

Performance enhancement using CUDA in a simulation of heat diffusion

Gonalo Loureno
n°55780
gm.lourenco@campus.fct.unl.pt

Joana Faria
n°55754
js.faria@campus.fct.unl.pt

I. INTRODUCTION

This assignment aims to optimize a base code, written in C, that computes a simulation for heat diffusion. For the optimization, we seek to take advantage of GPU programming using CUDA, and explore several alternatives to find the most efficient one.

To find the best performance we will explore different approaches, namely analyzing different configurations, exploring the impact of using shared memory, and the difference between using streams and not.

Knowing that the architecture of the system influences the performance obtained, the first step of our work is to understand the architecture of the system where our program will run. So we present the characteristics below:

```
Device: NVIDIA GeForce GTX 1050 Ti
CUDA Driver Version: 11.4
Runtime Version: 11.2
CUDA Capability Major/Minor version number: 6.1
6 Multiprocessors (SM)
Max Grid sizes: 2 147 483 647, 65 535, 65 535
Max Block sizes: 1024, 1024, 64
Max Blocks per SM: 32
Max threads per Block: 1024
Warp size: 32
Total global memory: 4 137 024 MB
Shared Mem per SM: 98 304
Max ShMem per Block: 49 152
```

II. RESULTS

To accurately compare the different versions all the tests were performed in the university cluster and averaged the execution times over ten execution. The parameters chosen were:

```
const int nx = 200; // Width of the area
const int ny = 200; // Height of the area
const float a = 0.5; // Diffusion constant
const float h = 0.005; // h=dx=dy grid spacing
const int numSteps = 100000; // Number of time
steps to simulate (time=numSteps*dt)
const int outputEvery = 100000; // How frequently
to write output image
```

A. Sequential Version

We start by analyzing and profiling the sequential version to understand what improvements can be done, and what parts are the more problematic.

By a quick analysis of the code, we expect the cycle present in the `main` function to be the biggest problem. This assumption is supported by the reports of the profiling tools. The output of these tools can be seen in the files `results/sequential/gprof` and `results/sequential/perf`.

The average execution time obtained with this version is 153.979 seconds.

B. V1 Version - CUDA

Next, we make a naive translation of the sequential code to CUDA code, which can be found in `proj1/v1.cu`. This implementation was based on the tutorial provided with the project, with some adaptations.[1]

With this version, we experimented with a variety of grid configurations. We follow Nvidia recommendations[2]:

- Threads per block should be a multiple of warp size to avoid wasting computation on under-populated warps and to facilitate coalescing.
- A minimum of 64 threads per block should be used, and only if there are multiple concurrent blocks per multiprocessor.
- Between 128 and 256 threads per block is a good initial range for experimentation with different block sizes.

So we start with a configuration having 256 threads per block and we test until the maximum of 1024 threads per block. We didn't find significant differences between these configurations.

For the sake of completeness, we tested with blocks smaller than 68 threads, and, as expected, we got worse results. The results are summarized in Figure 1

Analyzing this version using the profiling tool NVPROF we can conclude that the majority of execution time is spent in communication between the host and the device, with only 22.19% of the execution time dedicated to kernel execution. The program spends 41.74% of time copying data from the host to the device and 36.07% from the device to the host. The complete results are available at `results/v1/nvprof_results`.

C. V2 Version - Better communication

In this version, we intend to improve the previous naive implementation, reducing the communication between the host and the device, which we detect to be a big problem

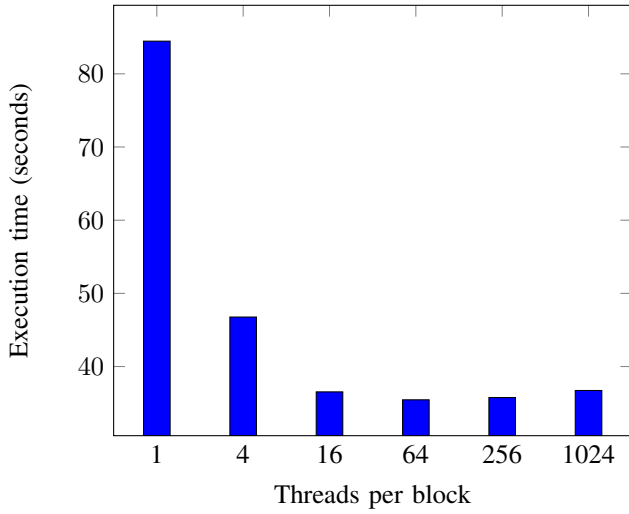


Fig. 1: Execution time of different grid configurations for V1

in the previous version. In this problem, there aren't any computations needed between the steps so we don't need to transfer the data to the host between steps.

As expected we got a huge improvement in execution time. This version has an average execution time of 2.106914 seconds. Using NVPROF we can see that now the kernel is executing 100.00% of the total execution time. The full report is in `results/v2/nvprof_results`.

We decided, from now on, to increase the complexity of the problem by increasing the amount of work, to better spot the improvements. So we define `numSteps = 100000`, and maintain the output of the image in the beginning and end.

With this new parameter, the average execution time is 25.280373 seconds.

D. V3 Version - Shared Memory

Looking deeper into our current solution we realize that the same array position is accessed by five different threads, thus revealing a great opportunity for the use of shared memory.

In this version, we take advantage of the shared memory between threads of the same block. Each thread starts by copying a value to the shared memory and after synchronization, the thread computes the designated value.

This version had an average execution time of 2.199574 seconds, which doesn't show any improvement from the previous version. This may be because of the overhead introduced by copying each element and the threads' synchronization is not balanced by faster access to the shared memory.

E. V4 Version - Streams

After analyzing our version with shared memory we focused decided to add streams to this version despite not expecting any big improvement when working with a small grid.

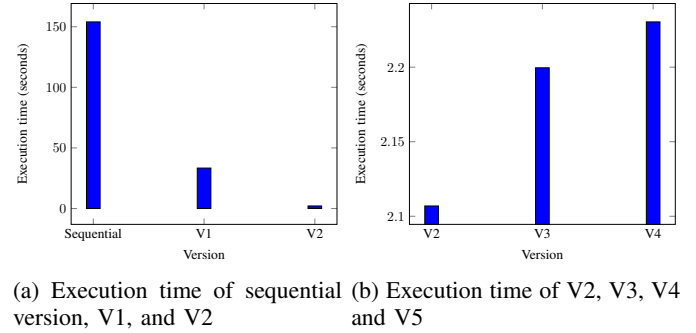
Our implementation adds streams that allow the device to run as soon as the first partition of our table is downloaded. Since our function works with smaller squares we have a two-dimension cycle in order to copy the necessary data.

We were already expecting the results we got, an average execution time of 2.230349 seconds when using 16 streams (4*4). Even if it is worse than our previous time, it's to be expected because our grid is too small. The majority of the time is spent on the execution of steps, so, the time saved in execution doesn't balance the time lost with the creation and initialization of streams.

We expect streams to prove advantageous when we have a larger amount of data to transfer between the host and the device. To prove this hypothesis we increase the grid size to 2000x2000 and compare the different versions, showing the results in

Another situation in that streams may be helpful is we want the output in intermediate steps, allowing the communication to overlap with the computation of the next step. So, once again, we rerun the different version with `outputEvery = 1000` to compare the execution time. The results can be seen in

III. RESULTS



(a) Execution time of sequential (b) Execution time of V2, V3, V4 version, V1, and V2 and V5

Fig. 2: Execution time of the different versions

REFERENCES

- [1] "Solving heat equation with CUDA — OpenACC/CUDA for beginners." (), [Online]. Available: https://enccs.github.io/OpenACC-CUDA-beginners/2.02_cuda-heat-equation/?utm_source=pocket_saves (visited on 11/09/2022).
- [2] "CUDA C++ Best Practices Guide." (), [Online]. Available: <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html> (visited on 11/09/2022).