

# GPGPU: Parallelization of the sciddicaT model using CUDA

Alessandro Fazio - 242422

February 8, 2024

## 1 Abstract

The assignment of this project is to parallelize the sciddicaT model using CUDA. The sciddicaT model is a cellular automaton that uses the minimization of differences to simulate non-inertial landslides. The serial implementation is provided, along with a set of test cases. The goal is to parallelize the model using CUDA, and to compare the performance of the parallel implementation with the serial one.

## 2 Introduction

The sciddicaT model is defined as follows:

$$SciddicaT = \langle R, X, S, P, \sigma \rangle$$

where:

- $R$  is the set of cells in the grid -  $X$  is Von Neumann's Neighborhood -  $S$  is the set of states -  $P$  are the parameters of the model -  $\sigma$  is the transition function

Furthermore, the model is defined as a cellular automaton, where each cell is defined as follows:

$$Cell = \langle r, c, h, s \rangle$$

where:

- $r$  is the row of the cell -  $c$  is the column of the cell -  $h$  is the height of the cell -  $s$  is the state of the cell

The transition function can be found in the provided serial implementation.

## 3 Implementations

The assignment requires the parallelization of the algorithm through different CUDA implementations. I will provide a brief description of each implementation. An interesting point of modification is to reset the flows to zero at each iteration, as it is not necessary to keep the flows from the previous iteration. This can be done by adding a `reset_flows` function that sets the `flow` field of each cell to zero.

### 3.1 Straightforward\_Unified

Namely a direct parallelization using the CUDA Unified Memory. This task is the most straightforward one, as it requires only the use of the `__global__` keyword to define the kernel function, and the use of the `'cudaMallocManaged'` function to allocate memory. The kernel function is then called using the `<<< ... >>>` syntax. We only need to replace the for loop with the kernel call, and to replace the memory allocation and deallocation with the CUDA Unified Memory. This implementation lays the foundation for the other implementations, as it provides a simple way to parallelize the algorithm. However, it is not the most efficient one, as it does not take advantage of the shared memory, and it does not use the best memory access patterns.

### 3.2 Straightforward\_Standard

Namely a direct parallelization using the CUDA standard host/device memory. This task is similar to the previous one, but it requires the use of the `cudaMalloc` and `cudaMemcpy` functions to allocate and copy memory to and from the device. After the computation, the memory must be copied back to the host using the `cudaMemcpy` function to be able to compute the **md5 hash** of the result.

### 3.3 Tiled\_no\_halos

Namely a CUDA shared memory based tiled parallelization without halo cells. This task is a great example of caching data in the shared memory, as it requires the use of the shared memory to store the cells of the tile. Caching the data in the shared memory allows for a faster access to the data, as the shared memory is much faster than the global memory. To achieve this, we need to use the `__shared__` keyword to define the shared memory, and to copy the data from the global memory to the shared memory. We also need to use the `__syncthreads` function to synchronize the threads in the block, as the shared memory is only visible to the threads in the same block.

### 3.4 Tiled\_with\_halos

Namely a CUDA tiled parallelization in which block's boundary threads perform extra work to copy halo cells from adjacent tiles in the shared memory. This task is similar to the previous one, but it requires the use of halo cells. Halo cells are the cells that are shared between the tiles, and they are necessary to compute the cells in the tile. To achieve this, we need to copy the halo cells from the adjacent tiles to the shared memory, and to perform the computation using the halo cells.

### 3.5 Tiled\_with\_halos\_larger\_block

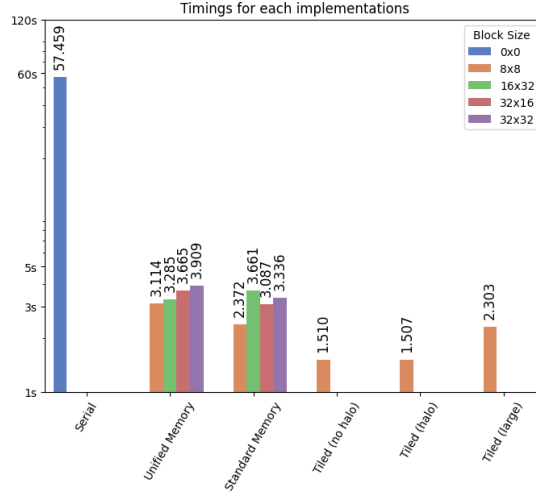
Namely a CUDA tiled parallelization in which the block size is larger than the tile size so that each thread just copies a single cell to the shared memory and only a number of threads equal to the tile size is used to update the tile. Similarly, here we just need to change the block size to be larger than the tile size. Unfortunately, I did not have time to properly implement this task. As a result the md5 hash of the result does not match the expected one, but the workflow is very similar to the previous task.

### 3.6 Tiled\_with\_halos\_larger\_block\_MPI

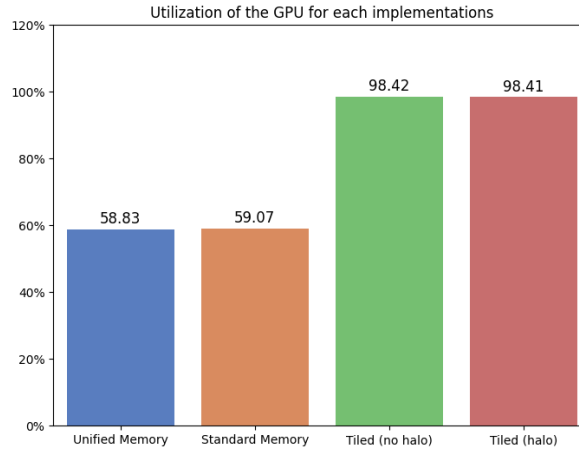
*No implementation provided* because of the lack of time. A possible implementation could be to use the MPI library to parallelize the algorithm across multiple nodes. This would require the use of the `MPI_Send` and `MPI_Recv` functions to send and receive the data between the nodes, and the use of the `MPI_Barrier` function to synchronize the nodes. We can simply divide the grid equally among the devices and let each device compute the cells in the tile taking care of the halo cells.

## 4 Results

The results of the parallel implementations will be compared with the serial implementation. The performance will be evaluated using the `nvprof` tool, and the correctness will be evaluated using the provided test cases. The plots will be generated using the `matplotlib` library<sup>1</sup>.

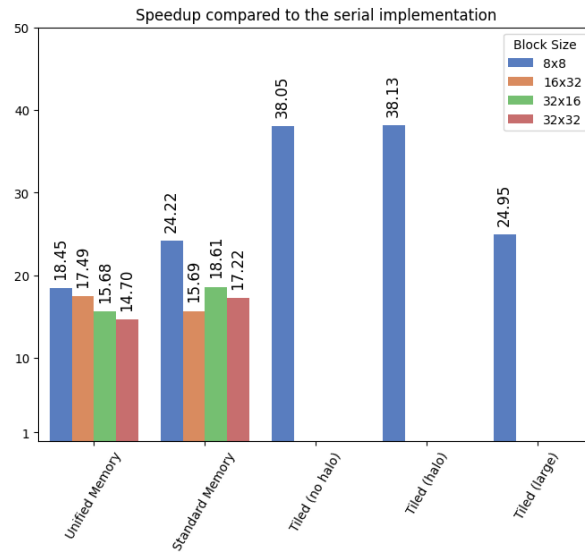


The results show that the parallel implementations are faster than the serial one, and that the Tiled (no halo) has the best performance. The correctness of the results is also guaranteed, as the md5 hash of the result matches the expected one.



<sup>1</sup>The missing timings are a result of the lack of time to properly implement the task, as such the md5 hash of the result does not match the expected one.

Here we can see the occupancy of the kernels. The occupancy is the ratio of the active warps to the maximum number of warps that can be active on a multiprocessor. The occupancy is an important metric, as it indicates how well the kernel is utilizing the resources of the GPU. Again, the Tiled (no halo) has the best performance, as it has the highest occupancy (it's a tie with the Tiled (with halo)). The Straightforward (Unified) has the lowest occupancy, as it does not take advantage of the shared memory, as expected.



The speedup is the ratio of the time taken by the serial implementation to the time taken by the parallel implementation. The speedup is an important metric, as it indicates how much faster the parallel implementation is compared to the serial one. The Tiled (no halo) has the best performance, as it has the highest speedup.

## 5 Roofline Model

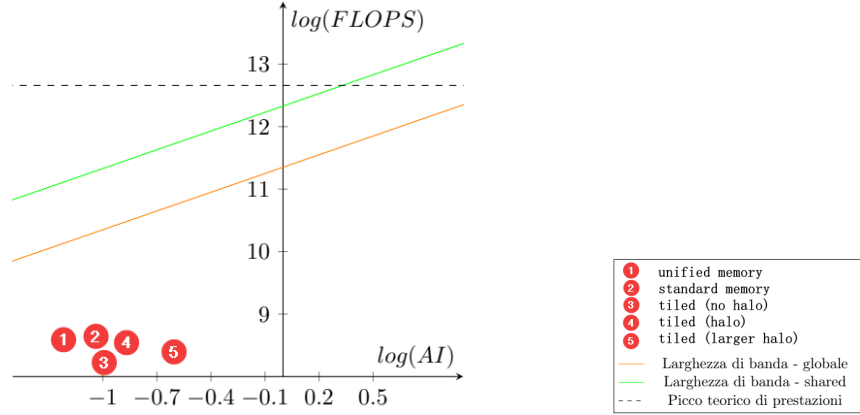


Figure 1: Roofline model for the implementations

Figure 2: legend

From the graph, it can be seen that all kernels are very much "bandwidth bound". In other words, the bandwidth used by the kernels acts as a bottleneck for the overall performance of the algorithm. The kernels simply do not use a sufficient amount of data to reach the theoretical peak performance of the GPU. As already mentioned several times, the domain of sciddicaT is relatively small in size, especially for a powerful GPU like the GTX 980.

### 5.1 Comments

An important clarification is that assessing the performance of the kernels was performed on a different number of iterations as it was unfeasible to run the kernels on the same number of iterations as the serial implementation. That would explain why the best performing kernel in terms of speedup is not the best performing kernel in terms of absolute time and occupancy.

## 6 Conclusion

The parallel implementations are faster than the serial one, and the correctness of the results is guaranteed for most configurations of Block Size. The difficulty of this task was mainly the lack of time for each implementations, as such the md5 hash of the result does not match the expected one for a few implementations. It was a hard time trying to debug why and I've yet to find the reason. The Tiled (no halo) has the best performance, as it has the highest occupancy and speedup. The Straightforward (Unified) has the lowest occupancy, as it does not take advantage of the shared memory. It was a very interesting project, and I learned a lot about parallel programming and CUDA. The main takeaway is that the hardest part of parallelizing an algorithm is to take advantage of the resources of the GPU. The trick is to map efficiently the data to the threads, and avoid at all costs expensive memory accesses. It was very satisfying to see the performance of the parallel implementations and to compare them with the serial one using tried and true technologies like CUDA and `nvprof`.