PBD - HLS - Première partie : Profilage

Groupe 1 :
LEGOUEIX Nicolas
MALLET Julien
VIGUIER Franck

24 novembre 2021

1 Fonctions

Le projet est composé des 3 fichiers sources main.c, timers_b.c et fonctions_ES.c main.c contient les fonctions *calculateLayer* et la fonction 'main'.

1.1 Fonctions de main.c

Le fichier **main.c** contient la fonction *main* et les fontions *calculateLayer*.

calculateLayer$_i$ La fonction *calculateLayer*$_i$ prend en entrée la couche $i-1$, les poids à appliquer et écrit le résultat dans la couche i dont l'adresse est passé en paramètres. Les couches et les poids sont des tableaux de float. La fonction *calculateLayer*$_1$ quant à elle prend en entrée non pas la couche $i-1$ mais le tableau contenant l'image à traiter. Elle ne fait pas de calcul mais copie l'image dans la variable stockant la couche 1.

main La fonction main a comme variables locales :

— Les tableaux de neurones

```
float
    Layer1_Neurons_CPU[IMGWIDTH*IMGHEIGHT],
    Layer2_Neurons_CPU[6*13*13],
    Layer3_Neurons_CPU[50*5*5],
    Layer4_Neurons_CPU[100];
```

Ces tableaux contiennent les couches successives de neurones de notre réseau.

— Le tableau spécifique à la dernière couche

```
double
    Layer5_Neurons_CPU[10];
```

Ce tableau contient la dernière couche qui est codé en flottant double précision.

— Un compteur pour les boucles du main.

```
int i;
```

— Un conteneur déterminant le score le plus grand (valeur max) détenu par l'ensemble des neurones de la couche 5. Utilisé à la fin du *main*.

```
double scoremax;
```

— Un tableau contenant un dessin du caractère à reconnaître.

```
float Input[29*29]
```

— Une variable représentant l'indexe du neurone ayant le meilleur score, donc le caractère reconnu

```
int indexmax;
```

main appelle dans un premier temps la fonction *InitHostMem* avant d'appeler successivement les fonctions *calculateLayer* puis lit la dernière couche du réseau de neurone : *Layer5_Neurons_CPU*. Cette fonction cherche la valeur max sur cette dernière couche. Le neurone ayant la plus grande valeur correspond au caractère identifié par le réseau. L'indice correspondant au neurone est dans la variable *indicemax* et est affiché à la fin du main et donc de l'application.
Les autres fonctions *calculateLayer* se chargent de mettre à jour la valeur du potentiel des neurones de chaque couche en fonction de la valeur du neurone de la couche précédente et du poids qui doit être appliqué à ce neurone.

1.2 Fonctions de fonctions_ES.c

Avant de pouvoir être utilisé pour la reconnaissance de chiffre, le réseau de neurones a été entraîné sur un dataset. Cet entraînement à permis d'ajuster les poids du réseau afin d'assurer un niveau de précision satisfaisant pour classifier de futur chiffres manuscrits. Dans le cadre de ce projet, nous souhaitons simplement utiliser le réseau. Les différents poids calculés à partir de l'entraînement sont ainsi enregistrés dans des fichiers .wei que nous utiliserons pour initialiser notre réseau de neurones.

InitHostMem La fonction *InitHostMem* permet d'initialiser les poids aux neurones de chaque couche en fonction des fichiers .wei fournis. Pour se faire, elle lit successivement les fichiers .wei associés à chaque couche de neurones et remplis les tableaux de poids associés.

ReadIn La fonction *ReadIn* est utilisée pour lire les fichiers d'entrées où sont présents les chiffres que le réseau de neurones doit classifier. Pour cela, elle ouvre le fichier d'entrée **in.neu** et le lit en remplissant la première couche du réseau de neurones.

output La fonction *output* écrit les 10 doubles contenus dans le tableau passé en argument dans un fichier **out.res**. Elle peut être utilisée par exemple pour écrire les valeurs de tous les neurones de la cinquième couche dans un fichier.

1.3 Fichier Timers_b.c

Timers_b.c est une librairie de fonctions de chronométrage en temps cpu selon l'architecture de la machine.

2 Compilation

Notre makefile utilise une règle unique de compilation :

```
%.o: %.c
	$(CC) $(CFLAGS) -pg -c $< -o $@
```

Cette règle génère les fichiers objets à partir des fichiers sources correspondants. Elle génère donc les fichiers **main.o, fonctions_ES.o** et **timers_b.o**.

```
neuralNetwork: $(OBJFILES)
	$(CC) $(CFLAGS) -pg $^ -lm -o $@
```

Cette règle s'occupe de l'édition des liens. Ces deux lignes doivent avoir l'option -pg pour activer le profilage et utiliser gprof.

Fichier complet :

```
CC = gcc
CFLAGS = -Wall
OBJFILES = main.o fonctions_ES.o timers_b.o wei.o

all: neuralNetwork

neuralNetwork: $(OBJFILES)
	$(CC) $(CFLAGS) -pg $^ -lm -o $@

%.o: %.c
	$(CC) $(CFLAGS) -pg -c $< -o $@

timers_b.c: timers_b.h

clean:
	rm -f *.o neuralNetwork gmon.out resnet.prof

.PHONY: all clean
```

Une fois compilé, il suffit d'appeler

```
./neuralNetwork
```

afin de lancer le programme. L'image à traiter dépendra de la valeur donnée tableau Input[] dans le fichier **main.c**.

3 Profilage - gprof

Nous avons ensuite modifié notre makefile afin d'activer l'option de profilage avec gprof grâce à l'argument -pg :

```
neuralNetwork: $(OBJFILES)
	$(CC) $(CFLAGS) -pg $^ -lm -o $@

%.o: %.c
	$(CC) $(CFLAGS) -pg -c $< -o $@
```

Une fois la compilation relancée avec ces modifications, un fichier **gmon.out** est créé lors de la prochaine exécution du programme. Ce fichier contient les données de profilage du programme, et peut être décodé par le logiciel gprof de la manière suivante :

```
gprof neuralNetwork gmon.out > resnet.prof
```

Cette commande affiche les résultats du profilage dans un fichier **resnet.prof** dans un format humainement lisible.
Cependant, nous avons remarqué que cette application n'a pas une résolution suffisante afin de chronométrer efficacement les temps d'exécutions de chaque fonctions. Nous avons donc décidé de

mesurer le temps pris pour 10.000 exécutions de chaque fonctions, c'est-à dire que nous avons enveloppé les appels au 5 fonctions $calculateLayer_i$ et *initHostMem* dans une boucle se répétant 10.000 fois. Voici la sortie obtenue :

% time	cumulative (s)	self (s)	calls	self (us / call)	total (us / call)	name
57.09	7.05	7.05	10000	705.00	705.00	calculateLayer3
34.49	11.31	4.26	10000	426.00	426.00	calculateLayer4
8.18	12.32	1.01	10000	101.00	101.00	calculateLayer2
0.24	12.35	0.03	10000	3.00	3.00	calculateLayer5
0.00	12.35	0.00	10000	0.00	0.00	InitHostMem
0.00	12.35	0.00	10000	0.00	0.00	calculateLayer1

On remarque que l'essentiel du temps est passé sur le calcul des niveaux 3 et 4 du réseau de neurones, ce qui est logique dans la mesure où layer 3 est la couche contenant le plus d'éléments (125.100). Pour Layer4, bien qu'il n'y ai moins d'éléments à calculer que pour layer2, les calculs sont plus coûteux car les boucles sont plus longues.
Le Layer5 est rapide à traiter, car il est relativement petit étant donné qu'il s'agit de la couche de sortie. De même, Layer1 ne termine pas en un temps mesurable par gprof malgré les nombreuses itérations car il ne s'agit que d'un seul appel à la fonction *memcpy*, très rapide à exécuter en comparaison aux nombreuses boucles imbriquées utilisées dans les fonctions *calculateLayer* .

4 Profilage - *dtime*

Le principal inconvénient de gprof est qu'il ne mesure le temps qu'a intervalles réguliers (0.01 secondes). Ce n'est pas très précis, et empêche de mesurer le temps pris par une fonction si ce dernier est inférieur à cette valeur. Pour cette raison, nous utilisons la fonction *dtime* fournie dans le fichier **timers_b.c**. Cette dernière se base sur des appels systèmes afin d'obtenir une définition bien plus précise du temps écoulé, toujours exprimé en secondes.
Il suffit alors d'appeler *dtime* avant et après l'exécution de chaque fonction individuellement et de soustraire les deux valeurs obtenues afin d'obtenir le temps qu'a pris la fonction à s'exécuter.
Afin de conserver la même échelle que pour notre trace utilisant gprof, nous avons gardé la boucle lançant 10.000 fois l'ensemble des fonctions du programme. Voici nos résultats :

Name	Time (s)
L3	7.2366870000
L4	4.6107110000
L2	1.3202630000
Init	0.8084680000
L5	0.0431170000
L1	0.0041850000

Globalement, les temps d'exécutions sont similaires, mais on obtient des temps cohérents vers le bas du classement. Le fait que Init a ici pris plus de temps que Layer5 est probablement dû à une combinaison du gain de précision donné par *dtime* et aux variations liées à l'occupation du processeur au moment de l'exécution.
Pour le reste, les observations restent identiques : L3 et L4 sont les plus gros consommateurs de temps d'exécution.