Artefact* : Évalution de l'impact énergétique du scheduling pour l'entraînement d'IA sur des CPU hétérogènes.

Andrew Mary Huet de Barochez, Stéphan Plassart, Sébastien Monnet

prénom.nom-composé@univ-smb.fr

Résumé
L'artéfact vise à évaluer la consommation énergétique de notre framework d'IA, nommé `DAHL`, et basé sur `StarPU`. Une référence est établie contre `Pytorch`, en les comparant sur un cas d'usage identique. Ensuite, différents schedulers de `StarPU` sont évalués, permettant d'indentifier des avantages et inconvénients pour chaque famille de schedulers. Également, une des contributions consiste à montrer que les CPU avec une topologie hétérogène demandent davantage d'attention à la configuration pour obtenir une exécution performante en temps et en énergie. Il faudra donc identifier la topologie du CPU de votre machine, puis la comparer avec vos voisins afin de constater les différence entre les topologies homogènes et hétérogènes !

Mots-clés : Nix, gros dataset à télécharger, pouvoir utiliser RAPL (CPU Intel), CPU hétérogène ?

---

1. Vue générale

1.1. Contributions

- C_1 Sur un CPU hétérogène, `Pytorch` ne s'adapte pas correctement à l'augmentation de la charge de travail : le temps d'exécution et potentiellement la consommation, ne sont plus linéaires.
- C_2 Sur-allouer le nombre de threads en `Pytorch` permet de pallier C_1.
- C_3 `DAHL`/`StarPU` montre de meilleures performances brutes, et s'adapte mieux à la charge.
- C_4 Les schedulers de la famille work-stealing donnent de meilleurs gains sur des machines homogènes. La famille eager semble plus adaptée à celles hétérogènes.

1.2. Artefact

La reproduction des résultats se base sur deux frameworks : `Pytorch`, qui sera instalable très facilement depuis l'environement Nix, et `DAHL`, notre framework qu'il faudra compiler. Les expériences dépendent ensuite d'un dataset à télécharger, puis qu'il faudra pre-processer, grace à un script bash. L'ensemble du code et scripts pour effectuer ces différentes étapes sont disponibles sur Codeberg : `https://codeberg.org/PhoqueEberlue/dahl/src/branch/compas`. Dernièrement, les résultats à reproduire se trouvent dans le papier lié à cette archive : `paper.pdf`. Le papier contient les résultas "officiels" sur des expérimentations de longue durée, avec des wattmètres, mais les fichiers `minimal-result-*.pdf` contiennent des exemples effectués en situation "Compas" que vous pourriez reproduire.

*. Format inspiré de SC : `https://github.com/jennfshr/sc26-repro/`

Artefact	Contributions	Éléments du papier
R_1	C_1, C_2	Figure 2
R_2	C_3	Figure 3
R_3	C_4	Figure 4

1.3. Détails des résultats

Résultat : R_1

Dans le cas d'un CPU homogène : on s'attend à ce que `pytorch-regular` et `pytorch-morethreads` ait des temps d'exécutions et des consommations similaires.
Dans le cas d'un CPU hétérogène : on s'attend à ce que `pytorch-morethreads` soit meilleur en consommation et en exécution que `pytorch-regular`.
Dans tous les cas : `pytorch-morethreads` utilise moins bien le CPU (%CPU usage), et `DAHL` offre de meilleures performances que `Pytorch`.

Résultat : R_2

Dans le cas d'un CPU homogène : les résultats sont plutôt similaires pour la plupart des ordonnanceurs. En, augmentant la taille des images, `random` se fait dépasser et la part de temps d'exécution ne fait qu'augmenter pour les autres schedulers.
Dans le cas d'un CPU hétérogène : on remarque de grande différences entre les schedulers sur les petites images, `ws`/`lws` donnent de moins bons résulats.
Dans tout les cas : `random` et `dmdar` sont moins bons en temps et en part d'exécution.

Résultat : R_3

Dans le cas d'un CPU homogène : les schedulers basés sur du work stealing ont les meilleurs gains.
Dans le cas d'un CPU hétérogène : les schedulers basés sur du eager ont les meilleurs gains.
Dans tous les cas : la baseline `random` doit arriver à être dépassée, `Pytorch` est moins bon que `DAHL` sur ce cas d'utilisation.

1.4. Temps de reproduction en minutes

Les expériences ont pris 4 jours complets pour tourner, évidemment ici nous n'aurons pas le temps. Nous pouvons donc jouer sur différents paramètres qui nous donnerons des résultats plus rapide, au dépit de courbes moins détaillées. Nous avons donc choisi un exemple minimal pour reproduire ces résultats. Il sera davantage détaillé dans la partie exécution.
Le temps humain pour la reproduction dépendera des pontentiels bugs. Le temps machine dépendra de la vitesse de téléchargement et de processing du dataset. La compilation prend également un peu de temps. De mon côté j'ai eu à peu près 5 minutes de compilation, 20 minute de téléchargement + processing, et 7 minutes pour faire tourner les expériences minimales.

2. Setup de l'artefact

Le setup se base sur l'utilisation du Nix pour aider à la reproductibilité. Cela devrait fonctionner sur tous les Linux et MacOS. Installation de Nix :

```
# Installer nix: https://nix.dev/install-nix.html
```

```
curl -L https://nixos.org/nix/install | sh -s -- --daemon

# Activer les flakes:
# Soit en passant ces flags aux commandes nix:
# --experimental-features 'nix-command flakes'
#
# Soit en faisant ce setup:
# https://wiki.nixos.org/wiki/Flakes#setup
# Qui consiste à ajouter:
# `experimental-features = nix-command flakes` au fichier
# .config/nix/nix.conf
```

On peut ensuite cloner le projet :

```
git clone ssh://git@codeberg.org/PhoqueEberlue/dahl.git
cd dahl
git checkout compas
nix develop

# Pour le lancer dans votre shell favoris:
nix develop --command fish
```

Vient une partie plutôt longue car nous devons télécharger le dataset :

```
cd datasets
./big_fashion_download.bash
```

Le script gère également le pre-processing du dataset. Il va en créer 10 différents avec images redimensionnées et de plus en plus petites.
Pendant le téléchargement, vous pouvez commencer à compiler `DAHL` :

```
cd dahl/dahl/

mkdir build
cd build
cmake ..
make

# Potentiellement lancer les tests:
make test
```

3. Exécution de l'artefact

Une fois tout le setup accompli l'exécution est très simple, mais longue. Nous faisons varier de deux dimensions : le framework (`Pytorch/DAHL`) ainsi que sa configuration (`Pytorch` simple, `Pytorch` more threads, les schedulers `StarPU`), et la taille du dataset utilisé ($26 \times 34 \times 3$ jusqu'à $234 \times 312 \times 3$).
Pour réduire le temps d'exécution, nous avons réduit de nombreux curseurs, le réduisant à 7 minutes sur mon pc portable. Il est possible d'augmenter, ou de diminuer ce temps dans

Paramètre	Valeur	Usage
CUSTOM_DATASETS	Un sur deux	Datasets utilisés
NUM_REPEATED_RUNS	1	Répetitions pour intervalle de confiance
NUM_SAMPLES	2048	Samples par epoch
NUM_EPOCHS	1	Nombre d'epoch
BATCH_SIZE	32	Taille des batchs
EXEC_MODES	Tous sauf WS/EAGER	Les frameworks et leurs configs

la configuration de l'expérience dans `experiments/workload/params.py`. Augmenter le nombre de datasets donnera des courbes avec plus de points. Le nombre de runs répétées donnera des intervalles de confiance. Quant aux nombre de sample et d'epoch, cela peut permettre de faire tourner les entraînement plus longtemps pour avoir plus de mesures d'énergie.
Dernièrement, vous trouverez des symboles `TODO-COMPAS` aux endroits où il est parfois nécessaire d'effectuer une modification pour coller à votre environnement. Par exemple, il faudra spécifier le hostname de votre machine et quelques autres informations utiles pour de l'affichage dans les plots.
On peut donc lancer les expériences, qui permettront des générer toutes les figures :

```
cd experiments

python3 -m workload.launch False

python3 -m workload.plot
```