Exploring GPGPU Acceleration of Process-Oriented Simulations

Communicating Process Architectures 2013

Fred Barnes
School of Computing, University of Kent, Canterbury

F.R.M.Barnes@kent.ac.uk

http://www.cs.kent.ac.uk/~frmb/

Programming Languages and Systems

University of Kent | Computing

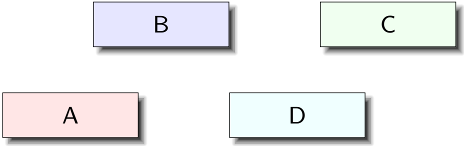
Contents

Process-Oriented Programming

- Building systems with **concurrent processes** as the bricks.
  - processes communicate and synchronise using **channels** and **barriers** (the mortar).
  - communication is synchronised, unidirectional and unbuffered.
- We use the **occam-π** language [1] for implementation.
  - based heavily on the semantics of CSP [2].
  - ideas of dynamics and mobility from the π-calculus [3].

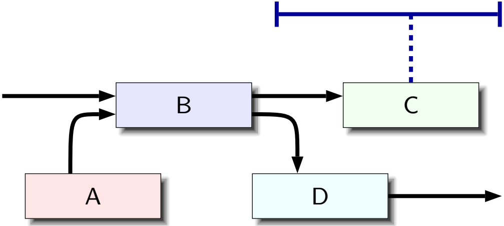
Process-Oriented Programming

- Building systems with **concurrent processes** as the bricks.
  - processes communicate and synchronise using **channels** and **barriers** (the mortar).
  - communication is synchronised, unidirectional and unbuffered.
- We use the **occam-π** language [1] for implementation.
  - based heavily on the semantics of CSP [2].
  - ideas of dynamics and mobility from the π-calculus [3].



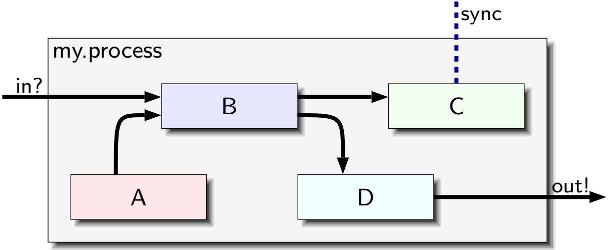
Process-Oriented Programming

- Building systems with **concurrent processes** as the bricks.
  - processes communicate and synchronise using **channels** and **barriers** (the mortar).
  - communication is synchronised, unidirectional and unbuffered.
- We use the **occam-π** language [1] for implementation.
  - based heavily on the semantics of CSP [2].
  - ideas of dynamics and mobility from the π-calculus [3].



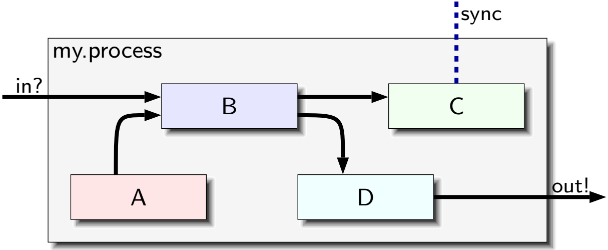
Process-Oriented Programming

- Building systems with **concurrent processes** as the bricks.
  - processes communicate and synchronise using **channels** and **barriers** (the mortar).
  - communication is synchronised, unidirectional and unbuffered.
- We use the **occam-π** language [1] for implementation.
  - based heavily on the semantics of CSP [2].
  - ideas of dynamics and mobility from the π-calculus [3].



Process-Oriented Programming

- Building systems with **concurrent processes** as the bricks.
  - processes communicate and synchronise using **channels** and **barriers** (the mortar).
  - communication is synchronised, unidirectional and unbuffered.
- We use the **occam-π** language [1] for implementation.
  - based heavily on the semantics of CSP [2].
  - ideas of dynamics and mobility from the π-calculus [3].



Process-Oriented Programming

- Channels are **first class** types, so can have channels carrying channels (or rather, **channel ends**).
  - enables networks of processes to **reconfigure** themselves dynamically.
  - can have **shared channel-ends**, whose mutually exclusive access is protected by a **fair-queueing semaphore**.
- Processes can **alternate** (select) between multiple **channel inputs** and **timeouts**, with optional priority.
  - external choice in CSP, more or less.
- Can build **large** systems ($10^4 - 10^6$ processes) using layered networks of communicating processes, that grow, shrink and evolve at run-time.
  - need to be aware of dangers such as **deadlock**, **livelock** and **starvation** (good design).

Process-Oriented Programming

- Channels are **first class** types, so can have channels carrying channels (or rather, **channel ends**).
  - enables networks of processes to **reconfigure** themselves dynamically.
  - can have **shared channel-ends**, whose mutually exclusive access is protected by a **fair-queueing semaphore**.
- Processes can **alternate** (select) between multiple **channel inputs** and **timeouts**, with optional priority.
  - external choice in CSP, more or less.
- Can build **large** systems ($10^4 - 10^6$ processes) using layered networks of communicating processes, that grow, shrink and evolve at run-time.
  - need to be aware of dangers such as **deadlock**, **livelock** and **starvation** (good design).

Process-Oriented Programming

- Channels are **first class** types, so can have channels carrying channels (or rather, **channel ends**).
    - enables networks of processes to **reconfigure** themselves dynamically.
    - can have **shared channel-ends**, whose mutually exclusive access is protected by a **fair-queueing semaphore**.
- Processes can **alternate** (select) between multiple **channel inputs** and **timeouts**, with optional priority.
    - external choice in CSP, more or less.
- Can build **large** systems ($10^4 - 10^6$ processes) using layered networks of communicating processes, that grow, shrink and evolve at run-time.
    - need to be aware of dangers such as **deadlock**, **livelock** and **starvation** (good design).

Not a Talk About occam-π

- For the purpose of this talk, pictures are sufficient.
    - the graphical representation we have for process networks maps **cleanly** to and from code.
- Not entirely dissimilar languages **Erlang** (Sony Ericsson) and **Go** (Google) do similar things — some intersection of features.
    - no assumption about **sequential** execution in occam-π: equal syntax standing with **concurrent** execution (`SEQ` vs. `PAR`).
- Perhaps more relevant is the tool-chain and the run-time system (**CCSP** [4]).
    - compiled to native code for **fast** execution (though not optimal).
    - **small** overheads for channels (4 bytes) and processes (32 bytes minimum).

Not a Talk About occam-π

- For the purpose of this talk, pictures are sufficient.
  - the graphical representation we have for process networks maps **cleanly** to and from code.
- Not entirely dissimilar languages **Erlang** (Sony Ericsson) and **Go** (Google) do similar things — some intersection of features.
  - no assumption about **sequential** execution in occam-π: equal syntax standing with **concurrent** execution (`SEQ` vs. `PAR`).
- Perhaps more relevant is the tool-chain and the run-time system (**CCSP** [4]).
  - compiled to native code for **fast** execution (though not optimal).
  - **small** overheads for channels (4 bytes) and processes (32 bytes minimum).

Not a Talk About occam-π

- For the purpose of this talk, pictures are sufficient.
    - the graphical representation we have for process networks maps **cleanly** to and from code.
- Not entirely dissimilar languages **Erlang** (Sony Ericsson) and **Go** (Google) do similar things — some intersection of features.
    - no assumption about **sequential** execution in occam-π: equal syntax standing with **concurrent** execution (`SEQ` vs. `PAR`).
- Perhaps more relevant is the tool-chain and the run-time system (**CCSP** [4]).
    - compiled to native code for **fast** execution (though not optimal).
    - **small** overheads for channels (4 bytes) and processes (32 bytes minimum).

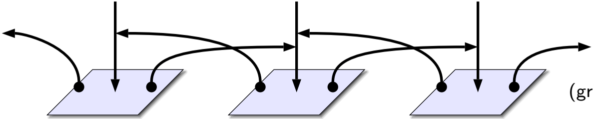
The Boids Simulation

- A good case study — it is not **trivially parallelisable**.
  - **fractal** generators and Conway's **game-of-life** *are* trivially parallelisable and give the expected speedups when running with the GPU (×300 or more).
- An n-body problem, but where n is kept manageable by **partitioning** the world into a regular grid.
- Produced originally as part of the **CoSMoS** project [5, 6].
  - based on Reynolds' "boids" [7].

The Boids Simulation

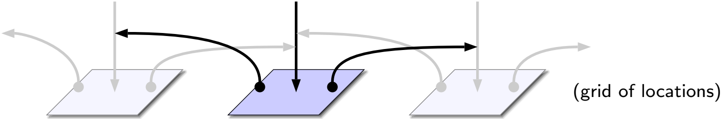
- A good case study — it is not **trivially parallelisable**.
  - **fractal** generators and Conway's **game-of-life** *are* trivially parallelisable and give the expected speedups when running with the GPU (×300 or more).
- An *n*-body problem, but where *n* is kept manageable by **partitioning** the world into a regular grid.
- Produced originally as part of the **CoSMoS** project [5, 6].
  - based on Reynolds' "boids" [7].

The Boids Simulation

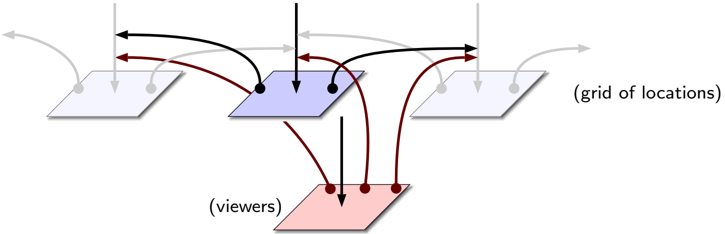


(grid of locations)

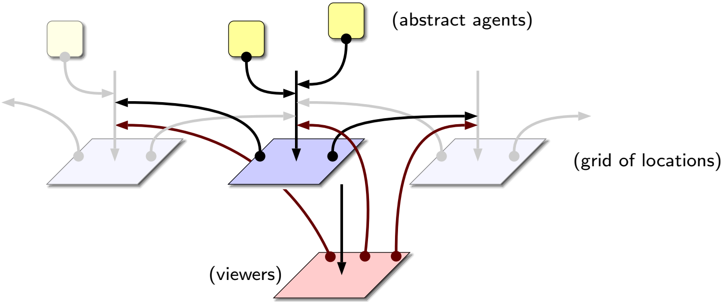
The Boids Simulation



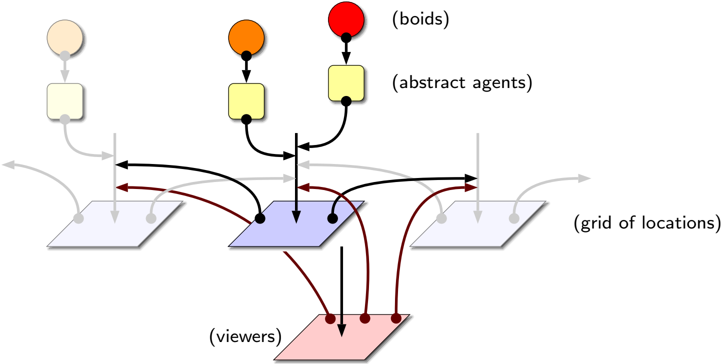
The Boids Simulation



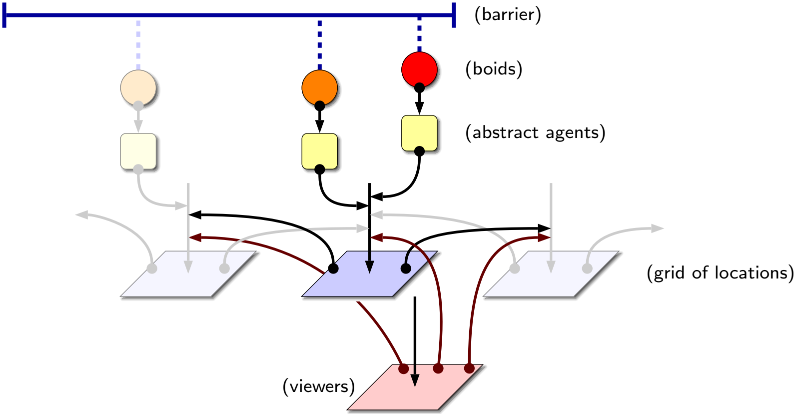
The Boids Simulation



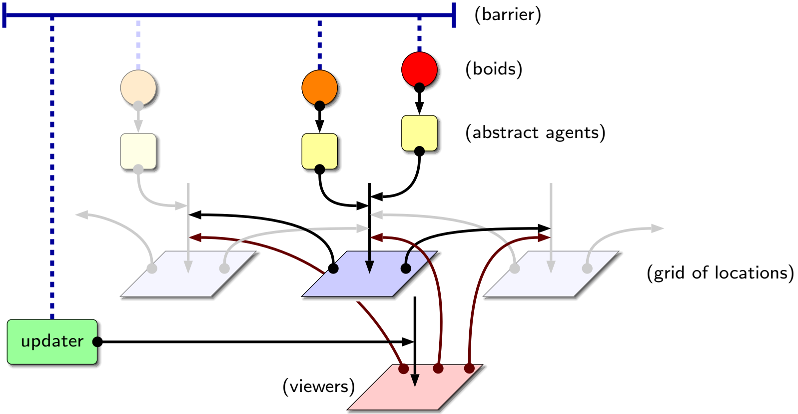
The Boids Simulation



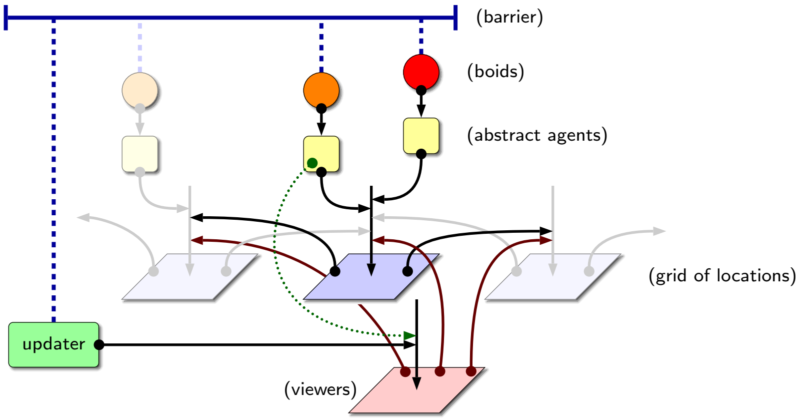
The Boids Simulation



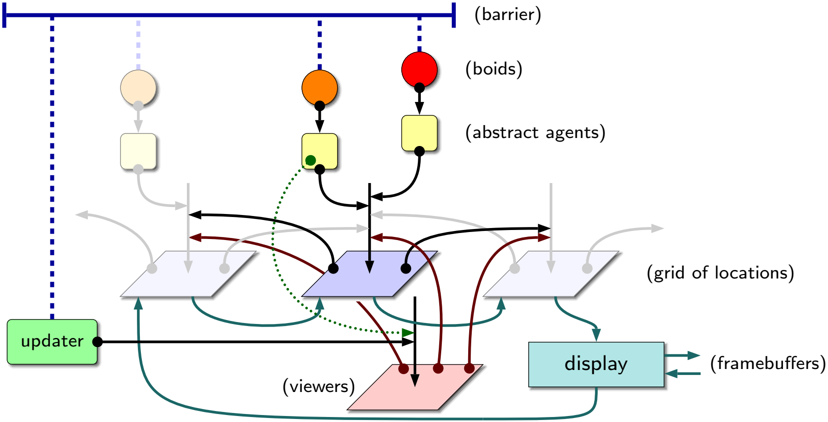
The Boids Simulation



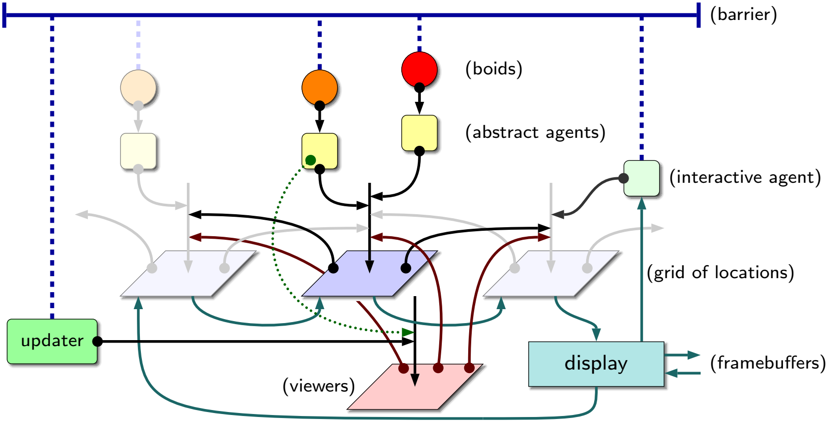
The Boids Simulation



The Boids Simulation



The Boids Simulation



Simulation Operation

- World is defined using a grid of **location** processes.
    - each location has a **viewer**, and each viewer has an **updater**.
- **Boid** processes do not interact with locations and viewers directly.
    - instead interacting with an **abstract agent**, that in turn handles interaction with the world (and its particular geometry).
- The **barrier** divides simulation execution into **two phases**.
    - Phase 1:
        - processes synchronise on the barrier.
        - via the abstract agent and viewer, neighbour discovery.
        - compute new acceleration and velocity.
        - move if needed.
    - Phase 2:
        - processes synchronise on the barrier.
        - viewers update from locations.

Simulation Operation

- World is defined using a grid of **location** processes.
    - each location has a **viewer**, and each viewer has an **updater**.
- **Boid** processes do not interact with locations and viewers directly.
    - instead interacting with an **abstract agent**, that in turn handles interaction with the world (and its particular geometry).
- The **barrier** divides simulation execution into **two phases**.
    - Phase 1:
        - processes synchronise on the barrier.
        - via the abstract agent and viewer, neighbour discovery.
        - compute new acceleration and velocity.
        - move if needed.
    - Phase 2:
        - processes synchronise on the barrier.
        - viewers update from locations.

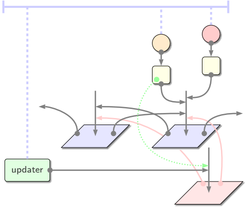
Simulation Operation

- World is defined using a grid of **location** processes.
  - each location has a **viewer**, and each viewer has an **updater**.
- **Boid** processes do not interact with locations and viewers directly.
  - instead interacting with an **abstract agent**, that in turn handles interaction with the world (and its particular geometry).
- The **barrier** divides simulation execution into **two phases**.

- Phase 1:
  - processes synchronise on the barrier.
  - via the abstract agent and viewer, neighbour discovery.
  - compute new acceleration and velocity.
  - move if needed.
- Phase 2:
  - processes synchronise on the barrier.
  - viewers update from locations.

Simulation Operation

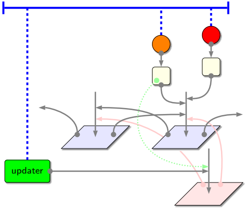
- World is defined using a grid of **location** processes.
  - each location has a **viewer**, and each viewer has an **updater**.
- **Boid** processes do not interact with locations and viewers directly.
  - instead interacting with an **abstract agent**, that in turn handles interaction with the world (and its particular geometry).
- The **barrier** divides simulation execution into **two phases**.



- Phase 1:
  - processes synchronise on the barrier.
  - via the abstract agent and viewer, neighbour discovery.
  - compute new acceleration and velocity.
  - move if needed.
- Phase 2:
  - processes synchronise on the barrier.
  - viewers update from locations.

Simulation Operation

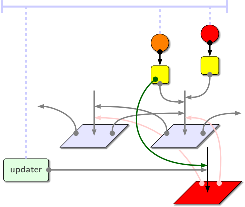
- World is defined using a grid of **location** processes.
    - each location has a **viewer**, and each viewer has an **updater**.
- **Boid** processes do not interact with locations and viewers directly.
    - instead interacting with an **abstract agent**, that in turn handles interaction with the world (and its particular geometry).
- The **barrier** divides simulation execution into **two phases**.



- Phase 1:
    - processes synchronise on the barrier.
    - via the abstract agent and viewer, neighbour discovery.
    - compute new acceleration and velocity.
    - move if needed.
- Phase 2:
    - processes synchronise on the barrier.
    - viewers update from locations.

Simulation Operation

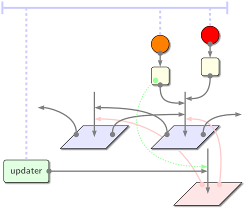
- World is defined using a grid of **location** processes.
  - each location has a **viewer**, and each viewer has an **updater**.
- **Boid** processes do not interact with locations and viewers directly.
  - instead interacting with an **abstract agent**, that in turn handles interaction with the world (and its particular geometry).
- The **barrier** divides simulation execution into **two phases**.



- Phase 1:
  - processes synchronise on the barrier.
  - via the abstract agent and viewer, neighbour discovery.
  - compute new acceleration and velocity.
  - move if needed.
- Phase 2:
  - processes synchronise on the barrier.
  - viewers update from locations.

Simulation Operation

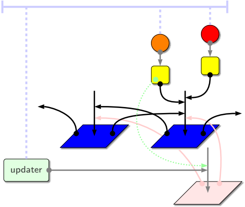
- World is defined using a grid of **location** processes.
  - each location has a **viewer**, and each viewer has an **updater**.
- **Boid** processes do not interact with locations and viewers directly.
  - instead interacting with an **abstract agent**, that in turn handles interaction with the world (and its particular geometry).
- The **barrier** divides simulation execution into **two phases**.



- Phase 1:
  - processes synchronise on the barrier.
  - via the abstract agent and viewer, neighbour discovery.
  - compute new acceleration and velocity.
  - move if needed.
- Phase 2:
  - processes synchronise on the barrier.
  - viewers update from locations.

Simulation Operation

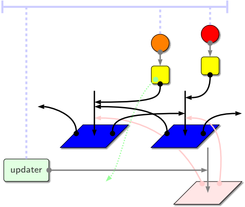
- World is defined using a grid of **location** processes.
    - each location has a **viewer**, and each viewer has an **updater**.
- **Boid** processes do not interact with locations and viewers directly.
    - instead interacting with an **abstract agent**, that in turn handles interaction with the world (and its particular geometry).
- The **barrier** divides simulation execution into **two phases**.



- Phase 1:
    - processes synchronise on the barrier.
    - via the abstract agent and viewer, neighbour discovery.
    - compute new acceleration and velocity.
    - move if needed.
- Phase 2:
    - processes synchronise on the barrier.
    - viewers update from locations.

Simulation Operation

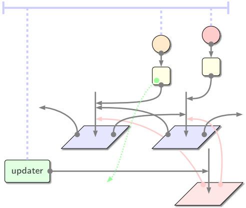
- World is defined using a grid of **location** processes.
  - each location has a **viewer**, and each viewer has an **updater**.
- **Boid** processes do not interact with locations and viewers directly.
  - instead interacting with an **abstract agent**, that in turn handles interaction with the world (and its particular geometry).
- The **barrier** divides simulation execution into **two phases**.



- Phase 1:
  - processes synchronise on the barrier.
  - via the abstract agent and viewer, neighbour discovery.
  - compute new acceleration and velocity.
  - move if needed.
- Phase 2:
  - processes synchronise on the barrier.
  - viewers update from locations.

Simulation Operation

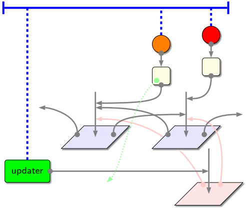
- World is defined using a grid of **location** processes.
    - each location has a **viewer**, and each viewer has an **updater**.
- **Boid** processes do not interact with locations and viewers directly.
    - instead interacting with an **abstract agent**, that in turn handles interaction with the world (and its particular geometry).
- The **barrier** divides simulation execution into **two phases**.



- Phase 1:
    - processes synchronise on the barrier.
    - via the abstract agent and viewer, neighbour discovery.
    - compute new acceleration and velocity.
    - move if needed.
- Phase 2:
    - processes synchronise on the barrier.
    - viewers update from locations.

Simulation Operation

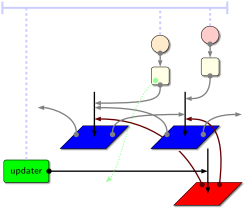
- World is defined using a grid of **location** processes.
  - each location has a **viewer**, and each viewer has an **updater**.
- **Boid** processes do not interact with locations and viewers directly.
  - instead interacting with an **abstract agent**, that in turn handles interaction with the world (and its particular geometry).
- The **barrier** divides simulation execution into **two phases**.



- Phase 1:
  - processes synchronise on the barrier.
  - via the abstract agent and viewer, neighbour discovery.
  - compute new acceleration and velocity.
  - move if needed.
- Phase 2:
  - processes synchronise on the barrier.
  - viewers update from locations.

Simulation Operation

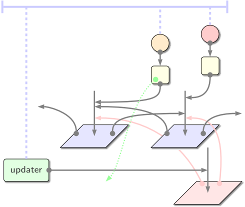
- World is defined using a grid of **location** processes.
  - each location has a **viewer**, and each viewer has an **updater**.
- **Boid** processes do not interact with locations and viewers directly.
  - instead interacting with an **abstract agent**, that in turn handles interaction with the world (and its particular geometry).
- The **barrier** divides simulation execution into **two phases**.



- Phase 1:
  - processes synchronise on the barrier.
  - via the abstract agent and viewer, neighbour discovery.
  - compute new acceleration and velocity.
  - move if needed.
- Phase 2:
  - processes synchronise on the barrier.
  - viewers update from locations.

Simulation Operation

- World is defined using a grid of **location** processes.
  - each location has a **viewer**, and each viewer has an **updater**.
- **Boid** processes do not interact with locations and viewers directly.
  - instead interacting with an **abstract agent**, that in turn handles interaction with the world (and its particular geometry).
- The **barrier** divides simulation execution into **two phases**.



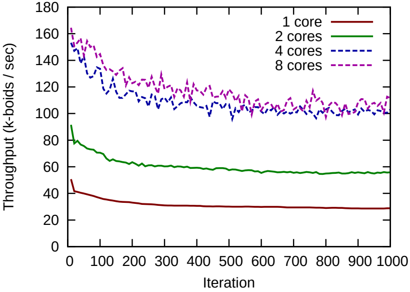
- Phase 1:
  - processes synchronise on the barrier.
  - via the abstract agent and viewer, neighbour discovery.
  - compute new acceleration and velocity.
  - move if needed.
- Phase 2:
  - processes synchronise on the barrier.
  - viewers update from locations.

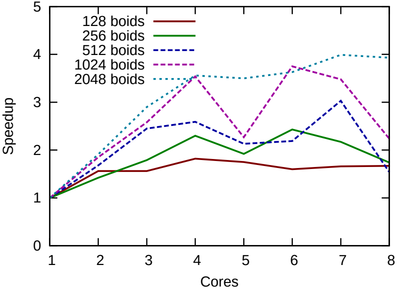
From the Boids' Perspective

```
procedure BOID(space link, barrier t)
    state me = initial_state ()
    while True do
        sync t                                        ▷ enter observation phase
        all = GET_VIEWABLE(link)
        vis, obs = PRUNE_VISIBLE(all, me)
        me = CENTRE_OF_MASS(vis, me)
        me = REPULSION(vis, me)
        me = MEAN_VELOCITY(vis, me)
        me = OBSTACLES(obs, me)
        UPDATE(link, me)
        sync t                                        ▷ enter update phase
    end while
end procedure
```

Performance

- For **2048 boids** and **9 obstacles** in an **8×6** grid.
    - test machine is an Intel Quad Core i7 (2600K) running at 3.4 GHz (fixed); 4 real cores & 4 hyperthreads.

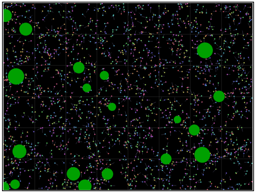




- Performance drops as flocks start to form (*n*-body effect).
    - levels out to around **50 cycles/sec**.

Visualisation

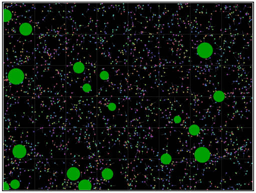
- Some of the process plumbing is used for a **display**:
  - uses **SDL** to display 2D framebuffers on a host display (and, separately, allows capture to files).
  - in **interactive** mode, can adjust simulation parameters and move an obstacle around.



- This is about as good as the original version will manage.
  - could tweak it for more performance based on parameter values, but not expecting substantial improvements.
- **Solution:** use the GPU to speed things up!

Visualisation

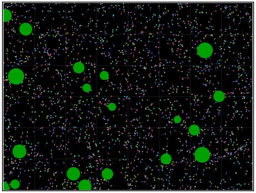
- Some of the process plumbing is used for a **display**:
  - uses **SDL** to display 2D framebuffers on a host display (and, separately, allows capture to files).
  - in **interactive** mode, can adjust simulation parameters and move an obstacle around.



- This is about as good as the original version will manage.
  - could tweak it for more performance based on parameter values, but not expecting substantial improvements.
- **Solution:** use the GPU to speed things up!

Visualisation

- Some of the process plumbing is used for a **display**:
  - uses **SDL** to display 2D framebuffers on a host display (and, separately, allows capture to files).
  - in **interactive** mode, can adjust simulation parameters and move an obstacle around.



- This is about as good as the original version will manage.
  - could tweak it for more performance based on parameter values, but not expecting substantial improvements.
- **Solution:** use the GPU to speed things up!

A Brief History of GPUs

Intel release the iSBX 275 multibus board, providing accelerated drawing of lines, arcs, rectangles and character bitmaps.

1983

A Brief History of GPUs

Intel release the iSBX 275 multibus board, providing accelerated drawing of lines, arcs, rectangles and character bitmaps.

1983

1985

first personal computer graphics processor appears in the Commodore Amiga: line drawing, area fill and blitter. Included a graphics co-processor with a primitive instruction set.

A Brief History of GPUs

Intel release the iSBX 275 multibus board, providing accelerated drawing of lines, arcs, rectangles and character bitmaps.

IBM release the 8514/A for the PS/2 (MCA bus): line drawing, area fill and blitter.

first personal computer graphics processor appears in the Commodore Amiga: line drawing, area fill and blitter. Included a graphics co-processor with a primitive instruction set.

A Brief History of GPUs

S3 Graphics introduce the S3 86C911, designed to accelerate specific software. Responsible for many similar (accelerator) cards.

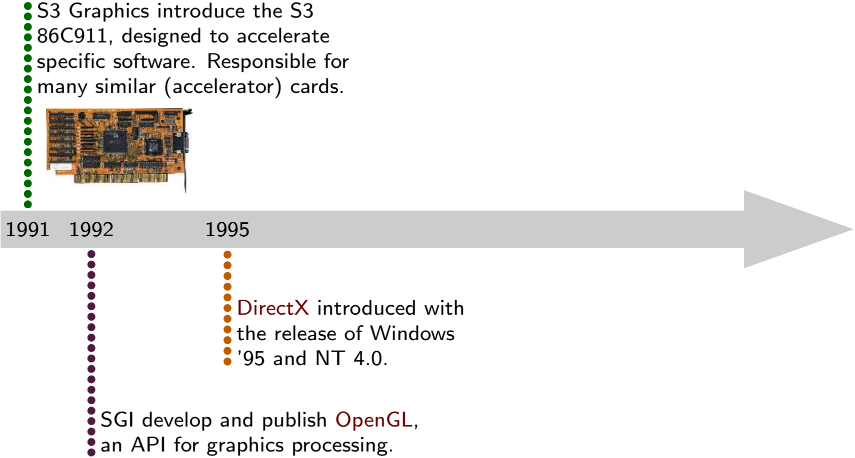
1991

A Brief History of GPUs

S3 Graphics introduce the S3 86C911, designed to accelerate specific software. Responsible for many similar (accelerator) cards.

SGI develop and publish OpenGL, an API for graphics processing.

A Brief History of GPUs



S3 Graphics introduce the S3 86C911, designed to accelerate specific software. Responsible for many similar (accelerator) cards.

DirectX introduced with the release of Windows ’95 and NT 4.0.

SGI develop and publish OpenGL, an API for graphics processing.

A Brief History of GPUs

1991

S3 Graphics introduce the S3 86C911, designed to accelerate specific software. Responsible for many similar (accelerator) cards.

1992

SGI develop and publish OpenGL, an API for graphics processing.

1995

DirectX introduced with the release of Windows '95 and NT 4.0.

1999

NVIDIA popularise the term Graphics Processing Unit.

A Brief History of GPUs

1991 S3 Graphics introduce the S3 86C911, designed to accelerate specific software. Responsible for many similar (accelerator) cards.

1992 SGI develop and publish OpenGL, an API for graphics processing.

1995 DirectX introduced with the release of Windows '95 and NT 4.0.

1999 NVIDIA popularise the term Graphics Processing Unit.

2001 NVIDIA releases the GeForce 3, included a programmable shader. Start of the GPGPU era.

A Brief History of GPUs

1991

S3 Graphics introduce the S3 86C911, designed to accelerate specific software. Responsible for many similar (accelerator) cards.

1992

SGI develop and publish OpenGL, an API for graphics processing.

1995

DirectX introduced with the release of Windows '95 and NT 4.0.

1999

NVIDIA popularise the term Graphics Processing Unit.

2001

NVIDIA releases the GeForce 3, included a programmable shader. Start of the GPGPU era.

2002

ATI try and introduce Visual Processing Unit (VPU) into the lexicon, unsuccessfully.

A Brief History of GPUs

1991

S3 Graphics introduce the S3 86C911, designed to accelerate specific software. Responsible for many similar (accelerator) cards.

1992

SGI develop and publish OpenGL, an API for graphics processing.

1995

DirectX introduced with the release of Windows '95 and NT 4.0.

1999

NVIDIA popularise the term Graphics Processing Unit.

2001

NVIDIA releases the GeForce 3, included a programmable shader. Start of the GPGPU era.

2002

ATI try and introduce Visual Processing Unit (VPU) into the lexicon, unsuccessfully.

GPUs able to handle looping and floating-point intensive shader 'mini-programs'.

A Brief History of GPUs

- Specific **graphics co-processors** existed in the 1980s and 1990s, but not in the general consumer market.
    - fixed-feature hardware accelerators (DirectX) cheaper and faster.
- Recent GPU cards offer significant computational ability, driven largely by the HPC and gaming industries.
    - fundamentally still **graphics processors**, not high-performance **scientific calculators**.

A Brief History of GPUs

- Specific **graphics co-processors** existed in the 1980s and 1990s, but not in the general consumer market.
  - fixed-feature hardware accelerators (DirectX) cheaper and faster.
- Recent GPU cards offer significant computational ability, driven largely by the HPC and gaming industries.
  - fundamentally still **graphics processors**, not high-performance **scientific calculators**.

General GPU Structure

- Bunch of different hardware units:
  - memory (VRAM) and host interfaces.
  - a large cache memory area.
  - thread scheduling logic.
  - a number of **stream processors**.
- Logical interpretation is **SIMD**: data is fixed (in a large register-file) and instructions are pumped through a number of processing cores.
- NVIDIA Fermi [8] used in **GF100** and **GF110** GPUs.
  - available on cards such as the Tesla C2050 and GeForce GTX 580.
  - around **3 billion** transistors in **512** CUDA cores.
  - more optimisations for double-precision arithmetic.
- Resulting silicon on a 40nm process is about the size of a stamp.
  - hard to fabricate, but regular structure means that parts can be disabled where defective.
  - e.g. GTX 570 has 1 of the 16 stream processors disabled.

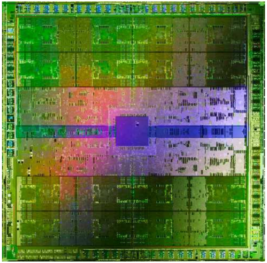
General GPU Structure

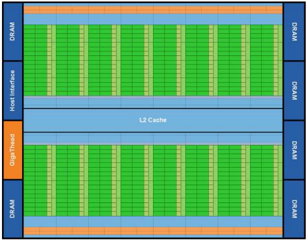
- Bunch of different hardware units:
  - memory (VRAM) and host interfaces.
  - a large cache memory area.
  - thread scheduling logic.
  - a number of **stream processors**.
- Logical interpretation is **SIMD**: data is fixed (in a large register-file) and instructions are pumped through a number of processing cores.
- NVIDIA Fermi [8] used in **GF100** and **GF110** GPUs.
  - available on cards such as the Tesla C2050 and GeForce GTX 580.
  - around **3 billion** transistors in **512** CUDA cores.
  - more optimisations for double-precision arithmetic.
- Resulting silicon on a 40nm process is about the size of a stamp.
  - hard to fabricate, but regular structure means that parts can be disabled where defective.
  - e.g. GTX 570 has 1 of the 16 stream processors disabled.

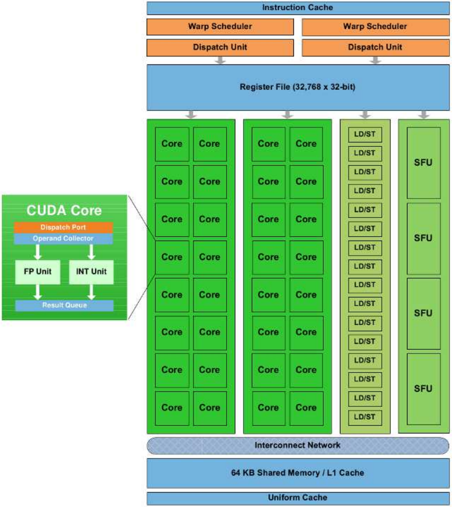
General GPU Structure

- Bunch of different hardware units:
    - memory (VRAM) and host interfaces.
    - a large cache memory area.
    - thread scheduling logic.
    - a number of **stream processors**.
- Logical interpretation is **SIMD**: data is fixed (in a large register-file) and instructions are pumped through a number of processing cores.
- NVIDIA Fermi [8] used in **GF100** and **GF110** GPUs.
    - available on cards such as the Tesla C2050 and GeForce GTX 580.
    - around **3 billion** transistors in **512** CUDA cores.
    - more optimisations for double-precision arithmetic.
- Resulting silicon on a 40nm process is about the size of a stamp.
    - hard to fabricate, but regular structure means that parts can be disabled where defective.
    - e.g. GTX 570 has 1 of the 16 stream processors disabled.

NVIDIA Fermi Architecture







On GPU Programming

- The GPU programming model, for **CUDA** [9] and **OpenCL** [10], is somewhat abstracted from the real hardware.
  - CUDA used for these experiments: more mature and well documented, but less portable.
- Programmer writes a **kernel** — a piece of code that is executed in parallel across the CUDA cores.
  - single **threads** organised into **thread blocks** (max. 512/1024).
  - blocks arranged into **grids** that can be **huge** (64k/2G × 2/3D).
  - threads scheduled in groups of 32 called **warps**, execution is interleaved (based on available resources).
- Arrangement of threads, blocks and grids can be tweaked for performance.
  - balanced with register and cache memory use.
  - “better” GPUs can do **shared memory** and **synchronisation** within thread blocks.

On GPU Programming

- The GPU programming model, for **CUDA** [9] and **OpenCL** [10], is somewhat abstracted from the real hardware.
    - CUDA used for these experiments: more mature and well documented, but less portable.
- Programmer writes a **kernel** — a piece of code that is executed in parallel across the CUDA cores.
    - single **threads** organised into **thread blocks** (max. 512/1024).
    - blocks arranged into **grids** that can be **huge** (64k/2G × 2/3D).
    - threads scheduled in groups of 32 called **warps**, execution is interleaved (based on available resources).
- Arrangement of threads, blocks and grids can be tweaked for performance.
    - balanced with register and cache memory use.
    - "better" GPUs can do **shared memory** and **synchronisation** within thread blocks.

On GPU Programming

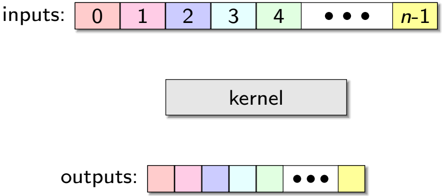
- The GPU programming model, for **CUDA** [9] and **OpenCL** [10], is somewhat abstracted from the real hardware.
  - CUDA used for these experiments: more mature and well documented, but less portable.
- Programmer writes a **kernel** — a piece of code that is executed in parallel across the CUDA cores.
  - single **threads** organised into **thread blocks** (max. 512/1024).
  - blocks arranged into **grids** that can be **huge** (64k/2G × 2/3D).
  - threads scheduled in groups of 32 called **warps**, execution is interleaved (based on available resources).
- Arrangement of threads, blocks and grids can be tweaked for performance.
  - balanced with register and cache memory use.
  - "better" GPUs can do **shared memory** and **synchronisation** within thread blocks.

GPU Programming

- For doing typical scientific calculations (e.g. boid algorithms) over a set of things (e.g. boid state) simplest to treat as a 1D problem:

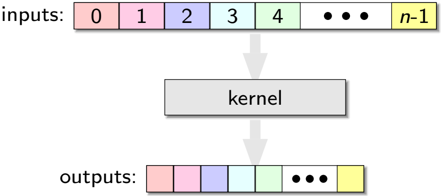
GPU Programming

- For doing typical scientific calculations (e.g. boid algorithms) over a set of things (e.g. boid state) simplest to treat as a 1D problem:



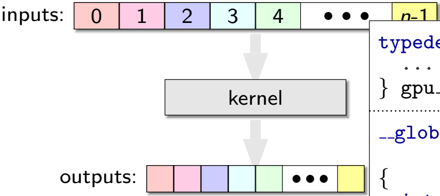
GPU Programming

- For doing typical scientific calculations (e.g. boid algorithms) over a set of things (e.g. boid state) simplest to treat as a 1D problem:



GPU Programming

- For doing typical scientific calculations (e.g. boid algorithms) over a set of things (e.g. boid state) simplest to treat as a 1D problem:



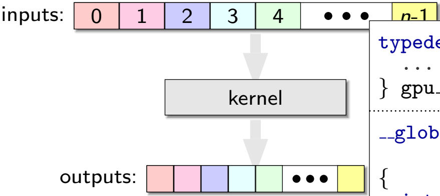
```
typedef struct {              typedef struct {
  ... stuff                     ... stuff
} gpu_in;                     } gpu_out;

__global__ void my_kernel (const gpu_in *in,
            gpu_out *out, const int count)
{
  int idx = (blockDim.x * blockIdx.x)
                 + threadIdx.x;

  if (idx < count) {
    out[idx] = sums (in, idx);
  }
}
```

GPU Programming

- For doing typical scientific calculations (e.g. boid algorithms) over a set of things (e.g. boid state) simplest to treat as a 1D problem:



```
typedef struct {            typedef struct {
  ... stuff                   ... stuff
} gpu_in;                   } gpu_out;
```

```
__global__ void my_kernel (const gpu_in *in,
                gpu_out *out, const int count)
{
  int idx = (blockDim.x * blockIdx.x)
                          + threadIdx.x;

 (idx < count) {
ut[idx] = sums (in, idx);
```

```
... allocate device memory
while (busy) {
  ... copy data to GPU
  my_kernel <<< 512, blks >>> (args)
  ... copy results from GPU
}
... free device memory
```

GPU Server Approach

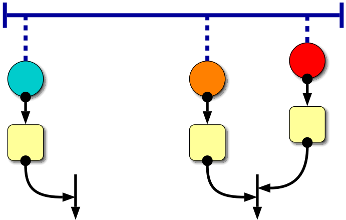
- As a starting point, a **GPU server** process is introduced.
    - clean abstraction: other processes send computation requests and collect results.

- server collects requests and dispatches them in fixed-size **batches** to the GPU.
- only a few parts of the boid algorithm to start with:

- Despite the additional infrastructure, overheads are not too significant.
    - but performance is not too great either.

GPU Server Approach

- As a starting point, a **GPU server** process is introduced.
  - clean abstraction: other processes send computation requests and collect results.



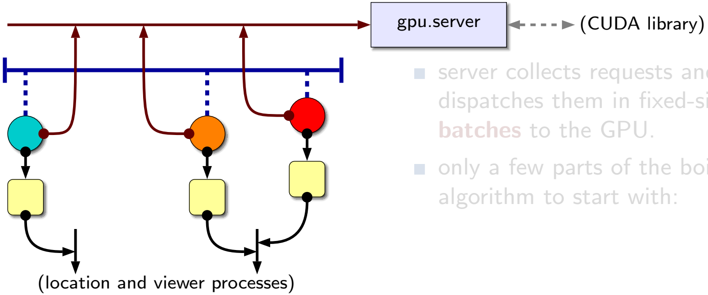
(location and viewer processes)

- server collects requests and dispatches them in fixed-size **batches** to the GPU.
- only a few parts of the boid algorithm to start with:

- Despite the additional infrastructure, overheads are not too significant.
  - but performance is not too great either.

GPU Server Approach

- As a starting point, a **GPU server** process is introduced.
  - clean abstraction: other processes send computation requests and collect results.

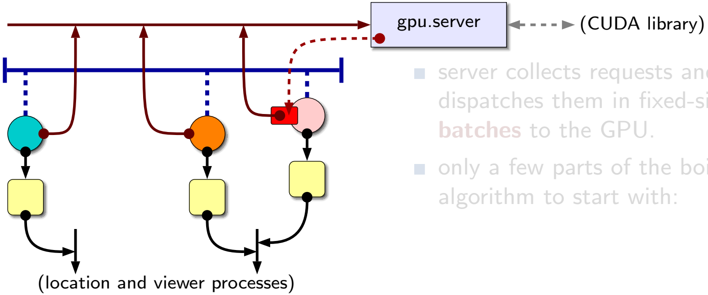


- server collects requests and dispatches them in fixed-size **batches** to the GPU.
- only a few parts of the boid algorithm to start with:

- Despite the additional infrastructure, overheads are not too significant.
  - but performance is not too great either.

GPU Server Approach

- As a starting point, a **GPU server** process is introduced.
    - clean abstraction: other processes send computation requests and collect results.

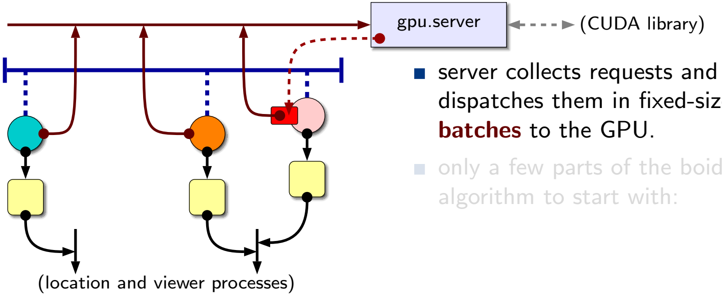


- server collects requests and dispatches them in fixed-size **batches** to the GPU.
- only a few parts of the boid algorithm to start with:

- Despite the additional infrastructure, overheads are not too significant.
    - but performance is not too great either.

GPU Server Approach

- As a starting point, a **GPU server** process is introduced.
  - clean abstraction: other processes send computation requests and collect results.

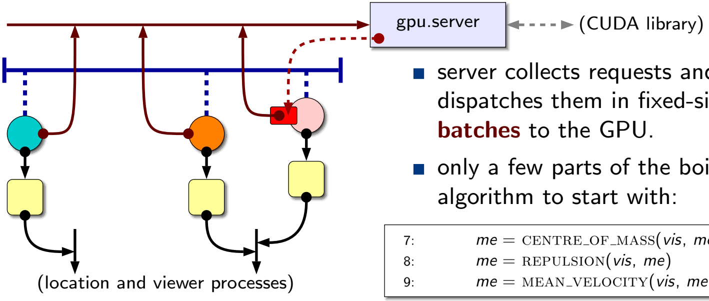


- server collects requests and dispatches them in fixed-size **batches** to the GPU.
- only a few parts of the boid algorithm to start with:

- Despite the additional infrastructure, overheads are not too significant.
  - but performance is not too great either.

GPU Server Approach

- As a starting point, a **GPU server** process is introduced.
  - clean abstraction: other processes send computation requests and collect results.



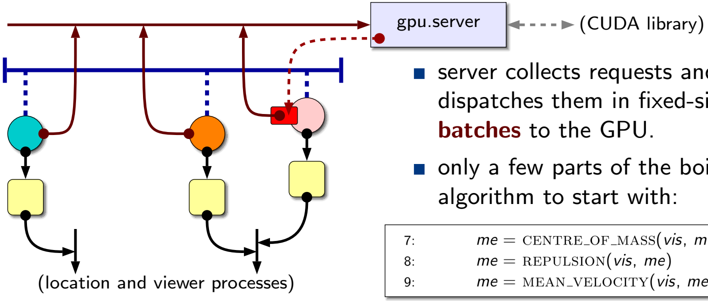
- server collects requests and dispatches them in fixed-size **batches** to the GPU.
- only a few parts of the boid algorithm to start with:

```
$me = \text{CENTRE\_OF\_MASS}(vis, me)$
$me = \text{REPULSION}(vis, me)$
$me = \text{MEAN\_VELOCITY}(vis, me)$
```

- Despite the additional infrastructure, overheads are not too significant.
  - but performance is not too great either.

GPU Server Approach

- As a starting point, a **GPU server** process is introduced.
    - clean abstraction: other processes send computation requests and collect results.

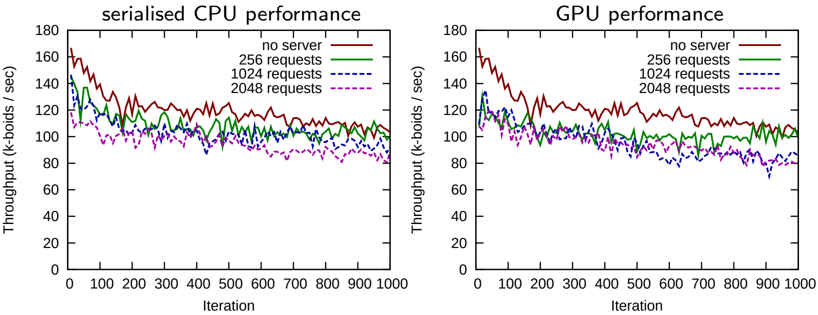


- server collects requests and dispatches them in fixed-size **batches** to the GPU.
- only a few parts of the boid algorithm to start with:

```
$me = \text{CENTRE\_OF\_MASS}(vis, me)$
$me = \text{REPULSION}(vis, me)$
$me = \text{MEAN\_VELOCITY}(vis, me)$
```

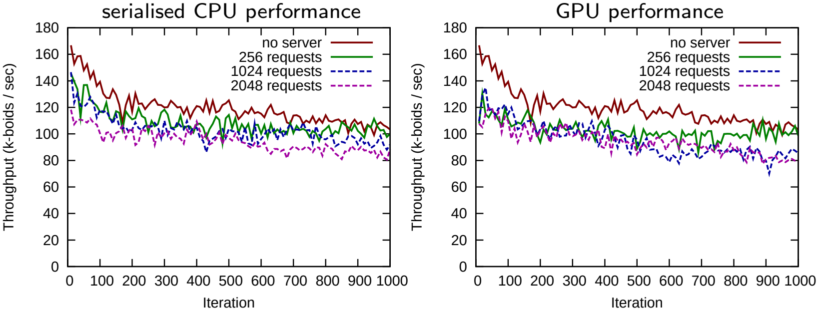
- Despite the additional infrastructure, overheads are not too significant.
    - but performance is not too great either.

GPU Server Approach



- Original choice of which parts of the algorithm to implement on the GPU not brilliant:
  - most computationally expensive part is the splitting of **viewable agents** into **visible boids** and **obstacles**.

GPU Server Approach



- Original choice of which parts of the algorithm to implement on the GPU not brilliant:
  - most computationally expensive part is the splitting of **viewable agents** into **visible boids** and **obstacles**.

GPU Server Approach: More GPU

- Putting more of the boid algorithm onto the GPU, does not help:

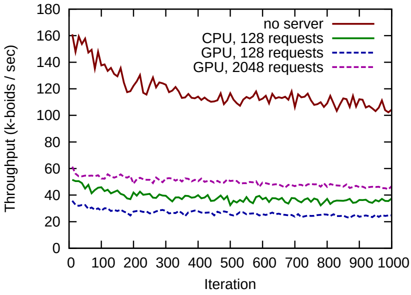
```
        $vis, obs = \text{PRUNE\_VISIBLE}(all,\ me)$
        $me = \text{CENTRE\_OF\_MASS}(vis,\ me)$
        $me = \text{REPULSION}(vis,\ me)$
        $me = \text{MEAN\_VELOCITY}(vis,\ me)$
        $me = \text{OBSTACLES}(obs,\ me)$
```

- Significant increase in the amount of data (*all*) copied to the GPU.
- for typical parameter sets, the number of **visible** agents (*vis*) is around 3–5% of those **viewable** (*all*) — circa 13MB for 2048 boids.

GPU Server Approach: More GPU

- Putting more of the boid algorithm onto the GPU, does not help:

```
        $vis, obs = \text{PRUNE\_VISIBLE}(all, me)$
        $me = \text{CENTRE\_OF\_MASS}(vis, me)$
        $me = \text{REPULSION}(vis, me)$
        $me = \text{MEAN\_VELOCITY}(vis, me)$
        $me = \text{OBSTACLES}(obs, me)$
```

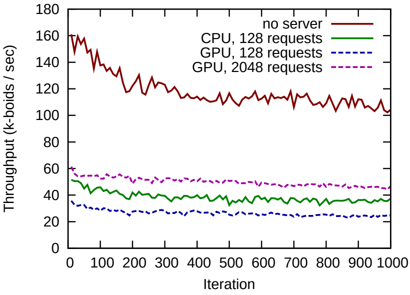


- Significant increase in the amount of data (*all*) copied to the GPU.
- for typical parameter sets, the number of **visible** agents (*vis*) is around 3–5% of those **viewable** (*all*) — circa 13MB for 2048 boids.

GPU Server Approach: More GPU

- Putting more of the boid algorithm onto the GPU, does not help:

```
$vis, obs = \text{PRUNE\_VISIBLE}(all,\ me)$
$me = \text{CENTRE\_OF\_MASS}(vis,\ me)$
$me = \text{REPULSION}(vis,\ me)$
$me = \text{MEAN\_VELOCITY}(vis,\ me)$
$me = \text{OBSTACLES}(obs,\ me)$
```

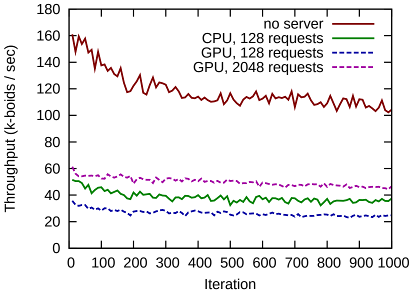


- Significant increase in the amount of data (*all*) copied to the GPU.
- for typical parameter sets, the number of **visible** agents (*vis*) is around 3–5% of those **viewable** (*all*) — circa 13MB for 2048 boids.

GPU Server Approach: More GPU

- Putting more of the boid algorithm onto the GPU, does not help:

```
$vis, obs = \text{PRUNE\_VISIBLE}(all, me)$
$me = \text{CENTRE\_OF\_MASS}(vis, me)$
$me = \text{REPULSION}(vis, me)$
$me = \text{MEAN\_VELOCITY}(vis, me)$
$me = \text{OBSTACLES}(obs, me)$
```



- Significant increase in the amount of data (*all*) copied to the GPU.
- for typical parameter sets, the number of **visible** agents (*vis*) is around 3–5% of those **viewable** (*all*) — circa 13MB for 2048 boids.

GPU Server Approach: More Optimisations

- Various attempts to further **optimise** the system (without changing anything too substantially) did *not* produce anything better than the CPU-only version.
  - limited by the memory bandwidth between host and GPU — might improve with host-stolen video-RAM.
  - strategies included **page locked** memory on the host (directly sharable over the PCIe bus) and the use of **streams** on the device to overlap memory copies with kernel execution.

Refactoring: Shared Data

- As a moderate change, introduce some **shared data** to the system.
    - in principle, means the actual boid (and other agent) state only needs to be copied to the GPU **once** each cycle.
    - **barrier phases** can be used to coordinate access to this shared state **safely** (CREW).
- Requires some less subtle changes in the system:
    - mostly **absolute** positioning and agent IDs not state.

- **Phase 1**: boids read global state and compute new (local) velocity.
- **Phase 2**: boids update global state and move.
- **Phase 3**: updates to viewable states occur (as before).

Refactoring: Shared Data

- As a moderate change, introduce some **shared data** to the system.
  - in principle, means the actual boid (and other agent) state only needs to be copied to the GPU **once** each cycle.
  - **barrier phases** can be used to coordinate access to this shared state **safely** (CREW).
- Requires some less subtle changes in the system:
  - mostly **absolute** positioning and agent IDs not state.

- **Phase 1**: boids read global state and compute new (local) velocity.
- **Phase 2**: boids update global state and move.
- **Phase 3**: updates to viewable states occur (as before).

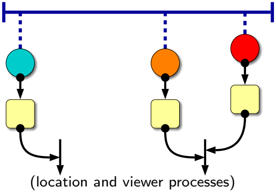
Refactoring: Shared Data

- As a moderate change, introduce some **shared data** to the system.
  - in principle, means the actual boid (and other agent) state only needs to be copied to the GPU **once** each cycle.
  - **barrier phases** can be used to coordinate access to this shared state **safely** (CREW).
- Requires some less subtle changes in the system:
  - mostly **absolute** positioning and agent IDs not state.

- **Phase 1**: boids read global state and compute new (local) velocity.
- **Phase 2**: boids update global state and move.
- **Phase 3**: updates to viewable states occur (as before).

Refactoring: Shared Data

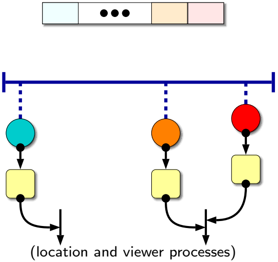
- As a moderate change, introduce some **shared data** to the system.
  - in principle, means the actual boid (and other agent) state only needs to be copied to the GPU **once** each cycle.
  - **barrier phases** can be used to coordinate access to this shared state **safely** (CREW).
- Requires some less subtle changes in the system:
  - mostly **absolute** positioning and agent IDs not state.



- **Phase 1**: boids read global state and compute new (local) velocity.
- **Phase 2**: boids update global state and move.
- **Phase 3**: updates to viewable states occur (as before).

Refactoring: Shared Data

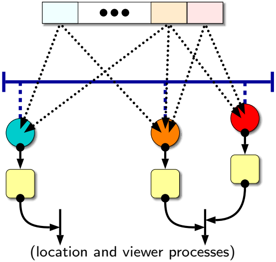
- As a moderate change, introduce some **shared data** to the system.
  - in principle, means the actual boid (and other agent) state only needs to be copied to the GPU **once** each cycle.
  - **barrier phases** can be used to coordinate access to this shared state **safely** (CREW).
- Requires some less subtle changes in the system:
  - mostly **absolute** positioning and agent IDs not state.



- **Phase 1**: boids read global state and compute new (local) velocity.
- **Phase 2**: boids update global state and move.
- **Phase 3**: updates to viewable states occur (as before).

Refactoring: Shared Data

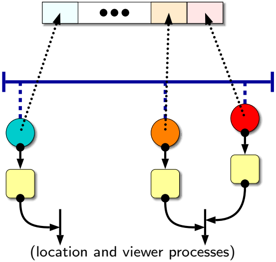
- As a moderate change, introduce some **shared data** to the system.
    - in principle, means the actual boid (and other agent) state only needs to be copied to the GPU **once** each cycle.
    - **barrier phases** can be used to coordinate access to this shared state **safely** (CREW).
- Requires some less subtle changes in the system:
    - mostly **absolute** positioning and agent IDs not state.



- **Phase 1**: boids read global state and compute new (local) velocity.
- **Phase 2**: boids update global state and move.
- **Phase 3**: updates to viewable states occur (as before).

Refactoring: Shared Data

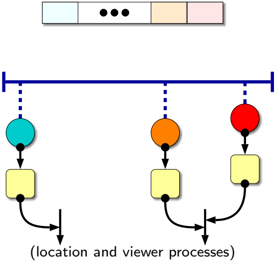
- As a moderate change, introduce some **shared data** to the system.
    - in principle, means the actual boid (and other agent) state only needs to be copied to the GPU **once** each cycle.
    - **barrier phases** can be used to coordinate access to this shared state **safely** (CREW).
- Requires some less subtle changes in the system:
    - mostly **absolute** positioning and agent IDs not state.



- **Phase 1**: boids read global state and compute new (local) velocity.
- **Phase 2**: boids update global state and move.
- **Phase 3**: updates to viewable states occur (as before).

Refactoring: Shared Data

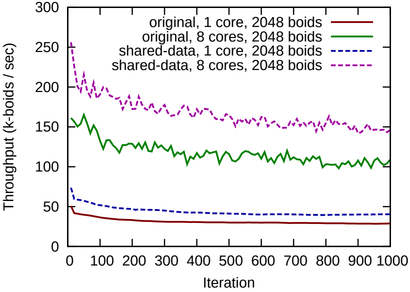
- As a moderate change, introduce some **shared data** to the system.
  - in principle, means the actual boid (and other agent) state only needs to be copied to the GPU **once** each cycle.
  - **barrier phases** can be used to coordinate access to this shared state **safely** (CREW).
- Requires some less subtle changes in the system:
  - mostly **absolute** positioning and agent IDs not state.

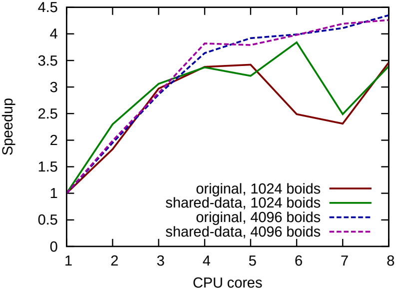


- **Phase 1**: boids read global state and compute new (local) velocity.
- **Phase 2**: boids update global state and move.
- **Phase 3**: updates to viewable states occur (as before).

Shared Data: Performance

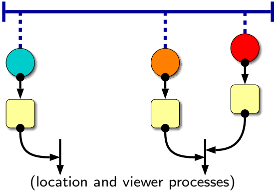
- Considering a CPU-only version to start with (based on the original), performance is **significantly improved**.
  - downside is our existing GPU results now look even worse...





Shared Data: Reintroducing the GPU

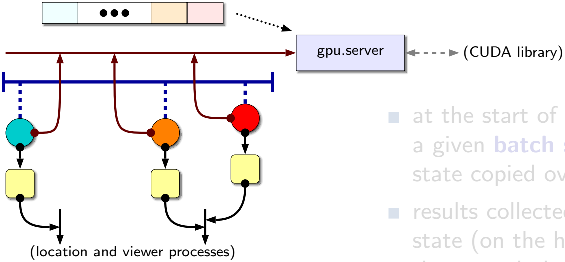
- Next, add a GPU-server process, operating on shared agent data.
  - still copying around arrays of **viewable** agents, but only integers now.



- at the start of the GPU cycle (for a given **batch size**), all agent state copied over.
- results collected locally and global state (on the host) updated before the second phase.

Shared Data: Reintroducing the GPU

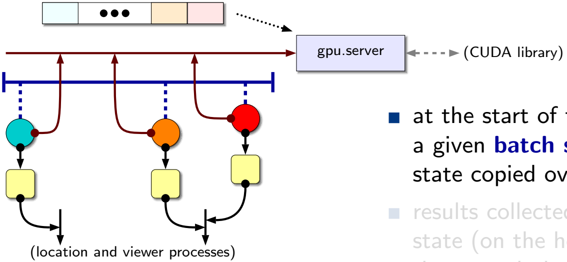
- Next, add a GPU-server process, operating on shared agent data.
  - still copying around arrays of **viewable** agents, but only integers now.



- at the start of the GPU cycle (for a given **batch size**), all agent state copied over.
- results collected locally and global state (on the host) updated before the second phase.

Shared Data: Reintroducing the GPU

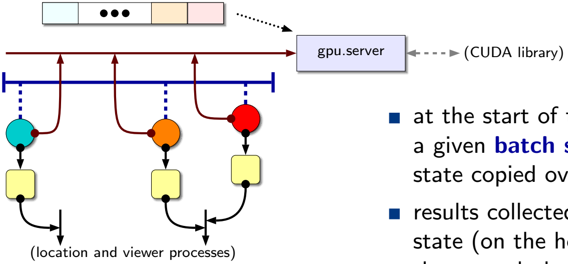
- Next, add a GPU-server process, operating on shared agent data.
  - still copying around arrays of **viewable** agents, but only integers now.



- at the start of the GPU cycle (for a given **batch size**), all agent state copied over.
- results collected locally and global state (on the host) updated before the second phase.

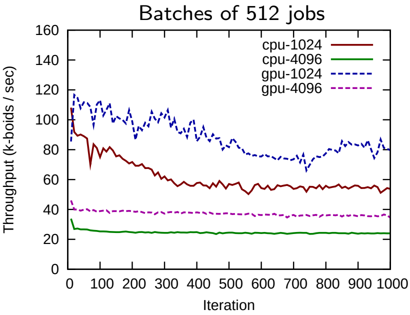
Shared Data: Reintroducing the GPU

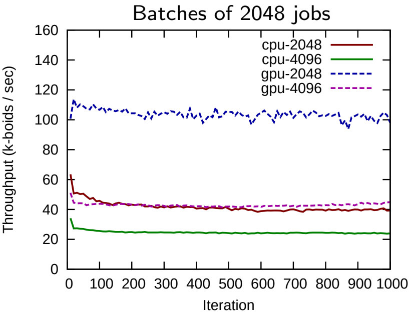
- Next, add a GPU-server process, operating on shared agent data.
  - still copying around arrays of **viewable** agents, but only integers now.



- at the start of the GPU cycle (for a given **batch size**), all agent state copied over.
- results collected locally and global state (on the host) updated before the second phase.

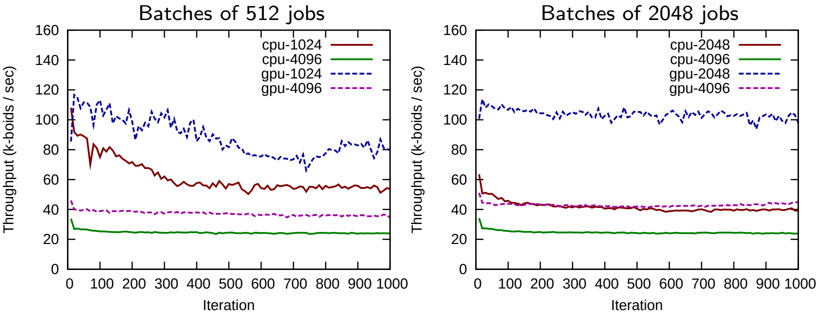
Reintroducing the GPU: Performance


Batches of 512 jobs


Batches of 2048 jobs

- Performance is unimpressive.
  - worse than the **shared-data CPU-only** version in all cases.
- Still a lot of **viewable** state manipulation.

Reintroducing the GPU: Performance



- Performance is unimpressive.
  - worse than the **shared-data CPU-only** version in all cases.
- Still a lot of **viewable** state manipulation.

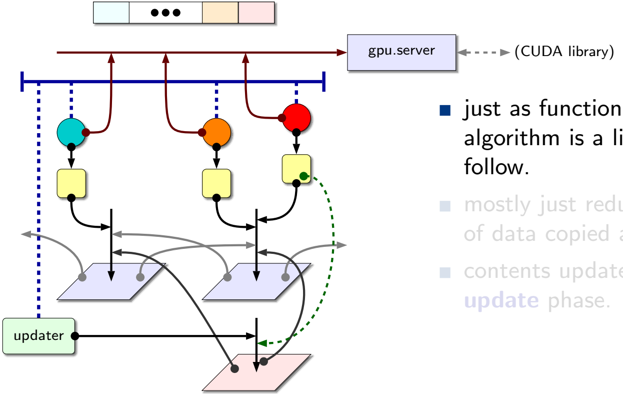
Sharing the Viewable State

- Sharing the viewable state (in each **viewer**) requires some changes in the boid algorithm.
    - a **single pass** over the viewable agents, instead of sorting into **visible boids** and **obstacles**.

- just as functional, but the boid algorithm is a little harder to follow.
- mostly just reducing the amount of data copied around.
- contents updated during the **update** phase.

Sharing the Viewable State

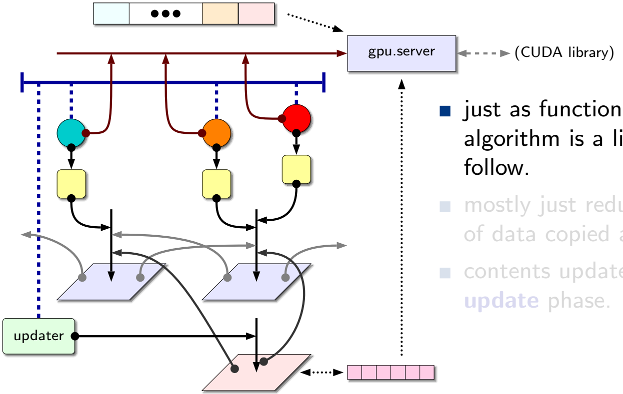
- Sharing the viewable state (in each **viewer**) requires some changes in the boid algorithm.
  - a **single pass** over the viewable agents, instead of sorting into **visible boids** and **obstacles**.



- just as functional, but the boid algorithm is a little harder to follow.
- mostly just reducing the amount of data copied around.
- contents updated during the **update** phase.

Sharing the Viewable State

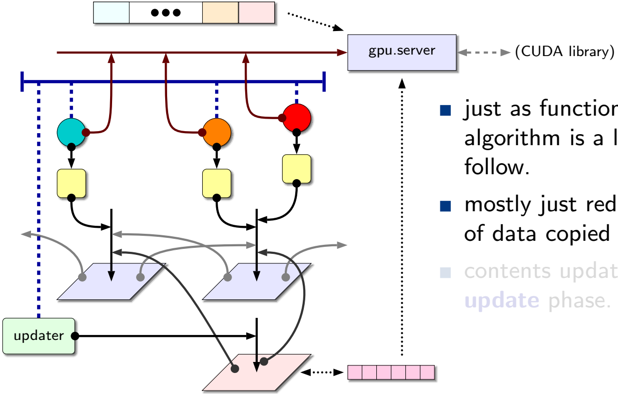
- Sharing the viewable state (in each **viewer**) requires some changes in the boid algorithm.
  - a **single pass** over the viewable agents, instead of sorting into **visible boids** and **obstacles**.



- just as functional, but the boid algorithm is a little harder to follow.
- mostly just reducing the amount of data copied around.
- contents updated during the **update** phase.

Sharing the Viewable State

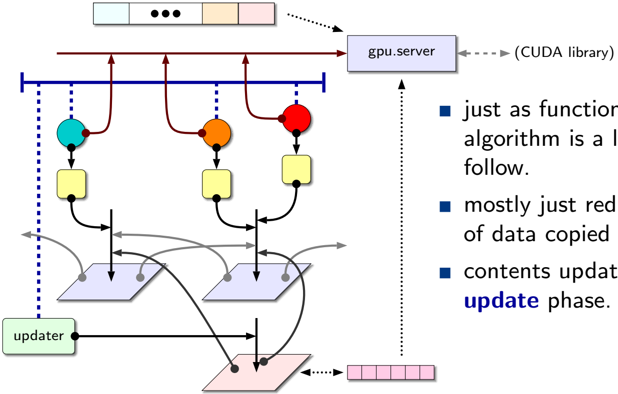
- Sharing the viewable state (in each **viewer**) requires some changes in the boid algorithm.
  - a **single pass** over the viewable agents, instead of sorting into **visible boids** and **obstacles**.



- just as functional, but the boid algorithm is a little harder to follow.
- mostly just reducing the amount of data copied around.
- contents updated during the **update** phase.

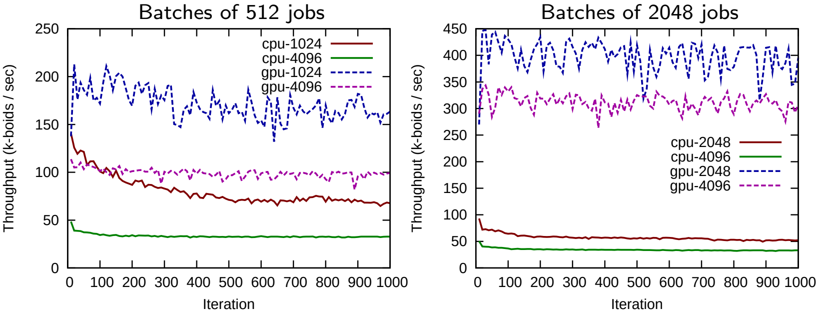
Sharing the Viewable State

- Sharing the viewable state (in each **viewer**) requires some changes in the boid algorithm.
  - a **single pass** over the viewable agents, instead of sorting into **visible boids** and **obstacles**.



- just as functional, but the boid algorithm is a little harder to follow.
- mostly just reducing the amount of data copied around.
- contents updated during the **update** phase.

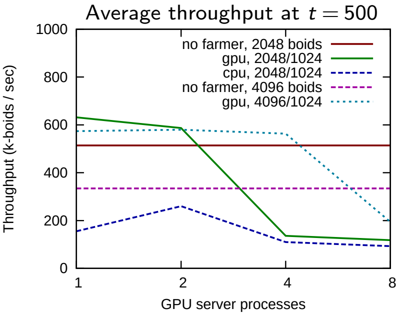
Sharing the Viewable State: Performance



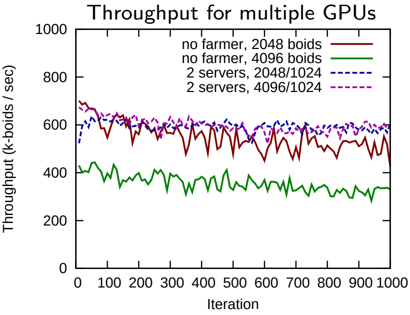
- For batches of 2048 jobs, start seeing some **performance gain** for the first time!
  - slow-down for 4096 boids is partially due to increased density (still in an 8×6 grid).

Parallel GPU Servers

- An obvious (and fairly straightforward) next step is to **parallelise** the GPU server.
    - to take advantage of **multiple GPUs**.
    - or allow a mix of GPU and CPU execution.



Average throughput at $t = 500$

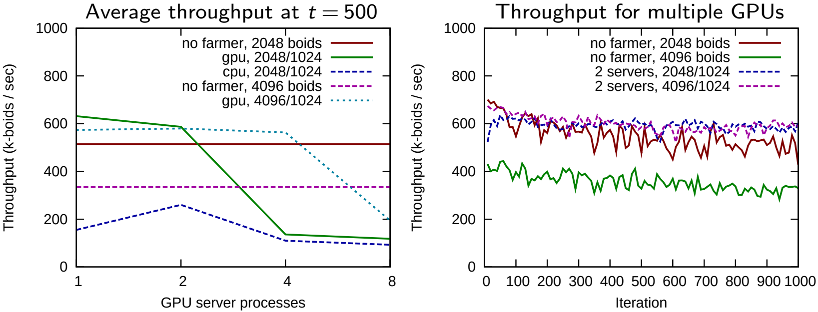


Throughput for multiple GPUs

- Improvement in throughput for 4096 boids is significant.
    - so worth doing when multiple GPUs are present.

Parallel GPU Servers

- An obvious (and fairly straightforward) next step is to **parallelise** the GPU server.
  - to take advantage of **multiple GPUs**.
  - or allow a mix of GPU and CPU execution.



- Improvement in throughput for 4096 boids is significant.
  - so worth doing when multiple GPUs are present.

Further Optimisation: Less Channel I/O

- Each cycle, the viewer processes update their **viewable** arrays from the contents of the 9 connected locations.
    - means agent IDs are **duplicated** 8 times (although that's not a huge overhead).
- Each boid goes through a sequence of communications with the GPU server process.
    - when dealing with large numbers of boids, this creates **significant** overheads (for something that is largely straightforward).
- Solutions to these damage the **clarity** of the system.
    - largely by **breaking** the abstractions of delegated computation (the GPU server process) and viewable state (in the viewer processes).

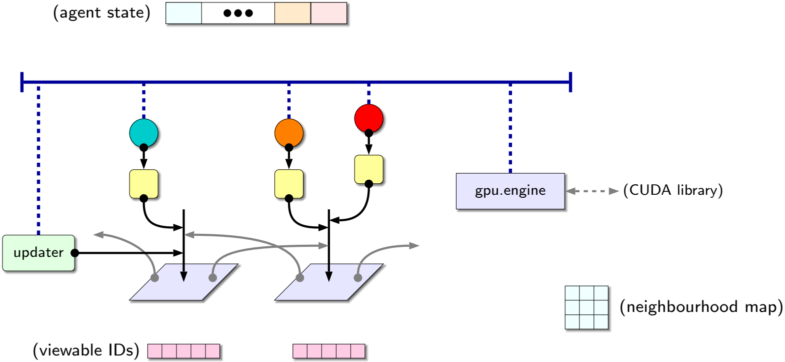
Further Optimisation: Less Channel I/O

- Each cycle, the viewer processes update their **viewable** arrays from the contents of the 9 connected locations.
  - means agent IDs are **duplicated** 8 times (although that's not a huge overhead).
- Each boid goes through a sequence of communications with the GPU server process.
  - when dealing with large numbers of boids, this creates **significant** overheads (for something that is largely straightforward).
- Solutions to these damage the **clarity** of the system.
  - largely by **breaking** the abstractions of delegated computation (the GPU server process) and viewable state (in the viewer processes).

Further Optimisation: Less Channel I/O

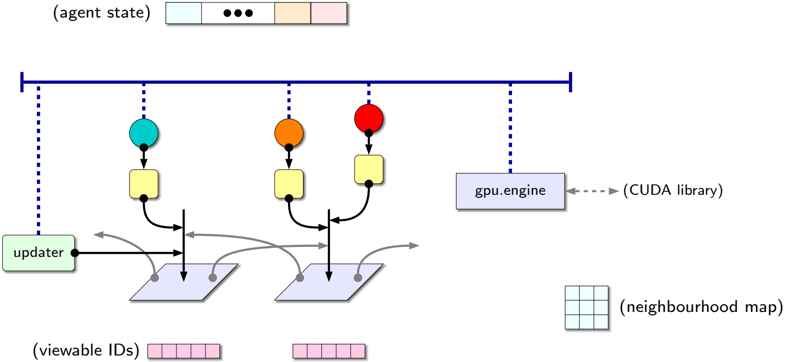
- Each cycle, the viewer processes update their **viewable** arrays from the contents of the 9 connected locations.
  - means agent IDs are **duplicated** 8 times (although that's not a huge overhead).
- Each boid goes through a sequence of communications with the GPU server process.
  - when dealing with large numbers of boids, this creates **significant** overheads (for something that is largely straightforward).
- Solutions to these damage the **clarity** of the system.
  - largely by **breaking** the abstractions of delegated computation (the GPU server process) and viewable state (in the viewer processes).

Further Optimisation: Less Channel I/O



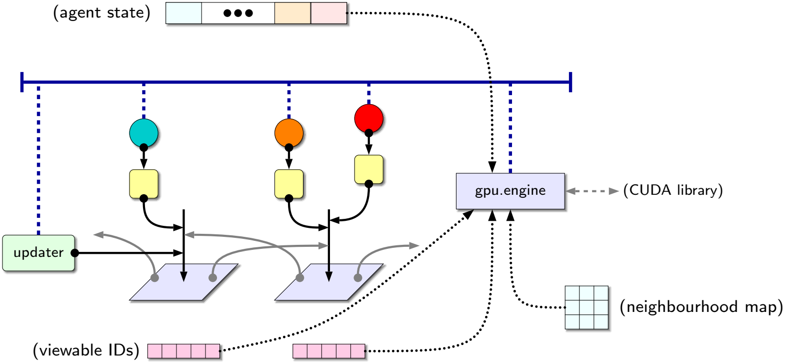
- **Three phases** of execution:
  - state copied to GPU, computations performed, results collected and written back.
  - boids initiate movement, moving if needed.
  - global viewable state updated.

Further Optimisation: Less Channel I/O



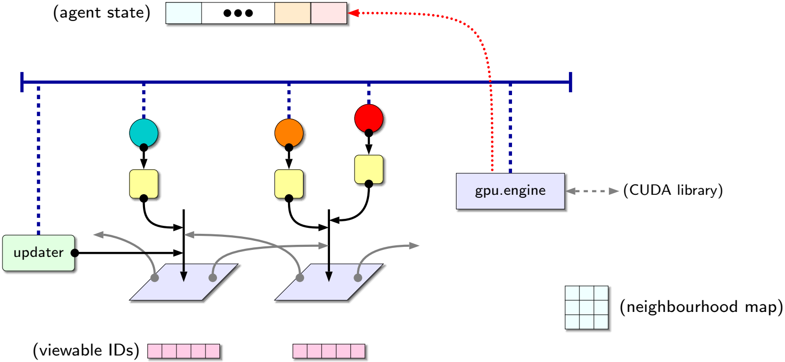
- **Three phases** of execution:
    - state copied to GPU, computations performed, results collected and written back.
    - boids initiate movement, moving if needed.
    - global viewable state updated.

Further Optimisation: Less Channel I/O



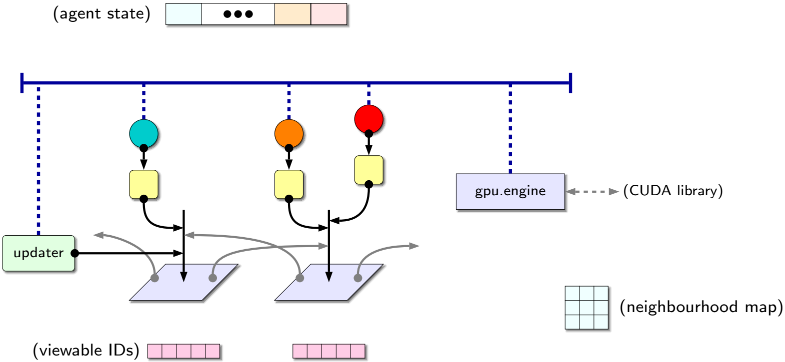
- **Three phases** of execution:
  - state copied to GPU, computations performed, results collected and written back.
  - boids initiate movement, moving if needed.
  - global viewable state updated.

Further Optimisation: Less Channel I/O



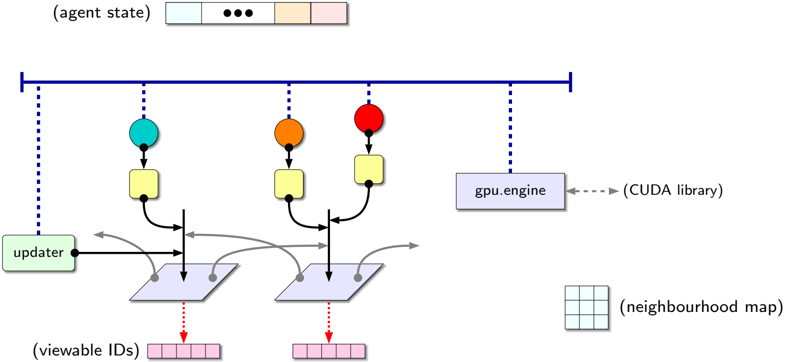
- **Three phases** of execution:
  - state copied to GPU, computations performed, results collected and written back.
  - boids initiate movement, moving if needed.
  - global viewable state updated.

Further Optimisation: Less Channel I/O



- **Three phases** of execution:
  - state copied to GPU, computations performed, results collected and written back.
  - boids initiate movement, moving if needed.
  - global viewable state updated.

Further Optimisation: Less Channel I/O

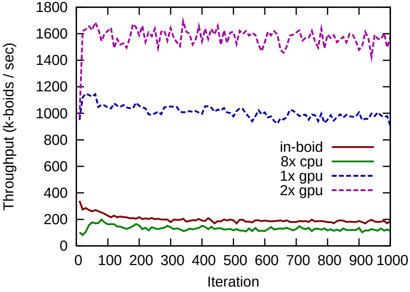


- **Three phases** of execution:
  - state copied to GPU, computations performed, results collected and written back.
  - boids initiate movement, moving if needed.
  - global viewable state updated.

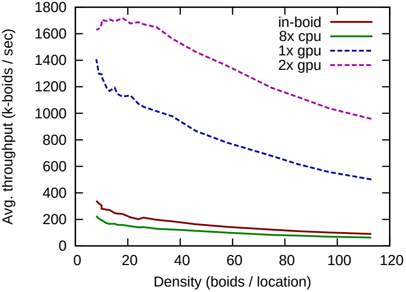
Less Channel I/O: Performance

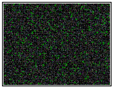
- Improvement in performance is **substantial**.
  - for **16384** boids, vary the **density** and execution mode.

Average throughput at 38 boids/location

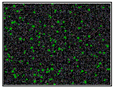


Throughput for varying densities

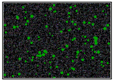




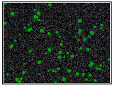
$\frac{16384}{50\times 38} = 9$



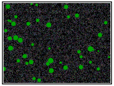
$\frac{16384}{26\times 20} = 32$



$\frac{16384}{20\times 14} = 59$



$\frac{16384}{16\times 12} = 85$



$\frac{16384}{12\times 9} = 152$

Centralising Movement

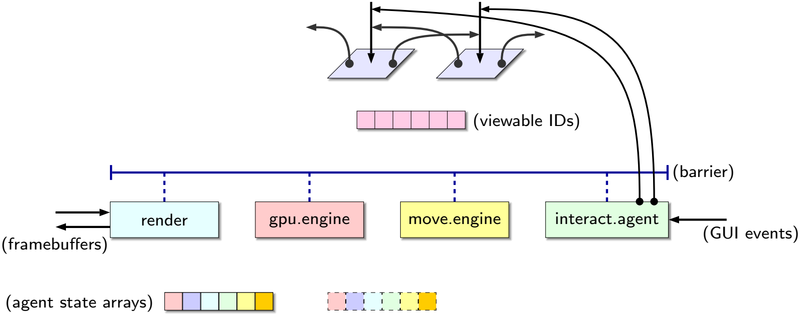
- Boid processes still (and only) initiate the **move** between locations.
    - centralising this makes the boid processes, locations and updaters **redundant**.
- We keep the locations for interaction, however.
    - and **double-buffer** the agent state for performance.

Centralising Movement

- Boid processes still (and only) initiate the **move** between locations.
    - centralising this makes the boid processes, locations and updaters **redundant**.
- We keep the locations for interaction, however.
    - and **double-buffer** the agent state for performance.

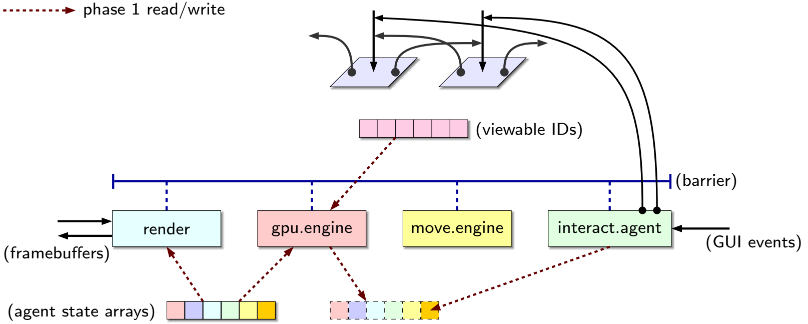
Centralising Movement

- Boid processes still (and only) initiate the **move** between locations.
  - centralising this makes the boid processes, locations and updaters **redundant**.
- We keep the locations for interaction, however.
  - and **double-buffer** the agent state for performance.



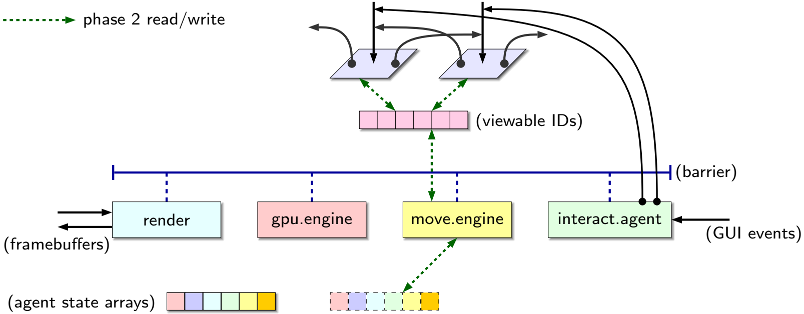
Centralising Movement

- Boid processes still (and only) initiate the **move** between locations.
  - centralising this makes the boid processes, locations and updaters **redundant**.
- We keep the locations for interaction, however.
  - and **double-buffer** the agent state for performance.



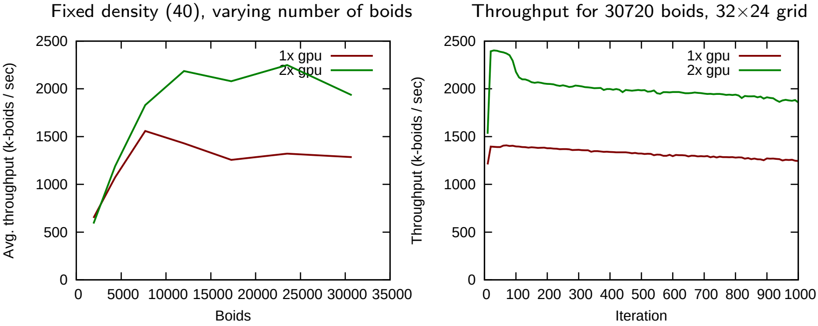
Centralising Movement

- Boid processes still (and only) initiate the **move** between locations.
    - centralising this makes the boid processes, locations and updaters **redundant**.
- We keep the locations for interaction, however.
    - and **double-buffer** the agent state for performance.

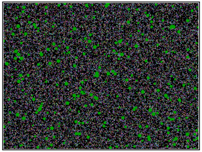


Centralising Movement: Performance

- Squeeze a little more performance out of the GPU(s).



- Could manage more in theory, but visualisation creates overheads.
  - from about 60 cycles/sec without visualisation, to 25 cycles/sec with (synchronised display).



Experimenting with Different Parameters

- Boid algorithm uses a number of different **parameters** internally.
  - **repulsion** radius and fraction, **viewing** angle and distance, **centre-of-mass** fraction, **mean-velocity** fraction.
  - and a few other things.
- Playing around produces substantially different behaviours.
  - previously difficult to explore with large numbers of agents.

Experimenting with Different Parameters

- Boid algorithm uses a number of different **parameters** internally.
  - **repulsion** radius and fraction, **viewing** angle and distance, **centre-of-mass** fraction, **mean-velocity** fraction.
  - and a few other things.
- Playing around produces substantially different behaviours.
  - previously difficult to explore with large numbers of agents.

Conclusions

- Have gone from a basic **occam-π only** implementation (managing around 110,000 boid-cycles per second) to a **hybrid CPU/GPU** implementation with good performance (2,000,000 boid-cycles per second).
  - could still **improve** though (future work).
- A process of **step-by-step change**, not a new implementation.
  - unlikely to have come up with this design from a fresh start.
- Despite the differences from the original, still retains nice high-level features:
  - can have other agents (e.g. the interactive one) in the system too — executing on the CPU, GPU or something else.
  - **distribution** still possible: use of locations (even if just data).

Conclusions

- Have gone from a basic **occam-π only** implementation (managing around 110,000 boid-cycles per second) to a **hybrid CPU/GPU** implementation with good performance (2,000,000 boid-cycles per second).
    - could still **improve** though (future work).
- A process of **step-by-step change**, not a new implementation.
    - unlikely to have come up with this design from a fresh start.
- Despite the differences from the original, still retains nice high-level features:
    - can have other agents (e.g. the interactive one) in the system too — executing on the CPU, GPU or something else.
    - **distribution** still possible: use of locations (even if just data).

Conclusions

- Have gone from a basic **occam-π only** implementation (managing around 110,000 boid-cycles per second) to a **hybrid CPU/GPU** implementation with good performance (2,000,000 boid-cycles per second).
    - could still **improve** though (future work).
- A process of **step-by-step change**, not a new implementation.
    - unlikely to have come up with this design from a fresh start.
- Despite the differences from the original, still retains nice high-level features:
    - can have other agents (e.g. the interactive one) in the system too — executing on the CPU, GPU or something else.
    - **distribution** still possible: use of locations (even if just data).

Future Work

- Now that we can have large numbers of boids, a **3D version**.
  - and perhaps an opportunity to do something interesting with the **haptics interface**.
- **Absolutely no attempt** (because of lack of time) has been made to optimise the code that runs on the GPU, other than getting it to work.
  - expect to squeeze a bit of performance out.
  - have not even experimented with different threads-per-block and similar.
- A **total GPU implementation**, to give a "best case" benchmark.
  - if not already; handling the moves on the GPU is **non-trivial**.

Acknowledgements

- Hardware:
  - NVIDIA GTX-570, GTX-590 and ATI Radeon 7970 funded by the Faculty of Sciences (REF fund 2012/2013, Tranche 1).
  - fast desktop (quad-core 2600K) funded by the School of Computing.
- Early experiments with occam-π and CUDA/OpenCL done by Tom Pressnell and Brendan Le Foll (graduated).
- Images, in no particular order:
  - Intel Corporation, Kaiiv (de.wikipedia), Editing by Pixel8, IBM Corporation, pcmag.com, IXBT Labs, anandtech.com, NVIDIA Corporation.
- Additional history/etc.: Wikipedia.

Questions?

References

[1] P.H. Welch and F.R.M. Barnes.
Communicating mobile processes: introducing occam-pi.
In *25 Years of CSP*, volume 3525 of *LNCS*. Springer, 2005.

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

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

[4] 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.

[5] Fiona A.C. Polack, Tim Hoverd, Adam T. Sampson, Susan Stepney, and Jon Timmis.
Complex systems models: engineering simulations.
In S. Bullock, J. Noble, R. Watson, and M. A. Bedau, editors, *Artificial Life XI: Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems*, pages 482–489. MIT Press, Cambridge, MA, 2008.

[6] 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.

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

[8] NVIDIA Corporation.
Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Fermi, 2009.

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

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