ROB306 - Accélérateur Matériel avec HLS

yan CHEN, kai ZHANG, mengyu PAN, Icare SAKR

October 2020

1 Introduction

Nous évaluons dans ce projet les performances en temps d'exécution et en ressources de la fonction Vision/traitement d'images filtres dans 3 configurations possibles : 1. sur processeur généraliste disponible sur PC (Intel) 2. sur processeur embarqué ARM9 3. en accélérateur matériel sur circuit reconfigurable FPGA XC7Z020 disponible sur carte zedboard.

L'opération de filtrage permet de modéliser aussi bon nombres de phénomènes comme le flou ou à contrario l'accentuation des contours. Cette opération consiste à appliquer un filtre (noyau de convolution) définit comme une fenêtre carrée de taille N*N à tous pixels de l'image. Ainsi le pixel va être modifié en fonction des informations de son voisinage. On définit que N est un entier impaire et R est le rayon du filtre. On pose $N = 2 * R + 1$. Si l'image et le filtre sont considérés comme des fonction à valeur de $N2 \rightarrow N$ alors l'équation de calcule de convolution comme suit.

$$image : (i, j) \in [0, length - 1] * [0, width - 1] \rightarrow im(i, j) \tag{1}$$

$$filtre : (m, n) \in [-R, R] * [-R, R] \rightarrow p(m, n) \tag{2}$$

$$imagefiltre(i, j) = \sum_{-R \leq m,n \leq R} image(i + m, j + n) \cdot filtre(m, n) \tag{3}$$

On utilise filtre moyenne dans ce projet pour simplifier le processus de convolution d'image. Le noyau de convolution comme indiqué ci-après.

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

FIGURE 1 – Le noyau de convolution

Les taches sont distribuées par personne dans ce projet.

1. T1 : ingénieur logiciel (PC) : Icare SAKR
2. T2 : Ingénieur logiciel embarqué (ARM9/FPGA) : Yan CHEN
3. T3 : Ingénieur HLS : kai ZHANG
4. T4 : Ingénieur intégration FPGA SoC (System on Chip) : Pan Mengyu

2 Evaluation de performances sur PC

Dans cette section nous évaluons notre fonction de filtre moyeneur sur PC, en testant différents méthodes et approches pour optimiser le temps d'exécution de notre fonction.

2.1 Description PC

La configuration d'exécution matérielle utilisée pour réalisé cette évaluation se résume dans les figures 2 et 3.

CPU Name:	Intel(R) Core(TM) i7-7820HQ CPU
Frequency	2.90GHz
Cores	4
Memory	16 GB LPDDR3
Os	MacOS
Compiler	gcc

FIGURE 2 – Descritpion PC

L1$	256 KiB	L1I$	128 KiB	4x32 KiB	8-way set associative	
		L1D$	128 KiB	4x32 KiB	8-way set associative	write-back
L2$	1 MiB			4x256 KiB	4-way set associative	write-back
L3$	8 MiB			4x2 MiB	12-way set associative	write-back

FIGURE 3 – Descritpion Caches Corei7-7820HQ

2.2 Implémentation du filtre moyen

2.2.1 Complexité de la convolution

Afin de résoudre le problème de convolution 2D, l'approche la plus simple consiste à boucler tous les pixels de l'image et tous les éléments du noyau en une seule fois. Cet algorithme est appelé algorithme naïf. Il utilise 4 boucles imbriquées, les 2 boucles extérieures sur les lignes et les colonnes de l'image et les 2 boucles intérieures sur les lignes et les colonnes du noyau. Pour un noyau 5×5, selon l'équation 3, il faut 25 opérations de multiplication-accumulation pour chaque pixel.

Ainsi pour une image $M \times N$ (échelle de gris) et un filtre $m \times n$, chaque pixel nécessite des calculs en $\Theta(mn)$, donc la convolution 2D aurait une complexité approximative de $\Theta(MNmn)$. (Bien sûr, ceci dépend de si on fait un padding sur les côtés des images ou non, mais en supposant que m et n sont petits par rapport à M et N, cela ne devrait pas faire une trop grande différence). En fait, ici on ne considère qu'une seule composante de l'image dans une base de couleur donnée, pour une image à 3 composantes (r,g,b par exemple), la complexité est multipliée par 3 (un filtrage par composante).

Dans ce qui suit, on ne traite qu'une seule composante de l'image.

2.3 Optimisations de l'exécution

Nous utilisons trois approches pour optimiser le temps d'exécution de notre fonction de filtre moyen : optimisation à la compilation, optimisation algorithmique, et l'optimisation par parallélisme OpenMP.

2.3.1 Optimisation à la compilation

Nous utilisons le compilateur gcc et faisons varier les options d'optimisation de compilation.

Les différents options d'optimisations à la compilation utilisés sont :

- O0 : Pas d'optimisation. Réduit le temps de compilation et fait en sorte que le debugging produise les résultats attendus. C'est la valeur par défaut.

- O1 : Avec -O1 (ou -O), le compilateur essaie de réduire la taille du code et le temps d'exécution, sans effectuer d'optimisations qui prennent beaucoup de temps de compilation.

- O2 : Optimiser encore plus. gcc réalise presque toutes les optimisations prises en charge qui n'impliquent pas de compromis entre l'espace et la vitesse. Par rapport à -O1, cette option augmente à la fois le temps de compilation et la performance du code généré.

- O3 : Optimiser encore plus. -O3 active toutes les optimisations spécifiées par -O2 et active également plus de drapeaux d'optimisation.

- Os : Il permet de régler le compilateur sur la taille du code plutôt que sur la vitesse d'exécution, et d'effectuer d'autres optimisations destinées à réduire la taille du code.

- Ofast : Ne tiens pas compte du respect de normes strictes. -Ofast active toutes les optimisations -O3. Il permet également les optimisations qui ne sont pas valables pour tous les programmes conformes aux normes.

Les performances de notre code sont évalués sur un jeu de test constitué d'un ensemble d'images de tailles variable, et une taille fixe du noyau de convolution (3x3) pour pouvoir exploiter l'aspect algorithmique (notamment le déroulement de boucle). Nous évaluons l'algorithme naïf avec déroulement des boucles relatifs au produit de convolution d'un pixel donné. Les temps d'exécutions relevés sont les moyennes de 4 exécutions successifs d'une configuration donnée.

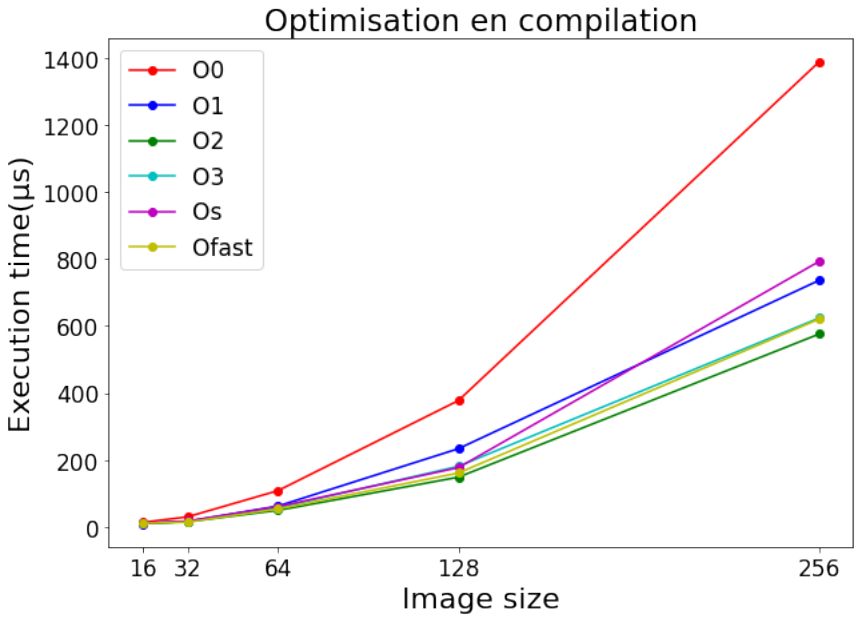

FIGURE 4 – Temps d'exécution en fonction de l'optimisation à la compilation

Nous remarquons que dans le cas de notre algorithme, la performance dépend de l'option d'optimisation et de la taille de notre image. Pour une image 256x256 l'option la plus performante (en termes de temps d'exécution) est -O2.

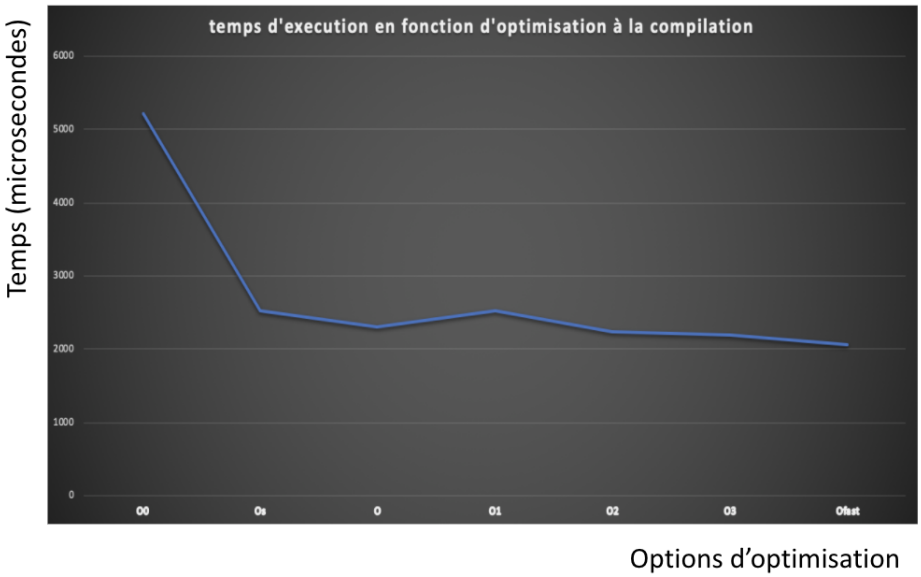

FIGURE 5 – Temps d'exécution en fonction de l'optimisation à la compilation pour un taille d'image fixe 500x500

2.3.2 Optimisation algorithmique

Dans cette section, nous allons évaluer l'impact de l'algorithme sur la performance de notre filtre moyenner. Pour cela, nous avons testé 3 variantes de l'algorithme : L'algorithme naïf, l'algorithme avec déroulement de boucles, et l'algorithme par séparation, que nous allons explicité dans la suite.

Algorithme Naïf (AlgoNaif)

Comme nous l'avons vu précédemment, l'algorithme naïf utilise 4 boucles imbriquées, les 2 boucles extérieures sur les lignes et les colonnes de l'image et les 2 boucles intérieures sur les lignes et les colonnes du noyau de convolution.

```
13
14      for(int i=1; i<rows-1;i++)
15      {//on ne fait pas de padding ici
16          //mais on ne traite pas les pixels de 1ere et derniere position
17          for(int j=1; j<cols-1; j++)
18          {
19              for(int ki = 0; ki< kernelSize; ki++)
20              {
21                  for(int kj = 0; kj< kernelSize; kj++)
22                  {
23                      int pad = kernelSize/2;
24                      B[i][j]+=A[i+ki-pad][j+kj -pad]*kernel[ki][kj];
25                  }
26
27              }
28
29          }
30      }
```

FIGURE 6 – Algorithme Naïf : 4 boucles imbriqués

Algorithme avec déroulement de boucles (AlgoUnroll)

La première optimisation algorithmique est le déroulement de boucle. Un speedup moyen de 2,5× peut être obtenu en déroulant manuellement la boucle imbriquée sur le noyau (3x3) en 9 multiplications. Ainsi l'algorithme se réduit à deux boucles uniquement, permettant le balayage de l'image (voir figure 7). Cependant ceci nécessite de connaître à priori la taille du noyau de convolution.

```
12
13      for(int i=1; i<rows-1;i++)
14      {//on ne fait pas de padding ici
15          //mais on ne traite pas les pixels de 1ere et derniere position
16          for(int j=1; j<cols-1; j++)
17          {
18              B[i][j] = A[i-1][j-1]*kernel[0][0] +
19                        A[i-1][j]*kernel[0][1]+
20                        A[i-1][j+1]*kernel[0][2]+
21                        A[i][j-1]*kernel[1][0]+
22                        A[i][j]*kernel[1][1]+
23                        A[i][j+1]*kernel[1][2]+
24                        A[i+1][j-1]*kernel[2][0]+
25                        A[i+1][j]*kernel[2][1]+
26                        A[i+1][j+1]*kernel[2][2];
27          }
28      }
```

FIGURE 7 – Algorithme par déroulement : 2 boucles

Algorithme par séparation (AlgoSep)

Un noyau de convolution séparable est un vecteur de nombres réels qui peut être décomposé en projections horizontales et verticales et peut donc être appliqué indépendamment aux lignes et aux colonnes du domaine spatial pour assurer le filtrage. Il s'agit d'un cas spécial de la convolution plus générale, mais sa mise en œuvre est plus efficace sur le plan algorithmique. Pour un noyau 3×3, ceci réduit le nombre de multiplications par pixel à 6. Du point de vue algorithmique, l'algorithme de convolution par séparation devrait toujours être préféré si le noyau est séparable. Il a une complexité temporelle linéaire en la taille du noyau, alors que la complexité de l'algorithme non séparé est quadratique.

Le noyeau d'un filtre moyen est séparable. En fait :

$$\frac{1}{3}\begin{bmatrix}1\\1\\1\end{bmatrix} * \frac{1}{3}\begin{bmatrix}1 & 1 & 1\end{bmatrix} = \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\1 & 1 & 1\\1 & 1 & 1\end{bmatrix}$$

FIGURE 8 – Décomposition du filtre moyen en un produit de deux filtres simples

```
9
10      //Passage horizontal
11      //#pragma omp parallel for
12      for(int i=1; i<rows-1;i++)
13      {
14          //#pragma simd //vectorization de la boucle
15          for(int j=1; j<cols-1; j++)
16          {
17              B[i][j] = A[i][j-1]*kernel[0] +
18                        A[i][j]*kernel[1] +
19                        A[i][j+1]*kernel[2];
20
21          }
22      }
23      //Passage vertical
24      //#pragma omp parallel for
25      for(int i=1; i<rows-1;i++)
26      {
27          //#pragma simd //vectorization de la boucle
28          for(int j=1; j<cols-1; j++)
29          {
30              A[i][j] = B[i-1][j]*kernel[0] +
31                        B[i][j]*kernel[1] +
32                        B[i+1][j]*kernel[2];
33
34          }
35      }
36  }
```

FIGURE 9 – Algorithme par séparation du filtre

Résultats comparaison

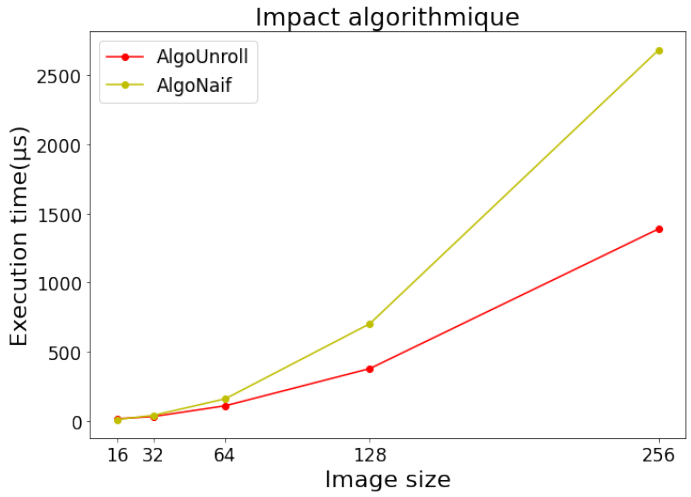

FIGURE 10 – Impact algorithmique sur la performance

2.3.3 Optimisation par parallélisme OpenMP/MPI

OpenMP est le standard de facto pour la programmation en mémoire partagée, et est basé sur un ensemble de directives de compilation ou de pragmas, combiné à une API de programmation pour spécifier les régions parallèles, l'étendue des données, la synchronisation, etc. Il prend également en charge la configuration du temps d'exécution grâce à l'utilisation de variables d'environnement d'exécution, par exemple $OMP\_NUM\_THREADS$ pour spécifier le nombre de threads au moment de l'exécution. OpenMP est une approche de programmation parallèle portable et est supporté en C et C++.

Pour notre algorithme, j'ai réalisé le parallélisme de l'algorithme par déroulement de boucle avec multi-threading et multi-coeur. L'image et donc décomposée en longueur n_coeurs parties et chaque partie est envoyée à un coeur pour filtrage. Pour un coeur donné, la boucle de balayage de lignes est parallélisée par multithreading avec OpenMP (avec pragma omp parallel 4). J'ai testé ceci avec différent nombres de coeurs à 4 threads chacun.

Résultats

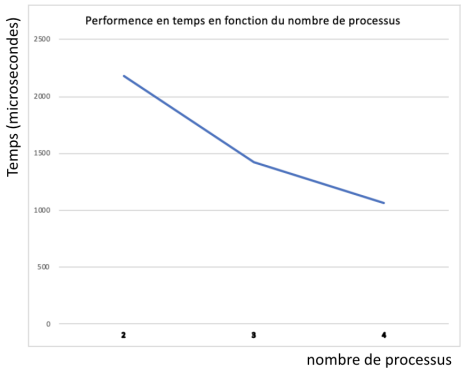

FIGURE 11 – Temps d'éxecution en fonction du nombre de coeurs (à 4 threads chacun) pour une image de taille 500x500

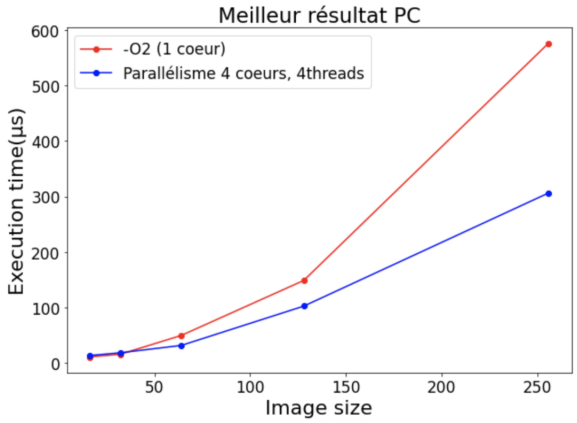

FIGURE 12 – Comparaison parallélisme, optimisation de compilation pour l'algorithme de déroulement de boucle.

2.4 Meilleur résultat PC

Finalement le meilleur résultat sur PC est obtenu pour un parallélisme avec 4 coeurs et 4 threads chacun en utilisant l'algorithme de déroulement de boucle.

2.5 Piste d'amélioration

2.5.1 Convolution dans l'Espace de Fourrier

Une convolution devient un produit si on l'applique dans l'espace de Fourier, on peut donc exprimer l'opération I $\star$ m (I étant l'image, m étant le masque avec lequel vous filtrez) comme I $\star$ m=F^{-1}(F(I).F(m)) (où F est la transformée de Fourier) qui nous coûtera des opérations $\Theta(MNlog(MN))$ aven l'algorithme FFT (Fast Fourier Transform). Le fait que ce soit moins cher que ce que les méthodes précédemment évoqués dépend de la taille du masque filtrant : Pour les petits filtres, une application directe est moins coûteuse, mais si le filtre applique un masque sur une zone qui est un sous-ensemble important de l'image, alors la transformée de Fourier peut devenir moins chère.

3 Evaluation de performances sur processeur emparqué ARM9

On test le temps d'exécution de code sur ARM9 de Zedboard dans cette partie. L'objectif est de faire la comparaison entre plateformes différents, PC et ARM9. Normalement, les deux palteformes sont très différentes pour le temps d'exécution de convolution d'image. Il est important de noter que le système de traitement Zynq comprend non seulement le processeur ARM, mais un ensemble de ressources de traitement associées formant une unité de traitement d'application (APU), et d'autres interfaces périphériques, la mémoire cache, les interfaces de mémoire, l'interconnexion et les circuits de génération d'horloge. Un schéma bloque montrant l'architecture du PS est illustré à la Figure 13, où l'APU est mis en évidence [2].

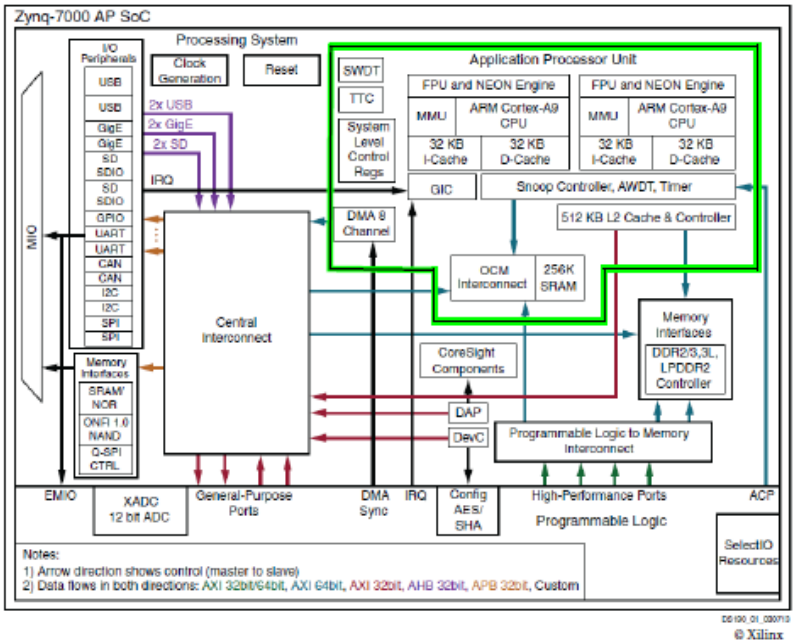

FIGURE 13 – PS de ZYNQ

3.1 Outil de test

On utilise principalement deux outils logiciels dans cette partie. Ceux sont Vivado 2019.1 et Xilinx SDK 2019.1. Premièrement, on crée une IP dans Vivado et puis teste le code dans Xlink SDK. Les détail d'opération, comme indique ci-après.

Vivado 2019.1

1. Création d'IP sur Vivado 2019.1
2. Génération de output produit et HDL Wrapper
3. Génération de bitstream
4. Exportation de plateform de matériel pour outil de développement de logiciel
5. Lancement de Xilinx SDK

Xilinx SDK 2019.1

1. Génération de projet
2. Création de source fichier et head fichier
3. Mesure de temps d'exécution
4. Analyse de données

3.2 Création d'IP

On ne considère pas les autres parties de Zedboard sauf ARM9 dans cette partie. Ici, l'IP est donc simple. Ensuit, la port M_AXI_GPO_ACLK faut se connecter à la port FCLK_CLK3 dans l'IP qui est très important étape. Comme indiqué sur Figure 14.

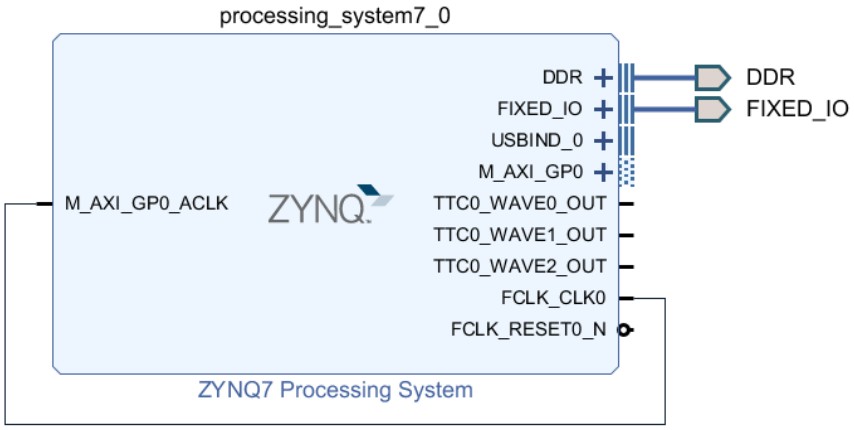

FIGURE 14 – IP pour ARM9

3.3 Algorithme d'optimisation

On utilise trois méthodes d'optimisation pour accélérer le code. Le première est compilation optimisation. Le deuxième est convolution par bloque. Le dernière est utilisation de Cache. Ils fonctionnent bien sur l'accélération de convolution d'image.

3.3.1 Compilation optimisation

Le principe de compilation optimisation sur ARM9 est le même que sur PC. On fait les compilation configuration sur Xilinx SDK. Comme indique sur la Figure 15.

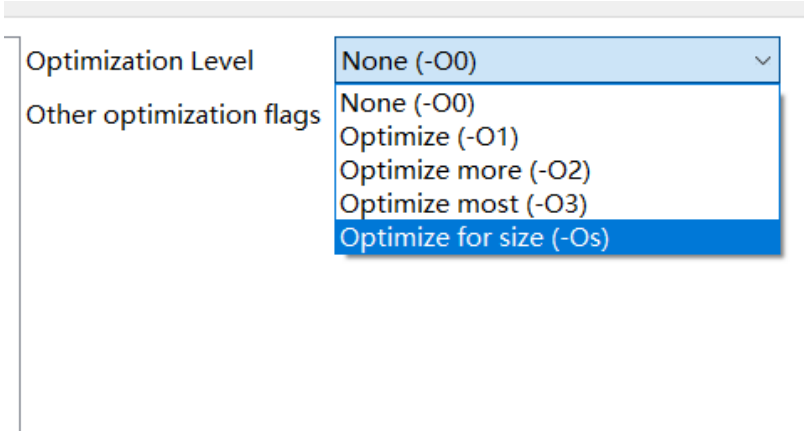

FIGURE 15 – Configuration de compilation optimisation

3.3.2 Convolution par bloque

La convolution par bloque signifie que une image est divisé à 4 bloques. Puis on fait respectivement la convolution à chaque bloque. Le résultat est met dans le position relevant de image de

résultat. L'idée viens de l'algorithme de multiplication de matrice par bloque. Cependant, ils sont différents. On fait 4 fois de convolution pour un "for boucle" dans cette algorithme. Le nombre total de boucle diminue donc beaucoup. Il réduit la surcharge de temps due à la long "for boucle". Figure 16 présente la procédure de convolution par bloque. Le code comme suit.

```
void block_convolution_mean(
                mat matrix[I_ROW][I_CLOM],
                mat kernel[K_ROW][K_CLOM],
                mat result[I_ROW][I_CLOM])
{
        int RED = (K_ROW-1)/2;
        int DEMI_SIZE_ROW = I_ROW/2-1
        int DEMI_SIZE_CLOM = I_CLOM/2-1
        for (int i = RED; i < I_ROW/2; i++)
        {
                for (int j = RED; j < I_CLOM/2; j++)
                {
                        int tmp_up_left = 0, tmp_down_left = 0;
                        int tmp_up_right = 0, tmp_down_right = 0;
                        for (int m = 0; m < K_ROW; m++)
                        {
                                for (int n = 0; n < K_CLOM; n++)
                                {
                                        //convolution en haut et a gauche
                                        tmp_up_left += matrix[i+m-RED][j+n-RED]*kernel[m][n];
                                        //convolution en base et a gauche
                                        tmp_down_left += matrix[i+DEMI_SIZE_ROW+m-RED][j+n-RED]*
                                             kernel[m][n];
                                        //convolution en haut et a droit
                                        tmp_up_right += matrix[i+m-RED][j+DEMI_SIZE_CLOM+n-RED]*
                                             kernel[m][n];
                                        //convolution en base et a droit
                                        tmp_down_right += matrix[i+DEMI_SIZE_ROW+m-RED][j+
                                             DEMI_SIZE_CLOM+n-RED]*kernel[m][n];
                                }
                        }
                        result[i][j] = tmp_up_left/(K_ROW*K_CLOM);
                        result[i+DEMI_SIZE_ROW-RED][j] = tmp_down_left/(K_ROW*K_CLOM);
                        result[i][j+DEMI_SIZE_CLOM-RED] = tmp_up_right/(K_ROW*K_CLOM);
                        result[i+DEMI_SIZE_ROW-RED][j+DEMI_SIZE_CLOM-RED] = tmp_up_right/(K_ROW*
                             K_CLOM);
                }
        }
}
```

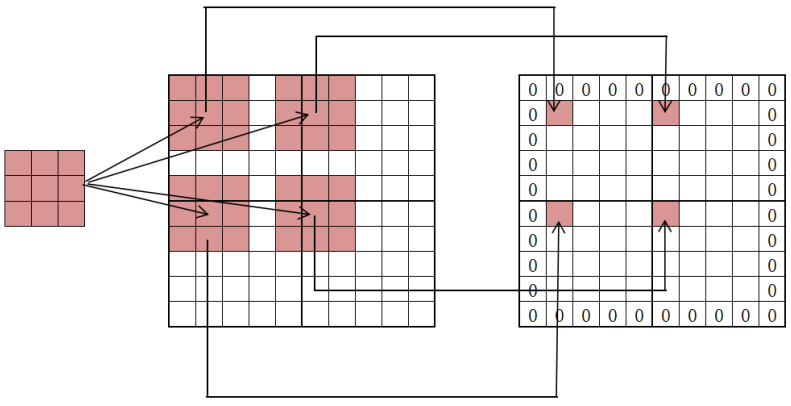

FIGURE 16 – Convolution par bloque

3.3.3 Utilisation de cache

Un cache de processeur est une mémoire plus petite et plus rapide, située au plus près d'une unité centrale de traitement (ou d'un cœur de microprocesseur), qui stocke des copies des données à partir d'emplacements de la mémoire principale qui sont fréquemment utilisés avant leurs transmissions aux registres du processeur pour réduire le temps de prendre donnée dans l'exécution de code [1]. ARM9 de Zedboard a deux processeur. Chaque processeur a trois caches. Un est L1 cache d'instructions (32KB), un est L1 cache de data (32KB), la dernière est L2 cache d'instructions et de data qui a plus grand taille de mémoire (512KB). La Figure 17 [2] indique la structure de cache dans APU de ARM9.

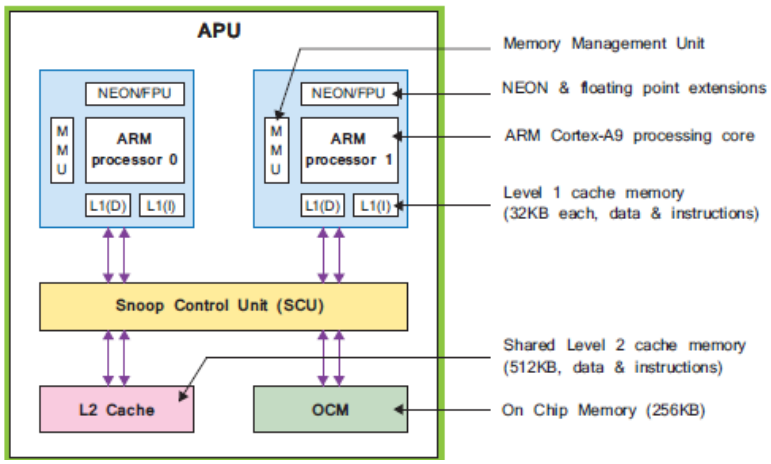

FIGURE 17 – Cache dans ARM9

Les cache sont ouverts en défaut. On utilise donc le code comme suit pour fermer respectivement les caches pour tester le temps d'exécution de convolution d'image.

```
#include "xil_cache.h"

Xil_DCacheDisable(); //Fermer le cache de data
Xil_ICacheDisable(); //Fermer le cache d'instruction
```

3.3.4 Test de temps d'exécution

On utilise les fonctions *XTime_GetTime(&start)*, *XTime_GetTime(&end)* dans laboratoire de *"xtime_l.h"* pour tester le temps d'exécution du code. Comme suit.

```
#include "xtime_l.h"

XTime_GetTime(&start1); //Noter le temps de debut
ass_Matrix(result, 0); // Le code teste
convolution_mean(imag, filter, result); //Le code teste
XTime_GetTime(&end1); //Noter le temps de cloture
printf("\n Output1 took %llu clock cycles.\n", 2*(end1 - start1)); //Exporter le nomnre de clock
    cycles
printf("Output1 took %g us.\n",
       1.0 * (end1 - start1) / (COUNTS_PER_SECOND/1000000)); // Exporter le temps d'execution de code
            , unite : us
```

3.4 Résultat

On concède la relation entre le temps d'exécution et taille d'image avec la utilisation des différents méthodes d'optimisation. On prend en compte l'image qui est 16*16, 32*32, 64*64 et 128*128.

3.4.1 Comparaison de temps d'exécution entre compilation optimisation et convolution par bloque

On discute l'influence de compilation optimisation et de convolution par bloque sur temps d'exécution. On trouve que compilation optimisation et convolution par bloque réduisent le temps d'exécution pour tous les tailles d image, surtout compilation optimisation dans la Figure 18. Ensuit, on essaie de combiner les deux méthodes d'optimisation pour réduire le temps d'exécution. Il obtient de bons résultats. En ajoutant la convolution par bloque sur la base de la compilation optimisation, la vitesse d'exécution du code devient plus rapide par rapport au cas de la convolution par bloque ou uniquement de la compilation optimisation. La méthode d'optimisation qui a le meilleurs résultats est compilation optimisation O3 plus convolution par bloque. Note : les caches sont ouverts pendant le test.

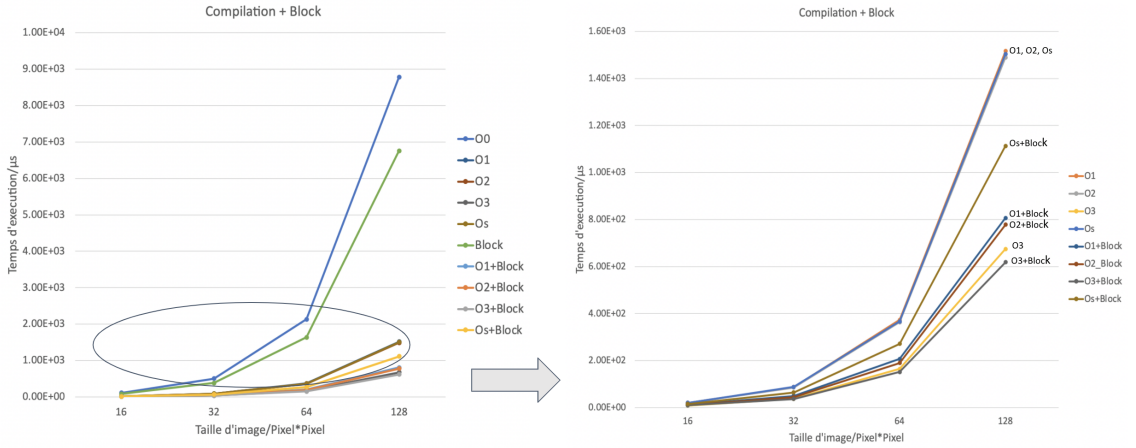

FIGURE 18 – Comparaison de temps d'exécution entre compilation optimisation et convolution par bloque

Selon les résultats de Figure 18, au fur et à mesure que la taille de l'image augmente, le temps d'exécution se réduit de plus en plus par les optimisations. Cependant, il n'est pas suffisant de prouver meilleur performance avec plus grand taille d'image. Il faut donc faire la comparaison de pourcentage de réduction de temps entre les différentes méthodes d'optimisation avec différentes taille d'image. Le pourcentage de réduction de temps est défini comme suit.

$$pourcentage = \frac{TNO - TO}{TNO} \times 100\% \tag{4}$$

où TNO : Temps d'exécution sans optimisation. TO : Temps d'exécution après optimisation.

Une schéma montant pourcentage de réduction de temps est illustré à la Figure 19. Presque même pourcentage de réduction de temps pour tous les tailles d'images. Cependant, à mesure que la taille de l'image augmente, le pourcentage du réduction de temps augmente légèrement. Au l'autre terme, il a petite meilleur performance avec plus grand taille d'image.

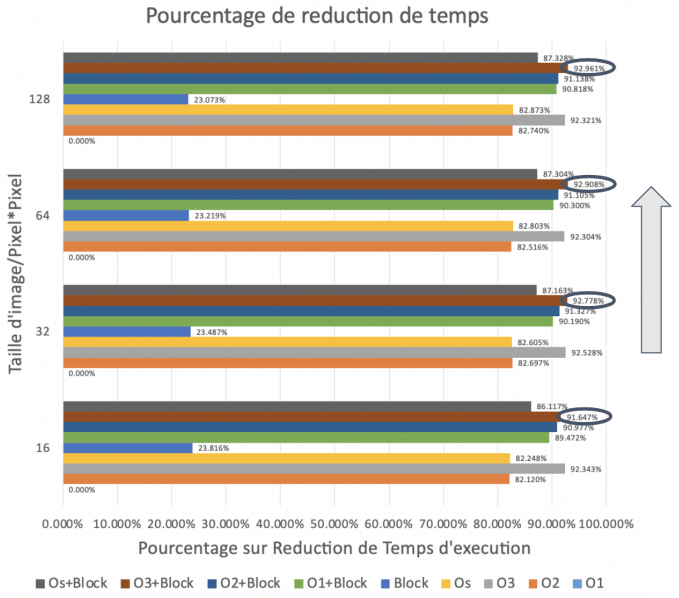

FIGURE 19 – Pourcentage de réduction de temps

3.4.2 Comparaison de temps d'exécution entre cache et sans cache

Normalement, ARM9 implémente le code avec les caches. On doit donc fermer les caches pour obtenir les résultats différents. On compare les quatre situations, cache, non cache d'instruction, non cache de data et non caches de instruction et de data. Le graphique montant comparaison de temps d'exécution entre cache et sans cache est illustré à la Figure 20. Il présente que cache peut considérablement améliorer la vitesse d'exécution du code et que la vitesse d'exécution du code sans cache de data est moindre que celle sans cache d'instruction.

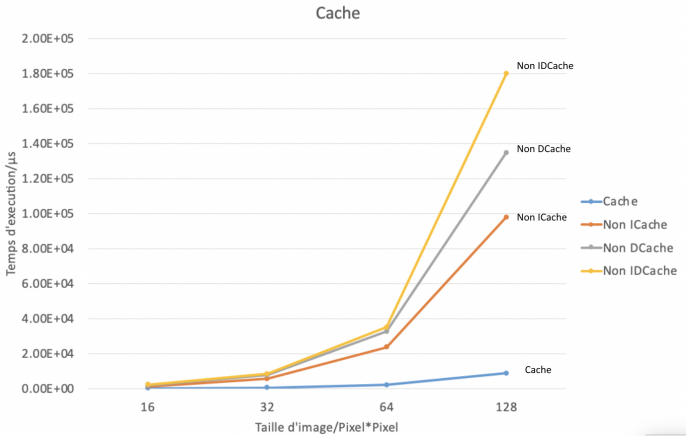

FIGURE 20 – Comparaison de temps d'exécution entre cache et sans cache

3.4.3 Comparaison de temps d'exécution entre ARM9 et PC

En fait, la vitesse d'exécution du code sur PC est normalement supérieur à celle sur ARM9. Ici on fait vérification et fait comparaison. La Figure 21 présent évidement qu'il faut plus de temps d'exécution du code sur ARM9. On réussit la vérification bien.

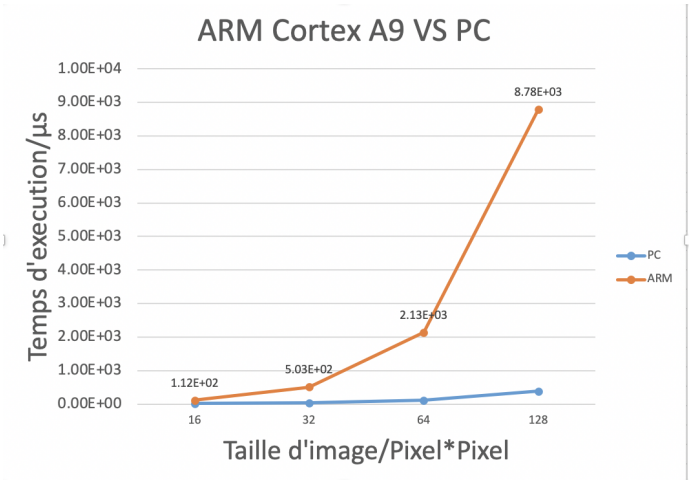

FIGURE 21 – Comparaison de temps d'exécution entre ARM9 et PC

4 Evaluation de performances sur HLS

4.1 Introduction

On exécute la programme de filtre moyenne sur Vivado HLS et évalue la performance dans les images avec différent taille. Vivado HLS aussi offre les directives différents que on peut les utiliser pour accélérer et réduire du temps de l'exécution.
Il fournit plusieurs directives pour améliorer les performances d'exécution. Voici quelques directives que nous avons utilisées dans notre projet :

— ARRAY_RESHAPE : Remodeler un tableau de un avec de nombreux éléments à un avec une plus grande largeur de mot. Il est utile pour améliorer les accès à la RAM de bloc sans utiliser plus de RAM de bloc.

— ARRAY_PARTITION : Partitionne les grandes baies en plusieurs baies plus petites ou en registres individuels, pour améliorer l'accès aux données et supprimer les goulots d'étranglement de la RAM de bloc.

— LOOP_MERGE :Fusionnez des boucles consécutives pour réduire la latence globale, augmenter le partage et améliorer l'optimisation logique.

— PIPELINE : Réduit l'intervalle d'initiation en permettant l'exécution d'opérations en chevauchement dans une boucle ou une fonction.

4.2 Solutions

On utilise plusieurs stratégies pour améliorer le programme. Comme le tableau 1,nous avons d'abord implémenté une version original de l'algorithme de filtre d'image. Ensuite, nous avons utilisé différentes stratégies pour améliorer les performances. Après avoir analysé la ressource et cherché le goulot d'étranglement de la performance, nous avons optimisé la version originale et développé raw1. Comme la figure 22, nous avons utilisé une variable temporaire pour stocker la somme des valeurs de pixel d'une petite région. Comparé au stockage de la somme dans le tableau, il réduit le temps de recherche de l'adresse dans un tableau qui a provoqué le goulot d'étranglement du programme. Enfin, nous avons appliqué différentes directives au code optimisé et comparé le temps d'exécution et les ressources.

Solution	Stratégies	
original(raw)	-	-
code optimisation(raw1)	-	-
solution1	pipeline(row,**col**)	-
solution2	pipeline+rewind(row, **col**)	-
solution3	pipeline(row,**col**)	Array_partition(block, **complete**, cyclic)
solution4	pipeline+rewind(row, **col**)	Array_partition(block, **complete**, cyclic)

TABLE 1 – Table de les solutions.Raw1 est la version optimisée de raw. Le mot en gras (stratégie) entre les bracets a obtenu les meilleures performances après comparaison entre différentes stratégies.

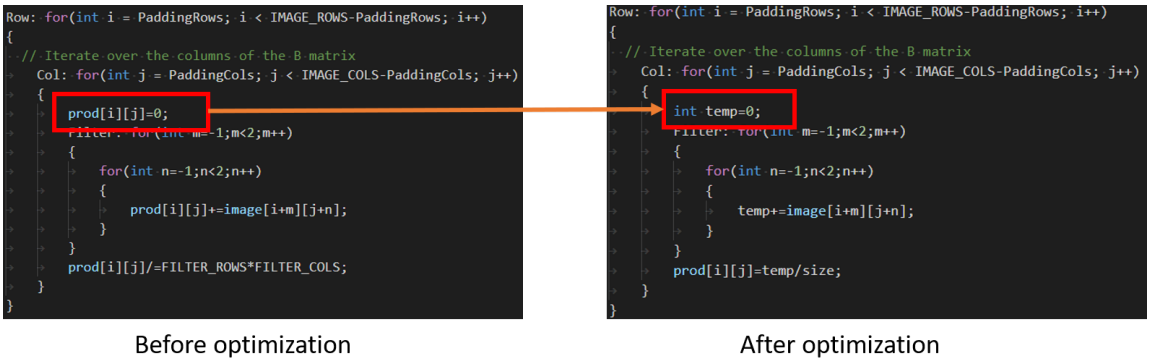

FIGURE 22 – Optimisation du code

4.3 Résultat

4.3.1 Résultat de code optimisation

Nous avons évalué l'impact de l'optimisation du code sur le temps d'exécution. Pour obtenir le résultat général, nous avons appliqué notre programme à trois images de tailles différentes ($16 \times 16, 32 \times 32, 64 \times 64$). Le temps d'exécution des différents programmes est représenté sur la figure 23. Nous pouvons voir qu'après l'optimisation code, le code est plus efficace avec moins de temps d'exécution.

4.3.2 Résultat de stratégies différents

Nous avons appliqué différentes directives basées sur un code optimisé afin de trouver la stratégie la plus pratique. Le temps de fonctionnement et le débit (bits flux) ont été pris en compte pour la comparaison quantitative et l'évaluation. Comme la figure 24, on peut voir que lorsque des directives plus compliquées sont appliquées, le temps d'exécution diminue et le débit augmente. Surtout pour la directive pipeline, elle a considérablement réduit le temps de fonctionnement. De plus, la partition de matrice a évidemment amélioré le débit, presque deux fois le pipeline. Lorsqu'on compare l'opération de rewind, elle peut accélérer légèrement le programme. Compte tenu de l'effet plafond, le temps de fonctionnement est principalement constitué du temps de fonctionnement nécessaire.

4.3.3 Diagramme de Pareto

Un graphique de Pareto est un type de graphique qui contient à la fois des barres et un graphique linéaire, où les valeurs individuelles sont représentées par ordre décroissant par des barres et le total cumulé est représenté par la ligne. Le but du graphique de Pareto est de mettre en évidence le plus important parmi un ensemble (généralement large) de facteurs.[4] Dans notre expérience, nous

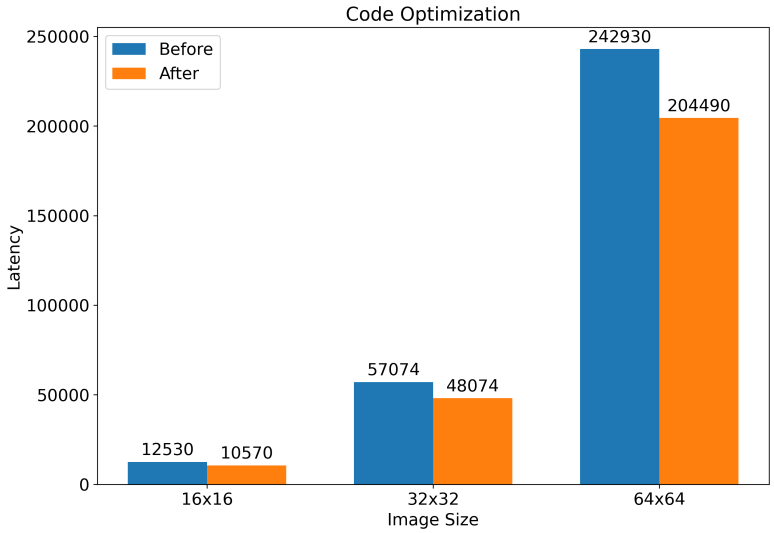

FIGURE 23 – Optimisation du code

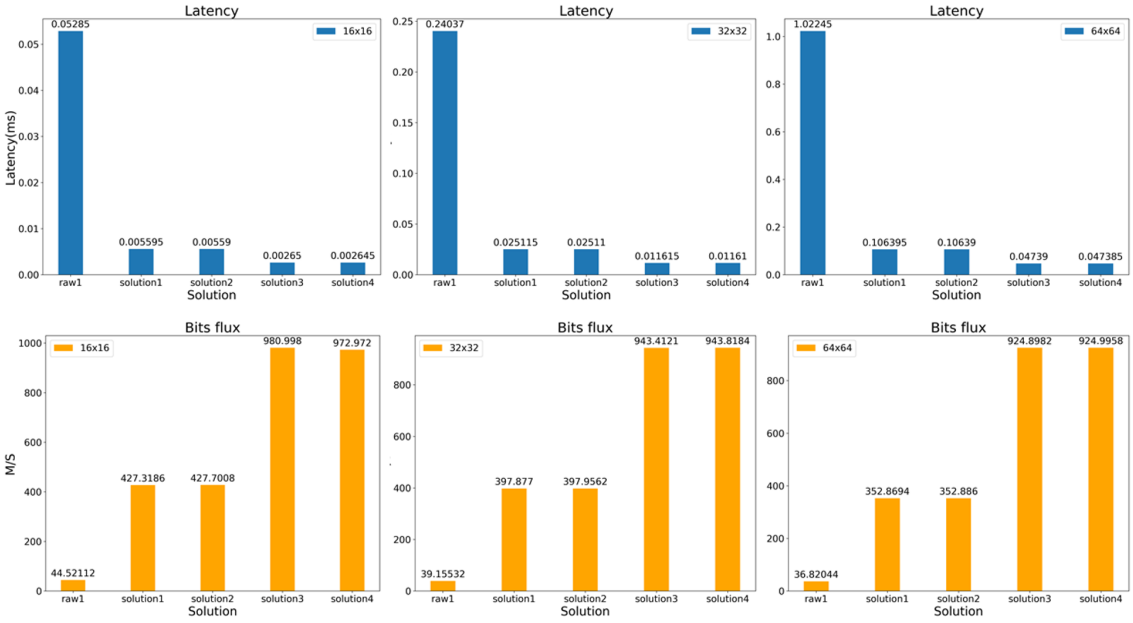

FIGURE 24 – Résultat de stratégies différents. Le premier ligne est du temps d'exécution et le deuxième est les débits

utilisons le graphique de Pareto pour analyser la relation entre le temps d'exécution et les ressources utilisées par le programme, comme illustré sur la figure **??**. Il a proposé que le temps d'exécution ait une corrélation négative avec les ressources de calcul. Avec plus de ressources de calcul, le code nécessite moins de temps d'exécution.

A travers le graphique de Pareto, nous pouvons tirer une conclusion : compte tenu de la ressource et du temps d'exécution, le pipeline + rewind est la stratégie la plus appropriée pour notre programme.

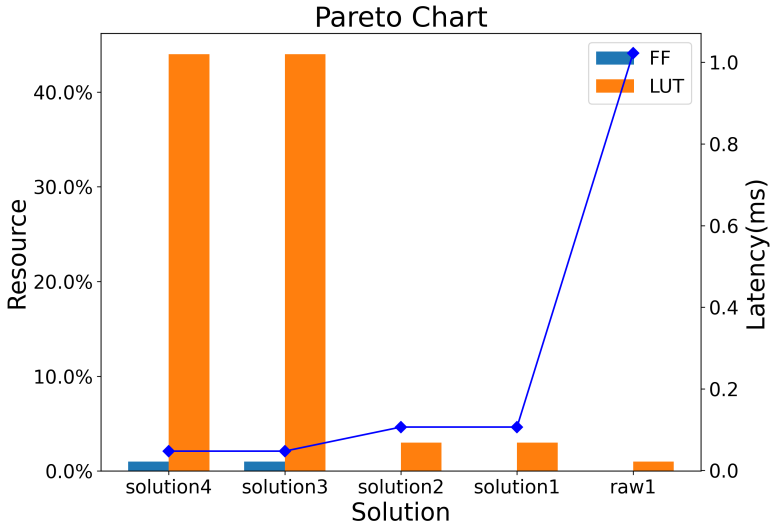

FIGURE 25 – Diagramme de Pareto.

5 Evaluation de performances intégration

5.1 Introduction du Méthode

Au début, nous avons essayé avec les bibliothèques de vidéo dans Vivado HLS mais il faut utiliser les bloques spéciales pour ces bibliothèques. La difficulté est que nous ne connaissons pas les bloques spéciales ni les connections parmi ces bloques. Dans ce cas, nous avons écrit tous les fonctions nécessaires . Les données sont 8 bits et nous voulons les transmettre par AXI pour accélère le calcul donc le schéma général est présenté dans le figure 26 .

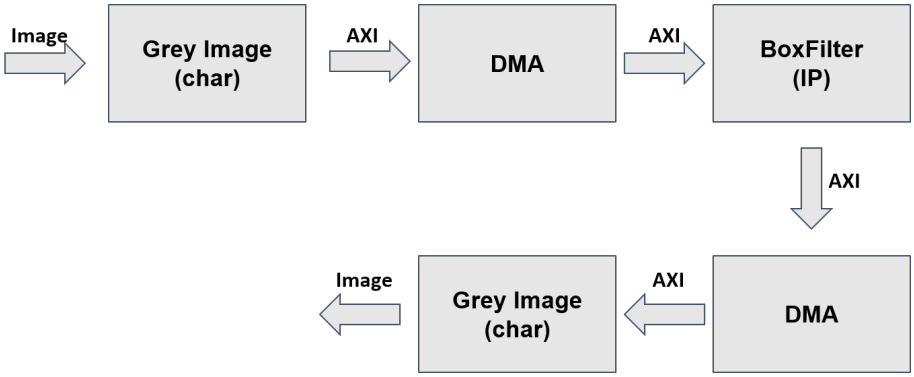

FIGURE 26 – schéma

5.2 Création du IP

Pour transmettre les données originales au données de AXI, nous avons consulté les documents nécessaires[3].Dans le design de Vivado HLS, nous avons défini le apaxiu<8,4,5,5> comme la donnée de AXI. Dans le top fonction, nous avons fait la filtrage moyenne par calculer la donnée moyenne. Après le synthesis de C et l'exportation RTL, on peut voir le IP dans le figure 27 dans le block design. Dans cet image, nous avons utilisé AXI Stream pour l'entre et la sortie de donnée et AXI Lite pour le contrôler.

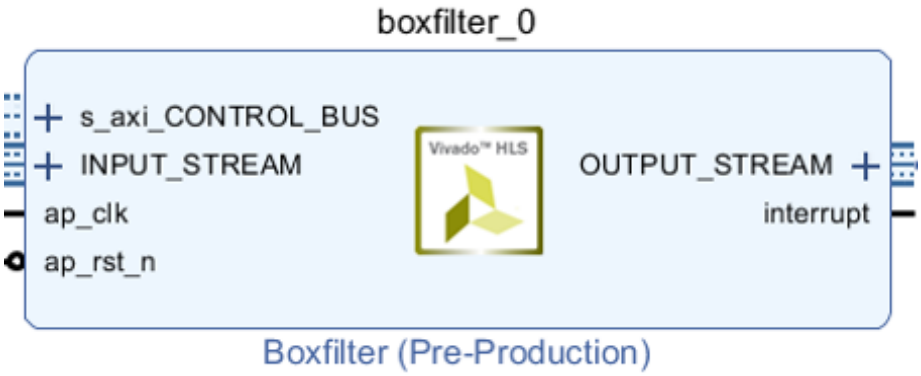

FIGURE 27 – mon IP

5.3 Implémentation du IP

Ensuite il faut implémenter un circuit pour mon IP. D'abord, c'est nécessaire d'implémenter une système de processeur et ses composants nécessaires comme le reset de système de processeur. On peut accéder les données par DMA (Data Memory Access). En sachant mon IP et DMA sont connecté par AXI, on doit ajouter deux interconnexions de AXI dans l'implémentation dans le figure28.

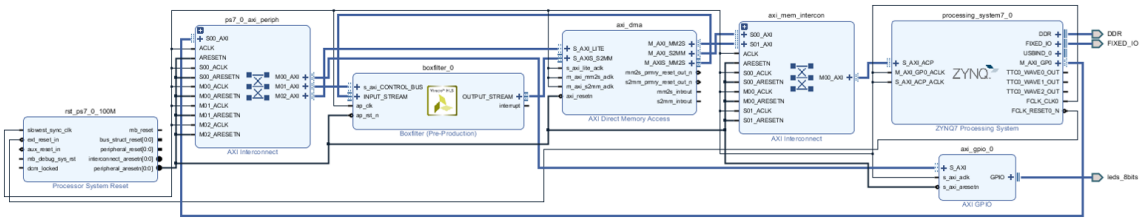

FIGURE 28 – Implémentation

5.4 Optimisation

La taille maximal de matrice est 64x64. Pour diminuer le temps de communication entre mon IP et RAM, l'entre et la sortie sont 64x64 bits donc on peut transmettre les données une seule fois. De plus, pour augmenter la vitesse de lire et écrire, nous avons ajouté le pipeline dans la lecture et l'écriture. D'ailleurs, nous avons essayé avec les options différents de compilation.

5.5 Résultats obtenus

Dans le test, nous avons testé la matrice avec les tailles différents (16x16, 32x32 et 64x64). Nous avons créé une matrice de test comme le figure29 et nous avons rempli la quatrième, la cinquième et la sixième rang avec 250 et les autres ranges restent vide. Dans mon IP, l'entre est 64x64 bits donc je peux importer 4 matrices de 32x32 ou 16 matrices de 16x16.

FIGURE 29 – Exemple de matrice

D'abord, nous avons calculé les temps d'exécution pour les trois matrices dans le figure 30. On peut voir que le temps augmenter avec l'augmentation du la taille de matrice.

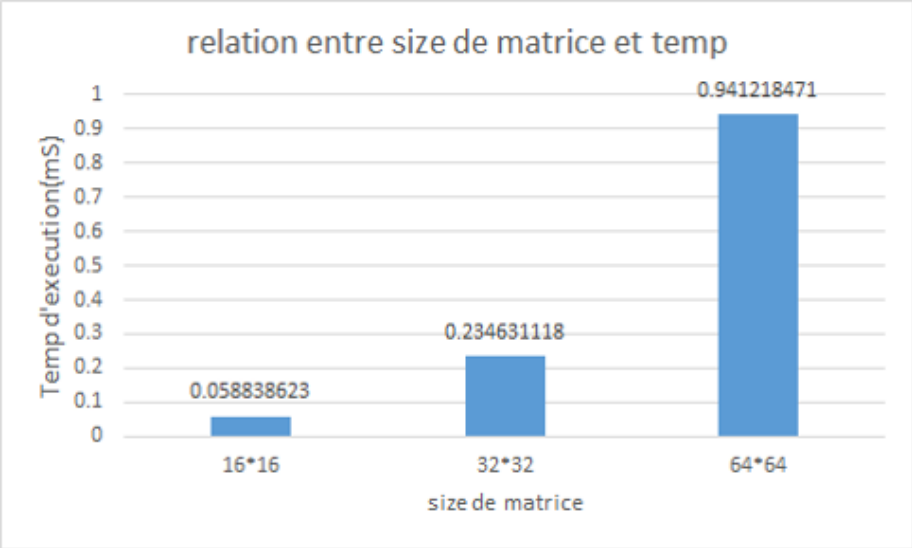

FIGURE 30 – relation entre la taille de matrice et le temps

Ensuite, nous avons comparé les optimisations différents des compilations pour ces trois matrices. Dans SDK, il y a quatre options de l'optimisation : "-O1" est d'optimiser un peu ; "-O2" est d'optimiser beaucoup ; "-O3" est d'optimiser le plus ; "-OS" est d'optimiser pour la taille. Dans le figure 31, "-O3" est le plus adapté pour la matrice 16x16.

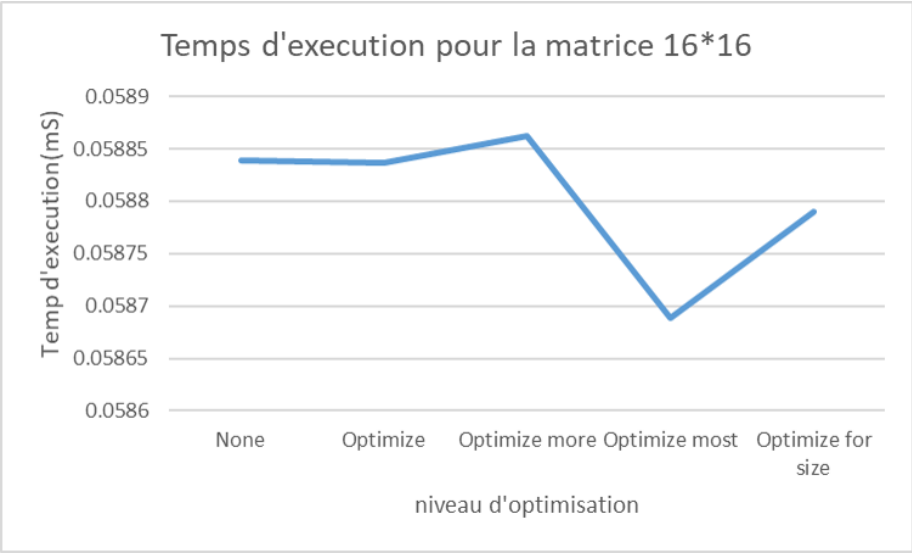

FIGURE 31 – relation entre le niveau de l'optimisation et le temps pour matrice 16x16

Dans le figure 32, "-OS" est le plus adapté pour la matrice 32x32, mais il n'y pas trop de différence entre l'option "-OS" et l'option "-O3".

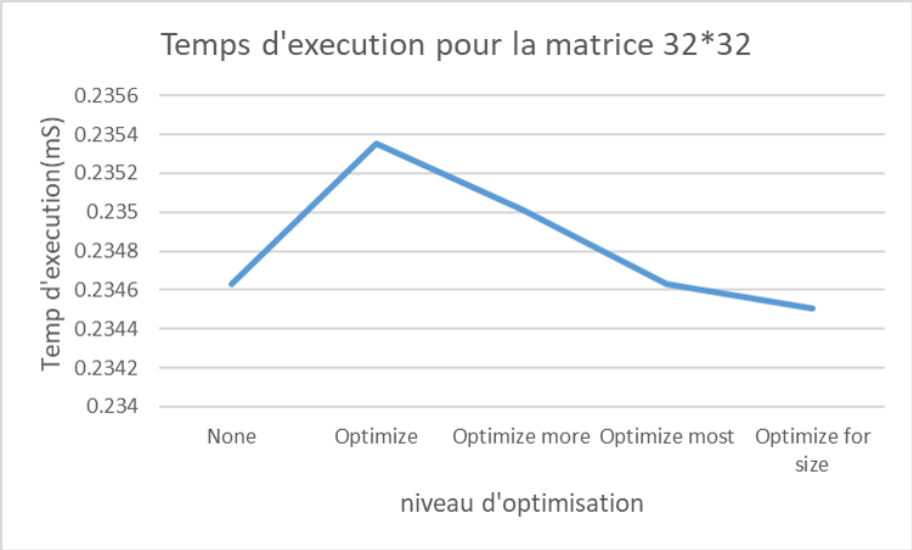

FIGURE 32 – relation entre le niveau de l'optimisation et le temps pour matrice 32x32

Dans le figure 32, c'est évident que l'option "-O3" est le plus adapté pour la matrice 64x64. Dans ce cas, on peut voir que l'option "-O3" peut améliorer la performance le plus pour les matrices avec les tailles différentes.

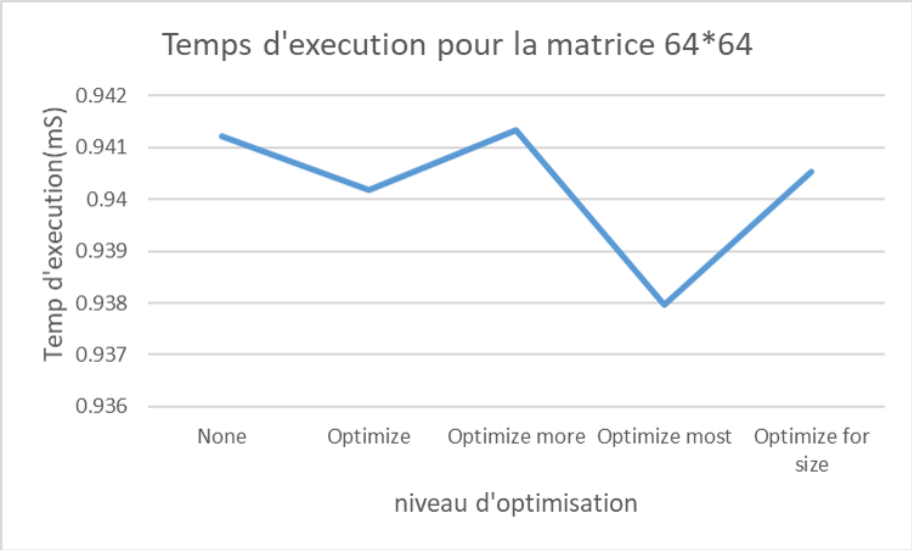

FIGURE 33 – relation entre le niveau de l'optimisation et le temps pour matrice 64x64

À la fin, nous avons comparé les résultats obtenus dans les quatre taches dans le figure 34. Dans cet image, le FPGA avec AXI est plus lente. C'est à cause de la communication entre DMA et mon IP qui prend beaucoup de temps.

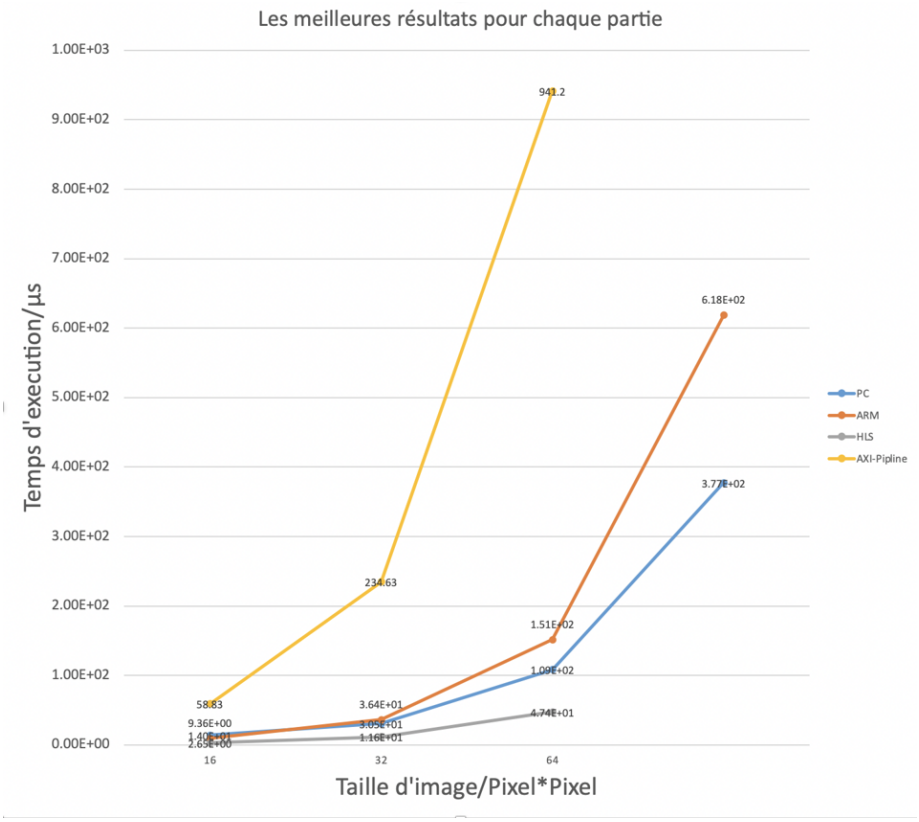

FIGURE 34 – résultats des notre tache

Références

[1] *Cache de processeur*. [EB/OL]. https://fr.wikipedia.org/wiki/Cache_de_processeur. 2020.

[2] M. A. Enderwitz L. H. Crockett R. A. Elliot et R. W. Stewart. *The Zynq Book : Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*. First Edition. Strathclyde Academic Media, 2014.

[3] PATSIATZ. *Sobel Vivado HLS Kernel using AXI Stream interface*. [EB/OL]. https://fpgaw0rld.wordpress.com/2017/05/16/sobel-vivado-hls-kernel-using-axi-stream-interface/ Accessed May 16, 2017.

[4] Wikipedia contributors. *Pareto chart*. [Online ; accessed 22-July-2004]. 2004. url : https://en.wikipedia.org/wiki/Pareto_chart.