

Rapport Master M1

Master Calcul Haute Performance Simulation (CHPS)

Optimisation du Code CNN pour la Reconnaissance des Plantes

Réalisé par: Aicha Maaoui

Encadré par: Prof. Hugo Bolloré

Mai 2022

Institut des Sciences et Techniques des Yvelines (ISTY)

Abstract

Contents

| 1 | Opt | imisation du code en séquentiel | 1 |
|--------------|-------|--|----|
| | 1.1 | Profilage du code | 1 |
| | 1.2 | Flags de compilation: Options d'optimisation | 1 |
| 2 | Par | allélisation du code avec MPI | 3 |
| | 2.1 | Avant propos | 3 |
| | 2.2 | Strucutre du code original | 3 |
| | 2.3 | 1ere tentative de parallélisation: Approche 1 | 4 |
| | | 2.3.1 Profilage du code avec <i>Gprof</i> et <i>Perf</i> | 4 |
| | | 2.3.2 Approche 1: Parallélisation du calcul de convolution | 5 |
| | | 2.3.3 Approche 1 bis: Tentative d'amélioration de l'approche 1 | 7 |
| | 2.4 | Approche 2: Décomposition de l'image | 9 |
| | 2.5 | Approche 3: Décomposition de l'ensemble des images | 10 |
| \mathbf{A} | Арр | pendix Chapter | 11 |
| Re | efere | nces | 11 |

List of Figures

| 1.1 | Speed-up obtenu en fonction de flags de compilation | 2 |
|-----|--|----|
| 2.1 | Structure du code original. | 3 |
| 2.2 | Résultat de profilage du code original avec <i>Gprof.</i> | 4 |
| 2.3 | Résultat de profilage du code original avec <i>Perf.</i> | 4 |
| 2.4 | Illustration de la procédure de convolution | 5 |
| 2.5 | Utilisation de 3 processeurs pour le calcul de la convolution | 6 |
| 2.6 | Speed-up obtenu en fonction du nombre de processus et taille des images d'entrée | 7 |
| 2.7 | Comparison du temps d'exécution avec les approches 1 et 1 bis | 8 |
| 2.8 | Précision obtenue en fonction du nombre de processus, pour les approches 1 et | |
| | 1 bis, tailles d'images 50×50 et $200\times 200.$ \ldots | 9 |
| 2.9 | Partie a paralléliser dans l'approche 2 | 10 |

List of Tables

| 1.1 | Ajout de flags d | de compilation p | our optimiser le tem | ps de calcul. | 1 |
|-----|------------------|------------------|----------------------|---------------|-------|
| | | | | | |

Chapter 1

Optimisation du code en séquentiel

1.1 Profilage du code

Pour faire un profilage du code, on utilise Perf. Pour se faire, on ajoute le flage -g dans le Makefile.

1.2 Flags de compilation: Options d'optimisation

On se propose dans cette partie de diminuer le temps d'exécution en ajoutant des flags de compilation pour le compilateur g++.

Pour se faire, on fixe le nombre d'Epochs et le taux d'apprentissage à 1 et 0.003, respectivement. La taille de chaque image du Data est égal à 50×50 (en cm^2). Le calcul étant séquentiel, l'exécution du code est faite sur un seul processus.

Le tableau (1.1) regroupe les flags de compilation ajoutés au fichier Makefile, dans CFLAGS.

| Flag | Description | Temps Exec. |
|---------------------|---|-----------------------|
| Default (gcc -Wall) | - | $82.12 \sec$ |
| -O2 | Recommandé par Intel pour utilisation générale | $11.76 \sec$ |
| -O3 | Optimisation (calcul intensif en virgule flottante) | $10.95 \mathrm{sec}$ |
| -Ofast | -O3 plus quelques extras | $10.94 \sec$ |
| -ffast-math | optimisation agressives en virgule flottante | $13.08 \sec$ |
| -ftree-vectorize | vectorisation automatique | $11.65 \sec$ |

Table 1.1: Ajout de flags de compilation pour optimiser le temps de calcul.

Le speed-up suite à l'ajout des flags de compilation est illustré dans la figure (1.1).

Ainsi, on en deduit que:

- Le flag -Ofast est équivalent au flag -O3 avec quelques extras. Cependant, le speedup obtenu en utilisant chaque flag d'eux est presque comparable pour notre code. On conservera dans ce cas le flag de compilation -O3,
- Le flag -ffast-math peut être activé en utilisant le flag -Ofast,
- L'auto vectorisation est obtenue en utilisant le flag de compilation *-ftree-vectorize*, mais elle est incluse dans le flag de compilation *-O3*.

Speed-up obtenu en fonction de flags de compilation

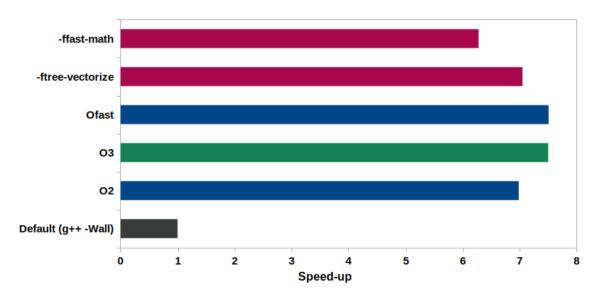


Figure 1.1: Speed-up obtenu en fonction de flags de compilation.

On conclut qu'on peut utiliser le flag de compilation -O3 pour avoir un speed-up considérable du code d'environ 7.5%.

Chapter 2

Parallélisation du code avec MPI

2.1 Avant propos

Dans ce chapitre, l'analyse de performance est effectué sur 3 nombre d'epochs pour des images de tailles 50×50 et 200×200 et un taux d'apprentissage égal à 0.003.

2.2 Strucutre du code original

Le code de la partie training est décrit par la figure (2.1).

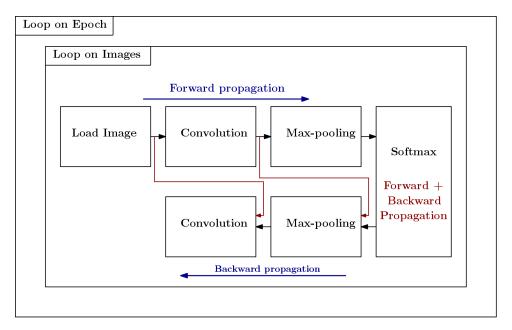


Figure 2.1: Structure du code original.

On a d'abord une première boucle sur le nombre d'epochs. A chaque epoch, on a une deuxième

boucle sur l'ensemble des images disponibles. Pour chaque image, on commence par la charger et réduire le nombre de canaux de 3 à 1. L'image réduite passe ensuite à la partie convolution. L'image convoluée passe ensuite à la partie max-pooling pour finir dans la couche softmax. Ces étapes sont effectuées ensuite dans le sens inverse dans la back-propagation. Il est important ici de noter que les parties max-pooling et convolution dans la back-propagation prenant en entrée non seulement les données issues de la couche softmax, mais aussi des entrées initiales comme l'indiquent les flèches rouges.

2.3 1ere tentative de parallélisation: Approche 1

2.3.1 Profilage du code avec Gprof et Perf

D'après les résultats de Gprof (illustrées dans la figure (2.2)), la partie convolution consomme environ 16% du temps d'exécution.

```
Flat profile:
                         Each sample counts as 0.01 seconds.
                                                    cumulative
                                                                                                                  self
                                                                                                                                                                                                          self
                                                                                                                                                                                                                                                      total
                                                                                                                                                                                                         s/call
0.00
                                                                                                                                                                                                                                                    s/call
0.00
                                                                                                                                                                                                                                                                                         name
void std::vector<double, std::allocator<double> >::_M_realloc_insert<double const&>(__gnu_cxx::
                              15.88
                                                                                 3.04
                                                                                                                             0.91
                                                                                                                                                                        6038
                                                                                                                                                                                                                   0.00
                                                                                                                                                                                                                                                             0.00 Convolution_layer::convolution_parameters(std::vector<double, std::allocator<double> >
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         const&, int, i
                                                                                                                                                                                                                                                                                        Convolution_layer::BockPropagation(std::vector<double, std::allocator<double> > consta, lnt, 1
Softmax_layer::BockPropagation(std::vector<double, std::allocator<double> > consta, double)
Softmax_layer::Softmax_start(std::vector<std::vector<double, std::allocator<double> >, std::allocator
Pooling_layer::BackPropagation(std::vector<std::vector<double, std::allocator<double> >, std:
                              13 79
                                                                                  3 83
                                                                                                                                                                         3019
                                                                                                                                                                                                                     0 00
                                                                                                                                                                        6038
3019
                                                                                                                                                                                                                   0.00
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
                                  5.93
                                                                                  5.21
                                                                                                                             0.34
                                                                                                                                                                        6038
                                                                                                                                                                                                                   0.00
                                                                                                                                                                                                                                                               0.00
                                                                                  5.40
                                                                                                                             0.19
                                                                                                                                                                         3019
                                                                                                                                                                                                                   0.00
                                                                                                                                                                                                                                                               0.00
                                                                                                                                                                                                                     0.00
                                                                                  5.66
5.73
                                                                                                                             0.10
                                                                                                                                                                        6038
                                                                                                                                                                                                                                                               0.00
                                                                                                                                                                                                                                                                                         Data::create_canal(cv::Mat*)
                                  1.22
                                                                                                                             0.07
                                                                                                                                                                                                                                                                                               init
                                                                                                                                                                                                                                                                                        _init

void std::vector<double, std::allocator<double> >:: M_realloc_insert<double>(_gnu_cxx::_normal_iter
Data::loadImage(std::_cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const
void std::vector<int, std::allocator<chirt> >:: M_realloc_insert<int const&<(_gnu_cxx::_normal_iterat
std::vector<double, std::allocator<double> >::_M_default_append(unsigned long)
Pooling_layer::Hidden(std::vector<double, std::allocator<double> >, std
                                                                                  5.73
5.73
                                                                                                                                                                                                                  0.00
0.00
0.00
                                                                                                                                                               313976
                                                                                                                                                                                                                                                               0.00
                                                                                                                                                                   54342
45311
                                  0.00
                                                                                                                             0.00
                                  0.00
                                                                                  5.73
                                                                                                                             0.00
                                                                                                                                                                    18114
                                                                                                                                                                                                                   0.00
                                  0.00
                                                                                    5.73
                                                                                                                                                                                                                     0.00
                                                                                                                                                                                                                                                                                           Convolution layer::Hidden(std::vector<double, std::allocator<double> > const&)
                                  0.00
                                                                                  5.73
                                                                                                                             0.00
                                                                                                                                                                        6038
                                                                                                                                                                                                                     0.00
                                                                                                                                                                                                                                                                                         output::prediction(int, int&, int&)
Data::get_fusion_canal() const
std::vector<std::vector<double, std::allocator<double> >, std::allocator<std::vector<double, std::all
                                  0.00
                                                                                  5.73
                                                                                                                             0.00
                                                                                                                                                                        6038
                                                                                                                                                                                                                     0.00
                                                                                  5.73
5.73
                                                                                                                                                                         6038
                                 0.00
                                                                                  5.73
                                                                                                                                                                                                                                                                                           void std::vector<std:: cxx11::basic string<char, std::char traits<char>, std::allocator<char> >, std
```

Figure 2.2: Résultat de profilage du code original avec *Gprof*.

Cependant, *Gprof* donne comme résultat le temps passé des fonctions en mode exclusif, différemment à *Perf* qui est en mode inclusif. Le profilage du code avec *Perf* est montré dans la figure (2.3).

```
program
11.66%
                              libc-2.31.sc
                                                                                           malloc
             program
                              program
                                                                                           Softmax_layer::BackPropagation
_int_malloc
_memmove_avx_unaligned_erms
cfree@GLIBC_2.2.5
Pooling_layer::BackPropagation
Pooling_layer::Pooling_parameter
 6,41%
                              program
libc-2.31.so
libc-2.31.so
             program
 5,67%
4,94%
             program
program
  4,67%
             program
                              libc-2.31.so
 4,23%
3,42%
                              program
program
             program
 2,80%
             program
                              ргодгам
                                                                                           Convolution_layer::BackPropagation
 2,41%
2,06%
             program
program
                              libstdc++.so.6.0.28
libc-2.31.so
                                                                                           operator new
malloc_consolidate
                                                                                          Softmax_layer::Softmax_start
unlink_chunk.isra.0
Data::create_canal
             program
program
  1.49%
                              ргодгам
                              libc-2.31.so
             program
                              program
```

Figure 2.3: Résultat de profilage du code original avec *Perf.*

D'après la figure (2.3), on remarque que la couche de convolution consomme 8.14% du temps d'exécution.

2.3.2 Approche 1: Parallélisation du calcul de convolution

Suite aux résultats obtenus lors du profilage du code, on a essayé de commencer par améliorer le calcul dans cette couche en le parallélisant. Cette première approche est implémentée dans la branche $Aicha_Optimization_MPI_Approach1$.

Initialement, cette partie applique sur une image de taille initiale 8 filtres de taille 3×3 chacun. A chaque étape, le filtre est superposé à l'image. On multiplie les nombres dans la matrice représentant l'image d'entrée et ceux du filtre, on les somme et on place le résultat dans la matrice convoluée. Ensuite, le filtre se déplace par le *stride* (ici égal à 1) et on refait cette procédure, jusqu'au remplissage de la matrice convoluée de sortie, comme illustré dans l'image (2.4).

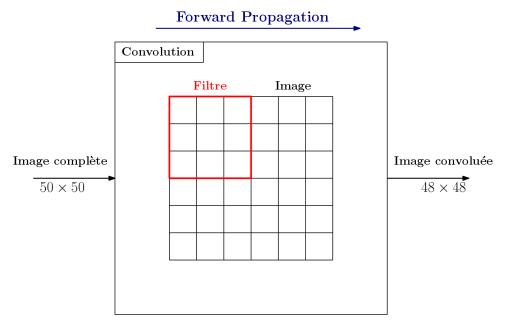


Figure 2.4: Illustration de la procédure de convolution.

Afin de paralléliser le calcul, on a lancé le code sur plusieurs processeurs. Donc, chaque processeur exécute la même chose que les autres et charge l'image entière. Le filtre cependant a été calculé sur un seul processeur (rank 0), ensuite partagé avec les autres (en utilisant MPI_Bcast). Avant de commencer le calcul de convolution, chaque processeur lui a été attribué une partie spécifique de l'image sur laquelle il va appliquer le filtre. La figure (2.5) illustre un exemple d'utilisation de 3 processeurs.

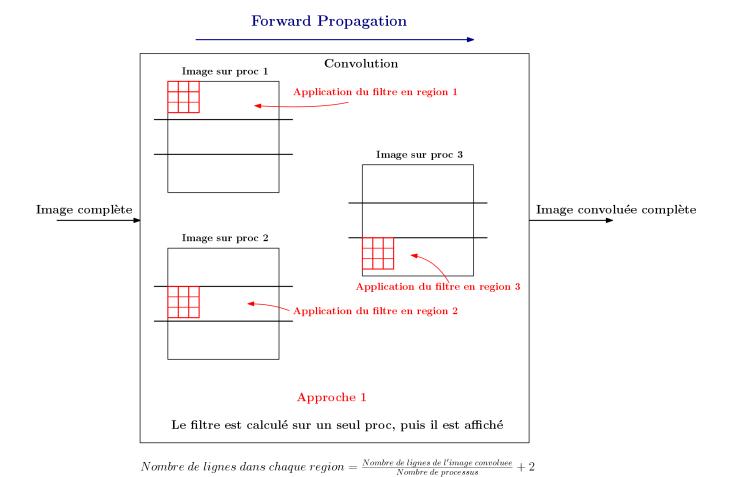


Figure 2.5: Utilisation de 3 processeurs pour le calcul de la convolution.

Le découpage se fait toujours d'une manière horizontale. Ce découpage est adapté à notre structure de données car toute l'image est stockée horizontalement dans un seul vecteur à 1 dimension.

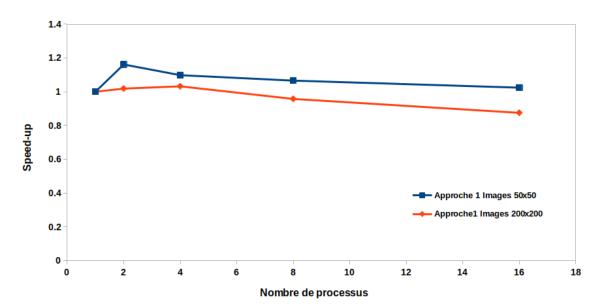
Le nombre de lignes attribuées à chaque processus est calculé par la formule suivante:

Nombre de lignes dans chaque région =
$$\frac{\text{Nombre de lignes de l'image convoluée}}{\text{Nombre de processeurs}} + 2$$
 (2.1)

Il faut toujours laisser une ligne de plus en haut et une ligne de plus en bas pour l'application du filtre à l'image. Donc, on peut penser que cette méthode est plus adaptée aux grandes images qu'aux petites images. Par exemple, si on décompose une image 50×50 sur 16 procs, chaque processeur va prendre en charge 3 lignes sur lesquelles il va appliquer le filtre, plus 2 autres lignes. Ainsi, les résultats des différents processeurs sont rassemblées pour former l'image convoluée totale. On utilise pour ça la commande $MPI_Gatherv$ qui va rassembler toutes les données dans le processus rank 0. Le calcul continue ensuite de manière normale sur le rank 0. On utilise

MPI_Gatherv et non pas MPI_Gather car cette commande rend possible un découpage qui n'est pas semblable pour tous les processeurs.

La figure (2.6) illustre le speed-up obtenu en utilisant cette première approche sur des images de tailles 50×50 et 200×200 .



Approche 1: Speed-up obtenu en fonction de la taille d'image et le nombre de processus

Figure 2.6: Speed-up obtenu en fonction du nombre de processus et taille des images d'entrée

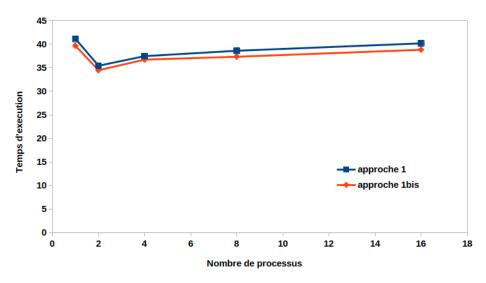
On remarque que le speed-up obtenu dans le cas d'une image de taille 50×50 est plus grand que celle de taille 200×200 , ce qui est contraire à notre attentes. Parmi les hypothèses qui peuvent expliquer ce résultat inattendu, on peut citer:

- le coût d'échange de données volumineux masque le coût gagné pendant le calcul,
- Il aussi probable que l'utilisation d'images plus grandes est accompagné par un surcoût dans d'autres parties du code qui masque le gain dans la couche de convolution

2.3.3 Approche 1bis: Tentative d'amélioration de l'approche 1

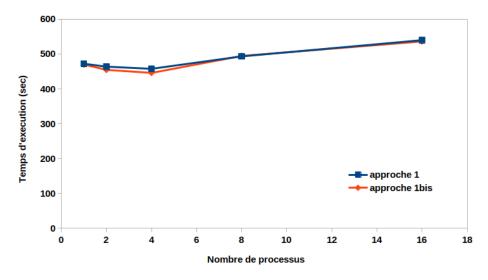
On a encore essayé d'optimiser cette approche. En réalité le calcul de filtre se fait 8 fois avec 8 filtres différents et donc le rassemblement de données avec $MPI_Gatherv$ se fait de même 8 fois. Afin de réduire les communications MPI, on a essayé d'envoyer les résultats des 8 filtres en 1 seul bloc, comme le montre le listing (2.1). La figure (2.7) montre le résultat du speed-up et on ne constate pas une grande amélioration au niveau temps d'exécution.

Temps d'execution en fonction du nombre de processus, images 50x50



(a) Temps d'exécution en fonction du nombre de processus, 3 epochs, images 50×50 .

Temps d'execution en fonction du nombre de processus, images 200x200



(b) Temps d'exécution en fonction du nombre de processus, 3 epochs, images 200×200 .

Figure 2.7: Comparison du temps d'exécution avec les approches 1 et 1bis.

On peut constater d'après la figure (2.7) une légère amélioration du temps d'exécution. On peut aussi vérifier que tout au long de ces implémentations, la précision pour la même taille d'images varie peut, comme l'indique la figure (2.7). Mais pour le même nombre d'epochs, la précision est meilleure pour l'image 200×200 . Ceci peut s'expliquer par la présence de plus de détails. On note qu'initialement, le code d'origine avait une précision de 42.0338% pour 3 epochs et une taille d'image 50×50 .

70 60 50 Précision (en %) 40 30 20 Approche 1 Images 50x50 10 Approche1 Images 0 10 12 16 18 14 Nombre de processus

Précision en fonction du nombre de processus

Figure 2.8: Précision obtenue en fonction du nombre de processus, pour les approches 1 et 1bis, tailles d'images 50×50 et 200×200 .

2.4 Approche 2: Décomposition de l'image

Jusqu'à maintenant, le programme est lancé sur plusieurs processus qui exécutent la même chose et seulement l'application du filtre dans la couche de convolution est parallélisée.

Il serait intéressant de décomposer l'image dès sa lecture. Ainsi, chaque processus reçoit uniquement un sous-domaine de l'image. L'application de cette approche est facile pour la couche de convolution et de max-pooling. Cependant, il n'est pas évident pour la partie softmax et calcul de backpropagation. On a essayé d'implémenter cette approche dans la branche Aicha_Optimisation_Approach2, mais malheureusement sans succès. La figure (2.9) montre la partie du code dans laquelle circule l'image décomposée.

L'image est lue (en utilisant *imread* de openCv) sur un seul processus, ensuite elle est décomposée et les sous-domaines sont partagés en utilisant *MPI_Scatterv*. Après la push de max-pooling, les différents sous-domaines doivent être assemblés pour continuer la couche de softmax avec une entrée complète. La difficulté d'implémentation réside dans le rassemblement des entrées des parties max-pooling et convolution dans la back-propagation.

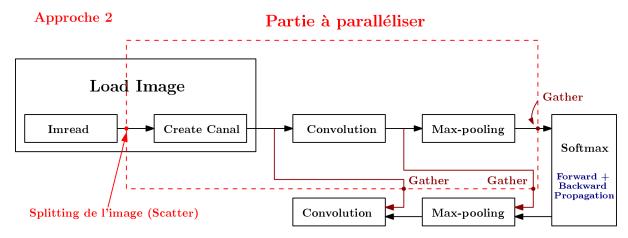


Figure 2.9: Partie a paralléliser dans l'approche 2.

2.5 Approche 3: Décomposition de l'ensemble des images

Appendix A

Appendix Chapter