Communicating Process Architectures 2013
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2013

Exploring GPGPU Acceleration of Process-Oriented Simulations

Frederick R.M. BARNES [1], Thomas PRESSNELL and Brendan LE FOLL

School of Computing, University of Kent, UK

Abstract. This paper reports on our experiences of using commodity GPUs to speed-up the execution of fine-grained concurrent simulations. Starting with an existing process-oriented 'boids' simulation, we explore a variety of techniques aimed at improving performance, gradually refactoring the original code. Successive improvements lead to a 10-fold improvement in performance, which we believe can still be improved upon, allowing us to explore simulations with larger numbers of agents (30,000 rather than 2,000) interactively and without significant performance degradation.

Keywords. GPGPU, process-oriented simulations

Introduction

This paper considers the issue of using GPUs, SIMD-style hardware architectures, to accelerate fine-grained concurrent systems, that employ and encourage MIMD-style parallelism and that use a range of mechanisms for inter-process communication and synchronisation. As a non-trivial example we consider a simulation based on Reynolds' boids [1], whose structure and operation is described in section 1, together with a brief overview of the occam-π concurrency model used. The boids simulation used as an example is not trivially parallelised, and in general can be considered an *n-body* problem.

Our background is in the area of fine-grained concurrency, developing languages (occam-π [2]) and run-time kernels (CCSP [3]) supporting the construction of massively concurrent software systems[2], and their efficient execution on multicore and multiprocessor platforms. A wide range of software systems have been built using these techniques, including embedded control systems, agent-based simulations and operating systems. Concurrency is used as a fundamental design principle, with language (compiler) guarantees of freedom from race-hazard and aliasing error, permitting the building of predictably scalable systems [4], based on the semantics of CSP [5] and the π-calculus [6]. Furthermore, such systems can lend themselves to speedup on parallel hardware, where there is enough 'parallel slack' to ensure execution cores are kept busy.

The recent emergence of GPGPU (general-purpose graphics processor) programming, and its increasing availability, leads us to ask how we can take advantage of this in order to speed up computationally heavy process-oriented systems. The principle programming abstractions for GPUs are data-parallel and C based, with CUDA [7] and OpenCL [8] foremost. The issue of parallelising existing code and algorithms for GPU execution is largely

[1]Corresponding Author: *Fred Barnes, School of Computing, University of Kent, Canterbury, Kent, CT2 7NF, UK*. E-mail: `f.r.m.barnes@kent.ac.uk`

[2]By *massively concurrent* we mean large numbers (10^4–10^6) of communicating concurrent processes on a single processing node.

well-studied [9,10], though optimal implementations can be difficult and likely tailored to the specific problem and hardware.

Section 2 explores three different approaches that treat the GPU as an abstract computational resource within a process-oriented program, with successive modifications aimed at improving performance. For each, the design and performance of the approach is discussed. Section 3 draws some concluding remarks on the work, with some discussion of limitations, applicability to other simulations and directions for future work.

1. A Process-Oriented Boids Simulation

Figure 1 shows the process-network for part of the "boids" simulation [4], developed as part of the CoSMoS project [11]. The boxes represent processes — independent threads of control with their own (private) state. The lines connecting them represent channels or bundles of channels (those with arrowheads), and barriers (no arrowheads). Channel communication is synchronous and unbuffered — readers must wait for writers and vice-versa. Furthermore, there is no *shared data* or uncontrolled *aliasing* in this system (enforced by the occam-π [2] compiler), with data copied or references *moved* between processes. Several of the processes in Figure 1 utilise *many-to-one* channel structures — the many writers compete for access using a fair-queueing semaphore with correct use enforced by the compiler.

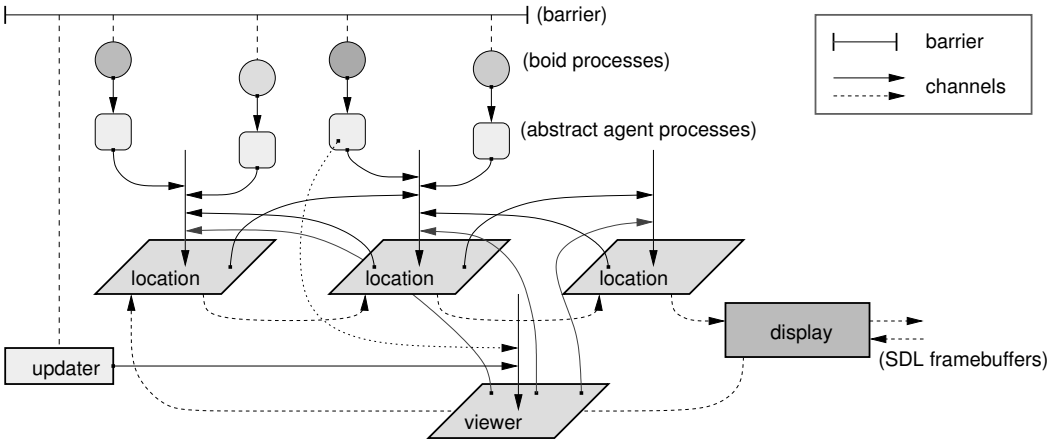


Figure 1. Process network for a dynamic agent-based simulation.

The topology of the world in which the boids exist is constructed from a toroidal 2-dimensional grid of 'location' processes (only 1 dimension is shown in Figure 1). Associated with each location are 'viewer' and 'updater' processes, the former keeping details of agents in the associated and neighbouring locations, the latter controlling updates to this view. Boid and updater processes are *enrolled* on a barrier that is used to control progression in the simulation, ensuring that all boids (and other processes) proceed at the same rate, and further ensuring that the view returned to all boids in any particular cycle is consistent.

The concurrent design of the simulation confers several advantages. Firstly, it is structured in a way that enables a consistent understanding of the system, its components and their interactions — we transition from design (Figure 1) to a concurrent implementation seamlessly. Secondly, modifying or extending the system, to change the behaviour of some or all boids, allow for the creation and destruction of boids at run-time, or to explore interesting non-Euclidean topologies, is largely straightforward.

It should be noted that this is the system we are starting with: a MIMD-style process-oriented CPU based multiprocessor implementation. If we were starting anew with the intention of GPU utilisation, the system would likely be designed and constructed in a very

```
 1: procedure BOID(space link, barrier t)
 2:     state me = initial_state ()
 3:     while True do
 4:         state[] all, vis, obs
 5:         sync t                                   ▷ enter observation phase
 6:         all = GET_VIEWABLE(link)
 7:         vis, obs = PRUNE_VISIBLE(all, me)
 8:         me = CENTRE_OF_MASS(vis, me)
 9:         me = REPULSION(vis, me)
10:         me = MEAN_VELOCITY(vis, me)
11:         me = OBSTACLES(obs, me)
12:         UPDATE(link, me)
13:         sync t                                   ▷ enter update phase
14:     end while
15: end procedure
```

Algorithm 1. Boid agent algorithm.

different way: a data-parallel design implemented in the preferred GPU language (C/C++ with OpenCL or C-style with CUDA). For a fairly small system such as the one described, a complete re-implementation on the GPU would ultimately yield better performance, and this has been investigated by others previously [12]. This, however, is not the primary objective of this work, which is to explore ways to improve performance of process-oriented systems using a GPGPU.

1.1. Simulation Operation and Performance

The behaviour of the individual boid processes is shown in Algorithm 1. In each cycle, the boids first synchronise on the barrier (step 5) entering the *observation* phase. Here, the 'viewer' processes associated with each boid's current location are queried (step 6) to retrieve information about viewable agents in the local and neighbouring locations. The information obtained here is essentially an array of data structures containing agent information such as relative position (calculated by the 'viewer' process) and velocity.

The retrieved information is split (step 7) into visible boids and obstacles, based on relative position and velocity. A series of rules are then applied (steps 8 – 11) to compute an acceleration value for the boid, in turn producing a new velocity. This is communicated to the current 'location' process (step 12) and used to update the boid's position. If crossing a boundary, the current location instructs the abstract agent to move to a new location. The boids synchronise on the barrier again (step 13) entering the (empty) *update* phase, before looping and synchronising once again.

To put this code into context, Figure 2 shows the logical state (position and velocity) of boids in a 2 dimensional world. Each of the grid locations shown is implemented by a 'location' and 'viewer' process pair, with agents (both boids and obstacles) connected via their abstract agents to the particular 'location' they inhabit. The positions, velocities and other agent state are represented as floating-point numbers in the program — individual grid locations (the unit space of the simulation) can contain any number of agents at different positions within, up to a predefined maximum. For the boid shown near the centre, whose field-of-vision is indicated, the array generated in step 6 (*all*) will contain information for all the agents shown, including itself. At step 7, this is pruned down into two arrays containing information about the three visible boids (*vis*) and the single visible obstacle (*obs*). For typical flocking behaviours, the ratio of viewable agents to visible ones is generally high, so much of the observed state is discarded at step 7. In practice this is not too significant, since the size

of the communicated agent state is relatively small (32 bytes). For denser simulations or with different parameter sets, however, this contributes significantly to computational overhead.

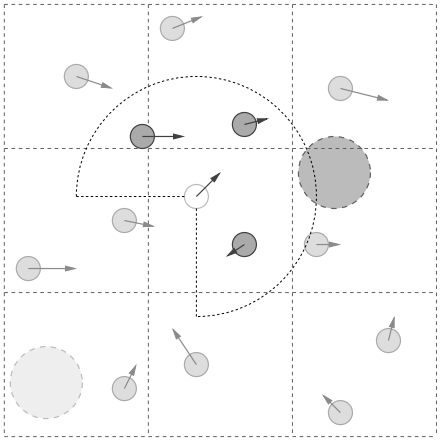


Figure 2. Boids and their world, from the perspective of the white boid near the centre.

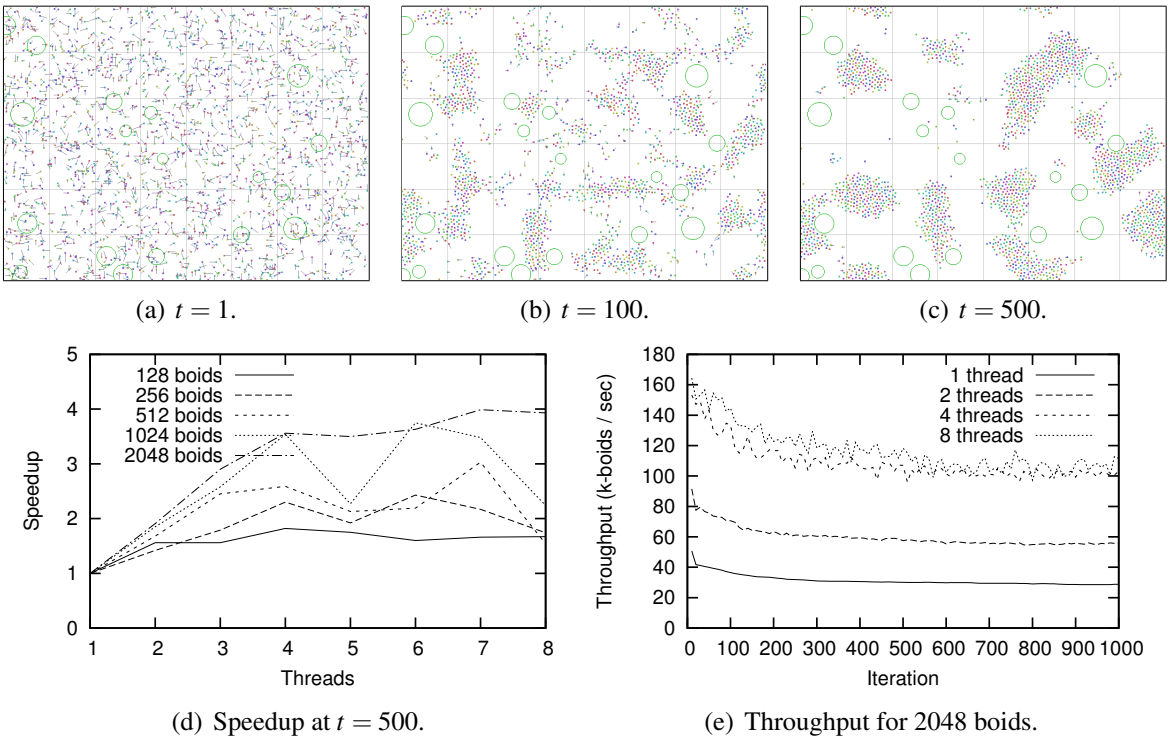
In parallel with the boids and synchronising on the barrier are the 'updater' processes associated with each location. In the update phase these instruct the corresponding viewer to collect state from the connected location processes, providing a consistent view for the next cycle of execution.

Figure 3 shows three screen-shots of a system containing 2048 boids in an 8×6 grid at time steps 1, 100 and 500. As can be seen, from the initial random placement, flocks soon emerge. Figure 3(d) shows the speedup of this simulation on a typical desktop computer: an Intel Quad Core i7 (2600K) running at a fixed 3.4 GHz, with 4 physical cores and 4 hyper-threads. Speedup is only achieved when dealing with large numbers of boids and does not exceed the number of physical cores (4 in this instance). Figure 3(e) shows the simulation throughput, in thousands of boid cycles per second, for 2048 boids over a period of time. Increasing the number of cores has a measurable advantage, but the simulation does slow down once large flocks start to form — roughly from $t = 500$ onwards. Enabling the visualisation reduces performance proportionally with the requested display frame rate and size. To remove these (and other CPU-loading effects) the benchmarks are run with the visualisation disabled.

The performance figures reported are for a single run of the simulation only, with the simulation reporting the cycle time every 10 iterations (the average of the last 10 iterations). The difference reported by separate runs of the simulation, given the same parameters, is small — typically less than 1%. Only showing the results for a single run (instead of averaged over a number of runs) gives a better indication of the variability of performance over time. the 'noise' experienced by 4 and 8 threads in Figure 3(e) (and elsewhere) is due to two factors: the way processes are scheduled by the run-time system [3], that becomes significantly less stable when the CPU is forcibly shared between the simulation and other OS goings on; and the changing distribution of work as boids move between locations, that can create significant workload differences from one cycle to the next.

1.2. Comments on Performance

Little attempt has been made to optimise the code at this point — the simulation was developed in a domain-specific context, using individual concurrent processes to represent different physical artefacts, with the interactions between them suitably abstracted. As a result, the boid behaviour as implemented in Algorithm 1 is straightforward. The boids *see* the relative position and velocities of other agents, deriving their new state from that information plus



(a) $t = 1$. (b) $t = 100$. (c) $t = 500$.

(d) Speedup at $t = 500$. (e) Throughput for 2048 boids.

Figure 3. Boid simulation at successive time steps, showing flocking behaviour (a,b,c); multicore CPU speedup (d) and throughput (e).

their current state. Furthermore, the code is largely obvious and easily understood or adapted by domain experts, not necessarily computer scientists.

For the applications and investigations this simulation was originally created for, performance is sufficient. With the visualisation enabled, 2048 boids in an 8×6 grid can be interacted with in real-time at 25 frames per second, roughly half the speed of the system with visualisation disabled. Given the way it is implemented, the decrease in performance as flocks start to form is not unexpected. Each flock is essentially a localised n-body problem and each boid's own view of the flock is unique (as the implementation deals with relative positions and velocities).

2. Exploring GPGPU Acceleration

To remain compatible with a wide range of GPU hardware, we limit our assumptions about what might be available. Specifically, we consider the GPU as a SIMD processor that supports integer and floating-point operations, without requirements for inter-thread communication, atomic actions or complex data sharing. In practice, a GPU is often much more capable than this, supporting inter-thread synchronisation, atomic manipulation of shared data and possessing significant internal memory bandwidth, with a capability for asynchronous data transfer to/from the host's memory. Taking advantage of such facilities, however, greatly increases the complexity of the resulting system. For the time being, our objective is to retain some simplicity in the resulting code — certainly to the extent that it is still within the grasp of domain experts. Despite limiting the potential performance where GPUs are concerned, not assuming too much makes the approach potentially applicable to other SIMD devices, such as FPGAs and DSPs, though these are outside the scope of this paper.

For our initial experiments as reported here, we have chosen to use CUDA [7] (Compute Unified Device Architecture) to implement code for the GPU. Invoking external C calls in order to interface with the CUDA run-time library is straightforward from occam-π, as are

considerations for data-type compatibility. To keep the implementation simple, the occam-π program handles all GPU interactions transparently via external function calls. Whilst there are other approaches that could perhaps be taken (see Section 3) this is certainly the simplest for the general programmer. The CUDA toolkit from NVIDIA provides a C-like langage for writing the actual GPU *kernel* (code that is executed in parallel on the available stream processors) and a run-time library for accessing the GPU. Although this limits our investigations to NVIDIA's devices, attempts to use other technologies including OpenCL [8] (that targets a wider range of SIMD-style processing hardware) have proven to be less successful (further investigation required).

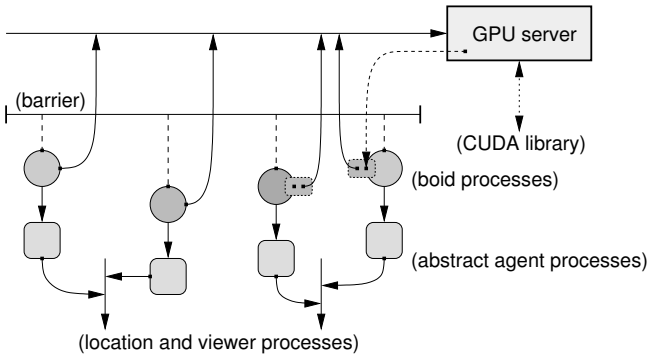
The following sections explore some different strategies aimed at improving performance of the simulation. The first (2.1) treats the GPU as an "on-the-side" computational resource, with all GPU interactions handled through a server process to which other processes can connect and make requests. The second (2.2) considers a different approach: refactoring the simulation to use *shared-memory* to hold the simulation state, in increasing amounts, whose safe access is controlled by the existing barrier. With some guided optimisations to restrict the amount of data copied to or from the GPU, this starts to demonstrate a significant improvement in performance. Section 2.3 considers the combination of these two approaches, with further improved performance as a result. The third approach considered here (2.4) refactors the system in a way that turns the parallelism 'inside-out' — moving from many long-lived high-level concurrent processes (the boids) to a few high-level concurrent processes, with short-lived concurrent processes internally. In effect, collecting all the individual boid processes into one "super-boid" process. The resulting system, containing many short-lived repeating concurrent processes, is more suited to the GPU execution style than the other approaches we consider, and demonstrates considerably better performance as a result.

2.1. A Process-Oriented GPU Server

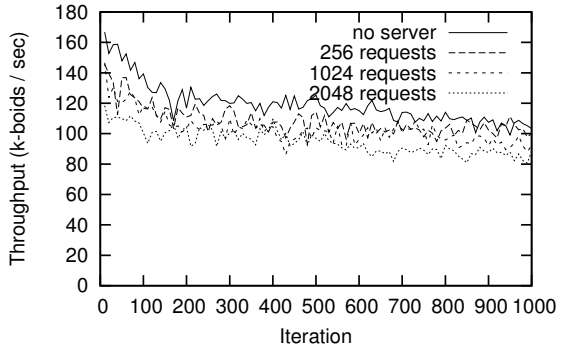
The first approach introduces a "GPU server" process, to which all similar boids are connected, that collects and executes computation requests — shown in Figure 4(a). The operation is straightforward: the server process collects requests (comprising the boid's state, the set of other visible boids, and a channel on which to return results), copies this to the GPU, executes the boid algorithms on the GPU, copies the results back to host memory, and finally communicates these results back to the individual boid processes. The specific parts of Algorithm 1 targeted for GPU execution initially are steps 8 – 10. The code that implements this is a direct by-hand translation from the occam-π, with each boid's algorithm executing in its own GPU thread with its own data.

Figure 4(b) shows the throughput for the basic CPU implementation ("no server") alongside a CPU implementation of the GPU server process, to measure the overhead of the infrastructure. The different lines (256, 1024 and 2048) are for different request-buffer sizes in the GPU server[3]. Since there is only one GPU server process, all the computation is necessarily serialised. Figure 4(c) shows the cycle times for the same basic "no server" implementation alongside those where the same algorithm is executed on the GPU — in this case, one half of an NVIDIA GTX 590 (which is in effect two GTX 580s side-by-side). This shows a general drop in performance, although the throughput is more consistent. Both sets of results suggest

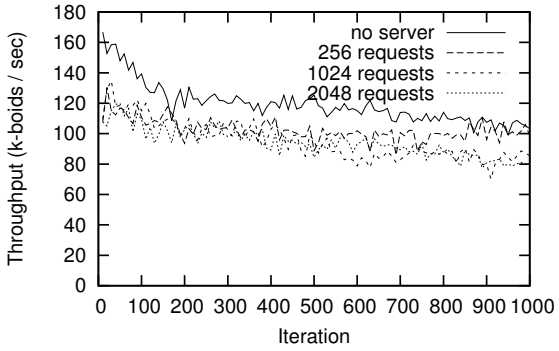
[3]The server process buffers requests from boids (each 32 bytes) together with information about what other boids are visible (each 32 bytes). As such, the amount of data copied to the GPU is $((n*32)+(v*32))$ bytes, where n is the number of requests (512, 1024, 2048) and v is the number of viewable agents, which is typically $\leq (50\times n)$, varying considerably. For 1024 requests, the total amount of data copied to the GPU is typically less than 2 megabytes. When the request buffer is full, or has timed out, these are sent to the GPU as a single job (to be distributed amongst the available stream processors).



(a) Abstracting the GPU resource using a parallel process.



(b) Serial infrastructure only (no GPU).



(c) Using half of a GTX 590 GPU.

Figure 4. Network structure and performance using a GPU server process (2048 boids).

that the targeted parts of Algorithm 1 (steps 8 – 10) account for only a small percentage of execution time.

With some basic profiling, it quickly becomes clear that the majority of time is spent in step 7, involving trigonometry based on the viewing angle and direction of the boids. Given this, the next logical might sensibly be to move more of the algorithm onto the GPU. In practice, however, this causes a significant drop in performance — down to between 20,000 and 50,000 boid cycles per second. The reason is a significant increase in the quantity of boid state that needs to be copied to the GPU. In the previous version (Figure 4) only the state of visible boids was copied (visibility having been determined on the CPU beforehand); here the state of all boids, provided by the viewer process, is copied. When multiplied by the number of boids this quickly becomes significant. In tests, only around 3 – 5 % of viewed boids are actually visible (as determined by typical simulation parameters). For a system of 2048 boids, this data is around 13 megabytes in size.

It is well known that copying data between the host and GPU memories is expensive, so the lack of performance here is not unexpected. Unfortunately, however, for this particular simulation the resulting performance is worse than the CPU-only version for 2 cores. Other similarly structured but operationally different simulations may perform better, however, particularly in cases where the amount of data copied between the host and GPU is small, and the computation is expensive.

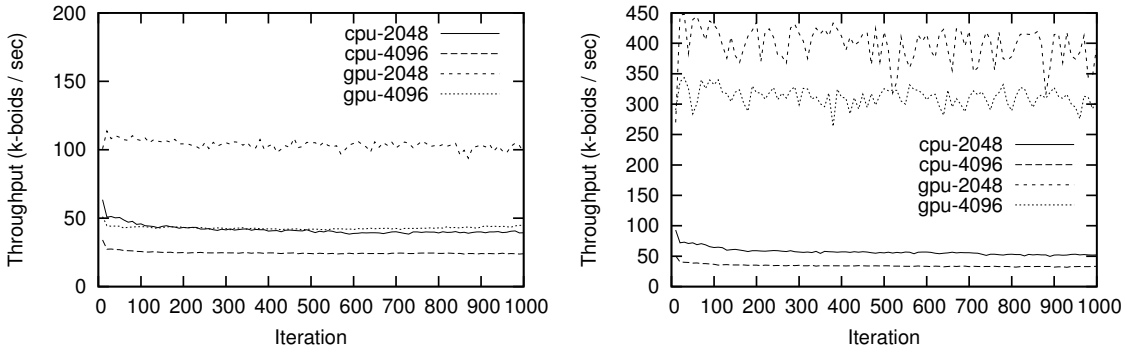
2.2. Rethinking the Problem: Shared Data

The results shown so far demonstrate a reduction in performance — the quantity of data involved and nature of the problem means that naïvely transplanting chunks of code onto the GPU, surrounded by data copies, does not always work well (though this is well known already) and definitely does not work well in this particular case. Whilst there is plenty of opportunity for optimising what we have, such as parallelising the GPU server's operation

for multicore CPUs and multiple GPUs, the limiting factor (and most significant overhead) will still be data copying.

The amount of data (and copies of) involved in the original simulation arises as a result of the way the simulation was implemented — communicating boid state between processes. Once updated, an individual boid's state will be copied for each other boid that can see it, modified to provide a relative position (a simple vector subtraction) as part of the observation. This eludes to a decision in the original implementation, that was to perform this computation for each viewable boid at the point of observation, rather than in each boid.

In an attempt to reduce the amount of data required by the algorithm when running on the GPU, we re-engineered the simulation to utilise shared data for the boid state. Whilst this required fairly extensive modifications to individual processes (reading and writing boid state, including positions that are now absolute) the overall structure of the system (shown in Figure 1) did not change. Furthermore, taking this approach (that we could have done initially) does not much weaken the flexibility of the simulation nor damage any of the existing concurrency abstractions present: global shared data can easily be modelled (or implemented) as a process containing that data, with which other processes communicate. Dynamically creating and destroying boids is no issue, nor is physically distributing the simulation. Instead of communicating boid state, however, the 'location' and 'viewer' processes now only need to be aware of which boids are present (an integer identifier). Race-hazards on the shared state are avoided by the distinct *phases* of operation, marshalled by the barrier.



(a) Non-optimised shared agents only.

(b) Shared agent and viewable data.

Figure 5. Benchmark results of initial shared-data implementations.

For the CPU only implementation, sharing the boid state improves performance by around 50%. Re-introducing the GPU server process infrastructure, but using shared boid state, produces varied results. As shown in Figure 5(a), performance overall for the GPU version is lacking — the CPU results are for a single server process only, thus all the computation is serialised on a single CPU core. With the GPU version, there is still a large amount of data copied to the device — many times the original state-space of the simulation — primarily consisting of the array of viewable boids (and obstacles) returned by the viewer processes. To reduce this, we share the array of viewable agents at each location. This necessitates a change in the boid algorithm implementation, that previously sorted the viewable and relative-position arrays into visible 'boids' and 'obstacles', to instead perform just one pass over the viewable array. The results of this are shown in Figure 5(b), demonstrating an improvement in performance when using the GPU (and for larger numbers of boids).

At this stage, the individual viewer processes extract lists of agents (boids and obstacles) from their accompanying and surrounding location processes, collate these and return to individual boids (via a pointer and size) when requested. As a result, the list of agents in each location is duplicated 8 more times in transit to the GPU — clearly inefficient.

Removing this overhead improves the best-case performance slightly — up to 450,000 boid cycles per second for 4096 boids, where all processing is done in a single GPU kernel run, with each GPU thread performing the computation cycle for a single boid. Although there is no duplicated data, there is some amount of redundant data in the global 'viewable' array that is copied to the GPU. Removing this redundant data (using a shared server process that allocates this space dynamically) had a negligible impact on performance.

An effect of making the viewable state shared is that the 'viewer' processes become redundant, since the contents of other locations can be observed directly. The 'updater' processes remain however, and are used to signal moves between locations, avoiding any races between observing and changing the contents of individual locations.

2.3. Reintroducing High-Level Parallelism

The graphs showing CPU performance in Figure 5 effectively measure the algorithm's execution on a single processor core, as there is only one GPU server process. A reasonable next step is to re-introduce parallelism at the application level, to support multiple CPU cores, multiple GPU devices or a mixture of both. The results of this are shown in Figure 6, with the average throughput against the number of GPU server processes (using the CPU or GPU), and specific throughput for the GPU (using both devices on the GTX-590). The results do not indicate a significant performance improvement — the infrastructure required to implement this (in a way that is compatible with the existing boid code) creates significant overheads, due to the need to synchronise the activities of the processes involved.

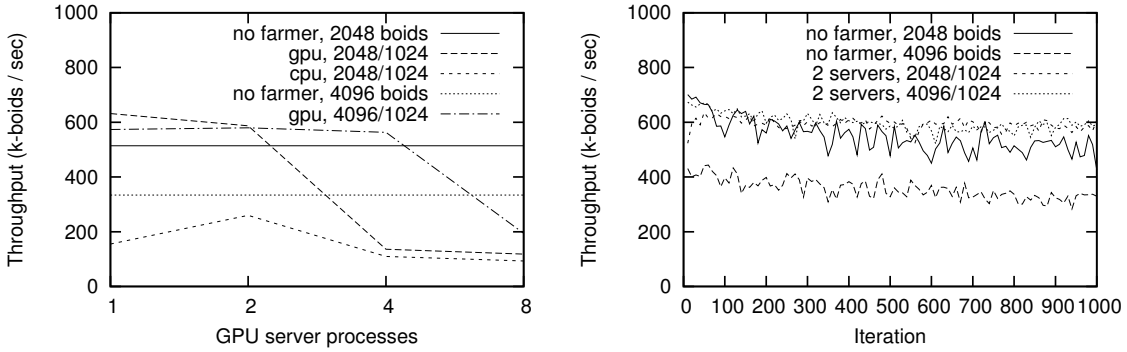


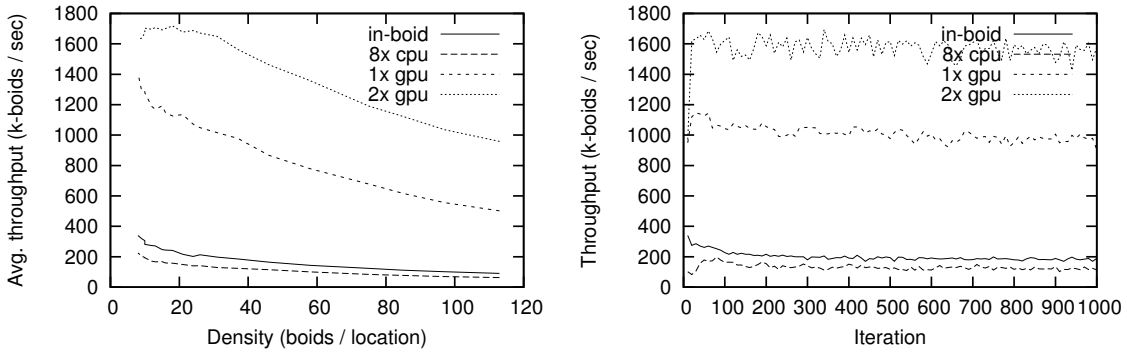
Figure 6. Parallelising the GPU server, 2048 and 4096 boids in varying batch sizes.

Despite the overall lack of performance (compared with a version "no farmer" where the boids execute their algorithms locally in parallel, scheduled over 8 CPU threads) some improvements are observed. Doubling the number of boids to 4096 produces significant improvements on two GPU devices in batches of 2048 (not shown on the graph) — as is partly expected, given where the overhead lies. An advantage of this particular approach is that the boids themselves still see the processing as an abstract resource, to which they send their current state and viewable neighbourhood (via global shared data) and receive back an updated state.

2.4. Removing Concurrency

So far, we have focussed on improving performance in a way that keeps the individual 'boid' processes intact — they are still in control of their own behaviours, even though computation is delegated. As a further optimisation we replace the GPU server process network with a single process that synchronises on the same barrier as the boids, but which performs all computation directly — either on the CPU or the GPU, in parallel as appropriate. To accom-

modate this we introduce an additional phase to the barrier, in which the newly computed velocities of individual boids are written back to the shared data. Removing the explicit channel communication between boids and a centralised server process has significant performance advantages, as shown in Figure 7, where a throughput of around 1.6×10^6 boids-per-second is attained when using two GPU devices at a density of 38 boids per location with $16,384$ boids (in a 24×18 grid). This is a significant increase in the number of boids: previously we had been unable to explore such large systems interactively in real-time, except with fairly specialised hardware platforms such as the *display wall* used in [4]. Furthermore, we are now able to explore aspects such as how the density of a simulation affects throughput (and ultimately the resulting emergent behaviour) — interesting *flocking* behaviours occur between 20 and 60 boids per location; more than this and the simulation becomes over crowded, until it is completely "jammed" at around 120 boids per location.



(a) Varying simulation density.

(b) Throughput at 38 boids/location.

Figure 7. Further optimisations removing channel communication, 16384 boids.

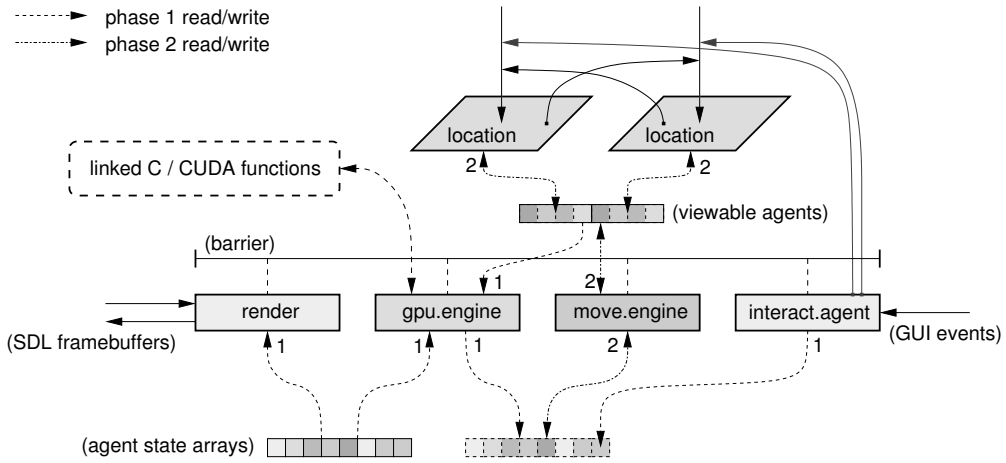
The only remaining activity in the boid processes is initiating movement between locations. As a further optimisation we centralise this in a similar way. The boid processes now contain no activity other synchronising on the barrier — as such, we remove them entirely. The result is a further improvement in performance, at the (arguable) expense of some design clarity (the boids now solely exist as shared global data). Other agents in the system, such as the 'interactive' one, are still easily implemented as concurrent processes, however — as could any complex agents (without interfering with the execution of others on the GPU).

As a further optimisation, we remove the copy-back phase (where the boid state is updated) and instead double-buffer the entire agent state array (including obstacles). Figure 8 shows the high-level design of this version together with benchmark results for up to $30,000$ boids. As the results show, performance is significantly improved, managing a throughput of around 2 million boids-per-second for (roughly) $10,000$ boids and higher using both GPU devices available on the GTX-590.

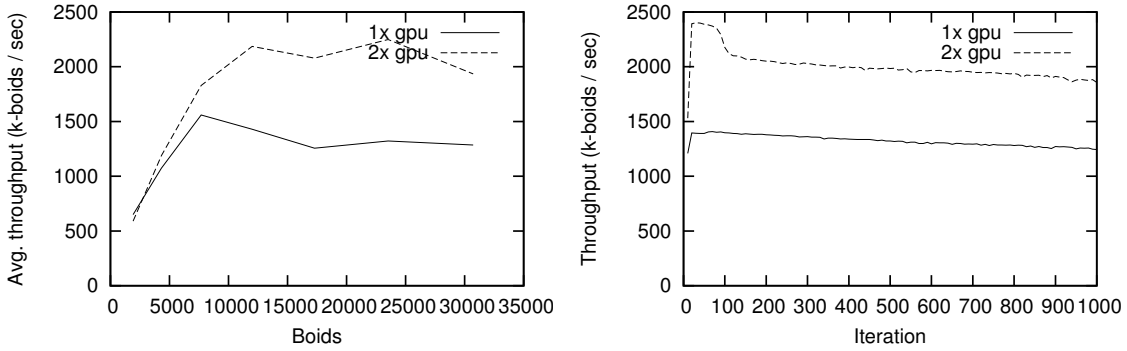
Although we have removed the individual boid and obstacle processes, extending the simulation (as done with the interactive agent and GUI rendering processes) is still a largely straightforward task — simply adding parallel processes with access to the shared state and synchronising on the barrier. The grid of location processes remains for convenience, used only during initialisation (populating the virtual world with boids and obstacles) and for moving the interactive agent that behaves like an obstacle.

3. Conclusions and Future Work

In this paper we have explored some different approaches for speeding-up the execution of fine grained concurrent simulations, using a commodity GPU resource. Starting from a



(a) Simulation structure showing two-phase reads and writes.



(b) Average throughput for varying grid sizes, 40 boids per location.

(c) Throughput for 30,720 boids in a 32×24 grid.

Figure 8. Version of the simulation utilising reads and writes on shared data instead of channel I/O, using a barrier to coordinate access.

CPU-only highly concurrent process-oriented simulation — that scales acceptably well on multicore platforms — we applied successive and generally small modifications with varying results.

Initial attempts at abstracting the GPU resource using a parallel process produced unimpressive results (Figure 4), due primarily to the amount of data copying involved. Further changes to the system included the introduction of shared data, for the agent state (Figure 5(a)) and viewable data (Figure 5(b)), with only minor performance improvements. Reducing the amount of data copied and parallelising the GPU server's execution (Figure 6) gave a 3-4× speedup over the original, but only slightly improved on the best CPU results. Removing the agent-initiated computation (Figure 7) and then the agent processes entirely (Figure 8) produced the best results, with up to 10× speedup over the original.

Although the design of the system has changed somewhat, it is an evolution of an existing system (not a clean-room implementation). The concurrent design of the system, and support for that concurrency in the implementation language used (occam-π) admits a certain flexibility, e.g. changing from a message-passing to a shared-data implementation. The design presented in Figure 8(a) has a good balance between concurrent design, performance and flexibility, but is not perhaps an obvious design. But neither is it the most optimal in terms of execution performance. Despite this, the fact that we can explore and evolve systems in this way is of clear benefit, here and elsewhere.

Being able to run larger (and more complex) simulations was one of our original objectives, as a way of exploring interesting and more complex behaviours in real-time that only

emerge when dealing with larger numbers of agent interactions. Following the same rules from Algorithm 1, Figure 9 shows results for different parameter sets, resulting in different emergent behaviours.

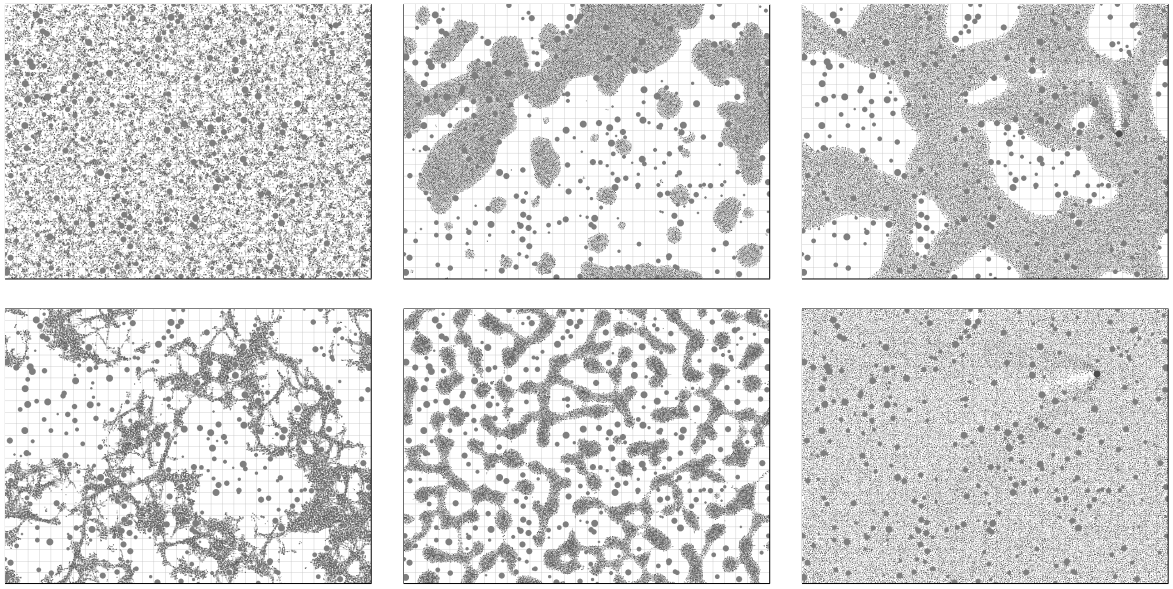


Figure 9. Screen-shots of 32,000 boids and 307 obstacles in a 32×24 grid, with varying parameters changing the behaviour; right-most images show the interactive agent.

The system in Figure 8 handles boid movement in the 'move.engine' process, iterating over the boids (sequentially) and manipulating the viewable arrays directly. This is computationally light and accounts for only a fraction of the overall cycle time ($< 0.05\%$). Running this on the GPU would *probably* be of some performance benefit, but doing so is not straightforward (due to the shared viewable state) — and would restrict the implementation to particular GPU platforms (because of the atomic increments or inter-thread synchronisations required to do this efficiently).

3.1. Lessons Learned

Here we reflect on some of the lessons learned in undertaking this work, and consider how the approaches can (or might) be applied in other contexts.

Perhaps the most obvious observation is that we chose a difficult problem: the way the system was originally implemented did not lend itself to a straightforward GPU implementation (abstracting and offloading the computation by way of a "GPU server" process). The principal overhead was one of memory copies to the GPU — that subsequent optimisations (shared memory) sought to overcome. In simulations where the amount of data involved (though not necessarily the number of agents) is significantly smaller or where the amount of computation performed is greater, we would expect the initial approach to perform substantially better.

In terms of applying communicating process ideas to GPU programming, we believe this has merit. In typical GPGPU simulations, the activities of the host and the GPU are interlocked, often with only one OS thread on the host side. The occam-π concurrency model provides a level of parallel flexibility that does not exist in ordinary (C-style) procedural programming — utilising all available CPU and GPU processing capabilities from within a single application becomes an easy prospect, saving the programmer from potentially complex thread synchronisation and race-hazard issues. Furthermore, adding extra functionality "on the side" is similarly straightforward, even if that extra functionality is largely independent of the existing system (e.g. a process that periodically samples the simulation for correctness).

3.2. Future Work

Whilst we have been successful in our efforts to improve performance, there remain several avenues of future work to be explored, some of which we would expect to yield a further improvement in performance.

The work presented here has focussed on one particular and quite simple approach to accessing SIMD-style GPU resources from MIMD-style process-oriented programs. Specifically, hiding all GPU interaction behind single external function calls, passing data to and from the GPU at that point. Whilst this is straightforward, and potentially lends itself to other SIMD architectures, it does not necessarily allow for optimal performance — every cycle of the simulation requires the entire simulation state to be copied to the GPU, and the changed state back again, before the next cycle can start.

In light of this, an alternative approach (and as a further modification to the simulation) is to maintain the simulation state on the GPU foremost, with all the computation there, transferring the resulting state to the CPU when necessary. Although this would likely result in better performance for this particular simulation, it is harder to guarantee in the general case, particularly where a range of different agent types is involved (not just boids and data-only obstacles).

No significant attempts have yet been made at optimising our hand-written CUDA. Some work with the CUDA profiling tools would reveal any deficiencies in our CUDA kernels, pointing the way to further optimisation. Similarly, we do not attempt to make efficient use of constant memory on the GPU, nor change the way CUDA threads are arranged, both of which can have a significant (and positive) impact on performance [13]. To better understand the performance ceiling, it would be worthwhile implementing a GPU-only version (with no occam-π concurrency at all). There has been past work looking specifically at GPU and other accelerated implementations of Reynolds' boids [14,12], that perform well with a variety of optimisation techniques, but little in recent years that considers the same fundamental algorithm.

Given the facilities present in modern GPUs, such as inter-thread synchronisation, atomic operations and direct access to host memory, one possibility would be to implement channel-style communication algorithms on the GPU itself, perhaps in each GPU thread, presenting the GPU resource as an array of parallel processes that can be communicated with directly. How this would work in practice is perhaps unclear, as different GPU devices provide different capabilities in differing amounts (e.g. CUDA GPU threads can only be synchronised within *thread blocks*, not across all threads). The general (developer community) view is that this sort of approach does not work well, given that the GPU infrastructure is foremost intended for, and largely assumes, highly-parallel short-lived tasks — i.e. pixel shading.

The work presented here has required the existing occam-π code to be re-implemented in CUDA, although the structure and operations on data are a direct hand translation. Making this automatic, by having the compiler generate CUDA kernels and appropriate interfaces for identified portions of the source program, would make the process of accessing GPU and other similar SIMD resources significantly simpler. Whether such a mechanism would prove to be useful (in terms of improved performance with minimal effort) remains to be seen, and would require some consideration of automatic refactorisation to be of benefit to simulations such as our original 'boids'.

Whilst not directly relevant to the benchmarks reported in this paper, there is scope for additional optimisation when visualisation is enabled. This is currently handled from the CPU, via the 'display' process (Figure 1). At each cycle of the display (which we allow to run slower than the simulation if desired) the painted framebuffer is communicated to the GPU, that then displays it on the screen (via the host's windowing system). Where the GPU being used for computation is the same as that being used for display, there is a clear

benefit in generating the framebuffer (and keeping it) on the GPU. Although not entirely straightforward, this is well documented and can be done through the use of OpenGL textures.

3.3. Source Code

For the CUDA-enabled occam-π user, this simulation is now part of the standard KRoC distribution (since version 1.6.0), together with a CUDA wrapper library "ocuda" and other small example programs. KRoC is now hosted on GitHub: `https://github.com/concurrency/kroc/`.

Acknowledgements

The authors would like to thank the anonymous reviewers for valuable feedback on an earlier version of this work, and acknowledge the support of the Faculty of Sciences at the University of Kent (who provided funding for the GPU hardware used).

References

[1] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *SIGGRAPH Comput. Graph.*, 21(4):25–34, August 1987.

[2] P.H. Welch and F.R.M. Barnes. Communicating mobile processes: introducing occam-pi. In A.E. Abdallah, C.B. Jones, and J.W. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005.

[3] C.G. Ritson, A.T. Sampson, and F.R.M. Barnes. Multicore scheduling for lightweight communicating processes. *Science of Computer Programming*, 77(6):727–740, June 2012.

[4] Adam T. Sampson, John Markus Bjørndalen, and Paul S. Andrews. Birds on the wall: Distributing a process-oriented simulation. In *2009 IEEE Congress on Evolutionary Computation (CEC 2009)*, pages 225–231. IEEE Press, 2009.

[5] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985. ISBN: 0-13-153271-5.

[6] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999. ISBN: 0-52165-869-1.

[7] NVIDIA. CUDA C programming guide 4.2, April 2012. `http://www.nvidia.com/content/cuda/cuda-developer-resources.html`.

[8] Khronos OpenCL Working Group. The OpenCL specification 1.2, November 2011. `http://www.khronos.org/registry/cl/`.

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using cuda. *Journal of Parallel and Distributed Computing*, 68(10):1370 – 1380, 2008.

[10] Richard Vuduc, Aparna Chandramowlishwaran, Jee Whan Choi, Murat Efe Guney, and Aashay Shringarpure. On the limits of GPU acceleration. In *Proc. USENIX Wkshp. Hot Topics in Parallelism (HotPar)*, Berkeley, CA, USA, June 2010.

[11] S. Stepney, P.H. Welch, J. Timmis, C. Alexander, F.R.M. Barnes, M. Bates, F.A.C. Polack, and A. Tyrrell. CoSMoS: Complex systems modelling and simulation infrastructure, April 2007. EPSRC grants EP/E053505/1 and EP/E049419/1. URL: `http://www.cosmos-research.org/`.

[12] Alessandro Ribeiro da Silva, Wallace Santos Lages, and Luiz Chaimowicz. Boids that see: Using self-occlusion for simulating large groups on GPUs. *Comput. Entertain.*, 7(4):51:1–51:20, January 2010.

[13] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Pearson Education, 2011.

[14] Craig Reynolds. Big fast crowds on PS3. In *Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, Sandbox '06, pages 113–121, New York, NY, USA, 2006. ACM.