# 13X007: Assignment #7

Due on 30.11.2023

*Parallelism*

UNIVERSITÉ
DE GENÈVE

**CHRISTOFOROU Anthony**

# Contents

# 1 Introduction

This report evaluates a heat simulation application, focusing on its parallelization approach and execution on different hardware configurations. The application is designed to solve a 2D heat equation, a typical problem in computational fluid dynamics and heat transfer simulations. The challenge lies in efficiently parallelizing the computations to leverage the capabilities of modern CPUs and GPUs. The parallelization strategy involves using standard C++ libraries and execution policies tailored to each hardware's strengths.

# 2 Methodology

## 2.1 Hardware Employed

- **Yggdrasil HPC:** Powerful Cluster. For this project we will use 32 Cores and a Single GPU (Unkown CUDA Device). (We used the `yggdrasil` cluster instead of the `baobab` because we had some issues with using the asked values for the `sbatch` file)
- **Personal Computer:** A setup comprising 12 CPU cores and an RTX 2070 Super GPU.

## 2.2 Implementation Details

- **Data Structure:** The 2D domain of the heat simulation is stored in a linear vector, a common technique in high-performance computing to simplify memory access patterns and enhance data locality.
- **Parallelization Technique:** The application utilizes `std::for_each` combined with execution policies from the C++ Standard Library. This approach enables efficient parallel processing on multi-core CPUs.
- **CPU Execution:** On CPUs, the parallel execution policy (`std::execution::par_unseq`) is employed, allowing for unsequenced, concurrent execution across multiple threads.
- **GPU Execution:** The GPU implementation details were not provided in the initial code snippet. However, it typically involves using GPU-specific programming models like CUDA or OpenCL for parallel processing.

## 2.3 Expanded Implementation Details

### 2.3.1 Utilizing std::for_each and Lambda Functions

`std::for_each` in the C++ Standard Library is employed to apply a function over a range of elements, and is a key part of parallelizing the heat simulation. This function template allows for the specification of an execution policy, enabling parallelization and vectorization.

Lambda functions in C++ offer a concise way to define operations to be performed on each element. These anonymous functions can capture variables from their enclosing scope, making them suitable for defining complex operations inline within `std::for_each`.

### 2.3.2 Using Value References vs. Pointers

The choice of using value references over pointers in C++ is significant for safety and efficiency, especially in parallel computing contexts. Value references, indicated by `&`, cannot be null and do not require dereferencing, reducing the risk of errors and improving code readability. They are also more efficient in certain scenarios due to reduced indirection.

#### 2.3.3 Lambda Capture Clause

Lambda capture clauses (`[&]`, `[=]`, etc.) determine how external variables are captured within the lambda function. Capturing by reference allows the lambda to modify external variables and is more efficient for large data structures. Capturing by value, however, is safer in concurrent environments as it avoids potential race conditions by creating a copy of the variables.

# 3 Results

## 3.1 Execution Time Ratio

The execution time ratio between CPU and GPU implementations was calculated based on the provided data:

- **Yggdrasil HPC:** CPU (32 cores) execution time was significantly higher than the GPU.
- **Personal Computer:** The execution times for the CPU (12 cores) and GPU (RTX 2070 Super) were almost identical.

## 3.2 Performance Metrics

1. **On Yggdrasil HPC:**
   - **CPU (32 cores):** Marked an execution time of 3,121,081 microseconds.
   - **GPU:** Recorded an execution time of 1,832,123 microseconds.
2. **On Personal Computer:**
   - **CPU (12 cores):** Notched an execution time of 27,182,565 microseconds.
   - **GPU (RTX 2070 Super):** Clocking in at 25,766,272 microseconds.

## 3.3 Graphical Representation

A bar graph vividly depicts the execution times, contrasting the performance across the varied hardware configurations.

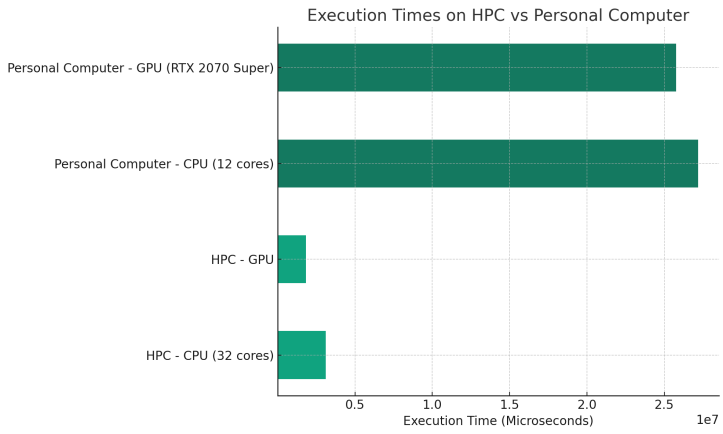


Figure 1: Execution Times

# 4 Discussion

## 4.1 Performance Analysis

- **CPU Performance:** The application demonstrates effective multi-threading on the CPU, particularly on the personal computer with fewer cores. The use of `std::for_each` with parallel execution policies contributes to this efficiency.
- **GPU Performance:** The relative underperformance on the GPU in the HPC environment suggests potential challenges, possibly related to memory transfer overhead or kernel optimization.

## 4.2 Parallelization Challenges

- **Scalability:** The application's scalability with increasing CPU cores is a point of consideration, particularly on HPC systems.
- **Optimization:** Tuning the application for optimal performance on GPUs may require addressing specific challenges like memory management and execution efficiency.

# 5 Conclusion

The analysis reveals that the heat simulation application performs comparably on CPU and GPU in a personal computer setup but faces challenges in the HPC environment, particularly in GPU optimization. The findings underscore the importance of tailored parallelization strategies and optimization techniques for different hardware configurations in high-performance computing applications. Further exploration and optimization, especially of the GPU implementation, could lead to significant performance improvements.