LA-UR-13-27416

Title:

High-Level Data Parallelism

Author(s):

Li-ta Lo
Chris Sewell
Jim Ahrens
Pat McCormick

Intended for: New Mexico Supercomputing Challenge Kick-off, Socorro, NM, October 2013

High-Level Data Parallelism

Li-Ta Lo
Chris Sewell
James Ahrens
Patrick McCormick

Los Alamos National Laboratory

LA-UR-13-27416

Parallel Programming Pays a Life-Long Dividend

- Supercomputer Hardware Advances Everyday
  - Higher and higher parallelism
  - Every toy you play with today will be a dinosaur when you graduate from college
  - Optimizations tailored to a certain architecture will be obsolete when you implement it
- Parallel Programming APIs Come and Go
  - Nobody programs with shaders for GPGPU anymore
  - Will this also happen to OpenCL, CUDA, etc. in the future?
- High-Level Parallelism
  - Will not change over time

Five Operations You Can Do with a Lot of Data in Parallel

- Generate/Create
  - Automatically fill with programmatically defined data
- Transform
  - Apply some "operation" to each element of the data
  - Also called "Map" in many contexts
- Compact
  - Take only the elements in which you are interested
  - Also called "Filter" in many contexts
- Expand
  - The opposite of Compact
  - Create a larger data set from a smaller data set
- Aggregate
  - Calculate a "summary" of your data (e.g., sum or average)
  - Also called "Reduce" or "Fold"
  - "Scan" also provides all intermediate values

Brief Introduction to Thrust

- Thrust is a NVidia C++ template library for CUDA. It can also target OpenMP and we are creating new backends to target other architectures
- Thrust allows you to program using an interface similar the C++ Standard Template Library (STL)
- Most of the STL algorithms in Thrust are data parallel

Simple Examples with Thrust Pseudocode

- Generate

```
thrust::sequence(0,4)  0  1  2  3  4
```

- Transform

```
input                  4  5  2  1  3
thrust::transform(+1)  5  6  3  2  4
```

- Compact

```
input                  4  5  2  1  3
thrust::copy_if(even)  4  2
```

- Expand

```
input                  4  5  2  1  3
thrust::for_each(x,2x) 4  8  5 10  2  4  1  2  3  6
```

- Aggregate

```
input                  4  5  2  1  3
thrust::reduce(+)      15
```

Generate Data in Parallel

- Many copies of a certain constant value

  -
    ```
    // Ten elements with initial value of integer 1
    thrust::device_vector<int> x(10, 1);
    ```
- A sequence of increasing or decreasing values

  -
    ```
    // Allocate space for ten integers, uninitialized
    thrust::device_vector<int> y(10);
    // Fill the space with integers
    thrust::sequence(y.begin(), y.end(), 5, 2);
    ```
- Random Values
  - Multiple copies of a random number generator
  - Give each one a different seed

Transform: Vector Addition

- Apply a binary operator “plus” to each element in x and y

  -
    ```
    thrust::transform(x.begin(), x.end(), // begin and end of the
                                          // first input vector
    y.begin(),                            // begin of the second
                                          // input vector
    result.begin(),                       // begin of the result
                                          // vector
    thrust::plus<int>());                 // predefined integer
                                          // addition
    ```

  -
    ```
         x: 1  1  1  1  1  1  1  1  1  1
                     +
         y: 5  7  9 11 13 15 17 19 21 23
                     =
    result: 6  8 10 12 14 16 18 20 22 24
    ```

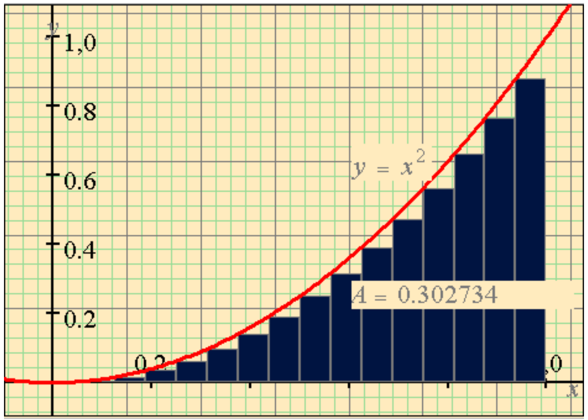
Transform: Uniform Sampling of a Mathematical Function

- Q: How are we going to generate something more complicated?
  A: Start from some sequence and apply some transformation
- Sampling the function f(x) = x^2 in the interval of [0, 1]

  -
    ```
    // Generate a sequence of xi in [0,1] with dx=0.1
    // in between each of them
    float dx = 1.0f/10.0f;
    thrust::sequence(x.begin(), x.end(), 0.0f, dx);

    // Apply the square operation to each of the xi
    // to transform into f(xi) = yi = xi2
    thrust::transform(x.begin(), x.end(),
                      y.begin(),
                      square());
    ```

Reduce: Simple Numerical Integration

- Apply what we learned to estimate the integral $\int_0^1 f(x)dx$ by $\sum_{i=1}^{n} f(x_i)dx$
- Create a constant vector of dx
- Sample the function on each x_i
- Apply multiply operation on each element of x_i and dx
-
```
thrust::transform(y.begin(), y.end(),
                  dx.begin(), y_dx.begin(),
                  thrust::multiply<float>());
```



-

dx	=	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
$f(x_i)$	=	0.0	0.01	0.04	0.09	0.16	0.25	0.36	0.49	0.64	0.81	1.0
$f(x_i)*dx$	=	0.0	0.001	0.004	0.009	0.016	0.025	0.036	0.049	0.064	0.081	0.1

- Sum all the $f(x_i)*dx$ to get $\int_0^1 f(x)dx$
-
```
float result = thrust::reduce(y_dx.begin(), y_dx.end());
```

Scan: Simple Numerical Integration

- What happens if we are interested in the integral $F(t) = F(0) + \int_0^t f(x)dt$

  on the interval [0, 1] instead of just a number?
- Calculate a running sum by using scan

-
  ```
  thrust::inclusive_scan(y_dx.begin(), y_dx.end(),
                         F.begin());
  ```

-
  ```
  f(x_i)*dx =  0.0 0.001 0.004 0.009 0.016 0.025 0.036 0.049 0.064 0.081   0.1
  F(t)      =  0.0 0.001 0.005 0.014 0.030 0.055 0.091 0.140 0.204 0.285 0.385
  ```
- The last element of the scan (0.385) is the same as the output of reduce
- In mathematical terms, $\int_0^1 f(x)dx = F(1) - F(0)$

Scan: Calculating the Fibonacci Sequence by Matrix Multiplication

- The reduce and scan operators can also work with a user defined type
- The Fibonacci Sequence is defined as

$$F_{n+1} = F_n + F_{n-1} \quad \text{with} \quad F_0 = 0, F_1 = 1$$

- By "unrolling" the recurrence we have

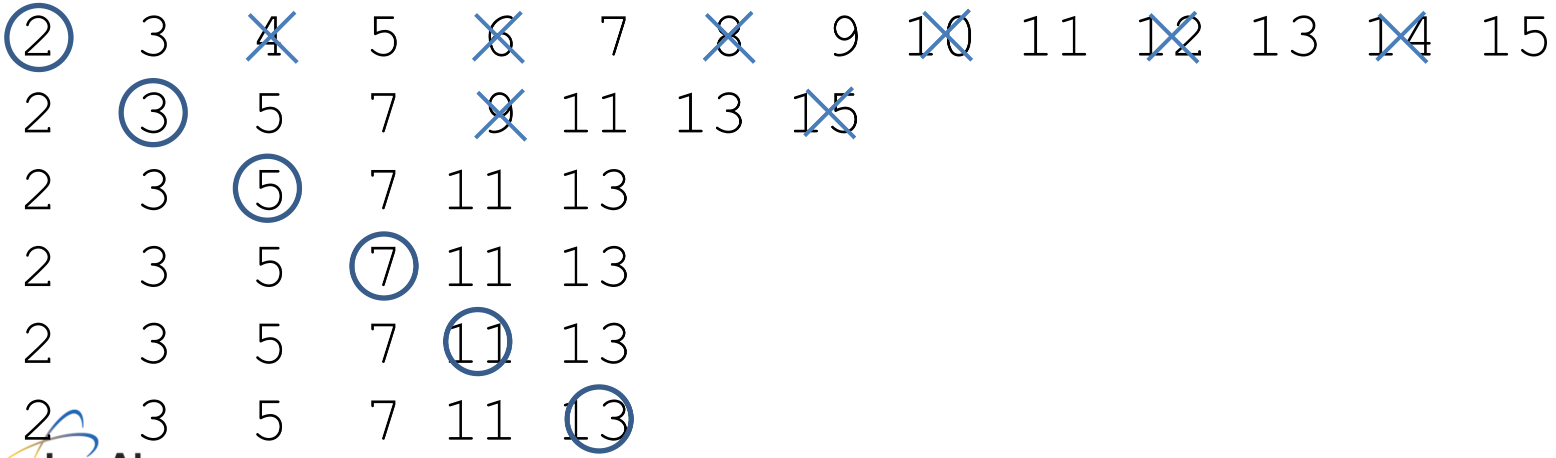
$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix}$$

- Thus we can compute F_n by matrix multiplication

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} \textcircled{1} & 1 \\ 1 & 0 \end{bmatrix} \quad \begin{bmatrix} \textcircled{2} & 1 \\ 1 & 1 \end{bmatrix} \quad \begin{bmatrix} \textcircled{3} & 2 \\ 2 & 1 \end{bmatrix} \quad \begin{bmatrix} \textcircled{5} & 3 \\ 3 & 2 \end{bmatrix} \quad \begin{bmatrix} \textcircled{8} & 5 \\ 5 & 3 \end{bmatrix}$$

Compaction: Finding Prime Numbers Using the Sieve of Eratosthenes

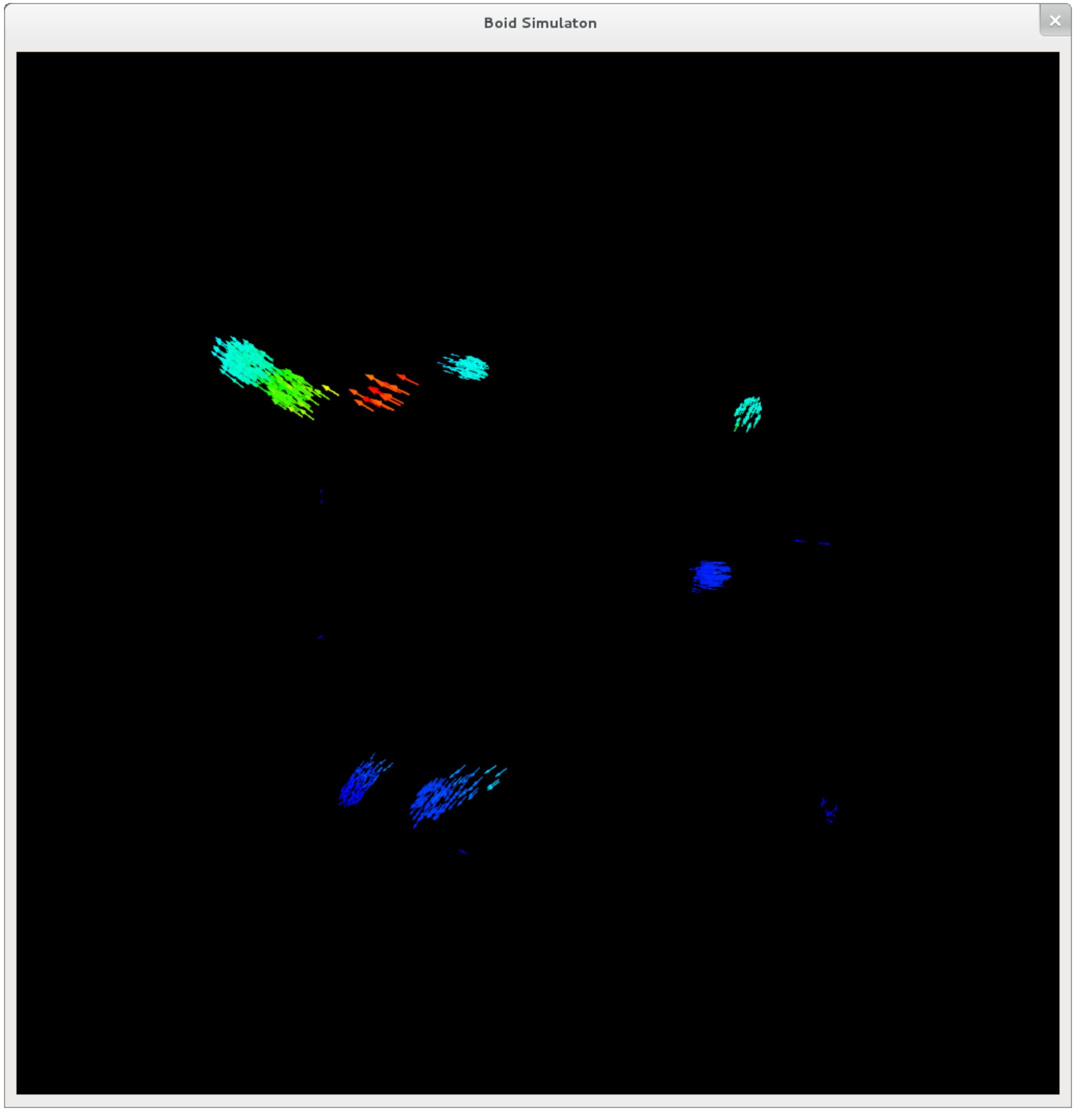
- Start with a vector containing the sequence of integers from 2 to N
- The first element in this vector is prime
- Use compaction to copy only elements of the vector not divisible by this prime into an updated vector
- The second element in this vector is prime
- Repeat the two steps above until you reach the end of the vector
- 

  2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15
  2 3 5 7 ~~9~~ 11 13 ~~15~~
  2 3 5 7 11 13
  2 3 5 7 11 13
  2 3 5 7 11 13
  2 3 5 7 11 13

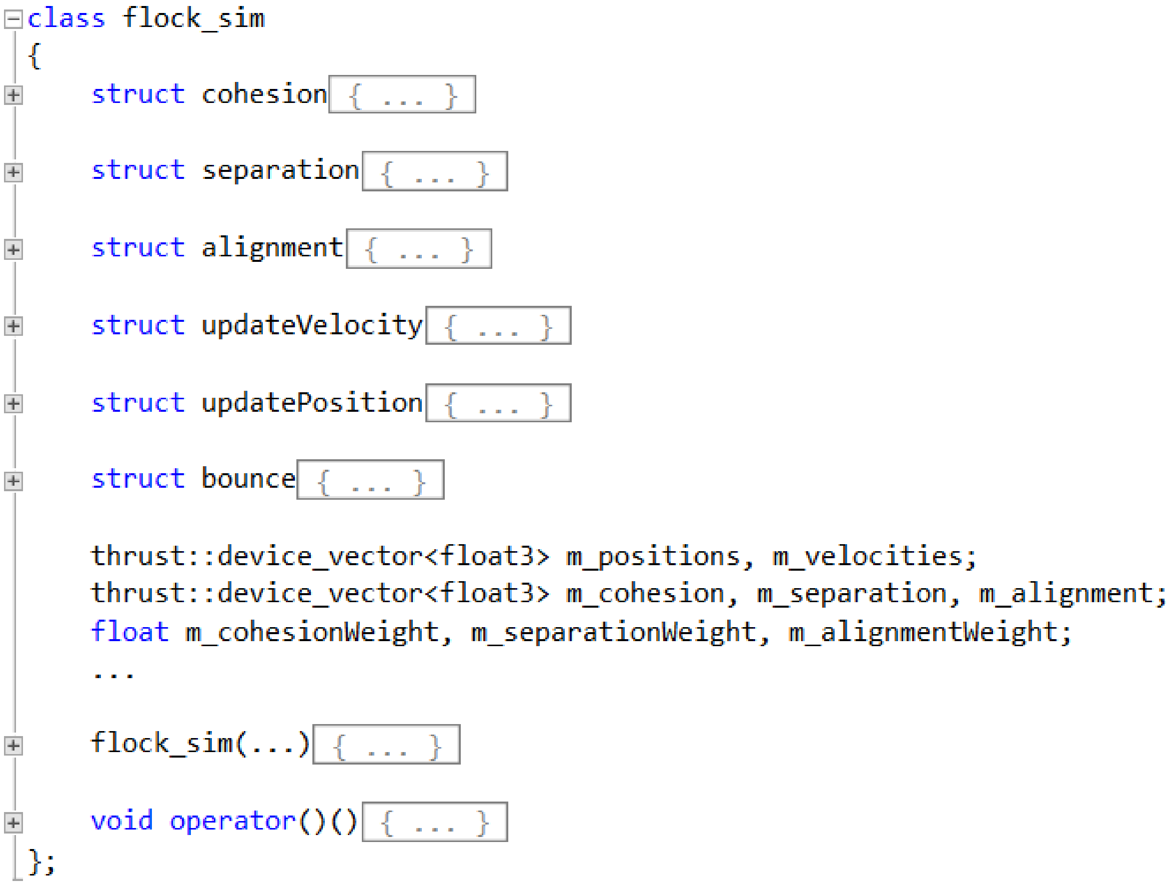
Boid Simulation

- Simulate flocking behavior of a group of “boids”
- At each time step, velocities are adjusted based on three parameters, each dependent only upon observing other nearby boids:
  - Cohesion: Each boid wants to move towards the centroid of other boids in its vicinity, to join the flock
  - Separation: Each boid wants to move away from other boids that are too close to it, to avoid collisions
  - Alignment: Each boid wants to adjust its velocity (direction and magnitude) to match that of other boids in its vicinity, to move in sync with the flock
- Positions are then updated for the next time step based on the new velocities
- Thrust transform and for_each functions are used in order to parallelize the computations for all the boids
- Functors are used to compute the cohesion, separation, and alignment parameters for a boid, and to update its velocity and position
- Reference: http://syntacticsalt.com/2011/03/10/functional-flocks/ by Matt Sottile

Boid Simulation Video

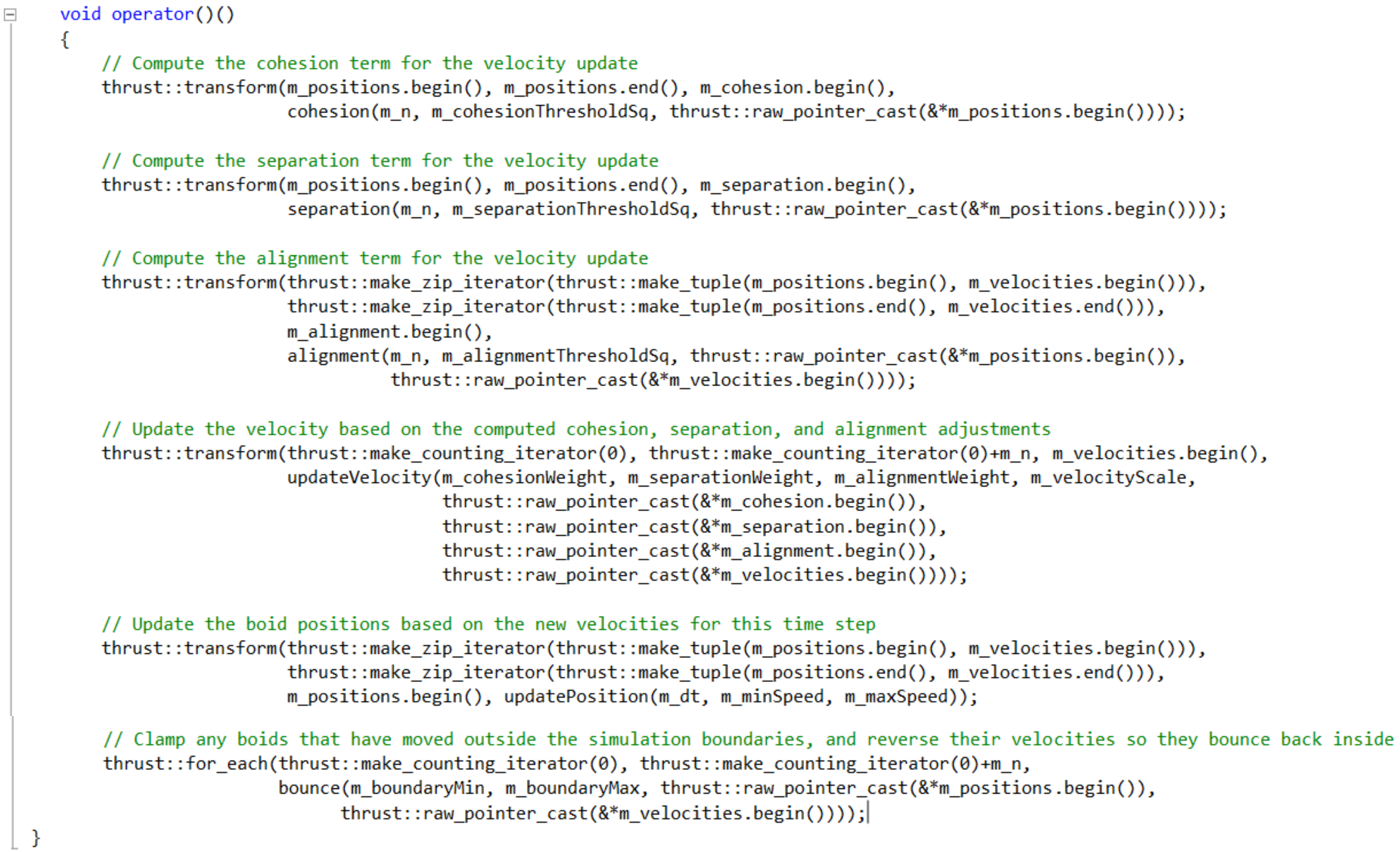


Outline of Flock Simulation Class



```
class flock_sim
{
    struct cohesion { ... }

    struct separation { ... }

    struct alignment { ... }

    struct updateVelocity { ... }

    struct updatePosition { ... }

    struct bounce { ... }

    thrust::device_vector<float3> m_positions, m_velocities;
    thrust::device_vector<float3> m_cohesion, m_separation, m_alignment;
    float m_cohesionWeight, m_separationWeight, m_alignmentWeight;
    ...

    flock_sim(...) { ... }

    void operator()() { ... }
};
```

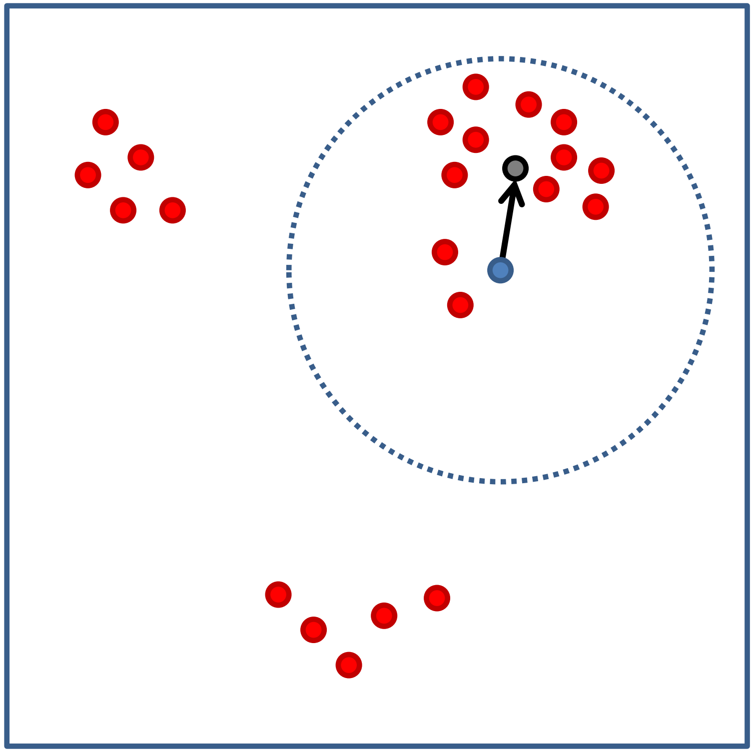
Main simulation loop



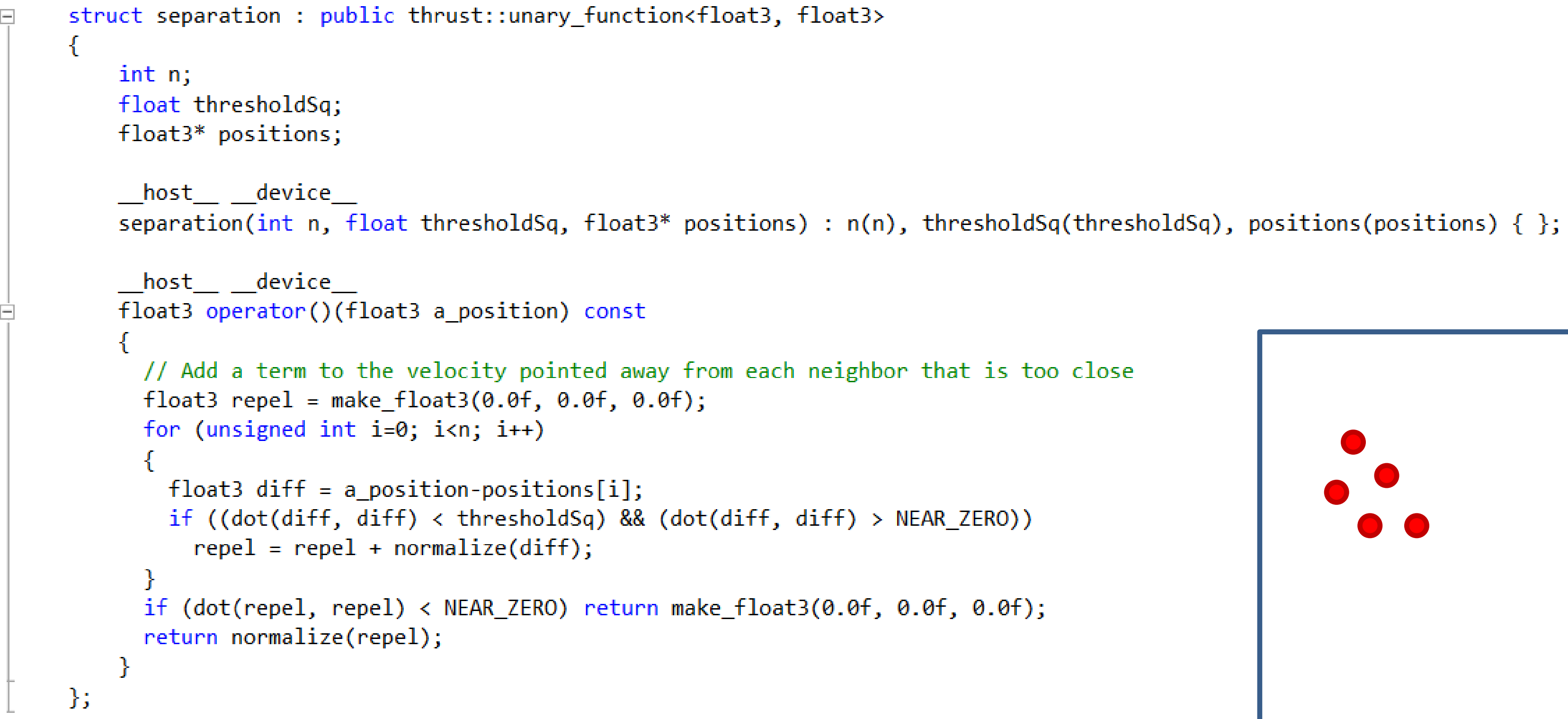
```
void operator()()
{
    // Compute the cohesion term for the velocity update
    thrust::transform(m_positions.begin(), m_positions.end(), m_cohesion.begin(),
                      cohesion(m_n, m_cohesionThresholdSq, thrust::raw_pointer_cast(&*m_positions.begin())));

    // Compute the separation term for the velocity update
    thrust::transform(m_positions.begin(), m_positions.end(), m_separation.begin(),
                      separation(m_n, m_separationThresholdSq, thrust::raw_pointer_cast(&*m_positions.begin())));

    // Compute the alignment term for the velocity update
    thrust::transform(thrust::make_zip_iterator(thrust::make_tuple(m_positions.begin(), m_velocities.begin())),
                      thrust::make_zip_iterator(thrust::make_tuple(m_positions.end(), m_velocities.end())),
                      m_alignment.begin(),
                      alignment(m_n, m_alignmentThresholdSq, thrust::raw_pointer_cast(&*m_positions.begin()),
                                thrust::raw_pointer_cast(&*m_velocities.begin())));

    // Update the velocity based on the computed cohesion, separation, and alignment adjustments
    thrust::transform(thrust::make_counting_iterator(0), thrust::make_counting_iterator(0)+m_n, m_velocities.begin(),
                      updateVelocity(m_cohesionWeight, m_separationWeight, m_alignmentWeight, m_velocityScale,
                                     thrust::raw_pointer_cast(&*m_cohesion.begin()),
                                     thrust::raw_pointer_cast(&*m_separation.begin()),
                                     thrust::raw_pointer_cast(&*m_alignment.begin()),
                                     thrust::raw_pointer_cast(&*m_velocities.begin())));

    // Update the boid positions based on the new velocities for this time step
    thrust::transform(thrust::make_zip_iterator(thrust::make_tuple(m_positions.begin(), m_velocities.begin())),
                      thrust::make_zip_iterator(thrust::make_tuple(m_positions.end(), m_velocities.end())),
                      m_positions.begin(), updatePosition(m_dt, m_minSpeed, m_maxSpeed));

    // Clamp any boids that have moved outside the simulation boundaries, and reverse their velocities so they bounce back inside
    thrust::for_each(thrust::make_counting_iterator(0), thrust::make_counting_iterator(0)+m_n,
                     bounce(m_boundaryMin, m_boundaryMax, thrust::raw_pointer_cast(&*m_positions.begin()),
                           thrust::raw_pointer_cast(&*m_velocities.begin())));
}
```

Cohesion Term

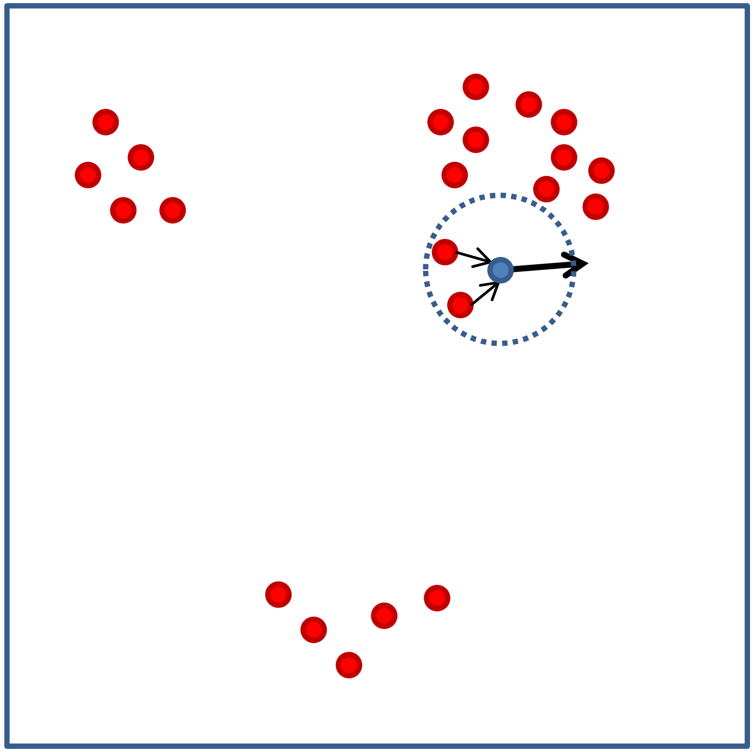
```
struct cohesion : public thrust::unary_function<float3, float3>
{
    int n;
    float thresholdSq;
    float3* positions;

    __host__ __device__
    cohesion(int n, float thresholdSq, float3* positions) : n(n), thresholdSq(thresholdSq), positions(positions) { };

    __host__ __device__
    float3 operator()(float3 a_position) const
    {
      // Compute centroid of all neighbors by searching through all other boids
      float3 centroid = make_float3(0.0f, 0.0f, 0.0f);
      int neighbors = 0;
      for (unsigned int i=0; i<n; i++)
      {
        float3 diff = a_position-positions[i];
        if (dot(diff, diff) < thresholdSq)
        {
          centroid = centroid + positions[i];
          neighbors++;
        }
      }
      if (neighbors == 0) return make_float3(0.0f, 0.0f, 0.0f);
      centroid.x /= neighbors;  centroid.y /= neighbors;  centroid.z /= neighbors;

      // Add a term to the velocity pointed towards the centroid of the neighbors
      return (centroid - a_position);
    }
};
```



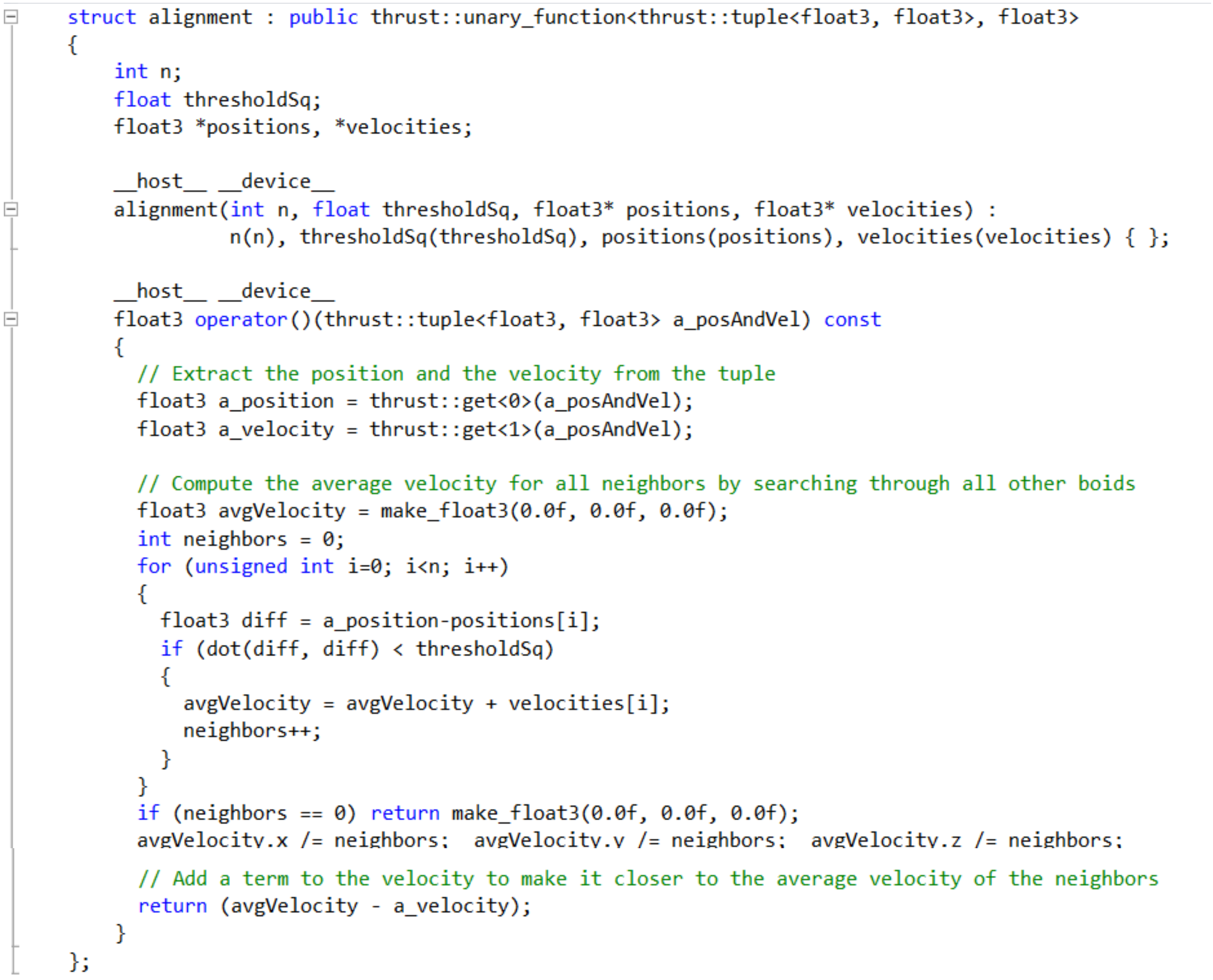
Separation Term



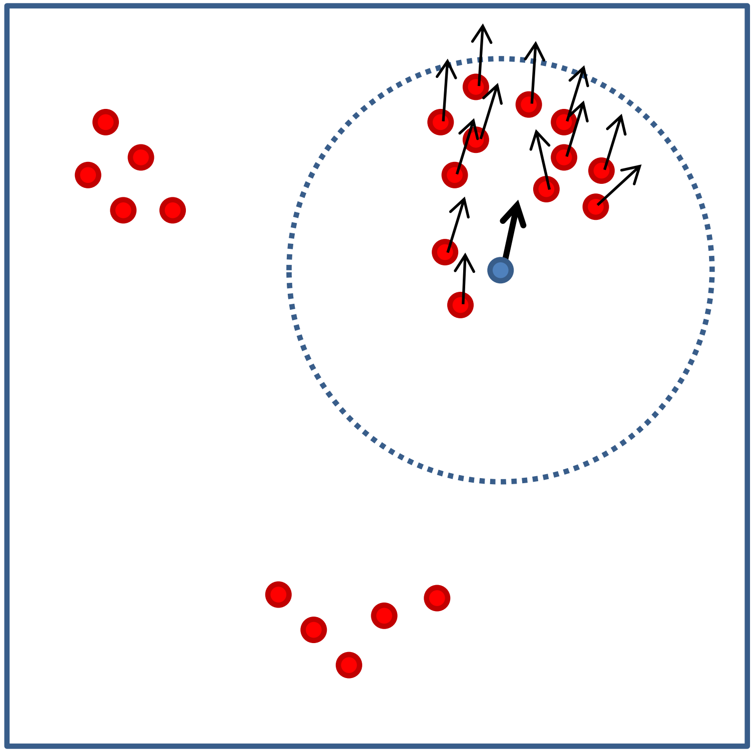
```
struct separation : public thrust::unary_function<float3, float3>
{
    int n;
    float thresholdSq;
    float3* positions;

    __host__ __device__
    separation(int n, float thresholdSq, float3* positions) : n(n), thresholdSq(thresholdSq), positions(positions) { };

    __host__ __device__
    float3 operator()(float3 a_position) const
    {
      // Add a term to the velocity pointed away from each neighbor that is too close
      float3 repel = make_float3(0.0f, 0.0f, 0.0f);
      for (unsigned int i=0; i<n; i++)
      {
        float3 diff = a_position-positions[i];
        if ((dot(diff, diff) < thresholdSq) && (dot(diff, diff) > NEAR_ZERO))
          repel = repel + normalize(diff);
      }
      if (dot(repel, repel) < NEAR_ZERO) return make_float3(0.0f, 0.0f, 0.0f);
      return normalize(repel);
    }
};
```



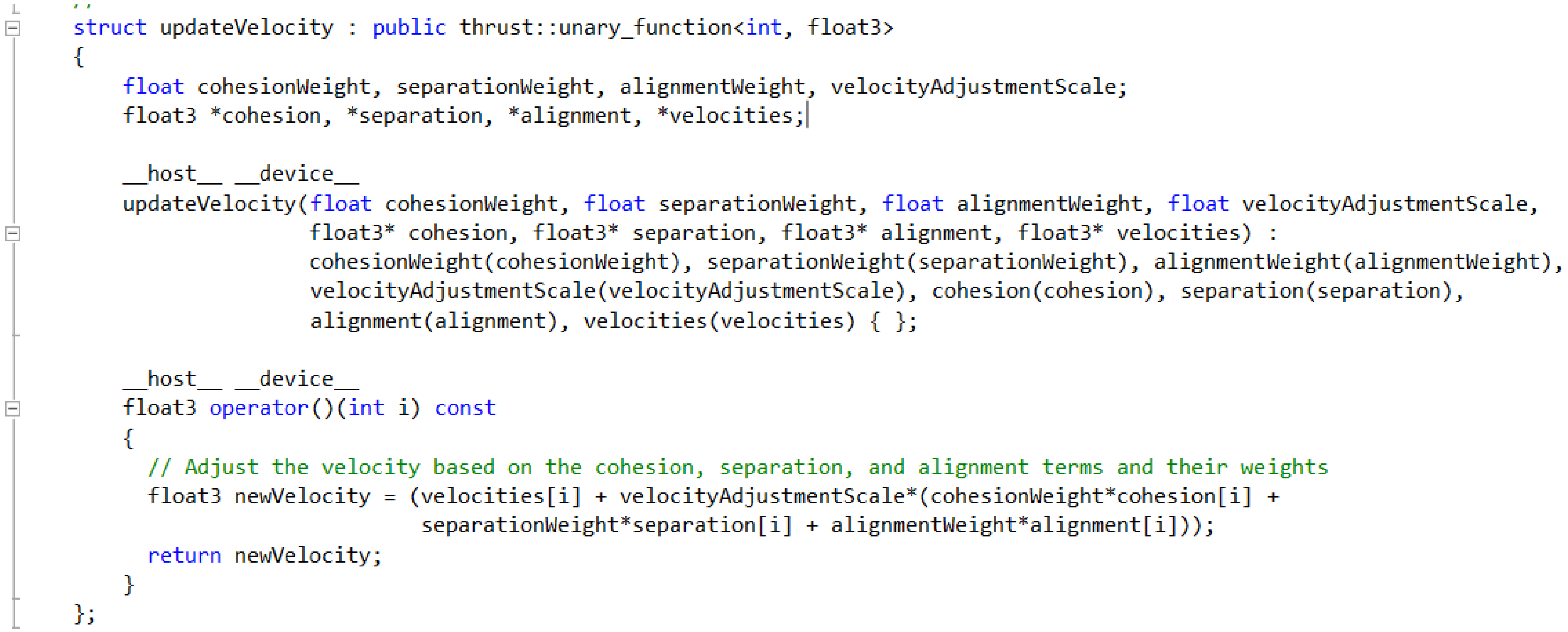
Alignment Term



```
struct alignment : public thrust::unary_function<thrust::tuple<float3, float3>, float3>
{
    int n;
    float thresholdSq;
    float3 *positions, *velocities;

    __host__ __device__
    alignment(int n, float thresholdSq, float3* positions, float3* velocities) :
            n(n), thresholdSq(thresholdSq), positions(positions), velocities(velocities) { };

    __host__ __device__
    float3 operator()(thrust::tuple<float3, float3> a_posAndVel) const
    {
      // Extract the position and the velocity from the tuple
      float3 a_position = thrust::get<0>(a_posAndVel);
      float3 a_velocity = thrust::get<1>(a_posAndVel);

      // Compute the average velocity for all neighbors by searching through all other boids
      float3 avgVelocity = make_float3(0.0f, 0.0f, 0.0f);
      int neighbors = 0;
      for (unsigned int i=0; i<n; i++)
      {
        float3 diff = a_position-positions[i];
        if (dot(diff, diff) < thresholdSq)
        {
          avgVelocity = avgVelocity + velocities[i];
          neighbors++;
        }
      }
      if (neighbors == 0) return make_float3(0.0f, 0.0f, 0.0f);
      avgVelocity.x /= neighbors;  avgVelocity.y /= neighbors;  avgVelocity.z /= neighbors;

      // Add a term to the velocity to make it closer to the average velocity of the neighbors
      return (avgVelocity - a_velocity);
    }
};
```



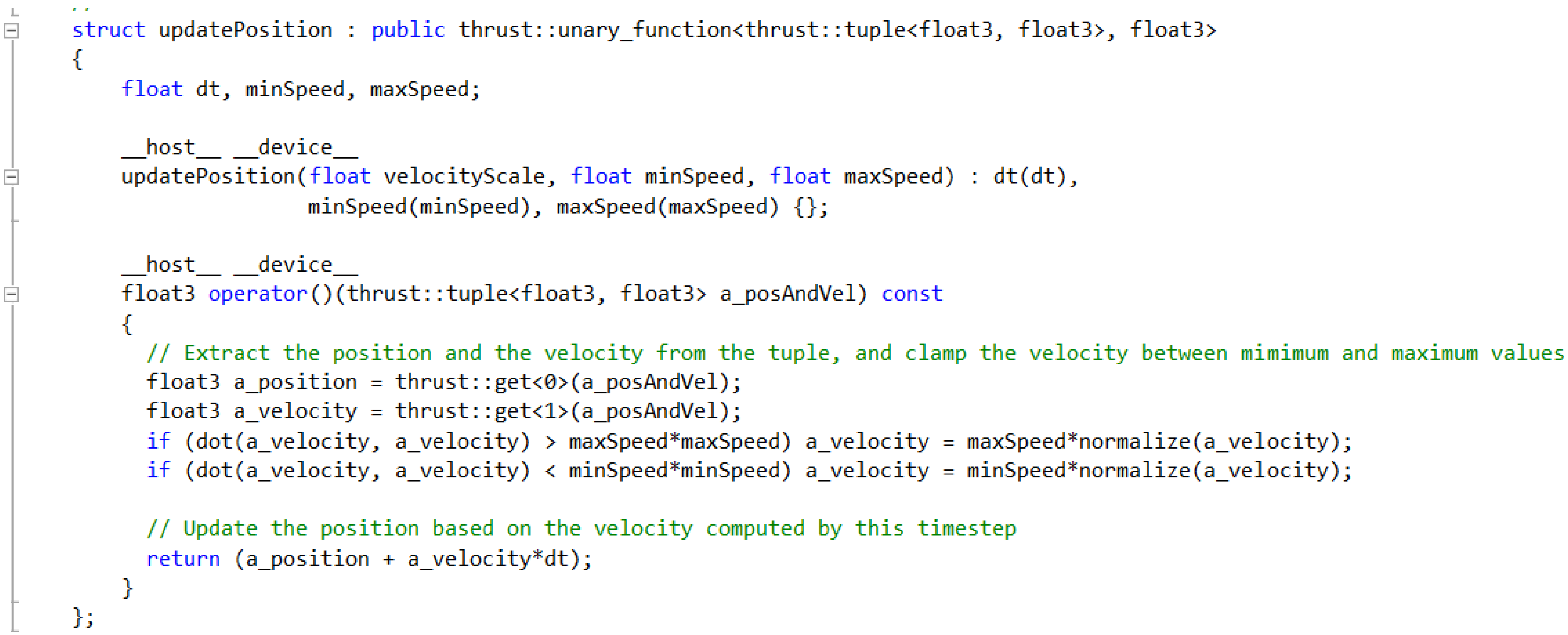
Updating Velocities



```
struct updateVelocity : public thrust::unary_function<int, float3>
{
    float cohesionWeight, separationWeight, alignmentWeight, velocityAdjustmentScale;
    float3 *cohesion, *separation, *alignment, *velocities;

    __host__ __device__
    updateVelocity(float cohesionWeight, float separationWeight, float alignmentWeight, float velocityAdjustmentScale,
                   float3* cohesion, float3* separation, float3* alignment, float3* velocities) :
                   cohesionWeight(cohesionWeight), separationWeight(separationWeight), alignmentWeight(alignmentWeight),
                   velocityAdjustmentScale(velocityAdjustmentScale), cohesion(cohesion), separation(separation),
                   alignment(alignment), velocities(velocities) { };

    __host__ __device__
    float3 operator()(int i) const
    {
      // Adjust the velocity based on the cohesion, separation, and alignment terms and their weights
      float3 newVelocity = (velocities[i] + velocityAdjustmentScale*(cohesionWeight*cohesion[i] +
                            separationWeight*separation[i] + alignmentWeight*alignment[i]));
      return newVelocity;
    }
};
```

$$v_{t+1} = v_t + w_v\left(w_c c + w_s s + w_a a\right)$$

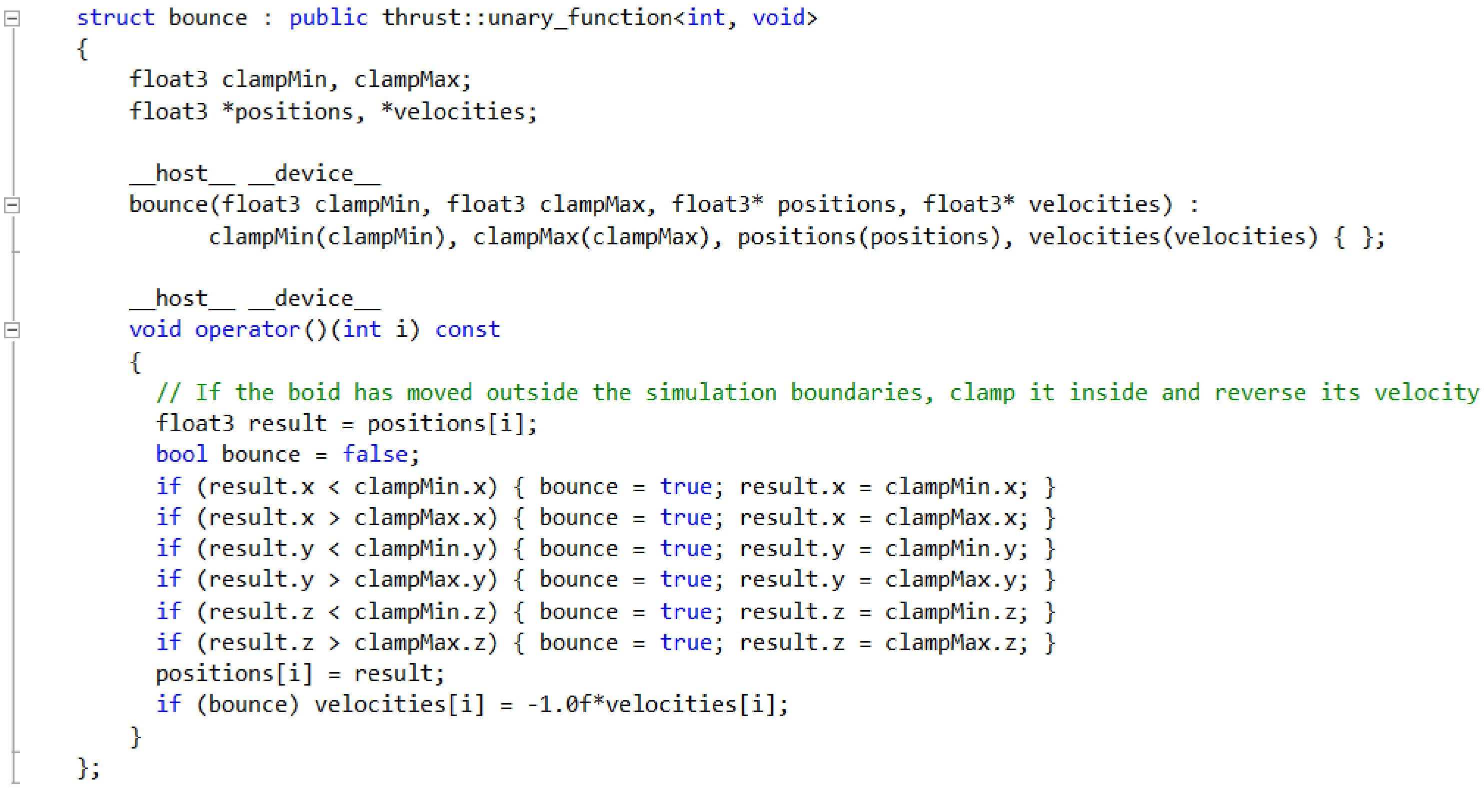
Updating Positions



```
struct updatePosition : public thrust::unary_function<thrust::tuple<float3, float3>, float3>
{
    float dt, minSpeed, maxSpeed;

    __host__ __device__
    updatePosition(float velocityScale, float minSpeed, float maxSpeed) : dt(dt),
                  minSpeed(minSpeed), maxSpeed(maxSpeed) {};

    __host__ __device__
    float3 operator()(thrust::tuple<float3, float3> a_posAndVel) const
    {
      // Extract the position and the velocity from the tuple, and clamp the velocity between mimimum and maximum values
      float3 a_position = thrust::get<0>(a_posAndVel);
      float3 a_velocity = thrust::get<1>(a_posAndVel);
      if (dot(a_velocity, a_velocity) > maxSpeed*maxSpeed) a_velocity = maxSpeed*normalize(a_velocity);
      if (dot(a_velocity, a_velocity) < minSpeed*minSpeed) a_velocity = minSpeed*normalize(a_velocity);

      // Update the position based on the velocity computed by this timestep
      return (a_position + a_velocity*dt);
    }
};
```

$$x_{t+1} = x_t + v_t \Delta t$$

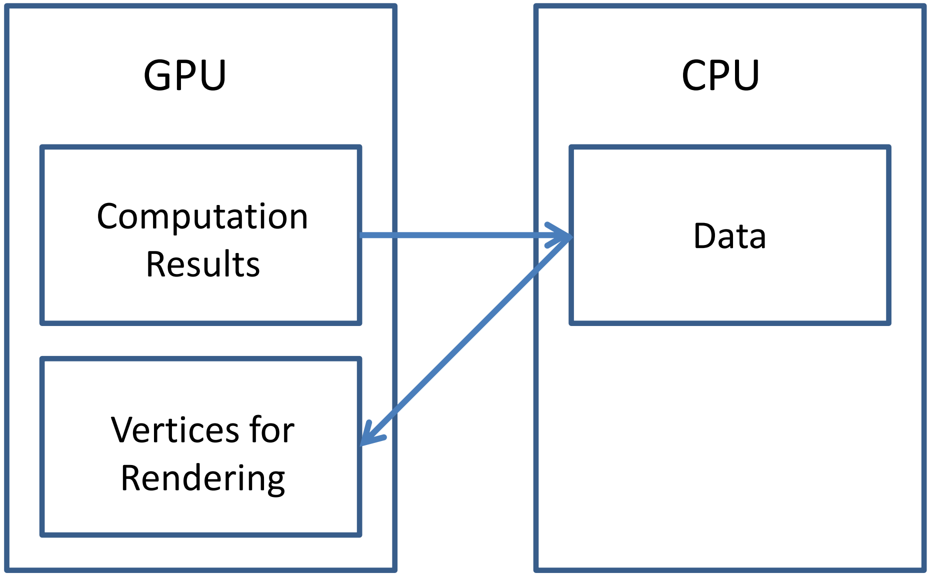
Bouncing off boundaries



```
struct bounce : public thrust::unary_function<int, void>
{
    float3 clampMin, clampMax;
    float3 *positions, *velocities;

    __host__ __device__
    bounce(float3 clampMin, float3 clampMax, float3* positions, float3* velocities) :
          clampMin(clampMin), clampMax(clampMax), positions(positions), velocities(velocities) { };

    __host__ __device__
    void operator()(int i) const
    {
      // If the boid has moved outside the simulation boundaries, clamp it inside and reverse its velocity
      float3 result = positions[i];
      bool bounce = false;
      if (result.x < clampMin.x) { bounce = true; result.x = clampMin.x; }
      if (result.x > clampMax.x) { bounce = true; result.x = clampMax.x; }
      if (result.y < clampMin.y) { bounce = true; result.y = clampMin.y; }
      if (result.y > clampMax.y) { bounce = true; result.y = clampMax.y; }
      if (result.z < clampMin.z) { bounce = true; result.z = clampMin.z; }
      if (result.z > clampMax.z) { bounce = true; result.z = clampMax.z; }
      positions[i] = result;
      if (bounce) velocities[i] = -1.0f*velocities[i];
    }
};
```

Interop

- Without interop, separate memory is used on the GPU for computation results and for rendering, and data transfer goes through the CPU
- With interop, a shared region of memory on the GPU is used both for computation and for rendering, eliminating the slow GPU-CPU data transfers



GPU

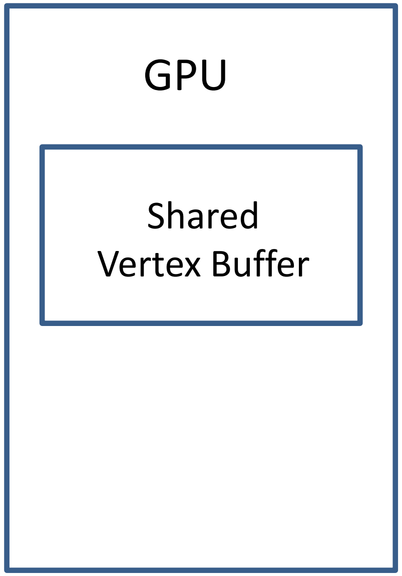
Computation Results

Vertices for Rendering

CPU

Data

Without interop



GPU

Shared Vertex Buffer

With interop

Your Free Ride Today

- The example codes we showed are independent of the location of data and execution
- It can be executed serially on CPU or parallel backends
- Debug on CPU during development; use parallel execution in "production"
- Extend to other languages and libraries
  - STL in C++
  - Copperhead in Python
  - SQL/LINQ for databases

Your Free Lunch Tomorrow

- The high-level parallel algorithms you write today will still work with new hardware in the future
- In fact, they will only get faster!
- The skills you learn in developing high-level parallel algorithms will still be applicable in the future even as computing technology improves

Conclusion

Think High-Level when Programming in Parallel