Open MP

Libraria folosita pentru OpenMP este : #include <omp.h>

1. Rularea unui program din comanda : g++ -fopenmp main.cpp –o testMain

2. Setarea numarului de threaduri :

omp_set_num_threads( );

(ex: omp_set_num_threads(8)
-> setarea a 8 threaduri);

3. Pentru a identifica threadul curent se foloseste:

omp_get_thread_num( );

4. Pentru a identifica cate threaduri avem in sectiunea paralela se foloseste:

omp_get_num_threads ( );

5. Aflarea numarului maxim de threaduri:

omp_get_max_threads( )

[returneaza cate threaduriare procesorul] ;

6. **#pragma omp parallel for** -> directiva de preprocesare. Va paraleliza primul for de care da. Acesta va imparti for-ul in functie de cate threaduri are procesorul, fara sa specificam cum il impartim.

7. **#pragma omp parallel { }** -> directiva de preprocesare. Va paraleliza sectiunea de care este intre acolade. Aici trebuie sa definim index-ul de start si index-ul de stop.

- index-ul de start = omp_get_thread_num() *size_of_matrix/ omp_get_num_threads();
- index-ul de stop = (omp_get_thread_num() + 1) *size_of_matrix/ omp_get_num_threads();

8. **#pragma omp atomic ->** se foloseste pentru sincronizarea threadurilor.

9. **#pragma omp parallel for reduction ({operator } : { varibila }). ->** primul for care este in aceasta structura va fi paralelizat, omp-ul avand grija sa sincronizeze variabila care a fost specificata in directiva in functie de ce operator are.

10. **Profiling** : calculam timpul cat dureaza pana se calculeaza sectiunea intre cea de start si cea de stop.

— se foloseste clock_t start = new clock();
— se foloseste clock_t stop = new clock();
— different ne ofera timpul exact in secunde, numai ca ne trebuie cast explicit catre double

double (stop - start) / CLOCKS_PER_SEC

1. Lansarea in executie a mai multor fire de executie utilizand OpenMP

Crearea firelor de executie se face printr-o directiva de preprocesare specifica OpenMp,

```
#pragma omp parallel
```

Sectiunea de cod ce se afla dupa aceasta linie va fi executata pe mai multe thread-uri. Initial numarul de thread-uri este egal cu num?rul de nuclee a procesorului. Specificarea în mod explicit a numarului de fire de executie se face cu functia ***omp_set_num_threads*()**. Pentru a obtine identificatorul firului de executie curent se utilizeaza functia ***omp_get_thread_num*()**.

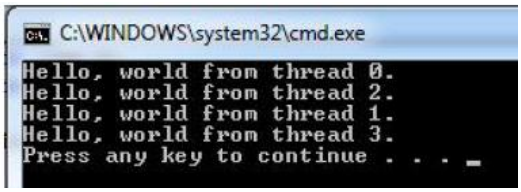
Exemplul urmator lanseaza patru fire de executie si afisaza identificatorul fiecaruia:

```
#include <omp.h>
#include <stdio.h>
int main(void)
{
	//se specifica numarul de fire de executie
	omp_set_num_threads(4);

	//directiva de preprocesare OpenMp
	#pragma omp parallel
		{
			//returneaza numarul(identificatorul) firului de executie curent
			int thread_id = omp_get_thread_num();

			//cod ce va fi executat pe fiecare fir de executie
			printf("Hello, world from thread %d.\n",id);
		}

	return 0;
}
```



2. Paralelizarea unei structure repetitive FOR

Paralelizarea unei structuri repetitive se face prin împartirea numarului de iteratii pe mai multe fire de executie.

Exemplul 1:Paralelizarea unei structuri repetitive ce atribuie valori unui tablou de elemente de tip întreg. Fiecare element al tabloului primește valoarea identificatorului firului de execuție curent.

```
#include <omp.h>
#include <iostream>

int main(void)
	{
		//numarul de elemente
		const int N = 10;

		//vector cu N elemente alocat static
		int i, a[N];

		#pragma omp parallel for
		for (i = 0; i < N; i++)
			{
				//se atribuie fiecarui element din vector identificatorul 	firului de executie curent
				a[i] = omp_get_thread_num();
			}

		//se afiseaza vectorul
		for (i = 0; i < N; i++)
			{
				Printf(“%d”, a[i]);
			}
	return 0;
}
```

Exemplul 2: Efectuarea operatiei de adunare a doi vectori pe CPU utilizând mai multe fire de executie.

```
#include <omp.h>
#include <iostream>

int main(void)
	{

		//num□rul de elemente
		const int N = 10;
		int i, a[N],b[N],c[N];

		//se initializeaz□ cei doi vectori
		for( int i = 0; i < N; i++ )
			{
				a[i] = 1;
				b[i] = 2;
			}

		#pragma omp parallel for
		for (i = 0; i < N; i++)
			{
				c[i] = a[i] + b[i];
			}

		//se afiseaza vectorul
		for (i = 0; i < N; i++)
			{
				printf("%d", c[i]);
			}

		return 0;
	}
```

3. Sincronizarea firelor de executie

-Este folosita pentru a controla ordinea in care fire de executie diferite acceseaza o resursa comuna
-Sunt : sectiuni critice / operatii atomice / bariera de sincronizare

Excludere mutuala :

* Sectiunea critica – un singur fir de execute poate intra intr-o sectiune critica la un momentdat.
#pragma omp critical

Exemplu :

```
int rezultat;
#pragma omp parallel
        {
                int id = omp_get_thread_num();
                int rezultat_partial = calculeaza_partial(id);

                #pragma omp critical
                        {
                                rezultat =calculeaza(rezultat_partial);
                        }
        }
```

* Operație atomică –similar cu o secțiune critică doar că e valabilă pentru scrierea unei singure locații de memorie (*x [operatorbinar]=[expresie]*)

#pragma omp atomic

Exemplu – estimarea lui pi

```
int rezultat;
#pragma omp parallel
        {
                int id = omp_get_thread_num();
                int rezultat_partial = calculeaza_partial(id);
                #pragma omp atomic
                        rezultat += rezultat_partial;
        }
```

Directiva barrier

- Sincronizeaza toate firele de executie
- Bariera de sincronizare trebuie intalnita de toate firele de executie sa u de nici unul
-

#pragma omp barrier

4. **Structuri repetitive paralele**

Cod secvențial:

```
for(int i = 0; < N; i++){do_stuff(i)};
```

Cod paralel (varianta 1):

```
#pragma omp parallel
{
        id = omp_get_thread_num();
        Nthreads = omp_get_num_threads();
        istart = id*N/Nthreads;
        iend(id+1)*N/Nthreads;
        for( int i = istart; i < iend; i++ )
        {
                do_stuff(i);
        }
}
```

Cod paralel (varianta 2) :

```
#pragma omp parallel for
for(int i = 0; < N; i++){do_stuff(i)};
```

5. **Reducere paralela**

```
double sum = 0.0; A[N];
```

```
for (int i = 0; i < N; i++)
        sum += A[i];
```

• Iterațiile nu sunt independente. Fiecare fir de execuție modifică variabila *sum*
• Soluția: reducere paralelă . Fiecare fir de execuție calculează o sumă parțială.
* Clauza OpenMP pentru reducere paralelă **reduction([operator, listă variabile]***)*

• Exemplu:

```
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < num_steps; i++)
        {
                sum += A[i];
        }
```

• Problema valorilor inițiale. OpenMP inițializează variabilele în funcție de operatorul specificat eg. „0” pentru „+” sau „1” pentru „*”

Tema 1 – OpenMp

1. Implementați un program pentru înmulțirea unei matrici pătratice A cu un vector $\mathbf{b}$:

$$x_i = \sum_{j=0}^{N} A(i,j)b_j$$

2. Paralelizați programul de la punctul 1 folosind librăria OpenMp. Utilizați directiva *omp parallel* astfel încât fiecare fir de execuție să calculeze o porțiune a vectorului rezultat x.
3. Măsurați timpul de execuție pentru varianta secvențială și separat pentru cea paralelă.

Indicatii:

- Matricea are dimensiunea 1000x1000
- Vectorul are dimensiunea 1000x1
- Atât matricea A cât și vectorul $\mathbf{b}$ trebuie inițializate.

```
1.  #include <stdio.h>
2.  #include <omp.h>
3.  #include <time.h>
4.
5.  // Se defineste numarul de linii si coloane ale matricei patratice A
6.  #define N 8 //1000
7.  #define NUMBER_OF_USED_THREADS 4
8.
9.  int main()
10. {
11.     // Se aloca memorie pentru matrice, vector si rezultat.
12.     double *A = new double[N * N]; // Matricea A
13.     double *B = new double[N];     // Vectorul B
14.     double *C1 = new double[N];    // Rezultatul C  secvential
15.     double *C2 = new double[N];    // Rezultatul C  parallel
16.
17.
```

```
18.     // Variabile utilizate pentru a masura timpul.
19.     clock_t timerStart, timerStop;
20.     double cpu_time_used;
21.
22.     // Initializarea matricei A cu valoarea 2.
23.     for (int i = 0; i < N; i++)
24.     {
25.         for (int j = 0; j < N; j++)
26.         {
27.             A[i, j] = 2.0;
28.         }
29.     }
30.
31.     // Initializarea vectorului B, C1 si C2 cu valoarea 1 respectiv 0.
32.     for (int i = 0; i < N; i++)
33.     {
34.         B[i] = 1;
35.         C1[i] = 0;
36.         C2[i] = 0;
37.     }
38.
39.     // Varianta secventiala ----------------------------------------------------------------------------------
    --------------------------------
40.     printf("\n Varianta secventiala ! ");
41.     // Se porneste contorizarea timpului.
42.     timerStart = clock();
43.
44.     for (int i = 0; i < N; i++)
45.     {
46.         for (int j = 0; j < N; j++)
47.         {
48.             C1[i] = A[i, j] * B[j] + C1[i];
49.         }
50.     }
51.
52.     // Se opreste contorizarea.
53.     timerStop = clock();
54.     cpu_time_used = ((double)(timerStop - timerStart)) / CLOCKS_PER_SEC;
55.
56.     printf("\n Rezultatul inmultirii varianta secventiala : ");
57.     for (int k = 0; k < N; k++)
58.     {
59.         printf("%.2lf  ", C1[k]);
60.     }
61.
62.     printf("\n CPU time used for secvential method: %.5lf \n", cpu_time_used);
63.
64.     // Varianta paralela ----------------------------------------------------------------------------------
    ----------------------------
65.     printf("\n Varianta paralela ! ");
66.
67.     // Se porneste contorizarea timpului.
68.     timerStart = clock();
69.
70.     // Se afiseaza numarul maxim de threaduri disponibile.
71.     printf("\n Numarul maxim de threaduri disponibile este %d\n", omp_get_max_threads);
72.
73.     // Se seteaza numarul de threaduri ce urmeaza a fi utilizate. Ini□ial num□rul de thread-
    uri este egal cu numarul de nuclee a procesorului.
74.     // Executia va fi impartita in NUMBER_OF_USED_THREADS parti: 0-250 / 250 - 500 ... sau ceva de genu.
75.     omp_set_num_threads(NUMBER_OF_USED_THREADS);
```

```
76.
77.  #pragma omp parallel
78.      {
79.          // Seidentifica ID threadului curent.
80.          int threadId = omp_get_thread_num();
81.
82.          // Se determina numarul de threaduri utilizate.
83.          int numberOfThreads = omp_get_num_threads();
84.
85.          // Se calculeaza indexul de start.
86.          int indexStart = threadId * N / numberOfThreads;
87.
88.          // Se calculeaza indexul de sfarsit.
89.          int indexEnd = (threadId + 1) * N / numberOfThreads;
90.
91.          for (int i = indexStart; i < indexEnd; i++)
92.          {
93.              for (int j = 0; j < N; j++)
94.              {
95.                  C2[i] = A[i, j] * B[j] + C2[i];
96.              }
97.          }
98.
99.          printf("\n Partea %d,%d a fost finalizata !", indexStart, indexEnd);
100.     }
101.
102.     timerStop = clock();
103.     cpu_time_used = ((double)(timerStop - timerStart)) / CLOCKS_PER_SEC;
104.
105.     printf("\n Rezultatul inmultirii varianta paralela : ");
106.     for (int k = 0; k < N; k++)
107.     {
108.         printf("%.2lf  ", C2[k]);
109.     }
110.
111.     printf("\n CPU time used for parrallel method: %.5lf \n", cpu_time_used);
112. }
```

Tema 2 – OpenMp: Estimarea lui π cu metoda Monte-Carlo

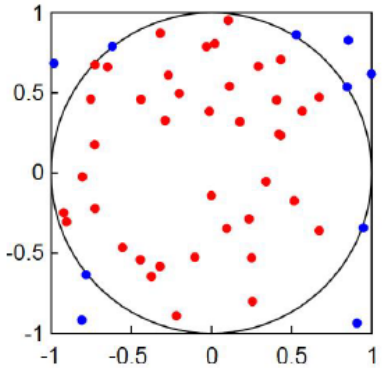
1. Implementați un program pentru estimarea valorii aproximative a lui π folosind metoda Monte-Carlo.
2. Paralelizați programul de la punctul 1 împărțind manual numărul de iterații pe mai multe fire de execuție.
3. Paralelizați programul de la punctul 1 folosind reducție paralelă. Folosiți directiva *omp parallel for reduction*.

Indicație: pentru generarea unei valori aleatoare în într-un interval (a, b) se poate folosii funcția *rand()* în felul următor:

$$x = a + (b - a) \cdot \frac{rand()}{RAND\_MAX}$$

Algoritmul de estimare a lui π prin metoda Monte-Carlo:

1. Se generează aleator N puncte într-un pătrat cu latura 2 și centrul în origine.
2. Se numără toate punctele ce se află în interiorul cercului cu raza $r = 1$ și centrul în origine.
3. Se aproximează valoarea lui π cu $4\frac{N_{cerc}}{N_{total}}$, unde N_{cerc} este numărul de puncte ce se află în interiorul cercului, calculate la punctul 2, iar N_{total} este numărul total de puncte.



```
#include <omp.h>
#include <stdio.h>
#include <iostream>
#include <time.h>
#include <math.h>

#define NUMBER_OF_POINTS 10000
//  Nu merge doar cu OMP PARALLEL FOR deoarece apar conflicte atunci cand dorim sa accesam in aceeasi variabila. Trebuie impartit pe bucati.
double generateRandNumbers(double a, double b);
double monteCarloSecventialFunction(int numberOfPoints, double a, double b, double radius);
double monteCarloParallelFunction(int numberOfPoints, double a, double b, double radius);
double monteCarloParallelForReductionFunction(int numberOfPoints, double a, double b, double radius);

int main()
{

    clock_t timerStart, timerStop;
    double timeElapsed;

    double piValue;

    // Varianta secventiala !
    printf("\n Monte-Carlo secvential function !");
    timerStart = clock();
    piValue = monteCarloSecventialFunction(NUMBER_OF_POINTS, -1.0, 1.0, 1.0);
    timerStop = clock();
    timeElapsed = (double)(timerStop - timerStart) / CLOCKS_PER_SEC;
    printf("\n Valoare PI : %.5f", piValue);
    printf("\n Time elapsed : %.3f \n", timeElapsed);

    // Varianta parallel !
    printf("\n Monte-Carlo PARALLEL function !");
    timerStart = clock();
    piValue = monteCarloParallelFunction(NUMBER_OF_POINTS, -1.0, 1.0, 1.0);
    timerStop = clock();
    timeElapsed = (double)(timerStop - timerStart) / CLOCKS_PER_SEC;
    printf("\n Valoare PI : %.5f", piValue);
    printf("\n Time elapsed : %.3f \n", timeElapsed);

    // Varianta parallel for reduction !
    printf("\n Monte-Carlo PARALLEL FOR REDUCTION function !");
    timerStart = clock();
    piValue = monteCarloParallelForReductionFunction(NUMBER_OF_POINTS, -1.0, 1.0, 1.0);
    timerStop = clock();
    timeElapsed = (double)(timerStop - timerStart) / CLOCKS_PER_SEC;
    printf("\n Valoare PI : %.5f", piValue);
    printf("\n Time elapsed : %.3f \n", timeElapsed);
}

double generateRandNumbers(double a, double b)
{
    // Generate a rundem number between the two limites a and b.
    double random = (b - a) * ((double)rand()) / RAND_MAX;

    return a + random;
}

double monteCarloSecventialFunction(int numberOfPoints, double a, double b, double radius)
{
    // Monte-Carlo function for calculate PI value secvential method.
```

```
61.
62.     int pointsInsideCircle = 0;
63.
64.     // Generate a number of points between the two limites and calculate the distance between the
    center of the cicle and the point.
65.     // If the distance is less then the radius the pointsInsideCircle is incremented
66.     for (int i = 0; i < numberOfPoints; i++)
67.     {
68.         float x = generateRandNumbers(a, b);
69.         float y = generateRandNumbers(a, b);
70.
71.         // Se calculeaza distanta de la punctul generat la centrul cercului din interiorul patratu
    lui.
72.         double distanceToCenter = sqrt(double(pow(x, 2) + pow(y, 2)));
73.
74.         if (distanceToCenter < radius)
75.         {
76.             pointsInsideCircle++;
77.         }
78.     }
79.
80.     // The formula to calculate the PI number
81.     return 4 * ((1.0 * pointsInsideCircle) / (numberOfPoints * 1.0));
82. }
83.
84. double monteCarloParallelFunction(int numberOfPoints, double a, double b, double radius)
85. {
86.     // Monte-Carlo function for calculate PI value PARALLEL method.
87.
88.     // We need to use omp parallel insted because we need to sync the threads.
89.     // We acces the same memmory and these is a critical space
90.
91.     int pointsInsideCircle = 0;
92.
93.     // Generate a number of points between the two limites and calculate the distance between the
    center of the circle and the point.
94.     // If the distance is less then the radius the pointsInsideCircle is incremented
95.     #pragma omp parallel
96.     {
97.         int threadId = omp_get_thread_num();
98.         int numberOfThreads = omp_get_num_threads();
99.
100.        // We calculate the start index and the stop index to calculate on chuncks the PI value.
101.        int startIndex = threadId * numberOfPoints / numberOfThreads;
102.        int stopIndex = (threadId + 1) * numberOfPoints / numberOfThreads;
103.
104.        for (int i = startIndex; i < stopIndex; i++)
105.        {
106.
107.            float x = generateRandNumbers(a, b);
108.            float y = generateRandNumbers(a, b);
109.
110.            // Se calculeaza distanta de la punctul generat la centrul cercului din interiorul pat
    ratului.
111.            double distanceToCenter = sqrt(double(pow(x, 2) + pow(y, 2)));
112.
113.            if (distanceToCenter < radius)
114.            {
115.                // For sync we use omp atomic
116.                #pragma omp atomic
117.                    pointsInsideCircle++;
```

```
            }
        }
    }

    // The formula to calculate the PI number
    return 4 * ((1.0 * pointsInsideCircle) / (numberOfPoints * 1.0));
}

double monteCarloParallelForReductionFunction(int numberOfPoints, double a, double b, double radius)
{
    // Monte-Carlo function for calculate PI value PARALLEL FOR REDUCTION method.

    // We make a parallel opperation, but these is sync to the + of the pointsInsideCircle.

    int pointsInsideCircle = 0;

    #pragma omp parallel for reduction(+ : pointsInsideCircle)
    for (int i = 0; i < numberOfPoints; i++)
    {
        float x = generateRandNumbers(a, b);
        float y = generateRandNumbers(a, b);

         // Se calculeaza distanta de la punctul generat la centrul cercului din interiorul patratului.
         double distanceToCenter = sqrt(double(pow(x, 2) + pow(y, 2)));

        if (distanceToCenter < radius)
            {
                pointsInsideCircle++;
            }
    }

    // The formula to calculate the PI number
    return 4 * ((1.0 * pointsInsideCircle) / (numberOfPoints * 1.0));
}
```

Tema 3 – OpenMp: Calculul paralel a histogramei unei imagini

2. Paralelizați calculul histogramei din programul de mai sus utilizând OpenMP.

Indicatii:

- Se alocă memorie și se inițializează câte o histogramă „privată” fiecărui fir de execuție
- Se folosește directiva *omp parallel for* pentru a asocia fiecărui fir de execuție o porțiune din imagine
- Fiecare fir de execuție va calcula histograma porțiunii de imagine ce-i corespunde
- În final se calculează histograma globală însumând histogramele „private” calculate de fiecare fir de execuție

Imaginea este definita ca un singur tablou unidimensional, iar pentru a parcurge imaginea folosim doua for-uri. Unul ce parcurge liniile iar cel de-al doilea coloanele.

— **imagine [I * imgSize + J]**

ne da elementul de pe linia i coloana j.

Este acelasi lucru cu

imagine[i][j]

daca am folosi un tablou bidimensional.

• Histograma unei imagini trebuie sa aiba 256 de elemente, fiind un tablou unidimensional. (0 - 255 niveluri de gri). Pentru a popula histograma in modul clasic, vedem intensitatea px-lului din imagine, aceasta trebuie sa fie intre 0 si 255, iar pe pozitia specifica incrementam valoarea din histograma.

• Calculul histogramei paralele. Pentru a calcula aceasta histograma se va initializa un tablou bidimensional. Va avea un numar de randuri egal cu numarul maxim de threaduri pe care il are procesorul, **omp_get_max_threads( ) ,** iar numarul de coloane este egal cu 256, adica numarul nivelurilor de gri. Se poate paraleliza cu **#pragma omp parallel for**, lasand omp-ul sa se ocupe de impartirea threadurilor. La calcularea histogramei, se ia valoarea px-lului din imagine si numele thread-ului pentru a stii pe ce rand din tablou ne aflam. La locatia specificata vom incrementa . La final vom avea un numar de N randuri de histogram. Aceste trebuiesc adunate pentru a afla histograma totala. **Astfel nu va aparea RACE CONDITION.**

• Se verifica daca histograma paralela si cea secventiala sunt aceleasi. Acestea trebuie sa fie egale deoarece au aceeasi imagine de intrare.

```cpp
#include <iostream>
#include <time.h>
#include <omp.h>
void simpleHistogram(int imgSize, int *hist, unsigned char *img);
void parallelHistogram(int imgSize, int *hist, int **global_hist, unsigned char *img, int number_of_threads);
bool identicalHistogramCheck(int *hist1, int *hist2);

int main()
{

    int imgSize = 10000;
    unsigned char *img = new unsigned char[imgSize * imgSize];
    int *secventialHistogramArray = new int[256];
    int *parallelHistogramArray = new int[256];

    int numberOfThreads = omp_get_max_threads();
    printf("\nNumarul de threaduri pe care lucreaza procesorul este : %d ",numberOfThreads);

    // Initializarea imagini cu elemente intre 0 si 255
    for (int i = 0; i < imgSize; i++)
    {
        for (int j = 0; j < imgSize; j++)
        {
            img[i * imgSize + j] = i % 256;
        }
    }

    // Initializarea histogramei cu toate elementele 0
    for (int i = 0; i < 256; i++)
    {
        secventialHistogramArray[i] = 0;
        parallelHistogramArray[i]   = 0;
    }

    // Se calculeaza o histograma cu metoda clasica, neparalelizata
    clock_t timerStart = clock();
    {
        simpleHistogram(imgSize, secventialHistogramArray, img);
    }
    clock_t timerStop = clock();
    double secventialElapsedTime = ((double)(timerStop - timerStart)) / CLOCKS_PER_SEC;

    // Se afiseaza timpul de executie pentru realizarea acestui calcul
    printf("\nTimp necesar pentru calcularea histogramei, metoda secventiala: %.3lf", secventialElapsedTime);

    // Trebuie definita o histograma pentru fiecare sectiune, adica fiecare thread al procesorului
    // va avea o histograma separata
    int **privareHistogramArray = new int *[numberOfThreads];
    for (int i = 0; i < numberOfThreads; i++)
    {
        privareHistogramArray[i] = new int[256];
    }

    // Aceasta histograma se initileaza cu 0 pentru a nu avea date redundante in cod
    for (int i = 0; i < numberOfThreads; i++)
    {
        for (int j = 0; j < 256; j++)
        {
            privareHistogramArray[i][j] = 0;
        }
    }
```

```
61.
62.     // Se calculeaza histograma cu metoda paralela de calcul, aceasta facandu-
    se pe mai multe threaduri
63.     clock_t timerParallelStart = clock();
64.     {
65.         parallelHistogram(imgSize, parallelHistogramArray, privareHistogramArray, img, numberOfThreads);
66.     }
67.     clock_t timerParallelStop = clock();
68.
69.     double parallelElapsedTime = ((double)(timerParallelStop - timerParallelStart)) / CLOCKS_PER_SEC;
70.     // Se calculeaza timpul necesar pentru calcularea histogramei cu metoda paralela
71.     printf("\nTimp necesar pentru calcularea histogramei, metoda paralela: %.3lf", parallelElapsedTime);
72.
73.     // Se verigica daca cele doua histograme sunt aceleasi
74.     // Practic acestea trebuie sa fie aceleasi, deoarece au aceeasi imagine de intrare

75.     // Daca acestea sunt diferite inseamna ca este o greseala la unul din algoritmi de
    calcul
76.     // ai histogramei
77.
78.     identicalHistogramCheck(secventialHistogramArray, parallelHistogramArray) == true ?
     printf("\nHistogramele sunt identice !")
79.                                                                                         :
     printf("\nHistogramele sunt diferite !");
80.
81.     // Se elibereaza memoria care a fost folosita
82.     delete img;
83.     delete secventialHistogramArray;
84.     delete parallelHistogramArray;
85.     delete privareHistogramArray;
86. }
87.
88. void simpleHistogram(int imgSize, int *histogram, unsigned char *img)
89. {
90.     // Se parcurge imaginea atat pe linii cat si pe coloane
91.     // Iar in functie de intensitatea px curent, se incrementeaza valoarea din histograma
92.     for (int i = 0; i < imgSize; i++)
93.     {
94.         for (int j = 0; j < imgSize; j++)
95.         {
96.             histogram[img[i * imgSize + j]]++;
97.         }
98.     }
99. }
100.
101.        void parallelHistogram(int imgSize, int *currentHistogram, int **globalHistogram
    , unsigned char *img, int numberOfThreads)
102.        {
103.            // Se imparte imaginea in functie de cate thread-uri are acasta
104.            #pragma omp parallel for
105.            for (int i = 0; i < imgSize; i++)
106.            {
107.                // Se ia thread-ul curent
108.                int current_thread = omp_get_thread_num();
109.                for (int j = 0; j < imgSize; j++)
110.                {
111.                    // La histograma corespunzatoare thread-ului curent, la valoarea px-
    ului se incrementeaza
112.                    globalHistogram[current_thread][img[i * imgSize + j]]++;
113.                }
114.            }
```

```
115.
116.            // Din toate histogramele se realizeaza doar una singura
117.            for (int i = 0; i < numberOfThreads; ++i)
118.            {
119.                for (int j = 0; j < 256; ++j)
120.                {
121.                    currentHistogram[j] += globalHistogram[i][j];
122.                }
123.            }
124.        }
125.
126.        bool identicalHistogramCheck(int *histogram1, int *histogram2)
127.        {
128.            // Se parcurg cele doua histograme si se verifica daca sunt exact aceleasi
129.            bool ok = true;
130.
131.            for (int i = 0; i < 256; i++)
132.            {
133.                if (histogram1[i] != histogram2[i])
134.                    ok = false;
135.            }
136.
137.            return ok;
138.        }
```

Inmultirea a doua matrici patratice

```
1.  #include <iostream>
2.  #include <omp.h>
3.  #include <time.h>
4.  #include <cmath>
5.
6.  #define PI 3.14
7.
8.  void generate_first_matrix(unsigned char *matrix, int rows, int cols);
9.  void generate_second_matrix(unsigned char *matrix, int rows, int cols);
10. void generate_first_matrix_parallel(unsigned char *matrix, int rows, int cols);
11. void generate_second_matrix_parallel(unsigned char *matrix, int rows, int cols);
12. void multiplication_2d_matrix_classic_method(unsigned char *result, unsigned char *matr
    ix1, unsigned char *matrix2, int imgSize);
13. void multiplication_2d_matrix_parallel_method(unsigned char *result, unsigned char *mat
    rix1, unsigned char *matrix2, int imgSize);
14.
15.
16. int main()
17. {
18.     printf("Numarul de thread-
    uri pe care lucram este : %d \n", omp_get_max_threads());
19.     int imgSize = 500;
20.
21.     unsigned char *first_matrix = new unsigned char[imgSize * imgSize];
22.     unsigned char *second_matrix = new unsigned char[imgSize * imgSize];
23.
24.     unsigned char *first_matrix_parallel = new unsigned char[imgSize * imgSize];
25.     unsigned char *second_matrix_parallel = new unsigned char[imgSize * imgSize];
26.
27.     unsigned char *result_classic_metod = new unsigned char[imgSize * imgSize];
28.     unsigned char *result_parallel_metod = new unsigned char[imgSize * imgSize];
29.
30.
31.     clock_t start;
32.     clock_t stop;
33.     double time_to_calculate;
34.
35.     for (int i = 0; i < imgSize; i++)
36.     {
37.         for (int j = 0; j < imgSize; j++)
38.         {
39.             first_matrix[i * imgSize + j] = 0;
40.             first_matrix_parallel[i * imgSize + j] = 0;
41.             second_matrix_parallel[i * imgSize + j] = 0;
42.             second_matrix[i * imgSize + j] = 0;
43.         }
44.     }
45.
46.     start = clock();
47.     generate_first_matrix(first_matrix, imgSize, imgSize);
48.     stop = clock();
49.     time_to_calculate = double(stop - start) / CLOCKS_PER_SEC;
50.     std::cout << "Time to generate first matrix classic method: " << time_to_calculate
    << "s\n";
51.
52.
53.     start = clock();
```

```
54.     generate_second_matrix(second_matrix, imgSize, imgSize);
55.     stop = clock();
56.     time_to_calculate = double(stop - start) / CLOCKS_PER_SEC;
57.     std::cout << "Time to generate second matrix classic method: " << time_to_calculate
    << "s\n";
58. 
59. 
60.     start = clock();
61.     generate_first_matrix_parallel(first_matrix_parallel, imgSize, imgSize);
62.     stop = clock();
63.     time_to_calculate = double(stop - start) / CLOCKS_PER_SEC;
64.     std::cout << "Time to generate first matrix parallel method: " << time_to_calculate
    << "s\n";
65. 
66. 
67.     start = clock();
68.     generate_second_matrix_parallel(second_matrix_parallel, imgSize, imgSize);
69.     stop = clock();
70.     time_to_calculate = double(stop - start) / CLOCKS_PER_SEC;
71.     std::cout << "Time to generate second matrix parallel method: " << time_to_calculat
   e << "s\n";
72. 
73. 
74.     start = clock();
75.     multiplication_2d_matrix_classic_method(result_classic_metod, first_matrix, second_
   matrix, imgSize);
76.     stop = clock();
77.     time_to_calculate = double(stop - start) / CLOCKS_PER_SEC;
78.     std::cout << "Time to multiplicate 2d matrix classic method: " << time_to_calculate
    << "s\n";
79. 
80. 
81.     start = clock();
82.     multiplication_2d_matrix_parallel_method(result_parallel_metod, first_matrix, secon
   d_matrix, imgSize);
83.     stop = clock();
84.     time_to_calculate = double(stop - start) / CLOCKS_PER_SEC;
85.     std::cout << "Time to multiplicate 2d matrix parallel method: " << time_to_calculat
   e << "s\n";
86. 
87. 
88.     // Free memory
89.     delete first_matrix;
90.     delete second_matrix;
91.     delete first_matrix_parallel;
92.     delete second_matrix_parallel;
93.     delete result_classic_metod;
94.     delete result_parallel_metod;
95. }
96. 
97. void generate_first_matrix(unsigned char *matrix, int rows, int cols)
98. {
99.     for (int i = 0; i < rows; i++)
100.         {
101.             for (int j = 0; j < cols; j++)
102.             {
103.                 matrix[i * rows + j] = pow(sin((2 * PI * i) / rows), 2) + pow(cos((2
    * PI * i) / cols), 2);
104.             }
105.         }
106.     }
```

```
107.
108.       void generate_second_matrix(unsigned char *matrix, int rows, int cols)
109.       {
110.           for (int i = 0; i < rows; i++)
111.           {
112.               for (int j = 0; j < cols; j++)
113.               {
114.                   matrix[i * rows + j] = pow(cos((2 * PI * i) / rows), 2) + pow(sin((2
    * PI * i) / cols), 2);
115.               }
116.           }
117.       }
118.
119.       void generate_first_matrix_parallel(unsigned char *matrix, int rows, int cols)
120.       {
121.           #pragma omp parallel for
122.           for (int i = 0; i < rows; i++)
123.           {
124.               for (int j = 0; j < cols; j++)
125.               {
126.                   matrix[i * rows + j] = pow(sin((2 * PI * i) / rows), 2) + pow(cos((2
    * PI * i) / cols), 2);
127.               }
128.           }
129.       }
130.
131.       void generate_second_matrix_parallel(unsigned char *matrix, int rows, int cols)

132.       {
133.           #pragma omp parallel for
134.           for (int i = 0; i < rows; i++)
135.           {
136.               for (int j = 0; j < cols; j++)
137.               {
138.                   matrix[i * rows + j] = pow(cos((2 * PI * i) / rows), 2) + pow(sin((2
    * PI * i) / cols), 2);
139.               }
140.           }
141.       }
142.
143.       void multiplication_2d_matrix_classic_method(unsigned char *result, unsigned cha
    r *matrix1, unsigned char *matrix2, int imgSize){
144.
145.           for( int i = 0; i < imgSize; i++)
146.           {
147.               for( int j = 0; j < imgSize; j++)
148.               {
149.                   result[i * imgSize + j] = 0;
150.
151.                   for (int k = 0; k < imgSize; k++){
152.                       result[i * imgSize + j] += matrix1[i * imgSize + k] * matrix2[k
    * imgSize + j];
153.                   }
154.               }
155.           }
156.
157.       }
158.
159.       void multiplication_2d_matrix_parallel_method(unsigned char *result, unsigned ch
    ar *matrix1, unsigned char *matrix2, int imgSize)
160.       {
```

```
161.           #pragma omp parallel
162.           {
163.               int start_idx = omp_get_thread_num() * imgSize / omp_get_num_threads();

164.               int stop_idx = ( omp_get_thread_num() + 1 ) * imgSize / omp_get_num_thre
    ads();
165.               for( int i = start_idx; i < stop_idx; i++)
166.               {
167.                   for( int j = 0; j < imgSize; j++)
168.                   {
169.                       result[i * imgSize + j] = 0;
170.
171.                       for (int k = 0; k < imgSize; k++){
172.                           result[i * imgSize + j] += matrix1[i * imgSize + k] * matrix
    2[k * imgSize + j];
173.                       }
174.                   }
175.               }
176.           }
177.       }
```

CUDA

1.*Alocarea memoriei*

◘ Host: adica CPU

```
float * floats_h = new float[N];
float * floats_h = (float*)malloc(N * sizeof(float));
```

◘ Device: adica GPU

```
float * floats_d;
cudaMalloc((void**)&floats_d, N*sizeof(float));
```

2.Copierea memoriei - > Primul parametru este destinația iar al 2-lea este sursa

◘ De la host la device:

```
cudaMemcpy(floats_d, floats_h, N*sizeof(float),cudaMemcpyHostToDevice);
```

◘ De la device la host:

```
cudaMemcpy(floats_h, floats_d, N*sizeof(float),cudaMemcpyDeviceToHost);
```

3.Kernel-ul CUDA

◘ Kernel-ul: funcție ce se este apelată de pe CPUși se execută pe GPU
◘ Implementarea porțiunii de cod ce se vaexecuta pe GPU se face într-un kernel
◘ Definirea unui kernel:

```
__global__ void exemplu_kernel(float * p1, float * p2, int p3)
{
}
```

◘ Lansarea în execuție:

```
exemplu_kernel <<<1, 1 >>>( [lista de parametrii] );
```

4.*Transferul parametrilor la un kernelCUDA*

◘ Într-un kernel CUDA nu se pot folosii decâtvariabile ce se află în memoria device
◘ Toți parametrii unui kernel sunt copiați automatîn memoria device la momentul lansării în execuție

◘ Transferul direct:

```
__global__ void ceva(float f)
	{
		// Aici f va fi 1.234
	}
void main()
	{
		float x = 1.234;
		ceva<<<1,1>>>(x);
	}
```

◘Transferul prin pointer:

```
__global__ void ceva(float *f)
	{
		...
	}
void main()
	{
		float *f_dev;
		cudaMalloc((void**)&f_dev, ...)
		ceva<<<1,1>>>(f_dev);
	}
```

◘Transferul instanței unei structuri.

```
struct A
	{
		int i;
		float x;
		double y;
	}
__global__ void ceva(A a)
	{
		...
	}
void main()
	{
```

```
        A a;
        a.i = 0; a.i = 1.0f; a.y = 2.0;
        ceva<<<1,1>>>(a)
}
```

Exemplu

```
__global__ void test(int *a, int *b, int *c)
{
        *c = *a + *b;
}

int main()
{
        int a = 1, b = 2, c;
        int *a_dev, *b_dev, *c_dev;

        //Alocare pe GPU
        cudaMalloc((void**)&a_dev, sizeof(int));
        cudaMalloc((void**)&b_dev, sizeof(int));
        cudaMalloc((void**)&c_dev, sizeof(int));

        //Copiere CPU → GPU
        cudaMemcpy(a_dev, &a, sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy(b_dev, &b, sizeof(int), cudaMemcpyHostToDevice);

        //Lansarea in executie a kernel-ului
        test << <1, 1 >> >(a_dev, b_dev, c_dev);

        //Copere GPU → GPU
        cudaMemcpy(&c, c_dev, sizeof(int), cudaMemcpyDeviceToHost);

        printf("a+b = %d\n", c);

        return 0;
}
```

```
int main()
{
        int a = 1, b = 2, c;

        c = a + b;

        printf("a+b = %d\n", c);

        return 0;
}
```

6.Organizarea firelor de execuție CUDA

Block și *Grid*

- Pot avea una, două sautrei dimensiuni
- Lungimea se specifică la lansarea în execuție
- Un bloc aredimensiunile maxime 1024 x 1024 x 64, **până la maxim 1024 fire pe bloc in total**
- Un grid poate avea maxim 655353 blocuri

7.Paralelismul CUDA

Variabile preefinite în orice funcție kernel

– **threadIdx**: id-ul thread-ului curent in bloc

– **blockIdx**: id-ul blocului curent în grid

– **gridDim**: dimensiunea grid-ului (nr total de blocuri)

– **blockDim**: dimensiunea unui bloc (nr de fire intr-un bloc)

◘ Aceste variabile sunt de tip *dim3* (au ca membrii x,y și z pentru a descrie o rețea 3D de fire). Exemplu: threadIdx.x, threadidx.y, threadIdx.z

```
__global__ void exemplu_kernel()
        {
                int thread_id = threadIdx.x;
                int blockid = blockIdx.x;
        }
```

Lansarea în execuție:

exemplu_kernel <<<lungime_grid, lungime_block >>>([*lista de parametrii*]);

◘ Se lansează în execuție *lungime_grid*lungime_bloc* fire de execuție

◘ D.p.d.v. logic este echivalent cu:

```
for (int i = 0; i < lungime_grid*lungime_block; i++)
        {
                exemplu_kernel( [lista de parametrii] );
        }
```

Exemplu – Adunarea a doi vectori

Cod C standard

```
void sum_serial(int n,
float * a,
float * b,
float * c
)
{
for (int i = 0; i < n; i++)
        c[i] = a[i] + b[i];
}

sum_serial(4096 * 256, a, b, c);
```

CUDA

```
__global__
void sum_parallel(int n,
float * a,
float * b,
float * c
)
{
int i = blockIdx.x*blockDim.x +
threadIdx.x;
c[i] = a[i] + b[i];
}
sum_parallel<<<4096, 256>>>(4096 *
256, a, b, c);
```

6. *Tipuri de memorii CUDA*

◘Memorie globală
– Se alocă și eliberează*doar de pe CPU (cudaMalloc, cudaFree)*
– Poate fi accesată atât deCPU cât și de GPU (*cudaMemset, cudaMemcpy).*
– La nivel de GPU estevizibilă orcărui thread.
– Accesul este foarte lent.

◘ Memorie partajată (shared memory)
– Vizibilă doar de pe GPUla nivelul unui bloc.
– Se alocă în interiorulunui kernel. Ex:

__shared__ float f[128];

– Accesul este foarterapid.
– Limitată, mult mai micădecât memoria globală

◘ Regiștrii
– Vizibili doar la nivel dethread.
– Variabilele declaratelocal într-un thread vor fiautomat plasate înregiștrii.
– Nu pot fi declarați ca un tablou.
– Accesul este foarterapid.

Durata de viață
– Regiștrii: kernel
– Mem. Partajată: kernel
– Mem. Globală: Aplicație

Utilizare
– Regiștrii:

```
__global__ void kernel()
        {
                //Registrii
                float f1,f2,f3;
        }
```

– Memoria partajata:

```
__global__ void kernel()
        {
                //Shared memory
                __shared__ float tmp[128];
```

```
}
```

Factori ce limitează paralelismul

- Dacă se depășește numărul maxim de regiștrii aceștia vor vi alocați în memoria globală.
- Dacă se depășește cantitatea de memorie shared se reduce paralelismul
- DeviceQuery
- Opțiunea --ptxas-options=-v

Bariera de sincronizare a thread-urilor dintrun bloc.

__synchtreads();

Reducere paralelă

- Implementare: varianta 1

```
__global__ void sum( float *data, float *data_output, int size )
{
	__shared__ float sdata[BLOCK_SIZE];
	unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
	unsigned int tid = threadIdx.x;


	if( index >= size )
		return;


	sdata[tid] = data[index];
	__syncthreads();
	if( tid == 0 )
	{
		for( unsigned int i = 1; i < blockDim.x; i++ )
		{
			sdata[tid] += sdata[i];
		}
	 data[ blockIdx.x ] = sdata[0];

	}
}
```

Reducere paralelă

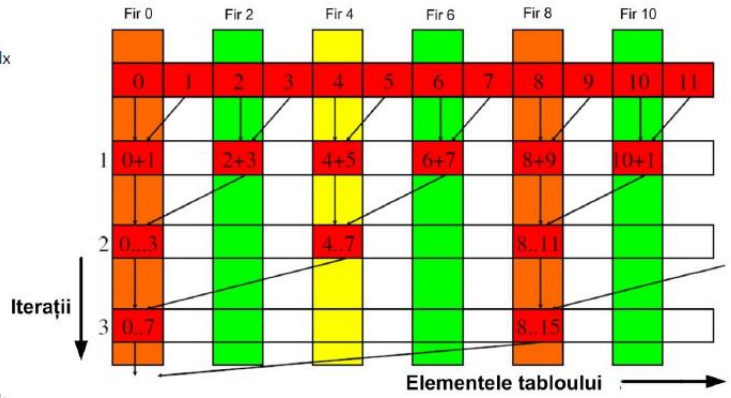
- Implementare: varianta 2

```
__global__ void sum( float *data, float *data_output, int size )
{
        __shared__ float sdata[128];
        unsigned int index = blockIdx.x * blockDim.x + threadIdx
        unsigned int tid = threadIdx.x;


        if( index >= size )
                return;


        sdata[tid] = data[index];
        __syncthreads();

        for(unsigned int s=1; s < blockDim.x; s *= 2)
        {
                if(tid % (2*s) == 0)
                {
                                sdata[tid] += sdata[tid + s];
                }
                __syncthreads();
        }
        if(tid == 0)
                g_odata[blockIdx.x] = sdata[0];
}
```



Reducere paralelă

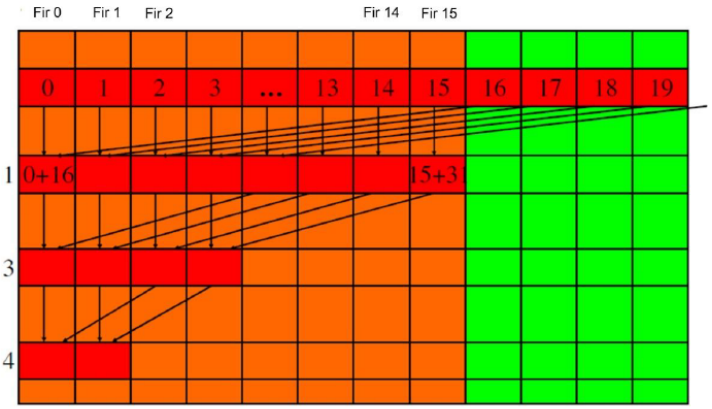
- Implementare: varianta 3

```
__global__ void sum( float *data, float *data_output, int size )
{
        __shared__ float sdata[128];
        unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
        unsigned int tid = threadIdx.x;

        if( index >= size )
                return;

        sdata[tid] = data[index];
        __syncthreads();
        for (unsigned int s = blockDim.x>>1; s > 0;s>>=1)
        {
                if (tid < s)
                                sdata[tid] += sdata[tid+s];
        }

        if(tid == 0)
                data[blockIdx.x] = sdata[0];
}
```



Tema 4 – CUDA: Matrici

1. Implementați un program CUDA ce generează două matrici cu valori date de două funcții de două variabile $f_1(i,j)$ și $f_2(i,j)$.

   Indicații:

   - Se va folosii un grid de thread-uri 2D
   - Funcțiile după care se calculează valoarea fiecărui element din prima matrice este $f_1(i,j) = \sin^2\left(\frac{2\pi i}{N}\right) + \cos^2\left(\frac{2\pi j}{M}\right)$ iar pentru a 2-a matrice $f_2(i,j) = \cos^2\left(\frac{2\pi i}{N}\right) + \sin^2\left(\frac{2\pi j}{M}\right)$ unde N și M reprezintă dimensiunea imaginii.
   - Fiecare matrice va fi alocată ca un singur tablou de $N \times M$ elemente de tip float

2. Modificați kernel-ul CUDA de la punctul 1 astfel încât să folosiți un grid de thread-uri 1D.

   Indicație: Valorile i și j se calculează din indicele global $i_g = threadIdx.x + blockIdx.x \cdot blockDim.x$

3. Implementați un kernel CUDA ce adună cele două matrici obținute la punctul 1 și afișați valorile matricei rezultat. Verificați ca valorile elementelor matricei rezultat să fie 2.

```
1.  #include "cuda_runtime.h"
2.  #include "device_launch_parameters.h"
3.
4.  #include <stdio.h>
5.  #include <cuda.h>
6.  #include <math.h>
7.
8.  #define PI 3.14f
9.
10. __global__ void generareMatrice2D(float *A, float *B, int N, int M)
11. {
12.     int i = blockIdx.x * blockDim.x + threadIdx.x;
13.     int j = blockIdx.y * blockDim.y + threadIdx.y;
14.
15.     if(i < N && j < M)
16.     {
17.         A[i*N + j] = powf( sinf( (2*PI*i)/N) , 2 ) + powf( cosf( (2*PI*j)/M ), 2 );
18.         B[i*N + j] = powf( cosf( (2*PI*i)/N) , 2 ) + powf( sinf( (2*PI*j)/M ), 2 );
19.     }
20. }
21.
22. __global__ void generareMatrice1D(float *A, float *B, int N, int M)
23. {
24.     int idx = blockIdx.x * blockDim.x + threadIdx.x;
25.
26.     int i = idx / N;
27.     int j = idx % N;
28.
29.     if (i < N && j < M)
30.     {
```

```
31.         A[i*N + j] = powf(sinf((2 * PI*i) / N), 2) + powf(cosf((2 * PI*j) / M), 2);
32.         B[i*N + j] = powf(cosf((2 * PI*i) / N), 2) + powf(sinf((2 * PI*j) / M), 2);
33.     }
34. }
35.
36. __global__ void sumOf2DMatrix(float *A, float *B, float *C, int N, int M)
37. {
38.     int i = blockIdx.x * blockDim.x + threadIdx.x;
39.     int j = blockIdx.y * blockDim.y + threadIdx.y;
40.
41.     if (i < N && j < M)
42.     {
43.         C[i*N + j] = A[i*N + j] + B[i*N + j];
44.     }
45. }
46.
47. __global__ void sumOf1DMatrix(float *A, float *B, float *C, int N, int M)
48. {
49.     int idx = blockIdx.x * blockDim.x + threadIdx.x;
50.
51.     int i = idx / N;
52.     int j = idx % N;
53.
54.     if (i < N && j < M)
55.     {
56.         C[i*N + j] = A[i*N + j] + B[i*N + j];
57.     }
58. }
59.
60. int main()
61. {
62.     int N = 512;
63.     int M = 512;
64.
65.     size_t size = N * M * sizeof(float);
66.
67.     // Alocare memorie pe host(CPU).
68.     float *A_H, *B_H, *C_H;
69.     A_H = (float*)malloc(size);
70.     B_H = (float*)malloc(size);
71.     C_H = (float*)malloc(size);
72.
73.     // Alocare memorie pe device (GPU).
74.     float *A_D, *B_D, *C_D;
75.     cudaMalloc((void**)&A_D, N*M * sizeof(float));
76.     cudaMalloc((void**)&B_D, N*M * sizeof(float));
77.     cudaMalloc((void**)&C_D, N*M * sizeof(float));
78.
79.
80.     // Dimensiuni grid si threads
81.     dim3 dimBlock2D(16, 16, 1);     // Numarul thread-urilor  continute de un bloc.
82.     dim3 dimGrid2D(32, 32, 1);      // Numarul de blocuri existente.
83.
84.     dim3 dimBlock1D(512, 1, 1); // Numarul thread-urilor  continute de un bloc.
85.     dim3 dimGrid1D(512, 1, 1);      // Numarul de blocuri existente.
86.
87.     generareMatrice2D <<< dimGrid2D, dimBlock2D >>> (A_D, B_D, N, M);
88.     sumOf2DMatrix <<< dimGrid2D, dimBlock2D >>> (A_D, B_D, C_D, N, M);
89.
90.     // Copy data from DEVICE(GPU) to HOST(CPU)
91.     cudaMemcpy(A_H, A_D, size, cudaMemcpyDeviceToHost);
```

```
92.     cudaMemcpy(B_H, B_D, size, cudaMemcpyDeviceToHost);
93.     cudaMemcpy(C_H, C_D, size, cudaMemcpyDeviceToHost);
94.
95.     // See some results :
96.     for (int i=0; i<10; i++)
97.     {
98.         for (int j = 0; j < 10; j++)
99.         {
100.                    printf("%.2f ", C_H[i*N+j]);
101.                }
102.            }
103.
104.            // Clear memory
105.            free(A_H);
106.            free(B_H);
107.            free(C_H);
108.            cudaFree(A_D);
109.            cudaFree(B_D);
110.            cudaFree(C_D);
111.
112.            return 0;
113.        }
```

Tema 5 – CUDA: Filtrarea unei imagini

1. Implementați un program CUDA ce aplică un filtru median unei imagini.

Indicații:

- Puteți folosii codul de la tema 3 pentru generarea unei imagini cu dimensiunea $N \times M$ (1000x1000)
- Imaginea este grayscale iar tipul pixelilor este float
- Rezultatele se vor scrie într-o nouă imagine p_{new}.
- Filtrul median se implementează în felul următor: Fiecare pixel $p_{new}(i,j)$ din noua imagine va avea valoarea mediei pixelilor aflați în regiunea dată de $(i-1, j-1)$ și $(i+1, j+1)$ din vechea imagine.

$$p_{new}(i,j) = \frac{1}{9} \sum_{i'=i-1}^{i+1} \sum_{j'=j-1}^{j+1} p_{old}(i',j')$$

```cpp
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include <cuda.h>
#include <math.h>


__global__ void medianFilter(float *inputImage, float *outputImage, int N, int M)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;    // ROW
    int j = blockIdx.y*blockDim.y + threadIdx.y;    // COLUMN

    float partialResult = 0;

    // Test if is a marginal pixel.
    if (i == 0 || i == N-1 || j == 0 || j == M-1)
        return;

    for (int k = i-1; k <= i+1; k++)
    {
        for (int l = j-1; l <= j+1; l++)
        {
            partialResult += inputImage[k*N + l];
        }
    }

    partialResult = partialResult / 9;

    outputImage[i*N + j] = partialResult;
}

int main()
{
    // Dimensiunile imaginii.
    int N = 256;
    int M = 256;

```

```
39.     // Dimensiunea.
40.     size_t size = N * M * sizeof(float);
41.
42.     // Imaginile pe HOST.
43.     float *imgOriginal_H;
44.     float *imgResult_H;
45.
46.     // Se aloca memorie pentru imaginile de le CPU.
47.     imgOriginal_H   = (float*)malloc(size);
48.     imgResult_H     = (float*)malloc(size);
49.
50.     // Imaginile pe DEVICE.
51.     float *imgOriginal_D;
52.     float *imgResult_D;
53.
54.     // Se aloca memorie in GPU pentru imagini.
55.     cudaMalloc((void**)&imgOriginal_D, size);
56.     cudaMalloc((void**)&imgResult_D, size);
57.
58.     // Dimensiunile gridului, adica numarul de blocuri si dimensiunea blocului,
59.     // adica numarul de fire de executie cuprinse intr.un bloc.
60.     dim3 numberOfThreadsPerBlock(16, 16, 1);
61.     dim3 numberOfBlocks(1, 1, 1);
62.
63.     // Initializarea imagini cu elemente intre 0 si 255
64.     for (int i = 0; i < N; i++)
65.     {
66.         for (int j = 0; j < M; j++)
67.         {
68.             // Imaginea primeste vvalori intre 0 si 255. (Grayscale)
69.             imgOriginal_H[i * N + j] = (i + j) % 2; //i % 256;
70.         }
71.     }
72.
73.     // Se transfera imaginea din CPU pe GPU.
74.     cudaMemcpy(imgOriginal_D, imgOriginal_H, size, cudaMemcpyHostToDevice);
75.
76.     // Se apeleaza filtrul median.
77.     medianFilter <<< numberOfBlocks, numberOfThreadsPerBlock >>> (imgOriginal_D, imgRes
    ult_D, N, M);
78.
79.     // Se transfera imaginea din GPU pe CPU.
80.     cudaMemcpy(imgResult_H, imgResult_D, size, cudaMemcpyDeviceToHost);
81.
82.     for (int i = 0; i < 10; i++)
83.     {
84.         for (int j = 0; j < 10; j++)
85.         {
86.             printf("%.2f ", imgResult_H[i + j * N]);
87.         }
88.         printf("\n");
89.     }
90.
91.     // Se elibereaza memoria care a fost folosita.
92.     free(imgOriginal_H);
93.     free(imgResult_H);
94.     cudaFree(imgOriginal_D);
95.     cudaFree(imgResult_D);
96.
97.     return 0;
98. }
```

Tema 6 – CUDA: Filtrarea unei imagini

1. Modificați programul de la tema 5 astfel încât să folosiți memoria partajată pentru a reduce numărul de accesări la memoria globală.

   Indicații:

   - Thread-urile CUDA sunt distribuite 2D
   - Numărul de thread-uri pe block este 1024 (distribuite 2D → 32*32)
   - Se ignoră pixelii aflați pe frontieră
   - Se lansează în execuție un singur block, deci dimensiunea imaginii va fi 32*32

```
1.  #include "cuda_runtime.h"
2.  #include "device_launch_parameters.h"
3.
4.  #include <stdio.h>
5.  #include <cuda.h>
6.  #include <math.h>
7.
8.
9.  __global__ void sharedMedianFilter(float *inputImage, float *outputImage, int N, int M)
10. {
11.     // Se declara o memorie shared.
12.     __shared__ float sharedMemory[32*32];
13.
14.     // Indicii de parcurgere a memoriei.
15.     int i = blockIdx.x*blockDim.x + threadIdx.x;     // ROW
16.     int j = blockIdx.y*blockDim.y + threadIdx.y;     // COLUMN
17.
18.     // Se copiaza imaginea in memoria partajata.
19.     sharedMemory[threadIdx.x + threadIdx.y * N] = inputImage[j+i*N];
20.
21.     // Se asteapta ca toate thread-urile sa fie terminate.
22.     __syncthreads();
23.
24.     float partialResult = 0;
25.
26.     // Test if is a marginal pixel.
27.     if (i == 0 || i == N - 1 || j == 0 || j == M - 1)
28.         return;
29.
30.     for (int k = i - 1; k <= i + 1; k++)
31.     {
32.         for (int l = j - 1; l <= j + 1; l++)
33.         {
34.             partialResult += inputImage[k*N + l];
35.         }
36.     }
37.
38.     outputImage[i*N + j] = partialResult / 9;
39. }
40.
```

```
41. int main()
42. {
43.     // Dimensiunile imaginii.
44.     int N = 32;
45.     int M = 32;
46. 
47.     // Dimensiunea.
48.     size_t size = N * M * sizeof(float);
49. 
50.     // Imaginile pe HOST.
51.     float *imgOriginal_H;
52.     float *imgResult_H;
53. 
54.     // Se aloca memorie pentru imaginile de le CPU.
55.     imgOriginal_H   = (float*)malloc(size);
56.     imgResult_H     = (float*)malloc(size);
57. 
58.     // Imaginile pe DEVICE.
59.     float *imgOriginal_D;
60.     float *imgResult_D;
61. 
62.     // Se aloca memorie in GPU pentru imagini.
63.     cudaMalloc((void**)&imgOriginal_D, size);
64.     cudaMalloc((void**)&imgResult_D, size);
65. 
66.     // Dimensiunile gridului, adica numarul de blocuri si dimensiunea blocului,
67.     // adica numarul de fire de executie cuprinse intr.un bloc.
68.     dim3 numberOfThreadsPerBlock(32, 32, 1);
69.     dim3 numberOfBlocks(1, 1, 1);
70. 
71.     // Initializarea imagini cu elemente intre 0 si 255
72.     for (int i = 0; i < N; i++)
73.     {
74.         for (int j = 0; j < M; j++)
75.         {
76.             // Imaginea primeste vvalori intre 0 si 255. (Grayscale)
77.             imgOriginal_H[i * N + j] = (i + j) % 2; //i % 256;
78.         }
79.     }
80. 
81.     // Se transfera imaginea din CPU pe GPU.
82.     cudaMemcpy(imgOriginal_D, imgOriginal_H, size, cudaMemcpyHostToDevice);
83. 
84.     // Se apeleaza kernelul de filtru median.
85.     sharedMedianFilter <<< numberOfBlocks, numberOfThreadsPerBlock >>> (imgOriginal_D,
    imgResult_D, N, M);
86. 
87.     // Se transfera imaginea din GPU pe CPU.
88.     cudaMemcpy(imgResult_H, imgResult_D, size, cudaMemcpyDeviceToHost);
89. 
90.     // Se afiseaza primele 10 elemente.
91.     for (int i = 0; i < 10; i++)
92.     {
93.         for (int j = 0; j < 10; j++)
94.         {
95.             printf("%.2f ", imgResult_H[i + j * N]);
96.         }
97.         printf("\n");
98.     }
99. 
100.          // Se elibereaza memoria care a fost folosita.
```

```
            free(imgOriginal_H);
            free(imgResult_H);
            cudaFree(imgOriginal_D);
            cudaFree(imgResult_D);

            return 0;
        }
```

Tema 5 – CUDA: Reducție paralelă

1. Implementați un program CUDA pentru căutarea valorii maxime într-un tablou de N elemente printr-o operație de reducție paralelă

   Indicații :

   - Elementele sunt de tip *int* iar *N* este 128
   - Elementele se inițializează cu valori aleatoare
   - Numărul de thread-uri pe bloc este 128 (distribuite 1D)
   - Se folosește memoria partajată pentru optimizarea operației de căutare a maximului la nivel de bloc

2. Modificați programul de la punctul 1 astfel încât acesta să funcționeze cu orice valoare a lui N (numărul de thread-uri pe bloc va rămâne 128).
3. Implementați același algoritm și pe CPU și comparați timpii de execuție pentru $N = 10^6$.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include <cuda.h>
#include <math.h>
#include <time.h>

__global__ void maximumValue(int *tablouIn, int *tablouOut)
{
    // Se declara memoria shared in care va fi copiat tabloul.
    __shared__ int sharedMemory[128];

    // Indexul de parcurgere a vectorului.
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    // Copiere vector in memoria partajata.
    sharedMemory[threadIdx.x] = tablouIn[i];

    // Sincronizare threaduri.
    __syncthreads();

    // Gasirea maximului.
    for (unsigned int s = blockDim.x >> 1; s > 0; s >>= 1)
    {
        if (threadIdx.x < s)
        {
```

```
            sharedMemory[threadIdx.x] = sharedMemory[threadIdx.x] > sharedMemory[threadIdx.x + s] ? sharedMemory[threadIdx.x] : sharedMemory[threadIdx.x + s];
        }
    }

    if (threadIdx.x == 0)
    {
        tablouOut[blockIdx.x] = sharedMemory[0];
    }
}


int main()
{
    // Numarul de elemente ale tabloului.
    int N = pow(10, 6);

    // Dimensiunea.
    size_t size = N * sizeof(float);

    // Se declara si aloca 2 tablouri de N elemente in HOST.
    float* table_H1;
    float* table_H2;
    table_H1 = (float*)malloc(size);
    table_H2 = (float*)malloc(size);

    // Se initializeaza cu elemente aleatoare.
    for (int i = 0; i < N; i++)
    {
        table_H1[i] = rand() % 1000;
    }

    // Declarare si alocare tablouri in DEVICE.
    int* table_D1;
    int* table_D2;
    cudaMalloc((void**)&table_D1, size);
    cudaMalloc((void**)&table_D2, size);

    // Se defineste gridul si threads.
    const int grid = N / 128;
    dim3 numberOfThreadsPerBlock(128, 1, 1);
    dim3 numberOfBlocks(grid, 1, 1);

    // Se defineste valoarea maxima pentru varianta secventiala.
    float maximumValue_H = -1;

    // Calculul maximului in varianta secventiala.
    clock_t timerSecventialStart = clock();
    for (int i = 0; i < N; i++)
    {
        if (table_H1[i] > maximumValue_H)
        {
            maximumValue_H = table_H1[i];
        }
    }
    clock_t timerSecventialStop = clock();
    double cpuTimeUsed = ((double)(timerSecventialStop - timerSecventialStart)) / CLOCKS_PER_SEC;


    // Se copiaza vectorul din Host pe Device
```

```
87.     cudaMemcpy(table_D1, table_H1, size, cudaMemcpyHostToDevice);
88.
89.     clock_t timerParallelStart = clock();
90.
91.     while (N > numberOfThreadsPerBlock.x)
92.     {
93.         cudaMalloc((void**)&table_D2, N / 128 * sizeof(float));
94.         dim3 numberOfBlocks(N / 128);
95.         maximumValue <<<numberOfBlocks, numberOfThreadsPerBlock >>> (table_D1, table_D2
    );
96.         cudaFree(table_D1);
97.         table_D1 = table_D2;
98.         N /= 128;
99.     }
100.
101.            cudaMemcpy(table_H2, table_D1, N * sizeof(float), cudaMemcpyDeviceToHost);
102.
103.            // Se defineste valoarea maxima pentru varianta paralela.
104.            float maximumValue_D = -1;
105.
106.            // Aici N are o valoare mult mai mica deoarece a fost impartit mai sus.
107.            for (int i = 0; i < N; i++)
108.            {
109.                if (table_H2[i] > maximumValue_D)
110.                {
111.                    maximumValue_D = table_H2[i];
112.                }
113.            }
114.            clock_t timerParallelStop = clock();
115.            double gpuTimeUsed = ((double)(timerParallelStop - timerParallelStart)) / CL
    OCKS_PER_SEC;
116.
117.            //Afisare valori maxime.
118.            printf("\nValoarea maxima CPU este : %.2f gasit in %.3lf secunde !", maximum
    Value_H, cpuTimeUsed);
119.            printf("\nValoarea maxima GPU este : %.2f gasit in %.3lf secunde !", maximum
    Value_D, gpuTimeUsed);
120.
121.            // Eliberare memorie.
122.            cudaFree(table_D1);
123.            cudaFree(table_D2);
124.            free(table_H1);
125.            free(table_H2);
126.
127.            return 0;
128.        }
```