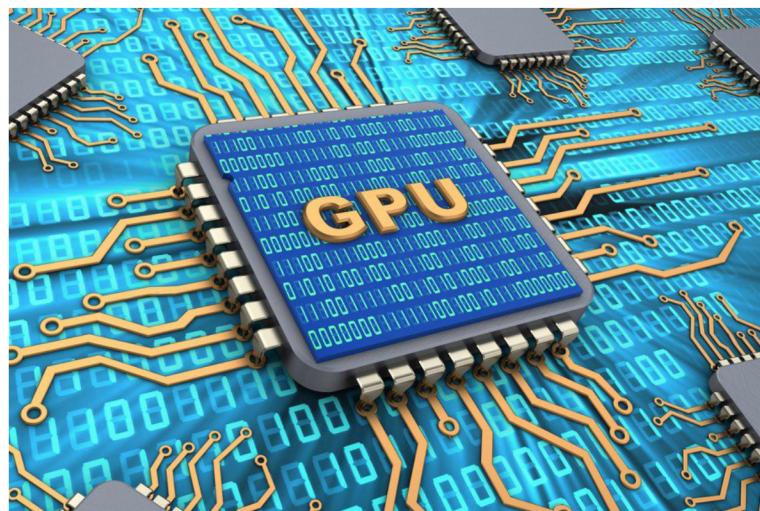


Rapport GPGPU : Détection de code barre

Merlin SCHOOSE Marie-Charlotte CALISTO

Simon QUESNEY Alan LEBLANC

Juin 2021



L'objectif de ce projet reposait sur la détection de codes barre sur des images en temps réel, afin de pouvoir appliquer cette solution à du traitement de flux vidéo.

Pour ce faire, le problème est divisé en plusieurs sous-catégories.

Tout d'abord les descripteurs visuels *LBP* (Local Binary Patterns) sont calculés pour permettre de représenter plus succinctement et efficacement l'image traitée. L'algorithme *LBP* repose sur la division de l'image en patchs de taille fixe, desquelles on extrait des "textons", permettant de représenter des données de textures. On peut alors calculer un histogramme des textons de notre patch et utiliser ce dernier pour représenter le patch pour la suite du pipeline.

Il est alors possible classifier les textures de l'image, grâce à un algorithme de clustering non-supervisé, dans notre cas l'algorithme *K-means*. Une fois les centroïdes trouvés, nous classifions chaque patch en fonction de leur plus proche centroïde avec un algorithme de plus proche voisins. On les colorie ensuite en fonction de leur centroïde.

Ce résultat peut permettre de trouver le code barre sur une image si on arrive à trouver le cluster qui représente le mieux les codes barres et détecter dans l'image les zones où ce cluster est très présent. Lors de ce projet nous nous sommes concentrés sur les deux premières parties de cette méthode, à savoir l'algorithme des *LBP* ainsi que le clustering et la colorisation de l'image en fonction des clusters.

Notre version CPU de l'algorithme est implémenté en C++, en utilisant la bibliothèque OpenCV pour la gestion des images et la classification *K-Means*. Cela nous permet d'éviter de passer les données générées par l'algorithme *LBP* en python avant de faire la classification.

De façon plus spécifique, l'image traitée est chargée en niveaux de gris et est traitée avec le type `cv::Mat` (d'OpenCV), qui permet de manipuler des matrices simplement. La taille de la matrice en question est alors mise à jour, afin que l'image soit divisible en patchs de 16×16 . On extrait ensuite chaque patch de l'image grâce à la classe `cv::Rect` afin d'en récupérer les textons. On crée un vecteur contenant les textons de chaque pixel (calculés à l'aide de décalages logiques de bits), avant de calculer l'histogramme du patch. Ce dernier est ajouté dans une autre `cv::Mat` qui contient donc finalement les histogrammes pour chaque patchs de l'image.

Une fois les histogrammes calculés, on applique un algorithme de *nearest neighbours* afin de lier chaque patch au centroïde de *K-Means* qui lui est le plus proche, avant d'associer une couleur aléatoire à chaque centroïde et de recolorer les patchs. Pour éviter de ré-entraîner *K-Means* à chaque exécution, ce dernier a été entraîné une fois, en prenant en compte de nombreuses images (issues de la base de données collaborative), puis les centroïdes ont été stockés dans un fichier. Les centroïdes peuvent donc être chargés à chaque exécution, ce qui augmente largement sa vitesse d'exécution totale.

On obtient alors une image reconstruite, avec chaque patch composée d'une couleur unique, dépendant du centroïde qui lui correspond le mieux.

1 Implémentation GPU naïve

Cette implémentation utilise le même chargement d'image que la version CPU avec OpenCV et sous forme de `cv::Mat`.

La partie GPU prend en paramètre l'image et ses dimensions allouées sur le host ainsi que la matrice d'histogrammes pré allouée sur le host.

Les premières étapes consistent donc à allouer de la mémoire sur le device (pour la matrice d'histogramme et l'image à traiter) et à copier les données de l'image du host vers le device. Dans les deux cas, ces opérations se font en 2D. On peut ensuite exécuter le kernel, avant de copier les histogrammes sur le host et de libérer la mémoire utilisée sur l'image par le device. Toute les étapes du *LBP* sont

faites en un seul kernel.

Le kernel est exécuté sur des blocs de 16×16 sur toute l'image pour émuler au mieux le découpage en patch de *LBP*. Pour chaque thread nous avons stocké le patch dans la mémoire partagée du bloc.

Nous avons utilisé les mêmes fonctions que l'implémentation CPU sur le device pour récupérer le texton du pixel actuel pour chaque thread du bloc. Les textons sont stockés dans la shared memory. Ensuite nous avons stocké l'histogramme du patch dans la shared memory et nous l'avons initialisé à 0 en single thread. Pour le remplissage de l'histogramme, nous avons utilisé des `atomicInc` avec les valeurs des textons. Nous avons ensuite remplis l'histogramme global du bloc avec les valeurs de l'histogramme local en single thread.

Cette version totalise 3 `__syncthreads` pour les phases de remplissage des trois variables shared. Cette version a comme défaut de faire des opérations en single thread avec l'histogramme ainsi que certaines shared variables inutiles.

De plus, le reste de la pipeline (nearest neighbour et colorisation) est toujours exécutée sur le CPU.

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	4.41%	4.6622ms	1	4.6622ms	4.6622ms	4.6622ms	[CUDA memcpy DtoH]
	42.15%	4.3801ms	1	4.3801ms	4.3801ms	4.3801ms	kernel(unsigned char*, int, int, unsigned long, unsigned char*, unsigned long)
	12.99%	1.3504ms	1	1.3504ms	1.3504ms	1.3504ms	[CUDA memcpy HtoD]
API calls:	92.43%	141.97ms	2	70.983ms	94.076us	141.87ms	cudaMallocPitch
	4.41%	6.7661ms	2	3.3830ms	1.3879ms	5.3782ms	cudaMemcpy2D
	2.89%	4.4368ms	1	4.4368ms	4.4368ms	4.4368ms	cudaDeviceSynchronize
	0.09%	141.53us	1	141.53us	141.53us	141.53us	cuDeviceTotalMem
	0.07%	103.54us	107	1.1240us	109ns	48.753us	cuDeviceGetAttribute
	0.07%	103.09us	1	103.09us	103.09us	103.09us	cudaFree
	0.02%	28.01us	1	28.01us	28.01us	28.01us	cuDeviceGetName
	0.02%	23.062us	1	23.062us	23.062us	23.062us	cudaLaunchKernel
	0.00%	6.1340us	1	6.1340us	6.1340us	6.1340us	cuDeviceGetCIBusId
	0.00%	1.5340us	3	511ns	215ns	1.1020us	cuDeviceGetCount
	0.00%	791ns	2	395ns	126ns	665ns	cuDeviceGet
	0.00%	355ns	1	355ns	355ns	355ns	cudaPeekAtLastError
	0.00%	215ns	1	215ns	215ns	215ns	cuDeviceGetUuid

FIGURE 1 – nvprof GPU

On voit avec ce nvprof que la majorité du temps passé par le GPU est dans l'allocation de la mémoire. Hors des appels API le kernel de *LBP* prend 4ms.

2 Rendu vidéo

Pour pouvoir traiter des vidéos en plus de photos, nous avons utilisé le module `Video I/O` d'`OpenCV` pour lire et écrire image par image.

On a ensuite utilisé nos implémentations précédentes pour appliquer la *LBP* puis la classification des histogrammes.

Le résultat est enregistré dans un fichier `mp4` (il n'est donc pas visualisable en temps réel).

Nous avons également essayé une entrée vidéo webcam à la place de l'entrée par fichier (facile à faire avec `OpenCV`).

Cela n'a cependant pas été inclus dans le rendu final à cause de problèmes matériels qui nous ont empêcher de suffisamment tester cette fonctionnalité... (checksum error on block 0, adieu manjaro 😢)

3 Benchmark et Optimisations

3.1 Benchmarks des versions naïves

Pour étudier et comparer les versions de l'algorithme implémentées, il est crucial de mettre en place des moyens de benchmark ces dernières. Dans cette optique, nous avons utilisé la librairie `Google`

Benchmark, qui permet de se munir de métriques pertinentes et efficaces pour évaluer une implémentation.

En particulier, nous avons utilisé le temps total d'exécution par itération de l'algorithme ainsi que le frame rate comme métriques principales. En effet, le frame rate est très pertinent pour ce problème, car l'objectif final serait de traiter une vidéo en temps réel.

```
2021-06-21T15:07:55+02:00
Running ./bench
Run on (8 X 3800 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x4)
  L1 Instruction 32 KiB (x4)
  L2 Unified 256 KiB (x4)
  L3 Unified 6144 KiB (x1)
Load Average: 0.56, 0.77, 0.65
***WARNING*** CPU scaling is enabled, the benchmark real time measurements may be noisy and will incur extra overhead.

Benchmark      Time      CPU Iterations UserCounters...
BM_Pipeline_cpu/real_time    597 ms      597 ms      1 frame_rate=1.67409/s
BM_Pipeline_gpu/real_time    515 ms      510 ms      1 frame_rate=1.94244/s
BM_LBP_cpu/real_time       168 ms      168 ms      4 frame_rate=5.95854/s
BM_LBP_gpu/real_time       11.4 ms     11.3 ms    61 frame_rate=87.416/s
```

FIGURE 2 – Benchmark des version naïves CPU et GPU

Pour le rendu vidéo, nous nous sommes intéressé au rapport entre le temps nécessaire au traitement de la vidéo et la durée de la vidéo.

Ce rapport est équivalent au ratio entre le nombre d'images calculées par seconde (`compute_fps`) et le nombre d'images par secondes de la vidéo (`real_fps`).

Pour que la détection de code barre puisse se faire en temps réel, il faut que ce ratio soit supérieur à 1 (c'est à dire qu'on finisse de calculer une image avant que la prochaine arrive dans la chronologie de la vidéo).

Comme on peut le voir, on s'approche d'un rendu temps réel avec l'implémentation GPU-OPTI (avec un ratio de 0.9).

```
Run on (8 X 2500 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x8)
  L1 Instruction 32 KiB (x8)
  L2 Unified 4096 KiB (x8)
  L3 Unified 16384 KiB (x8)
Load Average: 0.07, 0.26, 0.38

Benchmark      Time      CPU Iterations UserCounters...
BM_Pipeline_cpu/real_time    865 ms      857 ms      1 frame_rate=1.1566/s
BM_Pipeline_gpu/real_time   1364 ms     1286 ms      1 frame_rate=0.73337/s
BM_Pipeline_gpu_opti/real_time 123 ms      123 ms      6 frame_rate=8.11315/s
BM_LBP_cpu/real_time        241 ms      240 ms      3 frame_rate=4.15572/s
BM_LBP_gpu/real_time        28.7 ms     28.6 ms     26 frame_rate=34.8979/s
BM_LBP_gpu_opti/real_time   19.5 ms     19.5 ms    35 frame_rate=51.1884/s
BM_Pipeline_cpu_video/real_time 66.0 s      65.9 s      1 compute_fps=6.52845/s compute_fps/real_fps=0.217833 real_fps=29.97
BM_Pipeline_gpu_video/real_time 53.5 s      53.4 s      1 compute_fps=8.05262/s compute_fps/real_fps=0.208689 real_fps=29.97
BM_Pipeline_gpu_opti_video/real_time 15.9 s     15.8 s      1 compute_fps=27.1696/s compute_fps/real_fps=0.906559 real_fps=29.97
```

FIGURE 3 – Benchmark des version naïves CPU et GPU pour les images et les vidéos

3.2 Optimisation GPU

Pour commencer la version GPU optimisée nous avons tout d'abord minimisé le nombre de variables partagées que nous avions, plus particulièrement l'initialisation des textons qui en plus d'être partagée, rajoutait un appel à `__syncthreads`.

À la place, nous accédons directement à l'indice de l'histogramme via la fonction `get_texton()` prenant en argument le pixel actuel. Puis au lieu d'initialiser l'histogramme et les valeurs des pixels sur

un seul thread en utilisant une boucle, nous les avons calculé en parallèle. Notre nombre de thread est équivalent à la taille de l'histogramme ($16 \times 16 = 256$) ce qui permet à chaque thread d'initialiser une seule cellule de l'histogramme.

Nous avons également implémenté la recherche des plus proches voisins entre les histogrammes des patch et les centroïdes sur GPU. En effet, l'entraînement de *K-Means* n'étant plus exécuté dans le pipeline classique, il peut être intéressant de passer la recherche des centroïde correspondant à chaque patch et la colorisation de l'image finale sur GPU. Cela permet d'augmenter nettement le pipeline global du traitement de l'image fournie.

De plus, les histogrammes de chaque patchs étant déjà calculé sur le GPU, il est possible d'éviter de copier ces derniers sur le host pour calculer le nearest neighbours, ce qui optimise les accès mémoires.

Pour le kernel `nearest_neighbour`, chaque bloc s'occupe d'un histogramme et chaque thread d'un élément de l'histogramme.

Nous avons calculé la distance entre l'histogramme du bloc actuel et chaque centroïde. Pour cela, nous avons créé un tableau shared d'une taille K qui contient les distances de chaque centroïde que nous accumulons avec `atomicAdd`. Une fois le calcul des distances fait, nous faisons un argmin en single thread pour avoir le label.

Nous faisons aussi la partie colorisation sur GPU avec un autre kernel qui map seulement les labels en couleurs avec notre tableau de couleurs aléatoires.

3.3 Benchmarks et résultats finaux

Benchmark	Time	CPU	Iterations	UserCounters...
BM_Pipeline_cpu/real_time	592 ms	591 ms	1	frame_rate=1.69042/s
BM_Pipeline_gpu/real_time	427 ms	427 ms	2	frame_rate=2.3423/s
BM_Pipeline_gpu_opti/real_time	296 ms	296 ms	2	frame_rate=3.37848/s
BM_LBP_cpu/real_time	179 ms	179 ms	4	frame_rate=5.59939/s
BM_LBP_gpu/real_time	10.8 ms	10.8 ms	65	frame_rate=92.8561/s
BM_LBP_gpu_opti/real_time	8.18 ms	8.17 ms	87	frame_rate=122.234/s

FIGURE 4 – Benchmark Final

Grâce au benchmark, on peut voir que la version GPU optimisée est la plus rapide avec un frame rate de 122.234/s pour la partie *LBP* contre 5.59939/s en CPU et 92.8561/s avec le GPU classique. Pour la pipeline globale (donc *LBP* + *nearest_neighbour* + colorisation), le GPU optimisé est à 296ms tandis que la version GPU + *nearest neighbour* CPU est à 427ms et la version totalement CPU est à 600ms pour une image de 4032×3024 .

Type	Time(x)	Time	Calls	Avg	Min	Max	Name
GPU activities:							
	93.67%	296.19ms	1	296.19ms	296.19ms	296.19ms	nearest_neighbour(unsigned char*, unsigned int, unsigned int, unsigned long, float*, unsigned long, int*)
	4.02%	15.590ms	1	15.590ms	15.595ms	15.595ms	[CUDA memcpy DtoH]
	0.65%	2.452ms	1	2.452ms	2.452ms	2.452ms	[CUDA memcpy HtoD]
	0.41%	1.3059ms	3	435.30us	832ns	1.3029ms	[CUDA memcpy HtoD]
	0.35%	1.1219ms	1	1.1219ms	1.1219ms	1.1219ms	[CUDA memcpy HtoD]
	0.18%	2.299ms	1	2.299ms	2.299ms	2.299ms	colorize(unsigned char*, unsigned int, unsigned int, unsigned long, int const *, unsigned char const *)
	0.17%	2.2994ms	4	749.85us	165.25us	2.2994ms	[CUDA memcpy HtoD]
	3.60%	17.444ms	3	5.8147ms	6.9080ms	16.097ms	[CUDA memcpy DtoH]
	0.21%	1.0150ms	6	169.16us	3.5990us	737.21us	cudaMemcpy2D
	0.07%	342.41us	2	171.21us	3.0800us	342.41us	cudaFree
	0.04%	10.130ms	101	10.130ms	146.300us	10.130ms	cudaMalloc
	0.05%	255.22us	1	255.22us	255.22us	255.22us	cudaDeviceTotalMem
	0.02%	80.459us	1	80.459us	80.459us	80.459us	cuDeviceGetName
	0.01%	47.217us	3	15.739us	8.7180us	20.911us	cudaLaunchKernel
	0.01%	47.217us	1	8.4640us	8.4640us	8.4640us	cudaLaunchKernel
	0.00%	5.0640us	1	5.0640us	5.0640us	5.0640us	cuDeviceGetPCIBusId
	0.00%	1.9190us	3	639ns	303ns	1.2630us	cuDeviceGetCount
	0.00%	1.3400us	3	446ns	246ns	741ns	cudaPeekAtLastError
	0.00%	1.2400us	2	624ns	219ns	1.0000us	cuDeviceGetCudaEvent
	0.00%	446ns	1	446ns	446ns	446ns	cuDeviceGetCudaEvent

FIGURE 5 – nvprof pipeline GPU optimal

Sur nvprof, on voit que la grande majorité du temps de calcul GPU pour la pipeline optimale est dans le kernel de nearest neighbour. En effet le kernel utilise beaucoup de `atomicAdd` et d'opérations en single thread car il faut comparer chaque histogramme avec K centroïdes. Or, la taille maximale d'un bloc est de 1024 threads, or pour totalement paralléliser les calculs des distances il faudrait un bloc de taille $256 \times K$, ce qui n'est pas possible pour $K > 4$. Et l'accès à la mémoire des centroïdes est naïve.



FIGURE 6 – Exemple de Résultats

4 Conclusion

En conclusion, nous avons en premier lieu implémenté une version CPU fonctionnelle afin de servir de base et de commencer à appréhender le fonctionnement de l'algorithme.

Par la suite, nous avons mis en place une première version implémentée sur GPU de l'algorithme *LBP*. Cette version contenait déjà quelques optimisations mais servait principalement de comparaison pour commencer à optimiser.

Nous avons alors commencé à mettre en place des benchmarks pour comparer nos différentes versions, ce qui nous a permis d'implémenter une nouvelle version plus optimisée de l'algorithme *LBP*, ainsi qu'une version GPU de l'algorithme de recherche de plus proche voisin et de la colorisation de l'image finale.

Nous sommes finalement relativement satisfait des résultats de ce projet, même si nous aurions aimé passer plus de temps à optimiser nos versions GPUs, mais nous avons perdu beaucoup de temps au début du projet sur des problématiques d'organisations ou des problèmes annexes, comme le linkage des différentes librairies.

Globalement, le travail a été effectué en groupe, en pair programming, et en conséquence, la question d'une répartition spécifique du travail effectué n'a pas vraiment de sens, puisque tout a été fait par tout le groupe en simultané.