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

Alberto Baldrati
alberto.baldrati@stud.unifi.it

April 2020

Outline

Preview

- The Pattern Recognition technique consists in individuating a specific query in a data target
- We focus on 2D Pattern Recognition, our data are matrices (e.g. images)
- We have to define a matching metric, i.e. a measure that shows the similarity (closeness) between queries and targets.

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

Algorithm

1	2	1	3
2	1	2	4
0	1	2	3

Target Matrix

0	3
2	1

Query Matrix

1	2	1	3
2	1	2	4
0	1	2	3

compute SAD[1][1]

1	2	1	3
2	1	2	4
0	1	2	3

compute SAD[1][2]

1	2	1	3
2	1	2	4
0	1	2	3

compute SAD[1][3]

1	2	1	3
2	1	2	4
0	1	2	3

compute SAD[2][1]

1	2	1	3
2	1	2	4
0	1	2	3

compute SAD[2][2]

1	2	1	3
2	1	2	4
0	1	2	3

compute SAD[2][3]

2	6	4
6	4	5

SAD Matrix

Algorithm

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)
```

Time Complexity

- Our algorithm has 4 nested loops: 2 in the outer cycle (*Algorithm2*) and 2 in the computation of each SAD matrix value (*Algorithm1*).
  Time complexity in sequential implementation is

$$(Tr - Qr + 1) * (Tc - Qc + 1) * (Qr * Qc) \tag{2}$$

- **Algorithm 2** is embarrassingly parallel, so if we use a number of threads equal to **Nt**, in an ideal situation the complexity of this parallel algorithm becomes:

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

OpenMP

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)
```

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)
```

OpenMP

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)
```

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)
```

CUDA

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: PatternRecogntionKernel

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
            tV = T[i+row][j+col]
            qV = Q[i][j]
            localSadValue +=
              | tV - qV |
    S[row][col] = localSadValue
```

CUDA points of insterest

- 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]
- Target Matrix and Query Matrix are not modified, so we can mark them as **const** and **restrict**, 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.
- Tiling is useless due to the low reuse of same values within the same block

Equipment, metrics and profiling

- The tests have been conducted on an 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 750 2GB compute-capability 5.0 (running on CUDA 10.1)
- The metrics used are execution time and SpeedUp S_P, which is calculated as

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

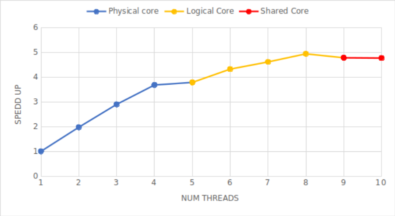
- The high precision C++11 library *chrono* has been used for measuring the execution time.

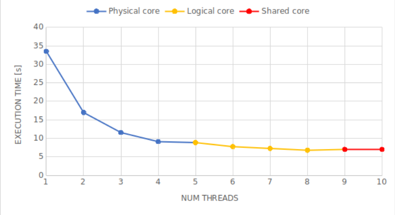
Tests

- Each time has been measured running each test 5 times and taking the average as a result
- 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

OpenMP Test1

Test1: target matrix with dimension 1500×1500 and query matrix with dimension 150×150

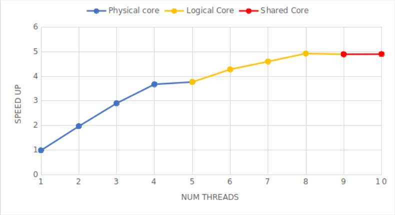


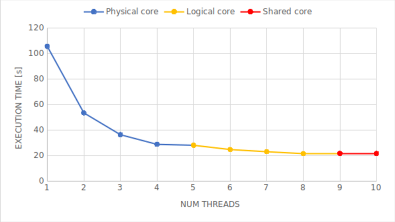


OpenMP Test1		
Num threads	Execution time	SpeedUp
1	33,52s	1,00x
2	16,98s	1,97x
3	11,57s	2,90x
4	9,11s	3,68x
5	8,86s	3,78x
6	7,76s	4,32x
7	7,27s	4,61x
8	**6,79s**	**4,94x**
9	7,01s	4,78x
10	7,03s	4,77x

OpenMP Test2

Test2: target matrix with dimension 2000×2000 and query matrix with dimension 200×200

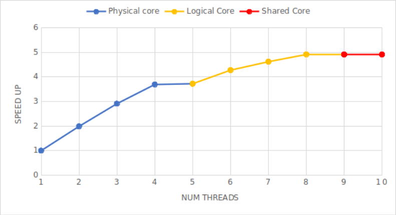


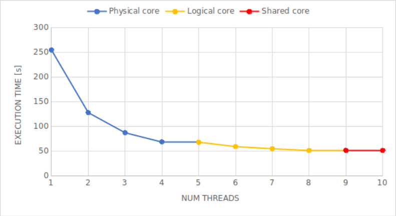


OpenMP Test2		
Num threads	Execution time	SpeedUp
1	105,68s	1,00x
2	53,47s	1,98x
3	36,34s	2,91x
4	28,76s	3,67x
5	28,04s	3,77x
6	24,70s	4,28x
7	22,99s	4,60x
8	**21,47s**	**4,92x**
9	21,61s	4,89x
10	21,59s	4,90x

OpenMP Test3

Test3: target matrix with dimension 2500×2500 and query matrix with dimension 250×250





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

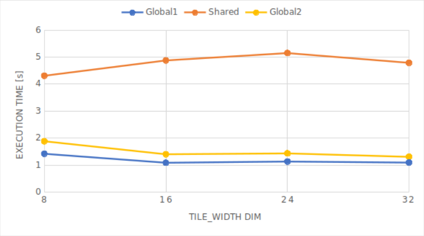
CUDA results

In the following slides the results of our CUDA implementations are shown, in each test we have used four TILE_WIDTH dimensions and three different implementations:

- 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 the graph with the name of **Shared**)

CUDA Test1

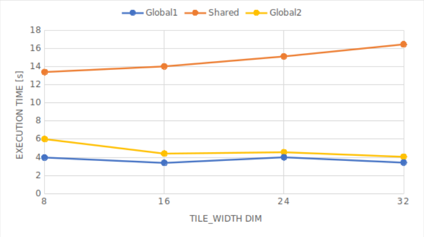
Test1: target matrix with dimension 1500×1500 and query matrix with dimension 150×150



CUDA Test1			
TILE_WIDTH	Global1	Shared	Global2
8	1,42s	4,31s	1,88s
16	**1,08s**	4,88s	1,40s
24	1,13s	5,15s	1,43s
32	1,10s	4,79s	1,31s

CUDA Test2

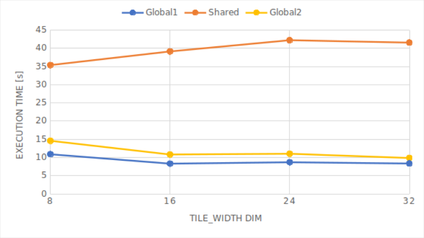
Test2: target matrix with dimension 2000×2000 and query matrix with dimension 200×200



CUDA Test2			
TILE_WIDTH	Global1	Shared	Global2
8	3,97s	13,40s	6,01s
16	**3,39s**	14,02s	4,41s
24	4,00s	15,12s	4,56s
32	3,42s	16,45s	4,05s

CUDA Test3

Test3: target matrix with dimension 2500×2500 and query matrix with dimension 250×250



CUDA Test3			
TILE_WIDTH	Global1	Shared	Global2
8	10,90s	35,40s	14,59s
16	**8,31s**	39,15s	10,82s
24	8,69s	42,23s	11,03s
32	8,34s	41,56s	9,86s

OpenMP and CUDA results comparison

- Both CUDA and OpenMP experiments use the same matrices, so the comparison between implementations is fair
- For each test has been taken the best time achieved with each different implementation.

	Sequential time	*OpenMP time*	*OpenMP speedUp*	*CUDA time*	*CUDA SpeedUP*
Test1	33,52s	6,79s	**4,94x**	1,09s	**30,75x**
Test2	105,68s	21,47s	**4,92x**	3,39s	**31,17x**
Test3	254,50s	51,80s	**4,91x**	8,31s	**30,62x**

Conclusions

- 2D Pattern Recognition algorithm achieves a powerful boost if executed in a parallel implementation
- In our tests OpenMP implementations reach a **5x** SpeedUP and the GPU version based on CUDA reach a **31x** SpeedUP
- In embarrassingly parallel algorithm the use of GPUs outperforms the traditional CPU processing