COMP426 Assignment 4

OpenCL Barnes-Hut Galaxy Simulator

First Name	Last Name	Student ID
Iulian	Rotaru	40102558

Introduction

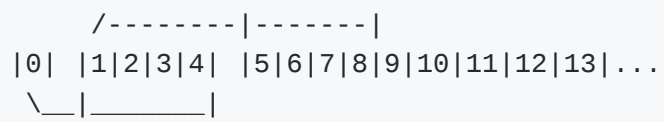
The goal of this last assignement was to use OpenCL in order to compute an n-body simulation. The particularity with OpenCL is that kernels can be written for CPU or GPU devices.
To complete this project, I restarted the simulation from scratch in C99, I'll explain why in the report.

How was divided the work between the CPU and GPU in order to minimize data movements and synchronization ?

So, the goal of the project was to make the less possible communication between CPU and GPU. In my approach, I saw way more potential in data parallelism with the GPU than in task parallelism with the CPU.
I completely reimplemented my code in C99 in order to have an algorithm where I could do data parallelism on EVERY step. This means that there is no logic running at all on the CPU.
The CPU code only contains the initial structure creation and kernel calls.
Also, all the data is stored only on the device memory, and recovered once for display purposes.

To achieve this, I had to:

- Switch from tree with pointers that are linking the nodes to a static sized array representation that could be allocated once



So in this array, cell 0's children are cells 1, 2, 3 & 4, cell 1's children are cells 5, 6, 7, & 8 etc ...

- Store bodies inside a static sized array. The array is sorted by cell index (index of the cell linked to the body)
- Bodies contains a link (index) to their cell
- Each cell contains an index to the beginning of its body chunk, and a body count
- Maximum depth is a key concept. A lot of computations are done layer by layer in the tree.

And now the algorithm is able to run in a data parallel fashion, all the informations needed are accessible in a direct and independant manner if computations are correctly dispatched.

How are kernel codes for the CPU and GPU performing the corresponding computations ?

I don't have any CPU kernel. All the kernels are GPU. I will describe every step and kernel, and assemble everything at the end.

init

Type	Work Size	Dimensions
Host Code	none	none

The initialisation occurs once.

- Allocate bodies, cells and galaxy_infos in heap
- Initialize values
- Create device buffers
- Copy bodies, cells and galaxy_infos in buffers
- Keep heap allocation from bodies and cells (to reuse for recovery)
- Initialize OpenCL context
- Load Kernels

- Compile Kernels

galaxy_contains_losts

Type	Work Size	Dimensions
GPU Kernel	Body Count	2 (one for each galaxy)

- Check if cell index of the body is 0
- If it is, it means that the body is "lost" (not assigned to any cell), set return value to true
- If it isn't, it means that the body is already assigned to a cell

galaxy_dispatch_losts

Type	Work Size	Dimensions
GPU Kernel	Body Count	2 (one for each galaxy)

- From (x,y) position, recover index of corresponding cell on the last layer of the tree (this can be done trivially without loops)
- Go up in tree (can be done without loops, by computing parent index from current cell index)
- When active cell found (cells have active flags), assign body to cell (set cell index on body)

galaxy_contains_sub_dispatchables

Type	Work Size	Dimensions
GPU Kernel	Cell Count	2 (one for each galaxy)

- Check on all cells at the same time if any contains more than one body and is not on the last layer

- Sets return value accordingly

galaxy_dispatch_sub_dispatchables

Type	Work Size	Dimensions
GPU Kernel	Cell Count on a layer	2 (one for each galaxy)

- Kernel called once per layer, starting from top layer, ignoring last layer
- Check if any cell contains more than one body
- If it does, correctly assign bodies to children cells (and activate them if needed)
- This will dispatch all the bodies as the bodies are "cascading" from higher levels to lower levels

galaxy_clear_inactivecells

Type	Work Size	Dimensions
GPU Kernel	Cell Count on a layer	2 (one for each galaxy)

- Kernel called once per layer, starting from last layer
- Check if cell has no children (or is on last layer), no bodies, and deactivate accordingly
- This will clear any superfluous cells, starting from the lower layers up to the top

galaxy_compute_com

Type	Work Size	Dimensions
GPU Kernel	Cell Count on a layer	2 (one for each galaxy)

- Kernel called once per layer, starting from last layer
- Computes center of mass by combining children centers of masses (unless the cells contains bodies)

- This will compute every cell's center of mass from lower layers up to the top

galaxy_compute_accelerations

Type	Work Size	Dimensions
GPU Kernel	Cell Count on a layer, Body Count	3 (one for each galaxy, one for cells on a layer, one for all bodies)

- Kernel called once per layer, starting from the top layer
- Computes at the same time all the bodies x all the cells of a layer
- Keeps track of computed cells for a specific body inside and compute history buffer
- Prevents useless computation by checking if parent cell has been computed (sets current as true if it is the case)
- Computes forces if cell contains bodies or if ratio tells us to consider a cell as a body and compute forces with its center of mass
- Sets current cell as computed in compute history

body_sort

Type	Work Size	Dimensions
GPU Kernel	Body Count	2 (one for each galaxy)

- For every body, compute amount of different bodies that have cell indexes lower than selected body
- Bodies with the same cell index are compared using their actual index in the body array
- Place into new position, inside a secondary body container initialized at the beginning of the process

- Host codes copies secondary buffer into primary

body_apply_accelerations

Type	Work Size	Dimensions
GPU Kernel	Body Count	2 (one for each galaxy)

- For every body, applies cached acceleration to speed, sets cache to 0, applies speed to position
- Check if body is getting out of cell. If it does, set cell index on the body to 0 (considered as lost)

cell_clear_idxs

Type	Work Size	Dimensions
GPU Kernel	Cell Count	2 (one for each galaxy)

- Sets body index to 0 (body index of 0 means there are no assigned bodies)
- Sets body count to 0

cell_set_idxs

Type	Work Size	Dimensions
GPU Kernel	Body Count - 1	2 (one for each galaxy)

- Every thread checks current and next body in the body array
- If next body has different index than current body, takes the linked cell of the next body, and set its start index

cell_set_amount

Type	Work Size	Dimensions
GPU Kernel	Body Count - 1	2 (one for each galaxy)

- Every thread checks current and next body in the body array
- If current body has different index than next body, takes the linked cell of the current body, and set its body count

```
init()

while (running)

    // While lost bodies are present, assign them to according c
    // Bodies are then resorted, and cells are relinked to bodie
    while (galaxy_contains_losts())
        galaxy_dispatch_losts()
        body_sort()
        cell_clear_idxs()
        cell_set_idxs()
        cell_set_amount()

    // While sub dispatchables cells are present, dispatch their
    while (galaxy_contains_sub_dispatchables())
        // Every time a layer is processed, bodies are resorted,
        // computation is correct
        for (layer_number in tree)
            galaxy_dispatch_sub_dispatchables(layer_number)
            body_sort()
            cell_clear_idxs()
            cell_set_idxs()
            cell_set_amount()

    // Check for cells to deactivate
    galaxy_clear_inactive_cells()

    // Compute centers of masses
    galaxy_compute_com()

    // Compute accelerations
    galaxy_compute_accelerations()

    // Apply computation
    body_apply_accelerations()

    // Recover bodies + cells, display
```

How the host program guarantees correctness and performance ?

The host program will take care of every kernel call, and will properly synchronise calls between each other. It will also take care of the display, and will limit computations per second depending on current FPS.

Conclusion

With Native Threads, I achieved smooth **700** body simulation.

With TBB, I achieved smooth **700** body simulation.

With CUDA, task parallelism and data parallelism, I achieved smooth **1.5k** simulation.

With OpenCL, this "new" algorithm and only data parallelism, I achieved smooth **10k** simulation on GTX 1060.