**Documentație**

**Introducere**

Acest program implementează un algoritm de convoluție pe o matrice de intrare, folosind atât un mod secvențial, cât și un mod paralel (cu CUDA) pentru a accelera procesul. Programul aplică un filtru de convoluție (matricea *convMat*) pe o matrice de intrare *(mat*), folosind tehnici de programare pe GPU pentru a îmbunătăți performanța. Acesta permite procesarea fișierelor de intrare și compararea rezultatelor cu un fișier de referință.

**Structura Programului**

1. **Functii**
2. **readInput()**
   * Descriere: Citeste matricea de intrare și matricea de filtrare (kernel) dintr-un fișier. Filtrul este o matrice 3x3 care poate fi ajustată în funcție de nevoile aplicației
   * Parametri: N/A (Citește dintr-un fișier predefinit *date.txt)*
   * Funcționalitate:
     + Deschide fișierul *date.txt*, citește dimensiunile matricei și valorile acesteia.
     + Definește o matrice de filtrare fixă (3x3).
     + Încarcă datele în variabilele globale *mat* și *convMat.*

void readInput() {  
 ifstream inputFile("date.txt");  
 if (!inputFile) {  
 cerr << "Error opening file." << endl;  
 return;  
 }  
 mat.resize(N, vector<int>(M));  
 for (int i = 0; i < N; ++i) {  
 for (int j = 0; j < M; ++j) {  
 inputFile >> mat[i][j];  
 }  
 }  
 // 1 0 1  
 // 0 2 0  
 // -1 0 -1  
 convMat.resize(n, vector<int>(n));  
 convMat.assign({{1, 0, 1}, {0, 2, 0}, {-1, 0, -1}});  
  
 inputFile.close();  
}

1. **writeOutput()**
   * Descriere: Scrie matricea rezultată într-un fișier de ieșire numit *output.txt*.
   * Parametri: N/A
   * Funcționalitate:
     + Salvează matricea procesată în fișierul *output.txt.*

void writeOutput() {  
 ofstream outputFile("output.txt");  
 if (!outputFile) {  
 cerr << "Error opening file." << endl;  
 return;  
 }  
 for (int i = 0; i < N; ++i) {  
 for (int j = 0; j < M; ++j) {  
 outputFile << mat[i][j];  
 if (j < M - 1) {  
 outputFile << " ";  
 }  
 }  
 if (i < N - 1) {  
 outputFile << endl;  
 }  
 }  
 outputFile.close();  
}

1. **secvential()**
   * Descriere: Aplica algoritmul de convoluție pe matricea de intrare într-un mod secvențial. Fiecare element din matricea rezultată este calculat prin aplicarea unui filtru de convoluție pe vecinătatea corespunzătoare a fiecărui element din matricea de intrare. Aceasta este o metodă de procesare în care fiecare element este calculat individual, fără a utiliza paralelism.
   * Parametri: N/A (Valorile pentru dimensiunile matricei și kernel-ul sunt setate în mod global).
   * Funcționalitate:
     + Pentru fiecare rând al matricei de intrare, se creează un vector auxiliar *auxRow*, care stochează valorile corespunzătoare pentru o fereastră de dimensiune n în jurul elementului curent.
     + Pentru fiecare element din matrice, se calculează suma produselor elementelor vecine din matricea de intrare și elementele corespunzătoare ale matricei de convoluție.
     + Rezultatul convoluției este stocat în matricea de intrare (*mat*).
     + Această metodă este secvențială, adică fiecare calcul pentru un element este făcut unul câte unul, fără utilizarea unui proces de paralelizare.

void secvential() {  
  
 vector<int> auxRow(M + n - 1);  
  
 // calculate new values for border cells and save them in separate vectors  
 for (int i = 0; i < N; ++i) {  
 for (int j = -n / 2; j < M + n / 2; ++j) {  
 int x = max(0, min(i, N - 1));  
 int y = max(0, min(j, M - 1));  
 auxRow[j + n / 2] = mat[x][y];  
 }  
 // calculate new values for non-border cells and save them directly in the matrix  
 for (int j = 0; j < M; ++j) {  
 int sum = 0;  
 for (int i1 = 0; i1 < n; ++i1) {  
 for (int j1 = 0; j1 < n; ++j1) {  
 int x = i - n / 2 + i1;  
 int y = j - n / 2 + j1;  
 if (x >= 0 && y >= 0 && x < N && y < M) {  
 sum += auxRow[y + n / 2] \* convMat[i1][j1];  
 }  
 }  
 }  
 mat[i][j] = sum;  
 }  
 }  
}

1. **linii()**
   * Descriere: Această funcție este responsabilă pentru efectuarea convoluției matricei de intrare folosind CUDA. Se ocupă cu alocarea și copierea datelor pe GPU, executarea kernel-ului și copierea rezultatelor înapoi pe CPU.
   * Parametri: N/A (Valorile pentru dimensiunile matricei și kernel-ul sunt setate în mod global).
   * Funcționalitate:
     + Alocă memorie pe GPU pentru matricea de intrare, kernel și matricea rezultat.
     + Transferă datele de la CPU la GPU.
     + Lansează kernel-ul CUDA *convolveKernel* pentru a efectua convoluția.
     + Copiază rezultatele înapoi pe CPU și le plasează în matricea originală.
     + Eliberează memoria GPU după finalizarea procesării.

void linii() {  
 size\_t matSize = N \* M \* sizeof(int);  
 size\_t convMatSize = n \* n \* sizeof(int);  
   
 // Allocate device memory  
 int \*d\_mat, \*d\_convMat, \*d\_result;  
 CHECK\_CUDA(cudaMalloc(&d\_mat, matSize));  
 CHECK\_CUDA(cudaMalloc(&d\_convMat, convMatSize));  
 CHECK\_CUDA(cudaMalloc(&d\_result, matSize));  
   
 // Flatten host matrices for device transfer  
 vector<int> flatMat(N \* M);  
 vector<int> flatConvMat(n \* n);  
 for (int i = 0; i < N; ++i) {  
 for (int j = 0; j < M; ++j) {  
 flatMat[i \* M + j] = mat[i][j];  
 }  
 }  
 for (int i = 0; i < n; ++i) {  
 for (int j = 0; j < n; ++j) {  
 flatConvMat[i \* n + j] = convMat[i][j];  
 }  
 }  
  
 // Copy data to device  
 CHECK\_CUDA(cudaMemcpy(d\_mat, flatMat.data(), matSize, cudaMemcpyHostToDevice));  
 CHECK\_CUDA(cudaMemcpy(d\_convMat, flatConvMat.data(), convMatSize, cudaMemcpyHostToDevice));  
  
 // Define grid and block dimensions  
 dim3 blockSize(16, 16);  
 dim3 gridSize((M + 15) / 16, (N + 15) / 16);  
  
 // Launch the kernel  
 convolveKernel<<<gridSize, blockSize>>>(d\_mat, d\_convMat, d\_result, N, M, n);  
 CHECK\_CUDA(cudaGetLastError());  
  
 // Copy results back to host  
 vector<int> flatResult(N \* M);  
 CHECK\_CUDA(cudaMemcpy(flatResult.data(), d\_result, matSize, cudaMemcpyDeviceToHost));  
  
 // Reshape flatResult into 2D result matrix  
 for (int i = 0; i < N; ++i) {  
 for (int j = 0; j < M; ++j) {  
 mat[i][j] = flatResult[i \* M + j];  
 }  
 }  
  
 // Free device memory  
 CHECK\_CUDA(cudaFree(d\_mat));  
 CHECK\_CUDA(cudaFree(d\_convMat));  
 CHECK\_CUDA(cudaFree(d\_result));  
}

1. **convolveKernel (Kernel CUDA)**
   * Descriere: Kernel-ul CUDA care efectuează convoluția pe GPU.
   * Parametri:
     + d\_mat: Pointer către matricea de intrare pe GPU.
     + d\_convMat: Pointer către kernel-ul de filtrare pe GPU.
     + d\_result: Pointer către matricea rezultat de pe GPU.
     + N: Numărul de rânduri din matricea de intrare.
     + M: Numărul de coloane din matricea de intrare.
     + n: Dimensiunea kernel-ului (3 în acest caz).
   * Funcționalitate:
     + Fiecare thread calculează suma elementelor din matricea de intrare care corespund cu kernel-ul, efectuând operația de convoluție.
     + Verifică limitele matricei de intrare pentru a evita accesul la elemente din afara acesteia.

\_\_global\_\_ void convolveKernel(int \*d\_mat, int \*d\_convMat, int \*d\_result, int N, int M, int n) {  
 int row = blockIdx.y \* blockDim.y + threadIdx.y;  
 int col = blockIdx.x \* blockDim.x + threadIdx.x;  
  
 if (row < N && col < M) {  
 int sum = 0;  
 for (int i = 0; i < n; ++i) {  
 for (int j = 0; j < n; ++j) {  
 int x = row - n / 2 + i;  
 int y = col - n / 2 + j;  
 if (x >= 0 && y >= 0 && x < N && y < M) {  
 sum += d\_mat[x \* M + y] \* d\_convMat[i \* n + j];  
 }  
 }  
 }  
 d\_result[row \* M + col] = sum;  
 }  
}

1. **Main**

* Descriere: Punctul de intrare al programului, care primește parametrii de comandă pentru dimensiunile matricei și metoda de procesare (secvențial sau paralel).
* Parametri: *argc, argv*: Parametrii de comandă (dimensiuni matrice, numele functiei, comparare, etc.).
* Funcționalitate:
  + Citește dimensiunile matricei din argumentele de comandă.
  + Apelează funcția *readInput()* pentru a încărca matricea de intrare și kernel-ul.
  + În funcție de modul ales (*linii*), lansează funcția *linii()* pentru a procesa matricea pe GPU.
  + Afișează timpul de execuție în milisecunde și scrie rezultatele în fișierul de ieșire *output.txt*.

int main(int argc, char \*argv[]) {  
 if (argc < 4) {  
 cerr << "Usage: " << argv[0] << " <matrix\_rows> <matrix\_cols> <compare-1/0>" << endl;  
 return 1;  
 }  
  
 N = stoi(argv[1]);  
 M = stoi(argv[2]);  
 string functionName = argv[3];  
 n = 3;  
 compare = stoi(argv[4]);  
  
 auto start = chrono::high\_resolution\_clock::now();  
 readInput();  
  
 if (functionName == "secvential") {  
 secvential();  
 } else if (functionName == "linii") {  
 linii();  
 } else {  
 cerr << "Invalid function name. Use one of: secvential, linii" << endl;  
 return 1;  
 }  
  
 auto stop = chrono::high\_resolution\_clock::now();  
 chrono::duration<double, std::micro> duration = stop - start;  
 double microseconds = duration.count();  
 double milliseconds = microseconds / 1000;  
  
 cout << milliseconds << " milliseconds" << endl;

writeOutput();  
  
 if (compare == 1) {  
 if (N != 100 && N != 1000) {  
 cout << "Invalid matrix size" << endl;  
 return 1;  
 }  
  
 ifstream outputFile("output.txt");  
 ifstream expectedFile = ifstream("expected" + to\_string(N) + "x" + to\_string(N) + ".txt");  
  
 string outputLine, expectedLine;  
 while (getline(outputFile, outputLine) && getline(expectedFile, expectedLine)) {  
 outputLine.erase(outputLine.find\_last\_not\_of(" \n\r\t") + 1);  
 expectedLine.erase(expectedLine.find\_last\_not\_of(" \n\r\t") + 1);  
 if (outputLine != expectedLine) {  
 cout << "Output is not correct" << endl;  
 return 1;  
 }  
 }  
 cout << "Output is correct" << endl;  
 }  
  
 return 0;  
}

1. **Descrierea Algoritmului**

**Mod Secvențial:**

Se aplică filtrul de convoluție pe fiecare element al matricei, fără a folosi paralelism.

**Mod Paralel cu CUDA:**

1. Citire Date pe GPU:
   * Matricele de intrare și kernel-ul sunt transferate din memoria principală (CPU) în memoria GPU.
2. Calculul Convoluției:
   * Kernel-ul CUDA *convolveKernel* este lansat pe GPU. Fiecare thread este responsabil pentru calcularea unui element din matricea rezultată folosind o fereastră de 3x3 din matricea de intrare și matricea de filtrare.
   * Convoluția se efectuează paralel pentru toate elementele matricei de intrare.
3. Transferul Rezultatelor:
   * După executarea kernel-ului, rezultatele sunt transferate înapoi în memoria CPU și sunt salvate într-un fișier de ieșire.
4. **Optimizări**

* Utilizarea CUDA pentru paralelizarea calculelor de convoluție, permițând astfel procesarea rapidă a matricei chiar și pentru dimensiuni mari.
* Alocarea eficientă a memoriei pe GPU: Programul alocă doar cantitatea de memorie necesară pentru matricele de intrare, kernel și rezultate.
* Împărțirea sarcinilor între threaduri: Fiecare thread este responsabil pentru un subset de date, astfel încât procesarea să fie realizată simultan pe multiple nuclee ale GPU-ului.

1. **Performanță**

* Timp de execuție: Programul calculează și afișează timpul de execuție al procesării în milisecunde.
* Comparare Performanță: Timpul de execuție pentru procesarea pe GPU este semnificativ mai rapid comparativ cu implementările secvențiale pe CPU, în special pentru matrice de dimensiuni mari.

1. **Instrucțiuni de Utilizare**
2. **Compilare**:
   * Utilizați comanda de mai jos pentru a compila programul cu CUDA:

**nvcc -o program.exe program.cu**

sau

**nvcc -o program.exe main.cu -I"C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v12.6/include" -L"C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v12.6/lib/x64" -lcudart**

1. **Rulare**:
   * Rulați programul utilizând comanda:

php

**./program.exe <număr\_linii> <număr\_coloane> <nume\_funcție> <mod\_comparare>**

* + Exemplu:

**./program 100 100 secvential 1**

Aceasta va procesa o matrice de 100x100 folosind convoluția secvențială și va compara rezultatul cu un fișier de referință (dacă compare == 1).

sau

**./program 1000 1000 linii 1**

Aceasta va procesa o matrice de 1000x1000 folosind convoluția paralelă prin CUDA și va compara rezultatul cu un fișier de referință (dacă compare == 1).

1. **Cazuri de Testare**

|  |  |  |  |
| --- | --- | --- | --- |
| Matrix Rows | Matrix Cols | Tip Convolutie | Timp executie mediu |
| 100 | 100 | secvential | 8.29947ms |
| 1000 | 1000 | secvential | 814.9318ms |
| 100 | 100 | linii (CUDA) | 93.2513ms |
| 1000 | 1000 | linii (CUDA) | 717.4479ms |

Putem observa că secvențial, pe o matrice de dimensiuni mai mici (100x100) avem timpuri de execuție mai bune decât execuția paralelă CUDA, pentru că un block are dimensiunea 16x16 (256 thread-uri), iar secvențial sunt mai ușor de parcurs, dar pe o matrice de dimensiuni mai mari (1000x1000), convoluția aplicată paralel prin CUDA este mai eficientă cu aproape 100ms.