to both reduce computation time, and

also to reach into even larger numbers of agents. In

large applications, boids will cluster more tightly in the middle of

size box. Figures 4(a) and 4(b) show a visual indication

goal rule to direct boids toward the center of a fixed

Reynolds [6, 7]. In our simulation, we make use of a

simulation model that we use for benchmarking

these techniques is the Boids model, originally by Craig

Merrill and Grimshaw [38].

Following the sorting phase of the algorithm, a single

kernel is executed to populate the grid box starting and

ending indices, as well as to perform a scatter write of

the boids into their sorted positions, so that the hashes

will be effective when the boid kernel is executed. The

boid kernel itself simply evaluates the boids within the

8 grid boxes surrounding the grid box that the current

boid is in. During this process, the sorting phase will af-

ford improved coalesced memory reads for the device,

and this greatly speeds up the process. The Morton or-

dering used when the hashes were calculated also in-

crease memory locality of the grid boxes surrounding

the current boid.

The kernel which calculates the grid box starting and

ending indices does so by having each thread keep track

of one grid box hash, and comparing itself against the

next thread’s hash. The threads which do not match

the hash of the next thread are the ones which mark the

boundary points of the grid box, and these threads sim-

ply write their starting and ending indices into an array

in global memory. To accelerate this process, shared

memory is used. The threads then reuse themselves by

copying the boid indicated by their indices into the new

sorted positions indicated by the index array reordered

by the Thrust sorting phase.

We modified this algorithm to execute across several

GPUs in order to both reduce computation time, and

also to reach into even larger numbers of agents. In

order to achieve this, we only parallelise one part of

the algorithm across more than one GPU - the agent-

agent interaction kernel. In larger systems, the com-

putation time required by this kernel dwarfs that of the

datastructure construction time and all other subsequent

steps taken. Therefore we find it most suitable to extend

the algorithm in this manner first.

The first modification we made was the addition of host

POSIX threads. One thread is created for every GPU,

and these control the memory and execution of each.

A POSIX thread barrier is used to synchronise between

the threads. Following this change, we modified the

kernel execution code to perform interaction to parts of

the total number of agents. For every POSIX thread,

the range \(r\) of agents to evaluate is \(r = t/g\) where \(t\)

is the total number of agents, and \(g\) is the total num-

ber of GPUs available. The starting index \(s\) is simply

\(s = it/g\). Once the agent interactions have been eval-

uated, the POSIX threads are synchronised and the com-

plete parts are copied from each GPU back to the host.

This process is effectively a gather operation. Once

a frame is drawn or finished, each POSIX thread copies

the entire host memory into its GPU, and the process

repeats. It is worth noting that one could either have

one GPU calculate the initial hashes, sort, and reorder-

ing, then copy (effectively scatter) the results to the

other GPUs via host memory, or simply have each GPU

independently perform these steps. It is a large waste

of computation, but copying between GPUs will likely

cause a higher overhead. In future, we hope to paral-

lelise the previous steps also.

Following from Algorithm 1, the modified algorithm

for use with more than one GPU is presented in Alg. 3.

This algorithm is a very simple parallelism of Alg. 1,

but it affords an excellent (albeit single-factor) increase

in performance, and it also brings into reach even higher

numbers of agents, with reasonable computing time.

Firstly, all vectors are copied onto all devices (a

scatter operation). Then, for \(n\) frames, or until the

application quits, each GPU will calculate hashes for

each boid, then sort by hash, then populate the grid box

starting and ending indices, scatter the boids into their

sorted positions, and finally, perform a partial agent-

agent interaction kernel. Once this is complete, the re-

sults are gathered back onto the host, where it is either
drawn on the screen, and/or scattered back to the GPUs

and the process repeats.

The simulation model that we use for benchmarking

these techniques is the Boids model, originally by Craig

Reynolds [6, 7]. In our simulation, we make use of a

goal rule to direct boids toward the center of a fixed

size box. Figures 4(a) and 4(b) show a visual indication

of boid clustering near the start and end of a simulation,

respectively. This shows more clearly that during sim-

ulation, boids will cluster more tightly in the middle of

the box. We ensure that this is the case so that we can
determine exactly how the effectiveness of grid-boxing

is lost as boid clustering increases, and gridboxes con-
tain more and more boids.

Algorithm 2 Data-parallel CUDA kernel which calcu-
lates hashes for all boids.

```c
func calc_hash_d(...) 
  i = blockIdx.x * blockDim.x + threadIdx.x 
  if i < numBoids then
    p = positions[i]
    //Morton ordering for hashes
    hash = calc_grid_hash(p)
    hashes[index] = hash
    indices[index] = index
  end if
end
```

Followings the sorting phase of the algorithm, a single

kernel is executed to populate the grid box starting and

ending indices, as well as to perform a scatter write of

the boids into their sorted positions, so that the hashes

will be effective when the boid kernel is executed. The

boid kernel itself simply evaluates the boids within the

8 grid boxes surrounding the grid box that the current

boid is in. During this process, the sorting phase will af-

ford improved coalesced memory reads for the device,

and this greatly speeds up the process. The Morton or-

dering used when the hashes were calculated also in-

crease memory locality of the grid boxes surrounding

the current boid.

The kernel which calculates the grid box starting and

ending indices does so by having each thread keep track

of one grid box hash, and comparing itself against the

next thread’s hash. The threads which do not match

the hash of the next thread are the ones which mark the

boundary points of the grid box, and these threads sim-

ply write their starting and ending indices into an array

in global memory. To accelerate this process, shared

memory is used. The threads then reuse themselves by

copying the boid indicated by their indices into the new

sorted positions indicated by the index array reordered

by the Thrust sorting phase.

We modified this algorithm to execute across several

GPUs in order to both reduce computation time, and

also to reach into even larger numbers of agents. In

order to achieve this, we only parallelise one part of

the algorithm across more than one GPU - the agent-

agent interaction kernel. In larger systems, the com-

putation time required by this kernel dwarfs that of the

datastructure construction time and all other subsequent

steps taken. Therefore we find it most suitable to extend

the algorithm in this manner first.

The first modification we made was the addition of host

POSIX threads. One thread is created for every GPU,

and these control the memory and execution of each.

A POSIX thread barrier is used to synchronise between

the threads. Following this change, we modified the

kernel execution code to perform interaction to parts of

the total number of agents. For every POSIX thread,

the range \(r\) of agents to evaluate is \(r = t/g\) where \(t\)

is the total number of agents, and \(g\) is the total num-

ber of GPUs available. The starting index \(s\) is simply

\(s = it/g\). Once the agent interactions have been eval-

uated, the POSIX threads are synchronised and the com-

plete parts are copied from each GPU back to the host.

This process is effectively a gather operation. Once

a frame is drawn or finished, each POSIX thread copies

the entire host memory into its GPU, and the process

repeats. It is worth noting that one could either have

one GPU calculate the initial hashes, sort, and reorder-

ing, then copy (effectively scatter) the results to the

other GPUs via host memory, or simply have each GPU

independently perform these steps. It is a large waste

of computation, but copying between GPUs will likely

cause a higher overhead. In future, we hope to paral-

lelise the previous steps also.

Following from Algorithm 1, the modified algorithm

for use with more than one GPU is presented in Alg. 3.

This algorithm is a very simple parallelism of Alg. 1,

but it affords an excellent (albeit single-factor) increase

in performance, and it also brings into reach even higher

numbers of agents, with reasonable computing time.

Firstly, all vectors are copied onto all devices (a

scatter operation). Then, for \(n\) frames, or until the

application quits, each GPU will calculate hashes for

each boid, then sort by hash, then populate the grid box

starting and ending indices, scatter the boids into their

sorted positions, and finally, perform a partial agent-

agent interaction kernel. Once this is complete, the re-

sults are gathered back onto the host, where it is either
drawn on the screen, and/or scattered back to the GPUs

and the process repeats.

The simulation model that we use for benchmarking

these techniques is the Boids model, originally by Craig

Reynolds [6, 7]. In our simulation, we make use of a

goal rule to direct boids toward the center of a fixed

size box. Figures 4(a) and 4(b) show a visual indication

of boid clustering near the start and end of a simulation,

respectively. This shows more clearly that during sim-

ulation, boids will cluster more tightly in the middle of

the box. We ensure that this is the case so that we can
determine exactly how the effectiveness of grid-boxing

is lost as boid clustering increases, and gridboxes con-
tain more and more boids.
Algorithm 3 Multiple-GPU (mGPU) implementation of grid-boxing in boids.

Allocate arrays
Copy simulation parameters to constant memory on all devices
HOST.copyVectorsToDevice()
for i ← 0 to n OR NOT exit_condition do
    i ← i + 1
    for each gpu g in parallel do
        for j ← 0 to NUM_BOIDS in parallel do
            hashes[j] = calculate_hash(position[j])
        end for
        Sort by hash key (hashes, indices)
        Populate gridBoxStart and gridBoxEnd
        Scatter write boids
        Perform Boid kernel
    end for
    copyVectorsPartiallyFromDevices()
    drawBoids()
    copyVectorsToDevice()
end for

(a) A histogram of the spatial locations of boids near the beginning of the simulation. (b) Another histogram of spatial locations of boids several hundred frames later.

Figure 4: Histogram plots of spatial locations of boids during the simulation. Red refers to a cell with 1 boid, green 2 boids, blue 3 boids, and yellow 4 boids.

3 Results

We tested the performance of our mGPU and single-GPU versions of the grid-boxing algorithm. The results for mGPU interaction and datastructure construction times are presented in Figures 5 and 6 in the form of a y-normalised plot for interaction computation, and a linear plot for the datastructure construction time. In both of these plots, the data points taken are 10-frame averages. For the mGPU implementation, we average the compute time across the four GPUs that we use. The testing GPUs we used are two NVidia GTX 590s. Each of these cards have two physical GPUs each.

Our simulation is simply a flock of boids moving toward the centre of the box they are in. Movement towards the centre causes a clustering effect in the centre of the box, which compresses and pulsates a number of times before reaching what can be described as equilibrium. This accounts for the initial increase in computation time, followed by slight harmonic fluctuations.

Figure 7 shows how the mGPU implementation scales against the single-GPU version of the algorithm. It shows an impressive performance gain over the single-GPU version, particularly as the system becomes larger. Smaller systems are simply more suited to being evaluated on one GPU to avoid the large overhead of memory copies across the PCI-E bus.
4 Discussion

Uniform grid-boxing might well be a more simplistic space partitioning method, but it is readily parallelisable, as opposed to other data structures inspired from single-threaded solutions. These solutions often rely on pointers for the simplest implementation, as well as recursion. These two techniques are the bane of data-parallel programs.

The CUDA-GPU memory architecture does not allow for a heap, and branch diversion is an even larger problem. Branching is allowed in CUDA, but it comes at a performance loss. Each time threads in a warp diverge from one another, some threads simply execute NOOPs until they converge again. Thus, a CUDA kernel is most effective when its threads execute the same instructions.

Other space partitioning algorithms also exist which perform well, but they only slightly resemble the algorithms which inspire them. For example, it may well be possible to replace a K-d tree with a hashtable to make it suitable for execution on GPU, but this multi-threaded hashtable requires mutexes - another branch diversion issue. It seems the only way to perform space partitioning in reasonable time on GPU to still make the effort of writing a CUDA-based program is to utilise some kind of sorting algorithm to place particles in very deliberate places in memory, to gain both spatial locality and also some structured method of traversing this data.

Sorting on GPU has been studied extensively, and many excellent solutions already exist, most notably the Radix sort by Merrill and Grimshaw merged into the Thrust library, which is shipped with CUDA [38].

The uniform nature of this grid-boxing data structure makes mGPU an attainable goal for multi-agent simulations, as it allows for distributing data with minimal copying between GPUs, while still preserving accuracy. This technique has seen a lot of use already.

Finally, it is worth noting also that grid-boxing is only well-suited to algorithms which have a fixed interaction distance. The smaller this distance is, the better the algorithm will perform, given that the grid is appropriately sized. For the results to remain fully accurate and representative of the original simulation code, the interaction distance must be smaller than the grid box size.

5 Conclusions

We have presented a multiple-GPU (mGPU) and single-GPU solution for grid-boxing in multi-spatial-agent simulations. Performance measurements were made, and the two algorithms compared, and an overwhelming clear advantage of mGPU was seen over the single-GPU implementation as expected. Even though memory sacrifices were made in the process, larger systems continued to receive an excellent increase in performance. These techniques are vital for facilitation of large system sizes of agents and the investigation of logarithmic scaled emergent phenomena in simulations.

In future we hope to parallelise the hash calculation and the sorting and reordering phases also. These phases are more difficult to parallelise over multiple GPUs, but this will greatly improve performance, and allow the ability to reach into much larger systems as well. These approaches will allow interactive-time visualisation of large scale spatial agent simulations and hence make it easier to search the parameter space of the models.

References