Projet PPAR : Parallélisation de la FFT

Romain CAPRON, Yannick ZHANG - MAIN4

1 Parallélisation avec OpenMP (`fft_omp.c`)

1.1 Démarche

Afin d'optimiser efficacement le code, nous avons utilisé l'outil de profilage `gprof` pour obtenir des informations détaillées sur le temps d'exécution de chaque fonction. Pour ce faire, l'option `-pg` a été ajoutée lors de la compilation. Puis après l'éxécution du code, la commande suivante a été utilisée pour enregistrer les résultats d'analyse :

```
gprof nom_du_executable > analysis.txt
```

Les résultats du profilage avec size = 67108864 ont montré que la fonction ont montré que que la fonction `PRF` monopolisait environ $43,63\%$ du temps d'exécution total, ce qui équivaut à un temps effectif de $21,85$ secondes. De manière similaire, la fonction `FFT_rec` représentait $38,43\%$ du temps d'exécution, soit $19,25$ secondes. Par ailleurs, la fonction `__muldc3`, essentielle dans le calcul des nombres complexes, constituait $10,52\%$ du temps d'exécution total, équivalant à $5,27$ secondes.

Ces résultats ont orienté notre concentration vers l'optimisation de ces fonctions, tout en parallélisant les boucles simples qui n'impactaient pas négativement les performances globales du code.

1.2 Exécution et Mesure des performances

Afin de mesurer les performances optimales, l'option `-pg` a été retirée pour ces mesures, car elle aurait pu augmenter le temps d'exécution. Après avoir effectué un `make`, le code parallèle a été exécuté pour différents nombres de coeurs en utilisant la commande suivante :

```
make run TARGET=fft_omp THREADS=x
```

Pour chaque configuration de coeurs, le code a été exécuté cinq fois de manière consécutive. Et entre chaque série de cinq exécutions, nous avons laissé la machine reposer pendant 10 minutes pour éviter la surchauffe et garantir des performances stables à chaque itération. Et pour eviter au maximum que des données déjà présentes dans le cache du processeur puissent fausser le temps d'execution. Pour l'ensemble des tests d'accélération et d'efficacité, nous avons utilisé un echantillon de size = 67108864. Pour les tests portés sur le weak-scaling, nous avons augmenté progressivement le nombre de coeurs de 1 à 8, (1, 2, 4, 8), en doublant à chaque fois la taille de l'échantillon (de 1048576 à 8388608). Cette progression est parfaitement proportionnelle pour une meilleure analyse.

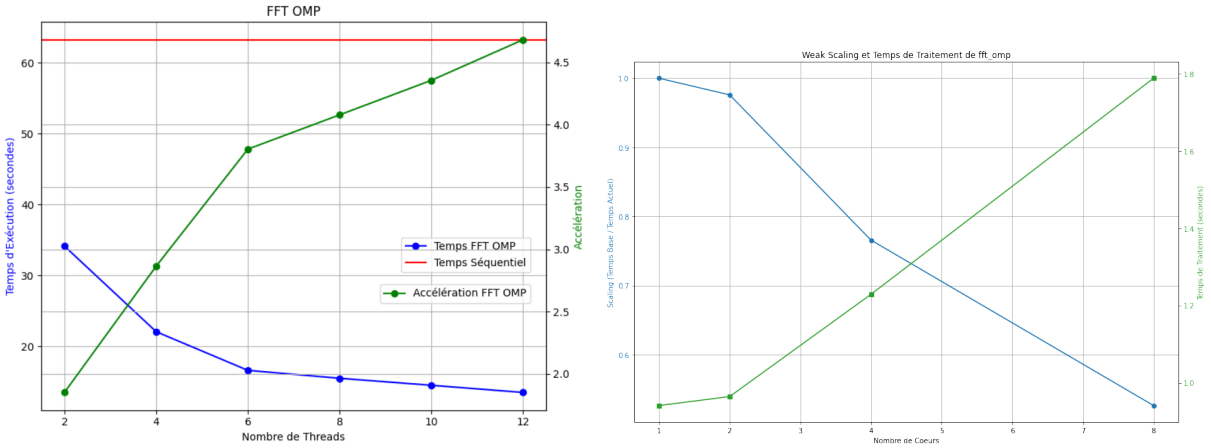
1.3 Commentaire sur les résultats

Vis à vis du **strong scaling**. Les résultats (figure 1(a)) révèlent une amélioration significative des performances grâce à la parallélisation du code FFT avec OpenMP. Initialement, le temps d'exécution moyen du code séquentiel était de 63.0561 secondes. Avec

l'application de la parallélisation, le temps moyen a chuté à 34.5482 secondes avec 2 coeurs, atteignant son minimum à 13.7107 secondes avec 12 coeurs.

Notons qu'une réduction substantielle du temps d'exécution a été observée jusqu'à 6 coeurs, prouvant l'efficacité de la parallélisation. Cependant, au-delà de ce nombre, bien que le temps continue de diminuer, le coefficient directeur de l'accélération diminue, suggérant une diminution des gains d'efficacité avec l'ajout de coeurs supplémentaires. On pourrait parler de mauvais **strong scaling** une pour plus de 6 coeurs ici.

On observe également, vis à vis du **weak scaling** (figure 1(b)), que l'on n'obtient pas une droite de coefficient directeur nul. Donc l'efficacité du code diminue lorsqu'on multiplie par 2 la taille des données, même en multipliant par 2 le nombre de coeurs.



(a) Évolution de la moyenne des temps et de l'accélération selon le nombre de coeurs

(b) Efficacité et analyse du Weak-Scaling - FFT OMP

FIGURE 1 – Analyse de `(fft_omp.c)`

2 Vectorisation `(fft_vec.c)`

2.1 Démarche

Les mêmes techniques de profilage ont été appliquées au code parallélisé Les résultats ont montré que les temps ont été significativement réduit mais les proportions du temps qu'ils occupaient sont rester pratiquement les même. Ainsi, bien que chaque fonction ait bénéficié de la parallélisation, la structure globale du temps d'exécution demeure similaire à celle observée dans le code séquentiel.

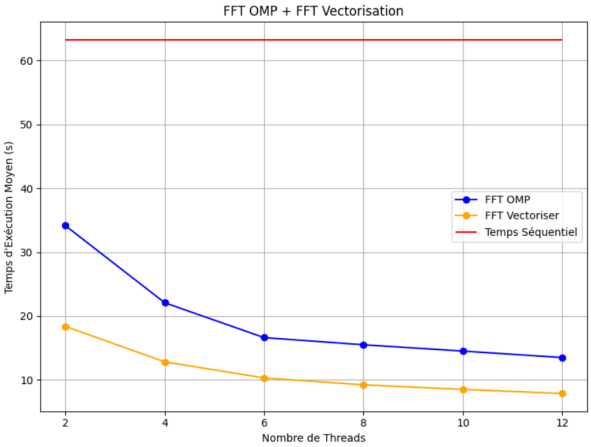
Pour explorer davantage les possibilités d'optimisation, nous avons ensuite appliqué la technique de vectorisation au code, permettant des calculs sur plusieurs éléments de données simultanément. Dans cette phase d'optimisation, des commandes telles que `pragma omp simd` ont été introduites pour indiquer au compilateur d'OpenMP de générer du code SIMD. Parallèlement, des instructions intrinsèques SIMD (AVX2 et NEON) ont été intégrées au code pour spécifier des opérations vectorielles spécifiques. Et des directives préprocesseur `#if defined` ont été utilisées avec afin de détecter l'architecture cible et d'appliquer les instructions SIMD correspondantes. Ainsi, le code s'ajuste automatiquement à l'architecture du processeur de l'utilisateur, que ce soit AVX2 pour l'architecture x86 ou NEON pour l'architecture ARM. Les graphes pour architecture ARM sont en annexe, compte tenu du code qui diffère (NEON), mais aussi de l'architecture du Macbook M1 qui diffère par sa mémoire et ses coeurs haute efficience/haute performance.

2.2 Exécution et Mesure des performances (avec size = 67108864)

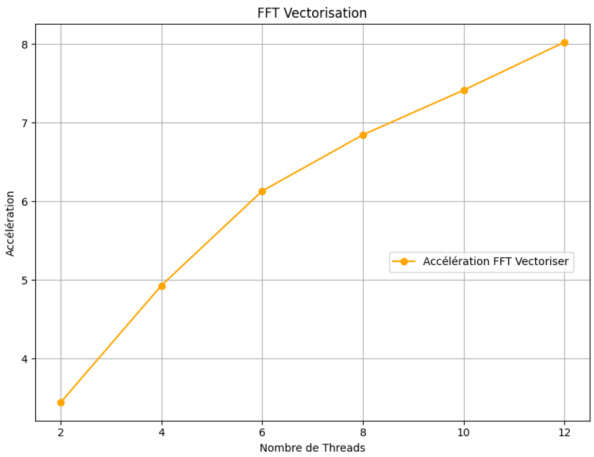
Après avoir préalablement adapté le `Makefile` pour compiler le code selon l'architecture de la machine (utilisant les options `-mavx2 -mfma` pour x86 ou `-march=armv8-a+simd` pour ARM, voir le makefile), il suffit d'exécuter la commande suivante :

```
make run TARGET=fft_vec THREADS=x
```

En choissisant x comme dit précédemmment pour effectuer les même test voici nos résultats.



(a) Comparaison des Temps d'exécution entre le code vectorisé et les autres



(b) Évolution de l'accélération

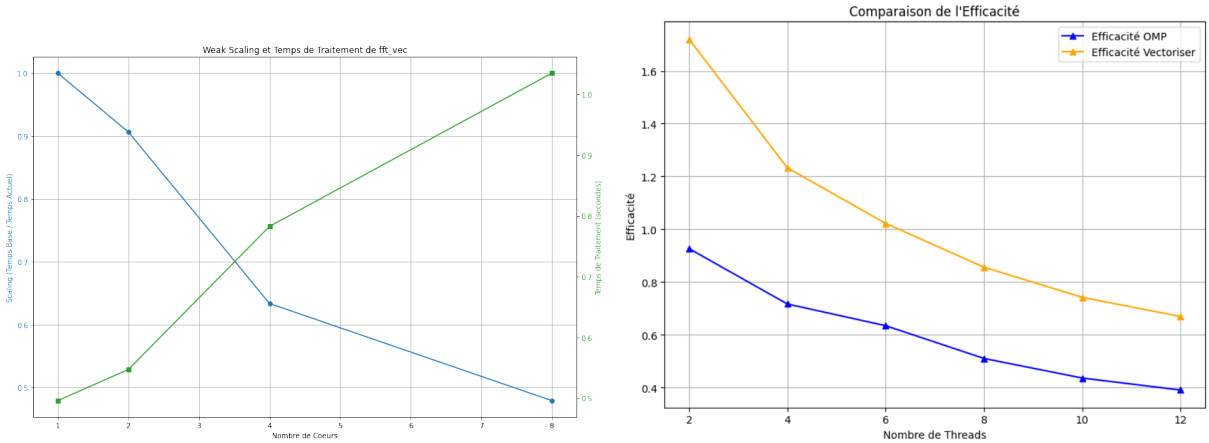
FIGURE 2 – Analyse de `(fft_vec.c)`

2.3 Commentaire sur les résultats

Vis à vis du **strong scaling**. La parallélisation avait déjà conduit à une réduction considérable du temps d'exécution par rapport au code séquentiel. Alors que le code séquentiel prenait en moyenne 63 secondes, l'introduction de la parallélisation a réduit ce temps à environ 14 secondes avec 6 coeurs, ce qui représente une accélération notable. Cependant, il est important de noter que l'accélération obtenue par la parallélisation n'était pas linéaire, surtout au-delà de 6 coeurs , où nous avons observé une diminution des gains de performance. Cette observation suggère que bien que la parallélisation soit efficace, elle atteint un point de saturation en raison des limitations matérielles et de la gestion des ressources parallèles. On a donc encore une limitation du **strong scaling**. Peut-être dûe à l'utilisation du "multi-threading" au delà de 6 coeurs. (Voir les configurations matérielles dans la partie 3.1)

Puis l'introduction de la vectorisation avec des instructions intrinsèques a apporté une amélioration supplémentaire des performances. Le code vectorisé a réduit le temps d'exécution moyen à environ 8 secondes avec 6 , surpassant ainsi les performances du code parallèle non vectorisé. Il est aussi intéressant de noter que la vectorisation a montré une tendance d'accélération plus constante avec l'augmentation du nombre de coeurs, contrairement à la parallélisation OpenMP.

Ensuite, on peut observer que lorsque l'on augmente progressivement la taille de l'echantillon et le nombre de coeurs, on n'obtient pas une droite de coefficient directeur nul, ce qui indique un **weak-scaling** imparfait. Pour 1 et 2 coeurs, les perfomances ne baissent pas trop, mais lors de l'utilisation de 4 puis 8 coeurs, on observe une chute des performances.



(a) Weak-scaling de fft_vec

(b) Comparaison de l'efficacité des codes

FIGURE 3 – Analyse de `(fft_vec.c)`, suite

Puis en examinant **l'efficacité** (figure 3(b)), on observe un début surprenant avec des valeurs supérieures à 1 pour le code vectorisé. Cela suggère que, comparé au code séquentiel de base, le code vectorisé exploite les ressources de calcul de manière remarquablement efficace. Cette efficacité supérieure à l'unité pourrait refléter un code séquentiel initial qui n'est pas complètement optimisé, laissant ainsi une marge significative d'amélioration lors de l'application de la vectorisation.

Toutefois, à mesure que le nombre de coeurs augmente, nous constatons une décroissance de l'efficacité pour les deux méthodes de parallélisation et de vectorisation. Cette baisse suggère que l'exploitation des ressources parallèles devient plus complexe avec un nombre croissant de coeurs. Cela met donc en évidence l'importance de trouver un équilibre optimal entre les performances brutes et une utilisation efficace des ressources. En choisissant judicieusement le nombre de coeurs, on peut maximiser l'efficacité globale du système, évitant ainsi les inefficacités liées à une gestion excessive des coeurs.

3 Analyse pour la Machine avec AVX2

3.1 Peak Performance

Pour déterminer la performance de pointe de notre système, nous avons fait le calcul suivant :

```
Peak Performance = Nombre de processeur x Frequence maximale
   par processeur x Operations par Cycle x Facteur FMA
```

Avec la commande `lscpu` nous avons obtenu les informations détaillées sur notre processeur, et les données pertinentes extraites sont les suivantes :

```
Architecture :                          x86_64
Processeur(s) :                         12
Nom de modele :                         Intel(R) Core(TM)
                                    i7-8750H CPU @ 2.20GHz
Thread(s) par coeur :                   2
Coeur(s) par socket :                   6
```

D'ailleurs, les instructions AVX2 permettent des opérations sur des registres de 256 bits. Pour les calculs en double précision (64 bits), cela signifie que chaque instruction AVX2 peut traiter 4 opérations simultanément. Et les processeurs qui supportent AVX2 supportent généralement aussi FMA, permettant d'effectuer une multiplication et une ad-

dition en une seule instruction, ce qui peut doubler le nombre d'opérations de virgule flottante par cycle d'horloge.

D'où notre Performance de Pointe = $12 \times 2.2 \times 4 \times 2 = 211.2$ GFLOPS

3.2 Estimation du Nombre d'opération

Nous avons procédé à un calcul manuel pour estimer le nombre d'opérations dans nos fonctions. Prenons, par exemple, la fonction `FFT_rec`, qui est au cœur de notre. Nous avons calculé le nombre d'opérations nécessaires en fonction de la taille de l'entrée n.

$$\text{Nombre total d'opérations pour FFT\_rec} = \sum_{k=1}^{\log_2(n)} \left(\frac{n}{2^k} \times \text{Nombre d'opérations par itération} \right)$$

Pour une taille d'entrée $n = 67108864$ et $n = 134217728$, les nombres totaux d'opérations pour `FFT_rec` et l'ensemble du code sont respectivement :

- Pour $n = 67108864$:
  - `FFT_rec` : 402653184 opérations
  - Ensemble du code : 10066329672 opérations
- Pour $n = 134217728$:
  - `FFT_rec` : 805306368 opérations
  - Ensemble du code : 19922944064 opérations

3.3 Observations et Limitations

Lors de l'exécution de notre code pour une taille d'entrée de $n = 134217728$, nous avons observé un arrêt prématuré, indépendamment de la version du code (séquentielle, parallélisée ou vectorisée). Ce comportement est indicatif d'un problème connu sous le nom de "mur de la mémoire" (**memory-bound**).

En effet, notre analyse a révélé que la performance théorique maximale de la machine est de 211.2 GFLOPS, tandis que le nombre total d'opérations effectuées par notre code pour cette taille d'entrée est de seulement 19.92 milliards d'opérations. Cette différence significative indique que le code n'exploite pas pleinement la capacité de calcul du processeur. Ainsi, il est peu probable que le problème rencontré soit un cas de limitation par la puissance de calcul (**compute-bound**).

En conséquence, il est raisonnable de supposer que le problème est plutôt lié à une contrainte de mémoire, connue sous le nom de "memory-bound". Ce phénomène pourrait être causé par une bande passante de mémoire insuffisante ou une saturation de la mémoire disponible. Cette limitation est particulièrement problématique pour les grandes tailles de données, où les besoins en mémoire excèdent les ressources disponibles, entraînant des échecs d'exécution ou une dégradation des performances.

4 Conclusion

En conclusion, ce projet a démontré que, même si la parallélisation et la vectorisation soient des stratégies puissantes pour améliorer les performances des algorithmes, elles doivent être appliquées en tenant compte des limitations matérielles, en particulier en ce qui concerne la mémoire. Une attention particulière devrait être accordée à la gestion de la mémoire et à la réduction de l'empreinte mémoire, afin de surmonter le "mur de la mémoire" et de se rapprocher davantage de la performance de pointe théorique des systèmes informatiques.

5 ANNEXE - Graphes des codes executés sur MAC M1 (et utilisation de NEON pour ARM)

En complément d'un code vectorisé à l'aide des protocoles AVX, nous avons fait une vectorisation à l'aide des protocoles NEON, afin que cela fonctionne sur les architectures de MAC M1 (ARM). Voici quelques graphes intéressants. (pour autant, nous avons choisit de baser nos analyses sur le code AVX, de plus, les resultats entre AVX et NEON semblent cohérents)

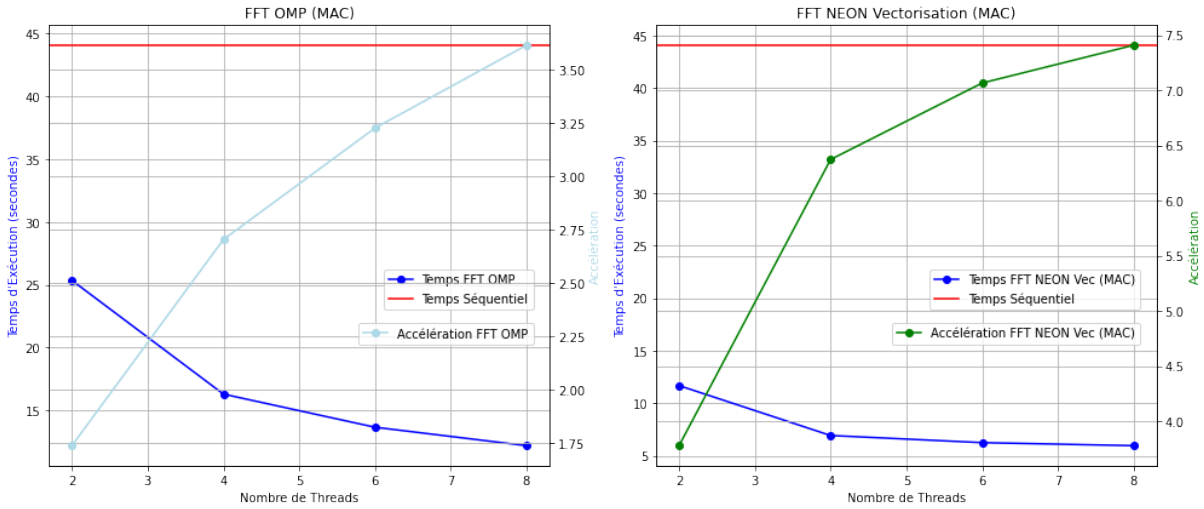


FIGURE 4 – Comparaison Temps d'execution et Accéleration - FFT_OMP/FFT_Vec (MAC)

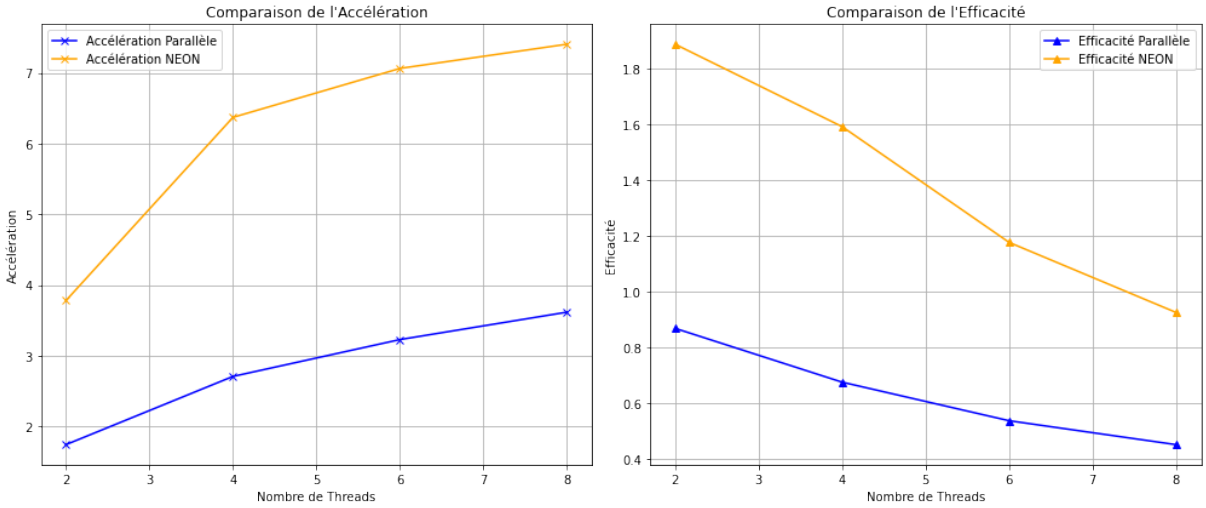


FIGURE 5 – Comparaison Accéleration et Efficacité - FFT_OMP/FFT_Vec (MAC)

L'accélération et l'efficacité sont bien meilleures dans le code **Vectorisé** en comparaison du code uniquement **Parallélisé**. Mais on observe dans les deux cas une efficacité qui décroit lorsqu'on augmente le nombre de coeurs, et une accélération dont le coefficient directeur diminue également.

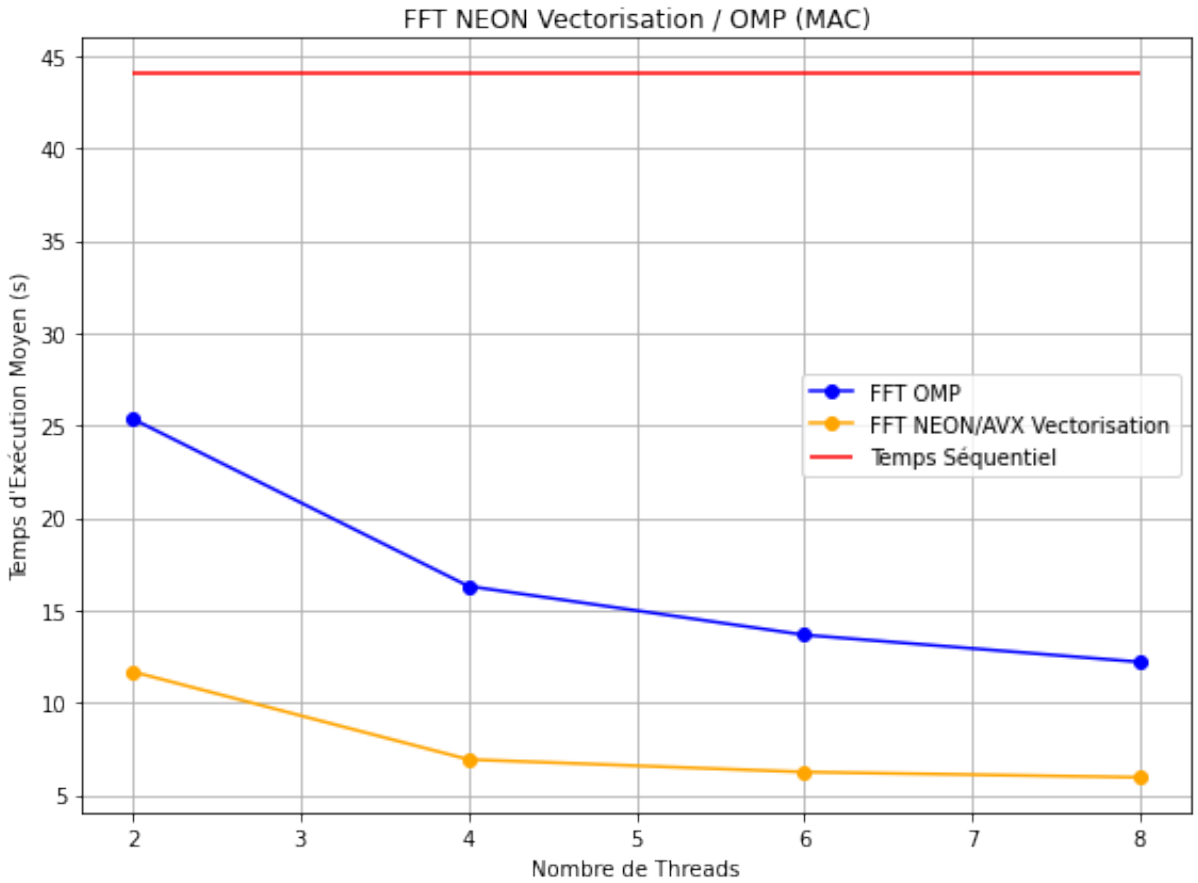
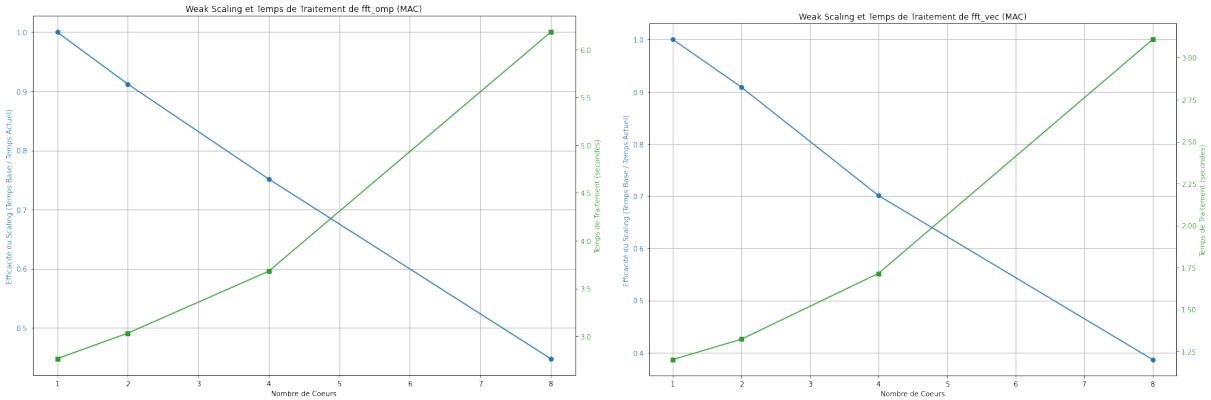


FIGURE 6 – Comparaison des temps d'execution (MAC)

Le weak-scaling a été fait selon ce protocole :
— 1 coeur : size = 4194304
— 2 coeurs : size = 8388608
— 4 coeurs : size = 16777216
— 8 coeurs : size = 33554432



(a) Weak-scaling FFT_OMP (b) Weak-Scaling FFT_Vec

FIGURE 7 – Weak-Scaling (MAC)