ÉCOLE NATIONALE D'INGÉNIEURS DE BREST

RÉVÉLATEUR D'INGÉNIEUR·E·S
DEPUIS 60 ANS

TESTIA
AN AIRBUS COMPANY

Stage Ingénieur 2024/2025

Traitement d'image avec CUDA

Du 01/07/2024 au 24/12/2024

Tuteur entreprise : BOITTIN Tanguy

Tuteur académique : DESMEULLES Gireg

Stagiaire :
ABDEL KADER
Omar

Remerciements

Je souhaite exprimer ma profonde reconnaissance envers toutes les personnes ayant contribué à la réussite de mon stage. Je remercie tout particulièrement Monsieur **Tanguy Boittin** et Monsieur **Christophe Dinh**, mes tuteurs tout au long de cette expérience, ainsi que Monsieur **Jean-Yves Pelé**, responsable de l'équipe R&D, et l'ensemble des membres de cette équipe, qui ont grandement facilité mon intégration chez Testia.

Je tiens également à adresser mes sincères remerciements à Monsieur **Gireg Desmeulles** pour son soutien et son encadrement attentif tout au long de mon stage. Son accompagnement a été d'une aide précieuse dans l'élaboration de ce rapport de stage.

Résumé/Abstract

Français

Durant mon stage de six mois chez Testia, j'ai eu l'occasion de renforcer significativement mes compétences dans divers domaines. Ce stage a été une opportunité idéale pour mettre en pratique les connaissances acquises durant mes études et approfondir mon expertise en génie logiciel. J'ai particulièrement amélioré mes compétences en programmation, notamment en C++ et CUDA, et j'ai mené des recherches approfondies sur l'architecture des GPU, en particulier celle de NVIDIA. J'ai également développé mon expertise en traitement d'images.

Cette expérience m'a permis de renforcer ma capacité à résoudre des problèmes complexes, à mieux communiquer et à travailler de manière autonome. Grâce aux échanges avec mes collègues, j'ai appris à aborder les défis techniques avec une approche plus professionnelle.

Testia, spécialisée dans le contrôle non destructif et l'intégrité structurelle, cherche à optimiser le temps de traitement des images dans son système. Mon stage s'est structuré en trois phases principales : tout d'abord, j'ai réalisé une étude approfondie sur les GPU NVIDIA et les technologies de calcul parallèle. Ensuite, j'ai analysé les performances en développant de nouveaux algorithmes d'extraction de raies laser à partir d'images, implémentés à la fois en CUDA pour le GPU et en CPU, puis en réalisant des tests comparatifs. Enfin, j'ai intégré cette technologie dans leur logiciel de traitement d'images, IDEEPIX.

Je tiens à exprimer ma gratitude envers mon maître de stage et toute l'équipe pour leur soutien et leur confiance tout au long de cette expérience. Ce stage m'a permis d'acquérir de nouvelles compétences et a clairement orienté mon parcours vers l'ingénierie.

English

During my six-month internship at Testia, I had the opportunity to significantly enhance my skills in various areas. This internship was an ideal occasion to put into practice the knowledge gained during my studies and to deepen my expertise in software engineering. I especially improved my programming skills, notably in C++ and CUDA, and conducted in-depth research on GPU architecture, particularly NVIDIA's. I also developed my expertise in image processing.

This experience allowed me to strengthen my ability to solve complex problems, communicate more effectively, and work independently. Through exchanges with my colleagues, I learned to approach technical challenges with a more professional perspective.

Testia, specializing in non-destructive testing and structural integrity, aims to optimize image processing time in its system. My internship was structured into three main phases: first, I conducted a thorough study on NVIDIA GPUs and parallel computing technologies. Next, I analyzed performance by developing new laser stripe extraction algorithms for images, implemented both in CUDA for the GPU and on the CPU, followed by comparative performance testing. Finally, I integrated this technology into their image processing software, IDEEPIX.

I would like to express my gratitude to my supervisor and the entire team for their support and trust throughout this experience. This internship allowed me to acquire new skills and has clearly directed my career path toward engineering.

Table des matières

Liste des figures

Liste des tableaux

Glossaire

A

AF-Inspect: Automated Fiber Inspect
AFP/ATL: Automated Fiber Placement / Automated Tape Laying
API: Application Programming Interface

C

CPU: Central processing Unit
CUDA: Compute Unified Device Architecture

F

FG: FrameGrabber

G

GPGPU: General-Purpose Graphics Processing Unit
GPU: Graphics Processing Unit

I

IDEEPIX: Un logiciel de traitement d'images développé spécifiquement par TESTIA

P

PLY: Polygon File Format

S

SM: Streaming Multiprocessors

Figure 1 Logo de Testia

1. Testia

1.1. Présentation

Testia (Figure 1) est une filiale d'Airbus, spécialisée dans le contrôle non-destructif et l'intégrité structurelle. La société existe depuis plus de 30 ans et est implantée dans plus de 17 pays, dont la France, l'Espagne, l'Allemagne, le Mexique, le Canada. Elle a réalisé environ 26M de chiffre d'affaires en 2023 en France. Testia emploie plus de 400 collaborateurs dans le monde. Elle possède aussi plusieurs centres de formation et d'accréditation. Testia a pour objectif de renforcer la sécurité dans l'aviation en proposant plusieurs types de contrôles non destructifs:

- Contrôle par ressuage
- Contrôle magnétique
- Test par ultrasons
- Contrôle par courants de Foucault
- Tomographie
- Contrôle par infrarouge

Ces contrôles peuvent être utilisés dans l'aéronautique, l'automobile, les énergies renouvelables, la défense, …

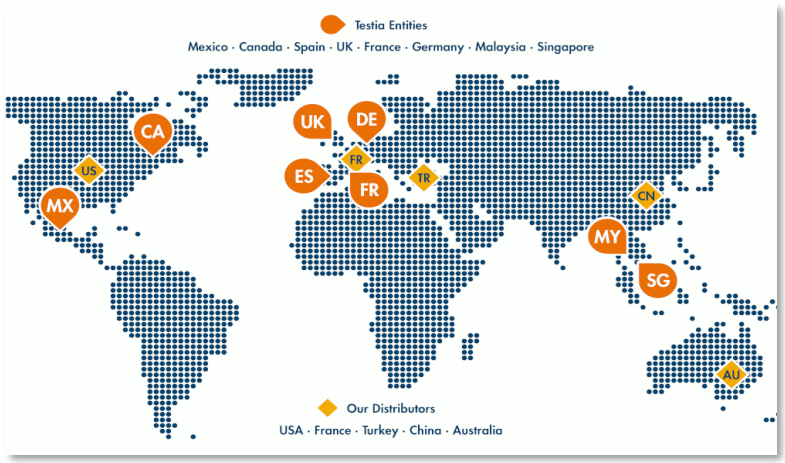

Figure 2 Carte des sites et distributeurs Testia

1.2. Testia Rennes

J'ai effectué ce stage au sein de Testia Rennes. Au début de l'année 2024, l'entreprise *Edixia Automation*, située à Vern-sur-Seiche (35), a été acquise par Testia pour son expertise en contrôle par vision industrielle. L'entreprise développe plusieurs produits :

- Station de contrôle de jeu et affleurement pour automobiles
- Banc de contrôle de défaut de disque de frein
- Contrôle de fibres de carbone
- Contrôle de rivets

1.3. Service R&D

Durant ce stage, j'ai rejoint le service **R&D Software** (*RDSW*), composé de 8 personnes. Ce service regroupe plusieurs compétences en optique, électronique et développement logiciel. L'équipe est responsable du développement et de l'amélioration des produits existants, tant du point de vue logiciel que matériel. Cela passe par l'intégration de nouveau matériel aux produits et la conception de nouvelles fonctionnalités pour rester compétitif. Le service RDSW est aussi responsable des migrations logicielles des systèmes.

1.4. Produits de la gamme Testia Rennes

Testia Rennes propose différents produits d'inspection visuelle, on retrouve par exemple :

- Le Système 3DCast (Figure 3) : Vérification de différentes pièces de fonderie, tel que des disques de freins, des culasses ou encore des blocs moteurs. Différents types de défauts peuvent être détectés : manque/surplus de matière, traces de chocs, etc.
- Le banc de test Celia (Figure 4) : Contrôle photométrique et géométrique des projecteurs (Phares de voitures, etc.), et qui permet de les régler afin d'être en conformité avec la législation.

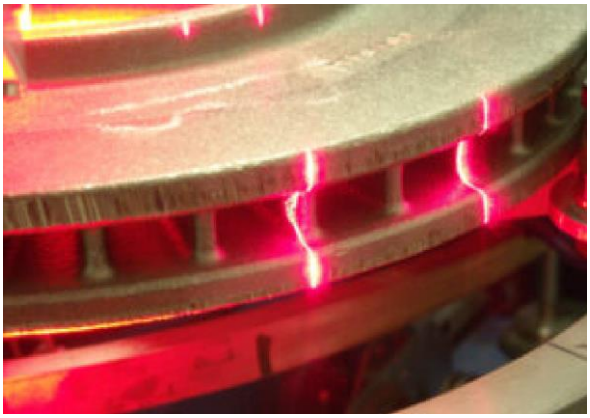

Figure 3 Système 3dCast

Figure 4 Banc de test Celia

- Le dispositif AF-Inspect (Figure 6) : Le système AF-Inspect réalise une inspection automatisée en temps réel des matériaux composites déposés lors des processus AFP/ATL, détectant les défauts tels que les écarts, replis, torsades, et débris étrangers pour garantir la qualité de fabrication.
- Scan-Rivet et Snap-Rivet (Figure 5) : Solutions de contrôle et de mesure adaptées aux procédés d'assemblage par rivetage, ils répondent aux contraintes des lignes de production des milieux de l'automobile et de l'aéronautique

Figure 6 AF-Inspect

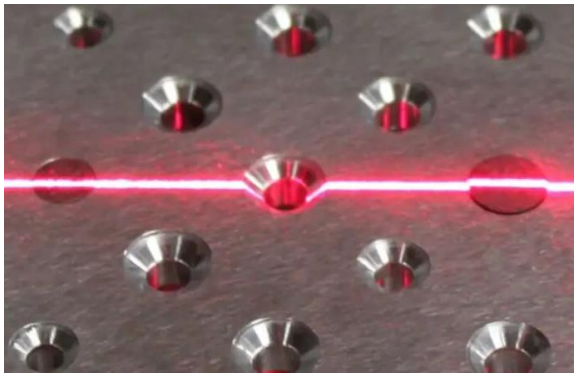

Figure 5 Scan-Rivet

1.5. Responsabilités Sociétales des Entreprises

1.5.1. Gouvernance de l'organisation

L'entreprise met en œuvre une gouvernance alignée sur les principes de la Responsabilité Sociétale des Entreprises (RSE). Elle a défini une stratégie claire, communiquée aux collaborateurs et déployée selon un plan d'actions précis. Les *publications des Essentials 2024* (objectifs 2024) illustrent cette démarche en alignant les actions de l'entreprise avec ses objectifs stratégiques, tout en impliquant activement les équipes dans leur réalisation. Des indicateurs environnementaux, sociaux et économiques sont suivis régulièrement pour évaluer l'impact des actions menées et assurer une transparence totale auprès des parties prenantes. Les dirigeants adoptent des valeurs en cohérence avec le Développement Durable et veillent à appliquer les réglementations en vigueur dans les territoires où l'entreprise opère.

1.5.2. Relations et Conditions de Travail

L'entreprise respecte scrupuleusement les droits sociaux, économiques et culturels, en garantissant des conditions de travail justes et appropriées conformément aux législations en vigueur et aux normes définies dans les conventions collectives. Afin de favoriser un dialogue social équilibré, des réunions du comité social et économique (CSE) sont organisées chaque mois, permettant ainsi des échanges réguliers entre les collaborateurs et la direction. De plus, des négociations d'accords d'entreprise sont menées pour répondre aux besoins des employés et améliorer leurs conditions de travail. En matière de santé et de sécurité, l'entreprise met en œuvre toutes les mesures nécessaires en fournissant des équipements de protection

individuelle (EPI) adaptés et en déployant un plan de prévention rigoureux pour limiter les risques professionnels et psychosociaux. Enfin, l'entreprise s'engage activement dans le développement du capital humain en assurant le développement des compétences et l'employabilité de chaque collaborateur.

1.5.3. Environnement

L'entreprise s'engage à réduire son impact environnemental en identifiant et en limitant les pollutions générées par ses activités, en optimisant sa consommation de ressources comme les matières premières et les énergies, tout en favorisant le digital pour limiter les consommables. Elle prend également des mesures concrètes pour réduire les impacts climatiques, notamment par la limitation des déplacements professionnels.

2. Sujet du stage

L'aquisition des images 3D se fait comme suit : Le laser projette une raie lumineuse sur l'objet, et la caméra capture son image, où la courbe correspond à la hauteur de l'objet. Chaque point de cette courbe est converti en une mesure de hauteur. En répétant cette opération sur plusieurs lignes, une image 3D complète est reconstruite, avec chaque pixel représentant une hauteur (Figure 7).

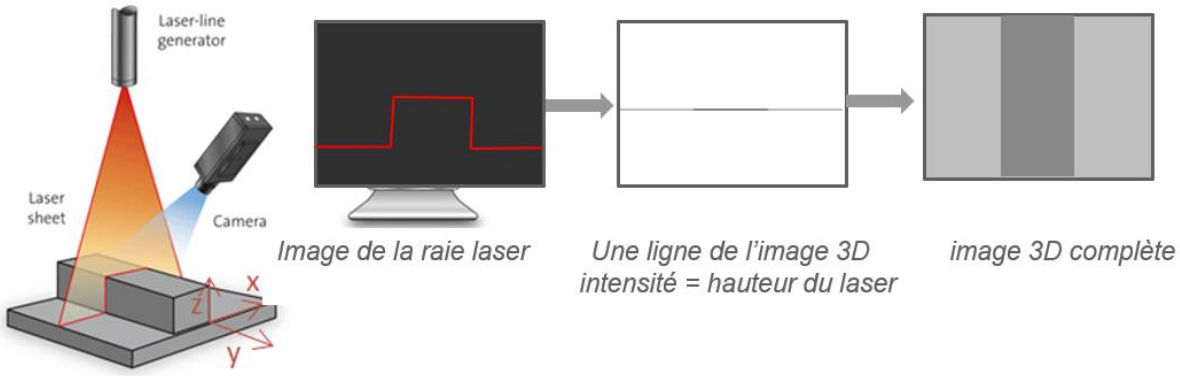

Figure 7 Reconstruction 3D à partir d'un laser ligne et d'une caméra

À Testia Rennes, les caméras associées à des lasers de leurs machines sont connectées à une carte FrameGrabber (FG), elle-même reliée à l'ordinateur, permettant de numériser, traiter et transférer les données d'image en temps réel vers un ordinateur hôte grâce à une couche logicielle appelée COAXLINK, qui gère cette carte (Figure 8). Ces images sont ensuite traitées par le logiciel IDEEPIX, qui applique divers traitements, tels que des moyennes locales et des filtres médians.

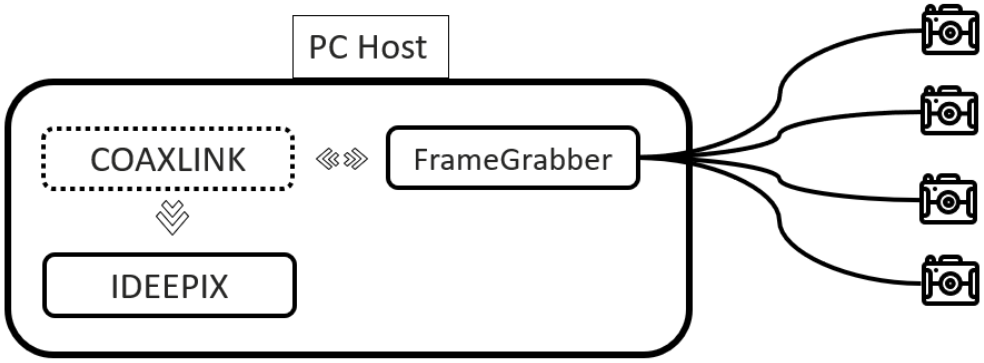

Figure 8 Architecture du système

Ces traitements sont actuellement effectués sur le CPU, ce qui allonge le temps d'exécution en raison de la nature séquentielle des calculs. Pour améliorer les performances, il est devenu nécessaire d'explorer l'utilisation d'une carte graphique (GPU), spécialement conçue pour les calculs parallèles, contrairement au CPU qui effectue les calculs de manière séquentielle.

Ma mission consistait à mener une recherche approfondie sur CUDA, une technologie développée par NVIDIA permettant de programmer leurs GPU, à développer de nouveaux algorithmes d'extraction de raies laser intégrés dans COAXLINK, et à effectuer des tests comparatifs entre CPU et GPU. Enfin, j'ai identifié les filtres et fonctions les plus exigeants en temps dans IDEEPIX afin de planifier leur adaptation pour une exécution sur GPU.

Le GPU cible est un NVIDIA RTX GeForce 4000 Quadro intégré dans le système, mais l'équipe m'a fourni un GPU NVIDIA RTX GeForce 3060 pour effectuer tous mes tests et travaux.

Durant mon stage, j'ai géré mon emploi du temps de manière autonome. À chaque étape, j'ai organisé des réunions avec mes tuteurs et le responsable d'équipe pour présenter l'avancement de mon travail et définir les étapes suivantes. J'ai également utilisé l'outil Git pour gérer mon projet. Le diagramme de Gantt est présenté dans la Figure 9.

	Semaines																							
Work Breakdown Structure	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	Progress Bar
Introduction à CUDA et configuration de l'environnement de travail																										100%
Réaliser une étude comparative entre CUDA et d'autres plateformes																										100%
Développement des 3 algorithmes d'extraction de raies laser																										100%
Découverte de Nsight Compute et apprentissage des techniques d'optimisation du code																										100%
Apprendre le concept d'occupation pour optimiser le temps de traitement																										100%
Concevoir un quatrième algorithme pour l'extraction des raies laser																										100%
Expérimenter l'utilisation des "Streams" sur l'un des algorithmes d'extraction de raies laser																										70%
Compilation avec CUDA pour générer une DLL 32 bits																										100%
Début de l'application de CUDA au filtre médian																										100%
Intégration de la fonction de filtre médian CUDA dans le logiciel IDEEPIX																										100%
Début de l'application de CUDA au filtre de moyenne locale																										20%
Analyser les causes de la variation du temps d'exécution d'une fonction CUDA lors de multiples appels																										100%
Retester les algorithmes d'extraction de raies laser à fréquence maximale																										100%
Analyser le comportement du GPU face à l'augmentation de sa température																										100%
Analyser le comportement du GPU en exécution simultanée de plusieurs programmes CUDA																										100%
Début de l'intégration de CUDA à l'algorithme d'extraction de raies laser existant dans le système TESTIA																										100%
Implémentation de la fonction de dilatation avec CUDA																										100%
Implémentation de la fonction de erosion avec CUDA																										100%
Evaluer les bénéfices GPU Vs CPU sur différentes tailles de fenêtres																										100%
Travaille sur le rapport de stage																										100%
Tester les fonctions implémentées avec CUDA avec l'IA d'IDEEPIX afin d'identifier les éventuels conflits.																										100%
Préparer la présentation de mon stage																										100%
Expliquer aux membres de l'équipe mon projet complet afin qu'ils puissent le gérer par la suite																										100%

Figure 9 Diagram de Gantt

3. CUDA

3.1. Présentation

CUDA est une technologie propriétaire de GPGPU c'est-à-dire utilisant un processeur graphique (GPU) pour exécuter des calculs généraux à la place du processeur central (CPU). CUDA permet de programmer des GPU en C, C++ et Python. Elle est développée par NVIDIA, initialement pour ses cartes graphiques. CUDA offre un moyen efficace de paralléliser les calculs. Dans le domaine du traitement d'image, où les opérations sont souvent répétitives sur de grandes quantités de données, CUDA permet de distribuer le travail sur des milliers de cœurs, optimisant ainsi le temps de calcul grâce à l'architecture du GPU (Figure 10) qui est conçue pour réaliser des calculs parallèles. Tandis qu'un CPU traite les pixels un par un, un GPU, grâce à CUDA, peut traiter des milliers de pixels simultanément.

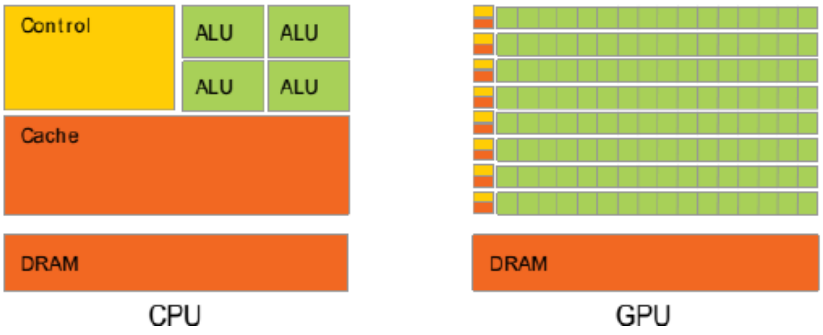

Figure 10 Architecture CPU vs GPU

3.2. Comparaison entre CUDA, OpenCL et SYCL

Il existe plusieurs plateformes, comme OpenCL et SYCL, qui permettent de programmer les GPUs. J'ai donc dû effectuer des recherches pour déterminer laquelle serait la plus adaptée [1].

CUDA :

Avantages :

- Facilité d'utilisation : Facile à apprendre en termes de programmation
- Documentation : Il existe beaucoup de documentation, tutoriaux et d'exemples
- Performance : Il est spécifique pour NVIDIA ce qui le rend performant sur cette carte
- Langage de programmation : Disponible en C, C++ et Python
- Intégration : Intégration aisée dans des applications existantes écrites en C++

Inconvénients :

- Compatibilité : Ne peut marcher que sur les cartes NVIDIA

SYCL :

Avantages :

- Facilité d'utilisation : Facile à comprendre en termes de programmation
- Langage de programmation : Disponible en C++
- Intégration : Intégration aisée dans des applications existantes écrites en C++
- Compatibilité : Fonctionne sur des GPU de différents fabricants

Inconvénients :

- Documentation : Il existe très peu de documentation, tutoriaux et d'exemples pour l'apprendre
- Performance : Peut ne pas tirer pleinement parti des optimisations spécifiques au matériel par rapport à CUDA sur les GPU NVIDIA

OPENCL :

Avantages :

- Compatibilité : Fonctionne sur des GPU de différents fabricants
- Documentation : Il existe beaucoup de documentation, tutoriaux et d'exemples pour l'apprendre
- Intégration : Peut-être intégrer dans un code C++

Inconvénients :

- Facilité d'utilisation : Plus difficile à apprendre que CUDA
- Langage de programmation : Disponible en C
- Performance : Peut ne pas tirer pleinement parti des optimisations spécifiques au matériel par rapport à CUDA sur les GPU NVIDIA

Aspect	CUDA	OpenCL	SYCL
Compatibilité de la plateforme	GPU NVIDIA uniquement	Multi-Plateforme	Multi-Plateforme
Langage de programmation	C, C++, Python	C, liaisons pour d'autres langues	C++
Facilité d'utilisation	Facile	Modérée	Modérée
Performance	Élevé sur NVIDIA	Variée	Compétitive
Documentation	Extensive	Compréhensive	Limitée

Tableau 1 Comparaison entre CUDA, OpenCL et SYCL

Après cette comparaison (Tableau 1), nous avons décidé de travailler avec CUDA car il offre les moyens les plus simples pour le prendre en main et intégrer l'utilisation du GPU dans nos systèmes.

3.3. Fonctionnement du code

Le code est divisé en deux parties :

- **Host (code principal)**, exécuté sur le CPU.
- **Device (code du kernel)**, exécuté sur le GPU.

Le code de l'Host appelle celui du Device, qui s'exécute directement sur le GPU. Cependant, le Device ne dispose pas d'un système d'exploitation et ne peut pas gérer sa mémoire de manière autonome. C'est donc l'Host qui s'occupe de la gestion de la mémoire du GPU [2] :

- Il déclare une variable du type pointeur sur le type de données à manipuler type *ptr ;
- Il alloue de la mémoire sur le GPU grâce à la fonction cudaMalloc (&ptr, nombre_octets) :
    - Réserve un espace mémoire sur le GPU, sous le contrôle du CPU.
    - Stocke l'adresse de cette zone mémoire dans le pointeur ptr (Figure 11)

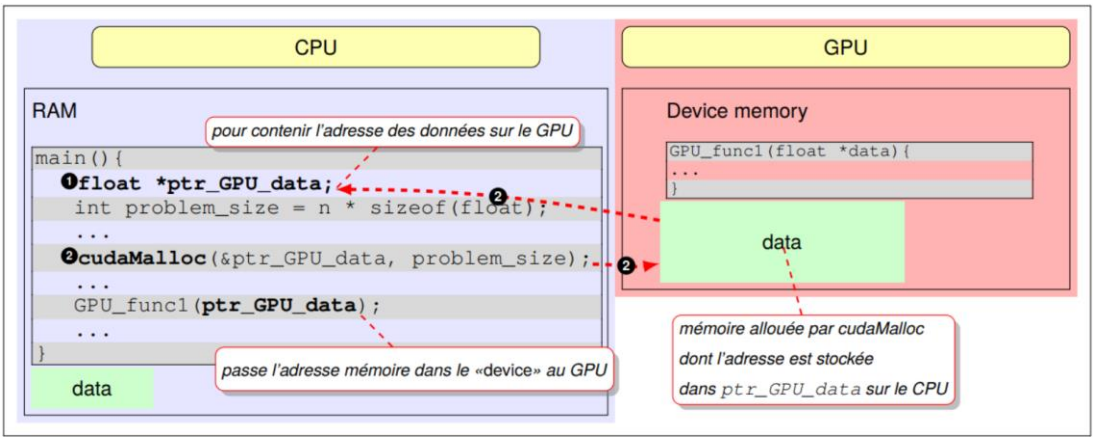

Figure 11 Allocation de mémoire dans le GPU

Les données sont transférées du CPU vers la mémoire du GPU via une opération «cudaMemcpy()», qui utilise le bus PCI (Figure 12).

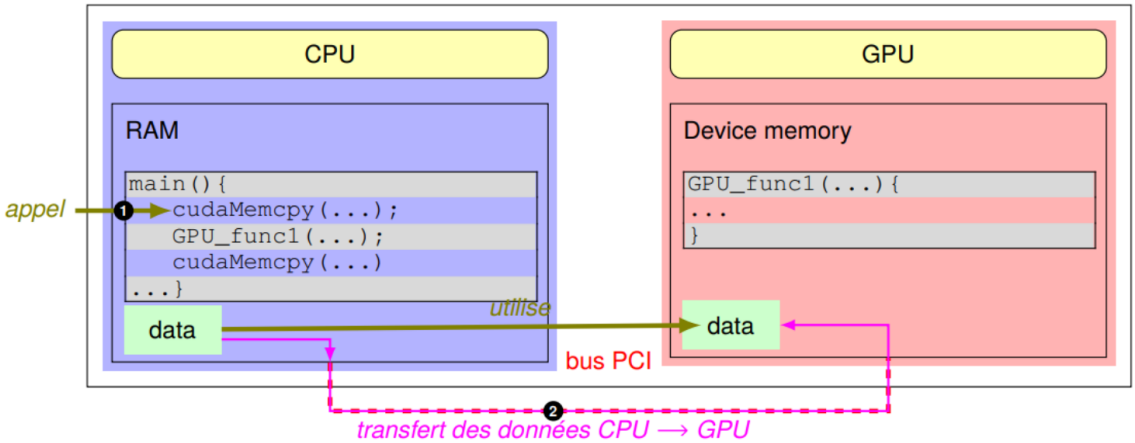

Figure 12 Transferts des données du CPU vers GPU

La fonction appelée pour s'exécuter sur le GPU, appelée kernel, utilise les données déjà transférées dans la mémoire du GPU (Figure 13).

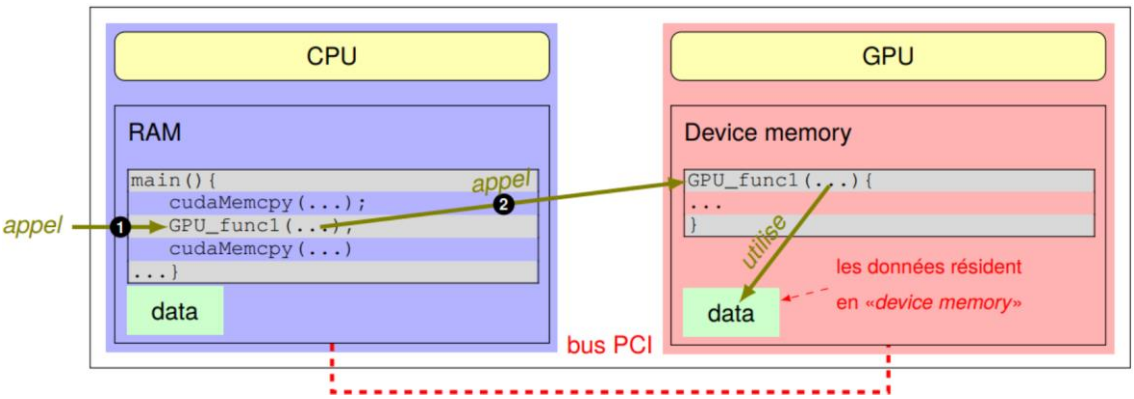

Figure 13 Appel du kernel

Pendant l'exécution du kernel, celui-ci modifie les données présentes dans sa mémoire, préalablement envoyées par le CPU (Figure 14).

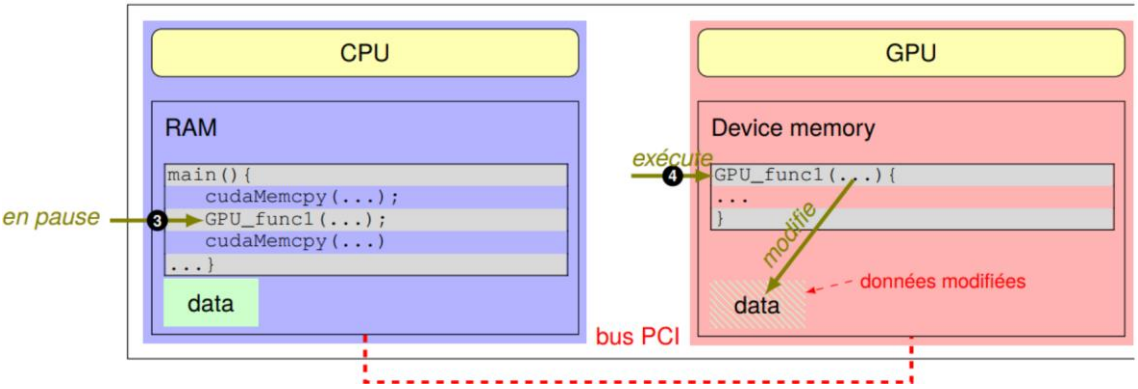

Figure 14 Modifications des données par le GPU

Une fois l'exécution du kernel terminée, le contrôle revient au code hôte (Figure 15).

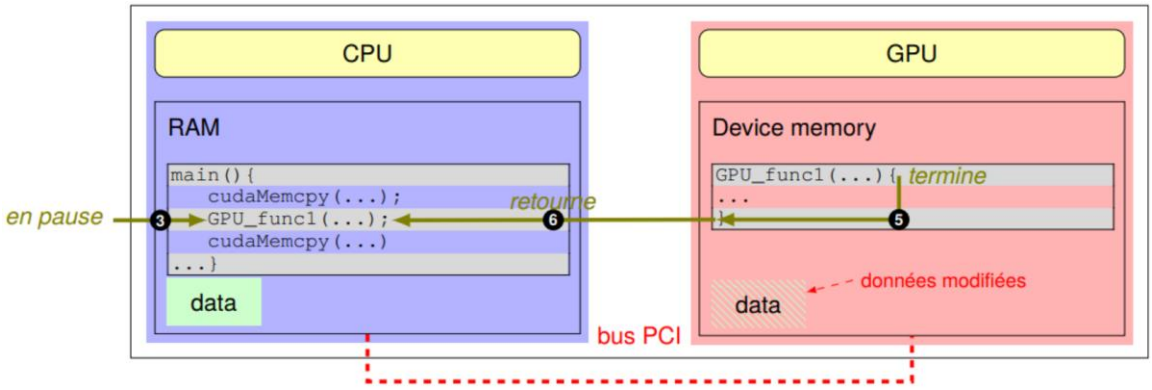

Figure 15 Retourne aux code Host

Enfin, une nouvelle opération « cudaMemcpy() » est utilisée pour rapatrier les données modifiées depuis la mémoire du GPU vers celle du CPU (Figure 16).

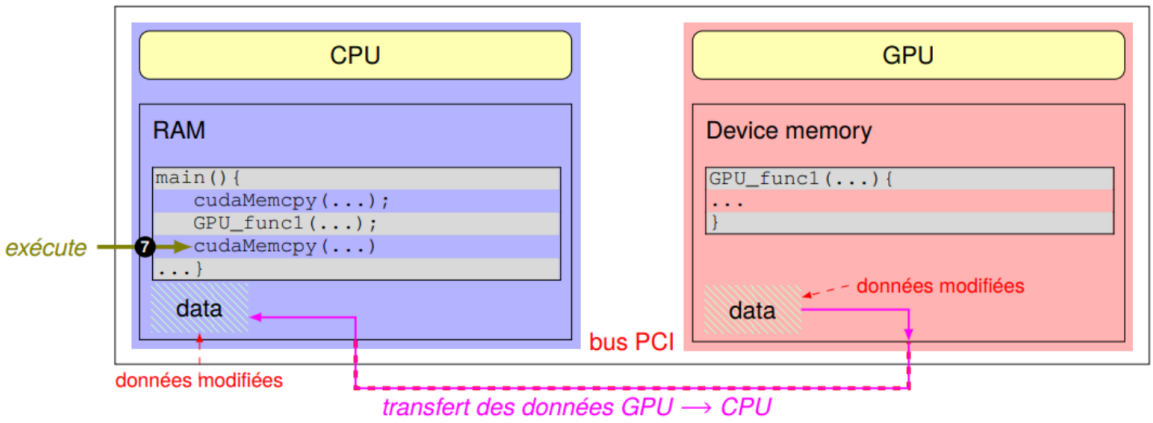

Figure 16 Retourne des données modifiés vers le CPU

L'appel de la fonction kernel se fait comme suit :

```
MyFunction<<<NbBlocks, NbThreads>>> (param1, param2, …);
```

En CUDA, les threads sont groupés en blocs et les blocs sont organisés en grilles (Figure 17). Ici, NbBlocks correspond au nombre de blocs, tandis que NbThreads correspond au nombre de threads par bloc, ce qui signifie que cette fonction sera exécutée NbBlocks x NbThreads fois en parallèle. Il faut savoir qu'actuellement, le nombre maximal de blocs est de 2 147 483 647 (ce nombre est valable pour tous les GPU avec une capacité de calcul supérieure ou égale à 3.0) et 1024 est le nombre maximal de threads par bloc (ce nombre est valable pour tous les GPU avec une capacité de calcul supérieure ou égale à 2.0). À titre d'exemple, la Quadro RTX 4000, qui utilise l'architecture « Turing », a une capacité de calcul de 7.5, et la RTX 3060, qui utilise l'architecture « Ampere », a une capacité de calcul de 8.6.

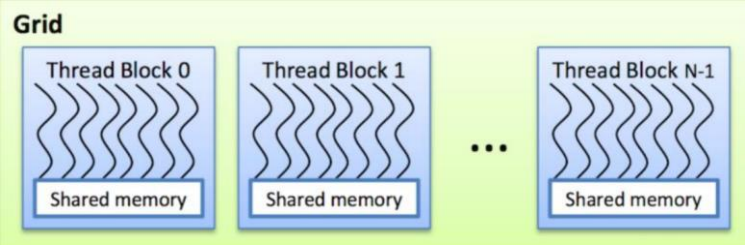

Figure 17 Architecture d'une grille

CUDA fournit des fonctions utiles telles que threadIdx.x et blockIdx.x, qui représentent l'index du thread ou du bloc en cours d'exécution. De plus, blockDim.x et gridDim.x correspondent respectivement aux valeurs du nombre de threads par bloc et du nombre de blocs dans une grille, telles qu'elles ont été spécifiées en paramètres lors du lancement du noyau CUDA. Le kernel est défini avec le mot-clé « __global__ ».

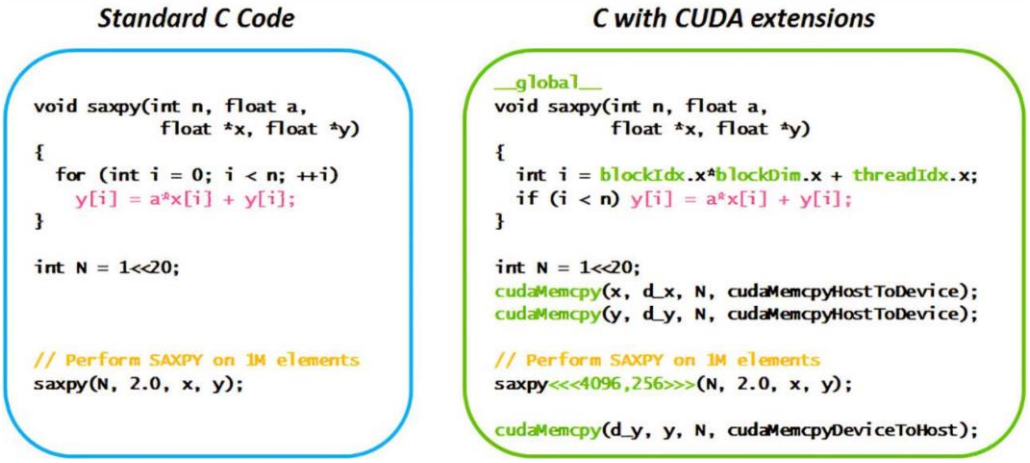

Figure 18 Différence entre le code CPU et le code CUDA

Dans le code illustré dans la Figure 18, le kernel aura en tout 4096 x 256 = 1 048 576 threads, et donc la variable « i » représente l'index du thread que le kernel exécute.

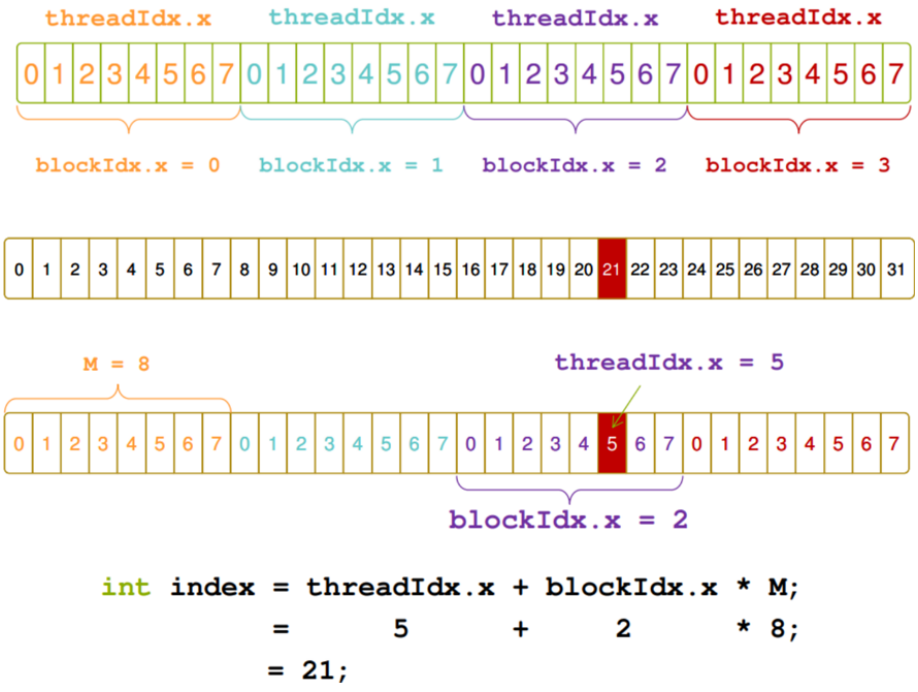

Figure 19 Calcul de l'indice du thread

Dans la Figure 19, "M" désigne le nombre de threads par bloc, équivalent à "blockDim.x" dans le kernel. Pour calculer l'index du thread exécuté par le kernel, il suffit de multiplier le nombre de threads dans un bloc par l'index du bloc auquel appartient le thread en cours d'exécution, puis d'ajouter l'index du thread dans son propre bloc.

```
const int K = 16; // tile size is K*K
__global__ void transpose_parallel_per_element(float in[], float out[])
{
  int i = blockIdx.x * K + threadIdx.x; // column
  int j = blockIdx.y * K + threadIdx.y; // row

  out[j + i*N] = in[i + j*N];
}
```

```
dim3 blocks(N/K, N/K); // blocks per grid
dim3 threads(K, K); // threads per blocks

transpose parallel per element<<<blocks, threads>>>(d in, d out);
```

Figure 20 Exécution bidimensionnelle des blocs et des threads

Dans la Figure 20, les blocs et les threads sont organisés en 2 dimensions à l'aide de la structure « dim3 », permettant de déclarer les variables sur 1, 2 ou 3 dimensions (par exemple, « dim3 threads(K, K) » est équivalent à « dim3 threads(K, K, 1) »). Dans cet exemple, le kernel utilise deux variables, i et j, qui correspondent aux indices du thread en colonne et en ligne, facilitant ainsi le traitement d'images en 2D. Il est cependant essentiel de respecter certaines limitations : la taille maximale d'un bloc est de 1024 en x, 1024 en y et 64 en z, tandis que la taille maximale d'une grille est de 2147483647 en x, 65535 en y et 65535 en z. Il est également important de noter que le produit des dimensions x × y × z des threads ne doit pas dépasser 1024 (le nombre maximal de threads par bloc) ou 2147483647 si l'on définit le nombre de blocs. Par exemple, « dim3 threads(32, 16, 16) » serait invalide, car 32 × 16 × 16 = 8192, ce qui dépasse la limite de 1024 [3].

3.4. Fonctionnement du matériel

Les cartes graphiques sont dotées d'architectures variées. Par exemple, la RTX 3060 repose sur l'architecture « Ampere » tandis que la RTX 4000 Quadro utilise « Turing ». Bien que ces architectures présentent des différences en termes de performances et de spécifications, elles partagent un socle commun : le principe de fonctionnement des cœurs CUDA et le calcul parallèle.

Figure 21 Architecture "Turing"

Dans la Figure 21, l'architecture Turing est décrite avec ses principaux composants matériels :

- **PCI Express Host Interface** : Interface PCIe permettant la communication entre le GPU et le CPU ou d'autres périphériques sur la carte mère.
- **SM (Streaming Multiprocessor)**: Multiprocesseur de streaming. Chaque SM contient des cœurs CUDA, des unités de mémoire partagée, des registres, et des unités spécialisées comme les Tensor Cores. Les SMs sont responsables de l'exécution parallèle des threads.
- **HBM2 (High Bandwidth Memory 2 / Global memory)** : Mémoire à large bande passante. HBM2 est utilisée pour offrir une très haute bande passante mémoire au GPU, essentielle pour les charges de travail intensives en données.

Figure 22 Architecture d'un SM

- **Warp Scheduler (32 threads/clk)** : Le scheduler de warps. Un warp est un groupe de 32 threads exécutés en parallèle. Le warp scheduler décide quels warps doivent être exécutés à chaque cycle d'horloge. Cela permet de maximiser l'utilisation des ressources disponibles et de masquer les latences.
- **Dispatch Unit (32 threads/clk)** : L'unité de dispatch gère l'envoi des instructions aux différents cœurs CUDA ou autres unités d'exécution au sein de la SM. Elle fonctionne en étroite collaboration avec le warp scheduler pour assurer une exécution fluide et efficace des instructions.
- **Register File (16,384 x 32-bit)** : Les registres sont des mémoires rapides et locales utilisées par les threads pour stocker des variables et des données temporaires. Chaque SM dispose de 16 384 registres de 32 bits chacun, accessibles rapidement par les cœurs CUDA.
- **INT32** : Les unités de traitement des entiers de 32 bits. Elles exécutent les opérations arithmétiques et logiques sur des entiers de 32 bits.
- **FP32** : Les unités de traitement des flottants en précision simple (32 bits). Elles exécutent les opérations en virgule flottante de précision simple.
- **FP64** : Les unités de traitement des flottants en double précision (64 bits). Elles exécutent les opérations en virgule flottante de double précision, souvent utilisées dans les calculs scientifiques et techniques.
- **192KB L1 Data Cache / Shared Memory** : Une mémoire cache de niveau 1 dédiée aux données, qui est partagée entre les threads d'un SM. Cette mémoire peut également être utilisée comme mémoire partagée, où les threads d'un même bloc peuvent communiquer et partager des données.

Le kernel envoyé au « Device » est organisé de manière hiérarchique :

- Threads identiques
- Threads organisés en blocs, chaque bloc s'exécutant sur un seul multiprocesseur
- Blocs organisés au sein d'une grille, qui répartit ses blocs sur tous les multiprocesseurs (Figure 23)
- Grille et blocs 1D, 2D ou 3D
    - Une grille 2D de blocs 2D
    - Une grille 2D de blocs 1D
    - …

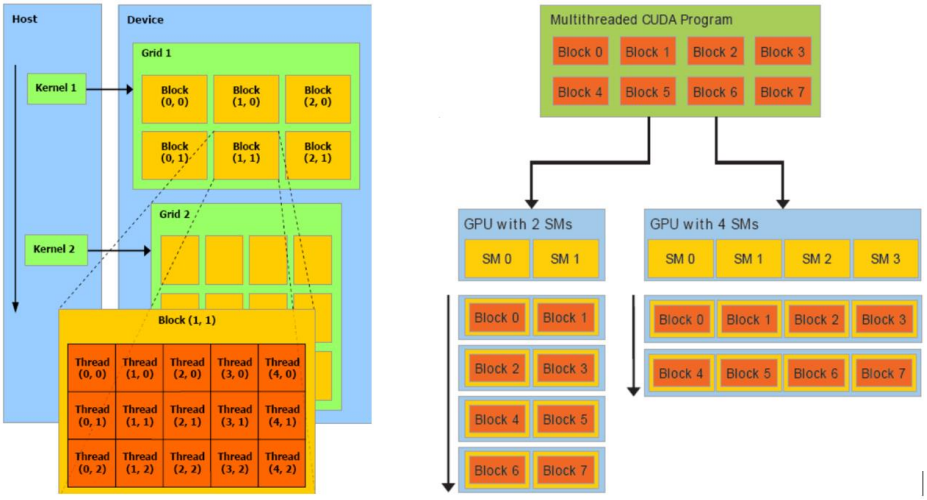

Figure 23 Organisation des threads et des blocs

Les blocs seront répartis sur les différents SM. À ce niveau, les blocs de threads sont divisés en groupes de 32 threads, appelés « warps ». Ensuite, ces « warps » passent vers le « Warp Scheduler », qui décide quels « warps » doivent être exécutés à chaque cycle d'horloge. Le « warp » est ensuite envoyé par le « Dispatch Unit » vers un ensemble dédié d'unités d'exécution arithmétiques (CUDA Cores) pour exécuter les instructions (Figure 24) [4].

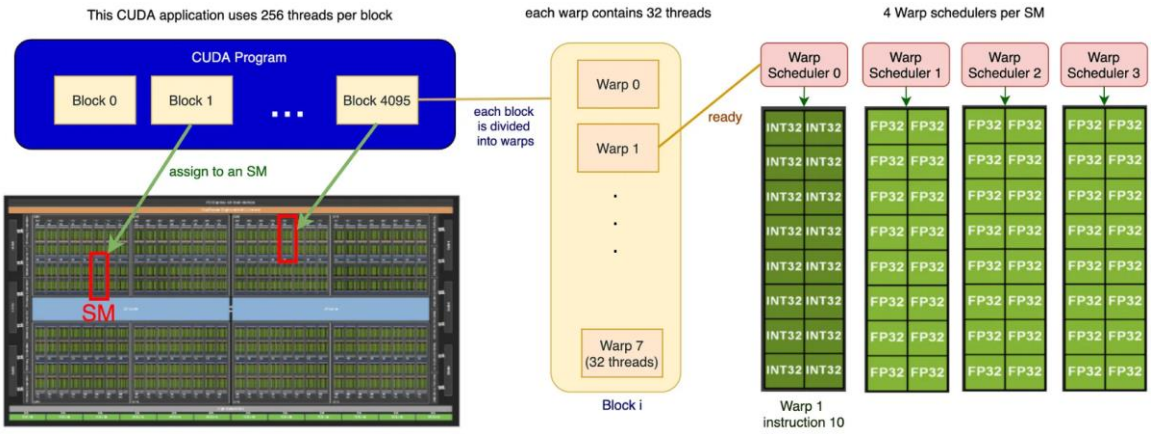

Figure 24 Mécanisme de répartition des threads sur les cœurs du GPU

4. Traitement d'images

4.1. COAXLINK

COAXLINK est une couche logicielle qui gère les opérations effectuées par la carte FG. C'est sur cette couche que l'algorithme d'extraction de raies laser est exécuté. Pour commencer à évaluer les bénéfices de l'utilisation de CUDA, on m'a fourni une image concaténée de raies laser de taille 1936x152000 pixels, ainsi qu'une image individuelle de raie laser de taille 1936x504 pixels. Ces images sont en niveaux de gris, avec les raies laser représentées en blanc. Mon objectif était de développer des algorithmes d'extraction de raies laser et de reconstruire

ces données sous forme d'images 3D au format PLY. Pour chaque algorithme que j'ai conçu, j'ai créé une version CPU et une version CUDA afin de comparer leurs performances. J'ai utilisé la bibliothèque OpenCV pour réaliser ces algorithmes. J'ai développé un programme pour tester la différence en sortie du programme avec CUDA et sans CUDA pour chaque algorithme, les résultats des tests montrent une parfaite correspondance entre les sorties des deux versions, suggérant que l'utilisation du GPU n'a pas introduit d'erreurs dans les calculs. J'ai également utilisé le logiciel Player3D pour visualiser les fichiers au format PLY, ce qui m'a permis d'observer l'image de sortie en 3D.

4.1.1. Algorithmes d'extraction de raies de laser

Algorithme 1 (calcul image par image)

Dans cet algorithme, j'ai pris une image d'une raie de laser (Dimension : 1936x504) et l'ai copiée plusieurs fois pour obtenir 100 images à traiter. L'algorithme consiste à traiter chaque image individuellement pour sauvegarder les coordonnées des raies dans des vecteurs. Au préalable, j'ai appliqué un filtre gaussien sur chaque image afin d'éliminer les bruits présents dans l'arrière-plan de l'image (Figure 25).

Figure 25 Image avant et après l'application du flou gaussien.

Dans mon code utilisant CUDA, j'ai utilisé le kernel dans trois fonctions : `gaussianBlurCUDA` (pour appliquer le filtre gaussien), `applyThresholdCUDA` (pour appliquer le seuil d'intensité de couleur à éliminer), et `findLaserLineCUDA` (pour remplir le vecteur avec les coordonnées de la raie). J'ai donc réalisé deux versions du code (l'une avec CUDA et l'autre sans CUDA) pour comparer les performances. En particulier, j'ai cherché à comparer la différence de rapidité entre l'exécution du kernel et son équivalent sans CUDA. L'algorithme utilise CUDA pour parcourir en parallèle les colonnes de l'image et vérifier si le pixel analysé présente une intensité blanche. Si c'est le cas, la coordonnée y de ce pixel est ajoutée au vecteur laser_line_y_vector, lequel regroupe plusieurs vecteurs, chacun correspondant à une image (Figure 26).

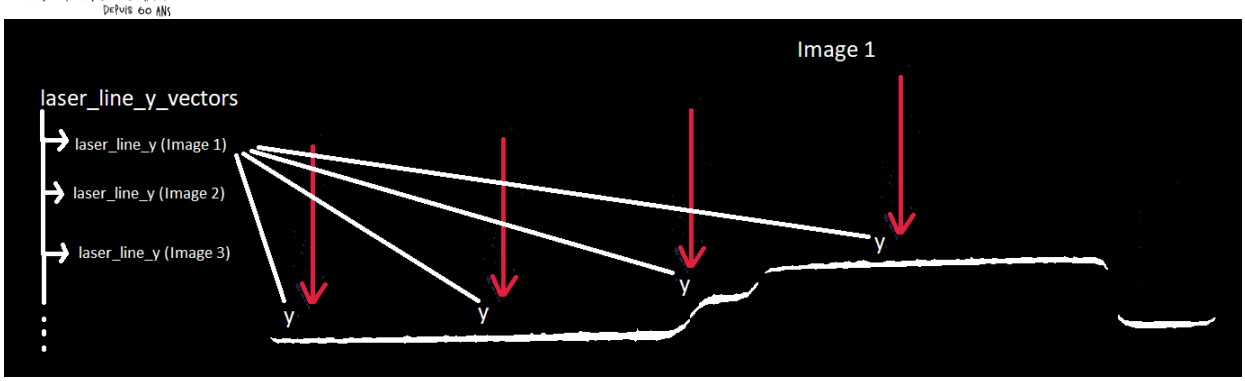

Figure 26 Fonctionnement du premier algorithme

	Avec CUDA	Sans CUDA
Temps de lecture des images	≈ 140.1 ms	≈ 140.1 ms
CUDA Init	≈ 112 ms	
Temps d'allouée de mémoire	≈ 0.22 ms	
Temps pour copier les données vers le GPU	≈ 7.3 ms	
Temps du kernel « applyGaussianBlur »	≈ 6.8 ms (x2573 plus rapide)	≈ 17500 ms
Temps du kernel « applyThreshold »	≈ 7.4 ms (x71 plus rapide)	≈ 526 ms
Temps du kernel « findLaserLine »	≈ 5.4 ms (x57 plus rapide)	≈ 308 ms
Temps pour copier les données vers le CPU	≈ 2.7 ms	
Temps pour Free data	≈ 0.44 ms	
Temps total du programme (sans chronos)	**≈ 310 ms** (x70 plus rapide)	**≈ 18. 9 Secondes**

Tableau 2 Comparaison des temps d'exécution de l'algorithme 1 avec et sans l'utilisation de CUDA

Comme on peut le constater dès le premier algorithme utilisant CUDA, il existe une différence significative dans le temps d'exécution entre le traitement sur CPU et celui sur GPU (Tableau 2). J'ai ensuite expérimenté en modifiant la taille du noyau du filtre gaussien (Figure 27) ainsi que la taille des images (Figure 28) pour évaluer l'impact sur les performances. Lorsque l'on exécute un programme en CUDA, la première fonction de l'API CUDA appelée déclenche l'initialisation du GPU (CUDA Init), un processus qui dure environ 100 ms. Cette initialisation n'a lieu qu'une seule fois pendant l'exécution du programme.

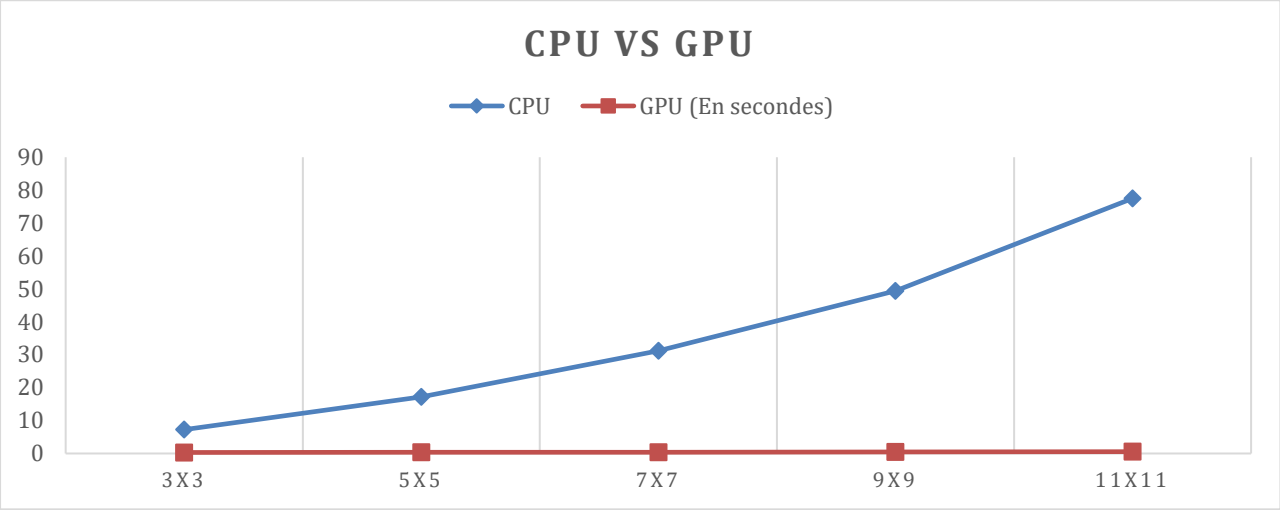

Figure 27 Temps du programme en fonction de la taille de la fenêtre du flou gaussien (image de 1936x504)

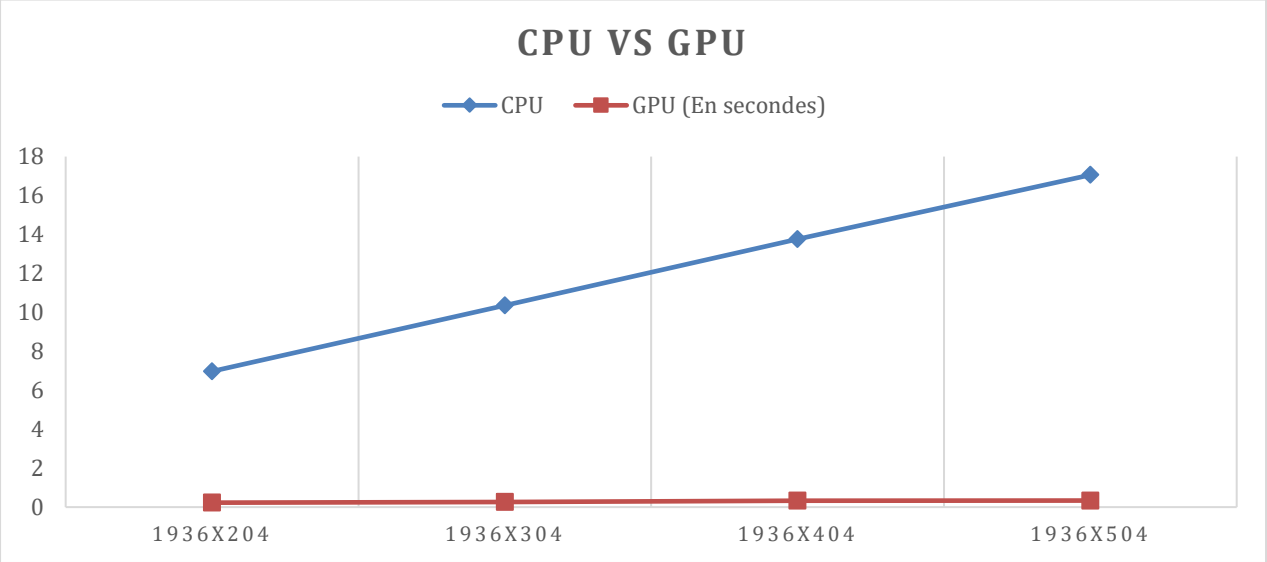

Figure 28 Temps du programme en fonction de la taille de l'image (fenêtre 5x5)

Ce qui m'a particulièrement intéressé dans ce test, c'est la performance du GPU, qui est restée relativement stable malgré l'augmentation des calculs, contrairement au CPU dont les performances ont été plus impactées. Voici dans la Figure 29 l'affichage du fichier PLY en sortie de l'algorithme 1 sur Player3D.

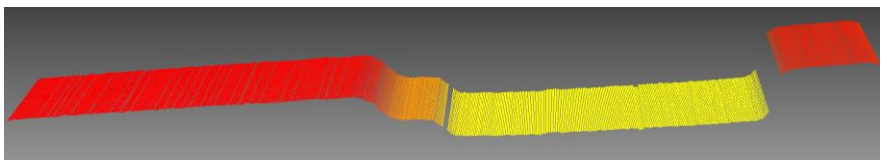

Figure 29 Image 3D générée par le premier algorithme

Algorithme 2 (*Scan complet*)

Dans cet algorithme, j'ai utilisé l'image concaténée en entrée pour l'analyser dans son intégralité et remplir les vecteurs de chaque ligne avec les coordonnées de tous les pixels

correspondants. L'image concaténée a été parcourue de manière parallèle, à la fois en lignes et en colonnes, chaque thread étant associé à un pixel unique. Le thread compare l'intensité du pixel à une valeur blanche : si le pixel répond au critère, sa coordonnée x est ajoutée au vecteur lines_x_coords[index_ligne][index_pixel], qui regroupe toutes les coordonnées x des pixels appartenant aux lignes de raies extraites, tandis que la coordonnée y est ajoutée au vecteur lines_y_coords[index_ligne][index_pixel] (Figure 30). Cela rend les vecteurs volumineux mais exhaustifs en termes de détails. Comme précédemment, j'ai développé deux versions du code, l'une utilisant CUDA et l'autre sans CUDA.

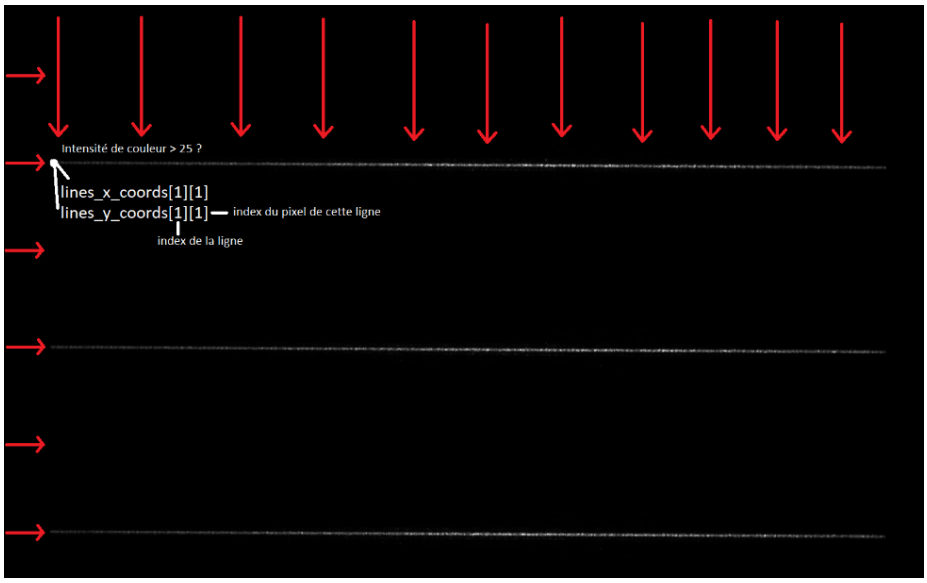

Figure 30 Fonctionnement du deuxième algorithme

	Avec CUDA	Sans CUDA
Temps de lecture de l'image (1936x152000)	≈ 300 ms	≈ 300 ms
CUDA Init	≈ 112 ms	
Temps d'allouée de mémoire	≈ 7.498 ms	
Temps pour copier les données vers le GPU	≈ 47.8 ms	
Temps du kernel	≈ 6.2458 ms (x83 plus rapide)	≈ 800 ms
Temps pour copier les données vers le CPU	≈ 39.92 ms	
Temps pour Free data	≈ 4.717 ms	
Temps total du programme (sans chronos)	**≈ 556.48 ms** (x1.97 plus rapide)	**≈ 1.1 s**

Tableau 3 Comparaison des temps d'exécution de l'algorithme 2 avec et sans l'utilisation de CUDA

Cette expérimentation met également en évidence une différence de temps d'exécution entre le traitement en CUDA et celui sur CPU (Tableau 3), bien que moins marquée que

précédemment. Cela s'explique principalement par le temps nécessaire pour transférer l'image du CPU vers le GPU et les données de sorties du GPU au CPU en CUDA, un processus ralenti par la taille importante de l'image. J'ai également effectué des tests en modifiant la taille de l'image afin d'évaluer l'impact sur les performances.

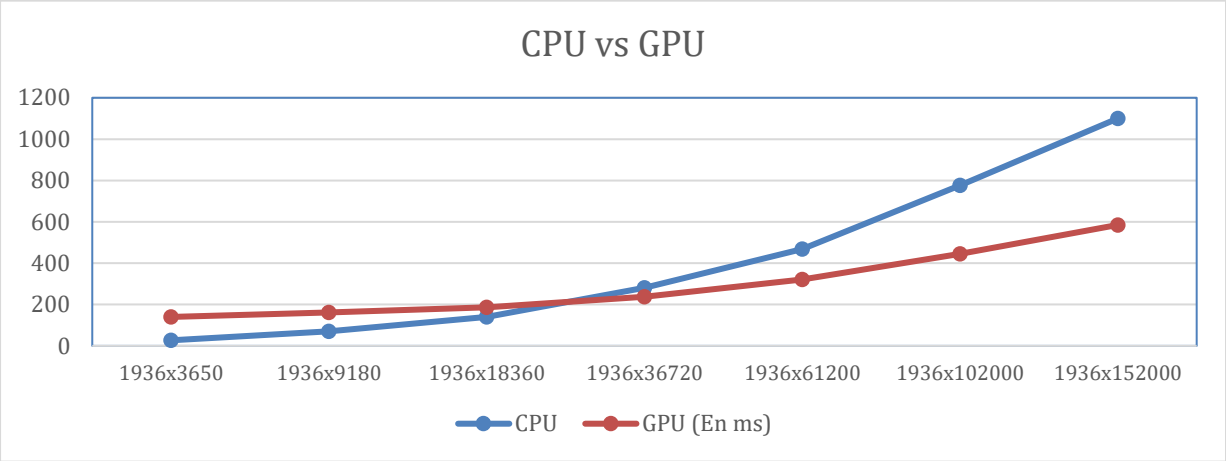

Figure 31 Temps du programme en fonction de la taille de l'image

Le CPU s'est révélé plus performant avec les tailles d'images faibles (Figure 31), ce qui permet de conclure que le CPU peut être plus rapide que le GPU lorsque le programme ne nécessite pas beaucoup de calculs. Voici dans la Figure 32 l'affichage du fichier PLY en sortie de l'algorithme 2 sur Player3D.

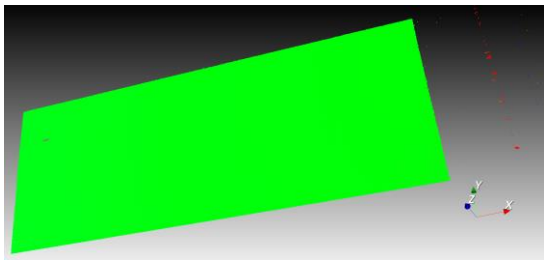

Figure 32 Image 3D générée par le deuxième algorithme

Algorithme 3 (*Scan vertical*)

Dans cet algorithme, j'ai parcouru l'image concaténée uniquement dans la direction vertical pour enregistrer la hauteur d'un seul pixel par abscisse, ce qui allège considérablement l'image en sortie. À chaque pixel, j'ai analysé les 10 pixels suivants verticalement afin de m'assurer de sélectionner celui ayant la meilleure intensité de couleur (Figure 33). Deux versions du code ont été développées : une implémentation avec CUDA et une autre sans CUDA.

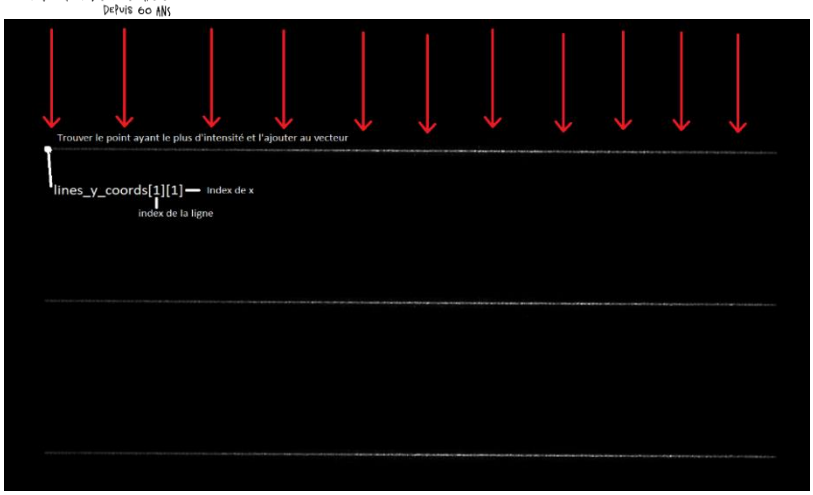

Figure 33 Fonctionnement du troisième algorithme

	Avec CUDA	Sans CUDA
Temps de lecture de l'image (1936x152000)	≈ 300 ms	≈ 300 ms
CUDA Init	≈ 112 ms	
Temps d'allouée de mémoire	≈ 6.566 ms	
Temps pour copier les données vers le GPU	≈ 49.9321 ms	
Temps du kernel	≈ 69.3882 ms (x70.56 plus rapide)	≈ 4896.6 ms
Temps pour copier les données vers le CPU	≈ 13.3756 ms	
Temps pour Free data	≈ 1.8173 ms	
Temps total du programme (sans chronos)	**≈ 570.385 ms** (x9.177 plus rapide)	**≈ 5234.6 ms**

Tableau 4 Comparaison des temps d'exécution de l'algorithme 3 avec et sans l'utilisation de CUDA

Dans ce test (Tableau 4), la complexité des calculs était plus élevée, ce qui explique les meilleures performances du GPU par rapport au CPU. De plus, le vecteur de sortie était plus léger, ce qui a permis un transfert des données vers le CPU plus rapide par rapport à l'algorithme 2.

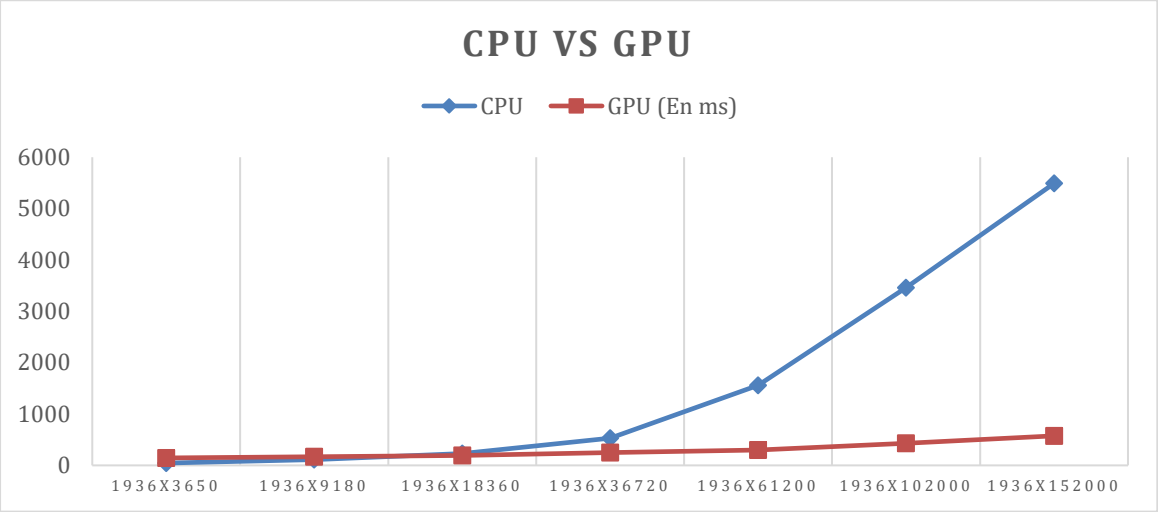

Figure 34 Temps du programme en fonction de la taille de l'image

D'après la Figure 34, en raison de la complexité de l'algorithme, le GPU s'est montré nettement plus performant à mesure que la taille de l'image augmentait.
Voici dans la Figure 35 l'affichage du fichier PLY en sortie de l'algorithme 3 sur Player3D.

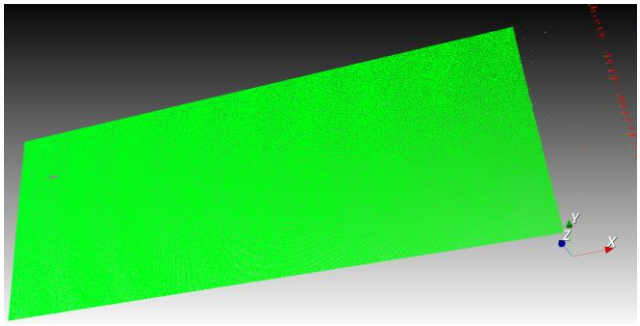

Figure 35 Image 3D générée par le troisième algorithme

Algorithme 4 (Scan complet avec comparaison)

Dans cet algorithme, j'ai parcouru l'image concaténée de manière complète (horizontalement et verticalement) et pour chaque pixel analysé, j'ai vérifié si les 10 pixels situés au-dessus et les 10 pixels situés en dessous étaient plus lumineux afin de conserver le pixel le plus lumineux (Figure 36). Ainsi, j'ai réalisé deux versions du code : une utilisant CUDA et l'autre sans CUDA.

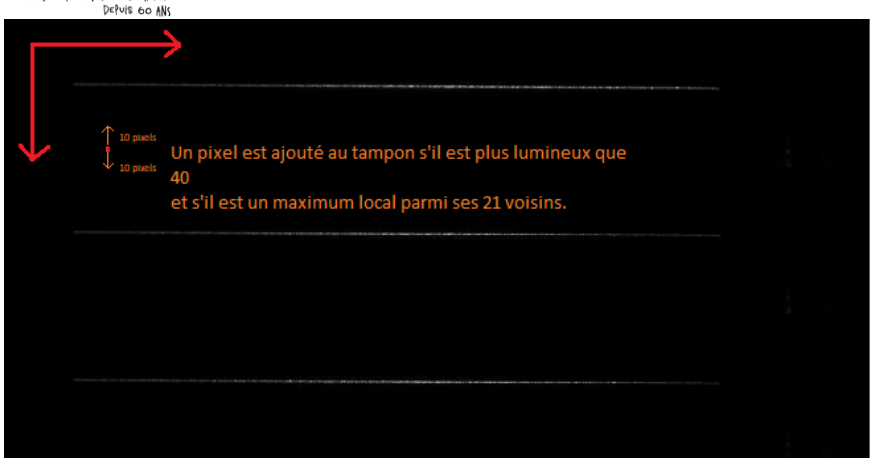

Figure 36 Fonctionnement du quatrième algorithme

	Avec CUDA	Sans CUDA
Temps de lecture de l'image (1936x152000)	≈ 300 ms	≈ 300 ms
CUDA Init	≈ 112ms	
Temps d'allouée de mémoire	≈ 8.07 ms	
Temps pour copier les données vers le GPU	≈ 47.69 ms	
Temps du kernel	≈ 43.83 ms (x57.72)	≈ 2530.28 ms
Temps pour copier les données vers le CPU	≈ 10.13 ms	
Temps pour Free data	≈ 1.184 ms	
Temps total du programme (sans chronos)	**≈ 510.73 ms** (x5.61)	**≈ 2865.72 ms**

Tableau 5 Comparaison des temps d'exécution de l'algorithme 4 avec et sans l'utilisation de CUDA

Un bon résultat a également été obtenu avec CUDA par rapport au CPU (Tableau 5), en raison de la complexité accrue des calculs.

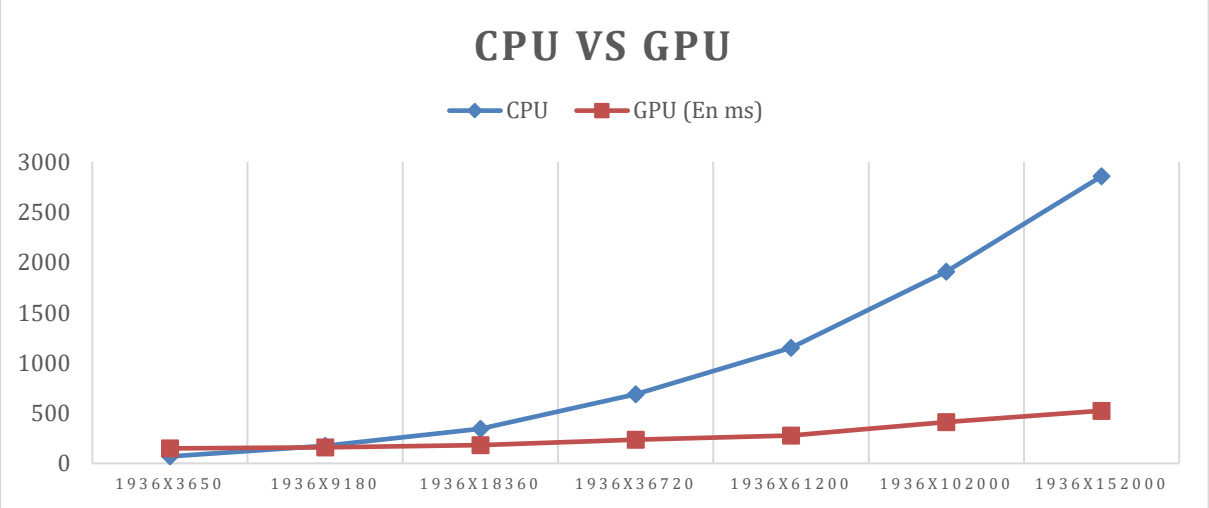

Figure 37 Temps du programme en fonction de la taille de l'image

De la même manière, la Figure 37 démontre que la complexité du code influence significativement la différence de performance entre le CPU et le GPU.
Voici dans la Figure 38 l'affichage du fichier PLY en sortie de l'algorithme 4 sur Player3D.

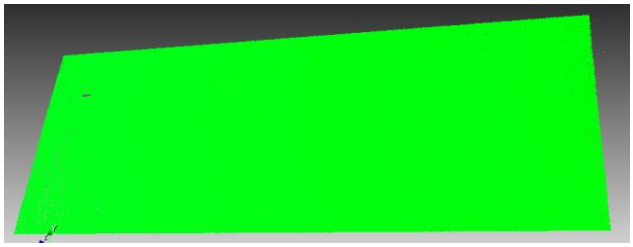

Figure 38 Image 3D générée par le quatrième algorithme

Algorithme de TESTIA

Après avoir terminé la présentation de mes propres algorithmes, j'ai entrepris d'implémenter sur CUDA l'algorithme d'extraction de raies laser déjà utiliser chez Testia. Cette fonction, issue de COAXLINK, a pour objectif de traiter chaque raie laser d'une image concaténée et de produire une ligne d'image en sortie pour chacune. Par exemple, si l'image concaténée mesure 2048x15200 pixels et que chaque raie laser correspond à une sous-image de 2048x52 pixels (soit un total de 300 raies), la sortie 3D aura une taille de 2048x300 pixels. Ainsi, chaque raie de taille 2048x52 est condensée en une ligne de 2048x1.
Sur CPU, cette fonction est appelée dans une boucle `for`, où chaque itération transforme une raie (2048x52) en une ligne (2048x1), nécessitant ainsi 300 itérations pour l'ensemble des raies. En CUDA, j'ai parallélisé ce processus pour permettre à la fonction de traiter simultanément les 300 raies, accélérant ainsi considérablement le traitement. Voici les résultats obtenus pour une image concaténée de taille 2048x15200, avec chaque raie laser mesurant 2048x52 pixels :

		GPU	CPU
TrtRaieSoft_y (CUDA vs CPU)	Avec Cuda Init	130 ms - 150 ms	48 – 50 ms
	Sans Cuda Init	15 ms - 16 ms	

Tableau 6 Comparaison des temps d'exécution entre CPU et GPU sur la fonction TrtRaieSoft_y

On peut observer une différence significative dans le temps d'exécution entre le GPU et le CPU (Tableau 6), à condition que l'initialisation CUDA ne soit effectuée qu'une seule fois dans le contexte du COAXLINK. J'ai effectué un test comparatif entre l'image générée avec le CPU et celle générée avec CUDA, et aucune différence n'a été observée.

4.1.2. Méthodes d'optimisation

J'ai effectué une recherche approfondie sur les techniques d'optimisation en CUDA pour accélérer le traitement et identifier deux approches principales pour améliorer les performances de mon code: l'utilisation efficace de la mémoire partagée et l'exploitation des streams. Par la suite, je me suis formé au concept d'occupation, et j'ai appris à utiliser Nsight Compute, un outil qui fournit des informations précises sur le temps d'exécution de chaque

fonction CUDA. Cet outil inclut également une calculatrice d'occupation permettant de déterminer le pourcentage d'occupation en entrant les paramètres requis.

Mémoire partagée

La mémoire partagée CUDA est un type de mémoire accessible à tous les threads au sein d'un même bloc. Elle réside sur la puce du GPU elle-même, ce qui la rend beaucoup plus rapide à accéder par rapport à la mémoire globale hors puce. On a 2 types de mémoire partagée : statique et dynamique [5].

Si la taille du tableau de mémoire partagée est connue à la compilation, il est possible de déclarer explicitement un tableau de cette taille dans le kernel. Ce tableau sera alors partagé entre tous les threads d'un même bloc.

Sinon, la taille de la mémoire partagée à allouer par bloc de threads doit être spécifiée (en octets) via un troisième paramètre optionnel lors de la configuration du lancement du kernel.

La mémoire partagée est utilisée pour réduire significativement le nombre d'accès à la mémoire globale, qui est beaucoup plus lente. En stockant les données fréquemment utilisées dans la mémoire partagée, on minimise les latences liées aux accès à la mémoire globale. Chaque bloc de threads accède une seule fois à la mémoire globale pour charger les données dans la mémoire partagée, permettant ensuite aux threads du bloc d'y accéder rapidement et de manière répétée.

Par exemple, si le kernel utilise une mémoire partagée dont la taille dépend de l'image à traiter, il faudra utiliser de la mémoire partagée dynamique. En revanche, si la taille est fixe, on peut opter pour la mémoire partagée statique.

La mémoire partagée a été bénéfique uniquement pour l'algorithme 1, car dans les autres algorithmes, chaque thread accède généralement à des variables différentes. Dans ces cas-là, la mémoire partagée ne présente aucun avantage.

	Avec CUDA (Avant)	**Avec CUDA (Après)**
Pour tous les fonctions (90 images)		
Temps de lecture des images	≈ 126.09 ms	≈ 126.09 ms
CUDA Init	≈ 112 ms	≈ 112 ms
Temps d'allouée de mémoire	≈ 40.581 ms	≈ 7.96 ms
Temps pour copier les données vers le GPU	≈ 46.26 ms	≈ 6.5 ms
Temps du kernel (gaussianBlurCUDA)	≈ 65.43 ms	≈ 5.91 ms
Temps du kernel (applyThresholdCUDA)	≈ 22.455 ms	≈ 5.11 ms
Temps du kernel (findLaserLineCUDA)	≈ 16.2 ms	≈ 3.28 ms
Temps pour copier les données vers le CPU	≈ 87.237 ms	≈ 2.86 ms
Temps pour Free data	≈ 22.96 ms	≈ 0.231 ms
Temps total du programme (sans chronos)	**≈ 686.7 ms**	**≈ 285.3 ms** (x2.4 plus rapide)

Tableau 7 Comparaison des temps d'exécution de l'algorithme 1 avant et après l'utilisation de la mémoire partagée

Après avoir intégré l'utilisation de la mémoire partagée dans l'algorithme 1, le temps d'exécution a été réduit de 2,4 fois (Tableau 7), mettant en évidence l'impact significatif des opérations de lecture et d'écriture sur la mémoire globale.

Occupation

1. **Warps par SM** [6] : Le SM (Streaming MultiProcessor) a un nombre maximal de warps qui peuvent être actifs simultanément. Étant donné que l'occupation est le rapport entre les warps actifs et le nombre maximal de warps actifs pris en charge, l'occupation est de 100 % si le nombre de warps actifs est égal au maximum. Si ce facteur limite les blocs actifs, l'occupation ne peut pas être augmentée. Par exemple, sur un GPU qui prend en charge 64 warps actifs par SM, 8 blocs actifs avec 256 threads par bloc (8 warps par bloc) donnent 64 warps actifs et une occupation théorique de 100 %. De même, 16 blocs actifs avec 128 threads par bloc (4 warps par bloc) donneraient également 64 warps actifs et une occupation théorique de 100 %.
2. **Blocs par SM** [6] : Le SM a un nombre maximal de blocs qui peuvent être actifs simultanément. Si l'occupation est inférieure à 100 % et que ce facteur limite les blocs actifs, cela signifie que chaque bloc ne contient pas suffisamment de warps pour atteindre 100 % d'occupation lorsque la limite de blocs actifs de l'appareil est atteinte. L'occupation peut être augmentée en augmentant la taille des blocs. Par exemple, sur un GPU qui prend en charge 16 blocs actifs et 64 warps actifs par SM, les blocs avec 32 threads (1 warp par bloc) génèrent au maximum 16 warps actifs (occupation théorique de 25 %), car seuls 16 blocs peuvent être actifs et chaque bloc n'a qu'un seul warp. Sur ce GPU, l'augmentation de la taille des blocs à 4 warps par bloc permet d'atteindre une occupation théorique de 100 %.
3. **Registres par SM** [6] : Le SM possède un ensemble de registres partagés par tous les threads actifs. Si ce facteur limite les blocs actifs, cela signifie que le nombre de registres par thread alloués par le compilateur peut être réduit pour augmenter l'occupation. Exemple : Si le GPU dispose de 65 536 registres par SM et que le programme utilise 48 registres par thread avec 1 536 threads par SM, la limite est dépassée, car cela nécessite 73 728 registres. Pour respecter cette limite, il est nécessaire de réduire le nombre de threads par bloc à 42, ce qui permet d'utiliser 64 512 registres et reste dans les capacités du GPU. Le nombre de registres par thread peut être Controller en utilisant l'option de compilation « maxrregcount » comme suit:
   `nvcc -maxrregcount=20 -o Executable.exe main.cu`
   Ou on peut utiliser la fonction « __maxnreg__() » comme suit :

```
__global__ void __
maxnreg__(maxNumberRegistersPerThread)
MyKernel(...)
{
 ...
}
```

On peut voir combien de registres de notre programme sont alloués avec Nsight Compute dans la partie « Launch Statistics » (Figure 39)

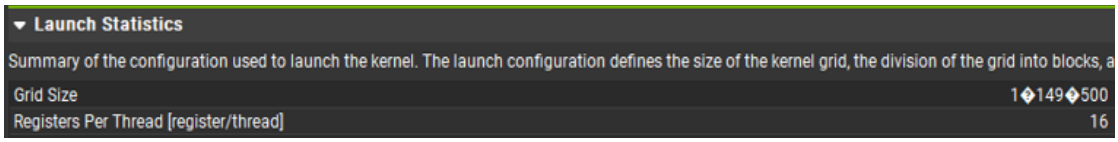

Figure 39 Visualisation du nombre de registres par thread dans Nsight Compute

4. **Mémoire partagée par SM** [6] : Chaque multiprocesseur de streaming (SM) dispose d'une quantité fixe de mémoire partagée accessible à tous ses threads actifs. Si cette mémoire partagée est entièrement utilisée, elle peut limiter le nombre de blocs pouvant s'exécuter simultanément sur le SM, réduisant ainsi l'occupation. Pour améliorer l'occupation, il est essentiel de minimiser la mémoire partagée requise par thread. Avec une taille de mémoire partagée par bloc de 12 Ko et une capacité maximale de 102 Ko par SM sur le GPU, un maximum de 8 blocs peut être alloué par SM. En diminuant la taille des blocs, le nombre de blocs pouvant s'exécuter simultanément sur un SM augmente, jusqu'à atteindre la limite maximale de blocs supportée par ce dernier.

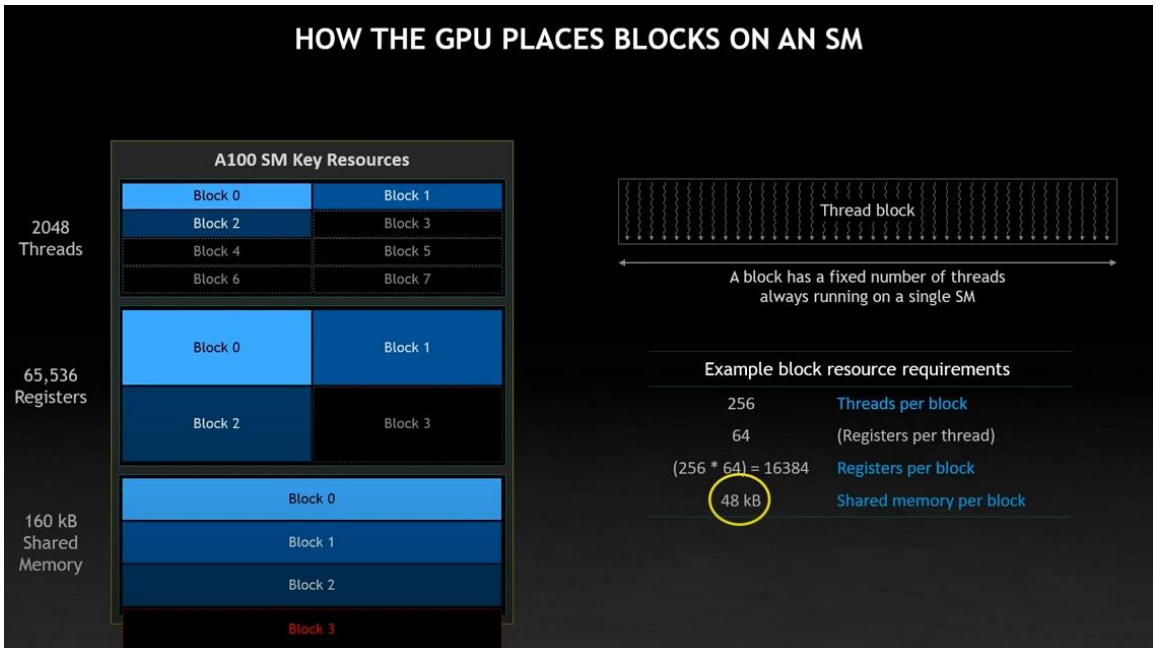

Figure 40 L'occupation sur le NVIDIA A100

Dans la Figure 40 [7], on observe qu'un SM d'un GPU A100 peut gérer jusqu'à 2048 threads, 65536 registres et 160 Ko de mémoire partagée. Par exemple, si on lance 256 threads par bloc, le SM peut contenir jusqu'à 8 blocs (en divisant le nombre total de threads pouvant être gérés par le SM par le nombre de threads par bloc). Avec 64 registres par thread, cela correspond à un total de 16384 registres par bloc. Dans ce cas, le SM pourra contenir jusqu'à 4 blocs (en divisant le nombre maximal de registres disponibles par le nombre de registres utilisés par bloc). Par ailleurs, si chaque bloc

utilise 48 Ko de mémoire partagée, le SM ne pourra accueillir que 3 blocs (en divisant la mémoire partagée maximale du SM par la mémoire partagée utilisée par bloc).
Ainsi, seulement 3 blocs seront alloués dans chaque SM, ce qui entraîne une sous-utilisation des ressources. Pour maximiser l'occupation des ressources, il serait nécessaire de réduire la taille de la mémoire partagée utilisée par bloc afin de permettre le lancement de 4 blocs. De plus, si possible, il conviendrait également de minimiser le nombre de registres utilisés et la mémoire partagée par bloc afin d'augmenter encore davantage l'occupation et l'efficacité du GPU.

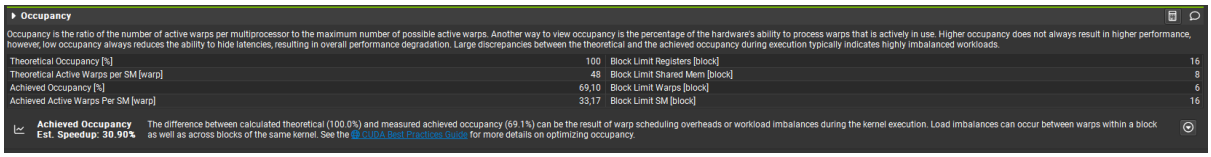

Figure 41 Visualisation de l'occupation dans Nsight Compute

Dans mon code, j'ai atteint une occupation théorique de 100 %, mais l'occupation effective était réduite à 69 % (Figure 41) en raison de divers facteurs pouvant influencer ce pourcentage, tels qu'un déséquilibre de charge de travail au sein des warps. Par exemple, si certains threads effectuent des calculs plus complexes que d'autres, cela peut entraîner des différences dans les temps d'exécution entre les warps.

Streams

Un stream en CUDA est une file d'attente d'actions que le GPU doit exécuter. Chaque fonction de l'API CUDA peut être exécutée de manière asynchrone, ce qui signifie que le code du CPU continue de s'exécuter pendant que l'instruction est en attente d'exécution, indépendamment du code hôte. Cependant, au sein d'un même stream, les instructions sont exécutées de manière synchrone les unes par rapport aux autres (Figure 42).

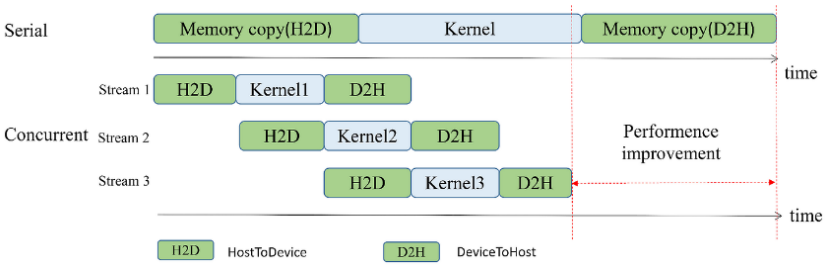

Figure 42 Fonctionnement des streams

Les streams sont particulièrement utiles lorsqu'il est nécessaire d'exécuter plusieurs kernels ou d'appeler un même kernel plusieurs fois. Cela s'est révélé pertinent dans le premier algorithme d'extraction de raies laser, où trois kernels étaient exécutés pour chaque image. En revanche, pour les autres algorithmes qui traitent une image concaténée, un seul appel au kernel est effectué, rendant l'utilisation des streams moins avantageuse.

J'ai intégré l'utilisation des streams dans le premier algorithme pour traiter 100 images de 1936x504 pixels. J'ai analysé les performances et visualisé les trames des opérations de copie et des kernels avec Nsight Compute, en étudiant cinq cas d'utilisation des streams.

1er cas : j'ai testé l'exécution d'un seul kernel par image pour observer le comportement du GPU (Figure 43). Il est essentiel de souligner que deux streams ne peuvent pas envoyer ou recevoir simultanément des données entre le GPU et le CPU, car ces opérations sont séquentielles et partagent les mêmes ressources.

```
// Processing each image in parallel using streams
for (int idx = 0; idx < NUM_IMAGES; ++idx) {
    // Copy image data from OpenCV Mat to pinned host memory
    memcpy(h_Image_Vector[idx], Image_Vector[idx].data, imgSize);

    // Asynchronously copy image data from host to device
    cudaMemcpyAsync(d_Image_Vector[idx], h_Image_Vector[idx], imgSize, cudaMemcpyHostToDevice, streams[idx]);

    // Perform image processing
    //gaussianBlurCUDA<<<gridSize, blockSize, sharedMemSize, streams[idx]>>>(d_Image_Vector[idx], d_Image_Vector[idx], rows, cols, kWidt
    //applyThresholdCUDA<<<gridSize, blockSize, 0, streams[idx]>>>(d_Image_Vector[idx], d_Image_Vector[idx], rows, cols, ThresholdValue,
    findLaserLineCUDA<<<(cols + 15) / 16, 16, 0, streams[idx]>>>(d_Image_Vector[idx], rows, cols, d_LaserLine_Vector[idx]);

    // Asynchronously copy the laser line results from device to pinned host memory
    cudaMemcpyAsync(h_LaserLine_Vector[idx], d_LaserLine_Vector[idx], lineSize, cudaMemcpyDeviceToHost, streams[idx]);
}
```

Figure 43 Code du premier cas d'étude sur les streams

La Figure 44 illustre les résultats obtenus avec la carte NVIDIA RTX 3060.

Figure 44 Graphique des streams sur le RTX 3060 pour le premier cas d'étude

On constate ici que seuls les kernels (indiquée en bleu) s'exécutaient en parallèle, tandis que les opérations de copie des données du CPU vers le GPU (indiquées en vert) et du GPU vers le CPU (indiquées en rose) ne s'exécutaient pas en parallèle avec les kernels.

Par la suite, j'ai testé ce programme sur la carte NVIDIA RTX 4000 Quadro, et les résultats sont illustrés dans la Figure 45.

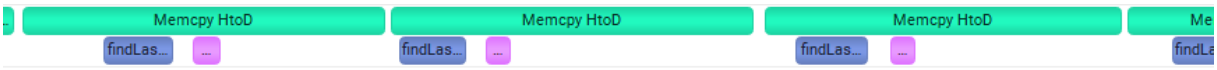

Figure 45 Graphique des streams sur le RTX 4000 Quadro pour le premier cas d'étude

Comme on peut le constater, le GPU s'est comporté de manière attendue, et la différence de comportement entre ces deux GPU s'explique par leur architecture respective : le NVIDIA RTX 3060 est conçu principalement pour le gaming, tandis que le RTX 4000 Quadro est un GPU destiné aux stations de travail.

2ème cas : J'ai augmenté le nombre de blocs et de threads par bloc pour tester son influence sur la parallélisation des streams (Figure 46).

```
// Processing each image in parallel using streams
for (int idx = 0; idx < NUM_IMAGES; ++idx) {
    // Copy image data from OpenCV Mat to pinned host memory
    memcpy(h_Image_Vector[idx], Image_Vector[idx].data, imgSize);

    // Asynchronously copy image data from host to device
    cudaMemcpyAsync(d_Image_Vector[idx], h_Image_Vector[idx], imgSize, cudaMemcpyHostToDevice, streams[idx]);

    // Perform image processing
    //gaussianBlurCUDA<<<gridSize, blockSize, sharedMemSize, streams[idx]>>>(d_Image_Vector[idx], d_Image_Vector[idx], r
    //applyThresholdCUDA<<<gridSize, blockSize, 0, streams[idx]>>>(d_Image_Vector[idx], d_Image_Vector[idx], rows, cols,
    findLaserLineCUDA<<<3872, 256, 0, streams[idx]>>>(d_Image_Vector[idx], rows, cols, d_LaserLine_Vector[idx]);

    // Asynchronously copy the laser line results from device to pinned host memory
    cudaMemcpyAsync(h_LaserLine_Vector[idx], d_LaserLine_Vector[idx], lineSize, cudaMemcpyDeviceToHost, streams[idx]);
}
```

Figure 46 Code du deuxième cas d'étude sur les streams

La Figure 47 illustre les résultats obtenus avec la carte NVIDIA RTX 3060.

Figure 47 Graphique des streams sur le RTX 3060 pour le deuxième cas d'étude

Le taux de parallélisation des kernels était inférieur par rapport au premier cas, ce qui met en évidence l'effet négatif de l'augmentation du nombre de blocs et de threads sur la parallélisation des streams.

Figure 48 Graphique des streams sur le RTX 4000 Quadro pour le deuxième cas d'étude

Dans la Figure 48, comme dans le premier cas, le RTX 4000 Quadro a montré un comportement similaire, avec un léger décalage dans l'exécution du kernel et lors de la copie des données du GPU vers le CPU.

3ème cas : j'ai testé l'influence des blocs et threads bidimensionnelle sur les streams (Figure 49).

```
for (int idx = 0; idx < NUM_IMAGES; ++idx) {
    // Copy image data from OpenCV Mat to pinned host memory
    memcpy(h_Image_Vector[idx], Image_Vector[idx].data, imgSize);

    // Asynchronously copy image data from host to device
    cudaMemcpyAsync(d_Image_Vector[idx], h_Image_Vector[idx], imgSize, cudaMemcpyHostToDevice, streams[idx]);

    // Perform image processing
    gaussianBlurCUDA<<<3872, 256, sharedMemSize, streams[idx]>>>(d_Image_Vector[idx], d_Image_Vector[idx], rows, cols, kWidth, kHei
    //applyThresholdCUDA<<<gridSize, blockSize, 0, streams[idx]>>>(d_Image_Vector[idx], d_Image_Vector[idx], rows, cols, ThresholdV
    //findLaserLineCUDA<<<3872, 256, 0, streams[idx]>>>(d_Image_Vector[idx], rows, cols, d_LaserLine_Vector[idx]);

    // Asynchronously copy the laser line results from device to pinned host memory
    cudaMemcpyAsync(h_LaserLine_Vector[idx], d_LaserLine_Vector[idx], lineSize, cudaMemcpyDeviceToHost, streams[idx]);
}
```

Figure 49 Code du troisième cas d'étude unidimensionnel sur les streams.

La Figure 50 illustre les résultats obtenus avec la carte NVIDIA RTX 3060.

Figure 50 Graphique des streams (1D) sur le RTX 3060 pour le troisième cas d'étude

Pour cette exécution en 1D, comme d'habitude, seuls les kernels sont légèrement parallélisés.

Figure 51 Graphique des streams (1D) sur le RTX 4000 Quadro pour le troisième cas d'étude

Cependant, sur le RTX 4000 Quadro (Figure 51), le kernel n'a pas pu s'exécuter en raison d'une mémoire partagée insuffisante. En effet, j'ai déclaré un espace de mémoire dépendant de la taille de l'image, et le nombre de blocs et de threads lancés a dépassé la capacité de cette mémoire.
En comparaison, lors de l'exécution en 2D (Figure 52), les résultats sont illustrés dans la Figure 53 avec la carte RTX 3060.

```
dim3 blockSize(16, 16);
dim3 gridSize((cols + blockSize.x - 1) / blockSize.x, (rows + blockSize.y - 1) / blockSize.y);

double ThresholdValue = 40;
double maxValue = 255;
size_t sharedMemSize = (blockSize.x + kWidth - 1) * (blockSize.y + kHeight - 1) * sizeof(uchar);

// Processing each image in parallel using streams
for (int idx = 0; idx < NUM_IMAGES; ++idx) {
    // Copy image data from OpenCV Mat to pinned host memory
    memcpy(h_Image_Vector[idx], Image_Vector[idx].data, imgSize);

    // Asynchronously copy image data from host to device
    cudaMemcpyAsync(d_Image_Vector[idx], h_Image_Vector[idx], imgSize, cudaMemcpyHostToDevice, streams[idx]);

    // Perform image processing
    gaussianBlurCUDA<<<gridSize, blockSize, sharedMemSize, streams[idx]>>>(d_Image_Vector[idx], d_Image_Vector[idx], rows, cols, kWidth, kHeight, half_kWidth, half_kHeight, d_GaussianKernel);
    //applyThresholdCUDA<<<gridSize, blockSize, 0, streams[idx]>>>(d_Image_Vector[idx], d_Image_Vector[idx], rows, cols, ThresholdValue, maxValue);
    //findLaserLineCUDA<<<(cols + 15) / 16, 16, 0, streams[idx]>>>(d_Image_Vector[idx], rows, cols, d_LaserLine_Vector[idx]);

    // Asynchronously copy the laser line results from device to pinned host memory
    cudaMemcpyAsync(h_LaserLine_Vector[idx], d_LaserLine_Vector[idx], lineSize, cudaMemcpyDeviceToHost, streams[idx]);
}
```

Figure 52 Code du troisième cas d'étude bidimensionnel sur les streams

Figure 53 Graphique des streams (2D) sur le RTX 3060 pour le troisième cas d'étude

Le taux de parallélisation était également de plus en plus faible, ce qui met en évidence l'effet négatif des blocs et threads bidimensionnels sur le RTX 3060.
En comparaison, sur le RTX 4000 Quadro, les résultats sont illustrés dans la Figure 54.

Figure 54 Graphique des streams (2D) sur le RTX 4000 Quadro pour le troisième cas d'étude

Dans ce cas, en raison du temps d'exécution du kernel qui est plus long que celui de la copie des données du CPU vers le GPU, ces opérations de transfert sont terminées avant que tous les kernels ne soient exécutés. Cela explique pourquoi une parallélisation parfaite n'a pas pu être atteinte.
4ème cas : J'ai simplement essayé de tester un petit nombre de blocs et de threads en configuration bidimensionnelle (Figure 55).

```
dim3 blockSize(8, 8);
dim3 gridSize(12, 8);

double ThresholdValue = 40;
double maxValue = 255;
size_t sharedMemSize = (blockSize.x + kWidth - 1) * (blockSize.y + kHeight - 1) * sizeof(uchar);

// Processing each image in parallel using streams
for (int idx = 0; idx < NUM_IMAGES; ++idx) {
    // Copy image data from OpenCV Mat to pinned host memory
    memcpy(h_Image_Vector[idx], Image_Vector[idx].data, imgSize);

    // Asynchronously copy image data from host to device
    cudaMemcpyAsync(d_Image_Vector[idx], h_Image_Vector[idx], imgSize, cudaMemcpyHostToDevice, streams[idx]);

    // Perform image processing
    gaussianBlurCUDA<<<gridSize, blockSize, sharedMemSize, streams[idx]>>>(d_Image_Vector[idx], d_Image_Vector[idx], rows, co
    //applyThresholdCUDA<<<gridSize, blockSize, 0, streams[idx]>>>(d_Image_Vector[idx], d_Image_Vector[idx], rows, cols, Thre
    //findLaserLineCUDA<<<3872, 256, 0, streams[idx]>>>(d_Image_Vector[idx], rows, cols, d_LaserLine_Vector[idx]);

    // Asynchronously copy the laser line results from device to pinned host memory
```

Figure 55 Code du quatrième cas d'étude sur les streams

Figure 56 Graphique des streams sur le RTX 3060 pour le quatrième cas d'étude

Avec le RTX 3060 (Figure 56), nous avons observé le même comportement qu'auparavant, avec toutefois une légère amélioration de la parallélisation des kernels par rapport au premier cas.

Figure 57 Graphique des streams sur le RTX 4000 Quadro pour le quatrième cas d'étude

Ensuite, sur le RTX 4000 Quadro (Figure 57), nous avons obtenu exactement le même résultat que dans le premier cas : une parallélisation entre les opérations de copie et l'exécution des kernels.

5ème cas (code original) : j'ai réactivé les deux autres kernels et relancé le programme (Figure 58).

```
dim3 blockSize(16, 16);
dim3 gridSize((cols + blockSize.x - 1) / blockSize.x, (rows + blockSize.y - 1) / blockSize.y);

double ThresholdValue = 40;
double maxValue = 255;
size_t sharedMemSize = (blockSize.x + kWidth - 1) * (blockSize.y + kHeight - 1) * sizeof(uchar);

// Processing each image in parallel using streams
for (int idx = 0; idx < NUM_IMAGES; ++idx) {
    // Copy image data from OpenCV Mat to pinned host memory
    memcpy(h_Image_Vector[idx], Image_Vector[idx].data, imgSize);

    // Asynchronously copy image data from host to device
    cudaMemcpyAsync(d_Image_Vector[idx], h_Image_Vector[idx], imgSize, cudaMemcpyHostToDevice, streams[idx]);

    // Perform image processing
    gaussianBlurCUDA<<<gridSize, blockSize, sharedMemSize, streams[idx]>>>(d_Image_Vector[idx], d_Image_Vector[idx], rows, cols, kWidth, kHe
    applyThresholdCUDA<<<gridSize, blockSize, 0, streams[idx]>>>(d_Image_Vector[idx], d_Image_Vector[idx], rows, cols, ThresholdValue, maxVa
    findLaserLineCUDA<<<(cols + 15) / 16, 16, 0, streams[idx]>>>(d_Image_Vector[idx], rows, cols, d_LaserLine_Vector[idx]);

    // Asynchronously copy the laser line results from device to pinned host memory
    cudaMemcpyAsync(h_LaserLine_Vector[idx], d_LaserLine_Vector[idx], lineSize, cudaMemcpyDeviceToHost, streams[idx]);
}
```

Figure 58 Code du cinquième cas d'étude sur les streams (code original)

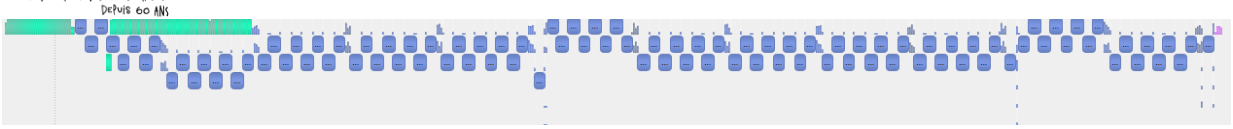

Figure 59 Graphique des streams sur le RTX 3060 pour le cinquième cas d'étude

Pour le RTX 3060 (Figure 59), une légère parallélisation a été observée entre les kernels et les opérations de copie au début, puis les kernels ont été exécutés en parallèle par la suite.

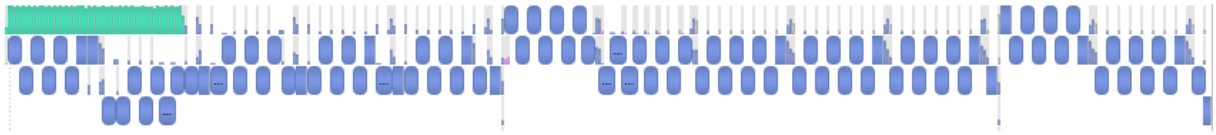

Figure 60 Graphique des streams sur le RTX 4000 Quadro pour le cinquième cas d'étude

Pour le RTX 4000 Quadro (Figure 60), nous avons observé un comportement similaire, à la différence près que la parallélisation entre les kernels et les opérations de copie des données a commencé dès le début de l'exécution.

	RTX 3060	RTX 4000 Quadro
Cas 1	184 ms (118 ms sans streams)	166 ms (120 ms sans streams)
Cas 2	189 ms	158 ms
Cas 3	1D : 190 ms ; 2D : 241 ms	1D : 341 ms ; 2D : 229 ms
Cas 4	188 ms	164 ms
Cas 5	237 ms (208 ms sans streams)	240 ms (220 ms sans streams)

Tableau 8 Comparaison des temps d'exécution entre la RTX 3060 et la RTX 4000 Quadro sur différents cas d'étude utilisant les streams

D'après le Tableau 8, il est évident que le temps d'exécution sans l'utilisation des streams était plus rapide. Par conséquent, nous avons constaté que leur utilisation dans notre algorithme n'apporte aucun avantage significatif.

4.2. IDEEPIX

Ideepix est un logiciel utilisé par Testia pour le traitement d'images 2D et 3D. Il permet d'appliquer divers filtres et de réaliser de nombreuses autres opérations. Cependant, tous ces traitements sont actuellement effectués sur des CPUs, ce qui ralentit le processus. Par exemple, appliquer un filtre médian sur une image de taille 1936x3650 prend environ 1,6 seconde. L'objectif est donc d'identifier les filtres les plus gourmands en temps sur le CPU et de tenter de les optimiser en les exécutant avec CUDA. Pour commencer il faut créer une DLL avec CUDA qui sera utilisé par IDEEPIX et compiler CUDA en 32 bits étant donné que IDEEPIX est en 32 bit.

4.2.1. Filtre Médian

La première fonction dans IDEEPIX à convertir était la fonction qui appliquait le filtre médian. Le filtre médian est une technique de filtrage numérique non linéaire, souvent utilisée pour supprimer le bruit d'une image ou d'un signal (Figure 61).

Figure 61 Exemple d'utilisation d'un filtre médian

La fonction Median3DAverageFen16To8 est une fonction de la bibliothèque Biblio_3D intégré dans IDEEPIX qui prend en paramètre une image source de 16 bits contenant l'image à traiter et une image de 8 bits vide, la fonction appliquera le filtre médian sur l'image 16 bit et ainsi nous donne l'image en sortie sur 8 bits.

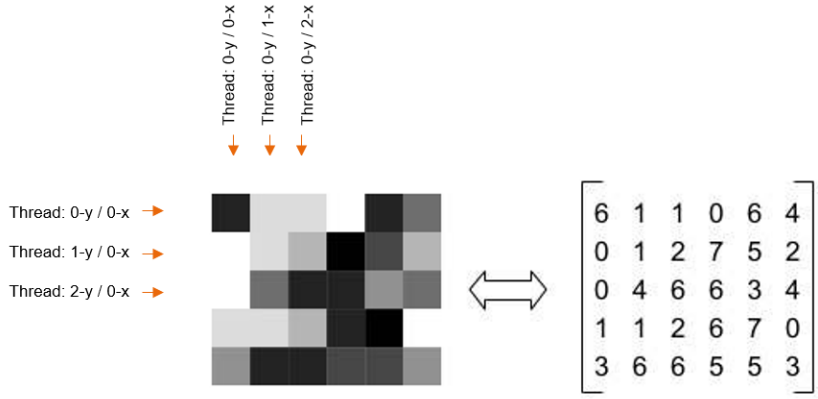

Figure 62 Processus du filtre médian avec CUDA

Grâce à CUDA, nous pouvons affecter un thread à chaque pixel de l'image (Figure 62). Cette parallélisation nous permet de traiter chaque pixel indépendamment, sans risque d'interférence. Chaque thread dispose de son propre buffer pour stocker les valeurs des pixels voisins, ce qui garantit que les calculs sont effectués sur des données cohérentes. En utilisant une image de sortie pour stocker les valeurs des pixels modifiés, nous garantissons que l'affectation des pixels résultants n'interfère pas entre eux.

Chez Testia, cette fonction était réalisée de manière légèrement différente, car elle effectuait un lissage de l'image. Pour cela, j'ai développé une version du filtre médian standard sur le CPU, puis je l'ai implémentée en CUDA. Les résultats obtenus sont présentés ci-dessous:

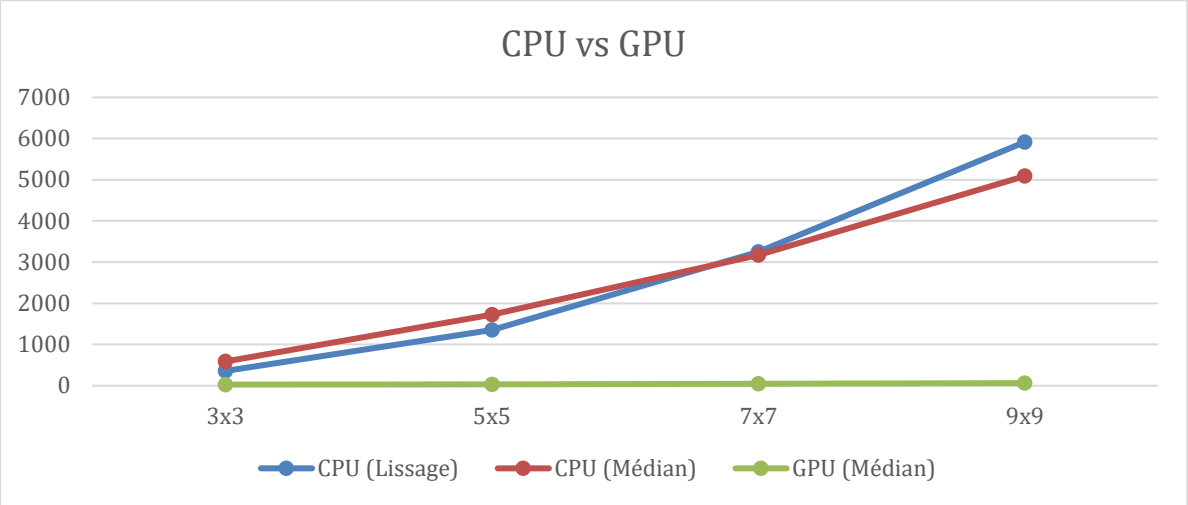

Figure 63 Temps d'exécution du filtre médian en fonction de la taille du noyau pour une image 2048x3650 (en ms)

D'après la Figure 63, on peut constater que, pour une grande image, un gain est observé quel que soit le filtre utilisé. Cependant, il est également important de noter que les temps d'exécution sont proches avec un filtre de petite taille, mais s'écartent davantage à mesure que la taille du filtre augmente.

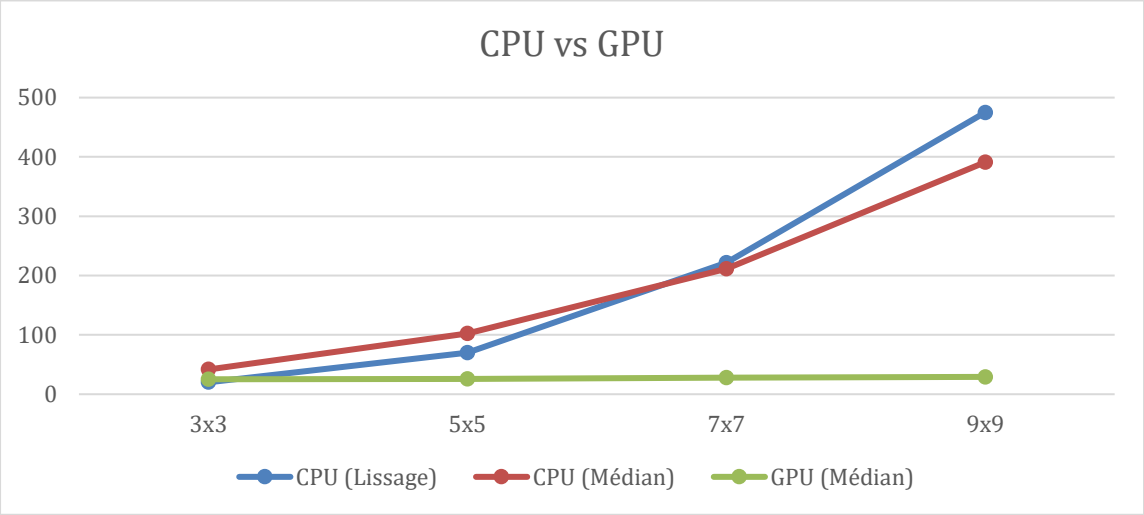

Figure 64 Temps d'exécution du filtre médian en fonction de la taille du noyau pour une image 700x700 (en ms)

Avec une image beaucoup plus petite (Figure 64), on observe généralement que le GPU reste plus rapide. Toutefois, lorsque la taille du filtre est très réduite, le temps d'exécution du GPU se rapproche de celui du CPU.

4.2.2. Dilatation

La dilatation est une opération fondamentale en traitement d'image morphologique, utilisée principalement pour manipuler des formes dans une image binaire ou en niveaux de gris. Son but est d'agrandir les zones lumineuses ou blanches de l'image en fonction d'un élément structurant donné (Figure 65).

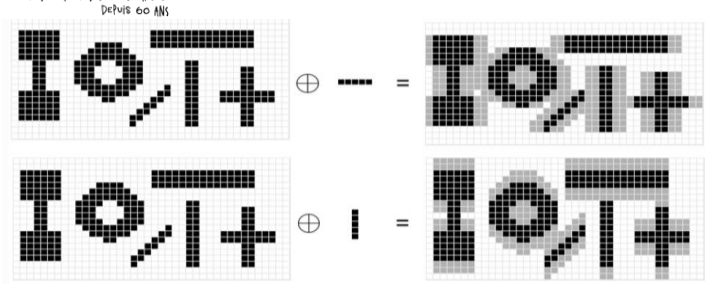

Figure 65 Exemple d'utilisation d'un filtre de dilatation

Dans le logiciel Ideepix, deux fonctions de dilatation sont disponibles : une pour le traitement horizontal et une autre pour le traitement vertical. La dilatation horizontale traite chaque colonne individuellement dans une boucle for, tandis que la dilatation verticale traite chaque ligne. En CUDA, j'ai pu paralléliser les colonnes pour la dilatation horizontale et les lignes pour la dilatation verticale. Ensuite, j'ai comparé les résultats obtenus sur le GPU avec ceux du CPU.

Temps d'exécution sur Image 3650x2048	**CPU**		**GPU**		**Gain en %**	
Sense	**Horizontale**	**Verticale**	**Horizontale**	**Verticale**	**Horizontale**	**Verticale**
Noyau 2	48.36 ms	120.25 ms	5.84 ms	10.44 ms	728%	1051%
Noyau 32	54.22 ms	119.38 ms	5.42 ms	9.11 ms	900%	1210%
Noyau 64	57.1 ms	118.32 ms	5.46 ms	9.95 ms	945%	1089%

Tableau 9 Comparaison des temps d'exécution entre CPU et GPU sur le filtre de dilatation (Image 3650x2048)

D'après le Tableau 9, on observe un gain significatif en performance avec le GPU par rapport au CPU, aussi bien pour la dilatation horizontale que verticale. Ainsi, cela démontre clairement l'efficacité du traitement parallèle sur GPU.

Temps d'exécution sur Image 700x700	**CPU**		**GPU**		**Gain en %**	
Sense	**Horizontale**	**Verticale**	**Horizontale**	**Verticale**	**Horizontale**	**Verticale**
Noyau 2	3 ms	4.38 ms	1.58 ms	6.1 ms	89.8%	-28%
Noyau 32	3.44 ms	5.04 ms	1.65 ms	5.25 ms	108.4%	-4%
Noyau 64	3.72 ms	5.2 ms	1.66 ms	5.26 ms	124%	-1.14%

Tableau 10 Comparaison des temps d'exécution entre CPU et GPU sur le filtre de dilatation (Image 700x700)

Avec une image de plus petite taille (Tableau 10), le CPU s'est avéré plus rapide en raison de la réduction du volume de données à traiter.

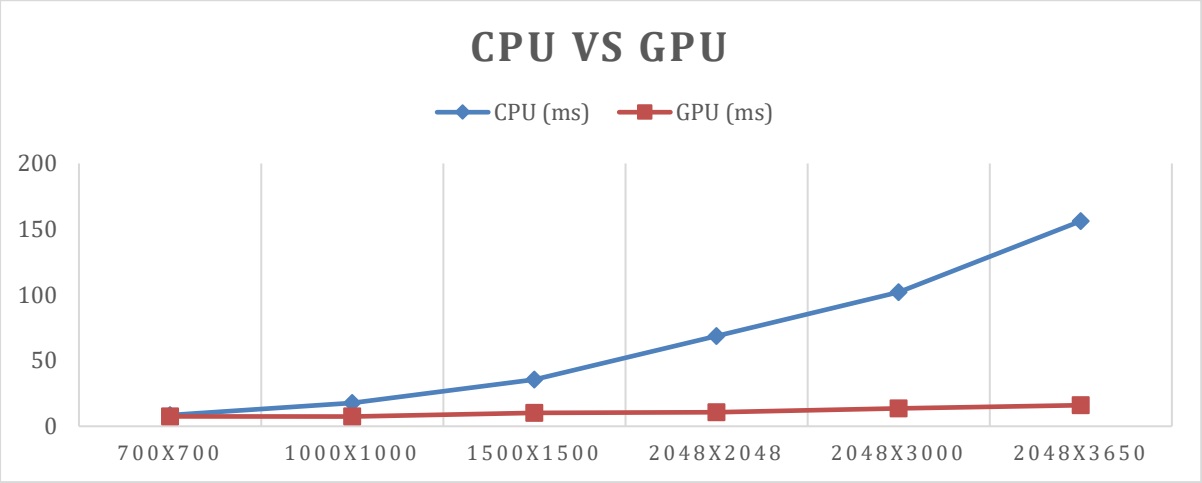

Figure 66 Comparaison du filtre dilatation (horizontal et vertical) selon la taille de l'image avec un noyau de 32

La Figure 66 illustre que le CPU est plus efficace pour des fenêtres d'image de petite taille, tandis que le GPU maintient une performance relativement stable, même lorsque la taille de la fenêtre augmente.

4.2.3. Erosion

L'érosion est une opération morphologique utilisée en traitement d'image pour réduire les objets présents dans une image binaire ou en niveaux de gris. Elle est particulièrement utile pour enlever les petites imperfections ou réduire les détails fins d'une image (Figure 67).

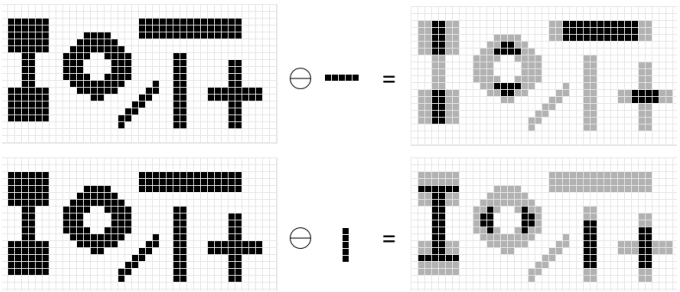

Figure 67 Exemple d'utilisation d'un filtre d'érosion

Dans le logiciel Ideepix, deux fonctions d'érosion sont disponibles : l'une pour le traitement horizontal et l'autre pour le traitement vertical. En CUDA, j'ai parallélisé les colonnes pour l'érosion horizontale et les lignes pour l'érosion verticale. Par la suite, j'ai comparé les résultats obtenus sur le GPU avec ceux du CPU.

Temps d'exécution sur Image 3650x2048	**CPU**		**GPU**		**Gain en %**	
Sense	**Horizontale**	**Verticale**	**Horizontale**	**Verticale**	**Horizontale**	**Verticale**
Noyau 2	48.31 ms	119.64 ms	5.22 ms	10.33 ms	825%	1058%
Noyau 32	57.11 ms	117.96 ms	5.08 ms	9.35 ms	1024%	1161%
Noyau 64	54.64 ms	119.78 ms	5.55 ms	10.25 ms	884%	1068%

Tableau 11 Comparaison des temps d'exécution entre CPU et GPU sur le filtre d'érosion (Image 3650x2048)

D'après le Tableau 11, un gain est systématiquement observé avec le GPU.

Temps d'exécution sur Image 700x700	CPU		GPU		Gain en %	
Sense	**Horizontale**	**Verticale**	**Horizontale**	**Verticale**	**Horizontale**	**Verticale**
Noyau 2	2.98 ms	4.69 ms	1.7 ms	6.46 ms	75.2%	-27.3%
Noyau 32	3.22 ms	4.84 ms	1.41 ms	5.69 ms	128.3%	-14.9%
Noyau 64	3.45 ms	4.78 ms	1.46 ms	7.64 ms	136.3%	-37.4%

Tableau 12 Comparaison des temps d'exécution entre CPU et GPU sur le filtre d'érosion (Image 700x700)

Avec une image de taille réduite (**Erreur ! Source du renvoi introuvable.**), on observe des r ésultats similaires, mettant en évidence la rapidité du CPU par rapport au GPU lorsque les volumes de données sont plus faibles.

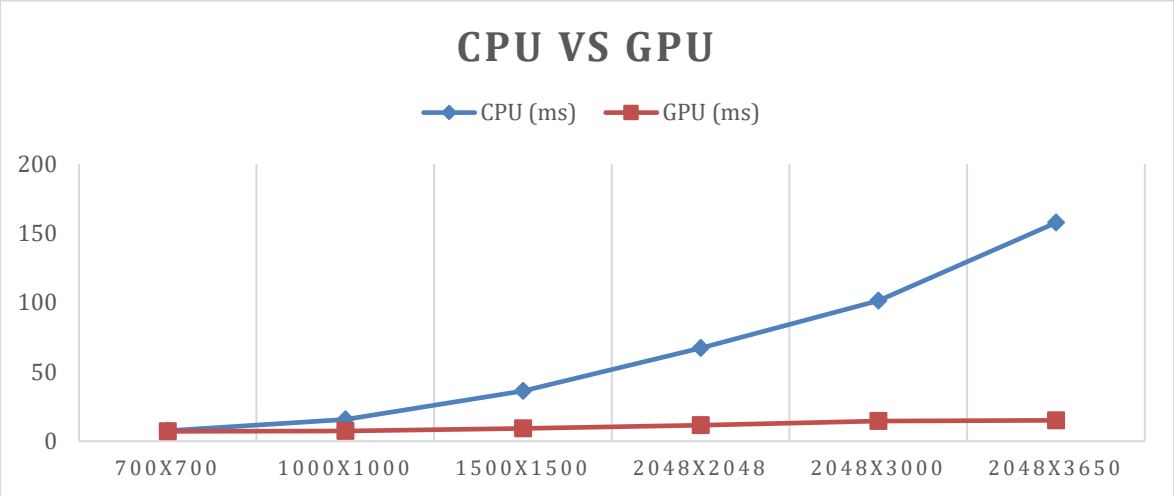

Figure 68 Comparaison du filtre d'érosion (horizontal et vertical) selon la taille de l'image avec un noyau de 32

La Figure 68 montre que les résultats obtenus restent cohérents avec les observations précédentes : le CPU est plus performant pour des fenêtres d'image de petite taille, tandis que le GPU conserve une performance stable, même avec l'augmentation de la taille de la fenêtre.

4.3. Problèmes/Solutions

- Le premier problème que j'ai rencontré concernait le filtre médian déjà implémenté chez Testia, qui différait du filtre médian standard. Dans leur version, chaque itération influençait la suivante : lorsqu'un pixel changeait de valeur lors d'une itération, la nouvelle valeur était utilisée pour calculer la valeur du pixel suivant dans l'itération suivante, contrairement au filtre médian standard qui utilise les valeurs initiales pour toutes les itérations. Cette approche rendait impossible son implémentation en CUDA. Par conséquent, j'ai choisi d'implémenter CUDA sur la version standard du filtre médian.
- Lors des tests du filtre médian sur GPU, j'ai constaté des variations dans les temps d'exécution à chaque appel de la fonction. Après investigation, j'ai découvert que ces fluctuations étaient dues à la fréquence d'horloge dynamique du GPU, qui s'accélère au démarrage du programme puis diminue automatiquement. Pour résoudre ce

problème, j'ai configuré la fréquence d'horloge du GPU en mode statique à l'aide de la commande `nvidia-smi`. En ouvrant l'invite de commande en mode administrateur, j'ai d'abord utilisé la commande `nvidia-smi -q -d CLOCK` pour identifier la fréquence maximale du GPU, puis fixé la plage de fréquences souhaitée, par exemple entre 2000 et 2115 MHz, avec la commande `nvidia-smi -lgc 2000,2115` et on peut remettre la fréquence en mode dynamique avec la commande `nvidia-smi -rgc`. Cette configuration a permis au GPU de maintenir une fréquence stable, améliorant ainsi ses performances et sa stabilité pendant les tests. J'ai également cherché une fonction dans l'API CUDA permettant de fixer l'horloge du GPU à une valeur statique. J'ai trouvé la fonction **nvmlDeviceSetGpuLockedClocks**, qui fait partie de la bibliothèque **NVML**, propre à CUDA. Cependant, cette bibliothèque n'était disponible qu'en 64 bits. Pour contourner ce problème, j'ai intégré cette fonction dans ma DLL en ajoutant une macro qui s'active uniquement lorsque la DLL est compilée en 64 bits.

J'ai ensuite commencé à observer l'impact de la maximisation de la fréquence d'horloge sur la température de mon GPU NVIDIA RTX 3060. J'ai remarqué un comportement cyclique : la température monte progressivement de 37 °C à 57 °C en l'espace de 9 minutes lorsque l'horloge est poussée au maximum, ce qui déclenche les ventilateurs. Ces derniers parviennent à abaisser rapidement la température à 40 °C en seulement 10 secondes, mais ce cycle se répète en boucle (Figure 69).

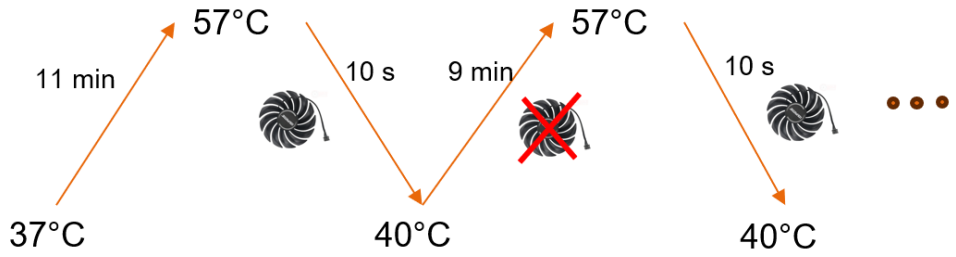

Figure 69 Comportement du GPU face à l'augmentation de sa température

- La compilation de la DLL en CUDA en 32 bits s'est avérée complexe, car j'utilisais initialement CUDA 12.6, une version qui ne supporte plus le 32 bits. J'ai donc dû réinstaller CUDA 9.2, la dernière version compatible avec la compilation en 32 bits. Lorsqu'on compile avec le compilateur nvcc de CUDA, celui-ci cherche automatiquement à utiliser le compilateur de Visual Studio. Pour compiler en 32 bits, j'ai tenté d'utiliser clang++ à la place de Visual Studio, puisque le projet Ideepix est nativement compilé avec clang++. Cependant, cette méthode n'a pas fonctionné. J'ai alors découvert que la dernière version de Visual Studio supportant la compilation en 32 bits était Visual Studio 12.0 (2013). Bien que cette version ne soit plus disponible sur le site officiel de Microsoft, j'ai pu la trouver en effectuant une recherche approfondie en ligne. Une fois installée, elle a fonctionné avec nvcc de CUDA 9.2, et j'ai pu générer avec succès une DLL en 32 bits.

- J'ai été chargé d'implémenter une fonction réalisant la moyenne locale sur une image à l'aide de CUDA. Cependant, cette fonction présentait une dépendance entre les itérations, ce qui la rendait incompatible avec CUDA. J'ai alors tenté de modifier la fonction pour la rendre compatible, mais le résultat obtenu était totalement différent de l'image attendue.
- Pour structurer la bibliothèque que j'ai développée en CUDA, j'ai essayé d'organiser mes fonctions au sein de classes, puis j'ai recompilé la DLL. Cependant, lorsque j'ai tenté d'appeler ces fonctions depuis Ideepix, compilé avec clang++, une erreur de liaison s'est produite. Cela m'a conduit à la conclusion que l'encodage des classes par Visual Studio 12.0 est incompatible avec celui du compilateur clang++.
- On m'a demandé d'étudier le comportement du GPU lorsque plusieurs processus utilisent CUDA simultanément. Cette analyse a révélé que les processus affectent les performances les uns des autres. Par la suite, j'ai découvert le service MPS (Multi-Process Service), conçu pour optimiser les performances du GPU en cas d'exécution concurrente en répartissant ses ressources, permettant ainsi à plusieurs processus d'y accéder simultanément [8].

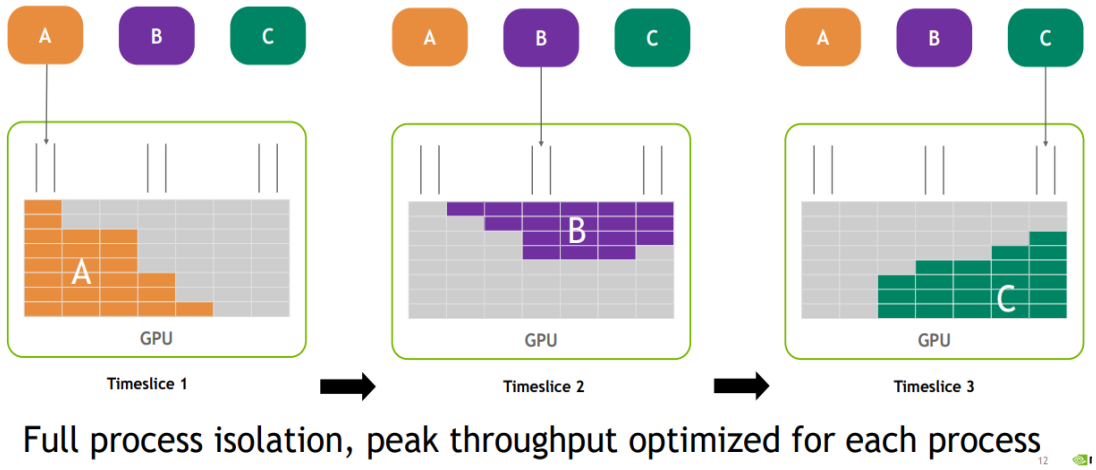

Figure 70 Comportement du GPU face à plusieurs processus simultanés

Sur la Figure 70, on a 3 processus qui s'executent les uns après les autres et qui n'utilisent pas tous les ressources du GPU et cela cause des ressources inutilisable et pour resoudre cela, le MPS gère les ressources du GPU pour pouvoir executer plusieurs processus en meme temps. Comme le montre la Figure 71.

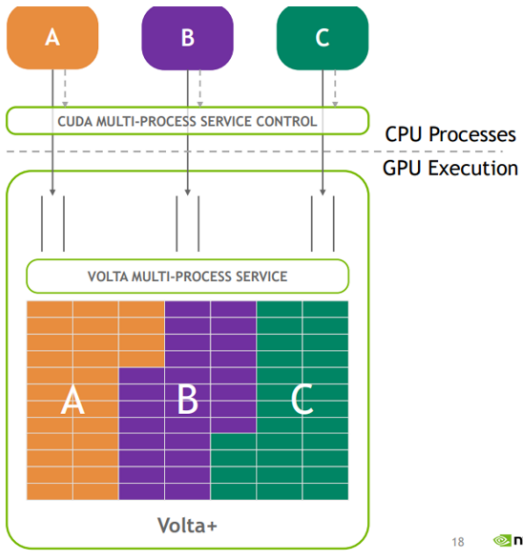

Figure 71 Comportement du GPU avec MPS face à plusieurs processus

Cependant, ce service n'est disponible que sur les systèmes Linux, ce qui constitue une limitation.

Bilan du stage

En conclusion, j'ai réussi à répondre à toutes les attentes de Testia. J'ai mené une étude approfondie sur CUDA, documentant chaque étape à travers des rapports détaillés. J'ai également conçu de nouveaux algorithmes d'extraction de raies laser et les ai implémentés en CUDA, tout en adaptant leur propre algorithme existant pour effectuer des tests comparatifs avec le CPU afin d'évaluer les gains de performance. De plus, j'ai optimisé les fonctions les plus consommatrices en temps dans leur logiciel Ideepix, dédié au traitement d'image, en les adaptant à CUDA. Ainsi, j'ai pleinement accompli les objectifs qui m'étaient fixés.

Ce stage m'a permis de découvrir et d'approfondir mes connaissances sur CUDA, une technologie essentielle dans le domaine du traitement d'images, ainsi que de comprendre en profondeur le fonctionnement des GPU, notamment leur architecture et leur gestion parallèle des calculs. J'ai également appris à utiliser Nsight Compute, un outil fourni par NVIDIA, pour analyser et optimiser les programmes CUDA. J'ai renforcé mes compétences en planification de projets et en résolution de problèmes de manière autonome. Ce stage m'a aussi donné l'occasion de mettre en pratique mes connaissances en C++, un langage que j'avais étudié à l'ENIB, tout en travaillant avec la bibliothèque OpenCV, spécifiquement utilisée pour le traitement d'images. J'ai appris à comprendre et à adapter des fonctions et des systèmes déjà existants pour répondre aux besoins de l'entreprise.

Sur le plan personnel, cette expérience m'a ouvert les yeux sur l'environnement de travail en entreprise, me permettant de mieux comprendre les attentes professionnelles envers un ingénieur. Travailler au sein d'une équipe m'a aidé à développer une approche plus professionnelle et à changer ma mentalité. Ce stage m'a également fait prendre conscience de l'importance des enseignements reçus à l'ENIB, renforçant ma motivation à approfondir mes connaissances. En résumé, mon stage chez Testia a été une expérience très enrichissante, à la fois sur le plan personnel et professionnel.

Bibliographie/Webographie

1. IWOCL, «A Comparison of SYCL, OpenCL, CUDA, & OpenMP for Massively Parallel Support Vector Classification,» 2022. [En ligne]. Available: https://www.youtube.com/watch?v=-yphY7Rtltc.

2. P.-F. Bonnefoi, «Développement GPGPU,» 2023.

3. T. Nurkkala, «CUDA Programming,» 2021. [En ligne]. Available: https://www.youtube.com/watch?v=xwbD6fL5qC8.

4. «AI Chips: A100 GPU with Nvidia Ampere architecture,» [En ligne]. Available: https://jonathan-hui.medium.com/ai-chips-a100-gpu-with-nvidia-ampere-architecture-3034ed685e6e.

5. «Using Shared Memory in CUDA C/C++,» 2013. [En ligne]. Available: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/.

6. Nvidia, «Achieved Occupancy,» [En ligne]. Available: https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm.

7. C. I. Rust, «GTC 2022 - How CUDA Programming Works - Stephen Jones, CUDA Architect, NVIDIA,» 2022. [En ligne]. Available: https://www.youtube.com/watch?v=QQceTDjA4f4&t=75s.

8. «INTRODUCTION TO CUDA's MULTI-PROCESS SERVICE (MPS),» [En ligne]. Available: https://www.olcf.ornl.gov/wp-content/uploads/2021/06/MPS_ORNL_20210817.pdf.