Ministère de l'Enseignement Supérieur
et de la Recherche Scientifique

Université des Sciences et Technologies
Houari Boumediene

Rapport Projet TP :
Segmentation par thresholding

MEKHAZNI Ryham & KEFSI Nourhane

30 décembre 2024

Table des matières

Introduction

Dans un monde où les volumes de données à traiter ne cessent de croître, l'optimisation des performances des systèmes de calcul devient une nécessité. Le traitement d'images, en particulier, est un domaine clé dans de nombreuses applications informatiques, nécessitant souvent des calculs intensifs. Avec l'évolution des technologies, deux approches majeures sont utilisées pour effectuer ces calculs : le calcul séquentiel, qui traite les tâches de manière linéaire, et le calcul parallèle, qui exploite la puissance des processeurs modernes, notamment les GPU (*Graphics Processing Units*), pour exécuter plusieurs tâches simultanément, réduisant ainsi le temps d'exécution.

Dans ce rapport, nous nous intéressons à la segmentation d'images par seuillage (*thresholding*), une méthode simple mais essentielle pour extraire des informations significatives à partir d'images. En utilisant cette méthode, nous mettrons en œuvre et comparerons deux approches : une implémentation séquentielle et une implémentation parallèle exploitant CUDA, une technologie développée par NVIDIA pour tirer parti des GPU.

Nous explorerons et analyserons :

- Les définitions de concepts clés tels que le calcul séquentiel, le calcul parallèle, les GPU, CUDA, et le traitement d'image.
- Les étapes principales du traitement d'image et la méthode de segmentation par seuillage.
- Une explication détaillée des implémentations séquentielle et parallèle, accompagnée de captures d'écran.
- Une analyse comparative des deux approches en termes de performances, notamment en termes de temps d'exécution.

Ce travail vise à illustrer comment les technologies modernes, comme CUDA, peuvent transformer des processus de calcul intensifs, en mettant en lumière les avantages et les limites des différentes approches dans le domaine du traitement d'images.

1 Concepts Clés et Définitions

1.1 Introduction au Calcul Hétérogène

Code parallèle Le code parallèle est un type de programme où plusieurs instructions s'exécutent simultanément sur plusieurs unités de calcul (processeurs, cœurs, GPU). Contrairement au code séquentiel, il divise une tâche en sous-tâches indépendantes pour accélérer l'exécution. Cette approche est

couramment utilisée en calcul haute performance (HPC) et en traitement d'images.

1.1.1 code séquentiel

Le code séquentiel est un type de programme où les instructions sont exécutées une par une, de manière linéaire, sur une seule unité de calcul (processeur ou cœur). Chaque tâche doit se terminer avant que la suivante ne commence, ce qui peut ralentir l'exécution par rapport au code parallèle.

1.1.2 GPU

Un **GPU** (Graphics Processing Unit) est un processeur spécialisé dans le traitement rapide et parallèle de grandes quantités de données, en particulier pour le rendu graphique. Contrairement au CPU, il possède des milliers de cœurs capables d'exécuter des tâches en parallèle, ce qui le rend idéal pour des applications de calcul intensif comme l'apprentissage profond, la vision par ordinateur et le traitement d'images.

1.1.3 CPU

Un **CPU** (Central Processing Unit) est le processeur central d'un ordinateur, responsable de l'exécution des instructions d'un programme de manière séquentielle ou légèrement parallèle (avec plusieurs cœurs). Contrairement au GPU, il est optimisé pour des tâches générales et des calculs complexes nécessitant des décisions logiques et des opérations séquentielles.

1.1.4 Nvidia

NVIDIA est une entreprise technologique américaine spécialisée dans la conception de GPU (Graphics Processing Units) et de plates-formes de calcul parallèle. Elle est largement reconnue pour ses cartes graphiques utilisées dans les jeux vidéo, mais aussi pour ses contributions majeures au domaine de l'intelligence artificielle (IA), du deep learning et du calcul haute performance (HPC) grâce à sa plateforme de programmation parallèle CUDA.

1.1.5 CUDA

CUDA (Compute Unified Device Architecture) est une plateforme de calcul parallèle et un modèle de programmation développé par NVIDIA. Elle permet aux développeurs d'exploiter la puissance des GPU pour exécuter des calculs intensifs de manière parallèle. Avec CUDA, les programmeurs peuvent

écrire du code en C, C++ ou Python pour effectuer des tâches complexes en traitement d'images, deep learning, simulation scientifique et HPC (High-Performance Computing).

1.1.6 calcul hétérogène

Le calcul hétérogène consiste à utiliser les unités de calcule : CPU et GPU pour exécuter différentes parties d'un programme. Chaque unité est exploitée en fonction de ses points forts : le CPU pour les tâches séquentielles complexes et le GPU pour le traitement parallèle massif. Cette approche permet d'optimiser les performances, la consommation d'énergie et le temps d'exécution des applications intensives en calcul.

1.2 Introduction au Traitement d'Images

1.2.1 Définition d'une image

Une image est une représentation visuelle d'un objet ou d'un concept, qu'elle soit produite par des moyens naturels (vision humaine) ou artificiels (dessin, photo, film, etc.). Mathématiquement, une image peut être définie comme une fonction bidimensionnelle f(x,y), où f(x,y) représente l'intensité (ou le niveau de gris) de l'image aux coordonnées (x,y). Cette intensité correspond à l'énergie irradiée ou réfléchie par un processus physique tel que l'émission lumineuse, le rayonnement infrarouge, ou les rayons X. Ainsi, une image est à la fois une perception du monde extérieur et une abstraction permettant d'analyser ou de traiter visuellement des données.

1.2.2 étapes du traitement d'images

Le traitement d'image est une branche du traitement du signal qui vise à effectuer des opérations sur des images afin d'améliorer leur lisibilité, corriger des défauts ou en extraire des informations utiles

Le traitement d'image se décompose généralement en plusieurs étapes clés :

- Acquisition de l'image : Capture d'une image via un capteur (caméra, scanner) ou importation d'une image existante et la convertir en un format numérique pour traitement informatique.
- Prétraitement (ou traitement de base) : améliorer la qualité visuelle et préparer l'image pour des étapes ultérieures.
- Segmentation : Division de l'image en régions homogènes pour isoler des objets d'intérêt.

- Extraction de caractéristiques : Identification des caractéristiques pertinentes de l'image (contours, formes, textures).
- Analyse et interprétation : Étape où les informations extraites sont analysées et interprétées pour des applications spécifiques, telles que la classification, la détection ou le diagnostic.
- Compression (facultatif) : Réduction de la taille de l'image pour le stockage ou la transmission, tout en conservant une qualité acceptable.

Pour ce projet nous avons choisi de faire une segmentation en utilisant le seuillage simple.

1.2.3 segmentation par seuillage (thresholding)

La segmentation est une décomposition d'une image en régions qui ont un sens : les "objets" de l'image. La segmentation basée sur des seuils est une technique de traitement d'image qui consiste à comparer les valeurs de pixels avec des seuils spécifiés pour les classer en différents groupes, tels que la végétation végétale et le fond du sol, en fonction des résultats de la comparaison. Il existent deux types de seuillage :

Seuillage simple

Il permet de segmenter les pixels d'une image en fonction d'une valeur seuil fixe. Cette méthode permet de convertir une image en niveaux de gris en une image binaire, où les pixels inférieurs au seuil sont mis à zéro (noirs) et les pixels supérieurs sont mis à 255 (blancs) en passant par deux étapes :

— le choix de seuil fixe

— Application de ce suil sur les pixels d l'image.

Seuillage otsu

Est une méthode automatique de binarisation d'une image, il permet de séparer les pixels en deux classes : l'objet (foreground) et l'arrière-plan (background). L'objectif est de trouver un seuil optimal qui minimise la variance intra-classe tout en maximisant la variance inter-classe en passant par plusieurs étapes :

— Calcul d'histogramme

— Calcul des caractéristiques : poids de la classe fond, poids de la classe premier plan, moyenne de la classe de fond, moyenne de la classe de premier plan.

— Calcul de la variance inter-classe

— Choix de seuil optimale, celui qui maximise la variance inter-classe

— Binarisation

2 Analyse de codes

2.1 Analyse du Code Séquentiel :

2.1.1 Bibliothèques utilisées

- cv2 (OpenCV) : Une bibliothèque de traitement d'images et de vision par ordinateur. Elle permet de lire, traiter et enregistrer des images. Elle offre aussi de nombreuses fonctionnalités de manipulation d'images, comme le redimensionnement, l'application de filtres, et plus encore.
- time : Une bibliothèque pour mesurer le temps d'exécution d'un programme. Elle est utilisée ici pour mesurer le temps que prend l'exécution du code, ce qui permet de comparer l'efficacité de différentes approches.
- matplotlib : une bibliotheèque utilisé pour les affichages dans notre cas pour affichier l'image originale et segmentée.

2.1.2 Seuilage simple

Étape 1 : défntion de la fonction de thresholding

Pour cette étape, nous avons développé une fonction en Python appelée **threshold_simple**, qui suit les étapes suivantes :

— Convertir l'image d'entrée en niveaux de gris à l'aide d'OpenCV.

— Comparer chaque pixel avec la valeur seuil donnée.

— Générer une image résultat seuillée.

```python
def threshold_simple(image, threshold):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    result_image = gray.copy()
    height, width = gray.shape
    for i in range(height):
        for j in range(width):
            result_image[i, j] = 0 if gray[i, j] < threshold else 255
    return result_image
```

FIGURE 1 – Fonction de Seuillage Simple

Étape 2 : fonction de mesure de temps

Cette étape vise à mesurer le temps d'exécution de la fonction de seuillage simple en appliquant l'algorithme à des images redimensionnées de différentes tailles. Cela permet de comparer le temps d'execution avec celui du parallele dans la partie comparaisn cette foncition comprend plusieurs étapes :

— Chargement de l'image

— Redimensionnement, L'image originale est redimensionnée selon plusieurs facteurs de redimensionnement spécifiés dans une liste. Ces facteurs modifient la taille de l'image

— Application du Seuillage, pour chaque image redimensionnée, la fonction **threshold_simple** est appliquée. Le temps d'exécution est mesuré en utilisant les fonctions time.time() avant et après l'exécution.

— Visualisation

— La sauvegarde des résultats, pour chaque facteur du redimensionnement de l'image nous avons sauvegardé son temps d'execution dans un fichier texte pour une analyse ultérieure.

```python
def measure_execution_times(image_path, resize_factors, output_file):
    original_image = cv2.imread(image_path)
    if original_image is None:
        print("Erreur : Impossible de charger l'image.")
        return

    results = []
    for factor in resize_factors:
        # Redimensionner l'image
        resized_image = cv2.resize(
            original_image,
            (original_image.shape[1] * factor, original_image.shape[0] * factor)
        )
        start_time = time.time()
        result_image = threshold_simple(resized_image, 188)
        end_time = time.time()
        execution_time = end_time - start_time
        results.append((factor, execution_time))
        print(f"Facteur de redimensionnement : {factor}, Temps d'exécution : {execution_time:.6f} secondes")

        if factor == 1:
            plt.figure(figsize=(10, 5))
            plt.subplot(1, 2, 1)
            plt.title("Image originale")
            plt.imshow(cv2.cvtColor(resized_image, cv2.COLOR_BGR2RGB))
            plt.axis("off")
            plt.subplot(1, 2, 2)
            plt.title("Image segmentée (Seuillage simple)")
            plt.imshow(result_image, cmap="gray")
            plt.axis("off")
            plt.show()
```

le chargement de l'image

le redimensionnement de l'image

le calcule du temps pour l'application du seuillage simple

l'affichage de l'image originale ainsi que sa version segmentée

FIGURE 2 – Mesure des Temps d'Exécution

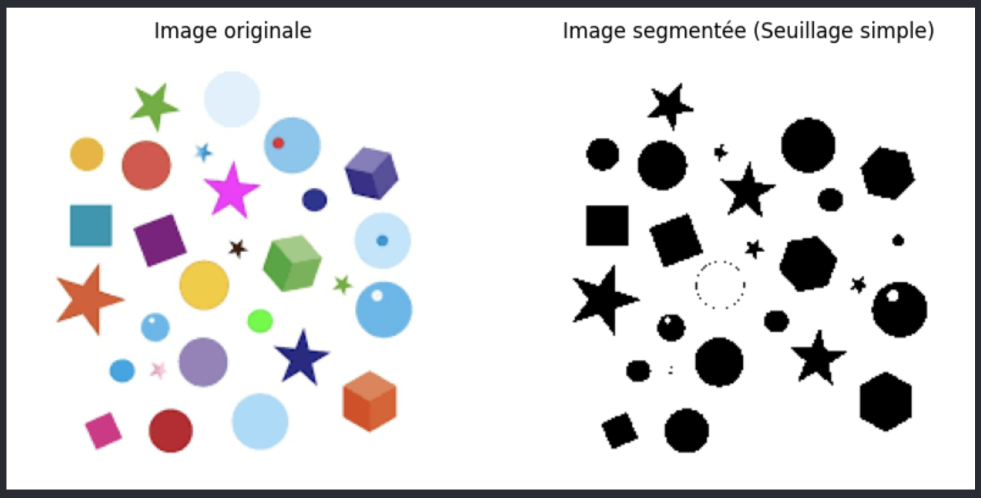
Application :

```python
# Exemple d'utilisation
if __name__ == "__main__":
    image_path = "../../images.png"
    resize_factors = [1, 3, 5, 7, 9, 11]
    output_file = "execution_times_simple.rtf"
    measure_execution_times(image_path, resize_factors, output_file)
```

FIGURE 3 – Application des fonctions

Résultat :

```
Facteur de redimensionnement : 1, Temps d'exécution : 0.118032 secondes
```




```
Facteur de redimensionnement : 3, Temps d'exécution : 1.063929 secondes
Facteur de redimensionnement : 5, Temps d'exécution : 3.012486 secondes
Facteur de redimensionnement : 7, Temps d'exécution : 5.765212 secondes
Facteur de redimensionnement : 9, Temps d'exécution : 9.571019 secondes
Facteur de redimensionnement : 11, Temps d'exécution : 14.287436 secondes
Facteur de redimensionnement : 13, Temps d'exécution : 20.150966 secondes
```

FIGURE 4 – Le résultat du seuillage simple en séquentiel

2.1.3 Seuilage Otsu

Étape 1 : Calcul d'histogramme

Cette fonction consiste à calculer l'histogramme qui permet de représenter la distribution des niveaux de gris dans l'image, ce qui est nécessaire pour déterminer automatiquement un seuil optimal.

```python
def calculate_histogram(image):
    histogram = [0] * 256
    height, width = image.shape
    for i in range(height):
        for j in range(width):
            pixel_value = image[i, j]
            histogram[pixel_value] += 1
    return histogram
```

FIGURE 5 – Le calcul d'histogramme

Étape 2 : la segmentation par Otsu

Cette étape consiste à calculer automatiquement un seuil optimal permettant de segmenter l'image en deux classes (arrière-plan et premier plan) de manière à maximiser la variance inter-classes.

- Prétraitement, si l'image est en couleur, elle sera convertie en niveaux de gris à l'aide de la fonction **cv2.cvtColor**. Cela simplifie les calculs puisque la méthode d'Otsu est conçue pour des images en niveaux de gris.
- Calcul de l'Histogramme, on a fait appel à la fonction qui permet de calculer l'histogramme pour l'image en niveaux de gris.
- Initialisation, Plusieurs variables sont initialisées pour le calcul : **total_pixels** qui est le nombre total de pixels dans l'image. **sum_all_pixels** c'est la somme pondérée des intensités, **current_max_variance** c'est pour suivre la variance inter-classes maximale rencontrée et threshold pour enregistrer le seuil optimal.
- Parcours des Niveaux de Gris, pour chaque intensité les poids de l'arrière-plan et du premier plan sont calculés et leurs moyennes, La variance

inter-classes est évaluée, si elle est supérieure à la variance maximale actuelle, le seuil et la variance maximale sont mis à jour.

— Application du Seuillage, Une fois le seuil optimal trouvé, la fonction **cv2.threshold** est utilisée pour produire une image binaire.

```python
def otsu_thresholding(image):
    if len(image.shape) == 3:
        gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    else:
        gray = image
    histogram = calculate_histogram(gray)
    total_pixels = gray.size
    current_max_variance = 0
    threshold = 0
    sum_all_pixels = sum(i * histogram[i] for i in range(256))
    sum_background = 0
    weight_background = 0
    for t in range(256):
        weight_background += histogram[t]
        if weight_background == 0:
            continue
        weight_foreground = total_pixels - weight_background
        if weight_foreground == 0:
            break
        sum_background += t * histogram[t]
        mean_background = sum_background / weight_background
        mean_foreground = (sum_all_pixels - sum_background) / weight_foreground
        inter_class_variance = (
            weight_background * weight_foreground * (mean_background - mean_foreground) ** 2
        )
        if inter_class_variance > current_max_variance:
            current_max_variance = inter_class_variance
            threshold = t
    _, binary_image = cv2.threshold(gray, threshold, 255, cv2.THRESH_BINARY)

    return threshold, binary_image
```

FIGURE 6 – La segmentation otsu séquentielle

Étape 3 : fonction de mesure de temps

Cette étape vise à mesurer le temps d'exécution de la fonction de seuillage Otsu en appliquant l'algorithme à des images redimensionnées de différentes tailles et faire un affichage

```python
def measure_execution_times_and_display(image_path, resize_factors, output_file):
    # Charger l'image
    original_image = cv2.imread(image_path)
    if original_image is None:
        print("Erreur : Impossible de charger l'image.")
        return
    results = []
    # Mesurer les temps d'exécution pour chaque facteur de redimensionnement
    for factor in resize_factors:
        resized_image = cv2.resize(
            original_image,
            (original_image.shape[1] * factor, original_image.shape[0] * factor)
        )
        start_time = time.time()
        threshold, binary_image = otsu_thresholding(resized_image)
        end_time = time.time()
        execution_time = end_time - start_time
        results.append((factor, execution_time))
        print(f"Facteur de redimensionnement : {factor}, Temps d'exécution : {execution_time:.6f} secondes, Seuil : {threshold}")
        # Affichage
        if factor == 1:
            plt.figure(figsize=(10, 5))
            plt.subplot(1, 2, 1)
            plt.title("Image originale")
            plt.imshow(cv2.cvtColor(resized_image, cv2.COLOR_BGR2RGB))
            plt.axis("off")

            plt.subplot(1, 2, 2)
            plt.title(f"Image segmentée (Seuil = {threshold})")
            plt.imshow(binary_image, cmap="gray")
            plt.axis("off")
```

FIGURE 7 – Le calcul du temps

Application :

```python
# Exemple d'utilisation
if __name__ == "__main__":
    image_path = "../../images.png"
    resize_factors = [1, 5, 10, 15, 20, 25, 30]
    output_file = "execution_times.rtf"
    measure_execution_times_and_display(image_path, resize_factors, output_file)
```

FIGURE 8 – Application des fonctions dans le cas otsu séquentiel

Résultat :

Le resultat affiche le teemps d'execution pour chaque taille d'image ainsi que le calcul de seuillage trouvé

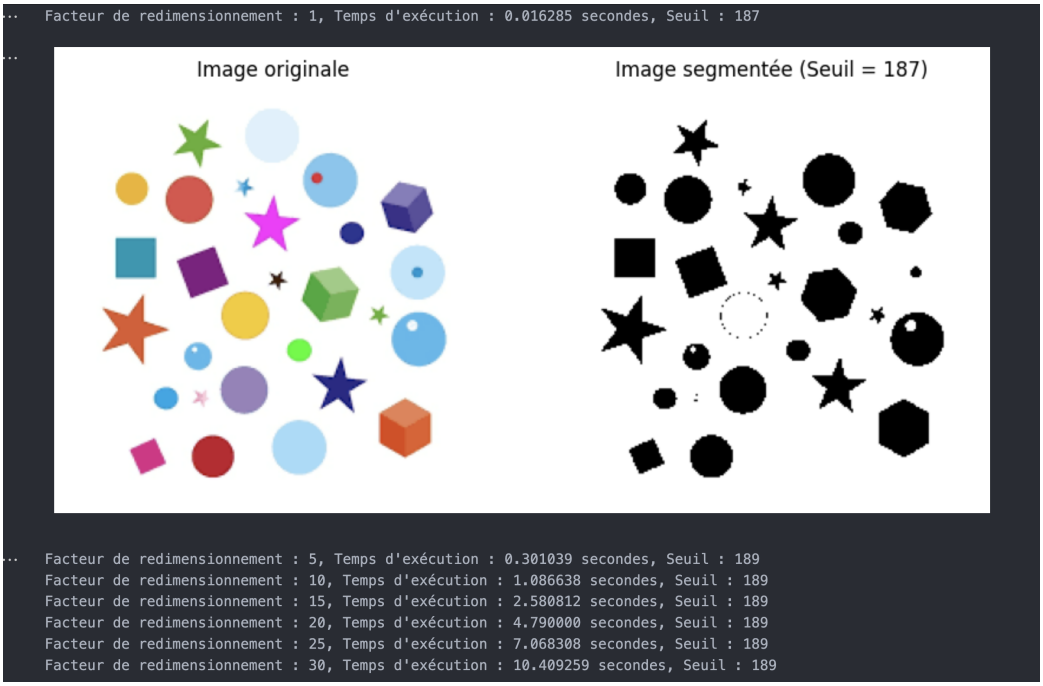


FIGURE 9 – Le résultat du seuillage otsu en séquentiel

2.2 Analyse du Code Parallèle

Ce projet utilise des bibliothèques externes et CUDA pour effectuer un traitement parallèle sur une image d'entrée. Les étapes détaillées de l'implémentation sont décrites ci-dessous.

2.2.1 Bibliothèques utilisées :

Le programme utilise les bibliothèques suivantes :

— `stb_image.h` et `stb_image_write.h` : pour le chargement et la sauvegarde des images au format **.png**.

— `math.h` : pour les calculs mathématiques nécessaires.

Les constantes et macros importantes incluent :

— `#define STB_IMAGE_IMPLEMENTATION` et `#define STB_IMAGE_WRITE_IMPLEMENTATION` : pour activer les implémentations des bibliothèques STB.

— `__constant__ int N` : constante CUDA pour stocker la taille totale de l'image.

2.2.2 Seuillage simple

Chargement et préparation de l'image : L'image est chargée en niveaux de gris depuis un fichier d'entrée (`/content/images.png`) en utilisant `stbi_load`. Cette fonction convertit automatiquement l'image en une seule composante par pixel, correspondant à son niveau de gris.

```
int main() {
    int width, height, channels;
    unsigned char *image = stbi_load("/content/images.png", &width, &height, &channels, 1);
    if (image == NULL) {
        printf("Erreur lors du chargement de l'image\n");
        return 1;
    }

    int factors[] = {1, 3, 5, 7, 9, 11};
    int num_factors = sizeof(factors) / sizeof(factors[0]);

    unsigned char *d_image, *d_imageResult;
    printf("Taille originale de l'image : %dx%d\n", width, height);
    printf("Facteur\tTemps moyen du kernel (ms)\n");

    for (int i = 0; i < num_factors; i++) {
        int factor = factors[i];
        int new_width = width / factor;
        int new_height = height / factor;
        int n = new_width * new_height;

        unsigned char* resized_image = resize_image(image, width, height, new_width, new_height);
```

FIGURE 10 – Chargement et préparation de l'image

Les dimensions de l'image (`width` et `height`) et le nombre de canaux (`channels`, forcé à 1 pour le niveau de gris) sont extraits lors du chargement.

```
unsigned char* resize_image(unsigned char* input_image, int width, int height, int new_width, int new_height) {
    unsigned char* resized_image = (unsigned char*)malloc(new_width * new_height * sizeof(unsigned char));
    float x_ratio = (float)width / new_width;
    float y_ratio = (float)height / new_height;

    for (int y = 0; y < new_height; y++) {
        for (int x = 0; x < new_width; x++) {
            int px = (int)(x * x_ratio);
            int py = (int)(y * y_ratio);
            resized_image[y * new_width + x] = input_image[py * width + px];
        }
    }
    return resized_image;
}
```

FIGURE 11 – redimensionnement de l'image

Une fonction de redimensionnement (`resize_image`) est utilisée pour générer des versions agrandies de l'image d'entrée selon différents facteurs. Cette étape est effectuée sur le CPU avant le transfert des données au GPU.

Allocation mémoire La mémoire est allouée sur :

— le **CPU**, pour stocker les images redimensionnées avant et après traitement ;

— le **GPU**, pour l'image d'entrée redimensionnée (`d_image`) et le résultat (`d_imageResult`).

Implémentation du Kernel CUDA Le traitement parallèle est effectué par le kernel CUDA `thresholding`, qui applique un seuil sur chaque pixel. Chaque thread traite un pixel en suivant l'équation suivante :

$$\texttt{imageResult[idx]} = (\texttt{image[idx]}/128) \times 255$$

Ainsi, chaque pixel est converti en noir (**0**) ou en blanc (**255**) selon qu'il est inférieur ou supérieur au seuil (128). Les identifiants des threads (`threadIdx.x`, `blockIdx.x`) sont utilisés pour accéder aux pixels de manière parallèle. Une condition (`idx < n`) garantit que seuls les pixels valides sont traités.

Paramètres d'exécution CUDA Le nombre de threads par bloc (`blockSize`) est fixé à 32. Le nombre de blocs nécessaires (`numBlocks`) est calculé dynamiquement pour couvrir l'ensemble des pixels :

$$\texttt{numBlocks} = \lceil \frac{n}{\texttt{blockSize}} \rceil$$

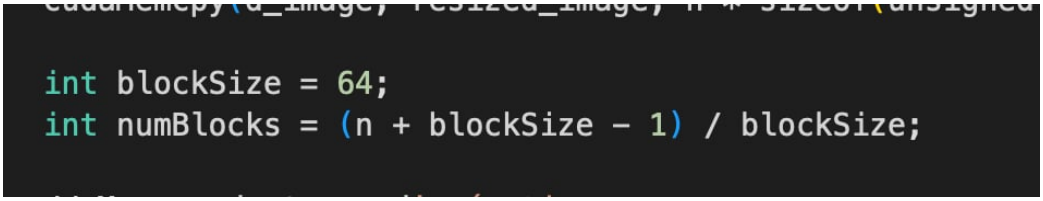


FIGURE 12 – Paramètres d'exécution CUDA

Mesure des performances Le temps d'exécution du kernel est mesuré pour différents facteurs de redimensionnement de l'image (`factors`) en utilisant les événements CUDA (`cudaEvent`). Les résultats sont enregistrés dans un fichier CSV (`kernel_performance.csv`), permettant une analyse comparative des performances en fonction de la taille des images.

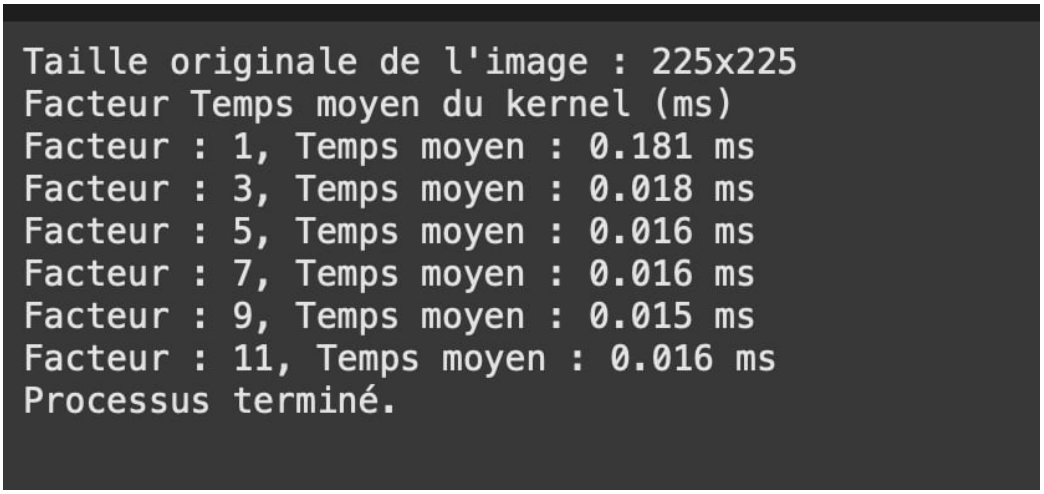


FIGURE 13 – Mesure des performances : factors - excetion time

Sauvegarde de l'image résultante Après le traitement, l'image seuilée est récupérée depuis la mémoire du GPU, puis sauvegardée au format `.png` (`/content/thresholded_image.png`) à l'aide de `stbi_write_png`.

Résultats attendus Les résultats du projet comprennent :

- Une image binaire résultante, sauvegardée dans le répertoire courant.
- Un fichier CSV récapitulant les temps d'exécution du kernel pour chaque facteur de redimensionnement.
- L'affichage des résultats dans la console, incluant :
  1. La taille originale et les tailles redimensionnées des images.
  2. Les temps d'exécution du kernel pour chaque facteur.

2.2.3 Seuil d'Otsu

Cette section présente un programme CUDA implémentant la segmentation d'image en utilisant la méthode du seuil d'Otsu. Le code est structuré en plusieurs étapes, chacune décrite ci-dessous.

Kernel CUDA : Segmentation par le Seuil d'Otsu Le kernel `otsu_segmentation` réalise les étapes suivantes :

1. **Calcul de l'histogramme :** Chaque thread incrémente atomiquement un élément de l'histogramme correspondant à l'intensité du pixel.

FIGURE 14 – Kernel CUDA

2. **Calcul des caractéristiques de l'histogramme :** Un seul thread (`threadIdx.x = 0`) calcule :

```
// Calcul des caractéristiques de l'histogramme
int tid = threadIdx.x;
if (tid < 256) {
    float total = 0.0f, sumB = 0.0f, maxVariance = 0.0f;
    int totalPixels = 0, bestThreshold = 0;

    for (int i = 0; i < 256; ++i) {
        total += i * histogram[i];
        totalPixels += histogram[i];
    }

    float weightB = 0.0f, weightF = 0.0f;
    for (int t = 0; t < 256; ++t) {
        weightB += histogram[t];
        if (weightB == 0) continue;

        weightF = totalPixels - weightB;
        if (weightF == 0) break;

        sumB += t * histogram[t];
        float meanB = sumB / weightB;
        float meanF = (total - sumB) / weightF;

        float betweenVar = weightB * weightF * powf(meanB - meanF, 2);
        if (betweenVar > maxVariance) {
            maxVariance = betweenVar;
            bestThreshold = t;
        }
    }

    if (tid == 0) {
        *threshold = bestThreshold;
    }
}
__syncthreads();
```

FIGURE 15 – Kernel CUDA

— Le poids (`weightB` et `weightF`) des pixels de fond et de premier

plan.
   — Les moyennes des intensités (`meanB` et `meanF`).
   — La variance inter-classes maximale pour déterminer le seuil optimal.
3. **Application du seuil :** Chaque pixel de l'image est comparé au seuil calculé, et son intensité est mise à 255 (**blanc**) ou 0 (**noir**).

```
    // Appliquer le seuil à l'image
    if (idx < n) {
        imageResult[idx] = (image[idx] > *threshold) ? 255 : 0;
    }
}
```

FIGURE 16 – Application du seuil

Fonction de Redimensionnement de l'Image La fonction `resize_image` ajuste les dimensions de l'image source (`old_width`, `old_height`) vers une nouvelle taille (`new_width`, `new_height`) en utilisant un sous-échantillonnage.

```
void resize_image(const unsigned char *input, unsigned char *output, int old_width, int old_height, int new_width, int new_height) {
    for (int y = 0; y < new_height; ++y) {
        for (int x = 0; x < new_width; ++x) {
            int src_x = x * old_width / new_width;
            int src_y = y * old_height / new_height;
            output[y * new_width + x] = input[src_y * old_width + src_x];
        }
    }
}
```

FIGURE 17 – Fonction de Redimensionnement de l'Image

Fonction Principale : Workflow Global La fonction `main` réalise les étapes suivantes :

1. **Chargement de l'image source :** L'image est chargée en niveaux de gris depuis `/content/images.png`.
2. **Traitement pour différents facteurs de redimensionnement :** Les tailles sont multipliées par les facteurs `{1, 5, 10, 15, 20, 25}`.

3. **Allocation mémoire :** La mémoire est allouée pour :
   — Les images d'entrée (`d_image`) et de sortie (`d_imageResult`) sur le GPU.

```
int main() {
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    int width, height, channels;
    unsigned char *image = stbi_load("/content/images.png", &width, &height, &channels, 1);
    if (image == NULL) {
        printf("Erreur lors du chargement de l'image\n");
        return 1;
    }
```

FIGURE 18 – Fonction principale main

— L'histogramme, le seuil, et d'autres caractéristiques nécessaires.

4. **Lancement du Kernel :** Le kernel est lancé avec un nombre de threads adapté à la taille de l'image :

$$\texttt{numBlocks} = \lceil \frac{\texttt{n}}{\texttt{blockSize}} \rceil$$

Le temps d'exécution est mesuré à l'aide des événements CUDA (`cudaEvent`).

```
// Initialiser N
cudaMemcpyToSymbol(N, &n, sizeof(int));

cudaEventRecord(start, 0);
otsu_segmentation<<<numBlocks, blockSize>>>(d_image, n, d_imageResult, d_histogram, moyennes, weights, caracteristiques, d_threshold);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

float time = 0;
cudaEventElapsedTime(&time, start, stop);
```

FIGURE 19 – temps d'execution avec cudaEvent

5. **Récupération des résultats :** Le seuil optimal et l'image résultante sont récupérés depuis la mémoire GPU.
6. **Sauvegarde des résultats :** L'image segmentée est sauvegardée au format `.png` sous le nom `/content/otsu_segmented_factor_{factor}.png`.

Analyse des Performances Pour chaque facteur de redimensionnement, le programme mesure et affiche :

— Le temps d'exécution du kernel (en millisecondes).

— Le seuil trouvé (`h_threshold`).

Ces résultats permettent d'évaluer l'impact de la taille de l'image sur les performances.

```
cudaEventElapsedTime(&time, start, stop);

int h_threshold;
cudaMemcpy(&h_threshold, d_threshold, sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(h_imageResult, d_imageResult, n * sizeof(unsigned char), cudaMemcpyDeviceToHost);

printf("Facteur %d : Temps d'exécution : %.3f ms, Seuil trouvé : %d\n", factor, time, h_threshold);

char output_filename[50];
sprintf(output_filename, "/content/otsu_segmented_factor_%d.png", factor);
stbi_write_png(output_filename, new_width, new_height, 1, h_imageResult, new_width);
```

FIGURE 20 – Sauvegarde de l'image

```
Facteur 1 : Temps d'exécution : 0.843 ms, Seuil trouvé : 188
Facteur 5 : Temps d'exécution : 14.046 ms, Seuil trouvé : 189
Facteur 10 : Temps d'exécution : 55.914 ms, Seuil trouvé : 188
Facteur 15 : Temps d'exécution : 120.444 ms, Seuil trouvé : 188
Facteur 20 : Temps d'exécution : 168.554 ms, Seuil trouvé : 254
Facteur 25 : Temps d'exécution : 265.408 ms, Seuil trouvé : 254
Facteur 30 : Temps d'exécution : 375.300 ms, Seuil trouvé : 254
```

FIGURE 21 – Analyse des Performances

```
    free(resized_image);
    free(h_imageResult);
    cudaFree(d_image);
    cudaFree(d_imageResult);
    cudaFree(d_histogram);
    cudaFree(d_threshold);
    cudaFree(moyennes);
    cudaFree(weights);
    cudaFree(caracteristiques);
}

stbi_image_free(image);

cudaEventDestroy(start);
cudaEventDestroy(stop);

return 0;
```

FIGURE 22 – Nettoyage

Résultats et Nettoyage Une fois toutes les étapes terminées :

Les événements CUDA (`cudaEvent`) sont détruits.

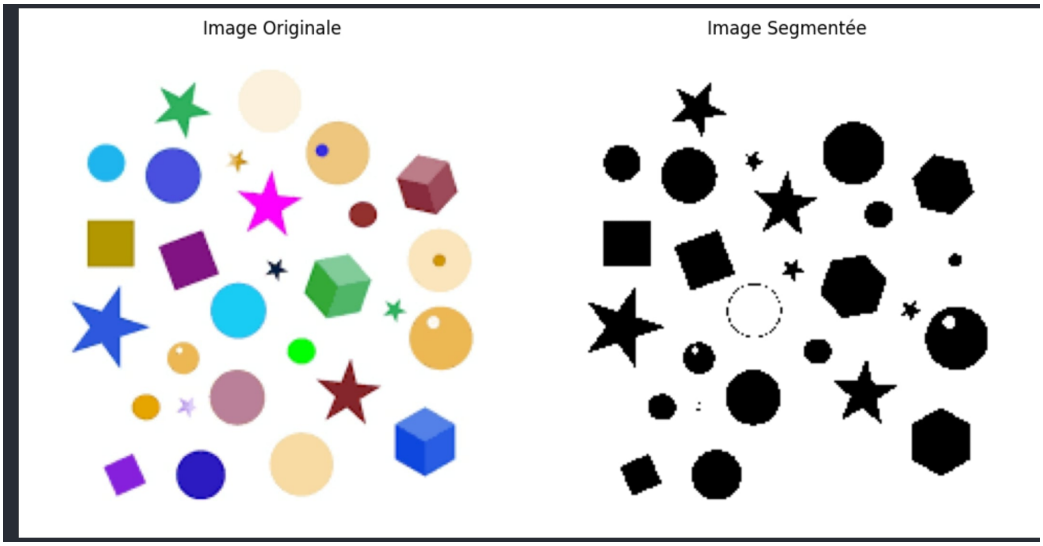


FIGURE 23 – image originales vs image segmente

— La mémoire allouée sur le CPU et le GPU est libérée.

— L'image source est désallouée à l'aide de `stbi_image_free`.

Le programme termine avec succès, produisant des images segmentées pour chaque facteur de redimensionnement.

2.3 Comparaison entre Code Séquentiel et Parallèle

Cette section compare les performances des implémentations séquentielle et parallèle pour deux cas : le seuillage simple et le seuillage d'Otsu.

2.3.1 Seuillage Simple

Le seuillage simple consiste à appliquer un seuil fixe sur une image pour en distinguer les pixels en deux classes (fond et premier plan). Voici les caractéristiques des deux implémentations :

Implémentation Séquentielle

- **Approche :** Le programme parcourt chaque pixel de l'image, compare son intensité au seuil, et met à jour le pixel en conséquence.
- **Complexité :** $O(n)$, où n est le nombre total de pixels dans l'image.
- **Performance :** Le temps d'exécution est directement proportionnel à la taille de l'image.
- **Limitations :** Inefficace pour des images de grande taille en raison de la nature linéaire du traitement.

Implémentation Parallèle

- **Approche :** Chaque thread du GPU traite un pixel en parallèle, ce qui permet de répartir la charge de travail.
- **Complexité :** $O(1)$ pour chaque pixel, avec une complexité totale limitée par la taille des blocs et des grilles GPU.
- **Performance :** Temps d'exécution considérablement réduit pour des images volumineuses grâce à l'exécution simultanée sur plusieurs threads.
- **Limitations :** Nécessite une configuration optimale des blocs et des grilles pour éviter les conflits de threads.

Résultats de la Comparaison

- **Temps d'exécution :** L'implémentation parallèle est plus rapide que la version séquentielle pour des images de grande taille.
- **Scalabilité :** Le code parallèle offre une scalabilité presque linéaire avec l'augmentation du nombre de threads et la taille des données.

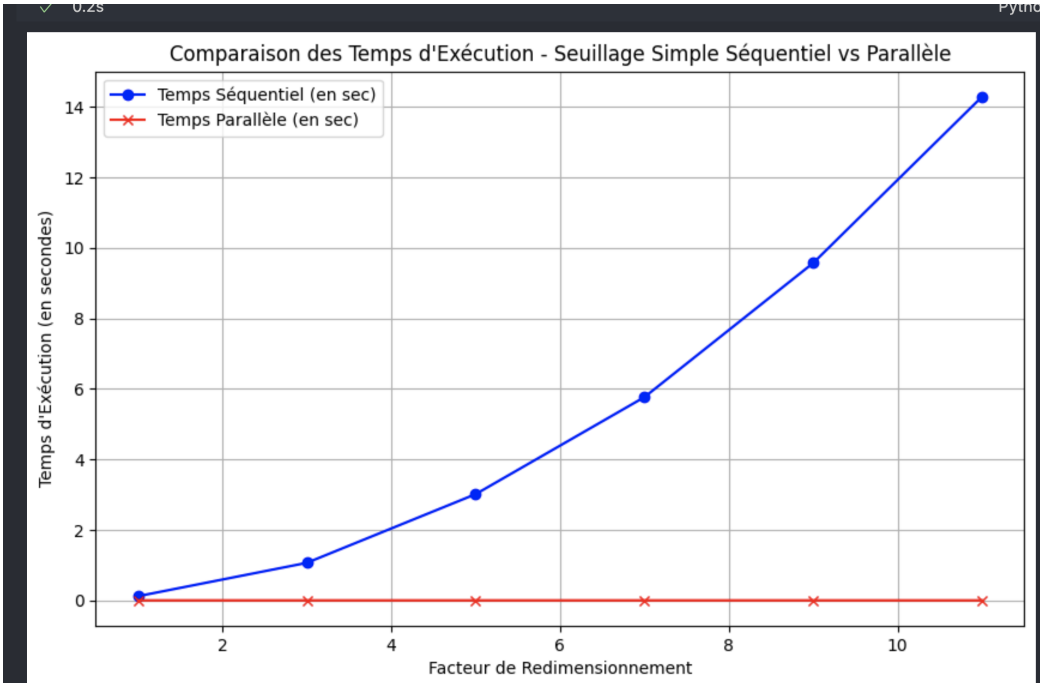


FIGURE 24 – comparaison de seuillage simple

2.3.2 Seuillage d'Otsu

Le seuillage d'Otsu est une méthode plus complexe, qui détermine automatiquement le seuil optimal en analysant l'histogramme de l'image.

Implémentation Séquentielle

- **Approche :** L'histogramme de l'image est calculé en premier, suivi par une itération sur les niveaux de gris pour trouver le seuil optimal en maximisant la variance inter-classes.
- **Complexité :** $O(n + m)$, où n est le nombre de pixels et $m = 256$ est le nombre de niveaux de gris.
- **Performance :** L'implémentation devient lente pour des images volumineuses, surtout à cause du calcul séquentiel de l'histogramme.
- **Limitations :** Non adaptée pour un traitement en temps réel.

Implémentation Parallèle

- **Approche :**
  1. **Étape 1 : Calcul de l'histogramme.** Chaque thread incrémente atomiquement un compteur correspondant à l'intensité du pixel.

2. **Étape 2 : Recherche du seuil optimal.** Un thread calcule les moyennes et la variance inter-classes en analysant les données de l'histogramme.

— **Complexité :** $O(1)$ pour chaque pixel et niveau de gris, avec une complexité totale dépendant de la configuration du GPU.

— **Performance :** Temps d'exécution significativement réduit par rapport à la version séquentielle grâce à la parallélisation des calculs.

— **Limitations :** Dépendance à la bande passante mémoire et aux conflits de threads dans l'étape de mise à jour de l'histogramme.

Résultats de la Comparaison

— **Temps d'exécution :** L'implémentation parallèle est plus rapide que la version séquentielle, selon la taille de l'image.

— **Précision :** Les deux implémentations produisent des résultats identiques, mais la version parallèle est bien plus efficace pour des images de grande taille.

— **Utilisation des ressources :** La version parallèle utilise pleinement les ressources GPU, réduisant ainsi le temps de calcul pour les grandes images.

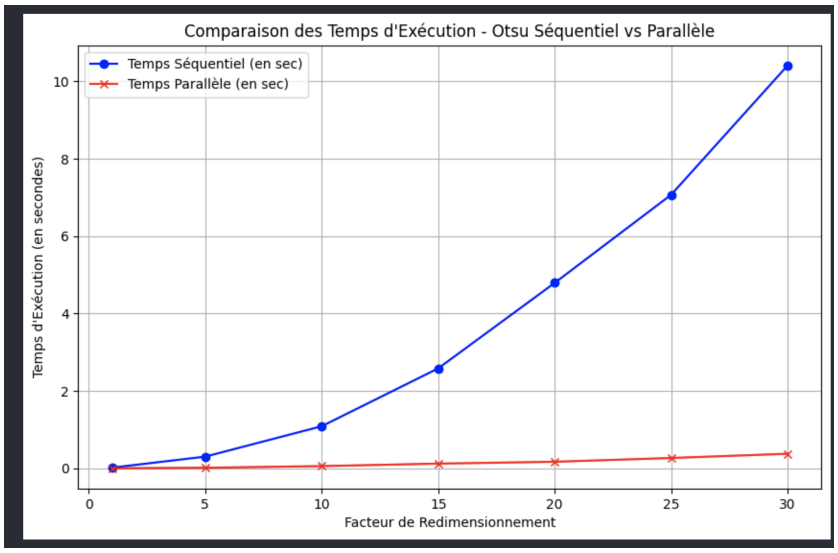


FIGURE 25 – comparaison de seuillage Otsu

Conclusion

Dans ce rapport, nous avons exploré l'application du traitement d'image par seuillage à l'aide de deux approches distinctes : le calcul séquentiel et le calcul parallèle. À travers cette étude, nous avons mis en évidence les principes fondamentaux du traitement d'image, les étapes clés, et les avantages qu'apportent les technologies modernes telles que CUDA pour accélérer le traitement.

Le code séquentiel, bien que simple à implémenter et suffisant pour des tâches de petite envergure, montre rapidement ses limites en termes de performance lorsqu'il est appliqué à des images de grandes dimensions. En revanche, l'utilisation du calcul parallèle via GPU permet de tirer parti de la puissance de calcul des architectures modernes, réduisant significativement le temps de traitement pour des volumes de données importants.

La comparaison des deux approches a permis de constater que, malgré la complexité relative du calcul parallèle, il offre un avantage considérable en termes de performances pour des applications intensives. Ces résultats soulignent l'importance de choisir la bonne approche en fonction des besoins et des contraintes spécifiques de chaque projet.

Enfin, ce travail illustre l'intérêt des technologies de calcul hétérogène dans le domaine du traitement d'image et ouvre la voie à des explorations futures, notamment l'application d'algorithmes de segmentation plus avancés.

Table des figures