Adunarea a doua matrici

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

void sumOfTwoMatrix()
{
	/* Adunarea a doua matrici cu N linii si N coloane si
	paralelizare a programului utilizand OpenMP*/

	int N = 4; //1000

	//Prima matrice
	int *A;
	// A doua matrice
	int *B;
	// Matricea rezultata
	int *C;

	// 1. Alocare memorie pe host pentru cele trei matrici:
	A = new int[N * N];
	B = new int[N * N];
	C = new int[N * N];

	/* 2. Initializati matricile A si B dupa cum urmeaza
		- elementele de pe prima coloana au valoarea 1
		- elementele de pe a doua coloana au valoarea 2
		- elementele de pe coloana a treia au valoarea 3*/

	// Initializare Matricea A
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			A[i * N + j] = j + 1;
		}
	}

	printf("Elementele matricei A sunt: \n");
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			printf("%3d ", A[i * N + j]);
		}
		printf("\n");
	}
```

```
	// Initializare Matricea B
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			B[i * N + j] = j + 1;
		}
	}

	printf("\nElementele matricei B sunt: \n");
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			printf("%3d ", B[i * N + j]);
		}
		printf("\n");
	}

	/*Implementati suma celor doua matrici si scrieti rezultatul in matricea c*/
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			C[i * N + j] = 0;
		}
	}

	// Suma celor doua matrici
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			C[i * N + j] = A[i * N + j] + B[i * N + j];
		}
	}

	printf("\nSuma celor doua matrici: \n");
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			printf("%3d ", C[i * N + j]);
		}
		printf("\n");
	}


	/* 4. Implementati varianta paralel a calculului de la cerinta precedenta*/

	omp_set_num_threads(4);

#pragma omp parallel
	{
		int threadId = omp_get_thread_num();
		int numberOfThreads = omp_get_num_threads();
		int iStart = threadId * N / numberOfThreads;
		int iStop = (threadId + 1) * N / numberOfThreads;
```

```
		// Suma celor doua matrici
		for (int i = iStart; i < iStop; i++)
		{
			for (int j = 0; j < N; j++)
			{
				C[i * N + j] = A[i * N + j] + B[i * N + j];
			}
		}

	}

	printf("\nSuma celor doua matrici: \n");
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			printf("%3d ", C[i * N + j]);
		}
		printf("\n");
	}

}
```

Scaderea a doua matrice

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


void loweringOfTwoMatrix()
{
	/* Scaderea a doua matrici cu N linii si N coloane si
	paralelizare a programului utilizand OpenMP*/

	int N = 4; //1000

	//Prima matrice
	int *A;
	// A doua matrice
	int *B;
	// Matricea rezultata
	int *C;
	// Matricea rezultata (paralela)
	int *C1;
	// Matricea rezultata (reductie)
	int *C2;

	// 1. Alocare memorie pe host pentru cele trei matrici:
	A = new int[N * N];
	B = new int[N * N];
	C = new int[N * N];
	C1 = new int[N * N];
	C2 = new int[N * N];

	/* 2. Initializati matricile A si B dupa cum urmeaza
		- elementele de pe prima coloana au valoarea 1
		- elementele de pe a doua coloana au valoarea 2
		- elementele de pe coloana a treia au valoarea 3*/

		// Initializare Matricea A
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			A[i * N + j] = j + 2;
		}
	}

	printf("Elementele matricei A sunt: \n");
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
```

```
                printf("%3d ", A[i * N + j]);
        }
        printf("\n");
    }

    // Initializare Matricea B
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            B[i * N + j] = j + 1;
        }
    }

    printf("\nElementele matricei B sunt: \n");
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            printf("%3d ", B[i * N + j]);
        }
        printf("\n");
    }

    /* 3. Implementati suma celor doua matrici si scrieti rezultatul in matricea c*/
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            C[i * N + j] = 0;
        }
    }

    // Suma celor doua matrici
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            C[i * N + j] = A[i * N + j] - B[i * N + j];
        }
    }

    printf("\nSuma celor doua matrici: \n");
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            printf("%3d ", C[i * N + j]);
        }
        printf("\n");
    }

    /* 4. Implementati varianta paralel a calculului de la cerinta precedenta*/

    omp_set_num_threads(4);

#pragma omp parallel
    {
```

```
			int threadId = omp_get_thread_num();
			int numberOfThreads = omp_get_num_threads();
			int iStart = threadId * N / numberOfThreads;
			int iStop = (threadId + 1) * N / numberOfThreads;
			// Suma celor doua matrici
			for (int i = iStart; i < iStop; i++)
			{
				for (int j = 0; j < N; j++)
				{
					C1[i * N + j] = A[i * N + j] - B[i * N + j];
				}
			}

		}

		printf("\nSuma celor doua matrici (parallel): \n");
		for (int i = 0; i < N; i++)
		{
			for (int j = 0; j < N; j++)
			{
				printf("%3d ", C1[i * N + j]);
			}
			printf("\n");
		}

#pragma omp parallel for reduction(+ : C2[i * N + j])
		{
			for (int i = 0; i < N; i++)
			{
				for (int j = 0; j < N; j++)
				{
					C2[i * N + j] = A[i * N + j] - B[i * N + j];
				}
			}
		}

		printf("\nSuma celor doua matrici(reduction): \n");
		for (int i = 0; i < N; i++)
		{
			for (int j = 0; j < N; j++)
			{
				printf("%3d ", C2[i * N + j]);
			}
			printf("\n");
		}
}
```

Produsul a doua matrice

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


void producOfTwoMatrix()
{
	/* Implementati un program ce calculeaza produsul a doua matrici
	cu N linii si N coloane iar apoi paralelizati programul utilizant OpenMp*/

	// Numarul N de linii/coloane
	int N = 4; //1000
	// Prima matrice
	int *A;
	// Adoua matrice
	int *B;
	// Matricea rezultata
	int *C;
	// Matricea rezultata (paralela)
	int *C1;
	// Matricea rezultata (reduction)
	int *C2;

	/* 1. Alocati memorie pe host pentru cele trei matrici*/
	A = new int[N * N];
	B = new int[N * N];
	C = new int[N * N];
	C1 = new int[N * N];
	C2 = new int[N * N];

	/* 2. Initializati matricile a si b dupa cum urmeaza:
		- diagonala principala a matricei A are valoarea 3
		- diagonala secundara a matricei B are valoarea 2*/

	// Initializarea matrice A
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			if (i == j)
			{
				A[i * N + j] = 5;
			}
			else
			{
				A[i * N + j] = 2;
			}
		}
	}

	printf("Elementele matricei A sunt: \n");
```

```
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			printf("%3d ", A[i * N + j]);
		}
		printf("\n");
	}

	// Initializarea matrice B
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			if ( j == N - 1 - i)
			{
				B[i * N + j] = 5;
			}
			else
			{
				B[i * N + j] = 2;
			}
		}
	}

	printf("\nElementele matricei B sunt: \n");
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			printf("%3d ", B[i * N + j]);
		}
		printf("\n");
	}


	/* 3. Implementati inmultirea celor doua matrici si scrieti rezultatul in matricea
C*/

	// Initializare matricei C
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			C[i * N + j] = 0;
			C1[i * N + j] = 0;
			C2[i * N + j] = 0;
		}
	}

	// Inmultirea celor doua matrici
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			for (int k = 0; k < N; k++)
			{
				C[i * N + j] += A[i * N + k] * B[k * N + j];
```

```
                }
            }
        }

        printf("\nInmultirea celor doua matrici este: \n");
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
            {
                printf("%3d ", C[i * N + j]);
            }
            printf("\n");
        }

        /* 4. Implementati varianta paralele a calculului de la cerinta precedenta*/

        omp_set_num_threads(4);

#pragma omp parallel
        {
            int threadId = omp_get_thread_num();
            int numberOfThreads = omp_get_num_threads();
            int iStart = threadId * N / numberOfThreads;
            int iStop = (threadId + 1) * N / numberOfThreads;

            for (int i = iStart; i < iStop; i++)
            {
                for (int j = 0; j < N; j++)
                {
                    for (int k = 0; k < N; k++)
                    {
                        C1[i * N + j] += A[i * N + k] * B[k * N + j];
                    }
                }
            }
        }

        // Afisarea inmultirii matricei C prin varianta paralela
        printf("\nMatricea C prin varianta paralela este: \n");
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
            {
                printf("%3d ", C1[i * N + j]);
            }
            printf("\n");
        }


#pragma omp parallel for reduction(+ : C2[i * N + j])
        {
            for (int i = 0; i < N; i++)
            {
                for (int j = 0; j < N; j++)
                {
                    for (int k = 0; k < N; k++)
                    {
                        C2[i * N + j] += A[i * N + k] * B[k * N + j];
```

```
                        }
                }
        }
    }

    // Afisarea inmultirii matricei C prin varianta paralela  (reduction)
    printf("\nMatricea C prin varianta reductiei paralele: \n");
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            printf("%3d ", C2[i * N + j]);
        }
        printf("\n");
    }

}
```

Produsul intre 2 matrici cu N linii si N coloane

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

// Numarul N de linii/coloane
#define N 4

int main()
{
	// Prima matrice
	int *A;
	// A 2-a matrice
	int *B;
	// Matricea rezultat
	int *C;

	// 1. Alocati memorie pe host pentru cele trei matrici
	// 0.333 puncte
	A = new int[N * N];
	B = new int[N * N];
	C = new int[N * N];

	/* 2. Initializati matricile a si b dupa cum urmeaza:
		- elementele de pe prima coloana au valoarea 1
		- elementele de pe a doua coloana au valoarea 2
		- elementele de pe a treia coloana au valoarea 4 */
	// 0.333 puncte

	printf("\n Matricea A este: \n");
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			A[i * N + j] = j+1;
		}
	}

	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			printf("%3d ", A[i * N + j]);
		}
		printf("\n");
	}

	printf("\n Matricea B este: \n");
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			B[i * N + j] = j + 1;
		}
```

```
	}

	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			printf("%3d ", B[i * N + j]);
		}
		printf("\n");
	}


	/* 3. Implementati inmultirea celor doua matrici si scrieti rezultatul in matricea
C*/
	// 0.333 puncte

	// Initializarea matricei C cu 0
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			C[i * N + j] = 0;
		}
	}

	// Inmultirea celor doua matrici
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			for (int k = 0; k < N; k++)
			{
				C[i * N + j] += A[i * N + k] * B[k * N + j];
			}
		}
	}


	printf("\n Produsul dintre cele doua matrici este: \n");
	// Afisarea matricei C
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			printf("%3d ", C[i * N + j]);
		}
		printf("\n");
	}




/* 4. Implementati varianta paralela ca calculului de la cerinta precedenta
		Indicatie: doar bucla for exterioara (ce parcurge liniile) se distribuie
		pe mai multe fire de executie */
	// 1 pct
```

```
#pragma omp parallel for reduction (+:C[i * N + j])
	{
		for (int i = 0; i < N; i++)
		{
			for (int j = 0; j < N; j++)
			{
				for (int k = 0; k < N; k++)
				{
					C[i * N + j] += A[i * N + k] * B[k * N + j];
				}
			}
		}
	}

	printf("\n Produsul matricelor varianta paralela: \n");
	// Afisarea matrice C varianta paralela
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			printf("%3d ",C[i * N + j]);
		}
		printf("\n");

		return 0;
	}
}
```

Produsul intre o matrice si vector

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


void product_matrixAndVector()
{
	/* Implementarea unui program pentru inmultirea unei matrici patratice A cu un
vector*/

	// Numarul N de linii/coloane
	int N = 4; //1000
	// Prima matrice
	float *A;
	// Vectorul B
	float *B;
	// Rezultatul secvential
	float *C1;
	// Rezultatul paralel
	float *C2;

	/* 1. Alocati memorie pe host pentru cele trei matrici */
	A = new float[N * N];
	B = new float[N];
	C1 = new float[N];
	C2 = new float[N];

	/* 2. Initializare matricea A si vectorii*/

	// Matricea A
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			A[i * N + j] = 2;
		}
	}

	// Initializarea vectorului B cu 1 si C1,C2 cu 0
	for (int i = 0; i < N; i++)
	{
		B[i] = 2;
		C1[i] = 0;
		C2[i] = 0;
	}

	/* 3. Implementati inmultirea matricei si a vectorului si afisati rezultatul*/
	// Inmultirea vectorului si a matricei
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
```

```
		{
			C1[i] += A[i * N + j] * B[j];
		}
	}


	//Afisarea vectorului obtinut
	printf("Vectorul rezultat este: ");
	for (int i = 0; i < N; i++)
	{
		printf("%.2f ", C1[i]);
	}
	printf("\n");

	/* 4. Implementati varianta paralela a calculului de la cerinta precedenta*/

	omp_set_num_threads(4);

#pragma omp parallel
	{
		int threadId = omp_get_thread_num();
		int numberOfThreads = omp_get_num_threads();
		int iStart = threadId * N / numberOfThreads;
		int iStop = (threadId + 1) * N / numberOfThreads;

		for (int i = iStart; i < iStop; i++)
		{
			for (int j = 0; j < N; j++)
			{
				C2[i] += A[i * N + j] * B[j];
			}

		}
	}

	// Afisare vectorului obtinut metoda paralela
	printf("\nVectorul rezultat este (metoda paralela): ");
	for (int i = 0; i < N; i++)
	{
		printf("%.2f ", C2[i]);
	}
	printf("\n");

}
```

Cuda ce efectueaza adunarea a doua matrici patratice

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cuda.h>
#include <cuda_runtime_api.h>
#include <device_functions.h>
#include <stdio.h>

/* CUDA
	Implementati un program CUDA ce efectueaza adunarea a doua matrici patratice
	cu N lini si N coloane */

__global__ void matrix_matrix_add(float *A, float *B, float *C, int N)
{
	// Indicele fiecatui thread
	int index; // trebuie sters

	/* 7. Calculati valoarea celor doi indici i si j utilizand variabilele predefinite
		blockDim, blocIdx si threadIdx astfel incat fiecare thread sa corespunda
unei pozitii din matrice*/

	int i = blockIdx.x * blockDim.x + threadIdx.x;
	int j = blockIdx.y * blockDim.y + threadIdx.y;

	/* 8. Efectuati adunarea matricilor a si b si scrieti rezultatul in C (C = A +
B)*/
	C[i * N + j] = A[i * N + j] + B[i * N + j];

}

int main()
{
	// Numarul de linii / coloane
	int N = 1024;
	// Prima matrice
	float *A_H;
	// A 2-a matrice
	float *B_H;
	// matricea rezultat
	float *C_H;

	size_t size = N * N * sizeof(float);

	/* 1. Alocati memorie pe host pentru cele trei matrici:
		Matricea este de N linii si N coloane si este alocata ca un
		bloc de memorie (o singura alocare de N*N elemente)*/

	A_H = (float*)malloc(size);
	B_H = (float*)malloc(size);
	C_H = (float*)malloc(size);

	/* 2. Initializati cele doua matrici a si b dupa cum urmeaza:
```

```
	- prima matrice va fi initializata cu zero peste tot exceptand diagonala
		principala unde elementele au valoarea 5
	- a doua matrice va fi initializata cu 1 peste tot exceptand diagonala
		secundara unde elementele au valoarea 3*/

// Initializarea matricei A
for (int i = 0; i < N; i++)
{
	for (int j = 0; j < N; j++)
	{
		if (i == j)
		{
			A_H[i * N + j] = 5;
		}
		else
		{
			A_H[i * N + j] = 0;
		}
	}
}

// Initializarea matricei B
for (int i = 0; i < N; i++)
{
	for (int j = 0; j < N; j++)
	{
		if ( j == N - 1 - i)
		{
			B_H[i * N + j] = 3;
		}
		else
		{
			B_H[i * N + j] = 1;
		}
	}
}

// Cele trei matrici entru memoria device

float *A_D;
float *B_D;
float *C_D;

/* 3. Alocati memorie pe device pentru cele trei matrici*/
cudaMalloc((void**)&A_D, size);
cudaMalloc((void**)&B_D, size);
cudaMalloc((void**)&C_D, size);

/* 4. Copiati continutul celor doua matrici a si b de pe host pe device*/
// IMP: Primul parametru este destinatia iar al doilea sursa !!
cudaMemcpy(A_D, A_H, size, cudaMemcpyHostToDevice);
cudaMemcpy(B_D, B_H, size, cudaMemcpyHostToDevice);

// Numarul de thread-uri pe bloc
dim3 threadsPerBlock;
threadsPerBlock.x = 32;
threadsPerBlock.y = 32;
threadsPerBlock.z = 1;
```

```
	// Numarul de blocuri
	dim3 numBlocks;

	/* 5. Calculati numarul de blocuri astfel incat:
		- se obtine un grid 2D de thread-uri
		- se va lasa in executie cate un thread pentru fiecare element din matrice
			deci NxN in total*/

	numBlocks.x = 32;
	numBlocks.y = 32;
	numBlocks.z = 1;

	// Se lanseaza in executie kernel-ul CUDA
	matrix_matrix_add << < numBlocks, threadsPerBlock >> > (A_D, B_D, C_D, N);

	/* 6. Copiati continutul matricei rezultat C de pe device pe host*/
	// IMP: Primul parametru este destinatia iar al doilea sursa !!
	cudaMemcpy(C_H, C_D, size, cudaMemcpyDeviceToHost);


	printf("\n Suma celor doua matrici este: \n");
	// Afisarea sumei matricei
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		{
			printf("%.2f ", C_H[i * N + j]);
		}
		printf("\n");
	}

	return 0;
}
```

Produsul scalar intre doi vectori CUDA

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cuda.h>
#include <cuda_runtime_api.h>
#include <device_functions.h>
#include <stdio.h>

/*
	Implementati un program CUDA ce efectueaza adunarea a doua matrici
	patratice cu N linii si N coloane
*/

__global__ void matrix_matrix_add(float *A, float *B, float *C, int N)
{
	__shared__ float sharedMemory[1024];

	/* 7. Calculati valoarea celor doi indici i si j*/
	int i = blockIdx.x * blockDim.x + threadIdx.x;

	sharedMemory[i] = A[i] * B[i];

	__syncthreads();

	/* Efectuati adunarea matricilor A si B si scrieti rezultatul in C*/
	atomicAdd(&C[0], sharedMemory[i]);

}

int main()
{
	// Numarul N de linii/coloane
	int N = 4; //1024

	// Primul vector
	float *A_H;
	// Al doilea vector
	float *B_H;
	// numar rezultat
	float *C_H;

	size_t size = N * sizeof(float);

	/* 1. Alocati memorie pe host pentru cei trei vectori*/

	// Alocare memorie pe host (CPU)
	A_H = (float*)malloc(size);
	B_H = (float*)malloc(size);
	C_H = new float[1];
	/* 2. Initializati cei doi vectori A si B dupa cum urmeaza */
```

```
	// Initializare vector A si B
	for (int i = 0; i < N; i++)
	{
		A_H[i] = 2;
		B_H[i] = 3;
	}

	// Cei trei vectori pentru memoria device
	float *A_D;
	float *B_D;
	float *C_D;

	/* 3. Alocati memorie pe device pentru cei trei vectori*/

	cudaMalloc((void**)&A_D, size);
	cudaMalloc((void**)&B_D, size);
	cudaMalloc((void**)&C_D, 1* sizeof(float));

	/* 4. Copiati continutul celor doi vectori a si b de pe host de device*/
	cudaMemcpy(A_D, A_H, size, cudaMemcpyHostToDevice);
	cudaMemcpy(B_D, B_H, size, cudaMemcpyHostToDevice);

	// Numarul de thread-uri pe bloc
	dim3 threadsPerBlock;

	threadsPerBlock.x = 2; //32
	threadsPerBlock.y = 1;
	threadsPerBlock.z = 1;

	// Numarul de blocuri
	dim3 numBlocks;

	/* 5. Calculati numarul de blocuri astfel incat:
		- se obtine un grid 2D de thread-uri
		- se va lasa in executie cate un thread pentru fiecare element din matrice
dece NxN in total*/

	numBlocks.x = 2; //32
	numBlocks.y = 1;
	numBlocks.z = 1;

	// Se lanseaza in executie kernel-ul CUDA
	matrix_matrix_add <<<numBlocks, threadsPerBlock >>> (A_D, B_D, C_D, N);

	/* 6. Copiati continutul vectorului rezultat C de pe device pe host*/
	cudaMemcpy(C_H, C_D, sizeof(float), cudaMemcpyDeviceToHost);

	printf("\n Suma celor doua matrici este: \n");
	printf(" %.2f ", C_H[0]);

	return 0;

}
```

CUDA produs intre matrice si vector

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cuda.h>
#include <cuda_runtime_api.h>
#include <device_functions.h>
#include <stdio.h>



/*	CUDA
	Implementati un program CUDA ce efectueaza produsul dintre o matrice  si un vector
	 */

__global__ void matrix_matrix_add(float *A, float *B, float *C, int N)
{

	/* 7. Calculati valoarea celor doi indici i si j utilizand variabilele predefinite
		blockDim, blocIdx si threadIdx astfel incat fiecare thread sa corespunda
unei pozitii din matrice*/

	int i = blockIdx.x * blockDim.x + threadIdx.x;
	int j = blockIdx.y * blockDim.y + threadIdx.y;

	/* 8. Efectuati adunarea matricilor a si b si scrieti rezultatul in C (C = A +
B)*/
	C[i] += A[i * N + j] + B[j];

}

int main()
{
	// Numarul de linii / coloane
	int N = 1024;
	// Prima matrice
	float *A_H;
	// Vectorul
	float *B_H;
	// matricea rezultat
	float *C_H;


	/* 1. Alocati memorie pe host pentru matrice s i cei doi vectori:*/

	A_H = (float*)malloc(N * N * sizeof(float));
	B_H = (float*)malloc(N * sizeof(float));
	C_H = (float*)malloc(N * sizeof(float));

	/* 2. Initializati matricea a si vectorul b dupa cum urmeaza:
		- prima matrice va fi initializata cu zero peste tot exceptand diagonala
			principala unde elementele au valoarea 5
		- vectorul va fi initializat cu 2*/
```

```
		// Initializarea matricei A
for (int i = 0; i < N; i++)
{
	for (int j = 0; j < N; j++)
	{
		if (i == j)
		{
			A_H[i * N + j] = 5;
		}
		else
		{
			A_H[i * N + j] = 0;
		}
	}
}

// Initializarea vectorul B si C
for (int i = 0; i < N; i++)
{
	B_H[i] = 2;
	C_H[i] = 0;
}

// Matricea si vectorii pentru memoria device

float *A_D;
float *B_D;
float *C_D;

/* 3. Alocati memorie pe device*/
cudaMalloc((void**)&A_D, N * N * sizeof(float));
cudaMalloc((void**)&B_D, N * sizeof(float));
cudaMalloc((void**)&C_D, N * sizeof(float));

/* 4. Copiati continutul matricei A si a vectorului B de pe host pe device*/
// IMP: Primul parametru este destinatia iar al doilea sursa !!
cudaMemcpy(A_D, A_H, N * N * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(B_D, B_H, N * sizeof(float), cudaMemcpyHostToDevice);

// Numarul de thread-uri pe bloc
dim3 threadsPerBlock;
threadsPerBlock.x = 32;
threadsPerBlock.y = 32;
threadsPerBlock.z = 1;

// Numarul de blocuri
dim3 numBlocks;

/* 5. Calculati numarul de blocuri astfel incat:
	- se obtine un grid 2D de thread-uri
	- se va lasa in executie cate un thread pentru fiecare element din matrice
		deci NxN in total*/

numBlocks.x = 32;
numBlocks.y = 32;
numBlocks.z = 1;
```

```
	// Se lanseaza in executie kernel-ul CUDA
	matrix_matrix_add << < numBlocks, threadsPerBlock >> > (A_D, B_D, C_D, N);

	/* 6. Copiati continutul matricei rezultat C de pe device pe host*/
	// IMP: Primul parametru este destinatia iar al doilea sursa !!
	cudaMemcpy(C_H, C_D, N * sizeof(float), cudaMemcpyDeviceToHost);


	printf("\n Suma matricei si a vectorului este: \n");
	// Afisarea sumei matricei
	for (int i = 0; i < 4; i++)
	{
		printf("%.2f ", C_H[i]);
	}

	return 0;
}
```