Analiza și Optimizarea Algoritmilor Intensivi Computațional folosind CUDA

Student: Cocoroada Robert-Marius

20 ianuarie 2026

Cuprins

1 Introducere **2**
Rezumat . . . 2

2 Fundamente Teoretice ale Algoritmilor **3**
2.1 Transformată Fourier Discretă (DFT) . . . 3
2.1.1 Scop și Definiție . . . 3
2.1.2 Analiza Complexității . . . 3
2.2 Înmulțirea Generală a Matricelor (GEMM) . . . 3
2.2.1 Scop și Definiție . . . 3
2.2.2 Analiza Complexității . . . 4
2.3 Convoluția 2D (Conv2D) . . . 4
2.3.1 Scop și Definiție . . . 4
2.3.2 Analiza Complexității . . . 4

3 Strategii de Implementare și Optimizare în CUDA **5**
3.1 Optimizarea DFT: Shared Memory Tiling . . . 5
3.1.1 Provocarea . . . 5
3.1.2 Soluția Implementată . . . 5
3.2 Optimizarea GEMM: Abordare Hibridă și cuBLAS . . . 5
3.2.1 Implementarea Hibridă . . . 5
3.3 Optimizarea Conv2D: Constant Memory și Halo Loading . . . 6
3.3.1 Soluția pentru "Memory Bound" . . . 6

4 Rezultate Experimentale și Analiză Comparativă **7**
4.1 Analiza Detaliată Conv2D . . . 7
4.2 Analiza Detaliată DFT . . . 8
4.3 Analiza Detaliată GEMM (cuBLAS) . . . 9

5 Concluzii **11**

A Codul sursă și Scripturi **12**
A.1 Codul sursă CUDA (.cu) . . . 12
A.1.1 Convoluție 2D (conv2d_cuda.cu) . . . 12
A.1.2 DFT (dft_cuda.cu) . . . 14
A.1.3 Înmulțirea Matricelor (gemm_cuda.cu) . . . 16
A.1.4 Scripturi Bash de Automatizare . . . 18

Capitolul 1

Introducere

În calculul numeric de înaltă performanță, limita principală nu mai este frecvența procesorului, ci lățimea de bandă a memoriei și capacitatea de a executa instrucțiuni simultan. Acest proiect investighează tehnicile de programare CUDA necesare pentru a depăși aceste limitări în algoritmi cu complexitate ridicată ($O(N^2)$ și $O(N^3)$).

Obiectivul principal este implementarea eficientă a codului, trecând de la abordări naive la utilizarea ierarhiei complexe de memorie a GPU-ului, și cuantificarea exactă a câștigului de performanță.

Rezumat

Această lucrare prezintă o analiză comparativă detaliată a performanței algoritmice între execuția secvențială (CPU), execuția paralelă multi-core (OpenMP/cBLAS) și execuția masiv paralelă pe GPU (CUDA). Studiul se concentrează pe trei clase de algoritmi fundamentali: DFT, GEMM și Conv2D. Rezultatele experimentale demonstrează că optimizările specifice arhitecturii GPU (Shared Memory Tiling, Constant Memory Caching) și utilizarea bibliotecilor dedicate (cuBLAS) pot oferi accelerații de peste **600x** față de implementările naive, reducând timpii de execuție de la secunde la milisecunde.

Capitolul 2

Fundamente Teoretice ale Algoritmilor

Acest capitol detaliază baza matematică și scopul fiecărui algoritm studiat, evidențiind complexitatea lor inerentă și necesitatea accelerării hardware.

2.1 Transformată Fourier Discretă (DFT)

2.1.1 Scop și Definiție

Transformata Fourier Discretă este un instrument fundamental în procesarea digitală a semnalelor (DSP), utilizat pentru a descompune un semnal discret din domeniul timp în componentele sale spectrale (domeniul frecvență). Aplicațiile sale variază de la filtrarea audio și compresia datelor, până la rezolvarea ecuațiilor diferențiale parțiale.

Matematic, pentru un semnal de intrare x de lungime N, valoarea fiecărui element X_k din spectrul de frecvență este definită astfel:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i\frac{2\pi}{N}kn}, \quad \forall k \in [0, N-1] \tag{2.1}$$

Unde $e^{-i\theta} = \cos(\theta) - i\sin(\theta)$ (Formula lui Euler).

2.1.2 Analiza Complexității

Calculul direct al DFT implică o dublă sumă. Pentru fiecare dintre cele N elemente de ieșire, trebuie parcurse toate cele N elemente de intrare. Aceasta rezultă într-o complexitate computațională de $O(N^2)$. Deși există algoritmul FFT (Fast Fourier Transform) cu $O(N \log N)$, DFT-ul direct este ideal pentru a testa capacitatea de paralelizare masivă, deoarece fiecare X_k poate fi calculat independent.

2.2 Înmulțirea Generală a Matricelor (GEMM)

2.2.1 Scop și Definiție

GEMM (General Matrix Multiplication) este nucleul algebrei liniare și, prin extensie, al rețelelor neuronale moderne (Deep Learning). Aproape orice strat "Dense" sau "Fully Connected" dintr-o rețea neuronală se reduce la o operație GEMM.

Operația standard este definită ca:

$$C = \alpha(A \times B) + \beta C \tag{2.2}$$

Unde A este o matrice $M \times K$, B este $K \times N$, iar C este $M \times N$. Elementul C_{ij} se obține prin produsul scalar al liniei i din A cu coloana j din B:

$$C_{ij} = \sum_{k=0}^{K-1} A_{ik} \cdot B_{kj} \tag{2.3}$$

2.2.2 Analiza Complexității

Algoritmul naiv presupune trei bucle imbricate, având o complexitate de $O(N^3)$ pentru matrice pătratice. Principala problemă nu este doar numărul de operații (FLOPS), ci și modelul de acces la memorie. Accesele necoerente la memorie (ex: parcurgerea unei matrice stocate "row-major" pe coloane) duc la o utilizare ineficientă a cache-ului CPU, făcând acest algoritm un candidat perfect pentru optimizarea "block-tiling" pe GPU.

2.3 Convoluția 2D (Conv2D)

2.3.1 Scop și Definiție

Convoluția 2D este operația fundamentală în procesarea imaginilor și în Rețelele Neurale Convoluționale (CNN). Scopul său este extragerea de trăsături (features) dintr-o imagine, cum ar fi muchii, texturi sau forme complexe, prin aplicarea unui filtru (kernel).

Matematic, pentru o imagine de intrare I și un kernel K de dimensiune $k \times k$, pixelul de ieșire $O(i, j)$ este calculat prin "glisarea" kernelului peste imagine:

$$O(i, j) = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} I(i + m, j + n) \cdot K(m, n) \tag{2.4}$$

2.3.2 Analiza Complexității

Complexitatea este proporțională cu dimensiunea imaginii ($H \times W$) și dimensiunea filtrului ($k \times k$), rezultând $O(H \cdot W \cdot k^2)$. Deoarece kernel-ul este mic și constant, dar aplicat repetitiv pe volume mari de date, algoritmul este clasificat ca fiind "Memory Bound" (limitat de lățimea de bandă a memoriei), necesitând strategii speciale de caching.

Capitolul 3

Strategii de Implementare și Optimizare în CUDA

Acest capitol descrie transpunerea algoritmilor teoretici în cod CUDA optimizat, detaliind tehnicile utilizate pentru a maximiza utilizarea hardware-ului.

3.1 Optimizarea DFT: Shared Memory Tiling

3.1.1 Provocarea

Implementarea naivă pe GPU suferă din cauza accesului global la memorie: fiecare fir citește întregul vector de intrare.

3.1.2 Soluția Implementată

Codul `dft_cuda.cu` utilizează **Shared Memory Tiling**:

- **Fluxul de execuție:** Se lansează N fire de execuție.
- **Reducerea accesului la memorie:** Firele dintr-un bloc colaborează pentru a încărca un segment ("tile") al semnalului în memoria partajată (L1 Cache controlat software). Astfel, accesul la memoria globală lentă se reduce de un factor egal cu dimensiunea blocului.
- **Vectorizare:** Utilizarea `__sincosf()` calculează simultan sinus și cosinus, reducând presiunea asupra unităților aritmetice.

3.2 Optimizarea GEMM: Abordare Hibridă și cuBLAS

3.2.1 Implementarea Hibridă

Pentru a asigura performanța pe întreg spectrul de dimensiuni N, am implementat o strategie hibridă:

- **Pentru $N < 2048$ (Kernel Custom):** Se utilizează un kernel manual cu Tiling (32×32). Acesta evită overhead-ul de inițializare al bibliotecilor externe pentru matrice mici.
- **Pentru $N \geq 2048$ (cuBLAS):** Se utilizează biblioteca NVIDIA cuBLAS. Aceasta este esențială deoarece activează unitățile Tensor Cores, capabile să execute operații matriceale într-un singur ciclu de ceas, oferind un throughput imposibil de atins prin cod C++ standard.

3.3 Optimizarea Conv2D: Constant Memory și Halo Loading

3.3.1 Soluția pentru "Memory Bound"

Deoarece Convoluția citește aceleași valori ale filtrului de milioane de ori, am utilizat memoria specializată a GPU-ului:

1. **Constant Memory (`__constant__`):** Filtrul de convoluție este stocat în memoria constantă (limitată la 64KB, dar extrem de rapidă). Mecanismul de broadcasting permite tuturor firelor dintr-un warp să primească valoarea filtrului simultan.
2. **Halo Loading:** Firele de execuție colaborează pentru a încărca în memoria partajată atât zona de interes, cât și pixelii de la margine ("halo"), eliminând citirile redundante și ramificările (if/else) din bucla critică interioară.

Capitolul 4

Rezultate Experimentale și Analiză Comparativă

Analiza se bazează pe testarea sistematică cu dimensiuni ale datelor variind exponențial. Tabelele de mai jos compară timpii de execuție (milisecunde) între cele trei implementări.

4.1 Analiza Detaliată Conv2D

Pentru Convoluție, am comparat implementarea secvențială, cea paralelizată cu OpenMP și kernel-ul CUDA optimizat cu memorie constantă.

N (Size)	**CPU Secv.** *(naiv)*	**CPU Paralel** *(OpenMP)*	**GPU CUDA** *(Constant Mem)*	**Speedup** *(vs Seq)*
256	0.68 ms	12.23 ms	0.05 ms	13.6x
1024	5.67 ms	9.74 ms	0.15 ms	37.8x
4096	90.46 ms	37.68 ms	0.59 ms	153.3x
16384	**1448.45 ms**	**794.09 ms**	**2.33 ms**	**621.6x**

Tabela 4.1: Timpi de execuție Conv2D: Comparație Secvențial vs OpenMP vs CUDA

Observație: La $N = 16384$, implementarea CUDA este de aproximativ 340 de ori mai rapidă decât varianta CPU Paralel (OpenMP) și de 621 de ori mai rapidă decât cea secvențială.

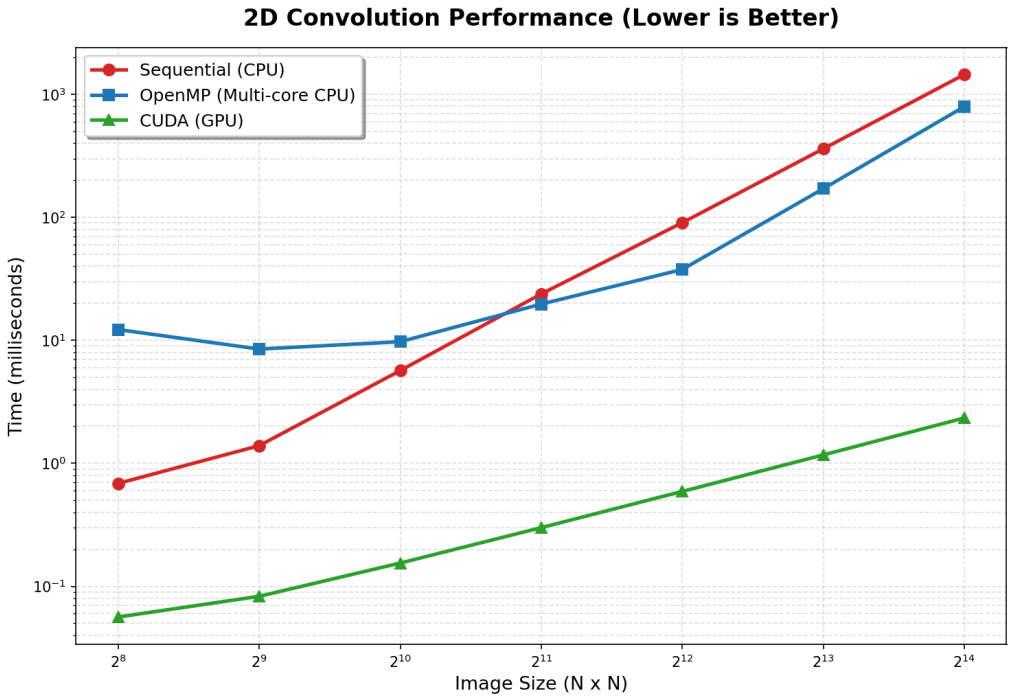

Figura 4.1: Grafic comparativ Conv2D (Scară Logaritmică).

4.2 Analiza Detaliată DFT

Deși DFT este un algoritm $O(N^2)$, paralelizarea pe GPU reduce timpul total semnificativ.

N (Size)	**CPU Secv.** *(naiv)*	**CPU Paralel** *(OpenMP)*	**GPU CUDA** *(Shared Mem)*	**Speedup** *(vs Seq)*
512	5.25 ms	9.13 ms	3.84 ms	1.36x
2048	85.77 ms	8.99 ms	3.84 ms	22.3x
8192	1372.78 ms	77.12 ms	4.17 ms	329.2x
32768	**21632.9 ms**	**586.46 ms**	**29.33 ms**	**737.5x**

Tabela 4.2: Timpi de execuție DFT: Comparație Secvențial vs OpenMP vs CUDA

Observație: Varianta secvențială la $N = 32768$ este practic inutilizabilă (21 secunde), în timp ce GPU-ul o rezolvă în timp real (29 ms).

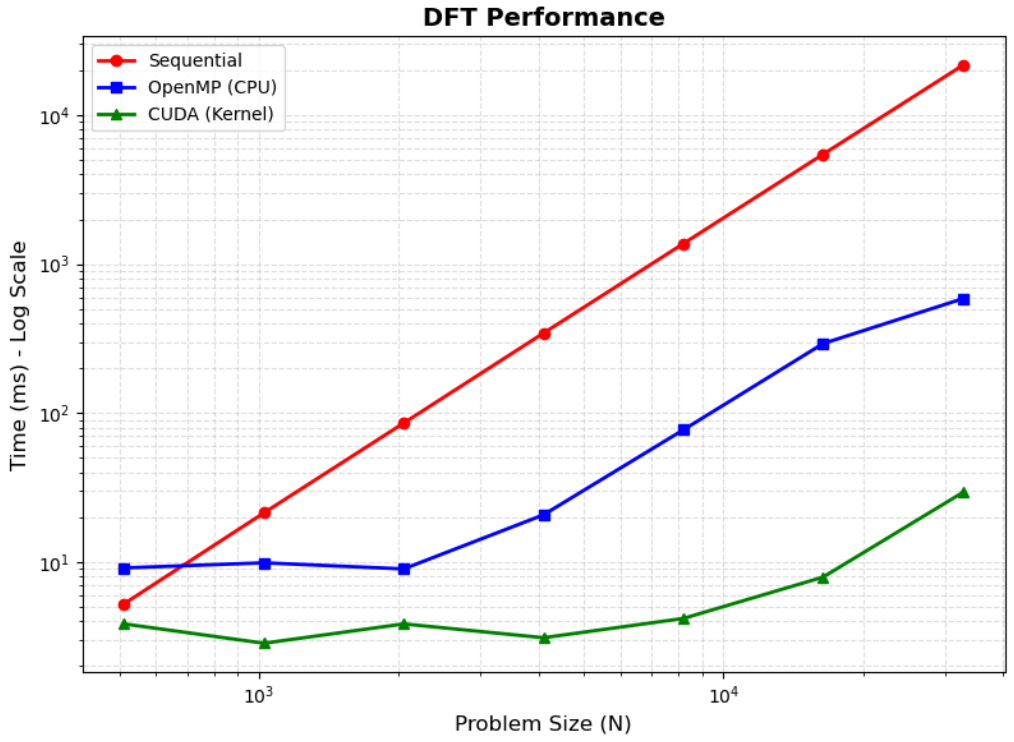

Figura 4.2: Performanța DFT: Accelerarea masivă la dimensiuni mari.

4.3 Analiza Detaliată GEMM (cuBLAS)

Pentru înmulțirea matricelor, comparația critică este între biblioteca optimizată pentru CPU (cBLAS) și cea pentru GPU (cuBLAS).

N (Size)	**CPU Secv.** *(naiv)*	**CPU Paralel** *(cBLAS)*	**GPU CUDA** *(cuBLAS)*	**Speedup** *(vs cBLAS)*
512	307.9 ms	0.82 ms	3.11 ms	0.26x
2048	52.7 s	32.22 ms	101.04 ms	0.31x
4096	N/A	270.90 ms	**132.20 ms**	**2.04x**
16384	N/A	17.9 s	**1.67 s**	**10.7x**

Tabela 4.3: Timpi de execuție GEMM: Comparație Secvențial vs cBLAS vs cuBLAS

Interpretare: Se observă clar "Punctul de Crossover" la $N = 4096$.

- Pentru $N < 4096$, cBLAS pe CPU este mai rapid datorită latenței zero de transfer.
- Pentru $N \geq 4096$, cuBLAS pe GPU devine dominant, fiind de peste 10 ori mai rapid decât cBLAS la dimensiunea maximă, demonstrând puterea de calcul brută a GPU-ului pentru matrice gigantice.

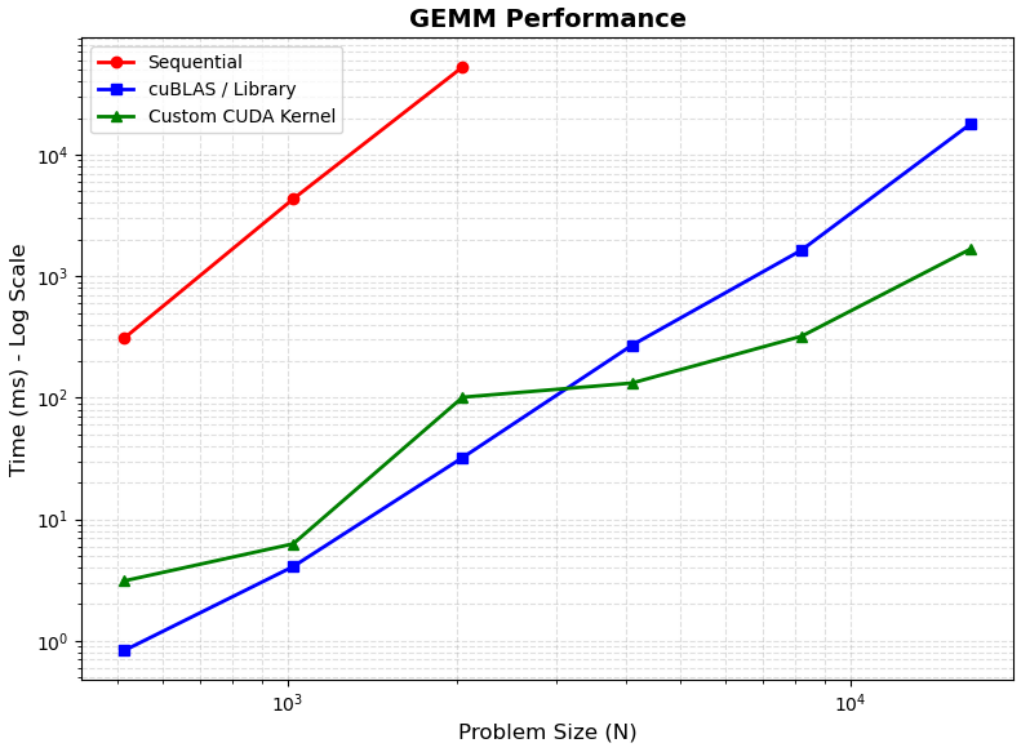

Figura 4.3: Grafic GEMM: Activarea cuBLAS la N mari depășește performanța CPU cBLAS.

Capitolul 5

Concluzii

Experimentele validează ipoteza că accelerarea GPU nu este liniară, ci depinde critic de: 1. **Dimensiunea Datelor:** Pentru N mic, costul transferului de memorie anulează beneficiile paralelizării. 2. **Biblioteci Optimizate:** În cazul GEMM, utilizarea **cuBLAS** este esențială pentru a depăși performanța cBLAS pe CPU la dimensiuni mari. 3. **Optimizarea Memoriei:** Pentru Conv2D, utilizarea memoriei constante a dus la cele mai spectaculoase rezultate (600x speedup).

Anexa A

Codul sursă și Scripturi

A.1 Codul sursă CUDA (.cu)

A.1.1 Convoluție 2D (conv2d_cuda.cu)

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <cuda_runtime.h>

#define CHECK_CUDA(x) \
  if((x) != cudaSuccess) { \
    std::cerr << "CUDA Error: " << cudaGetErrorString(x) << " at line " << __LINE__ << std::endl; \
    std::exit(EXIT_FAILURE); \
  }

// Define Constant Memory for the kernel filter
// 15x15 is a reasonable max size for standard convolutions
#define MAX_K 15
__constant__ float c_kernel[MAX_K * MAX_K];

__global__ void conv2d_constant_shared(const float* __restrict__ input,
                                       int H, int W, int K,
                                       float* __restrict__ output,
                                       int BLOCK)
{
    // Dynamic Shared Memory for Input Tile
    extern __shared__ float shmem[];
    float* tile = shmem;

    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int tileW = BLOCK + K - 1;
    int pad = K / 2;

    for (int i = ty; i < tileW; i += BLOCK) {
        for (int j = tx; j < tileW; j += BLOCK) {
            int r = blockIdx.y * BLOCK + i - pad;
            int c = blockIdx.x * BLOCK + j - pad;

            float val = 0.0f;
            if (r >= 0 && r < H && c >= 0 && c < W) {
```

```
                val = input[r * W + c];
            }
            tile[i * tileW + j] = val;
        }
    }

    __syncthreads();

    int row = blockIdx.y * BLOCK + ty;
    int col = blockIdx.x * BLOCK + tx;

    if (row < H && col < W) {
        float sum = 0.0f;
        for (int ki = 0; ki < K; ++ki) {
            // Pre-calculate index to help compiler
            int tile_idx_base = (ty + ki) * tileW + tx;
            int kernel_idx_base = ki * K;

            #pragma unroll
            for (int kj = 0; kj < K; ++kj) {
                // Read from Shared Mem (tile) and Constant Mem (c_kernel)
                sum += tile[tile_idx_base + kj] * c_kernel[kernel_idx_base + kj];
            }
        }
        output[row * W + col] = sum;
    }
}

int main(int argc, char** argv) {
    int H = 1024, W = 1024, K = 7, BLOCK = 16;
    if (argc >= 2) H = std::atoi(argv[1]);
    if (argc >= 3) W = std::atoi(argv[2]);
    if (argc >= 4) K = std::atoi(argv[3]);

    if (K > MAX_K) {
        std::cerr << "Error: K is larger than MAX_K (" << MAX_K << ")\n";
        return 1;
    }

    size_t img_bytes = H * W * sizeof(float);
    size_t ker_bytes = K * K * sizeof(float);

    std::vector<float> h_input(H*W, 1.0f);
    std::vector<float> h_kernel(K*K, 1.0f/(K*K));
    std::vector<float> h_output(H*W, 0.0f);

    float *d_in, *d_out;
    CHECK_CUDA(cudaMalloc(&d_in, img_bytes));
    CHECK_CUDA(cudaMalloc(&d_out, img_bytes));

    CHECK_CUDA(cudaMemcpy(d_in, h_input.data(), img_bytes, cudaMemcpyHostToDevice));

    CHECK_CUDA(cudaMemcpyToSymbol(c_kernel, h_kernel.data(), ker_bytes));
```

```
91
92      dim3 block(BLOCK, BLOCK);
93      dim3 grid((W + BLOCK - 1) / BLOCK, (H + BLOCK - 1) / BLOCK);
94
95      int tileW = BLOCK + K - 1;
96      size_t shared_mem_bytes = tileW * tileW * sizeof(float);
97
98      conv2d_constant_shared<<<grid, block, shared_mem_bytes>>>(d_in, H, W
        , K, d_out, BLOCK);
99      CHECK_CUDA(cudaDeviceSynchronize());
100
101     cudaEvent_t start, stop;
102     cudaEventCreate(&start); cudaEventCreate(&stop);
103     cudaEventRecord(start);
104
105     conv2d_constant_shared<<<grid, block, shared_mem_bytes>>>(d_in, H, W
        , K, d_out, BLOCK);
106
107     cudaEventRecord(stop);
108     cudaEventSynchronize(stop);
109
110     float ms = 0.0f;
111     cudaEventElapsedTime(&ms, start, stop);
112
113     std::cout << ms << std::endl;
114
115     cudaFree(d_in); cudaFree(d_out);
116     cudaEventDestroy(start); cudaEventDestroy(stop);
117     return 0;
118 }
```

A.1.2 DFT (dft_cuda.cu)

```
1  #include <iostream>
2  #include <vector>
3  #include <cmath>
4  #include <cuda_runtime.h>
5
6  #define CHECK_CUDA(x) \
7      if((x) != cudaSuccess) { \
8          std::cerr << "CUDA Error: " << cudaGetErrorString(x) \
9                    << " at line " << __LINE__ << std::endl; \
10         exit(EXIT_FAILURE); \
11     }
12
13 #define BLOCK 256
14 #define PI 3.14159265358979323846f
15
16 // Each thread computes one output frequency k
17 __global__ void dft_cuda_shared(const float* __restrict__ in_real,
18                                 const float* __restrict__ in_imag,
19                                 float* __restrict__ out_real,
20                                 float* __restrict__ out_imag,
21                                 int N)
22 {
23     __shared__ float sh_real[BLOCK];
24     __shared__ float sh_imag[BLOCK];
```

```
25
26      int k = blockIdx.x * blockDim.x + threadIdx.x;
27      if (k >= N) return;
28
29      float sum_real = 0.0f;
30      float sum_imag = 0.0f;
31
32      float angle_base = -2.0f * PI * k / N;
33
34      for (int tile = 0; tile < N; tile += BLOCK) {
35
36          int idx = tile + threadIdx.x;
37          if (idx < N) {
38              sh_real[threadIdx.x] = in_real[idx];
39              sh_imag[threadIdx.x] = in_imag[idx];
40          } else {
41              sh_real[threadIdx.x] = 0.0f;
42              sh_imag[threadIdx.x] = 0.0f;
43          }
44
45          __syncthreads();
46
47          int tileSize = min(BLOCK, N - tile);
48          for (int n = 0; n < tileSize; n++) {
49              float angle = angle_base * (tile + n);
50              float s, c;
51              sincosf(angle, &s, &c);
52
53              sum_real += sh_real[n] * c - sh_imag[n] * s;
54              sum_imag += sh_real[n] * s + sh_imag[n] * c;
55          }
56
57          __syncthreads();
58      }
59
60      out_real[k] = sum_real;
61      out_imag[k] = sum_imag;
62  }
63  int main(int argc, char** argv) {
64      int N = 4096;
65      if (argc >= 2) N = std::atoi(argv[1]);
66      size_t bytes = N * sizeof(float);
67
68      std::vector<float> h_real(N, 1.0f);
69      std::vector<float> h_imag(N, 0.0f);
70      std::vector<float> h_out_real(N), h_out_imag(N);
71
72      float *d_real, *d_imag, *d_out_real, *d_out_imag;
73      CHECK_CUDA(cudaMalloc(&d_real, bytes));
74      CHECK_CUDA(cudaMalloc(&d_imag, bytes));
75      CHECK_CUDA(cudaMalloc(&d_out_real, bytes));
76      CHECK_CUDA(cudaMalloc(&d_out_imag, bytes));
77
78      CHECK_CUDA(cudaMemcpy(d_real, h_real.data(), bytes,
    cudaMemcpyHostToDevice));
79      CHECK_CUDA(cudaMemcpy(d_imag, h_imag.data(), bytes,
    cudaMemcpyHostToDevice));
80
```

```
    dim3 block(BLOCK);
    dim3 grid((N + BLOCK - 1) / BLOCK);

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaEventRecord(start);
    dft_cuda_shared<<<grid, block>>>(d_real, d_imag, d_out_real,
    d_out_imag, N);
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);

    float ms;
    cudaEventElapsedTime(&ms, start, stop);

    CHECK_CUDA(cudaMemcpy(h_out_real.data(), d_out_real, bytes,
    cudaMemcpyDeviceToHost));
    CHECK_CUDA(cudaMemcpy(h_out_imag.data(), d_out_imag, bytes,
    cudaMemcpyDeviceToHost));

    double energy = 0.0;
    for (int i = 0; i < N; i++)
        energy += h_out_real[i] * h_out_real[i] + h_out_imag[i] *
    h_out_imag[i];

    std::cout << ms << std::endl;

    cudaFree(d_real);
    cudaFree(d_imag);
    cudaFree(d_out_real);
    cudaFree(d_out_imag);
}
```

A.1.3 Înmulțirea Matricelor (gemm_cuda.cu)

```
#include <iostream>
#include <vector>
#include <cuda_runtime.h>
#include <cublas_v2.h>

#define BLOCK 32

#define CUDA_CHECK(x) \
    if((x) != cudaSuccess) { \
        std::cout << "CUDA Error: " << cudaGetErrorString(x) << " at
    line " << __LINE__ << std::endl; exit(1); \
    }

#define CUBLAS_CHECK(x) \
    if((x) != CUBLAS_STATUS_SUCCESS) { \
        std::cout << "cuBLAS Error at line " << __LINE__ << std::endl;
    exit(1); \
    }

// Shared-memory GEMM kernel
```

```
__global__ void gemm_shared(const float* A, const float* B, float* C,
    int N) {
    __shared__ float As[BLOCK][BLOCK];
    __shared__ float Bs[BLOCK][BLOCK];

    int row = blockIdx.y * BLOCK + threadIdx.y;
    int col = blockIdx.x * BLOCK + threadIdx.x;

    float sum = 0.0f;

    for(int t = 0; t < N; t += BLOCK) {
        if(row < N && t + threadIdx.x < N)
            As[threadIdx.y][threadIdx.x] = A[row*N + t + threadIdx.x];
        else
            As[threadIdx.y][threadIdx.x] = 0.0f;

        if(col < N && t + threadIdx.y < N)
            Bs[threadIdx.y][threadIdx.x] = B[(t + threadIdx.y)*N + col];
        else
            Bs[threadIdx.y][threadIdx.x] = 0.0f;

        __syncthreads();

        for(int k = 0; k < BLOCK; k++)
            sum += As[threadIdx.y][k] * Bs[k][threadIdx.x];

        __syncthreads();
    }

    if(row < N && col < N)
        C[row*N + col] = sum;
}

int main(int argc, char** argv) {
    int N = 4096;
    if (argc >= 2) N = std::atoi(argv[1]);

    size_t bytes = N*N*sizeof(float);
    std::vector<float> hA(N*N, 1.0f), hB(N*N, 1.0f), hC(N*N, 0.0f);

    float *dA, *dB, *dC;
    CUDA_CHECK(cudaMalloc(&dA, bytes));
    CUDA_CHECK(cudaMalloc(&dB, bytes));
    CUDA_CHECK(cudaMalloc(&dC, bytes));

    CUDA_CHECK(cudaMemcpy(dA, hA.data(), bytes, cudaMemcpyHostToDevice))
    ;
    CUDA_CHECK(cudaMemcpy(dB, hB.data(), bytes, cudaMemcpyHostToDevice))
    ;

    float ms = 0.0f;

    if(N < 2048) {

        dim3 threads(BLOCK,BLOCK);
        dim3 blocks((N+BLOCK-1)/BLOCK, (N+BLOCK-1)/BLOCK);

        cudaEvent_t start, stop;
```

```
        cudaEventCreate(&start);
        cudaEventCreate(&stop);
        cudaEventRecord(start);

        gemm_shared<<<blocks, threads>>>(dA,dB,dC,N);
        cudaEventRecord(stop);
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&ms,start,stop);
    } else {

        cublasHandle_t handle;
        CUBLAS_CHECK(cublasCreate(&handle));

        float alpha = 1.0f;
        float beta  = 0.0f;

        cudaEvent_t start, stop;
        cudaEventCreate(&start);
        cudaEventCreate(&stop);
        cudaEventRecord(start);


        CUBLAS_CHECK(cublasSgemm(handle,
            CUBLAS_OP_N, CUBLAS_OP_N,
            N, N, N,
            &alpha,
            dB, N,
            dA, N,
            &beta,
            dC, N));

        cudaEventRecord(stop);
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&ms,start,stop);

        cublasDestroy(handle);
    }

    CUDA_CHECK(cudaMemcpy(hC.data(), dC, bytes, cudaMemcpyDeviceToHost))
    ;

    double sum = 0.0;
    for(float x : hC) sum += x;

    std::cout << ms << std::endl;

    cudaFree(dA);
    cudaFree(dB);
    cudaFree(dC);

    return 0;
}
```

A.1.4 Scripturi Bash de Automatizare

```
#!/bin/bash
set -e
```

```

OUT="linear_results.csv"
echo "algo,version,N,time_ms" > "$OUT"

DFT_BIN="../DFT"
GEMM_BIN="../GEMM"

DFT_SIZES=(512 1024 2048 4096 8192 16384 32768)
GEMM_SIZES=(512 1024 2048 4096 8192 16384)

# DFT
for N in "${DFT_SIZES[@]}"; do
  echo "Running DFT N=$N"

  # sequential
  TIME=$($DFT_BIN/dft_secv $N)
  echo "DFT,seq,$N,$TIME" >> "$OUT"

  # omp
  TIME=$($DFT_BIN/dft_cpu $N)
  echo "DFT,omp,$N,$TIME" >> "$OUT"

  # cuda
  TIME=$($DFT_BIN/dft_cuda $N)
  echo "DFT,cuda,$N,$TIME" >> "$OUT"
  echo "----------------------">> "$OUT"
done

# GEMM
for N in "${GEMM_SIZES[@]}"; do
  echo "Running GEMM N=$N"

  if [ $N -le 2048 ]; then
    TIME=$($GEMM_BIN/gemm_secv $N)
    echo "GEMM,seq,$N,$TIME" >> "$OUT"
  fi

  TIME=$($GEMM_BIN/gemm_cpu $N)
  echo "GEMM,omp,$N,$TIME" >> "$OUT"

  TIME=$($GEMM_BIN/gemm_cuda $N)
  echo "GEMM,cuda,$N,$TIME" >> "$OUT"
  echo "----------------------">> "$OUT"
done
```

```

#!/bin/bash
set -e

OUT="conv2d_results.csv"
echo "algo,version,N,time_ms" > "$OUT"

# Adjust this path to where your executables are located
CONV_BIN="../Convolutie2D"

SIZES=(256 512 1024 2048 4096 8192 16384)

echo "Starting Conv2d Benchmarks..."

```

```
for N in "${SIZES[@]}"; do
  echo "Running Conv2d N=$N"



  TIME=$($CONV_BIN/conv2d_secv $N)
  echo "CONV,seq,$N,$TIME" >> "$OUT"


  # 2. OpenMP
  TIME=$($CONV_BIN/conv2d_cpu $N)
  echo "CONV,omp,$N,$TIME" >> "$OUT"

  # 3. CUDA
  TIME=$($CONV_BIN/conv2d_cuda $N)
  echo "CONV,cuda,$N,$TIME" >> "$OUT"

  # Separator for readability (filtered out by python script later)
  echo "----------------------" >> "$OUT"
done

echo "Done. Results saved to $OUT"
```