Accélérateur Matériel avec HLS
Hardware-Software codesign

Zheyi SHEN[1], Dajing GU[2], Zhaoyi GUAN[3], and Hanin HAMDI[4]

[1]U2IS, ENSTA Paris, France

***Résumé*—Ce projet est une comparaison entre différentes méthodes de calcul avancé. On souhaite évaluer les performances de calcul sous PC, un processeur ARM9 et un accélérateur matériel implémenté sur un circuit FPGA. Pour ce faire, on s'appuie sur l'application de la multiplication de matrices : une tâche qui requiert de bonnes aptitudes de calcul. Dans la première partie de ce papier, on souhaite faire des optimisations algorithmiques de l'application ainsi que des optimisations logicielles sous PC. La deuxième partie est dédiée pour le calcul avec le processeur ARM9 de la carte *Zedboard*. On étudiera l'influence de divertes parties du processeur à l'instar du cache, ainsi que ses performances en terme du temps d'exécution. La troisième partie est réservée pour la conception d'un accélérateur matériel sous *Vivado HLS* ainsi que l'estimation de ses performances. Enfin, la quatrième partie est une implémentation sur la FPGA de cet IP Core HLS à travers un réseau AXI4-Stream. On calculera également le temps réel d'exécution pour chaque solution HLS.**

***Mots clés*—hardware software codesign, accélération matériel, FPGA, Xilinx, Vivado, Vivado HLS, ARM9**

I. Introduction

Dans ce projet, on souhaite évaluer les performances en temps d'exécution et en ressources d'une fonction dans 3 configurations possibles: sur le processeur généraliste sur PC, sur le processeur embarqué ARM9 et en accélérateur matériel sur circuit reconfigurable FPGA XC7Z020 disponible sur *Zedboard*.

La fonction que l'on a choisi est la multiplication de matrices qui est représentée dans le code ci-dessous:

```
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])
{
  // Iterate over the rows of the A matrix
   Row: for(int i = 0; i < IN_A_ROWS; i++) {
       // Iterate over the columns of the B
           matrix
       Col: for(int j = 0; j < IN_B_COLS; j++) {
          prod[i][j] = 0;
          // Do the inner product of a row of A
              and col of B
          Product: for(int k = 0; k < IN_B_ROWS;
               k++) {
             prod[i][j] += a[i][k] * b[k][j];
          }
       }
   }
}
```

C'est un algorithme séquentiel original sur lequel on va faire la modification pour améliorer la performance de la fonction.

II. Évaluation des performances sur PC

Dans cette partie, on évalue notre code sur PC. La table II.1 représente les paramètres principaux de performance de PC que l'on a utilisé pour cette partie.

CPU Name	Intel(R) Core(TM) i7-8550 CPU @1.80GHz
Frequency	3100 MHz
Cache L1	32KB I + 32KB D
Cache L2	256 kB
Cache L3	8192 kB
Memory	8.00 GB
OS	Ubuntu
Compiler	GCC

TABLE II.1
Description du PC

On teste plusieurs algorithmes pour la multiplication de matrices et on voit l'impact algorithmique. De plus, en faisant varier les options d'optimisation en compilation, on compare les performances de différentes options (Aucune optimisation, O1, O2, O3, Os, Og et Ofast). Les différences entre ces options sont introduites ci-dessous:

- O1 : Dans l'hypothèse de ne pas affecter la vitesse de compilation, utiliser certains algorithmes d'optimisation pour réduire le temps d'exécution.
- O2 : Au détriment un peu la vitesse de compilation, en plus de toutes les optimisations effectuées par -O1, utiliser presque tous les algorithmes d'optimisation supportés par la configuration cible pour améliorer la vitesse d'exécution.

- O3 : En plus de la mise en œuvre de toutes les options d'optimisation de -O2, de nombreux algorithmes de vectorisation sont généralement adoptés pour améliorer le degré d'exécution parallèle du code (utiliser le pipeline, le cache, etc. dans les CPUs modernes).
- Os : Optimiser pour la taille. Activer toutes les optimisations -O2 sauf celles qui augmentent souvent la taille du code.
- Og : Sélectionner des options d'optimisation qui n'entrent pas en conflit avec l'option -g. Fournir un niveau d'optimisation raisonnable, tout en générant de meilleures informations de débogage et en respectant les normes linguistiques.
- Ofast : Il ne suivra pas strictement la norme de langue. En plus d'activer toutes les options d'optimisation -O3, certaines optimisations seront également utilisées pour certaines langues, par exemple, -ffast-math

Il y a aussi plusieurs paramètres qui peuvent influencer le temps d'exécution, notamment la dimension de matrices, le *blocksize* dans l'algorithme de multiplication par blocs, le nombre de threads dans *OpenMP*, etc.

Dans ce projet, on utilise la fonction `clock()` pour mesurer le temps d'exécution et pour chaque configuration, on calcule le moyen de dix tests. On trace des courbes ou dessine des histogrammes pour comparer la performance de différents choix et l'objectif est de sélecter ainsi la meilleure configuration.

A. Algorithme séquentiel original

L'algorithme original sur lequel on a travaillé est montré dans le code ci-dessous:

```
void matrix_mult_sequentiel(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])
{
    for(int i = 0; i < IN_A_ROWS; i++){
        for(int j = 0; j < IN_B_COLS; j++){
            int temp = 0;
            for(int k = 0; k < IN_B_ROWS; k++){
                temp += a[i][k] * b[k][j];
            }
            prod[i][j] = temp;
        }
    }
}
```

Ici, par rapport au code dans la première section, on a ajouté la variable intermédiaire $temp$ pour réduire les lectures et les écritures du tableau (array) $prod$, cela peut réduire le temps d'exécution.

Les courbes de performance en temps sont représentées dans la figure II.1. C'est évident que

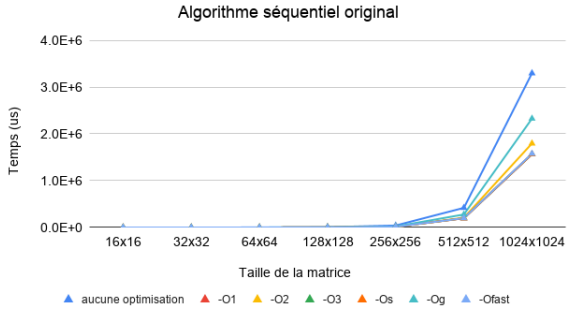


Fig. II.1. Le temps d'exécution de l'algorithme séquentiel original

le temps d'exécution augmente à mesure que la dimension de matrices augmente. En outre, toutes les options en compilation peuvent améliorer la performance en temps en comparant avec le test sans optimisation ($\sim 3.3s$ pour les matrices 1024×1024). Les options O3, Os, Ofast sont les plus rapides ($\sim 1.6s$ pour les matrices 1024×1024).

B. Algorithme séquentiel amélioré

Dans l'algorithme séquentiel original, l'accès au cache n'est pas continu, c'est-à-dire, pour chaque itération (i, j, k), on accède à $a[i][k]$ et $b[k][j]$ et ensuite on accède à $a[i][k+1]$ et $b[k+1][j]$ dans l'itération suivante. Si on considère que la matrice est stockée en ligne, alors l'accès en colonne à la matrice b est discontinu. Cela entraînera un faible taux de réussite des accès au cache (cache hit ratio). Donc, pour accélérer, il est nécessaire de rendre l'accès au cache aussi continu que possible. On propose ainsi un algorithme séquentiel amélioré comme ci-dessous:

```
void matrix_mult_sequentiel_2(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 temp;
    for(int i = 0; i < IN_A_ROWS; i++){
        for(int k = 0; k < IN_B_ROWS; k++){
            temp = a[i][k];
            for(int j = 0; j < IN_B_COLS; j++){
                prod[i][j] += temp * b[k][j];
            }
```

```
        }
    }
}
```

Dans cet algorithme, on change l'ordre des itérations j et k. De cette façon, la variable $temp = a[i][k]$ est accédée IN_B_COLS fois continûment. De plus, l'accès à la matrice b est en ligne. Les courbes de performance en temps de cet algorithme sont représentées dans la figure II.2.

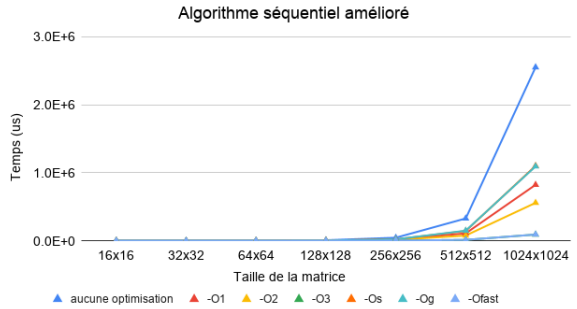


Fig. II.2. Le temps d'exécution de l'algorithme séquentiel amélioré

En comparant avec la figure II.1, le temps d'exécution est plus petit pour l'algorithme amélioré. Dans ce cas, les options O3 et Ofast sont les plus rapides et l'optimisation de ces deux options est beaucoup plus évidente. Par exemple, pour les matrices 1024×1024, les options O3 et Ofast prennent environs $90ms$ pour le calcul, mais l'algorithme avec aucune optimisation prend environs $2550ms$.

C. Algorithme de multiplication par blocs

Un autre algorithme qu'on a proposé est l'algorithme de multiplication par blocs. On sait que si le cache reste le même, plus la matrice est grande, plus il est difficile de l'insérer dans le cache, et plus il y aura de manquant de cache (cache miss). Le cache hit ratio est donc faible.

Dans le PC, il y a trois types de cache, L1 est le plus petit, L3 est le plus grand. Plus la capacité du cache est grande, plus le temps d'accès est long. Donc, diviser la matrice en blocs peut la faire entrer dans un cache plus rapide. Le code de l'algorithme par blocs est représenté ci-dessous.

```
// This method can only be used when IN_A_ROWS
    = IN_A_cols = IN_B_ROWS = IN_B_COLS
void matrix_mult_block(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])
{
    for (int i_block=0; i_block<IN_A_ROWS;
        i_block+=BLOCKSIZE){
        for (int j_block = 0; j_block<IN_B_COLS
            ; j_block+=BLOCKSIZE){
            for (int i = 0; i < IN_A_ROWS; i++)
                {
                for (int j = j_block; j <
                    j_block + BLOCKSIZE; j++) {
                    for (int k = i_block; k <
                        i_block + BLOCKSIZE; k
                        ++){
                        prod[i][j] += a[i][k]*
                            b[k][j];
                    }
                }
            }
        }
    }
}
```

Le temps d'exécution pour le bloc de dimension 8×8 avec différentes options en compilation est montré dans la figure II.3.Dans ce cas, on peut voir que les options O3 et Ofast sont les plus rapides ($\sim 300ms$).

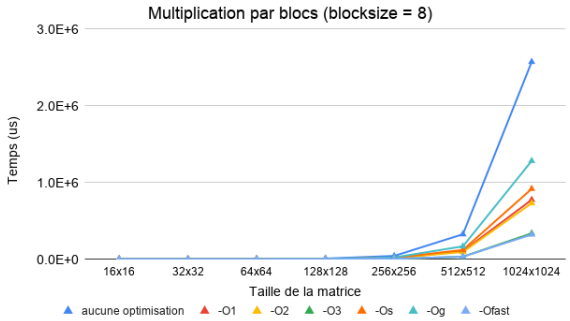


Fig. II.3. Le temps d'exécution de l'algorithme par blocs

L'algorithme de multiplication par blocs a un paramètre *blocksize* qui influencera la performance. On teste donc les dimensions de bloc 4×4, 8×8 et 16×16. Pour les comparer clairement, on met les courbes en *log*.

Selon la figure II.4, les configurations avec $blocksize = 8$ et $blocksize = 16$ sont plus vite que $blocksize = 4$, parce que quand on réduit la taille du bloc, le nombre de cycles augmente et cela ralentit l'exécution. De plus, on voit que lorsque la dimension de matrice est plus petite que 512, la configuration de $blocksize = 8$ est plus rapide mais quand la dimension est supérieure à 512, celle de $blocksize = 16$ a une meilleure performance.

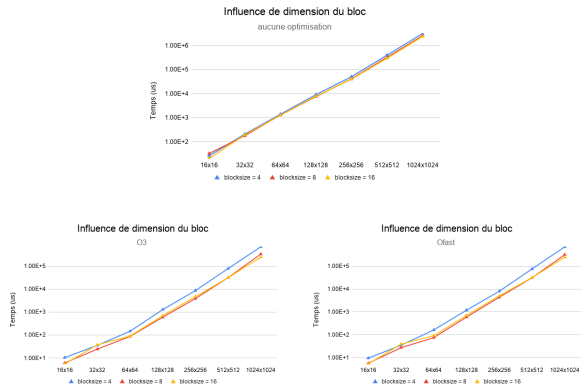


Fig. II.4. Comparaison de performances avec la dimension du bloc différentes

D. Comparaison de trois algorithmes

Dans cette section, on fait la comparaison entre les trois algorithmes que l'on a introduit dans les sections précédentes. Les résultats sont représentés sur la figure II.5, d'où la couleur bleue représente l'algorithme séquentiel original, le rouge représente l'algorithme séquentiel amélioré et la multiplication par blocs est représentée par l'orange. Ici, on choisit 8×8 comme la taille du bloc.

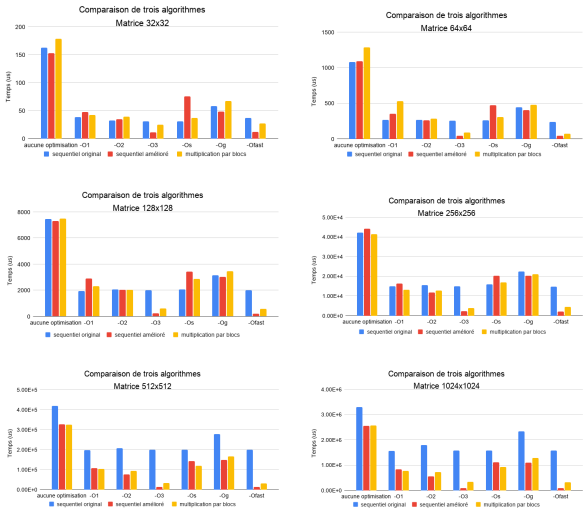


Fig. II.5. Comparaison de trois algorithmes

D'après la figure II.5, quand la dimension de matrice est assez grande, l'optimisation de l'algorithme séquentiel amélioré et par blocs est beaucoup plus évidente. Quand la dimension de matrice est inférieure à 128×128, on ne voit pas beaucoup de différences entre ces trois algorithmes pour la plupart d'options d'optimisation en compilation. Sous l'option Os, l'optimisation de l'algorithme a même un contre-effet. Cependant, sous les options O3 et Ofast, l'effet est toujours évident.

Quand la dimension de matrice est supérieure à 512, les deux algorithmes diminuent tout le temps d'exécution en comparant avec l'algorithme original. Sous les options O3 et Ofast, le niveau d'optimisation est plus élevé.

Selon ces résultats, on en déduit que les options O3 et Ofast sont les plus rapides et que l'algorithme séquentiel amélioré a une meilleur performance pour réduire le temps d'exécution surtout sous les options O3 et Ofast. On se concentre donc sur l'algorithme séquentiel amélioré dans la section suivante.

E. Utilisation de OpenMP

Le processeur du PC que l'on utilise pour cette partie est de quatre coeurs, alors on utilise OpenMP pour réaliser une programmation multi-threads. Le code est comme ci-dessous:

```
1  void matrix_mult_openMP(mat_a a[IN_A_ROWS][
       IN_A_COLS],
2                          mat_b b[IN_B_ROWS][
                               IN_B_COLS],
3                          mat_prod prod[IN_A_ROWS
                               ][IN_B_COLS])
4  {
5      # pragma omp parallel num_threads(
           thread_count)
6      {
7          int my_rank = omp_get_thread_num();
8          int i,j,k,temp;
9          int my_first_row = my_rank*local_size;
10         int my_last_row = (my_rank+1)*
               local_size - 1;
11         for(i = my_first_row; i <= my_last_row;
               i++){
12             for(k = 0; k < IN_B_ROWS; k++){
13                 temp = a[i][k];
14                 for(j = 0; j < IN_B_COLS; j++){
15                     prod[i][j] += temp * b[k][j
                           ];
16                 }
17             }
18         }
19     }
20 }
```

On utilise la directive `# pragma omp parallel` pour identifier les segments parallèles dans le code. La variable *thread_count* définit le nombre de threads. La figure II.6 montre la comparaison entre l'algorithme séquentiel amélioré sans openMP et avec openMP et on compare aussi les différents nombres de threads.

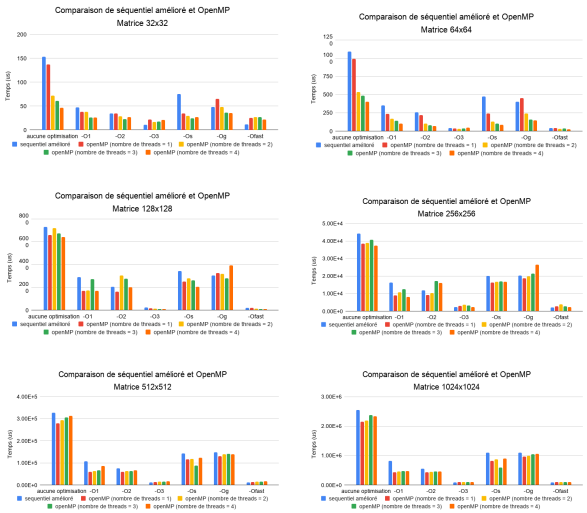


Fig. II.6. Comparaison de algorithme séquentiel amélioré et OpenMP

D'après la figure II.6, quand la taille de matrice est petite, l'effet de multi-threads est plus évident. Lorsque la dimension de matrice est élevée, l'effet est plus faible. On observe que les options O3 et Ofast sont toujours les plus rapides entre ces sept options, et sur ces deux cas, on ne voit pas beaucoup de différences entre l'algorithme avec openMP et sans OpenMP. C'est parce que les options O3 et Ofast ont déjà fait l'amélioration sur l'exécution parallèle.

En outre, on voit un phénomène que le temps d'exécution de l'algorithme avec OpenMP n'est pas stable surtout pour les petites matrices. C'est parce que pour OpenMP, il faut demander des threads, le temps d'exécution est donc dépendant de la rapidité de la distribution de ressources sur PC.

F. Performance en temps pour la meilleure configuration

En analysant toutes les données des tests, on a trouvé quatre meilleures méthodes d'accélération. La table II.2 montre la comparaison du temps d'exécution entre l'algorithme original et ces quatre méthodes. Pour bien comparer, on a mis leurs pourcentages en considérant l'original comme 100%.

Dimension de la matrice	16x16	32x32	64x64	128x128	256x256	512x512	1024x1024	Average
Séquentiel original sans optimisation	100%	100%	100%	100%	100%	100%	100%	100%
Séquentiel amélioré, -O3	11.65%	6.76%	4.05%	2.96%	5.36%	2.86%	2.75%	5.20%
Séquentiel amélioré, -Ofast	26.42%	7.19%	3.72%	2.69%	4.78%	2.86%	2.76%	7.20%
Séquentiel amélioré, 4 threads, -O3	13.92%	12.84%	4.40%	1.14%	5.76%	3.85%	3.14%	6.44%
Séquentiel amélioré, 4 threads, -Ofast	13.92%	13.39%	2.50%	1.15%	5.28%	4.20%	3.13%	6.22%

TABLE II.2
RATIO D'OPTIMISATION DE 4 MEILLEURES CONFIGURATIONS

On sélectionne la configuration de l'algorithme séquentiel amélioré sans openMP sous l'option O3 qui a la plus basse valeur moyenne comme la meilleure configuration. La figure II.7 montre la courbe de performance en temps pour cette configuration.

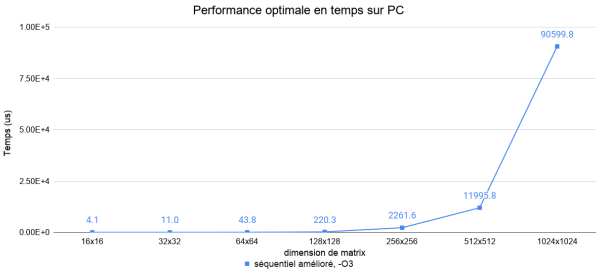


Fig. II.7. Le temps d'exécution de l'algorithme séquentiel amélioré

III. ÉVALUATION DES PERFORMANCES SUR PROCESSEUR EMBARQUÉ ARM9

Dans cette partie, on porte le code C de notre application sur le procersseur ARM9 en exploitant l'environnement Vivado 2019 SDK et on fait des optimisations sur la micro-architecture, l'option de compilation (O0, O1, O2, O3 et Os) et l'algorithme. Le fichier de tête "$xtime\_l.h$" est inclus pour mesurer le temps d'exécution du code et le nombre d'itération est 10 afin de réduire l'erreur.

A. Optimisation sur la micro-architecture du processeur

Maintenant on va lancer le code sur le processeur ARM (Advanced RISC Machine ou Acorn RISC Machine) du zedboard. Par rapport à d'autres séries de processeurs, l'architecture d'ARM est relativement plus simple et sa consommation d'énergie est plus faible. Dans cette section, on va faire des optimisations sur la micro-architecture du processeur.

1) L'influence du cache: Parmi toutes les opérations du processeur, l'accès à la mémoire est l'opération la plus fréquente. Mais la vitesse de fonctionnement de la Mémoire Principale est toujours inférieure à celle du processeur. Pour résoudre le problème causé par la différence entre la vitesse d'écriture et de lecture de données dans CPU et dans la Mémoire Principale, le cache est ajouté, comme présenté dans la figure III.1.

Quand le CPU veut trouver des données, il va tout d'abord chercher dans le cache. Si les données dont l'on a besoin sont dans le cache, le CPU peut directement les obtenir. Sinon, le cache va les chercher dans la Mémoire Principale, ensuite le cache va être mis à jour et le CPU

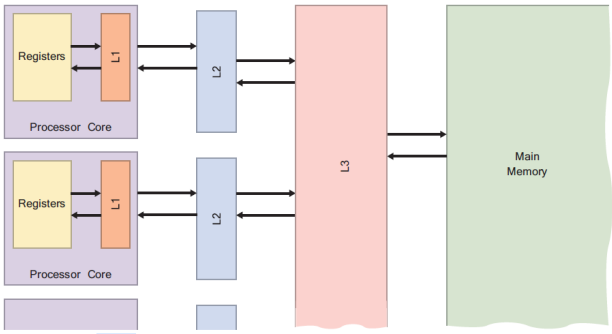


Fig. III.1. Le fonctionnement du cache

va obtenir tout ce dont il a besoin. L'existence du cache réduit le nombre de fois où le CPU accède directement à la Mémoire Principale. Cela a amélioré la performance du code.

```
#include "xil_cache.h"

Xil_DCacheDisable();
```

Sur l'ARM, l'utilisation du cache est validée en défaut, à l'aide du code ci-dessus, on a réussi à invalider l'utilisation du cache.

Dimension de matrix	16x16	32x32	64x64	128x128	256x256	512x512
temps (μs) sans cache	2.53×10^3	1.95×10^4	1.53×10^4	1.26×10^5	8.04×10^7	6.44×10^8
temps (μs) avec cache	1.89×10^2	1.44×10^3	1.13×10^4	9.09×10^4	1.07×10^6	8.76×10^6

TABLE III.1
PERFORMANCES AVEC ET SANS LE CACHE

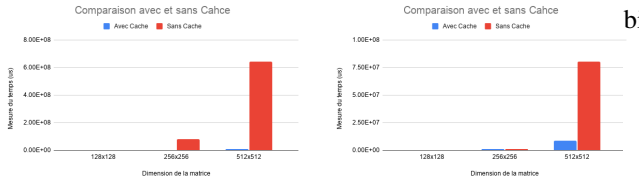


Fig. III.2. Performances avec et sans cache

Selon la table III.1 et la figure III.2, on peut trouver facilement que le temps d'exécution devient beaucoup plus grand ($>$ 10 fois) après l'invalidation de l'utilisation du cache, qui a vérifié le fonctionnement du cache dans la micro-architecture de l'ARM.

2) L'influence de prédiction de branchement: A part le cache, la prédiction de branchement est une autre partie dans la micro-architecture de l'ARM qui influence la performance.

La prédiction de branche peut nous aider à éviter le ralentissement de la vitesse d'exécution causé par le retard des pipelines. Quelque fois certain pipeline du processeur est en retard, cela ralentira la vitesse d'exécution de l'ensemble du programme. Dans ce cas la prédiction de branche est nécessaire de prédire et d'exécuter l'étape suivante.

Le mécanisme de travail consiste à prédire et à exécuter la prochaine instruction possible selon un ensemble de règles de prédiction de branchement sur un processus de branchement quand une instruction est en train d'être exécutée. Une fois l'instruction de processus principale est exécutée, il est jugé si l'instruction suivante est la même que le résultat prédit. S'il en est de même, le résultat du calcul du processus de branchement est directement copié dans le processus principal. Si la prédiction est incorrecte, le processus principal continue de s'exécuter et le processus de branchement effectue la prédiction suivante.

```
#include "xil_mmu.h"

Xil_DisableMMU();
```

Sur l'ARM, l'utilisation de la prédiction de branchement est validée en défaut, à l'aide du code ci-dessus, on a réussi à invalider les TLB, la prédiction de branchement et l'utilisation de cache, dont le résultat est présenté dans la figure III.3 et par la table III.2.

Dimension de matrix	16x16	32x32	64x64	128x128	256x256
temps (μs) sans cache	5.56×10^3	4.35×10^4	3.45×10^5	2.78×10^6	2.22×10^7
temps (μs) avec cache	1.89×10^2	1.44×10^3	1.13×10^4	9.09×10^4	1.07×10^6

TABLE III.2
PERFORMANCES AVEC ET SANS MMU

Le fonctionnement du MMU et de la prédiction est bien vérifié par la figure III.3 et par la table III.2.

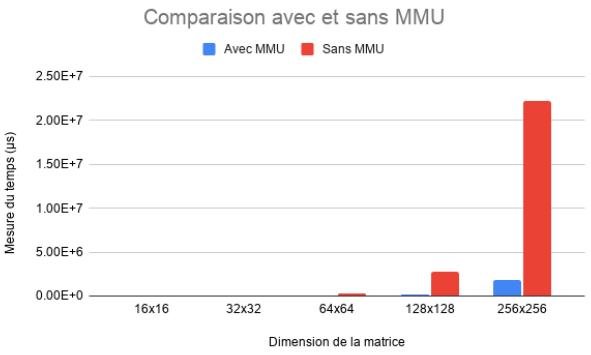


Fig. III.3. Performances avec et sans MMU

B. Optimisation sur l'option de compilation

Après avoir essayé des optimisations sur la micro-architecture, on se concentre maintenant sur l'option

en compilation. Dans *Xilinx SDK 2019*, il y a 5 choix d'option : O0, O1, O2, O3 et Os.

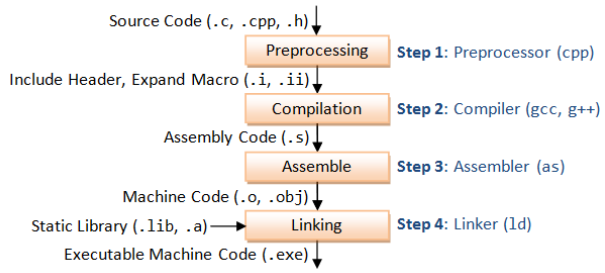


Fig. III.4. Le processus de la compilation

Pour langage C, le processus du programme du code source au programme binaire est constitué de 4 parties (*pre-processing*, *compilation*, *essemble* et *linking*).

Le *pre-processing* est utilisé pour remplacer tous les fichiers d'en-tête `#include` et les définitions de macro par leur contenu réel. Après le *pre-processing*, la taille du fichier sera beaucoup plus grande. La *compilation* ici est le processus de conversion de programmes pré-traités en code d'assemblage spécifique. Le processus *essemble* convertit le code d'assemblage de l'étape précédente en code machine. Le fichier généré à cette étape est au format binaire. Le processus *linking* lie plusieurs fichiers cibles et les fichiers de bibliothèque requis dans le fichier exécutable final.

En fonction de méthodes d'optimisation différentes:
1) Rationaliser les instructions d'opération;
2) Satisfaire l'opération de pipeline du processeur;
3) Ajuster l'ordre d'exécution du code en devinant le comportement du programme;
4) Utiliser adéquatement des registres, etc. *GCC* nous a fourni beaucoup d'options de compilation, qui sont O0, O1, O2, O3 et Os sur ARM.

Option de compilation	Effet
O0	Aucune optimisation
O1	Un peu d'optimisation
O2	Plus d'optimisation
O3	La plus d'optimisation
Os	Optimisation sur la taille du code

TABLE III.3
OPTION DE COMPILATION EN ARM

Selon le résultat présenté dans la table III.4 et dans la figure III.5, on en déduit que le temps d'exécution augmente avec l'augmentation de la dimension de la matrice. Plus grande la matrice, plus de temps il faut pour exécuter le code. Ensuite, c'est dans l'option O3 que la performance est la plus forte. L'option O2 a un fonctionnement similaire comme l'option Os.

Dans les sub-sections suivantes, O3 est utilisé comme l'option en compilation.

Option /Dimension	16x16	32x32	64x64	128x128	256x256	512x512
O0 (μs)	1.89×10^2	1.44×10^3	1.13×10^4	9.09×10^5	1.07×10^6	8.76×10^6
O1 (μs)	2.68×10^1	2.27×10^2	1.70×10^3	2.09×10^4	6.10×10^5	5.06×10^6
O2 (μs)	2.63×10^1	2.24×10^2	1.68×10^3	1.56×10^4	3.77×10^5	3.16×10^6
O3 (μs)	1.41×10^1	2.24×10^2	1.68×10^3	1.23×10^4	2.27×10^5	2.12×10^6
Os (μs)	2.63×10^1	2.24×10^2	1.68×10^3	1.56×10^4	3.77×10^5	3.16×10^6

TABLE III.4
PERFORMANCES DE DIFFÉRENTS OPTIONS DE COMPILATION

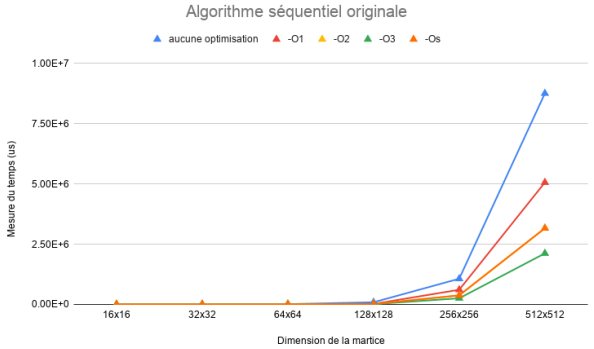


Fig. III.5. Performances de différents options de compilation

Selon la table III.5, une bonne option de compilation peut nous aider de réduire le temps d'exécution de 20%.

Dimension de matrix	16x16	32x32	64x64	128x128	256x256	512x512
O0 (μs)	1.89×10^2	1.44×10^3	1.13×10^4	9.09×10^5	1.07×10^6	8.76×10^6
O3 (μs)	1.41×10^1	2.24×10^2	1.68×10^3	1.23×10^4	2.27×10^5	2.12×10^6
Ratio (O3/O0)	7.45%	15.59%	14.94%	13.52%	24.02%	24.21%

TABLE III.5
RAPPORT D'AMÉLIORATION D'OPTION DE COMPILATION

C. Optimisation sur l'algorithme

Dans cette sub-section on fait des optimisations sur l'algorithme. On a écrit et a testé 3 algorithmes : Algorithme séquentiel original, Algorithme séquentiel amélioré et Algorithme de multiplication par blocs.

1) Réduction de l'écriture et de la lecture par variable intermédiaire: On fait tout d'abord une modification sur l'algorithme original comme dans la section 2. En ajoutant une variable intermédiaire, on a réussi à réduire le nombre d'écritures et de lectures pendant la multiplication de matrices, ce qui a réduit le temps d'exécution.

```
for (int j = 0; j < IN_B_COLS; j++){
// Iterate over the columns of the B matrix
    int sum = 0;
    // Do the inner product of a row of A and
        col of B
    for(int k = 0; k < IN_B_ROWS; k++){
        sum += a[i][k] * b[k][j];
```

```
    }
    prod[i][j] =sum;
}
```

Selon le tableau III-C1 et la figure III.6, on peut trouver facilement que le temps d'exécution est beaucoup moindre qu'avant, qui a vérifié notre choix.

Dimension de matrix	16x16	32x32	64x64	128x128	256x256	512x512
temps (μs) avant	3.27×10^2	2.52×10^3	1.99×10^4	1.62×10^5	1.87×10^6	1.58×10^7
temps (μs) arpès	1.89×10^2	1.44×10^3	1.13×10^4	9.09×10^4	1.07×10^6	8.76×10^6

TABLE III.6
PERFORMANCES AVANT ET APRÈS L'ADDITION D'UNE VARIABLE INTERMÉDIAIRE

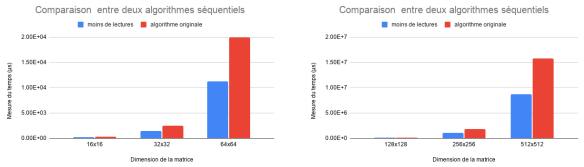


Fig. III.6. Performances avant et après l'addition d'une variable intermédiaire

2) Algorithme séquentiel original: Le code de l'algorithme séquentiel original est présenté ci-dessous. C'est l'algorithme de niveau le plus bas. La multiplication de la matrice est implémentée par 3 boucles. Le résultat de cet algorithme est présenté dans la table III.4 et dans la figure III.5.

```
void matrix_mult_sequentiel(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])
{
    for(int i = 0; i < IN_A_ROWS; i++){
        for(int j = 0; j < IN_B_COLS; j++){
            int sum = 0;
            for(int k = 0; k < IN_B_ROWS; k
                ++){
              sum += a[i][k] * b[k][j];
            }
            prod[i][j] =sum;
        }
    }
}
```

3) Algorithme séquentiel amélioré: Dans cet algorithme, on a inversé l'ordre des itérations de la ligne et de la colonne pour la deuxième matrice. Comme ça, pendant la lecture d'une valeur $a[i][k]$, toute la ligne j de la matrice *prod* est mise à jour. Le nombre de la lecture est réduit, ce qui rend un temps d'exécution beaucoup plus court.

```
void matrix_mult_sequentiel_2(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 s;
    for(int i = 0; i < IN_A_ROWS; i++){
            for(int k = 0; k < IN_B_ROWS; k++){
                s = a[i][k];
                for(int j = 0; j < IN_B_COLS; j
                    ++){
                  prod[i][j] += s * b[k][j];
            }
        }
    }
}
```

Option /Dimension	16x16	32x32	64x64	128x128	256x256	512x512	1024x1024
O0 (μs)	3.06×10^2	2.37×10^3	1.88×10^4	1.50×10^5	1.23×10^6	9.8×10^6	7.85×10^7
O1 (μs)	3.03×10^1	2.49×10^2	1.88×10^3	1.50×10^4	1.52×10^5	1.22×10^6	1.00×10^7
O2 (μs)	2.70×10^1	2.25×10^2	1.69×10^3	1.34×10^4	1.39×10^5	1.11×10^6	9.24×10^6
O3 (μs)	3.28×10^1	2.27×10^2	1.69×10^3	1.23×10^4	1.03×10^5	7.92×10^6	6.73×10^7
Os (μs)	3.88×10^1	3.05×10^2	1.94×10^3	1.58×10^4	1.56×10^5	1.24×10^6	1.02×10^7

TABLE III.7
PERFORMANCES DE L'ALGORITHME SÉQUENTIEL AMÉLIORÉ

Selon le résultat présenté dans la figure III.7 et dans la table III.7, le temps d'exécution de l'algorithme est le plus petit en option O3. Dans la section suivante, une comparaison complète va être faite entre les algorithmes que l'on a créé.

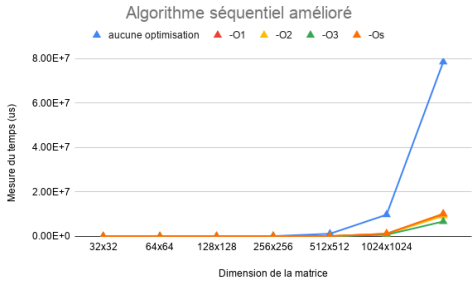


Fig. III.7. Performances de l'Algorithme séquentiel amélioré

4) Algorithme de multiplication par blocs: Plus grande la taille de la matrice, plus de temps il faut pour faire la multiplication. Dans cet algorithme, on a séparé la matrice en blocs plus petits, en considérant que la somme du temps de la multiplication des matrices plus petites est plus petite que le temps pour faire la multiplication d'une grande matrice.

```
void matrix_mult_block(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])
{
```

```
    for (int i_block=0; i_block<IN_A_ROWS;
        i_block+=BLOCKSIZE){
        for (int j_block = 0; j_block<IN_B_COLS
            ; j_block+=BLOCKSIZE){
            for (int i = 0; i < IN_A_ROWS; i++)
                {
                for (int j = j_block; j <
                    j_block + BLOCKSIZE; j++) {
                  int sum = 0;
                  for (int k = i_block; k <
                      i_block + BLOCKSIZE; k++)
                      {
                      sum += a[i][k]*b[k][j];
                  }
                  prod[i][j] += sum;
                }

            }
        }
    }
}
```

Option /Dimension	16x16	32x32	64x64	128x128	256x256	512x512
O0 (μs)	2.47×10^2	1.96×10^3	1.57×10^4	1.28×10^5	1.11×10^6	8.79×10^6
O1 (μs)	7.11×10^1	5.64×10^2	4.51×10^3	3.72×10^4	3.25×10^5	2.74×10^6
O2 (μs)	4.44×10^1	3.52×10^2	2.81×10^3	2.39×10^4	1.45×10^5	2.05×10^6
O3 (μs)	2.20×10^1	1.75×10^2	1.39×10^3	1.23×10^4	1.35×10^5	1.23×10^6
Os (μs)	7.93×10^1	6.32×10^2	7.01×10^3	6.03×10^4	4.93×10^5	2.96×10^6

TABLE III.8
PERFORMANCES DE L'ALGORITHME DE MULTIPLICATION PAR BLOCS

Selon le résultat présenté dans la figure III.8 et dans la table III.8, le temps d'exécution de l'algorithme est le plus petit en option O3. Dans la section suivante, une comparaison complète va être faite entre les algorithmes que l'on a créés.

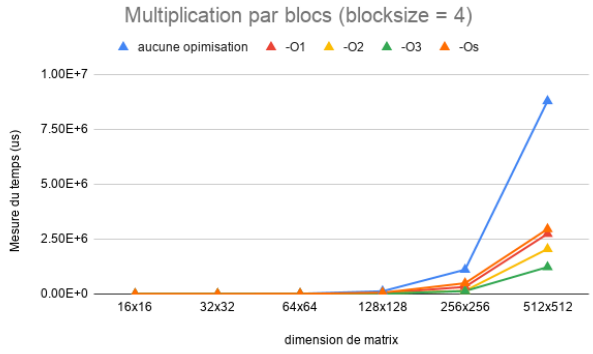


Fig. III.8. Performances de l'Algorithme de multiplication par blocs

5) Comparaison des algorithmes: La comparaison des performances de trois algorithmes est présentée dans la figure III.9. On peut en déduire facilement que quand la taille de la matrice est petite ($< 128 \times 128$), c'est l'algorithme original qui se comporte le mieux, mais quand la taille de la matrice est plus grande ($> 128 \times 128$), c'est l'algorithme séquentiel amélioré et l'algorithme de multiplication par blocs qui sont meilleurs.

En plus, s'il n'y a aucune optimisation (O0), c'est toujours l'algorithme original qui est le plus rapide, mais quand d'autres options sont choisies, le résultat change complètement.

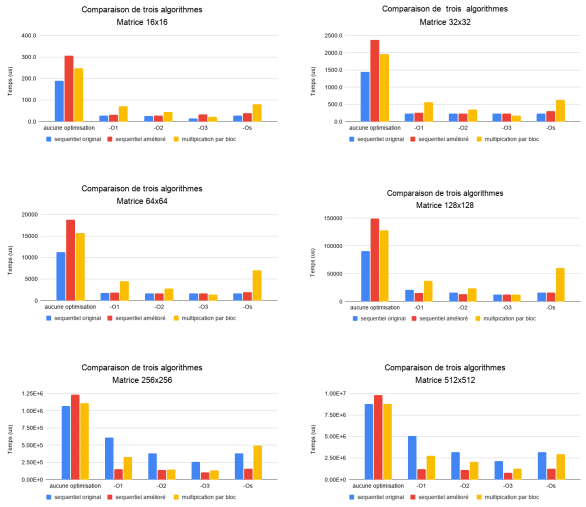


Fig. III.9. Comparaison des trois algorithmes

En conclusion, l'algorithme séquentiel amélioré est le meilleur parmi ces trois algorithmes même si son comportement en O0 n'est pas assez bien.

D. Optimisation à venir

Dans l'environnement de compilation croisée de Linux, la technologie de *multi-threads* est possible mais elle n'est pas autorisée dans *Xilinx sdk*. Pourtant c'est quand même possible de faire la communication entre les deux coeurs l'ARM9.

Ce problème s'agit des connaissances sur *Noc (Network on chips)* de ROB307 et dans ce TP on n'a pas fait beaucoup sur cette partie, mais on pense que la coopération entre deux coeurs peut nous aider à améliorer la performance.

E. Comparaison entre le PC et l'ARM

Dans cette sub-section, on fait une comparaison entre la performance de PC et celle de l 'ARM. Selon la table III.9, la performance de PC est plus forte que celle d'ARM. En plus, plus grande la taille de la matrice, plus grand il y a un écart entre les performances. En moyenne, la performance du PC est 20 fois plus forte que celle de l'ARM, ce qui peut être expliqué par la

fréquence beaucoup plus basse de l'ARM et le nombre plus petit des coeurs de l'ARM.

Comparaison PC/ARM de l'algorithme originale (O0)						
Dimension de matrix	16x16	32x32	64x64	128x128	256x256	512x512
ARM (μs)	1.89×10^2	1.44×10^3	1.13×10^4	9.09×10^5	1.07×10^6	8.76×10^6
PC (μs)	3.52×10^1	1.63×10^2	1.08×10^3	7.45×10^3	4.22×10^4	4.19×10^5
Ratio ARM/PC)	5.4	8.8	10.4	12.2	25.3	20.9

TABLE III.9
COMPARAISON PC/ARM DE L'ALGORITHME ORIGINALE (O0)

Dans la table III.10, on a comparé le temps d'exécution de l'algorithme séquentiel améliorée en option O3 (le plus rapide) sur PC et en ARM. Même si la performance de l'ARM a été amélioré par l'optimisation d'algorithme et d'option de compilation, il y a encore un grand écart entre le PC et l'ARM. Dans ce cas, PC est 40 fois plus fort que ARM.

Comparaison PC/ARM de l'algorithme séquentiel amélioré (O3)						
Dimension de matrix	16x16	32x32	64x64	128x128	256x256	512x512
ARM (μs)	3.28×10^1	2.27×10^2	1.69×10^3	1.23×10^4	1.03×10^5	7.92×10^6
PC (μs)	4.10×10^0	1.10×10^1	4.38×10^1	2.20×10^2	2.26×10^3	1.20×10^4
Ratio ARM/PC)	8.0	20.6	38.6	55.7	45.3	66.0

TABLE III.10
COMPARAISON PC/ARM DE L'ALGORITHME SÉQUENTIEL AMÉLIORÉ (O3)

IV. ESTIMATION DES PERFORMANCES ET DES RESSOURCES PAR ACCÉLÉRATEUR MATÉRIEL HLS SUR CIRCUIT FPGA

A partir de cette partie, on va chercher à optimiser l'algorithme de multiplication de matrices avec un circuit FPGA. Dans cette partie, le but est de faire l'optimisation dans l'aspect IP avec le logiciel HLS.

A. Démarches en Vivado HLS

Vivado HLS est le logiciel utilisé pour dessiner un IP FGPA, on peut y rajouter des méthodes d'optimisation et estimer les effets des optimisations. Ici on introduit les étapes principales pour exporter un ip, afin qu'il puisse être utilisé dans le block design en Vivado.

1) Codes C++: Dans la création d'un projet, il faut choisir le board FPGA utilisé. Dans notre cas, c'est le *Zedboard Zynq Evaluation and développement Kit*. Ensuite on peut ajouter des *Source file* dans le projet, qui sont des codes C++ pour réaliser la fonction. Dans le *Source file*, on n'a pas besoin d'une fonction *main*. Par contre, il doit y avoir une *Top function*, qui est une fonction de type *void* et décrit la principale fonction implémentée par cette ip. Dans notre projet, cette fonction est de multiplier les deux matrices et de donner une matrice comme résultat. Et d'après nos expériences, il est plus profitable d'optimiser l'algorithme *Sequence1 (séquentiel original)* parmi les trois algorithmes proposés, où il y a trois échelles d'itérations. On les note séparément *Row*, *Col* et *Product*. Alors notre *Top function* est comme la figure suivante :

```
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])
{

  // Iterate over the rows of the A matrix
   Row: for(int i = 0; i < IN_A_ROWS; i++) {
      // Iterate over the columns of the B
          matrix
      Col: for(int j = 0; j < IN_B_COLS; j++) {
         float sum = 0;
         // Do the inner product of a row of A
             and col of B
         Product: for(int k = 0; k < IN_B_ROWS;
              k++) {
            sum += a[i][k] * b[k][j];
         }
         prod[i][j] = sum;
      }
   }
}
```

A part les *Source File*, il est préférable d'ajouter encore les fichiers *Test Bench*, dans le but de vérifier si le programme de *Source* marche de la manière qu'on veut. Dans le *Test Bench*, on a besoin d'une fonction *main*. Pour vérifier si le résultat de la multiplication est correct, on initialise deux matrices et on impose les valeurs de ces deux matrices. Ensuite on calcule la multiplication de ces matrices dans *Test Bench* avec l'algorithme le plus simple et on obtient une matrice de résultat *idéale*. Puis on appelle la *top function* dans le *Test Bench* avec les deux matrices initiales comme paramètres. Alors on peut obtenir une matrice de résultat *test*. Dans le reste du programme, on compare les valeurs de la matrice *idéale* et celles de la matrice *test*. Si elles sont exactement les mêmes, on peut dire que le test a réussi. Sinon on peut imprimer les résultats dans les différents endroits des codes pour chercher les fautes.

Pour appeler la *Top Function* dans le *Test Bench*, il faut que les deux fichiers incluent le même fichier *.h* et que dans le fichier *.h* on définisse un nom spécial, qui est *HW_COSIM* dans notre cas. Quand on a besoin de l'appeler dans *Test Bench*, on utilise les expressions *ifdef* et *endif* comme ci-dessous:

```
#define HW_COSIM
...
#ifdef HW_COSIM
matrix_mult(A, B, C);
#endif
```

Dans certains cas, par exemple, si on veut régler les ports *input* et *output* avec protocole AXI4-Stream, on doit beaucoup modifier les codes dans *Source File* et

Test Bench, ce qui est présenté dans la partie *Protocoles et Ports*.

2) Rapports de Performance: Une fois les codes C++ sont bien ajoutés, on peut commencer à optimiser le programme en temps avec les différentes directives dans HLS au prix de l'espace. Pour estimer les effets d'un programme avec des directives, on fait un *C Synthesis* dans HLS. Comme cela, HLS va nous donner un rapport qui estime le temps d'exécution et les ressources utilisées. Avant tout, il est nécessaire de connaître les notions essentielles décrivant les performances d'un programme dans le rapport.

Ci-dessous on peut voir un rapport typique après une opération *C Synthesis* en Vivado, qui nous donne des informations nécessaires sur le temps d'exécution et les ressources utilisées. Dans la première partie *Summary*, on voit les estimations sur le nombre de nanosecondes (ns) par cycle de clock dans le circuit. Par défaut, le but de ce terme est 10ns. À condition de ne pas dépasser la limite de ressources, Vivado tentera de rendre la valeur de ce terme inférieure au but pour que le temps d'exécution soit plus petit. Dans l'intégration du circuit, on peut ajuster la fréquence de cette ip pour correspondre à l'estimation (le nombre de cycles de clock par seconde). Les détailles seront présentées dans la partie d'intégration.

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.730	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
131849	131849	131849	131849	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	64	-	-	-
Expression	-	0	0	23820	-
FIFO	-	-	-	-	-
Instance	0	-	36	40	-
Memory	66	-	0	0	0
Multiplexer	-	-	-	410	-
Register	0	-	2567	96	-
Total	66	64	2603	24366	0
Available	280	220	106400	53200	0
Utilization (%)	23	29	2	45	0

Fig. IV.1. Un rapport typique après C Synthesis

Le *latency* et le cycle de clock déterminent ensemble le temps d'exécution. Dans Vivado HLS, la définition de *latency* est le nombre de cycles de clock estimé entre un *input* et un *output*. Un autre terme similaire est *interval*, qui définit le nombre de cycles de clock estimé entre un *input* et un autre *input*. Souvent, l'*interval* est égal ou un peu supérieur au *latency* (1 cycle) pour un programme donné. Alors, ici on considère le *latency* comme le critère du temps d'exécution. Avec *latency* et le nombre de nanosecondes par cycle, noté *clock_size*, on peut calculer le temps d'exécution du programme par:

$$Temps = latency \times clock\_size$$

Un autre critère pour évaluer la performance sur le temps qui n'apparaît pas dans le rapport est le débit (Throughput), qui décrit la vitesse de *input* et *output*. Le débit n'influence pas de manière directive le temps d'exécution. Cependant, si le débit de l'ip est trop petit, il n'y a pas assez de données à traiter dans l'ip donc ce n'est pas utile d'avoir une trop grande puissance. A la création d'une ip, on doit y ajouter des *ports* pour transformer les données. Alors le débit d'une ip dépend de la largeur de ces ports (*data_width*) et la fréquence de transporter les données:

$$Débit = data\_width \times fréquence\_transformation$$

Ensuite, on regarde l'estimation des ressources utilisées dans le rapport. Dans notre programme, on a besoin de 4 types de ressources: BRAM, DSP, LUT et FF.

-**LUT**(Look-up table): Une ressource flexible capable de mettre en oeuvres: (i) une fonction logique jusqu'à six entrées; (ii) une petite mémoire morte (ROM); (iii) une petite mémoire vive (RAM);

-**FF**(Flip-flop): un élément de circuit séquentiel mettant en œuvre un registre à 1 bit, avec une fonctionnalité de réinitialisation.

-**DSP**(Digital Signal Processor): un élément logique de traitement de signal numérique, qui peut effectuer différents types d'opérations arithmétiques, y compris la multiplication et l'addition.

-**BRAM**(Block RAM): élément dans le circuit FPGA pour stocker temporairement des données.

Pour différents circuits FPGA, les ressources disponibles varient avec les différents modèles. Dans notre test (XC7Z020), on a 53200 LUTs, 106400 FFs, 220 DSP et 280 BRAM_18K (4,9MB), qui sont présentés dans le rapport. Et dans les lignes voisines, le rapport liste la quantité de ressources utilisée avec les

optimisations imposées dans le programme, et le pourcentage d'utilisation pour chaque type de ressources. En général, le processus d'optimisation est d'échanger le temps en espace. Il y a une tendance telle que plus vite le programme se déroule, plus de ressources il occupe (front de Pareto). Dans l'estimation, on doit confirmer que les ressources utilisées ne peuvent pas dépasser 100%. Si cela se passe, soit on utilise un meilleur board, soit on réduit le niveau d'optimisation.

3) Optimisations avec directives: Vivado HLS nous fournit des directives fixées. Il y a deux façons d'insérer ces directives dans les codes. On peut écrire des lignes de codes qui commencent par `#pragma`. Ou bien on peut simplement cliquer le bouton *directive* au droit de l'écran et insère les directives pour les termes dans le code. Ces directives données par HLS peuvent faire des optimisations dans différents aspects (fonctions, itérations, etc). Quelques directives qu'on a testées sont les suivantes:

Directive	Effet
Pipeline	réduit l'intervalle d'initiation
Array_partition	Partitionne un array en arrays plus petits
Array_map	Combine plusieurs petits arrays en un seul grand array
Array_reshape	Combinaison de Array_partition et Array_map
Inline	Supprime une fonction dans la hiérarchie
Dataflow	active pipeline au niveau des fonctions
Allocation	Spécifie les restrictions d'instance
Unroll	crée plusieurs opérations indépendantes plutôt qu'une seule itération

TABLE IV.1
DIRECTIVES UTILISÉES EN HLS

Parmi toutes les directives qu'on a testées, on trouve que les plus utiles d'entre eux pour réduire le temps d'exécution sont *Pipeline* et *Array_partition*.

Dans le processus de la multiplication, on a besoin de calculer plusieurs itérations. Si on n'utilise pas *pipeline*, ces itérations seront traitées une par une : après avoir sorti le résultat de cette itération, le programme va commencer à traiter la prochaine. Par contre, la directive *Pipeline* permet le programme de traiter à la fois plusieurs itérations en prenant plus de ressources, comme décrit dans la figure IV-A3:

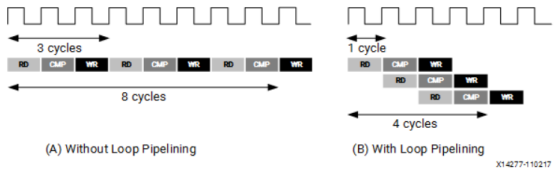


Fig. IV.2. Le principle de *pipeline*

Array_Partition divise un grand array en plusieurs petits arrays en prenant plus de mémoire. En faisant cela, les petits arrays peuvent être lus en même temps donc cela accélère le processus de *input* et *output*. Afin d'équilibrer le temps d'exécution et l'utilisation de ressource, on utilise très souvent *Array_Reshape* en réel, qui est une combinaison de *Array_Partition* et de *Array_Map*, comme présenté dans la figure suivante. Par conséquent, le débit augmente. Comme on a dit dans la dernière sous-partie, la seule augmentation du débit ne peut pas accélérer le programme. Donc il est indispensable d'utiliser *Array_Reshape* et d'autres directives en même temps. Dans notre expérience, l'utilisation de *Array_Reshape* et *Pipeline* peut donner des excellents effets.

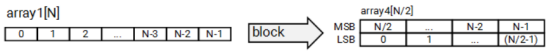


Fig. IV.3. Le principle de *Array_Reshape*

4) protocoles et ports: Quand on construit une ip, on ne considère pas que les fonctions réalisées à l'intérieur de l'ip. Il faut aussi penser comment l'ip se connecte avec les autres dans le circuit. Alors il faut ajouter des *ports* en utilisant les directives pour transporter les informations et donner des signaux de contrôle. Dans notre expérience, on a testé deux types de protocoles pour transporter les données: AXI4-Lite et AXI4-Stream. Ils sont tous les deux des protocoles de bus souvent utilisés dans FPGA. Leurs propriétés sont présentées dans le tableau ci-dessous:

Propriété	AXI4-Stream	AXI4-Lite
sens de transmission	simple	double
occupation de mémoire	grande	petite
adresse requise	non	oui
débit	grand	petit

TABLE IV.2
PROPRIÉTÉS DE AXI4-STREAM ET DE AXI4-LITE

D'après ces propriétés, on peut en conclure que AXI4-Stream est un protocole plus puissant que AXI4-Lite mais il occupe plus de ressources. Pour ajouter ces ports dans l'ip, la manière la plus simple est d'insérer les directives avec les boutons dans HLS. Cependant, de cette façon, on ne peut pas modifier les ports selon notre besoin. Dans la partie d'intégration, on veut que le protocole utilisé soit AXI4-Stream et que les deux matrices entrées partagent un port. Avec les boutons de directive, on ne peut que les diviser en deux ports. Dans ce cas, on modifie les codes C++ et on y ajoute directement des ports.

Pour réaliser ça, tout d'abord, on doit changer la *Top Function*. Au lieu d'entrer directement les matrices, on prend deux *streams* qui contiennent les valeurs des matrices comme entré. On lit les valeurs une par une et puis les impose aux matrices. Les contenus des *streams* sont déjà définis dans le fichier *ap_axi_sdata.h*. Une chose importante est que pour n'importe quel type de

variable, si on veut les transformer par un *stream*, il faut les transformer en type *unsigned char*. Pour assurer la compatibilité du programme, on prend les valeurs des matrices comme *char* à l'intérieur de l'ip et on fait des transformations avant et après les calculs par le class *converter* qui est aussi défini par HLS.

Une fois les valeurs sont bien lues, le programme calcule la multiplication comme avant. Pour sortir le résultat, on insère les valeurs une par une dans un *output stream* et l'envoie par un port *output*. Une remarque est qu'à la fin de *output stream*, il faut noter dans le code que la valeur à présent est la dernière donnée dans le *stream* pour que le programme dans SDK finisse de lire le stream.

Ensuite on modifie les codes dans le *Test Bench* de la manière similaire. Les codes complets sont posés dans le drive. Après avoir tout fini les codes, on re-fait la *C Synthesis* pour vérifier le résultat de calculs et on fait la *C/RTL Cosimulation* pour vérifier si le résultat est toujours bon dans l'aspect hardware. Si toutes ces vérification sont réussies, on peut exporter l'ip en VHDL. Ces ips sont comme les figures suivantes dans Vivado Block Design:

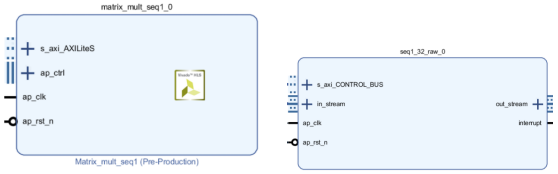


Fig. IV.4. IPs avec AXI4-Lite (gauche) et AXI4-Stream (droit)

B. Résultats

Dans nos expériences, on fait varier les tailles de matrices, les directives d'optimisation, les protocoles de transmission utilisés et on estime les effets d'optimisation dans les différents groupes.

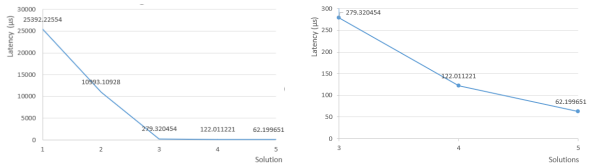
On prend les tests pour des matrices 64×64 comme exemple. Dans ce groupe, on impose le *data_width* égal 32bit. On utilise les ports *s_axilite* (AXI4-Lite) comme ports de transmission et le but de cycle de clock est 10ns. Les 5 solutions (méthodes d'optimisation) différentes sont choisies comme suivant:

Numéro	directives
Solution 1	Aucune Optimisation
Solution 2	Pipeline Product; Array_Partition a et b complet
Solution 3	Pipeline Col; Array_Partition a et b en 8
Solution 4	Pipeline Col; Array_Partition a et b en 16
Solution 5	Pipeline Col; Array_Partition a et b en 32

TABLE IV.3
SOLUTIONS AXI4-LITE POUR MATRICES 64×64

Quelques remarques: Quand on utilise *Array_Partition*, on peut choisir le paramètre qui décide le niveau de division d'un array. Plus chaque array est petit, plus la lecture des données est rapide mais plus le programme va occuper de la mémoire. Si on choisit *complet*, ça veut dire que chaque petit array contient un seul élément. Si on choisit le facteur comme 8, ça veut dire qu'on divise un grand array en 8 petits arrays, où chaque portion contient 8 éléments dans ce cas.

La raison pour laquelle on ne prend pas les solutions avec *pipeline* en *Row* est la limite de ressources. On a testé ces cas mais l'utilisation de LUTs va dépasser 100%. Les différents critères de ces 5 solutions sont présentés ci-dessous:



(a) 1-5 complet (b) 3-5 agrandi

Fig. IV.5. Variations de Latency pour les matrices 64

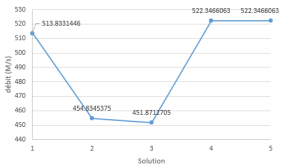


Fig. IV.6. Variations de débit pour les matrices 64

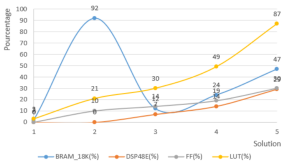


Fig. IV.7. Variations de ressources utilisées pour les matrices 64

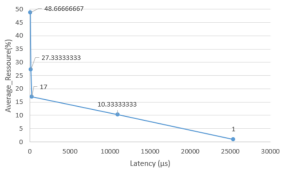
Pour les *latency*, les résultats correspondent bien à ce qu'on a prévu : quand on fait le *pipeline* dans une plus grande échelle, tant que la vitesse de lecture est suffisante, le temps d'exécution va diminuer. Pour les

solutions 1-3, l'élément qui limite le *latency* est la vitesse de calcule et pour 3-5 la limite est la vitesse de lecture des données.

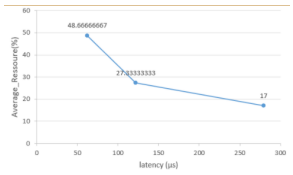
Pour le débit des données, comme on a fixé le *data_width* à 32 bit, le seul élément qui influence cette variable est le nombre de nanosecondes par cycle de clock. Alors on ne peut pas voir une loi pour décrire ces variances.

Pour les ressources utilisées, la variance est opposée à celle de *latency* : Quand le temps d'exécution diminue, le programme va occuper plus de ressources du système. On peut voir que pour la solution 2, l'utilisation de *BRAM* est trop haute. Cela nous dit que ce n'est pas une bonne idée de faire *pipeline* dans l'échelle *Product*.

Alors avec les résultats obtenus, on peut tracer un front de Pareto : une courbe pour décrire la relation entre le temps d'exécution et les ressources utilisées. Afin de bien afficher la courbe, on prend la moyenne de pourcentage des 3 types de ressources mentionnées dans cete partie sauf *BRAM*, noté *Average_Ressource*, pour représenter l'utilisation de ressources. Le front de Pareto est présenté dans la figure **??** :



(a) 1-5 complet



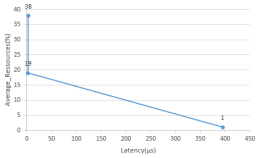
(b) 3-5 agrandi

Fig. IV.8. Front de Pareto pour les matrices 64

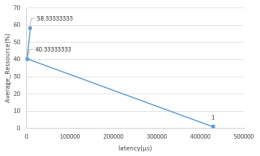
D'après le front de Pareto, on voit bien la loi entre le temps d'exécution et les ressources utilisées. Dans la suite de cette partie, pour gagner de l'espace et du temps, on va ne poser que quelques fronts de Pareto pour les différentes tailles de matrices et avec différentes configurations de l'ip. Les fichiers détaillés sont posés dans le drive de notre groupe.

On a fait aussi des ips avec le protocole AXI4-Stream et les fronts de Pareto sont présentés ci-dessous:

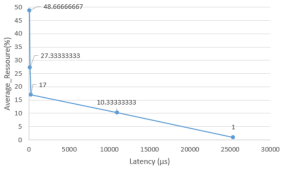
Alors on peut voir que les formes de ces courbes sont toutes similaires, ce qui correspond à l'équilibre entre le temps d'exécution et les ressources utilisées.



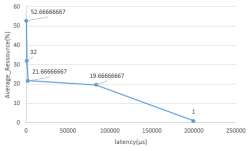
(a) matrice 16



(b) matrice 32

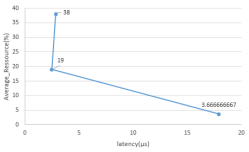


(c) matrice 64

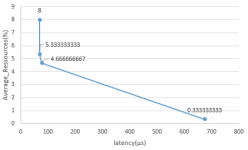


(d) matrice 128

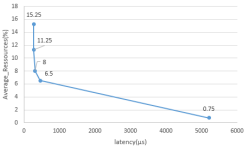
Fig. IV.9. Fronts de Pareto pour différentes tailles en AXI4-Lite



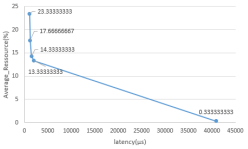
(a) matrice 16



(b) matrice 32



(c) matrice 64



(d) matrice 128

Fig. IV.10. Fronts de Pareto pour différentes tailles en AXI4-Stream

V. Mesures des performances réelles de l'accélérateur matériel sur le circuit FPGA

Cette dernière partie du projet a pour objectif d'optimiser les performances des accélérateurs matériels créés dans la troisième partie en les connectant au processeur ARM9 du Zedboard via un réseau AXI. Ainsi, cette phase du projet est une implémentation sur chip de tout le travail qui a été réalisé par mes collègues : écriture algorithmique et Conception HLS.

A. *Démarche suivie*

Pour accomplir cette tâche, on a suivi les étapes suivantes :

- Créer un bloc design qui relie entre l'IP HLS pré-produite et le processeur Zynq à travers un réseau AXI4-Stream.
- Vérifier la synthèse du design en analysant les rapports fournis par Vivado : rapport d'énergie, d'utilisation et du timing.
- Générer les wrappers et le bitStream : le hardware avec lequel on va programmer le FPGA.
- Exporter le hardware et ouvrir SDK.
- Écrire le code SDK (en mode strandalone) qui permet de communiquer avec le DMA, l'IP HLS, l'axi-timer et le processeur ARM9 dans le but d'évaluer les performances de l'implémentation.

B. Le design utilisé

En se référant aux considérations précédentes, on trouve que les performances du bus AXI4-Lite sont limitées vu qu'il effectue un transfert séquentiel des données à 32 bits. Ceci engendre un temps de communication énorme et rend le transfert un peu lent. Pour cette raison, on a pris comme choix de travailler avec le réseau AXI4-stream. Dans ce cas, il suffit de fixer la taille maximale du paquet transmis (dans notre cas 32 bits) en fonction du nombre total des entrées/sorties communiquées. Ensuite, les entrées sont transférées sous forme d'un flux de données vers la mémoire DDR en utilisant un accès DMA (Direct Memory Access).

Enfin, la figure V.1 explique le diagramme qu'on a utilisé afin d'associer l'IP HLS pré-produite (avec une interface AXI-stream) et le processeur *Zynq*.

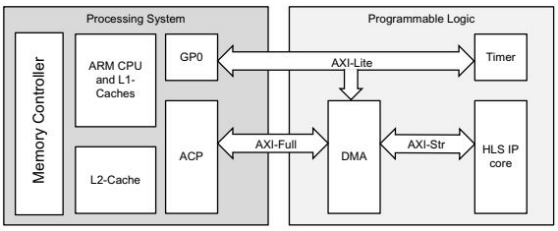


Fig. V.1. PS et PL dans le Zynq-7000 AP SOC

1) AXI4-DMA: L'IP AXI4-DMA permet un accès direct en *memory mapped* entre la mémoire DDR et les périphériques. Cet IP core contient plusieurs interfaces de transfert avancé. Les interfaces qu'on a utilisé pour notre design sont :

- Slave AXI4-Lite
- AXI4 à AXI4-stream Stream Master (MM2S)
- AXI4-stream à AXI4 Stream Slave (S2MM)
- AXI Control AXI4-Stream Master
- AXI Status AXI4-Stream Slave

Pour ce travail, nous utilisons le DMA en mode *Simple DMA*. Dans ce mode, on effectue des transferts simples à travers les interfaces matser-MM2S et slave-S2MM ce qui est caractérisé par une petite consommation des ressources matérielles. Comme expliqué dans la figure V.2, le principe de ce transfert DMA est le suivant:

L'interface master MM2S reçoit les données depuis le DDR et l'envoie vers l'interface Slave de l'IP core : S-IN qui est le port input-stream de l'IP HLS. Dans notre cas, bien entendu la multiplication de deux matrices, pour chaque test le master MM2S envoie à l'input-stream de l'IP deux streams successifs qui contiennent le data des matrices A et B en entrée. Puis l'IP HLS va traiter la demande et il prépare le res-hw résultat de la multiplication de A et B et l'envoie à travers l'interface slave output-stream vers l'interface du DMA slave S2MM qui l'envoie à son tour au DDR. Enfin, la communication entre le DMA et le DDR est effectuée, entre autres, à travers le bus ACP.

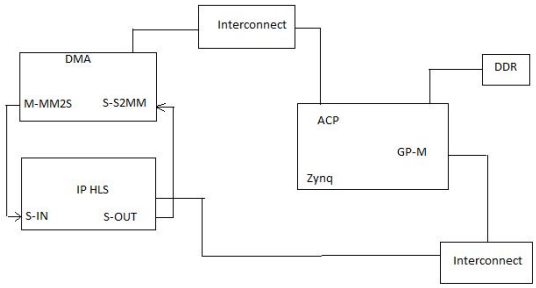


Fig. V.2. Explication des transferts DMA

2) ACP port: Les FPGA SoC basés sur le processeur ARM Cortex-A9 incluent une fonctionnalité appelée Accelerator Coherency Port : ACP. Grâce à l'ACP, les nouvelles données produites par notre accélérateur matériel sont transférées directement au cache L2 du processeur *Zynq*, via une connexion directe à faible latence. Cette opération est effectuée non seulement rapidement, mais aussi de manière cohérente. Ainsi, on a choisi d'utiliser cette interface pour effectuer les transferts entre le DMA et le DDR.

3) AXI timer: Afin de suivre les performances de l'accélérateur matériel sur chip, en temps réel, on ajoute au block design l'IP AXI timer. Ce dernier, configuré correctement sous SDK, permet de repérer le nom-

bre total des cycles effectués par le hardware lors de l'exécution sur la carte.

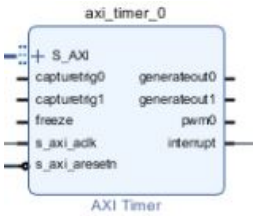


Fig. V.3. Le core AXI timer

4) l'IP HLS utilisée: La dernière partie de notre design est l'accélérateur matériel. On ajoute l'IP HLS créées sous Vivado HLS pour les différentes dimensions et avec différentes solutions. Cette phase du travail n'utilise que les IPs configurées avec des interfaces AXI4-stream.

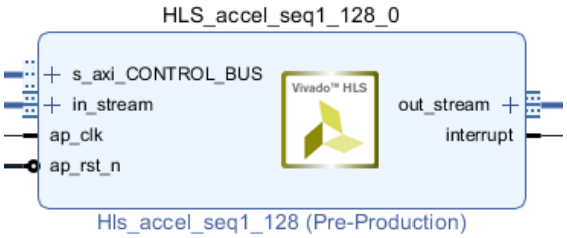


Fig. V.4. Le design utilisé

On a fait la synthèse de plusieurs designs pour les dimensions 16, 32, 64 et 128 et pour diverses solutions en axi4-stream. Par exemple, le tableau suivant illustre les solutions IP HLS utilisées avec axi-stream et pour la dimension 128 :

Numéro	Directives
Solution 1	Pipeline COLS ; Array_Partition a et b en 8
Solution 2	Pipeline COLS ; Array_Partition a et b en 16
Solution 3	Pipeline COLS ; Array_Partition a et b en 32
Solution 4	Pipeline COLS ; Array_Partition a et b en 64
Solution 5	Pipeline COLS ; Array_Partition a et b complet
Solution 6	Aucune optimisation

TABLE V.1
SOLUTIONS AXI4-STREAM POUR LES MATRICES 128×128

5) Conclusion: En conclusion, les principales IP Cores de notre design sont : axi DMA + ACP, *Zynq* processing system, l'accélérateur matériel HLS et l'AXI-timer. Les principaux paramètres de l'implémentation :

- La fréquence de l'IP HLS (celle de l'horloge FCLK) : 200 MHz (sauf quelques cas spécifiés dans la partie 4 du drive du groupe).
- Le data width est 32 bits.
- La fréquence du processeur *Zynq* est maintenue inchangée (666 MHz).

Enfin, la totalité de notre block design est donnée par la figure V.5 :

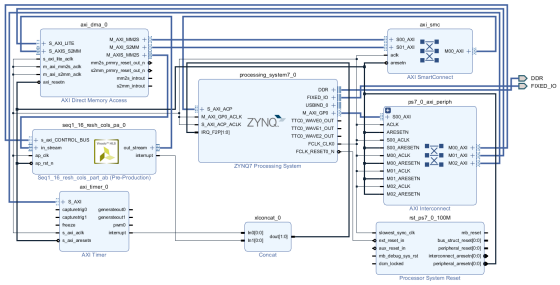


Fig. V.5. Le design utilisé

C. Optimisation sous Vivado

L'outil Vivado utilise plusieurs directives d'optimisation afin d'ajuster le timing du design et pour optimiser le routage et parsuite la consommation d'énergie. Après la synthèse de notre design, on peut voir les rapports générés par Vivado afin de vérifier si tout marche bien dans le circuit ou pas. Pour ce faire, on peut analyser les rapports d'énergie, d'utilisation et du timing.

1) Utilisation matérielle: On peut repérer sous Vivado les taux de la consommation des ressources matérielles ainsi que la répartition de ces ressources sur le circuit. Soit par exemple ce circuit synthétisé pour des matrices de dimension 32 et la solution HLS : Pipeline Cols et Partition a et b Complet (figure 5.6).

On remarque une grande consommation des DSP (73 %). De même, la consommation des Flip-Flops (18 %) et des LUTs (29 %) est assez élevée. Cette hyper consommation est aussi claire sur le circuit synthétisé (figure 5.7). Pour résoudre ce problème on a deux solutions : soit réajuster le bloc design soit changer les directives de l'IP HLS en essayant de maintenir des performances de calcul assez proches. La solution qu'on a adopté est la deuxième et effectivement le problème a été résolu : avec la nouvelle IP (pour cette dimension 32) on a désormais une moyenne de consommation matérielle égale à 9.75 % avec un facteur d'accélération par rapport à ARM9 égal à 3.9.

2) Timing: Outre la consommation matérielle, on analyse le rapport du timing pour voir s'il y a des *path* critiques avec un timing négatif et aussi pour voir si la

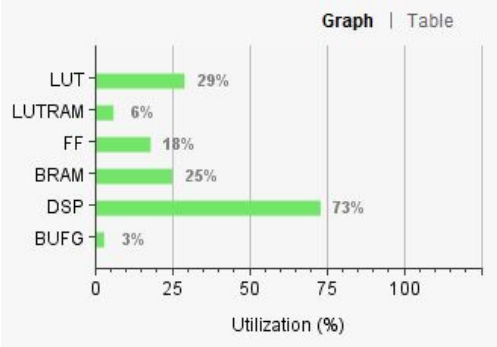


Fig. V.6. Utilisation matérielle pour la solution pipeline cols + Partition a et b complet, dim 32

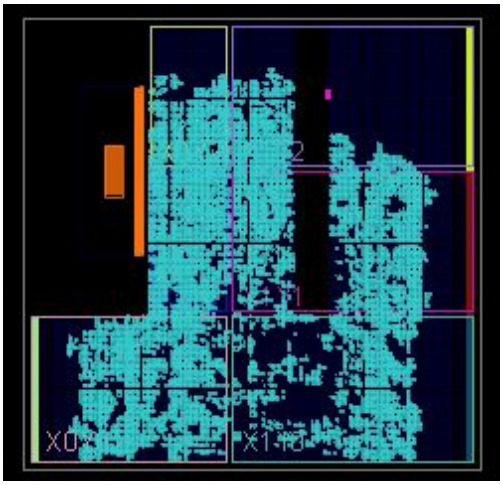


Fig. V.7. Design synthétisé pour la solution pipeline cols + Partition a et b complet, dim 32

fréquence du FCLK choisie est adéquate avec le design ou non. Pour la solution considérée dans cette partie (Pipeline cols + partition a et b complet), toutes les contraintes du temps sont respectées comme montré par la figure 5.8. En outre, si l'on accède au schéma des paths de ce "worst negative slack", on trouve qu'ils sont tous compacts (un exemple est donné par la figure 5.9), donc le design est bon côté timing. Si ce n'est pas le cas (et c'était pas pour les derniers tests où on a augmenté considérablement la fréquence), on devra vérifier toutes les sources des paths à timing négatifs pour fixer les contraintes non respectées.

3) Consommation d'énergie: Une dernière chose à voir est la consommation d'énergie. En fait, lors de la synthèse, Vivado utilise des directives d'optimisation du design pouvant réduire la consommation d'énergie jusqu'à 30 %. On peut effectuer un rapport d'énergie

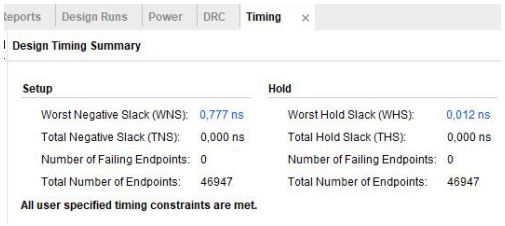


Fig. V.8. Rapport du timing pour la solution pipeline cols + Partition a et b complet, dim 32

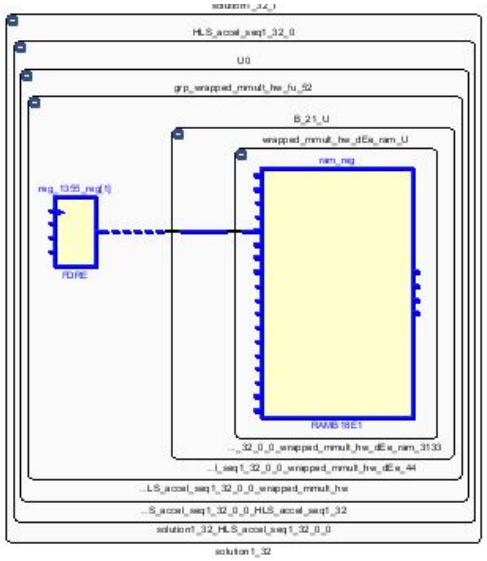


Fig. V.9. Schéma d'un path à timing postif

à la fin de la synthèse, ce qui donne la répartition de l'énergie dissipée sur les différentes parties du circuit de la FPGA.

D. Software design avec SDK

Après avoir créé le hardware avec lequel on programme la carte, on l'exporte et on va utiliser maintenant SDK. L'objectif est de tester le fonctionnement du hardware et de le comparer avec le software (traitements donnés par le ARM9). Ce qu'on fait sous SDK est :

Étape 1 :: On crée des instantiations de l'IP core, l'AXI DMA et l'AXI timer. et on les initialise.

```
1 // IP core instance
2 XSeq1_32_raw xmmult_dev;
3 XSeq1_32_raw_Config *mmult_dev_cfg ;
4
5 // axiDMA instance
6 XAxiDma axiDMA ;
7 XAxiDma_Config *axiDMA_cfg ;
8
```

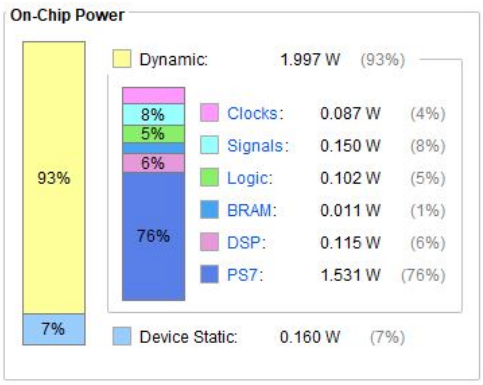


Fig. V.10. Rapport d'énergie pour la solution Pipeline Cols + Partition en a et b complet

```
// TIMER Instance
XTmrCtr timer_dev;
```

```
// Initialising HLS accelerator core
mmult_dev_cfg = XSeq1_32_raw_LookupConfig(
    XPAR_SEQ1_32_RAW_0_DEVICE_ID);

// Initialising AxiDMA core
axiDMA_cfg = XAxiDma_LookupConfig(
    XPAR_AXIDMA_0_DEVICE_ID);
```

Étape 2 :: A l'intérieur de la fonction main, on crée les matrices input A et B et on calcule le résultat donné par le software pour 1024 tests et on utilise l'AXI timer pour repérer le nombre moyen des cycles par exécution.

Étape 3 :: On calcule le résultat par le hardware aussi comme une moyenne sur 1024 tests et en utilisant le AXI timer. Pour chaque test, on envoie A puis B depuis le DMA vers l'IP core puis on reçoit res_hw = A*B depuis l'IP Core vers le DMA.

```
status = XAxiDma_SimpleTransfer(&axiDMA,(int)A,
    SIZE*sizeof(int), XAXIDMA_DMA_TO_DEVICE);

status = XAxiDma_SimpleTransfer(&axiDMA,(int)B,
    SIZE*sizeof(int), XAXIDMA_DMA_TO_DEVICE);

status = XAxiDma_SimpleTransfer(&axiDMA,(int)
    res_hw,SIZE*sizeof(int),
    XAXIDMA_DEVICE_TO_DMA);

// SIZE = DIM*DIM
```

Etape 5 :: On change le STACK et le HEAP size selon la dimension des matrices input. le STACK size est calculé de façon à libérer de l'espace mémoire en fonction des variables déclarées dans la fonction main et les fonctions qu'elle appelle (dans notre cas une seule fonction). On déclare au début de la fonction main 4 matrices DIM*DIM de type int. Donc le STACK size est 4*4*1024*2 bits = 32 octets. Ici, on multiplie par 2 pour assurer que ça fonctionne dans le pire des cas.

Étape 6 :: On programme le FPGA par le hardware et on crée une application SDK (à partir du code qu'on vient de décrire) qu'on exécute sur la carte. On obtient l'affichage suivant :

```
**********************************************************************
********** ROB306, Matrix Multiplication, Groupe 2  **********
**********      Solution : seq1_16_pip_cols_resh_ab  **********
**********************************************************************

 Initialising HLS accelerator core

 Initialising AxiDMA

Running Matrix Multiplication in the SOFTWARE

---> Total run time for SOFTWARE on Processor is 5996 cycles over 1024 tests.

Running Matrix Multiplication in the HARDWARE
Cache cleared !

---> Total run time for AXI DMA + HLS accelerator is 2600 cycles over 1024 tests


Acceleration factor: 2.306

SW and HW results match!
```

Fig. V.11. Output du SDK

E. Résultats

On va tracer dans cette section les graphes qui résument les résultats obtenus après exécution en temps réel. Les résultats numériques sont tous détaillés dans le drive de notre groupe.

1) Le temps d'exécution: Afin de calculer le temps d'exécution mis par la carte FPGA+IP HLS pour donner le résultat de la multiplication, on repère tout d'abord le nombre moyen des cycles par exécution (le output du SDK). Ensuite, le temps d'exécution est alors :

$$\text{Temps d'exécution} = \frac{Nombretotaldescycles}{Fréquence}$$

On obtient alors les figures suivantes pour différentes tailles des matrices : 16, 32, 64 et 128.

- Pour la dimension 16, les quatre solutions ont quasiment les mêmes performances et apportent une amélioration par rapport au processeur ARM9 d'un ratio de 2.3.
- Pour les dimensions 32, 64 et 128, les 3 premières solutions sont proches en performance et apportent une très bonne amélioration par rapport à la qua-

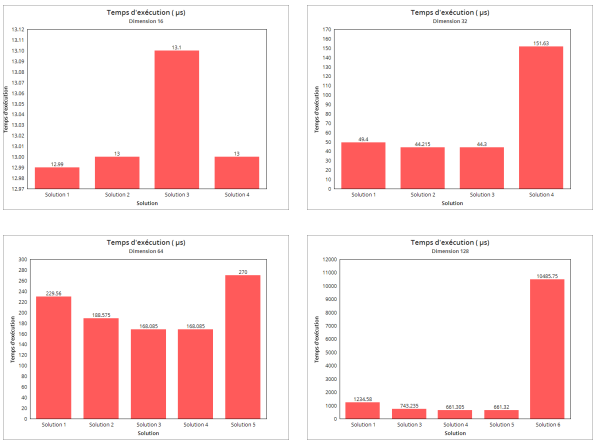


Fig. V.12. Le temps d'exécution sur Chip pour les dimensions 16, 32, 64 et 128 et pour différentes solutions HLS

trième solution (aucune optimisation HLS) et par rapport au processeur ARM9.

2) Utilisation des ressources matérielles: L'analyse du temps d'exécution pour chaque solution ne suffit pas pour décider si une solution est efficace ou pas. Ainsi, on analyse en plus la moyenne de la consommation des ressources matérielles :

$$\text{Moyenne ressources (\%)} = \frac{LUTs(\%)+BRAM(\%)+DSP(\%)+FF(\%)}{4}$$

On obtient alors les figures suivantes pour différentes tailles des matrices : 16, 32, 64 et 128.

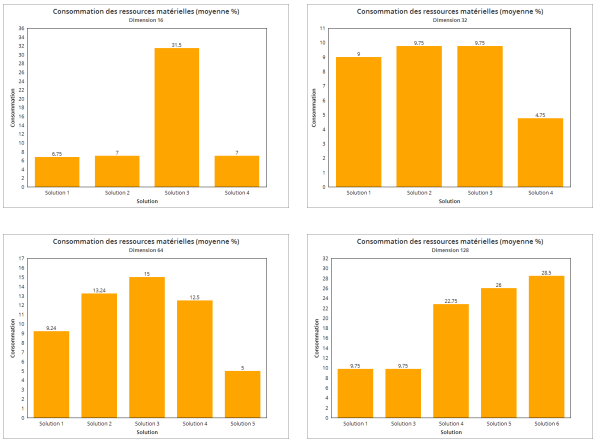


Fig. V.13. Moyenne de l'utilisation matérielle pour les dimensions 16, 32, 64 et 128 et pour différentes solutions HLS

- Pour la dimension 16, les résultats ne reflètent rien, il y a un pic de consommation pour la solution 3 par rapport aux solutions 1 et 2 malgré qu'elles ont quasiment le même temps d'exécution.
- Pour les trois autres dimensions, on peut voir que, quasiment, moins une solution met du temps lors de son exécution plus elle utilise des ressources matérielles.

Il est aussi intéressant d'analyser aussi l'utilisation de chaque ressource matérielle pour chaque solution. On obtient alors les figures suivantes pour différentes tailles des matrices : 16, 32, 64 et 128.

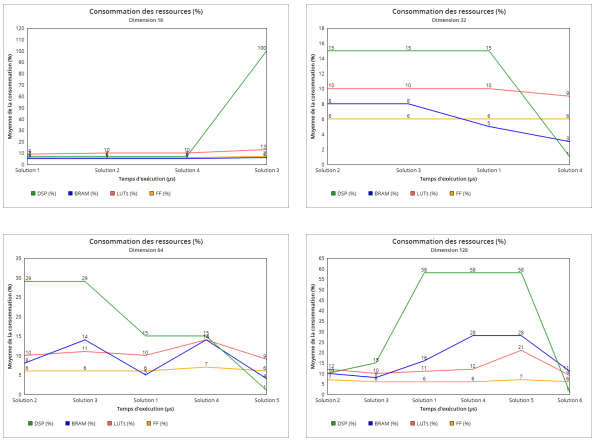


Fig. V.14. L'utilisation matérielles pour les dimensions 16, 32, 64 et 128 et différentes solutions HLS

- Pour la dimension 16, l'utilisation des ressources matérielles est quasiment constante pour toutes les solutions sauf pour les DSPs. On voit que la solution 3 consomme toutes les 220 DSP ce qui explique le pic qu'on a trouvé pour la moyenne. Cette IP HLS utilise alors un très grand nombre de multiplications.
- L'utilisation des FLIP-FLOPs est quasiment constante : 6 ou 7 %.
- La consommation des DSPs est plus grande que les autres ressources. Ceci est prévisible puisque l'IP effectue plusieurs multiplications.
- Une autre remarque attendue est que la solution sans optimisation consomme toujours très peu de ressources.

Enfin, Pour comprendre la relation temps d'exécution - utilisation matérielle, on trace les figures suivantes :

moyenne de l'utilisation matérielle = f(temps d'exécution)

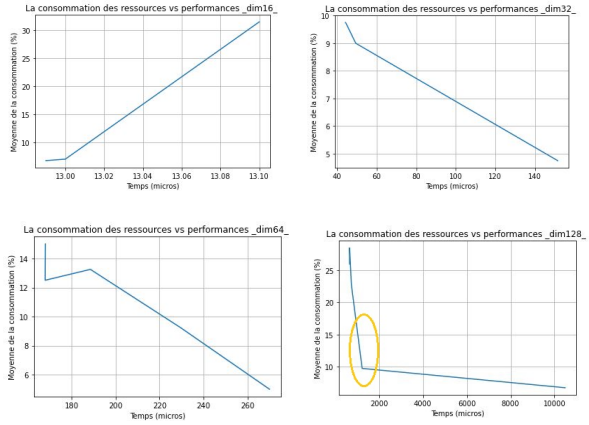


Fig. V.15. L'utilisation matérielle pour les dimensions 16, 32, 64 et 128 en fonction du temps d'exécution

- Pour les solutions de la dimension 16 les résultats ne sont pas cohérents : on ne peut rien décider surtout que les trois solutions ont les mêmes performances et les différentes directives HLS ne permettent pas des performances différentes.
- Pour la dimension 32, moins le temps d'exécution est, plus la consommation des ressources matérielles est. Le même résultat reste valable pour la dimension 64 (sauf pour la solution 3).
- Par contre, pour les solutions de la dimension 128, il est claire que plus on augmente les performances de la solution plus on consomme de ressources. On remarque l'apparition d'un point de Pareto qui met en évidence cette relation.

F. Conclusion

On présente maintenant Les meilleurs résultats obtenus pour chaque dimension. Une remarque très importante à donner est que tous ces temps d'exécution représentent : le temps d'exécution mis par l'IP Core pour effectuer la multiplication + le temps d'exécutions mis pour effectuer les transferts des données depuis et vers l'AXI DMA.

Dimension	meilleur temps d'exécution μs
16	12.99
32	44.215
64	168.085
128	661.305

TABLE V.2
MEILLEURES PERFORMANCES POUR CHAQUE DIMENSION

VI. CONCLUSION

Ce projet a eu comme objectif la comparaison entre différentes directives d'optimisation de l'algorithme naïf de multiplication de matrices.

La première partie a été dédiée pour l'optimisation algorithmique et sous PC de cette application. Un algorithme séquentiel amélioré qui permet de réduire les accès mémoire a été introduit. Ensuite, on a adopté la démarche de la multiplication par bloc au lieu de la multiplication terme à terme. En second lieu, on a essayé de paralléliser le traitement de ces trois algorithmes en utilisant OpenMP. Outre que l'optimisation algorithmique on a opté pour des optimisations logicielles sous PC en utilisant des options de compilation à l'instar de : $-O_1$, $-O_2$, $-O_3$, $-O_s$, $-O_g$ et $-O_{fast}$.

La deuxième partie a été dédiée pour le calcul et l'optimisation sous le processeur ARM9. On a exécuté le code C qui traduit l'algorithme séquentiel amélioré en effectuant encore quelques améliorations algorithmiques. On a étudié aussi l'influence du cache et de la prédiction de branchement sur les performances du processeur. Enfin, il s'est avéré qu'il est trop lent.

Ensuite, on a créé lors de la troisième partie des accélérateurs matériels sous Vivado HLS. On s'est appuyé sur l'algorithme séquentiel amélioré et on a ajouté des directives d'optimisation (Pipeline, partition, etc.). Puis, pour faciliter l'implémentation sur chip, on a créé des IP HLS avec des interfaces AXI4-Lite ou AXI4-Stream. La dernière étape était d'estimer les performances de ces accélérateurs matériels ainsi que leur utilisation du matériel en utilisant *C synthesis*.

La quatrième et dernière partie a été dédiée pour l'implémentation sur chip de l'accélérateur matériel à travers un réseau AXI4-Stream. Pour cette phase, on a utilisé AXI-DMA pour effectuer les transferts de données entre l'IP core et le DDR ainsi que l'AXI-timer afin de repérer le nombre des cycles exécutés en temps réel. On a trouvé que l'IP HLS + FPGA sont beaucoup plus performants que le processeur ARM9 et s'approchent des performances sur PC.

Pour conclure, on trace le tableau suivant qui résume les meilleurs résultats qu'on a eu sur PC, ARM9 et HLS + FPGA (+temps de transferts DMA).

Dimension	PC	ARM9	HLS + FPGA
16	4.1	14.1	12.99
32	11.0	226.7	44.215
64	43.8	1692.6	168.085
128	220.3	12281.4	661.305

TABLE VI.1
MEILLEURS TEMPS D'EXÉCUTION (μs) POUR LES DIFFÉRENTES CONFIGURATIONS

Enfin, on trouve que même les meilleures solutions utilisées pour les $3^{ème}$ et $4^{ème}$ n'utilise que le $\frac{1}{5}$ (ou

moins) des ressources matérielles disponibles. Ainsi, une piste à suivre maintenant est de recréer de nouvelles IP qui utilisent plus de ressources tout en optimisant au maximum le temps d'exécution.

REFERENCES

[1] J. L. Sanchez-Lopez, J. Pestana, S. Saripalli, and P. Campoy, "An approach toward visual autonomous ship board landing of a vtol uav," *Journal of Intelligent & Robotic Systems*, vol. 74, no. 1, pp. 113–127, 2014. URL: https://link.springer.com/article/10.1007/s10846-013-9926-3.

[2] Xlinx, *Embedded System Tools Reference Manual*, 2019.1 ed., 5 2019. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1043-embedded-system-tools.pdf.

[3] Xlinx, *Gererating Basic Software Platforms Reference Guide*, 2019.1 ed., 5 2019. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1138-generating-basic-software-platforms.pdf.

[4] Xlinx, *Xlinx Software Development Kit (SDK) User Guide*, 2019.1 ed., 5 2019. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1145-sdk-system-performance.pdf.

[5] Xlinx, *Vivado Design Suite User Guide High-Level Synthesis*, 2019.1 ed., 7 2019. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug902-vivado-high-level-synthesis.pdf.

[6] Xlinx, *Vivado Design Suite Tutorial High-Level Synthesis*, 2019.1 ed., 5 2019. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug871-vivado-high-level-synthesis-tutorial.pdf.

[7] Xlinx, *Vivado HLS Optimization Methodology Guide*, 2017.4 ed., 12 2017. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1270-vivado-hls-opt-methodology-guide.pdf.