#include <stdio.h>

#include <cuda.h>

// Kernel pour le produit de convolution 1D

\_\_global\_\_ void conv1D(int n, float \*x, int p, float \*h, float \*y) {

    int idx = threadIdx.x + blockIdx.x \* blockDim.x; // Calcul de l'index global

    if (idx < n) {

        float sum = 0.0;

        for (int k = max(0, idx - p + 1); k <= idx; k++) {

            sum += x[k] \* h[idx - k];

        }

        y[idx] = sum; // Stocke le résultat dans le vecteur de sortie

    }

}

// Kernel pour le produit de convolution 2D

\_\_global\_\_ void conv2D(int m, int n, float \*x, int p, int q, float \*h, float \*y) {

    int i = blockIdx.y \* blockDim.y + threadIdx.y; // Index ligne

    int j = blockIdx.x \* blockDim.x + threadIdx.x; // Index colonne

    if (i < m && j < n) {

        float sum = 0.0;

        for (int k = max(0, i - p + 1); k <= i; k++) {

            for (int l = max(0, j - q + 1); l <= j; l++) {

                sum += x[k \* n + l] \* h[(i - k) \* q + (j - l)];

            }

        }

        y[i \* n + j] = sum; // Stocke le résultat dans la matrice de sortie

    }

}

int main() {

    // Dimensions du problème pour l'exemple

    const int n = 10; // Taille du vecteur 1D

    const int m = 4, nc = 4; // Dimensions de la matrice 2D

    const int p = 3, q = 3; // Dimensions des filtres

    // Allocation mémoire sur l'hôte

    float h\_x1D[n] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // Vecteur d'entrée

    float h\_h1D[p] = {1, 0, -1}; // Filtre 1D

    float h\_y1D[n] = {0}; // Résultat 1D

    float h\_x2D[m \* nc] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}; // Matrice d'entrée

    float h\_h2D[p \* q] = {0, 1, 0, 1, -4, 1, 0, 1, 0}; // Filtre 2D

    float h\_y2D[m \* nc] = {0}; // Résultat 2D

    // Allocation mémoire sur le device

    float \*d\_x1D, \*d\_h1D, \*d\_y1D;

    float \*d\_x2D, \*d\_h2D, \*d\_y2D;

    cudaMalloc(&d\_x1D, n \* sizeof(float));

    cudaMalloc(&d\_h1D, p \* sizeof(float));

    cudaMalloc(&d\_y1D, n \* sizeof(float));

    cudaMalloc(&d\_x2D, m \* nc \* sizeof(float));

    cudaMalloc(&d\_h2D, p \* q \* sizeof(float));

    cudaMalloc(&d\_y2D, m \* nc \* sizeof(float));

    // Copie des données sur le device

    cudaMemcpy(d\_x1D, h\_x1D, n \* sizeof(float), cudaMemcpyHostToDevice);

    cudaMemcpy(d\_h1D, h\_h1D, p \* sizeof(float), cudaMemcpyHostToDevice);

    cudaMemcpy(d\_x2D, h\_x2D, m \* nc \* sizeof(float), cudaMemcpyHostToDevice);

    cudaMemcpy(d\_h2D, h\_h2D, p \* q \* sizeof(float), cudaMemcpyHostToDevice);

    // Lancement des kernels

    dim3 block1D(256);

    dim3 grid1D((n + block1D.x - 1) / block1D.x);

    conv1D<<<grid1D, block1D>>>(n, d\_x1D, p, d\_h1D, d\_y1D);

    dim3 block2D(16, 16);

    dim3 grid2D((nc + block2D.x - 1) / block2D.x, (m + block2D.y - 1) / block2D.y);

    conv2D<<<grid2D, block2D>>>(m, nc, d\_x2D, p, q, d\_h2D, d\_y2D);

    // Copie des résultats vers l'hôte

    cudaMemcpy(h\_y1D, d\_y1D, n \* sizeof(float), cudaMemcpyDeviceToHost);

    cudaMemcpy(h\_y2D, d\_y2D, m \* nc \* sizeof(float), cudaMemcpyDeviceToHost);

    // Affichage des résultats

    printf("Résultat de la convolution 1D :\n");

    for (int i = 0; i < n; i++) {

        printf("%.2f ", h\_y1D[i]);

    }

    printf("\n\n");

    printf("Résultat de la convolution 2D :\n");

    for (int i = 0; i < m; i++) {

        for (int j = 0; j < nc; j++) {

            printf("%.2f ", h\_y2D[i \* nc + j]);

        }

        printf("\n");

    }

    // Libération de la mémoire

    cudaFree(d\_x1D);

    cudaFree(d\_h1D);

    cudaFree(d\_y1D);

    cudaFree(d\_x2D);

    cudaFree(d\_h2D);

    cudaFree(d\_y2D);

    return 0;

}

#include <stdio.h>

#include <stdlib.h>

#define NROW 100 // Nombre de lignes de la matrice d'entrée

#define NCOL 100 // Nombre de colonnes de la matrice d'entrée

#define NRH 10 // Nombre de lignes du filtre

#define NCH 10 // Nombre de colonnes du filtre

// Fonction CPU pour la convolution 2D

void cpu\_convo2d(float \*A, float \*B, float \*C, int nrows, int ncols, int kr, int kc) {

// Parcourir chaque pixel de la matrice d'entrée

for (int i = 0; i < nrows; i++) {

for (int j = 0; j < ncols; j++) {

float result = 0.0f;

// Parcourir chaque élément du filtre

for (int m = 0; m < kr; m++) {

for (int n = 0; n < kc; n++) {

int a\_row = i + m - kr / 2;

int a\_col = j + n - kc / 2;

// Vérifier si les indices sont dans les limites de la matrice

if (a\_row >= 0 && a\_row < nrows && a\_col >= 0 && a\_col < ncols) {

result += A[a\_row \* ncols + a\_col] \* B[m \* kc + n];

}

}

}

// Mettre à jour la valeur dans la matrice de sortie

C[i \* ncols + j] = result;

}

}

}

int main() {

// Exemple d'utilisation

float A[NROW \* NCOL]; // Matrice d'entrée

float B[NRH \* NCH]; // Filtre

float C[NROW \* NCOL]; // Matrice de sortie

// Initialisation des matrices A et B

for (int i = 0; i < NROW \* NCOL; i++) {

A[i] = 1.0f; // Exemple de valeur

}

for (int i = 0; i < NRH \* NCH; i++) {

B[i] = 0.1f; // Exemple de filtre

}

// Appel de la fonction CPU

cpu\_convo2d(A, B, C, NROW, NCOL, NRH, NCH);

// Afficher une partie de la matrice de sortie (pour vérifier les résultats)

for (int i = 0; i < 5; i++) {

for (int j = 0; j < 5; j++) {

printf("%f ", C[i \* NCOL + j]);

}

printf("\n");

}

return 0;

}