DM n° 1 — Compression d'image en niveaux de gris

Antoine de Roquemaurel (Groupe 1.1)

1 Compilation exécution et tests

L'archive qui à été déposée sur Moodle est organisé comme suit :

`Report_deRoquemaurelAntoine_G1.1.pdf` Le rapport que vous êtes en train de consulter

`Compressor/` Programme de compression contenant les fichiers détaillés section 1.2.

1.1 Fichier sources

- `block.c` : Fonctions et structures de données concernant les blocks.
- `compressor.c` : Fonctions de compression.
- `decompressor.c` : Fonctions de décompression.
- `main.c` : Fichier principal gérant les arguments
- `ZIterator.c` : Itérateur effectuant un parcours en Z.
- `blockiterator.c` : Itérateur permettant de parcourir une image block par block.
- `dct-idct.c` : Fonctions appliquant la dct et son inverse
- `image.c` : Fonctions et structure de données concernant les images
- `util.c` : Fonctions utiles
- `Makefile` : Fichier Makefile

1.2 Fichiers de tests

L'exécution des tests s'effectue de la même manière que l'archive fournie :

```
aroquemaurel@aokiji < master > : ~/projets/c/oim/jpg-compression/report
[0] % time make tests
dct             [OK]
quantify        [OK]
vectorize       [OK]
compression     [OK]
decompression [OK]
make tests  0,38s user 0,07s system 85% cpu 0,532 total
```

Listing 1 – Execution des tests

Comme montré listing 1, l'intégralité des tests passent, ceux-ci sont basés sur les images d'origine. Un image de taille plus importante à été ajoutée afin de pouvoir mieux observer les différentes optimisation, plus d'informations sont fournies section 4.

2 Compression

Afin d'effectuer la compression, plusieurs tests ont été effectués afin de garantir que chaque module était bien fonctionnel :

– Dct
– Quantify
– Vectorize
– Compression

Chaque test est incrémental : un test ne peut pas passer sans passer le précédent.

2.1 Dct et Quantify

Afin d'appliquer la dct et de quantifier la matrice, il faut utiliser la fonction `applyDct` qui en fonction du 3e paramètre applique soit la normalisation, c'est-à-dire une division par `8.f`, soit il divise par la matrice de quantification. Cette factorisation nous permet de réutiliser cette méthode soit pour le test dct, soit pour le test quantify.

2.2 Vectorize

La vectorisation consiste en l'utilisation d'un Itérateur permettant de parcourir un block en Z comme le montre l'image ci-contre.

Cet itérateur, `ZIterateur`, possède les méthodes `has_next` et `next` permettant de parcourir l'intégralité du block comme le montre le listing 2. La méthode `next` doit vérifier tous les cas possibles avant de renvoyer la bonne valeur et de déplacer le curseur dans le tableau.

```
for(i = 0 ; i < input->h ; i +=8 ) {
  for(j = 0 ; j < input->w ; j += 8) {
    block = block_new();
    dct(input, block.data, j, i);
    block_applyQuantify(&block, quantify);
    zit = zIterator_new(block, 8);

    output->data[(output->size)++] = round(zIterator_value(zit));
    while(zIterator_hasNext(zit)) {
      output->data[(output->size)++] = round(zIterator_next(&zit));
    }

    zIterator_delete(&zit);
    block_delete(&block);
  }
}
```

Listing 2 – Vectoriser une image

Une solution plus optimisée que la présente aurait pu être de générer la liste des positions nécessaires à un déplacement en Z avec un script externe tel que Python ou Bash. Cependant cette solution était moins souple en terme d'évolution : l'execution

Une autre solution peut cependant être envisager pour optimiser un peu plus l'algorithme : stocker les valeurs des cases devant être parcourues dans un tableau.

3 Décompression

La décompression utilise globalement les modules déjà développés précédemment, cependant en raison de l'`idct` particulièrement lente, une étape d'optimisation section 4 sera nécessaire.

Figure 1, nous pouvons voir l'image `mystery.xxx` une fois décompressée grâce au développement du décompresseur.

La décompression fonctionne correctement sur les systèmes 64bits, il semblerait cependant que le test de décompression renvoit `KO` sur un système 32bits.

Cela vient d'une approximation des arrondies, bien que l'image reste tout à fait lisible. On peut supposer que la compression / décompression sous un système 32 bits à un taux d'erreur plus important.

Il est conseillé de tester le compresseur sur un système 64bits.

Figure 1 – Image mystère

4 Optimisations

Le traitement d'image, et particulièrement la décompression sont des opérations couteuses. Ainsi afin d'avoir un traitement qui soit correct en mémoire ou en cpu, des optimisations étaient nécessaires.

Deux axes d'optimisations ont été travaillés :
- D'une part, limiter les calculs, notamment dans les boucles, et particulièrement pour la fonction idct.
- D'autres part, l'utilisation de plusieurs cœurs en parallélisant le système semblait être intéressant.

Afin de pouvoir observer facilement le temps d'exécution gagné, une image à été ajouté aux scripts de tests. En effet, les images proposées ne dépassaient pas 800×800 pixels, un traitement qui était déjà relativement rapide. J'ai donc téléchargé une image `nasa.pgm` visible figure 2 possédant une taille de 1800×1800, cela m'a permis d'effectuer un test de charge tout en pouvant plus facilement observer les différences d'optimisations.

R Cependant, ces optimisations ne doivent pas altérer la lisibilité du code afin de pouvoir maintenir notre compresseur sans aucun problème.

Section 4.3, figure 3 est disponible une figure montrant les gains de performance obtenus suite aux optimisations.

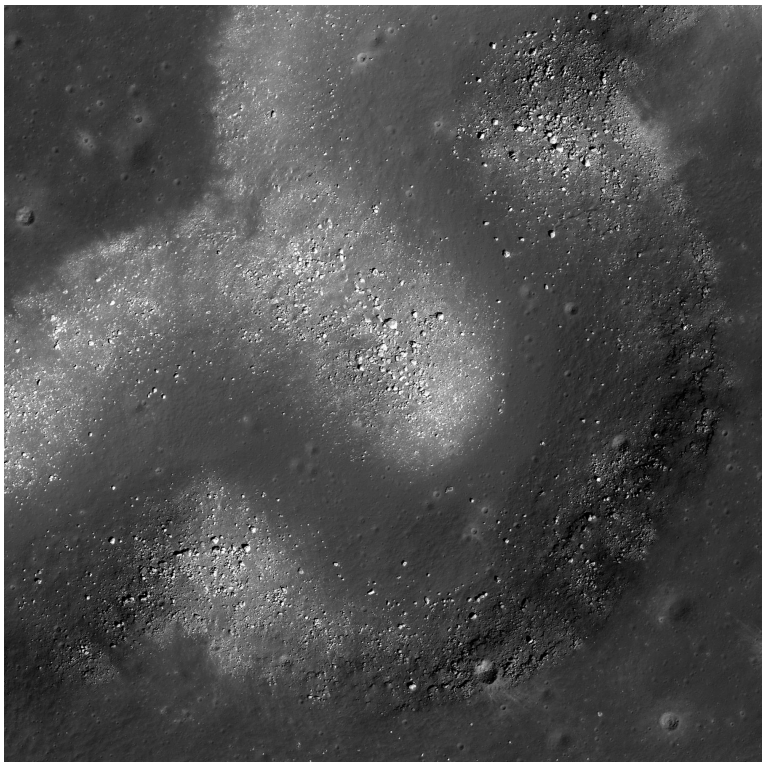

Figure 2 – Image utilisée pour les tests d'optimisation

4.1 Inverse de la dct

4.1.1 Constantes calculées dans les boucles

Dans la fonction `idct`, plusieurs calculs de constantes étaient effectués à l'intérieur des boucles, dans la 4e boucle, ces calculs étaient donc effectués $8^4 = 4096$ fois, par image. Le compilateur doit effectuer quelques optimisations afin d'éviter de recalculer trop souvent, cependant il était tout de même inutile d'effectuer autant de fois ces calculs, d'autant plus que ce sont des divisions, l'opération la plus couteuse pour un processeur.

Afin d'éviter la redondance de calculs, j'ai donc mis la valeur du calcul dans des variables statiques : celui-ci sera effectué au premier lancement de `idct` et sera ensuite conservé jusqu'à la fin du programme.

Les deux opérations concernées étaient :
- $\frac{\Pi}{16}$, utilisé dans les calculs de cosinus

```
static double invSqrt2 = 1 / sqrt(2);
```

Listing 3 – Déclaration d'une variable contenant $\frac{1}{\sqrt{2}}$

- $\frac{1}{\sqrt{2}}$, utilisé dans la macro `COEFFS`

```
static double piDivideBy16 = M_PI / 8;
```

Listing 4 – Déclaration d'une variable contenant $\frac{\Pi}{16}$

4.1.2 Cosinus

Le principal problème de `idct` était le calcul des cosinus. En effet, un cosinus est un calcul lourd à effectuer, or celui-ci à l'instar des calculs de $\frac{\Pi}{8}$ ou de $\frac{1}{\sqrt{2}}$, était placé à l'intérieur des boucles. Ainsi un très grand nombre de calcul étaient effectués.

Sauf qu'en regardant plus en détail, ces cosinus étaient effectués en fonction des variables x, u, y et v : variables ne dépendant nullement des paramètres de la fonction. Le processeur effectuait donc toujours les même calculs à chaque appel de `idct`.

La solution que j'ai trouvé est qu'au premier lancement de `idct`, je stock les cosinus qui seront nécessaires quel que soit l'appel de `idct`, dans le cas ou on aurait plusieurs calculs identiques pour un seul appel, j'ai appliqué un système de `Hash`.

Les résultats des cosinus vont être stockés dans un tableau de `float`, l'indice du tableau sera le paramètre fourni à `cos`, c'est-à-dire une des deux valeurs suivantes :
– `(2*y+1)*v`
– `(2*x+1)*u`
Le résultat d'un cosinus est donc obtenu de la manière suivante :

```
allCos[(2*y+1)*v];
allCos[(2*x+1)*u];
```

Listing 5 – Obtention du résultat d'un cosinus

Une dernière modification en rapport avec le cosinus à été de remonter la récupération de `allCos[(2*y+1)*v]` d'une boucle : en effet, cette instruction ne dépend que de y et v, il est inutile de la mettre dans la boucle u.

> R Cette modification n'a cependant pas fait gagné de temps de calculs, étant donné que ce n'était qu'une lecture RAM très rapide à faire.

4.1.3 Déclarations dans les boucles

La dernière modification effectuée pour `idct` était mineure : déplacer les déclarations en dehors de la boucle. En effet, il me paraissait inutile d'effectuer une déclaration au début de la boucle, et de détruire la variable à la fin, et ceci 4096 fois. Cependant, en me renseignant, j'ai découvert que le compilateur `gcc` optimisait cela en se rendant compte qu'il était inutile de libérer la variable : aucun temps n'a donc été gagné grâce à cette étape car `gcc` faisait déjà le travail.

Cependant, il est tout de même plus propre de ne pas se reposer sur le compilateur et d'effectuer nous même les optimisations, en effet, se reposer sur le compilateur peut être dangereux en fonction des versions du compilateur. Quant est-t-il de Mingw, Cygwin, Visual C++, ... ?

4.2 Parallélisme avec `openmp`

La parallélisation permet d'utiliser plusieurs cœurs simultanément, ce qui pourrait hypothétiquement accélérer le calcul sur les matrices. En effet certaines boucles pourraient être faites par plusieurs threads.

Cependant, en pratique, cette optimisation est inutile pour notre problème. En effet, le temps qui pourrait être gagné par le traitement paralléliser est perdu à cause de la communication entre threads et de leur gestion : sur un petit volume de données, cela est donc contre-productif.

Or, notre projet comporte des calculs qui s'effectuent principalement sur des blocs de 8×8 : ce sont donc des petites itérations, bien que nombreuses.
L'utilisation de OpenMP serait donc plus intéressantes pour des boucles parcourant de gros volumes de données.

Les boucles les plus intéressantes à paralléliser sont celles qui peuvent s'effectuer en parallèle indépendamment de l'itération courante, ce sont donc les boucles concernant la vectorisation du côtés de la compression et la « dévectorisation » du côtés de la compression.

4.3 Gain de temps grâce à l'optimisation

La figure 3 montre les différents gains de performances effectués tout au long de l'optimisation.

Aucune Aucune optimisation

Nombre de boucles Optimisation de mon code afin de diminuer au maximum le nombre de boucles

`idct` Optimisation de la fonction `idct` comme détaillé section 4.1

OpenMP Utilisation de la bibliothèque OpenMP afin de paralléliser le travail

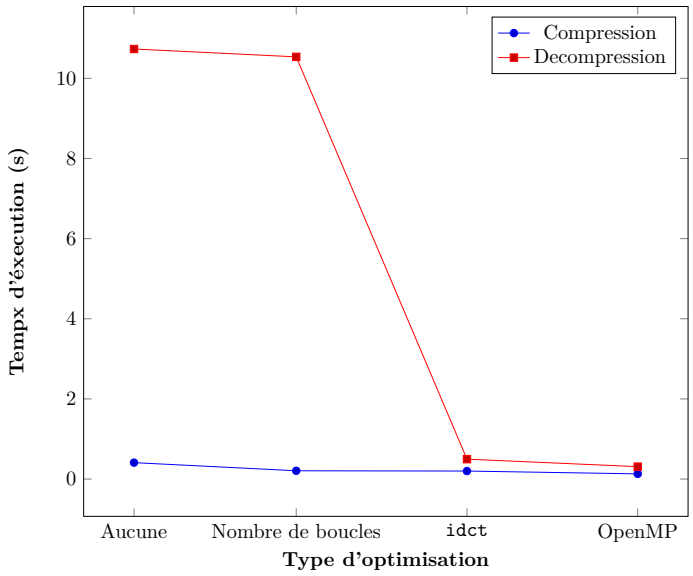

Figure 3 – Statistiques sur le temps gagné grâce aux optimisations

Nous pouvons donc observer qu'au vu de sa rapidité, la compression n'était pas optimisable : il parait extrêmement difficile de gagner ne serait-ce que 30ms, notamment grâce à la fonction dct qui effectue la majorité de ses calculs directement en binaire.

Cependant, grâce à notre optimisation, nous avons pu avoir une décompression $\frac{10.510}{0.502} = 21$ fois plus rapide !

```
(ssh) aroquemaurel@aokiji <noOptimize>
[0] % time ./compressor 0 nasa.xxx nasa.pgm

./compressor 0 nasa.xxx nasa.pgm  10,47s user ↩
    0,02s system 99% cpu 10,510 total
```

Listing 6 – Sans optimisation

```
(ssh) aroquemaurel@aokiji <master>
[0] % time ./compressor 0 nasa.xxx nasa.pgm

./compressor 0 nasa.xxx  nasa.pgm 0,48s user ↩
    0,01s system 99% cpu 0,502 total
```

Listing 7 – Avec optimisations

A Table des figures

B Listings