PROJET ASR

Analyse de performance de TensorFlow

Rédigé par:
BOUZGHAIA Fatma
MESSAOUDI Mehdi

Encadrant:
TRAHAY François

Année Universitaire: 2020 - 2021

Table des matières

Introduction

Le Deep Learning est un domaine d'intelligence artificielle qui permet aux ordinateurs d'apprendre avec l'expérience et de comprendre le monde en termes de hiérarchie de concepts, chaque concept étant défini par sa relation avec des concepts plus simples. Si nous dessinons un graphique montrant comment ces concepts sont construits les uns sur les autres, le graphique est profond, avec de nombreuses couches. Pour cette raison, il est appelé apprentissage profond (Deep Learning). Bien que les concepts théoriques sous-jacents ne soient pas nouveaux, le Deep Learning est devenu une tendance au cours de la dernière décennie en raison de nombreux facteurs, y compris son application réussie dans une variété de solutions de problèmes, la conception de Convolutional Neural Network (CNN) et une meilleure accessibilité aux ordinateurs haute performance.

A cet effet de nombreuses bibliothèques d'apprentissage en profondeur sont apparues, telles que TensorFlow, Theano, CNTK, Caffe, Torch, Neon et PyTorch. Comme il existe une variété de bibliothèques open source disponibles, les développeurs ont besoin d'études expérimentales scientifiques qui indiquent quelle bibliothèque est la plus appropriée pour une application déterminée et comment utiliser les outils d'analyse de performance existants pour optimiser l'utilisation de ce framework. Dans ce contexte, le présent travail évalue la bibliothèque Tensorflow et les outils existants pour optimiser son utilisation.

Tensorflow est un framework python qui permet de faire de l'IA. Depuis son lancement en 2015, Tensorflow est devenu en un temps record la référence pour faire du Deep Learning, utilisé aussi bien dans la recherche qu'en entreprise.

Au-delà de la popularité qui entoure ce framework et les projets émergents grâce à ce dernier, Tensorflow reste un outil assez difficile à maîtriser pour être utilisé pleinement et efficacement. Afin d'optimiser les projets utilisant ce framework, des outils d'analyse de performance ont été introduits afin d'enrichir l'utilisation de Tensorflow.

L'analyse des performances est une discipline spécialisée impliquant des observations systématiques pour améliorer les performances et améliorer la prise de décision. Dans le contexte de ce rapport, l'analyse ou le profilage «performance» se référera à l'analyse de la vitesse à laquelle l’entraînement d'un modèle de réseau convolutifs est effectué (telle que mesurée, par exemple, par le débit de formation en itérations par seconde), et la manière dont la session utilise les ressources systèmes pour atteindre cette vitesse.

En effet, TensorFlow prend en charge à la fois les CPU et les GPU pour entraîner des modèles. Dans un cas idéal, un programme doit avoir une utilisation élevée du GPU, une communication minimale entre le CPU et le GPU et aucune surcharge du pipeline d'entrée. Et c’est grâce aux outils d’analyse de performance que l'optimisation de ces critères est possible.

1. Objectif du projet

Notre projet consiste à étudier les outils d'analyse existants de TensorFlow qui est un framework python qui permet de faire de l'IA et de déployer des calculs sur GPU et de développer de nouvelles analyses, notamment en regardant ce qu'il se passe à bas niveau (au niveau du cuda).

Pour cela, nous avons à notre disposition deux modèles que nous expliquerons dans la section 3. En premier lieu, ces deux modèles nous permettent de prendre en main la technologie TensorFlow. En second lieu, d'appliquer différentes optimisations sur les deux modèles dans le but de voir leur impact sur l'utilisation de la GPU par TensorFlow.

2. État de l'art

2.1. TensorFlow

TensorFlow est une bibliothèque de logiciels open source pour le calcul numérique à l'aide de graphiques de flux de données.

La bibliothèque Tensorflow intègre différentes API pour construire une architecture d'apprentissage en profondeur à grande échelle comme CNN ou RNN. TensorFlow est basé sur le calcul de graphes; il permet de visualiser la construction du réseau neuronal avec Tensorboard qui permet de déboguer le programme. Enfin, Tensorflow est conçu pour être déployé à grande échelle. Il fonctionne sur CPU et GPU.

Nous privilégions l'utilisation de la GPU puisque la GPU est très rapide pour faire des calculs surtout que l'apprentissage profond repose sur une multiplication matricielle importante.

2.2. Outils d'analyse des performances

Nous voulons toujours que nos modèles s'entraînent plus rapidement, nous optimisons donc notre code et nous favorisons l'utilisation de la GPU pour accélérer l'exécution des opérations.

Cependant, il est possible que même après avoir accéléré les calculs, le modèle présente des inefficacités dans la pipeline lui-même (présence de Bottlenecks), et par conséquent, l'exécution est ralentie. Dans de tels cas, il est très difficile de déboguer son code, ou même de dire ce qui cloche.

Cela peut être résolu en utilisant différents outils d'analyse de performances, appelés aussi Profilers. Les profileurs sont des outils qui aident à comprendre la consommation de différentes ressources matérielles (temps et mémoire) des différentes opérations

TensorFlow et à résoudre les goulots d'étranglement, accélérant ainsi l'exécution du modèle.

Dans cette partie nous présentons deux outils d'analyse de performance, TensorBoard et NVIDIA Profiler.

2.2.1. TensorBoard

TensorFlow propose le TensorFlow Profiler appelé également Tensorboard pour le profilage des modèles tf. Fondamentalement, Tensorboard surveille la formation du modèle. Il stocke le temps nécessaire à l'exécution des opérations ainsi que le temps nécessaire à l'exécution des différentes étapes du modèle. Il note aussi des informations concernant l'utilisation des ressources coûteuses (CPU et GPU) en termes de mémoire et de temps. Toutes ces informations sont regroupées sous la forme de plusieurs graphes visuels permettant d'interpréter les résultats de l'exécution du modèle.

L'intégration et l'activation du profileur est assez simple. Il suffit de faire appel à la méthode callbacks.TensorBoard de l'API tf.keras

```
# Create a TensorBoard callback
logs = "logs/" + datetime.now().strftime("%Y%m%d-%H%M%S")
tboard_callback = tf.keras.callbacks.TensorBoard(log_dir = logs,
                                                 histogram_freq = 1,
                                                 profile_batch = '500,520')
```

2.2.1.1. Utilisation du profileur:

- **Overview Page**

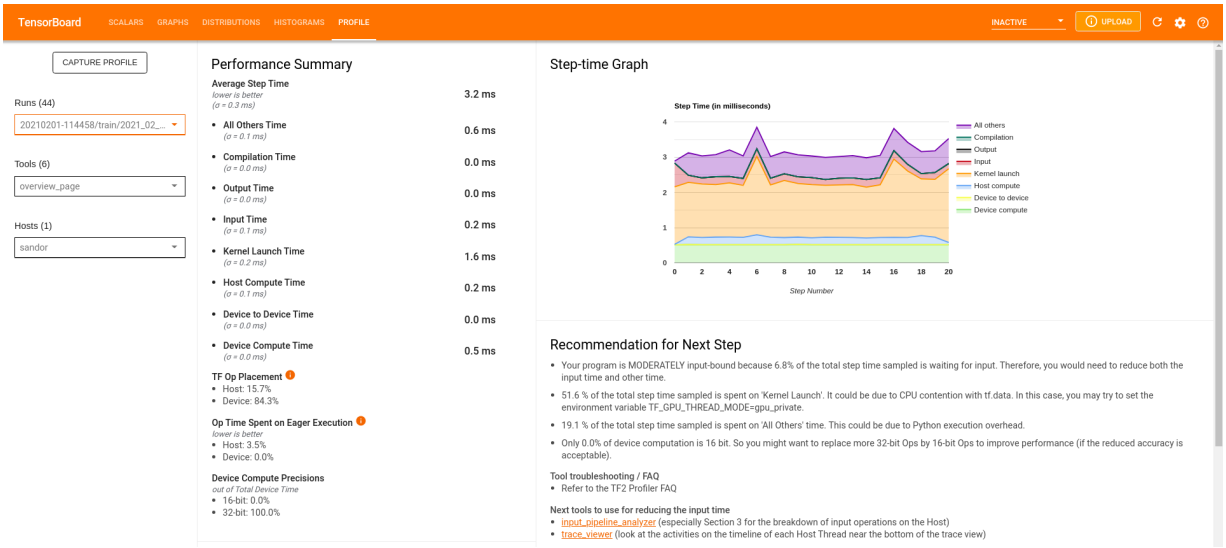

Figure 2.1: Profiler Overview Page of a simple_CNN execution

Cette section donne un résumé du niveau des performances du modèle. Parmi les éléments on peut distinguer:

- Step-time Graph : est un tracé du nombre de pas en fonction du temps nécessaire pour exécuter l'étape correspondante. Il montre aussi quelle partie de temps est utilisée par chaque catégorie. Dans la figure ci-dessus, présentant une exécution du modèle CNN Simple, on peut voir que notre modèle est "modérément lié à l'entrée" puisque 6.8% du temps correspond à des opérations d'entrée.
- Average Step Time : Donne un rapport détaillé sur le temps de pas moyen. Dans un cas idéal, nous voulons que notre modèle passe la plupart de son temps à réellement s'entraîner, c'est-à-dire à maintenir le GPU occupé (le Device Compute Time doit être le plus élevé et tous les autres temps doivent être le plus faible possible).
- TF Op Placement : (sous Performance Summary) affiche le % des opérations exécutées sur l'Host (CPU) par rapport au Device (GPU). Pour maximiser l'utilisation du GPU, le pourcentage d'utilisation du device doit être maximal.

- **Trace Viewer**

Figure 2.2: Profiler Trace Viewer Page of a simple_CNN execution

Cette section permet de d'avoir une visualisation détaillée des durées de chaque opération Tensorflow. À gauche, on peut voir deux sections principales (Device et Host). Cela nous indique quelle opération Tensorflow a été exécutée sur quel périphérique (GPU ou CPU). À droite, les barres colorées sont les durées pendant lesquelles les opérations TensorFlow respectives ont été exécutées.

Idéalement, GPU compute timeline doit être encombrée (c'est-à-dire que le GPU est occupé). Les copies d'hôte à périphérique et de périphérique à hôte (H2D et D2H) doivent être minimales.

- **Autres sections**

Le reste des outils donne des informations complètes sur les catégories ou opérations spécifiques couvertes dans la Overview Page et le Trace Viewer. parmi ces outils

- Input Pipeline Analyzer : Vérifie le pipeline d'entrée et indique s'il existe un goulot d'étranglement des performances.
- GPU Kernel Stats : Affiche les statistiques de performances pour chaque noyau GPU.
- Memory Profile : Profile l'utilisation de la mémoire du GPU.
- TensorFlow Stats : donne un aperçu des performances de chaque opération TensorFlow exécutée.

2.2.2. NVIDIA Profiler

Pour comprendre l'utilisation efficace des différentes hiérarchies de mémoire, il est important d'analyser les caractéristiques des applications au moment de l'exécution. Les profilers sont des outils très pratiques qui mesurent et affichent différentes métriques qui nous aident à analyser la façon dont la mémoire, la GPU, les cœurs et d'autres ressources sont utilisés. NVIDIA a pris la décision de fournir une API que les développeurs d'outils de profilage peuvent utiliser pour se connecter à une application CUDA.

Cette API utilise l'interface CUDA Profiler Tools Interface (CUPTI) pour fournir des informations de profilage pour les applications CUDA. NVIDIA développe et entretient lui-même des outils de profilage fournis dans le cadre du CUDA Toolkit.

CUDA Toolkit fournit deux outils de profil d'application GPU, le NVIDIA Profiler (NVPROF) et le NVIDIA Visual Profiler (NVVP). NVPROF est un outil de ligne de commande, tandis que NVVP a une interface visuelle. La fenêtre NVVP Profiler que nous utiliserons se présente comme suit:

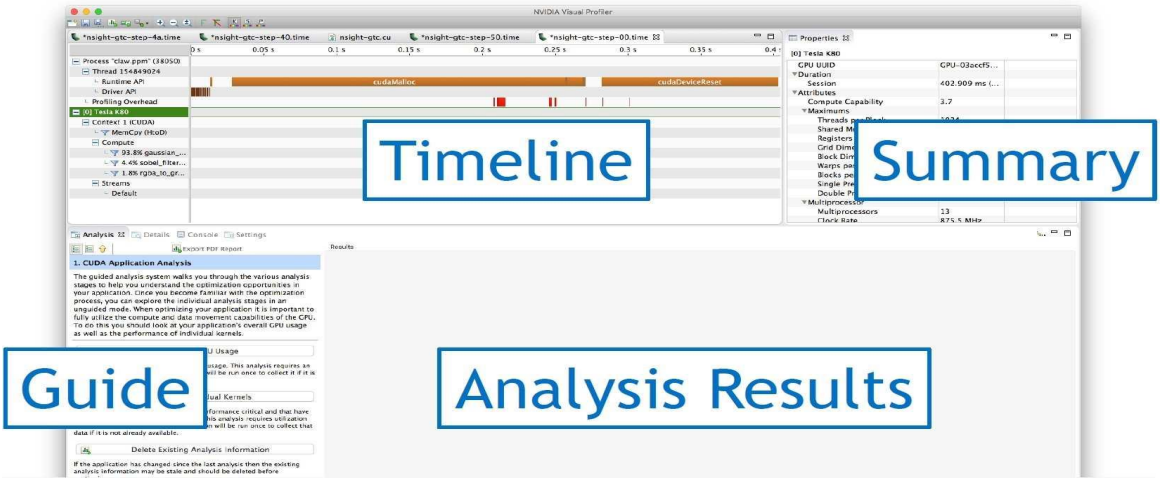

Figure 2.3: NVidia Profiler Viewer Page of a simple_CNN execution

Cette interface présente 4 fonctionnalités: Chronologie, Guide, Résultats d'analyse et Résumé.

L'interface **Chronologie**, comme son nom l'indique, montre l'activité du CPU et du GPU qui s'est produite au fil du temps.

L'interface **Résultats d'analyse** affiche le résultat de l'analyse. Le Visual Profiler propose deux modes d'analyse:

- Analyse guidée: elle guide le développeur en adoptant une approche étape par étape pour comprendre les principaux limiteurs de performance.
- Analyse non guidée: le développeur doit regarder manuellement les résultats dans ce mode pour comprendre le limiteur de performance.

2.3. La base de données CIFAR-10

L'ensemble de données CIFAR-10 se compose de 60,000 images couleur 32x32 en 10 classes, avec 6000 images par classe. Il y a 50,000 images d'entraînement et 10,000 images de test.

L'ensemble de données est divisé en cinq échantillons de training et un échantillon de test, chacun contenant 10,000 images. L'échantillon de test contient exactement 1,000 images sélectionnées au hasard dans chaque classe. Les échantillons de training contiennent les images restantes dans un ordre aléatoire, mais certains de ces échantillons peuvent contenir plus d'images d'une classe que d'une autre. Parmi eux, les échantillons de training contiennent exactement 5,000 images de chaque classe.

L'image ci-dessous nous montre les 10 classes de la base de données CIFAR-10 ainsi que 10 images aléatoires de chacune des classes.

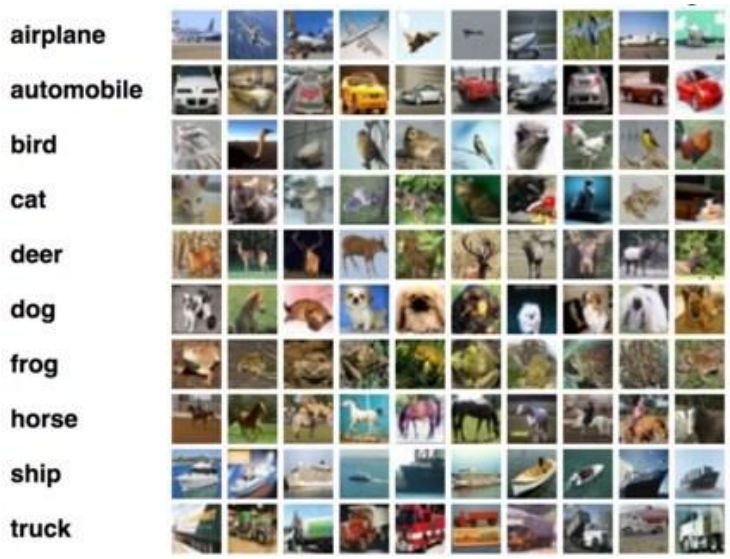

Figure 2.4: CIFAR-10 dataset overview

Dans notre cas, pour les deux modèles que nous travaillons avec, nous utilisons que 5% de tout l'échantillon qui a une taille totale de 50,000 images.

3. Nos modèles et leurs architectures

3.1. Le modèle CNN simple

Notre premier modèle est un réseau de neurones convolutifs simple qui est formé par un empilement de couches de traitement qui sont les suivantes:

- **Une couche de convolution:** Les couches convolutives traitent une image d'entrée en faisant glisser un certain nombre de petits filtres à travers chaque région possible et en sortant le produit scalaire du filtre et l'image au niveau de chaque région. Les paramètres apprenables pour une couche conventionnelle sont les poids de chaque filtre et une valeur du biais pour chaque filtre. Au niveau de notre modèle, on réalise en succession l'opération de convolution de l'image en entrée avec un filtre de taille précisée pour chaque couche qui est de 3×3.
- **Max pooling Layers:** Pour chaque sous matrice de taille 2×2 le maximum des 4 pixels est choisi comme valeur de sortie. L'objectif de cette couche est de sous-échantillonner l'image en réduisant sa dimension.
- **Flatten:** cette couche permet de recaler la sortie du dropout au bon format pour la couche suivante.
- **Dense:** c'est une couche régulière de neurones dans un réseau neuronal. Chaque neurone reçoit l'entrée de tous les neurones de la couche précédente, donc densément connectés. La couche a une matrice de poids W, un vecteur de biais b et les activations de la couche précédente a. Dans notre modèle, nous avons deux

couches denses de poids respectifs 64 et 10, sans vecteur de biais et avec une activation ReLu.

- **ReLU:** ou Rectified Linear Unit est une fonction d'activation ayant pour expression $f(x) = max(x, 0)$. Les couches ReLu n'ont pas de paramètres apprenables.

Le graphe suivant représente les différentes couches de notre premier modèle:

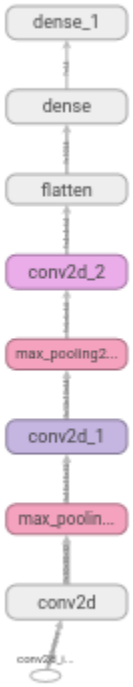

Figure 3.1: Architecture du modèle CNN

Nous avons utilisé cette architecture pour notre première application qui est **simple cnn**.

3.2. Le modèle RESNET

Notre deuxième modèle est le RESNET50 est un réseau neuronal convolutif d'une profondeur de 50 couches. l'architecture resnet 50 contient l'élément suivant:

Une convolution avec une taille de noyau de 7 * 7 et 64 noyaux différents, le tout avec une foulée de taille 2 nous donnant 1 couche. Ensuite, nous voyons la mise en commun maximale avec également une taille de foulée de 2. Dans la convolution suivante, il y a un noyau 1 * 1,64 qui suit un noyau 3 * 3,64 et enfin un noyau 1 * 1,256. Ces trois couches sont répétées au total 3 fois, ce qui nous donne 9 couches dans cette étape. Ensuite, nous voyons un noyau de 1 * 1,128 après cela, un noyau de 3 * 3,128 et enfin un noyau de 1 * 1,512 cette étape a été répétée 4 fois, ce qui nous a donné 12 couches dans cette étape. Après cela, il y a un noyau de 1 * 1,256 et deux autres noyaux avec 3 * 3,256 et 1 * 1,1024

et ceci est répété 6 fois, ce qui nous donne un total de 18 couches. Et puis à nouveau un noyau 1 * 1,512 avec deux autres de 3 * 3,512 et 1 * 1,2048 et cela a été répété 3 fois nous donnant un total de 9 couches. Après cela, nous faisons un pool moyen et le terminons avec une couche entièrement connectée contenant 1000 nœuds et à la fin une fonction softmax, donc cela nous donne 1 couche.

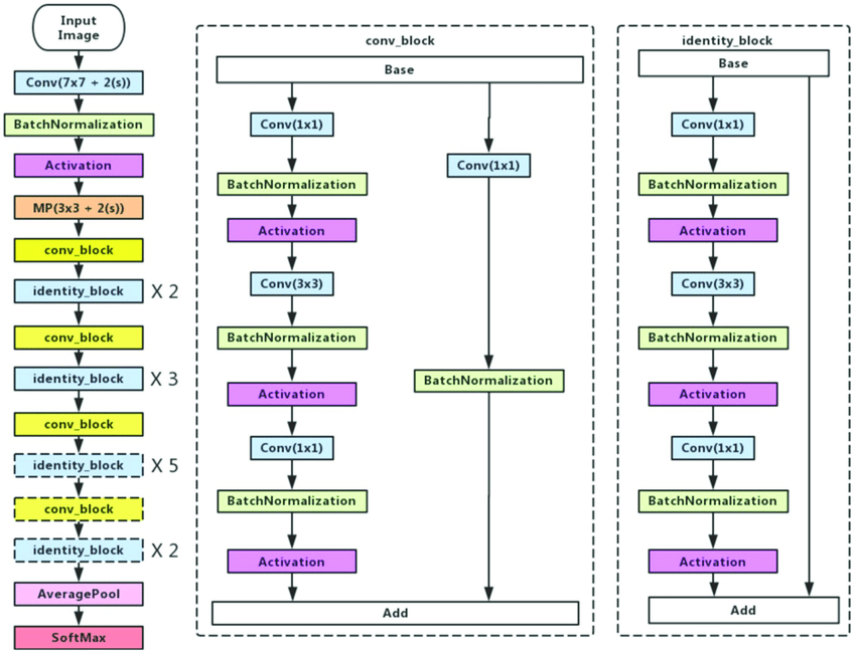

Figure 3.2: Architecture du modèle Resnet50

4. Réalisation des optimisations

Nous allons maintenant analyser les performances de TensorFlow à l'aide de TensorBoard en utilisant les deux applications que nous avons expliquées dans la section précédente.

Nous avons, en effet, utilisé que TensorBoard pour analyser les performances de TensorFlow puisque Nvidia Profiler est trop bas niveau pour cette technologie et ne nous donne pas de résultats compréhensibles pour les interpréter.

4.1. Contexte des optimisations

Avant de montrer et synthétiser nos résultats, nous expliquons tout d'abord le contexte de nos optimisations.

Tout d'abord, pour les deux applications que nous avons, nous utilisons un nombre d'échantillons de 50,000 images. C'est-à-dire que nous entraînons notre modèle sur 50,000 images de la base de données CIFAR-10 et nous testons sur les 10,000 images restantes.

Pour une meilleure gestion des différentes optimisations, nous avons utilisé un analyseur d'arguments pour:

- Le batch size: cet argument nous permettra de varier le nombre d'échantillons propagés dans le réseau.
- Le mode de la GPU: nous pouvons utiliser la GPU en mode privé (section 4.3.1). Si cet argument a une valeur de 1 alors nous utilisons la GPU en mode privé dans notre application.
- La stratégie de précision: définir une stratégie de précision et varier les types de précision existants. Si cet argument est à 1, alors une stratégie de précision est demandée. Dans ce cas, il faut voir l'argument qui est le type de politique à mettre en stratégie. Nous avons soit une stratégie mixte de float16 ou bien une stratégie float32. Cette stratégie sera expliquée dans la section 4.3.2.1.

Ainsi, pour faire fonctionner nos applications, nous avons ces deux lignes de commande:

```
$ python3 app_cnn.py --batch_size <batch_size> --gpu_mode <1 or 0>
--mixed_precision <1 or 0> --policy_type <16 or 32>
```

```
$ python3 app_resnet50.py --batch_size <batch_size> --gpu_mode <1 or 0>
--mixed_precision <1 or 0> --policy_type <16 or 32>
```

Pour gagner du temps, nous avons écrit un script bash pour chaque application qui nous permet de les lancer avec les différents arguments, tout en changeant les valeurs à chaque fois. Tout ça, est dans une boucle qui permet de doubler le batch size d'une itération à une autre.

Dans le but d'analyser les profils de nos applications avec TensorBoard, nous avons choisi de travailler avec un batch size de 256 pour l'application **simple cnn** et un batch size de 64 pour l'application **resnet**. Nous faisons ainsi des comparaisons entre les différentes optimisations que nous appliquons à nos modèles à l'aide de ces deux cas que nous avons choisis.

4.2. Variation du batch size

Tout d'abord, le batch size permet de définir le nombre d'échantillons qui seront propagés à travers le réseau convolutif.

Ainsi pour notre première analyse de performance, nous avons fait varier le batch size allant de 8 jusqu'à 32,768 pour l'application **simple cnn** et pour l'application **resnet** nous avons varié le batch size de 16 jusqu'à 64 faute de mémoire. En effet, au-delà d'un batch size de 64, la GPU n'a pas assez de mémoire pour le faire fonctionner à cause des lourdes couches que le modèle resnet possède.

Nous avons tracé les graphes suivants qui nous renseignent le temps pris par chaque itération de chaque modèle (**simple cnn** et **resnet**) pour une valeur différente du batch size:

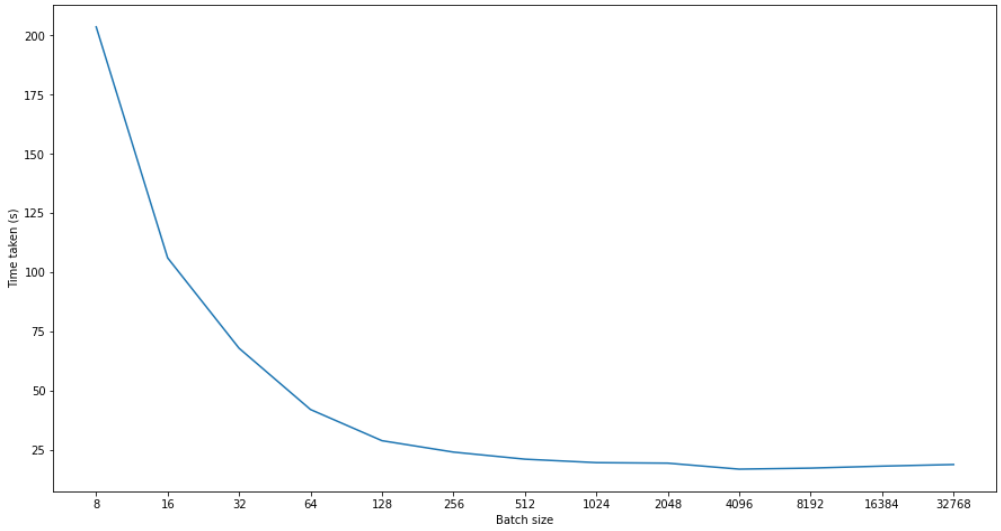

Figure 4.1: Tracé de la variation du batch size pour l'application simple cnn

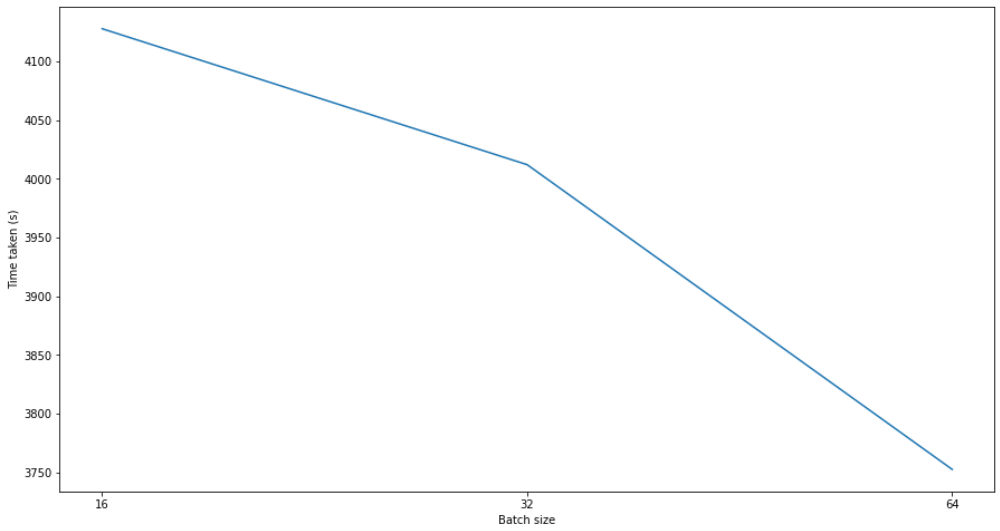

Figure 4.2: Tracé de la variation du batch size pour l'application resnet

Nous pouvons clairement constater que le temps d'exécution diminue considérablement à chaque fois que nous augmentons le batch size.

Nous utilisons maintenant la plateforme TensorBoard pour interpréter les résultats que nous avons. Les figures 4.3 et 4.4 présentent les captures du profil respectivement pour un batch size de 256 pour l'application **simple cnn** et un batch size de 61 pour l'application **resnet**.

Dans le premier profil que nous avons, nous constatons que notre application **simple cnn** passe près de 34,9% du temps à lancer le kernel (couleur orange) qui est le temps que le CPU passe au lancement du GPU calculer les noyaux sur l'appareil. Pour faire face à ce problème, TensorBoard nous recommande d'utiliser la GPU en mode privé.

Dans le deuxième profil, notre application **resnet** est totalement optimisée puisqu'elle passe presque la totalité du temps dans le calcul sur la GPU.

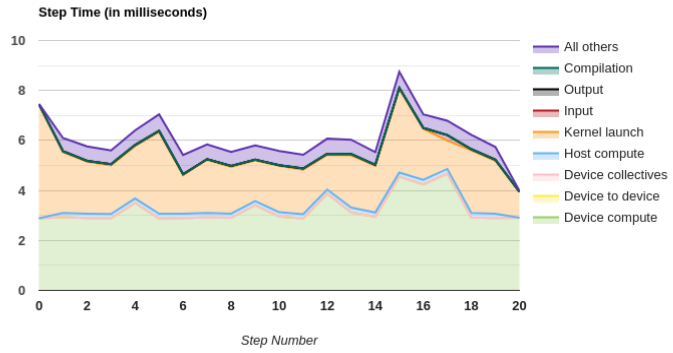

Figure 4.3: Profil d'exécution de l'application simple cnn avec un batch size de 256

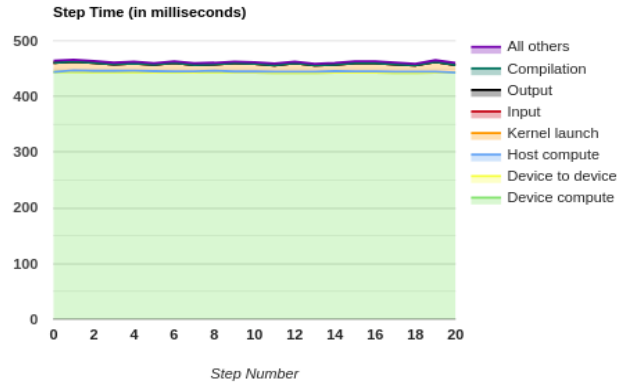

Figure 4.4: Profil d'exécution de l'application resnet avec un batch size de 64

De plus, nous constatons aussi, que notre application **simple cnn** passe beaucoup de temps dans le calcul (couleur verte claire). Ceci est peut être dû à l'utilisation des décimaux à 32 bits et non pas à 16 bits. Ceci dit, TensorBoard nous recommande d'utiliser des opérations à 16 bits au lieu de 32 bits.

Néanmoins, nous pouvons clairement voir que nos modèles ne sont pas fortement liés à l'entrée (c'est-à-dire qu'il ne passe pas beaucoup de temps dans le pipeline d'entrée de données (couleur rouge)). Ce qui est une bonne chose puisque généralement c'est un problème très courant dans les modèles.

4.3. Application des différentes optimisations

4.3.1. Utilisation du la GPU en mode privée

Suite aux recommandations de TensorBoard, nous avons configuré nos deux applications **simple cnn et resnet** pour qu'elle utilise la GPU en mode privé. Cela garantit que les noyaux GPU sont lancés à partir de leurs propres threads dédiés et ne sont pas mis en file d'attente derrière le travail de tf.data.

D'après les figures 4.5 et 4.6, nous observons que le passage vers la GPU privée diminue légèrement le temps d'exécution de nos applications.

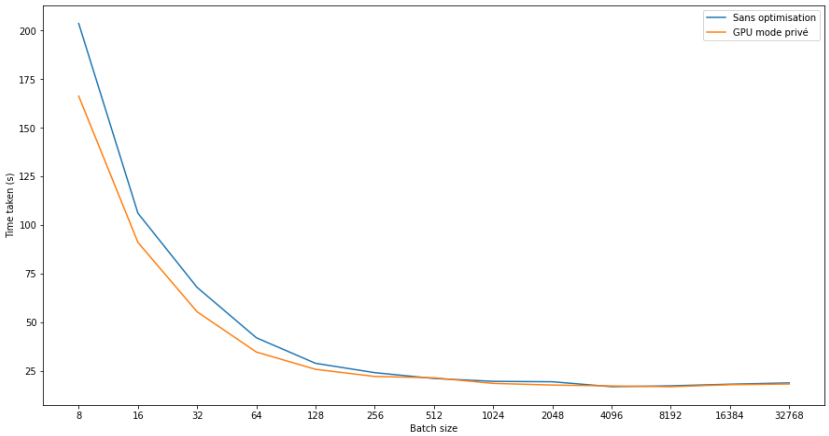

Figure 4.5: Comparaison de l'exécution de l'application simple cnn avec et sans GPU privée

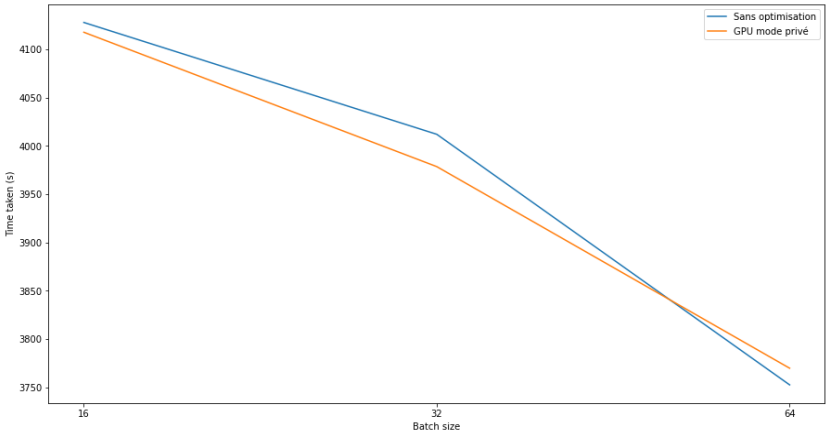

Figure 4.6: Comparaison de l'exécution de l'application resnet avec et sans GPU privée

D'après le résumé des performances de TensorBoard, nous collectons les valeurs données par les tableaux 4.1 et 4.2.

Optimisations sur simple cnn	**Average step-time (en ms)**	**Device Compute Time (en ms)**	**TF Op Placement -> Device (en %)**
Sans GPU privée	6.1	3.2	85.7
Avec GPU privée	6.5	3.2	85.7

Tableau 4.1: Résumé de performance de l'application simple cnn avec et sans GPU privée

Optimisations sur resnet	**Average step-time (en ms)**	**Device Compute Time (en ms)**	**TF Op Placement -> Device (en %)**
Sans GPU privée	461.7	443	99.6
Avec GPU privée	461.6	442.8	99.6

Tableau 4.2: Résumé de performance de l'application resnet avec et sans GPU privée

Nous remarquons qu'avec le mode privé de la GPU, le temps de pas moyen augmente pour le cas de l'application **simple cnn**. Ceci est dû au fait que le temps que le CPU passe au lancement du GPU a augmenté par rapport à l'application sans cette optimisation. Nous pouvons confirmer nos dires avec cette capture de TensorBoard qui nous présente un pourcentage de 39.9% de temps total est dédié au lancement de la GPU. Nous remarquons aussi que le temps de calcul sur la GPU reste toujours le même (3.2 ms).

Quant à l'application **resnet**, nous n'apercevons pas une réelle différence en performance en utilisant la GPU en mode privée et le profil reste inchangé par rapport à la figure 5.4.

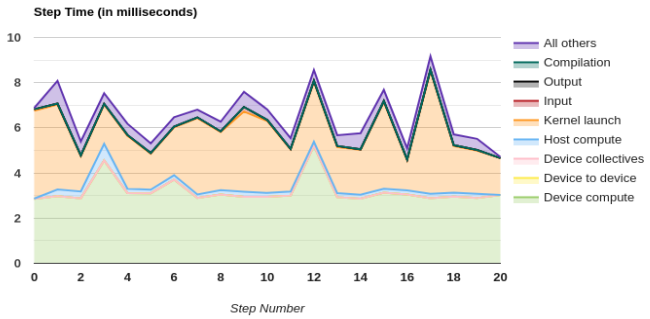

Figure 4.7: Profil de l'exécution simple cnn par une GPU privée

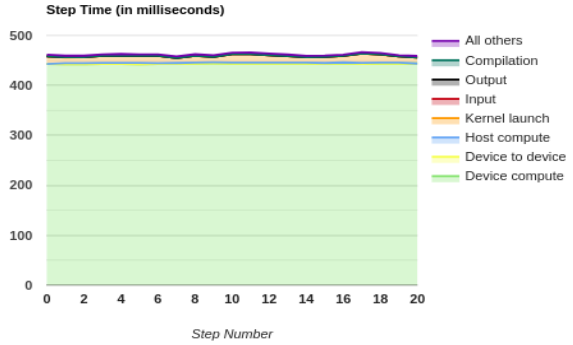

Figure 4.8: Profil de l'exécution resnet par une GPU privée

Malgré le fait que nous avons mis notre GPU en mode privée, TensorBoard nous recommande encore d'utiliser le mode privé.

4.3.2. Utilisation de la précision mixte

4.3.2.1. Définition de la précision mixte

La précision mixte est l'utilisation de types à virgule flottante 16 bits et 32 bits dans un modèle pendant l'entraînement pour le rendre plus rapide et utiliser moins de mémoire. En conservant certaines parties du modèle dans les types 32 bits pour la stabilité numérique, le modèle aura un temps de pas inférieur.

La différence entre les types float16 et float32 revient au fait que float32 est un nombre de 32 bits alors que float16 utilise 16 bits. Cela signifie que les float32 occupent deux fois plus de mémoire - et les opérations sur eux peuvent être beaucoup plus lentes dans certaines architectures de machines.

Cependant, les float32 peuvent représenter des nombres beaucoup plus précis que les nombres à virgule flottante 16 bits et donc ils permettent également de stocker des nombres beaucoup plus importants.

4.3.2.2. Résultats de l'optimisation

Pour de meilleurs résultats de l'application des stratégies de précision, nous avons testé nos applications sur deux cas:

- Utilisation des stratégies sans la GPU privée
- Utilisation du mode privé de la GPU avec les stratégies de précision

Utilisation de la précision mixte sans le mode privé de la GPU

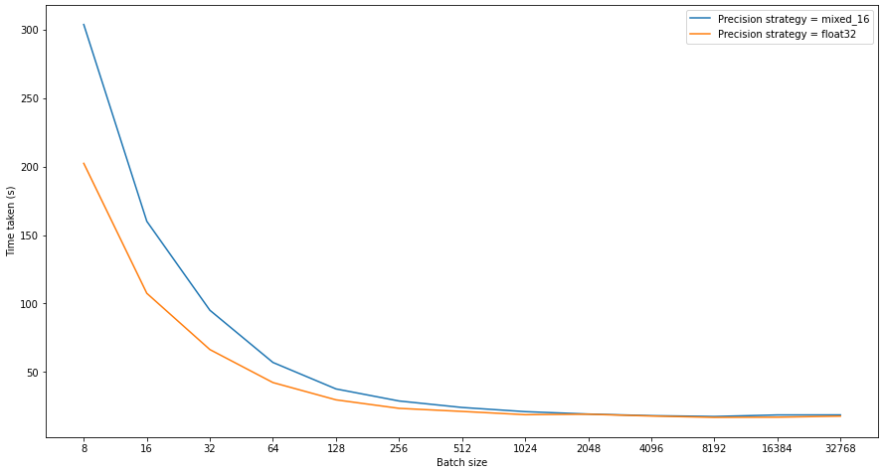

Figure 4.9: Comparaison de l'exécution de l'application simple cnn avec une stratégie mixed_float16 et float32

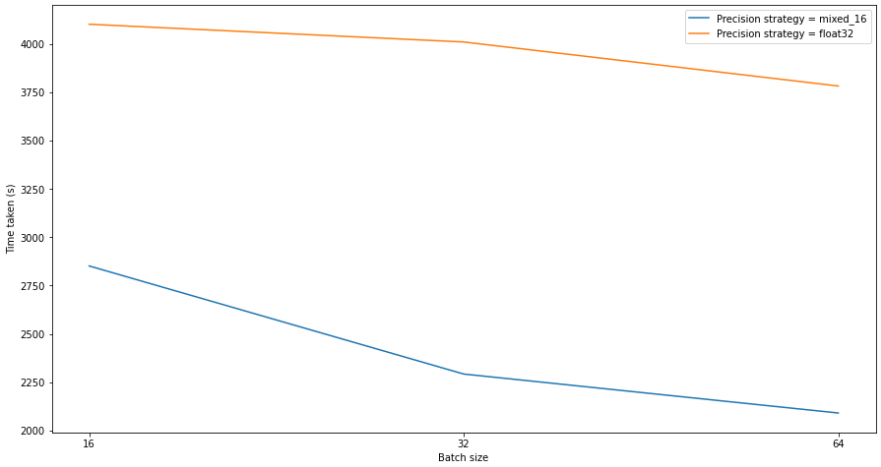

Figure 4.10: Comparaison de l'exécution de l'application resnet avec une stratégie mixed_float16 et float32

Les figures 4.9 et 4.10 nous indiquent que le temps d'exécution de nos applications avec une stratégie de précision du type float32 est meilleur que celui avec une stratégie du type float16, surtout pour l'application resnet. Cependant, la figure 5.6 nous indique que les performances avec une stratégie float16 est bien meilleure que celle en float32.

Par ailleurs, nous observons une augmentation du temps total lié à l'entrée et une diminution du temps de calcul sur la GPU pour la stratégie float32.

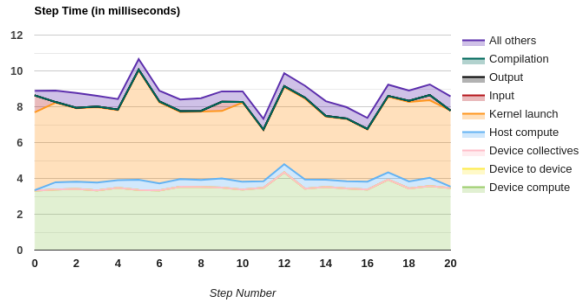

Figure 4.11 (a) : Stratégie de précision du type mixed_float16

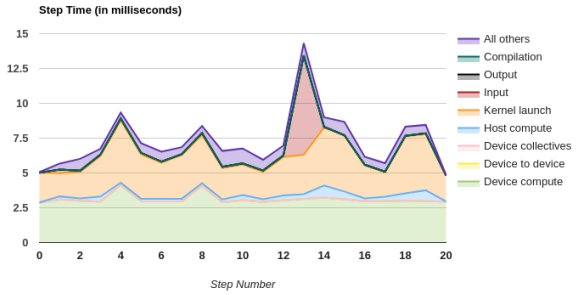

Figure 4.11(b): Stratégie de précision du type float32

Figure 4.11: Comparaison pour l'application simple cnn entre les deux types de stratégies sans GPU privée

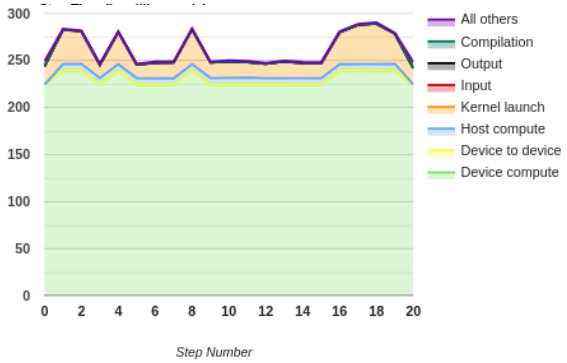

Figure 4.12 (a) : Stratégie de précision du type mixed_float16

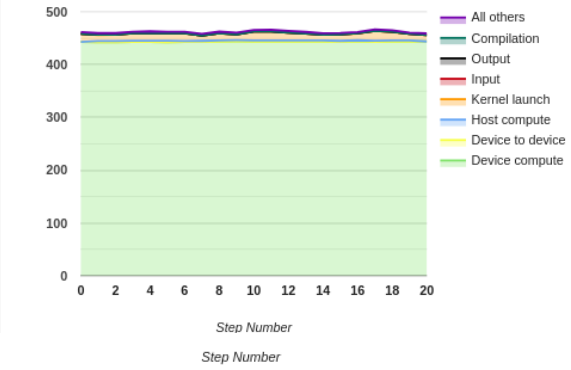

Figure 4.12(b): Stratégie de précision du type float32

Figure 4.12: Comparaison pour l'application simple cnn entre les deux types de stratégies sans GPU privée

Pour l'application **resnet**, nous constatons une augmentation du kernel launch, qui était négligeable sans l'utilisation d'une stratégie de précision du type mixed float16.

Optimisations	**Average step-time (en ms)**	**Device Compute Time (en ms)**	**TF Op Placement -> Device (en %)**
Stratégie mixed_float16	8.7	3.5	91.9
Stratégie float32	7.3	3.1	83.0

Tableau 4.3: Résumé de performance de l'application simple cnn sans GPU privée avec une stratégie de précision

Optimisations	**Average step-time (en ms)**	**Device Compute Time (en ms)**	**TF Op Placement -> Device (en %)**
Stratégie mixed_float16	261.8	230.1	99.6
Stratégie float32	431	443	99.6

Tableau 4.4: Résumé de performance de l'application resnet sans GPU privée avec une stratégie de précision

Les tableaux 4.3 et 4.4 nous confirment le fait que la stratégie mixed_float16 sans utilisation privée de la GPU est meilleure que la stratégie float32 au niveau de l'utilisation de la GPU pour les deux applications. En effet, la première stratégie nous donne une utilisation de 91.9% qui est très bien pour l'application **simple cnn** et un temps de pas moyen largement inférieur pour l'application **resnet**. Cependant, pour **simple cnn**, la deuxième stratégie nous permet d'avoir un temps moyen par pas plus faire, qui est donc meilleure que la première stratégie. Malgré ce faible temps, la stratégie float32 ne nous permet pas d'utiliser la GPU à son maximum.

Au final, pour exécuter les applications sans le mode privée de la GPU, une stratégie de précision du type mixed_float16 est bien meilleure qu'une stratégie float32 au niveau des performances données par TensorBoard.

Utilisation de la stratégie de précision avec le mode privé de la GPU

Pour faire suite à notre analyse de performance de TensorFlow, nous continuons avec la dernière optimisation qui est l'application d'une stratégie de précision avec une GPU privée.

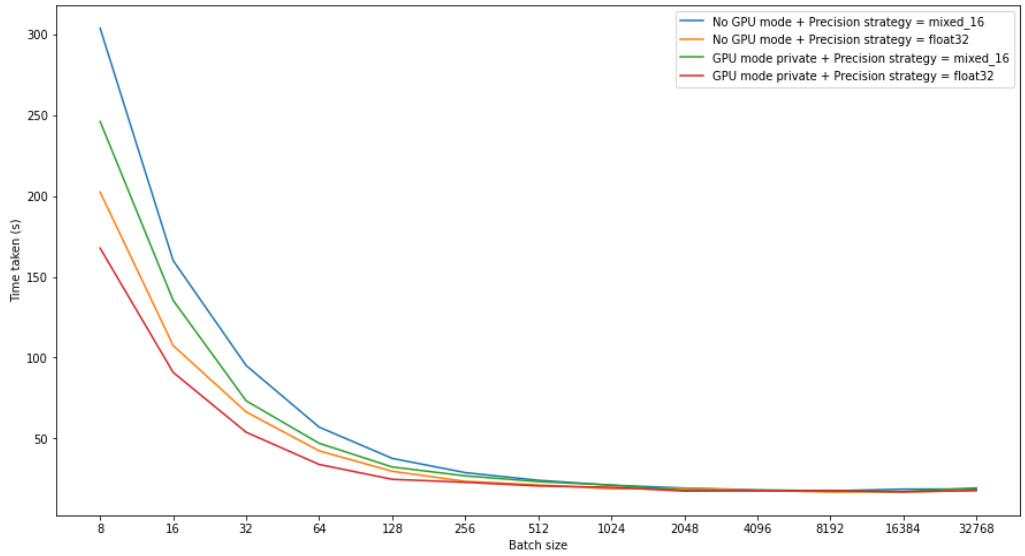

Figure 4.13: Comparaison de l'exécution de l'application simple cnn avec et sans GPU privée en utilisant des stratégies de précision

La figure 4.13 nous indique que la stratégie de précision du type float32 est la meilleure dans tous les cas (avec la GPU privée ou sans). En effet, les deux courbes rouge et orange ont les meilleurs temps d'exécution pour l'application **simple cnn**. Néanmoins, en utilisant le mode privé de la GPU, nous notons un temps meilleur pour n'importe quel type de stratégie.

Pour l'application **resnet**, le meilleur temps d'exécution revient à l'utilisation de la GPU en mode privée avec une stratégie de précision du type mixed float16. En effet, lors de

l'utilisation de la stratégie float32, les temps d'exécution sont très lent et donc non recommandés.

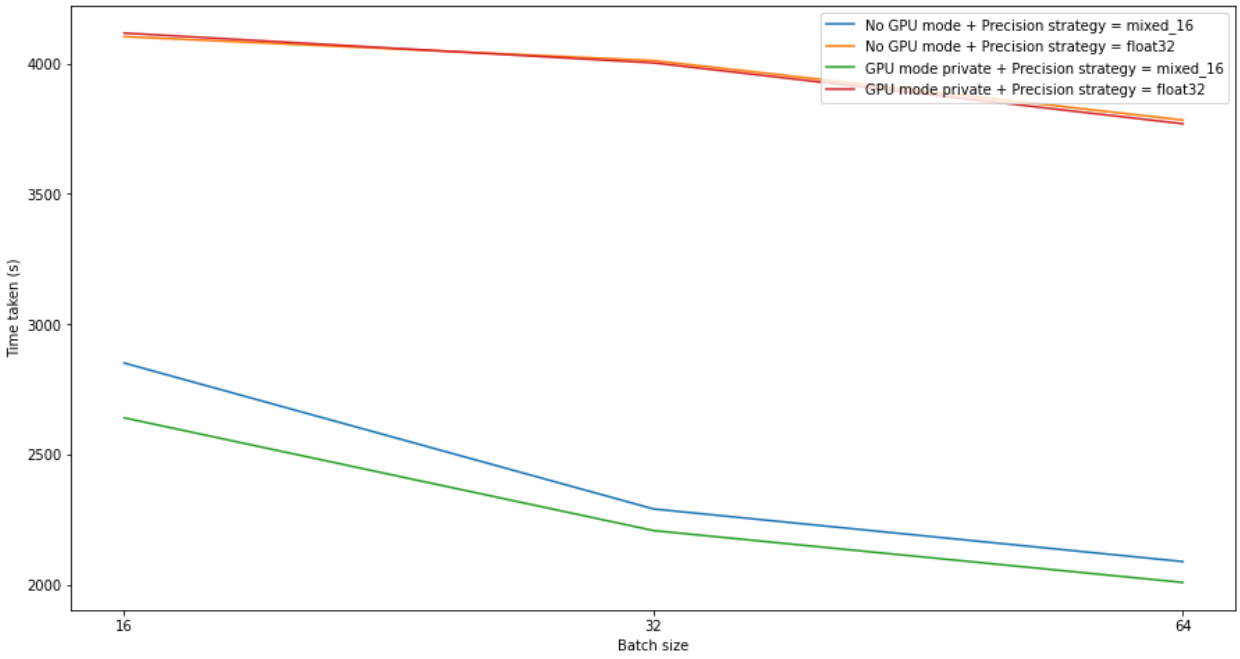

Figure 4.14: Comparaison de l'exécution de l'application resnet avec et sans GPU privée en utilisant des stratégies de précision

En utilisant TensorBoard, nous constatons que le temps que, pour l'application **simple cnn**, le CPU passe au lancement du GPU diminue considérablement en utilisant le mode privé de la GPU avec une stratégie de précision du type float32. De plus, nous observons une augmentation du temps de calcul sur la GPU qui, nous le rappelons, est le but.

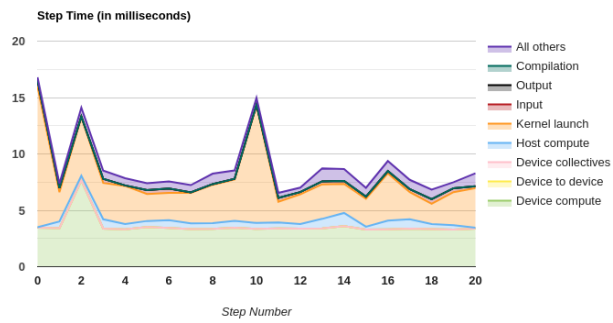

Figure 4.15(a): Stratégie mixed_float16 avec GPU privée

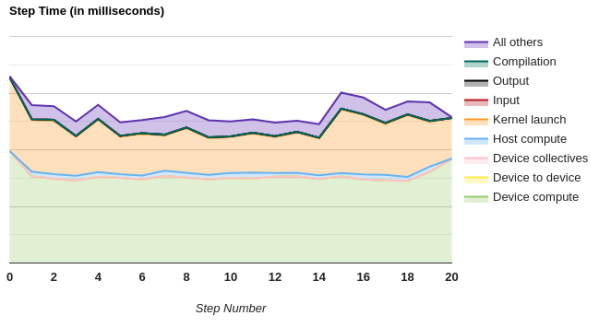

Figure 4.15(b): Stratégie float32 avec GPU privée

Figure 4.15: Comparaison entre les deux types de stratégies avec GPU privée pour l'application simple cnn

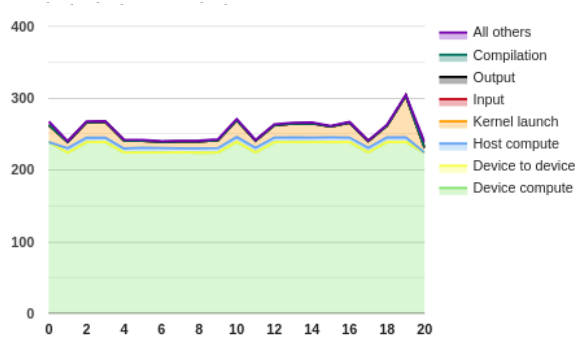

Figure 4.16(a): Stratégie mixed_float16 avec GPU privée

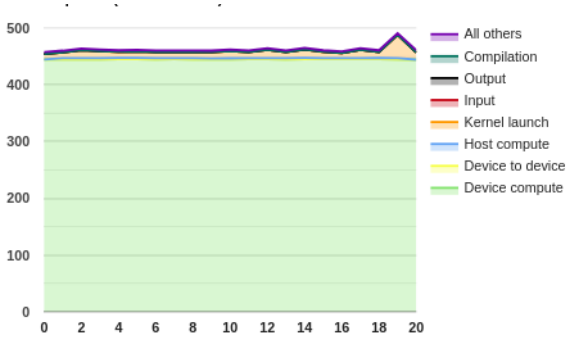

Figure 4.16(b): Stratégie float32 avec GPU privée

Figure 4.16: Comparaison entre les deux types de stratégies avec GPU privée pour l'application resnet

D'après les profils de la figure 4.16, nous constatons une augmentation du temps passé au kernel launch avec l'utilisation d'une stratégie mixed float16. Cependant, avec une stratégie float32, nous remarquons une augmentation du temps de calcul sur la GPU et une diminution du temps kernel launch.

Nous voyons donc que, pour un même batch size, les résultats retournés par Tensorboard valident le fait que passer en mode GPU privée permet d'améliorer les performances. Prenons l'exemple des résultats retournés par un batch size égal à 256 pour l'application **simple cnn**, représentés dans le tableau 5.5, et les résultats pour un batch size de 64 pour l'application **resnet** représentés dans le tableau 5.6.

Optimisations	**Average step-time (en ms)**	**Device Compute Time (en ms)**	**TF Op Placement -> Device (en %)**
Sans GPU privée + mixed_float16	8.7	3.5	91.9
GPU privée + mixed_float16	8.5	3.6	92.4
Sans GPU privée + float32	7.3	3.1	83
GPU privée + float32	5.4	3.1	86.6

Tableau 4.5: Résumé de performance de l'application simple cnn

Nous pouvons ici clairement observer l'impact du passage en mode GPU Privée. Pour un même type à virgule flottante, le temps moyen d'un pas diminue, le device compute time (le temps pendant lequel le GPU est occupé) augmente, ainsi que le pourcentage des opérations exécutées sur Device (GPU) par rapport au l'Host (CPU).

Optimisations	Average step-time (en ms)	Device Compute Time (en ms)	TF Op Placement -> Device (en %)
Sans GPU privée + mixed_float16	261.8	230.1	99.6
GPU privée + mixed_float16	255.8	231.8	99.6
Sans GPU privée + float32	431	443	99.6
GPU privée + float32	462.4	444.7	99.6

Tableau 4.6: Résumé de performance de l'application resnet

D'après le tableau ci-dessus, nous pouvons clairement observer l'impact du passage en stratégie de précision du type float16. Le temps moyen d'un pas diminue, le device compute time (le temps pendant lequel le GPU est occupé) augmente, ainsi que le temps de pas moyen qui baisse largement pour atteindre presque la moitié du temps donné pour la stratégie float32. Le passage au mode privé de la GPU nous permet gagner encore plus de temps par pas.

Lecture des données en avance

Durant le profilage de TensorFlow, nos applications passent 100% du temps à prétraiter les images de la base de données lors de l'entraînement, nous avons eu cette information grâce à TensorBoard qui nous fournit l'information de la répartition du temps de traitement des entrées sur la CPU. Ceci peut notamment causer des goulots d'étranglement puisque la GPU attend le traitement des images pour pouvoir calculer la précision de nos modèles comme le montre la figure 4.17.

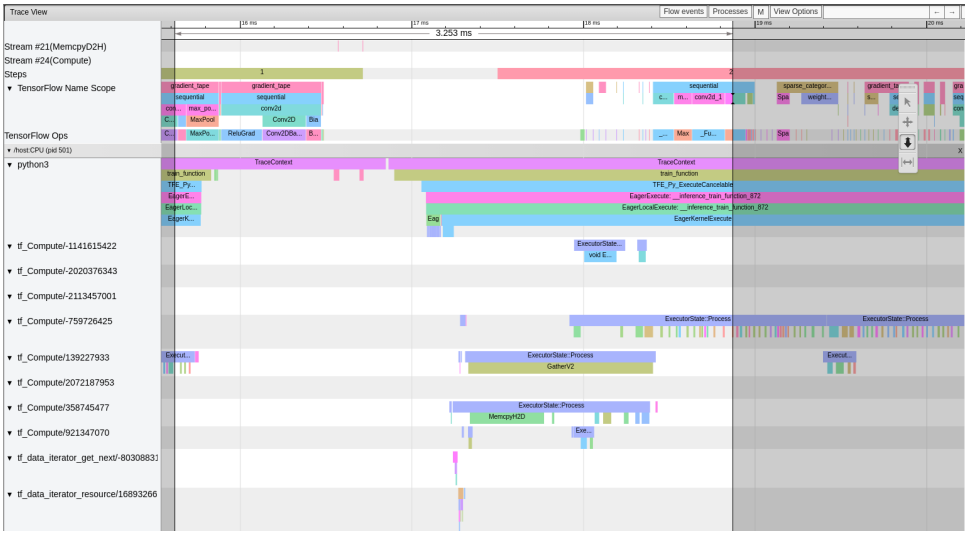

Figure 4.17: Repérage du goulot d'étranglement à l'aide de Trace Viewer de TensorBoard

Pour faire face à ce problème, nous faisons une prélecture de nos données avec la fonction **prefetch** de TensorFlow. Nous avons essayé cette technique avec l'application **simple cnn** et nous avons tracé le temps d'exécution de notre modèle avec une prélecture des images pour différents batch size et avec les optimisations que nous avons fait tout au long. Ce tracé est présenté par la figure 4.18 et 4.19.

Malheureusement, TensorBoard n'a pas voulu marcher avec cette prélecture des données afin de voir le profil de notre application avec cette nouvelle technique. Néanmoins, d'après la figure que nous voyons, nous pouvons clairement constater, et à notre grande surprise, que les temps d'exécution se sont détériorés par rapport à nos dernières optimisations sans prélire les images. Nous remarquons surtout une augmentation du temps lors de l'utilisation du mode privé de la GPU avec une stratégie de précision du type mixed float16.

Finalement, cette optimisation n'est pas bonne pour notre travail puisqu'elle nous donne une mauvaise performance et de mauvais résultats.

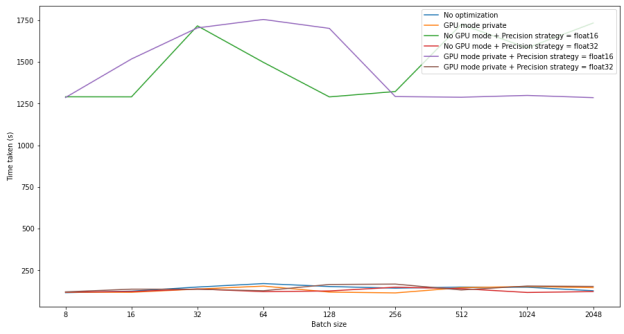

Figure 4.18: Comparaison de l'exécution de l'application simple cnn avec prélecture de données

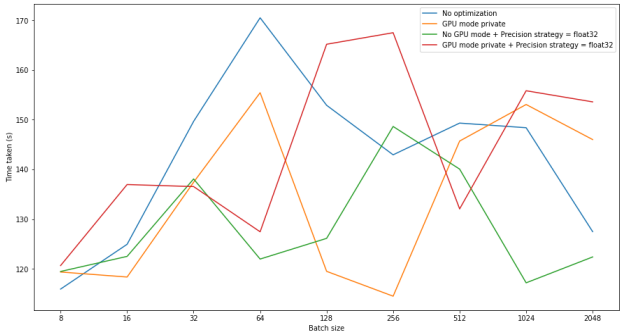

Figure 4.19: Comparaison des temps d'exécution avec prélecture des données sans la stratégie mixed float16

5. Synthétisation des résultats

Afin de compléter notre analyse de performance, nous consacrons cette partie à la synthétisation des résultats que nous avons obtenu suite aux multiples optimisations que nous avons essayé sur nos applications et en suivant les recommandations de TensorBoard.

5.1. L'application simple cnn

La figure 5.1 résume tous les temps d'exécution que nous avons eu suite aux optimisations que nous avons ajoutées à notre application **simple cnn**.

En premier lieu, nous constatons qu'il y a deux courbes qui se superposent à chaque fois. La première est le temps de l'application sans optimisation et l'ajout de la stratégie de précision en float32 sans la GPU privée. La deuxième est le temps de l'application avec les optimisations en mode privé de la GPU avec et sans stratégie de précision en float32. Ceci

alors nous confirme le fait que notre application utilise par défaut une stratégie de précision du type float32. En second lieu, nous remarquons qu'à partir d'un batch size de 1024, tous les temps d'exécution de l'application avec n'importe quelle optimisation ne diffèrent pas du tout les uns des autres.

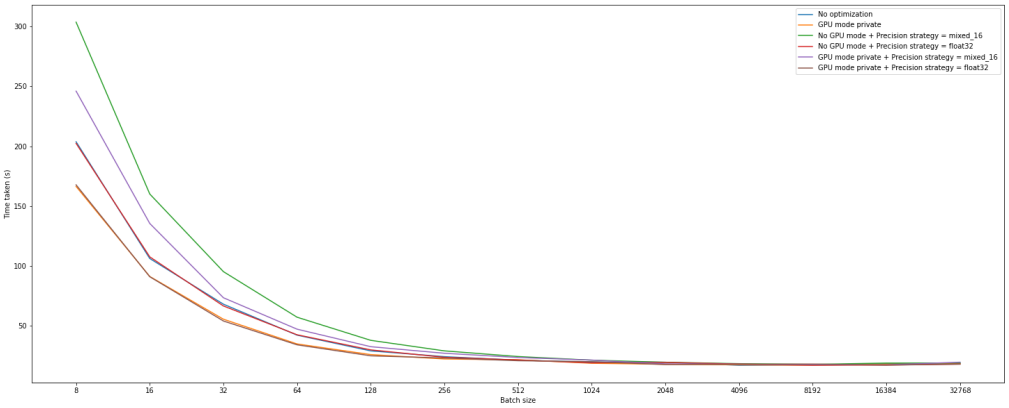

Figure 5.1: Comparaison des temps d'exécution de l'application simple cnn

Ci-dessous, nous avons un tableau qui résume les performances de notre application **simple cnn** avec les différentes optimisations. Nous remarquons que malgré un temps moyen par pas assez élevé, l'application avec une GPU privée et une stratégie mixte float16 est la meilleure d'entre toutes puisque ça nous permet d'utiliser presque 93% de la GPU qui est bien évidemment notre but. Avec ces optimisations, nous sommes sûrs que nous allons utiliser la GPU pour le calcul. Puisque TensorFlow repose essentiellement sur des méthodes de calculs, alors cette application est la meilleure pour notre cas au niveau des performances.

Optimisations	**Average step-time (en ms)**	**Device Compute Time (en ms)**	**TF Op Placement -> Device (en %)**
Sans GPU privée	6.1	3.2	85.7
Avec GPU privée	6.5	3.2	85.7
Sans GPU privée avec stratégie mixed_float16	8.7	3.5	91.9
GPU privée avec stratégie mixed_float16	8.5	**3.6**	**92.4**
sans GPU privée avec stratégie float32	7.3	3.1	83
GPU privée avec stratégie float32	**5.4**	3.1	86.6

Tableau 5.1: Résumé de performance de l'application simple cnn avec toutes les optimisations

5.2. L'application resnet

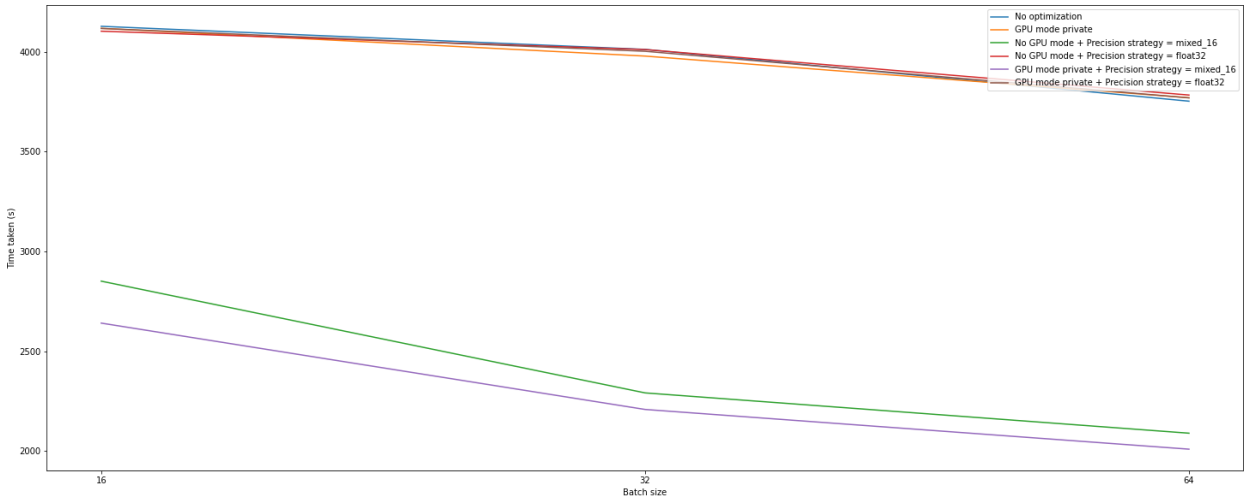

Figure 5.2: Comparaison des temps d'exécution de l'application resnet

Pour une application aussi lourde que resnet, nous observons que la meilleure stratégie à faire est de passer en float16, soit en utilisant une GPU privée ou pas. Néanmoins, l'utilisation d'une GPU privée nous fait gagner un peu de temps en exécution.

Optimisations	**Average step-time (en ms)**	**Device Compute Time (en ms)**	**TF Op Placement -> Device (en %)**
Sans GPU privée	461.6	442.8	99.6
Avec GPU privée	461.7	443	99.6
Sans GPU privée avec stratégie mixed_float16	261.8	**230.1**	99.6
GPU privée avec stratégie mixed_float16	**255.8**	231.8	99.6
sans GPU privée avec stratégie float32	461	443	99.6
GPU privée avec stratégie float32	462.4	444.7	99.6

Tableau 5.2: Résumé de performance de l'application resnet avec toutes les optimisations

Comme mentionné dans la fin de la section 4.3.2.2, la meilleure utilisation de l'application resnet est avec une stratégie de précision float16. Puisque ça nous donne les meilleures performances ainsi qu'un meilleur temps d'exécution.

Conclusion et perspectives

En guise de conclusion, ce projet fut pour nous une occasion d'approfondir nos connaissances de Tensorflow et de découvrir les outils d'analyse de performance ainsi que leurs rôles dans l'optimisation des projets Tensorflow, que ce soit en temps ou en utilisation des ressources mémoires (CPU et GPU).

La mise en place des deux modèles CNN simple et RESNET nous ont permis de mettre en valeur les outils d'analyse de performance, notamment Tensorboard, et d'observer l'impact des optimisations réalisées sur les performances des applications. La variation des différentes optimisations nous a mené à dégager des observations importantes. En ce qui concerne le modèle CNN simple, nous avons constaté qu' opter pour une stratégie mixte float16 permettait de tirer les meilleures performances du modèle, que ce soit en mode GPU privé ou pas. Une autre observation importante est que pour une stratégie mixte float16 le average step time augmente mais l'utilisation du GPU augmente aussi ce qui permet de combler l'augmentation du temps de pas moyen. En ce qui concerne le modèle RESNET, la différence est assez remarquable, le passage vers une stratégie GPU privée avec une stratégie mixte float16 permet de diviser le temps de pas moyen et le temps d'exécution total quasiment par deux par rapport au modèle basique sans aucune optimisation tout en laissant une utilisation optimale du GPU.

Quant au deuxième profiler abordé durant ce rapport qui est NVIDIA Profiler, et bien que outil d'analyse très intéressant au bas niveau, nous n'avons pas eu l'occasion de l'utiliser à sa juste valeur faute de temps, mais les résultats observés sur NVVP étaient cohérents avec nos analyses.

Finalement, nous avons pu accomplir une première partie de notre objectif durant ce travail. Il reste alors un des objectifs phares du projet qui est de développer de nouvelles analyses, notamment en regardant ce qu'il se passe à bas niveau. Également, nous pouvons aussi continuer notre analyse de performance avec la prélecture des données surtout en essayant d'analyser le profil TensorBoard.