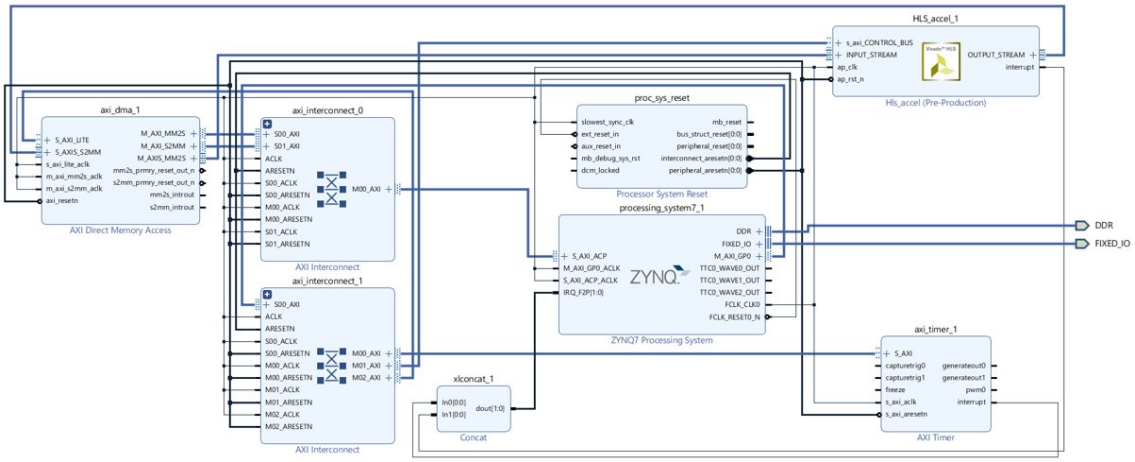
Conception et implémentation d'un accélérateur matériel pour la multiplication matricielle

ROB306 - Systèmes Électroniques Embarqués

Alexis OLIVEIRA DA SILVA, Bastien HUBERT,
Marcelo BRAGA E SILVA, Safa IDRANI



ENSTA Paris - novembre 2022

1 Introduction

Ce projet a le but d'évaluer différentes implémentations de l'application Multiplication de Matrices en fonction des performances et des ressources. Ce rapport est composé des parties suivantes:

- Évaluation des performances sur PC: dans cette partie l'optimisation de l'application est faite pour un processeur généraliste, où l'objectif de performance est limité à seulement la vitesse de calcul, avec toutes les ressources disponibles de l'ordinateur.
- Évaluation de performances sur processeur embarqué ARM9: la même optimisation est faite pour le processeur embarqué, en utilisant l'environnement Vivado SDK.
- Estimation de performances et de ressources par accélérateur matériel sur circuit FPGA: conception d'un accélérateur matériel et estimation des performances (temps de latence, débits, ressources utilisées)
- Mesures de performances par accélérateur matériel sur circuit FPGA: Intégration des IP HLS et ensuite évaluation des performances sur la carte ZedBoard.

2 Évaluation de performances sur PC

Cette partie est consacrée à l'évaluation des performances sur PC de différentes versions de l'algorithme de multiplication de matrices. Après une analyse des différentes méthodes, la meilleure solution (la plus rapide) est retenue pour chaque taille de matrice. Dans cette partie, on souhaite seulement optimiser la vitesse de calcul, en utilisant toutes les ressources disponibles par l'ordinateur en question. Les méthodes sont annoncées par ordre de complexité d'implémentation.

2.1 Architecture PC

Les codes ont été développés sous environnement Windows en C++ avec le compilateur **g++ version std=c++17**. Pour l'évaluation sur PC, l'ordinateur avec les caractéristiques suivantes a été utilisé:

Processeur	AMD Ryzen 7 3700U
Coeurs	4x 2.3 GHz
Cache L1	384 Ko
Cache L2	2 MB
Cache L3	4 MB
RAM	8 GB

Table 1: Architecture PC

2.2 Évaluation

L'évaluation á été faite pour les matrices carrés de nombres entiers (integer de taille 4 octets en C++) de 2 dimensions pour les tailles suivantes:

- 32x32
- 64x64
- 128x128
- 256x256
- 512x512
- 1024x1024
- 2048x2048
- 4096x4096*

* résultats sauf pour les méthodes séquentielles.

Les matrices sont générées pseudo-aléatoirement avec le même grain pour qu'à chaque fois les mêmes données soient traitées. Pour chaque combinaison d'algorithme et taille de matrice, la médiane de 10 temps d'exécution a été prise pour être une mesure plus robuste aux valeurs extrêmes qui apparaissent parfois.

2.3 Codes séquentiels

Les méthodes séquentielles sont les méthodes qui utilisent qu'un seul cœur du CPU pour le calcul. Elles sont plus simples à implémenter car elles ne nécessitent pas de synchronisation et de cohérence de donnés entre cœurs.

La première méthode plus naïve implémente la multiplication classique avec une complexité O(n^3) par l'équation 1:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj} \tag{1}$$

On a exploré le code avec **l'allocation dynamique de mémoire** et avec **l'allocation statique de mémoire**. Les résultats nous montrent que l'allocation statique est la plus rapide, car les zones mémoires de matrices sont garanties à être contigus et donc, l'accès est plus rapide que l'allocation dynamique par pointeurs.

En plus de l'allocation mémoire, on a permuté l'ordre des boucles pour tirer avantage de la **spatialité de cache** pour réduire le nombre de cache-misses. En C++, les matrices d'entiers sont allouées dans la mémoire avec l'ordre Row-Major, comme le montre la Figure 1. De cette façon, l'ordre naïf de multiplication n'est pas optimal, car la boucle la plus interne 'k' parcourt la matrice B en colonne. La solution adaptée a été alors, de mettre comme boucle plus interne 'j', en changeant l'ordre 'i,j,k' pour 'i,k,j'.

La Figure 2 affiche les résultats de cette section.

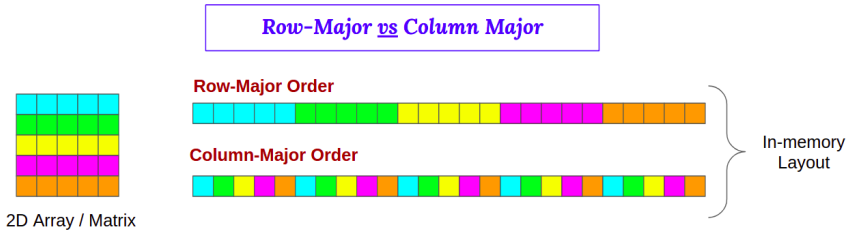


Figure 1: Ordre Row-Major vs Column-Major

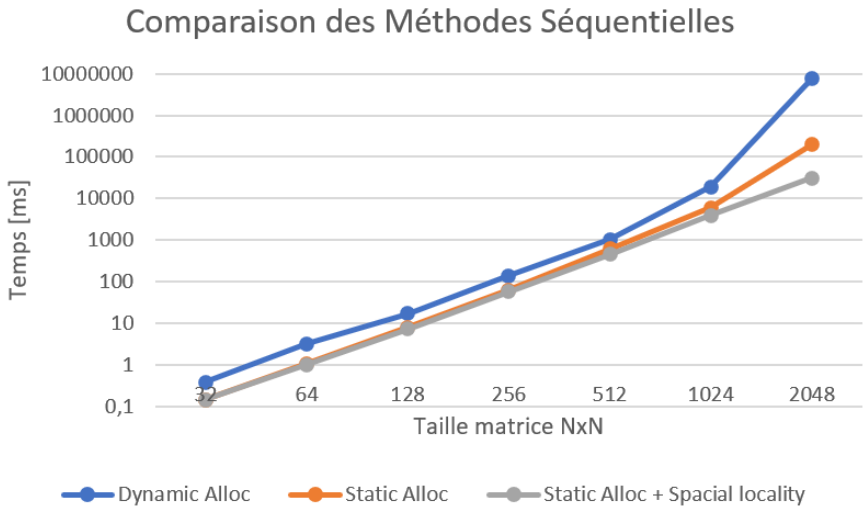


Figure 2: Comparaison des méthodes séquentielles

2.4 Parallélisation avec OpenMP

Le code précédent avec allocation statique et nouvel ordre des boucles a été parallélisé avec les prérogatives pragma de **OpenMP** pour profiter des multicoeurs de la CPU. La multiplication de matrice étant une application où les données d'entrée ne sont pas changées et le calcul d'une ligne ne dépend pas de l'autre, fait de la multiplication un candidat parfait pour être parallélisé. La Figure 3 montre un gain de performances pour les matrices de taille plus grandes que 128. L'utilisation d'OpenMP est très coûteuse pour les petites matrices. Par contre, pour les grandes matrices, le speed-up est presque de 4, ce qui confirme le grand pourcentage parallélisable du code.

2.5 Directives d'optimisation g++

g++ est équipé d'une ensemble de flags (drapeaux) pour optimiser le code exécutable. Le compilateur tente d'améliorer les performances et/ou la taille du code au détriment du temps de compilation et éventuellement de la capacité à déboguer le programme. Les flags suivants ont été utilisés dans la version parallèle du code et leur performance évalués:

- -O1: réduit la taille du code et le temps d'exécution, sans effectuer d'optimisations qui prennent beaucoup de temps de compilation.

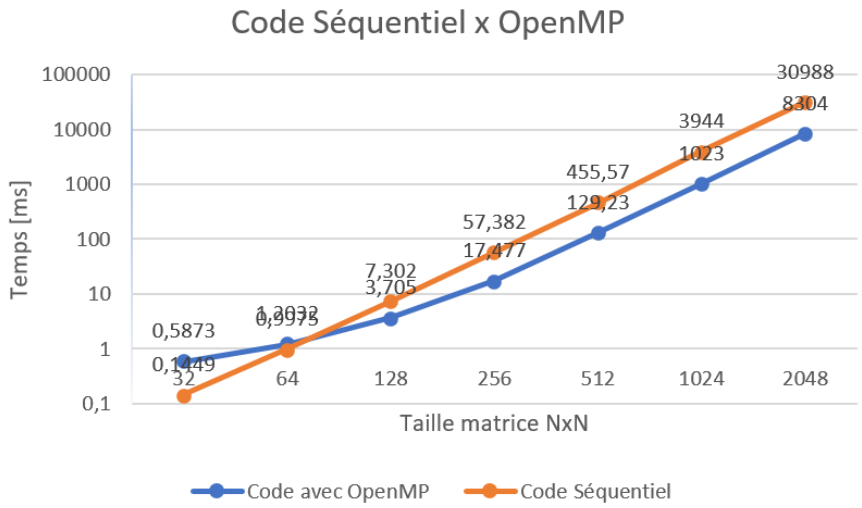


Figure 3: Comparaison code séquentiel et parallèle

- -O2: réalise presque toutes les optimisations prises en charge qui n'impliquent pas de compromis entre taille du code et vitesse.
- O3: Optimise encore plus. Les flags ajoutés plus pertinents pour notre application sont -floop-unroll-and-jam -fpeel-loops -fsplit-loops et -ftree-loop-distribution, qui optimisent les boucles.
- -Ofast: utilise O3 et d'autres flags qui peuvent compromettre le résultat comme -ffast-math qui accélère les calculs mathématiques en arrondissant les valeurs.

Les résultats montrent que la meilleure directive est **-O3**. Les directives -Os et -Og n'étaient pas considérées car leur but est de réduire la taille de l'exécutable et d'optimiser l'expérience de débogage respectivement.

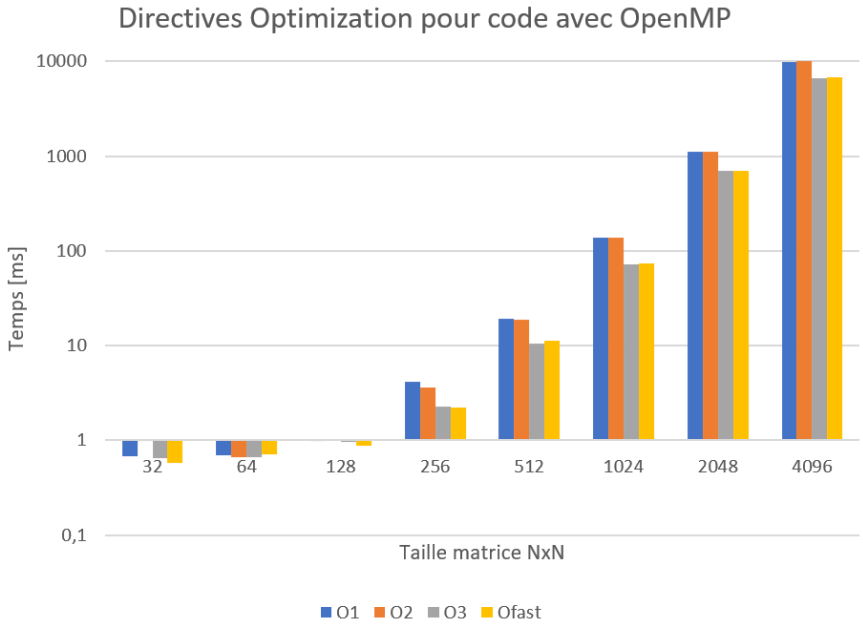


Figure 4: Comparaison flags d'optimisation

2.6 Division par blocs

La division par blocs est une façon d'exploiter la redondance des données pour la mémoire cache. La méthode consiste à diviser la matrice en blocs de taille Nb et à exécuter la multiplication pour chaque sous-bloc. L'évaluation a été faite pour différentes tailles de blocs comme l'affiche la Figure 5.

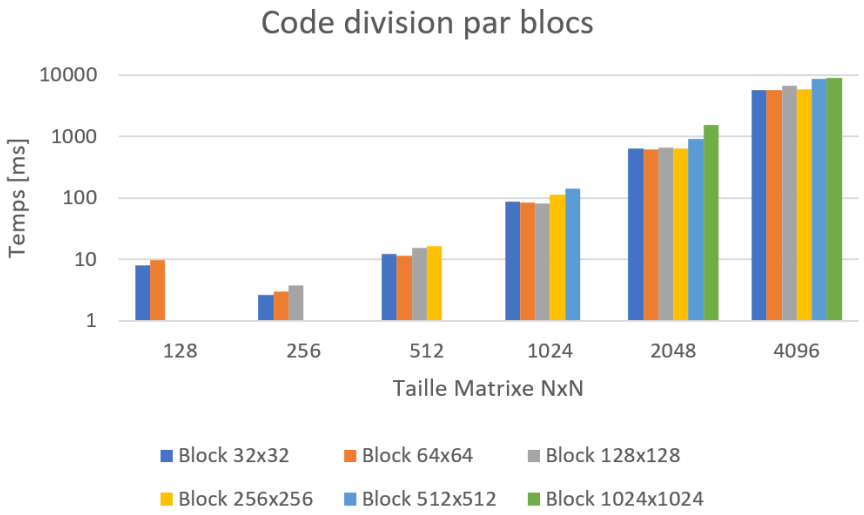


Figure 5: Comparaison temps pour différents taille de bloc

Les résultats obtenus donnent une meilleure performance seulement pour les grandes matrices de taille 2048x2048 et 4096x4096 avec blocs de taille 64x64, car l'algorithme devient coûteux pour les matrices plus petites. On peut estimer grossièrement le remplissage du Cache qui sera remplie par les trois sous-matrices pour chaque thread lancé 2:

$$NxN(tailleMatrice) * 4(tailleEntier) * 3(matricesA, BetC) * 4(NombreThreads) \quad (2)$$

Pour N = 128, on a une utilisation de 768 Ko qui est plus grande que le niveau L1 de cache (384 Ko) et donc, la cache sera écrasé quelques fois par les différents threads. Par contre, N = 64 donne une valeur de 192 Ko qui sera stockée dans le plus haut niveau de cache.

2.7 Méthode de Strassen

La méthode consiste à diviser récursivement les matrices A et B en 4 sous-matrices et à utiliser un ensemble de calculs qui utilisent moins de multiplications et plus d'additions, ce qui rend l'exécution plus rapide. Cette méthode a l'avantage d'avoir une complexité de calcul de $O(n^{2.82})$ au lieu de $O(n^3)$. Par contre, à cause de la création de nouvelles sous-matrices, on a une complexité d'utilisation de mémoire presque 3 fois plus grande que les approches précédentes. Comme la méthode est récursive, on ne pourrait pas allouer statiquement les matrices. Pour cela, la méthode utilisée n'implémente qu'une itération de Strassen. Les résultats montrent une accélération pour des matrices de taille plus grandes que 1024x1024 quand comparés au méthode par blocs (Figure 6).

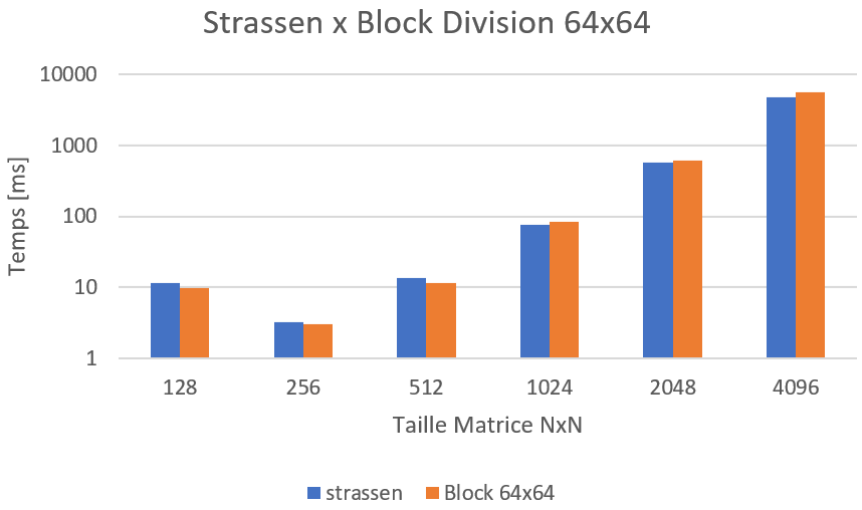


Figure 6: Comparaison Strassen et Division par Bloc

2.8 Conclusion des évaluations sur PC

La Table 2 résume les résultats de cette section. Pour les matrices de taille plus petite que 128x128 entiers, la version séquentielle qui exploite le cache se montre être la meilleure version. Pour des tailles plus grandes, l'ajout des directives OpenMP et multithreading commencent à surpasser les méthodes séquentielles.

On peut conclure que la réduction du temps d'exécution sur PC est beaucoup plus efficace pour les matrices de grande taille, grâce au haut taux d'utilisation des ressources. Pour les matrices petites par contre, toute l'overhead ajouté pour le multithreading et de la complexité des méthodes comme Strassen ne sont pas viables. En plus, le pourcentage de cache miss est trop grand pour les petites matrices. Il faut noter surtout que cette partie n'a pas considéré des indices comme énergie et ressources utilisés, qui sont d'extrêmes importance pour une application embarqué.

Taille Matrice	**Meilleure Performance**	**Temps [ms]**	**Gain par rapport à version de base**
32x32	Séquentiel + Perm boucles + O3	0,0790	4,81x
64x64	Séquentiel + Perm boucles + O3	0,0671	46,86x
128x128	Séquentiel + Perm boucles + O3	0,5405	31,64x
256x256	OpenMP + Perm boucles + O3	2,25	61,85x
512x512	OpenMP + Perm boucles + O3	10,44	96,93x
1024x1024	OpenMP + Perm boucles + O3	72,72	272,84x
2048x2048	OpenMP + Strassen + O3	577,2	13.292x
4096x4096	OpenMP + Strassen + O3	4737	>100.000x

Table 2: Résultats Évaluation sur PC

3 Évaluation de performances sur processeur embarqué ARM9

L'objectif de cette partie était l'évaluation des performances sur un processeur embarqué ARM en réalisant différents types d'optimisations : par algorithme, par code et par option de compilation.

Plus précisément, les différents types d'optimisation ont été fait sur les points suivant:

- sur l'architecture de l'ARM : avec le cache et la prédiction de branchement
- sur l'option de compilation : O0, O1, O2, O3 ou Os.
- sur l'algorithme en lui-même : séquentiel, séquentiel avec inversement de boucle, ou multiplication par bloc.

3.1 Configuration matérielle

La carte Zedboard 7020 a été utilisée pour les différentes simulations. Ses caractéristiques sont les suivantes :

Processeur	ARM 9
Coeurs	2x 667 MHz
Cache L1	32 kB
Cache L2	512 kB
RAM	512 MB

Table 3: Architecture Zedboard 7020

3.2 Évaluation des performances

L'évaluation a été faite pour des matrices carrées (contenant des integer de taille 4 octets) avec les dimensions suivantes:

- 16x16
- 32x32
- 64x64
- 128x128
- 256x256
- 512x512

Les calculs n'ont pas été réalisés pour des matrices de tailles supérieures ou égales à 1024 car les calculs étaient trop couteux en temps (et que chaque calcul doit être effectué 10 fois afin d'obtenir des résultats plus robustes).

Les matrices sont générées pseudo-aléatoirement avec le même seed que dans la partie PC, afin qu'à chaque fois les mêmes données soient traitées. Pour chaque combinaison d'algorithmes et de taille de matrice, la médiane de 10 temps d'exécution a été prise, afin d'obtenir des mesures plus robustes.

Tout d'abord, la comparaison des performances avec/sans cache d'instructions a été réalisée, et les résultats sont présents sur la figure 7.

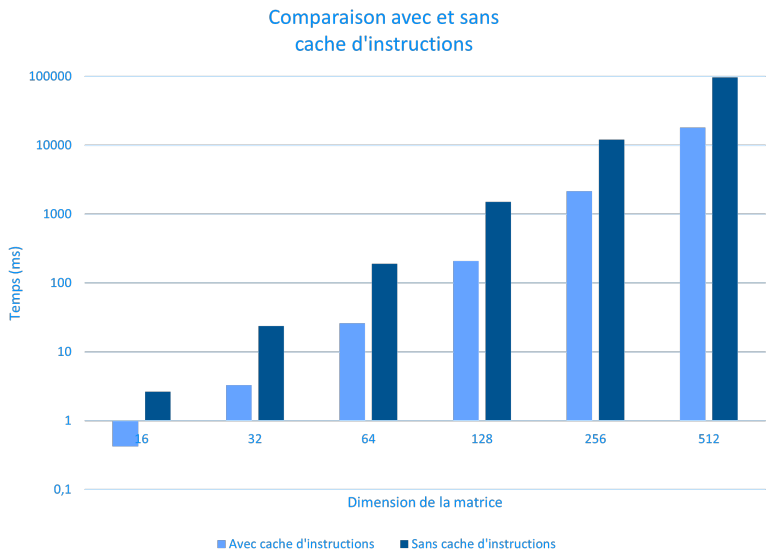


Figure 7: Performances avec/sans cache d'instructions

La désactivation du cache d'instructions se fait avec les instructions suivantes :

```
#include "xtime_l.h"
#include "xil_cache.h"

Xil_ICacheDisable();
```

Pour une matrice carrée de taille 32, on obtient des temps de 3.3 ms et 23.9 ms respectivement avec et sans cache d'instructions, et pour une matrice carrée de taille 512, on obtient des temps d'environ 18101 ms et 96903 ms respectivement avec et sans cache d'instruction. En moyenne on a un facteur 6-7 sur les temps d'exécutions suivant si la configuration est avec ou sans cache d'instruction.

Il a ensuite été fait de même avec la comparaison avec ou sans cache de données. Les résultats sont présent sur la figure 8.

La désactivation du cache de données se fait avec les instructions suivantes :

```
#include "xtime_l.h"
#include "xil_cache.h"

Xil_DCacheDisable();
```

On peut voir qu'on a des temps de calcul encore plus long que sans cache d'instructions. Par exemple pour une matrice carrée de taille 32 on a les temps suivants : 3 ms et 32 ms (respectivement avec et sans cache de données). Pour une matrice carrée de taille 512 les temps sont de 18 101 ms et de 136 308 ms. Le facteur moyen entre avec ou sans cache de données est ici d'environ 10. Il paraît normal d'obtenir de pire résultats en inhibant le cache de données qu'en inhibant celui d'instructions, puisque dans le cas du cache d'instructions, seulement quelques instructions sont sauvegardées dans le cache, tandis qu'avec le cache de données, des morceaux entiers sont mis en cache et accessible bien plus rapidement.

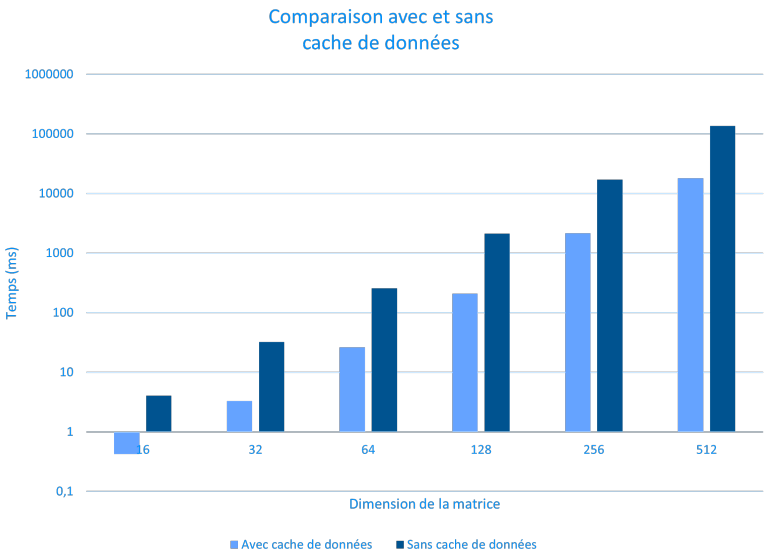


Figure 8: Performances avec/sans cache de données

Pour finir avec cette partie cache, nous avons ensuite comparé les performances avec et sans cache d'instructions et de données.

Les résultats obtenus sont ceux de la figure 9.

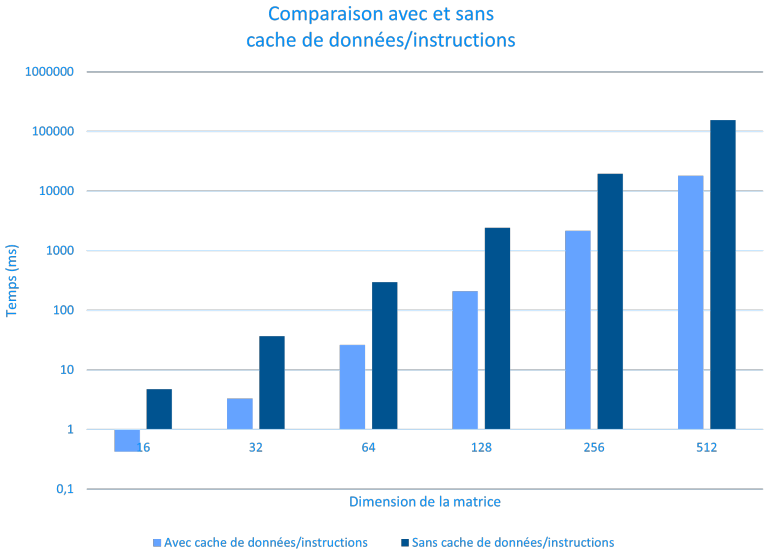


Figure 9: Performances avec/sans cache d'instructions/données

Le cas où le cache de données et d'instructions sont inhibés correspond au pire cas par rapport à ce qui a été vu avant. Pour une matrice carrée de taille 32 on a des temps de 3,28 ms et 37,1 ms et pour une de taille 512 des temps de calcul de 18 101 ms et 155 900 ms respectivement avec et sans cache d'instructions/données.

Enfin, les performances avec et sans MMU (l'unité de gestion mémoire) ont été comparées. Pour faire cela, il a d'abord fallu invalider les TLB (mémoire cache du processeur utilisée par l'unité de gestion mémoire), le tableau des prédicteurs de branchement et vider les caches de données avant de désactiver la MMU.

Les instructions pour désactiver la MMU sont les suivantes :

```
#include "xtime_l.h"
#include "xil_cache.h"
#include "xil_mmu.h"

Xil_DisableMMU();
```

Les performances obtenues sont présentes en figure 10

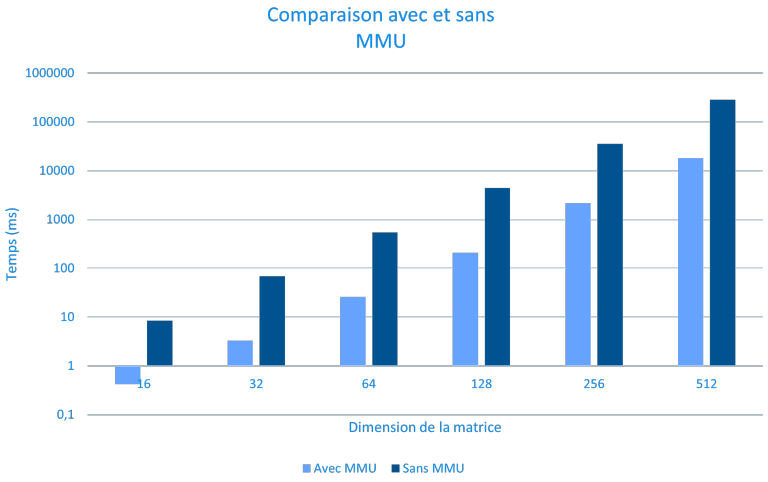


Figure 10: Performances avec/sans MMU

Maintenant, nous allons nous intéresser aux performances pour différentes options de compilation.

Sur la figure 11, on peut voir les performances pour différentes options de compilation.

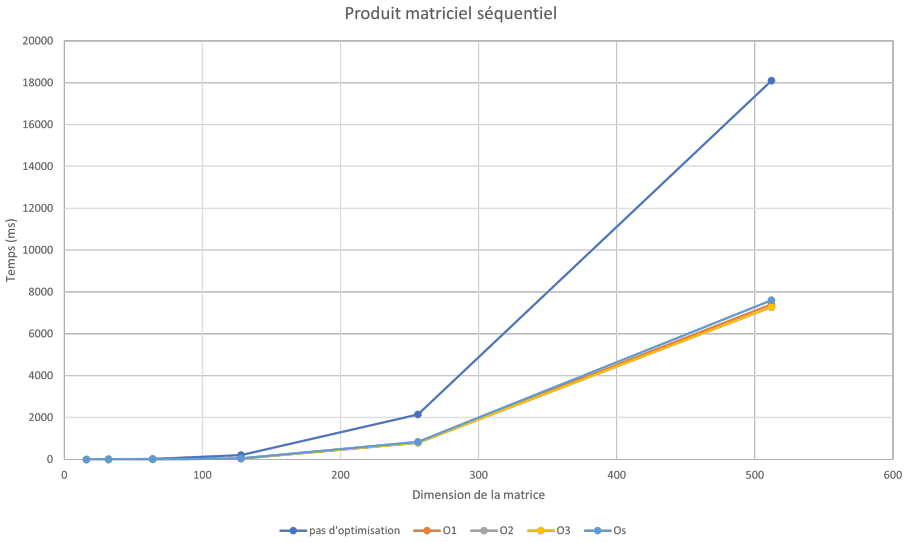


Figure 11: Performances pour différentes options de compilation

On peut voir ici que pour les 3 options de compilations pour le produit matriciel séquentiel on est assez proche,

mais on peut tout de même voir que -O3 donnera des temps de calcul bien plus rapides, et ce sera prépondérant pour des tailles de matrices très grandes par exemple de 2048.

Dans un deuxième temps, les performances de l'algorithme séquentiel inversé ont été comparées, c'est-à-dire en permutant l'ordre des boucles, en passant de i j k à i k j (figure 12).

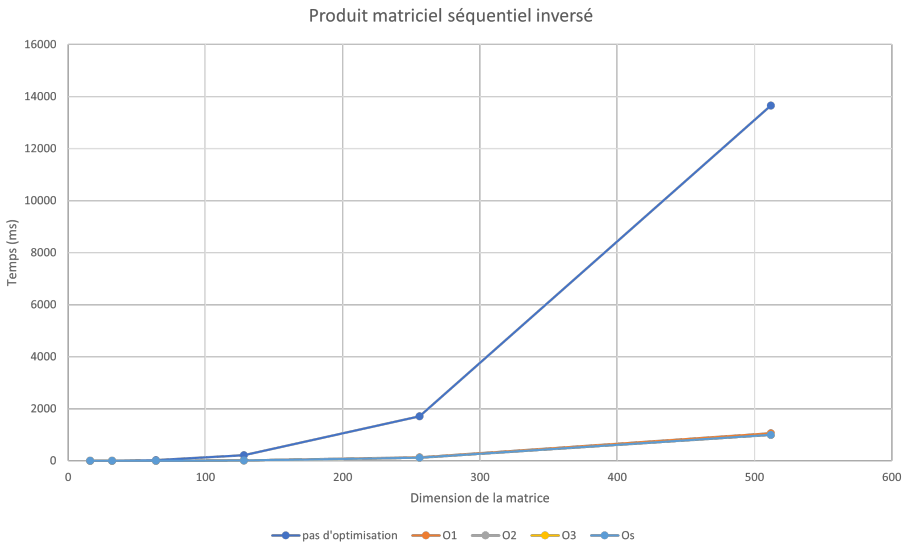


Figure 12: Performances algorithme séquentiel inversé

Ici c'est l'optimisation -Os qui domine légèrement, mais qui reste proche de -03.

Enfin, les comparaisons des performances pour le produit par blocs ont été réalisées et les résultats sont présents sur la figure 13.

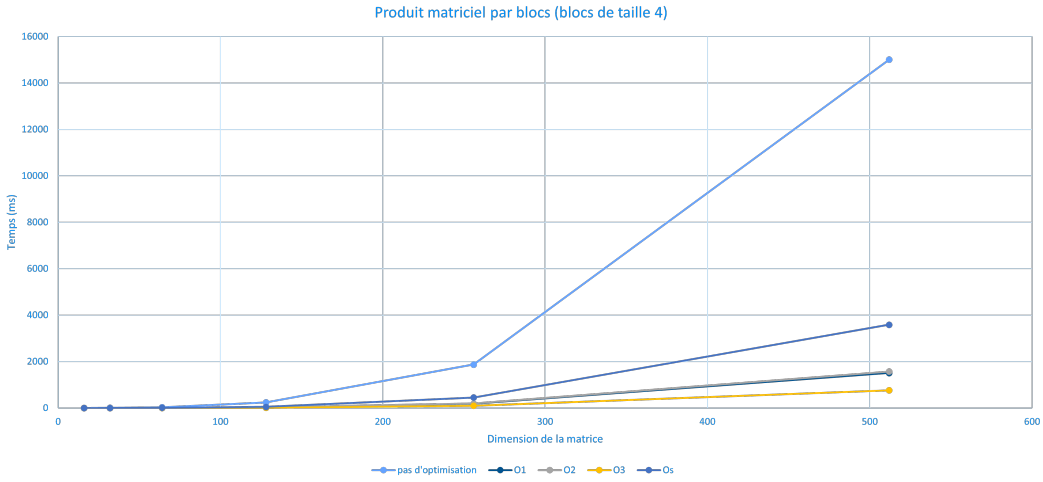


Figure 13: Performances algorithme produit par blocs

Le produit par bloc a été réalisé arbitrairement avec des blocs de taille 4. On voit également que le calcul le plus rapide est atteint avec l'option -O3.

Enfin, une comparaison globale a été effectuée entre les 3 algorithmes pour les différentes optimisations figure

14.

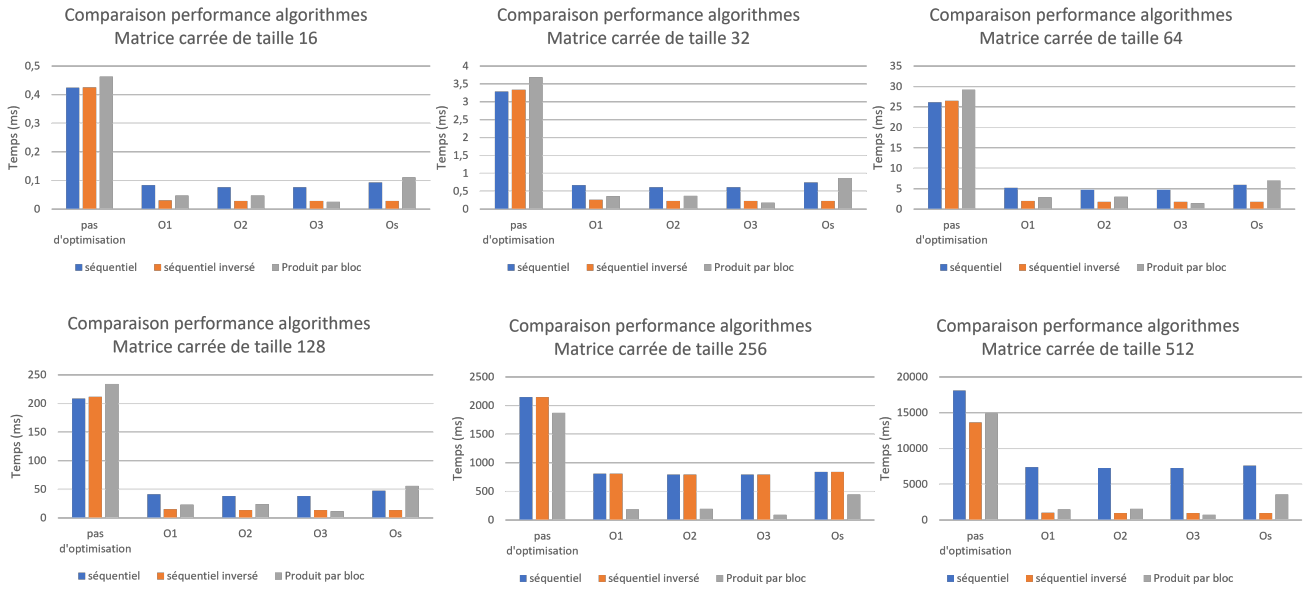


Figure 14: Comparaison des algorithmes

On voit vraiment là l'intérêt d'utiliser le produit par bloc pour les grandes tailles de matrices, ainsi que l'option O3 du compilateur. On peut voir ici que pour des petites tailles de matrice, on va perdre du temps à effectuer un produit par bloc et qu'on optera plutôt sur une inversion de boucle jusqu'à des matrices carrées de taille 128, et qu'au-delà il est clairement intéressant de passer à du produit par bloc où par exemple pour 256 on divise quasiment d'un facteur 10 le temps de calcul. Il faudra clairement privilégier l'option de compilation O3 dans tous les cas.

Pour finir, voici un tableau récapitulatif (figure 15) des comparaisons de temps d'exécution entre les différents algorithmes pour l'optimisation -O3 en pourcentage de temps de calcul avec l'algorithme original sans optimisation (-O0).

Comparaison du temps d'exécution						
Dimension	16	32	64	128	256	512
Algorithme originale (O0)	100%	100%	100%	100%	100%	100%
Algorithme originale (O3)	18,13%	18,58%	18,44%	18,29%	36,90%	40,21%
Multiplication par bloc (O3)	5,95%	5,63%	5,52%	5,50%	4,39%	4,19%
Séquentiel inversé (O3)	6,85%	7,14%	6,81%	6,67%	5,69%	5,54%

Figure 15: Comparaison des temps d'exécutions

En somme, les meilleurs temps de calcul sont obtenus pour de la multiplication par bloc (blocs de taille 4 ici) avec l'optimisation -O3.

3.3 Comparaison des performances PC/ARM

Pour finir cette partie, j'ai réalisé une comparaison entre les performances obtenues avec le PC et celles obtenues avec la Zedboard (ARM 9) pour l'algorithme original de produit matriciel et celui avec inversion (en -O3) :

Comparaison de l'algorithme originale (O0)						
Dimension	16	32	64	128	256	512
PC (ms)	0,03	0,18	1,06	7,88	62,31	609,49
ARM (ms)	0,42	3,29	26,13	208,42	2150,27	18101,23
Ratio	12,3	18,4	24,7	26,5	34,5	29,7

Figure 16: Comparaison de l'algorithme originale (O0)

Comparaison de l'algorithme avec inversion (O3)						
Dimension	16	32	64	128	256	512
PC (ms)	0,65	0,66	0,97	2,25	10,44	72,72
ARM (ms)	0,03	0,23	1,78	13,91	122,37	1003,26
Ratio	0,04	0,35	1,84	6,18	11,72	13,80

Figure 17: Comparaison de l'algorithme avec inversion (O3)

Hormis pour des matrices de dimension 16 et 32 avec l'algorithme avec inversion (en -O3) où le ARM 9 affiche de meilleures performances, on peut voir que les temps de calculs les plus faibles sont obtenus avec le PC et non pas avec l'ARM, avec des ratio temps de calculs ARM/PC montant jusqu'à 29,7 pour des matrices de dimensions 512 et l'algorithme original en -O0, et jusqu'à un ratio de 13,8 pour des matrices de dimension 512 et l'algorithme avec inversion (en -O3).

L'objectif du projet est finalement d'obtenir à la fin de meilleurs résultats en implémentant l'accélérateur matériel sur la Zedboard. Je recomparerai alors les performances entre le PC et l'ARM 9 à la fin du projet, dans un tableau final, pour confirmer que notre solution permet bien un gain de performance.

4 Estimation de performances et de ressources par accélérateur matériel sur circuit FPGA

4.1 Structure du code

Afin d'accélérer l'exécution de la fonction de multiplication matricielle, nous allons à présent concevoir un accélérateur matériel qui réalisera la travail de la fonction C++ `matrix_mult` décrite dans les sections précédentes. Dans cette partie, nous utilisons cette fonction comme une boîte noire, de sorte que n'importe laquelle des versions et optimisations exposées dans les sections 2 et 3 peut être réutilisée.

Nous rappelons que le prototype de cette fonction est le suivant :

```
void matrix_mult( mat_a a[IN_A_ROWS][IN_A_COLS],
                  mat_b b[IN_B_ROWS][IN_B_COLS],
                  mat_prod prod[IN_A_ROWS][IN_B_COLS] );
```

Avec `mat_a`, `mat_b` et `mat_prod` des types définis spécialement pour la manipulation de `matrix_mult`, et `IN_A_ROWS`, `IN_A_COLS`, `IN_B_ROWS` et `IN_B_COLS` des variables globales, tous définis dans un header.

Puisque cette section a pour objectif de passer de la fonction logicielle `matrix_mult` à un accélérateur matériel, nous devons envelopper (*wrapper*) cette fonction dans une autre, `matrix_mult_wrapper`, définie comme suit :

```
void matrix_mult_wrapper( AXI_VAL INPUT_STREAM[IN_A_ROWS * IN_A_COLS + IN_B_ROWS * IN_B_COLS],
                          AXI_VAL OUTPUT_STREAM[IN_A_ROWS * IN_B_COLS] ) {

  // Define matrices
  mat_a a[IN_A_ROWS][IN_A_COLS];
  mat_b b[IN_B_ROWS][IN_B_COLS];
  mat_prod prod[IN_A_ROWS][IN_B_COLS];

  int i, j, k;

  // Stream in the 2 input matrices
  int A_SIZE = IN_A_ROWS * IN_A_COLS;
  for( i = 0; i < IN_A_ROWS; i++ ) {
    for( j = 0; j < IN_A_COLS; j++ ) {
      #pragma HLS PIPELINE II=1
      k = i * IN_A_COLS + j;
      a[i][j] = pop_stream<int, 4, 5, 5>( INPUT_STREAM[k] );
    }
  }

  int B_SIZE = IN_B_ROWS * IN_B_COLS;
  for( i = 0; i < IN_B_ROWS; i++ ) {
    for( j = 0; j < IN_B_COLS; j++ ) {
      #pragma HLS PIPELINE II=1
```

```
      k = i * IN_B_COLS + j + A_SIZE;
      b[i][j] = pop_stream<int, 4, 5, 5>( INPUT_STREAM[k] );
    }
  }

  // Do multiplication
  matrix_mult( a, b, prod );

  // Stream in the 2 input matrices
  for( i = 0; i < IN_A_ROWS; i++ ) {
    for( j = 0; j < IN_B_COLS; j++ ) {
      #pragma HLS PIPELINE II=1
      k = i * IN_B_COLS + j;
      OUTPUT_STREAM[k] = push_stream<int, 4, 5, 5>( prod[i][j], k == A_SIZE + B_SIZE - 1 );
    }
  }
}
```

Remarquons tout d'abord que cette fonction est bien plus proche de la réalité matérielle du circuit sur lequel elle sera programmé, car ses entrées et sorties sont de type `AXI_VAL`, qui est un type spécifiquement conçu pour la transmission de données via des protocoles AXI : ce n'est qu'au sein de la fonction que sont définies les entrées et sorties de `matrix_mult`.

L'analyse du code de `matrix_mult_wrapper` montre que la fonction s'exécute en 4 temps :

- augmentation du niveau d'abstraction avec la définition des variables `mat_a`, `mat_b` et `mat_prod`
- lecture de `mat_a` et `mat_b` grâce à la fonction `pop_stream`, dont nous détaillerons le fonctionnement en sous-section 4.3
- exécution du produit matriciel à proprement parler
- écriture de `mat_prod` grâce à la fonction `push_stream`, dont le fonctionnement sera également décrit en sous-section 4.3

Toutefois, cette fonction reste trop abstraite pour pouvoir être directement convertie en accélérateur matériel, car elle ne spécifie ni les ports physiques ni les protocoles de communication. C'est pourquoi nous définissons une dernière fonction :

```
void HLS_accel( AXI_VAL INPUT_STREAM[IN_A_ROWS * IN_A_COLS + IN_B_ROWS * IN_B_COLS],
                AXI_VAL OUTPUT_STREAM[IN_A_ROWS * IN_B_COLS] ) {

  // Map ports to Vivado HLS interfaces
  #pragma HLS INTERFACE s_axilite port=return bundle=CONTROL_BUS
  #pragma HLS INTERFACE axis port=INPUT_STREAM
  #pragma HLS INTERFACE axis port=OUTPUT_STREAM

  matrix_mult_wrapper( INPUT_STREAM, OUTPUT_STREAM );
}
```

L'utilisation des `#pragma HLS` permet de définir des protocoles de communication de type AXI entre l'accélérateur et le reste du circuit, c'est donc ce que nous utilisons pour finaliser le code de notre accélérateur.

L'étape suivante consiste à faire synthétiser le code C++ et à l'exporter au format Verilog ou VHDL pour pouvoir l'intégrer dans un design Vivado, comme le fera la section 5.

4.2 Optimisations de calcul

Il existe de nombreux outils dans Vivado HLS pour optimiser un accélérateur matériel, mais deux d'entre eux sont particulièrement utiles : *pipeline* et *array reshape*.

Le pipeline est une optimisation matérielle qui permet de réduire drastiquement le chemin critique en insérant des bascules D dans la logique de l'accélérateur. De plus, Vivado HLS procède automatiquement au dépliage des boucles sur lesquelles du pipeline a été effectué, ce qui permet de dupliquer un fragment de circuit logique pour exécuter simultanément toutes les itérations de ladite boucle. Il est donc nécessaire de trouver un compromis entre le temps d'exécution qu'on gagne en calculant directement toutes les itérations d'une boucle et les ressources requises pour dupliquer le circuit autant de fois que voulu.

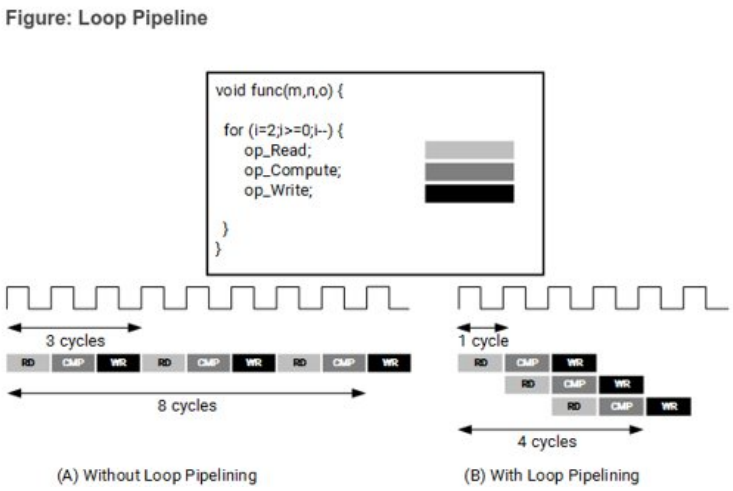


Figure 18: Schéma explicatif du fonctionnement du pipeline

L'array reshape est une optimisation similaire au dépliage de boucle, mais portant sur les données. Il s'agit de modifier la structure d'une liste de données pour permettre la lecture simultanée de plusieurs données, ce qui a également pour résultat de réduire le nombre de cycles d'horloge nécessaire au calcul du produit matriciel. Ici encore, on fait face à un compromis temps/ressources.

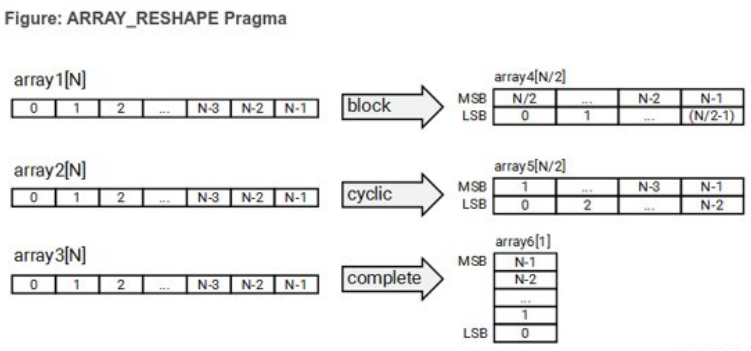


Figure 19: Schéma explicatif du fonctionnement de l'array reshape

4.3 Optimisations de transmission

Le protocole AXI (Advanced eXtensible Interface) est un protocole de communication standard développé par ARM, c'est donc celui que nous utilisons pour établir la communication entre l'accélérateur matériel et le processeur ARM embarqué sur la Zedboard. Cependant, plusieurs versions de ce protocole existent, parmi lesquelles on peur citer AXI4, AXI4-lite et AXI4-stream.

Le protocole AXI4 est très lourd, et n'est pas recommandé pour l'utilisation que nous faisons du bus. De même, si AXI4-lite est plus léger que sa version standarde, le débit qu'il propose reste insuffisant car il reste basé sur une communication *memory-mapped*. Un tel protocole de communication nécessiterai la mise en place d'un système de type maître-esclave complet, ce qui est bien trop complexe pour ce que nous souhaitons faire.

En revanche, le protocole AXI4-stream, conçu pour transmettre des flux de données à haut débit, est parfait pour notre cas. Le protocole n'a en outre besoin que d'un accès DMA (Direct Memory Access) simple et d'un couple de ports MM2S (memory-mapped to stream) et S2MM (stream to memory-mapped) pour fonctionner, ce qui le rend d'autant plus facile à implémenter par la suite dans le design Vivado.

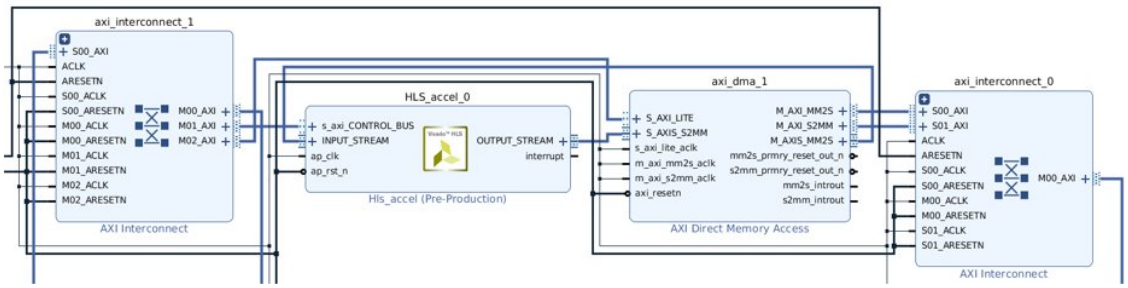


Figure 20: Partie du design Vivado incluant l'accélérateur matériel et les blocs de communication

4.4 Choix des paramètres et des métriques

Afin de correctement représenter le compromis temps/ressources, des configurations issues de différents paramètres et optimisations ont été simulées. Ainsi, 5 solutions ont été proposées :

- solution1 : sans optimisation
- solution2 : pipeline de la boucle Prod
- solution3 : pipeline de la boucle Col
- solution4 : array reshape des matrices A et B et pipeline de la boucle Prod
- solution5 : array reshape des matrices A et B et pipeline de la boucle Row

Ces solutions ont toutes été testées sur différentes données :

- data1 : matrices 5×5
- data2 : matrices 16×16
- data3 : matrices 64×64
- data4 : matrices 128×128
- data5 : matrices 512×512
- data6 : matrices 1024×1024

Les simulations ainsi effectuées donnent accès au nombre de cycles nécessaire à l'exécution de l'accélérateur matériel, ainsi qu'à la période minimale de l'horloge cadençant le circuit. La synthèse C donne en outre accès au nombre de blocs RAM, DSPs, FFs et LUTs utilisés. Ces valeurs permettent de définir deux métriques, une pour le temps et une pour les ressources, comme suit :

$$T(c) = log_{10}\left(nb_{cycles}(c) * T_{ck}(c) \right)$$
$$R(c) = log_{10}\left(nb_{BRAM}(c) + nb_{DSP}(c) + nb_{FF}(c) + nb_{LUT}(c) \right)$$

4.5 Résultats et analyse

Commençons par étudier les résultats obtenus pour chaque métrique séparément. Il faut d'abord noter que la Vivado HLS ne parvient pas à synthétiser solution5 pour des matrices de 64×64 ou plus en raison du dépliage total du produit matriciel : il faudrait disposer de taille 64^3 circuits multiplicatifs et 64^2 circuits additifs, ce qui représente une charge de calcul trop importante pour le logiciel, et dépasse largement les ressources disponibles sur la FPGA.

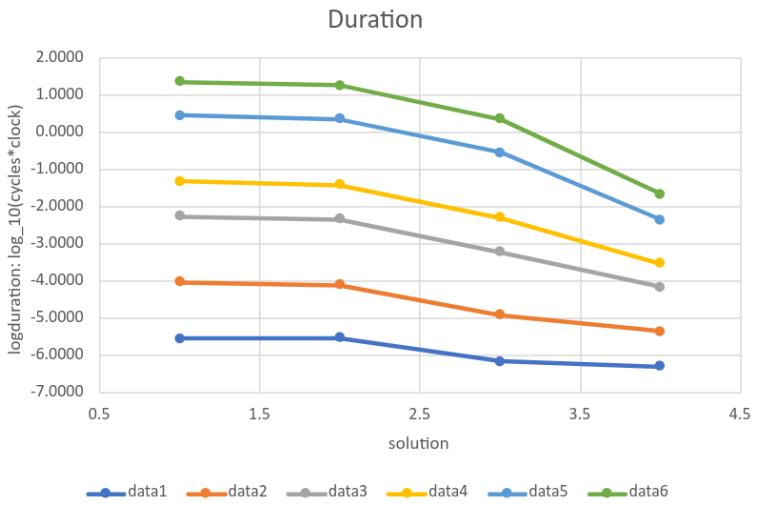


Figure 21: $T(c)$ pour différentes configurations

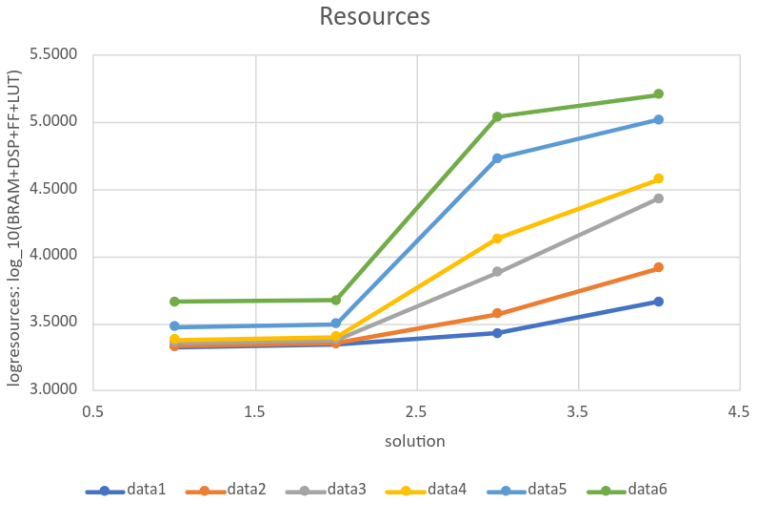


Figure 22: $R(c)$ pour différentes configurations

Comme on pouvait s'y attendre, les valeurs de T et R sont des fonctions croissantes de la taille des matrices. En effet, des matrices de dimensions importantes prennent plus de temps à traiter, mais requièrent également plus de ressources à cause des loop unrolling et des array reshape. De plus, chaque métrique montre une tendance inverse sur l'utilisation des optimisations : plus on optimise des boucles extérieures (c'est-à-dire plus on optimise de boucles simultanément), plus le temps d'exécution diminue et l'utilisation des ressources augmente.

Ces tendances contradictoires sont l'expression directe du compromis temps/ressources qui été mis en lumière dans la sous-section précédente, et nous amènent à comparer directement les deux métriques.

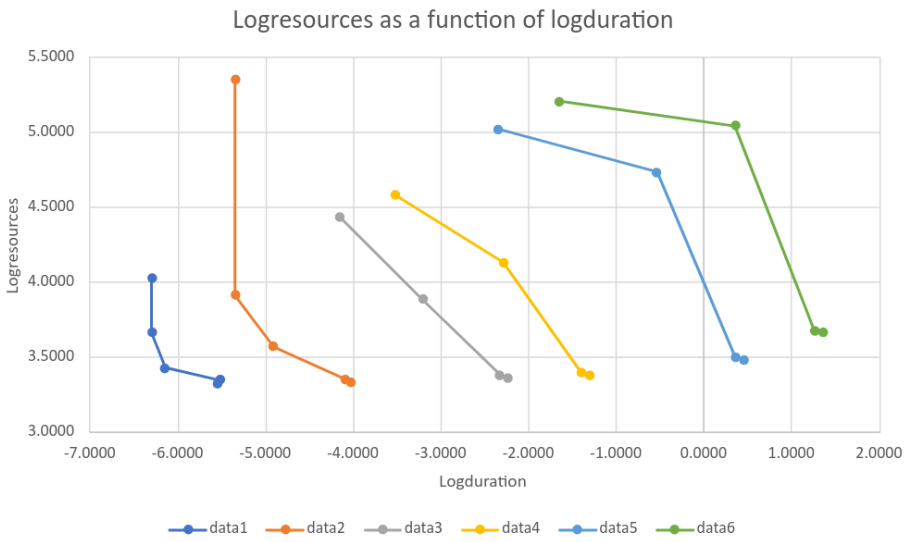


Figure 23: $R(c)$ en fonction de $T(c)$

Avec cette représentation, on voit clairement apparaître des fronts de Pareto pour chaque jeu de données, ce qui affirme une fois de plus la présence d'un compromis entre temps d'exécution et ressources utilisées. Puisque toute ressource non utilisée est perdue sur un circuit FPGA, le choix à faire est simple ici : tant que les ressources nécessaires sont disponibles, on prend la solution la plus rapide. Dans notre cas, pour toutes les tailles de matrices testées, la solution 4 est la solution à choisir.

5 Mesures de performances par accélérateur matériel sur circuit FPGA

L'objectif de cette étape est premièrement l'intégration des IP HLS développés dans la phase précédente sur un bloc design en utilisant Vivado pour mesurer les performances sur la carte ZedBoard. Apporter un axe d'optimisation sur le design pour améliorer le calcule de l'accélérateur.

5.1 Intégration de l'IP sur le blog design Vivado :

Pour créer un bloc design contenant l'IP HLS, on commence par l'importation de l'IP sur vivado. Puis on crée un schéma bloc qui permet de connecter l'accélérateur au processeur ARM9 via AX14-STREAM.

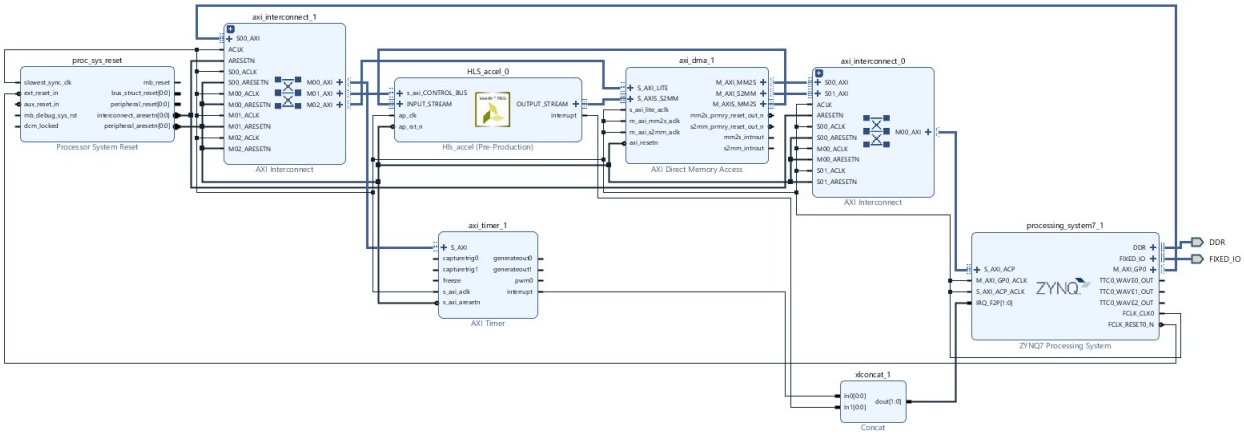


Figure 24: Bloc Design avec interface en AX14-STREAM

L'AXI-timer est utilisé pour calculer le nombre des cycles exécutés en temps réel.

5.2 Intégration SDK de l'IP :

Pour l'intégration SDK il est implicite de générer les différents fichiers wrappers et BitStream qui seront chargés sur la carte.

Pour l'intégration SDK il est implicite de générer les différents fichiers de sorties, wrappers et le fichier hardward qui seront chargés sur la carte. Après le chargement de BitStream sur FPGA, on adapte le code C pour que le processeur ARM9 interagit avec l'IP et nous permet de mesurer les performances.

5.3 Résultat :

Différents IP avec des interfaces en AXI4-Stream, correspondant aux tailles de matrices n × n avec n = 64, 128 ,256, avec pour chaque taille une version basique et 4 versions optimisées ont été intégrés pour tester et calculer leurs performances.

D'après le calcul des performances, on remarque une diminution du temps d'exécution. L'IP HLS a un effet plus

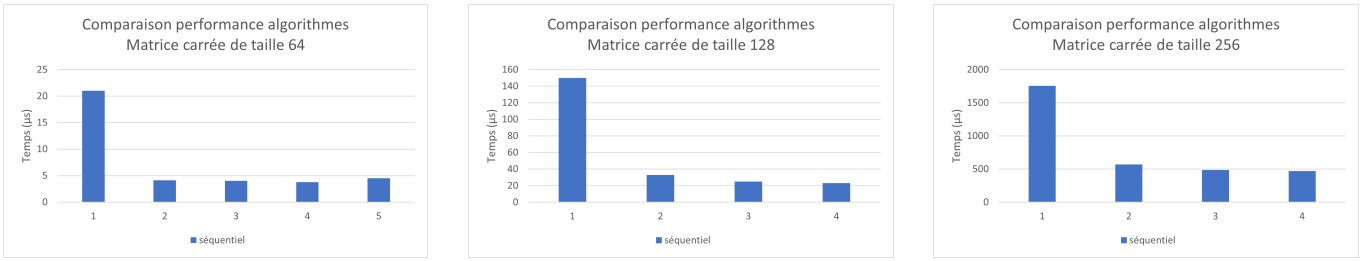


Figure 25: Résultat des performances après l'intégration de l'accélérateur

significatif avec les matrices de grande taille et moins remarquable avec les matrices de petite taille et c'est pour cela que dans cette étape on n'a pas travaillé avec les matrices de taille 16.

5.4 Comparaison performance ARM9 et FPGA- HLS

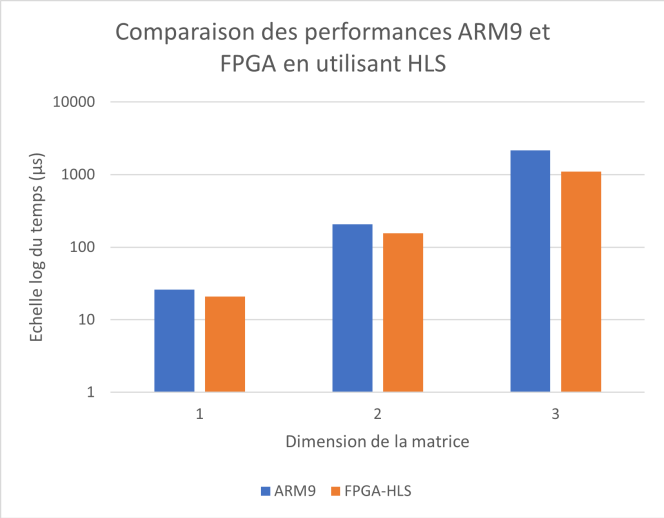


Figure 26: Comparaison des performances ARM9 et FPGA-HLS

Pour les dimensions des matrices :

* 1 : représente la taille 64

* 2 : représente la taille 128

* 3 : représente la taille 256

On constate le calcule avec en utilisant l'accélérateur HLS a des temps d'exécution inférieur à celle de ARM9 même pour les matrices de petite taille. L'amélioration de la performance est plus significative sur les matrices de plus grande taille.

Pour la partie intégration, il reste toujours des directives pour l'amélioration puisqu'on a une consommation dynamique élevée comme elle montre la figure 6. Cette consommation atteint 92%

Pour la réduction de consommation des ressources, il est recommandé d'ajouter des niveaux Pipeline à la logique

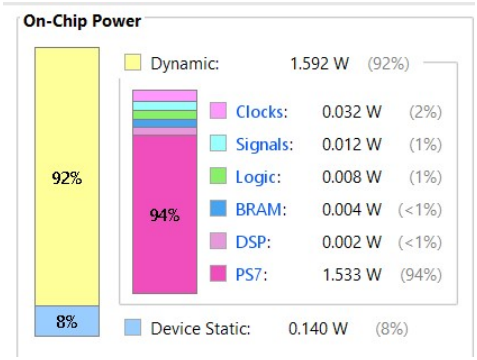


Figure 27: rapport de l'énergie pour une matrice de dimension 64 et solution 1

et de minimiser les signaux de contrôle asynchrone.

5.5 Conclusion :

Comme montre le tableau récapitulatif de calcule de performance, L'implémentation d'accélérateurs dans notre application de calcule améliore les performances de temps d'exécution pour les différentes dimensions de matrices mais cette amélioration est plus significative avec les grandes tailles. En effet, si on compare l'accélérateur avec le calcule avec le PC, on une légère amélioration pour les dimensions 16 et 32 mais elle devient très remarquable pour la dimension 512. On peut dire que l'accélérateur convient plus aux gros volumes de données.

Cependant, la solution sur ARM9 elle est moins bonne que le PC pour les différentes tailles de donnée.

Comparaison de l'algorithme originale (O0)						
Dimension	16	32	64	128	256	512
PC (ms)	0,03	0,18	1,06	7,88	62,31	609,49
ARM (ms)	0,42	3,29	26,13	208,42	2150,27	18101,23
ARM avec accélérateur matériel (ms)	0,01	0,02	0,09	0,39	2,28	5,70
Ratio	12,3	18,4	24,7	26,5	34,5	29,7
Ratio ARM accélérateur / PC	0,03	0,10	0,41	1,63	8,40	35,35
Accélération ARM	73,28	145,23	294,63	535,66	944,84	3175,20

Figure 28: Tableau comparatif final