

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:

```
return (t->tv_sec - t0->tv_sec) + 1.0e-9 * (
    double)(t->tv_nsec - t0->tv_nsec);
}

int main()
{
    const int nx = 200;           // Width of the
    area
```

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.**SolvingHeatEquation**

With this version, we experimented with a variety of grid configurations. We follow Nvidia recommendations**CUDABestPractices**:

- 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 ??

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`. From this iteration, we concluded that the better configuration is to use 16x16 threads. We will use this configuration in the next versions.

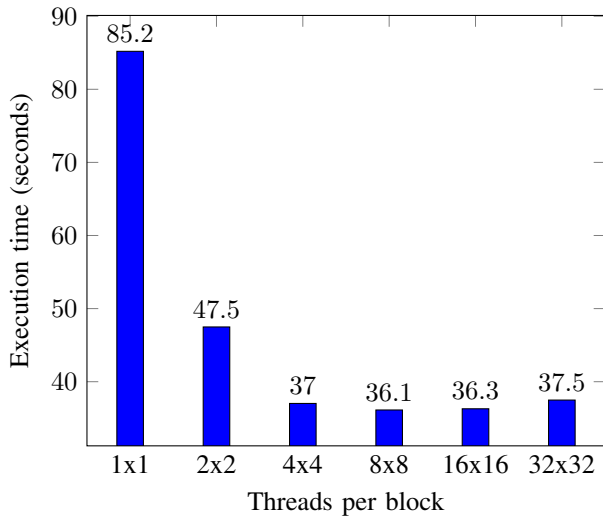


Fig. 1: Execution time of different grid configurations for V1

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 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 2.172697 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 want to understand the impact that this new communication profile is affected by the number of threads so we conduct an analysis similar to the previous one, showing the results in ??.

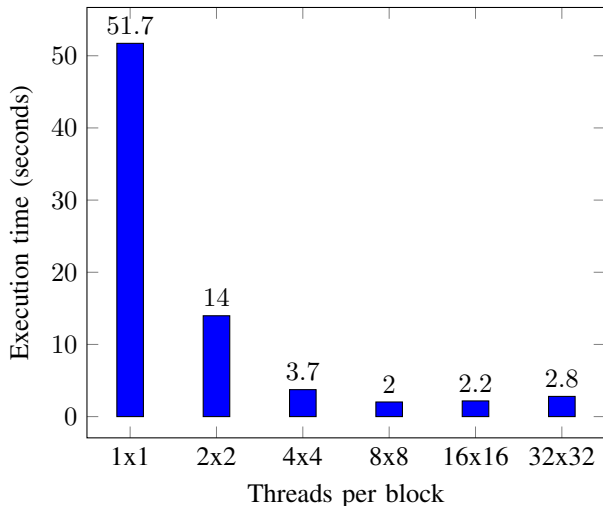


Fig. 2: Execution time of different grid configurations for V2

We maintain the use of the 16x16 configuration as the default for this version.

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 2.206946 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 with shared memory

After analyzing our version with shared memory we 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 2.203838 seconds when using 16 streams (4*4). Even if it is worse than our previous time, it's to be expected because of a few factors between which the biggest is that the majority of the time is spent on the execution of steps, so, the time saved in the execution of the first step is relatively small.

F. V5 Version - Streams

After V4 we already knew that since we weren't solving any of the major issues listed earlier we would get similar results, however, for the sake of verifying our expectations, we decided that the work implementing the same stream system as before but without the shared memory could be justified. In conclusion, we confirmed our previsions without any improvements whatsoever.

G. V6 Version - Streams to copy output

We expect streams to prove advantageous when we want the output in intermediate steps, allowing the communication to overlap with the computation of the next step. To perform this analysis we compare the V2 and V6 versions, varying the parameter `outputEvery` to values of 1000, 100, 10, and 1.

For larger values of `outputEvery`, we cannot see significant differences revealing that the costs required for synchronization are not balanced by the overlap of communication with synchronization. But with lower values, we have begun to see the advantages in the use of streams. We can see the results in ??.

With the results on every step is where we see the better improvement allowed by streams, in the end of every step we

can overlap the execution of the next step with the transfer of the previous result and its writing on a host file.

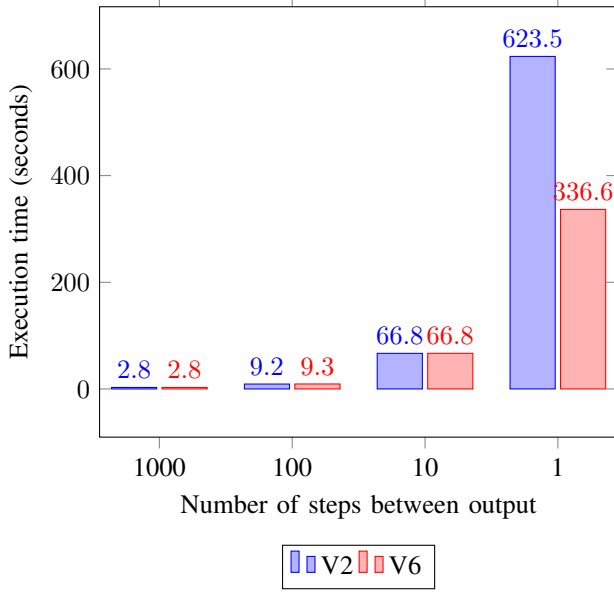


Fig. 3: Comparing execution time of V2 and V6, varying outputEvery

Another situation in that streams may be helpful is when we have a larger amount of data to transfer between the host and the device. To prove this hypothesis we vary the grid size and compare the different versions, showing the results in ??.

As we can see the use of streams does not appear to improve the performance when using larger data sets. This may be because the communication time is greater than the execution time of each kernel.

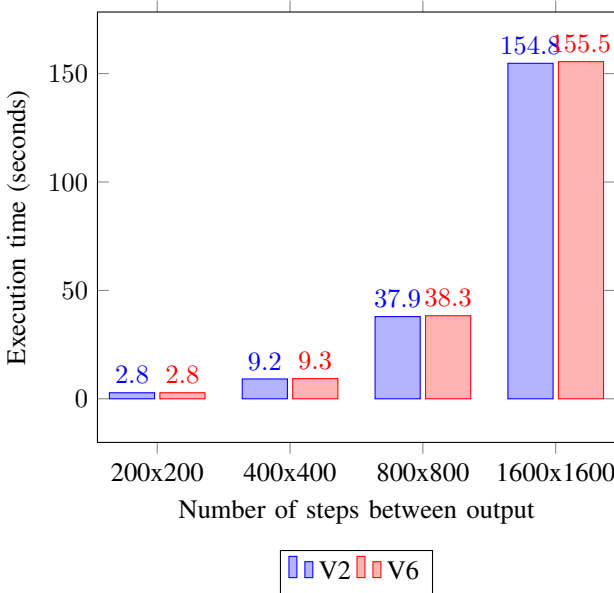
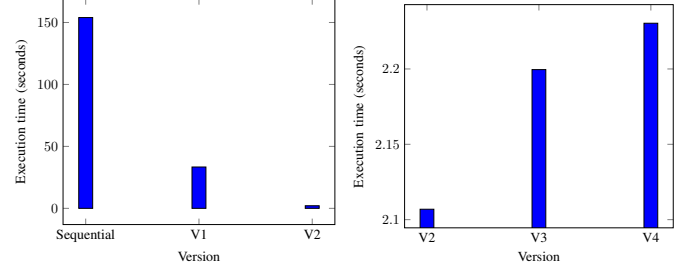


Fig. 4: Comparing execution time of V2 and V6, varying the input size

III. RESULTS



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

Fig. 5: Execution time of the different versions

A. V9 Version - More Work p/ Thread

Being a bit more creative in our approach we decided to have a different evaluation process. In this version, we went back to our V2 and changed it in order for each thread to process more than a single point of the table.

Our first attempt was to adapt each thread to process one line of values. This experiment turned out to be a horrible solution with 250+ seconds of execution in the same conditions as the cases above, but as we said this was only the first attempt. The result was surprising because we weren't expecting such a high value but we were basically overworking our threads, with 256 threads per block and 200 lines to process we were making a step work in "less" than one block, far from maximizing the use of our GPU.

What we did after this was precisely to work our texttt-threadSize, the number of elements that our thread would process.

B. V10 Version - More Work p/ Thread w/ Shared Memory

Finally, our last implementation was to make use of shared memory together with the V9 implementation to make use of the great search speeds when searching a bigger number of values, since starting this memory was also quite a heavy task. This way we were expecting quite an improvement when compared to V9.