Optimisation de calcul parallèle
Calcul de l'attention

* *

Quentin Marret - Matthieu Olekhnovitch - Joris Chenevas

Introduction :

L'optimisation du calcul du mécanisme d'attention constitue un enjeu critique pour les modèles de langage qui alimentent les dernières technologies de NLP, et sa complexité quadratique en fonction de la longueur des séquences représente un bottleneck important lors de l'entraînement et de l'inférence. Une optimisation du calcul de ce mécanisme est donc primordiale pour optimiser le temps d'inférence/entraînement et les coûts.

Rappels des consignes :

Dans le cadre du cours *Programmation parallèle pour le Machine Learning,* nous avons cherché à effectuer un benchmark approfondi d'un certain nombre d'approches différents du calcul de l'attention.

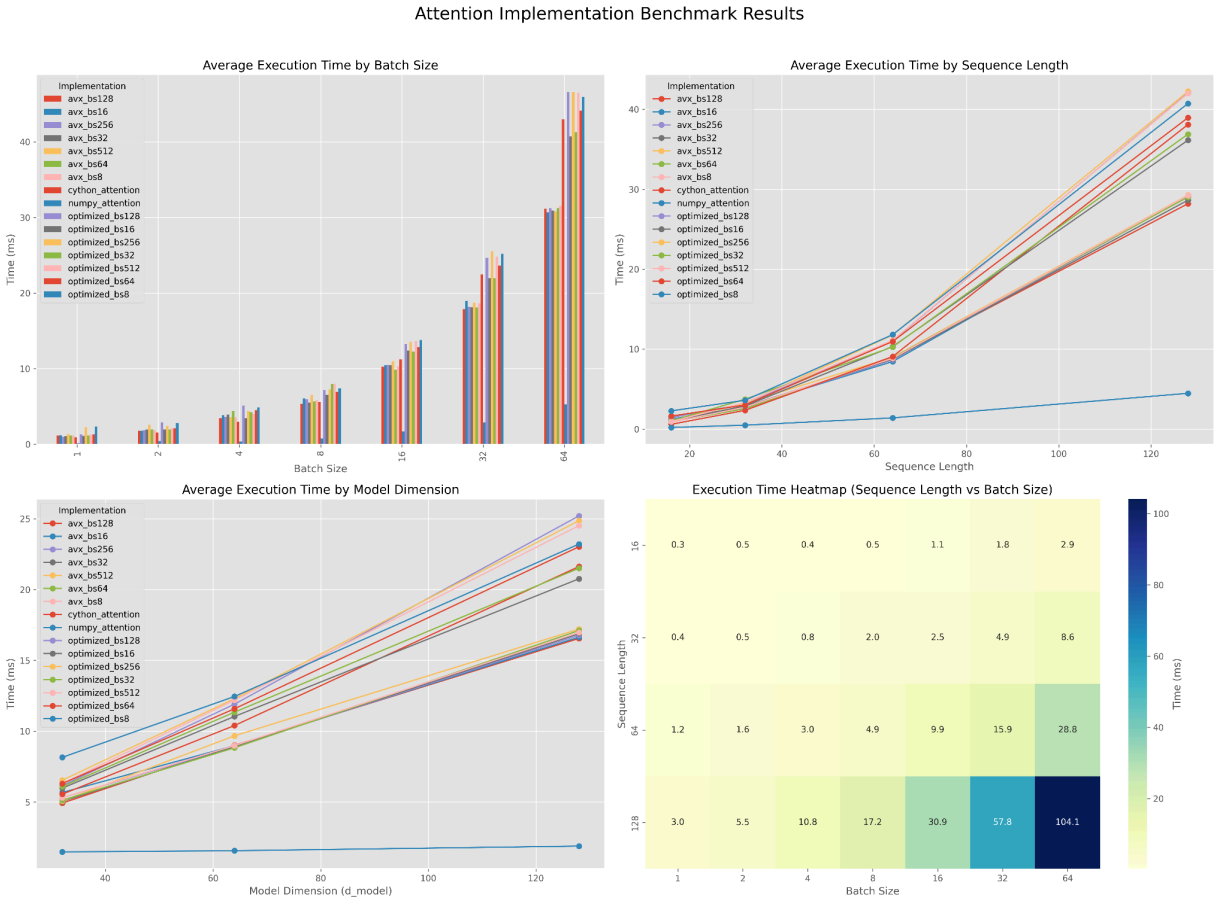
Parallèlement, nous avons cherché à l'optimiser le calcul d'une matrice d'attention en iditifient une taille de bloc optimal pour décomposer le calcul matriciel, sans passer par un système de recherche par grille consistant à tester plusieurs toutes les tailles de bloc pour identifier la plus rapide.

I. Comparatif global des approches (Naive Grid Search):

Nous avons d'abord implémenté quatre approches du mécanisme d'attention différents pour comparer leurs performances :

- numpy_attention sert de baseline optimisée utilisant les opérations vectorielles de NumPy
- cython_attention introduit un traitement par blocs qui utilise cython
- optimized_attention ajoute la parallélisation OpenMP et les optimisations de cache
- avx_attention tire parti des instructions SIMD AVX pour maximiser les performances.

Cette progression permet d'évaluer progressivement la performance. Nous avons construit le benchmark de facon à pouvoir tester autant de fonctions différentes que nécessaires tant qu'elles prennent en arguments Q, K et V les matrices de la formule d'attention. Dans l'approche de grille, nous définissons des block sizes différents pour les fonctions utilisant des blocs et effectuons 10 répetitions du calcul par set de paramètre + fonction. La version AVX semble apporter les meilleurs résultats, mais reste faible devant numpy en grande dimension. Le graphique *Attention Implementation Benchmark Results* ci-dessous propopse des visualisations des performances comparées des méthodes (Nous précisons que pour la heatmap, les temps d'executions sont averagés sur toutes les fonctions).



II. Code Cython pour le calcul de l'attention des matrices Q, K et V :

Tout d'abord, nous déclarons quand même la variable __AVX2__ afin que le compilateur puisse utiliser les instructions AVX afin d'optimiser le code lui-même.
Ensuite le code est divisé en trois grands parties :
- Première multiplication matricielle : On va tout d'abord multiplier les deux première matrice Q et K', en échangeant directement les lignes et colonnes de K à la volée pour utiliser directement sa transposée. On va aussi directement diviser le résultat obtenu pour chaque itération dans la boucle, par la racine carrée du nombre de colonnes de K. Enfin, on récupère aussi directement le maximum de chaque ligne, afin de stabiliser le futur softmax

- Calcul du softmax : Pour calculer le softmax de la matrice résultat du calcul précédent, on va parcourir les lignes deux fois. Une première fois afin de calculer l'exponentiel des valeurs (soustrait du max de la ligne pour stabiliser le calcul), et sommer la somme des exponentiels, et une deuxième fois pour diviser la valeur exponentielle par la somme des exponentielles de la ligne.

- Deuxième multiplication matricielle : Enfin, on calcule la multiplication matricielle du résultat du softmax, avec la matrice V. Pour cela, on parcourt donc les trois dimensions différentes des deux matrices.

On envoie ensuite la matrice résultat de la deuxième multiplication matricielle, qui est donc le résultat du calcul de l'attention des matrices Q, K et V.

III. Implémentation d'un Benchmark adaptatif permettant d'identifier la taille de bloc optimal pour réaliser le calcul de l'attention

Le calcul de l'attention sur une matrice de grande taille, typiquement 2048 x 2048, peut être optimisé sur les ordinateurs actuels en subdivisant la matrice en plusieurs sous-matrices de taille réduite. Cette division permet d'effectuer en parallèle les calculs sur ces sous-matrices, avant de recomposer la matrice finale à partir des résultats intermédiaires. Cependant, la subdivision de la matrice pour la parallélisation peut également introduire un coût supplémentaire en temps de calcul, notamment à cause de la gestion des ressources et des communications nécessaires entre sous-tâches. Étant donné que les performances parallèles dépendent fortement de l'architecture (os, processeurs, mémoires) spécifique à chaque ordinateur, la taille optimale des blocs peut varier d'un système à l'autre. Il est ainsi pertinent d'effectuer un benchmark personnalisé pour déterminer la taille optimale de blocs permettant le calcul le plus rapide de l'attention.

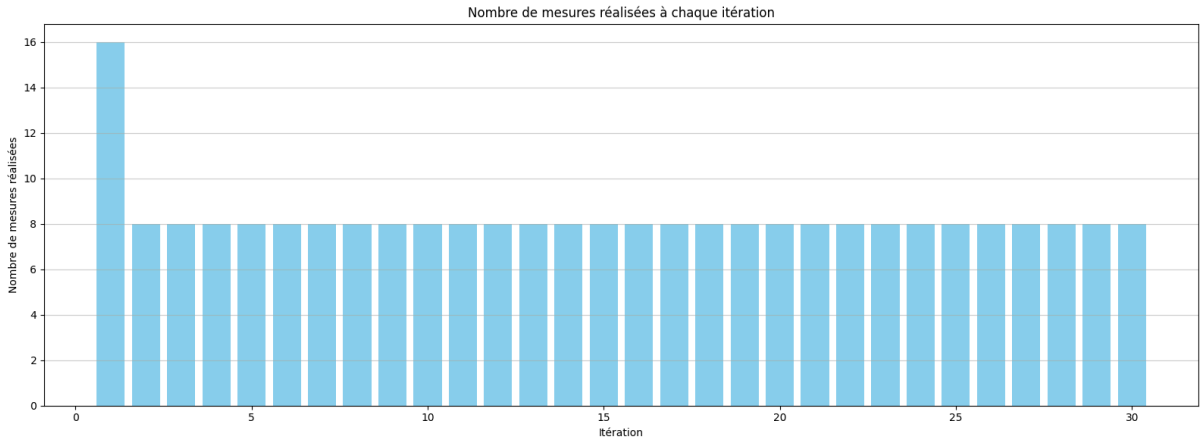
Dans cette partie, nous détaillons les stratégies de benchmarking utilisées dans nos notebooks :

- Benchmark Cython (calcul de l'attention implémenté en Cython)
- Benchmark Numpy (calcul de l'attention implémenté en Numpy)

Le benchmark par grille exhaustive évalue systématiquement chaque taille de bloc selon un schéma prédéfini. À chaque itération, toutes les tailles candidates sont mesurées plusieurs fois, permettant d'obtenir des statistiques robustes (moyennes et variances). Après chaque vague de mesures, l'algorithme identifie la meilleure taille de bloc sur la base d'un t-test avec un seuil de confiance de 95 %. Lorsque ce seuil est atteint, l'algorithme s'arrête et retourne la taille optimale. Bien que cette approche assure une exploration exhaustive et impartiale, elle peut nécessiter un grand nombre de mesures, en particulier quand le nombre de candidats ou le bruit des mesures est important (voir Annexe 1 pour davantage de résultats détaillés sur la grille exhaustive).

Le graphique associé montre clairement qu'à chaque itération, des mesures sont réalisées pour chacune des huit tailles de blocs testées ([64, 128, 192, 256, 320, 384, 448, 512]). Notons qu'au premier passage, deux mesures par taille de bloc sont réalisées pour pouvoir effectuer un test statistique fiable (basé sur l'écart-type, ce qui requiert au minimum deux valeurs).

Graph 1 : Mesure par itération - Grille exhausitive

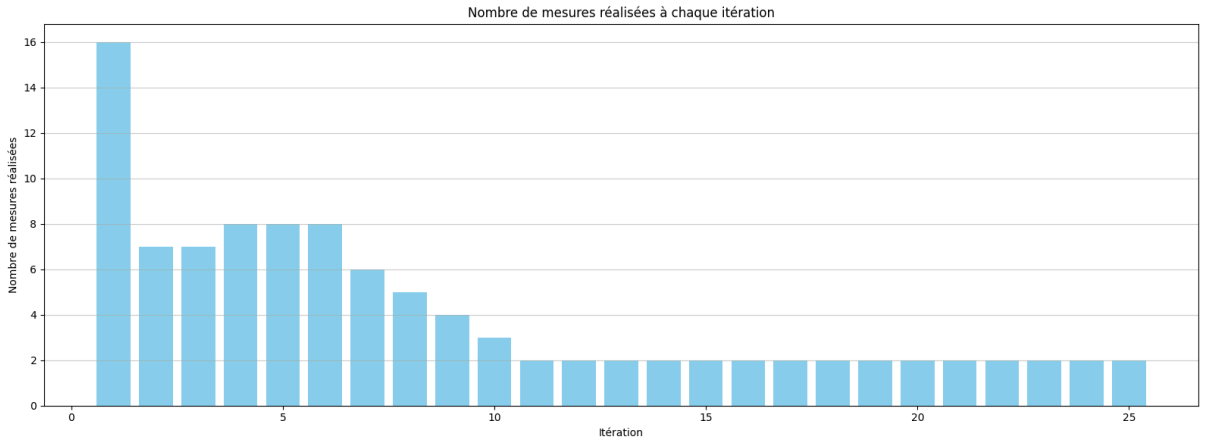


En revanche, le benchmark adaptatif probabiliste utilise une stratégie plus dynamique et flexible. Plutôt que de mesurer systématiquement toutes les tailles à chaque itération, il concentre rapidement ses mesures sur les tailles les plus prometteuses. À chaque étape, il estime, via un test statistique de Welch, la probabilité qu'une taille de bloc alternative ("challenger") soit réellement meilleure que la taille actuellement considérée comme optimale. Si cette probabilité dépasse un seuil prédéfini (paramètre `p_switch`), l'algorithme bascule sur cette nouvelle taille. Sinon, si aucune alternative n'est statistiquement significativement meilleure, il s'arrête. Cet algorithme

échantillonne principalement la meilleure taille actuelle et ses challengers plausibles. De plus, les paramètres `p_switch` et `extra_repeats_best` (nombre supplémentaire de mesures pour la meilleure taille courante) renforcent la robustesse du choix final en réduisant les changements dus à des fluctuations aléatoires. Bien que le paramètre `p_switch` doive être adapté au contexte spécifique du benchmark, on recommande généralement de fixer ce seuil entre 0.7 et 0.9. Nous avons choisi par défaut la valeur de 0.7 pour équilibrer prudence et rapidité de convergence.

Contrairement à l'algorithme par grille, le graphique associé au benchmark adaptatif montre que le nombre de mesures varie à chaque itération, diminuant progressivement au fur et à mesure que le nombre de tailles candidates plausibles décroît (voir Annexe 2 pour davantage de visualisations).

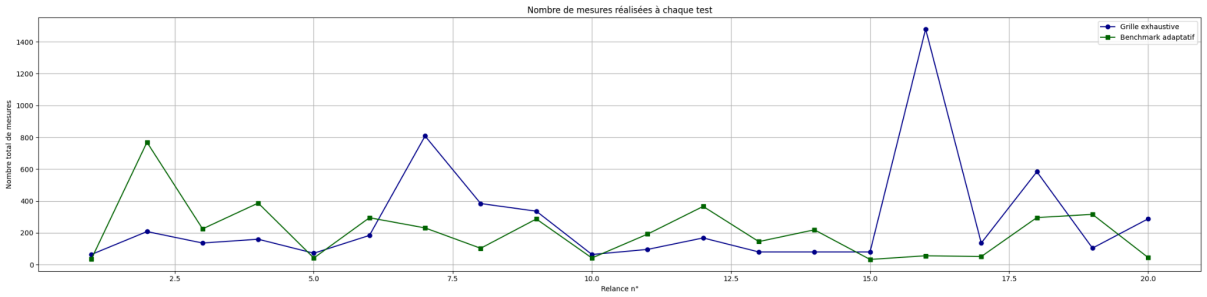
Graph 2 : Mesure par itération - Benchmark adaptatif



Ainsi, le benchmark adaptatif ne réévalue que les tailles de blocs ayant une probabilité supérieure à 5 % d'être meilleures que la taille actuelle optimale, contrairement à la grille exhaustive qui mesure systématiquement toutes les tailles à chaque itération. Ce recentrage rapide sur les tailles pertinentes permet une convergence plus rapide et efficace, notamment en présence de nombreux candidats ou d'un bruit élevé. C'est ce mécanisme adaptatif qui explique pourquoi, dans la majorité des cas, cette méthode atteint un résultat optimal en nécessitant beaucoup moins de mesures que la grille exhaustive, tout en garantissant une confiance statistique similaire.

En comparant ces deux méthodes sur 20 calculs distincts utilisant 20 matrices différentes, nous observons sur le graphique ci dessous que le benchmark adaptatif obtient presque systématiquement de meilleures performances en termes de rapidité d'identification de la taille optimale. Cependant, les deux approches convergent systématiquement vers la même taille optimale, à savoir 64, que ce soit pour l'implémentation Numpy ou Cython. Ce résultat récurrent de la taille optimale à 64 s'explique probablement par l'architecture des processeurs actuels, optimisée spécifiquement pour des calculs sur des blocs carrés de taille 64.

Graph 3 : Comparaison des 2 méthode “exhautive grid” et “adaptative benchmark” sur 20 tests

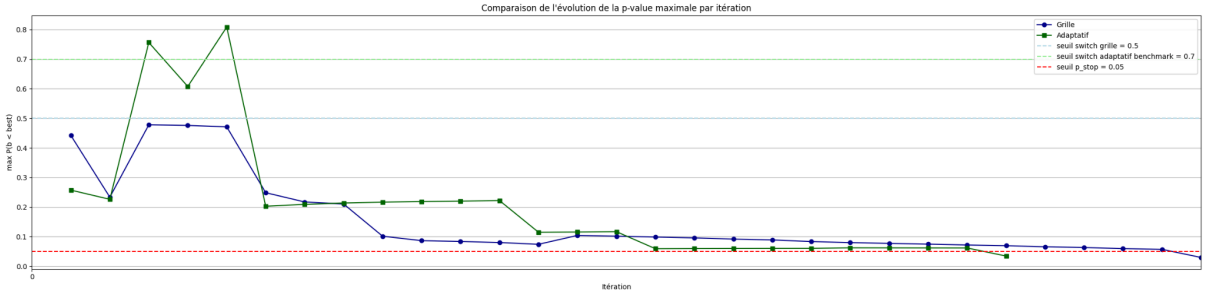


Annexe : Resultat Benchmark Grille exhaustive VS adaptavite Benchmark

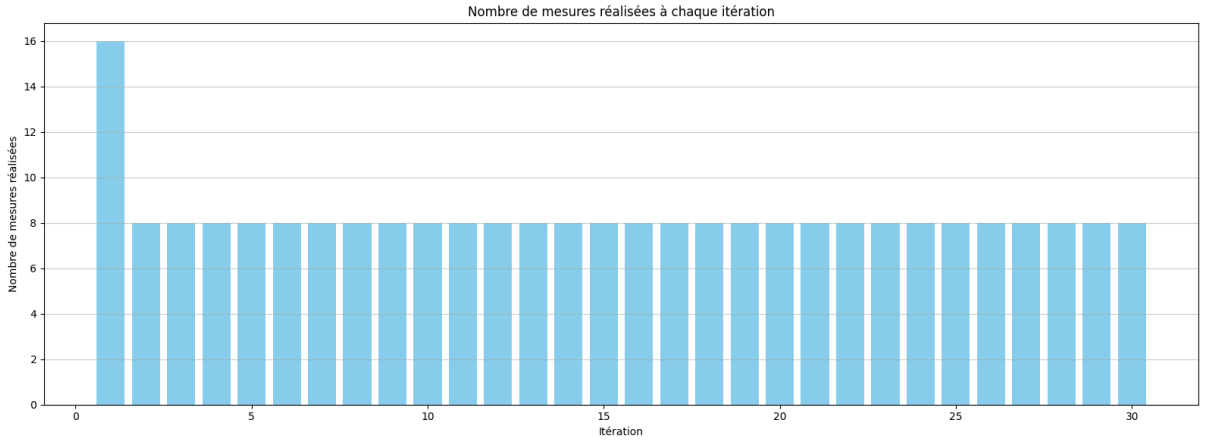
Paramètrage du benchmark :

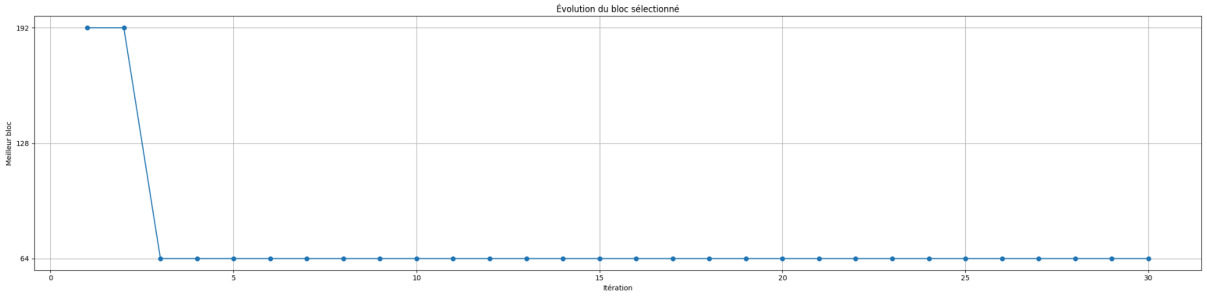
- calcul de l’attention implémenté en Cython (par nos soins)
- Blocksize testé : [64, 128, 192, 256, 320, 384, 448, 512]
- Taille de matrice Q, K, V = 2048 x 2048, d = 64 x 64

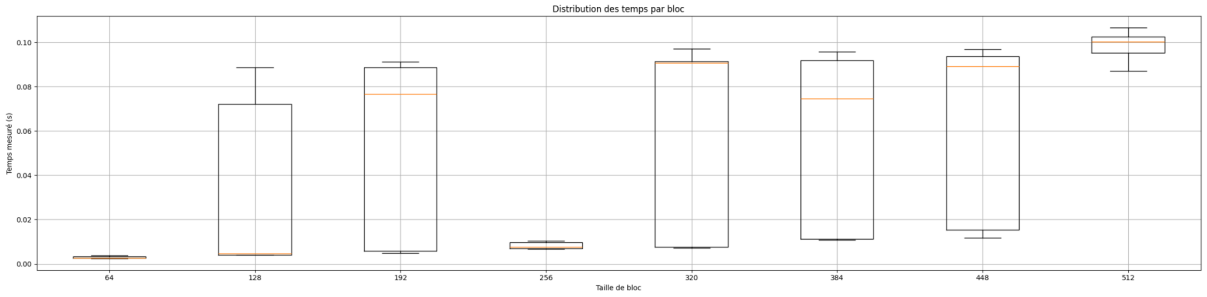
Annexe 1 - Resultat au global **Grille exhaustive VS adaptavite Benchmark**

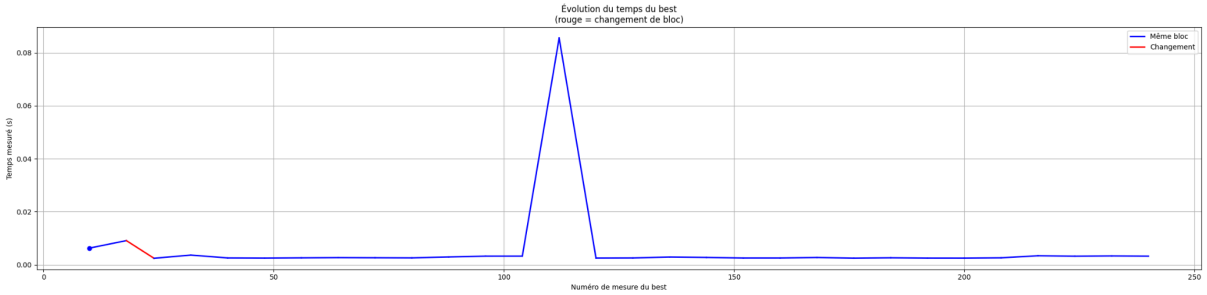


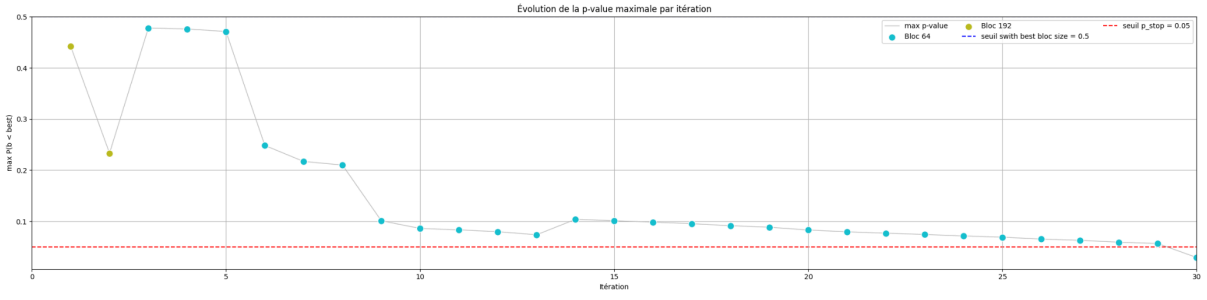
Annexe 2 – Visualisation résultats de **Grille exhaustive**



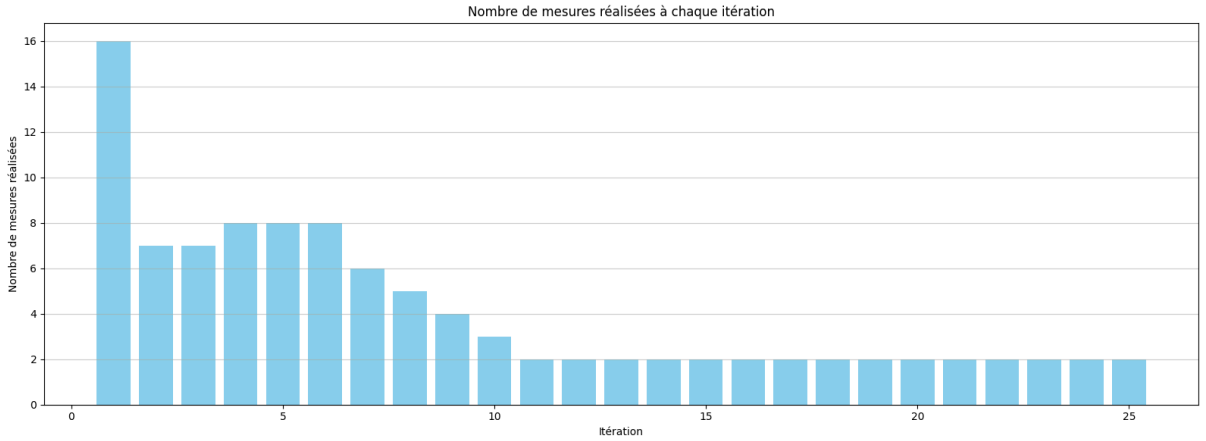


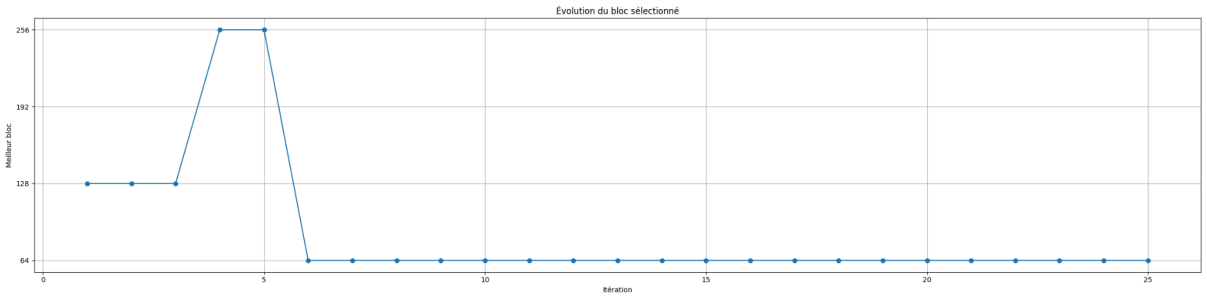


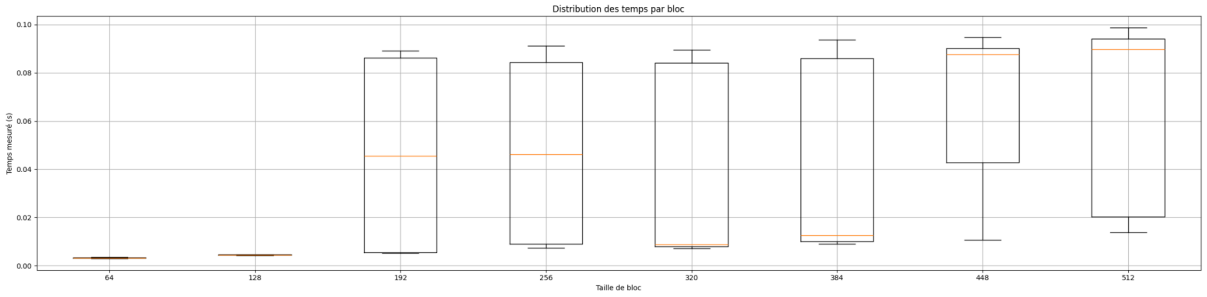


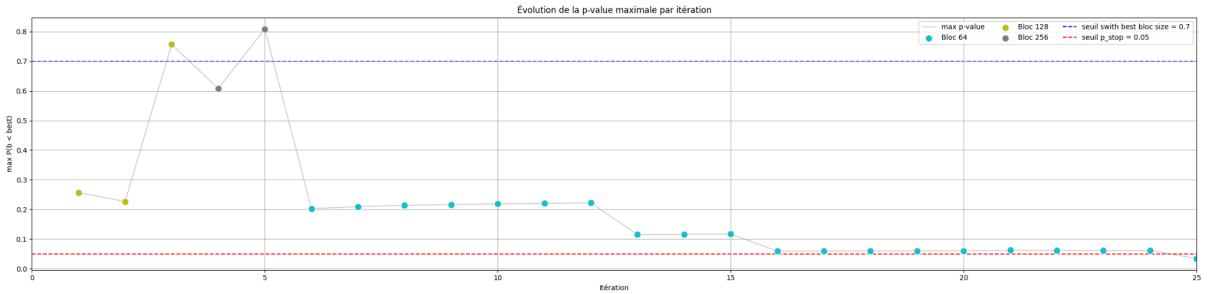


Annexe 3 – Visualisation résultats du **Benchmark adaptatif**









Précisions ici que c'est toujours la taille de bloc 64 qui est sélectionné in fine. Cela s'explique très certainement le fait que les ordinateurs actuelle sont souvent optimiser pour faire des calcules de matrices (carrés) de taille 64.