```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cuda_runtime_api.h>
#include <device_functions.h>
#include <cuda.h>
#include <iostream>
#include <stdio.h>
using namespace std;

#define N 1000

__global__ void produsScalar(int *A, int *B, int *rez)
{
	// Declarare memorie shared pentru 1000 de elemente
	__shared__ int sdata[1000];

	// Declarare index
	int i = threadIdx.x + blockDim.x * blockIdx.x;

	// Initializare memorie shared
	sdata[threadIdx.x] = A[i] * B[i];

	// Primul thread initializeaza rez cu 0
	if (i == 0) *rez = 0;

	// Sincronizare threaduri
	__syncthreads();

	// Insumarea elementelor pentru a obtine rezultatul final
	if (threadIdx.x == 0)
	{
```

```
		int sum = 0;
		for (int j = 0; j < 1024; j++)
		{
			sum += sdata[j];
			atomicAdd(rez, sum);
		}
	}



}


int main()
{



	// ---------------------------- Punctul 1 ----------------------------------


	int vectHostA[N], vectHostB[N], vectHostRez;



	vectHostRez = 0;
	// Initializare vectori
	for (int i = 0; i < N; i++)
	{
		vectHostA[i] = 2;
		vectHostB[i] = 3;
	}
```

```
	for (int i = 0; i < N; i++)
	{
			vectHostRez += vectHostA[i] * vectHostB[i];

	}

	printf("%d", vectHostRez);

	// ------------------------ Punctul 2 ---------------------------------

	// Alocare si declarare vectori pe host
//	int *vectHostA = new int[N];
//	int *vectHostB = new int[N];
//	int vectHostRez = 0;
//
//	// Initializare vectori
//	for (int i = 0; i < N; i++)
//	{
//		vectHostA[i] = 2;
//		vectHostB[i] = 3;
//	}
//
//#pragma omp parallel for reduction(+:sum)
//	{
//		for (int i = 0; i < N; i++)
//		{
//			vectHostRez += vectHostA[i] * vectHostB[i];
//		}
//	}
//
//	printf("%d", vectHostRez);
```

```
// ------------------------ Punctul 3 ----------------------------------
/*
// Alocare si declarare vectori pe host
int *vectHostA = new int[N];
int *vectHostB = new int[N];
int *vectHostRez = new int;

*vectHostRez = 0;
// Initializare vectori
for (int i = 0; i < N; i++)
{
	vectHostA[i] = 2;
	vectHostB[i] = 3;
}

// Alocare si declarare vectori pe device
int *vectDeviceA, *vectDeviceB, *vectDeviceRez;

cudaMalloc((void**)&vectDeviceA, N * sizeof(int));
cudaMalloc((void**)&vectDeviceB, N * sizeof(int));
cudaMalloc((void**)&vectDeviceRez, sizeof(int));

// Copiere vectori initializati pe host
cudaMemcpy(vectDeviceA, vectHostA, N * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(vectDeviceB, vectHostB, N * sizeof(int), cudaMemcpyHostToDevice);
//cudaMemcpy(vectDeviceRez, vectHostRez, sizeof(int), cudaMemcpyHostToDevice);

// Declarare dimensiuni gridsize si blocksize
// Avand in vedere ca trebuie sa inmultim 1000 elemente vom folosi
// un singur bloc ce contine 1024 de fire de executie sub forma unui
```

```
	// grid 2D de fire de executie
	dim3 gridsize(1, 1);    // 1 bloc
	dim3 blocksize(1024,1); // 32x32 de threaduri sub forma 2D

	// Apel kernel
	produsScalar << <gridsize, blocksize >> > (vectDeviceA, vectDeviceB, vectDeviceRez);

	// Copiere rezultat de pe device pe host
	cudaMemcpy(vectHostRez, vectDeviceRez, sizeof(int), cudaMemcpyDeviceToHost);

	// Afisare elemente
	printf("%d", *vectHostRez);



	free(vectHostA); free(vectHostB); free(vectHostRez);
	cudaFree(vectDeviceA); cudaFree(vectDeviceB); cudaFree(vectDeviceRez);
	*/
	return 0;
}
```