TP Performance énergétique d'un code

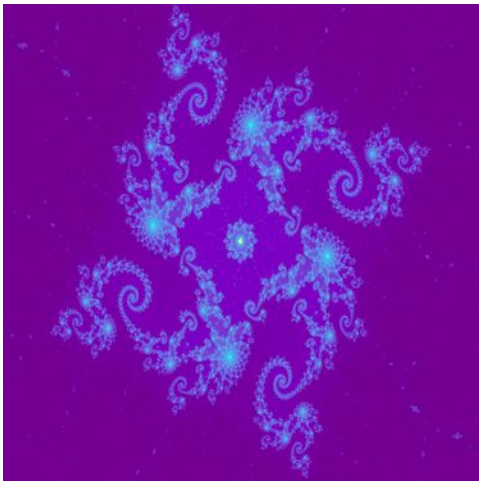
Gaël Guennebaud

TP Performance énergétique d'un code
A faire en bi- ou trinôme

Gaël Guennebaud (Inria Université de Bordeaux) - 2022

L'objectif de ce TP est d'explorer quelques leviers d'actions pour rendre un code de calcul plus efficace énergétiquement. Il s'agit d'un sujet extrêmement vaste et dans le cadre de ce TP nous explorerons que quelques aspects :

- Langage : python vs python compilé vs C++
- Parallélisme : code séquentiel vs SIMD vs multithreading vs pipelining



Pour cela nous prendront comme cas d'étude le calcul de l'ensemble de Mandelbrot qui, en quelques lignes de code, permet de générer des images captivantes.

Préambule

1. Vous travaillerez en binôme **sur deux PC** : un pour lire le sujet et reporter les mesures, et un second sur lequel vous effectuerez les mesures. Avant de démarrer ce second PC, débranché le et rebranché le sur une prise-wattmètre.
2. Sur le PC de mesure, utilisez un éditeur de texte « léger » comme *gedit*, et pas de « gros » éditeurs comme *vs code* (ou bien fermé le pendant les mesures).
3. Après démarrage, notez la puissance active en W et la puissance apparente en V.A observées. Cette puissance au repos est souvent noté P_{idle}.
4. Télécharger et décompresser l'archive.

Python

Nous allons commencer les mesures avec le script python mandelbrot.py :

```
$ python3 mandelbrot.py
```

Ce script affiche le temps de calcul de l'image, puis la stocke dans le fichier mandelbrot.jpg. Le temps d'écriture de l'image n'est pas comptabilisé car 1) nous n'avons pas la main sur cette étape, et 2) l'énergie absorbée lors de cette étape n'est pas homogène avec celle du calcul.

Ce premier « run » est assez long, c'est normal. Après avoir noté la puissance active, profitez en pour préparer un tableur libreoffice dans lequel vous noterez toutes vos mesures. Celui-ci doit posséder au moins 4 colonnes :

1. description de l'expérience, ex. « python, 2024, numba+parallel »
2. durée d'exécution en seconde
3. puissance active moyenne observée pendant l'exécution en W
4. une formule calculant l'énergie absorbée en mWh
5. optionnel : une formule calculant les émissions de CO2 (en mg)

Après exécution, vérifiez que l'image produite, mandelbrot.jpg, correspond bien à l'image ci-dessus, et n'oubliez pas de reporter le temps d'exécution !

Numba

Pour la deuxième expérimentation, nous allons tester de compiler la fonction mandelbrot via numba qui un compilateur JIT (« just in time »)[1]. Pour cela, il vaut faut ajouter :

1. `from numba import jit,prange`
2. et le décorateur `@jit(nopython=True)` juste avant la définition de la fonction mandelbrot

Lancez une mesure et comparer la **consommation** énergétique.

Pour la suite nous allons augmenter la difficulté en passant à des images de 2048x2048 (remplacez la ligne n=256 de la fonction main).

Relancez une mesure (cela peut prendre plus d'une minute).

Multi-threading

La version précédente est nettement plus rapide et moins énergivore, mais elle n'exploite qu'un seul des coeurs de votre CPU. En affichant le contenu du fichier `/proc/cpuinfo`, trouvez combien de coeurs **physiques** possède votre CPU (tips : ce fichier liste les coeurs virtuels, pour connaître le nombre de coeurs réels, faites une recherche sur internet à partir du modèle).

Remarquez que chaque pixel de l'image peut être calculée indépendamment des autres, nous pouvons donc faire calculer chaque ligne de l'image par un coeur différent. Nous pouvons demander à Numba de faire ce travail pour nous :

1. Ajoutez l'option `parallel=True` au décorateur `@jit`, c'est à dire : `@jit(nopython=True,parallel=True)`
2. et demandez explicitement à Numba de paralléliser la boucle sur les lignes en remplaçant `range` par `prange` (ici le préfix « p » est pour *parallèle*)

Relancez une mesure. Le calcul est effectivement plus rapide sur plusieurs coeurs, mais l'énergie absorbée est-elle aussi plus importante ? Qu'en est il coté consommation énergétique ?

C++

Peut-on faire mieux avec un langage compilé comme C++ ? Pour vérifier cela, nous allons commencer par une version équivalente au code python et compilé sans parallélisation :

```
$ g++ -I . -I eigen -std=c++17 -O3 mandelbrot.cpp -o mandelbrot
$ ./mandelbrot
```

Comme Numba, le compilateur clang est basé sur LLVM pour la génération du code. Comparez les temps de calcul et la consommation énergétique avec la version équivalente de python+Numba. Vous noterez également le nombre moyen d'itérations effectuées par pixel.

Multithreading

Pour tester le multi-threading :

1. Ajoutez la directive `#pragma omp parallel for` avant la boucle à paralléliser (ici la boucle sur les colonnes `for(int i = 0; i < n; ++i)`
2. Ajoutez l'option de compilation `-fopenmp` à la ligne de commande

Faites une mesure et comparez à la mesure précédente.

Tips : pour faciliter la lecture de la puissance, vous pouvez allonger la durée d'exécution en recalculant plusieurs fois la même image, par exemple 4 fois avec :

1 En fait, il s'agit plus d'une interface à LLVM

```
$ ./mandelbrot 2048 4
```

Le premier argument correspond à la taille de l'image : il est important de ne pas la changer car il est impossible de comparer les temps de calculs de deux images de Mandelbrot différentes.

SIMD

Cette version sous-exploite encore grandement les capacités de calcul de nos CPU dont chacun des coeurs est capable de réaliser une même opération sur plusieurs données en même temps. Ce type de parallélisation s'appel SIMD (Single Instruction Multiple Data). Par exemple, vos CPUs sont capable de réaliser 4 opérations flottantes double précision en même temps via le jeu d'instruction AVX. On parle de jeux d'instructions « vectorielles ». Dans certains cas, le compilateur est capable de « vectoriser » automatiquement certaines boucles de calcul. Ce n'est pas le cas ici. Nous avons donc du écrire une version spécifique de la fonction mandelbrot calculant 2, 4, 8 ou plus de pixels en même temps. Il s'agit de la fonction `mandelbrot_simd` du fichier du même nom. Testez cette version. Il faudra en plus ajouter les options `-mavx -mfma` au compilateur pour exploiter au mieux vos CPUs.

Faites deux mesures de cette version : une sans multithreading (sans -`fopenmp`) et une avec.

Pipelining

Le *pipelining* est un parallélisme au niveau des instructions inspiré du travaille à la chaine. Sur les CPUs x86-64 actuels, une opération élémentaire nécessite 4 cycles pour être exécuté. Si il y a suffisamment d'instructions **indépendantes** à traiter, ces CPUs sont capables de les traiter à la chaine de telle sorte qu'à chaque cycle une instruction démarre et une autre se termine. Dans ce but, la version précédente de Mandelbrot est capable de traiter plusieurs colonnes de l'image en même temps, par exemple 4. Pour cela, il vous faut remplacer le 1 par 4 dans l'appel de la fonction mandelbrot_simd.

Faites deux mesures de cette version : une sans multithreading (sans -`fopenmp`) et une avec (les deux mesures seront faite avec `-mavx -mfma`).

Il est temps de tirer quelques conclusions. Du point de vu de la **consommation énergétique** :

- Quelle version est la meilleure ?
- Une version plus rapide qu'une autre est-elle toujours **clairement** la meilleure ? Ou, autrement dit, peut on dire qu'il est suffisant d'optimiser de temps d'exécution pour optimiser la consommation énergétique ?
- Entre optimiser l'utilisation d'un seul coeur (SIMD+pipelining) ou simplement utiliser le multi-threading seul, quelle option est la plus intéressante ?

Et les coûts de compilation ?

Dans les expérimentations précédentes nous n'avons pas pris en compte le coût de la compilation de code C++. Dans un contexte où le binaire généré est utilisé de très nombreuse fois pour explorer les méandres de l'ensemble de Mandelbrot, ce coût est négligeable. Mais à partir de combien d'images calculées la meilleure solution C++ est moins gourmande en énergie que la meilleure solution python ? A vous de répondre à cette question en faisant la ou les mesures adéquates.

Tips : pour mesurer le temps de compilation vous pouvez utiliser la commande `time`, ex.:

```
$ time g++ …
```

GPU vs CPU ?

Les cartes graphiques actuelles sont équipées de GPUs permettant d'exécuter des centaines, voire quelques milliers, d'opérations flottantes en parallèle. Testons cela avec le code CUDA fourni avec ce TP (CUDA est un langage/compilateur développé par Nvidia pour ses propres GPUs qui est très proche du C/C++) :

```
$ nvcc -I . -I eigen -O3 mandelbrot.cu -o mandelbrot_cuda
$ ./mandelbrot_cuda
```

Pour votre configuration matériel, où ce situe cette option GPU en terme de temps de calcul et de consommation énergétique ?

Pour aller plus loin

1. Pourquoi, avec les versions SIMD, le nombre moyen d'itérations par pixel est-il plus élevé ?
2. Afin de trouver le meilleur compromis énergétique, quel(s) autre(s) paramètre(s) serait-il intéressant d'étudier ?
3. Vous pouvez évaluer l'effet du *pipelining* seul (sans SIMD) en compilant `mandelbrot_simd` avec `-DEIGEN_DONT_VECTORIZE` .

Mode d'emploi des wattmètres

Mode d'emploi wattmètre bluetooth Voltcraft SEM6000
Cet appareil est accompagné de son manuel que vous veillerez à rendre dans la boîte.

1) Télécharger et installer l'application Voltcraft SEM6000 sur votre smartphone
2) Brancher le wattmètre dans une prise
3) Allumez le bluetooth sur votre téléphone
4) Ouvrir l'application ; Autoriser l'accès à la localisation ;
5) Dans les paramètres (roue en haut à droite), vous pouvez changer la langue
6) Si cela ne se fait pas automatiquement, cliquer sur « trouver les nouveaux appareils »
7) Votre wattmètre doit apparaître dans la liste ;
    a. Le bouton power permet d'allumer où éteindre la prise – Tester le
    b. Suppression de l'historique
        1. Appuyer sur crayon en haut à gauche qui permet d'accéder aux paramètres,
        2. Cliquer sur le wattmètre dans la liste
        3. Puis sur « réinitialiser les paramètres par défaut » et « effacer l'historique de consommation »
        4. Retourner à la liste avec la touche « < » en haut à gauche puis appuyer de nouveau sur le crayon
    c. Visualiser la consommation
        1. Cliquer sur la flèche « > » à droite du wattmètre dans la liste
        2. Observez les valeurs de puissance et d'énergie

Mode d'emploi wattmètre Voltcraft 4000F
Cet appareil vous est fourni avec une carte SD et une pile déjà mis en place. Merci de les rendre comme ils vous ont été remis. Il est aussi accompagné du manuel que vous veillerez à rendre dans la boîte.

1) Branchez le wattmètre dans une prise.
   Remarque : Les pré-réglages du format de l'heure et de l'heure ont déjà été effectué sur l'appareil. Si vous observez lors de vos expérimentations que ces réglages ont été modifié, nous vous renvoyons vers le manuel (section 10 page 9) pour le re-régler.
2) Vous pouvez commencer les mesures. La touche MODE vous permet de naviguer entre différentes valeurs. Testez et observez les valeurs de puissance et d'énergie. Pour plus d'information, vous pouvez vous référer au manuel (section 11, pages 10-11).
3) Une fois la mesure de consommation d'un appareil réalisé, transférez les données sur la carte SD en appuyant sur la touche CONTINUER (triangle en bas). Si la valeur MEM est à 99, le transfert est terminé, laissez la carte encore 10 secondes avant de la retirer.
   Remarque : les mesures reprennent après retrait de la carte

Lecture des données de la carte :
Merci de laisser vos fichiers .BIN sur la carte et de ne pas effacer ceux existant.

1) Insérez la carte dans un lecteur de carte. Copiez les fichiers .BIN correspondant à vos mesures (observables avec l'heure du fichier) dans un dossier travail (ex : wattmetre)
2) Téléchargez depuis Moodle l'archive *el4000.zip* dans ce dossier
3) Depuis un terminal, lancez la commande et observez les résultats
   `python el4000.py –p csv FICHIER.BIN`