PC-2019/20 Course Project: 2D Pattern Recognition, CUDA and OpenMP impementations

Alberto Baldrati
alberto.baldrati@stud.unifi.it

Abstract

In this paper, after a brief introduction to the algorithm we have used in the 2D Pattern Recognition, we're going to compare a sequential version of it, with two parallel version: the first based on CPU written in C++ using OpenMP library, the second one based on GPU written in CUDA.
The focus of this paper is not showing how the algorithm works, but rather is analyzing the performance improvements obtainable with a multithreading version compared with a sequential one.

1. Introduction

The Pattern Recognition technique consists in individuating a specific query in a data target, therefore we want to find the closest portion to our query in our target.

In our paper we focus on 2D Pattern Recognition, so our data are matrices e.g. images). After defining the pattern we want to spot in our data (the query matrix), we have to define a matching metric, i.e. a measure that shows the similarity (closeness) between queries and targets.

The metric used in this paper is the Sum of Absolute Difference (SAD) described in the following equation:

$$SAD(T, Q, i, j) = \sum_{k=0}^{r} \sum_{l=0}^{c} |T_{i+k,j+l} - Q_{k,l}| \qquad (1)$$

Where r, c are the numbers of rows and columns in the query matrix.

The Pattern Recognition algorithm will be explained in more details later, for the moment we will say it consists in computing a similarity value for each possible translation of the query matrix into the target matrix.
After that we search for the minimum value in the SAD Matrix which indicates the portion of target matrix closest to the query matrix.

In **figure 1** we can see a trivial example of our algorithm.

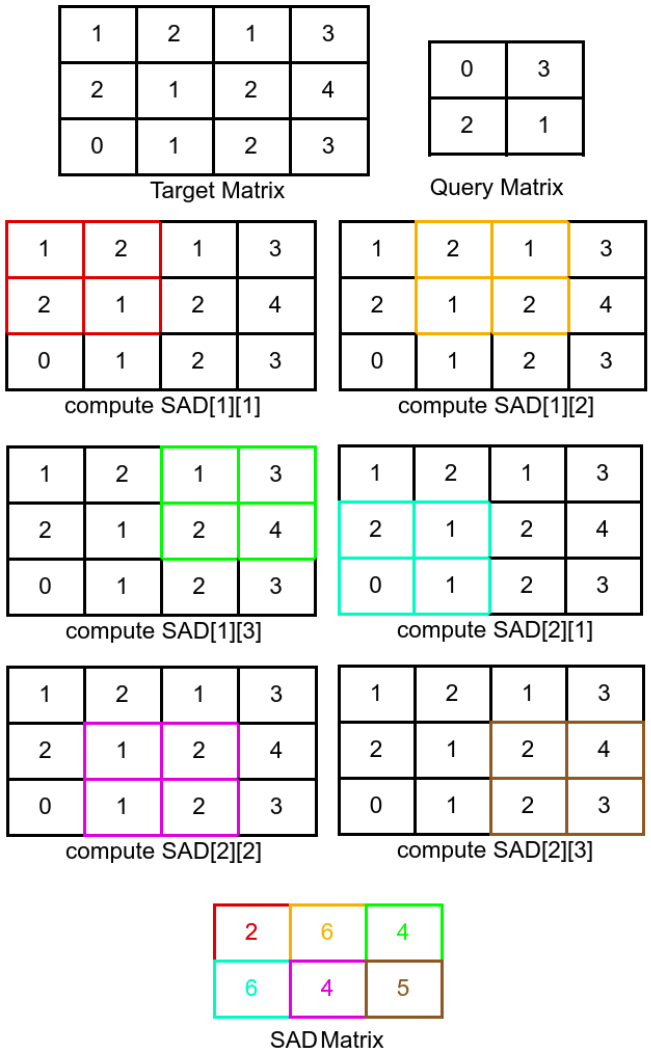

Figure 1. Trivial example of Pattern Recognition algorithm

As we said before the 2D Pattern Recognition can be applied to images (in fact they are matrices), **figure 2 on the next page** shows into the green box the area that has the best match with a random query matrix.

Figure 2. Pattern Recognition applied to an image

2. Algorithm

2.1. Sequential version

First of all we have implemented the sequential version of the algorithm briefly described above. This implementation runs on CPU and it is written in C++.
Below is reported the pseudocode, it is split in two section:

- The first, **Algorithm 1**, is fundamentally the pseudo code of **equation** (1) **on the preceding page**, this part of the algorithm will remain the same in the parallel version too.

- The second, **Algorithm 2**, is the outer cycle that iterates over rows and columns for finding the best match, this portion of the algorithm is embarrassingly parallel, so it can be easily parallelized and it is what we are going to do later in this paper.

Algorithm 1: computeSAD

```
Data: queryMatrix Q, targetMatrix T,
      startRowIndex i, startColIndex j
Result: localSadValue
localSadValue = 0
for from k = 0 to Q.rows do
    for from l = 0 to Q.cols do
        targetV = T[i+k][j+l]
        queryV = Q[k][l]
        localSadValue += | targetV - queryV |
return localSadValue
```

Algorithm 2: PatternRecognition

```
Data: queryMatrix Q, targetMatrix T
Result: SADMatrix S
Define SADMatrix S
S.rows = T.rows - Q.rows + 1
S.cols = T.cols - Q.cols + 1
for from i = 0 to S.rows do
    for from j = 0 to S.cols do
        S[i][j] = computeSAD(P,Q,i,j)
cx, cy = argmin(S)
```

2.2. Time Complexity Analysis

Our algorithm has 4 nested loops: 2 in the outer cycle that iterates over rows and columns and 2 in the computation of each SAD matrix value.
So before moving forward it's important to analyze the time complexity of the full algorithm.
For simplicity in our notation we define: **Tr** and **Tc** as target matrix rows and columns and **Qr** and **Qc** as query matrix rows and columns.

2.2.1 Sequential example

The complexity of **Algorithm 1** (SADcomplexity) is

$$Qr * Qc \tag{2}$$

The complexity of the **Algorithm 2** is:

$$(Tr - Qr + 1) * (Tc - Qc + 1) * SADcomplexity \tag{3}$$

So the full 2D Pattern Recognition algorithm has complexity:

$$(Tr - Qr + 1) * (Tc - Qc + 1) * (Qr * Qc) \tag{4}$$

2.2.2 Parallel example

As we said before the **Algorithm 2** is embarrassingly parallel, so it can be easily parallelized. If we suppose that we use a number of thread equal to **Nt**, in an ideal situation the **equation** (3) becomes:

$$\frac{(Tr - Qr + 1) * (Tc - Qc + 1)}{Nt} * SADcomplexity \tag{5}$$

And the **equation** (4) in parallel case becomes:

$$\frac{(Tr - Qr + 1) * (Tc - Qc + 1)}{Nt} * (Qr * Qc) \tag{6}$$

From **equation** (6) we can infer that in an ideal case, i.e. without overhead and with a number of core equal to **Nt**, we might achieve a linear SpeedUp up to **Nt** due to the structure of our algorithm

3. Parallel implementations

As briefly said in the introduction we have deployed two parallel version of the sequential algorithm, the first one written in C++ using the OpenMP library, the second one written in CUDA for using the parallel power of GPUs.

3.1. OpenMP

OpenMP is a library that can easily transform a sequential program into a parallel one, with the only addition of #pragma directives.

It obviously has some limitations, especially when a program must synchronize data among threads, but in our case that is not a problem. In fact the part of the algorithm we want to parallelize is embarrassingly parallel, so OpenMP suits very well for our purpose.[3]

Below is reported the pseudo code of the OpenMP implementation.

Algorithm 3: PatternRecognition *OpenMP version*

```
Data: queryMatrix Q, targetMatrix T, numThread
      Nt
Result: SADMatrix S
Define SADMatrix S
S.rows = T.rows - Q.rows + 1
S.cols = T.cols - Q.cols + 1
#pragma omp parallel for num_threads(Nt)
  collapse(2) schedule(static)
for from i = 0 to S.rows do
    for from j = 0 to S.cols do
        S[i][j] = computeSAD(P,Q,i,j)
cx, cy = argmin(S)
```

In the #pragma directives there are few points of interest:

- For our purpose we don't need dynamic schedule because the workload is pretty well balanced, in fact the number of tasks is fixed a priori. So to avoid useless overhead we use static schedule
- Nested parallelism could be useful in this case, in fact we want to parallelize over rows and columns, actually if the size of target matrix increases due to the limited number of core of CPUs the difference might not be relevant
- For testing purpose we can set the number of threads we want

3.2. CUDA

With this parallel implementation of the 2D Pattern Recognition we want to exploit the massive parallel power of GPUs.

One of the disadvantages of the CPUs over the GPUs, is that they suffer from the context switching cost when they are exposed to an high number of active threads, number too much superior to the cores count. Conversely, the GPUs, having a larger number of register files and hardware scheduler, they are not just able to manage an high quantity of threads, but this practice is actually encouraged to facilitate the memory latency hiding.
So, given a SAD Matrix M, it is an obvious consequence to start a number $|M| = n$ of threads and assign the processing of each element to them. Therefore in our case we launch a thread for each SAD value that we need to compute.

From now on, the used terms will be from the CUDA terminology [1]. In the GPU, threads are organized in *blocks*. Blocks can have different dimensions adapting the shape of data, in our case they are 2-dimensional because our data are matrices. The threads composing each block, are executed in parallel (by the **SIMT** paradigm) in group of 32 named warps. It is good practice keep the block size multiple of this number.[5]

Below is reported the pseudo code of our CUDA implementation, it is split in two part: the kernel lanuch in **Algorithm 4** and the kernel function in **Algorithm 5**

Algorithm 4: Kernel Launch

```
Data: queryMatrix Q, targetMatrix T, SADMatrix
      S, TILE_WIDTH
DimGrid(ceil(S.rows / TILE_WIDTH), ceil(s.cols /
  TILE_WIDTH));
dimBlock(TILE_WIDTH, TILE_WIDTH)
PatternRecognitionKernel<<<dimGrid,
  dimBlock>>>(Q,T,S)
```

Algorithm 5: PatternRecognitionKernel

```
Data: queryMatrix Q, targetMatrix T, SADMatrix S
bx = blockIdx.x
by = blockIdx.y
tx = threadIdx.x
ty = threadIdx.y
col = bx * blockDim.x + tx
row = by * blockDim.y + ty
if row<S.rows and col<S.cols then
    for from i = 0 to Q.rows do
        for from j = 0 to Q.cols do
            targetV = T[i+row][j+col]
            queryV = Q[i][j]
            localSadValue += | targetV - queryV |
    S[row][col] = localSadValue
```

In **Algorithm 4** we can see that our 2-dimensional blocks have dimension TILE_WIDTH× TILE_WIDTH, so in our test in **section 4** we will use TILE_WIDTH dimension equal to 8,16,24 and 32 for maintaining a number of threads in a block multiple of 32.

In **Algorithm 5** we can notice that the access to the target matrix is coalesced, in fact T[i+row][j+col] is of the form T[(expression with terms independent of tx) + tx], this means that with a single acces more requests to the VRAM will be satisfied at the same time. [1]
It's also important to notice that in the Kernel function Target Matrix and Query Matrix are not modified, so we can mark them as **const** and **restrict**. In this way the compiler is sure that both matrices are read-only data and can use a cache designed for this type of data, available for devices with compute capability greater than 3.5. [2]

3.2.1 Tiled implementation

For an extra performance boost we have tried to implement a tiled version of the 2D Pattern Recognition algorithm, which doesn't only use the global memory of the GPU but the shared memory as well. [4]

The shared memory can be seen as a programmable cache: it's a memory inside the GPU, characterized by a little dimension and very low latency. The values of this memory are logically shared between the element of a block, so it's useful when there is a massive reuse of the same values in the same block.

Unfortunately our algorithm does not suit well for these conditions and our tiled implementation performs worse than the version of the algorithm which use only the global memory.

4. Experimental Result

Several experiments have been made to compare execution time and SpeedUp of the sequential version, written in C++, with the parallel versions written in CUDA (using GPUs) and in C++ (using OpenMP).
The SpeedUp S_P is calculated as

$$S_P = \frac{t_s}{t_p} \tag{7}$$

The tests have been conducted on a Ubuntu 18.04 LTS machine equipped with:

- Intel Core i7-4790 3.6GHz with Turbo Boost up to 4Ghz, 4 core/8 thread processor
- RAM 16 GB DDR4
- NVidia GeForce 940MX 2GB (running on CUDA 10.1)

The high precision C++11 library *chrono* has been used for measuring the execution time.

Each time has been measured running each test 5 times and taking the average as a result, in this section we provide mostly graphs for a better understanding of results, in **section 6 on page 7** are available the complete numerical results. We have conducted experiments on three different combinations of query matrix and target matrix size:

- **Test1**: target matrix with dimension 1500×1500 and query matrix with dimension 150×150
- **Test2**: target matrix with dimension 2000×2000 and query matrix with dimension 200×200
- **Test3**: target matrix with dimension 2500×2500 and query matrix with dimension 250×250

4.1. OpenMP Result

The following graphs show the results of our OpenMP implementations, in each test we have used a number of threads from 1 (sequential version) to 10.

In the graphs we can identify three sections, which correspond to the three different colors:

- **Blue** section where each thread can be executed on a different **physical core**
- **Yellow** section where each thread can be executed on a different **logical core**
- **Red** section where threads has to **share** both physical and logical core

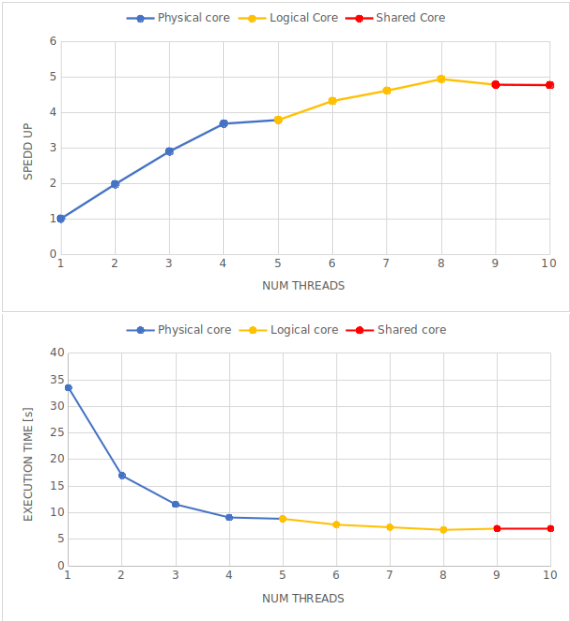

Figure 3. Test1 SpeedUp and execution time varying the number of threads using OpenMp

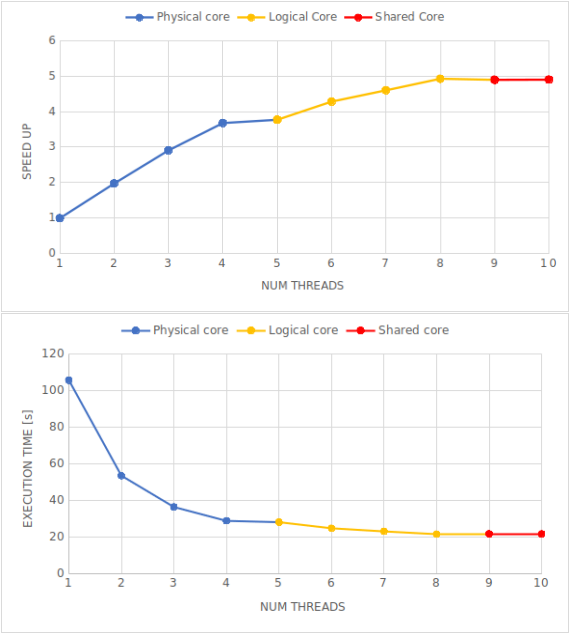

Figure 4. Test2 SpeedUp and execution time varying the number of threads using OpenMp

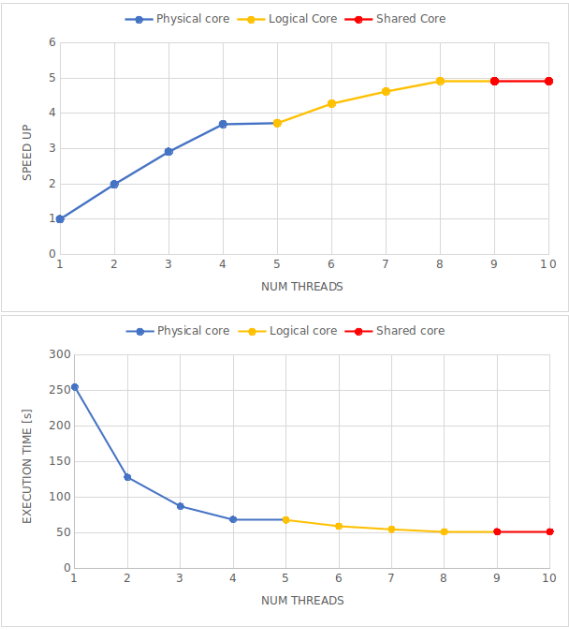

Figure 5. Test3 SpeedUp and execution time varying the number of threads using OpenMp

All three combinations of target and query matrix show very similar (almost identical) behavior, therefore we can infer that the performance of our implementation is independent from the size of the matrices.

In every SpeedUp graph we can notice that in each area we have a different SpeedUp evolution, in the blue area it is almost **linear**, in the yellow area is **sub-linear** and in the red area we have no more SpeedUp. Considering all areas together we can define our SpeedUp as *SpeedUp with an optimal number of processors*, this number in our case is 8 which is the number of logical core in the machine we used for our tests.

4.2. CUDA results

The following graphs show the results of our CUDA implementations, in each test we have used four TILE_WIDTH dimensions and three different implementation:

- The implementation which uses global memory and the optimization for pointer aliasing (blue line in the graph with the name **Global1**)
- The implementation which uses global memory but does not use the optimization for pointer aliasing (yellow line in the graph with the name **Global2**)
- The implementation which uses shared memory (orange line in he graph with the name of **Shared**)

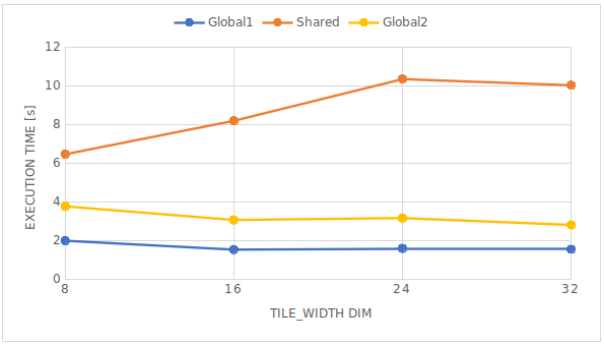

Figure 6. Test1 execution time using different implementation and TILE_WIDTH dimension

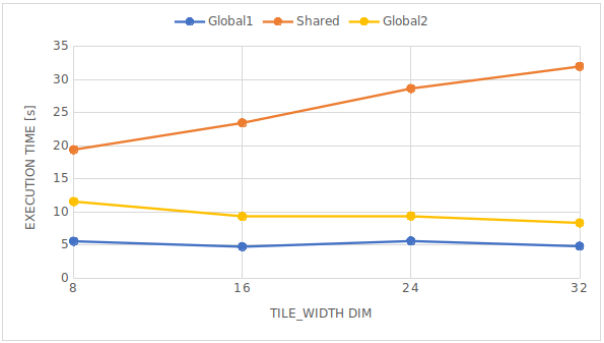

Figure 7. Test2 execution time using different implementation and TILE_WIDTH dimension

	Sequential time	*OpenMP time*	*OpenMP speedUp*	*CUDA time*	*CUDA SpeedUP*
Test1	33.52s	6.79s	**4.61x**	1.51s	**22.20x**
Test2	105.68s	21.47s	**4.92x**	4.73s	**22.34x**
Test3	254.50s	51.80s	**4.91x**	11.63s	**21.88x**

Table 1. Comparison and SpeedUp between CUDA and OpenMP version

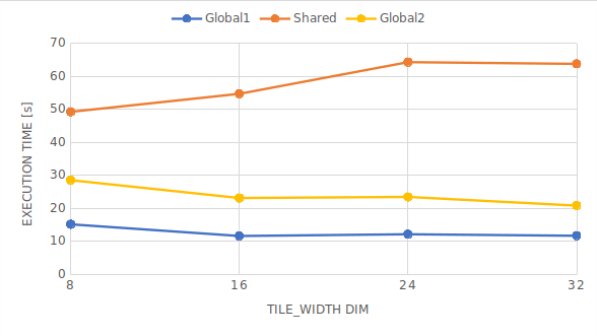

Figure 8. Test3 execution time using different implementation and TILE_WIDTH dimension

From the graphs we can confirm what we stated before, that is the shared version implementation is not good and the pointer aliasing optimization takes a good performance boost. In all three test the best version is the one which use such optimization and has TILE_WIDTH equal to 16.

4.3. OpenMP and CUDA comparison

In both CUDA and OpenMP experiments we have used the same matrices and we have measured the time in the same way, so we can compare the execution time between the different implementations. For each test we take only the best time achieved with each different implementation. Results are displayed in **table 1**.
We can see that also here the results are very similar in each Test, therefore in each implementation the algorithm is size independent.

5. Conclusions

In this paper we show how the 2D Pattern Recognition algorithm, which is computationally expensive, can be parallelized. The OpenMP library allow us to do this process in a very easy way with a satisfying SpeedUp (with our architecture we reach about **5x**). However the uses of GPUs bring another improvement, in fact the parallel implementation written in CUDA reaches a SpeedUp about **22x**.
Therefore we can state that for this embarrassingly parallel algorithm use of GPUs appears heavily faster then the traditional CPU processing

References

[1] Nvidia,cuda documentation, `https://docs.nvidia.com/cuda/`, 2019.

[2] J. Appleyard. Cuda pro tip: Optimize for pointer aliasing, `https://devblogs.nvidia.com/cuda-pro-tip-optimize-pointer-aliasing/`, 2014.

[3] M. Bertini. Openmp slides, `http://www.micc.unifi.it/bertini/download/parallel/2016-2017/8_shared_memory_openmp.pdf`, 2019.

[4] M. Harris. Using shared memory in cuda c/c++, `https://devblogs.nvidia.com/using-shared-memory-cuda-cc/`, 2013.

[5] F. Vaccaro. Cuda mean shift, `https://github.com/fede-vaccaro/CUDA_MeanShift/blob/master/meanshift_cuda.pdf`, 2019.

6. Appendix

In this section we provide all numerical results of tests.

OpenMP Test1		
Num threads	Execution time	SpeedUp
1	33,52s	1,00x
2	16,98s	1,97x
3	11,57s	2,90x
4	9,11s	3,68x
5	8,86s	3,78x
6	7,76s	4,32x
7	7,27s	4,61x
8	**6,79s**	**4,94x**
9	7,01s	4,78x
10	7,03s	4,77x

Table 2. Test1 OpenMP SpeedUp and execution time numerical results, graph displayed in figure 3 on page 4

OpenMP Test2		
Num threads	Execution time	SpeedUp
1	105,68s	1,00x
2	53,47s	1,98x
3	36,34s	2,91x
4	28,76s	3,67x
5	28,04s	3,77x
6	24,70s	4,28x
7	22,99s	4,60x
8	**21,47s**	**4,92x**
9	21,61s	4,89x
10	21,59s	4,90x

Table 3. Test2 OpenMP SpeedUp and execution time numerical results, graph displayed in figure 4 on page 5

OpenMP Test3		
Num threads	Execution time	SpeedUp
1	254,50s	1,00x
2	128,09s	1,99x
3	87,53s	2,91x
4	69,01s	3,69x
5	68,40s	3,72x
6	59,60s	4,27x
7	55,16s	4,61x
8	**51,80s**	**4,91x**
9	51,85s	4,91x
10	51,88s	4,91x

Table 4. Test3 OpenMP SpeedUp and execution time numerical results, graph displayed in figure 5 on page 5

CUDA Test 1			
TILE_WIDTH	Global1	Shared	Global2
8	1,97s	6,45s	3,76s
16	**1,51s**	8,19s	3,05s
24	1,58s	10,35s	3,15s
32	1,53s	10,03s	2,79s

Table 5. Test1 CUDA execution time numerical results, graph displayed in figure 6 on page 5

CUDA Test 2			
TILE_WIDTH	Global1	Shared	Global2
8	5,55s	19,38s	11,55s
16	**4,73s**	23,42s	9,28s
24	5,58s	28,62s	9,36s
32	4,79s	31,95s	8,32s

Table 6. Test2 CUDA execution time numerical results, graph displayed in figure 7 on page 5

CUDA Test 3			
TILE_WIDTH	Global1	Shared	Global2
8	15,17s	49,26s	28,56s
16	**11,62s**	54,71s	23,14s
24	12,15s	64,31s	23,47s
32	11,70s	63,79s	20,84s

Table 7. Test3 CUDA execution time numerical results, graph displayed in figure 8 on the preceding page