Parallel Performance Analysis of Brute Force N-Body Gravitational Application

Georgi Mirazchiyski

40208393

Concurrent and Parallel systems

SET10108

School of Computing

December 16, 2018

Abstract

The purpose of this report is to present and analyse multiple parallel implementations of a sequential application that computes a `brute force N-Body` algorithm. The final scene produced, simulates a dynamical system of particle-like box objects in 3 dimensions, simulating Newtonian gravity[1]. The methods for achieving parallelism, covered in the scope of this project are using `OpenMP`, for an initial parallel version of the application, and the recently released modern parallel programming model `SYCL` that adds OpenCL accelerator backend to the computation of the simulation.

1 Introduction and Background

The main idea of the project is to compare, analyse and discuss the parallelisation of the algorithm with every one of the mentioned methods and utilise both CPU and GPU devices.

The concept behind an N-Body system is a simulation of a dynamical system of bodies (or particles) that interact with one another under the influence of physical forces, such as gravity[ref:wikipedia]. The most wide-spread and intuitive way to perform this simulation is to apply the Newtonian gravitational force equation, which, in this system, was adopted in a way suiting for a reasonable performance within a real-time graphics application.

The formulation is as follows:

`Fnewton` = `Gconstant` * Mass * Mass / Distance * Distance.

Each body's acceleration is derived by calculating the forces applied to it. Therefore, allowing to compute the new velocity and position, using a time step constant. This final calculation is called integration. More specifically, the (semi-) euler variation was chosen for the implementation in code.

The algorithm implemented for this project is split in two parts; one for calculating the forces and one for computing the new positions of the bodies (integration). As the approach used if the brute force way of doing it, their time complexities are of O(n*n) and O(n), respectively, resulting in a total n*O(n*n).

The following `Listing 1.` shows exactly how the algorithm is designed in pseudo-code.

Listing 1: n-body application - pseudo code

```
1  // task 1: computes the gravity forces for each body
2  function computeForces(list<Body> bodies)
3      // gravity constant
4      real grav = 6.67408;
5      // apply gravitational force to each body
6      foreach (body1 in bodies)
7          foreach (body2 in bodies)
8              // calculate distance between each body
9              vec3 distance = body2.position - body1.position;
10             real length = distance.length;
11             // making sure we don't count length of 0
12             if (length > 1.0)
13                 // calculating gravity force's direction
14                 vec3 direction = distance.normalize;
15                 // applying the gravity calculation formula
16                 real num = grav * body1.mass * body2.mass;
17                 real denom = pow(length, 2) * direction;
18                 // update the final gravity force for the body
19                 body1.gravity += num / denom;
20             else  // do nothing
21
22 // task 2: inegrates the movement of each body
23 function integrateBodies(list<Body> bodies, real dt)
24     // apply semi-eueler integration
25     foreach (body in bodies)
26         body.acceleration = body.gravity_force;
27         body.velocity += body.acceleration * dt;
28         body.position += body.velocity * dt;
29
30 // runs the full simulation
31 function nbodySimulation(list<Body> bodies, real dt)
32     computeForces(bodies);
33     integrateBodies(bodies, dt);
```

2 Initial Analysis

2.1 Profiling

Once a working sequential version of the problem was implemented, it was followed by tuning and profiling of the code. I have used scoped timers to determine the baseline performance and analysed it via the use of the `Valgrind` profiler toolset that allowed for inspecting the `IR` and hot paths of the program. This helped to identify the main bottlenecks in the algorithm that are suitable for for optimization through parallelism.

`Figure 1` below shows output of the intensive bits of the application captured by the profiler.

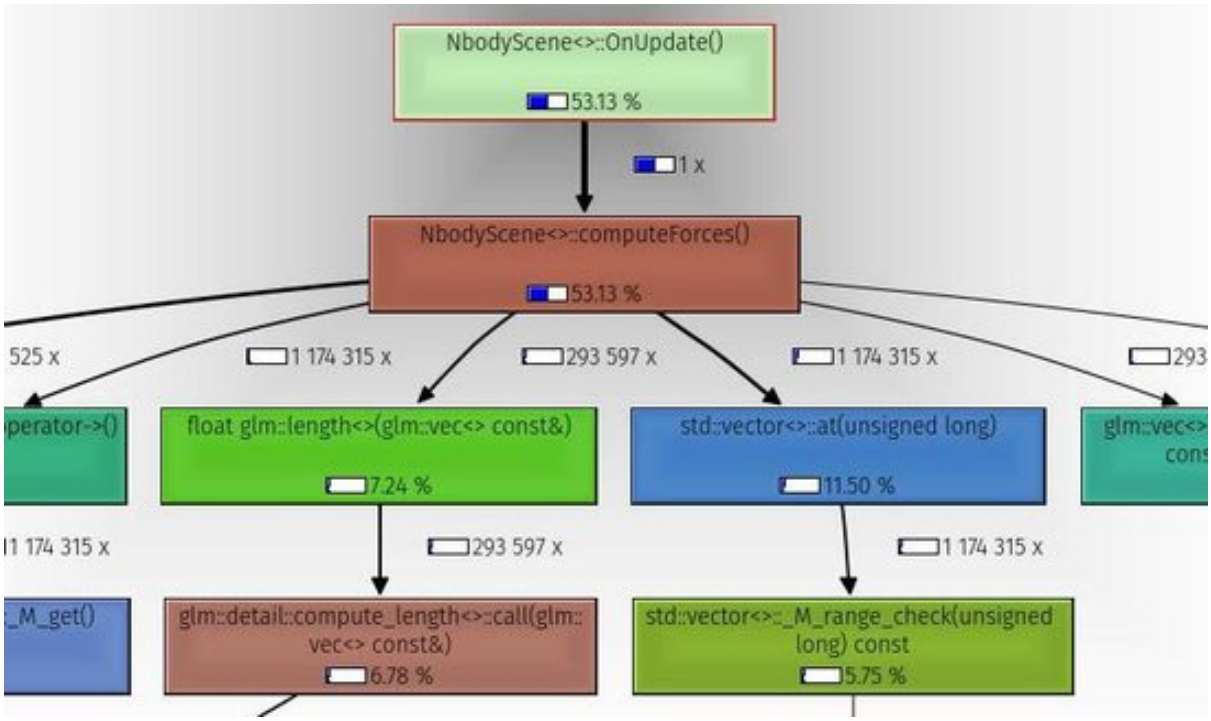

Figure 1: **Profiling tool.** Shot of the call-graph for the `OnUpdate()` function.

On the call-graph figure above can be seen that `53.13%` of the application's workload is in the OnUpdate function where the calculations of the simulation happen. The profiler has skipped function call for integration leaving the only the one for computing forces as the most performance impacting callee in the graph. To be skipped by the profiling tool, the integration function must not really impact much as it only reads and mutates values over 1-level for loop. The `computeForces()` function involves the heaviest mathematical calculations of the simulation, computed in almost perfectly nested for loops (2 levels). Although, optimising can be achieved by just adding parallelism to this function, I will consider the integration as well to try and squeeze every bit of possible performance gain.

2.2 Sequential benchmark

The baseline performance testing consists of simulating 4096 bodies with gravitational force applied to each one in a 200x100x50 axis-bounded field. As this is a real-time application, explicit instructions for termination were added on reaching 600 frames of execution, which were more than enough to find out the average time taken to compute a single frame of simulation. The time is measured in milli and micro seconds to allow for more precise comparison with the parallel solutions discussed in the next chapters.

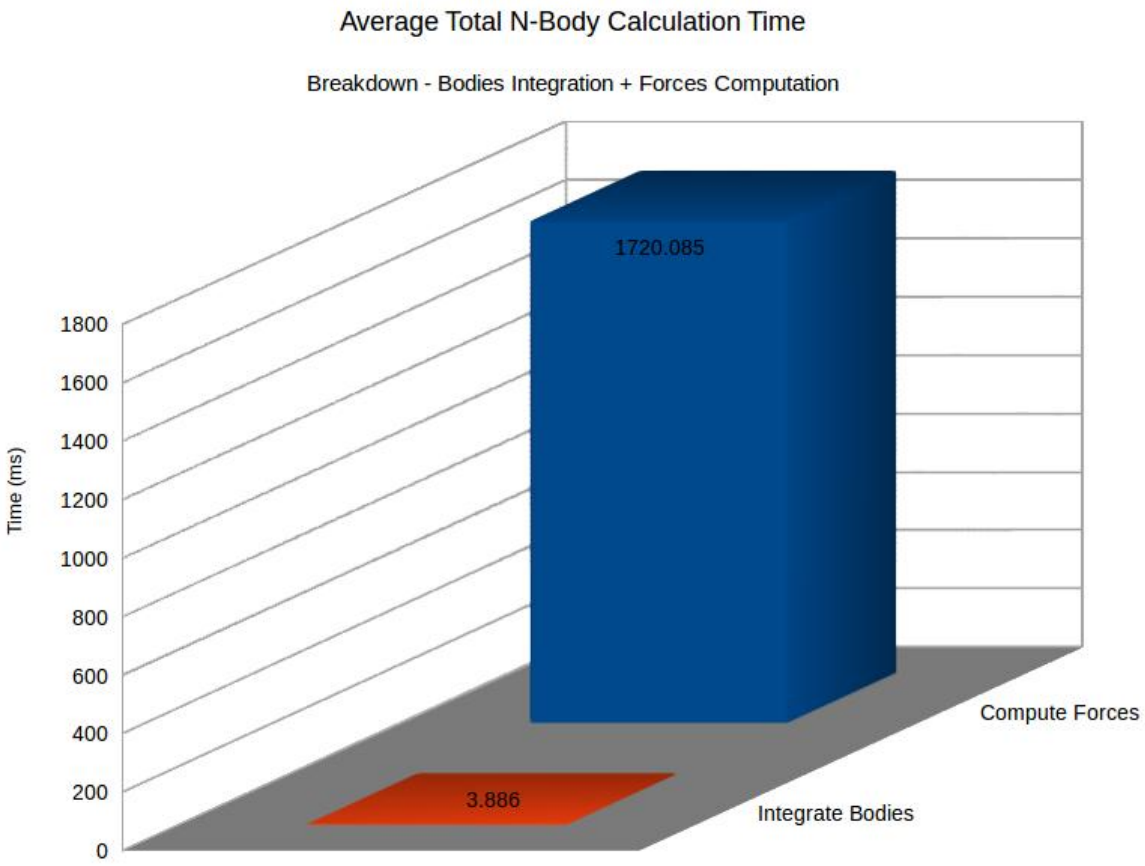

Figure 2: **Sequential benchmark - bar chart.** Average Time Taken to Integrate Bodies and Compute Forces per one time step of the simulation.

By analysing the chart in `Figure 2`, we can see that after splitting the main problem into two smaller tasks - computation of gravity forces and integration of the bodies, that together achieve the final simulation, it is now clear that the most critical bottleneck comes from calculating the forces that impact the system. In fact, it is approximately a 500 times slower operation to execute than the integration.

Operation (Time unit)	**Mean**	**Standard Deviation**	**Standard Error**
`Compute Forces (ms)`	`1720.085 ms`	`180.029`	`60.009`
`Compute Forces (µs)`	`1720085 µs`	`180029.658`	`60009.886`
`Integrate Bodies (ms)`	`3.886 ms`	`0.158`	`0.052`
`Integrate Bodies (µs)`	`158.869 µs`	`52.956`	`60009.886`

Table 1: Sequential Benchmark - `statistical summary of the measured time.`

3 Methodology

Hardware configuration	*General compute devices specs*
CPU Device	Intel(R) Core i7-8550U @ 1.80GHz
Max compute units	8 with Intel Hyper Threading
GPU Device	Intel(R) Gen9 HD Graphics
Max compute units	24
Software configuration	*General compute devices specs*
OS	Linux Ubuntu-x64
Compiler	gcc 7.4 with support for OpenMP 4.5
OpenCL/SYCL configuration	*General compute devices specs*
Device Version	**CPU**: OpenCL 1.2 / **GPU**: OpenCL 2.1
Max work item dimensions	**CPU**: 3 / **GPU**: 3
Max work item sizes	**CPU**: 8192x8192x8192 / **GPU**: 256x256x256
Max work group size	**CPU**: 8192 / **GPU**: 256
Global memory cache size (64 bytes)	**CPU**: 262144 / **GPU**: 524288
Local memory size (64 KiB)	**CPU**: 32768 / **GPU**: 65536
Benchmark configuration	*General compute devices specs*
Number of bodies	4096
Number of time steps	600

Table 2: Hardware and Software configuration specifications.

3.1 Method 1: OpenMP

OpenMP is a multi-platform multi-processing programming model with a simple and flexible API for developing parallel applications using high-level compiler directives (`pragmas`).

To begin with the adding of multithreading acceleration support to the simulation, I have used OpenMP as it is the easiest way of achieving this without having to do any initial setup. Both parts of the n-body algorithm remained unchanged. The gravity force calculation executes separately from the body integration, and does not require a synchronised access to shared memory resources making the parallelisation with the framework-provided functionality even

simpler to achieve.

The only addition is the `pragma directives` before the outer `for loop` of `computeForces()` and `integrateBodies();`

```
#pragma omp parallel for num_threads(max_hardware_threads) schedule(static)
```

Based on the hardware specifications of the machine used to execute the code, the number of threads is equal to 8. After experimenting with the options for loop scheduling, `static` seemed to gain the most performance. The correctness of the simulation is `1:1` with the sequential algorithm, only faster, resulting in a successful data-parallelism with this approach.

3.2 Method 2: SYCL

SYCL is a cross-platform abstraction layer that builds on the OpenCL concepts for heterogenous systems programming (SYstemCL). It enables single-source modern C++ style of coding for both its host and kernel language.

Logically, the next step was to go on a lower hardware level and try to fully utilise the capabilities of the processing devices. I have chosen to implement OpenCL support to the computations via the SYCL parallel programming model because both my CPU and GPU are OpenCL-enabled.

In the n-body algorithm, it is outlined that the forces calculation is a more compute-bound task, while the integration was more memory-bound, therefore I have implemented several variations of the kenrels involving different techniques for memory accessing and transfering, as follows:

- **global** - utilised global memory access with a 1-D work-item index space of 4096 elements with no work-group partitioning defined for both CPU and GPU versions of the kernels implementations. No explicitly specified barier synchronisation was required within the work-group. After the computation is completed the result is written back to the host buffer.
- **local** - utilised local memory access with a work-item 1-D index space of 4096 elements that are split into 32 work-groups of 128 work-items each for the CPU implementation. On the GPU, the elements were partitioned into 16 work-groups of 256-work items. A synchronisation between the threads in a work-group is applied after reading the host

data to the local memory of the device and after the computation is completed, before writing back to the host buffer.

- **coalesced** - utilised coalesced memory access that allows for memory to be accessed by a warp (consecutive threads) in a single transaction. On the CPU, 32 threads per warp were accessed using the global bandwidth, while on the GPU only 16 but for twice the amount of work-groups. Unlike the above, this technique was implemented only for the integration of the bodies and not for the forces computation because the main issue with the integration as mentioned is memory transfer and access.

4 Results and Discussion

4.1 (Log2) Time comparison between approaches

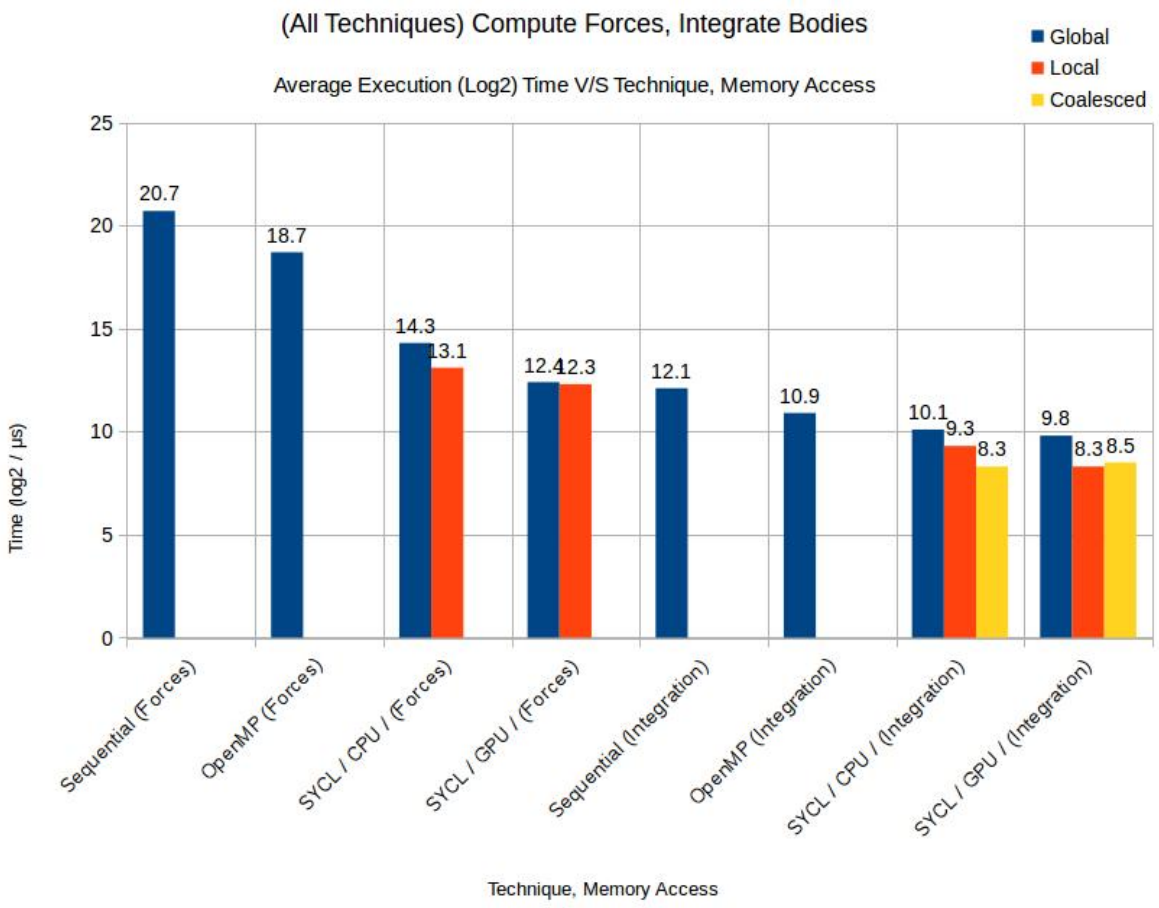

Figure 3: **Techniques comparison chart.** (Log2) Time vs Technique, Memory Access

The above Figure 3 shows how each the functions sequential performance compare to both OpenMP and SYCL approaches. In addition to that, the different techniques used in SYCL are included with different colors.

It can be observed that SYCL in general is the fastest approach without a doubt. The trend here is that when running the computations on the GPU they perform faster than on the CPU.

There are two trends considering the CPU: the utilisation of local memory location access is the most optimal technique for computing forces, while the coalesced access using the global memory bandwidth of the device seems more performant for the integration calculation.

Although, I was hoping that it will be similar for the GPU, it seems that the use of local memory in that case outperforms the coalesced access version for bodies integration.

4.2 Host/Device execution breakdown for SYCL

The Execution time is broken down into **kernel submission**, **kernel execution** and **overhead**, each of which is represented as percentage of the total time.

The following figures, Figure 4. and Figure 5., display a chart that represents the breakdown of the execution of the compute forces and integrate bodies operations on the CPU, respectively.

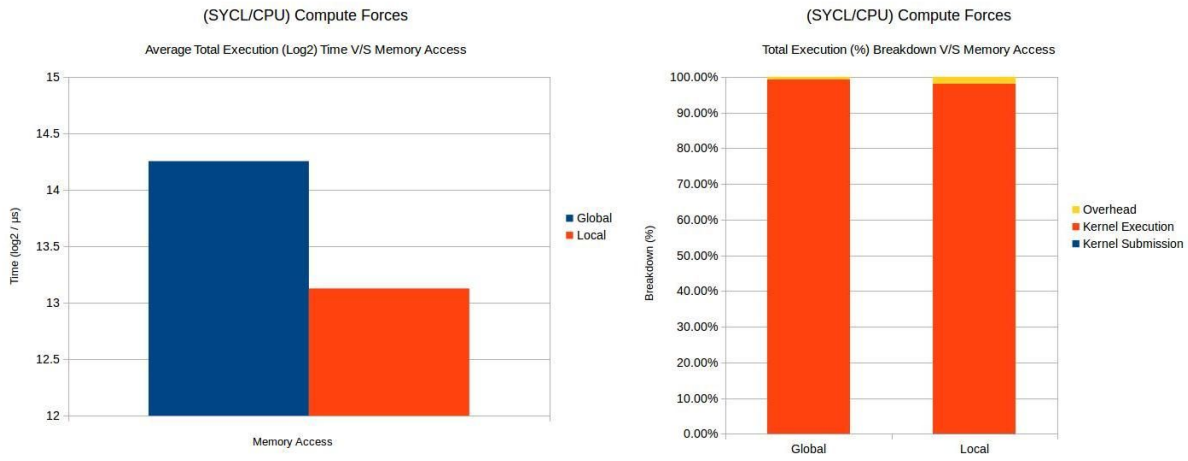

Figure 4: **SYCL / CPU - Compute Forces chart.** Total Execution vs Memory Access.

The total execution breakdown chart above shows the impact of each stage of the total

execution and memory movement from host to device and back to host for the Compute Forces operation on SYCL-CPU device.

It seems that kernel submission here has a very minimal to none impact on the whole execution for both global and local memory access versions. The kernel execution on the device-side that utilises local memory space is slightly bit more optimal than the global. Whereas the opposite is the case for the host overhead of memory allocation and transfers.

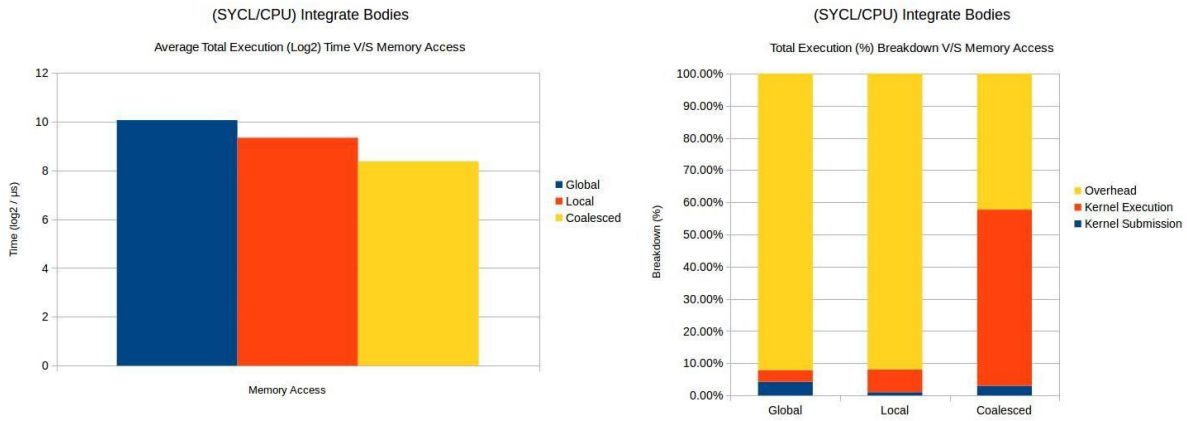

Figure 5: **SYCL / CPU - Integrate Bodies chart.** Total Execution vs Memory Access.

The total execution breakdown chart above shows the impact of each stage of the total execution and memory movement from host to device and back to host for the Integrate Bodies operation on SYCL-CPU device.

The kernel submission here has a little impact minimal on the whole execution for both local and coalesced memory access versions and slightly higher for global. The kernel execution on the device-side that utilises local memory space seems to have higher impact than the global one, whereas in the coalesced version it takes more than 50% of the whole execution. Hoewever, the coalesced one benefits from very little overhead due to the consecutive memory space achieved with this technique which is the main reason why this version outperforms the former two because of the nature of the problem here being memory-bound. Thus a more efficient way for threads to access memory locations is the most optimal solution to this computation.

The following figures, `Figure 6.` and `Figure 7.`, display a chart that represents the breakdown of the execution of the compute forces and integrate bodies operations on the GPU, respectively.

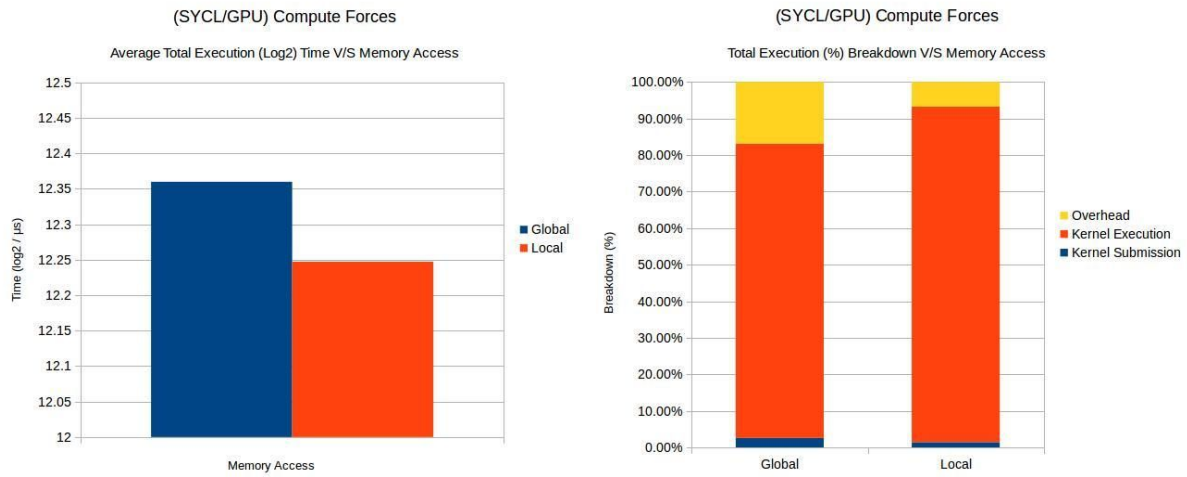

Figure 6: **SYCL / GPU - Compute Forces chart.** Total Execution vs Memory Access.

The total execution breakdown chart above shows the impact of each stage of the total execution and memory movement from host to device and back to host for the Compute Forces operation on SYCL-GPU device.

The kernel submission in the global version here is more impacting on performance than in the local one. However, it seems that the kernel execution seems to take off some of the heat in former version in comparsion to the latter one. But the end result is a faster local version due to the less overhead from memory transfers as it is a quite significant percentage in the global version of the kernel.

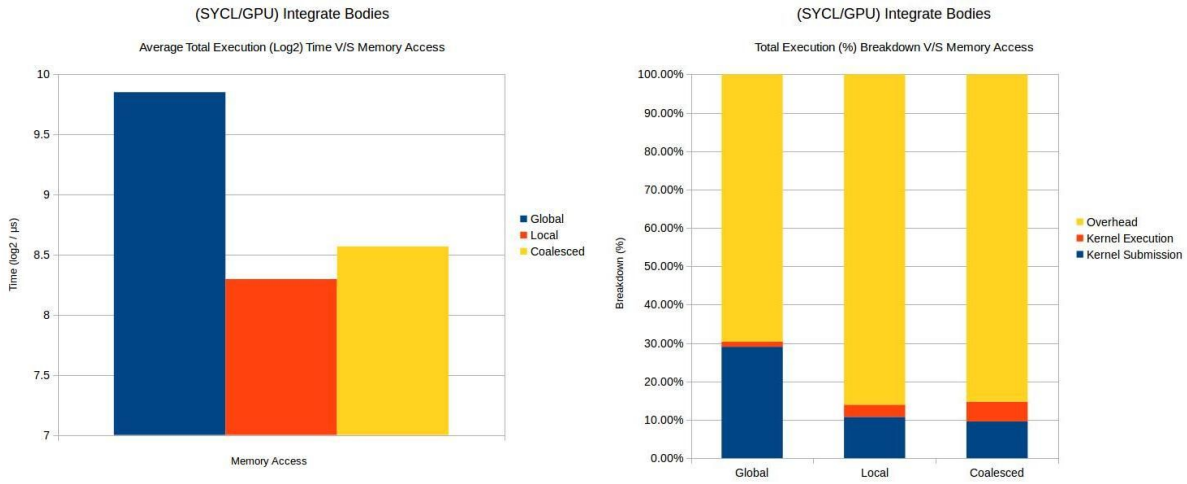

Figure 7: **SYCL / GPU - Integrate Bodies chart.** Total Execution vs Memory Access.

The total execution breakdown chart above shows the impact of each stage of the total

execution and memory movement from host to device and back to host for the Integrate Bodies operation on SYCL-GPU device.

The kernel submission is quite impacting on the performance in the global memory access version, while in the local and coalesced ones it is a reasonably small portion. The kernel execution is faster when utilising the local memory space rather than the global space bandwitdh with the coalesced version. The overhead is an equal percentage between local and coalesced versions and a good bit lower in the global version. The result is that the local and coalesced perform nearly equally fast. However, it seems that the GPU has managed to optimise local memory access a bit better, leading to a slightly faster overall execution.

4.3 Speedup and Efficiency

Next in line are speedup and efficiency comparison between all techniques, represented by the following figures, `Figure 8.` and `Figure 9.`.

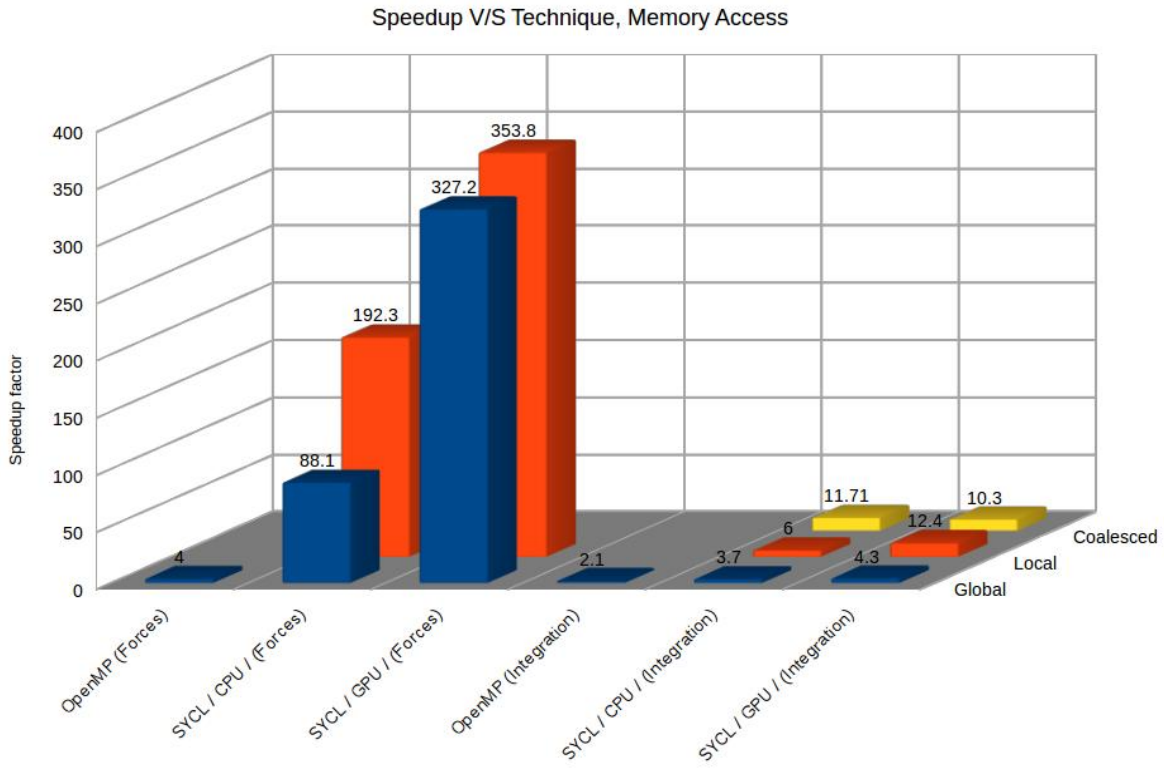

Figure 8: **Speedup chart.**

In the bar chart above we can see that SYCL on GPU with the utilisation of local memory access outperforms any other approach for the forces computation, the reason of which has

been discussed above in the **Breakdown** section. However, integration runs faster on the CPU, because of a better utilisation of the global memory coalescing implemented in the coalesced version of the SYCL kernel. Based on these findings, it can be recommended that the simulation is most optimal on the machine's configuration when run on the GPU with si, despite the little difference in the integration between the CPU and the GPU results with the thread coalescing implementation.

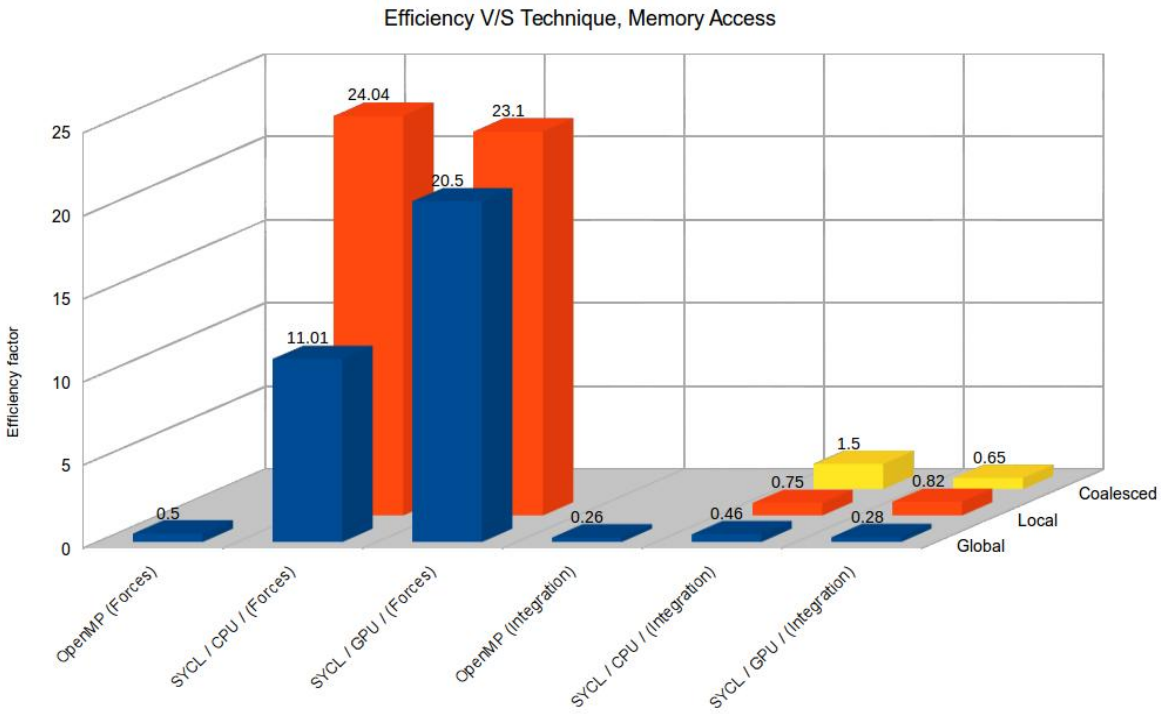

Figure 9: **Efficiency chart.**

It seems that the most efficient techhique to use for computing forces will be SYCL. The compuation is a lot faster on the GPU with local memory, however it runs on a lot more cores than the CPU and depending on how much other unrelated work is thrown at the GPU, it may reasonable to switch to computing it on the CPU. Best would to combine both devices and find the golden mean. As for the bodies integration, the SYCL coalesced version running on the CPU is both fastest and most efficient.

5 Conclusion

The investigation into the algorithm shows that it is highly paralellisable reagarding the computation of forces (gravity force) and scales really well with using GPGPU techniques. Speedup has been acheived in all cases with all of the computations involved into the n-body simulation. The variery of conducted experiments with SYCL show that there is no one generalised best way when it comes heterogenous systems that run on multiple devices, especially when we are talking graphics accelerators.

A potential improvement in this implementation can be to integrate the bodies within the force computation kernel because the velocities and positions only depend on the acceleration that is derived from the sum of all forces, thus it is a more general operation. This will save some overhead from the multiple transfers of position data from host to device and backwards, and only one command group per time step will be submitted by the SYCL queue instead of two.

Additionally, it would be really interesting to experiment with adding parallelism to the graphics part of the application with all of the used techniques. For the GPGPU approach, it would be sensible to continue with SYCL, only if the drivers support OpenCL/OpenGL buffers sharing. Improvements could possibly be really high if well implemented because the OpenGL vertex data of each body is connected its translation and orientation on the screen.

Overall, the parallelisation of the application was quite successful considering that the n-body calculation is more than 50% of the whole performance impact breakdown, the rest being the small graphics frameworks behind the visualisation of the simulation.

References

[1] Julia Layton. *Article on how does gravity work.* url: https://science.howstuffworks.com/environmental/earth/geophysics/question2321.htm