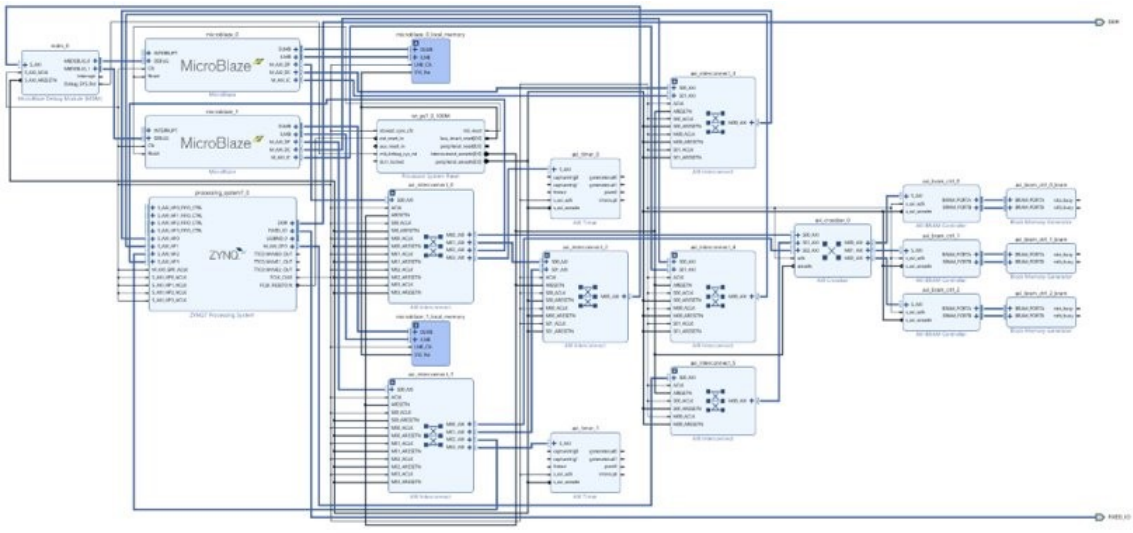
Groupe 2: Systèmes électroniques embarqués
FPGA MPSoC

ROB307: MPSoC Multiprocesseurs sur puce

Alexis OLIVEIRA DA SILVA, Bastien HUBERT,
Benjamin TIBI, Dimitri FRÉON,
Marcelo BRAGA E SILVA



ENSTA Paris - janvier 2023

1 Introduction

Ce projet vise à concevoir un système MPSoC embarqué pour une application robotique. Dans le cadre du projet, l'application donnée est le pilotage autonome d'un mobot mobile à roues, avec la plateforme robotique HUSKY fournie par l'école. Le projet a été divisé en deux groupes, le Groupe 1, qui sera responsable de l'implémentation des fonctions nécessaires au robot pour réaliser la tâche et la mission qui lui sont destinées, et le Groupe 2, responsable de l'implémentation MPSoC des fonctions proposées par le Groupe 1 dans le ZedBoard en utilisant un système multi-cœur composé du processeur ARM Cortex-A9, des processeurs synthétisés (soft processors) MicroBlaze et des accélérateurs matériels si nécessaire.

Figure 1: Robot HUSKY

Ce rapport est destinée à la documentation de la partie développé par le **Groupe 2**. Il est composé des parties suivantes:

- ÉVALUATION NETWORK ON CHIP (NOC) 3x3: tests et validation
- MULTICOEUR 1 ARM + 1 MICROBLAZE: Design et évaluation des performances
- MULTICOEUR 1 ARM + 2 MICROBLAZE: Design et évaluation des performances
- IMPLÉMENTATION DU NOC 3x3 DANS LE DESIGN
- POSSIBILITÉS D'AMÉLIORATION DU SYSTÈME

2 ÉVALUATION NETWORK ON CHIP (NOC) 3x3: tests et validation

2.1 Le Network on Chip

Un Network-On-Chip (NoC) est une architecture de communication utilisée dans les systèmes sur puce (SoC) pour connecter les différents blocs fonctionnels. Il utilise un réseau de routeurs et de canaux de communication interconnectés pour acheminer les données entre les différents blocs, ce qui permet une communication efficace et réduit la consommation d'énergie. Les NoC sont de plus en plus populaires dans les conceptions de SoC modernes car ils peuvent améliorer les performances et l'évolutivité du système, tout en réduisant la complexité de la conception. Parmi les composants clés d'un NoC figurent les routeurs, les interconnexions et les protocoles de communication. Étant donné qu'une partie importante d'une puce est composée de canaux de communication, il est essentiel de disposer d'une IP capable de réaliser efficacement la communication entre les blocs.

L'intérêt d'utiliser un NOC dans un véhicule autonome multicoeur est donc de communiquer efficacement chaque noeud de calcul, laissant plus d'espace à la Processing Logic pour être utilisé pour implémenter les fonctionnalités nécessaires au robot.

2.2 Tests

Les premiers tests avec le NOC donné seront effectués pour vérifier quelle configuration du NOC est la plus intéressante pour notre système. Il est composé de trois entrées et de trois sorties, comme le montre la figure ci-dessous. 2:

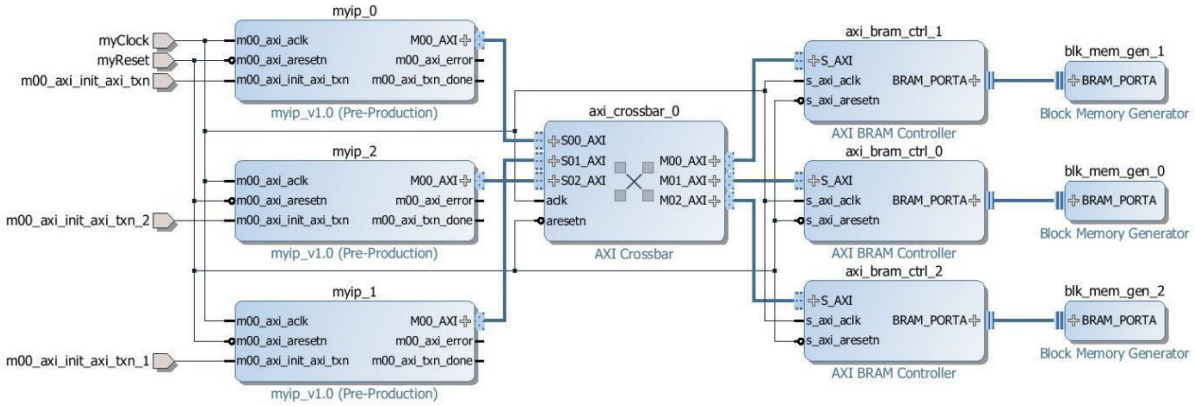


Figure 2: NOC 3x3

XXXXXXXXXXXXXXXXXXXXL'utilisation de générateurs de trafic: XXXXXXXXXXXXXXXXXXXXXXXXparler bande passante et laatence obtenus

XXXXXXXXXXXXXXXXexploration options du NOC (options du crossbar? XXXXXXXXXXXXXXXXxtype interconnexions AXI4, AXI4-Lite, and AXI3,

XXXXXXXXXXXXXXXXXXXXplacement et routage. Proposez des XXXXXXXXXXXXXXXXaméliorations.

XXXXXXXXXXXXXXXXXXXXXXXXXXArbitration options: Fixed Priority: In this scheme, masters are assigned a fixed priority level, and transactions from higher priority masters are given priority over those from lower priority masters. Round Robin: In this scheme, the crossbar cycles through all the masters in a fixed order, giving each one the opportunity to initiate a transaction in turn. Weighted Round Robin: It's similar to the round-robin scheme, but the weights assigned to each master are taken into account to determine the priority. Age-based: In this scheme, the crossbar keeps track of how long each master has been waiting for its transaction to be serviced and gives priority to the master that has been waiting the longest.

2.3 Évaluation NoC sur ZYNQ

XXXXXXXXXXXXXXXXXXXXXXXXXXxévaluation NOC sur ZedBoard

3 MULTICOEUR 1 ARM + 1 MICROBLAZE: Design et vérification

L'objectif de cette partie est de valider la conception avec 1 ARM et 1 MicroBlaze pour l'utiliser comme ligne directrice pour l'ajout ultérieur d'un MicroBlaze.

Le design Vivado du système est représenté par la Figure 3.

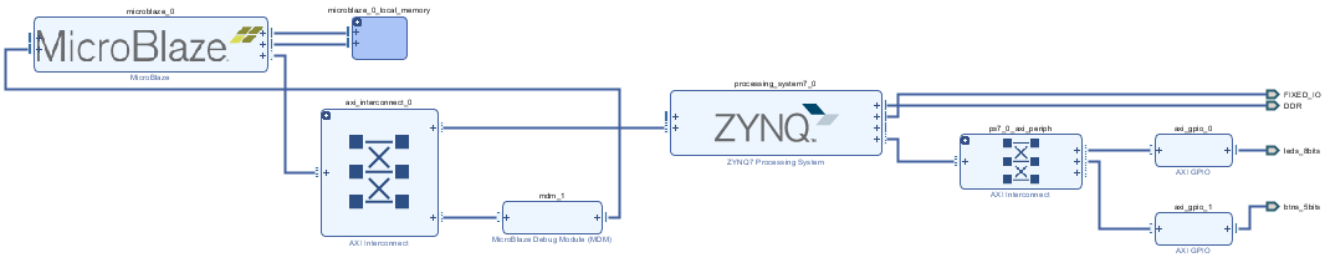


Figure 3: 1 ARM + 1 MicroBlaze

Une remarque doit être faite au design. Bien que nous puissions allouer une mémoire locale dans la PL au Microblaze, nous allons rapidement manquer d'espace disponible, en particulier pour les tâches exigeant une grande quantité de stockage de données (comme c'est le cas pour les tâches de navigation et de localisation normalement utilisées par les véhicules autonomes). Il serait donc préférable de mettre la mémoire externe DDR à la disposition du Microblaze. Toutefois, l'utilisation des BRAM de la PL étant suffisants dans notre cas, nous n'utiliserons pas la mémoire externe dans la suite de ce projet.

Pour que le MicroBlaze puisse accéder à la mémoire externe, il doit d'abord connecter sa sortie périphérique au Zynq PS via l'interconnexion AXI. Cela permet au MicroBlaze d'envoyer des transactions mémoire au Zynq PS, qui les transmet ensuite à la mémoire externe. Le Zynq PS agit donc comme un pont entre le processeur MicroBlaze et la mémoire externe, leur permettant de communiquer entre eux. En connectant la sortie périphérique du MicroBlaze au Zynq PS, celui-ci est en mesure d'accéder à la mémoire externe, et il peut également accéder à d'autres périphériques connectés au côté PS (si nécessaire plus tard).

Le design a été implementé dans la ZedBoard et des applications test en langage C testé pour vérifier le fonctionnement de deux processeurs et surtout l'accès à la DDR par le MicroBlaze.

4 MULTICOEUR 1 ARM + 2 MICROBLAZE: Design et évaluation des performances

Dans cette section, nous commencerons à évaluer les performances du système en considérant les applications et les fonctions indiquées par le Groupe 1.

4.1 Fonctions évalués

Les 4 fonctions suivantes ont été récupérées et adaptées en langage C afin que l'ARM ainsi que le MicroBlaze puissent les exécuter de la même manière.

4.1.1 Dijsktra

L'algorithme de Dijkstra est un algorithme de recherche de graphe largement utilisé qui résout le problème du plus court chemin, en produisant un arbre du plus court chemin. Cet algorithme est souvent utilisé dans le routage et comme sous-programme dans d'autres algorithmes graphiques. Dans le contexte des véhicules autonomes, l'algorithme de Dijkstra sera utilisé pour planifier le chemin le plus court et le plus sûr à emprunter par le véhicule, en tenant compte des facteurs externes.

4.1.2 K-means

K-means est un algorithme largement utilisé dans de nombreux domaines, notamment l'apprentissage automatique, la vision par ordinateur et l'exploration de données. K-means est un algorithme de clustering qui divise un ensemble de points en K clusters, où chaque cluster est représenté par son centroïde (moyenne). Il vise à minimiser la somme des carrés des distances entre les points d'un groupe et le centroïde du groupe. L'algorithme sera utilisé pour regrouper les pixels en K groupes, où chaque groupe représente un objet ou une région différente de l'image.

4.1.3 Filtre de Kalman

Le filtre de Kalman est un algorithme qui utilise une série de mesures observées dans le temps, normalement contenant du bruit, et produit des estimations de variables inconnues qui tendent à être plus précises que celles basées sur la mesure en soit. Il s'agit d'une technique puissante qui peut contribuer à améliorer la précision de l'estimation en combinant plusieurs sources d'information et en tenant compte du bruit et de l'incertitude inhérents aux mesures.

4.1.4 Conversion Image RGB en niveau de gris

La dernière fonction est une fonction de traitement d'image, représentée ici par une conversion en niveau de gris d'une image. Cette conversion est utile dans les situations où la couleur n'affecte pas l'information extraite de l'image, ce qui permet de réduire de manière significative la taille de l'image et l'effort de calcul (puisque RGB

nécessite 3 canaux de couleur). Compte tenu de la grande taille de l'image d'une caméra, la rendre plus petite réduira aussi considérablement le temps de transfert entre le processeur et la mémoire.

4.2 Premier design et évaluation

Ce premier design a été conçu pour vérifier le fonctionnement du système en ajoutant un Microblaze de plus (Figure 4).

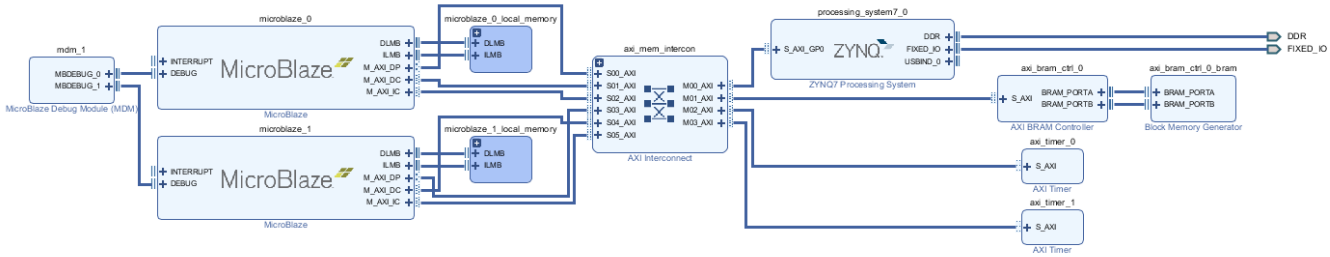


Figure 4: 1 ARM + 2 MicroBlaze (1)

Les MicroBlazes ont été configurés avec l'optimisation Minimum Area pour vérifier combien de ressources la conception minimale occuperait dans le PL et ainsi avoir de l'espace pour ajouter plus de MicroBlazes ou même des accélération matérielles pour certaines fonctions du robot. Les modifications apportées par rapport à la conception précédente sont l'utilisation des interfaces M_AXI_DC et M_AXI_IC pour les caches d'instructions et de données, et l'ajout des IP AXI_Timers pour mesurer le temps d'exécution des MicroBlazes. Par rapport au temps d'exécution sur l'ARM, un timer de la librarie "xtmrctr.h" a été utilisé pour mesurer les temps.

XXXXXXXXXXXXXXXXDonnez les informations de placement et routage. Quelle est la fréquence d'horloge du multicoeur XXXXXXXXXXXXXXXXXXXXAméliorations de fréquence d'horloge (mettre tout en meilleure perfoamnce?? optimiser MEILLEURE PERFORMANCE, mettre/retirer FPU, Branch Target Cache, IC, DC)

XXXXXXXXXXXXXXXXXXXXXXXXXXXXsynthèse placement routage du multicoeur avec les différentes configurations des 2 Microblaze. Donnez les informations de placement et routage et fréquences d'horloge.

Les fonctions de cette première design ont été exécutées en parallèle, une dans chaque processeur, mais indépendamment (les fonctions ne peuvent pas communiquer entre elles). Leur temps d'exécution dans chaque processeur a été enregistré comme indiqué dans le tableau **??**.

Fonction	Temp d'exécution sur ARM (us)	Temp d'exécution sur MB (us)
Dijkstra	4	25
K-means	1185	313571
Filtre Kalman	11	7763
RGB2GRAY	10	4841

Table 1: Temps d'exécution premier design

Le tableau **??** nous montre déjà que la fonction K-means est la plus coûteuse en temps d'exécution, et sera

probablement la fonction qui sera exécutée sur un des deux processeurs de l'ARM. D'autre part, en considérant les applications reçues, la taille des données d'entrée de chaque fonction est essentielle pour savoir comment bien évaluer le temps d'exécution de chaque fonction. La fonction de Dijkstra par exemple, dont la complexité est normalement de O(n2), s'exécute extrêmement vite car les graphes utilisés ne sont pas assez grands. Tenant compte de cela, on considéra que cela seront les données d'entrée utilisées car ils nous ont été passé de cette forme.

4.3 Deuxième design

Ce deuxième design a été conçe pour séparer le AXI interconnect des deux MicroBlazes en deux canaux différents et aussi l'utilisation d'un AXITimer par chaque MicroBlaze, de sorte que la mesure des temps soit exécutée en parallèle dans les deux cœurs. Le dernier point est l'utilisation du PS pour donner aux MicroBlazes l'accès à la mémoire externe comme mentionné dans la version avec un MicroBlaze, en utilisant les bus de communication AXI_Interconnect (Figure 5).

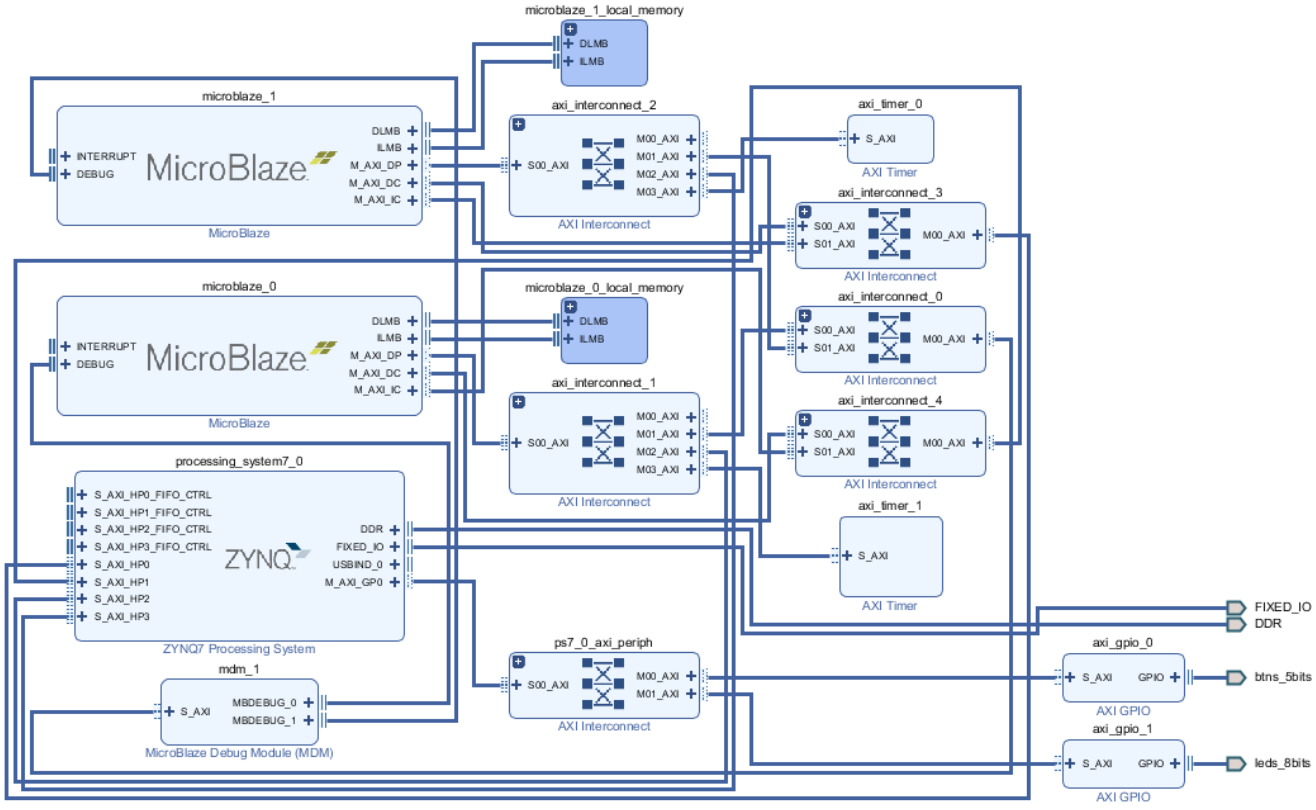


Figure 5: 1 ARM + 2 MicroBlaze (2)

4.4 Troisième Design

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXcensé à être le design d'avant plus accelerateurs materielles (just parler et montrer le squellete du blok design)

5 IMPLÉMENTATION DU NOC 3x3 DANS LE DESIGN

Ce design a été conçu pour intégrer au système multicoeur le NOC proposé dans la section 2 de ce rapport (Figure 6).

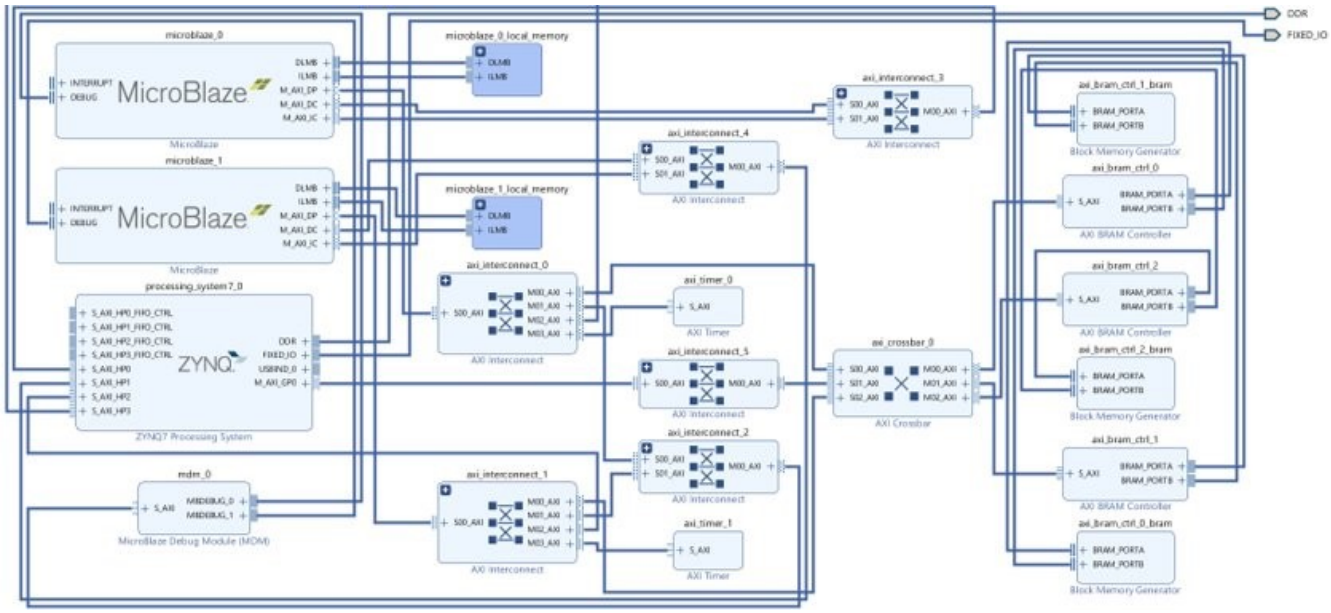


Figure 6: 1 ARM + 2 MicroBlaze + NOC

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

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

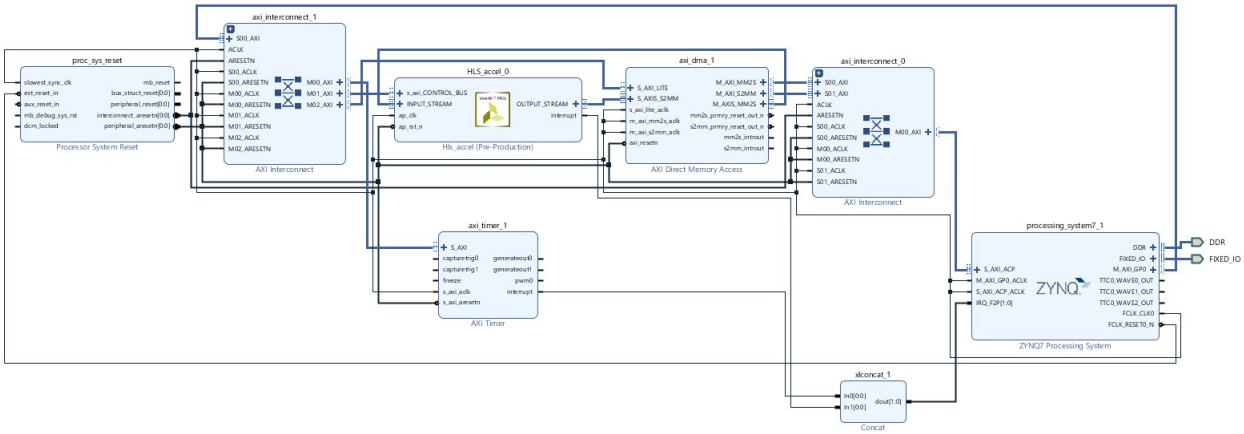


Figure 7: Bloc Design avec interface en AX14-STREAM

Afin d'accélérer l'exécution de la fonction de multiplication matricielle, nous allons à présent concevoir un accélérateur matériel qui réalisera la travail de la fonction C++ `matrix_mult` décrite dans les sections précédentes.

Dans cette partie, nous utilisons cette fonction comme une boîte noire, de sorte que n'importe laquelle des versions et optimisations exposées dans les sections 2 et 3 peut être réutilisée.

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

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

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

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

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

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

  int i, j, k;

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

  int B_SIZE = IN_B_ROWS * IN_B_COLS;
  for( i = 0; i < IN_B_ROWS; i++ ) {
    for( j = 0; j < IN_B_COLS; j++ ) {
      #pragma HLS PIPELINE II=1
      k = i * IN_B_COLS + j + A_SIZE;
      b[i][j] = pop_stream<int, 4, 5, 5>( INPUT_STREAM[k] );
    }
  }

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

  // Stream in the 2 input matrices
```

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

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

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

- augmentation du niveau d'abstraction avec la définition des variables `mat_a`, `mat_b` et `mat_prod`
- lecture de `mat_a` et `mat_b` grâce à la fonction `pop_stream`, dont nous détaillerons le fonctionnement en sous-section 5.3
- exécution du produit matriciel à proprement parler
- écriture de `mat_prod` grâce à la fonction `push_stream`, dont le fonctionnement sera également décrit en sous-section 5.3

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

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

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

  matrix_mult_wrapper( INPUT_STREAM, OUTPUT_STREAM );
}
```

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

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

5.2 Optimisations de calcul

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

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

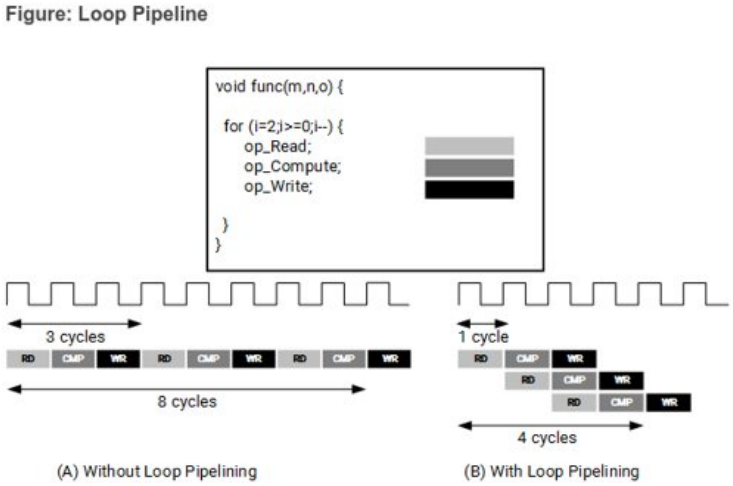


Figure 8: Schéma explicatif du fonctionnement du pipeline

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

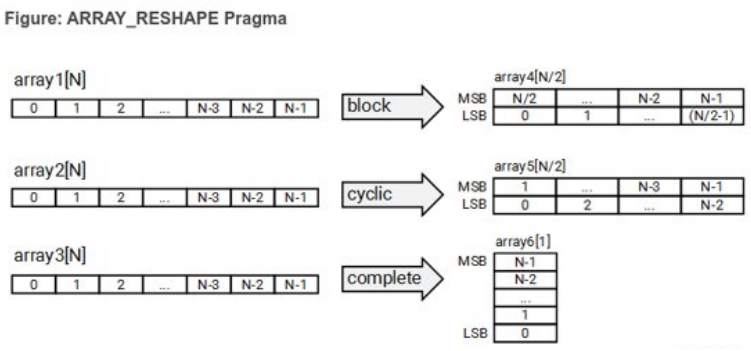


Figure 9: Schéma explicatif du fonctionnement de l'array reshape

5.3 Optimisations de transmission

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

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

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

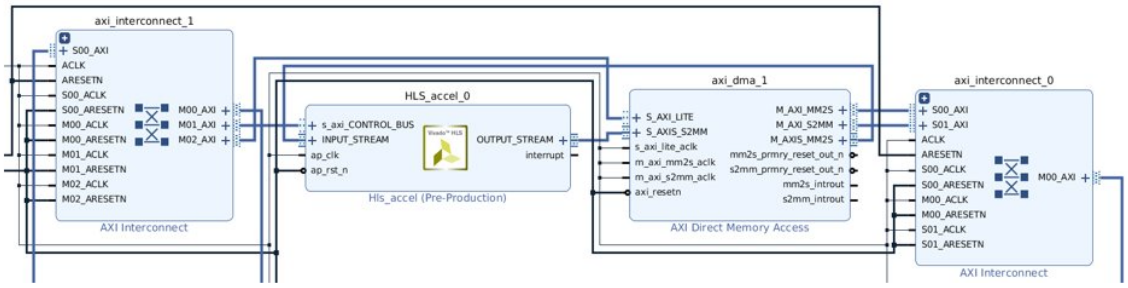


Figure 10: Partie du design Vivado incluant l'accélérateur matériel et les blocs de communication

5.4 Choix des paramètres et des métriques

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

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

Ces solutions ont toutes été testées sur différentes données :

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

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

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

5.5 Résultats et analyse

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

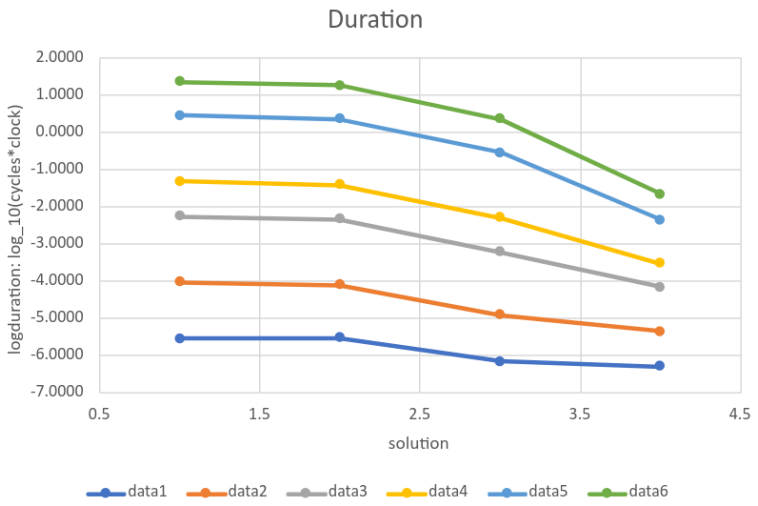


Figure 11: $T(c)$ pour différentes configurations

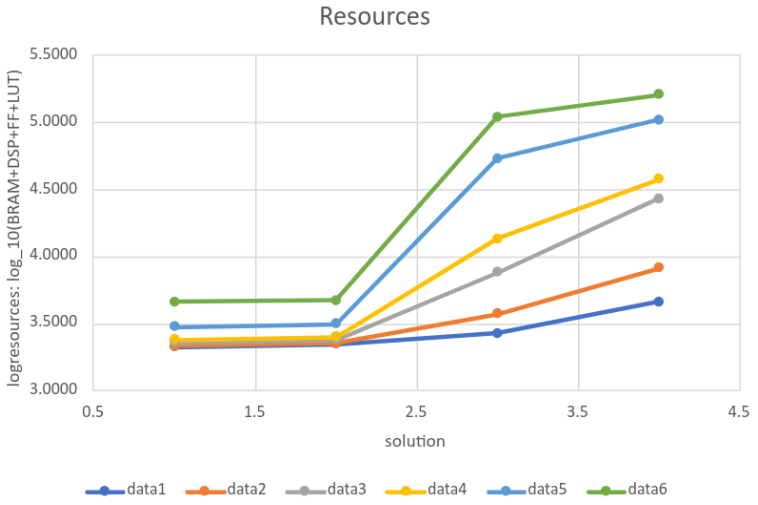


Figure 12: $R(c)$ pour différentes configurations

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

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

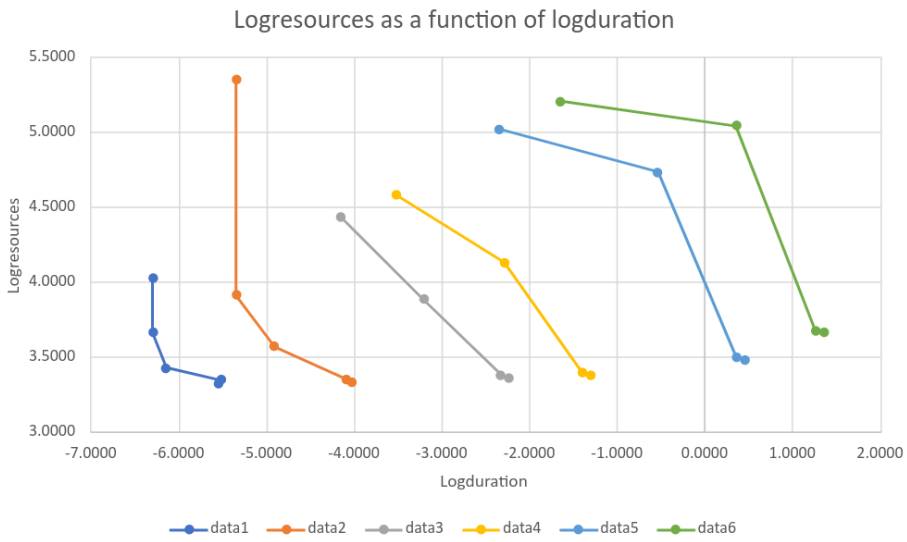


Figure 13: $R(c)$ en fonction de $T(c)$

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

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

5.6 Intégration SDK de l'IP :

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

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

5.7 Résultat :

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

D'après le calcul des performances, on remarque une diminution du temps d'exécution. L'IP HLS a un effet plus significatif avec les matrices de grande taille et moins remarquable avec les matrices de petite taille et c'est pour cela que dans cette étape on n'a pas travaillé avec les matrices de taille 16.

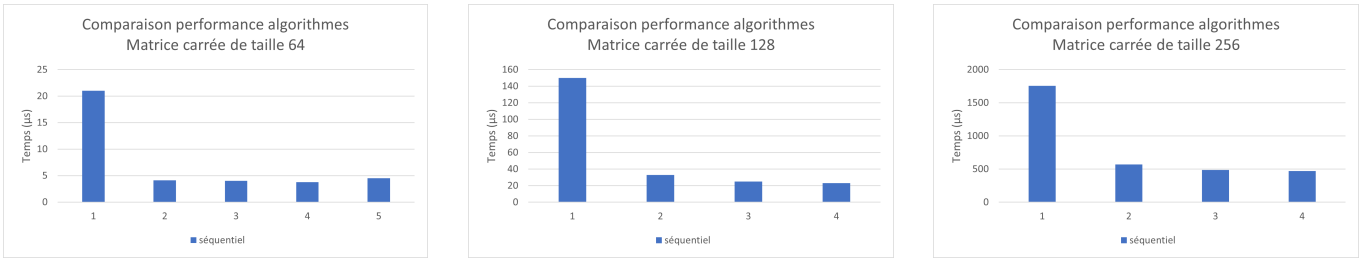


Figure 14: Résultat des performances après l'intégration de l'accélérateur

5.8 Comparaison performance ARM9 et FPGA- HLS

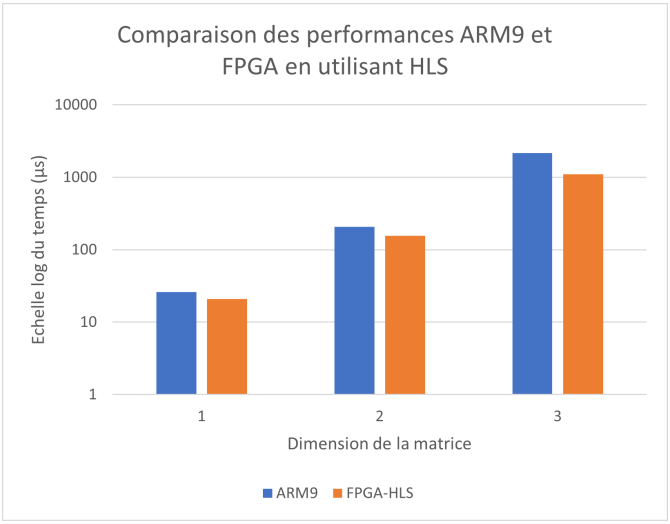


Figure 15: Comparaison des performances ARM9 et FPGA-HLS

Pour les dimensions des matrices :

* 1 : représente la taille 64

* 2 : représente la taille 128

* 3 : représente la taille 256

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

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

Pour la réduction de consommation des ressources, il est recommandé d'ajouter des niveaux Pipeline à la logique et de minimiser les signaux de contrôle asynchrone.

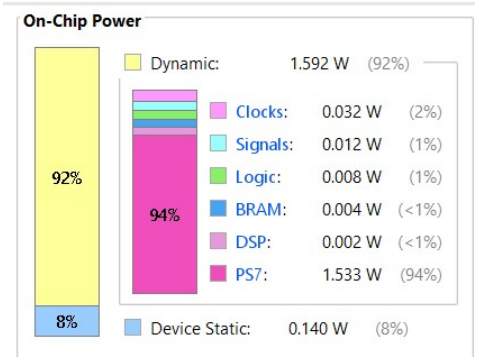


Figure 16: rapport de l'énergie pour une matrice de dimension 64 et solution 1

5.9 Conclusion :

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

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

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

Figure 17: Tableau comparatif final