18645 How To Write Fast Code

Mini-Project 2

Team 003

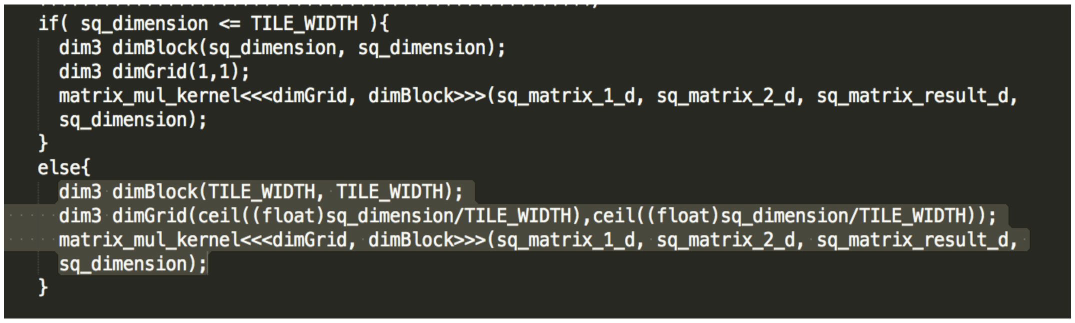
Andrew ID	Name
siqiaow	Siqiao Wu
chul1	Chu Lin
zchai	Zijie Chai

Part 1 Matrix_mul

1. Correct the original program

The original program used only one block and (sq_dimension * sq_dimension) threads. Once the dimension of the matrix exceeds 32, the number of threads in a block will be exceed 1024 and it will cause CUDA program crashing. To avoid this situation, we have to use more than one block when the size of matrix is bigger than 32 * 32.

First thing we should do is judging whether the dimension is exceed over 32(Figure 1).



```
if( sq_dimension <= TILE_WIDTH ){
  dim3 dimBlock(sq_dimension, sq_dimension);
  dim3 dimGrid(1,1);
  matrix_mul_kernel<<<dimGrid, dimBlock>>>(sq_matrix_1_d, sq_matrix_2_d, sq_matrix_result_d,
  sq_dimension);
}
else{
  dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
  dim3 dimGrid(ceil((float)sq_dimension/TILE_WIDTH),ceil((float)sq_dimension/TILE_WIDTH));
  matrix_mul_kernel<<<dimGrid, dimBlock>>>(sq_matrix_1_d, sq_matrix_2_d, sq_matrix_result_d,
  sq_dimension);
}
```

Figure 1

In figure 1, if sq_dimension is bigger that TILE_WIDTH which is defined as 32, we will use multiple blocks to handle this situation. Also we change the Makefile to use O3 optimization. The performance become 315Gflops.

2. Techniques To Speed up

(1) Optimized by using blocking

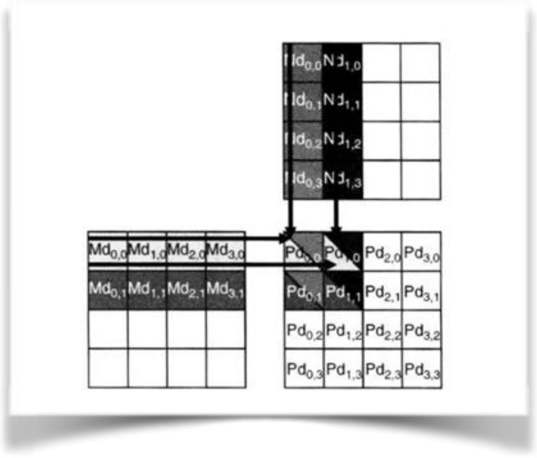
a. Optimization Goal:

This approach optimizes for many-core and cache. We use CUDA. For each block in grid, we use share memory. Share memory are more efficient that global memory and we could reduce time of accessing memory by using share memory.

b. Optimization process:

In this approach, we consider data. In normal matrix multiplication approach, same memory address will be access several times. By using share memory and blocking skill, we could reduce the time of accessing same memory address, so that the performance will be much better.

In textbook "programming massively parallel processors" chapter 5 section 3, it introduces an approach using blocking to reducing the time of reading data from memory in order to make computation resources focus on calculation rather than getting data. So for each matrix, we divide it to many 32 * 32 sub-matrix. Figure 2 use a 2*2 sub-matrix to illustrate this approach.



$Nd_{0,0}$ $Nd_{1,0}$
$Nd_{0,1}$ $Nd_{1,1}$
$Nd_{0,2}$ $Nd_{1,2}$
$Nd_{0,3}$ $Nd_{1,3}$
$Md_{0,0}$ $Md_{1,0}$ $Md_{2,0}$ $Md_{3,0}$
$Md_{0,1}$ $Md_{1,1}$ $Md_{2,1}$ $Md_{3,1}$
$Pd_{0,0}$ $Pd_{1,0}$ $Pd_{2,0}$ $Pd_{3,0}$
$Pd_{0,1}$ $Pd_{1,1}$ $Pd_{2,1}$ $Pd_{3,1}$
$Pd_{0,2}$ $Pd_{1,2}$ $Pd_{2,2}$ $Pd_{3,2}$
$Pd_{0,3}$ $Pd_{1,3}$ $Pd_{2,3}$ $Pd_{3,3}$

Figure 2(from "programming massively parallel processors")

c. Optimization result::

By using this, the performance is improved from 315 Gflops to 1102 Gflops.

Because the block size is 32 * 32, the times accessing memory will decreased to 1/32 of original accessing times, the expected speedup is 32/4 = 8 times. Here the '4' means each floating number is 4 byte. However, we only get 1102 / 315 =3.5 times speedup. The reason is bank conflict.

(2) Optimized by using loop unrolling

a. Optimization Goal:

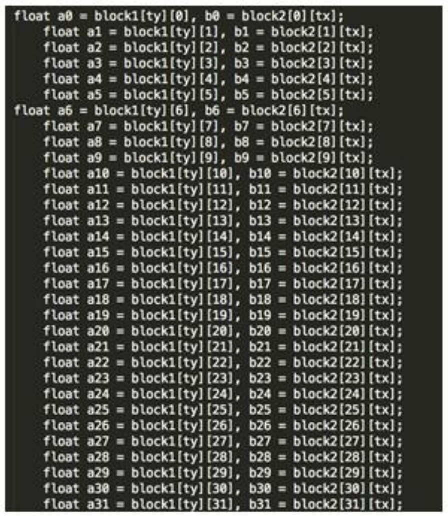
Reduce predicting condition branches since each wrong prediction will waste 3 clocks.

b. Optimization process:

When we multiply blocks in two matrixes, since the block size is determined (in our program it is 32), we don’t have to use for loop to calculate the result. Instead, we could unroll the loop, so that the condition branches will be eliminated and the performance will be improved.

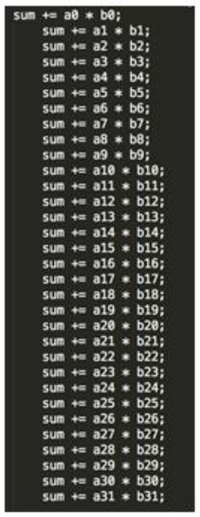
c. Optimization result:

To eliminate 32 times condition branch judgements, the program will get 32 times improvement theoretically. But since this loop is a very small part of the whole program, so the effect will be much smaller than our expectation. Our experiment shows that the performance is improved from 1102 Gflops to 1150 Gflops. Figure 3(a) and 3(b) show how we do it.



```
float a0 = block1[ty][0], b0 = block2[0][tx];
    float a1 = block1[ty][1], b1 = block2[1][tx];
    float a2 = block1[ty][2], b2 = block2[2][tx];
    float a3 = block1[ty][3], b3 = block2[3][tx];
    float a4 = block1[ty][4], b4 = block2[4][tx];
    float a5 = block1[ty][5], b5 = block2[5][tx];
float a6 = block1[ty][6], b6 = block2[6][tx];
    float a7 = block1[ty][7], b7 = block2[7][tx];
    float a8 = block1[ty][8], b8 = block2[8][tx];
    float a9 = block1[ty][9], b9 = block2[9][tx];
    float a10 = block1[ty][10], b10 = block2[10][tx];
    float a11 = block1[ty][11], b11 = block2[11][tx];
    float a12 = block1[ty][12], b12 = block2[12][tx];
    float a13 = block1[ty][13], b13 = block2[13][tx];
    float a14 = block1[ty][14], b14 = block2[14][tx];
    float a15 = block1[ty][15], b15 = block2[15][tx];
    float a16 = block1[ty][16], b16 = block2[16][tx];
    float a17 = block1[ty][17], b17 = block2[17][tx];
    float a18 = block1[ty][18], b18 = block2[18][tx];
    float a19 = block1[ty][19], b19 = block2[19][tx];
    float a20 = block1[ty][20], b20 = block2[20][tx];
    float a21 = block1[ty][21], b21 = block2[21][tx];
    float a22 = block1[ty][22], b22 = block2[22][tx];
    float a23 = block1[ty][23], b23 = block2[23][tx];
    float a24 = block1[ty][24], b24 = block2[24][tx];
    float a25 = block1[ty][25], b25 = block2[25][tx];
    float a26 = block1[ty][26], b26 = block2[26][tx];
    float a27 = block1[ty][27], b27 = block2[27][tx];
    float a28 = block1[ty][28], b28 = block2[28][tx];
    float a29 = block1[ty][29], b29 = block2[29][tx];
    float a30 = block1[ty][30], b30 = block2[30][tx];
    float a31 = block1[ty][31], b31 = block2[31][tx];
```

Figure 3(a)



```
sum += a0 * b0;
    sum += a1 * b1;
    sum += a2 * b2;
    sum += a3 * b3;
    sum += a4 * b4;
    sum += a5 * b5;
    sum += a6 * b6;
    sum += a7 * b7;
    sum += a8 * b8;
    sum += a9 * b9;
    sum += a10 * b10;
    sum += a11 * b11;
    sum += a12 * b12;
    sum += a13 * b13;
    sum += a14 * b14;
    sum += a15 * b15;
    sum += a16 * b16;
    sum += a17 * b17;
    sum += a18 * b18;
    sum += a19 * b19;
    sum += a20 * b20;
    sum += a21 * b21;
    sum += a22 * b22;
    sum += a23 * b23;
    sum += a24 * b24;
    sum += a25 * b25;
    sum += a26 * b26;
    sum += a27 * b27;
    sum += a28 * b28;
    sum += a29 * b29;
    sum += a30 * b30;
    sum += a31 * b31;
```

Figure 3(b)

(3) Optimized by eliminating redundant calculation

a. Optimization Goal:

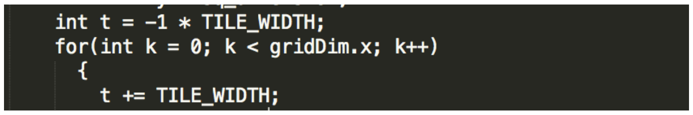
The goal is to eliminate the unnecessary computation.

b. Optimization process:

In our code, there exist some redundant calculations. For example, we have calculate row*sq_dimension and ty * sq_dimension which is not going to change in a for-loop. So we calculate them outside the for-loop to reduce those unnecessary calculations. Another thing is that in a for-loop, the loop variable is k and we are going to use k * TILE_WIDTH. In this case, since k will plus one at each iteration, so we don't have to compute the multiplication. Instead, we add TILE_WIDTH to the result at each iteration and we could replace multiply with addition, which will improve the performance.

c. **Optimization result:**

In figure 4, t will be the result of k * TILE_WIDTH, and we replace multiply with addition.As result, the performance improve from 1150 Gflops to 1185Gflops.



```
    int t = -1 * TILE_WIDTH;
    for(int k = 0; k < gridDim.x; k++)
      {
        t += TILE_WIDTH;
```

Figure 4

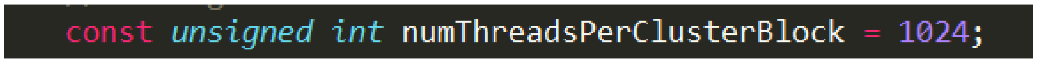
Part 2 K-means

1. Correct the original program

(1) Modify the number of threads per block

Each SM(streaming multiprocessor) can hold 2048 threads, and 8 blocks. If number of threads per block is 128, then there will be 16 blocks in one SM which exceed the limit number of the blocks for one SM.

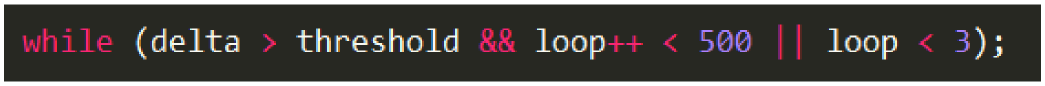
So, we should increase the number of threads in one block to decrease the number of blocks in one SM.



```
const unsigned int numThreadsPerClusterBlock = 1024;
```

(2) Modify the least iteration times

The error we get after we fixed the problem mentioned before is: mismatching, which means that the membership of the objects is not correct. So we think about that it may be because the iteration number is too small and the objects are not classified to the right cluster. And after we check the termination condition of the loop, we think that maybe it is because threshold is too large and let the loop is executed for once. So we add the condition to make sure that the loop can execute at least three times.



```
while (delta > threshold && loop++ < 500 || loop < 3);
```

2. Techniques To Speed up

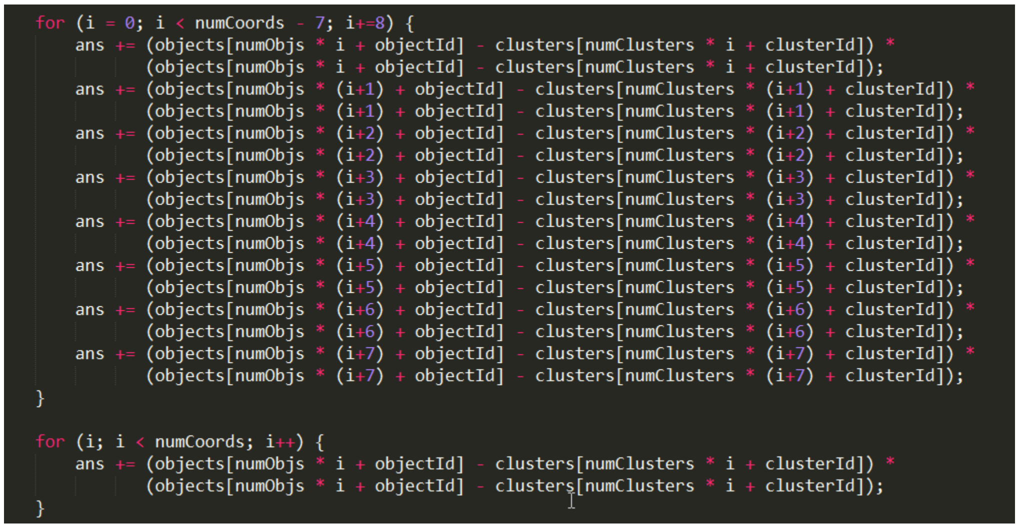
(1) Loop unrolling

a. **Optimization Goal:**

Try to decrease the number of the iteration by doing more calculations in one loop.

b. **Optimization Process:**

When computing the distance between an object point and a cluster center, or computing the new coordinates of a cluster center, we can calculate the value of every dimension in one loop instead of numCoords loops by using loop unrolling. And the implementation is shown in figure 5.



```
for (i = 0; i < numCoords - 7; i+=8) {
    ans += (objects[numObjs * i + objectId] - clusters[numClusters * i + clusterId]) *
           (objects[numObjs * i + objectId] - clusters[numClusters * i + clusterId]);
    ans += (objects[numObjs * (i+1) + objectId] - clusters[numClusters * (i+1) + clusterId]) *
           (objects[numObjs * (i+1) + objectId] - clusters[numClusters * (i+1) + clusterId]);
    ans += (objects[numObjs * (i+2) + objectId] - clusters[numClusters * (i+2) + clusterId]) *
           (objects[numObjs * (i+2) + objectId] - clusters[numClusters * (i+2) + clusterId]);
    ans += (objects[numObjs * (i+3) + objectId] - clusters[numClusters * (i+3) + clusterId]) *
           (objects[numObjs * (i+3) + objectId] - clusters[numClusters * (i+3) + clusterId]);
    ans += (objects[numObjs * (i+4) + objectId] - clusters[numClusters * (i+4) + clusterId]) *
           (objects[numObjs * (i+4) + objectId] - clusters[numClusters * (i+4) + clusterId]);
    ans += (objects[numObjs * (i+5) + objectId] - clusters[numClusters * (i+5) + clusterId]) *
           (objects[numObjs * (i+5) + objectId] - clusters[numClusters * (i+5) + clusterId]);
    ans += (objects[numObjs * (i+6) + objectId] - clusters[numClusters * (i+6) + clusterId]) *
           (objects[numObjs * (i+6) + objectId] - clusters[numClusters * (i+6) + clusterId]);
    ans += (objects[numObjs * (i+7) + objectId] - clusters[numClusters * (i+7) + clusterId]) *
           (objects[numObjs * (i+7) + objectId] - clusters[numClusters * (i+7) + clusterId]);
}

for (i; i < numCoords; i++) {
    ans += (objects[numObjs * i + objectId] - clusters[numClusters * i + clusterId]) *
           (objects[numObjs * i + objectId] - clusters[numClusters * i + clusterId]);
}
```

Figure 5. Loop unrolling

c. **Optimization result:**

Because we unroll the loop by 8 times, the program should speed up 8 times. However, because the loop is a small part of the program, much time in the program is used to schedule threads. So before we use loop unrolling the speed up is 2.3(use –o3 optimization), and after we use loop unrolling the speed up is 2.37

(2) SIMT

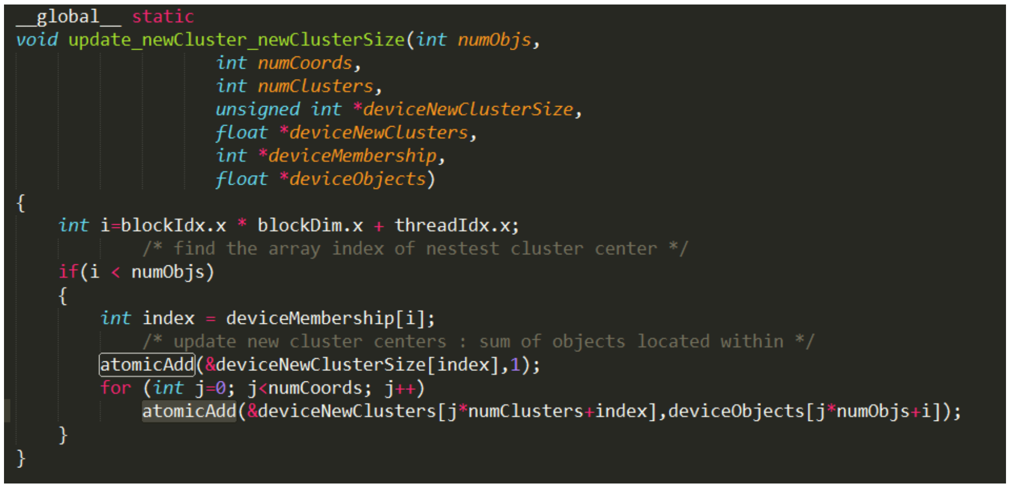
a. **Optimization Goal:**

SIMT means that use multiple threads to execute one instruction. It can be applied to reducing iteration by put the calculation in every loop to different threads, then execute all the threads at the same to reduce execution time.

b. **Optimization Process:**

There are many loops in the program, however we should optimize the loops executing on the host. So we modify the parts that calculates the new cluster centers and the number of objects belonging to different clusters.

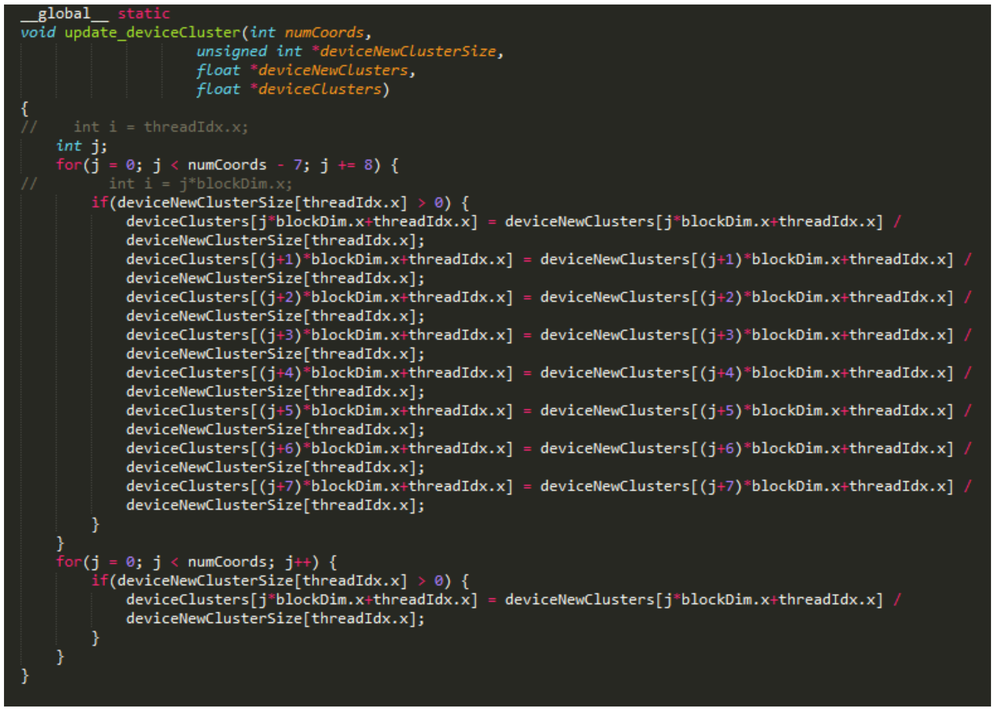
For the part that calculates the number of objects belonging to different clusters is shown in figure 6. For every object, we use one thread to access the membership and coordinate values. And because here multiple objects can belong to one cluster which may lead to the fact that multiple threads write to one memory location which store the data related to one cluster, so we use atomicAdd() function instead of simple addition.



```
__global__ static
void update_newCluster_newClusterSize(int numObjs,
                        int numCoords,
                        int numClusters,
                        unsigned int *deviceNewClusterSize,
                        float *deviceNewClusters,
                        int *deviceMembership,
                        float *deviceObjects)
{
    int i=blockIdx.x * blockDim.x + threadIdx.x;
            /* find the array index of nestest cluster center */
    if(i < numObjs)
    {
        int index = deviceMembership[i];
            /* update new cluster centers : sum of objects located within */
        atomicAdd(&deviceNewClusterSize[index],1);
        for (int j=0; j<numCoords; j++)
            atomicAdd(&deviceNewClusters[j*numClusters+index],deviceObjects[j*numObjs+i]);
    }
}
```

Figure 6

And for the part that calculates the coordinate values of every cluster center after every update, we also use numCluster threads to calculate the values of every cluster at the same time. And here we also use loop unrolling to speed up. The implementation is shown in figure 7.



```
__global__ static
void update_deviceCluster(int numCoords,
                    unsigned int *deviceNewClusterSize,
                    float *deviceNewClusters,
                    float *deviceClusters)
{
//    int i = threadIdx.x;
    int j;
    for(j = 0; j < numCoords - 7; j += 8) {
//        int i = j*blockDim.x;
        if(deviceNewClusterSize[threadIdx.x] > 0) {
            deviceClusters[j*blockDim.x+threadIdx.x] = deviceNewClusters[j*blockDim.x+threadIdx.x] /
            deviceNewClusterSize[threadIdx.x];
            deviceClusters[(j+1)*blockDim.x+threadIdx.x] = deviceNewClusters[(j+1)*blockDim.x+threadIdx.x] /
            deviceNewClusterSize[threadIdx.x];
            deviceClusters[(j+2)*blockDim.x+threadIdx.x] = deviceNewClusters[(j+2)*blockDim.x+threadIdx.x] /
            deviceNewClusterSize[threadIdx.x];
            deviceClusters[(j+3)*blockDim.x+threadIdx.x] = deviceNewClusters[(j+3)*blockDim.x+threadIdx.x] /
            deviceNewClusterSize[threadIdx.x];
            deviceClusters[(j+4)*blockDim.x+threadIdx.x] = deviceNewClusters[(j+4)*blockDim.x+threadIdx.x] /
            deviceNewClusterSize[threadIdx.x];
            deviceClusters[(j+5)*blockDim.x+threadIdx.x] = deviceNewClusters[(j+5)*blockDim.x+threadIdx.x] /
            deviceNewClusterSize[threadIdx.x];
            deviceClusters[(j+6)*blockDim.x+threadIdx.x] = deviceNewClusters[(j+6)*blockDim.x+threadIdx.x] /
            deviceNewClusterSize[threadIdx.x];
            deviceClusters[(j+7)*blockDim.x+threadIdx.x] = deviceNewClusters[(j+7)*blockDim.x+threadIdx.x] /
            deviceNewClusterSize[threadIdx.x];
        }
    }
    for(j = 0; j < numCoords; j++) {
        if(deviceNewClusterSize[threadIdx.x] > 0) {
            deviceClusters[j*blockDim.x+threadIdx.x] = deviceNewClusters[j*blockDim.x+threadIdx.x] /
            deviceNewClusterSize[threadIdx.x];
        }
    }
}
```

Figure 7

c. **Optimization result:**

Theoretically, the speed up should equal to the number of threads that execute at the same time. However, this part is a small part of the whole program, so there are many parts in the program has not been speed up. And when we use multiple threads to execute program, we need much time to schedule threads. So the actual speed up is lower than our expectation. Before we use SIMT, the speed up is 2.37, and after using SIMT the speed up is 3.24.

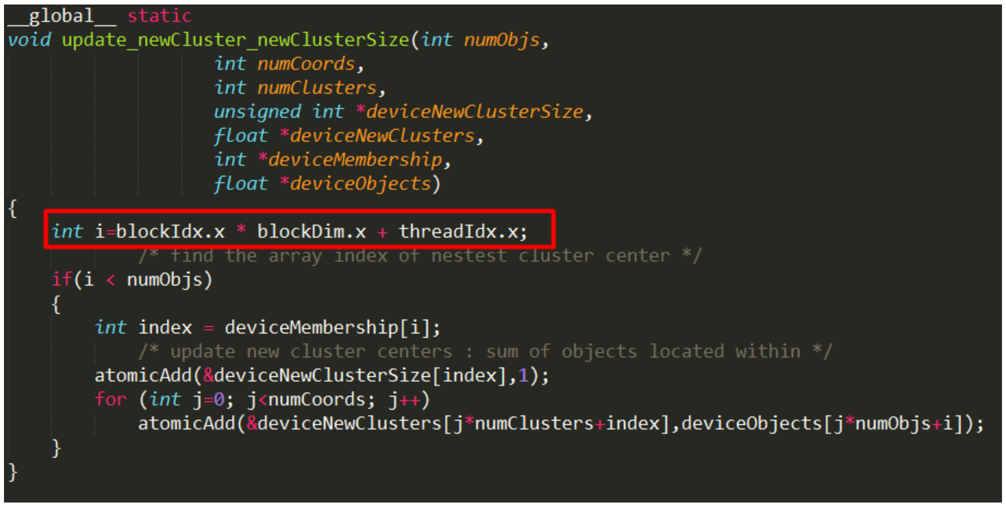
(3) Optimized by eliminating redundant calculation and using –O3 optimization

a. **Optimization Goal:**

The goal is to eliminate the unnecessary computation.

b. **Optimization Process:**

In our code, there exists some redundant calculation. For example, when accessing elements in an array, if the index is calculated by an expression, then we can calculate it for once and use it multiple times. And the implementation is shown in figure 9.



```
__global__ static
void update_newCluster_newClusterSize(int numObjs,
                                      int numCoords,
                                      int numClusters,
                                      unsigned int *deviceNewClusterSize,
                                      float *deviceNewClusters,
                                      int *deviceMembership,
                                      float *deviceObjects)
{
    int i=blockIdx.x * blockDim.x + threadIdx.x;
        /* find the array index of nestest cluster center */
    if(i < numObjs)
    {
        int index = deviceMembership[i];
            /* update new cluster centers : sum of objects located within */
        atomicAdd(&deviceNewClusterSize[index],1);
        for (int j=0; j<numCoords; j++)
            atomicAdd(&deviceNewClusters[j*numClusters+index],deviceObjects[j*numObjs+i]);
    }
}
```

c. **Optimization result:**

Before using -O3 optimization, the speed up is 0.98. And after using –O3 optimization, the speed up is 2.21.