1. Introduction

This project is about to simulate 10,000 to 1,000,000 asteroids in space. Those asteroids will start with an initial velocity and position, and they will be affected by the gravitational force of the sun and near asteroids. The simulation will be parametrizable, which means we will be able the number of asteroids, how many frames/repetitions, and how much time is spent between frames. Also, we can restart the simulation and continue it once.

The most important thing you have to have clear is what task of your problem we are going to do in the GPU or in the CPU. We are going to do all the calculations for the new position on the GPU and the rest on the CPU.

CUDA.

With CUDA we only have to tell which one of our devices are we going to use. We are going to allocate memory for all the data we are going to need in our simulation, including mass, position, velocity, force, acceleration, time and quadtree information for each of them. All of the initialization we are going to make in a function will be called only once. Also as we are allocating memory, we have to free them, so we will have another function to make that.

CUDA INITIALIZATION

```
cudaError_t cudaStatus = cudaSetDevice(0);
if (cudaSuccess != cudaStatus) {
  fprintf(stderr, "cudaSetDevice failed!  Do you have a CUDA-capable GPU installed?");
  goto Error;
}


//Allocation of the device memory
cudaStatus = cudaMalloc((void**)&asteroidDataGPU.dev_t, sizeof(float));
if (cudaSuccess != cudaStatus) {
  fprintf(stderr, "cudaMalloc failed!");
  goto Error;
}

cudaStatus = cudaMalloc((void**)&asteroidDataGPU.dev_m, size * sizeof(float));
if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMalloc failed!");
  goto Error;
}

cudaStatus = cudaMalloc((void**)&asteroidDataGPU.dev_x, size * sizeof(double));
if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMalloc failed!");
  goto Error;
}
                ■
                ■
                ■
cudaStatus = cudaMalloc((void**)&asteroidDataGPU.dev_outputData, size * kMaxNumberAsteroid * sizeof(int));
if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMalloc failed!");
  goto Error;
}

cudaStatus = cudaMalloc((void**)&asteroidDataGPU.dev_outputSize, size * sizeof(int));
if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMalloc failed!");
  goto Error;
}
```

Set the device we want to work in.

Allocate all needed memory

CUDA MEM RELEASE

```
void endAsteroidBeltCuda(AsteroidBeltDeviceData& asteroidDataGPU) {

  cudaFree(asteroidDataGPU.dev_t);
  cudaFree(asteroidDataGPU.dev_m);

  cudaFree(asteroidDataGPU.dev_x);
  cudaFree(asteroidDataGPU.dev_y);
  cudaFree(asteroidDataGPU.dev_z);

  cudaFree(asteroidDataGPU.dev_vx);
  cudaFree(asteroidDataGPU.dev_vy);
  cudaFree(asteroidDataGPU.dev_vz);

  cudaFree(asteroidDataGPU.dev_ax);
  cudaFree(asteroidDataGPU.dev_ay);
  cudaFree(asteroidDataGPU.dev_az);

  cudaFree(asteroidDataGPU.dev_fx);
  cudaFree(asteroidDataGPU.dev_fy);
  cudaFree(asteroidDataGPU.dev_fz);

  cudaFree(asteroidDataGPU.dev_outputData);
  cudaFree(asteroidDataGPU.dev_outputSize);
}
```

In addition, we are going to have another function in which we will call all the kernels, measure times, and copy data from host to divide and device to host. Also, we are going to calculate the number of blocks of 1024 we need for our simulation.

1st copy host to device.

```
cudaStatus = cudaMemcpy(asteroidDataGPU->dev_x, x, size * sizeof(double), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMemcpy failed!");
  goto Error;
}

cudaStatus = cudaMemcpy(asteroidDataGPU->dev_y, y, size * sizeof(double), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMemcpy failed!");
  goto Error;
}

cudaStatus = cudaMemcpy(asteroidDataGPU->dev_z, z, size * sizeof(double), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMemcpy failed!");
  goto Error;
}
```

2nd Call the kernels.

```
//Force Kernel
{
  // Launch a kernel on the GPU with one thread for each element.
  forceKernel << <numberBlocks, kNumberThreadPerBlock >> > (asteroidDataGPU->dev_m,
  asteroidDataGPU->dev_fx, asteroidDataGPU->dev_fy, asteroidDataGPU->dev_fz,
  asteroidDataGPU->dev_x, asteroidDataGPU->dev_y, asteroidDataGPU->dev_z, size,
  asteroidDataGPU->dev_outputData, asteroidDataGPU->dev_outputSize);


  // Check for any errors launching the kernel
  cudaStatus = cudaGetLastError();
  if (cudaSuccess != cudaStatus) {
    fprintf(stderr, "forceKernel launch failed: %s\n", cudaGetErrorString(cudaStatus));
    goto Error;
  }
  // ...
  cudaStatus = cudaDeviceSynchronize();
  if (cudaSuccess != cudaStatus) {
    fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launching forceKernel!\n", cudaStatus);
    goto Error;
  }
```

3rd Copy back device to host.

```
  cudaStatus = cudaMemcpy(x, asteroidDataGPU->dev_x, size * sizeof(double), cudaMemcpyDeviceToHost);
  if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
  }

  cudaStatus = cudaMemcpy(y, asteroidDataGPU->dev_y, size * sizeof(double), cudaMemcpyDeviceToHost);
  if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
  }

  cudaStatus = cudaMemcpy(z, asteroidDataGPU->dev_z, size * sizeof(double), cudaMemcpyDeviceToHost);
  if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
  }
```

OpenCL.

OpenCL is the second API to work with the GPU. With this one, we also are going to have three parts, initialization, update and end. The initialization is the hardest part as OpenCL lot of preparation before starting.

1. We search for available platforms and link them to a device.
2. Create our context using the previous device.
3. Create a program with our .cl file
4. Build it.
5. Create the queue.
6. Creates kernels (4).
7. Creates GPU memory, and buffers. One for each data we want to be in the GPU, position, velocity, acceleration, force, time, and quadtree information.
8. Bind the arguments of the kernels.

```c
/* Identify a platform */
{
  CLStatus = clGetPlatformIDs(1, &platform, NULL);
  if (CLStatus < 0) {
    perror("Couldn't identify a platform");
    exit(1);
  }

  //Access a device
  //GPU
  CLStatus = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
  if (CL_DEVICE_NOT_FOUND == CLStatus) {
    //Try CPU
    CLStatus = clGetDeviceIDs(platform, CL_DEVICE_TYPE_CPU, 1, &device, NULL);
  }

  if (CLStatus < 0) {
    perror("Couldn't access any device");
    exit(1);
  }
}
```

1st

```c
context = clCreateContext(NULL, 1, &device, NULL, NULL, &CLStatus);
if (CLStatus < 0) {
  perror("Couldn't create a context");
  exit(1);
}
```

2nd

```c
program_handle = fopen(kProgramFile, "r");
if (NULL == program_handle) {
  perror("Couldn't find the program file");
  exit(1);
}

fseek(program_handle, 0, SEEK_END);
program_size = ftell(program_handle);
rewind(program_handle);
program_buffer = (char*)malloc(program_size + 1);
program_buffer[program_size] = '\0';

fread(program_buffer, sizeof(char), program_size, program_handle);
fclose(program_handle);

program = clCreateProgramWithSource(context, 1,
  (const char**)&program_buffer, &program_size, &CLStatus);

if (CLStatus < 0) {
  perror("Couldn't create the program");
  exit(1);
}

free(program_buffer);
```

3rd

```c
CLStatus = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
if (CLStatus < 0) {
  // Find size of log and print to std outpur
  clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG,
    0, NULL, &log_size);

  program_log = (char*)malloc(log_size + 1);
  program_log[log_size] = '\0';
  clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG,
    log_size + 1, program_log, NULL);
  printf("%s\n", program_log);

  free(program_log);
  exit(1);
}
```

4th

```c
queue = clCreateCommandQueue(context, device, 0, &CLStatus);
if (CLStatus < 0) {
  perror("Couldn't create a command queue");
  exit(1);
};
```

5th

```c
/* Create a kernel */
{
  forceKernel = clCreateKernel(program, "updateAsteroidForce", &CLStatus);
  if (CLStatus < 0) {
    perror("Couldn't create a updateAsteroidForce");
    exit(1);
  };

  accelerationKernel = clCreateKernel(program, "updateAsteroidAcceleration", &CLStatus);
  if (CLStatus < 0) {
    perror("Couldn't create a updateAsteroidAcceleration");
    exit(1);
  };

  velocityKernel = clCreateKernel(program, "updateAsteroidVelocity", &CLStatus);
  if (CLStatus < 0) {
    perror("Couldn't create a updateAsteroidVelocity");
    exit(1);
  };

  positionKernel = clCreateKernel(program, "updateAsteroidPosition", &CLStatus);
  if (CLStatus < 0) {
    perror("Couldn't create a updateAsteroidPosition");
    exit(1);
  };
}
```

6th

```c
//Mass & time
massBuffer = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
  bAsteroids * sizeof(float), asteroidData.m, &CLStatus);

timeBuffer = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
  sizeof(float), &timeStep_, &CLStatus);

//Position
positionXBuffer = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
  bAsteroids * sizeof(double), asteroidData.x, &CLStatus);

positionYBuffer = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
  bAsteroids * sizeof(double), asteroidData.y, &CLStatus);

positionZBuffer = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
  bAsteroids * sizeof(double), asteroidData.z, &CLStatus);

//Velocity
velocityXBuffer = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
  bAsteroids * sizeof(float), asteroidData.vx, &CLStatus);

velocityYBuffer = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
  bAsteroids * sizeof(float), asteroidData.vy, &CLStatus);

velocityZBuffer = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
  bAsteroids * sizeof(float), asteroidData.vz, &CLStatus);

//Acceleration
accelerationXBuffer = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
  bAsteroids * sizeof(float), asteroidData.vz, &CLStatus);

accelerationYBuffer = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
  bAsteroids * sizeof(float), asteroidData.vz, &CLStatus);

accelerationZBuffer = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
  bAsteroids * sizeof(float), asteroidData.vz, &CLStatus);

//Force
forceXBuffer = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
  bAsteroids * sizeof(float), asteroidData.vz, &CLStatus);

forceYBuffer = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
  bAsteroids * sizeof(float), asteroidData.vz, &CLStatus);

forceZBuffer = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
  bAsteroids * sizeof(float), asteroidData.vz, &CLStatus);

outputDataBuffer = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
  bAsteroidsPerAsteroid * bAsteroids * sizeof(int), outputAsteroids, &CLStatus);

outputSizeBuffer = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
  bAsteroids * sizeof(int), indexOutputAsteroids, &CLStatus);

if (CLStatus < 0) {
  perror("Couldn't create a buffer");
  exit(1);
};
```

7th

```c
//Bind kernel arguments
{
  //Force Kernel
  CLStatus = clSetKernelArg(forceKernel, 0, sizeof(cl_mem), &massBuffer); // <======INPUT
  CLStatus |= clSetKernelArg(forceKernel, 4, sizeof(cl_mem), &positionXBuffer); // <======INPUT
  CLStatus |= clSetKernelArg(forceKernel, 5, sizeof(cl_mem), &positionYBuffer); // <======INPUT
  CLStatus |= clSetKernelArg(forceKernel, 6, sizeof(cl_mem), &positionZBuffer); // <======INPUT
  CLStatus |= clSetKernelArg(forceKernel, 7, sizeof(cl_mem), &outputDataBuffer); // <======INPUT
  CLStatus |= clSetKernelArg(forceKernel, 8, sizeof(cl_mem), &outputSizeBuffer); // <======INPUT
  CLStatus |= clSetKernelArg(forceKernel, 1, sizeof(cl_mem), &forceXBuffer); // <======OUTPUT
  CLStatus |= clSetKernelArg(forceKernel, 2, sizeof(cl_mem), &forceYBuffer); // <======OUTPUT
  CLStatus |= clSetKernelArg(forceKernel, 3, sizeof(cl_mem), &forceZBuffer); // <======OUTPUT


  //Acceleration Kernel
  CLStatus = clSetKernelArg(accelerationKernel, 0, sizeof(cl_mem), &massBuffer); // <======INPUT
  CLStatus |= clSetKernelArg(accelerationKernel, 1, sizeof(cl_mem), &forceXBuffer); // <======INPUT
  CLStatus |= clSetKernelArg(accelerationKernel, 2, sizeof(cl_mem), &forceYBuffer); // <======INPUT
  CLStatus |= clSetKernelArg(accelerationKernel, 3, sizeof(cl_mem), &forceZBuffer); // <======INPUT
  CLStatus |= clSetKernelArg(accelerationKernel, 4, sizeof(cl_mem), &accelerationXBuffer); // <======OUTPUT
  CLStatus |= clSetKernelArg(accelerationKernel, 5, sizeof(cl_mem), &accelerationYBuffer); // <======OUTPUT
  CLStatus |= clSetKernelArg(accelerationKernel, 6, sizeof(cl_mem), &accelerationZBuffer); // <======OUTPUT


  //Velocity Kernel
  CLStatus = clSetKernelArg(velocityKernel, 6, sizeof(cl_mem), &timeBuffer); // <======INPUT
  CLStatus |= clSetKernelArg(velocityKernel, 3, sizeof(cl_mem), &accelerationXBuffer); // <======INPUT
  CLStatus |= clSetKernelArg(velocityKernel, 4, sizeof(cl_mem), &accelerationYBuffer); // <======INPUT
  CLStatus |= clSetKernelArg(velocityKernel, 5, sizeof(cl_mem), &accelerationZBuffer); // <======INPUT
  CLStatus |= clSetKernelArg(velocityKernel, 0, sizeof(cl_mem), &velocityXBuffer); // <======OUTPUT
  CLStatus |= clSetKernelArg(velocityKernel, 1, sizeof(cl_mem), &velocityYBuffer); // <======OUTPUT
  CLStatus |= clSetKernelArg(velocityKernel, 2, sizeof(cl_mem), &velocityZBuffer); // <======OUTPUT


  //Position Kernel
  CLStatus = clSetKernelArg(positionKernel, 9, sizeof(cl_mem), &timeBuffer); // <======INPUT
  CLStatus |= clSetKernelArg(positionKernel, 3, sizeof(cl_mem), &velocityXBuffer); // <======INPUT
  CLStatus |= clSetKernelArg(positionKernel, 4, sizeof(cl_mem), &velocityYBuffer); // <======INPUT
  CLStatus |= clSetKernelArg(positionKernel, 5, sizeof(cl_mem), &velocityZBuffer); // <======INPUT
  CLStatus |= clSetKernelArg(positionKernel, 6, sizeof(cl_mem), &accelerationXBuffer); // <======INPUT
  CLStatus |= clSetKernelArg(positionKernel, 7, sizeof(cl_mem), &accelerationYBuffer); // <======INPUT
  CLStatus |= clSetKernelArg(positionKernel, 8, sizeof(cl_mem), &accelerationZBuffer); // <======INPUT
  CLStatus |= clSetKernelArg(positionKernel, 0, sizeof(cl_mem), &positionXBuffer); // <======OUTPUT
  CLStatus |= clSetKernelArg(positionKernel, 1, sizeof(cl_mem), &positionYBuffer); // <======OUTPUT
  CLStatus |= clSetKernelArg(positionKernel, 2, sizeof(cl_mem), &positionZBuffer); // <======OUTPUT
}
```

8th

In the update part, we will copy the input data to the device, call the kernels and copy back the result from the device to the host. And in the end part, we will free all the memory and CL resources we allocate.

OpenCL update

```
clEnqueueWriteBuffer(queue, timeBuffer, CL_TRUE, 0, sizeof(float), &timeStep_,
  0, NULL, NULL);

clEnqueueWriteBuffer(queue, outputDataBuffer, CL_TRUE, 0,
                     kMaxAsteroidPerAsteroid * kMaxAsteroids * sizeof(int), outputAsteroids,
  0, NULL, NULL);

clEnqueueWriteBuffer(queue, outputSizeBuffer, CL_TRUE, 0,
                     kMaxAsteroids * sizeof(int), indexOutputAsteroids,
  0, NULL, NULL);
```

Write Input data

```
CLStatus = clEnqueueNDRangeKernel(queue, accelerationKernel, 1, NULL, &global_size,
  &local_size, 0, NULL, NULL);
if (CLStatus < 0) {
  perror("Couldn't enqueue the accelerationKernel");
  exit(1);
}
```

Call the kernels

```
CLStatus = clEnqueueReadBuffer(queue, positionXBuffer, CL_TRUE, 0,
  currentAsteroidsNumber * sizeof(double),
  asteroidData.x, 0, NULL, NULL);

CLStatus = clEnqueueReadBuffer(queue, positionYBuffer, CL_TRUE, 0,
  currentAsteroidsNumber * sizeof(double),
  asteroidData.y, 0, NULL, NULL);

CLStatus = clEnqueueReadBuffer(queue, positionZBuffer, CL_TRUE, 0,
  currentAsteroidsNumber * sizeof(double),
  asteroidData.z, 0, NULL, NULL);
```

Read the output

CL Memory Release

```
clReleaseCommandQueue(queue);
clReleaseProgram(program);
clReleaseContext(context);

clReleaseMemObject(forceXBuffer);
clReleaseMemObject(forceYBuffer);
clReleaseMemObject(forceZBuffer);
clReleaseMemObject(accelerationXBuffer);
clReleaseMemObject(accelerationYBuffer);
clReleaseMemObject(accelerationZBuffer);
clReleaseMemObject(velocityXBuffer);
clReleaseMemObject(velocityYBuffer);
clReleaseMemObject(velocityZBuffer);
clReleaseMemObject(positionXBuffer);
clReleaseMemObject(positionYBuffer);
clReleaseMemObject(positionZBuffer);
clReleaseMemObject(massBuffer);
clReleaseMemObject(timeBuffer);
```

For both APIs we are going to use four different kernels to make all the calculations to update the position of an asteroid.

1º In the first one we are going to calculate the force that an asteroid is affected by. To do that we use the formula: $F = G * ((m1 * m2)/(d * d))$ We use it with the sun and with the asteroids that are near to it. That information is obtained using the quadtree. With the formula, we obtain the magnitude so now we have to multiply the magnitude by its direction, in the case of the Sun it will be the one from the asteroid pointed to the Sun.

```
__global__ void forceKernel(float* m, float* fx, float* fy, float* fz,
                            double* x, double* y, double* z, const int size,
                            const int* asteroidID, const int* numberAsteroid){

int i = threadIdx.x;
int j = blockIdx.x;
i = i + j * blockDim.x;
if (size > i) {

 // Sun force
 double distance = x[i] * x[i] + y[i] * y[i] + z[i] * z[i];
 if (distance == 0) { distance = 0.01f; }
 float moduleF= 1.0f/-sqrt(distance);
 if (distance <= 100) { moduleF *= -1.0f; }
 float dirX = x[i] * moduleF;
 float dirY = y[i] * moduleF;
 float dirZ = z[i] * moduleF;

 float force = kG * ((kMSun * m[i]) / distance);

 fx[i] = force * dirX * 10000000.0f;
 fy[i] = force * dirY * 10000000.0f;
 fz[i] = force * dirZ * 10000000.0f;
```

CUDA

```
 int n = numberAsteroid[i];
 int offsetData = kMaxNumberAsteroidDev * i;
 for (int a = 0; a < n; ++a) {

  int asteroid = asteroidID[offsetData + a];
  float tmpX = x[asteroid] - x[i];
  float tmpY = y[asteroid] - y[i];
  float tmpZ = z[asteroid] - z[i];


  distance = tmpX * tmpX + tmpY * tmpY + tmpZ * tmpZ;
  if (distance == 0) { distance = 0.01f; }
  moduleF = 1/sqrt(distance);
  if (distance <= 100) { moduleF *= -1.0f; }

  dirX = tmpX * moduleF;
  dirY = tmpY * moduleF;
  dirZ = tmpZ * moduleF;

  force = kG * ((m[i] * m[asteroid]) / distance);


  fx[i] += force * dirX;
  fy[i] += force * dirY;
  fz[i] += force * dirZ;

 }
```

CUDA

```
__kernel void updateAsteroidForce(__global float* m,
__global float* fx, __global float* fy, __global float* fz,
__global double* x, __global double* y, __global double* z,
__global int* asteroidID, __global int* numberAsteroid) {


uint i = get_global_id(0);


double distance = x[i] * x[i] + y[i] * y[i] + z[i] * z[i];
float moduleF = sqrt(distance);

float dirX = x[i] / -moduleF;
float dirY = y[i] / -moduleF;
float dirZ = z[i] / -moduleF;

float force = kG * ((kMSun * m[i]) / distance);

fx[i] = force * dirX * 8500000;
fy[i] = force * dirY * 8500000;
fz[i] = force * dirZ * 8500000;
```

OPENCL

```
int n = numberAsteroid[i];
int offsetData = kMaxNumberAsteroidDev * i;
for (int a = 0; a < n; ++a) {

  int asteroid = asteroidID[offsetData + a];
  float tmpX = x[asteroid] - x[i];
  float tmpY = y[asteroid] - y[i];
  float tmpZ = z[asteroid] - z[i];


  distance = tmpX * tmpX + tmpY * tmpY + tmpZ * tmpZ;
  if (distance == 0) { distance = 0.01f; }
  moduleF = 1 / sqrt(distance);
  if (distance <= 100) { moduleF *= -1.0f; }

  dirX = tmpX * moduleF;
  dirY = tmpY * moduleF;
  dirZ = tmpZ * moduleF;

  force = kG * ((m[i] * m[asteroid]) / distance);


  fx[i] += force * dirX;
  fy[i] += force * dirY;
  fz[i] += force * dirZ;

}
```

OPENCL

2º Once we have the force, we have to calculate the acceleration using the following formula.
$a = F/m;$

```
__global__ void accelerationKernel(float* m, float *fx, float* fy, float* fz,
                                   float *ax, float* ay, float* az, const int size){
  int i = threadIdx.x;
  int j = blockIdx.x;
  i = i + j * blockDim.x;
  float mInv = 1 / m[i];
  if (size > i) {
    ax[i] = fx[i] *mInv;
    ay[i] = fy[i] *mInv;
    az[i] = fz[i] *mInv;
  }
```

CUDA

```
__kernel void updateAsteroidAcceleration(__global float* m,
  __global float* fx, __global float* fy, __global float* fz,
  __global float* ax, __global float* ay, __global float* az) {

  uint i = get_global_id(0);

  ax[i] = fx[i] / m[i];
  ay[i] = fy[i] / m[i];
  az[i] = fz[i] / m[i];


}
```

OPENCL

3º Next the velocity, using: $v = v0 + a * t$

```cuda
__global__ void velocityKernel(float* vx, float* vy, float* vz,
                               float* ax, float* ay, float* az,
                               float* t, const int size){
  int i = threadIdx.x;
  int j = blockIdx.x;
  i = i + j * blockDim.x;
  if (size > i) {
    float velX = vx[i] + ax[i] * t[0];
    float velY = vy[i] + ay[i] * t[0];
    float velZ = vz[i] + az[i] * t[0];

    vx[i] = velX;
    vy[i] = velY;
    vz[i] = velZ;
  }
}
```

CUDA

```opencl
__kernel void updateAsteroidVelocity(
  __global float* vx, __global float* vy, __global float* vz,
  __global float* ax, __global float* ay, __global float* az,
  __global float* t) {

  uint i = get_global_id(0);
  float time = t[0];

  float velX = vx[i] + ax[i] * time;
  float velY = vy[i] + ay[i] * time;
  float velZ = vz[i] + az[i] * time;

  vx[i] = velX;
  vy[i] = velY;
  vz[i] = velZ;


}
```

OPENCL

4º Finally, we calculate the next position, using: $p = p0 + v * t + a * t * t * 1/2$

```cuda
__global__ void positionKernel(double* x, double* y, double* z, float* vx,
                               float* vy, float* vz, float* ax, float* ay,
                               float* az, float* t, const int size){
  int i = threadIdx.x;
  int j = blockIdx.x;
  i = i + j * blockDim.x;
  if (size > i) {
    float time = t[0];

    float pX = x[i] + vx[i] * time + (ax[i] * time * time * 0.5f);
    float pY = y[i] + vy[i] * time + (ay[i] * time * time * 0.5f);
    float pZ = z[i] + vz[i] * time + (az[i] * time * time * 0.5f);

    x[i] = pX;
    y[i] = pY;
    z[i] = pZ;
  }
}
```

CUDA

```opencl
__kernel void updateAsteroidPosition(
  __global double* x, __global double* y, __global double* z,
  __global float* vx, __global float* vy, __global float* vz,
  __global float* ax, __global float* ay, __global float* az,
  __global float* t) {


  int i = get_global_id(0);
  float time = t[0];

  float pX = x[i] + vx[i] * time + (ax[i] * time * time * 0.5f);
  float pY = y[i] + vy[i] * time + (ay[i] * time * time * 0.5f);
  float pZ = z[i] + vz[i] * time + (az[i] * time * time * 0.5f);

  x[i] = pX;
  y[i] = pY;
  z[i] = pZ;

}
```

OPENCL

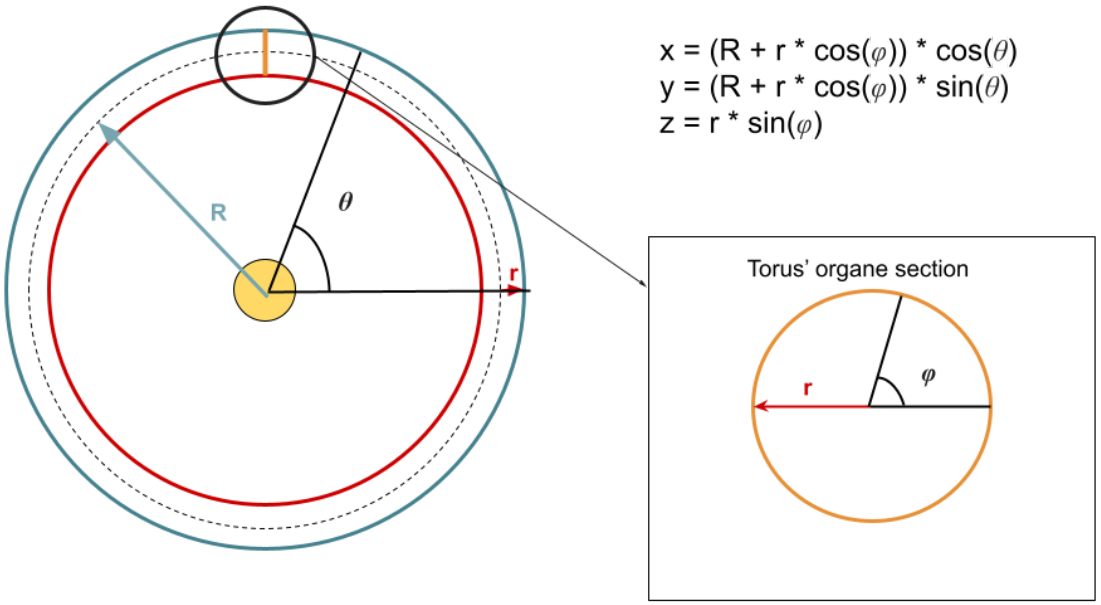
All of them have to be executed in sequence, as there is data dependence between them. All formulas are based on the Euler integration method.

Common implementation.

Initialise parameters.

The first we have to do is initialise all the positions, mass, radius, and velocity of each of them. We will do the initialization using std::thread to make it faster. We will use three different worker functions, the first one to initialise the position inside the torus, the second one to initialise the radius and the mass and the third one to generate the initial velocity. The first two tasks can be done at the same time as they do not have data dependencies, and when those two are finished, the third one can start.

For the position, we use the polar equations of the torus to get a position in the torus.



If we follow those equations, we will only obtain points on the surface of the torus, so to upgrade it, and obtain points in the torus (surface included), we have to make that $r \in (0, r]$. R is the distance of the centre of the tube of the torus to the sun. And $\varphi, \theta \in [0, 2\pi)$.

Initialization of position in code

```
float alfai = ((float)(rand() % 3600))/10.0f;
float betai = ((float)(rand() % 3600))/10.0f;
float alfa = (alfai * 3.141592f) / 180.0f;
float beta = (betai * 3.141592f) / 180.0f;
float radM = ((float)(rand() % 1000))*0.001f;
float cBeta = cosf(beta);
double innerRadi = kInnerTubeRadi * radM;
asteroidData.x[i] = (kDistanceCenterToCenterTorus + innerRadi * cBeta) * cosf(alfa);
asteroidData.y[i] = (kDistanceCenterToCenterTorus + innerRadi * cBeta) * sinf(alfa);
asteroidData.z[i] = kInnerTubeRadi * sin(beta);
```

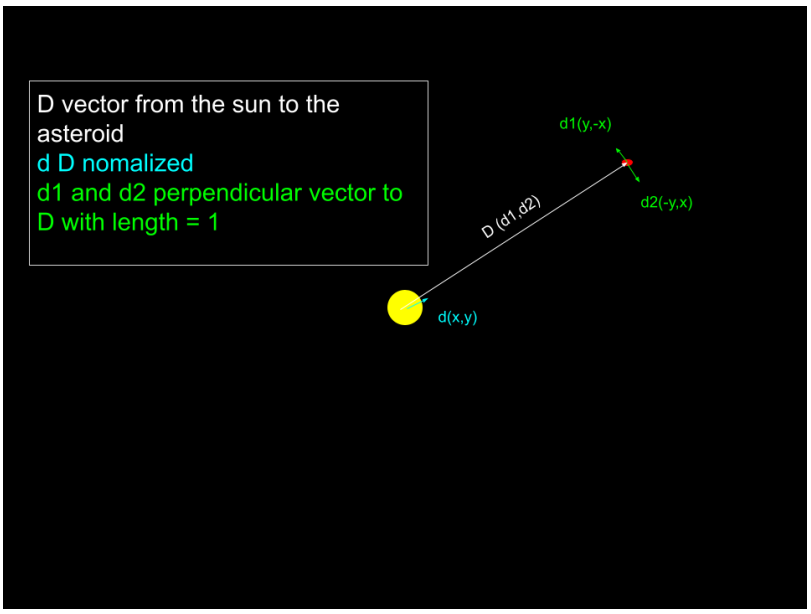
For the masses, first I decide the type of the asteroid and give a random radius according to the type and its density. With the radius, we calculate the volume of each one $V=r^3*\pi *4/3$. With the volume, we can also calculate the mass using ρ = m/V.

In the first image, we choose the type and second we set the radius and calculate its mass.

```
int type = rand() % 100;

int minR = 0;
int maxR = 0;
float density = 0.0f;
// C Type
if (type < kPercentageC) {
  minR = kMinRC;
  maxR = kMaxRC;
  density = kDensityC;
}
// S Type
else if (type > kPercentageC && type < kPercentageC + kPercentageS) {
  minR = kMinRS;
  maxR = kMaxRS;
  density = kDensityS;
}
// M Type
else {
  minR = kMinRM;
  maxR = kMaxRM;
  density = kDensityM;
}
```

```
  float r = float(rand() % (minR - maxR) + kMinRC);
  asteroidData.r[i] = r;
  asteroidData.m[i] = 4.0f / 3.0f * 3.141592f * r * r * r * density;
}
```

For the velocity, to calculate the modulus we use the following formula: $v = G * Msun/r$. And to calculate the direction of it we are going to use the perpendicular vector of the one that goes from the sun to the asteroid. As the initial velocity does no have *z* component, we can simplify the calculation of the perpendicular vector, we only have to change the component *x* for the *y* and *y* for the *x* and multiply by *-1* to one of them. The problem with that is that we can obtain two vectors, so if we multiply by *-1* the *x* component, and we do not obtain the desired one, we have to multiply the *y* component. We are going to multiply the initial velocity by 0.1 and adjust it for the simulation.



Code to calculate the velocity

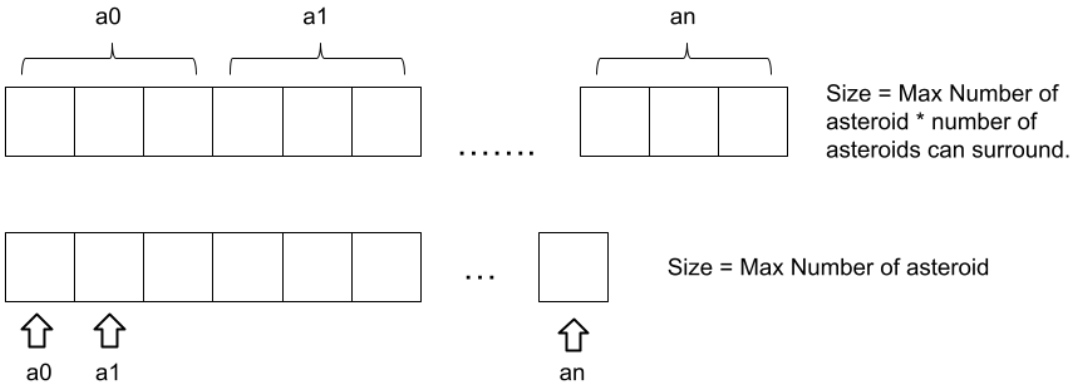
```
float dx = asteroidData.x[i];
float dy = asteroidData.y[i];

float distance = (dx * dx) + (dy * dy);
distance = sqrt(distance);

float cosAsteroid = dx / (distance);
float sinAsteroid = dy / (distance);

double mv = kG * kMSun / distance;
mv *= 0.1;

asteroidData.vx[i] =  mv* sinAsteroid;
asteroidData.vy[i] = -mv * cosAsteroid;
asteroidData.vz[i] = 0.0f;
```

QuadTree.

That is what we will use to have sorted all the asteroids and get in the faster way which asteroids are near a given one. That saves us to see through all the asteroids and compare their position to know if they are near or not. That is used when we calculate the force.

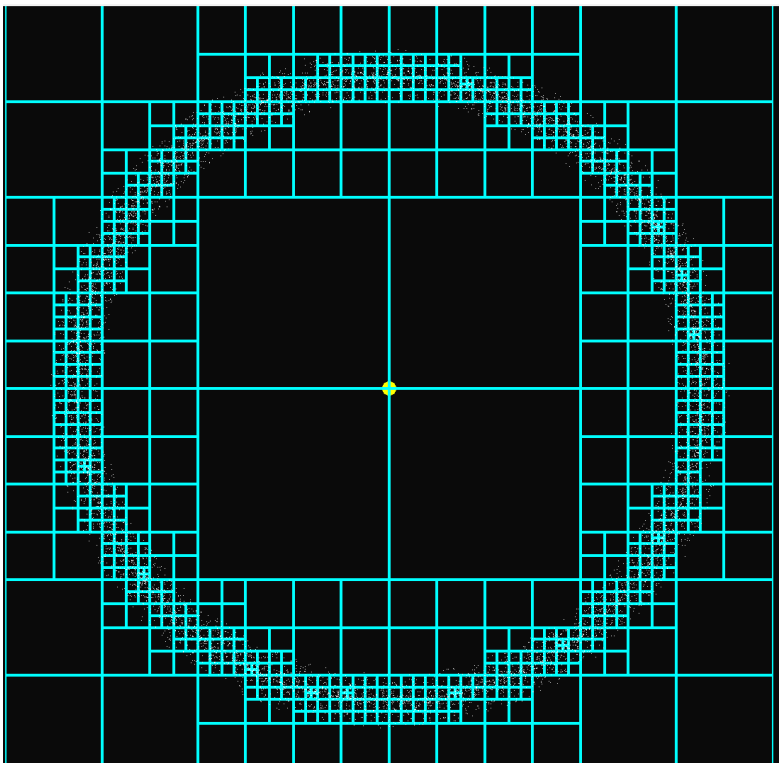
In the quadtree, we have two different functions. The first one is to update the quadtree with the updated positions each frame. And the second is the query, which will tell each asteroid which asteroids are near to it and how many.

However, we have to save that information so we will use two pointers: the first one will have the id of the asteroids that are near, and the second, is how many. So the allocation of the memory will be like the image below. The first one represents the asteroid that each asteroid is surrounded by and the second, how many.



The amount of near asteroids that one asteroid could have is limited in our case by 200 as the distance is really short and is very rare to have more than 200.

Representation of the quadtree

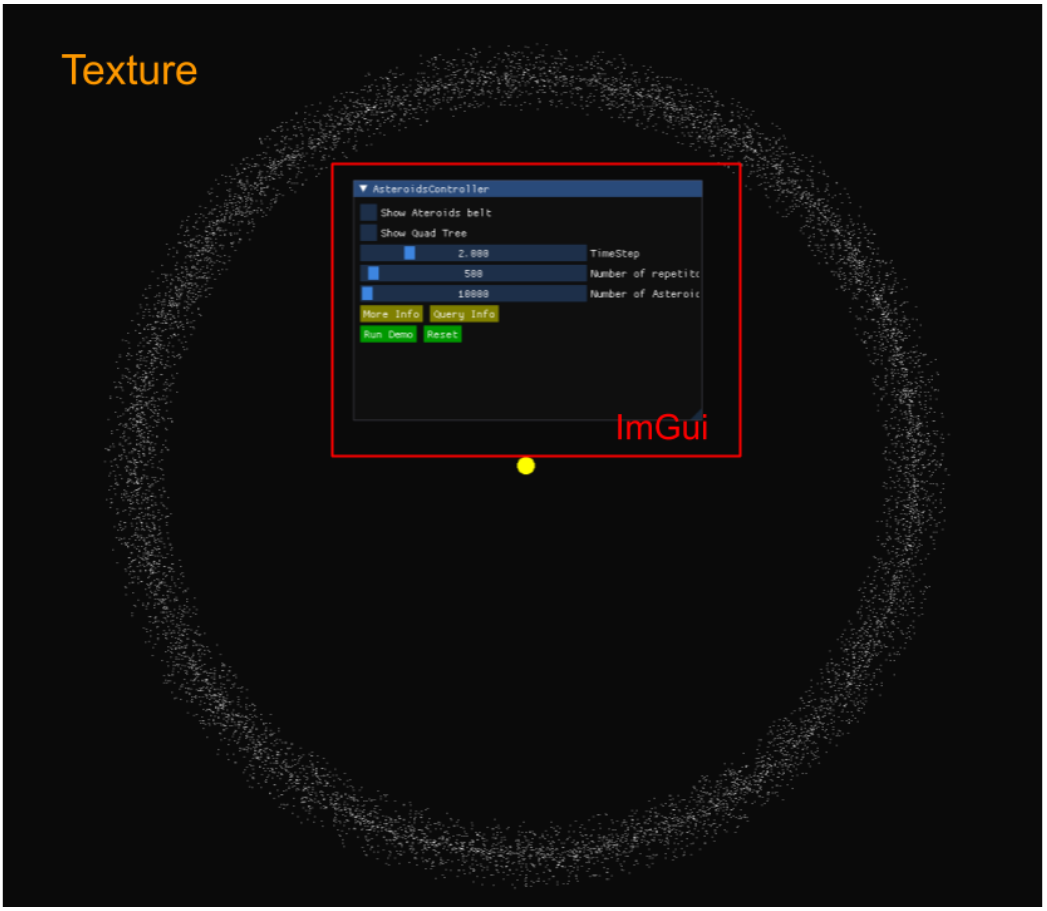


Render & texture painting.

To make all the render we are going to use a graphic library, SFML. We are going to paint each asteroid as a little 1x1 square. To go further we are going to do only one draw call to pain all asteroids. Instead, we are going to paint a texture in which each asteroid is going to be represented with a pixel painted on that texture. To make that we are going to create a class that inherits from both, *sf::Drawable* and *sf::Transformable*.

The good part is that you only need one draw call, but for each frame. But for each frame we will have to tell the new position of the asteroid in the texture, so to make it faster we will use threads. To make the window with the parameter of the simulation we use ImGui.

Simulation window



Time measurement.

To measure the times we will use *std::chrono.* First, we have to limit the number of frames/repetitions we want our simulation to have. That is because we are going to measure times in a certain way. While the execution of the simulation we will only get time, as you can see in the image below.

We do that to not waste time calculating the time each frame. For this reason, we are going to use a std::chrono array to store all the times, that is why we have to limit the number of frames for the simulation. When the simulation is finished, we are going to calculate all the times and show the results in a file in /log/log.txt

Measure CUDA kernels vs OpenCL

```
chrono->startF[id] = std::chrono::high_resolution_clock::now();
//Force Kernel
{
  // Launch a kernel on the GPU with one thread for each element.
  forceKernel << <numberBlocks, kNumberThreadPerBlock >> > (asteroidDataGPU->dev_m,
    asteroidDataGPU->dev_fx, asteroidDataGPU->dev_fy, asteroidDataGPU->dev_fz,
    asteroidDataGPU->dev_x, asteroidDataGPU->dev_y, asteroidDataGPU->dev_z, size,
    asteroidDataGPU->dev_outputData, asteroidDataGPU->dev_outputSize);


  // Check for any errors launching the kernel
  cudaStatus = cudaGetLastError();
  if (cudaSuccess != cudaStatus) {
    fprintf(stderr, "forceKernel launch failed: %s\n", cudaGetErrorString(cudaStatus));
    goto Error;
  }
  // cudaDeviceSynchronize waits for the kernel to finish, and returns
  // any errors encountered during the launch.
  cudaStatus = cudaDeviceSynchronize();
  if (cudaSuccess != cudaStatus) {
    fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launching forceKernel!\n", cudaStatus);
    goto Error;
  }

}
chrono->startA[id] = std::chrono::high_resolution_clock::now();
```

```
chronoSS.startF[id] = std::chrono::high_resolution_clock::now();
CLStatus = clEnqueueNDRangeKernel(queue, forceKernel, 1, NULL, &global_size,
  &local_size, 0, NULL, NULL);
if (CLStatus < 0) {
  perror("Couldn't enqueue the forceKernel");
  exit(1);
}

chronoSS.startA[id] = std::chrono::high_resolution_clock::now();
```

Analysis result.

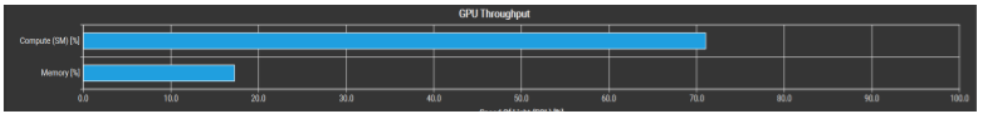
In our simulation, we will fix the time step to 2 and the number of repetitions to 100. The reason is to have coherence in the results and make all the simulations with the same conditions. To get information about our simulation we will use Nvidia Nsight.

In addition, we can also see the duration of our kernels which is as we expected and similar to the result we obtain in our time measurements. We can see that the slowest is the force kernel and the fastest is the acceleration one.

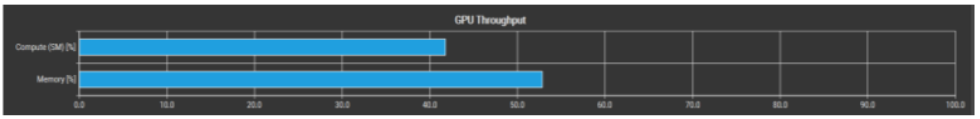
Function Nam	Demangled Name	Process	Device Name	Grid Size	Block Size	Cycles [cycle]	Duration [usecond]	Compute Throughput [%]	Memory Throughput [%]	#
forceKernel	**forceKernel(float *,...**	**[26488] Integrating...**	**NVIDIA GeForce RTX 30...**	**977, 1, 1**	**1024, 1, 1**	**1,102,976**	**795.71**	**71.04**	**17.25**	
acceleratio...	accelerationKernel(...	[26488] Integrating ...	NVIDIA GeForce RTX 306...	977, 1, 1	1024, 1, 1	243,107	176.48	41.79	52.85	
velocityKern...	velocityKernel(float...	[26488] Integrating ...	NVIDIA GeForce RTX 306...	977, 1, 1	1024, 1, 1	310,558	225.54	37.91	53.71	
positionKer...	positionKernel(dou...	[26488] Integrating ...	NVIDIA GeForce RTX 306...	977, 1, 1	1024, 1, 1	560,973	406.27	53.48	60.56	

Also, we can obtain more information about each kernel the occupancy and the GPU throughput. Force kernel has more than 70% on computing. For the acceleration kernel, we have a 40% and more than 50% on memory pretty similar to the velocity. And in position, we have more than 50% in computing and 60% in memory.

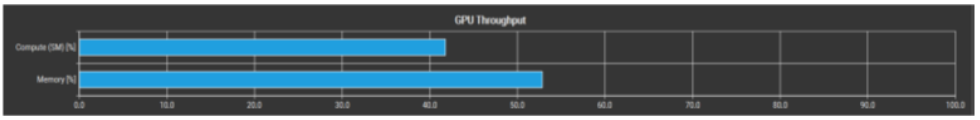
Force Kernel



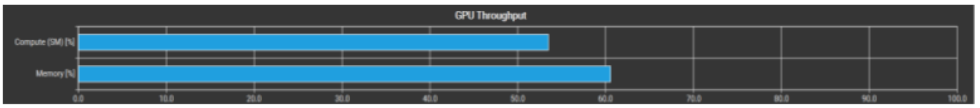
Acceleration Kernel



Velocity Kernel



Position Kernel



Talking about occupancy we have a 64% in most of them. In our case, we have four kernels that sometimes are just for one operation so that one can be joined into a bigger one, but always try to not make them so much more complex.

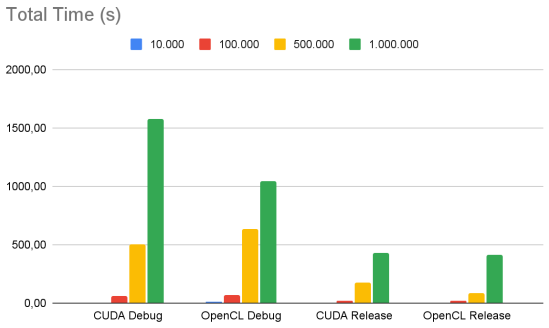
Force Kernel

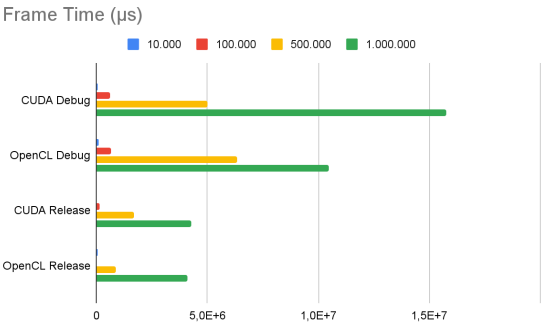
Acceleration Kernel

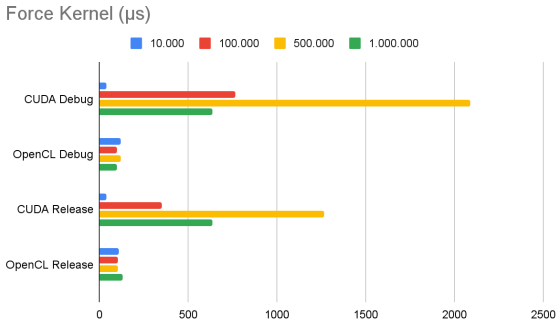
Velocity Kernel

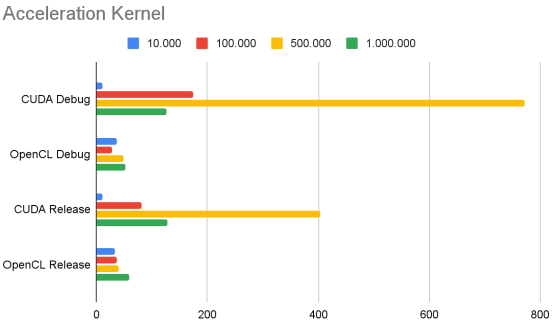
Position Kernel

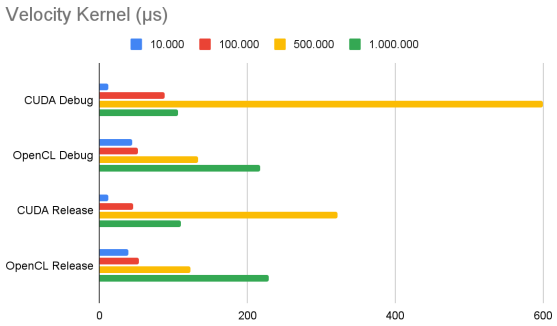
And also we are going to measure the time of each simulation. All of them are measured with the same conditions, only changing the number of asteroids. Time step = 2.0, number repetitions = 100, and x86.

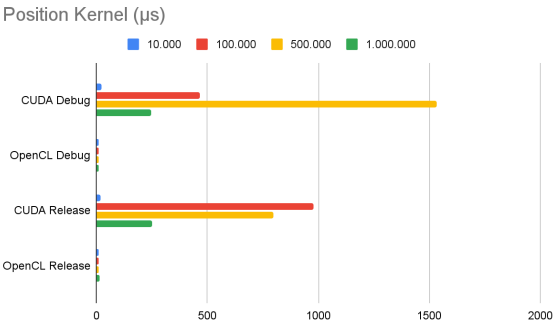












The graph shows results that are what we can expect. The total time of the simulation increases with the number of asteroids that there are. But with the kernels happens something very interesting. Talking about OpenCL, most of the kernels except velocity, stay more or less at the same time and velocity increases the time with the number of asteroids.

On the other hand, while using CUDA, as the amount of asteroids increases, it also increases the time that kernels take. Also, when we do the simulation with 500,000 asteroids, CUDA is terribly bad, duplicating the time that has 1,000,000. One reason could be that

In the end, the time is pretty much the same with a huge amount of asteroids. That is because the bottleneck of the simulation is in updating the quadTree and the texture, and the time kernels take is derisory tiny compared to that. As we can see in the 1,000,000 simulations as OpenCL is faster than CUDA but, both take a similar time.

Conclusion.

For beginners to GPU programming, CUDA is better as the initialization and finalization you have to do are pretty simple, only the allocation of the memory you will need and then free them, compared with the ones you have to do in OpenCL. If you are used to OpenGL, you might get used really fast to OpenCL as they are pretty similar.

Talking about the kernels both have kernels that are very similar. The main difference is what you call them. Regarding profiling, CUDA is much more accessible, than OpenCL.To debug CUDA kernels you only have to download the Nsight extension for Visual Studio or download Nsight software. In the last case, you will need to register as a developer for Nvidia to be able to download it. Once you used it, it will show you tons of different information about your kernels.

According to the result and the comparison above, I will choose CUDA for this project. Even if the result is worst than OpenCL, you can “easily” profile it and see how you can improve the performance, even in our case, we have to create a new project to be able to work with Nsight. In terms of performance, even OpenCL has a better one, with CUDA you can profile and see the possible improvements. Furthermore, if you are only interested in kernels performance, in that case, we can choose OpenCL, but if we are interested in simulation time, as we saw, the results were pretty much the same, so, in that case, we will choose CUDA.

References

- Code Train. (2018, March 26). *Coding Challenge #98.1: Quadtree - Part 1*. YouTube. Retrieved December 3, 2022, from https://www.youtube.com/watch?v=OJxEcs0w_kE
- Ocornut. (n.d.). *ocornut/imgui: Dear ImGui: Bloat-free Graphical User interface for C++ with minimal dependencies*. GitHub. Retrieved December 5, 2022, from https://github.com/ocornut/imgui
- SFML. (n.d.). *SFML*. SFML. Retrieved December 5, 2022, from https://www.sfml-dev.org/

Repo: https://github.coventry.ac.uk/beltranmur/AdvanceGPU.git