

Rapport Lab1 CUDA

POSNIC Antoine, CUSSON Thomas

4 avril 2018

Résumé

Rapport concernant le premier travail pratique en parallélisation.

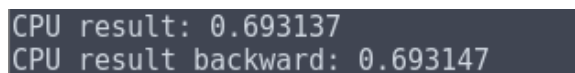
1 Introduction

Connecté à une machine distante fournie, avec deux GPU, le but de ce TP est de calculer en C la fonction \log_2 .

2 CPU

2.1 L'ordre des itérations

Lorsque l'on somme par ordre décroissant d'indice, on remarque que la fonction \log_2 est plus précise que son homologue dans l'ordre croissant. Car les nombre sur les grands indices ne sont pas approximés si additionnés en premiers (comportement du type float). On peut voir sur cette figure ci jointe la différence du résultat entre ces deux méthodes :



```
CPU result: 0.693137  
CPU result backward: 0.693147
```

FIGURE 1 – Différence Résultats

Le résultat du CPU backward étant plus proche, on optera pour cette méthode d'addition à l'avenir.

2.2 Vitesse d'exécution du calcul

```
int data_size = 1024 * 1024 * 128;
```

Data-size est le nombre d'itérations. Et pour cette valeur, le CPU met plus d'une seconde pour trouver le résultat de \log_2 .

On met donc beaucoup de temps pour trouver ce résultat, et nous allons donc paralléliser le calcul, à l'aide de GPUs dédiés.

```
log(2)=0.693147
time=1.062818s
```

FIGURE 2 – Temps CPU

3 GPU

3.1 Distributions des taches pour les threads

On ne peut avoir N threads qui calculent chacun leur valeur avant de tous s'additionner en pointer jumping. Ainsi, il faut trouver une manière de partager la tache à chaque threads.

On peut par exemple avoir une distribution une à une. C'est à dire, le premier thread s'occupera de toutes les :

```
num_element%num_thread == 1
```

Le deuxième, de l'élément de la série qui suit :

```
num_element%num_thread == 2
```

Et ainsi de suite.

Mais avec cette approche, une fois que chaque threads additionnent ses éléments calculés, on peut rencontrer une approximation comme expliqué au début. Si le nombre de thread est trop grand, il y aura un trop gros écart entre chaque valeur de chaque thread, et l'approximation aura lieu.

Par conséquent on va distribuer une partition à chaque thread. Ainsi chaque thread se voit attribuer un ensemble de valeurs adjacentes de la série. On évite ainsi des approximations en les faisant calculer de droite à gauche au sein de leur élément.

3.2 Vitesse de calcul

Une fois parallélisé sur les gpu, contrairement au single threaded algorithme séquentiel sur le cpu, on obtient un important gain de temps.

```
$ ./summation
CPU result: 0.693137
CPU result backward: 0.693147
log(2)=0.693147
time=1.545145s
GPU results:
Sum: 0.693147
Total time: 0.0845056 s,
Per iteration: 0.629616 ns
Throughput: 6.35308 GB/s
```

FIGURE 3 – Temps CPU/GPU

Avec pour paramètre :

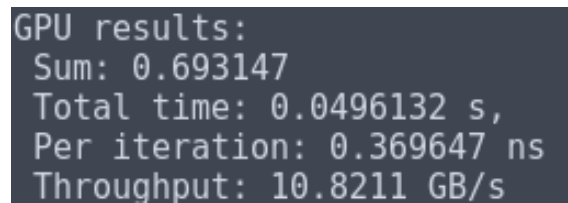
```
// Parameter GPU
```

```
int threads_per_block = 4 * 32;  
int blocks_in_grid = 8;
```

3.3 Modifions ces paramètres

Le nombre de thread par block et le nombre de block sont des paramètres modifiables qui jouent un rôle dans la vitesse d'exécution du calcul. En les augmentant on augmente les ressources fournies pour le calcul.

Ainsi, en doublant le nombre de threads par block sans changer le nombre de block :



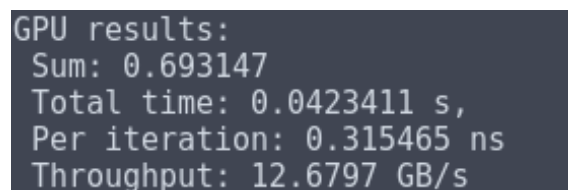
```
GPU results:  
Sum: 0.693147  
Total time: 0.0496132 s,  
Per iteration: 0.369647 ns  
Throughput: 10.8211 GB/s
```

FIGURE 4 – Temps GPU avec 2*thread

```
// Parameter GPU  
int threads_per_block = 8 * 32;  
int blocks_in_grid = 8;
```

On remarque que le calcul est presque deux fois plus rapide avec deux fois plus de thread par block, mais avec comme impact une augmentation significative (de près d'un tiers) de la bandwidth vers le CPU en retour.

Ensuite, en doublant le nombre de block sans changer le nombre de thread par block :



```
GPU results:  
Sum: 0.693147  
Total time: 0.0423411 s,  
Per iteration: 0.315465 ns  
Throughput: 12.6797 GB/s
```

FIGURE 5 – Temps GPU avec 2*block

```
// Parameter GPU  
int threads_per_block = 4 * 32;  
int blocks_in_grid = 16;
```

On remarque que le calcul est deux fois plus rapide avec deux fois plus de block (un peu mieux qu'avec 2*thread). Mais avec une augmentation encore plus grande de la bandwidth vers le CPU en retour.

Il semblerait donc qu'il faille maximiser le nombre de threads par blocks, plutôt que les blocks eux même.

4 Régression

Un souci avec notre précédente implémentation est qu'une fois que chaque block a calculé sa partie, ils repassent leur résultat au CPU. Un résultat possiblement très lourd pour la bandwidth. Une raison pour laquelle il faudrait que tout les thread d'un block calculent leur résultat combiné, pour diminuer l'impact sur la sortie du GPU.

C'est la partie régression, que nous avons pas implémenté.