Mini Project Report – CS398S21-A.SG

Ang Hiap Lee, a.hiaplee, 390000318

Chloe Lim Jia-Han, j.lim, 440003018

1. Summary

We implemented a simple demo of an N-body Simulation with both CPU and GPU implementations. Differences in performance output between the two implementations were recorded through runs of the simulation using the following specifications:

CPU	Intel Xeon E6-1620v4
GPU	Nvidia GeForce GTX 1060 6 GB
RAM	16GB

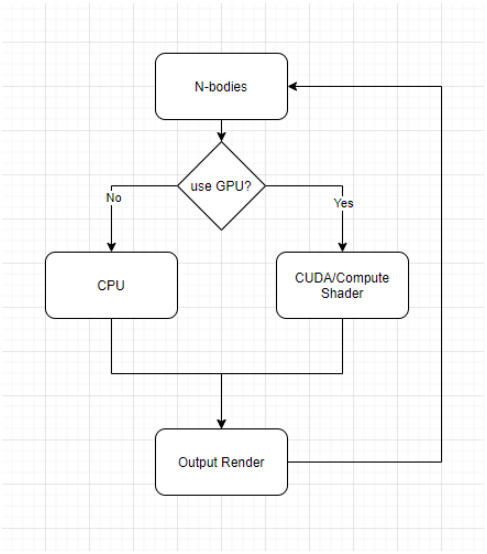
2. Background

a. Experiment Context:

In physics and astronomy, an N-body simulation is a simulation of a dynamic system of particles, usually under the influence of physical forces like gravity.

In the mini project, we will simulate the behaviour of the bodies represented by particles in the form of a demo. Demo will be implemented ideally with a GUI that allows the user to choose between a CPU sequential update and a parallel GPU update that is handled using either CUDA or compute shaders.

We should be able to observe a performance difference (frame time) between both implementations due to the complexity of the calculations.



(Application flow)

b. Algorithm Description/Implementation:

Both the CPU and the GPU end follows the same N-body interaction simulation algorithm, the Direct Sum approach acting on a 2-dimensional universe with a predetermined N number of bodies. Despite its time complexity of $O(N^2)$, it is explicit in its calculations towards applying the gravitational force on each particle onto every other particle. Minimal mathematical changes that can contribute towards a beautification of output was done to allow for more emphasis to be placed on the difference between the CPU and the GPU end. Constants such as the gravitational constant has been changed to suit our current implementation to allow a more visually clear output.

Of the N objects, the first object will be separately initialised to spawned in the centre of the world with no initial velocity and a predetermined solar mass of 0.00198892. Every other object will have a randomly generated position within a unit circle around the central mass. Using this initial position, the orbital speed around the central body (magnitude of velocity vector) will be calculated using the following base function:

$$\sqrt{\frac{GravitationalConstant * SolarMass}{Distance}}$$

Further elaboration of this implementation can be found in the function InitialForceCalcNormalObject.

```
float Application::InitialForceCalcNormalObject(const glm::vec2& pv)
{
    // math time
    /* This function will determine the magnitude of the initial velocity
       vector of the body, given the position within the unit circle it was
       initially generated in.

       Given the universe as a unit circle, the central body is in the origin.
       The length of pv gives an idea of how far the object is from the center,
       with the furtherst it could be being sqrt(0.5) units away, given that
       we are treating the bodies to be within a ciruclar orbit (circle of radius
       0.5).

       This implementation plays on the function to calculate orbital speed
       around the central body. The original function is
       sqrtf(G * solarMass / distance). What we have done was include an
       additional constant coefficient of 7500 / (N * N), to allow for a more
       suitable approximation of speed given the number of objects in the
       world. This coefficient is a custom implementation for our project
       and has a primary role of beautifying the thing over any actual scientific
       application. The rationale was to take a large enough value and scale it
       down by the square of the number of bodies in the system. */
    return sqrtf(G * (7500.0f / (float(N * N))) * solarMass / glm::length(pv));
}
```

Due to the use of a 2D system, only a theta angle will be computed from the generated position within the unit circle. This theta angle will be scaled accordingly by the signs of each element within the position vector, the multiplied by the orbital speed to get the orbital velocity. This orbital velocity has a fifty percent chance of being negated to allow for further variation of orientation.

Each subsequent body's mass after the central body will be anywhere between 0 to 10 times that of the central body mass.

The algorithm itself involves two loops. One with a complexity of $O(N^2)$ and another with a complexity of $O(N)$. The first loop is the actual Direct Sum algorithm. Each object will have the force acting on it nulled before its update. This results in the stored force of object being localised to the frame it is in. After having its force reset, it is them recalculated by comparing it against every other body in the universe. This comparison is

used to calculate the force acting on the object from the object it is compared to. This step involves the calculation of the gravitational force:

$$\frac{G * mass_1 * mass_2}{distanceBetweenObjects^2}$$

And the calculation of the force vector to be applied:

$$GravitationalForce * \widehat{\text{UnitDisplacement}}$$

With regards to the calculation of the gravitational force, we have chosen to follow typical astrophysical simulations and ignore collisions between bodies. Should two bodies collied, the distance between them would be 0, and the function would have a division by 0 error within it. To avoid this problem, following standard practice, a softening factor is applied within the denominator.

Further elaboration of this implementation can be found in the function AddForceNormalObject.

```
void Application::AddForceNormalObject(NormalObject& obja, NormalObject& objb)
{
    // calculating the force objb acts on obja and applying it
    static const float softeningsq = 9.0f; /* force no division by 0 */
    glm::vec2 difference = objb.translate - obja.translate; /* difference in position */
    float magnitudesq = difference.x * difference.x + difference.y * difference.y; /* square of magnitude */

    // math time
    /* F here represents the gravitational force of the two objects. The formula
       to calculate F is Gravitational_Constant * m_1 * m_2 / distance^2. The following
       line implements the above equation. Per object, Gravitaional_Constant * m_1
       will be constant, with m_2 and distance^2 varying per the object it is
       interacting with.

       The magnitude of a displacement vector (difference vector between two objects)
       is the distance between them. To get distance^2, we squared the magnitude.

       Our implementation includes an additional variable in the denominator called
       softeningsq, used to prevent a division by 0. This division by 0 in the original
       function is only there when the two objects are in the same position, which
       results in the displacement vector being a null vector. Our implementation aims
       to be as accurate as possible, such as ignoring collisions between bodies (which
       is actually a thing in astrophysical simulations it is really cool). Thus the
       ultimate addition of the softening factor. */
    float F = (G * obja.mass * objb.mass) / (magnitudesq + softeningsq);
    // math time part 2
    /* The final force applied onto the object will be the force calculated above (F)
       multiplied by the unit vector of the displaceent vector. This force, as stated
       above, represents the gravitational force both objects are acting on each other.
       Thus the application of the force will be having both objects move towards each
       other by the magnitude of the force (F) in each other's direction (unit vector
       of the displacement vector). */
    obja.force += F / sqrtf(magnitudesq) * difference; /* apply the forcee acting on obja from objb */
}
```

The final loop is merely the updating of the object velocity, modified by the force acting on the object multiplied by the delta time of the frame, and the updating of the object position, modified by the velocity of the object multiplied by the delta time of the frame.

Further elaboration of this implementation can be found in the function UpdateNormalObject.

```
void Application::UpdateNormalObject(NormalObject& obj)
{
    // basic application of force
        /* apply the force acting on the body onto the velocity
          second differential application applied to first differential */
    obj.velocity += (float)_deltaTime * obj.force;
    /* transform the body with respect to movement defined
      by newly updated velocity from above (simple moving) */
    obj.translate += (float)_deltaTime * glm::vec3(obj.velocity.x, obj.velocity.y, 0.0f);
}
```

3. Approach

a. Technologies used:

A simple OpenGL rendering framework was developed using GLFW and Dear ImGui. A simple GLSL shader was written to take advantage of OpenGL's instancing to feature (`glDrawElementsInstanced`) to render multiple instances of a mesh using a buffer of multiple transformation matrices.

In both implementations the VBO is used to stored transformation matrices of the objects.

b. Linking CUDA kernel with OpenGL:

Using CUDA's Graphics Interoperability API, we were able to perform I/O operations on OpenGLVBOs (vertex buffer object) directly inside CUDA kernel functions.

After the creation of the VBO using OpenGL functions, the buffer would need to be registered with CUDA as a graphics resource like below:

```
cudaGraphicsGLRegisterBuffer(&MyResourcePtr, VBO_ID , cudaGraphicsMapFlagsNone);
```

A kernel function would be need surrounded with `cudaGraphicsMapResources` and `cudaGraphicsUnmapResources` to allow CUDA to perform I/O operations on the resources specified.

```
cudaGraphicsMapResources(1, resources);
compute_cuda
(
    d_Objects, resources[0], _objects.size(), DimBlock, DimGrid2, N, G, _deltaTime, useBaseColor
);
cudaGraphicsUnmapResources(1, resources);
```

(example)

Within the kernel, the user is required to call `cudaGraphicsResourceGetMappedPointer` to retrieve the address of resource on the device.

```
RenderData* cuda_data;
size_t size;
cudaGraphicsResourceGetMappedPointer((void**)(&cuda_data), &size, resource);
```

With the above steps the user can perform I/O operations in the specified resource, in our project's case was writing directly into the VBO that holds the transformation matrices which would be used by the GLSL shaders to render.

c. GPU implementation

First, we copied the initial data of the N-bodies into device memory and map the VBO with CUDA, this process is done every time the N number is changed in the GUI.

```
auto size = _objects.size() * sizeof(NormalObject);

if (d_Objects)
{
    checkCudaErrors(cudaFree(d_Objects));
}

checkCudaErrors(cudaMalloc((void**)&d_Objects, size));
checkCudaErrors(cudaMemcpy(d_Objects, copy_objs.data(), size, cudaMemcpyHostToDevice));
checkCudaErrors(cudaDeviceSynchronize());
checkCudaErrors(cudaGraphicsGLRegisterBuffer(resources, objs.transforms, cudaGraphicsMapFlagsNone));
```

At this point we have the Object Data in the device global memory and the Render Data in the VBO data buffer.

```
struct NormalObject
{
    glm::vec3 scale = {1.0f, 1.0f, 1.0f};
    float rotate = 0.0f;
    glm::vec3 translate = { 0.0f, 0.0f, 0.0f }; // position

    glm::vec2 velocity = { 0.0f, 0.0f };
    glm::vec2 force = { 0.0f, 0.0f };
    float mass = 1.0f;

    float color[4] = { 1.0f,1.0f,1.0f,1.0f };
    float basecolor[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
    float altcolor[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
};

struct RenderData
{
    glm::mat4 transform;
    float color[4] = { 1.0f,1.0f,1.0f,1.0f };
};
```

(Two structs we used in the simulation)

With CUDA now being able to write into the VBO, the CPU sequential algorithm was then broken down into different parts in the GPU kernel. Looking at the UpdateNBody function, we can easily break the outer for loop into simple GPU O(1) implementation.

```
void Application::UpdateNBody()
{
    for (size_t i = 0; i < N; ++i)
    {
        _objects[i]->force = glm::vec2(0.0f, 0.0f); /* resetting the force acting on the body */
        // N squared algo
        for (size_t j = 0; j < N; ++j) /* handle interaction with every object */
            if (i != j) AddForceNormalObject(*_objects[i], *_objects[j]);
    }
    for (size_t i = 0; i < N; ++i)
    {
        UpdateNormalObject(*_objects[i]); /* handle the movement */
    }
}
```

Breaking the outer loop into the main kernel function and helper functions into device only functions we arrived at a simple O(N) implementation of the algorithm.

```
__device__ void GPU_AddForceNormalObject(NormalObject& obja, NormalObject& objb, float G)
{
    static float softeningsq = 3.0f * 3.0f;
    glm::vec2 difference = objb.translate - obja.translate;
    float magnitudesq = difference.x * difference.x + difference.y * difference.y;
    float F = (G * obja.mass * objb.mass) / (magnitudesq + softeningsq);
    obja.force += F / sqrtf(magnitudesq) * difference;
}

__device__ void UpdateNormalObject(NormalObject& obj, float _deltaTime)
{
    obj.velocity += (float)_deltaTime * obj.force;
    obj.translate += (float)_deltaTime * glm::vec3(obj.velocity.x, obj.velocity.y, 0.0f);
}
```

```
__global__ void compute_kernel(
    NormalObject* d_Objects,
    RenderData* vbo_data,
    size_t N,
    float G,
    float _deltaTime,
    bool _useBaseColor
)
{
    uint x = blockIdx.x * blockDim.x + threadIdx.x;

    if (x < N)
    {
        RenderData& data = vbo_data[x];
        NormalObject* obj = &(d_Objects[x]);


        obj->force = glm::vec2(0.0f, 0.0f);
        for (size_t j = 0; j < N; ++j)
            if (x != j) GPU_AddForceNormalObject(d_Objects[x], d_Objects[j], G);


        UpdateNormalObject(d_Objects[x], _deltaTime);

        data.transform = glm::mat4(1.0f);
        data.transform = glm::scale(data.transform, obj->scale);
        data.transform = glm::rotate(data.transform, glm::radians(obj->rotate), glm::vec3(0.0, 0.0, 1.0));
        data.transform = glm::translate(data.transform, obj->translate);

        if (_useBaseColor)
        {
            data.color[0] = obj->basecolor[0];
            data.color[1] = obj->basecolor[1];
            data.color[2] = obj->basecolor[2];
            data.color[3] = obj->basecolor[3];
        }
        else
        {
            data.color[0] = obj->altcolor[0];
            data.color[1] = obj->altcolor[1];
            data.color[2] = obj->altcolor[2];
            data.color[3] = obj->altcolor[3];
        }
    }
}
```

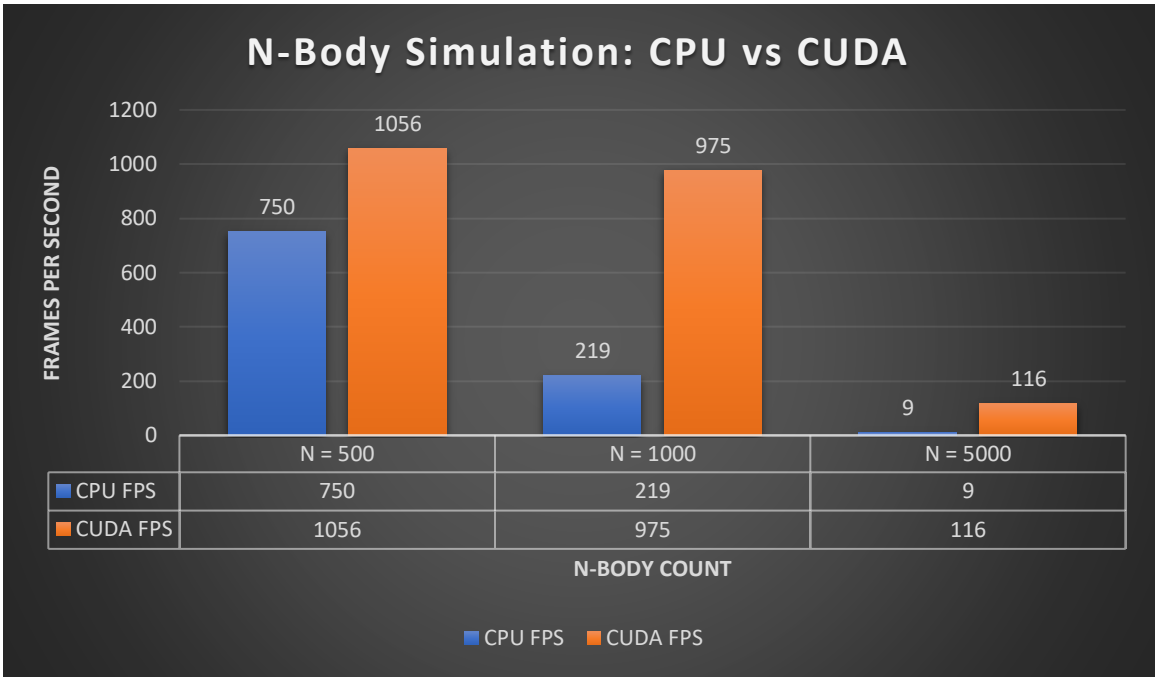
Using a (N x 1) grid & block each thread is able access the elements required for its calculation at O(1) access now. Shared memory was not used here as threads need to write into both buffers after calculation, using of shared memory for read only access is not ideal. Finally, we just call a helper function that launches the kernel.

```
void compute_cuda(
    NormalObject* d_Objects,
    cudaGraphicsResource* resource,
    uint max_objects,
    dim3& DimBlock,
    dim3& DimGrid2,
    size_t N,
    float G,
    float _deltaTime,
    bool _useBaseColor
)
{
    RenderData* cuda_data;
    size_t size;
    cudaGraphicsResourceGetMappedPointer((void**)(&cuda_data), &size, resource);

    compute_kernel << < DimGrid2, DimBlock >> >
        (
            d_Objects,
            cuda_data,
            N,
            G,
            _deltaTime,
            _useBaseColor
        );
    getLastCudaError("compute_kernel failed\n");
    cudaDeviceSynchronize();
}
```

4. Performance Analysis

The experiment was conducted using 3 different N values (500, 1000 , 5000). Raw Data is stated recorded down below, it is recommended to the run simulation to experience hands on.



We observed significant performance boost when the N count goes into non-trivial numbers with the CPU FPS dropping into one digit range when N = 5000 while CUDA manages to maintain more than a 100 frames per second.

Due the complexity of the different implementations of $O(N^2)$ for CPU and $O(N)$ for CUDA, it is obvious that the CUDA implementation will scale better than its CPU counterpart.

In addition, the CPU implementation requires copying of transformation matrices from host buffer to device VBO buffer every frame, which can create a significant overhead when the N count becomes huge while the CUDA implementation is able to write directly into the VBO buffer.

5. Work Division

Task	Handled By
N-body algorithm Research	Chloe Lim Jia-Han, Ang Hiap Lee
CPU implementation	Chloe Lim Jia-Han
GPU implementation	Chloe Lim Jia-Han, Ang Hiap Lee
CUDA Graphics Interoperability	Ang Hiap Lee
Base framework	Chloe Lim Jia-Han, Ang Hiap Lee

6. References

https://rauwendaal.net/2011/02/10/how-to-use-cuda-3-0s-new-graphics-interoperability-api-with-opengl/

https://www.3dgep.com/opengl-interoperability-with-cuda/

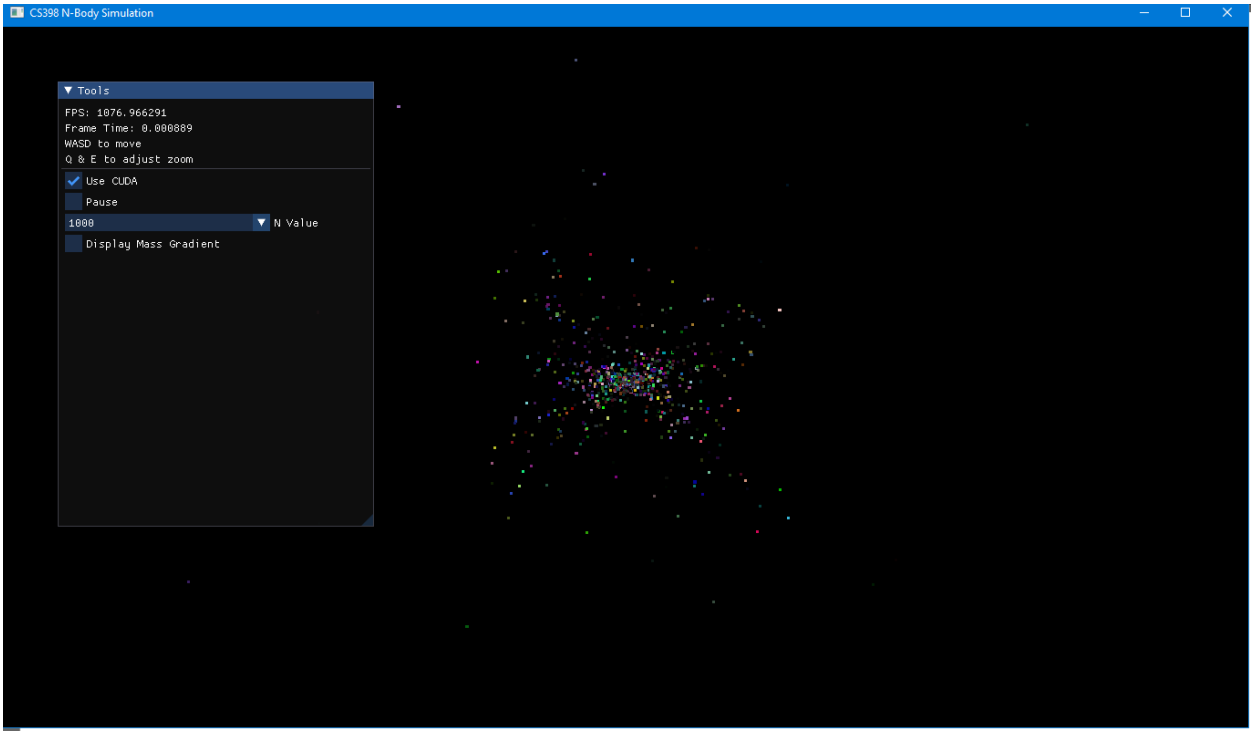
https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-31-fast-n-body-simulation-cuda

https://www.astronomynotes.com/gravappl/s8.htm

https://astronomy.swin.edu.au/cosmos/g/Gravitational+Constant#:~:text=In%20SI%20units%2C%20G%20has,therefore%20subject%20to%20%E2%80%9Cgravity%E2%80%9D.

https://www.evl.uic.edu/sjames/cs525/project2.html

Appendix



(In-App Screenshot)