1. Parallel Implementation

The CUDA mode use N*N threads and the N threads to compute the force between each planet.

Definite the Block and Grid as:

```
dim3 Block = { 256 };
dim3 Grid = { (unsigned)(N / 256) + 1 };
```

(1) When use N threads:

```
computeForce << <Grid, Block >> > (N, dfx, dfy, dplanet_soa->x, dplanet_soa->y,
dplanet_soa->m);

update << <Grid, Block >> > (N, dfx, dfy, dplanet_soa->x, dplanet_soa->y, dplanet_soa->vx,
dplanet_soa->vy);

computeHeat << <Grid, Block >> > (N, dplanet_soa->x, dplanet_soa->y, dheat, D);
```

`Use` tid thread to compute the all the forces acting on planet i in `computeForcePair` function.

can get thread id/ i by:

```
tid/i = blockIdx.x * blockDim.x + threadIdx.x;
```

Then use i to compute the all the forces acting on planet i, then save it in `fx[tid] and fy[tid]`, the length of `fx[tid] and fy[tid]]` is N.

(2) When use N*N threads:

```
computeForcePair << <N * N / 256 + 1, 256 >> > (N, dfmatx, dfmaty, dplanet_soa->x,
dplanet_soa->y, dplanet_soa->m, dfx, dfy);

update << <Grid, Block >> > (N, dfx, dfy, dplanet_soa->x, dplanet_soa->y, dplanet_soa->vx,
dplanet_soa->vy);

computeHeat << <Grid, Block >> > (N, dplanet_soa->x, dplanet_soa->y, dheat, D);
```

Use the tid thread to compute the force between planet i and planet j, in `computeForcePair` function
can get thread id by:

```
tid = blockIdx.x * blockDim.x + threadIdx.x;
```

can get i and j by:

```
int i = tid / num;
int j = tid % num;
```

Use i and j to calculate the force, then save it in `fmatx[tid] and fmaty[tid]`, the length of `fmatx[tid] and fmaty[tid]` is N*N.

the runtime of N*N threads and N threads for the I number of iterations:

	N thread	N*N thread
N=500, D=10, I=1000	2496ms	1130ms
N=500, D=10, I=2000	7640ms	1951ms
N=1000, D=10, I=100	1789ms	547ms
N=2000, D=10, I=100	3285ms	1579ms
N=10000, D=10, I=100	18212ms	31690ms

When the N is small, the runtime of N*N threads for the I number of iterations is smaller than N threads, because more threads calculate the force at the same time.

But when N is very large, the N*N threads is slower than N thread, I think there are two reasons:

(1) There are too many threads when use N*N threads, not enough cores, so threads need to queue and wait for cores
(2) N*N threads need more transmission data, however, the I/O speed is limited.

Because when N is smaller, N*N threads is faster, so I chose to use N*N threads.

2. Data in GPU memory

The CUDA mode use Arrays of Structures and the Structures of Arrays to represent data in GPU memory.
When use SOA, there are just one structure, there are 5 array pointers in this structure. Define structure type as:

```
struct nbody_soa {
        float* x, * y, * vx, * vy, * m;
        };
```

When use AOS, there are N structures, there are 5 float data in this structure. Define structure type as:

```
struct nbody{
        float x, y, vx, vy, m;
    };
```

the runtime of AOS and SOA for the I number of iterations:

	AOS	SOA
N=500, D=10, I=100; N*N thread	427ms	347ms
N=500, D=10, I=100; N thread	547ms	478ms
N=2000, D=10, I=100; N*N thread	1874ms	1579ms
N=2000, D=10, I=100; N thread	4512ms	3285ms

No matter what the values of N and D are, no matter how many threads are used, SOA is always faster.

SOA is more efficient than AOS, because continuous storage of frequently accessed data will greatly improve access speed.
For example, when we run :

```
for (int i = 0; i < num; i++)
{
        fx[tid] += fmatx[tid*num + i];
        fy[tid] += fmaty[tid*num + i];
}
```

`fmatx[tid*num + i]` and `fmatx[tid*num + i+1]`, the two data in one array is close, this improves our efficiency in extracting data.

3. The use of various GPU memory caches

In CUDA mode, I use constant, because the read is not written and the amount of data is small.
Define N and D as constant,

```
__constant__ float s = 2.0f;
__constant__ int N_gpu;
__constant__ int D_gpu;
```

then copy them to GPU

```
cudaMemcpyToSymbol(N_gpu, &N,sizeof(int));
cudaMemcpyToSymbol(D_gpu, &D, sizeof(int);
```

I did not use 2D/3D array or Normalisation/interpolation, I did not choose to use texture memory.
I did not use shared memory, because the data shared between threads is too large, and the capacity of share is small, so I do not think it is need to share data.

	do not use constant	use constant
N=1000, D=10, I=100; N*N thread	1423ms	1347ms
N=500, D=10, I=1000; N*N thread	1874ms	1654ms

Use constant is faster than without constant.It can effectively reduce memory bandwidth.

4. Avoid race conditions

(1) Activity Heat-map
I use N thread to calculate Activity Heat-map
Threads will visit h[ ] at the same time, so I use atomicAddfunction to avoid race conditions

```
atomicAdd(&h[y_index * d + x_index], 1.0f / num * d);
```

And the another part, when the point over map ,it will not be calculated in Activity Heat-map to avoid race conditions.

```
int tid = blockIdx.x * blockDim.x + threadIdx.x;
int x_index = (int)(x[tid] / (1.0 / d));
int y_index = (int)(y[tid] / (1.0 / d));
if (x_index < 0 || x_index >= d || y_index < 0 || y_index >= d) {
        return;
}
```

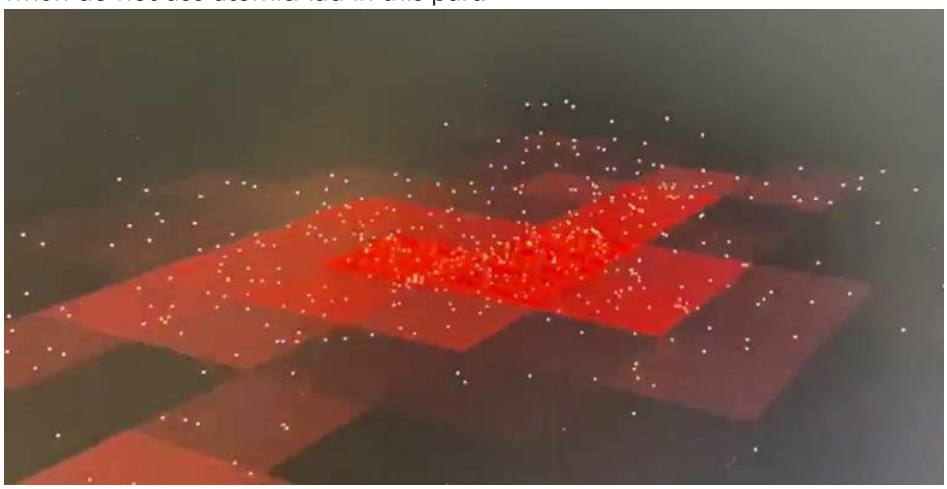
(2) computeForcePair function

I use N*N thread to calculate the all forces acting on one planet,

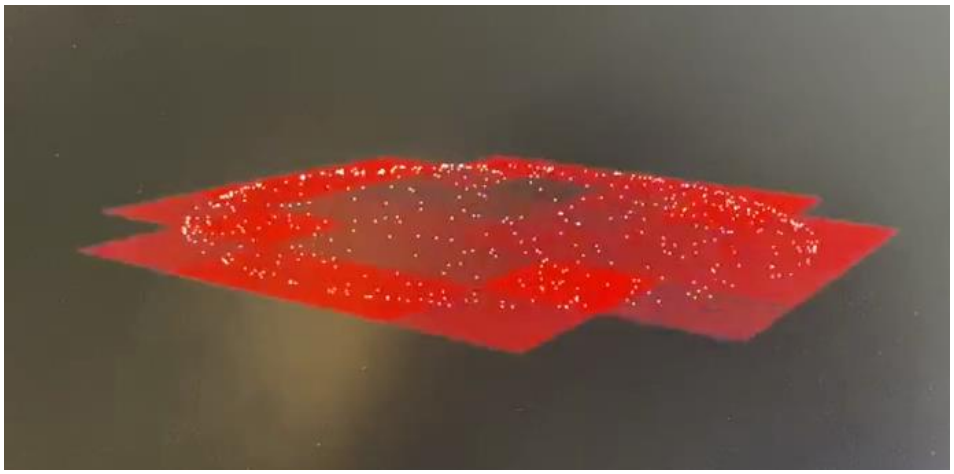
Threads will visit fx[ ] and fy[ ] at the same time, so I use atomicAddfunction to avoid race conditions.

After my comparison, I found that if this part do not use atomicAdd the visualization is not right.

when do not use atomicAdd in this part:



This is when use atomicAdd:



5.Improve the main N-Body simulation loop

In update and computeForcePair function

Before optimization,

```
coef = G * m[j]*m[i] / powf(distance + s2, 1.5);
atomicAdd(&fx[i], fmatx[tid]);
vx[tid] += dt * fx[tid]/ m[tid];
```

Here m can offset, After optimization:

```
coef = G * m[j] / powf(distance + s2, 1.5);
atomicAdd(&fx[i], fmatx[tid]);
vx[tid] += dt * fx[tid]/;
```

the runtime of AOS and SOA for the I number of iterations:

	Before optimization	After optimization
N=500, D=10, I=100; N*N thread	407ms	364ms
N=500, D=10, I=100; N thread	595ms	474ms
N=2000, D=10, I=100; N*N thread	1712ms	1445ms
N=2000, D=10, I=100; N thread	4478ms	3124ms

Because it reduces the amount of computation, the efficiency is improved.

6. Performance difference between all three version of the code

the runtime of CUDA, CPU and OPENMP for the I number of iterations:

	CUDA	CPU	OPENMP
N=500, D=10,I=100; N*N thread	407ms	645ms	487ms
N=2000,D=10,I=100;	1712ms	2187ms	2047ms
N=500,D=10,I=1000;	645ms	854ms	784ms

CUDA's performance is always the best, followed by OpenMP. Opoenmp is run by multithreading, CUDA is run by a large number of cores.