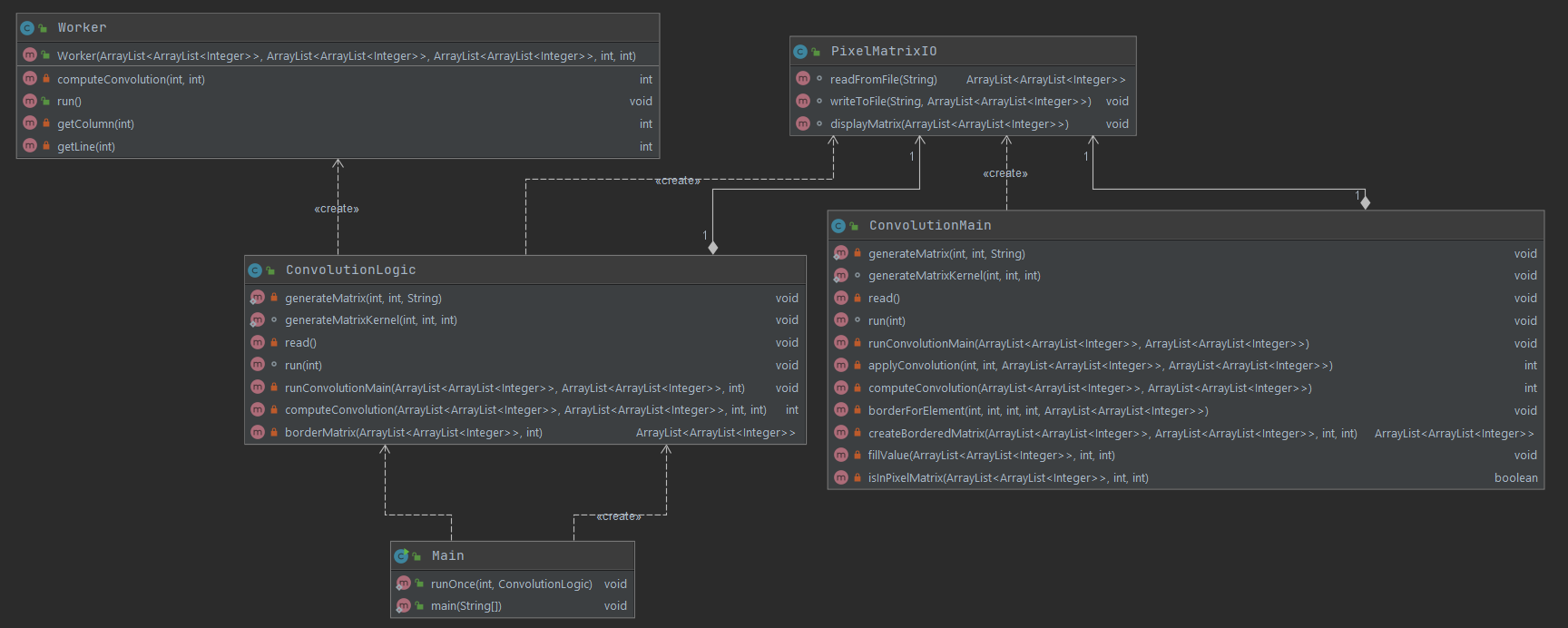
Laborator 1 PDD – Salsigan Razvan-Dan

## Cerinta

Se da o matrice de pixeli si un kernel, se cere sa se aplice o convolutie cu acel kernel pe matricea intiala si sa se calculeze timpul de executie, folosind un numar diferit de threaduri.

## Arhitectura



## Impartirea elementelor

Elementele au fost distribuite threadurilor linear, pe chunkuri de elemente consecutive, fiecare thread executand operatia pe numar\_elemente/numar\_threaduri [+1] pixeli. (Threadul primeste un start si un end, corespunzand indicilor din matricea liniarizata)

Determin cate elemente am (n\*m), aflu numarul de chunkuri impartind numarul de elemente la numarul de threaduri si distribui restul, cate un element pe fiecare thread pana il epuizez.

La fiecare iteratie, finalul ia valoarea start\_precedent+chunk\_size+[1]

int elements = pixelMatrixNSize \* pixelMatrixMSize;

int start = 0, end = 0;

int chunk = elements / noOfThreads;

int remainder = elements % noOfThreads;

for (int i = 0; i < noOfThreads; i++) {

end = start + chunk + (remainder-- > 0);

t.emplace\_back([this, start, end] {doWork(start, end); });

start = end;

}

## Padding

Pentru padding am o metoda care primeste matricea initiala si dimensiunea marginii (@border = 1 pentru kernel de 3x3, 2 pentru kernel de 5x5 etc) si returneaza o matrice bordata corespunzator.

Parcurg pe rand primele @border randuri si le completez cu elementele de pe prima linie din matricea initiala (si cele de pe colt in colt – primele/ultimele @border elemente de pe primele/ultimele @border linii vor primi valoarea din coltul matricii initiale), analog pentru ultimele @border randuri cu elementele de pe ultima linie din matricea initiala, dupa care fac acelasi procedeu pentru bordarea pe coloane.

int lines = pixelMatrixNSize;

int columns = pixelMatrixMSize;

//top and bottom rows

for (int i = 0; i < border; i++) {

//first @border elements (corners)

int col = 0;

//colturile din partea stanga

for (int j = 0; j < border; j++) {

borderedMatrix[i][col] = pixelMatrix[0][0];

borderedMatrix[i + border + lines][col] = pixelMatrix[lines - 1][0];

col++;

}

//copiez elementul de dedesubt

for (int j = 0; j < columns; j++) {

borderedMatrix[i][col] = pixelMatrix[0][j];

borderedMatrix[i + border + lines][col] = pixelMatrix[lines - 1][j];

col++;

}

//last @border elements (corners)

//colturile din partea dreapta

for (int j = 0; j < border; j++) {

borderedMatrix[i][col] = pixelMatrix[0][columns - 1];

borderedMatrix[i + border + lines][col] = pixelMatrix[lines - 1][columns - 1];

col++;

}

}

//the rest of the matrix

for (int i = 0; i < lines; i++) {

int col = 0;

for (int j = 0; j < border; j++) {

borderedMatrix[i + border][col] = pixelMatrix[i][0];

col++;

}

for (int j = 0; j < columns; j++) {

borderedMatrix[i + border][col] = pixelMatrix[i][j];

col++;

}

for (int j = 0; j < border; j++) {

borderedMatrix[i + border][col] = pixelMatrix[i][columns - 1];

col++;

}

}

## Analiza timpilor de executie

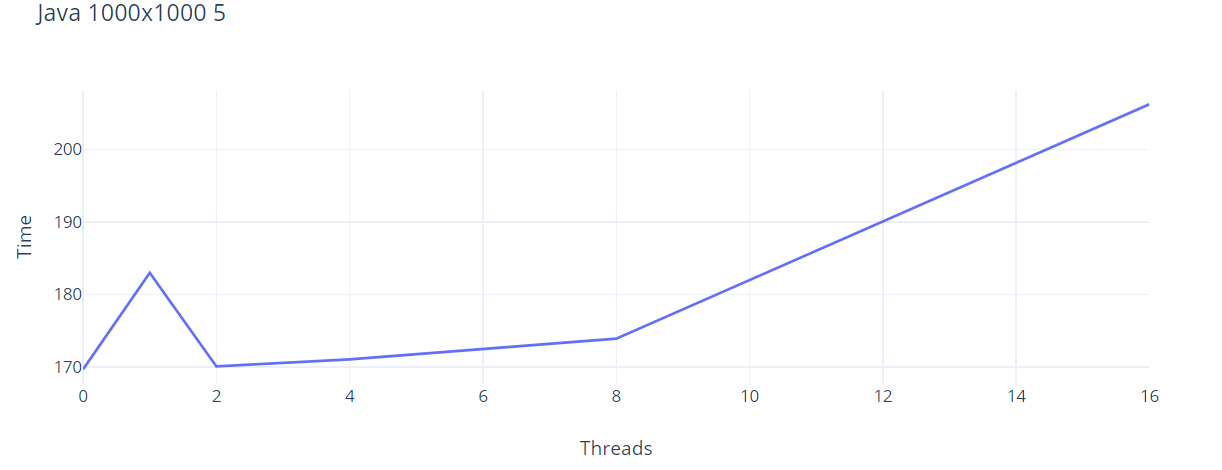
Testele au fost efectuate pe un procesor i5-9300H (4C, 8T) , in Windows

### Java

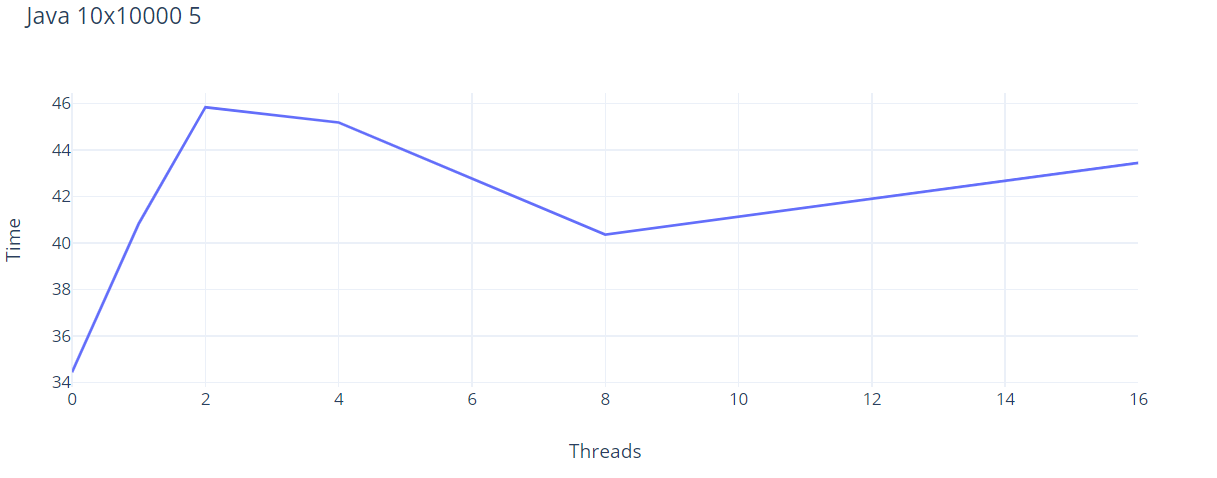
|  |  |  |
| --- | --- | --- |
| Tip matrice | Nr threads | Timp executie |
| N=M=10  n=m=3 | secvential | 0.49793 |
| 1 | 0.79112 |
| 4 | 0.9697 |
| N=M=1000  n=m=5 | secvential | 169.7087 |
| 1 | 183.0019 |
| 2 | 170.0949 |
| 4 | 171.0596 |
| 8 | 173.9188 |
| 16 | 206.2419 |
| N=10 M=10000  n=m=5 | secvential | 34.4335 |
| 1 | 40.82756 |
| 2 | 45.83078 |
| 4 | 45.17749 |
| 8 | 40.35461 |
| 16 | 43.4442 |
| N=10000 M=10  n=m=5 | secvential | 41.59292 |
| 1 | 44.85427 |
| 2 | 51.44558 |
| 4 | 45.83344 |
| 8 | 43.31275 |
| 16 | 48.7761 |

#### Grafice

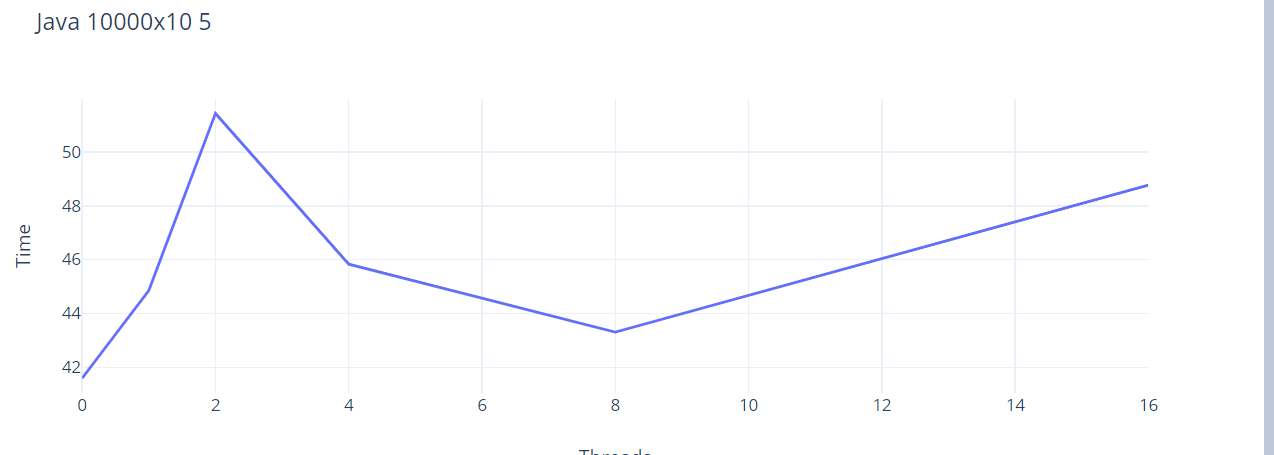
1. N=M=1000 n=m=5



1. N=10 M=10000 n=m=5



1. N=10000 M=10 n=m=5

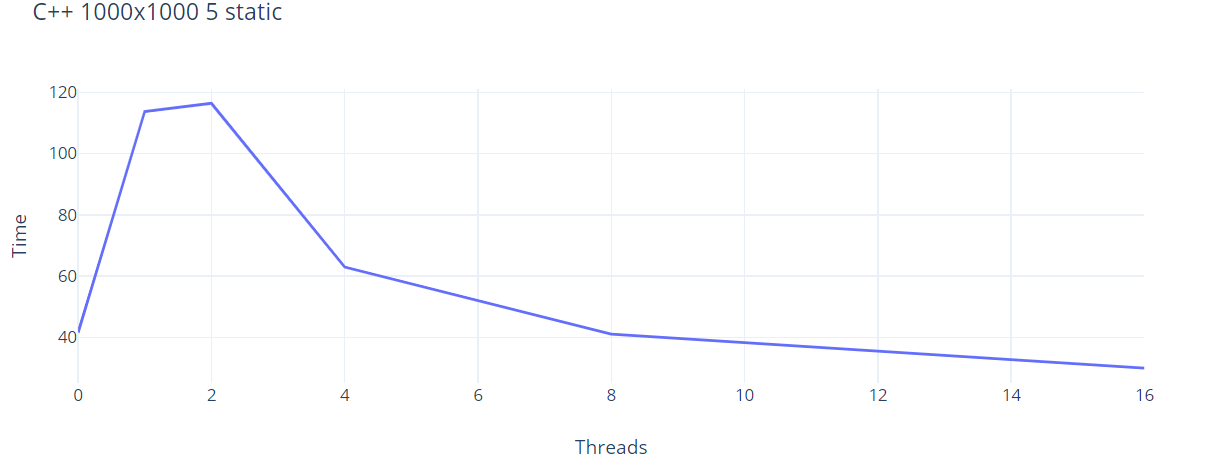
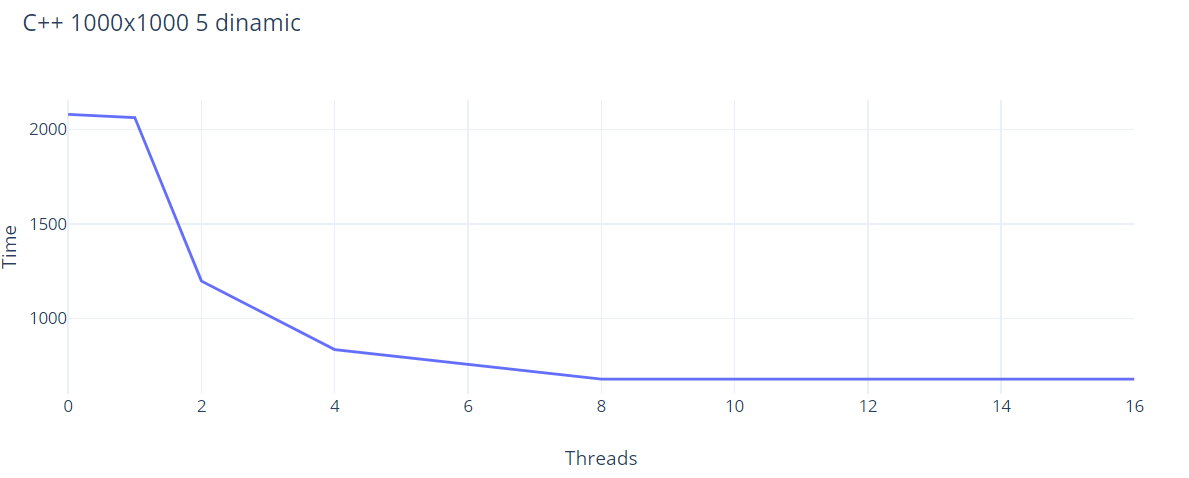


### C++

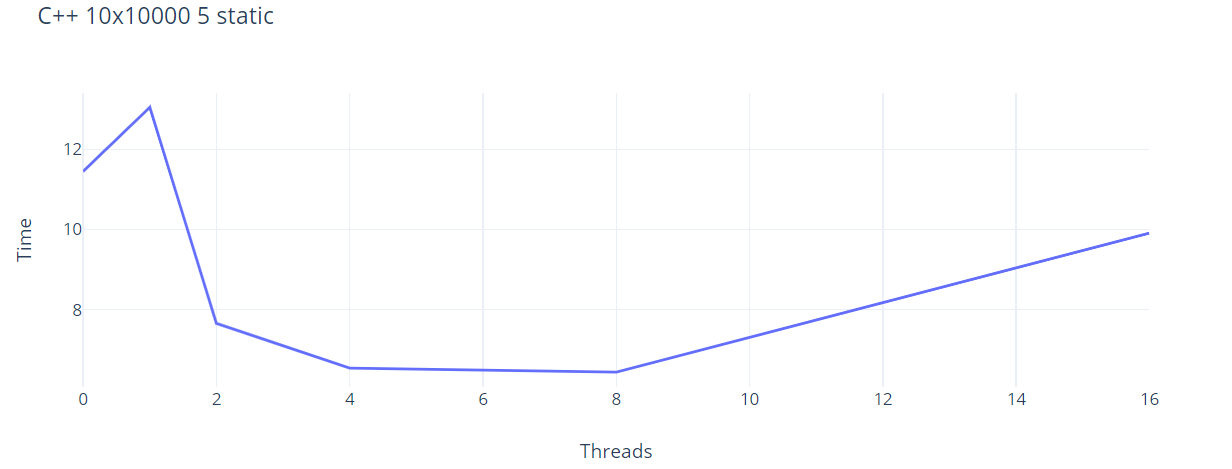
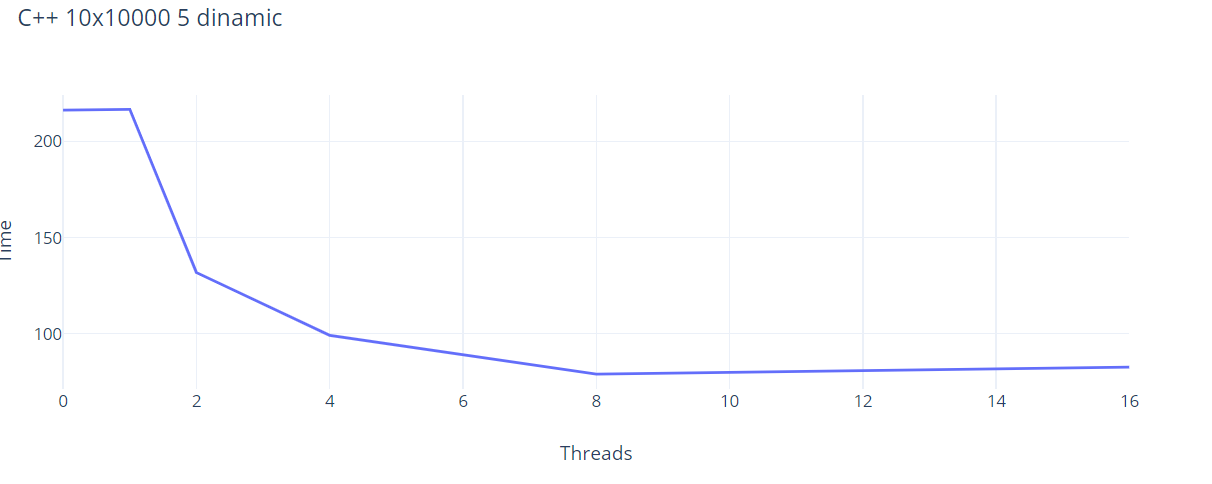
|  |  |  |  |
| --- | --- | --- | --- |
| Tip matrice | Tip alocare | Nr threads | Timp executie |
| N=M=10  n=m=3 | Static | secvential | 0.00989 |
| 1 | 1.74461 |
| 4 | 2.59428 |
| Dinamic | secvential | 0.25919 |
| 1 | 1.72215 |
| 4 | 2.3594 |
| N=M=1000  n=m=5 | Static | secvential | 113.7524 |
| 1 | 116.4724 |
| 2 | 62.97333 |
| 4 | 41.0734 |
| 8 | 29.91659 |
| 16 | 32.41549 |
| Dinamic | secvential | 2080.428 |
| 1 | 2062.734 |
| 2 | 1198.878 |
| 4 | 836.5035 |
| 8 | 679.5502 |
| 16 | 679.9415 |
| N=10 M=10000  n=m=5 | Static | secvential | 11.44737 |
| 1 | 13.05209 |
| 2 | 7.66016 |
| 4 | 6.54454 |
| 8 | 6.44114 |
| 16 | 9.91049 |
| Dinamic | secvential | 216.4554 |
| 1 | 216.8595 |
| 2 | 131.8933 |
| 4 | 99.21389 |
| 8 | 78.94706 |
| 16 | 82.67439 |
| N=10000 M=10  n=m=5 | Static | secvential | 14.06365 |
| 1 | 14.83604 |
| 2 | 9.03779 |
| 4 | 7.73879 |
| 8 | 7.54975 |
| 16 | 10.85079 |
| Dinamic | secvential | 333.9892 |
| 1 | 338.8771 |
| 2 | 251.4098 |
| 4 | 220.1405 |
| 8 | 208.4409 |
| 16 | 202.8371 |

#### Grafice

1. N=M=1000 n=m=5



1. N=10 M=10000 n=m=5



1. N=10000 M=10 n=m=5

