# Benchmark et matériel

## CPU

AMD Ryzen 7 3700x 8-core, 2 threads par cœur

3.6 GHz, max 4.4 GHz

## Caches

|  |  |
| --- | --- |
| Cache level | Taille |
| 1 | 32kB (par cœur) |
| 2 | 512kB (par cœur) |
| 3 | 16 mB (cœur 0-3 et 4-7) |

## Benchark des performances

Pour mesurer les performances de la machine utilisée, j’ai utilisé likwid dans sa version 1.18.0.

Les commandes suivantes ont été utilisées :

likwid-bench -t peakflops -W N:256KB

* Selon la topologie obtenue grâce à likwid, le groupe « N » permet d’utiliser tous les cœurs.
* La taille du banc de test est de 256KB car on a 8 cœurs avec chacun un cache L1 de 32KB => 8 \* 32KB = 256KB.

Les valeurs obtenues sont :112006.34 MFlops/s, 56003, MByte/, et 134317.660339 MUops/s.

Ainsi, le *roofline model* est (en terme de MFLops) :

Une image contenant texte, capture d’écran, ligne, diagramme

Description générée automatiquement

et en terme de MUops :

Une image contenant texte, capture d’écran, ligne, Tracé

Description générée automatiquement

## Mesures de base de l’application

Toutes les mesures se feront avec l’image half-life.png.

Pour calculer l’intensité opérationnelle, il nous faut le nombre de MFlops/s, de MUops/s ainsi que la largeur de la bande passante de notre programme. Cependant, seule la première partie du programme (le *greyscaling*) est en opérations à virgule. Pour les filtres gaussiens et de sobel, je me baserai sur le nombre de UOPS. Pour aider aux mesures, j’ai créé un groupe de performance « AI.txt » (cf annexe) permettant de calculer ces mesures ainsi que l’intensité opérationnelle (IO).

Il est également nécessaire de séparer ces trois fonctions car le volume de donnée traité n’est pas le même (greyscale traitant des données en couleur, soit 3-4x plus grand que pour gauss et sobel).

J’ai entouré les régions mentionnées avec les marqueurs de likwid (pré-enregistrés) et ai mesuré les métriques ci-dessous, qui serviront de baselines. Les temps sont mesurés en moyenne sur 5 exécutions.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Région | MUops/s | M/Flops/s | IO | IPC | Temps [ms] |
| greyscale | / | 2675.1780 | 0.181 | 3.6046 | 7.812 |
| gauss | 16518.1684 | / | 10.1171 | 4.0710 | 23.259 |
| sobel | 17378.6076 | / | 39.5213 | 4.0591 | 41.3043 |

Le temps total d’exécution est de 72.37 ms et c’est sans surprise les filtres gaussiens et de sobel qui prennent la majorité du temps. Il faudra donc se concentrer notamment sur ces deux fonctions.

# Mesures de l’efficacité de l’usage des caches

Ensuite, je mesure les caches d’instruction de données L1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Région** | **Instruction L1** | | **Data L1** | |
|  | Request rate | Miss ratio [%] | Request rate | Miss ratio [%] |
| **Greyscale** | 0.04% | 13% | 36% | 0.02% |
| **Gauss** | 0.01% | 15,03% | 28% | 0.003% |
| **Sobel** | 0.01% | 3,68% | 31% | 0.0002% |

Ce tableau, dont la forme est digne d’une œuvre d’art antique, nous montre que le taux de *cache miss* est extrêmement faible pour les données mais est supérieur pour le i-cache. Cependant, le taux de requête de cache pour le I-cache étant très faible, je peux l’ignorer.

En conclusion, le programme respecte très bien le principe de la localité des données et il ne serait pas nécessairement pertinent ou rentable d’essayer d’optimiser les structures de données. Ceci nous montre également que les performances du programme sont plus liés aux opérations qu’aux données en elles-mêmes.

Malgré tout, le taux d’accès aux données reste haut (de l’ordre de 30%). Chaque accès au cache L1 coutant environ 4 cycles sur mon architecture, il peut tout à fait être rentable de minimiser le nombre d’accès à la couche L1 et donc d’augmenter l’intensité opérationnelle du logiciel.

# Analyses et optimisations

Les exemples de code ci-dessous seront pris du filtre de Gauss mais les optimisations restent applicables pour Sobel également.

## Calculs en trop

Je commence par le simple en retirant les calculs superflus. A chaque pixel, la position de la ligne est recalculée. Je déplace ce calcul au début de chaque ligne :

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

Je passe les détails des mesures, mais cette fonction s’exécute désormais en ~22.71ms avec un IPC moyen qui passe à 4.16, soit < 5% de performance en plus pour un effort modeste. Une grande victoire pour l’écologie.

Le gain de performance étant faible, je ne vais pas afficher de nouveau roofline ici.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Région | MUops/s | M/Flops/s | IO | IPC | Temps [ms] |
| greyscale | / | 2675.1780 | 0.181 | 3.6046 | 7.812 |
| gauss | 16715.5378 | / | 9.3373 | 4.1640 | 22.710 |
| sobel | 17781.6076 | / | 38.9260 | 4.1181 | 40.598 |

## Branches

Si nous regardons le code du filtre de gauss, la copie des pixels des bordures se fait à l’aide d’un branche (l-131):

Une image contenant texte, capture d’écran

Description générée automatiquement

Après mesures, le taux d’erreur de prédiction est de l’ordre de 0.01%. Cependant, environ 6% des instructions servent aux branchements. Il parait judicieux d’essayer de les retirer.

La logique de copie étant toujours la même, je retire la branche

Annexe A – AI.txt

SHORT AI calculations for UOps et FLOPS

EVENTSET

FIXC1 ACTUAL\_CPU\_CLOCK

FIXC2 MAX\_CPU\_CLOCK

PMC0 RETIRED\_INSTRUCTIONS

PMC1 RETIRED\_UOPS

PMC2 RETIRED\_SSE\_AVX\_FLOPS\_ALL

PMC3 MERGE

DFC0 DATA\_FROM\_LOCAL\_DRAM\_CHANNEL

DFC1 DATA\_TO\_LOCAL\_DRAM\_CHANNEL

METRICS

Runtime (RDTSC) [ms] 1000\*time

Runtime unhalted [s] FIXC1\*inverseClock

MFLOP/s 1.0E-06\*(PMC2)/time

Retired MOps/s 1.0E-06\*(PMC1)/time

Memory bandwidth [MBytes/s] 4.0E-06\*(DFC0+DFC1)\*(4.0/num\_numadomains)\*64.0/runtime

AI (Uops) [OP/Byte] (PMC1)/((DFC0+DFC1)\*(4.0/(num\_numadomains))\*64.0)

AI (Flops) [Flop/byte] (PMC2)/((DFC0+DFC1)\*(4.0/(num\_numadomains))\*64.0)

LONG

Truc