Heat Diffusion over 2D Surface Group 1 Report

Jacinta Sousa, and João Bordalo

Abstract—The present paper presents the steps followed to improve the performance of a slow sequential program for heat diffusion simulation over a 2D surface. The goal was to reduce its execution time leveraging the GPU. We tried different optimization strategies, such as altering grid configuration and block size, increasing the amount of work per thread, overlapping communication and computation and using GPU shared memory. We found that solely leveraging GPU execution, without any further optimization concerns, greatly improved performance. We highlight that our best results were achieved by using 1D grid configurations, overlapping communication and computation and using GPU shared memory.

Index Terms—Stencil Pattern, Parallel Programming, Performance, CUDA, GPU Programming

1 Introduction

The provided code, simulates heat diffusion over a 2D surface and runs sequentially. As the surface considered grows, the program takes longer to execute, since the number of calculations and runs through the surface matrix that represents the heat of each point in a certain time step rises. With that problem at hand, the approach planned to solve it consists of taking advantage of the possibility to parallelize the computations using CUDA programming to enable running code on the GPU.

2 Methods

We followed the article [1] provided in the project statement to arrive at the initial solution of the problem as well as to introduce some of the optimizations. The method employed followed an identify-change-test-benchmark policy. To this end, we would identify a possible problem with our current version or possible optimization and make a fixing change. After changing the implementation, we assured correctness of the program by comparing its output with that of the original version. To assess the effectiveness of the changes made to the program, we benchmarked and profiled it, to obtain average execution times.

We worked from the original C program, using it as a baseline and programmed the equivalent in CUDA. Following that, we progressively made optimizations, some of which leveraging the previously fastest option and some of which starting from the same CUDA base. In the end we ended up testing different grid configurations, varying the number of threads, using the GPU's shared memory, varying the amount of work per thread and overlapping computation and communication.

All results can be found in section 3 and were obtained with the same program parameters, $nx = ny = 200$ and $h = 0.005$.

- *Jacinta Sousa, 55075, and João Bordalo, 55697, are Master's students in the Department of Computer Science, NOVA School of Science and Technology, Lisbon.*

3 Experiments

The original C program, which we call the *sequential* version is used as a baseline. We refer to our first CUDA solution as a naïve implementation since it parallelizes the heat diffusion computation with no regard for memory or code efficiency. Following this initial naïve version of the program, we optimized memory efficiency by reducing the number of calls to *cudaMemcpy* to the strictly necessary. The article [1] refers to this as moving the ownership of the data to the device. This version of the CUDA program is the one we use as a base for testing different grid configurations and block sizes.

For reference, table 5 shows all the obtained results.

The following table shows execution times over several runs of three versions of the program: sequential, naïve and cuda with an 8×8 grid.

As one can see from the results, the naïve version, despite neglecting optimization, already shows how leveraging the GPU for these types of problem can be beneficial. We can achieve a speedup $S = \frac{153.963}{35.185} = 4.376$ without worrying about code efficiency and optimization. By refactoring, so as to take into account memory efficiency, we arrived at a stable CUDA version with a cumulative speedup $S = \frac{35.185}{1.960} = 17.952$ and a total speedup $S = \frac{153.963}{1.960} = 78.553$. We chose the 8×8 grid to compare as it was the base version with lowest execution times. [1]

TABLE 1
Execution times for the sequential, the Naïve Implementation in CUDA, with a 16×16 grid, and the base version in CUDA, with an 8×8 grid.

	Mean	Maximum	Minimum
Sequential	153.964	156.053	153.697
CUDA 2D Grid			
Naïve Implementation	35.186	35.279	35.096
Base Implementation (8×8)	**1.960**	**2.018**	**1.948**

1. We're not comparing the two CUDA versions directly but illustrating the difference between the naïve version and an optimized version.

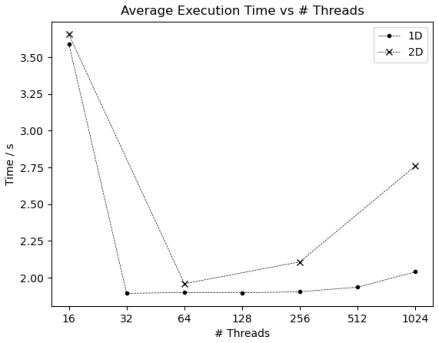

Fig. 1. Average execution times per # threads for different grid configurations

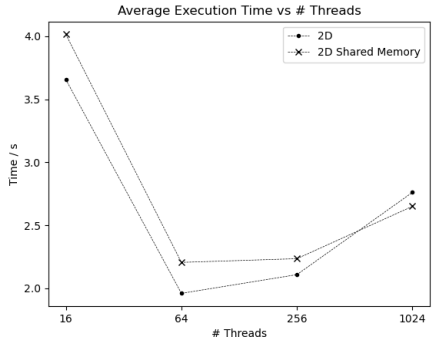

Fig. 2. Average execution times of the 2D version per # threads with and without shared memory

3.1 Strong Scaling and Grid Configurations

We tried different grid configurations and varied the number of threads so as to arrive at the fastest possible configuration and leverage it for further optimizations.

Figure 1 shows the average execution times as a function of the number of threads for two different grid configurations.

By analyzing the figure we can see, firstly, the behaviour of the program as the number of threads increases, known as strong scaling. We can see that both versions improved a lot as the number of threads grew from 16 to 64. Scaling from 64 to 256, for the 1D version has little impact on the execution time and ends up taking longer for the 2D version. Scaling it further to 1024 threads increases average execution times for both versions.

Secondly, it shows how the two different grid configurations compare. We only draw conclusions from the points that share the number of threads. For example, a 32×32 grid in the 2D version is comparable with the 1D version with 1024 threads. For a low number of threads, the two versions are very similar, with the 1D version being slightly faster as we scale up to 64 threads. As we increase the number of threads to around 256, we can see that the difference starts increasing, with the 1D version being faster.

We hypothesized the decrease in the number of operations executed in the 1D version can be a factor, as there is less need for index calculations and they're faster, with less multiplications.

3.2 Shared Memory

We experimented with using the device's Shared Memory to improve performance. The computation at hand reuses the same memory cells up to 5 times so the shared memory approach is an adequate attempt. To assess performance, we varied the number of threads on the shared memory version and compared to the version without it. We did this for both grid configurations to see which one had better results.

Figures 2 and 3 show average execution times as a function of the number of threads for our 2D and 1D versions with and without shared memory.

In the 2D version, using Shared Memory only improved results for 1024 threads, performing close to but worse than the initial version for other number of threads.

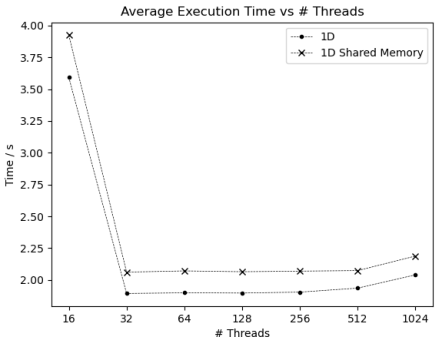

Fig. 3. Average execution times of the 1D version per # threads with and without shared memory

In the 1D version, the reported results show that the shared memory approach didn't improve execution times. In our initial tests for this approach, we found that it could be slightly faster than the version without shared memory. However, the final benchmark doesn't reflect this.

3.3 Work per Thread

We tried increasing the amount of work each thread does as a possible optimization strategy. The idea was to have a thread calculate more than one pixel. For the 1D grid configuration, the thread simply accesses the next pixel, doing this as many times as the increase of the user defined work per thread. This access pattern may not be the best in terms of cache hits since now threads start their execution separated by a stride equal to the defined work per thread.

For the 2D grid, the work follows the same logic as the 1D grid but in two dimensions, so each thread does a square of pixels. The given parameter, WPT, represents the size of the side of the pixel square. The total work made by a thread is WPT^2.

In tables 2 and 3, we can see that, for both grid configurations, increasing the amount of work per thread also increases the execution times of the program, compared to the implementation in which each thread calculates a single pixel.

As mentioned before, memory access patterns may deteriorate performance, increasing the number cache misses.

TABLE 2
Execution times for the CUDA implementation using an 8×8 grid with varying work per thread.

Work per Thread	Mean	Maximum	Minimum
1×1	1.960	2.018	1.948
2×2	12.806	12.867	12.743
3×3	108.614	108.737	108.122

TABLE 3
Execution times for the CUDA implementations using an 1D grid of 32 and 64 threads with varying work per thread.

	Mean	Maximum	Minimum
32 Threads			
$1\times$ work per thread	1.894	1.894	1.893
$2\times$ work per thread	6.591	6.592	6.591
$4\times$ work per thread	25.370	25.946	24.801
$8\times$ work per thread	96.247	98.068	95.746
64 Threads			
$1\times$ work per thread	1.899	1.903	1.898
$2\times$ work per thread	3.439	3.450	3.437
$4\times$ work per thread	12.942	13.381	12.777
$8\times$ work per thread	49.344	50.373	48.924

3.4 Overlapping Computation with Communication

The final optimization we tried, was overlapping computation and communication.

In this approach, while we're uploading a given partition of the data to the GPU, the kernel can be applied to another. Simultaneously, a third partition can even be downloading from the GPU. The goal is to hide the cost of communication between the host and the device.

We implemented it on top of a 1D Grid with shared memory since experimental results pointed to it being faster (see section 3.2).

Figure 4 shows how the average execution time changes as we increase the number of parallel streams.

We can see improvements with this approach, by comparing the version with 2 streams with the results for the original version of 1D Grid, 32 threads, with Shared Memory (see table 4).

However, as the number of streams increase, we see that average execution time also increases.

TABLE 4
Execution times for fastest CUDA 1D grid, 32 threads, with Shared Memory with and without overlapping computation and communication.

CUDA 1D Grid, 32 threads, with Shared Memory	Mean
No Overlapping	2.062
Overlapping, 2 streams	1.891

4 Conclusion

By analyzing the results one can see that GPU programming has benefits and can improve the execution time of sequential programs dramatically. We found that many proposed optimizations, in practice, ended up increase execution time or showing diminishing returns. It may be the case that our problem size is too small or our resources too limited.

Our final and fastest version used a 1D Grid with 32 threads, leveraging Shared Memory and Overlapping Computation and Communication with 2 streams. We had a final speedup of $S = \frac{153.963}{1.891} = 81.419$.

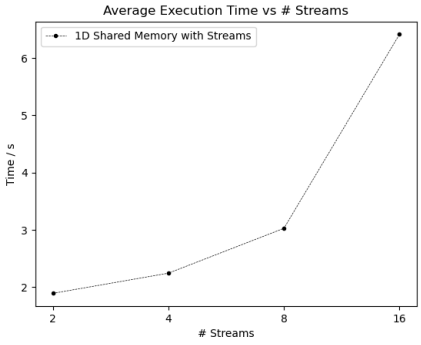

Fig. 4. Average execution times per # streams for a 1D Grid using Shared Memory

Some possible further optimizations include:

1) Improving work per thread with coalesced memory accesses
2) Overlapping Computation and Communication in 1D grid without shared memory

References

[1] Solving heat equation with CUDA (n.d.). Retrieved November 2022, from https://enccs.github.io/OpenACC-CUDA-beginners/2.02_cuda-heat-equation/

TABLE 5
Execution times for every version implemented.

	Mean	Maximum	Minimum
CUDA 2D Grid			
32×32 threads	2.761	2.770	2.754
16×16 threads	2.108	2.114	2.106
8×8 threads	1.960	2.018	1.948
4×4 threads	3.656	3.736	3.641
CUDA 2D Grid with Shared Memory			
32×32 threads	2.650	2.651	2.649
16×16 threads	2.235	2.410	2.215
8×8 threads	2.205	2.206	2.205
4×4 threads	4.014	4.019	4.013
CUDA 2D Grid (8×8 threads)			
2×2 work per thread	12.806	12.867	12.743
3×3 work per thread	108.614	108.737	108.122
CUDA 1D Grid			
1024 threads	2.040	2.042	2.040
512 threads	1.936	2.007	1.928
256 threads	1.905	1.911	1.903
128 threads	1.897	1.898	1.897
64 threads	1.899	1.903	1.898
32 threads	1.894	1.894	1.893
16 threads	3.593	3.593	3.592
CUDA 1D Grid with Shared Memory			
1024 threads	2.188	2.191	2.188
512 threads	2.074	2.075	2.073
256 threads	2.070	2.070	2.069
128 threads	2.065	2.068	2.064
64 threads	2.071	2.091	2.069
32 threads	2.062	2.165	2.049
16 threads	3.924	3.977	3.918
CUDA 1D Grid (64 threads)			
8 work per thread	49.344	50.373	48.924
4 work per thread	12.942	13.381	12.777
2 work per thread	3.439	3.450	3.437
CUDA 1D Grid (32 threads)			
8 work per thread	96.247	98.068	95.746
4 work per thread	25.370	25.946	24.801
2 work per thread	6.591	6.592	6.591
CUDA 1D Grid (32 threads) with Shared Memory and Streams			
16 streams	6.415	7.118	6.156
8 streams	3.027	3.171	2.940
4 streams	2.243	2.244	2.242
2 streams	**1.891**	1.892	1.890