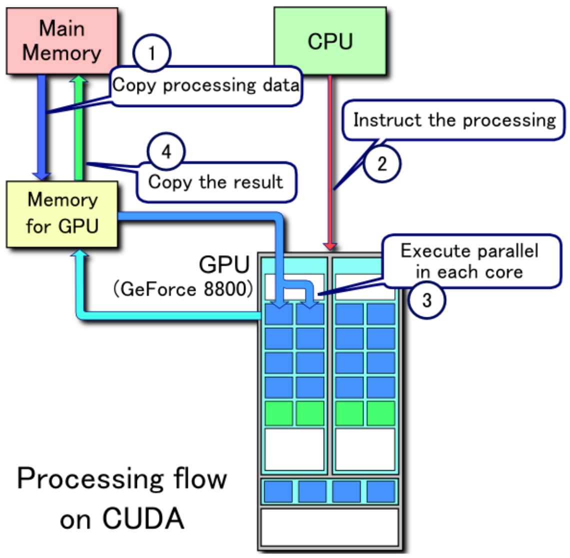
Lab 1 - Flocking

做这个实验之前，需要了解CUDA的结构。

CUDA Introduction

CUDA(Compute Unified Device Architecture)是老黄的NVIDIA在2006年发布的全新的运算集成技术，本质上CUDA是一种GPGPU，通过这个技术，用户可以利用NVIDIA的GPU进行图像处理、图形渲染，物理等计算。

如图为CUDA流水线：



1. 将主存的数据传入到GPU存储器
2. CPU指令指使GPU（这时需要驱动）
3. GPU平行计算
4. 将运算的结果从GPU存储器传回到主存

CUDA编程时控制这一流水线的具体代码，代码的语法兼容C++，语法门槛不高。

所以，多个CUDA模块就可以组成一个高效的多线程运算器，所以GPU在处理大型计算是非常快的，而CPU无法做到那么快的速度：

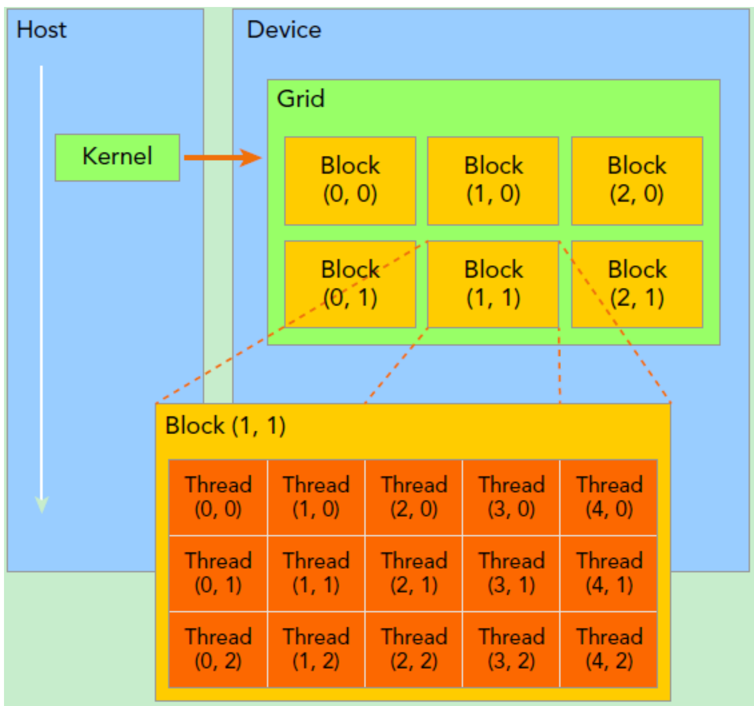
在CUDA中，host指代CPU及其内存，而用device指代GPU及其内存。CUDA程序中既包含host程序，又包含device程序，它们分别在CPU和GPU上运行。

将上述的流水线细节化，典型的CUDA程序的执行流程就出来了：

1. 分配host内存，并进行数据初始化；
2. 分配device内存，并从host将数据拷贝到device上；
3. 调用CUDA的核函数在device上完成指定的运算；
4. 将device上的运算结果拷贝到host上；
5. 释放device和host上分配的内存。

其中最重要的过程就是调用CUDA的核函数在device上完成指定的运算，这个时候kernel就登场了。kernel是在device上线程中并行执行的函数，核函数用 `__global__` 符号声明，在调用时需要用 `<<<grid, block>>>` 来指定kernel要执行的线程数量，在CUDA中，每一个线程都要执行核函数，并且每个线程会分配一个唯一的线程号thread ID，这个ID值可以通过核函数的内置变量 `threadIdx` 来获得。

要想学会如何调用核函数，那么需要对CUDA的结构了如指掌。在执行流水线的过程时，GPU和CPU的交互如图：



首先GPU上很多并行化的轻量级线程。kernel在device上执行时实际上是启动很多线程，一个kernel所启动的所有线程称为一个**网格**（grid），同一个网格上的线程共享相同的全局内存空间，grid是线程结构的第一层次，而网格又可以分为很多**线程块**（block），一个线程块里面包含很多线程，这是第二个层次。线程两层组织结构如下图所示，这是一个gird和block均为2-dim的线程组织。grid和block都是定义为 `dim3` 类型的变量， `dim3` 可以看成是包含三个无符号整数(x, y, z)成员的结构体变量，在定义时，缺省值初始化为1。因此grid和block可以灵活地定义为1-dim，2-dim以及3-dim结构，对于图中结构（主要水平方向为x轴），定义的grid和block如下代码所示，kernel在调用时也必须通过执行配置 `<<<grid, block>>>` 来指定kernel所使用的线程数及结构。

```
dim3 grid(3, 2);
dim3 block(5, 3);
//执行核函数
kernel_fun<<< grid, block >>>(prams...);
```

有时候，我们要知道一个线程在blcok中的全局ID，此时就必须还要知道block的组织结构，这是通过线程的内置变量blockDim来获得。它获取线程块各个维度的大小。对于一个2-dim的block(D_x, D_y)，线程(x, y)的全局ID为$x + yD_x$；对于一个3-dim的block (D_x, D_y, D_z)，线程(x, y, z)的全局ID为$x + yD_x + zD_xD_y$；另外线程还有内置变量gridDim，用于获得网格块各个维度的大小。

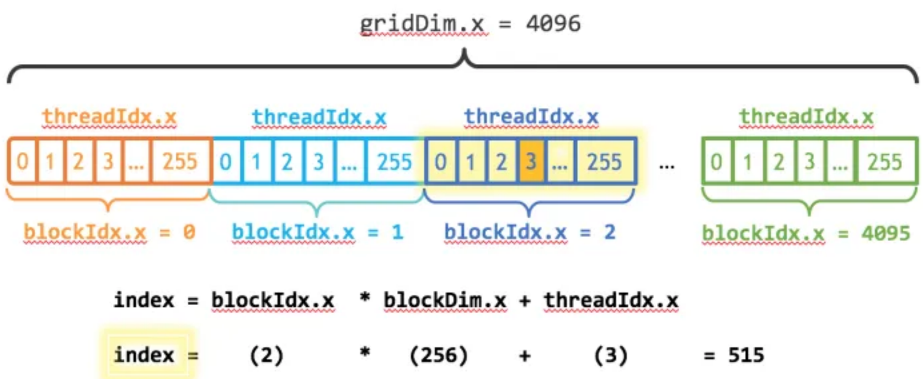
所以这种结构非常适合进行向量和矩阵的运算。比如要计算$N \times N$的矩阵之间的加法，我们可以使用16×16的线程块来来进行运算：

```
// Kernel定义
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}
int main()
{
    ...
    // Kernel 线程配置
    dim3 threadsPerBlock(16, 16);
    // 分块
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    // kernel调用
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

另外一个例子是执行两个N维向量的相加：

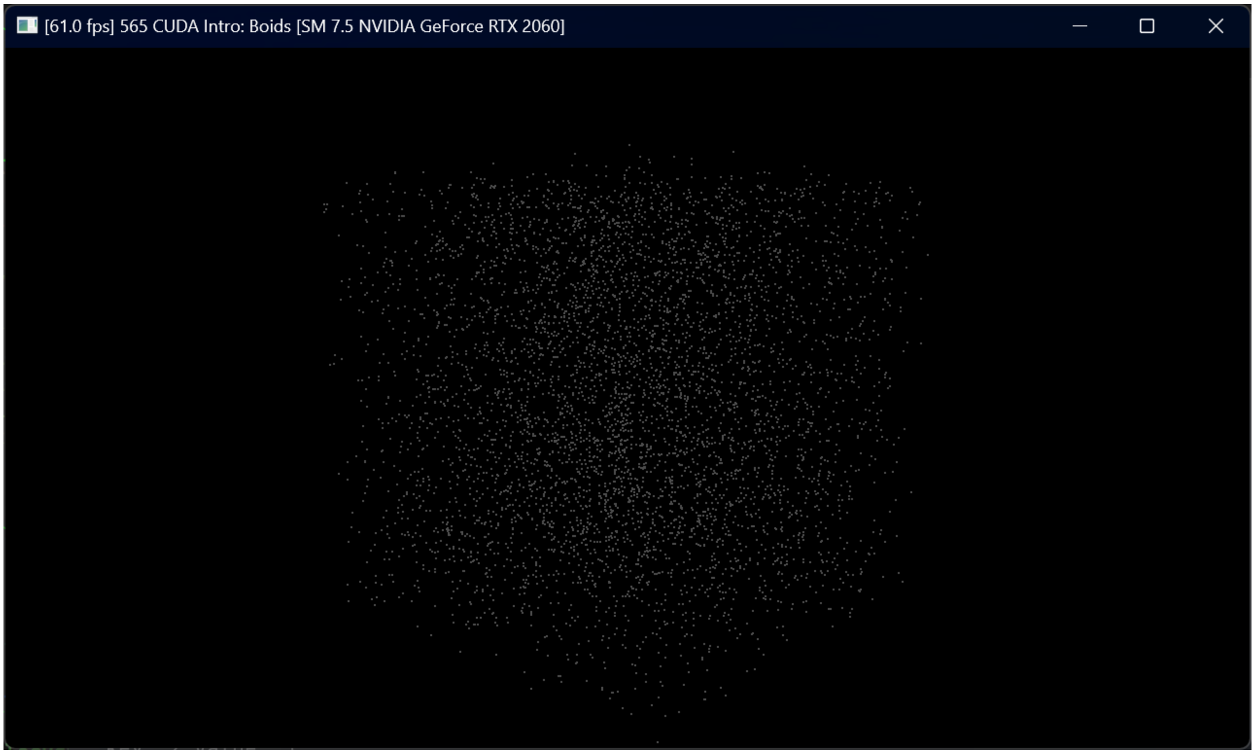
```
#include <iostream>
#include <cuda.h>

// 两个向量加法kernel，grid和block均为一维
__global__ void add(float* x, float * y, float* z, int n)
{
    // 获取全局索引
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    // 步长
    int stride = blockDim.x * gridDim.x;
    for (int i = index; i < n; i += stride)
    {
        z[i] = x[i] + y[i];
    }
}

int main()
{
    int N = 1 << 20;
    int nBytes = N * sizeof(float);
    // 申请host内存
    float *x, *y, *z;
    x = (float*)malloc(nBytes);
    y = (float*)malloc(nBytes);
    z = (float*)malloc(nBytes);

    // 初始化数据
    for (int i = 0; i < N; ++i)
    {
        x[i] = 10.0;
        y[i] = 20.0;
    }

    // 申请device内存
    float *d_x, *d_y, *d_z;
    cudaMalloc((void**)&d_x, nBytes);
    cudaMalloc((void**)&d_y, nBytes);
    cudaMalloc((void**)&d_z, nBytes);

    // 将host数据拷贝到device
    cudaMemcpy((void*)d_x, (void*)x, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy((void*)d_y, (void*)y, nBytes, cudaMemcpyHostToDevice);
    // 定义kernel的执行配置
    dim3 blockSize(256);
    // 简单算术题，如果要将1 << 20大小均分给256个块，那么一块处理多少？
    // 答案是[1 << 20 / blocksize] = 4096（向上取整），每一块处理4096个向量元素
    // 对于向上取整的公式为：(N + blockSize.x - 1) / blockSize.x
    dim3 gridSize((N + blockSize.x - 1) / blockSize.x);
    // 执行kernel
    add <<< gridSize, blockSize >>>(d_x, d_y, d_z, N);
```

```
    // 将device得到的结果拷贝到host
    cudaMemcpy((void*)z, (void*)d_z, nBytes, cudaMemcpyDeviceToHost);

    // 检查执行结果
    float maxError = 0.0;
    for (int i = 0; i < N; i++)
        maxError = fmax(maxError, fabs(z[i] - 30.0));
    std::cout << "最大误差: " << maxError << std::endl;

    // 释放device内存
    cudaFree(d_x);
    cudaFree(d_y);
    cudaFree(d_z);
    // 释放host内存
    free(x);
    free(y);
    free(z);

    return 0;
}
```



Part 1: Naive Boids Simulation

这一步我们将实现鸟群算法的基本实现，鸟群算法是一种模拟鸟类（Boids）大规模迁徙的算法。在这里，我们将用粒子模拟每个Boids的运动，目前还是静止的：



研究表明，Boids在大规模迁徙的时候遵循下面三个规则：

1. 聚集性：Boids会朝着质心的方向运动；
2. 分离性：Boids之间会保持一定的距离，不会靠得太近；
3. 一致性：Boids总体会尽可能往相同的方向和速度运动。

有如下伪代码解释这三个规则：

- Rule 1

```
function rule1(Boid boid)
    Vector perceived_center
    foreach Boid b:
        if b != boid and distance(b, boid) < rule1Distance then
            perceived_center += b.position
        endif
    end
    perceived_center /= number_of_neighbors
    return (perceived_center - boid.position) * rule1Scale
end
```

- Rule 2

```
function rule2(Boid boid)
    Vector c = 0
    foreach Boid b
        if b != boid and distance(b, boid) < rule2Distance then
            c -= (b.position - boid.position)
        endif
    end
    return c * rule2Scale
end
```

- Rule 3

```
function rule3(Boid boid)
    Vector perceived_velocity
    foreach Boid b
        if b != boid and distance(b, boid) < rule3Distance then
            perceived_velocity += b.velocity
        endif
    end
    perceived_velocity /= number_of_neighbors
    return perceived_velocity * rule3Scale
end
```

这三个Rule求出的值之和，就是速度的增量了。其中Rule 1和Rule 2描述了速度方向的增量，而Rule 3描述了速度大小的增量。于是可以写成C++代码，这一部分在device执行：

```
__device__ glm::vec3 computeVelocityChange(int N, int iSelf, const glm::vec3 *pos, const glm::vec3 *vel) {
  // Rule 1: boids fly towards their local perceived center of mass, which excludes themselves
  // Rule 2: boids try to stay a distance d away from each other
  // Rule 3: boids try to match the speed of surrounding boids
    glm::vec3 boidPos = pos[iSelf];
    glm::vec3 perceived_center(0.0f), perceived_velocity(0.0f), c(0.0f), ret(0.0f);
    int n1 = 0, n3 = 0;
    for(int i = 0; i < N; i++)
    {
        if (i == iSelf) continue;
        float distance = glm::distance(boidPos, pos[i]);
        if(distance < rule1Distance) perceived_center += pos[i], n1++;
        if(distance < rule2Distance) c -= pos[i] - boidPos;
        if(distance < rule3Distance) perceived_velocity += vel[i], n3++;
    }
    if (n1 > 0) perceived_center /= static_cast<float>(n1), ret += (perceived_center - boidPos)* rule1Scale;
    ret += c * rule2Scale;
    if (n3 > 0) perceived_velocity /= static_cast<float>(n3), ret += perceived_velocity * rule3Scale;
    return ret;
}
```

计算完速度之后，应当更新：

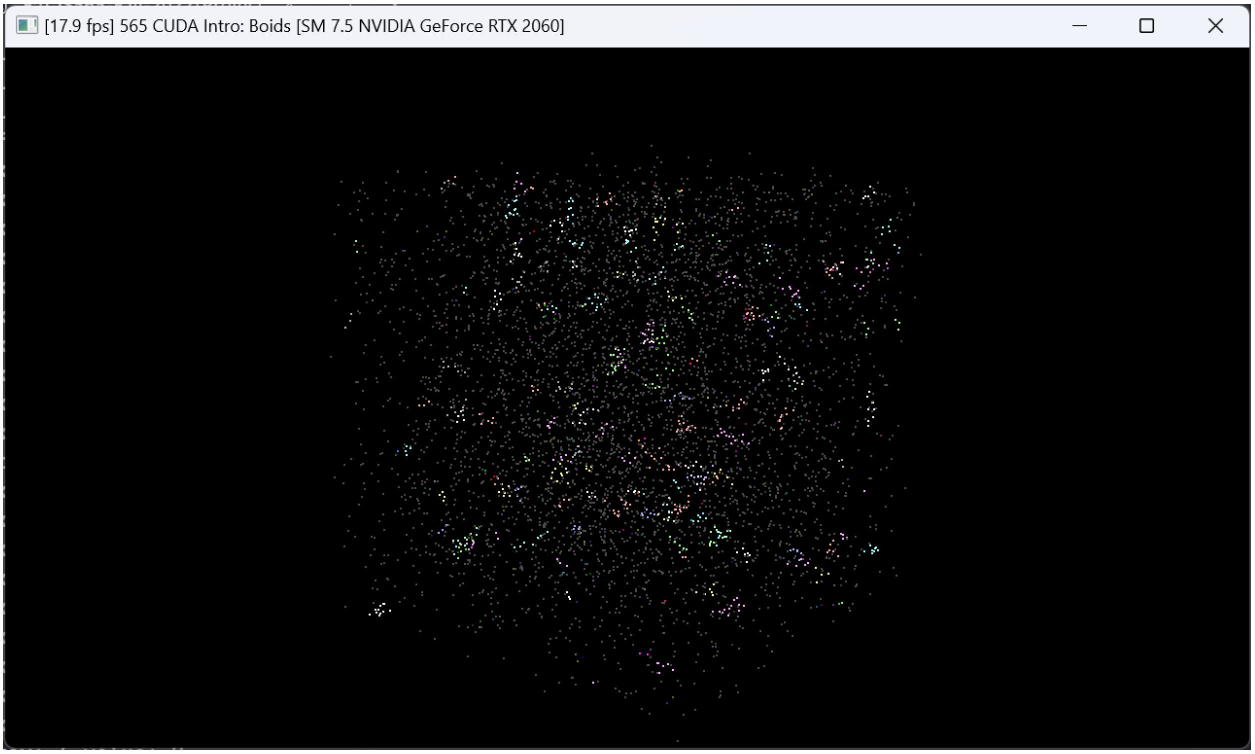
```
/**
* TODO-1.2 implement basic flocking
* For each of the `N` bodies, update its position based on its current velocity.
*/
__global__ void kernUpdateVelocityBruteForce(int N, glm::vec3 *pos,
  glm::vec3 *vel1, glm::vec3 *vel2) {
  // Compute a new velocity based on pos and vel1
  // Clamp the speed
  // Record the new velocity into vel2. Question: why NOT vel1?
    int index = (blockIdx.x * blockDim.x) + threadIdx.x;
    glm::vec3 deltaVel = computeVelocityChange(N, index, pos, vel1);
    glm::vec3 curVel = vel1[index];
    vel2[index] = glm::clamp(curVel + deltaVel, -maxSpeed, maxSpeed);
}
```

其中新的速度存储到vel2数组中，我们首先需要获得需要更新速度的粒子，用之前介绍的方法获得index，然后调用函数，最后限定范围在允许的范围内即可。

最后，在host中执行核函数。同时，为了防止Ping-pong效应，应当交换vel1和vel2，因为更新后的速度存储在vel2中：

```
void Boids::stepSimulationNaive(float dt) {
    // TODO-1.2 - use the kernels you wrote to step the simulation forward in time.
    dim3 blocksPerGrid((numObjects + blockSize - 1) /  blockSize); //分块，每块的大小是[n / blockSize]，向上取整，所以(numObjects + blockSize - 1
    kernUpdateVelocityBruteForce <<< blocksPerGrid, blockSize >>> (numObjects, dev_pos, dev_vel1, dev_vel2);
    kernUpdatePos <<< blocksPerGrid, blockSize >>> (numObjects, dt, dev_pos, dev_vel2);
    // TODO-1.2 ping-pong the velocity buffers
    std::swap(dev_vel1, dev_vel2);
}
```

运行结果：

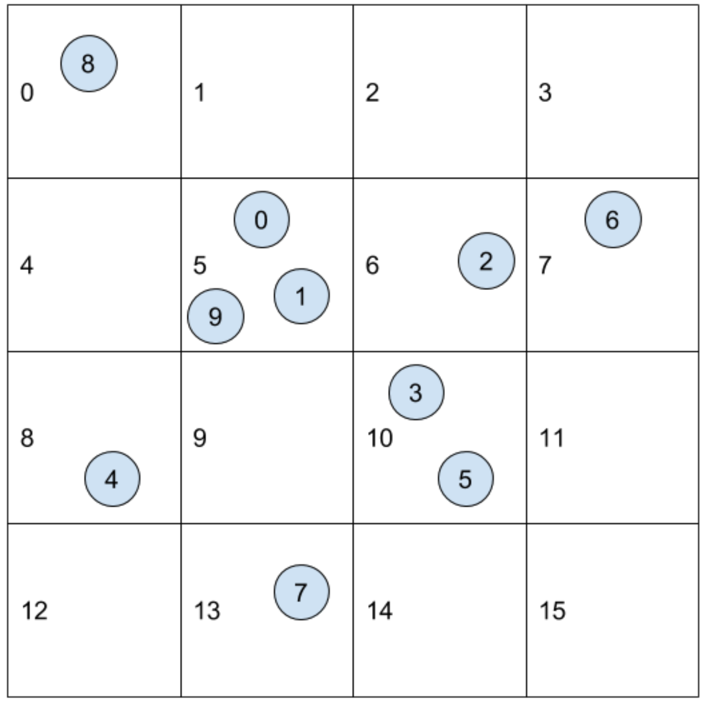


Part 2: 优化 - 时间优化

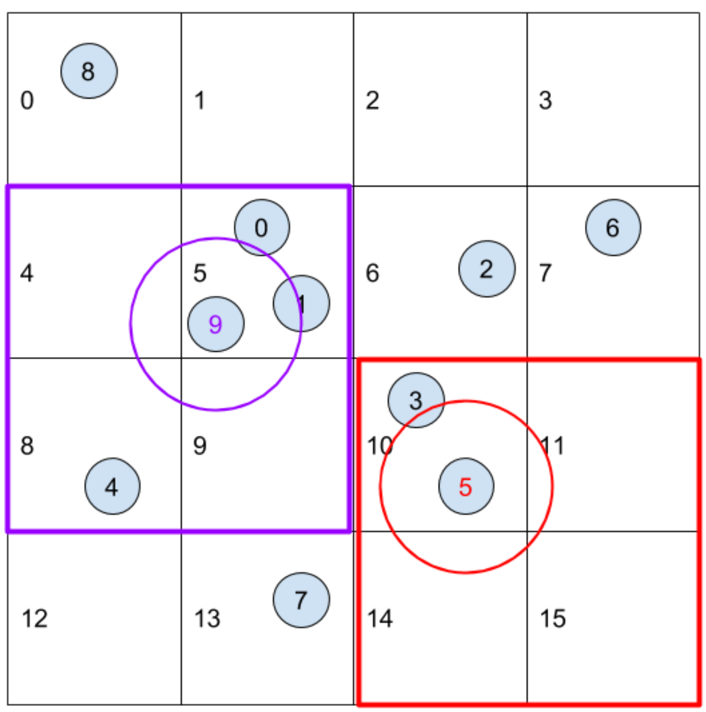
Part 1我们使用暴力法实现了Navie Boids Simulation，将每个Boid都强制和其他N-1个Boid进行比较，所以这个算法的复杂度为$O(n^2)$，效率非常低，我们的平均FPS也就10左右。所以我们需要对这个算法进行优化。在Part 1中，我们限定了Boids之间的邻域距离（Neighborhood distance)。可以按照邻域距离来优化算法。

一种简单的方法是采用划分格子，也就是构建Uniform Grid（单元格）：

如何构建Uniform Grid？传统上，我们一般在CPU上构建，通过遍历每个Boids，找到他们的Grid，然后将指向Boids的指针保存在表示Grid的可变数组中。但这方法不太适合将数据传输到GPU。因为首先GPU是没有可变数组的，然后因为CUDA多线程运算会导致并



如果单元格宽度是邻域距离的两倍，则每个Boid只需检查8个单元格中的其他Boid，或者2D情况下的4个boid，如图所示：

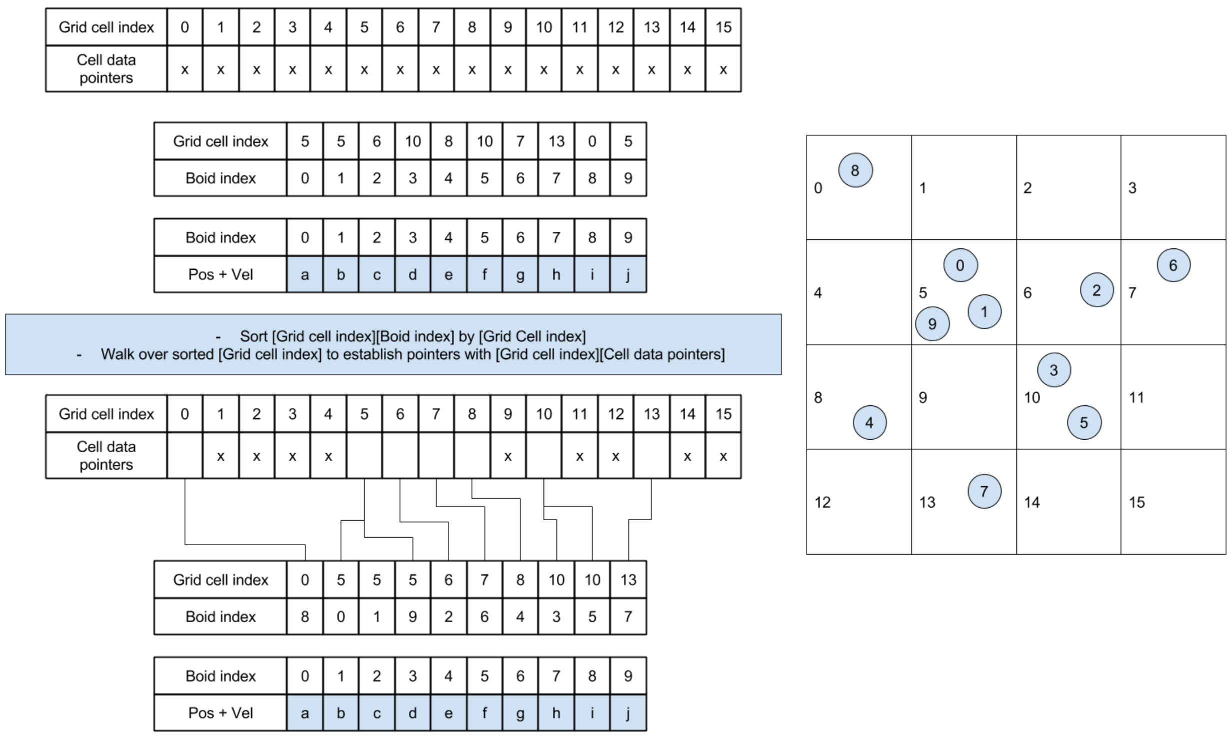


如何构建Uniform Grid？传统上，我们一般在CPU上构建，通过遍历每个Boids，找到他们的Grid，然后将指向Boids的指针保存在表示Grid的可变数组中。但这方法不太适合将数据传输到GPU。因为首先GPU是没有可变数组的，然后因为CUDA多线程运算会导致并

行迭代的时候数据之间可能会有线程竞争。所以这种方法不可取。

因此，我们可以利用排序来构建Uniform Grid。首先，定义一个数组Grid Array来记录每个Boids的Grid Index，然后根据Grid Index进行从小到大的排序。这样就可以确保在相同的Grid下指向Boid的指针在内存上是连续的。

之后，我们遍历已经排好序的Grid Array，并查看每个值。如果彼此的Grid Index不同，就说明他们不是在一个格子的（废话）。如图所示为这个数组的构建过程：



因此，我们需要构建两个数组，两个数组的Index都是Grid，但是值分别是指针和Boid Index。

需要注意的是，我们在GPU是无法用STL的sort函数进行排序的，我们可以使用GPU自带的Thrust库的sort函数来排序，下面是使用Thrust按照key的大小从小到大排序的示例：

```
void Boids::unitTest() {
  // LOOK-1.2 Feel free to write additional tests here.

  // test unstable sort
  int *dev_intKeys;
  int *dev_intValues;
  int N = 10;

  std::unique_ptr<int[]>intKeys{ new int[N] };
  std::unique_ptr<int[]>intValues{ new int[N] };

  intKeys[0] = 0; intValues[0] = 0;
  intKeys[1] = 1; intValues[1] = 1;
  intKeys[2] = 0; intValues[2] = 2;
  intKeys[3] = 3; intValues[3] = 3;
  intKeys[4] = 0; intValues[4] = 4;
  intKeys[5] = 2; intValues[5] = 5;
  intKeys[6] = 2; intValues[6] = 6;
  intKeys[7] = 0; intValues[7] = 7;
  intKeys[8] = 5; intValues[8] = 8;
  intKeys[9] = 6; intValues[9] = 9;

  cudaMalloc((void**)&dev_intKeys, N * sizeof(int));
  checkCUDAErrorWithLine("cudaMalloc dev_intKeys failed!");
```

```
  cudaMalloc((void**)&dev_intValues, N * sizeof(int));
  checkCUDAErrorWithLine("cudaMalloc dev_intValues failed!");

  dim3 fullBlocksPerGrid((N + blockSize - 1) / blockSize);

  std::cout << "before unstable sort: " << std::endl;
  for (int i = 0; i < N; i++) {
    std::cout << "  key: " << intKeys[i];
    std::cout << " value: " << intValues[i] << std::endl;
  }

  // How to copy data to the GPU
  cudaMemcpy(dev_intKeys, intKeys.get(), sizeof(int) * N, cudaMemcpyHostToDevice);
  cudaMemcpy(dev_intValues, intValues.get(), sizeof(int) * N, cudaMemcpyHostToDevice);

  // Wrap device vectors in thrust iterators for use with thrust.
  thrust::device_ptr<int> dev_thrust_keys(dev_intKeys);
  thrust::device_ptr<int> dev_thrust_values(dev_intValues);
  // LOOK-2.1 Example for using thrust::sort_by_key
  thrust::sort_by_key(dev_thrust_keys, dev_thrust_keys + N, dev_thrust_values);
  // 这里可以看出，用thrust排序，必须先定义thrust指针指向要排序的数组，然后用这个指针排序

  // How to copy data back to the CPU side from the GPU
  cudaMemcpy(intKeys.get(), dev_intKeys, sizeof(int) * N, cudaMemcpyDeviceToHost);
  cudaMemcpy(intValues.get(), dev_intValues, sizeof(int) * N, cudaMemcpyDeviceToHost);
  checkCUDAErrorWithLine("memcpy back failed!");

  std::cout << "after unstable sort: " << std::endl;
  for (int i = 0; i < N; i++) {
    std::cout << "  key: " << intKeys[i];
    std::cout << " value: " << intValues[i] << std::endl;
  }

  // cleanup
  cudaFree(dev_intKeys);
  cudaFree(dev_intValues);
  checkCUDAErrorWithLine("cudaFree failed!");
  return;
}
```

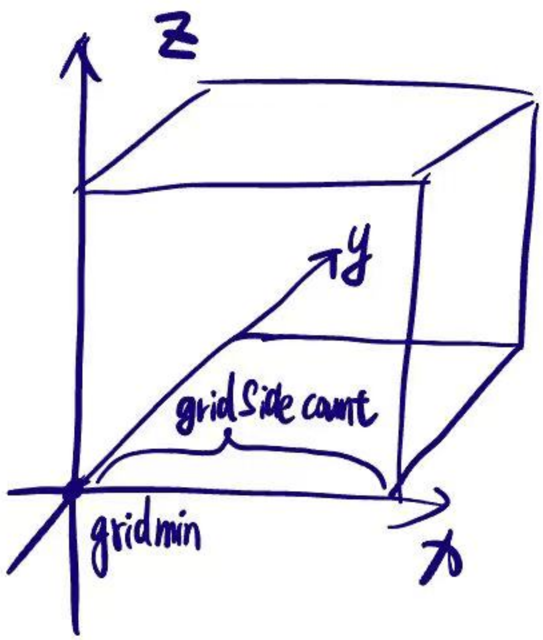
下面是控制台的输出：

```
file shaders/boid.vert.glsl loaded
file shaders/boid.geom.glsl loaded
file shaders/boid.frag.glsl loaded
before unstable sort:
  key: 0 value: 0
  key: 1 value: 1
  key: 0 value: 2
  key: 3 value: 3
  key: 0 value: 4
  key: 2 value: 5
  key: 2 value: 6
  key: 0 value: 7
  key: 5 value: 8
  key: 6 value: 9
after unstable sort:
  key: 0 value: 0
  key: 0 value: 2
  key: 0 value: 4
  key: 0 value: 7
  key: 1 value: 1
  key: 2 value: 5
  key: 2 value: 6
  key: 3 value: 3
  key: 5 value: 8
  key: 6 value: 9
```

根据代码的信息，可以知道如下几个全局变量的信息：

```
gridCellWidth - 所有格子的个数（展开成一维数组，所以有width的含义）
gridMinimum - 作为格子三维坐标系的原点，为了后续以它为原点计算格子的坐标
gridInverseCellWidth - 所有格子个数的倒数
gridSideCount - 格子所围成的三维立方体中的边长，因此有gridCellWidth = gridSideCount^3
dev_particleArrayIndices - [Grid cell index][Boid Index]中的Boid Index数组，为key
```

```
dev_particleGridIndices - [Grid cell index][Boid Index]中的Grid cell index数组，为value
dev_gridCellStartIndices - 键为GridIndex，值为当前Grid的第一个Boid Index
dev_gridCellEndIndices - 键为GridIndex，值为当前Grid的最后一个Boid Index
```



我们发现最后四个变量是数组，所以需要利用cudaMalloc函数申请内存空间，所以首先在Boids::initSimulation中申请内存：

```
/**
 * Initialize memory, update some globals
 */
void Boids::initSimulation(int N) {
    ......
    // TODO-2.1 TODO-2.3 - Allocate additional buffers here.
    cudaMalloc((void **)&dev_particleArrayIndices, N * sizeof(int));
    checkCUDAErrorWithLine("Malloc dev_particleArrayIndices failed!\n");

    cudaMalloc((void **)&dev_particleGridIndices, N * sizeof(int));
    checkCUDAErrorWithLine("Malloc dev_particleGridIndices failed!\n");

    cudaMalloc((void **)&dev_gridCellStartIndices, gridCellCount * sizeof(int));
    checkCUDAErrorWithLine("Malloc dev_gridCellStartIndices failed!\n");

    cudaMalloc((void **)&dev_gridCellEndIndices, gridCellCount * sizeof(int));
    checkCUDAErrorWithLine("Malloc dev_gridCellEndIndices failed!\n");

    dev_thrust_particleArrayIndices = thrust::device_ptr<int>(dev_particleArrayIndices);
    dev_thrust_particleGridIndices = thrust::device_ptr<int>(dev_particleGridIndices);

    cudaDeviceSynchronize();
}
```

然后首先计算Index，即计算dev_particleArrayIndices和dev_particleGridIndices数组：

```
__device__ int gridIndex3Dto1D(int x, int y, int z, int gridResolution) {
    return x + y * gridResolution + z * gridResolution * gridResolution;
}
```

```
// indices = dev_particleArrayIndices
// gridIndices = dev_particleGridIndices
__global__ void kernComputeIndices(int N, int gridResolution,
                                   glm::vec3 gridMin, float inverseCellWidth,
                                   glm::vec3 *pos, int *indices, int *gridIndices) {
    // TODO-2.1
    // - Label each boid with the index of its grid cell.
    // - Set up a parallel array of integer indices as pointers to the actual
    //   boid data in pos and vel1/vel2
    int boidIndex = blockIdx.x * blockDim.x + threadIdx.x; // boid index
    if (boidIndex >= N) return;
    glm::vec3 offset = pos[boidIndex] - gridMin;
    float xIndex = offset.x * inverseCellWidth;
    float yIndex = offset.y * inverseCellWidth;
    float zIndex = offset.z * inverseCellWidth;
    int gridIndex = gridIndex3Dto1D(xIndex, yIndex, zIndex, gridResolution);
    // key
    indices[boidIndex] = boidIndex;
    // value
    gridIndices[boidIndex] = gridIndex;
}
```

可以理解为，CUDA没有类似map的这样的键值对，所以为了引入键值对，我们需要用两个数组dev_particleArrayIndices和dev_particleGridIndices，其中前面的数组值和下标都是相等的，而后面的就是gridIndex。这样可以好利用dev_particleGridIndices排序。

另外，我们计算了offset，这个向量就是以gridMin为原点，当前boid的坐标，通过gridIndex3Dto1D可以转换成gridIndex。

接下来计算dev_gridCellStartIndices和dev_gridCellEndIndices数组：

```
__global__ void kernIdentifyCellStartEnd(int N, int *particleGridIndices,
                                         int *gridCellStartIndices, int *gridCellEndIndices) {
    // TODO-2.1
    // Identify the start point of each cell in the gridIndices array.
    // This is basically a parallel unrolling of a loop that goes
    // "this index doesn't match the one before it, must be a new cell!"

    // We must do this after sort
    int boidIndex = (blockIdx.x * blockDim.x) + threadIdx.x;
    if (boidIndex >= N) return;
    int gridIndex = particleGridIndices[boidIndex];
    if (boidIndex > 0) {
        int preGridIndex = particleGridIndices[boidIndex - 1];
        if (preGridIndex != gridIndex) {
            gridCellStartIndices[gridIndex] = boidIndex;
            gridCellEndIndices[preGridIndex] = boidIndex;
        }
        if (preGridIndex == N - 1)
            gridCellEndIndices[gridIndex] = boidIndex;
    } else {
        gridCellStartIndices[gridIndex] = boidIndex;
    }
}
```

因为核函数是对每个block的Thread进行一次核函数，所以对于计算起始点和终止点，不需要循环。

该函数被设计为使用块和线程的网格，其中线程的总数等于N。每个线程负责识别单个粒子单元格的起始和结束索引。

该函数通过迭代粒子GridIndices数组并将每个索引与其前面的索引进行比较来工作。如果索引不同，则当前索引是新单元格的起点，上一个索引是上一个单元格的终点。该函数相应地更新gridCellStartIndices和gridCellEndIndices数组。

请注意，在调用此函数之前，必须对particleGridIndices数组进行排序，以便将同一单元格中的粒子分组在一起。

接下来是计算的主体部分，核心内容不变，还是按照前面说的三个规则，但是需要进行格子的识别和划分，这样大大减少了计算量：

```
__global__ void kernUpdateVelNeighborSearchScattered(
    int N, int gridResolution, glm::vec3 gridMin,
    float inverseCellWidth, float cellWidth,
    int *gridCellStartIndices, int *gridCellEndIndices,
```

```
    int *particleArrayIndices,
    glm::vec3 *pos, glm::vec3 *vel1, glm::vec3 *vel2) {
    // TODO-2.1 - Update a boid's velocity using the uniform grid to reduce
    // the number of boids that need to be checked.
    // - Identify the grid cell that this particle is in
    // - Identify which cells may contain neighbors. This isn't always 8.
    // - For each cell, read the start/end indices in the boid pointer array.
    // - Access each boid in the cell and compute velocity change from
    //   the boids rules, if this boid is within the neighborhood distance.
    // - Clamp the speed change before putting the new speed in vel2
    int boidIndex = blockDim.x * blockIdx.x + threadIdx.x;
    if (boidIndex >= N) return;
    glm::vec3 curPos = pos[boidIndex];
    glm::vec3 offset = curPos - gridMin;
    float xIndex = offset.x * inverseCellWidth;
    float yIndex = offset.y * inverseCellWidth;
    float zIndex = offset.z * inverseCellWidth;
    int gridIndex = gridIndex3Dto1D(xIndex, yIndex, zIndex, gridResolution);

    // find the bounding box of A GRID, 26个
    int X_MAX = imin(xIndex + 1, gridResolution - 1);
    int X_MIN = imax(xIndex - 1, 0);
    int Y_MAX = imin(yIndex + 1, gridResolution - 1);
    int Y_MIN = imax(yIndex - 1, 0);
    int Z_MAX = imin(zIndex + 1, gridResolution - 1);
    int Z_MIN = imax(zIndex - 1, 0);

    glm::vec3 perceived_center(0.0f), perceived_velocity(0.0f), c(0.0f), ret(0.0f);
    int n1 = 0, n3 = 0;

    for (int x = X_MIN; x <= X_MAX; x++) {
        for (int y = Y_MIN; y <= Y_MAX; y++) {
            for (int z = Z_MIN; z <= Z_MAX; z++) {
                int neighborGridIndex = gridIndex3Dto1D(x, y, z, gridResolution);
                int neighborGridStart = gridCellStartIndices[neighborGridIndex];
                int neighborGridEnd = gridCellEndIndices[neighborGridIndex];
                for (int i = neighborGridStart; i <= neighborGridEnd; i++) {
                    int this_boidIndex = particleArrayIndices[i];
                    if (this_boidIndex != boidIndex) {
                        float distance = glm::distance(pos[this_boidIndex], curPos);
                        if (distance < rule1Distance) perceived_center += pos[this_boidIndex], n1++;
                        if (distance < rule2Distance) c -= pos[this_boidIndex] - curPos;
                        if (distance < rule3Distance) perceived_velocity += vel1[this_boidIndex], n3++;
                    }
                }
            }
        }
    }
    if (n1 > 0) perceived_center /= static_cast<float>(n1), ret += (perceived_center - curPos) * rule1Scale;
    ret += c * rule2Scale;
    if (n3 > 0) perceived_velocity /= static_cast<float>(n3), ret += perceived_velocity * rule3Scale;

    vel2[boidIndex] = glm::clamp(vel1[boidIndex] + ret, -maxSpeed, maxSpeed);
}
```

该函数使用块和线程的网格，其中每个线程负责更新单个粒子的速度。该函数的工作流程如下：

- 确定该粒子所在的网格单元格
- 确定哪些单元格可能包含邻居。这并不总是8个单元格。所以需要在该网格的周围26个格子中判断。
- 对于每个单元格，读取指向粒子指针数组中的起始/结束索引。
- 访问单元格中的每个粒子，并根据其规则计算速度变化，如果该粒子在邻居距离内。
- 将速度变化限制在合理范围内，然后将新的速度放入vel2中。

该函数使用三重循环遍历网格中的所有单元格和粒子，并计算每个粒子的速度变化。在计算速度变化时，该函数根据三个规则计算每个粒子与其邻居的交互作用。然后，将速度变化应用于原始速度，并将结果限制在最大速度范围内，然后将更新后的速度存储到vel2数组中。

最后是优化方法的入口函数，main.cpp将调用该函数开始CUDA计算：

```
void Boids::stepSimulationScatteredGrid(float dt) {
    // TODO-2.1
    // Uniform Grid Neighbor search using Thrust sort.
    // In Parallel:
    // - label each particle with its array index as well as its grid index.
    //   Use 2x width grids.
    // - Unstable key sort using Thrust. A stable sort isn't necessary, but you
    //   are welcome to do a performance comparison.
    // - Naively unroll the loop for finding the start and end indices of each
    //   cell's data pointers in the array of boid indices
    // - Perform velocity updates using neighbor search
    // - Update positions
    // - Ping-pong buffers as needed
    dim3 blocksPerGridBoids((numObjects + blockSize - 1) / blockSize);
    kernComputeIndices<<<blocksPerGridBoids, blockSize>>>(numObjects, gridSideCount, gridMinimum, gridInverseCellWidth, dev_pos, dev_partic
    thrust::sort_by_key(dev_thrust_particleGridIndices, dev_thrust_particleGridIndices + numObjects, dev_particleArrayIndices);

    dim3 blocksPerGridCells((gridCellCount + blockSize - 1) / blockSize);
    // 初始化dev_gridCellStartIndices和dev_gridCellEndIndices为-1
    kernResetIntBuffer<<<blocksPerGridCells, blockSize>>>(gridCellCount, dev_gridCellStartIndices, -1);
    kernResetIntBuffer<<<blocksPerGridCells, blockSize>>>(gridCellCount, dev_gridCellEndIndices, -1);
    kernIdentifyCellStartEnd<<<blocksPerGridCells, blockSize>>>(numObjects, dev_particleGridIndices, dev_gridCellStartIndices, dev_gridCell

    kernUpdateVelNeighborSearchScattered<<<blocksPerGridBoids, blockSize>>>(numObjects, gridSideCount, gridMinimum, gridInverseCellWidth, g
                                                                            dev_gridCellStartIndices, dev_gridCellEndIndices, dev_particleA
                                                                            dev_pos, dev_vel1, dev_vel2);
    kernUpdatePos<<<blocksPerGridBoids, blockSize>>>(numObjects, dt, dev_pos, dev_vel2);
    std::swap(dev_vel1, dev_vel2);
}
```

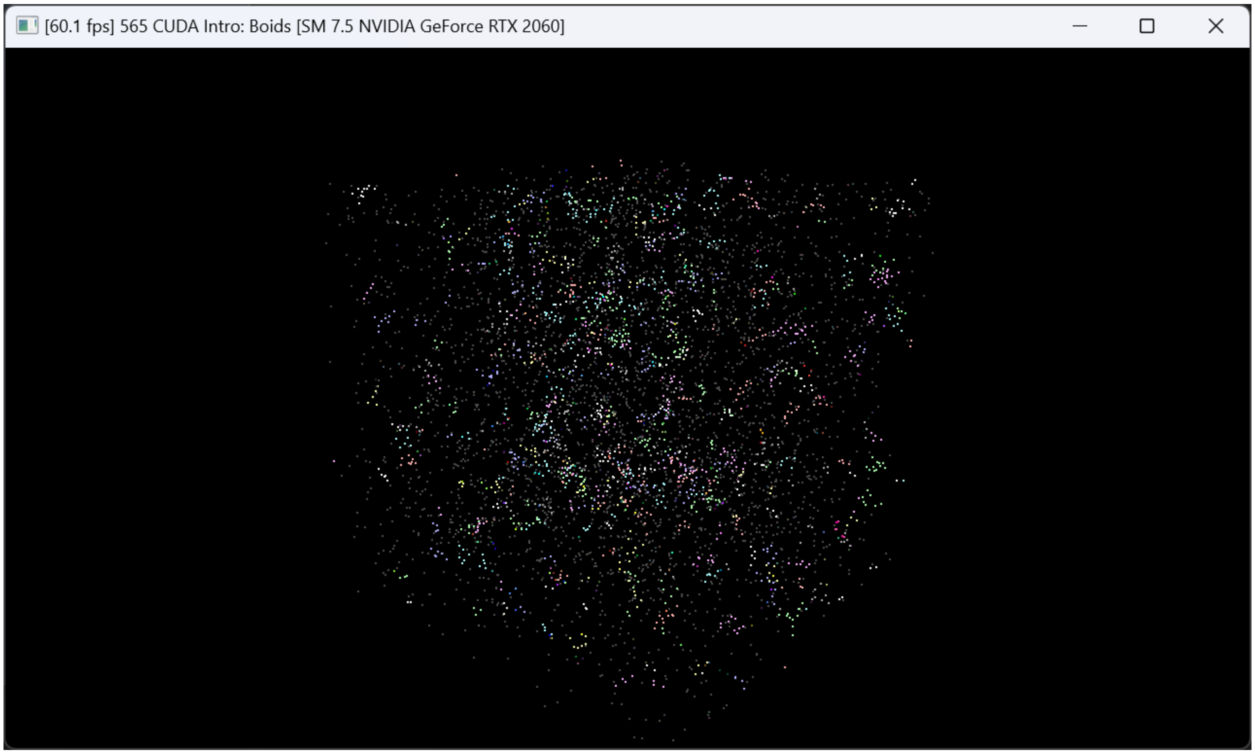
- 对于每个粒子，将其标记为其数组索引以及其网格索引。使用2倍的宽度网格。
- 使用Thrust进行不稳定的键排序。稳定排序不是必需的，但是你可以进行性能比较。
- 对于每个单元格，展开循环以查找数组中该单元格数据指针的起始和结束索引。
- 使用邻居搜索执行速度更新
- 更新位置
- 需要时交替使用缓冲区

该函数使用CUDA内核函数来实现上述步骤。首先，使用kernComputeIndices内核函数计算每个粒子的网格索引和数组索引。然后，使用Thrust排序按网格索引对粒子数组索引进行排序。接下来，使用kernIdentifyCellStartEnd内核函数确定每个单元格的起始和结束索引。最后，使用kernUpdateVelNeighborSearchScattered内核函数执行邻居搜索并更新速度，使用kernUpdatePos内核函数更新位置。

最后需要释放内存：

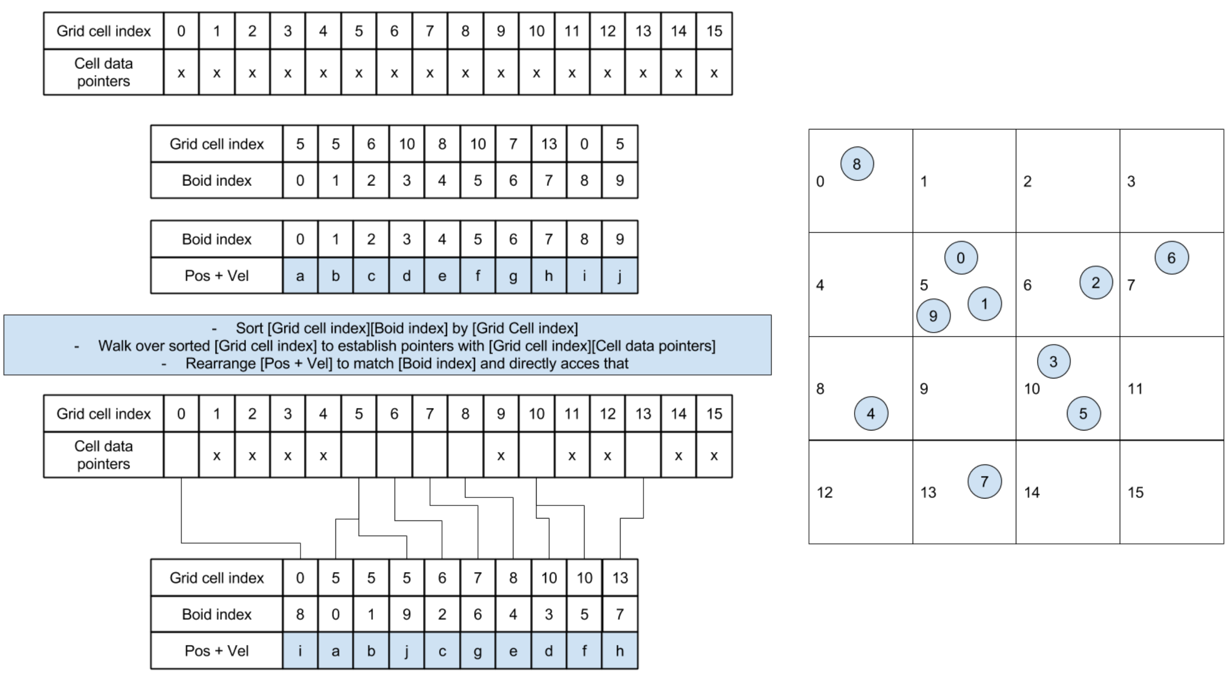
```
void Boids::endSimulation() {
    ......
    // TODO-2.1 TODO-2.3 - Free any additional buffers here.
    cudaFree(dev_particleArrayIndices);
    cudaFree(dev_particleGridIndices);
    cudaFree(dev_gridCellStartIndices);
    cudaFree(dev_gridCellEndIndices);
}
```

最后效果很不错，维持在60FPS：



Part 3: 进一步优化 - 内存空间优化

考虑2.1中概述的统一网格邻居搜索:在单个单元中指向boid的指针在内存中是连续的，但是boid数据本身(速度和位置)分散在各处。尝试重新安排boid数据本身，以便一个单元格中boid的所有速度和位置在内存中也是连续的，这样就可以直接使用dev_gridCellStartIndices和dev_gridCellEndIndices而不需要考虑dev_particleArrayIndices：



这种方法被称为“一致性网格”，其主要思想是重新排列粒子数组，使得同一单元格中的所有粒子数据在内存中是连续的，从而更好地利用了缓存。对于一致性网格邻居搜索，可以按照以下步骤重新排列粒子数组：

1. 将所有粒子按照它们在网格中的顺序排序，即先按照Z轴坐标排序，然后按照Y轴坐标排序，最后按照X轴坐标排序。这样可以保证同一单元格中的粒子在排列后是相邻的。
2. 创建新的缓冲区dev_sortedPos和dev_sortedVel，用于存储重新排列后的位置和速度数据。
3. 使用kernSortPosAndVel内核函数将位置和速度数据按照排序后的顺序存储到新的缓冲区中。

```
__global__ void kernSortPosAndVel(int N, int *particleArrayIndices, glm::vec3 *pos, glm::vec3 *sortedPos, glm::vec3 *vel, glm::vec3 *sorted
    int index = blockDim.x * blockIdx.x + threadIdx.x;
    if (index >= N) return;
    int boidIndex = particleArrayIndices[index];
    sortedPos[index] = pos[boidIndex];
    sortedVel[index] = vel[boidIndex];
}
```

所以先开辟dev_sortedPos和dev_sortedVel的内存空间

```
void Boids::initSimulation(int N) {
    ......
    // TODO-2.1 TODO-2.3 - Allocate additional buffers here.
    cudaMalloc((void **)&dev_particleArrayIndices, N * sizeof(int));
    checkCUDAErrorWithLine("Malloc dev_particleArrayIndices failed!\n");

    cudaMalloc((void **)&dev_particleGridIndices, N * sizeof(int));
    checkCUDAErrorWithLine("Malloc dev_particleGridIndices failed!\n");

    cudaMalloc((void **)&dev_gridCellStartIndices, gridCellCount * sizeof(int));
    checkCUDAErrorWithLine("Malloc dev_gridCellStartIndices failed!\n");

    cudaMalloc((void **)&dev_gridCellEndIndices, gridCellCount * sizeof(int));
    checkCUDAErrorWithLine("Malloc dev_gridCellEndIndices failed!\n");

    dev_thrust_particleArrayIndices = thrust::device_ptr<int>(dev_particleArrayIndices);
    dev_thrust_particleGridIndices = thrust::device_ptr<int>(dev_particleGridIndices);

    cudaMalloc((void **)&dev_sortedPos, N * sizeof(glm::vec3));
    checkCUDAErrorWithLine("Malloc dev_sortedPos failed!\n");

    cudaMalloc((void **)&dev_sortedVel, N * sizeof(glm::vec3));
    checkCUDAErrorWithLine("Malloc dev_sortedVel failed!\n");

    cudaDeviceSynchronize();
}
```

主体计算部分，我们不需要dev_particleArrayIndices，这样也可以获得邻近的boid，只需要知道i和pos就可以了：

```
__global__ void kernUpdateVelNeighborSearchCoherent(
    int N, int gridResolution, glm::vec3 gridMin,
    float inverseCellWidth, float cellWidth,
    int *gridCellStartIndices, int *gridCellEndIndices,
    glm::vec3 *pos, glm::vec3 *vel1, glm::vec3 *vel2) {
    // TODO-2.3 - This should be very similar to kernUpdateVelNeighborSearchScattered,
    // except with one less level of indirection.
    // This should expect gridCellStartIndices and gridCellEndIndices to refer
    // directly to pos and vel1.
    // - Identify the grid cell that this particle is in
    // - Identify which cells may contain neighbors. This isn't always 8.
    // - For each cell, read the start/end indices in the boid pointer array.
    //   DIFFERENCE: For best results, consider what order the cells should be
    //   checked in to maximize the memory benefits of reordering the boids data.
    // - Access each boid in the cell and compute velocity change from
    //   the boids rules, if this boid is within the neighborhood distance.
    // - Clamp the speed change before putting the new speed in vel2
    int boidIndex = blockDim.x * blockIdx.x + threadIdx.x;
    if (boidIndex >= N) return;
    glm::vec3 curPos = pos[boidIndex];
    glm::vec3 offset = curPos - gridMin;
    float xIndex = offset.x * inverseCellWidth;
    float yIndex = offset.y * inverseCellWidth;
```

```
    float zIndex = offset.z * inverseCellWidth;
    int gridIndex = gridIndex3Dto1D(xIndex, yIndex, zIndex, gridResolution);

    // find the bounding box of A GRID, 26个
    int X_MAX = imin(xIndex + 1, gridResolution - 1);
    int X_MIN = imax(xIndex - 1, 0);
    int Y_MAX = imin(yIndex + 1, gridResolution - 1);
    int Y_MIN = imax(yIndex - 1, 0);
    int Z_MAX = imin(zIndex + 1, gridResolution - 1);
    int Z_MIN = imax(zIndex - 1, 0);

    glm::vec3 perceived_center(0.0f), perceived_velocity(0.0f), c(0.0f), ret(0.0f);
    int n1 = 0, n3 = 0;

    for (int x = X_MIN; x <= X_MAX; x++) {
        for (int y = Y_MIN; y <= Y_MAX; y++) {
            for (int z = Z_MIN; z <= Z_MAX; z++) {
                int neighborGridIndex = gridIndex3Dto1D(x, y, z, gridResolution);
                int neighborGridStart = gridCellStartIndices[neighborGridIndex];
                int neighborGridEnd = gridCellEndIndices[neighborGridIndex];
                for (int i = neighborGridStart; i <= neighborGridEnd; i++) {
                    if (i != boidIndex) {
                        float distance = glm::distance(pos[i], curPos);
                        if (distance < rule1Distance) perceived_center += pos[i], n1++;
                        if (distance < rule2Distance) c -= pos[i] - curPos;
                        if (distance < rule3Distance) perceived_velocity += vel1[i], n3++;
                    }
                }
            }
        }
    }
    if (n1 > 0) perceived_center /= static_cast<float>(n1), ret += (perceived_center - curPos) * rule1Scale;
    ret += c * rule2Scale;
    if (n3 > 0) perceived_velocity /= static_cast<float>(n3), ret += perceived_velocity * rule3Scale;

    vel2[boidIndex] = glm::clamp(vel1[boidIndex] + ret, -maxSpeed, maxSpeed);
}
```

这是一个使用一致性网格进行邻居搜索的CUDA内核函数，用于更新粒子速度。与之前的内核函数kernUpdateVelNeighborSearchScattered相比，它不再使用dev_particleArrayIndices数组来查找粒子数据，而是直接使用dev_gridCellStartIndices和dev_gridCellEndIndices数组来查找同一单元格中的粒子数据。

该内核函数的主要步骤如下：

1. 计算当前粒子所在的网格索引gridIndex。
2. 根据当前粒子所在的网格索引，确定其周围可能存在邻居的网格索引范围。这里使用了一个优化，即只考虑距离当前粒子最近的27个网格，而不是所有的网格。
3. 对于每个可能存在邻居的网格，遍历该网格中的所有粒子，并根据粒子之间的距离计算速度变化ret，分别计算出三个规则（对应于Boids算法中的三个行为）的速度变化。
4. 将计算出的速度变化ret与当前粒子的速度vel1相加，并通过glm::clamp函数将其限制在最大速度maxSpeed和最小速度-maxSpeed之间，得到新的速度vel2。

需要注意的是，由于使用了一致性网格，因此不再需要考虑dev_particleArrayIndices数组，而是直接通过dev_gridCellStartIndices和dev_gridCellEndIndices数组访问同一单元格中的粒子数据。这样可以避免额外的内存开销，并提高邻居搜索的效率。

入口函数需要增加对pos和vel的排序：

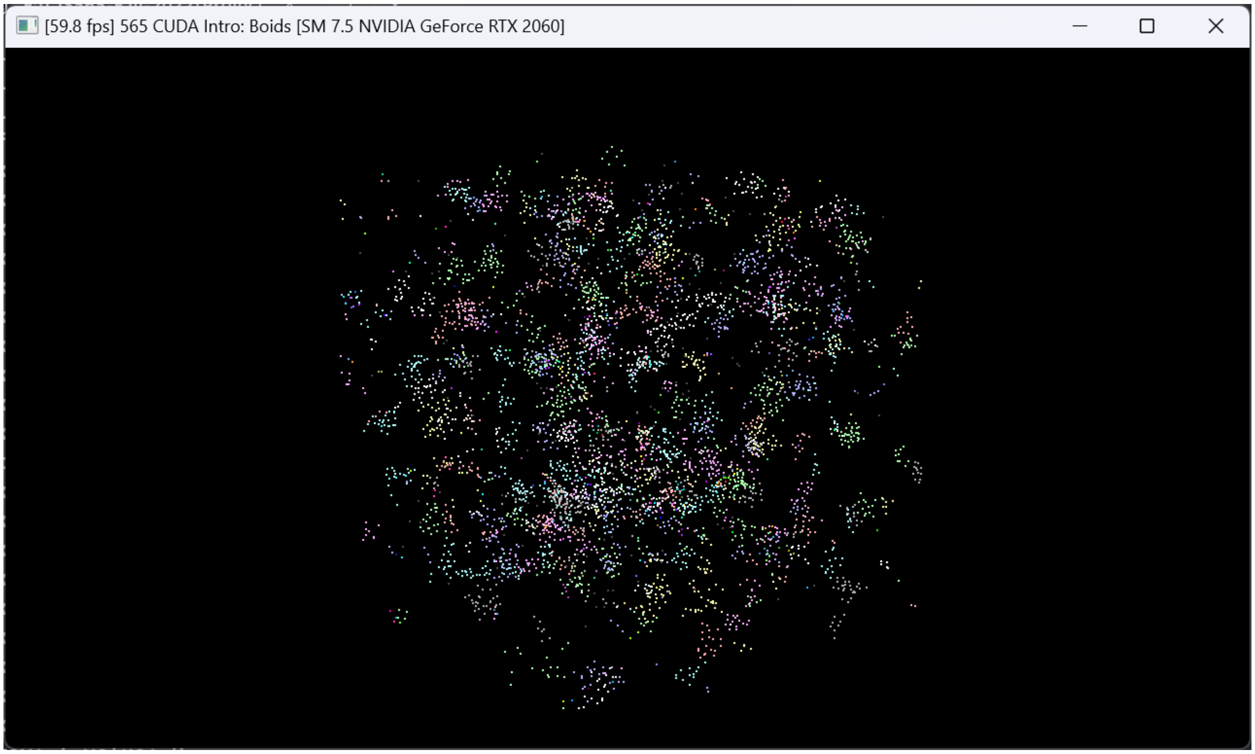
```
__global__ void kernSortPosAndVel(int N, int *particleArrayIndices, glm::vec3 *pos, glm::vec3 *sortedPos, glm::vec3 *vel, glm::vec3 *sorted
    int index = blockDim.x * blockIdx.x + threadIdx.x;
    if (index >= N) return;
    int boidIndex = particleArrayIndices[index];
    sortedPos[index] = pos[boidIndex];
    sortedVel[index] = vel[boidIndex];
}

void Boids::stepSimulationCoherentGrid(float dt) {
    // TODO-2.3 - start by copying Boids::stepSimulationNaiveGrid
    // Uniform Grid Neighbor search using Thrust sort on cell-coherent data.
    // In Parallel:
    // - Label each particle with its array index as well as its grid index.
    //   Use 2x width grids
```

```
    // - Unstable key sort using Thrust. A stable sort isn't necessary, but you
    //   are welcome to do a performance comparison.
    // - Naively unroll the loop for finding the start and end indices of each
    //   cell's data pointers in the array of boid indices
    // - BIG DIFFERENCE: use the rearranged array index buffer to reshuffle all
    //   the particle data in the simulation array.
    //   CONSIDER WHAT ADDITIONAL BUFFERS YOU NEED
    // - Perform velocity updates using neighbor search
    // - Update positions
    // - Ping-pong buffers as needed. THIS MAY BE DIFFERENT FROM BEFORE.
    dim3 blocksPerGridBoids((numObjects + blockSize - 1) / blockSize);
    kernComputeIndices<<<blocksPerGridBoids, blockSize>>>(numObjects, gridSideCount, gridMinimum, gridInverseCellWidth, dev_pos, dev_partic
    thrust::sort_by_key(dev_thrust_particleGridIndices, dev_thrust_particleGridIndices + numObjects, dev_particleArrayIndices);

    dim3 blocksPerGridCells((gridCellCount + blockSize - 1) / blockSize);
    kernResetIntBuffer<<<blocksPerGridCells, blockSize>>>(gridCellCount, dev_gridCellStartIndices, -1);
    kernResetIntBuffer<<<blocksPerGridCells, blockSize>>>(gridCellCount, dev_gridCellEndIndices, -1);
    kernIdentifyCellStartEnd<<<blocksPerGridCells, blockSize>>>(numObjects, dev_particleGridIndices, dev_gridCellStartIndices, dev_gridCell

    kernSortPosAndVel<<<blocksPerGridBoids, blockSize>>>(numObjects, dev_particleArrayIndices, dev_pos, dev_sortedPos, dev_vel1, dev_sorted

    kernUpdateVelNeighborSearchCoherent<<<blocksPerGridBoids, blockSize>>>(numObjects, gridSideCount, gridMinimum, gridInverseCellWidth, gr
                                                                        dev_gridCellStartIndices, dev_gridCellEndIndices,
                                                                        dev_sortedPos, dev_sortedVel, dev_vel2);
    kernUpdatePos<<<blocksPerGridBoids, blockSize>>>(numObjects, dt, dev_sortedPos, dev_vel2);
    std::swap(dev_vel1, dev_vel2);
    std::swap(dev_pos, dev_sortedPos);
}
```

这是Boids类中的另一个函数，用于在一致网格上进行模拟步骤。该函数与在散布网格上进行模拟步骤的主要区别在于，该函数使用连续内存中的一致数据来执行所有计算。具体步骤如下：

- 对于每个粒子，将其标记为其数组索引以及其网格索引。使用2倍的宽度网格。
- 使用Thrust排序按网格索引对粒子数组索引进行排序。
- 对于每个单元格，展开循环以查找数组中该单元格数据指针的起始和结束索引。
- 使用kernSortPosAndVel内核函数将位置和速度数据按排序后的顺序存储到新的缓冲区中。
- 使用kernUpdateVelNeighborSearchCoherent内核函数执行邻居搜索并更新速度，使用kernUpdatePos内核函数更新位置。
- 需要时交替使用缓冲区。

该函数使用CUDA内核函数来实现上述步骤。首先，使用kernComputeIndices内核函数计算每个粒子的网格索引和数组索引。然后，使用Thrust排序按网格索引对粒子数组索引进行排序。接下来，使用kernIdentifyCellStartEnd内核函数确定每个单元格的起始和结束索引。然后，使用kernSortPosAndVel内核函数将位置和速度数据按排序后的顺序存储到新的缓冲区中。最后，使用kernUpdateVelNeighborSearchCoherent内核函数执行邻居搜索并更新速度，使用kernUpdatePos内核函数更新位置。在更新完成后，需要交替使用缓冲区。


[59.8 fps] 565 CUDA Intro: Boids [SM 7.5 NVIDIA GeForce RTX 2060]