

# IRGPUA Projet

David CHEMALY

Thuraya SHANBARI

## Table des matières

|           |                                   |          |
|-----------|-----------------------------------|----------|
| <b>1</b>  | <b>Introduction</b>               | <b>3</b> |
| <b>2</b>  | <b>Gestion des erreurs</b>        | <b>3</b> |
| <b>3</b>  | <b>Structure</b>                  | <b>3</b> |
| <b>4</b>  | <b>Initialisation</b>             | <b>5</b> |
| <b>5</b>  | <b>Step 1 : Build Predicate</b>   | <b>6</b> |
| 5.1       | Implémentation . . . . .          | 6        |
| 5.2       | Benchmark . . . . .               | 6        |
| <b>6</b>  | <b>Step 2 : Scan</b>              | <b>6</b> |
| 6.1       | Implémentation . . . . .          | 6        |
| 6.2       | Benchmark . . . . .               | 7        |
| 6.3       | Scan . . . . .                    | 7        |
| 6.4       | Shift . . . . .                   | 7        |
| <b>7</b>  | <b>Step 3 : Scatter Addresses</b> | <b>7</b> |
| 7.1       | Implémentation . . . . .          | 7        |
| 7.2       | Benchmark . . . . .               | 8        |
| <b>8</b>  | <b>Step 4 : Apply Map</b>         | <b>8</b> |
| 8.1       | Implémentation . . . . .          | 8        |
| 8.2       | Benchmark . . . . .               | 8        |
| <b>9</b>  | <b>Step 5 : Histogram</b>         | <b>9</b> |
| 9.1       | Implémentation . . . . .          | 9        |
| 9.2       | Benchmark . . . . .               | 9        |
| <b>10</b> | <b>Step 6 : First Non-Zero</b>    | <b>9</b> |
| 10.1      | Implémentation . . . . .          | 9        |
| 10.2      | Benchmark . . . . .               | 10       |
| 10.2.1    | Predicate Zeros . . . . .         | 10       |

|   |           |
|---|-----------|
| 10.2.2 Find First Non Zero . . . . .        | 10        |
| <b>11 Step 7 : Apply Map Transformation</b> | <b>10</b> |
| 11.1 Implémentation . . . . .               | 10        |
| 11.2 Benchmark . . . . .                    | 11        |
| <b>12 Step 8 : Reduce</b>                   | <b>11</b> |
| 12.1 Implémentation . . . . .               | 11        |
| 12.2 Intégration . . . . .                  | 11        |
| 12.3 Benchmark . . . . .                    | 11        |
| <b>13 Step 9 : Sort</b>                     | <b>12</b> |
| 13.1 Implémentation . . . . .               | 12        |
| 13.2 Benchmark . . . . .                    | 12        |
| 13.2.1 Order Checking . . . . .             | 12        |
| 13.2.2 Radix Sort . . . . .                 | 12        |
| <b>14 Transfert de données</b>              | <b>12</b> |
| <b>15 Implémentation : Thrust</b>           | <b>13</b> |
| 15.1 Implémentation . . . . .               | 13        |
| <b>16 Final benchmark</b>                   | <b>13</b> |
| <b>17 Conclusion</b>                        | <b>14</b> |

## 1 Introduction

Pour ce projet, nous avons opté pour le pair-programming, particulièrement important étant donné le niveau de réflexion requis. Tout d’abord, nous nous sommes installés et avons pris le temps de parcourir le code CPU fourni. Ensuite, nous avons procédé à la compilation du projet et avons entamé la démarche de compréhension du code.

## 2 Gestion des erreurs

**Fichier: error.cuh**

Dans un premier temps, nous avons implémenté la gestion des erreurs en CUDA. Pour ce faire, nous avons utilisé deux macros afin de faciliter le processus de débogage.

- La première, `cudaCheckError()`, permet de vérifier s’il y a eu une erreur CUDA après l’exécution d’une fonction CUDA. Elle utilise `cudaGetLastError()` pour récupérer le dernier code d’erreur.
- La deuxième macro, `gpu_err_check(ans)`, prend en argument `ans`, qui correspond généralement à une fonction CUDA. Elle invoque la fonction `gpu_assert` avec `ans`, ainsi que le nom du fichier et le numéro de ligne. Cette macro est employée pour contrôler les erreurs après l’exécution d’une fonction CUDA spécifique.
- La fonction `gpu_assert` prend en argument un code d’erreur CUDA, le nom du fichier et le numéro de ligne. Si le code d’erreur n’est pas `cudaSuccess`, cette fonction affiche un message d’erreur contenant le message associé à ce code, ainsi que le nom du fichier et le numéro de ligne. Si le paramètre `abort` est vrai (ce qui est la valeur par défaut), le programme se termine avec le code d’erreur.
- Les ”wrappers” sont des fonctions CUDA encapsulées dans des fonctions simples. Chaque wrapper appelle la fonction CUDA correspondante et utilise `gpu_err_check` pour vérifier s’il y a une erreur après l’appel de la fonction CUDA. Si une erreur est détectée, `gpu_err_check` invoque `gpu_assert` pour gérer cette erreur.

## 3 Structure

**Fichier: deviceArray.cuh**

Tout d’abord, nous avons pris le temps de nous asseoir et de réfléchir à la

meilleure façon d'implémenter le code pour le GPU. En analysant le code pour le processeur (CPU), nous avons anticipé qu'il y aurait de nombreuses opérations telles que l'allocation de mémoire (malloc), l'initialisation de mémoire (memset) et la copie de tampons. C'est pourquoi nous avons créé une classe appelée DeviceArray qui contient des méthodes servant de wrappers pour les fonctions CUDA. Nous avons suivi cette approche pour mettre en place cette classe.

**Fichiers: fix\_gpu.cu, fix\_gpu.cuh**

Ensuite, pour effectuer des mesures de performance (benchmarking), nous avons divisé le code en deux parties distinctes : une pour le GPU et une pour le CPU.

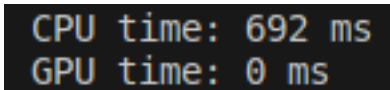
**Fichiers: parse.cu, parse.hh**

Pour faciliter le processus de benchmarking et décider quel composant tester (CPU ou GPU), nous avons mis en place une petite interface en ligne de commande (CLI).

```
./main [--version <cpu|gpu>] [--benchmark]
```

**Fichier: main.cu**

Ensuite, nous avons utilisé la bibliothèque chrono afin de mesurer le temps d'exécution de chaque implémentation.



```
CPU time: 692 ms
GPU time: 0 ms
```

**Fichier: pipeline.hh**

Nous avons également étendu la classe pipeline en ajoutant une méthode appelée set\_images. Cela nous permet d'accéder à nos images dans la fonction principale (main) afin de pouvoir comparer les versions CPU et GPU.

**Fichier: main.cu**

Ensuite, nous avons mis en place une fonction qui compare les résultats obtenus avec le CPU et le GPU. Cette fonction compare la somme totale et vérifie la taille du buffer.

**Fichier: fix\_gpu.cu**

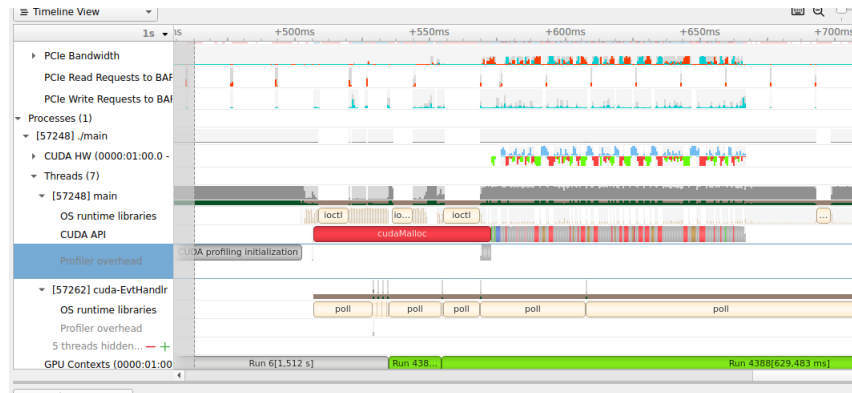
Par la suite, nous avons rassemblé toutes les fonctions de la pipeline GPU et les avons implémentées dans la fonction fix\_image\_gpu, que nous appelons ensuite dans la fonction main\_gpu.

Passons maintenant à la discussion de chaque kernel GPU que nous avons implémenté.

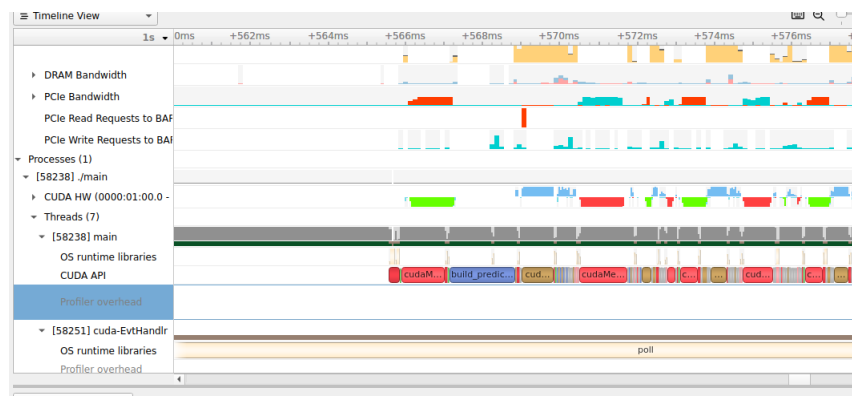
## 4 Initialisation

Après avoir achevé le projet, nous avons procédé à une phase de benchmarking. Nous avons utilisé Nsight Systems pour analyser notre pipeline. Nous avons remarqué qu'au début, l'allocation de mémoire CUDA(surtout le premier `cudaMalloc`) prenait du temps. Pour remédier à cela, nous avons pris l'initiative de définir notre device avant d'entamer les calculs.

Avant :



Après :



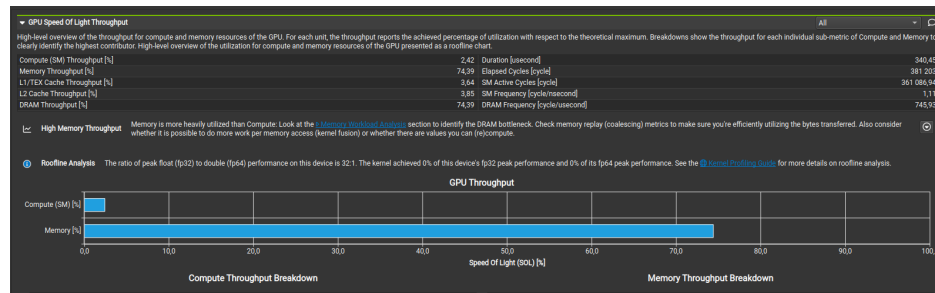
## 5 Step 1 : Build Predicate

### 5.1 Implémentation

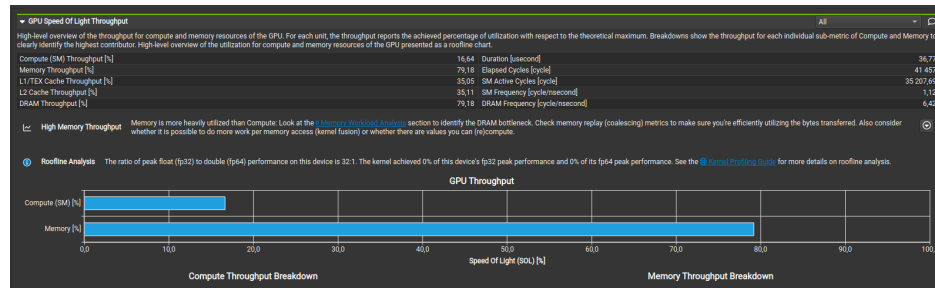
Pour la première étape, nous avons mis en place la construction de notre prédicat afin de pouvoir utiliser le compact pattern.

Le kernel que nous utilisons consiste simplement à attribuer la valeur 1 si le prédicat est valide, sinon 0.

### 5.2 Benchmark



On remarque que le kernel est memory bound car il effectue principalement des opérations d'accès à la mémoire plutôt que des calculs complexes. Nous avons tenté d'appliquer le grid stride loop pattern en effectuant quatre itérations, en réduisant la taille de notre grille de calcul par un facteur de quatre.



Nous avons réussi à augmenter légèrement notre utilisation de la mémoire.

## 6 Step 2 : Scan

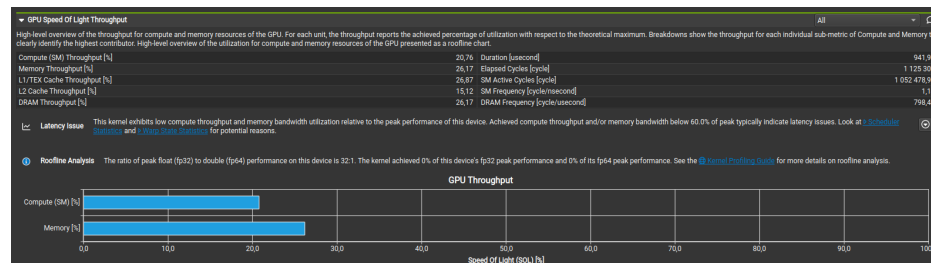
### 6.1 Implémentation

Le scan pattern est l'un des plus intéressants. Tout d'abord, nous avons créé une version qui utilise trois kernels. Ensuite, nous avons migré vers la version "decoupled look-back" afin de minimiser le nombre de noyaux requis.

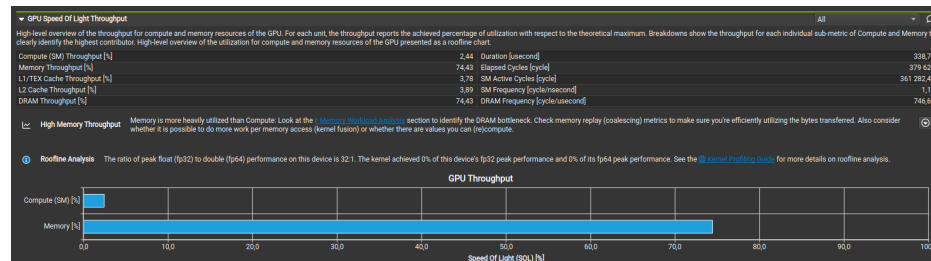
Nous avons rencontré un défi lors de la mise en oeuvre d'un scan exclusif en utilisant nos deux méthodes. C'est pourquoi nous avons développé un kernel qui décale les données de manière à ce que le premier élément soit 0 et que le dernier élément soit la valeur précédente à l'avant-dernier élément.

## 6.2 Benchmark

## 6.3 Scan



## 6.4 Shift

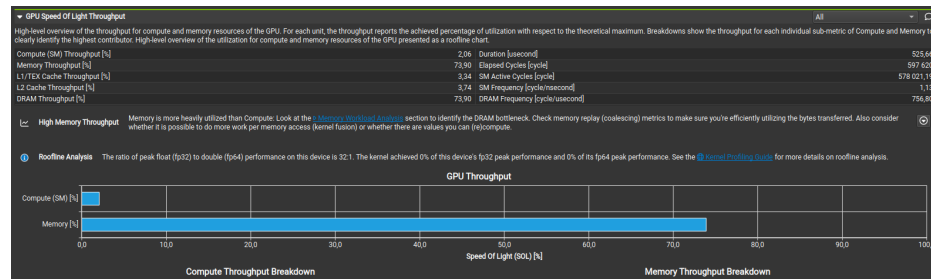


## 7 Step 3 : Scatter Addresses

### 7.1 Implémentation

Ensuite, nous avons mis en place le kernel qui applique le prédicat sur le buffer de notre image sur lequel nous avons appliqué le scan.

## 7.2 Benchmark



## 8 Step 4 : Apply Map

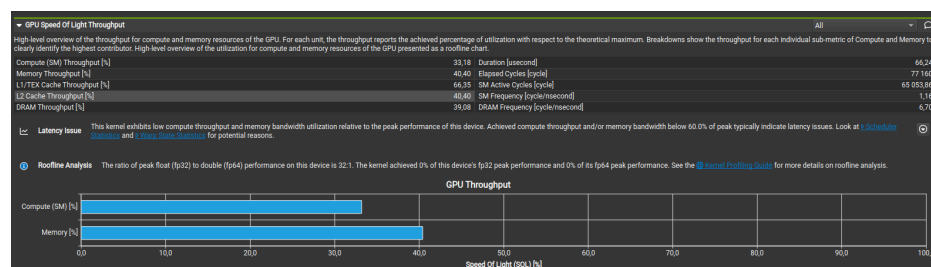
### 8.1 Implémentation

Par la suite, nous faisons un mapping du buffer en utilisant une suite de quatre valeurs (+1, -5, +3, -8).

Au début, nous avons développé une version qui utilisait des if-else. Cependant, cela entraînait des divergences de branches. C'est pourquoi nous avons choisi de passer à une version qui utilise de la constant memory en appliquant un opérateur "& 3" à la place d'un "% 4" sur le pixel, ce qui permet de mapper avec quatre valeurs distinctes.

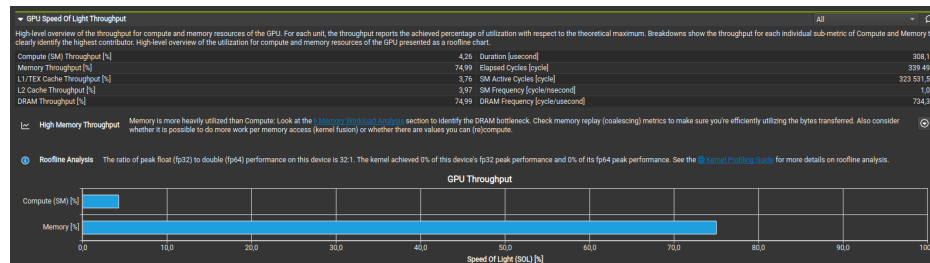
### 8.2 Benchmark

Voici la première version avec la divergence de branches.



et voici la dernière version avec la constant memory et le remplacement du "%" avec "&".



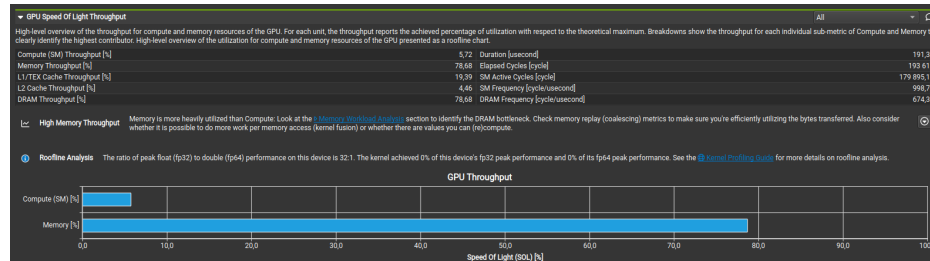


## 9 Step 5 : Histogram

### 9.1 Implémentation

Ensuite, nous avons mis en place un kernel pour calculer l'histogramme. Nous avons implémenté deux kernels pour effectuer ce calcul : l'un utilise des opérations atomiques et l'autre utilise la shared memory.

### 9.2 Benchmark



## 10 Step 6 : First Non-Zero

### 10.1 Implémentation

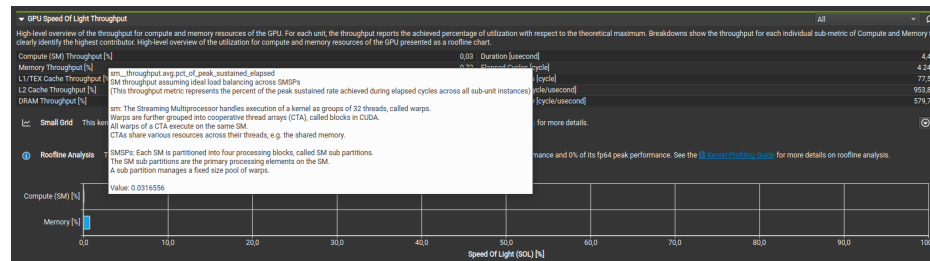
Dans un premier temps, pour cette étape, nous avons implémenté un kernel qui tente de retourner la première valeur différente de zéro en utilisant simplement une instruction conditionnelle (if). Cependant, CUDA fonctionne en mode SIMT, ce qui signifie que 32 threads exécutent la même instruction en parallèle. Par conséquent, on ne peut pas prédire quel thread trouvera la première valeur différente de zéro si nos valeurs au début de notre histogramme sont 0, 0, 2, 3, 4 par exemple. Il est possible que le programme renvoie 3 ou 4 plutôt que 2, ce qui pose problème. C'est pourquoi nous avons opté pour l'utilisation du pattern compact pour résoudre ce problème.

Dans un premier temps, nous avons créé un prédicat qui renvoie 1 si la valeur est différente de 0. Prenons un exemple : supposons que dans notre histogramme,

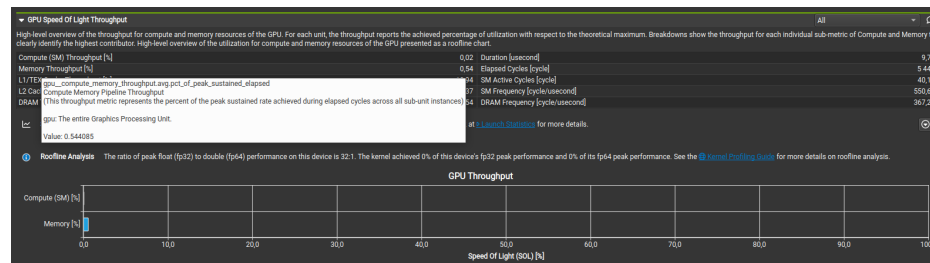
nous avons les valeurs 0, 0, 2, 3, 4. Notre prédicat renverra 0, 0, 1, 1, 1. Ensuite, nous appliquons un scan inclusive sur ce prédicat, obtenant ainsi 0, 0, 1, 2, 3. Enfin, nous renvoyons l'élément correspondant à la valeur 1 dans le prédicat cumulé.

## 10.2 Benchmark

### 10.2.1 Predicate Zeros



### 10.2.2 Find First Non Zero

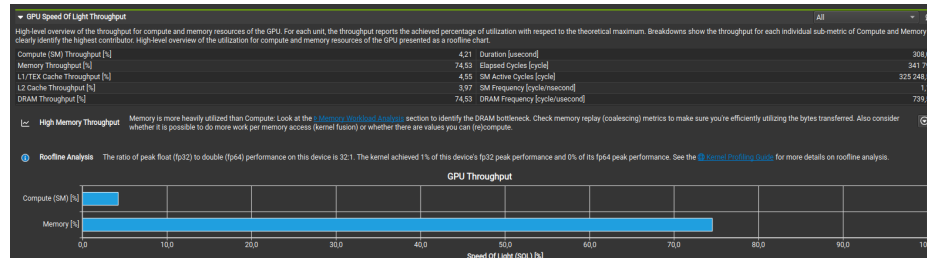


## 11 Step 7 : Apply Map Transformation

### 11.1 Implémentation

Pour cette fonction, nous avons simplement effectué une normalisation en utilisant `std::roundf` pour chaque pixel.

## 11.2 Benchmark



## 12 Step 8 : Reduce

### 12.1 Implémentation

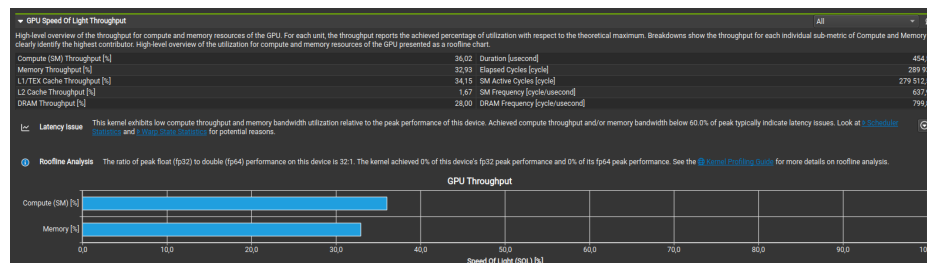
Fichiers: `kernels.cu`, `kernels.cuh`

Tout d'abord, nous avons implémenté la version naïve de la réduction, puis nous avons continué jusqu'à mettre en place neuf versions différentes.

### 12.2 Intégration

Après avoir examiné le code existant, nous avons noté que l'opération de réduction était effectuée dans un autre bloc omp parallèle for. Pour éviter cette boucle supplémentaire, nous avons introduit une fonction appelée `compute_reduce` qui déclenche un kernel de réduction immédiatement après l'exécution de la fonction `fix_image_gpu`.

### 12.3 Benchmark



## 13 Step 9 : Sort

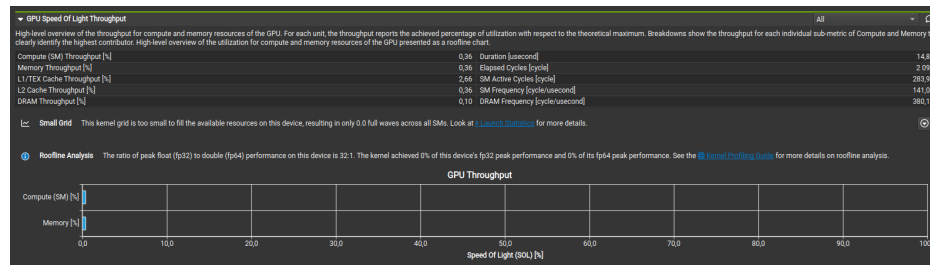
### 13.1 Implémentation

Nous avons décidé d’approfondir notre travail et avons donc mis en place l’algorithme de tri radix en suivant un article, qui traite 2 bits à chaque passage des entiers.

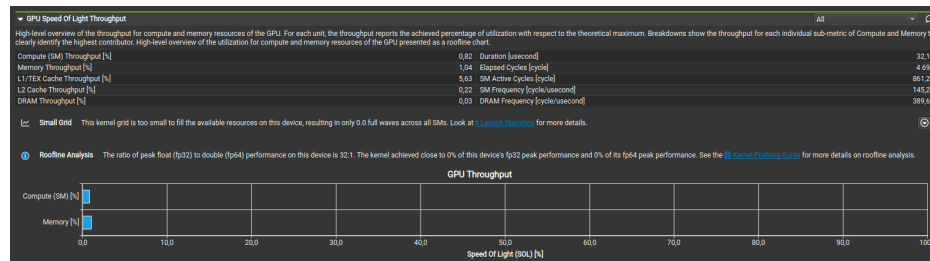
Voici le lien de l’article dont nous nous sommes inspiré : ”article link”.

### 13.2 Benchmark

#### 13.2.1 Order Checking



#### 13.2.2 Radix Sort



## 14 Transfert de données

Pour le transfert de données, nous avons remplacé les allocations de mémoire dynamique (malloc) dans la classe Image par des appels à cudaMallocHost. Cela permet de réserver de la pinned memory, ce qui accélère le transfert entre le CPU et le GPU.

## 15 Implémentation : Thrust

### 15.1 Implémentation

Tout d'abord, nous définissons des functor: une structure `Predicate` qui est utilisée comme prédicat pour filtrer les éléments du vecteur. Dans ce cas, la fonction `operator()` de `Predicate` renvoie `true` si l'entier donné n'est pas égal à -27. Ensuite, nous avons une structure `ApplyMapFunctor` qui est utilisée pour appliquer une transformation aux éléments du vecteur. Elle prend l'indice global du thread (`gid`) en compte et modifie la valeur de l'élément du vecteur en lui ajoutant la valeur correspondante depuis le tableau constant `map_thrust`. La structure `NonZeroPredicate` est utilisée pour filtrer les éléments du vecteur en ne gardant que ceux qui ne sont pas égaux à zéro. La structure `ApplyMapTransformation` est une fonction de transformation qui prend en compte un vecteur d'histogramme, le premier élément non nul et la taille, et renvoie un nouvel élément calculé en fonction de ces paramètres. Cela ressemble à une étape de normalisation.

Pour le calcul, nous utilisons `thrust::copy_if` pour appliquer le prédicat `Predicate` afin de filtrer les éléments du vecteur `d_image`. Ensuite, nous utilisons `thrust::for_each_n` avec le `ApplyMapFunctor` pour appliquer la transformation aux pixels de l'image.

Nous effectuons une égalisation d'histogramme en suivant les étapes suivantes :

- Nous créons un vecteur `d_histo` de taille 256 initialisé à zéro pour stocker l'histogramme.
- Nous faisons une copie triée du vecteur `d_image` et utilisons `thrust::upper_bound` pour calculer l'histogramme.
- Puis, nous calculons la différence adjacente pour obtenir l'histogramme cumulatif.
- Ensuite, nous effectuons une somme préfixe inclusive sur l'histogramme cumulatif.

Nous trouvons le premier élément non nul dans l'histogramme cumulatif en utilisant `thrust::find_if` et utilisons cette valeur pour normaliser les pixels de l'image à l'aide de `thrust::transform`.

## 16 Final benchmark

Voici le temps de nos version :

```

CPU time: 832 ms
GPU time: 646 ms
Thrust time: 624 ms
Comparing results CPU & GPU
Success: images are equal
Comparing results CPU & Thrust
Success: images are equal

```

et voici le benchmark avec nvprof pour savoir le pourcentage que prennent nos kernels sur la pipeline.

```

==38184== Profiling application: ./main --version gpu
==38184== Profiling result:

```

| Type            | Time(%) | Time     | Calls | Avg      | Min      | Max      | Name   |
|-----------------|---------|----------|-------|----------|----------|----------|--|
| GPU activities: | 38.19%  | 22.587ms | 32    | 705.83us | 480ns    | 2.4811ms | [CUDA memcpy HtoD]   |
|                 | 29.58%  | 22.132ms | 77    | 287.42us | 927ns    | 2.4994ms | [CUDA memcpy DtoH]   |
|                 | 22.51%  | 16.846ms | 106   | 158.93us | 5.5670us | 1.8326ms | decoupled look_back(int*, int*, int*, cuda::std::atomic<char>*, int) |
|                 | 6.19%   | 4.6326ms | 30    | 154.42us | 11.041us | 518.19us | reduce1(int const*, int*, int)                                       |
|                 | 2.10%   | 1.5679ms | 30    | 52.263us | 3.1370us | 177.76us | scatter_adresses(int*, int*, int)                                    |
|                 | 1.93%   | 1.4421ms | 537   | 2.6859us | 416ns    | 41.121us | [CUDA memcpy]  |
|                 | 1.72%   | 1.2893ms | 30    | 42.976us | 3.2320us | 142.05us | compute_histogram2(int*, int*, int)                                  |
|                 | 1.56%   | 1.1700ms | 30    | 38.999us | 3.2960us | 136.74us | apply_map_transformation1(int*, int*, int*, int)                     |
|                 | 1.56%   | 1.1658ms | 46    | 25.343us | 1.5360us | 127.33us | shift_buffer(int*, int*, int)  |
|                 | 1.38%   | 1.0315ms | 30    | 34.383us | 2.7200us | 117.60us | apply_map4(int*, int)  |
|                 | 1.01%   | 752.75us | 30    | 25.091us | 2.6240us | 82.817us | build_predicate3(int const*, int*, int)                              |
|                 | 0.87%   | 51.873us | 16    | 3.2420us | 3.2000us | 3.3200us | radix_sort(int*, int*, int*, int*, int)                              |
|                 | 0.87%   | 51.749us | 30    | 1.7240us | 1.6600us | 1.7610us | find_first_non_zero(int*, int*, int*, int)                           |
|                 | 0.86%   | 46.945us | 30    | 1.5640us | 1.5360us | 1.6320us | build_predicate_zeros1(int const*, int*, int)                        |
|                 | 0.84%   | 28.194us | 16    | 1.7620us | 1.6960us | 2.1120us | order_checking(int*, int*, int)                                      |
|                 | 0.84%   | 28.128us | 16    | 1.7580us | 1.7280us | 1.9200us | compute_new_position(int*, int*, int*, int*, int, int)               |
| API calls:      | 32.54%  | 88.706ms | 1     | 88.706ms | 88.706ms | 88.706ms | cudaSetDevice  |
|                 | 18.03%  | 49.141ms | 109   | 450.83us | 4.5260us | 2.5283ms | cudaMemcpy   |
|                 | 14.57%  | 39.725ms | 30    | 1.3242ms | 10.787us | 4.3288ms | cudaMallocHost   |
|                 | 11.08%  | 30.214ms | 440   | 68.667us | 2.7520us | 1.8359ms | cudaDeviceSynchronize  |
|                 | 7.34%   | 20.000ms | 1     | 20.000ms | 20.000ms | 20.000ms | cudaDeviceReset  |
|                 | 6.53%   | 17.799ms | 30    | 593.38us | 7.6450us | 1.8929ms | cudaFreeHost   |
|                 | 5.85%   | 15.950ms | 440   | 36.249us | 2.3170us | 14.365ms | cudaLaunchKernel   |
|                 | 1.95%   | 5.3019ms | 310   | 17.102us | 2.1760us | 169.86us | cudaMalloc   |
|                 | 1.40%   | 3.8266ms | 280   | 13.645us | 1.7260us | 102.39us | cudaFree   |
|                 | 0.66%   | 1.6463ms | 537   | 3.0650us | 1.2630us | 35.320us | cudaMemset   |
|                 | 0.09%   | 237.37us | 114   | 2.0820us | 120ns    | 84.394us | cudaDeviceGetAttribute   |
|                 | 0.06%   | 10.029us | 60    | 167ns    | 121ns    | 333ns    | cudaGetLastError   |
|                 | 0.00%   | 9.4360us | 1     | 9.4360us | 9.4360us | 9.4360us | cudaDeviceGetName  |
|                 | 0.00%   | 6.4950us | 1     | 6.4950us | 6.4950us | 6.4950us | cudaDeviceGetPCIBusId  |
|                 | 0.00%   | 2.6120us | 1     | 2.6120us | 2.6120us | 2.6120us | cudaDeviceTotalMem   |
|                 | 0.00%   | 1.1700us | 3     | 390ns    | 177ns    | 788ns    | cudaDeviceGetCount   |
|                 | 0.00%   | 652ns    | 2     | 326ns    | 129ns    | 523ns    | cudaDeviceGet  |
|                 | 0.00%   | 248ns    | 1     | 248ns    | 248ns    | 248ns    | cudaModuleGetLoadingMode   |
|                 | 0.00%   | 206ns    | 1     | 206ns    | 206ns    | 206ns    | cudaDeviceGetUuid  |

## 17 Conclusion

En conclusion, notre projet intègre de nombreuses techniques de parallélisation, en mettant particulièrement l'accent sur l'utilisation de CUDA. Nous avons cherché à tirer parti des streams pour permettre le chargement et le calcul en parallèle, afin d'exploiter davantage les capacités de parallélisation du GPU. Cela aurait permis d'optimiser davantage les performances de notre programme.

Nous avons appris énormément sur la conception de systèmes parallèles, l'optimisation de code GPU, l'utilisation efficace de la shared memory, et la gestion des transferts de données entre le CPU et le GPU. De plus, l'intégration de Thrust a enrichi notre compréhension des bibliothèques CUDA et de leur potentiel pour accélérer les calculs complexes. Enfin, l'exploration des streams CUDA a ouvert de nouvelles perspectives pour exploiter pleinement les capacités de parallélisation de notre projet. Ce voyage d'apprentissage continu a été riche en découvertes et a renforcé notre expertise dans le domaine de la programmation parallèle.