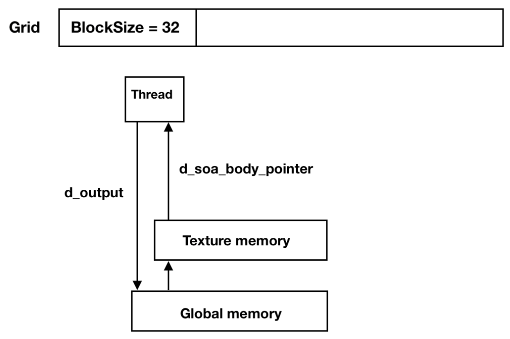
Assignment 2

acz19yz

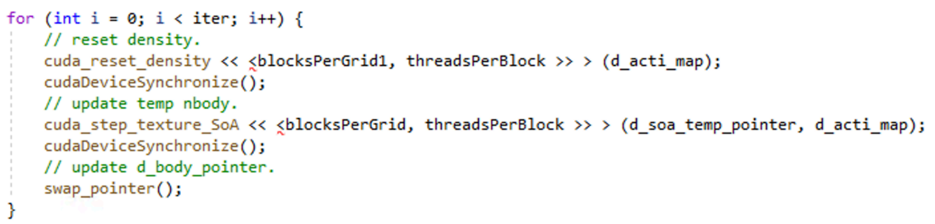
1. N threads

This section includes four parts. Those are description, comparison of performance using variant techniques, optimization and result. According to the result, SoA based on texture memory is less time-consuming and this strategy could decrease running time greatly compared with CPU and openMP. The next figure shows the basis steps in step function.



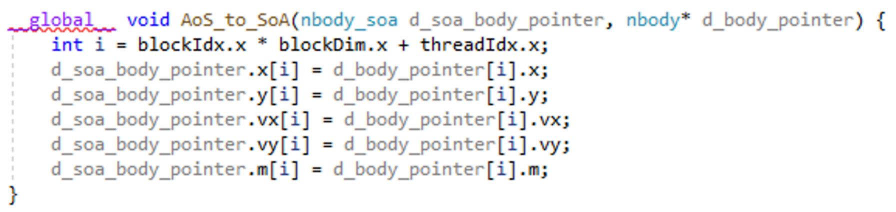
1.1 Description:

The approach that updates body situation in parallel but the calculation of join force is in serial has several fixed steps. Firstly, it is the initialisation of density array. Secondly, it is the calculation of the latest position of bodies and updates density array. Thirdly, it updates body position according to the latest information from step 2. the next figure displays those step in step function.



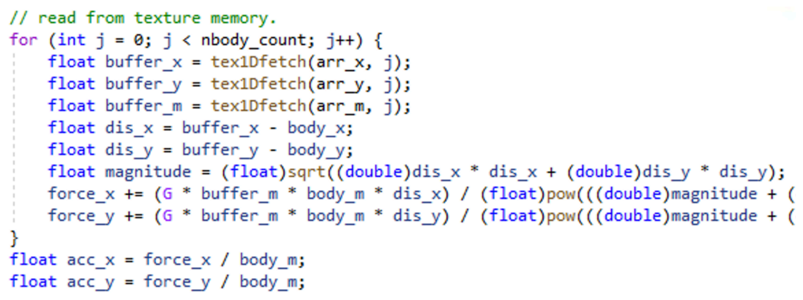
```
for (int i = 0; i < iter; i++) {
    // reset density.
    cuda_reset_density << <blocksPerGrid1, threadsPerBlock >> > (d_acti_map);
    cudaDeviceSynchronize();
    // update temp nbody.
    cuda_step_texture_SoA << <blocksPerGrid, threadsPerBlock >> > (d_soa_temp_pointer, d_acti_map);
    cudaDeviceSynchronize();
    // update d_body_pointer.
    swap_pointer();
}
```

The detail of each steps will be discussed as follow. First of all, the original structure of data (AoS) was converted into struct of array (SoA) if we want to use SoA to improve bandwidth.

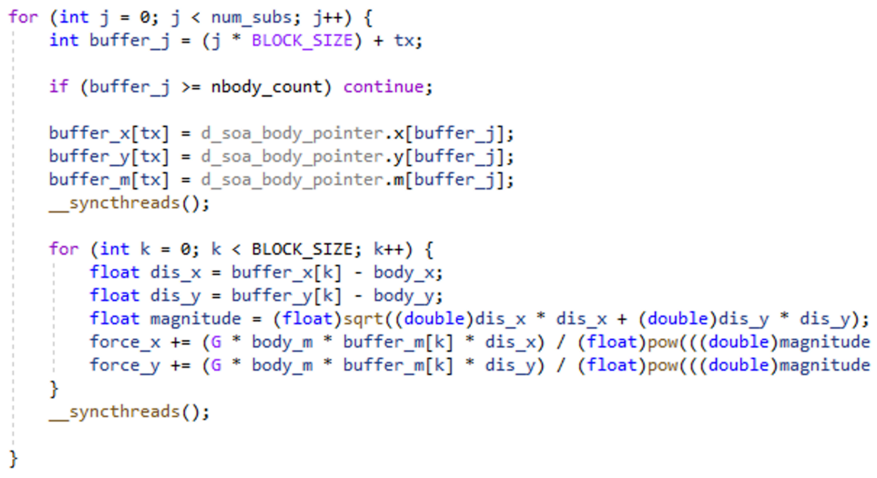


```
__global__ void AoS_to_SoA(nbody_soa d_soa_body_pointer, nbody* d_body_pointer) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    d_soa_body_pointer.x[i] = d_body_pointer[i].x;
    d_soa_body_pointer.y[i] = d_body_pointer[i].y;
    d_soa_body_pointer.vx[i] = d_body_pointer[i].vx;
    d_soa_body_pointer.vy[i] = d_body_pointer[i].vy;
    d_soa_body_pointer.m[i] = d_body_pointer[i].m;
}
```

After that, in force calculation and density update, two different strategies based on texture and shared memory will be illustrated. As for texture memory, a temporary body heap pointer was used to store the latest data. In this case, original body data will not be updated within one iteration.


```
// read from texture memory.
for (int j = 0; j < nbody_count; j++) {
    float buffer_x = tex1Dfetch(arr_x, j);
    float buffer_y = tex1Dfetch(arr_y, j);
    float buffer_m = tex1Dfetch(arr_m, j);
    float dis_x = buffer_x - body_x;
    float dis_y = buffer_y - body_y;
    float magnitude = (float)sqrt((double)dis_x * dis_x + (double)dis_y * dis_y);
    force_x += (G * buffer_m * body_m * dis_x) / (float)pow(((double)magnitude + (
    force_y += (G * buffer_m * body_m * dis_y) / (float)pow(((double)magnitude + (
}
float acc_x = force_x / body_m;
float acc_y = force_y / body_m;
```

As for shared memory, there is a little difference compared with texture memory. To be specific, whole data, especially big size data, is unable to be loaded into shared memory once. As a consequence, whole data was loaded into fixed size shared memory several times. Details are as follow.


```
for (int j = 0; j < num_subs; j++) {
    int buffer_j = (j * BLOCK_SIZE) + tx;

    if (buffer_j >= nbody_count) continue;

    buffer_x[tx] = d_soa_body_pointer.x[buffer_j];
    buffer_y[tx] = d_soa_body_pointer.y[buffer_j];
    buffer_m[tx] = d_soa_body_pointer.m[buffer_j];
    __syncthreads();

    for (int k = 0; k < BLOCK_SIZE; k++) {
        float dis_x = buffer_x[k] - body_x;
        float dis_y = buffer_y[k] - body_y;
        float magnitude = (float)sqrt((double)dis_x * dis_x + (double)dis_y * dis_y);
        force_x += (G * body_m * buffer_m[k] * dis_x) / (float)pow(((double)magnitude
        force_y += (G * body_m * buffer_m[k] * dis_y) / (float)pow(((double)magnitude
    }
    __syncthreads();

}
```

Every SMP loads data with block size into shared memory and sync-threads function was used before and after forces calculation. There is one detail that num_subs, number of block, might be bigger than it's needed actually. Therefore, boundary should be checked before data loading. The update of density was implemented using atomicAdd function to avoid race condition.

Finally, the latest situation was updated by swapping pointers.

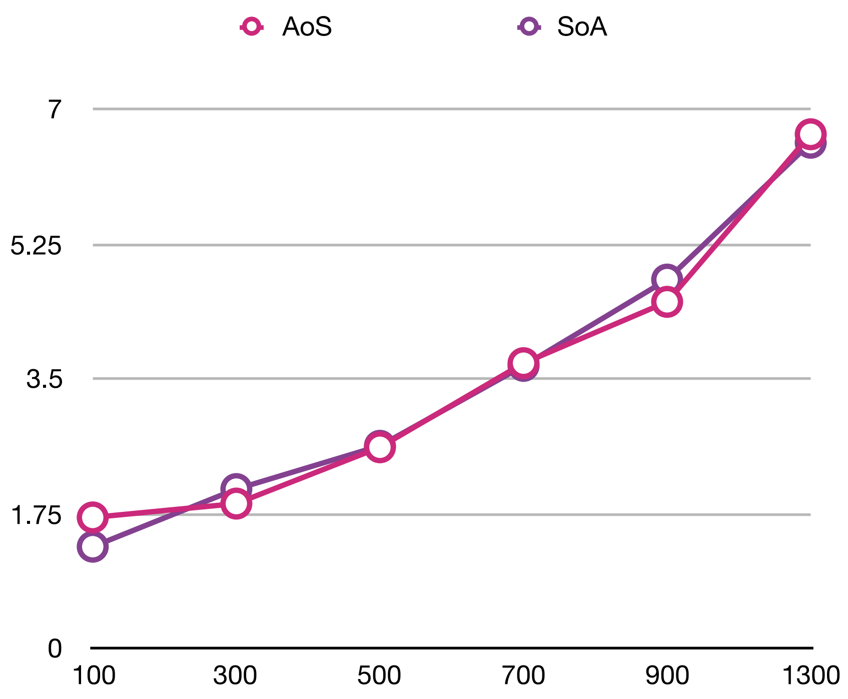
1.2 Comparison:

In this approach, there are four different GPU memory pattern including global, read-only, texture as well as shared memory and two different data layout. The next table is the result of average running time (3 times) in different strategies.

blockSize		**32**	
Grid = 5, I = 1000	**Num**	**AoS**	**SoA**
Global memory	**1000**	5.169	5.159
	3000	19.212	19.312
	5000	45.382	45.076
	1000	5.089	4.937
Read-only memory	**3000**	19.062	19.046

blockSize		32	
Grid = 5, I = 1000	**Num**	**AoS**	**SoA**
	5000	44.828	44.593
Texture memory	**1000**	-	**4.599**
	3000	-	**18.353**
	5000	-	**43.201**
Shared memory	**1000**	4.895	5.018
	3000	19.377	19.319
	5000	45.516	45.153

The next figure compare the performance between AoS and SoA in global memory.



According to the line chart above, the performances based on two variant memory access are quite similar.
With aspect to the result of table, the best one is SoA based on texture memory. Several details and phenomenons will be discussed as follow.

Firstly, the test is not implemented in AoS structure data in texture memory strategies because AoS structure is not suitable in it. To be specific, texture <float, 1, cudaReadModeElementType> can only accept several type of data. Unfortunately, structure data type is not in the list. Moreover, constant memory is not implemented in here because it cannot be allocated dynamically. The size of it must be fixed and declared as constant type before code running. Therefore, constant memory was not chosen in N threads.

Secondly, in general, the running time of global memory access is longer than others memory access whatever any data layout I choose.

Thirdly, the performance of shared memory is not better than that of texture memory. The reason might be that loading data from global memory to shared memory might be time consuming every time and there is no approach to load all data into shared memory. Therefore, shared memory approach needs long running time.

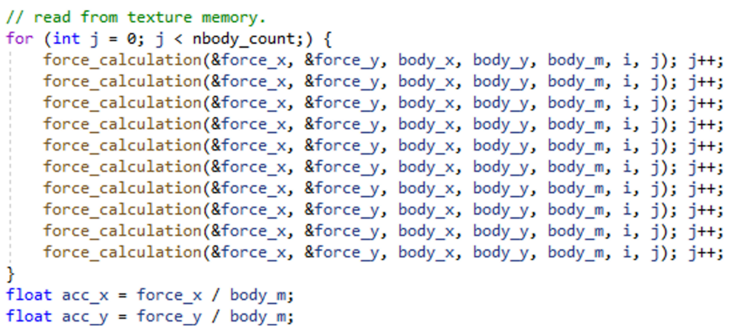
Thirdly, there is no big difference of performance between AoS and SoA, which is contradicted to my opinion that approach based on SoA layout should be faster than AoS. The reason might be that in the force calculation loop, every threads have to load every body from memory (global, read-only or texture) to L2 to register. This process might be time-consuming because of lower bandwidth although SoA layout could improve bandwidth.

1.3 optimisations:

There are several approaches for improving performance, such as unrolling loop, using shared memory to cache the latest data, increasing block size. Therefore, let's try to implement both methods to check whether the performance will be improved.

Firstly, let's try to unroll for loop. The next displays the code and comparison of running time.

	1000	3000	5000
Unroll for loop	4.86 > 4.599	18.633 > 18.353	43.832 > 43.201

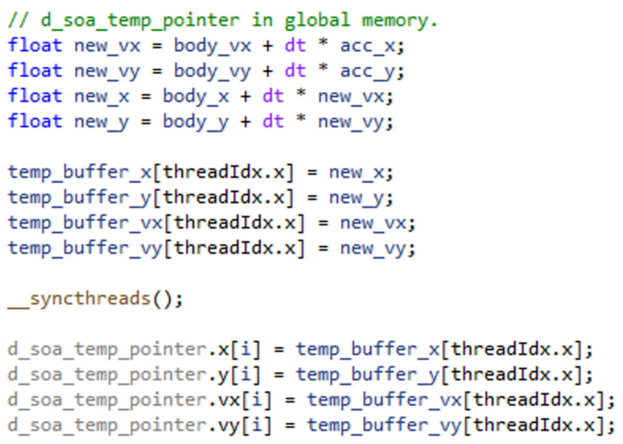


```
// read from texture memory.
for (int j = 0; j < nbody_count;) {
    force_calculation(&force_x, &force_y, body_x, body_y, body_m, i, j); j++;
    force_calculation(&force_x, &force_y, body_x, body_y, body_m, i, j); j++;
    force_calculation(&force_x, &force_y, body_x, body_y, body_m, i, j); j++;
    force_calculation(&force_x, &force_y, body_x, body_y, body_m, i, j); j++;
    force_calculation(&force_x, &force_y, body_x, body_y, body_m, i, j); j++;
    force_calculation(&force_x, &force_y, body_x, body_y, body_m, i, j); j++;
    force_calculation(&force_x, &force_y, body_x, body_y, body_m, i, j); j++;
    force_calculation(&force_x, &force_y, body_x, body_y, body_m, i, j); j++;
    force_calculation(&force_x, &force_y, body_x, body_y, body_m, i, j); j++;
    force_calculation(&force_x, &force_y, body_x, body_y, body_m, i, j); j++;
}
float acc_x = force_x / body_m;
float acc_y = force_y / body_m;
```

According to the result above, unrolling loop is more time consuming. The reason is that calling a function make performance worse. Many variables have to be push on the stack and allocate new variable space. In this case, this approach have negative impact on running time. However, function loop could be avoid if code cleanliness was ignored. Therefore, this way will not be considered.

Secondly, in texture memory strategy, the latest data will be written to global memory directly. It there any improvement if those latest information cached into shared memory was moving to global memory after all threads in a block were finished. The next is the comparison and code of second optimization.

	1000	3000	5000
Cache temp data	5.337 > 4.599	18.767 > 18.353	43.309 > 43.201



```
// d_soa_temp_pointer in global memory.
float new_vx = body_vx + dt * acc_x;
float new_vy = body_vy + dt * acc_y;
float new_x = body_x + dt * new_vx;
float new_y = body_y + dt * new_vy;

temp_buffer_x[threadIdx.x] = new_x;
temp_buffer_y[threadIdx.x] = new_y;
temp_buffer_vx[threadIdx.x] = new_vx;
temp_buffer_vy[threadIdx.x] = new_vy;

__syncthreads();

d_soa_temp_pointer.x[i] = temp_buffer_x[threadIdx.x];
d_soa_temp_pointer.y[i] = temp_buffer_y[threadIdx.x];
d_soa_temp_pointer.vx[i] = temp_buffer_vx[threadIdx.x];
d_soa_temp_pointer.vy[i] = temp_buffer_vy[threadIdx.x];
```

According to the result, this way is also not a good idea.
Thirdly, using variant block dimensions might also have different performance. Let's compare the performance using different block size.

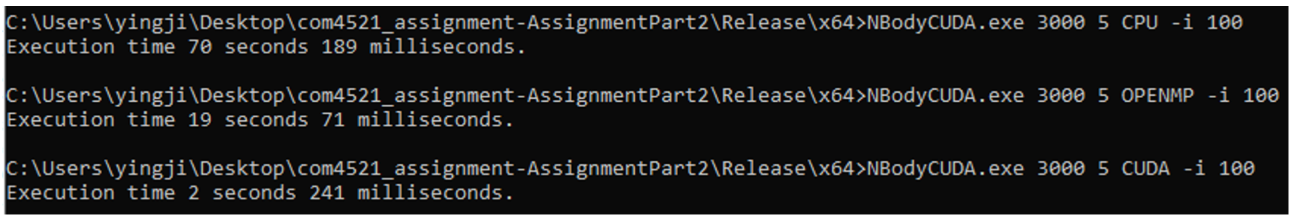
Num = 1000, I = 1000	16	32	64
Running time (texture)	4.595	4.599	4.726

The results are similar using 16, 32 and 64 block size. Usually, the waste of threads will be more serious with the increasing of block size. However, the delay of loading data will be covered up in bigger block size. Therefore, 32 block size was chosen.

1.4 Result:

The next figure is the performance comparison between CPU, OPENMP and CUDA. Obviously, GPU parallel could improve performance greatly.

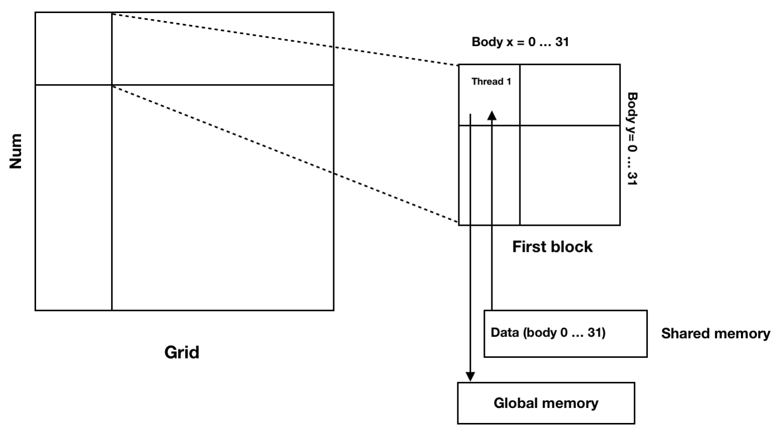
Grid = 5, I = 300			
Num	**CPU**	**OPENMP**	**CUDA**
500	5.744	1.583	0.796
700	11.238	3.104	0.967
900	18.619	5.114	1.233
1100	27.775	7.663	1.476



```
C:\Users\yingji\Desktop\com4521_assignment-AssignmentPart2\Release\x64>NBodyCUDA.exe 3000 5 CPU -i 100
Execution time 70 seconds 189 milliseconds.

C:\Users\yingji\Desktop\com4521_assignment-AssignmentPart2\Release\x64>NBodyCUDA.exe 3000 5 OPENMP -i 100
Execution time 19 seconds 71 milliseconds.

C:\Users\yingji\Desktop\com4521_assignment-AssignmentPart2\Release\x64>NBodyCUDA.exe 3000 5 CUDA -i 100
Execution time 2 seconds 241 milliseconds.
```

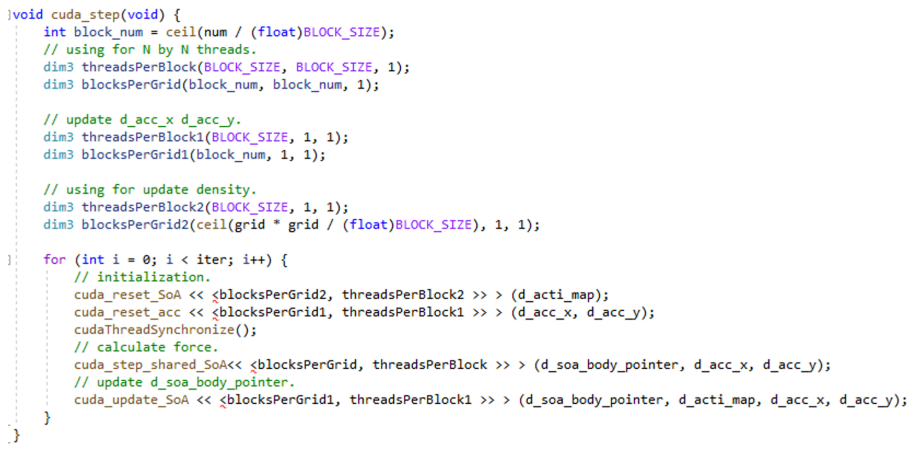
2. N*N threads

In this section, N by N threads will be described. Moreover, the performance of several strategies will be compared. According to the result, SoA layout based on texture memory could achieve the best performance. The next figure displays the basis process of simulation.



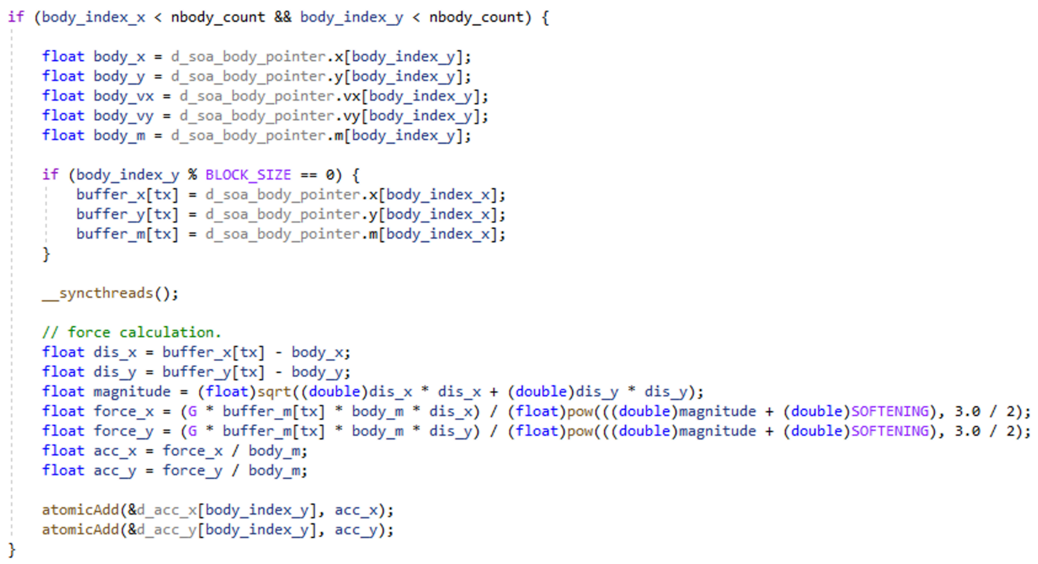
2.1 Description:

There are several steps in N by N implementation. First of all, it is the initialisation of density array and two temporary arrays in which accelerated speed was stored. After that, the force of every pair of bodies was calculated by N*N threads. Finally, the density and the latest situation of body were updated. The next figure is the step function.



```
void cuda_step(void) {
    int block_num = ceil(num / (float)BLOCK_SIZE);
    // using for N by N threads.
    dim3 threadsPerBlock(BLOCK_SIZE, BLOCK_SIZE, 1);
    dim3 blocksPerGrid(block_num, block_num, 1);

    // update d_acc_x d_acc_y.
    dim3 threadsPerBlock1(BLOCK_SIZE, 1, 1);
    dim3 blocksPerGrid1(block_num, 1, 1);

    // using for update density.
    dim3 threadsPerBlock2(BLOCK_SIZE, 1, 1);
    dim3 blocksPerGrid2(ceil(grid * grid / (float)BLOCK_SIZE), 1, 1);

    for (int i = 0; i < iter; i++) {
        // initialization.
        cuda_reset_SoA << <blocksPerGrid2, threadsPerBlock2 >> > (d_acti_map);
        cuda_reset_acc << <blocksPerGrid1, threadsPerBlock1 >> > (d_acc_x, d_acc_y);
        cudaThreadSynchronize();
        // calculate force.
        cuda_step_shared_SoA<< <blocksPerGrid, threadsPerBlock >> > (d_soa_body_pointer, d_acc_x, d_acc_y);
        // update d_soa_body_pointer.
        cuda_update_SoA << <blocksPerGrid1, threadsPerBlock1 >> > (d_soa_body_pointer, d_acti_map, d_acc_x, d_acc_y);
    }
}
```

The next two figure displays the calculation of forces in “cuda_step_shared_SoA” function. There are several details that need to be consider. Firstly, the boundary should be consider. To be specific, the number of bodies can not be divided by block size without remainder in general. In this scenario, many threads in SMP should’t be executed because their index might out of the boundary of heap. Therefore, their index (“blockIdx.x + blockDim.x + threadIdx.x” or “blockIdx.y + blockDim.y + threadIdx.y”) should be check before force calculation. Secondly, data should be loaded into shared memory by the threads of the first rows in every block only, which could decrease the times of access global memory. Thirdly, race condition should be consider when summing accelerated speed. The next is the code of force calculation.

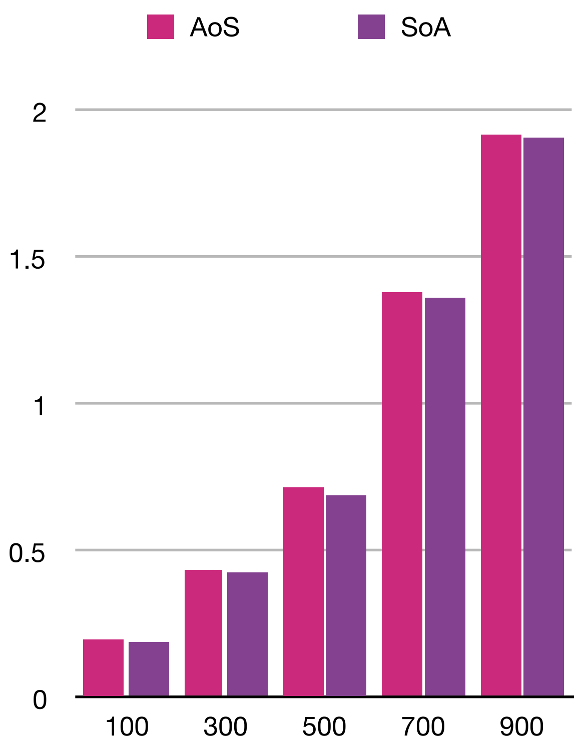


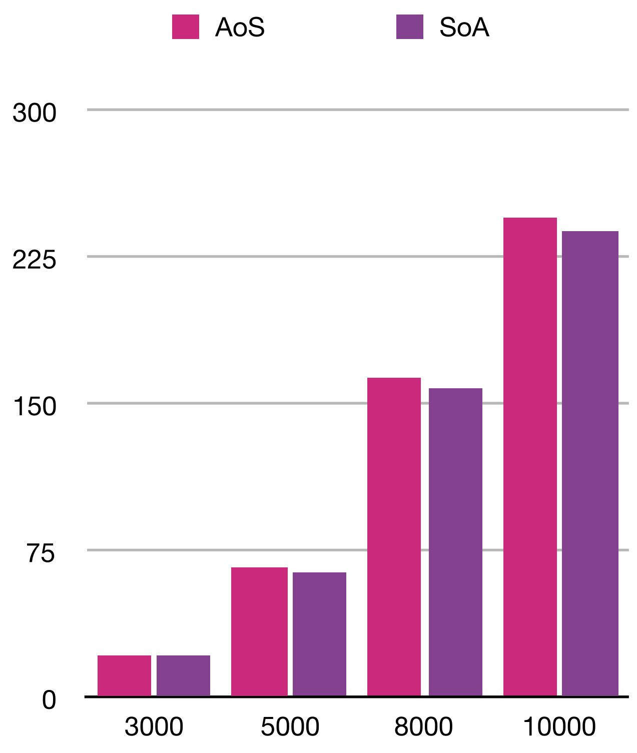
```
if (body_index_x < nbody_count && body_index_y < nbody_count) {

    float body_x = d_soa_body_pointer.x[body_index_y];
    float body_y = d_soa_body_pointer.y[body_index_y];
    float body_vx = d_soa_body_pointer.vx[body_index_y];
    float body_vy = d_soa_body_pointer.vy[body_index_y];
    float body_m = d_soa_body_pointer.m[body_index_y];

    if (body_index_y % BLOCK_SIZE == 0) {
        buffer_x[tx] = d_soa_body_pointer.x[body_index_x];
        buffer_y[tx] = d_soa_body_pointer.y[body_index_x];
        buffer_m[tx] = d_soa_body_pointer.m[body_index_x];
    }

    __syncthreads();

    // force calculation.
    float dis_x = buffer_x[tx] - body_x;
    float dis_y = buffer_y[tx] - body_y;
    float magnitude = (float)sqrt((double)dis_x * dis_x + (double)dis_y * dis_y);
    float force_x = (G * buffer_m[tx] * body_m * dis_x) / (float)pow(((double)magnitude + (double)SOFTENING), 3.0 / 2);
    float force_y = (G * buffer_m[tx] * body_m * dis_y) / (float)pow(((double)magnitude + (double)SOFTENING), 3.0 / 2);
    float acc_x = force_x / body_m;
    float acc_y = force_y / body_m;

    atomicAdd(&d_acc_x[body_index_y], acc_x);
    atomicAdd(&d_acc_y[body_index_y], acc_y);
}
```

2.2 Performance:

Grid = 5, I = 1000	Num	AoS	SoA
Global memory	1000	2.339	2.431
	3000	21.769	21.352
	5000	66.221	64.194
Read-only memory	1000	2.337	2.52
	3000	21.777	21.141
	5000	66.211	63.882
Texture memory	1000	-	**2.403**
	3000	-	**21.150**
	5000	-	**61.441**
Shared memory	1000	2.515	2.732
	3000	22.004	21.772
	5000	66.746	63.565

According to the table above, using SoA layout and texture memory is less time-consuming, which is the same as N threads approach. The other details are similar to N threads, which I have mentioned above. The next figure displays the running time of AoS and SoA in global memory. According to the graph below, the running time is similar when num is not big enough. However, running time of AoS is longer than that of SoA when num is bigger than 5000. The reason might because of the occupancy of GPU.

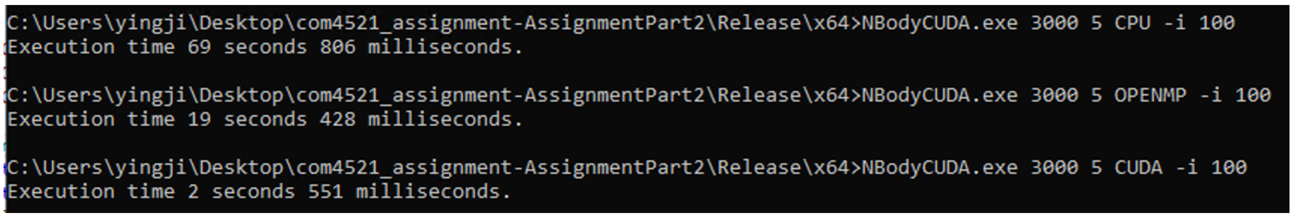




2.3 Result:

The next is the comparison between CPU, OPENMP and GPU.

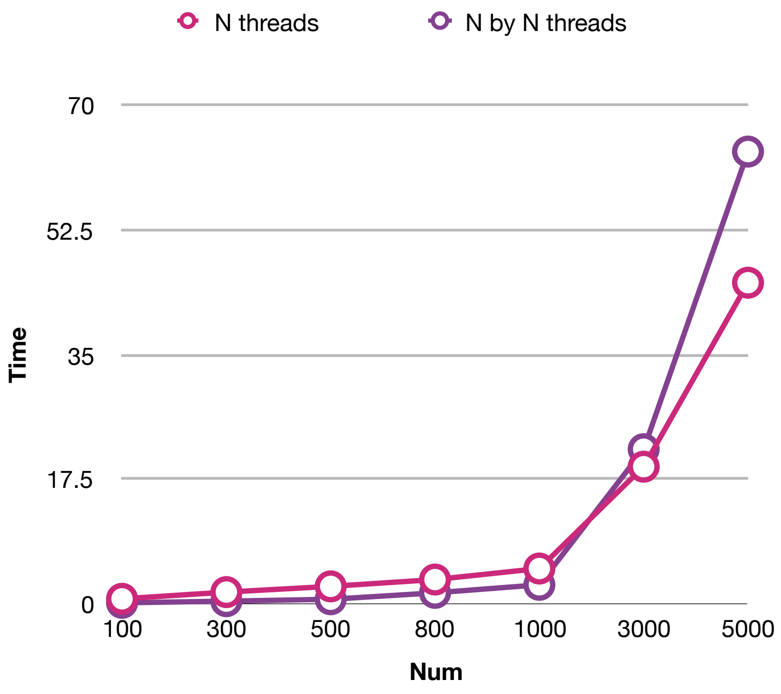
Num	CPU	OPENMP	CUDA
200	3.70	0.857	0.341
400	12.283	3.386	0.737
600	27.689	7.584	1.10
800	49.173	13.457	1.984



```
C:\Users\yingji\Desktop\com4521_assignment-AssignmentPart2\Release\x64>NBodyCUDA.exe 3000 5 CPU -i 100
Execution time 69 seconds 806 milliseconds.

C:\Users\yingji\Desktop\com4521_assignment-AssignmentPart2\Release\x64>NBodyCUDA.exe 3000 5 OPENMP -i 100
Execution time 19 seconds 428 milliseconds.

C:\Users\yingji\Desktop\com4521_assignment-AssignmentPart2\Release\x64>NBodyCUDA.exe 3000 5 CUDA -i 100
Execution time 2 seconds 551 milliseconds.
```

3. Comparison:

The next figure displays the performance between two variant strategies that were mentioned above. According to the result, N threads approach is faster than N by N threads when num is not big. However, the running time of the N by N threads will be increasing exponentially, which is much bigger than N threads. The reason might be that atomicAdd was used for calculating accelerated speed in N by N threads. In this case, more threads have to wait to write array, which greatly decrease the amount of paralleled threads. This phenomenon will be more serious with the increasing of amount of bodies.



4. Conclusion:

This report implements two variant threads approach including N threads and N by N threads and two memory access pattern (AoS and SoA) as well as four cacheing techniques including global memory, read-only memory, texture memory and shared memory. According to the discussion above, texture memory with SoA is less time-consuming in both approaches. N by N threads is less time consuming when num is less than a thousand. However, it will spend more time on big amount of bodies, which is worse than N threads. As a consequence, N threads will be used in this project.