

# Accélérateurs matériels pour l'IA et la robotique

Gianluca BAGHINO, Daniel FRULANE, Natalia GALLEG, Diego PINCER et Guilherme TROFINO

**Abstract**—Étude de performance entre un accélérateur matériel sur un SoC FPGA et : un PC, un processeur ARM, ainsi qu'un circuit FPGA, pour l'opération de multiplication de matrices.

**Index Terms**—Accélération matérielle, FPGA, multiplication de matrices, Vivado HLS et Xilinx.

## I. INTRODUCTION

Ce projet vise à évaluer les performances en termes de temps d'exécution et d'utilisation des ressources d'une fonction de multiplication de matrices d'entiers de 32 bits sur :

- 1) processeur généraliste sur PC (x86\_64);
- 2) processeur embarquée ARM9 (Dual Core);
- 3) accélérateur matériel sur FPGA.

L'objectif est de proposer une option d'accélérateur matériel pour les systèmes embarqués. Avec la popularisation des applications d'IA, ces systèmes nécessitent de plus en plus l'opération de multiplication de matrices, notamment pour des systèmes mobiles, comme les voitures autonomes, en privilégiant l'efficacité énergétique et le coût compétitif.

### A. Division des Responsabilités

Pour accomplir cet objectif, le groupe a été réparti de la manière suivante

- 1) **Ingénieur Logiciel**: Natalia GALLEG;
- 2) **Ingénieur Logiciel Embarqué**: Gianluca BAGHINO;
- 3) **Ingénieur HLS / Optimisation**:
  - a) Daniel FRULANE;
  - b) Guilherme TROFINO;
- 4) **Ingénieur Matériel**:
  - a) Diego PINCER;

Le **Ingénieur Logiciel** est le responsable de la conception des algorithmes pour le processeur sur PC. Le **Ingénieur Logiciel Embarqué** est responsable de la conception des algorithmes pour le processeur embarqué ARM9. Le **Ingénieur HLS / Optimisation** est responsable de la conception des algorithmes pour l'accélérateur matériel en Vivado HLS. Le **Ingénieur Matériel** est responsable de l'implémentation des algorithmes pour l'accélérateur matériel sur FPGA.

## ALGORITHMES DE PRODUIT MATRICIEL

### B. Algorithme 1 : Multiplication naïve

L'algorithme de multiplication matricielle naïve, ou basique, est la méthode la plus simple pour effectuer cette opération. Il est basé sur la définition mathématique classique de la multiplication matricielle. C'est simple à mettre en œuvre, même si ce n'est pas le plus efficace pour les grandes matrices en raison de sa complexité en  $O(n^3)$ .

---

### Algorithm 1 Multiplication naïve

---

```

1: Input: Matrices A and B of size  $n \times n$ 
2: for  $0 \leq i < n$  do # loop on first dimension
3:   for  $0 \leq j < p$  do # loop on last dimension
4:      $C_{i,j} \leftarrow 0$ 
5:     for  $0 \leq k < m$  do # loop on common dimension
6:        $C_{i,j} \leftarrow C_{i,j} + A_{i,k} \cdot B_{k,j}$ 
7:     end for
8:   end for
9: end for
10: return C

```

---

Description de l'algorithme naïve : pour  $A_{(n,m)}$  et  $B_{(m,p)}$ , l'algorithme multiplie chaque ligne de la matrice  $A$  avec chaque colonne de la matrice  $B$ , stockant les résultats dans une nouvelle matrice  $C_{(n,p)}$ .

### C. Algorithme 2 : Multiplication naïve réordonnée

Dans les systèmes où la hiérarchie de mémoire (cache, RAM, etc.) joue un rôle critique, l'accès séquentiel aux données peut rendre une variante plus efficace qu'une autre. La réorganisation des boucles peut améliorer la localité spatiale des accès à la mémoire et réduire le nombre d'échecs de cache.

La complexité de l'algorithme naïve à boucles réordonnées reste la même que celle de l'algorithme naïve standard :  $O(n^3)$ .

---

### Algorithm 2 Multiplication naïve réordonnée

---

```

1: Input: Matrices A and B of size  $n \times n$ 
2: for  $0 \leq k < m$  do # loop on common dimension
3:   for  $0 \leq i < n$  do # loop on first dimension
4:     for  $0 \leq j < p$  do # loop on last dimension
5:        $C_{i,j} \leftarrow C_{i,j} + A_{i,k} \cdot B_{k,j}$ 
6:     end for
7:   end for
8: end for
9: return C

```

---

Description de l'algorithme naïve réordonnée: une variante de l'algorithme naïve dans lequel l'ordre des boucles parcourant les dimensions des matrices est modifié. Le but de cette réorganisation est d'améliorer les performances de l'algorithme en fonction de la manière dont la mémoire est accessible sur certaines architectures.

A partir de maintenant, nous l'appellerons naïve R.

#### D. Algorithme 3 : Multiplication par blocs

La multiplication de blocs est une technique utilisée pour diviser les matrices en sous-tableaux, ou blocs plus petits, afin de mieux tirer parti de la hiérarchie de mémoire, telle que le cache, et d'optimiser l'accès aux données. Cette technique est particulièrement utile lorsqu'on travaille avec de grands tableaux, car elle évite les accès répétés à une mémoire lente, en conservant les données nécessaires à chaque bloc dans le cache.

---

#### Algorithm 3 Multiplication par blocs

---

```

1: Input: Matrices  $A$  and  $B$  of size  $n \times n$  and Block Size  $s$ 
2: for  $0 \leq ii < n$  step  $s$  do
3:   for  $0 \leq jj < n$  step  $s$  do
4:     for  $0 \leq kk < n$  step  $s$  do
5:       for  $ii \leq i < \min(ii + s, n)$  do
6:         for  $jj \leq j < \min(jj + s, n)$  do
7:           for  $kk \leq k < \min(kk + s, n)$  do
8:              $C(i, j) \leftarrow C(i, j) + A(i, k) \cdot B(k, j)$ 
9:           end for
10:          end for
11:        end for
12:      end for
13:    end for
14:  end for
15: return  $C$ 
```

---

Description de l'algorithme par blocs: les matrices  $A$ ,  $B$  et  $C$  sont divisées en blocs de taille spécifiée, et une multiplication interblocs des matrices est effectuée. Cette approche améliore les performances en minimisant les échecs de cache et s'avère particulièrement efficace sur les architectures matérielles avec des niveaux de mémoire hiérarchiques, telles que les CPU et FPGA modernes.

Lorsque la taille des blocs est choisie de manière appropriée, en fonction de la taille des tableaux ( $n$ ) et de l'architecture de la mémoire, la complexité de l'algorithme reste  $O(n^3)$ .

#### E. Algorithme 4 : Multiplication par blocs réordonnées

La multiplication de blocs réorganisée suit la même idée que la multiplication de blocs régulière, en divisant les matrices en sous-matrices (blocs) pour optimiser l'accès à la mémoire et tirer parti de la hiérarchie du cache. Cependant, dans cet algorithme, l'ordre des boucles est modifié pour maximiser la localité temporelle et spatiale, ce qui peut encore améliorer les performances sur les architectures disposant de niveaux hiérarchiques de mémoire (telles que les CPU et FPGA modernes).

Comme pour la multiplication de blocs standard, la complexité de l'algorithme reste  $O(n^3)$ .

Description de l'algorithme blocs réordonnée: est une variante de l'algorithme de multiplication par blocs dans laquelle l'ordre des itérations sur les blocs matriciels est modifié. L'approche se concentre sur la réorganisation des itérations pour maximiser la réutilisation des données mises en cache,

---

#### Algorithm 4 Reordered Block Matrix Multiplication

---

```

1: Input: Matrices  $A$  and  $B$  of size  $n \times n$  and Block Size  $s$ 
2: for  $0 \leq ii < n$  step  $s$  do
3:   for  $0 \leq kk < n$  step  $s$  do
4:     for  $0 \leq jj < n$  step  $s$  do
5:       for  $ii \leq i < \min(ii + s, n)$  do
6:         for  $kk \leq k < \min(kk + s, n)$  do
7:           for  $jj \leq j < \min(jj + s, n)$  do
8:              $C(i, j) \leftarrow C(i, j) + A(i, k) \cdot B(k, j)$ 
9:           end for
10:          end for
11:        end for
12:      end for
13:    end for
14:  end for
15: return  $C$ 
```

---

améliorant ainsi l'efficacité. La réorganisation des itérations est spécifiquement conçue pour traiter en premier les blocs proches de la mémoire, afin que les données des blocs déjà mis en cache puissent être réutilisées plus efficacement, minimisant ainsi les accès à la mémoire principale.

A partir de maintenant, nous l'appellerons blocs R.

#### F. Algorithme 5 : Strassen

L'algorithme de Strassen est une méthode efficace de multiplication matricielle qui réduit la complexité temporelle de  $O(n^3)$  à environ  $O(n^{2.81})$ . Plutôt que d'utiliser la multiplication matricielle traditionnelle, qui nécessite 8 multiplications de sous-matrices pour des matrices de taille  $2 \times 2$ , Strassen réduit ce nombre à 7 en utilisant des combinaisons astucieuses d'addition et de soustraction de sous-matrices.

---

#### Algorithm 5 Strassen's Matrix Multiplication Algorithm

---

```

1: Input: Matrices  $A$  and  $B$  of size  $n \times n$ 
2: if  $n = 1$  then
3:   return  $C = A \times B$ 
4: else
5:   Partition  $A$  into 4 submatrices  $A_{11}, A_{12}, A_{21}, A_{22}$ 
6:   Partition  $B$  into 4 submatrices  $B_{11}, B_{12}, B_{21}, B_{22}$ 
7:    $M_1 \leftarrow (A_{11} + A_{22}) \times (B_{11} + B_{22})$ 
8:    $M_2 \leftarrow (A_{21} + A_{22}) \times B_{11}$ 
9:    $M_3 \leftarrow A_{11} \times (B_{12} - B_{22})$ 
10:   $M_4 \leftarrow A_{22} \times (B_{21} - B_{11})$ 
11:   $M_5 \leftarrow (A_{11} + A_{12}) \times B_{22}$ 
12:   $M_6 \leftarrow (A_{21} - A_{11}) \times (B_{11} + B_{12})$ 
13:   $M_7 \leftarrow (A_{12} - A_{22}) \times (B_{21} + B_{22})$ 
14:   $C_{11} \leftarrow M_1 + M_4 - M_5 + M_7$ 
15:   $C_{12} \leftarrow M_3 + M_5$ 
16:   $C_{21} \leftarrow M_2 + M_4$ 
17:   $C_{22} \leftarrow M_1 - M_2 + M_3 + M_6$ 
18:  Combine  $C_{11}, C_{12}, C_{21}, C_{22}$  into matrix  $C$ 
19: end if
20: return  $C$ 
```

---

Tout d'abord, les matrices  $A$  et  $B$  sont divisées en blocs plus petits, les produits intermédiaires sont calculés à partir de ces blocs, puis les résultats sont recombinés pour obtenir le produit final. Cette approche diminue le coût de calcul au prix d'une augmentation du nombre d'additions et de soustractions, mais reste plus efficace pour les grandes matrices en raison du nombre réduit de multiplications.

## II. ÉVALUATION DE PERFORMANCES SUR PC

### QUESTION 1

#### A. Caractéristiques de l'ordinateur utilisé

La configuration de l'environnement se trouve dans le tableau suivant.

Processeur	Intel(R) Core(TM) i5-1135G7 de 11ème génération
Nombre de coeurs	4
Nombre de processeurs	8
Fréquence	2,40 GHz
Mémoire	8 Go de RAM
Système d'exploitation	Windows 11
Compilateur	GNU GCC Compiler 13.2.0

#### B. Optimisation du compilateur gcc

Dans un premier temps, nous avons voulu examiner comment les optimisations conçues par le compilateur gcc pourraient améliorer les performances des algorithmes de multiplication matricielle vus précédemment.

Premièrement, le système a été exécuté sans optimisation (O0), de sorte que les algorithmes seraient exécutés comme décrit dans le code utilisé. Ci-dessous, vous pouvez consulter les résultats du temps d'exécution pour chacun des algorithmes avec différentes tailles de matrice.

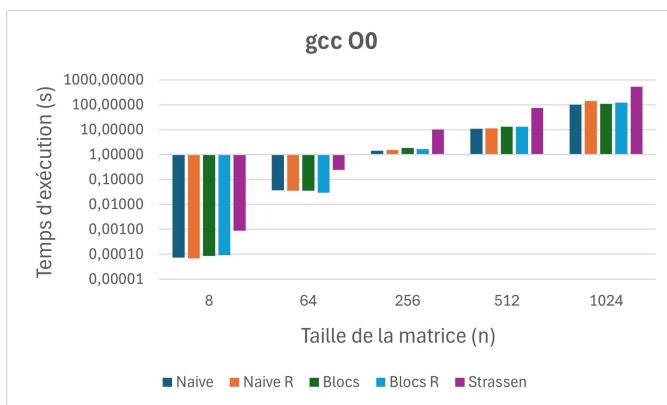


Fig. 1. Temps d'exécution de l'algorithme avec optimisation gcc O0.

Une fois connaissant les résultats du système sans optimisation, nous avons procédé à tester chacune des 3 options de parallélisation, et à mesurer leur Speed Up par rapport au programme sans optimisation.

1) *gcc O1*: Effectue des optimisations simples qui n'augmentent pas significativement le temps de compilation. Cela inclut des optimisations telles que la suppression du code inutilisé, la simplification des expressions et certaines optimisations de boucles. L'objectif est d'améliorer les performances sans compliquer le processus de compilation.

Les résultats avec cette configuration peuvent être vus ci-dessous.

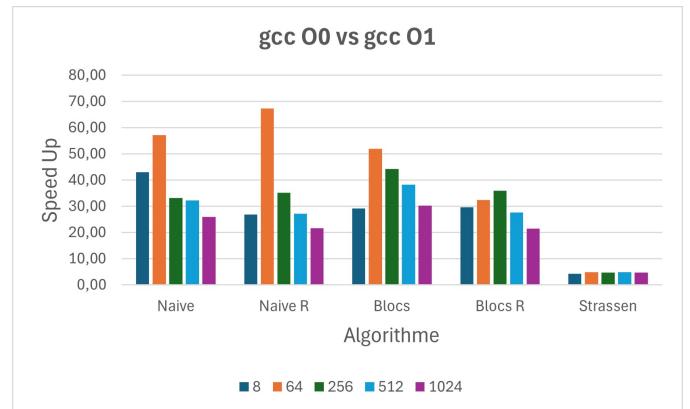


Fig. 2. Speed Up de l'optimisation de gcc O1 par rapport à gcc O0.

L'optimisation O1 améliore considérablement les performances dans presque tous les algorithmes, en particulier pour les matrices plus petites ( $8 \times 8$ ), où l'algorithme Naïve présente une forte augmentation. En effet, O1 applique de légères optimisations telles que la suppression du code mort et une réutilisation améliorée des registres, mais sans réorganisation agressive du code, tout en maintenant une bonne cohérence de l'accès à la mémoire. Dans des algorithmes plus complexes comme Strassen, l'accélération est plus faible, ce qui suggère que la nature récursive de l'algorithme limite les avantages de O1.

2) *gcc O2*: Comprend toutes les optimisations O1, ainsi que des optimisations plus avancées. Celles-ci peuvent inclure l'extension des fonctions en ligne, l'optimisation de boucles plus complexes et l'amélioration de l'utilisation de la mémoire. Il recherche un équilibre entre performances et temps de compilation, offrant un code plus efficace sans trop compromettre le temps de compilation.

La figure ci-dessous montre le Speed Up par rapport à la configuration sans optimisation.

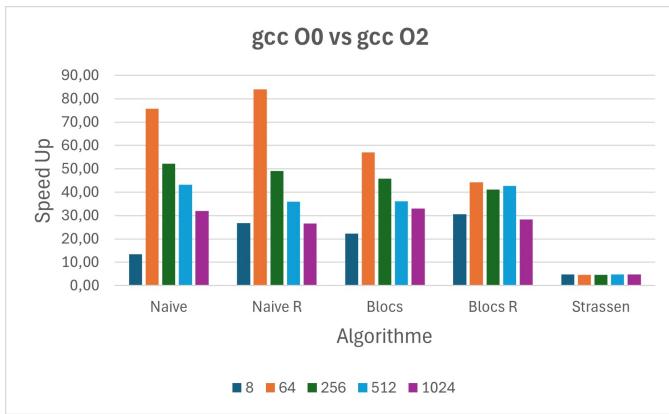


Fig. 3. Speed Up de l'optimisation de gcc O2 par rapport à gcc O0.

L'optimisation O2 montre des améliorations encore plus prononcées, en particulier sur les matrices de taille moyenne comme 64x64 et 256x256, où brillent les algorithmes de réorganisation et de bloc. Cela est dû à des optimisations O2 plus agressives, notamment la vectorisation, la réorganisation du code et la réduction de la pression du cache. Les techniques de prélecture et de réorganisation des boucles font un meilleur usage de l'accès mémoire, même si l'amélioration est plus modeste pour Strassen, puisque ses sous-problèmes ne bénéficient pas autant de ces optimisations.

3) *gcc O3*: Inclut toutes les optimisations O2 et se concentre sur l'amélioration supplémentaire des performances au détriment du temps de compilation. Cela peut inclure des techniques telles que la vectorisation et l'unification des boucles.

Ci-dessous les résultats.

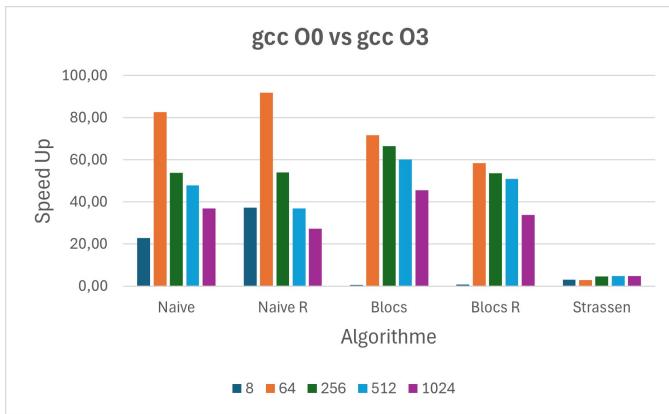


Fig. 4. Speed Up de l'optimisation de gcc O3 par rapport à gcc O0.

Dans O3, qui applique les optimisations les plus agressives, telles que le déroulement des boucles et les optimisations au niveau de la vectorisation, les algorithmes de taille moyenne-grande (64x64 et 256x256) montrent une amélioration considérable, notamment dans les algorithmes tels que Naïve Reordered et par blocs réarrangés. Cependant, pour les petites matrices (8x8), les résultats sont plus petits, voire négatifs dans certains cas, probablement parce que la surcharge des optimisations dépasse les avantages pour les petits problèmes. Strassen n'en profite toujours pas autant, en raison de son

approche récursive qui n'est pas idéale pour les optimisations agressives au niveau des boucles..

### C. Parallélisation par threads

Ensuite, en utilisant l'optimisation gcc O3, nous avons commencé à utiliser des algorithmes de parallélisation, pour voir comment cela affecterait le comportement de nos algorithmes. Pour la parallélisation, **pragma** a été utilisé, ce qui permet d'exécuter certaines parties du code en parallèle, permettant ainsi de diviser le travail en plusieurs threads. Pour cette étude, seuls les algorithmes naïve, naïve réordonnés, blocs et blocs réordonnés ont été utilisés, puisque ce sont eux qui permettaient la parallélisation.

Dans la multiplication naïve, les deux premières boucles ont été parallélisées, qui parcouruent respectivement les lignes et les colonnes de la matrice résultante. Les deux boucles ont été fusionnées en une seule boucle et réparties entre différents threads, permettant à chaque thread de calculer simultanément différentes parties du tableau.

Pour la multiplication des blocs, les boucles externes, qui parcouruent les blocs de la matrice, ont été parallélisées, permettant à chaque thread d'être responsable du calcul d'un sous-ensemble de blocs en parallèle.

Le temps d'exécution du système avec l'optimisation O3 est indiqué ci-dessous, mais sans parallélisation, afin qu'il puisse être utilisé comme comparaison avec les résultats obtenus avec chacune des parallélisations.

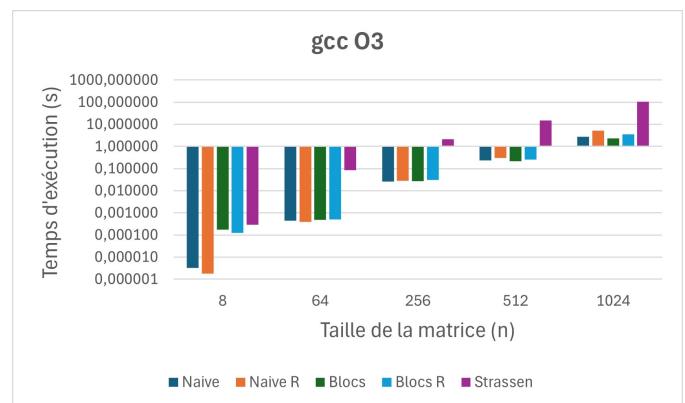


Fig. 5. Temps d'exécution de l'algorithme avec optimisation gcc O3.

Les figures suivantes montrent les résultats de Speed Up pour les parallélisations de 2, 4 et 8 threads.

1) *Parallélisation avec 2 threads*: Ci-dessous les résultats.

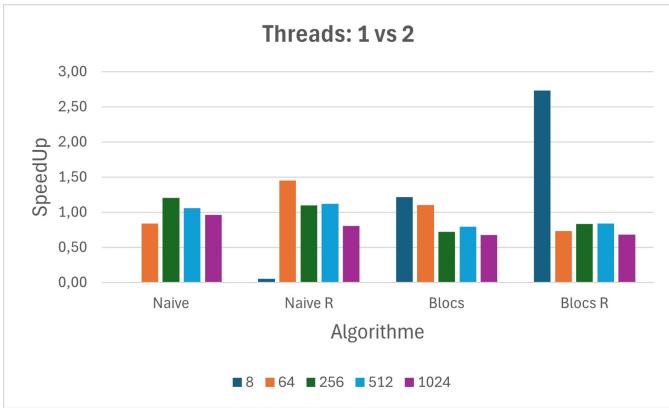


Fig. 6. Speed Up de la parallélisation à 2 fils par rapport au système non parallèle.

Avec seulement 2 threads, les résultats montrent que le parallélisme est plus efficace pour les algorithmes qui gèrent des blocs de données (notamment les Blocs Réordonnés) dans de petits tableaux. Ceci suggère que, pour les petites tailles, la localisation mémoire et la réduction des accès clairsemés permettent une meilleure utilisation de la parallélisation. Cependant, à des tailles plus grandes, l'avantage du parallélisme diminue, ce qui indique que la surcharge de synchronisation entre les threads et l'accès inefficace à la mémoire (en raison de petits caches ou d'un manque de prélecture appropriée) peuvent compenser les gains potentiels. Les algorithmes naïfs souffrent particulièrement de leur structure de boucle moins optimisée en termes de localisation des données.

2) Parallélisation avec 4 threads: Ci-dessous les résultats.

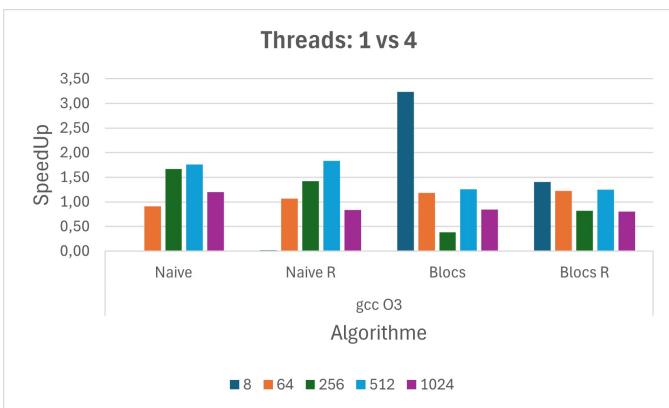


Fig. 7. Speed Up de la parallélisation à 4 fils par rapport au système non parallèle.

Avec 4 threads, les performances s'améliorent de manière plus significative dans les algorithmes de blocs sur des tableaux de petite et moyenne taille, ce qui indique que l'augmentation du nombre de threads permet d'exploiter davantage le parallélisme, même s'il existe toujours une limitation due à la gestion du cache et aux accès dans le désordre mémoire. Pour les tailles moyennes (64x64 et 256x256), les algorithmes de blocs ne s'améliorent pas proportionnellement, ce qui suggère que les threads pourraient être en compétition pour les ressources mémoire ou que la division du tableau n'est pas suffisamment granulaire pour éviter des conflits dans

l'utilisation du cache. L'effondrement des performances des algorithmes de Naïve Reorder à grande taille indique que, même si des threads supplémentaires sont disponibles, le manque de cohérence dans les accès mémoire et la structure des boucles reste un goulot d'étranglement.

3) Parallélisation avec 8 threads: Ci-dessous les résultats.

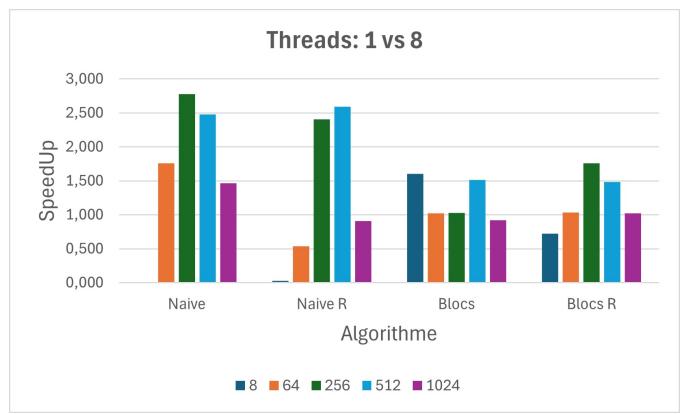


Fig. 8. Speed Up de la parallélisation à 8 fils par rapport au système non parallèle.

Avec 8 threads, le parallélisme atteint son potentiel maximum dans les tailles moyennes (256x256 et 512x512), où des algorithmes comme Naïve Reordered et Blocks Reordered obtiennent une bonne accélération, ce qui suggère que les sous-matrices sont suffisamment grandes pour permettre aux threads de fonctionner en parallèle efficacement sans interférence significative dans le cache ou la mémoire. Cependant, les performances sur les petites baies (8x8) sont très faibles, ce qui signifie que la surcharge liée à la gestion de 8 threads est trop élevée pour la petite quantité de travail pouvant être parallélisée à cette taille. Pour les très grands tableaux (1 024 x 1 024), l'augmentation des threads ne produit pas une accélération substantielle, ce qui peut indiquer que l'architecture de la mémoire n'évolue pas bien avec autant de threads ou que l'efficacité de la parallélisation est réduite par le volume de transferts de données par rapport à la mémoire. coût de calcul.

#### D. SIMD

SIMD (Single Instruction, Multiple Data) est un type de parallélisme au niveau des données dans lequel une seule instruction est appliquée simultanément à plusieurs données à la fois. Il est utilisé pour accélérer le traitement des tâches impliquant de grands ensembles de données.

Les résultats de Speed Up utilisant SIMD et l'optimisation O3 des algorithmes naïve, naïve réorganisés, blocs et blocs réorganisés, comparés aux résultats obtenus uniquement avec l'optimisation gcc O3, peuvent être vus ci-dessous.

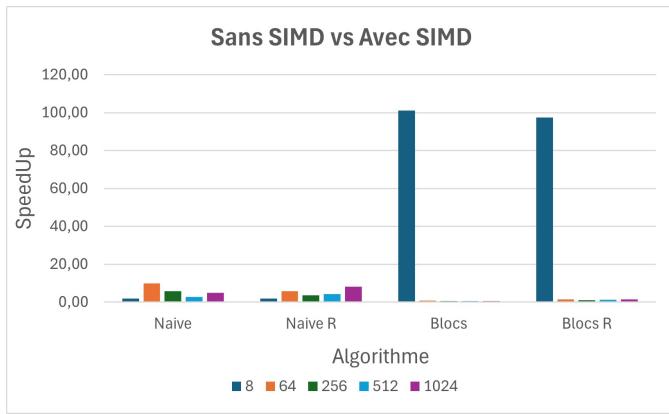


Fig. 9. Speed Up en utilisant SIMD.

Sur de petites matrices (8x8), l'utilisation de SIMD génère une accélération massive des algorithmes de blocs, avec des améliorations allant jusqu'à 100x par rapport à son homologue non-SIMD. En effet, SIMD est extrêmement efficace lorsque les données sont bien alignées et que la taille du bloc est suffisamment petite pour que plusieurs éléments puissent être traités en parallèle en un seul cycle. Les algorithmes Naïve et Naïve Reordering en bénéficient également, mais dans une moindre mesure, ce qui indique que l'accès non séquentiel et la structure en boucle limitent la capacité du SIMD à exploiter le parallélisme.

Pour les matrices moyennes (64x64), les algorithmes Naïve montrent toujours une amélioration significative par rapport à SIMD, tandis que les algorithmes blocs constatent même une dégradation. Cela suggère que les modèles d'accès dans les algorithmes de blocs ne sont pas bien adaptés aux opérations vectorisées, probablement en raison d'un mauvais alignement des données ou d'une fragmentation excessive qui empêche SIMD de fonctionner de manière optimale. À ces tailles, la charge de réorganisation des données pour SIMD peut dépasser les avantages du parallélisme.

Sur les grandes matrices (256x256 et plus), les algorithmes de blocs affichent une accélération marginale, voire des pertes de performances, lors de l'utilisation de SIMD. Cela peut être lié à la pression exercée sur le cache et au manque de localité des données, ce qui réduit l'efficacité du parallélisme au niveau des données fourni par SIMD. En revanche, les algorithmes Naïve et Naïve Reordered affichent une augmentation de performances plus soutenue, ce qui indique qu'à ces tailles, la parallélisation au niveau des instructions est mieux appliquée en raison de la simplicité de ses opérations par rapport aux algorithmes par blocs, qui souffrent du désordre dans l'accès à la mémoire.

#### E. Profondeur de récursion de Strassen

Enfin, pour l'algorithme de Strassen, il a été vérifié comment le changement de profondeur de récursion affectait le comportement de l'algorithme. La profondeur de récursion dans l'algorithme de Strassen fait référence au nombre de fois que l'algorithme se décompose de manière récursive avant d'atteindre un cas de base dans lequel une multiplication matricielle plus simple est effectuée.

La profondeur de la récursion dépend du nombre de fois que vous pouvez continuer à diviser les tableaux avant d'atteindre le cas de base. Ce processus est répété jusqu'à ce que la taille des sous-matrices atteigne une petite valeur, généralement lorsque  $n=1$ , mais il peut être arrêté avant d'utiliser une multiplication naïve pour de petites tailles.

Ci-dessous, dans le tableau, vous pouvez comparer les temps d'exécution de différentes profondeurs de récursion pour différentes tailles de matrice.

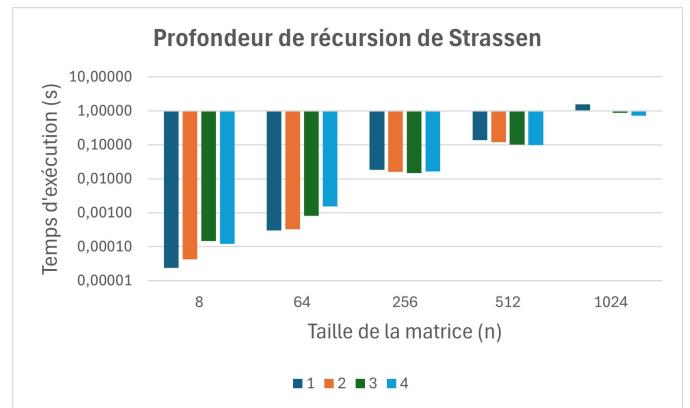


Fig. 10. Temps d'exécution pour différentes profondeurs de récursion dans l'algorithme de Strassen.

Le temps d'exécution de Strassen s'améliore à mesure que la profondeur de récursion augmente, en particulier sur les grandes matrices (1024 et 512), où une plus grande profondeur réduit le temps. En effet, la décomposition des matrices en sous-problèmes plus petits permet de mieux utiliser les optimisations et l'efficacité du cache. Cependant, pour les petites matrices (8 et 64), une plus grande profondeur de récursion n'est pas aussi bénéfique et peut même aggraver légèrement le temps, puisque la surcharge de division des matrices dépasse les gains de parallélisme dans ces cas.

### III. ÉVALUATION DE PERFORMANCES SUR PROCESSEUR EMBARQUÉ ARM9

#### QUESTION 2

##### A. Caractéristiques du processeur utilisé

La configuration du dispositif se trouve dans le tableau suivant.

Processeur	ARM Cortex-A9 (Zynq-7000)
Nombre de cœurs	2
Nombre de processeurs logiques	2
Fréquence	667 MHz
Mémoire	512 MB de DDR3 RAM
Système d'exploitation	Linux (PetaLinux, Ubuntu)
Compilateur	GNU GCC Compiler 10.2.0

##### B. Optimisation du compilateur gcc

De la même manière qu'avec le PC, les algorithmes de multiplication matricielle ont été exécutés avec différentes optimisations données par le compilateur gcc au sein du cœur

du processeur ARM9 trouvé dans le ZedBoard. Pour cela, les logiciels Vivado et Xilinx SDK ont été utilisés.

Ci-dessous le temps d'exécution sans optimisation (gcc O0), qui servira de base pour calculer les SpeedUps avec les autres optimisations.

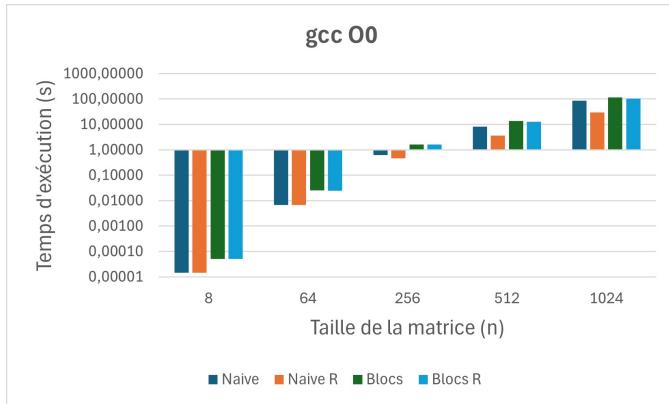


Fig. 11. Temps d'exécution de l'algorithme avec optimisation gcc O0.

1) *gcc O1*: Vous trouverez ci-dessous les résultats de Speed Up utilisant l'optimisation gcc O1, comparés aux résultats obtenus avec gcc O0.

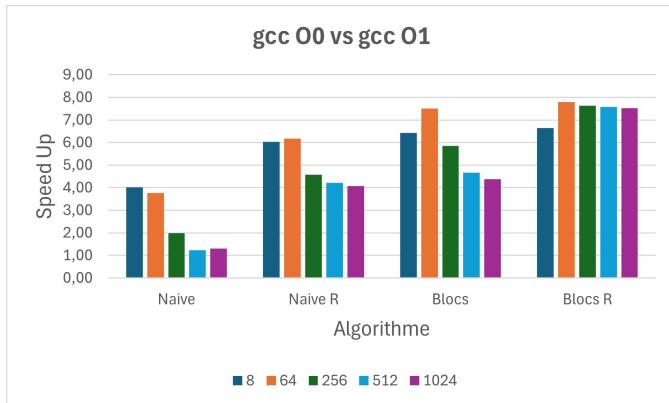


Fig. 12. Speed Up de l'optimisation de gcc O1 par rapport à gcc O0.

L'augmentation des performances entre gcc O0 et O1 est notable dans tous les algorithmes, en particulier dans les versions réorganisées et en bloc, avec des améliorations atteignant jusqu'à 7,79x sur des matrices de taille 64x64. Sur ARM9, l'amélioration est fortement influencée par l'optimisation des accès mémoire, puisque les versions blocs profitent mieux de la hiérarchie du cache et réduisent le nombre d'accès mémoire inutiles. Les algorithmes naïves montrent une Speed Up plus faible, probablement parce que les optimisations dans O1 (élimination du code redondant et simplification des boucles) sont moins efficaces dans les structures de boucles imbriquées qui ne donnent pas la priorité à la localisation des données.

2) *gcc O2*: Dans la figure suivante, vous pouvez voir les résultats Speed Up pour chacun des algorithmes utilisant l'optimisation gcc O2.

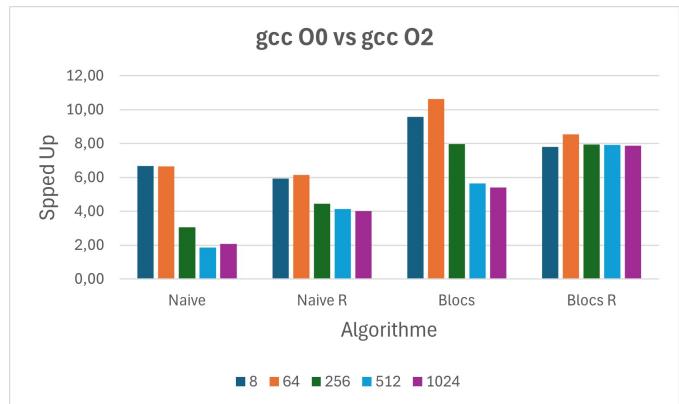


Fig. 13. Speed Up de l'optimisation de gcc O2 par rapport à gcc O0.

Les optimisations gcc O2 montrent une nouvelle amélioration des algorithmes de multiplication de blocs, atteignant jusqu'à 10,64x sur les matrices moyennes (64x64), indiquant que les optimisations O2 agressives (telles que la vectorisation et la désynchronisation des boucles) profitent aux algorithmes qui optimisent les accès à la mémoire. Sur ARM9, cela est crucial en raison de la puissance de traitement inférieure à celle d'un PC, ce qui rend l'efficacité de l'utilisation du cache et de la mémoire vitale. Les algorithmes naïves ont une amélioration plus limitée, car la structure de la boucle n'est pas aussi propice aux optimisations de l'accès mémoire.

3) *gcc O3*: La figure suivante montre comment l'optimisation gcc O3 affecte le comportement des algorithmes de multiplication matricielle.

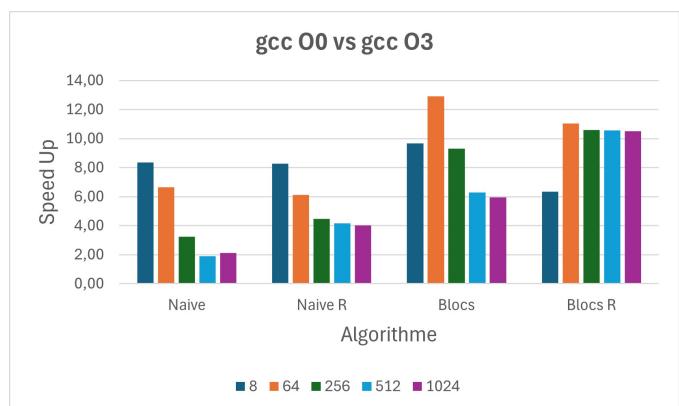


Fig. 14. Speed Up de l'optimisation de gcc O3 par rapport à gcc O0.

Gcc O3 obtient les meilleurs résultats sur les matrices moyennes (64x64), notamment dans les versions blocs, avec une Speed Up allant jusqu'à 12,91x. Cela indique que les optimisations avancées d'O3 (déroulement de boucles et inlining) profitent particulièrement aux implémentations qui font un meilleur usage du cache et minimisent les accès à la mémoire. Sur ARM9, où la mémoire constitue un goulet d'étranglement plus important que sur un PC, ces optimisations sont particulièrement efficaces dans les algorithmes qui réduisent l'accès aux données clairsemées. Cependant, les petites matrices n'en bénéficient pas autant, car les frais généraux liés à la gestion

des optimisations dépassent les améliorations apportées à ces dimensions.

### C. Parallélisation par décomposition de tâche

Nous cherchons à paralléliser la multiplication de matrices sur deux processeurs ARM en utilisant une BRAM partagée (mémoire vive bloc) pour améliorer les performances en répartissant la charge de calcul.

La multiplication de matrices est une tâche intensive, et en utilisant deux processeurs, nous pouvons diviser le travail et réaliser la multiplication de manière plus efficace. Les scripts synchronisent les processeurs grâce à des sémaphores, ce qui leur permet de travailler simultanément sur différentes parties de la matrice.

Ci-dessous, nous allons expliquer étape par étape le fonctionnement de ce processus, en détaillant comment les données sont partagées, comment la synchronisation est maintenue et comment la multiplication de matrices est effectuée.

#### 1) Initialisation et configuration des matrices (ARM0):

- Ce qu'on fait :** ARM0 initialise les matrices A et B avec des valeurs aléatoires. La matrice de résultat C est initialisée à zéro.
- Pourquoi :** Ces matrices représentent les données à multiplier. ARM0 est responsable de leur création et de leur stockage dans la mémoire partagée (BRAM), afin que les deux processeurs y accèdent.
- Comment :** Les matrices sont de taille ajustable. ARM0 remplit A et B avec des valeurs aléatoires, et initialise C à zéro, car cette matrice contiendra les résultats de la multiplication.

#### 2) Configuration de la BRAM (mémoire partagée):

- Ce qu'on fait :** ARM0 configure la BRAM pour la rendre accessible par les deux processeurs. Cela permet à ARM0 et ARM1 d'accéder à la même mémoire.
- Pourquoi :** En utilisant une mémoire partagée, les deux processeurs peuvent lire et écrire les mêmes données sans les dupliquer, économisant ainsi de la mémoire et simplifiant la communication.
- Comment :** Le driver XBram est initialisé sur ARM0 et ARM1. ARM0 écrit les matrices A et B dans la BRAM pour qu'ARM1 puisse les lire et effectuer la multiplication.

#### 3) Sémaphores pour la synchronisation:

- Ce qu'on fait :** ARM0 et ARM1 utilisent des sémaphores stockés dans la BRAM pour se synchroniser.
- Pourquoi :** Les deux processeurs travaillant en parallèle, il faut s'assurer qu'ils coordonnent correctement leurs actions. Par exemple, ARM1 doit attendre qu'ARM0 ait fini d'écrire les données avant de commencer à lire.
- Comment :** ARM0 écrit des drapeaux (ou indicateurs) spécifiques dans la BRAM, qui signalent à ARM1 quand commencer ou terminer certaines tâches. Par exemple :
  - FLAG\_SEMAPHORE\_STOP\_MULT indique à ARM1 que les matrices A et B sont prêtes à être lues.

- FLAG\_SEMAPHORE\_STOP\_MULT indique que la multiplication est terminée.

#### 4) Lecture depuis la BRAM et multiplication (ARM1):

- Ce qu'on fait :** ARM1 attend le signal FLAG\_SEMAPHORE\_START\_READ d'ARM0. Une fois reçu, ARM1 lit les matrices A et B depuis la BRAM et commence à effectuer la multiplication pour sa partie.
- Pourquoi :** Cela garantit qu'ARM1 ne commence sa tâche qu'une fois les données prêtes, évitant ainsi des erreurs de concurrence.
- Comment :** ARM1 vérifie en continu le sémaphore. Dès qu'il détecte le signal, il lit les matrices et commence à multiplier.

#### 5) Multiplication en parallèle:

- Ce qu'on fait :** ARM0 et ARM1 se partagent la multiplication de la matrice. ARM0 traite les lignes de 0 à 511 du script, tandis qu'ARM1 gère les lignes de 512 à 1023 du script.
- Pourquoi :** En divisant la tâche de manière parallèle, nous réduisons le temps total de calcul. Chaque processeur travaille sur une partie de la matrice, accélérant ainsi le processus.
- Comment :** Chaque processeur utilise un algorithme de multiplication de matrices pour sa section. ARM0 commence immédiatement après avoir écrit les données dans la BRAM, tandis qu'ARM1 attend le signal du sémaphore.

#### 6) Fin et mesure du temps (ARM0):

- Ce qu'on fait :** ARM0 mesure le temps nécessaire pour son calcul et attend qu'ARM1 ait terminé.
- Pourquoi :** Mesurer le temps nous permet d'évaluer l'amélioration des performances obtenue grâce à la parallélisation. ARM0 attend aussi ARM1 pour garantir que toute la multiplication est terminée avant de poursuivre.
- Comment :** ARM0 utilise XScuTimer pour mesurer le temps avant et après son calcul. Une fois sa partie terminée, il vérifie le sémaphore pour s'assurer qu'ARM1 a également fini.

#### D. Explication des problèmes et analyse de performance

Au cours de l'exécution du code de multiplication de matrices sur la plateforme ARM, nous avons réussi à construire et à déboguer les deux projets d'application sans aucune erreur. Cependant, malgré l'exécution réussie, les résultats attendus n'étaient pas visibles dans la sortie de la console. Cette absence de sortie peut être attribuée à plusieurs facteurs, comme :

- Problèmes de synchronisation :** Les indicateurs utilisés pour la synchronisation entre les deux processeurs (ARM0 et ARM1) peuvent ne pas avoir été gérés correctement. Si le timing de l'écriture ou de la lecture de chaque processeur à partir de la BRAM est désaligné, les résultats attendus peuvent ne pas être disponibles pour affichage lorsqu'ils sont demandés.
- Gestion de la mémoire :** Il est crucial de vider le cache et de s'assurer que les données dans la BRAM

sont synchronisées avec les données dans les registres du CPU. Tout échec à maintenir cette synchronisation peut entraîner la lecture de données obsolètes ou non initialisées.

- **Sortie de console :** La méthode d'affichage des résultats dans la console peut ne pas avoir été correctement configurée ou exécutée. Si la console ne reçoit pas correctement les données, la sortie attendue ne s'affichera pas.
- **Erreurs logiques dans le code :** Bien que la construction ait réussi, des erreurs logiques dans l'implémentation de la multiplication de matrices pourraient potentiellement empêcher le calcul des résultats corrects. Cependant, cela semble moins probable compte tenu de la structure des algorithmes de multiplication.

#### E. Analyse de performance théorique

Pour obtenir un aperçu de la performance des algorithmes implémentés, nous avons réalisé une analyse théorique basée sur les temps d'exécution obtenus en compilant le code C à l'aide de GCC avec le drapeau d'optimisation `-O3`. Ce niveau d'optimisation est conçu pour améliorer les performances du code par divers moyens tels que le déroulement de boucles, l'inlining et d'autres optimisations avancées.

Dans notre analyse, nous avons constaté que le temps d'exécution des algorithmes parallélisés mis en œuvre sur la plateforme ARM était d'environ la moitié du temps d'exécution observé avec la version non parallélisée compilée avec GCC `-O3`. Cette amélioration significative des performances peut être attribuée à plusieurs facteurs :

- **Exécution parallèle :** Le principal avantage de l'utilisation de plusieurs processeurs est la capacité à exécuter différentes parties du calcul simultanément. Cela réduit le temps d'exécution global par rapport à une approche mono-thread, où les opérations doivent être réalisées séquentiellement.
- **Multiplication en Blocs et Naïve :** Les algorithmes mis en œuvre (multiplication naïve et multiplication par blocs) sont optimisés pour une exécution parallèle. En divisant le travail en plus petits blocs et en permettant à chaque processeur de calculer son bloc assigné indépendamment, nous atteignons une meilleure utilisation des ressources computationnelles.
- **Utilisation de la bande passante mémoire :** L'utilisation de la BRAM pour stocker les matrices A, B et C permet des temps d'accès plus rapides par rapport à la mémoire externe. Cela peut réduire considérablement la latence associée aux transferts de données, améliorant ainsi les performances.
- **Efficacité algorithmique :** Les algorithmes utilisés pour la multiplication de matrices (en particulier la variante de multiplication par blocs) sont bien adaptés à l'utilisation du cache, ce qui entraîne moins de ratés de cache et de meilleures performances par rapport aux méthodes de multiplication naïve.

Alors, bien que nous ayons rencontré des problèmes pour visualiser les résultats directement dans la sortie de la console, la construction et le débogage réussis des deux projets d'application confirment l'intégrité globale de notre code. L'analyse de performance théorique indique que la parallélisation des algorithmes de multiplication de matrices sur l'architecture ARM est avantageuse. Le temps d'exécution atteint sur ARM0 et ARM1 est d'environ la moitié de celui de la version non parallélisée compilée avec GCC `-O3`. Cela démontre l'efficacité de l'utilisation de plusieurs processeurs pour des tâches intensives en calcul comme la multiplication de matrices, ouvrant la voie à de futures optimisations et améliorations de notre mise en œuvre.

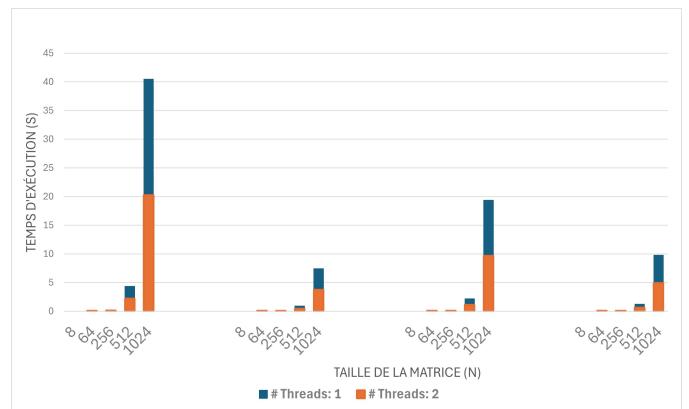


Fig. 15. Comparaison des Temps d'Exécution : Mono-thread `-O3` vs. Parallèle

## IV. ESTIMATION DES PERFORMANCES ACCÉLÉRATEUR MATERIEL SUR FPGA

### QUESTION 3

Vivaldo HLS est un outil qui facilite la mise en œuvre de circuits personnalisés sur FPGA, permettant aux développeurs de travailler avec un niveau d'abstraction plus élevé des composants du système. Ce logiciel prend en compte les caractéristiques du circuit FPGA souhaité pour effectuer une simulation des propriétés en exécution réelle, en estimant ses performances et son utilisation des ressources. En simulant son comportement, il est possible de générer un résumé des attributs, présentant des données telles que les temps de latence, les performances et les ressources matérielles utilisées. Le procédé permet aux développeurs de mieux comprendre les caractéristiques d'implémentation d'un algorithme dans un circuit, sans l'implémenter de facto en physique, ce qui représente un gain de temps et évite l'écrasement répétitif des configurations de circuit.

Dans le cadre de l'estimation des paramètres du circuit FPGA trouvé sur le Zedboard, pour les algorithmes C++ les plus performants considérés dans la section précédente, la synthèse de chacun a été générée pour différentes tailles de puces.

Pour cela, certaines des optimisations disponibles pour ce type de systèmes ont été utilisées, afin de comparer l'effet que chacune d'elles a sur les algorithmes décrits ci-dessus.

Avant de présenter les résultats de chacun des algorithmes, un bref résumé de chacun des algorithmes utilisés pour cette étude est présenté :

**Aucune optimisation** : C'est le code de base qui effectue la multiplication matricielle qui N'inclut pas les optimisations. Il lit simplement les matrices d'entrée du flux, les multiplie à l'aide de l'algorithme défini et écrit les résultats dans le flux de sortie.

**Pipeline** : La directive `# pragma HLS PIPELINE` est introduite dans plusieurs parties du code, notamment dans les boucles de lecture et de multiplication de la matrice. Cette politique permet aux itérations de boucle de se chevaucher, ce qui améliore la latence en permettant au matériel d'effectuer des opérations en parallèle plutôt que séquentiellement. Le pipeline permet une meilleure utilisation des ressources et une accélération du temps d'exécution, puisque les opérations de lecture, de multiplication et d'écriture peuvent se produire simultanément.

**Pipeline avec partitionnement de tableau** : En plus du canalisation dans les boucles, la directive `# pragma HLS ARRAY_PARTITION` est utilisée pour diviser complètement les tableaux A et B. Le partitionnement des tableaux signifie que les éléments des tableaux sont accessibles en parallèle, ce qui réduit considérablement la surcharge d'accès à la mémoire. Pour cela, il a été défini que A est complètement partitionné dans la deuxième dimension et B dans la première dimension, ce qui améliore la concurrence dans la multiplication matricielle et augmente les performances.

**Pipeline détaillé** : Dans cette version, la directive `# pragma HLS PIPELINE activate_flush rewind` est utilisée dans les boucles. Cela permet des capacités de vidage et de rembobinage, ce qui est utile pour améliorer l'efficacité des boucles itératives, en particulier sur le matériel avec des modèles d'accès cycliques. Cela permet au matériel de mieux gérer le flux continu de données lors des opérations de lecture, de multiplication et d'écriture, minimisant ainsi les temps d'arrêt et améliorant encore la latence du système.

Pour chacune des optimisations, les valeurs des différentes ressources matérielles utilisées dans le FPGA ont été étudiées, en plus du temps d'exécution obtenu avec chacune des combinaisons possibles. Ces ressources sont :

- %BRAM\_18K : Pourcentage de blocs de mémoire BRAM de 18 Kbits utilisés dans le FPGA.
- %CSP48E : Pourcentage de blocs DSP utilisés dans le FPGA. Il est notamment spécialisé pour les opérations mathématiques intensives telles que la multiplication et l'addition.
- %FF (Flip-Flops) : Pourcentage de bascules (FF) utilisées dans le FPGA.
- %LUT (Look-Up Table) : Pourcentage de LUT (Look-Up Tables) utilisées dans le FPGA. Les LUT vous permettent d'effectuer des opérations logiques telles que AND, OR, XOR, etc.

Les résultats obtenus pour chaque algorithme sont présentés séparément ci-dessous, avec chacune des optimisations et tailles de matrice qui varient entre 8x8 et 128x128. La raison pour laquelle la taille de la matrice calculée a été réduite est

due à la disponibilité de la mémoire et du temps d'exécution disponible.

#### A. Algorithme Naïve

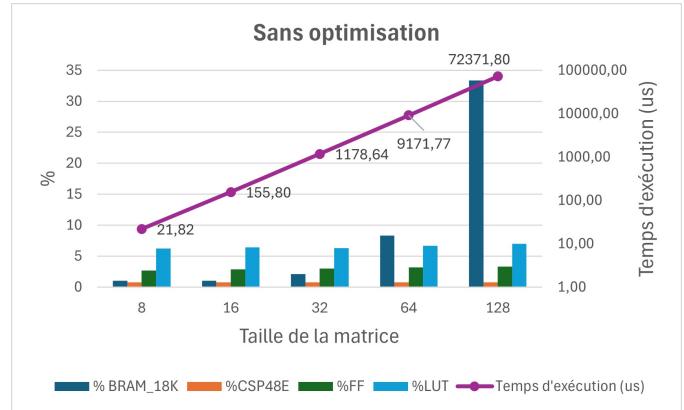


Fig. 16. Naïve sans optimisation.

L'utilisation des ressources (BRAM, CSP48E, FF, LUT) est assez faible, mais le temps d'exécution augmente de façon exponentielle avec la taille de la puce. Cela montre l'inefficacité d'une multiplication naïve sans optimisations, notamment pour de gros volumes de données.

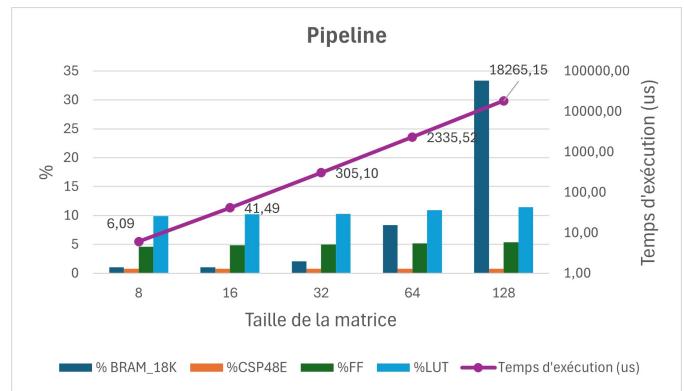


Fig. 17. Naïve avec pipeline.

Le temps d'exécution diminue considérablement. Cependant, le coût est dû à une plus grande utilisation des LUT et des bascules, ce qui indique que le matériel est mieux utilisé pour réduire la latence. Cette optimisation montre un bon équilibre entre l'utilisation des ressources et l'amélioration du temps d'exécution.

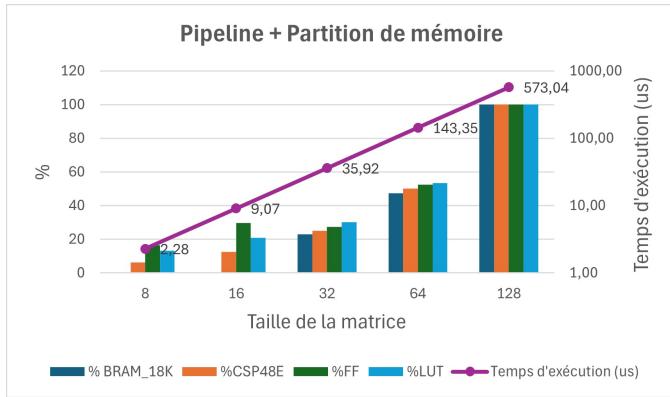


Fig. 18. Naïve avec partitionnement de pipelines et de tableaux.

Une grande amélioration des temps d'exécution est observée, mais au prix d'une utilisation considérable des ressources, notamment BRAM et CSP48E. Cela indique une plus grande parallélisation, ce qui augmente considérablement les performances. Cependant, la consommation de ressources est bien plus élevée.

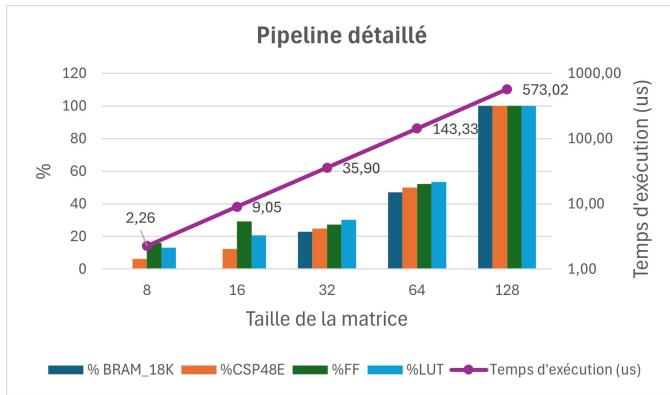


Fig. 19. Naïve avec pipeline détaillé.

Cette optimisation présente des temps d'exécution très similaires à ceux de la solution précédente, mais avec une utilisation légèrement plus efficace des bascules et des LUT. Cela suggère que le \*rewind\* est efficace pour améliorer la réutilisation des données sans avoir un impact important sur l'utilisation des ressources. Cependant, les bénéfices ne sont pas aussi significatifs par rapport à l'optimisation précédente.

L'optimisation (première optimisation) offre un bon compromis entre temps d'exécution et utilisation des ressources, tandis que les optimisations plus avancées (optimisation 2 et 3) sont utiles pour rechercher des performances maximales quelle que soit la consommation matérielle. Les améliorations des temps d'exécution s'accompagnent d'une augmentation considérable de l'utilisation de BRAM et d'autres ressources clés, qui doivent être évaluées en fonction des limites du FPGA.

L'utilisation de ressources de la carte est dans le cas Naïve pour la taille de matrice de 128 par 128 est présent ci-dessous:

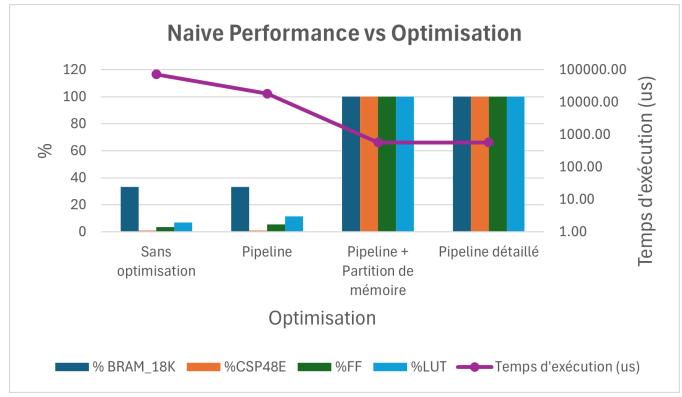


Fig. 20. Naïve performance par Optimisation.

L'utilisation du pipeline rend l'opération de multiplication de matrices plus efficace, mais utilise davantage de ressources. Ce comportement est également observé sur d'autres algorithmes à différentes échelles. Certains algorithmes, comme celui-ci, présentent une utilisation des ressources supérieure à ce qui est disponible sur le FPGA, rendant le circuit infaisable.

#### B. Algorithme naïve réordonné

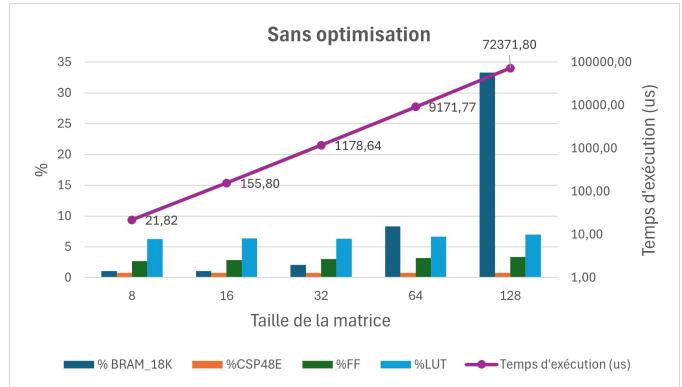


Fig. 21. Naïve Réordonné sans optimisation.

Le comportement est similaire à Naive sans optimisation. Les temps d'exécution sont très élevés pour les grandes matrices et l'utilisation des ressources est faible, ce qui indique une inefficacité de la multiplication sans optimisations.

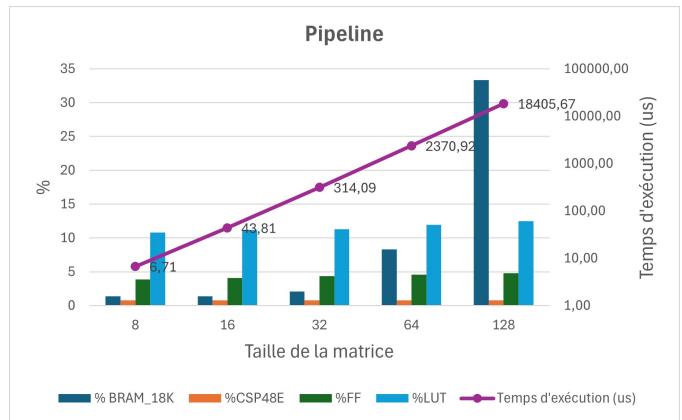


Fig. 22. Naïve Réordonné avec pipeline.

Le temps d'exécution est considérablement réduit, avec une utilisation modérée des ressources, améliorant les performances sans consommation matérielle excessive.

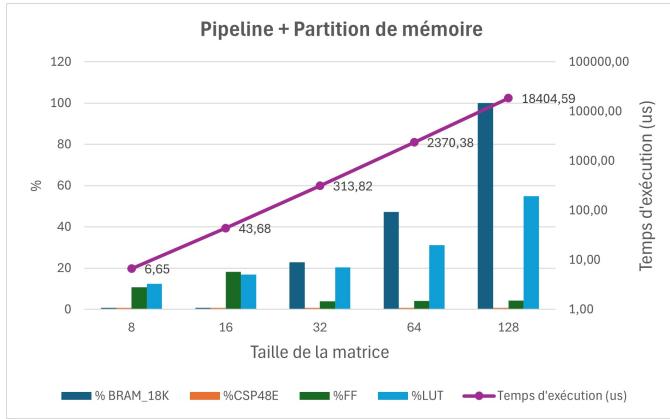


Fig. 23. Naïve Réordonné avec partitionnement de pipelines et de tableaux.

Le temps d'exécution reste quasiment le même, mais l'utilisation des ressources s'élèvent considérablement. Alors, nous comprenons que l'algorithme réordonné est modulé de façon qu'une routine additionnelle de partition de mémoire soit inefficace pour la multiplication de matrices.

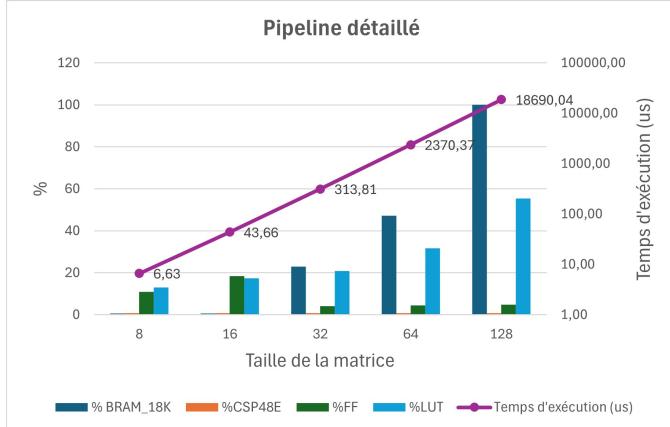


Fig. 24. Naïve Réordonné avec pipeline détaillé.

Des performances sont similaires à la solution précédente. Nous comprenons que cette optimisation aussi utilise des ressources inefficacement par rapport à la première pipeline.

### C. Algorithme des Blocs

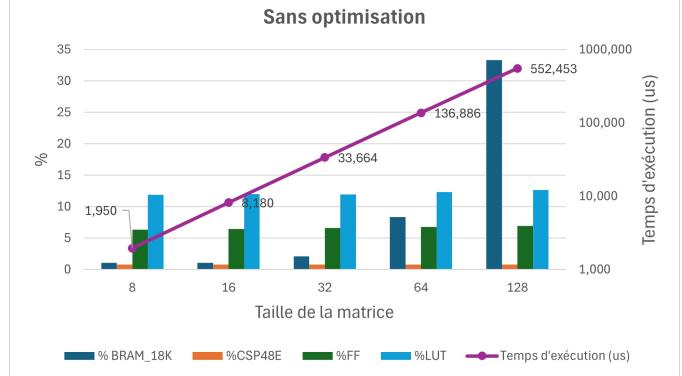


Fig. 25. Blocs sans optimisation.

Les temps d'exécution sont extrêmement faibles par rapport à Naive, du fait de l'efficacité intrinsèque de la méthode par blocs. L'utilisation des ressources est minime, ce qui le rend idéal pour les petites baies, mais pourrait bénéficier d'optimisations pour améliorer encore les performances sur les grandes baies.

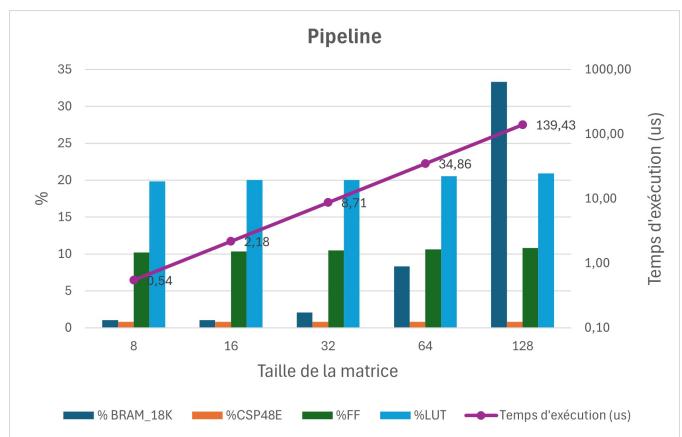


Fig. 26. Blocs avec pipeline.

Le temps d'exécution diminue, et l'utilisation des ressources devient plus homogène avec différents tailles de tableau. La valeur %FF est la seule qu'augmente considérablement avec la taille de matrices, mais est encore réduite en comparaison avec l'algorithme sans optimisation. Cette configuration indique un équilibre entre temps d'exécution et utilisation des ressources.

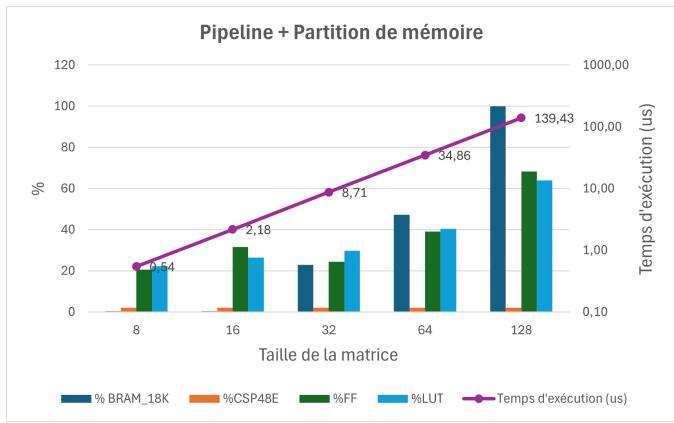


Fig. 27. Blocs avec partitionnement de pipelines et de tableaux.

Il y a beaucoup plus d'utilisation de BRAM et FF, mais avec des temps d'exécution similaires à ceux de l'optimisation précédente. Ceci suggère que cette configuration consomme plus de ressources sans amélioration significative du temps d'exécution.

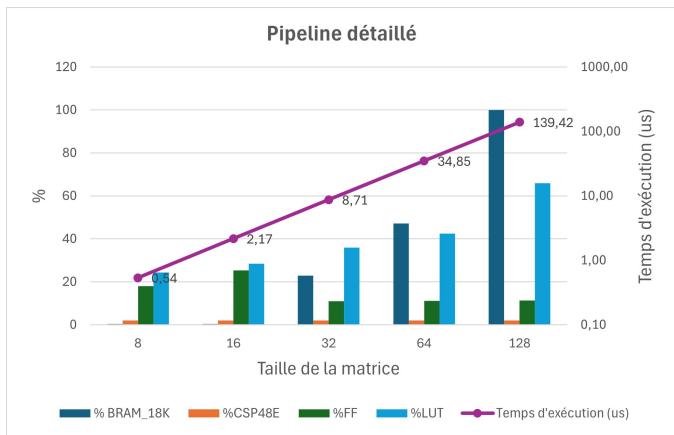


Fig. 28. Blocs avec pipeline détaillé.

Les performances sont pratiquement les mêmes que celles de la solution de baie partitionnée. Cela augmente légèrement l'utilisation des ressources, mais il n'y a pas de grandes différences dans les temps d'exécution.

La multiplication de blocs est nettement plus efficace en termes de temps d'exécution par rapport à Naive, notamment sur les petites matrices. Cependant, les optimisations introduisent une utilisation plus élevée des ressources (FF, LUT, BRAM) sans amélioration drastique des temps d'exécution, notamment dans les configurations plus avancées telles que le partitionnement des baies. Cela suggère que pour certaines applications, la solution sans optimisation pourrait être plus adaptée si une utilisation réduite du matériel est prioritaire.

#### D. Algorithme des Blocs réordonné

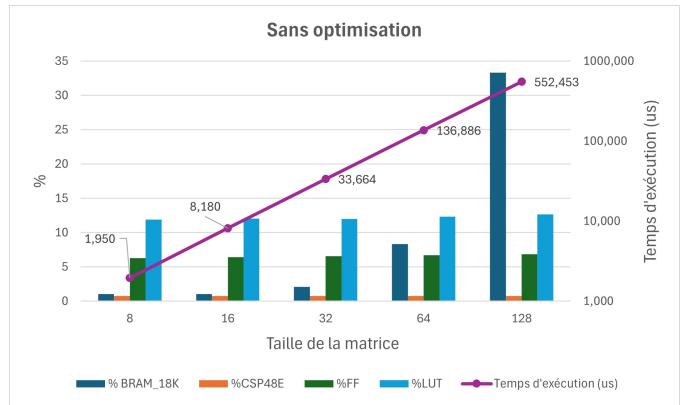


Fig. 29. Blocs Réordonné sans optimisation.

Il se comporte pratiquement de la même manière que la version non réorganisée, ce qui suggère que la réorganisation n'affecte pas de manière significative les petits tableaux.

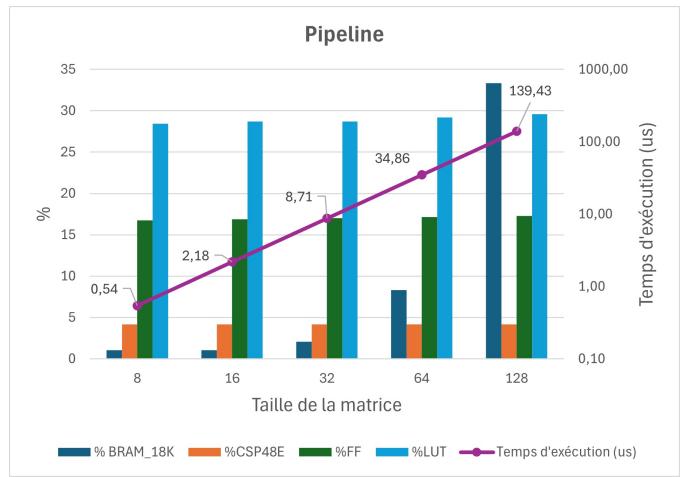


Fig. 30. Blocs Réordonné avec pipeline.

Cela augmente légèrement l'utilisation des ressources par rapport à la version non réorganisée, mais les temps d'exécution est considérablement réduit. Alors, cette optimisation suggère que la réorganisation consomme plus de matériel sans gain de vitesse notable.

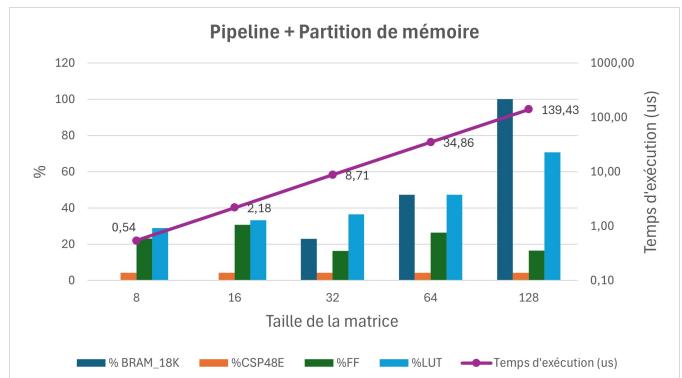


Fig. 31. Blocs Réordonné avec partitionnement de pipelines et de tableaux.

Semblable à la version non réordonnée, la consommation de ressources est plus élevée dans cette optimisation sans qu'une amélioration substantielle des temps d'exécution ne soit observée. Encore, les routines de partition de mémoire augmentent l'utilisation des ressources par rapport à l'optimisation de pipeline simple.

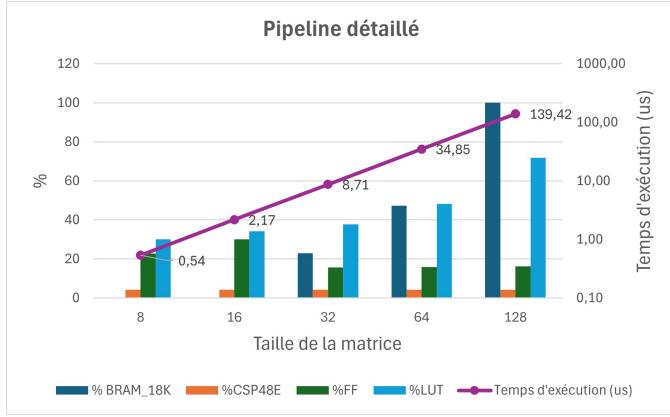


Fig. 32. Blocs Réordonné avec pipeline détaillé.

Bien que les ressources matérielles augmentent par rapport à la version bloc standard, les temps d'exécution sont pratiquement les mêmes.

En général, les optimisations liées à une pipeline représentent un gain significatif de temps, bien comme consomment plus de ressources.

## V. MESURES DES PERFORMANCES ACCÉLÉRATEUR MATÉRIEL SUR FPGA

### QUESTION 4

#### A. Création des IP's Integrator

Après la mise en œuvre d'une première configuration d'un accélérateur matériel de multiplication de matrices, un algorithme TCL a été conçu afin que diverses configurations puissent être générées automatiquement.

Dans cette implémentation, un même fichier contenait tous les différents algorithmes dans un même ensemble de fichiers, lesquels étaient sélectionnés lors de la création du projet en passant comme argument pour l'importation de fichiers le flag `flag_importation`. Cela permettait la compilation d'un fichier avec uniquement la structure nécessaire et rien de plus.

De plus, cette configuration a garanti que la même structure de simulation soit maintenue au cours des différentes simulations, assurant ainsi la comparabilité des résultats. En outre, cela a permis d'exécuter des simulations avec des tailles de matrices arbitraires, offrant une grande flexibilité dans le choix des algorithmes. Lors de l'exécution, un patch correctif de Vivado HLS a été nécessaire pour corriger un bug du système.

Par la suite, un algorithme en Python a été développé pour effectuer l'extraction automatique des informations des rapports de synthèse de chaque configuration et les enregistrer dans un fichier CSV, qui serait ensuite utilisé pour générer les graphiques nécessaires afin de présenter les résultats obtenus.

Ci-dessous se trouve le graphique avec les résultats de la multiplication naïve de matrices.

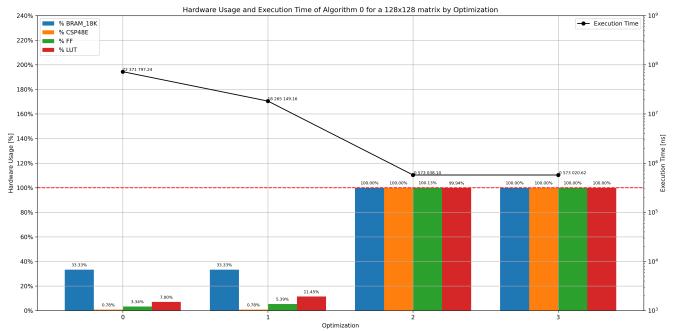


Fig. 33. Naïve Algorithme par Optimisation

Même si, dans cette simulation, l'utilisation des ressources physiques du système était théoriquement optimale en utilisant toutes les ressources disponibles, en pratique, ce circuit ne peut pas être implanté.

#### B. Mettre en Oeuvre sur Vivado

Une fois générés, l'algorithme HLS produisait les IPs nécessaires utilisés dans la représentation du Block Diagram pour synthétiser le circuit à implémenter sur la carte ZedBoard:

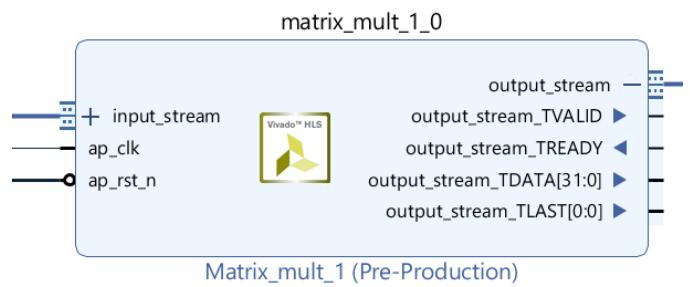


Fig. 34. IP créée

Lorsqu'il est mis en œuvre dans la conception par blocs, le circuit final reliant les autres dispositifs est le suivant :

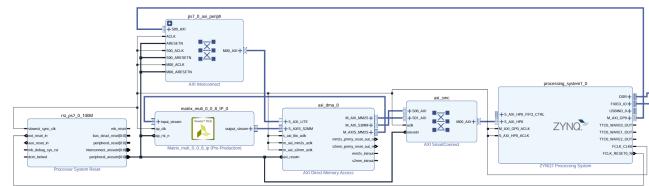


Fig. 35. Diagramme Block Design sur Vivado

Dans ce cas, il s'agit de la représentation du Block Diagram responsable de l'algorithme naïf, sans optimisation pour une matrice de taille 16 x 16. Toutefois, la structure du diagramme ne changera pas pour différentes variantes. Dans ce cas, il a été décidé de concevoir un accélérateur doté d'une seule entrée et d'une seule sortie afin de réduire le circuit nécessaire pour son interface avec l'extérieur, laissant ainsi plus de matériel

disponible pour l'optimisation, même si cela implique un temps accru pour la lecture de deux matrices via la même entrée.

Il a été nécessaire d'utiliser un DMA (Direct Memory Access) pour que le processeur du FPGA puisse stocker les données nécessaires à la réalisation de l'opération en question. Après la création du Block Diagram via Vivado, le circuit a été validé afin que les fichiers matériels nécessaires puissent ensuite être exportés. Après la création du wrapper et la génération du bitstream, un projet SDK a pu être créé.

Une couche de communication a été nécessaire pour gérer l'interaction entre le matériel physique et la partie simulée, mais les résultats n'ont pas pu être lus car la communication avec la carte via le terminal n'a pas réussi.

### C. Mesure des Performances et Utilisation des Ressources

Une fois le design implémenté sur le FPGA, nous devons mesurer le temps d'exécution réel en connectant le processeur ARM9 à l'accélérateur via l'interface AXI. Le logiciel exécuté sur le processeur ARM envoie les matrices à multiplier à l'accélérateur et récupère les résultats une fois la multiplication terminée.

L'idée serait de tester les 4 solutions créées dans hls pour les différentes tailles de matrice, 8, 16, 32, 64, 128 et 256.

Cependant, après avoir validé le projet, généré des produits de sortie, créé HDL Wrapper, Generate Bitstream et exporté les fichiers vers le SDK, nous avons rencontré un problème. Lorsque nous avons exécuté le code dans le SDK, bien que la connexion à la carte ait réussi, aucune réponse n'est apparue dans le terminal.

Pour comparer les résultats, il serait intéressant de montrer les temps d'exécution de chaque solution pour chaque taille de matrice.

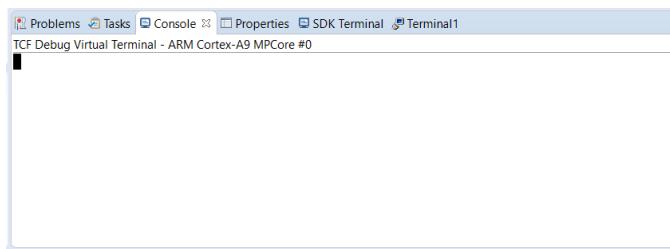


Fig. 36. SDK Terminal après avoir exécuté le code

### D. Utilisation des Ressources

Vivado génère un rapport détaillé de la synthèse matériel, qui inclut :

- Les **LUTs** (Look-Up Tables) utilisées, en pourcentage du total disponible.
- La mémoire **BRAM** (Block RAM) utilisée.
- Les **DSP Slices** (Digital Signal Processing) exploitées pour accélérer les calculs mathématiques complexes.

Ces informations nous permettent d'optimiser la conception tout en assurant que les ressources disponibles sur le FPGA ne sont pas épuisées.

Les données collectées sur le temps d'exécution et l'utilisation des ressources pour différentes configurations nous permettent de tracer les courbes de performance :

- **Courbes de latence** : montrent le temps d'exécution en fonction des configurations.
- **Courbes de Pareto** : illustrent le compromis entre la réduction du temps d'exécution et l'augmentation de l'utilisation des ressources.

La **frontière de Pareto** est essentielle pour optimiser notre système. Elle montre qu'il est difficile d'améliorer une métrique (comme la latence) sans aggraver une autre (comme l'utilisation des ressources). Nos résultats devraient suivre cette logique.

### E. Optimisations Potentielles

Pour améliorer les performances du système, plusieurs optimisations peuvent être appliquées :

- **Unrolling** : permet d'exécuter plusieurs itérations d'une boucle en parallèle, augmentant ainsi le parallélisme et réduisant la latence.
- **Pipelining** : permet aux différentes étapes d'un calcul de s'exécuter simultanément, améliorant ainsi le débit global.
- **Resource Sharing** : partage des ressources matérielles comme les DSP entre plusieurs opérations, ce qui réduit l'utilisation globale des ressources au prix d'une légère augmentation de la latence.

Ces stratégies d'optimisation sont cruciales pour maximiser l'efficacité du système tout en minimisant l'empreinte matérielle.

## RÉFÉRENCES

[1] A Zynq Accelerator for Floating Point Matrix Multiplication Designed with Vivado HLS. Daniele Bagni, A. Di Fresco, J. Noguera, F. M. Vallina. Application Note : Zynq-7000 AP SoC [Online]. Available : <https://docs.xilinx.com/v/u/en-US/xapp1170-zynq-hls>

[2] Vivado Design Suite Tutorial : High-Level Synthesis. XILINX. Nov 18, 2015. Available : <https://docs.xilinx.com/v/u/2015.4-English/ug871-vivado-high-level-synthesis-tutorial>

[3] Vivado Design Suite User Guide : High-Level Synthesis. XILINX. May 4 2021. Available : <https://docs.xilinx.com/v/u/en-US/ug902-vivado-highlevel-synthesis>