Comparison of Parallel Performance using Mojo, SYCL, and OpenCL

Mehmed Mulalić
University of Vienna

Supervised by:
ao. Univ.-Prof. Dipl.-Ing. Dr. Eduard Mehofer

July, 2025

1 Introduction

Sequential applications are executed using a single processing unit, and to increase performance, there are two methods: increasing optimization and/or utilizing multiple processing units, the latter of which is the focus of this report. The use of multiple processing units – parallel processing – enables applications to execute tasks over multiple cores or threads, significantly improving performance. For computationally intensive problems, such as molecular simulations or physics-based modeling, parallelism becomes essential. This study focuses on comparing the parallel performance of three heterogeneous parallel frameworks – Modular's Mojo, SYCL, and OpenCL – by implementing the Direct Coulomb Summation (DCS) computational method to calculate the electrostatic potential of charged atoms inside a three-dimensional mesh grid. To evaluate the performance of each framework, implementations were executed on a high-performance computing (HPC) cluster consisting of 6 nodes, each comprised of an AMD EPYC 7543 32-core CPU, as well as an NVIDIA Tesla A100 GPU.

2 Environment Setup

All experiments were tested using locally installed frameworks. Following the dedicated website of the Mojo framework to install it, CURL was used with the provided website and later exported to PATH with a source script. CUDA (Compute Unified Device Architecture) is a parallel programming platform framework and an API designed specifically for NVIDIA GPUs which allows software to run GPU-accelerated processing by utilizing NVIDIA GPUs. SYCL and OpenCL, in contrast, are open-source frameworks which also allow the use of GPU-accelerated processing but are general-purpose, allowing developers to use one framework while targeting multiple hardware, including FPGAs, CPUs, GPUs, etc. For the mentioned two frameworks to target NVIDIA GPUs, they are required to be compiled with linked CUDA drivers, meaning that CUDA must first be installed beforehand. CUDA drivers version 12.8.1_570.124.06 were installed using wget to download the extraction bash script from NVIDIA vendors, running the script and extracting to the local user directory, and finally running the installation script with the local user installation path. After installing CUDA drivers, packages gcc, CMake, clang, libstdcxx, and LLVM were installed using Conda and then added to PATH in bashrc. There are two main methods to using the SYCL framework: either Intel's oneAPI, which focuses on Intel drivers, or with AdaptiveCpp, which is a general-purpose SYCL framework. The latter method was used since the target device is an NVIDIA GPU. After installing previous packages, the SYCL framework was compiled using the AdaptiveCpp repository with the following parameters: local installation, Conda CXX compiler, CUDA backend enabled, toolkit root defined to local CUDA installation, and with CXX standard 17. Finally, to enable the use of OpenCL, the OpenCL ICD Loader and the OpenCL headers were installed, vendors were set, and the environment path was defined.

3 Methodology

After finishing the environmental setup, the sequential variation was completed beforehand using C++. The DCS method calculates the electrostatic potential value of each point in the grid as the sum of contributions from all atoms in the system [1]; this can be demonstrated with the following formula:

$$V = k_e \sum_{i=1}^{n} \frac{q_i}{r_i}; k_e = \frac{1}{4\pi\epsilon_0} = 8.987551786214 \times 10^9 Nm^2C^{-2}$$

Since this simulation is tested in a 3D space, the distance r is the Euclidean distance between the atom and the iteration's grid point. Two variables must be declared for the simulation: the atom count and the size of the grid mesh, defined as a 3D box of sizes $N \times N \times N$. The sequential variation consists of four loops, three to iterate over the x, y, z coordinates of the grid mesh, and one loop to iterate over the atoms and summarize the energy $\frac{q_i}{r_i}$, after which the outer loop will multiply the sum with the constant k_e and save the output for that specific point on the grid. Before calling this method, the sequential variant includes argument parsing to define the atom count and the grid mesh size. In contrast, heterogeneous parallel frameworks must also manage communication between two devices – the host (CPU) and the device (GPU). To enable GPU-accelerated computing, explicit buffer management is required to enable host-device communication. For the parallel variant, the DCS computational method is parallelized over the grid mesh by creating a 2D ND-range, creating a 2D slice of the 3D grid, where each thread is assigned to one grid point of a fixed z-axis. Five values are passed to the kernel: the output energy, atom data defined as a one-dimensional vector, the z index, and the number of atoms. Inside the kernel, the thread's global id is defined, one loop iterates through the atom data, energy is calculated, and the energy is saved to the output vector. This kernel is enqueued and read N times by the host.

Both the sequential and parallel versions have argument parsing to define the atom count and the grid size, as well as random atom data generation, which defines a pointer to contain three randomly distributed integer values to represent the atom's 3D position, followed by a randomly distributed floating-point number to represent the atom charge. In addition to the previous components, the parallel variant includes additional steps unique to each framework. After initializing OpenCL platforms and devices, the source and program are built, followed by the kernel which is read from an external file. Afterwards, two buffers are instantiated: a write-only buffer to represent the kernel output and a read-only input buffer representing atom data. Once the kernel arguments are defined, the computational loop begins by iterating over the z-axis. On each iteration, a new z value is set, after which an ND-range is enqueued, and the output buffer is read. The SYCL framework contains slight differences compared to OpenCL, namely that the only initialization needed is the queue and the device. When discussing SYCL computational models, there are two approaches: Unified Shared Memory (USM) and buffers with memory accessors. USM is AdaptiveCpp's recommended method and it offers more control over data manipulation and requires manual synchronization, while buffers offer a lax alternative represented by objects, including automatic memory management and synchronization. Compared to the previous frameworks, the Mojo framework skips a large part of initialization, as it requires only a device context and buffer allocations, of which there are three: a host buffer and two device buffers. One device buffer is mandatory to transfer the data from the device to the host, while the second device buffer is used to represent the atom data which were randomly generated by the host. In each z-axis iteration, an ND-range is enqueued, the device copies the kernel output to the host buffer, and the partial output is copied to an output array.

4 Results

To automate the process of acquiring the final results, bash shell scripts were used to execute the three frameworks with different values for the amount of atoms and the size of the grid. The results were obtained by averaging three executions using a Mersenne Twister pseudo-random generator with a fixed seed of 11 to ensure reproducibility between SYCL and OpenCL. Since Mojo does not provide an identical method for pseudo-random devices, it produces different seeded pseudo-random values compared to the C-based implementation, resulting in marginal deviations in the results. Figure 4.1 compares each framework's fixed-load speedup over two test cases, one with a grid size of 500 colored blue, and another with a grid size of 800 depicted in a red color, clearly demonstrating the trend of each framework's performance. Mojo performed similarly to SYCL's USM, with both methods showing an increased speedup on a larger grid size. SYCL's buffer method scored a decreased speedup with a higher grid size, similarly to OpenCL which, ultimately, provided the highest speedup.

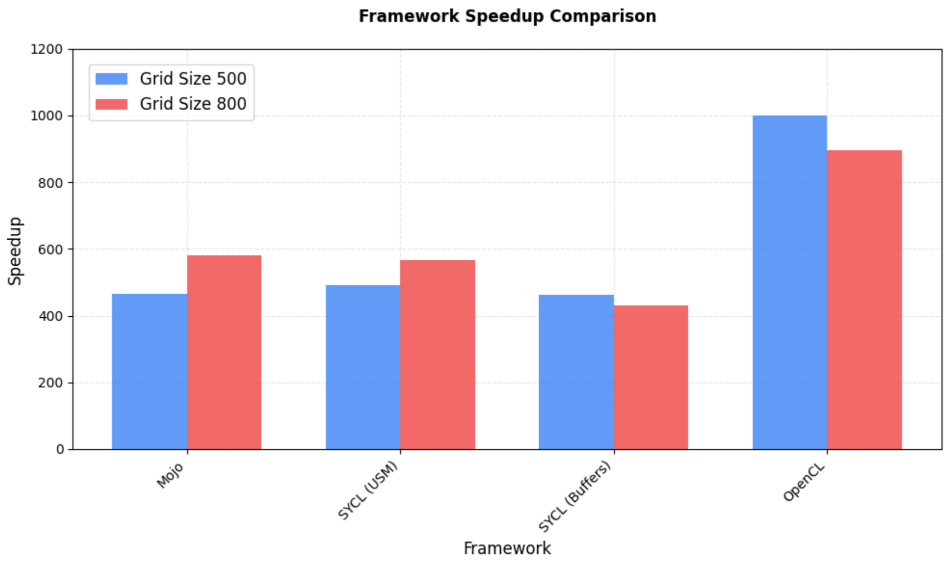

Figure 4.1: Parallel Speedup Comparison

Table 4.1 shows the sequential baseline results compiled with C++ with different grid sizes, showcasing how much grid size slows the execution of the program. With a global size of NxN and an automated local size, Table 4.2 shows that, although OpenCL provides the best results, Mojo showcased an outstanding performance, scoring similar results to SYCL and even outperforming SYCL's buffer method in one test case.. Although OpenCL outperforms Mojo, the latter proves to be competitive, achieving 65% of OpenCL's speedup in the second test case of a larger grid size. Notably, despite AdaptiveCpp's recommendation to use the USM model, both methods resulted in similar results, with the USM method achieving better results in the second test of case. Due to OpenCL's low-level memory management and fine-grained control, it allows a high level of control over optimization, resulting in the lowest execution time. In comparison to OpenCL, Mojo and SYCL provide lower amounts of control due to the abstraction layer the two frameworks provide. Although SYCL provides some memory management, it is limited by it's overhead due to it being an API, and surprisingly, with Mojo providing the lowest amount of optimization and development, it still managed to offer an optimized platform.

Parameter	Test 1	Test 2	Test 3	Test 4
Atom Count	1000	1000	1000	1000
Grid Size	100	200	500	800
Exec. Time	2.40 s	19.19 s	300.34 s	1232.24 s

Table 4.1: Baseline Sequential Performance

Framework	**Atom Count**	**Grid Size**	**Exec. Time**	**Speedup**
Mojo	1000	500	0.64 s	466.46
SYCL (USM)	1000	500	0.61 s	492.36
SYCL (Buffers)	1000	500	0.65 s	462.65
OpenCL	1000	500	**0.3 s**	**1001.14**
Mojo	1000	800	2.12 s	581.24
SYCL (USM)	1000	800	2.18 s	565.94
SYCL (Buffers)	1000	800	2.86 s	430.11
OpenCL	1000	800	**1.38 s**	**894.91**

Table 4.2: Parallel Performance. *Speedup is relative to baseline shown in Table 4.1*

To provide a deeper understanding of why these frameworks achieved their respective results, attempts were made to assess the low-level GPU instructions by acquiring the PTX (Parallel Thread Execution) files from each framework. Since Mojo does not embed GPU instructions inside the executable, nor does it offer an option to offer a PTX file, and since SYCL failed to provide a PTX file due to an error in compilation, only OpenCL was able to offer GPU instructions via the PTX file by using OpenCL's built-in program information command. Because only one PTX was created, no comparisons could be made to suggest a reason why any one framework did not achieve results close to OpenCL. In addition to analyzing low-level code, Linux's time command was used to summarize the usage of system resources for each executable. This time command highlighted that Mojo's implementation included an average of 1320% CPU usage over 64 logical units, averaging a 22% thread usage, which correlates to the amount of logical units inside one cluster's socket with hyperthreading. To create an equal comparison, all frameworks were restricted to one single CPU core using the taskset command. This restriction noticeably affected Mojo's results, increasing execution time by a factor of 46.8%, resulting in an execution time of 3.13s, depicting that Mojo contains a higher CPU reliance despite the heterogeneous focus. In addition to Mojo's CPU usage, both SYCL implementations reserved up to 10% of one additional thread. Finally, the performance analysis tool for NVIDIA devices, NVIDIA Nsight Systems (NSYS), was used to profile each executable, providing insight into data transfer and overall GPU activity. Regarding memory transfer, the test case of 1000 atoms and a grid size of 800 showed that all frameworks communicated 2048MB ($800 \times 800 \times 800 \times 4B$) in 800 calls from device to host and 0.016MB ($1000 \times 4 \times 4B$) in 1 call from host to device, except for SYCL's buffer method which transferred an additional 528MB with 1 call from device to host, and an additional 2560MB with 2 calls from host to device due to how SYCL handles automatic buffer synchronization. Host-to-device memory transfer is negligible (around 3000ns), but device-to-host transfer varies for each framework; it took OpenCL 0.207s, for Mojo it took 0.859s, SYCL's USM took 0.925s, and it took 0.001s for SYCL's buffer method. Finally, nsys summarized the OS runtime to be the following: OpenCL waited 81% for thread synchronization and 17% for I / O. Mojo waited 94% for I/O and 5% in device control. SYCL's USM waited 63% in thread synchronization and 34% for I/O. Finally, SYCL's buffer waited 56% in thread synchronization, 20% waiting for I/O, and 20% in nanosleep. This analysis provides insight as to how each framework handles heterogeneous workflow; all frameworks offer some way of communication between the CPU and the GPU, whilst Mojo distributes some tasks to the GPU which is why Mojo's summary shows the largest wait time in I/O.

5 Conclusion

Three different parallel frameworks – Mojo, SYCL, and OpenCL – were evaluated by performing a DCS computational method using an HPC cluster. Consistent tests were conducted to determine the performance of each framework, of which OpenCL delivered the best performance, achieving speedups of up to 3x over the alternatives. Despite OpenCL's results, Modular's Mojo, while still under development, demonstrated promising results, managing to score similar, or higher, speedup compared to SYCL, with speedup being 65% of OpenCL's speedup in one test case and. It is worth noting that although SYCL scored the lowest performance, it featured developer accessibility at the cost of performance, compared to OpenCL, which offers the highest performance gains but the lowest user-friendliness, with Mojo offering a compelling balance between performance and usability.

References

[1] David B. Kirk and Wen-mei W. Hwu *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier Inc., 2010