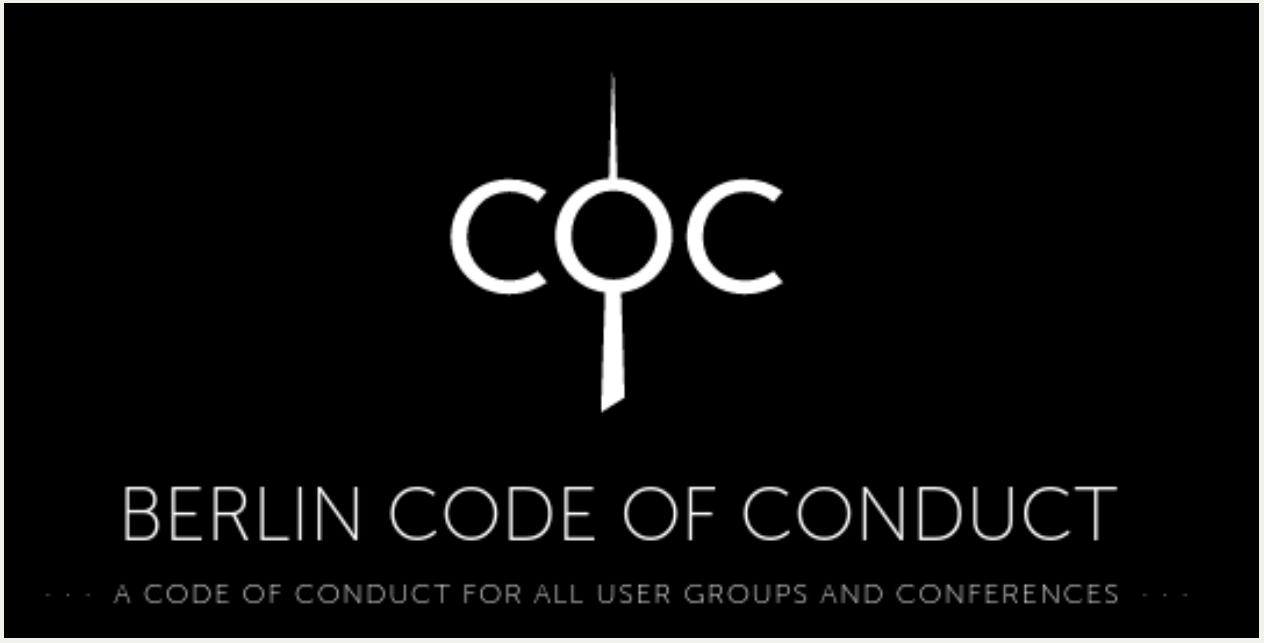
Graphics Programming Virtual Meetup


CoC
BERLIN CODE OF CONDUCT
A CODE OF CONDUCT FOR ALL USER GROUPS AND CONFERENCES

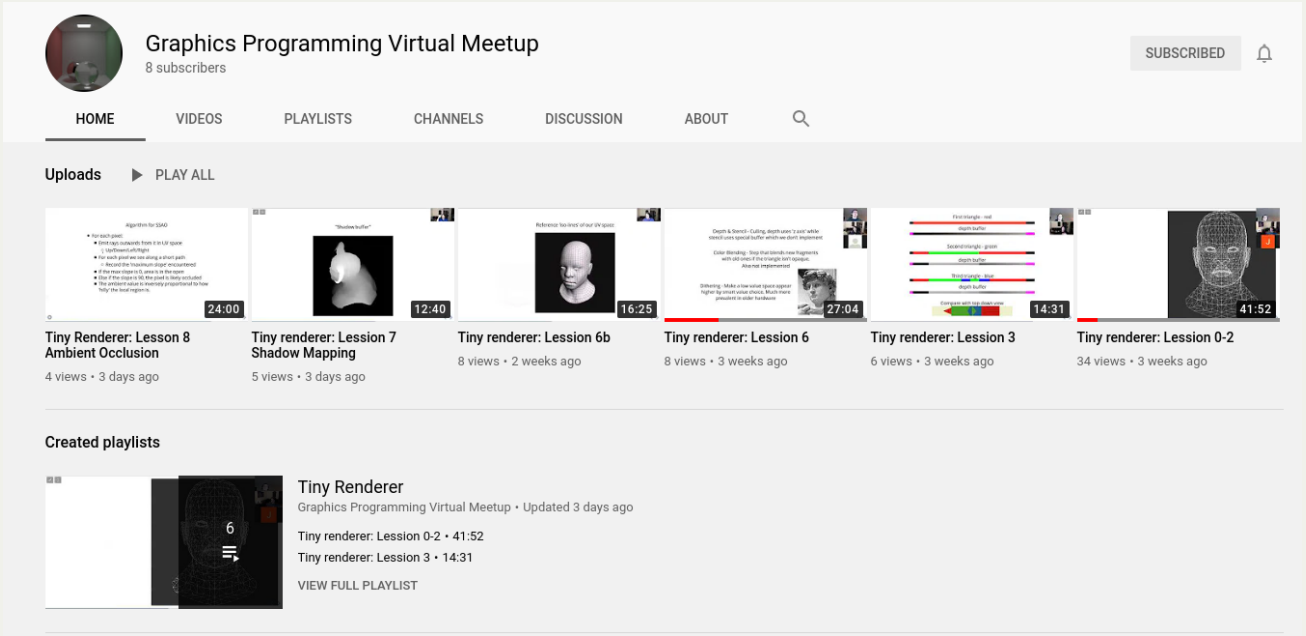
Discord

https://discord.gg/6TTRA5h

Twitter

https://twitter.com/GraphicMeetup

Youtube Channel



https://www.youtube.com/channel/UCbX05PBAE-582PYaRXdjRnw/

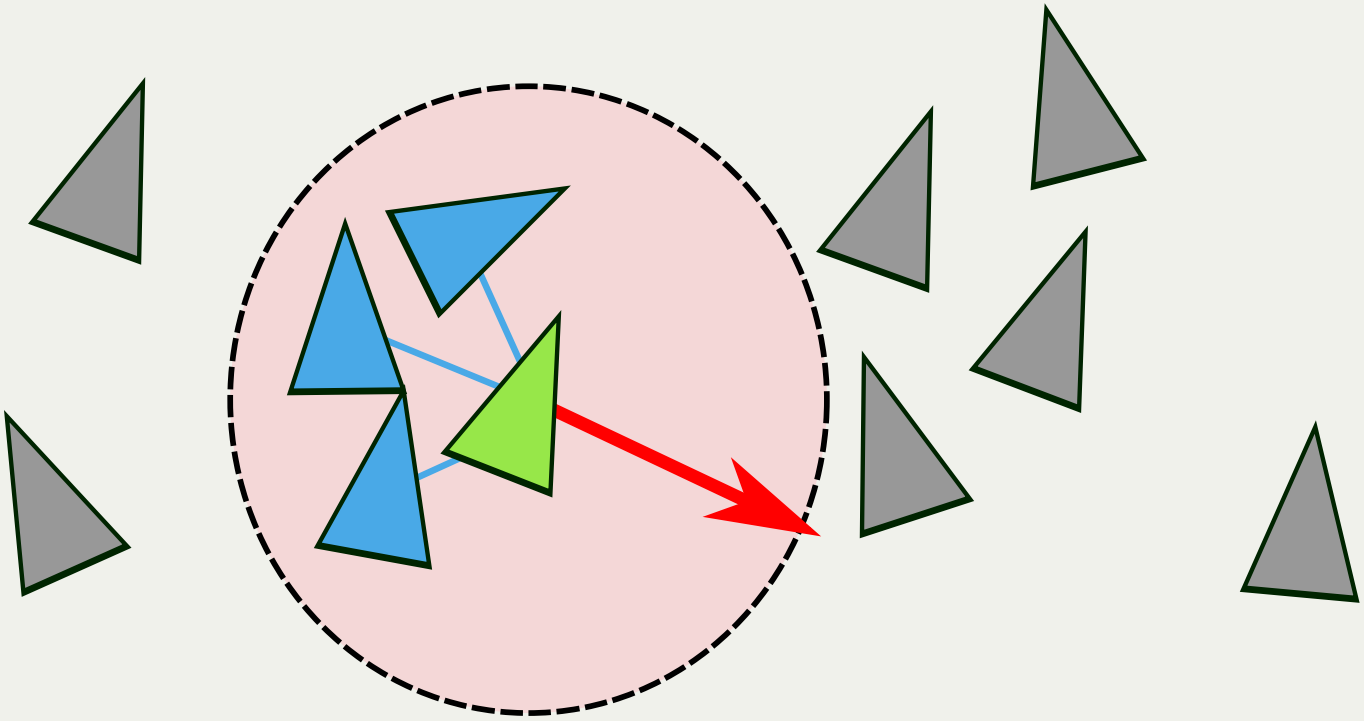
CUDA Flocking Simulation

Credits

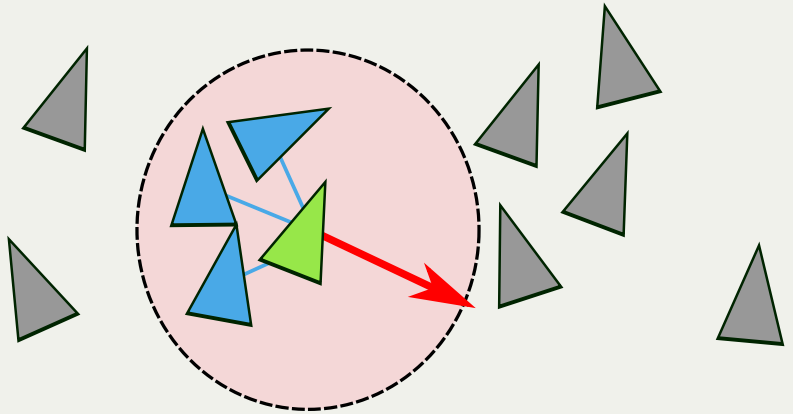
- University of Pennsylvania CIS 565 2020 Project 1
- "Efficient Neighbor Searching for Agent-based Simulation on GPU"

Boid Simulation

Rule 1: Separation

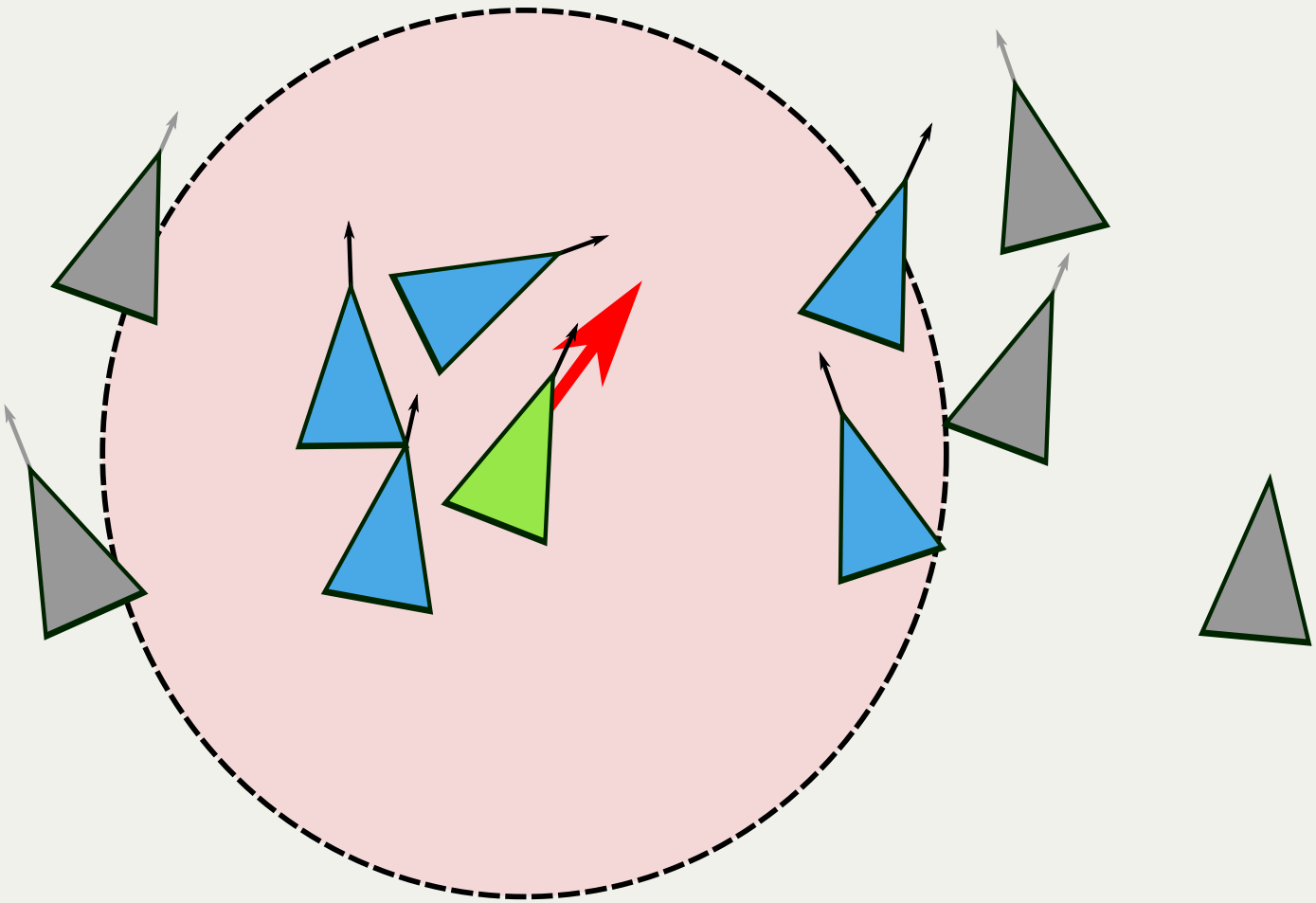


Separation Pseudocode

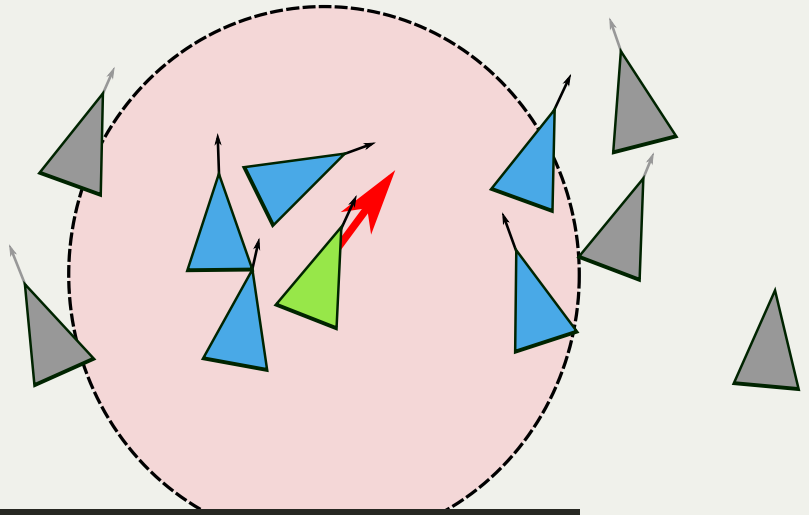


```
def seperation(boid: Boid, boids: Boid[]) {
  c: Vec3 = 0

  for (b : boids) {
    if b != boid and distance(b, boid) < seperation_distance
      c -= (neighbor.position - boid.position)
  }

  return c * seperation_scale
}
```

Rule 2: Alignment

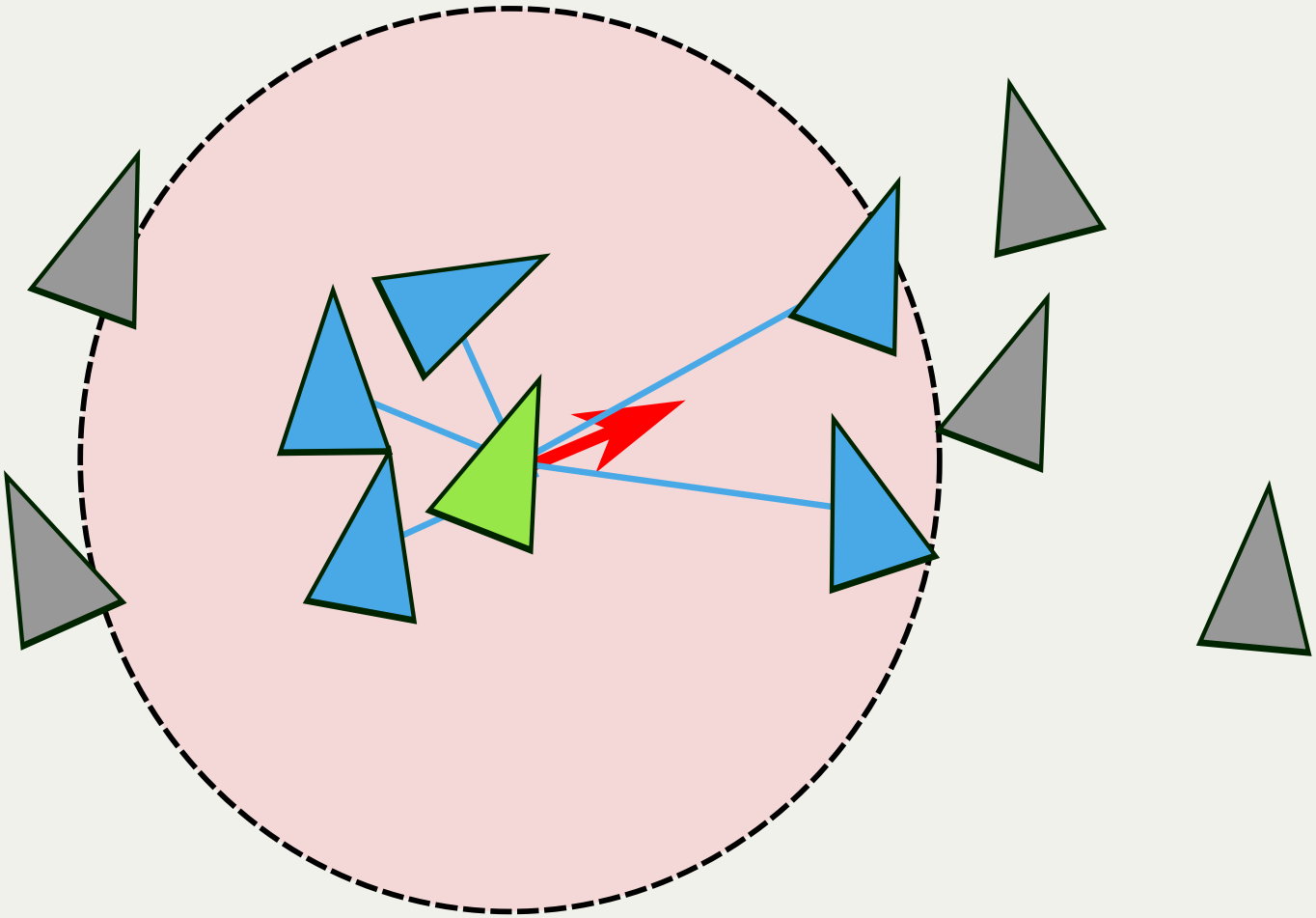


Alignment Pseudocode

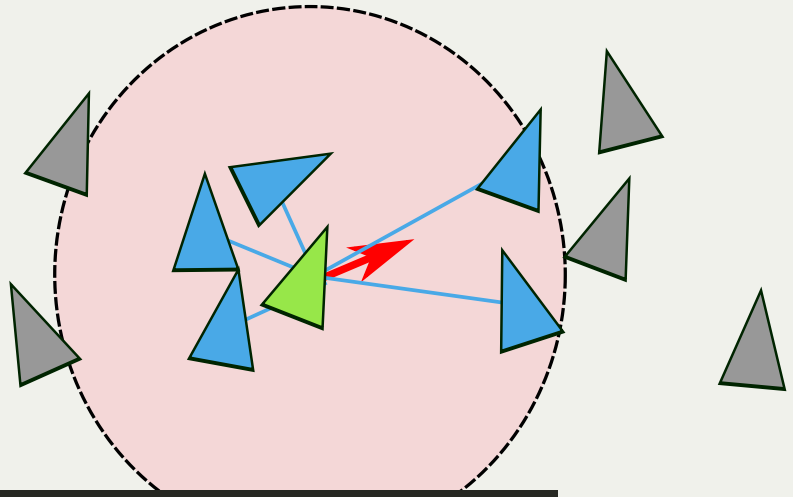


```
1  def alignment(boid: Boid, boids: Boid[]) {
2    perceived_velocity: Vec3 = 0
3    neighbors_count = 0
4
5    for (b : boids) {
6      if b != boid and distance(b, boid) < alignment_distance
7        perceived_velocity += b.velocity - boid.velocity
8        ++neighbors_count
9    }
10
11   if (neighbors_count >= 0) perceived_velocity /= neighbors_count
12
13   return perceived_velocity * alignment_scale
14 }
```

Rule 3: Cohesion



Cohesion Pseudocode



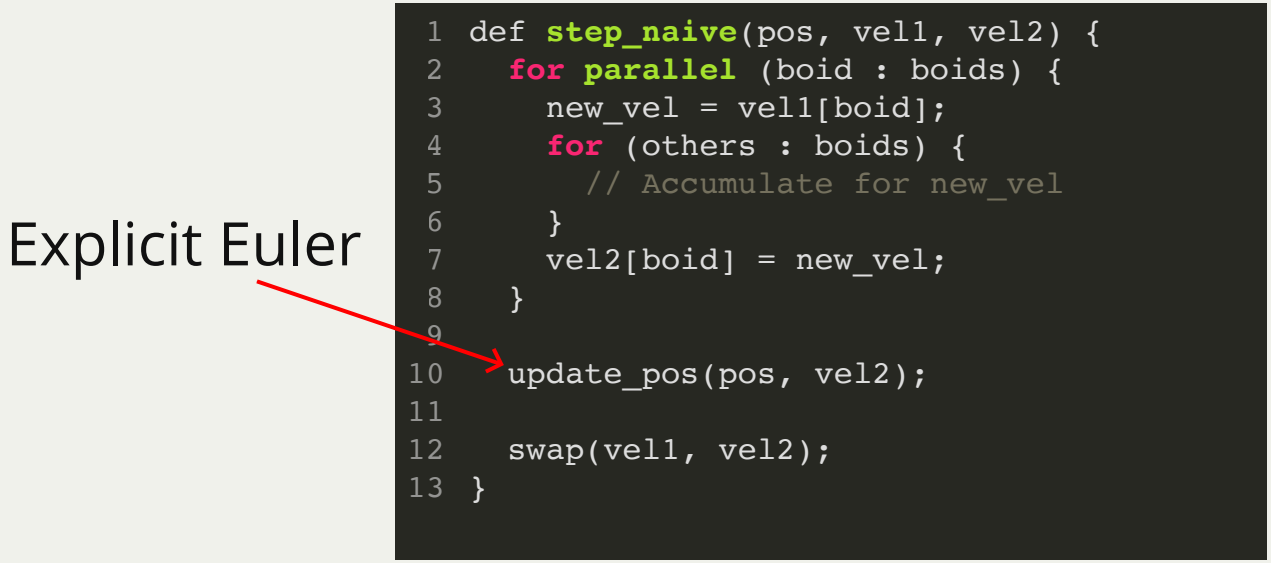
```
1  def cohesion(boid: Boid, boids: Boid[]) {
2    center_of_mass: Vec3 = 0
3    neighbors_count = 0
4
5    for (b : boids) {
6      if b != boid and distance(b, boid) < cohesion_distance
7        center_of_mass += b.position
8        ++neighbors_count
9    }
10
11   if (neighbors_count > 0) {
12     center_of_mass /= neighbors_count
13     return (center_of_mass - boid.position) * cohesion_scale
14   }
15
16   return 0
17 }
```

Naive Implementation: Pseudocode

```
1  def step_naive(pos, vel1, vel2) {
2    for parallel (boid : boids) {
3      new_vel = vel1[boid];
4      for (others : boids) {
5        // Accumulate for new_vel
6      }
7      vel2[boid] = new_vel;
8    }
9
10   update_pos(pos, vel2);
11
12   swap(vel1, vel2);
13 }
```

Naive Implementation: Pseudocode

Explicit Euler



```
1  def step_naive(pos, vel1, vel2) {
2    for parallel (boid : boids) {
3      new_vel = vel1[boid];
4      for (others : boids) {
5        // Accumulate for new_vel
6      }
7      vel2[boid] = new_vel;
8    }
9
10   update_pos(pos, vel2);
11
12   swap(vel1, vel2);
13 }
```

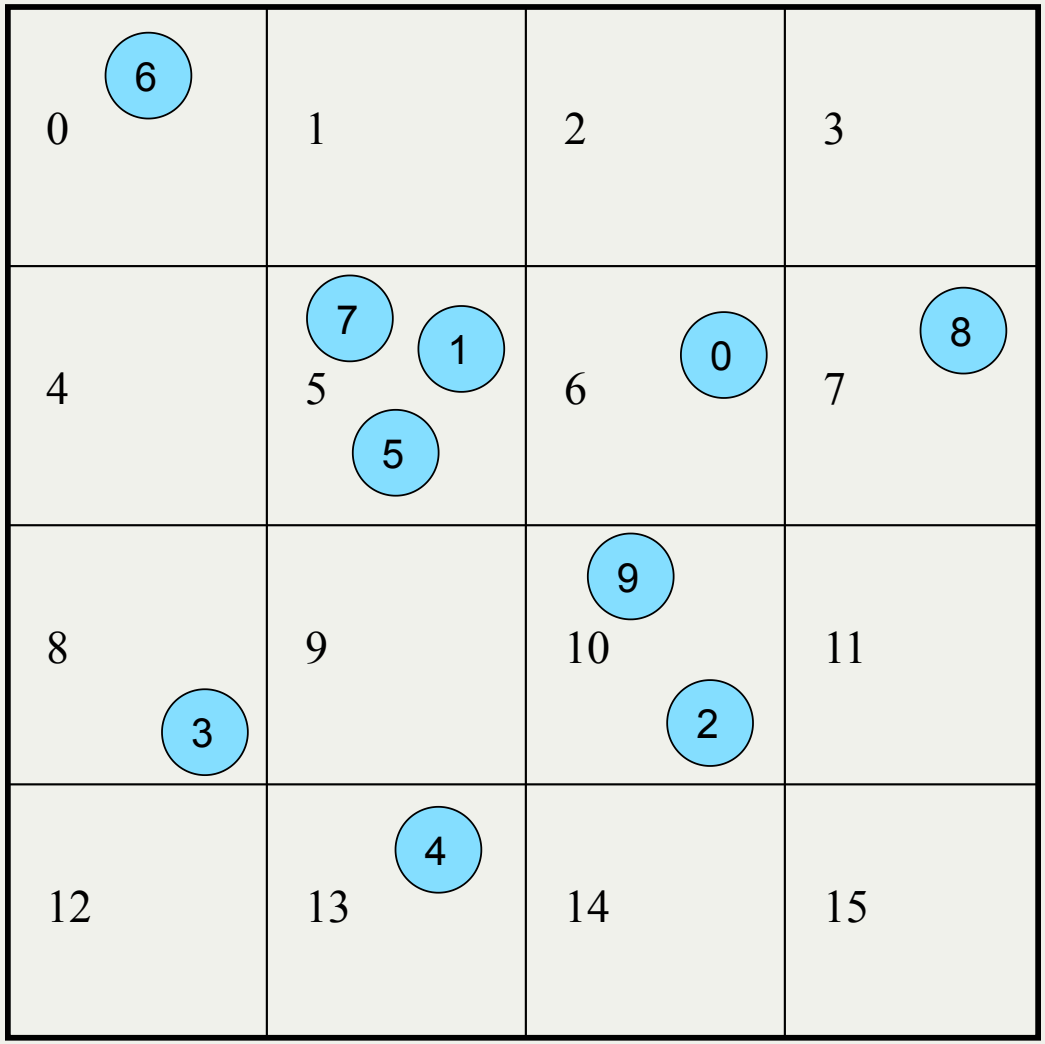
Naive Implementation: Pseudocode

```
1  def step_naive(pos, vel1, vel2) {
2    for parallel (boid : boids) {
3      new_vel = vel1[boid];
4      for (others : boids) {
5        // Accumulate for new_vel
6      }
7      vel2[boid] = new_vel;
8    }
9
10   update_pos(pos, vel2);
11
12   swap(vel1, vel2);
13 }
```

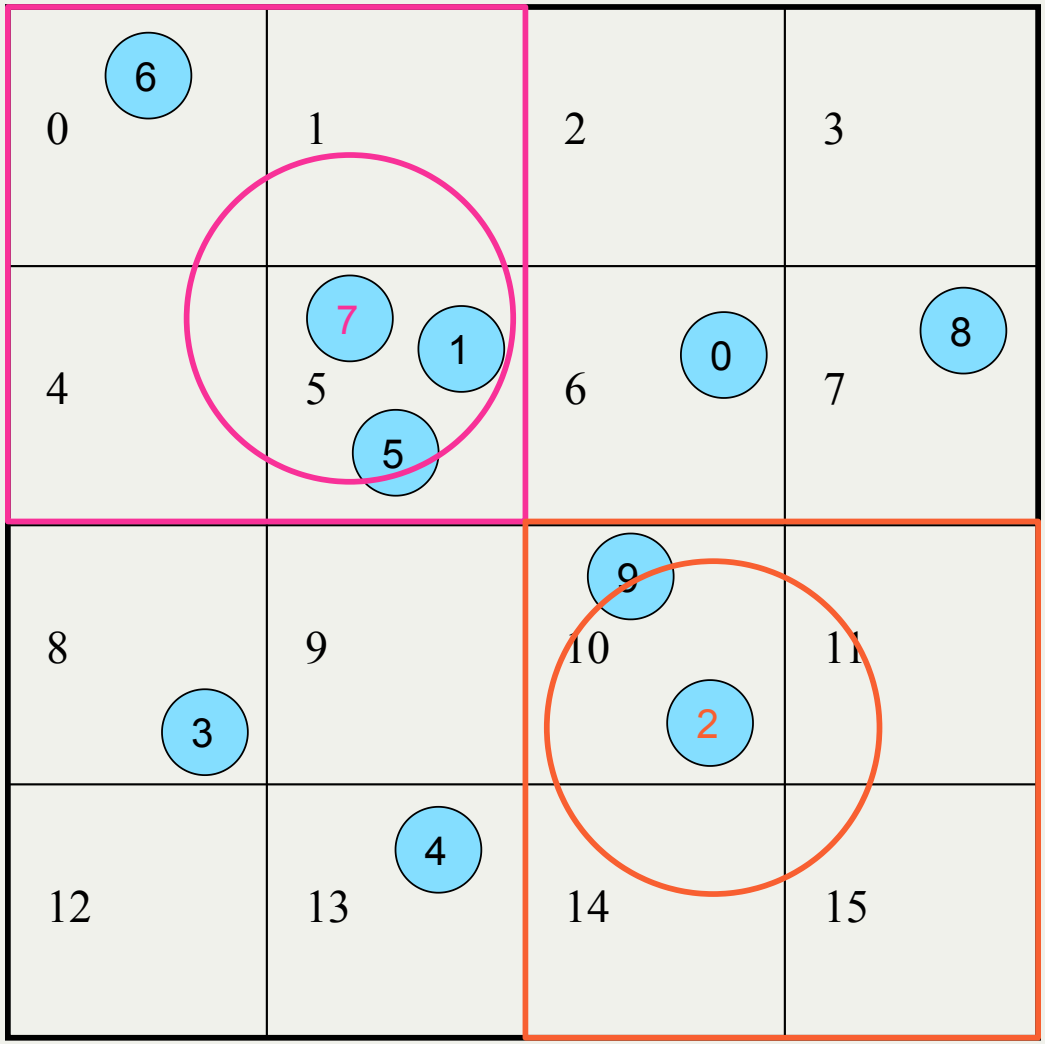
Buffer Ping-Ponging

Uniform Grid

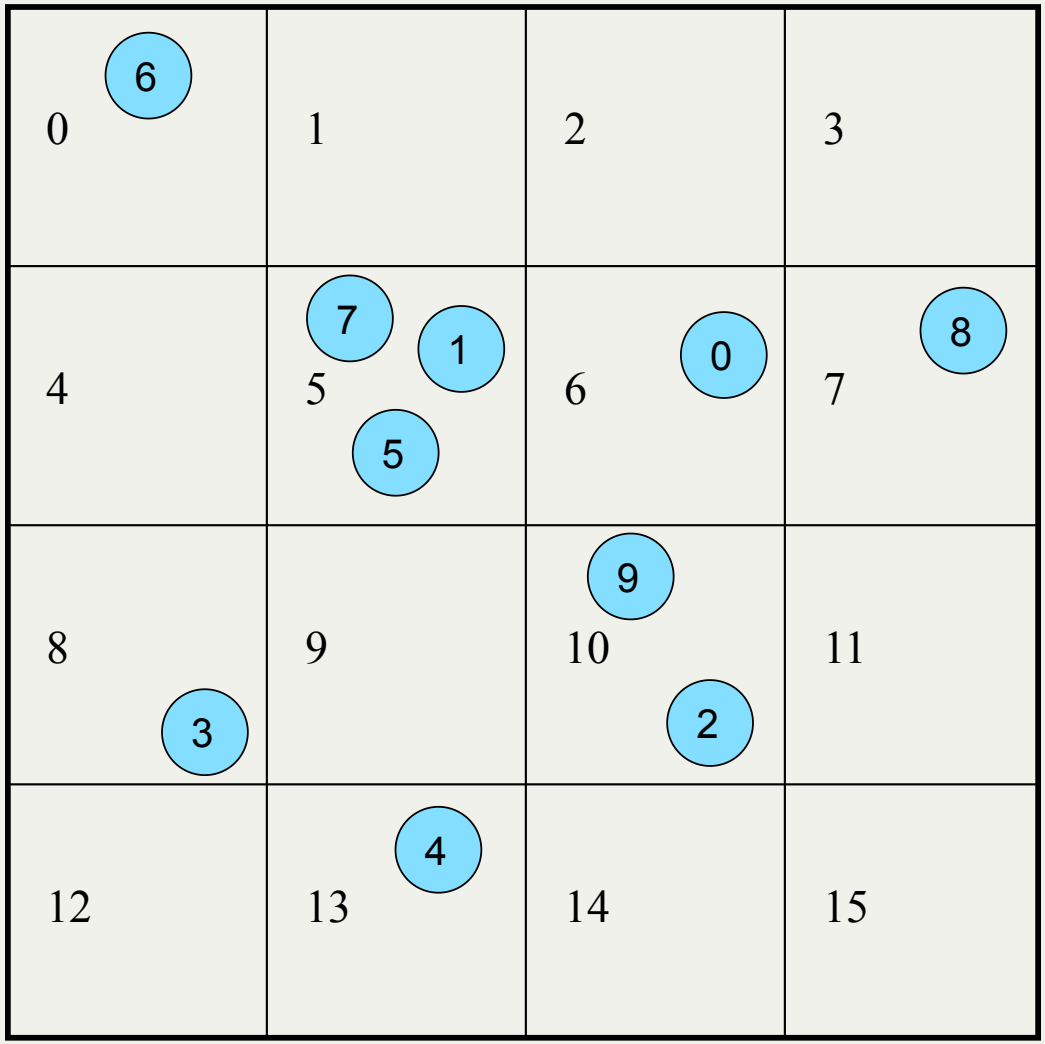
Uniform Grid



Grid Search



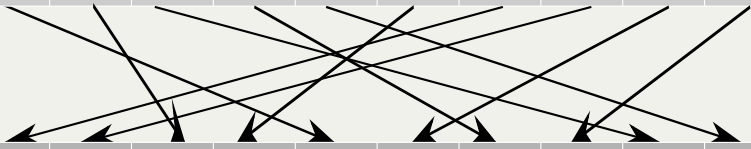
How to store the grid?



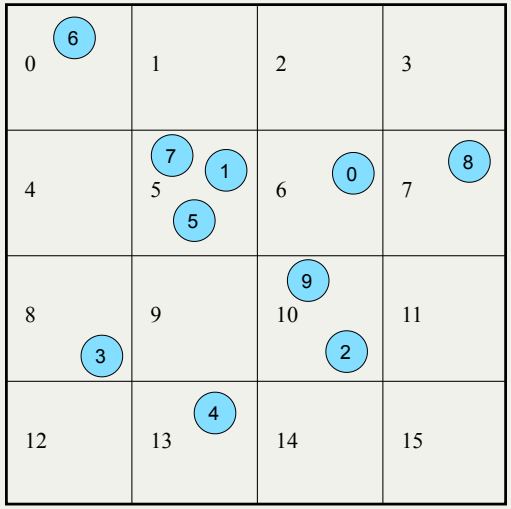
Sort boids according to grid

Boid ID	0	1	2	3	4	5	6	7	8	9
Grid ID	6	5	10	8	13	5	0	5	7	10

Sort:



Boid ID	6	7	1	5	0	8	3	9	2	4
Grid ID	0	5	5	5	6	7	8	10	10	13



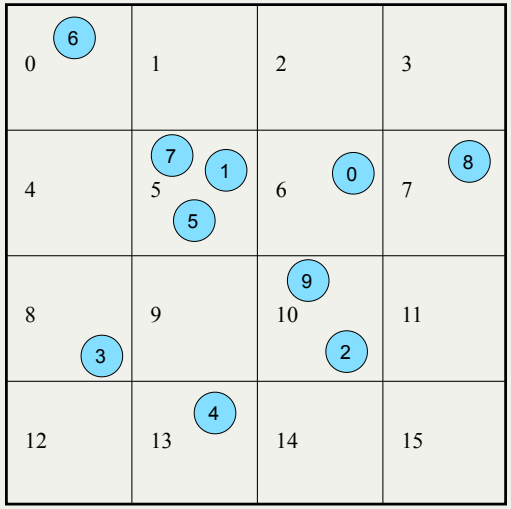
Thrust Library to Rescue

Boid ID	0	1	2	3	4	5	6	7	8	9
Grid ID	6	5	10	8	13	5	0	5	7	10

Sort:

Boid ID	6	7	1	5	0	8	3	9	2	4
Grid ID	0	5	5	5	6	7	8	10	10	13

```
1 thrust::sort_by_key(grid_indices,
2                     grid_indices + objects_count,
3                     boid_indices);
```



Note: Data not sorted

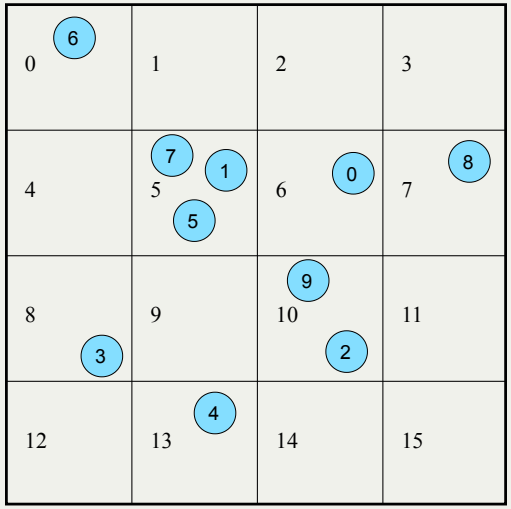
Boid ID	0	1	2	3	4	5	6	7	8	9
Grid ID	6	5	10	8	13	5	0	5	7	10

Sort:

Boid ID	6	7	1	5	0	8	3	9	2	4
Grid ID	0	5	5	5	6	7	8	10	10	13

Indirection:

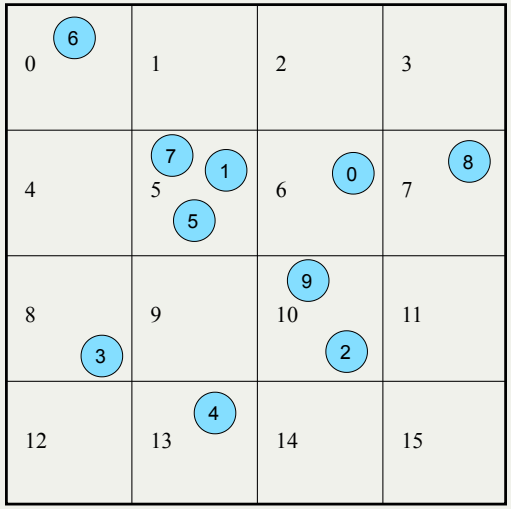
Boid ID	0	1	2	3	4	5	6	7	8	9
Boid Position	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
Boid Velocity	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9



The first and last boid of each grid

Boid ID	6	7	1	5	0	8	3	9	2	4
Grid ID	0	5	5	5	6	7	8	10	10	13

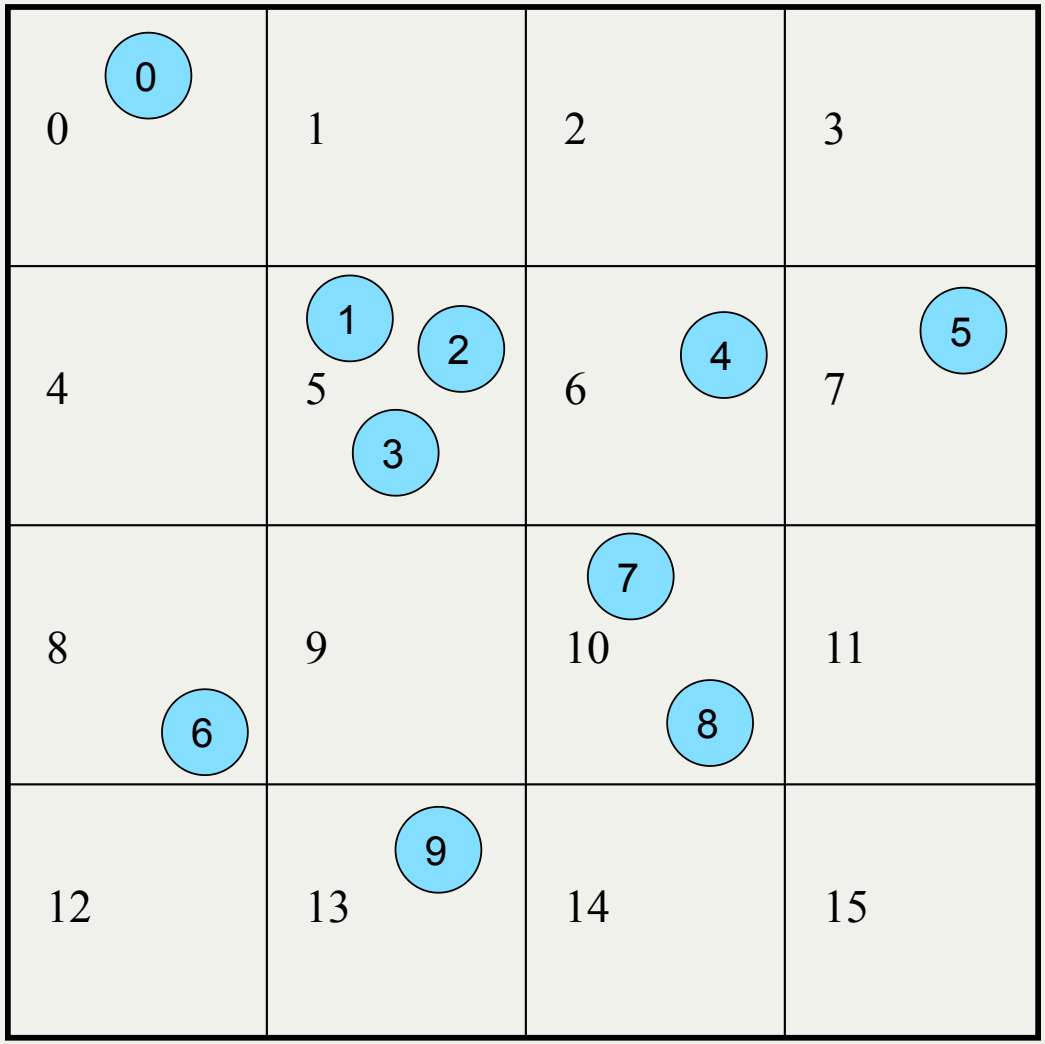
Grid Cell Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Grid First	0	-1	-1	-1	-1	1	4	5	6	-1	7	-1	-1	9	-1	-1
Grid Last	0	-1	-1	-1	-1	3	4	5	6	-1	8	-1	-1	9	-1	-1



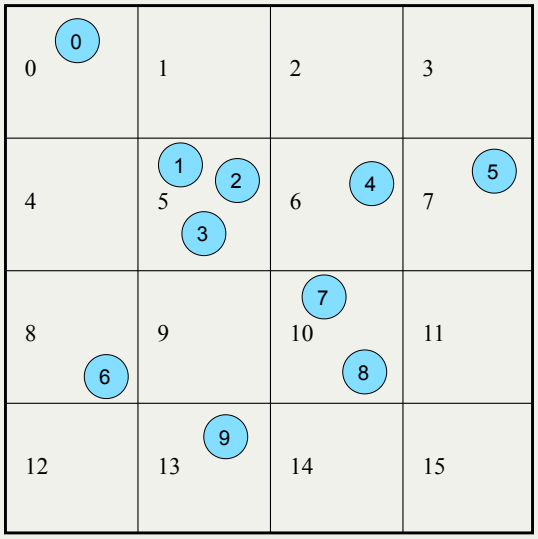
Uniform Grid: Pseudocode

```
1  def step_uniform_grid(pos, vel1, vel2) {
2    indices, grid_indices = compute_indices();
3
4    sort_by_key(grid_indices, indices);
5
6    grid_first, grid_last = identify_first_last(grid_indices);
7
8
9    for parallel (boid : boids) {
10         neighbor_grid_cells = calculate_neighbor_grid_cells(boid);
11
12     new_vel = vel1[boid];
13     for (cell : neighbor_grid_cells) {
14       for (neighbor from grid_first[cell] to grid_last[cell]) {
15         neighbor_pos = pos[indices[neighbor]];
16         neighbor_vel = vel[indices[neighbor]];
17         // Accumulate for new_vel
18       }
19     }
20     vel2[boid] = new_vel;
21   }
22
23   update_pos(pos, vel2);
24   swap(vel1, vel2);
25 }
```

Cutting the middleman

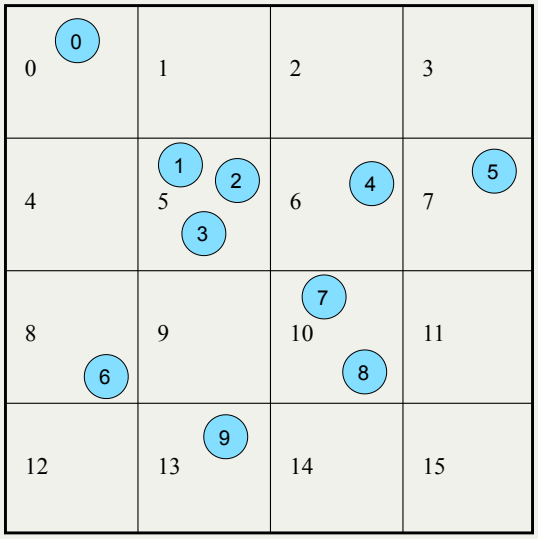


Cutting the middleman



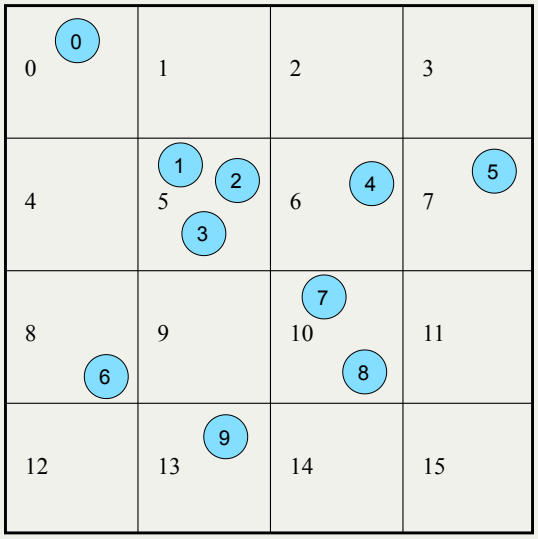

```
1 thrust::copy(pos, pos + objects_count, pos_sorted);
2 thrust::copy(vel, vel + objects_count, vel_sorted);
3 thrust::sort_by_key(grid_indices,
4                      grid_indices + objects_count,
5                      pos_sorted,
6                      vel_sorted);
```

Cutting the middleman




```
1  thrust::copy(po                    _count, pos_sorted);
2  thrust::copy(v      ve      bj     count, vel_sorted);
3  thrust::sort_       y(gr  _in
4                      grid_          + objects_count,
5                       pos_so
6                       v    sort
```

Cutting the middleman



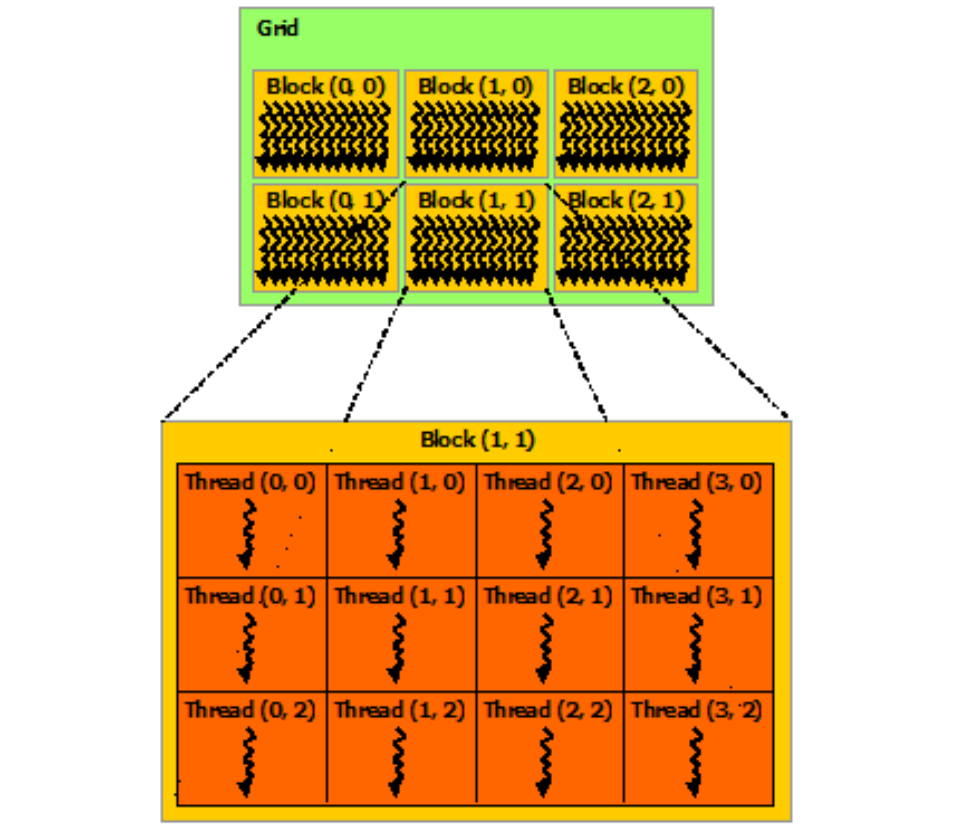

```
1 thrust::sort_by_key(grid_indices,
2                     grid_indices + objects_count,
3                     indices);
4 thrust::gather(indices, indices + objects_count, pos, pos_sorted);
5 thrust::gather(indices, indices + objects_count, vel, vel_sorted);
```

Coherent Grid: Pseudocode

```
1  def step_coherent_grid(pos, vel1, vel2) {
2    indices, grid_indices = compute_indices();
3
4    pos_sorted, vel_sorted = pos, vel1
5    sort_by_key(grid_indices, pos_sorted, vel_sorted);
6
7    grid_first, grid_last = identify_first_last(grid_indices);
8
9
10   for parallel (boid : boids) {
11         neighbor_grid_cells = calculate_neighbor_grid_cells(boid);
12
13     new_vel = vel1[boid];
14     for (cell : neighbor_grid_cells) {
15       for (neighbor from grid_first[cell] to grid_last[cell]) {
16         neighbor_pos = pos_sorted[neighbor];
17         neighbor_vel = vel_sorted[neighbor];
18         // Accumulate for new_vel
19       }
20     }
21     vel2[boid] = new_vel;
22   }
23
24   update_pos(pos, vel2);
25   swap(vel1, vel2);
26   swap(pos, pos_sorted);
27 }
```

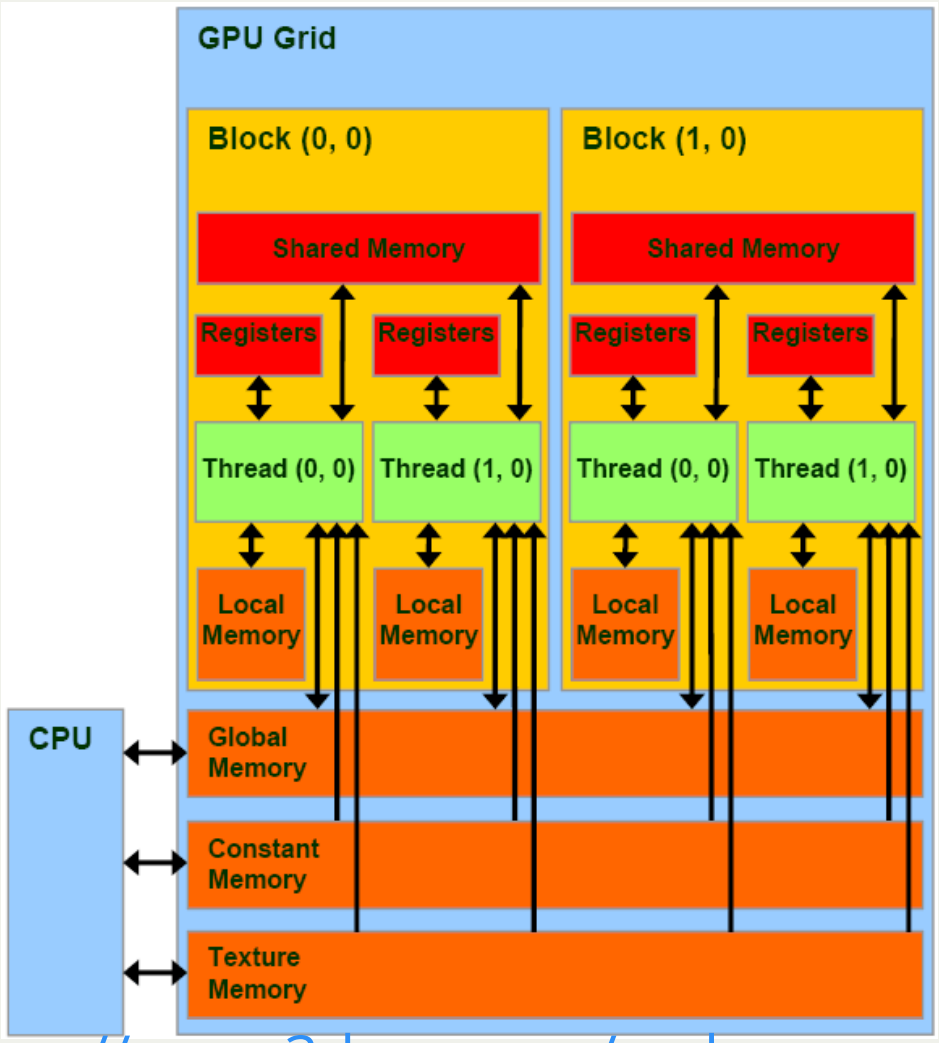
Shared-memory Optimization

CUDA Thread Hierarchy



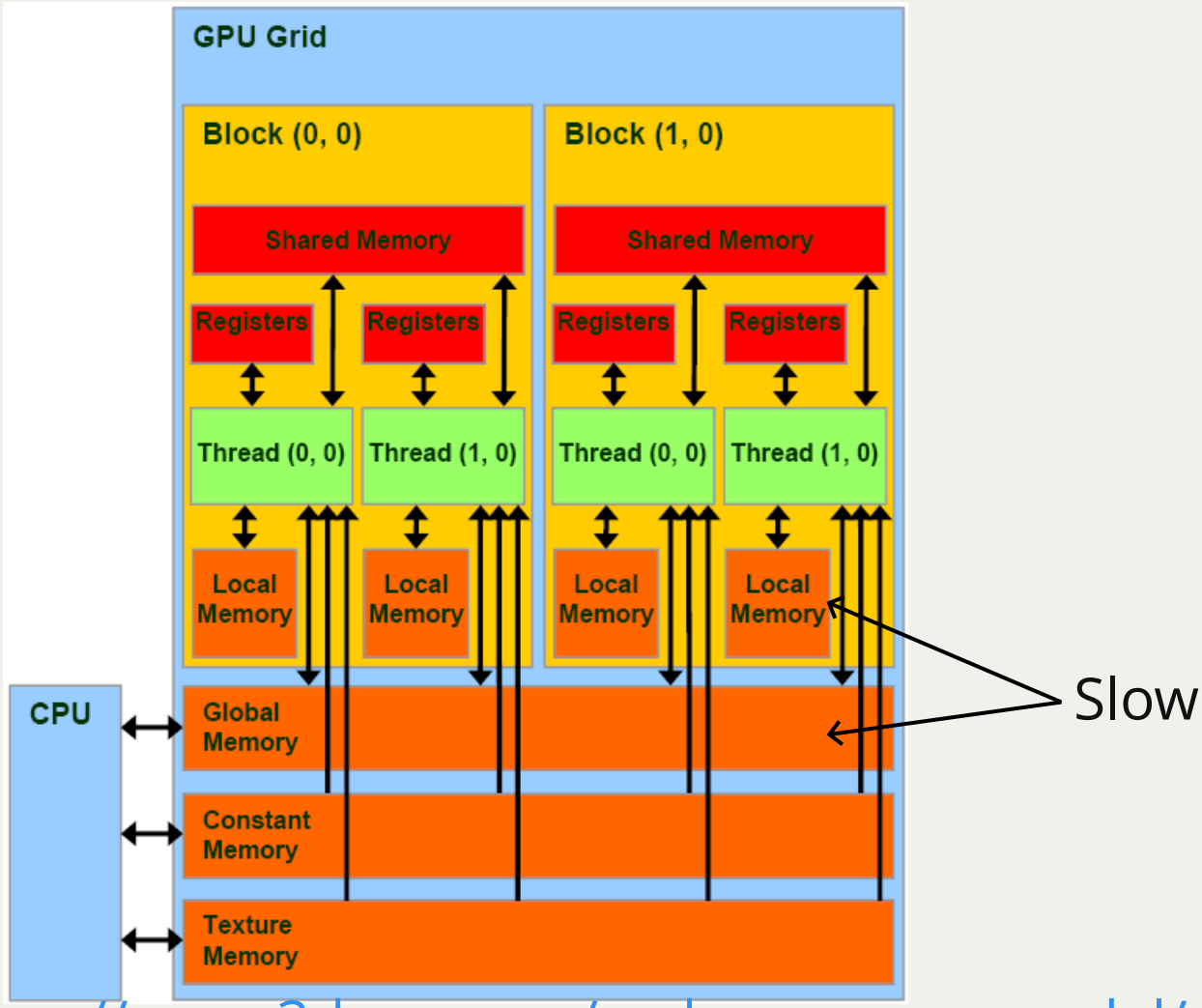
Credit: docs.nvidia.com/cuda/cuda-c-programming-guide/graphics/grid-of-thread-blocks.png

CUDA Memory



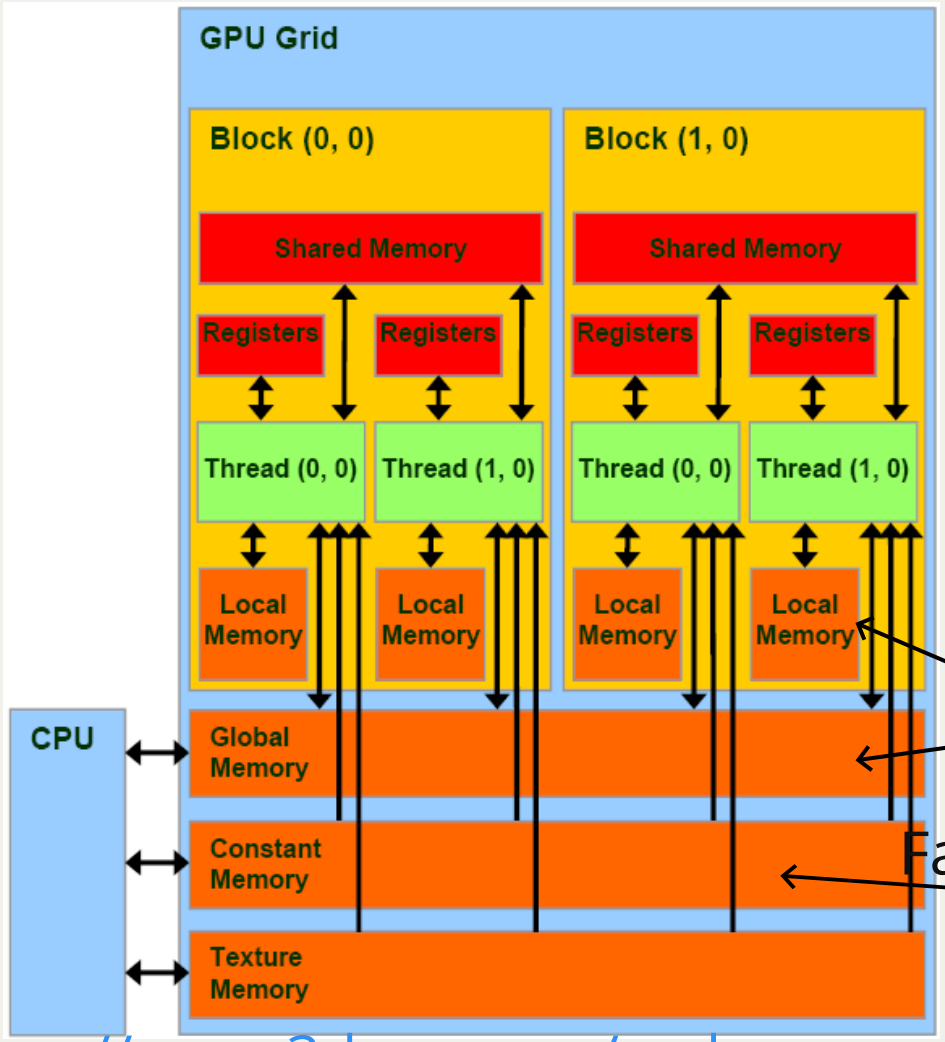
Credit: https://www.3dgep.com/cuda-memory-model/

CUDA Memory



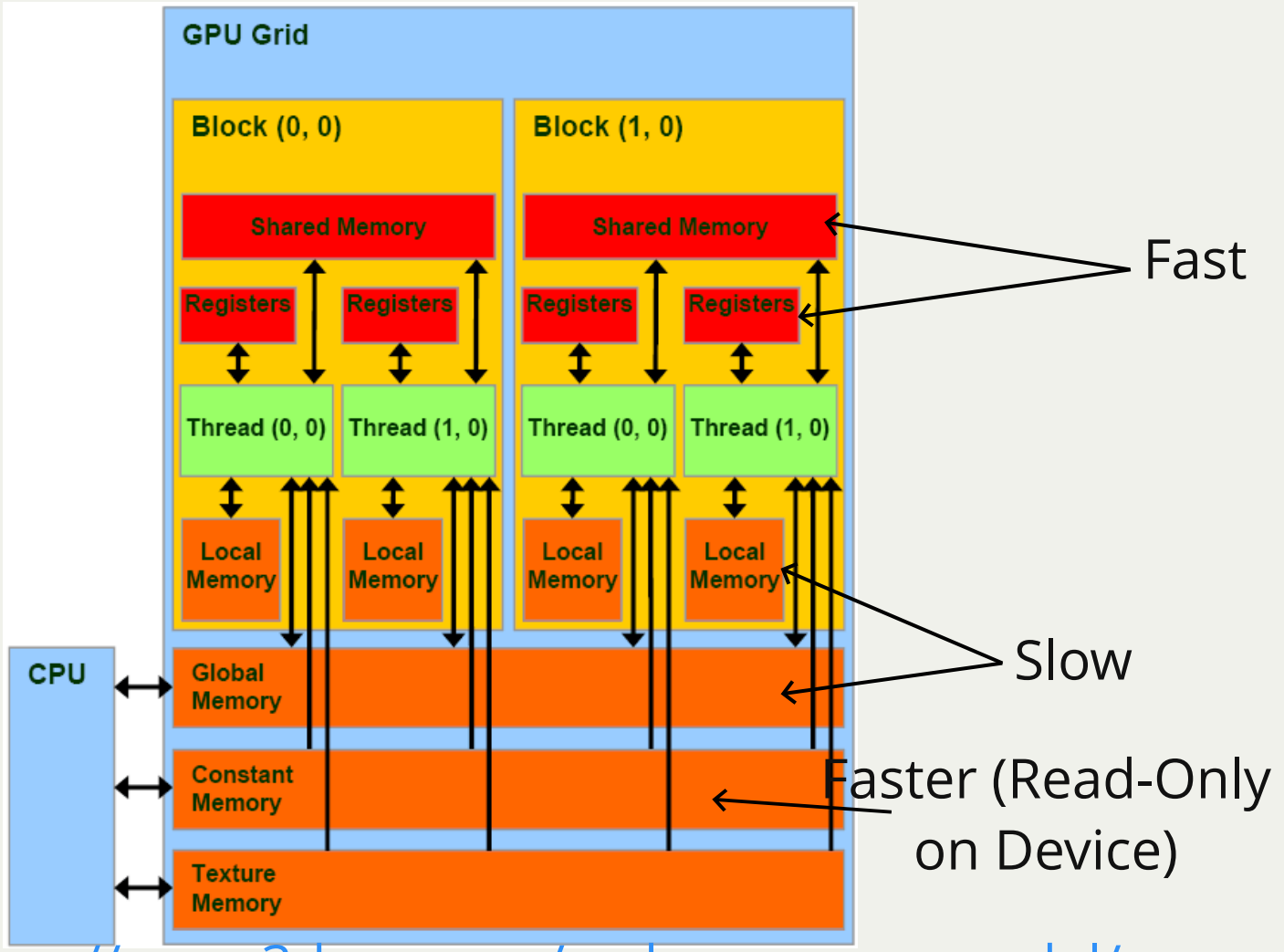
Credit: https://www.3dgep.com/cuda-memory-model/

CUDA Memory



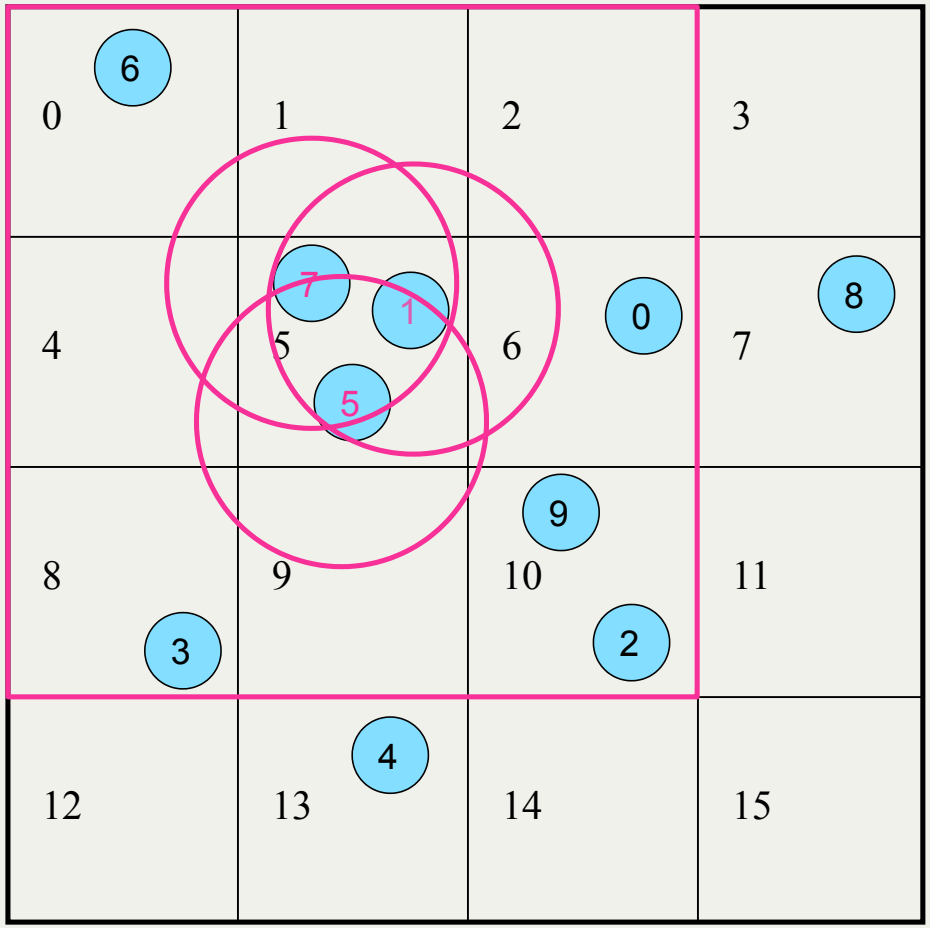
Credit: https://www.3dgep.com/cuda-memory-model/

CUDA Memory



Credit: https://www.3dgep.com/cuda-memory-model/

shovel all neighbor grids of a warp to shared memory



Shared Grid Pseudocode

```
1  def step_shared_grid(pos, vel1, vel2) {
2    // same
3
4    for parallel (warp: warps) {
5      while (not process all neighbors) {
6
7        // calculate all neighbor grid cells in a wrap
8
9        // copy some boids in neighbor cells to shared memory based on capacity
10
11       for parallel boids in wrap {
12         for (neighbor from shared_first to shared_last) {
13           // Accumulate for new_vel
14         }
15         vel2[boid] = new_vel;
16       }
17     }
18   }
19
20   // same
21 }
```

Live Demo

Graphics Programming Virtual Meetup