A07 - Assignment 7

Student: **Deidier Simone - 133020**

Answers to the theory questions

Theory Questions

1. In CUDA, functions can be classified based on where they are executed and where they can be called from. A function declared as `__global__` is a **kernel function**. This type of function is launched by the CPU (*host*) but is executed on the GPU (*device*). When using `__global__`, the function is accessible from the host and can be called with specific launch parameters to define the number of threads and blocks to be used for its execution. A function declared as `__device__` is a **device function**, meaning it is both called and executed on the GPU. This type of function cannot be called from the host; it can only be invoked by other functions that are already executing on the device (*e.g., `__global__` or `__device__` functions*). The `__device__` functions enable code reuse and modularity on the GPU, similar to inline functions in host code, and do not use launch parameters like `__global__` functions.

2. Parallelizing with CUDA offers several advantages over MPI, particularly for applications suited to GPU acceleration. CUDA has advantages over MPI for **massive poarallelism**, in fact CUDA leverages the hundreds or thousands of cores on GPUs, providing extensive parallelization for tasks that are computationally intensive and data-parallel. This is often much faster than what MPI can achieve with a limited number of CPU cores. Also in **memory bandwidth**, CUDA takes advantage of the GPU's high memory bandwidth, allowing for faster data transfer and efficient memory usage in parallel operations. GPUs typically have much higher bandwidth compared to CPUs, which benefits applications that require large volumes of data movement. In **simplified programming for data parallel tasks** CUDA provides a framework that simplifies the parallelization of operations where the same instructions need to be applied to many data elements (*SIMD paradigm*). This can reduce the programming complexity relative to MPI, which often requires explicit data decomposition, communication, and synchronization across distributed CPU processes. For **fine-grained parallelism**, in fact CUDA supports fine-grained parallelism using warps, which can be beneficial for applications needing synchronized and small-scale parallel computations within larger tasks. MPI, on the other hand, typically operates on a coarser level with inter-process communication, which can lead to higher latency for similar operations.

3. **Pros** of the cooperative groups are **simplified synchronization**, cooperative groups provide a more structured way to synchronize threads within blocks and even across multiple blocks, which can be essential for complex

algorithms that require precise coordination, **flexibility in grouping threads**, in fact cooperative groups allow developers to define groups within blocks or across multiple blocks in a grid. This flexibility can improve performance and make code more readable, as the logical grouping of threads can reflect algorithmic requirements more closely. **Performance optimization**, by enabling finer control over thread synchronization and communication, cooperative groups can reduce the need for global synchronization, helping improve overall GPU efficiency and throughput. **Cons** of using the cooperative groups are the **hardware dependency**, the use of cooperative groups, especially those that span multiple blocks, is restricted to specific hardware that supports CUDA's Cooperative Groups API (*typically only newer GPU architectures*), **limited portability**, because not all CUDA-compatible devices support inter-block cooperative groups, applications using this feature may not run on all GPUs, potentially limiting the code's portability and **potential complexity**, in fact, while cooperative groups simplify some synchronization patterns, they also introduce new abstractions that require understanding of group management, which can add complexity to debugging and may have a learning curve.

4. With an achieved occupancy of **0.02**, there is substantial room for improvement. Low occupancy may indicate that the kernel is not fully utilizing the GPU's resources and to improve occupancy, several strategies could be applied like **optimize block size and grid dimensions** ensuring that block sizes are optimized for the GPU's architecture (*e.g., in multiples of 32 threads for optimal warp utilization*) can help increase the active warps per multiprocessor and **minimize shared memory usage per block**, in fact reducing shared memory usage per block allows more blocks to reside concurrently on a multiprocessor, which can improve occupancy.