Tema proiect 2 PPD – 03.01.2022

Documentatie proiect 2 - CUDA

**Gaussian Blur:**

Pentru o matrice F de dimensiuni NxM se cere transformarea acesteia aplicand functia de convolutie , unde W de dimensiuni nxm este matricea suport. Implementarea trebuie sa fie in Java secvential si paralel si in C++ secvential si paralel avand matricile fie initializate static, fie dinamic. La final trebuie sa realizam tabele cu timpii de executie pentru fiecare setup si cateva grafice pentru a compara performantele intre limbaje.

De exemplu, pentru matricea de forma:

1 2 3 4 5

6 7 8 9 0

7 8 9 0 1

8 9 0 1 2

9 0 1 2 3

Si kernelul

* 1. 0.2 0.3
  2. 0.3 0.4
  3. 0.4 0.5

Pentru efectuarea calculelor, in continuare vom adauga o bordura egala cu cea mai apropiata extrema a matricei.

1 1 2 3 4 5 5

1 1 2 3 4 5 5

6 6 7 8 9 0 0

7 7 8 9 0 1 1

8 8 9 0 1 2 2

9 9 0 1 2 3 3

9 9 0 1 2 3 3

Pentru calcularea elementului de pe pozitia (2, 1), adica elemental cu valoarea 6, vom avea de efectuat urmatorul calcul:

1 \* 0.1 + 1 \* 0.2 + 2 \* 0.3 + 6 \* 0.2 + 6 \* 0.3 + 6 \* 0.4 + 7 \* 0.3 + 7 \* 0.4 + 8 \* 0.5

Deoarece functia de convolutie acceseaza elemente din afara matricei, am hotarat sa aplic o serie de verificari astfel incat elementul returnat sa fie cel pe care il cautam. Astfel, daca elementul accesat se afla pe:

1. i < 0, j < 0 -> F[0][0]
2. i < 0, j este valid -> F[0][j]
3. i < 0, j >= M -> F[0][M – 1]
4. i este valid, j < 0 -> F[i][0]
5. i este valid, j >= M -> F[i][M – 1]
6. i >= N, j < 0 -> F[N – 1][0]
7. i >= N, j valid -> F[N – 1][j]
8. i >= N, j >= M -> F[N – 1][M – 1]
9. i valid, j valid -> F[i][j]

Pentru implementarea paralela am folosit threaduri care procesau matricea F pe chunkuri astfel:

1. Citim matricea initiala si kernel-ul din fisier.
2. Impartirea in N threaduri.
3. Calcularea pentru fiecare thread a pozitiei de start si de final.
4. Iterarea prin fiecare element al intervalului calculat si calcularea valorii output corespunzatoare acestei pozitii.
5. Adaugarea valorii calculate intr-o noua matrice.
6. Asteptarea tuturor threadurilor sa isi termine calculul.
7. Scrierea matricei calculate intr-un fisier.

Implementarea am facut-o folosind libraria CUDA din C++. Am folosit **cudaMallocManaged** pentru a aloca matricile citite in memoria partajata de CPU si GPU. CUDA ne ofera directive care specifica daca codul ruleaza pe GPU sau CPU cum ar fi “**\_\_global\_\_**” care se apeleaza de pe CPU si se executa pe GPU si “**\_\_device\_\_**” care se apeleaza de pe GPU si se executa pe GPU. La final, pentru a putea salva matricea rezultat in fisier a fost nevoie sa asteptam terminarea executiei threadurilor de pe GPU, iar pentru asta am folosit **cudaDeviceSynchronize**.

Pentru apelarea functiilor de pe CPU care ruleaza pe GPU se dau ca parametri numarul de blocuri de threaduri si numarul de threaduri din fiecare bloc.

Pentru determinarea elementelor care sunt procesate de fiecare thread, am calculat urmatoarele:

threadID = blockDim \* blockIdx + threadIdx (numarul threadului care se executa) (determina daca threadul curent trebuie sa calculeze un element nedistribuit uniform)

start = cat \* threadIdx + min(threadID, rest) (pozitia de inceput din matricea liniarizata de unde incepe procesarea), unde rest = numarul elementelor care nu sunt uniform distribuite threadurilor, min(threadID, rest) reprezinta numarul de elemente procesate care nu au fost uniform distribuite pana la threadul curent