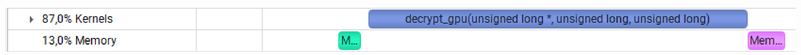
Exploiter les CUDA streams pour optimiser un code de chiffrement symétrique

1 Contexte

Les codes tel que le chiffrement symétrique Feistel, sont souvent limités par les temps d'attente liés aux transferts mémoire et à l'ordre séquentiel des tâches. Pour passer outre ces contraintes, nous avons découvert grâce à un cours NVIDIA, l'utilisation de multi-streams CUDA, qui permettent de superposer les opérations de transfert host↔device avec les calculs GPU. Sur la nouvelle plateforme ROMEO 2025, nous avons donc tester de mettre en place les différentes versions améliorées du code utilisant chacune différentes méthodes. Au long de ce poster nous verrons les spécificités de chaque version. ainsi qu'un petit profiling graphique avec NSIGHT Systems.

2 Version de base

Dans la version de base, le code commence par chiffrer en parallèle un tableau de 2^26 entiers sur le CPU à l'aide d'OpenMP et de la fonction permute64. Elle alloue ensuite de la mémoire page-locked sur l'hôte et sur le device, copie les données chiffrées vers le GPU, puis lance le kernel decrypt_gpu (avec un schéma grid-stride) pour inverser le chiffrement via unpermute64. Après le calcul, les résultats sont rapatriés sur le CPU pour être vérifiés grâce à check_result_cpu, et chaque phase (allocation, transferts, calcul, validation) est chronométrée via la classe Timer avant que la mémoire soit libérée. On peut voir ci-dessous qu'il y à la copie (en vert HostToDevice, le kernels sur le GPU et ensuite la copie DeviceToHost)



3 Kernel stream non par défaut

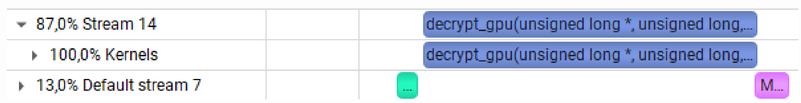
Pour exploiter l'exécution asynchrone et permettre un meilleur chevauchement entre calcul GPU et transferts, nous créons d'abord un stream CUDA dédié en ajoutant :

```
cudaStream_t stream;
cudaStreamCreate(&stream);
```

Le kernel est ensuite lancé sur ce stream non par défaut en passant stream comme quatrième paramètre de la commande suivante :

```
decrypt_gpu<<<gridDim, blockDim, 0, stream>>>(data_gpu, num_entries, num_iters);
```

Enfin, cudaStreamSynchronize(stream) garantit que toutes les opérations du stream sont terminées avant de détruire le stream avec cudaStreamDestroy(stream). Cette méthode isole le décryptage sur un autre stream.



Comme on peut voir la copie mémoire ce fait sur le stream par défaut 7 cependant le kernel lui s'execute sur le stream 14.

4 Memcpy in Non-Default Streams

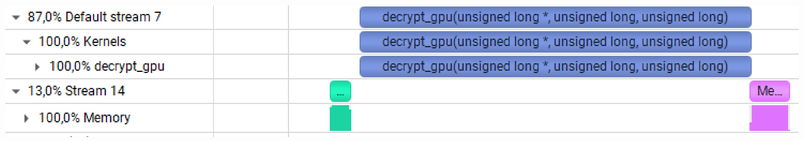
Pour optimiser le pipeline mémoire et overlaper copie et calcul GPU, on peut privilégier cudaMemcpyAsync sur un stream dédié plutôt que le cudaMemcpy bloquant. On peut alors spécifier un stream comme dernier argument.
Pour le HostToDevice :

```
cudaMemcpyAsync(data_gpu, data_cpu, sizeof(uint64_t)*num_entries,
                cudaMemcpyHostToDevice, stream);
```

Pour DeviceToHost :

```
cudaMemcpyAsync(data_cpu, data_gpu, sizeof(uint64_t)*num_entries,
                cudaMemcpyDeviceToHost, stream);
```

Cette méthode permet de réaliser un overlap efficace entre les transferts mémoire et l'exécution GPU, ce qui réduit les temps d'attente du CPU et du GPU, et maximise l'utilisation du bus PCIe.



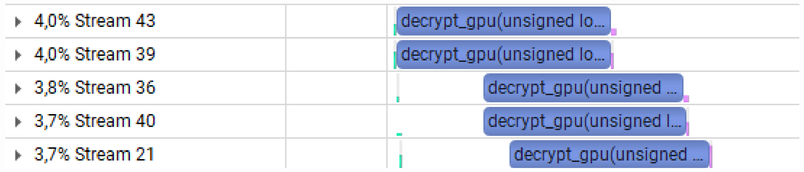
On peut maintenant voir que les transferts mémoires se font sur un stream non par défaut

5 Copy/compute overlap

Une autre technique qu'on peut ajouter est la découpe les données en N segments et, pour chaque chunk, on crée un stream dédié et on fait une boucle sur i allant de 1 à nb_streams :

```
cudaStreamCreate(&s[i]);
uint64_t offset = i * chunk;
uint64_t current_chunk = std::min(chunk, num_entries - offset);
cudaMemcpyAsync(d+offset, h+offset, sz, cudaMemcpyHostToDevice,
s[i]);
decrypt_gpu<<<blocks,threads,0,s[i]>>>(d+off,chunk,num_iters);
cudaMemcpyAsync(h+offset, d+offset, sz, cudaMemcpyDeviceToHost,
s[i]);
```

Chaque segment effectue H→D→GPU→D→H en parallèle, ce qui génére un overlap maximal entre transferts mémoire et calcul. Le CPU peut lancer tous les envois sans attendre, tandis que le GPU traite simultanément plusieurs blocs de données. J'ai également modifié le calcul des blocks afin de n'en lancer qu'assez pour qu'un seul kernel n'occupe qu'une fraction, par exemple 2 blocs par SM

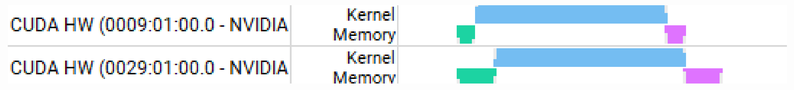


on voit bien ici que les barres vertes (H→D) et violettes (D→H) ne sont pas strictement enchaînées avant ou après tous les blocs bleus

6 Multi-GPUs

Voici maintenant la méthode qui utilise plusieurs GPU.. On segmente d'abord le tableau de données en « chunks » quasi-égaux, puis, pour chaque GPU g, on procède ainsi :

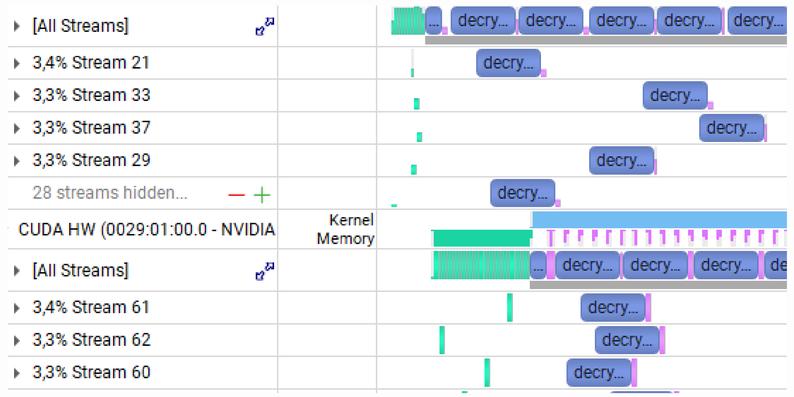
1. Choix du device : **cudaSetDevice(num_gpu);**
2. MemCpyAsync Sur chaque GPU, on fait les transfert avec data_gpu qui est maintenant un tableau de nul_gpu.
3. Lancement du kernel en spécifiant le gpu
4. Transfert inverse



On voit ici que les deux GPU exécutent en parallèle

7 Copy/Compute Overlap Mult-GPU

Pour tirer parti du multi-GPU / multi-stream et overlaper au mieux copies et calcul, on remplace tous les cudaMemcpy bloquants par des cudaMemcpyAsync sur les streams dédiés streams[g][s]. Il faut également limiter les blocs (maxB = smCount×2) pour ne pas saturer tous les SMs et permettre le co-scheduling des kernels sur un même GPU.



8 Conclusion

En combinant les CUDA streams asynchrones et le déploiement multi-GPU, on a pu superposer efficacement les transferts H->D/D->H et le calcul sur GPU. Cette approche a permis de réduire les temps d'attente CPU/GPU, de saturer le bus PCIe et de maximiser l'utilisation des SMs grâce à une limitation adaptée du nombre de blocs par SM. Au final, l'overlap copy/compute se révèle important pour tirer pleinement parti des architectures et s'adapte facilement à plus de GPUs ou à différents profils de code.

Simao CORTINHAL - URCA - 2024/2025