HPC - Laboratoire 2

Profiling

Rayane Annen

24 mars 2024

Benchmarks

Machine utilisée pour les tests :

- Architecture : Intel x86_64
- CPU : i9-9900K 8 Cores / 16 Threads @ 3.60GHz / Turbo 5.00GHz
- OS: Debian 12
- Compilateur :
  - gcc 12.2.0
  - target: x86_64-linux-gnu
  - Flags de compilation: `-O3 -g -Wall -fno-inline -DLIKWID_PERFMON`
  - Librairies: `stb`, `math.h` et `Likwid`

Analyse du code

Dans cette première partie, il est question de trouver emplacements dans le code qui prennent le plus de temps à calculer. J'ai identifié plusieurs endroits dans le code demandant des performances :

- Filtre niveau de gris (listes chaînées et tableaux 1D): `grayscale_1d` et `grayscale_chained`
- Filtre gaussien (listes chaînées et tableaux 1D): `gaussian_1d` et `gaussian_chained`
- Filtre sobel (listes chaînées et tableaux 1D): `sobel_1d` et `sobel_chained`

En effectuant un benchmark avec likwid voici les résultats obtenus pour les listes chaînées:

- Temps d'exécution total : 0.325850 secondes
- Commande exécutée :

```
sudo likwid-perfctr -C E:N:8 -g MEM_DP -m ./lab01 ../images/half-life.png ../images/out.png 2
```

	Temps d'exécution [s]	MFLOPs/s	Operational Intensity
`grayscale_chained`	0.086963 (26%)	144.1993	0.0152
`gaussian_chained`	0.141852 (43%)	0.0001	1.336258e-08
`sobel_chained`	0.095893 (29%)	43.5051	0.0035

Voici le résultat pour les tableaux 1D

- Temps d'exécution total : 0.117468
- Commande exécutée :

```
sudo likwid-perfctr -C E:N:8 -g MEM_DP -m ./lab01 ../images/half-life.png ../images/out.png 1
```

	Temps d'exécution [s]	MFLOP/s	Operational Intensity
`grayscale_1d`	0.030708 (26%)	144.1993	0.5390
`gaussian_1d`	0.038637 (32%)	0.0001	1.172069e-06

	Temps d'exécution [s]	MFLOP/s	Operational Intensity
`sobel_1d`	0.047003 (40%)	88.7568	1.544869e-06

Amélioration des performances

Baseline

Voici ci-dessous un tableau récapitulatif des tests effectués ainsi qu'un graphe des temps de traitements mesurés pour chacun des tests.

Images d'entrées :

Nom du fichier	Dimensions [pixels]	Nombre de composantes par pixel
`half-life.png`	$2000 \times 2090 = 4038000$	3 (8-bit RGB)
`medalion.png`	$1267 \times 919 = 1164373$	3 (8-bit RGB)
`nyc.png`	$1150 \times 710 = 816500$	3 (8-bit RGB)

Nombre de type :

- Tableau 1D : `1`
- Liste chaînée : `2`

Sur la base de ces deux variables (l'image et le type de structures de données) une matrice de tests a été effectuée. Chaque test est effectué 50 fois et le résultat gardé est la moyenne de toutes les runs.

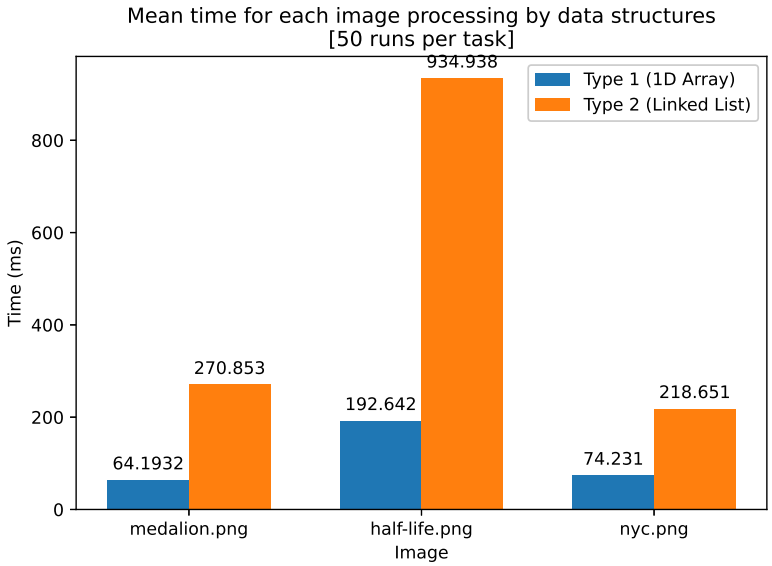

Figure 1: Résultats du benchmark obtenu avec la machine de test (baseline).

Accélération du calcul de la norme du gradient

Lorsqu'on applique le filtre sobel sur l'image on effectue en chaque point de l'image une convolution permettant d'approximer les gradients horizontaux ($\boldsymbol{G_x}$) et verticaux ($\boldsymbol{G_y}$).

Une fois les gradients calculés, on peut obtenir une approximation de la norme euclidienne du gradient :

$$\boldsymbol{G} = \sqrt{\boldsymbol{G_x^2} + \boldsymbol{G_y^2}}$$

Ce dernier calcul est ensuite utilisé pour déterminer si la norme est plus grande qu'un certain seuil, selon les cas on affiche un pixel blanc ou noir.

Le calcul de la norme peut être simplifié en comparant simplement les distances au carré :

$$\text{Couleur} = \begin{cases} \text{blanc}, & \text{si } \boldsymbol{G}^2 > \text{seuil} \times \text{seuil}, \\ \text{noir}, & \text{sinon} \end{cases}$$

Le code modifié résultant est le suivant :

```
    magnitude = gx * gx + gy * gy;
    res_img->data[i] = magnitude > SOBEL_BINARY_THRESHOLD ? PIXEL_WHITE : PIXEL_BLACK
```

Finalement voici les résultat obtenus

- Temps d'exécution total (liste chaînées) : 0.323816 [s] (amélioration par rapport au baseline: -0.62%)
- Temps d'exécution total (tableaux 1D) : 0.112129 [s] (-1.64%)

	Temps d'exécution [s]	MFLOPs/s	Operational Intensity
`sobel_chained`	0.093865 (-2.11%)	44.4452	0.0035
`sobel_1d`	0.043430 (-7.6%)	0.0004	1.544869e-06

Globalement on remarque que la différence reste minime, toutefois localement pour le filtre sobel, particulièrement avec des tableaux unidimensionnels la différence se voit, en effet on constate une amélioration de 7.6%.

Optimisation des boucles `for`

- Optimisation de la boucle
  - Pas de passage sur les pixels des bords inférieurs et supérieurs lors de l'application des filtres Code modifiée de la manière suivante :

```
for (i = width; i < img_size - width; ++i) {
  // ...
}
```

  - Simplification de la vérification si un pixel est en bordure (on ne vérifie plus que les bords latéraux) Code modifiée de la manière suivante :

```
if (position % width == 0 || position % width == width - 1) {
  // on border
}
```

- Précalcule des offsets pour les convolutions:

```
// ...
const int32_t offsets[] = {-width - 1, -width, -width + 1, -1, 0, 1, width - 1, width, width + 1}
// ...
sum = 0;
for (j = 0; j < 9; ++j) {
    sum += img->data[i + offsets[j]] * kernel[j];
}
res_img->data[i] = sum / gauss_ponderation;
//...
```

Retrait des autres invariants de boucles des boucles `for`:

- Taille et dimensions de l'image

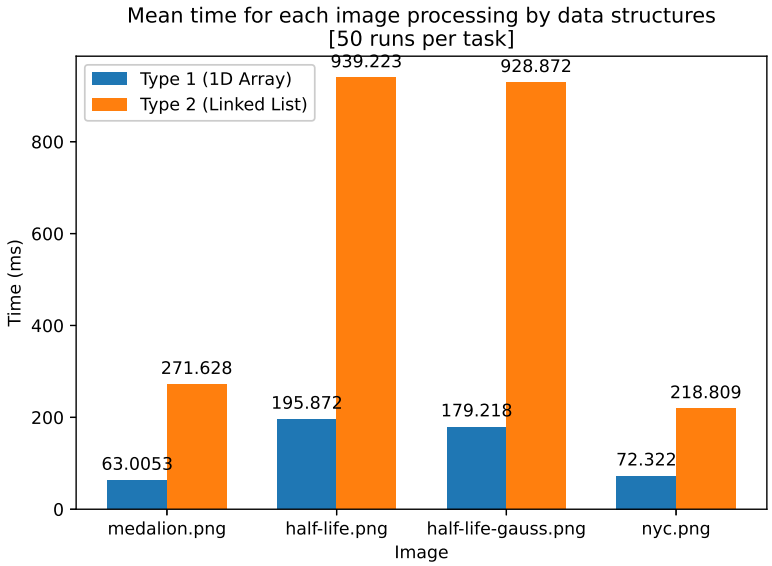

Figure 2: Résultats du benchmark obtenu avec la machine de test (après accélération sobel).