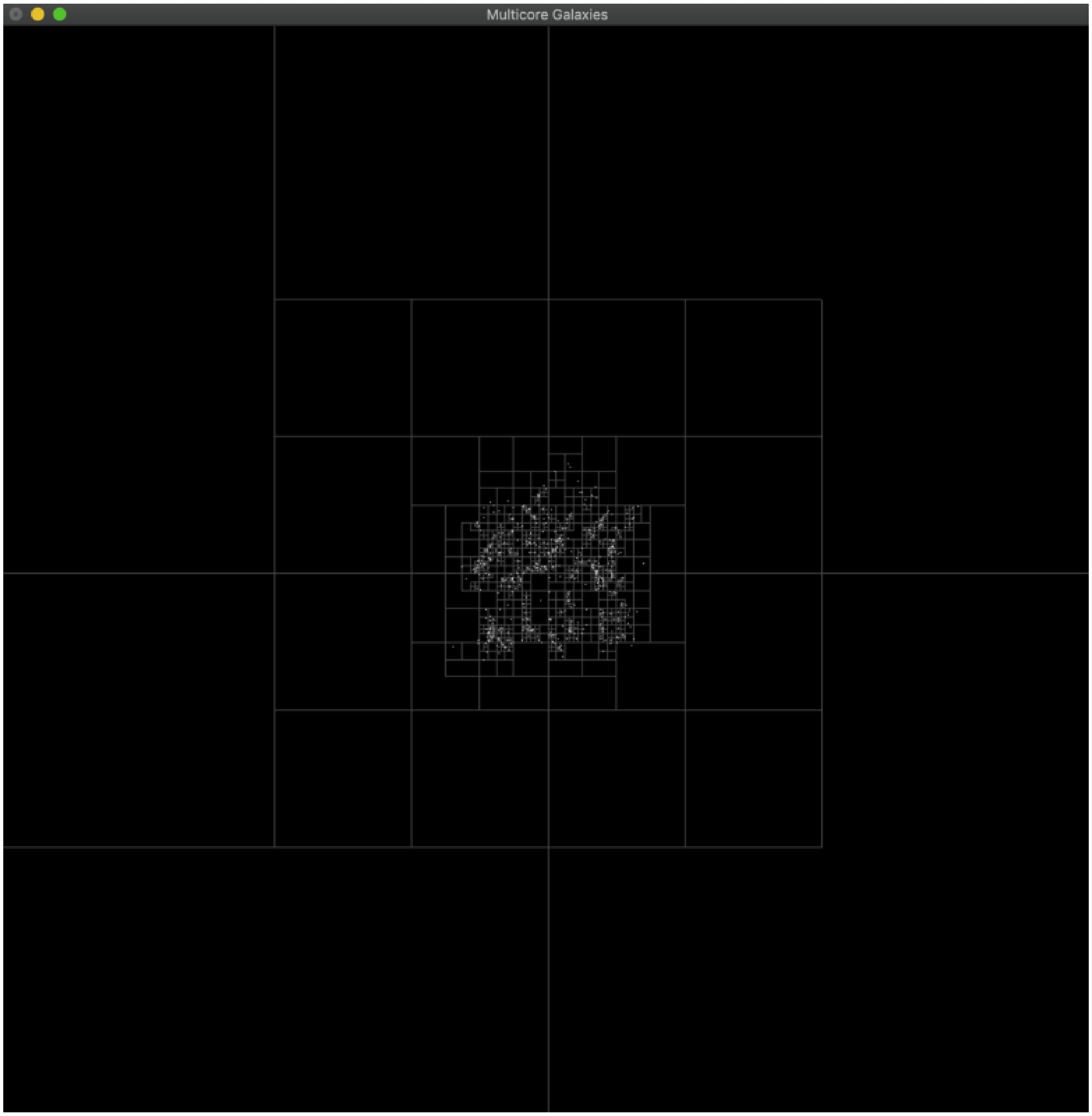
COMP426 MulticoreProgramming
Project 3 report



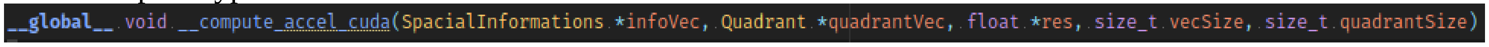
Clément Péau
40102553

CUDA

On this assignment we were tasked with using Nvidia’s CUDA to parallelise our computation.
On my development machine I am using a Gforce GTX 980. This card while old still pack a punch with its 2048 CUDA cores.

KERNEL

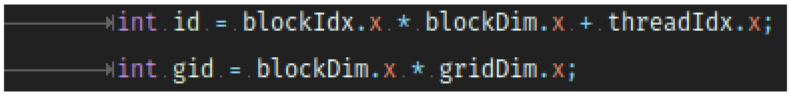
The prototype of the kernel is as follow



```
__global__ void __compute_accel_cuda(SpacialInformations *infoVec, Quadrant *quadrantVec, float *res, size_t vecSize, size_t quadrantSize)
```

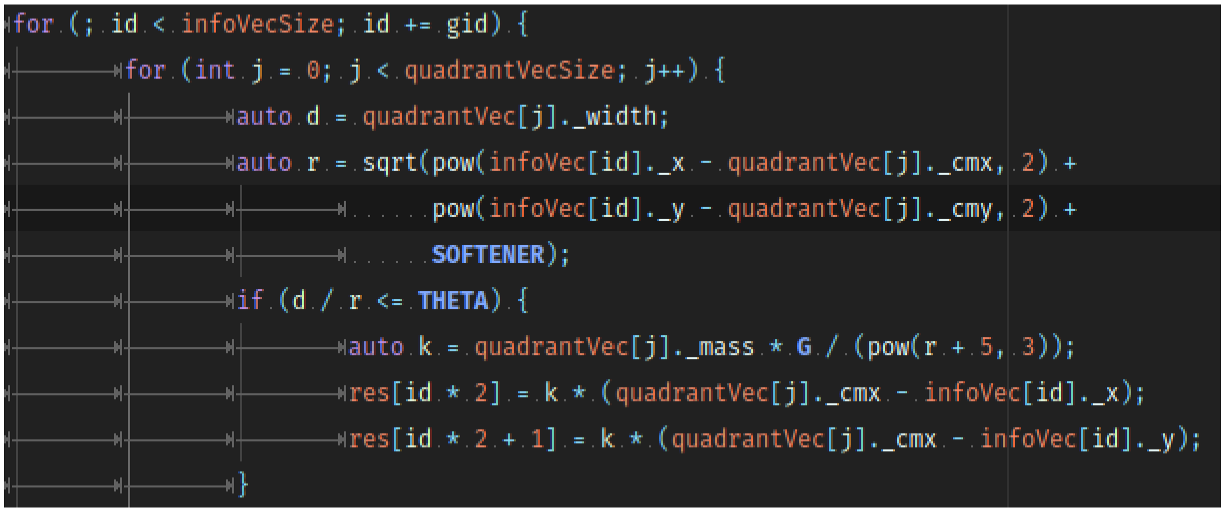
First parameter is a vector of spacialInformations that correspond to a Body. Each spacialInformations contains the coordinates, the weight, the size, and the mass. The second parameter is an array of Quadrant. Each quadrant each contains spacialInformation fields, this array correspond to every nodes at a given depth of the quad tree. The third parameter is an array of float of size 2 * NUMBER_OF_STARS, it is designed to store the x and y acceleration values.
The last two last parameters are the size of the two first arrays

First we calculate the thread ID and its grid ID using cuda’s global variables.



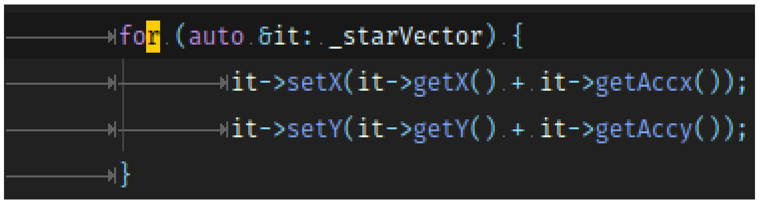
```
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    int gid = blockDim.x * gridDim.x;
```

This will be used to direct each thread onto its assigned workload.



```
for (; id < infoVecSize; id += gid) {
    for (int j = 0; j < quadrantVecSize; j++) {
        auto d = quadrantVec[j]._width;
        auto r = sqrt(pow(infoVec[id]._x - quadrantVec[j]._cmx, 2) +
                      pow(infoVec[id]._y - quadrantVec[j]._cmy, 2) +
                      SOFTENER);
        if (d / r <= THETA) {
            auto k = quadrantVec[j]._mass * G / (pow(r + 5, 3));
            res[id * 2] = k * (quadrantVec[j]._cmx - infoVec[id]._x);
            res[id * 2 + 1] = k * (quadrantVec[j]._cmx - infoVec[id]._y);
        }
```

We then make sure that we can’t have an id over our array size, and we increment the id by its gid. This way we make sure that we have the most granularity possible. Then we are going through each quadrant from this layer one by one. We then calculate our variables for the Barnes Hut Algorithm and store them inside our result array. This act as a cache. After the computation of every values. We loop through the Star array outside the kernel and apply them.



```
    for (auto &it: _starVector) {
        it->setX(it->getX() + it->getAccx());
        it->setY(it->getY() + it->getAccy());
    }
```