8 JUNE 2025

HIGH PERFORMANCE COMPUTING

CUDA PRACTICE

Oriol Vilalta Arrondo
Joel Lloret Vidal

Escola Politècnica Superior
Universitat de Lleida

Table of contents

Design decisions **2**
Thread mapping 2
CUDA kernel 2
Steps loop 2
Testing the CUDA code **3**
Comparing CUDA to OMP and OMP+MPI **5**
Evaluating CUDA and Moore-UdL performance 5
Evaluation of CUDA's concurrency 5
Our experience with CUDA **7**
Conclusions **8**

Our code can be accessed in this repository.

Design decisions

Thread mapping

The main concept of this approach is to map each CUDA thread to a single cell of the 2D matrix. To do that, the first thing to understand is the thread and grid organization.

```
dim3 block(blockSizeX, blockSizeY);

dim3 grid((ny + blockSizeX - 1) / blockSizeX, (nx + blockSizeY - 1) / blockSizeY);
```

These two variables define the block size and grid size, the latter computed in relation to the matrix dimension. With this structure in mind, we map the threads to each cell as shown below:

```
int i = blockIdx.y * blockDim.y + threadIdx.y;

int j = blockIdx.x * blockDim.x + threadIdx.x;
```

CUDA kernel

Inside the parallel region defined as the CUDA kernel, each thread computes its own coordinates (as seen above) in the 2D grid; and the new value for those coordinates using neighbors from the previous iteration's matrix.

```
if (i > 0 && i < nx - 1 && j > 0 && j < ny - 1)
{
        next[i * ny + j] = current[i * ny + j]
        + r * (current[(i+1) * ny + j] + current[(i-1) * ny + j]
        - 2 * current[i * ny + j])+ r * (current[i * ny + j + 1]
        + current[i * ny + j - 1] - 2 * current[i * ny + j]);
}
```

The conditional statement controls the boundary checks.

Steps loop

The loop executed on the CPU launches one kernel each time step, where *d_current* holds the grid's values at time t and *d_next* will hold values at t+1. After each step, buffers are swapped.

```
for (int t = 0; t < steps; t++)
{
    heat_step_kernel<<<grid, block>>>(d_current, d_next, nx, ny, r);
    std::swap(d_current, d_next);
}
```

Testing the CUDA code

Before testing the modified CUDA code, it is important to first evaluate the serial performance on this new machine. Since we are now working on a different system than the one used in previous assignments, we need to establish a baseline by running the unmodified heat_serial.c code. This involves testing a range of image sizes and different amounts of steps to understand the serial execution characteristics on the current hardware.

Those should be the same as the previous assignments to be able to compare the results afterward.

SERIAL (in s)				
Matrix Size	Steps			
	100	1000	10000	100000
100x100	0,0102	0,0876	0,8499	8,5743
1000x1000	1,0158	10,6156	94,5712	926,1802
2000x2000	4,0712	42,2441	422,2795	3.751,2566

Table 1. Time spent to calculate the result of the serial version.

Firstly, we need to determine the optimal block size to achieve the best performance from our CUDA implementation. To do this, we evaluate our version of the code—already adapted with CUDA—using different block sizes on two grid sizes: 1000×1000 and 2000×2000. We begin by testing block sizes that are powers of 2, progressively increasing until we observe a performance drop compared to the previous configuration. Once that threshold is identified, we narrow our search within the range of block sizes that yielded the best results, fine-tuning to find the most suitable block size for our setup.

CUDA SCALABILITY (1000 steps) (ms)								
Matrix Size	Block Size							
	2	4	8	10	12	14	16	32
1000x1000	366	221	178	171	170	171	176	173
2000x2000	1.184	630	483	483	480	485	479	509

Table 2. Evaluation of the optimal block size.

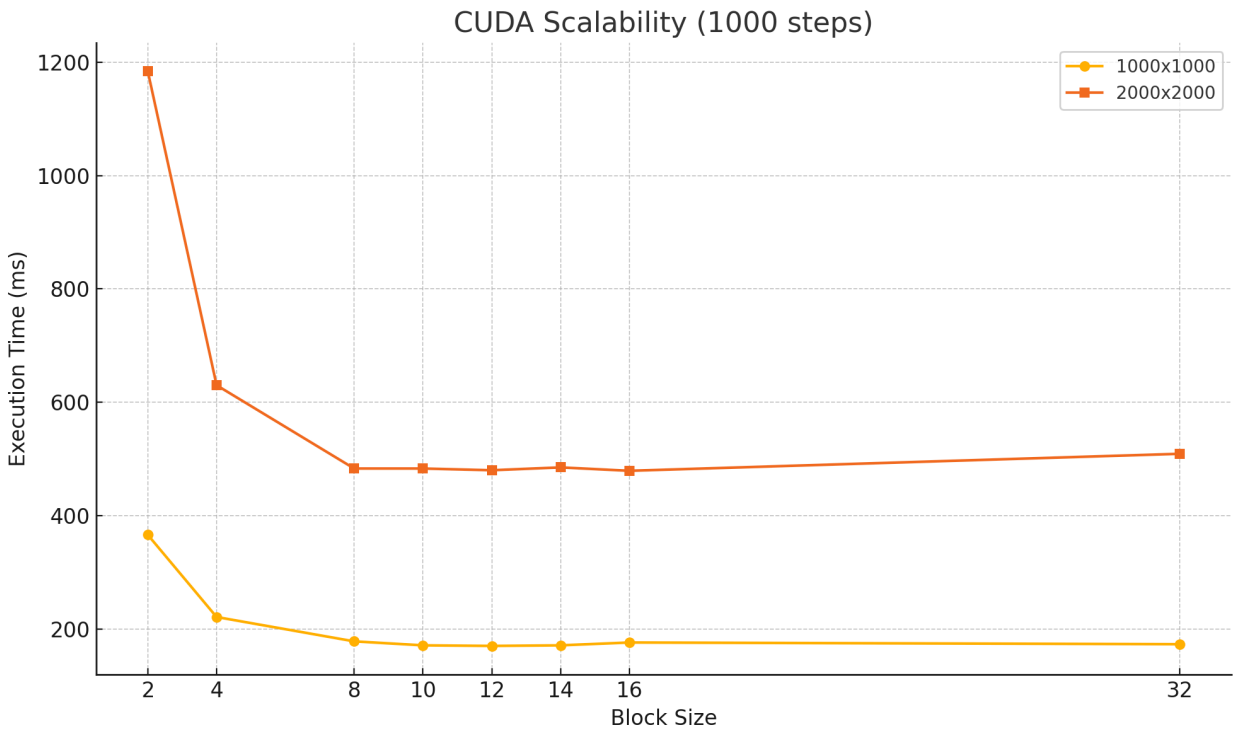

Image 1. Plot of the scalability depending on the block size. (1000 steps)

As shown, performance improves significantly as we increase the block size from 2 up to 12. Beyond this point, the execution time begins to slightly fluctuate or worsen, indicating diminishing returns or even overhead from excessive thread block sizes. While the differences between block sizes 10, 12, and 14 are minimal, especially compared to the much higher times observed with smaller sizes like 2 and 4, **block size 12** offers the best results.

Comparing CUDA to OMP and OMP+MPI

Evaluating CUDA and Moore-UdL performance

Before comparing the concurrent performance, it is important to note the baseline differences between the two machines used in our tests: the Moore system and the CUDA-enabled machine.

SERIAL				
Matrix Size	Steps			
	100	1000	10000	100000
100x100	0,01	0,11	1,02	10,2
1000x1000	1,14	12,11	109,47	1073,03
2000x2000	4,52	47,73	476,43	4.316,25

Table 3. Moore serial values.

SERIAL (in s)				
Matrix Size	Steps			
	100	1000	10000	100000
100x100	0,0102	0,0876	0,8499	8,5743
1000x1000	1,0158	10,6156	94,5712	926,1802
2000x2000	4,0712	42,2441	422,2795	3.751,2566

Table 4. CUDA serial values.

When running the non-concurrent version of the code, we observed that the Moore machine performs slightly slower than the CUDA machine. This performance gap provides useful context for interpreting the concurrency results. On those values the serial version of the CUDA machine is about **1.14** times faster than the Moore.

Evaluation of CUDA's concurrency

Based on the previous results, a block size of 12 was identified as the optimal configuration, offering the best execution time across different grid sizes. These are the results:

CONCURRENT (Block Size: 12)				
Matrix Size	Steps			
	100	1000	10000	100000
100x100	0,0648	0,0669	0,0884	0,3047
1000x1000	0,1502	0,1714	0,3782	2,4471
2000x2000	0,4080	0,4821	1,2489	8,9450

Table 5. Parallel results of the CUDA machine (12 of block size).

This way, we can obtain the speed-up by comparing the execution times of the serial and parallel versions:

SPEED-UP				
Matrix Size	Steps			
	100	1000	10000	100000
100x100	0,16	1,31	9,61	28,14
1000x1000	6,76	61,93	250,06	378,48
2000x2000	9,98	87,63	338,12	419,37

Table 6. Speed-Up of the CUDA machine.

On the previous assignments we already analyzed the optimal amounts of threads and processes in details. These were the optimal results obtained on the previous results:

SPEED UP (OMP 4 Threads)				
Matrix Size	Steps			
	100	1000	10000	100000
100x100	1,27	2,91	3,25	3,35
1000x1000	2,61	3,57	3,68	3,70
2000x2000	2,67	3,59	3,71	3,71

Table 7. Speed Up of concurrent OMP.

SPEED UP (8 Threads and 8 Processes)				
Matrix Size	Steps			
	100	1000	10000	100000
100x100	0,34	0,50	0,95	0,96
1000x1000	4,54	2,57	10,10	10,21
2000x2000	4,97	10,70	4,52	12,17

Table 8. Speed Up of concurrent OMP and MPI.

As a result, CUDA clearly outperforms both OpenMP and the hybrid OMP+MPI approach across all matrix sizes, especially as the problem size and number of steps increase. This demonstrates the massive parallel processing power of the GPU, which scales more effectively with larger workloads. While OMP and MPI-based approaches offer some improvements over serial execution, their scalability is limited in comparison.

Our experience with CUDA

In our experience, CPUs handle general tasks well but are slower for parallel workloads like training CNN models. We don't have much experience with CUDA directly, but it's a tool that truly maximizes GPU power and greatly speeds up complex problems. In the case of this project, we have been very surprised to see how much faster CUDA is in comparison to MPI or OMP in relation to the heat diffusion problem; and this gives us an idea about the massive role that CUDA plays in very complex problems, like video rendering and AI model training.

Conclusions

Before testing the CUDA implementation, evaluating the serial performance on the new system was essential to establish a reliable baseline. The results confirmed consistent execution times across various matrix sizes and steps, providing a solid foundation for comparison.

When analyzing CUDA scalability, we tested multiple block sizes to determine the most efficient configuration. The performance improved notably up to a block size of 12, after which gains plateaued or slightly regressed.

The results clearly show that CUDA significantly outperforms both the Moore system and CPU-based concurrency approaches like OpenMP and the hybrid OMP+MPI. Even though the CUDA machine is only about 1.14 times faster than Moore in serial execution, the performance difference becomes much more pronounced when using parallel execution. With a block size of 12, CUDA achieved speed-ups of over 400x on the largest problem sizes, while OpenMP and MPI approaches plateaued at much lower improvements, typically under 12x.