ROB306 - Accélérateur Matériel avec HLS

Madeleine Becker, Matheus Monteverde, Mateus Ricci, Fabricio Vellone

INTRODUCTION

a) : Dans ce projet, on se propose à étudier et évaluer les performances en temps d'exécution et en ressources d'une fonction de multiplication de matrices à différents tailles. On l'étudiera dans 3 configurations suivants : 1. sur processeur généraliste, comme ceux disponible dans un PC ; 2. sur processeur embarqué ARM9 ; 3. sur un circuit reconfigurable FPGA XC7Z020, disponible sur carte zedboard.Dans la Figure 0.1 on peut regarder la carte utilisé :

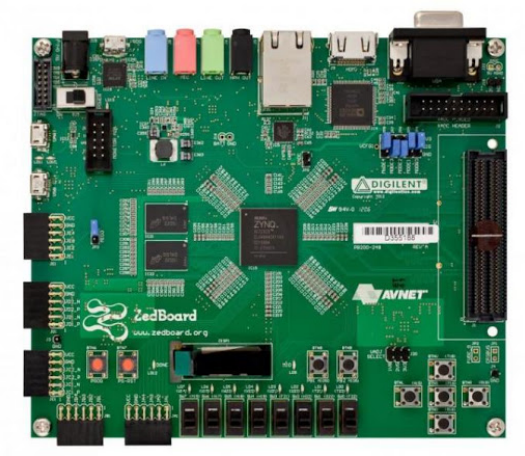


Figure 0.1: Carte ZedBoard

Le rapport est divisé en 4 sections : dans la section 1, "Évaluation de performances sur PC", nous décrivons la méthodologie suivie pour l'optimisation de la multiplication de matrices pour un processeur généraliste, en abordant des différents méthodes de compilation et stratégies d'algorithmes. Dans la section 2, "Évaluation de performances sur processeur embarqué ARM9", on optimisera la multiplication de matrices pour le processeur ARM9, en exploitant l'environnement VIVADO 2019 SDK et on étudiera les différents possibilités d'optimisation. La section 3 est destinée à l'estimation de performances et de ressources sur circuit FPGA, de cette façon, on essaie d'optimiser le circuit avec l'utilisation des directives dans la synthèse HLS en évaluant les temps de latence, les débits et ressources matérielles utilisées. La section 4 est destinée aux mesures de performances dans le circuit FPGA. Finalement, on présente une section de conclusion avec un réusmé de ce qu'on a obtenu comme résultats dans ce projet.

1. EVALUATION DE PERFORMANCES SUR PC

Cette partie vise à évaluer les performances de différentes configurations et méthodes sur une même fonction sur le PC.

A. *Description PC*

Tous les résultats concernant le temps d'exécution du CPU qui seront présentés dans cette section ont été obtenus en utilisant le PC qui présente les caractéristiques suivantes :

CPU Name	Intel(R) Core(TM) i7-7500U
Frequency	2.70GHz 2.90 GHz
Cache L1	128 KB
Cache L2	512 KB
Cache L3	4 MB
Memory	8 GB
OS	Windows
Compiler	g++

B. *Optimisation*

Afin d'optimiser le temps d'exécution du CPU, plusieurs configurations différentes ont été testées, variant non seulement la taille des matrices multipliées, mais aussi les flags d'otmisation et les méthodes de multiplication. Il est important de souligner que toutes les matrices multipliées sont des carrés de même taille.

Tous les paramètres utilisés peuvent être consultés ci-dessous :

Taille des matrices :

— 64x64*
— 128x128
— 256x256
— 512x512
— 1024x1024
— 2048x2048

Flags d'otmisation :

— Aucune
— -O
— -O1
— -O2
— -O3
— -Os
— -Ofast
— -Og

Méthodes de multiplication :

— Méthode séquentielle
— Méthode parallèle
— Méthode de Strassen
— Méthode adaptée à la mémoire cache
— Méthode combinée

* Pour les multiplications de matrices de taille 64x64, le temps d'exécution était si petit que le résultat était indiqué comme étant 0. Ainsi, pour ce cas, un temps moyen a été fait de 5 essais de la même configuration. Le résultat peut avoir été affecté par la mise en cache du processeur.

1) Méthode séquentielle: Tout d'abord, le code de la fonction de base de la multiplication par la méthode séquentielle a été exécuté. Dans cette méthode, la matrice résultante est obtenue par la formule suivante :

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

À partir de cette fonction, l'accélération du temps de fonctionnement du CPU sera analysée. Le temps d'exécution a été tracé sans aucune flag d'optimisation et en faisant varier la taille de la matrice. Les résultats obtenus sont présentés ci-dessous.

Ensuite, la même fonction a été compilée en utilisant différents types de flag d'optimisation comme indiqué sur la figure 1.2.

Ces résultats permettent déjà de constater une bonne accélération du temps d'exécution par rapport à la première configuration.

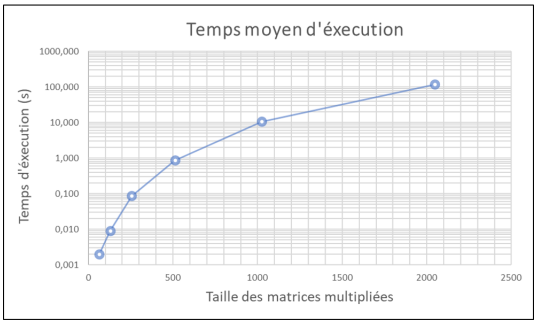


Figure 1.1: Temps d'éxecution pour la méthode séquentielle non optimisée

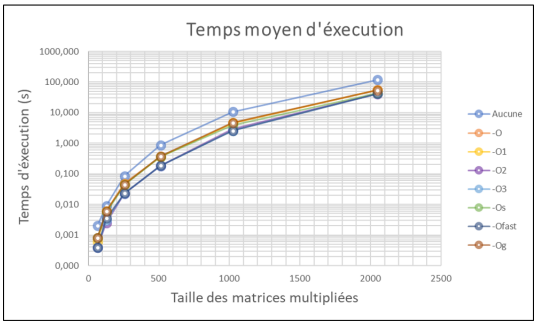


Figure 1.2: Temps d'éxecution pour la méthode séquentielle

2) Méthode parallèle: Il était alors souhaité d'inclure à la méthode précédante la parallélisation. Pour ce faire, nous avons utilisé l'extension de compilation OpenMP. L'OpenMP permet d'exploiter le code en parallèle, qui utilise les cœurs disponibles dans une machine multi-cœurs. Ainsi, le partage de la multiplication entre les threads est tel que différents threads calculeront différentes lignes de la matrice de résultats. Les accès aux matrices d'entrée sont en lecture et n'introduisent pas de problèmes.

Différents types de flags d'optimisation ont également été utilisés pour cette méthode. On a donc trouvé la meilleure configuration pour cette méthode avec la compilation au format "-O3". On peut le voir ci-dessous, par rapport à la méthode précédente.

3) Méthode de Strassen: Une autre méthode utilisée pour effectuer la multiplication des matrices était la méthode Strassen. L'algorithme de Strassen est une méthode récursive de multiplication de matrice où l'on divise les matrices en 4 sous-matrices de dimensions n/2 x n/2 à chaque étape récursive, comme indiqué

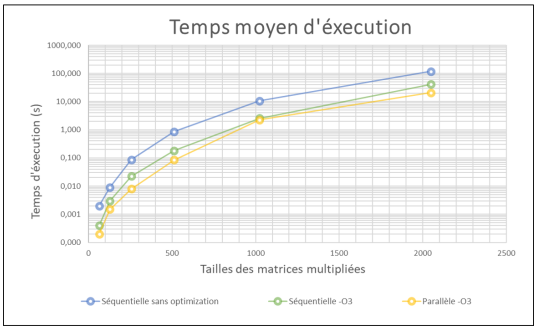


Figure 1.3: Temps d'éxecution pour la méthode parallèle

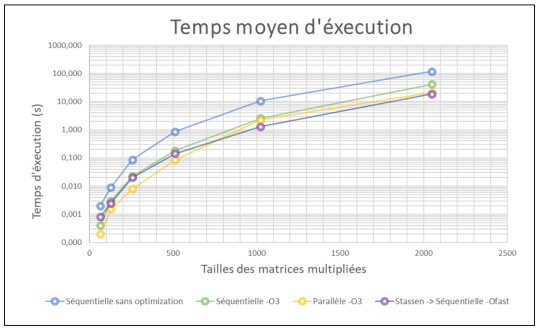


Figure 1.4: Temps d'éxecution pour la méthode de Strassen

ci-dessous.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \quad (2)$$

Ensuite, la matrice C, résultant de la multiplication des matrices A et B, est obtenue par les équations suivantes :

$$\begin{cases} P_1 = A_{11} \cdot (B_{12} - B_{22}) \\ P_2 = (A_{11} + A_{12}) \cdot B_{22}) \\ P_3 = (A_{21} + A_{22}) \cdot B_{11}) \\ P_4 = A_{22} \cdot (B_{21} - B_{11}) \\ P_5 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\ P_6 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \\ P_7 = (A_{11} + A_{21}) \cdot (B_{11} + B_{12}) \end{cases}$$

La complexité dans le temps de l'algorithme séquentiel est O(n^3) alors que celle de la méthode Strassen n'est que O($n^{2,81}$). Par contre, en appliquant cette méthode dans le code, il a été possible de constater qu'en divisant les matrices d'entrée en sous-matrices plusiers fois, le temps passé à allouer la mémoire de ces matrices auxiliaires était plus important que le gain de temps dû au nombre plus faible d'opérations arithmétiques. Ainsi, afin de pouvoir accélérer le processus, les matrices ont été divisées une seule fois, puis la multiplication des sous-matrices a été effectuée par la méthode séquentielle.

Le temps d'exécution de la meilleure configuration pour cette méthode a été tracé (figurr 1.4) pour être comparé avec les précédentes.

4) Méthode adaptée à la mémoire cache: Ensuite, une méthode plus adaptée à la mémoire de Cache a été proposée. Dans cette méthode, avant d'effectuer la multiplication entre les matrices en séquentiel, les colonnes de la matrice B sont stockée dans un vecteur. Cela permet de réduire le nombre de manques dans la mémoire Cache. Pour cette méthode également, la meilleure configuration a été tracée dans le graphique du temps d'exécution du CPU (figure 1.5).

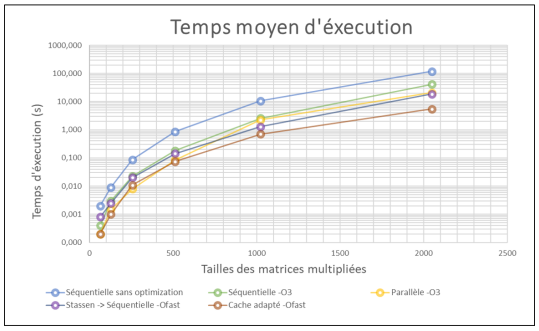


Figure 1.5: Temps d'éxecution pour la méthode Cache adapté

5) Méthode combinée: Enfin, une dernière méthode a été élaborée, combinant les trois méthodes précédentes. Les matrices sont d'abord divisées une fois en sous-matrices, puis ces sous-matrices sont multipliées par la méthode la plus adaptée à la mémoire cache. Dans cette dernière multiplication, la méthode parallèle est également utilisée.

Cette méthode a été la plus efficace et la comparaison avec les méthodes précédentes e est présentée sur la figure 1.6.

C. Analyse des résultats

Grâce aux résultats de toutes les configurations, il a été possible de tracer les temps d'exécution pour chaque taille de matrice différente. Ces graphiques sont visibles dans les figures suivantes.

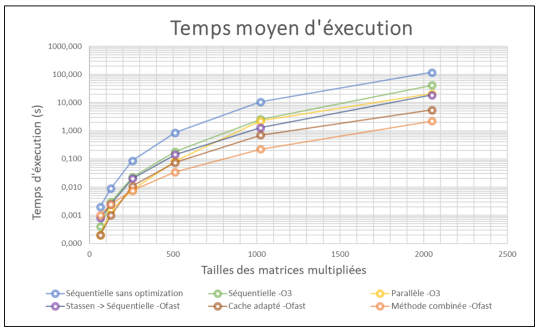
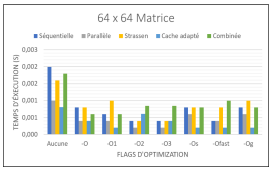
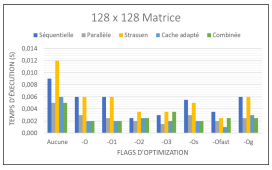


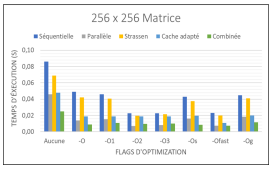
Figure 1.6: Temps d'éxecution pour la méthode combinée



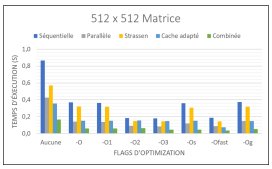
(a) Matrice de taille 64 x 64



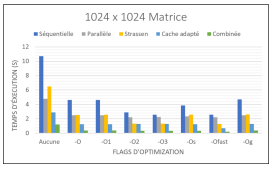
(b) Matrice de taille 128 x 128



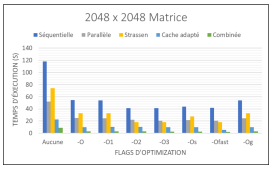
(c) Matrice de taille 256 x 256



(d) Matrice de taille 512 x 512



(e) Matrice de taille 1024 x 1024



(f) Matrice de taille 2048 x 2048

Figure 1.7: Temps d'exécution pour plusieurs configurations

De cette façon, il a été possible de trouver la meilleure configuration pour chaque taille de matrice différente et quelle est la valeur de cette accélération. Les meilleurs résultats sont présentés dans le tableau ci-dessous.

Taille de la matrice	Temps d'éxecution pour méthode séquentielle sans optimisation	Meilleure configuration	Temps d'éxecution pour la meilleure configuration	Accélération
64	0,0020	Cache adapté	0,0002	**10,02**
128	0,0090	Cache adapté	0,0010	**9,00**
256	0,0860	Combinée -Ofast	0,0075	**11,47**
512	0,8660	Combinée -Ofast	0,0345	**25,10**
1024	10,7030	Combinée -Ofast	0,2230	**48,00**
2048	118,3890	Combinée -Ofast	2,2100	**53,57**

Ces résultats montrent qu'il y a eu un grand gain de performance en termes de temps d'exécution sur la fonction choisie. Pour les matrices de petite taille, le gain était environ 9 fois plus rapide et pour les matrices de grande taille, était environ 53 fois plus rapide.

D. Conclusion

Par l'évaluation de la performance dans le temps d'exécution d'une fonction de multiplication des matrices en PC, il a été possible de voir l'influence de plusieurs paramètres dans le temps final obtenu. Ces paramètres étaient les différentes types de compilations, les tailles des matrices et les méthodes de multiplication.

L'étude des différentes configurations proposées a permis de constater qu'il n'y avait pas de configuration meilleure pour tous les cas. Pour les plus petites matrices choisies, la méthode du cache adapté était celle qui présentait de meilleures performances car elle ne nécessitait pas beaucoup d'allocation dynamique de mémoire et permettait d'accéder plus rapidement aux données souhaitées en mémoire. Cependant, pour les grandes matrices, la méthode combinée est plus adaptée car elle permet de diviser la matrice en sous-matrices de taille réduite et d'appliquer les méthodes les plus appropriées à ces petites matrices.

2. Évaluation de performances sur processeur embarqué ARM9

L'objet de cette partie est d'étudier les performances du processeur ARM9 de la carte Zedboard dans différentes configurations.

A. Résultats initiaux

Pour commencer cette étude, nous avons déterminé les performances par défaut du processeur. Les paramètres par défaut du processeur sont :

— algorithme naïf,

— compilation non optimisée,

— accès aux caches.

Dans ces conditions, nous avons mesuré les temps d'exécution de la multiplication de deux matrices carrées, en faisant varier la tailles des dites matrices de 2 à 2048. Les

résultats sont visibles sur la figure 2.1. On observe que les temps croissent exponentiellement, avec deux coefficients d'exponentielle principaux : un très grand pour de petites tailles de matrices, et un coefficient plus petit pour de plus grandes tailles de matrices. La transition se fait pour des matrices de taille environ 200. Le temps d'exécution maximal observé est d'environ $1,11 \cdot 10^9$ microsecondes pour des matrices de taille 2048.

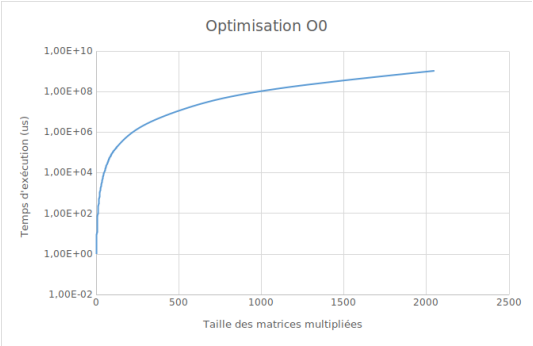


Figure 2.1: Résultats initiaux

B. *Options de compilation gcc*

Le premier paramètre que l'on a fait varier est l'option de compilation du compilateur gcc. Elle indique au compilateur le degré d'optimisation souhaité.

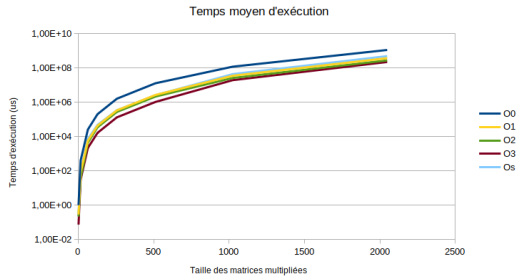


Figure 2.2: Options de compilation gcc

Les résultats sont visibles sur la figure 2.2. On observe que ce paramètre a une forte influence, puisque le temps d'exécution est 10 fois moindre entre la pire optimisation `-O0` et la meilleure `-O3`. Normalement, l'option de compilation `-Os` est censée être la meilleure, puisqu'elle devrait être adaptée à l'architecture. Cependant ça n'est pas le cas ici, donc on peut supposer que l'architecture décrite pour cette option ne correspond pas à l'architecture réelle. Le meilleur temps d'exécution pour des matrices de taille 2048 est $2,22 \cdot 10^8$ microsecondes.

C. *Utilisation du cache*

Le paramètre suivant que l'on a fait varier est l'accès au cache. Le processeur Zynq dispose de deux types de cache, le cache données et le cache instructions.

Pour ces mesures, la taille maximale de matrice que l'on a utilisée est 512 et non 2048 comme précédemment, parce que les temps d'exécution sont déjà très long pour cette taille de matrice.

1) Cache instructions: Tout d'abord, nous avons mesuré les temps d'exécution en privant le processeur de son cache instructions uniquement.

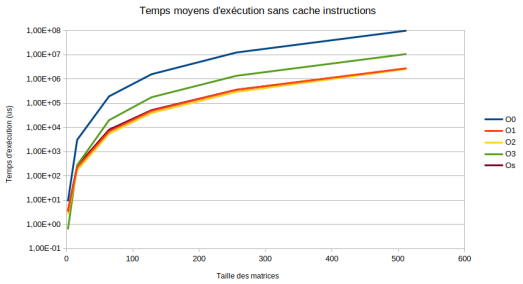


Figure 2.3: Sans cache instructions

Les résultats sont présentés sur la figure 2.3. Nous observons que les différentes options de compilation ont un impact sur les résultats, donc l'optimisation de la compilation se fait en optimisant le cache données. La meilleure option de compilation est toujours `-O3`, mais les temps sont environ 10 fois supérieurs aux temps avec cache instructions. Le meilleur temps pour une matrice de taille 512 est de $1,06 \cdot 10^7$ microsecondes, contre $1,04 \cdot 10^6$ microsecondes avec le cache instructions pour la même taille de matrices.

2) Cache données: Ensuite, nous avons mesuré les performances du processeur lorsqu'il est privé du cache données.

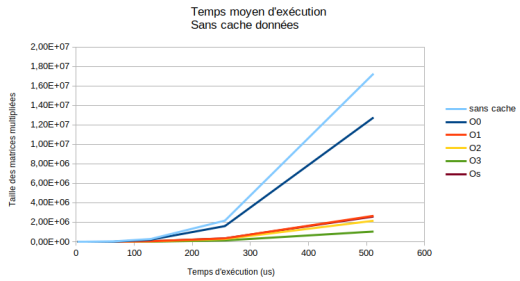


Figure 2.4: Sans cache données

Les résultats sont visibles sur la figure 2.4. Nous avons observé que les options de compilation n'avaient cette fois quasiment aucun effet. Nous en déduisons donc que les optimisation de la compilation se font essentiellement sur le cache de données. C'est assez logique puisque surtout dans une application comme la multiplication de deux matrices, il y a beaucoup plus de données que d'instructions à chercher en mémoire. Sur la figure, nous avons ajouté les courbes des différentes options de compilation avec le cache, pour faciliter la comparaison. Ce qui ressort, c'est que les performances du processeur sans cache instructions sont légèrement moins bonnes que la pire option de compilation. Pour des matrices de taille 512, le temps moyen d'exécution est de $1,73 \cdot 10^7$ microsecondes, contre $1,04 \cdot 10^6$ microsecondes avec le cache données pour la même taille de matrices.

3) Sans cache données ni cache instructions: Ensuite, nous avons mesuré les performances du processeur Zynq lorsqu'il est privé de ses deux caches.

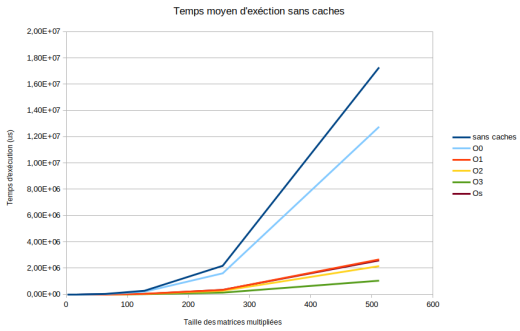


Figure 2.5: Sans caches données et instructions

Les résultats sont visibles sur la figure 2.5. De même ici, les options de compilations n'ont que peu d'influence, ce qui est cohérent avec les observations faites dans la partie précédente. Comme précédemment, nous avons également ajouté sur le graphique les résultats avec caches pour les différentes options de compilation, afin de faciliter la comparaison. Là encore, les résultats sont moins bons, ce qui est attendu. Pour des matrices de taille 512, le temps moyen d'exécution est de $1,73 \cdot 10^7$ microsecondes, contre $1,04 \cdot 10^6$ microsecondes avec le cache données pour la même taille de matrices. C'est le même temps que sans cache données, donc le cache instructions n'a que peu d'influence pour cette application.

D. Algorithme non naïf

Le dernier paramètre que nous avons étudié est un changement d'algorithme. Pour cela, nous avons utilisé un code développé en partie 1. Nous avons utilisé l'algorithme *Cache adapté*, puisque c'était le code le plus optimisé qui n'utilise pas de libraire particulière. En effet, puisque nous travaillons sur le processeur en *standalone*, nous avons accès qu'aux fonctions disponibles sur le Zynq.

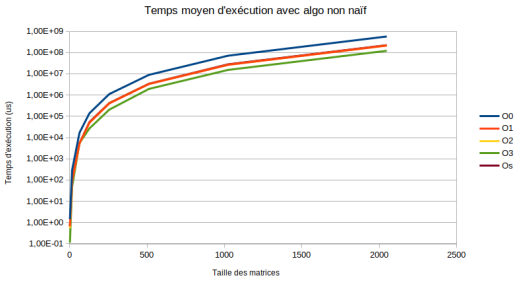


Figure 2.6: Algorithme non naïf

Les résultats sont visibles en figure 2.6. Ici les options de compilation ont bien été appliquées à l'algorithme naïf. On observe que par rapport à l'algorithme naïf, les temps d'exécution sont globalement divisés par 2, ce qui est bien mais moins bien que les autres paramètres. La meilleure option de compilation est toujours `-O3`, et le temps d'exécution pour des matrices de taille 2048 est de $1,21 \cdot 10^8$ microsecondes contre $2,22 \cdot 10^8$ microsecondes pour l'algorithme naïf.

E. Conclusions

Nous avons exploré les effets sur le processeur de l'optimisation de la compilation, des caches, et de l'algorithme. Cependant, d'autres paramètres n'ont pas été explorés. Par manque de temps, nous n'avons pas exploré l'effet de la variation de fréquence du processeur sur les performances. Nous aurions également pu tenter de paralléliser l'exécution puisque le processeur dispose de deux cœurs. Le problème est que cela aurait nécessité de programmer manuellement la synchronisation des cœurs pour l'accès à la mémoire, l'accès au cache, l'accès au port UART etc. Nous n'avions ni le temps ni les connaissances pour nous lancer dans ce chantier, donc nous avons choisi de ne pas le faire.

Le meilleur temps obtenu ici pour une matrice de taille 2048 est $1,21 \cdot 10^8$ microsecondes sur le processeur utilisant tous ses caches, avec un algorithme non naïf, et l'option de compilation `-O3`.

3. Estimation de performances et de ressources par accélérateur matériel sur circuit FPGA

Dans ce partie, on se propose a concevoir une accélérateur depuis un code C de multiplication de matrices. Pour y arriver on utilise le logiciel Vivado 2019 HLS pour generer une synthèse de ce circuit.

Premièrement, on a développé un code en C pour traiter le calcul et test de la multiplication des matrices. Ensuite, dans l'environnement VIVADO HLS, on a applique différents directives et on a proposé quelques solutions évalues par rapport au temps d'exécution et ressources utilisés. Les tests sont fait à plusieurs tailles de matrices.

Finalement, on exporte le code RTL pour l'exécuter dans le circuit FPGA, comme sera montré dans la section 4.

Le tableau ci-dessous résume les principaux solutions analysés, et les directives appliqués au HLS pour obtenir le code VHDL.

Solution 0	Solution 1	Solution 2	Solution 3	Solution 4	Solution 5	Solution 6
-	Partition in half	Partition complete	Reshape complete	Reshape in half	-	Partition in quarter
-	Pipeline	Pipeline	Pipeline	Pipeline	Pipeline	Pipeline

Figure 3.1: Tableau Stratégies de Solutions

La solution 0 consiste à la solution originale ;

La solution 1 consiste a l'utilisation de la directive "partition" par la moitie de la taille de la matrice dans le fonction de la multiplication de matrices et l'application de pipeline dans les boucles ;

La solution 2 correspond a l'utilisation d'une directive en partition complète dans le fonction de la multiplication de matrices et l'application de pipeline dans le boucles ;

La solution 3 correspond a l'application de la directive "reshape" complète dans le fonction de la multiplication de matrices et l'application de pipeline dans le boucles ;

La solution 4 correspond correspond a l'application de la directive "partition" par la moitie de la taille de la matrice dans le fonction de la multiplication de matrices et l'application de pipeline dans le boucles ;

La solution 5 correspond à seulement l'application de pipeline dans le boucles. La solution 6 consiste a l'utilisation de la directive "partition" par le quart de la taille de la matrice dans le fonction de la multiplication de matrices et l'application de pipeline dans les boucles ;

A. Multiplication Matrices 32x32

Les ressources utilisés pour la multiplication de matrices de taille 32x32 sont montrés dans le tableau ci-dessous :

32x32 Solution	BRAM_18K (%)	DSP48E (%)	FF (%)	LUT (%)	URAM (%)
0	2	1	0	2	0
1	24	40	4	11	0
2	24	40	4	13	0
3	22	40	4	57	0
4	22	40	4	49	0
5	2	2	1	7	0
6	12	20	3	10	0

Figure 3.2: Ressources utilisés pour la multiplication de matrices de taille 32

Les résultats pour les temps de exécution et pour le throughput sont montrés dans les figures 3.3 et 3.4.

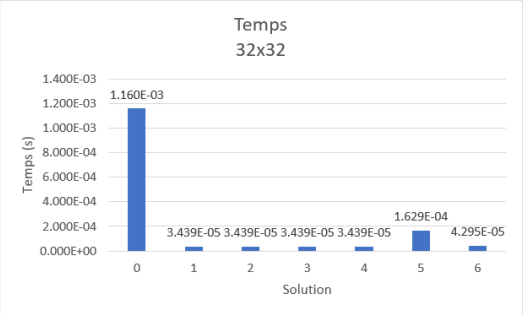


Figure 3.3: Latency pour la multiplication de matrices de taille 32

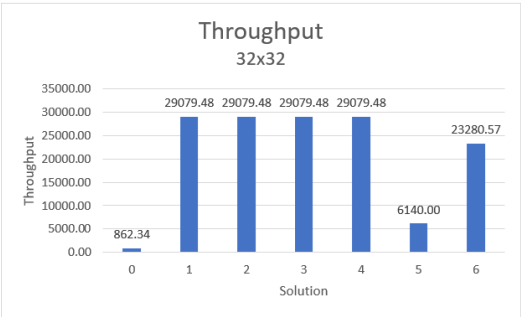


Figure 3.4: Throughput pour la multiplication de matrices de taille 32

D'après les résultats obtenus, on eut percevoir que dans la simulation, les solutions 1, 2, 3 et 4 présentent le même temps de exécution et throughput. De cette façon, en ce qui concerne la performance, le choix de l'une ou l'autre de ces solutions est envisageable. Par rapport à l'utilisation des ressources, la solution 1 présent en moyen les valeurs plus bas d'utilisation, mais les autres solutions restent viables.

B. Multiplication Matrices 64x64

Les ressources utilisés pour la multiplication de matrices de taille 64x64 sont montrés dans le tableau ci-dessous :

64x64

Solution	BRAM_18K (%)	DSP48E (%)	FF (%)	LUT (%)	URAM (%)
0	8	1	0	4	0
1	50	80	7	18	0
2	50	80	7	22	0
3	45	80	7	79	0
4	45	80	7	62	0
5	8	2	1	10	0
6	26	40	5	16	0

Figure 3.5: Ressources utilisés pour la multiplication de matrices de taille 64

Les résultats pour les temps de exécution et pour le throughput sont montrés dans les figures 3.6 et 3.7.

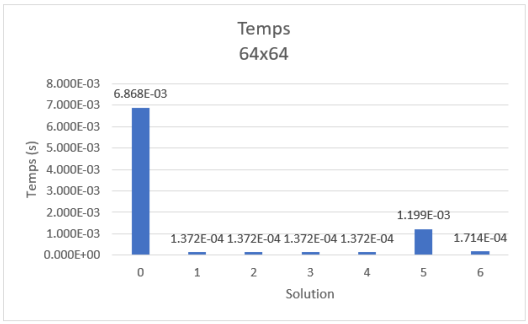


Figure 3.6: Latency pour la multiplication de matrices de taille 64

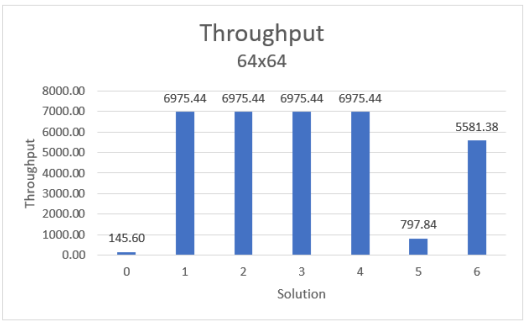


Figure 3.7: Throughput pour la multiplication de matrices de taille 64

À partir des résultats d'utilisation de ressources en 3.7, on voit bien que tous les solutions proposés sont d'accord avec les ressources disponibles dans le circuit. Par rapport au temps, on note un différence de plus en plus grand entre les solutions proposés et la solution originale. Les solutions 1, 2, 3 et 4 présentent des résultats similaires pour le temps d'exécution et throughput. Avec la solution 1 étant la plus efficace en termes de ressources.

C. Multiplication Matrices 128x128

Les ressources utilisés pour la multiplication de matrices de taille 128x128 sont montrés dans le tableau ci-dessous :

128x128

Solution	BRAM_18K (%)	DSP48E (%)	FF (%)	LUT (%)	URAM (%)
0	34	1	0	3	0
1	106	160	14	33	0
2	106	160	14	40	0
3	96	160	14	124	0
4	96	160	14	88	0
5	35	2	1	15	0
6	59	80	9	28	0

Figure 3.8: Ressources utilisés pour la multiplication de matrices de taille 128

Les résultats pour les temps de exécution et pour le throughput sont montrés dans les figures 3.9 et 3.10.

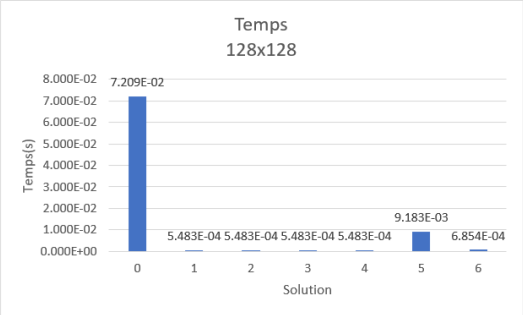


Figure 3.9: Latency pour la multiplication de matrices de taille 128

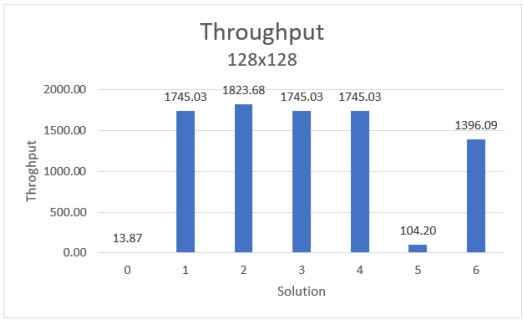


Figure 3.10: Throughput pour la multiplication de matrices de taille 128

Pour la multiplication de matrices de taille 128x128, l'utilisation de ressources devient très important, et presque tous les solutions proposés surpassent la capacité du circuit. Les seules solutions à ne pas surpasser la capacité du circuit sont les solutions original, 5 et 6. D'entre elles, la solution 6 présent un amélioration notable dans le temps de exécution, étant la meilleur solution pour ce cas là.

D. Multiplication Matrices 256x256

Les ressources utilisés pour la multiplication de matrices de taille 128x128 sont montrés dans le tableau ci-dessous :

256x256

Solution	Latency	BRAM_18K (%)	DSP48E (%)	FF (%)	LUT (%)	URAM (%)
0	50659851	142	1	0	4	0
1	262160	237	320	28	63	0
2	262160	237	320	28	76	0
3	262160	216	320	28	141	0
4	262160	216	320	28	141	0
5	8585231	142	2	3	29	0
6	327696	142	160	18	52	0

Figure 3.11: Ressources utilisés pour la multiplication de matrices de taille 128

Pour le multiplication des matrices 256x256, on note bien que dans tous le solutions proposés et même la solution originale, ont surpassé la capacité du circuit FPGA. De cette façon, pour le code C utilisé, nous n'avons pas trouvé des solutions envisageables.

E. Conclusion

À partir de la simulation dans l'environnement Vivado HLS, il est possible de proposer différents solutions pour améliorer le performance de multiplication des matrices dans le FPGA d'une façon très intuitif, seulement à partir d'une code C ou C++. Avec cette méthodologie est possible de trouver des améliorations dans le temps d'exécution d'un facteur de 33.72, 50.07 et 105.17 pour les tailles de 32x32, 64x64, 128x128, respectivement. Pour les matrices plus grandes, avec le code développé ce n'était pas possible de effectuer la multiplication des matrices, une fois que les ressources demandés pour les solutions sont plus élevés dont le circuit peut fournir.

4. Mesures de performances par accélérateur matériel sur circuit FPGA

A. Implémentassions

a) : Pour la partie intégration, comme mentionné dans la section 3, après l'exportation du RTL de l'application HLS, la connexion de ce bloc IP avec les autres parties nécessaires à son fonctionnement a été

effectuée. Comme le montre la Figure 4.1, le bloc IP créé ne contient qu'une entrée et une sortie au format Stream, et donc 3 autres entrées de contrôle, dont une est liée à le clock, deux autres responsables de la réinitialisation et de l'interruption, et la partie connectée à l'AXI.

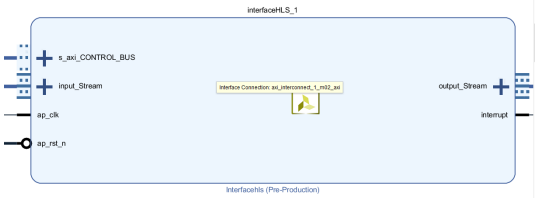


Figure 4.1: Bloque IP crée

Le schéma de connexion a été généré par un script .tcl, où celui-ci connecte toutes les parties nécessaires, à savoir un AXI DMA, 2 AXI pour l'interconnexion, et les parties connectées au processeur et au clock/reset. La figure 4.2 montre comment s'est terminé le système final :

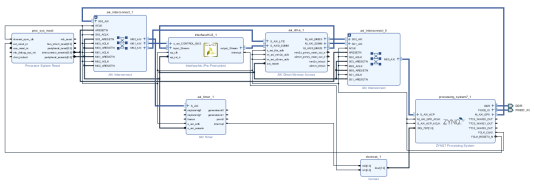


Figure 4.2: Diagramme de bloque sur Vivado

Il a été nécessaire, lors de cette étape, de modifier la valeur de la largeur de bit maximale transmise par le canal DMA, où elle a été portée à 26 (maximum possible), comme le montre la Figure 4.3. Après avoir vérifié toutes les connexions et validé le système, ses sorties et le HDL Wrapper ont été générés automatiquement par le logiciel Vivado.

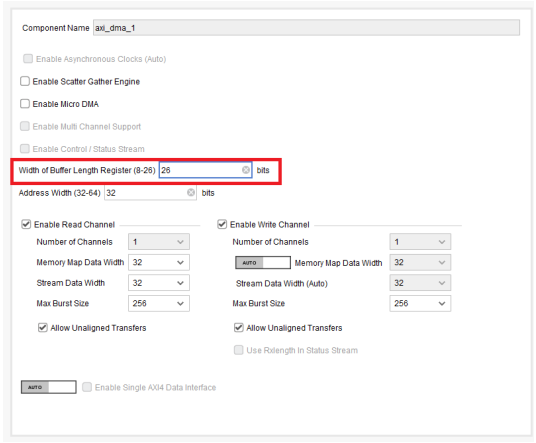


Figure 4.3: Changement des configurations dans le canal DMA

Pour terminer la mise en œuvre dans ce logiciel, le bitstream du schéma réalisé a été généré et son "matériel" a été exporté afin de pouvoir être utilisé dans le FPGA. A la fin de cette étape, il a été possible de voir comment les composants de la carte étaient occupés. La Figure 4.4 représente l'une des simulations, comme cela a été fait pour une matrice de taille 64, avec les optimisations proposées dans la solution 3. Il est donc possible de regarder et de confirmer la théorie, où le système distribue ses composants le plus près possible pour augmenter l'efficacité du processus.

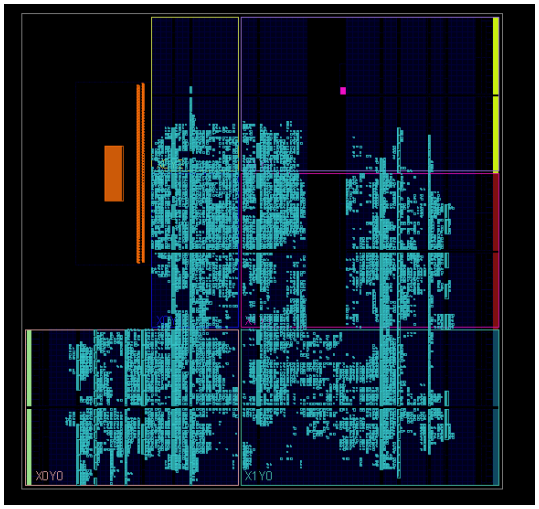


Figure 4.4: Exemple de l'utilisation des composants sur la FPGA après implantation du bloque IP, pour une matrice de 64x64 avec quelques optimisations

En passant à la partie d'implémentation matérielle, on a modifié un code C, selon

le modèle utilisé, où il sera possible de voir les performances des propositions créées, et de vérifier combien de temps il a fallu pour que la multiplication soit effectuée afin de les comparer. Avant de charger ce code sur la board, il a été nécessaire, surtout pour les matrices de grande taille, de modifier les valeurs de pile et de tas utilisées par le code. Pour s'assurer qu'il n'y aurait pas de problèmes avec les calculs et les allocations des variables, le script de lien a été ouvert, un outil qui aide à l'allocation des mémoires, et les valeurs de celles-ci ont été changées en 1 MB, comme le montre la Figure ci-dessous :

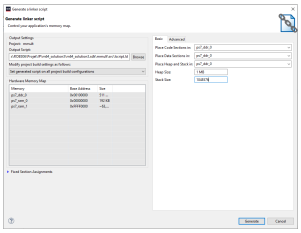


Figure 4.5: Configuration du Link Scripter

Après toutes ces vérifications et implémentations,il suffit d'exécuter le code proposé sur ZedBoard avec l'accélérateur créé et, par l'intermédiaire du terminal en connecté en série avec la carte, il est possible de voir une sortie comme indiqué ci-dessous :

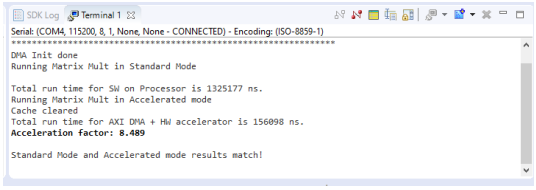


Figure 4.6: Sortie du terminal

B. *Résultats*

a) : Pour les résultats suivants, après toute la routine présentée précédemment dans cette section, nous avons utilisé les solutions proposées dans la section 3, selon la Figure 3.1, où ces stratégies sont mises en évidence. Ainsi, les matrices mises en œuvre au sein de la ZedBoard étaient celles de taille légèrement supérieure, puisque pour les matrices inférieures à 32, ces valeurs étaient déjà très faibles même pour la partie PC. Et pour les valeurs supérieures à 128, celles-ci dépassaient les composants de la ZedBoard et ne pouvaient pas être mises en œuvre.

1) *Matrice 32:*

a) : Comme le montre la Figure 4.7, le graphique représente les valeurs trouvées pour les solutions proposées. Pour cette taille de matrice, toutes les propositions ont pu être mises en œuvre et, comme on peut l'analyser, la plupart des solutions ont présenté des performances proches. La solution 0 concerne le code naïf, il est donc possible de constater une bonne amélioration par rapport aux autres techniques. Par conséquent, dans le cas présent, la première solution proposée comme étant la plus performante, soit une amélioration de 27 fois par rapport à la solution de base.

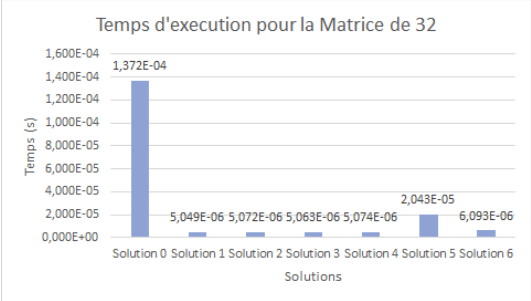


Figure 4.7: Table avec les temps par rapport à les propositions pour la matrice de taille 32

2) *Matrice 64:*

a) : Dans ce cas, toutes les solutions ont pu être mises en œuvre sur le circuit, mais seules quelques valeurs ont été prises, car les propositions avaient des performances très proches. Encore une fois, en comparant avec le code naïf (solution 0), il est possible de voir une grande amélioration avec l'application de ces lignes directrices, les trois premières solutions ont toutes montré de bons résultats pour cette exécution, avec une amélioration de 7 fois. Les résultats sont présentés dans la figure ci-dessous :

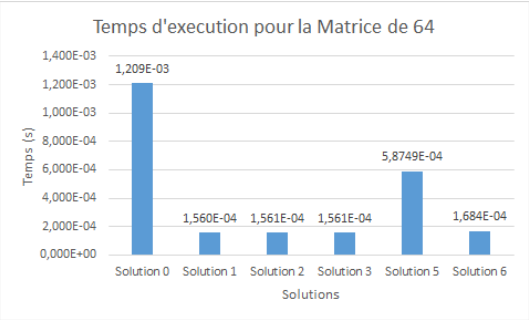


Figure 4.8: Table avec les temps par rapport à les propositions pour la matrice de taille 64

3) Matrice 128:

a) : La matrice de taille 128 avait déjà quelques problèmes, étant donné les simulations présentées dans la section précédente, les solutions 2 et 3 pour présenter une partition de bloc complète, étaient impossibles à mettre en pratique pour cette taille, car cela consommerait plus de composants que ceux disponibles sur la carte. Ainsi, les trois solutions possibles ont été mises en œuvre, mais une fois de plus, la solution 1 s'est avérée plus performante avec la solution 6. Pour ces plus grandes tailles, il est déjà possible de constater que la seule mise en œuvre du pipeline ne garantit plus une bonne performance (un ordre de grandeur différent), car les matrices deviennent trop grandes. La Figure 4.9 montre ces valeurs, et l'amélioration entre la solution la plus performante et la base a été de 38 fois.

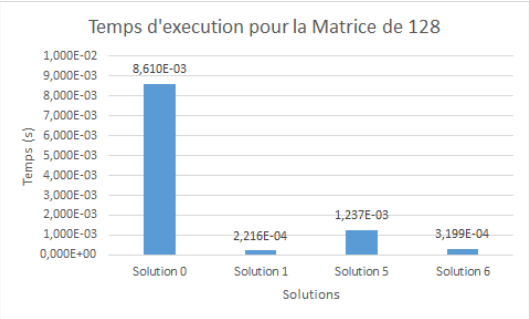


Figure 4.9: Table avec les temps par rapport à les propositions pour la matrice de taille 128

4) Comparaison entre les optimisations:

a) : Dans la Figure 4.10, le temps d'exécution de chaque matrice a été tracé par le code de base. De cette façon, il est possible d'avoir une idée de la vitesse à laquelle les valeurs augmentent et ont tendance à exploser en de très grandes matrices.

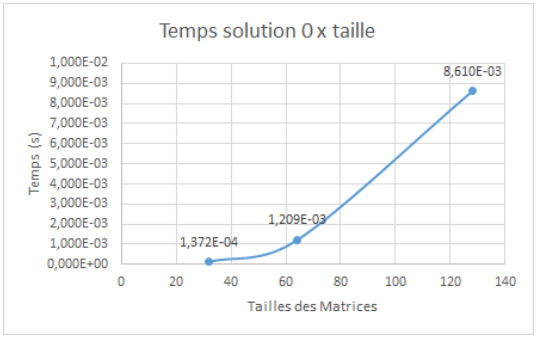


Figure 4.10: Implémentassions du code naïf

D'autre part, dans Figure 4.11, il est possible de voir pour les différentes solutions comment se déroulaient les exécutions de leur temps, de sorte qu'il était possible de mieux les visualiser et les comparer. Ainsi, il est possible de comparer visuellement que les solutions de type 1, 2 et 3 ont obtenu le même résultat pour presque toutes leurs applications, et la solution 6 présente une baisse de performance lorsque le taille était 128. Pour la solution cohérente du pipeline (solution 5), cette échelle est considérée comme conforme à la taille de la matrice proposée.

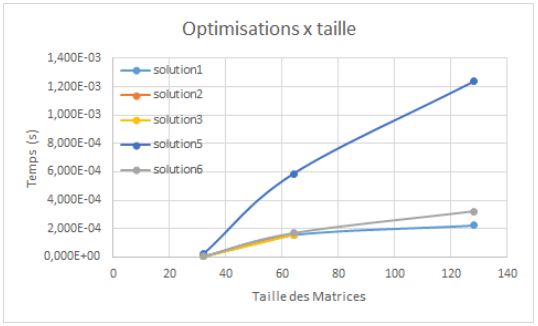


Figure 4.11: Optimisations x Taille de la matrice

C. Conclusions

a) : Après l'exécution de chaque cas, il est apparu clairement que la mise en œuvre directe dans le FPGA de l'accélérateur pour le cas spécifique augmente considérablement la performance du calcul proposé et, pour toutes les solutions proposées, la première a obtenu la meilleure performance par rapport à toutes les autres. Dans le Tableau I sont exposés les temps d'exécution pour chaque matrice de la solution la plus performante.

TABLE I: Tableau avec les performances

	Matrice 32	Matrice 64	Matrice 128
Temps (s)	$5,049 \cdot 10^{-6}$	$1,56 \cdot 10^{-4}$	$2,216 \cdot 10^{-4}$

Compte tenu des matrices de plus grande taille, en raison du type de code mis en œuvre et selon la simulation effectuée par la section précédente, pour ces matrices, leur exécution n'a pas été possible car elle a nécessité plus de BRAM que la quantité disponible sur la board considéré.

5. COMPARAISON ENTRE LES RÉSULTATS

Après avoir effectué l'évaluation des performances de la fonction choisie sur PC, ARM9 et le circuit FPGA, il a été possible de comparer les meilleurs résultats obtenus pour les matrices de taille 64x64 et 128x128. Étant donné que, pour les petites tailles, vu le différence de temps minimale entre les exécutions (pratiquement nulle) et pour les très grandes matrices (256+), il n'était pas possible de comparer les valeurs. Les résultats du temps d'exécution ainsi que l'accélération respective sont présentés dans les deux tableaux suivants.

	PC	**ARM9**	**FPGA**
Temps d'éxecution (mS)	0,20	5,50	0,16

	PC	**ARM9**	**FPGA**
Temps d'éxecution (mS)	1,00	26,87	0,22

TABLE II: 128x128

TABLE III: Temps d'éxecution pour les différentes configurations

On peut remarquer à travers ces résultats que les performances du circuit FPGA sont supérieures à celles du PC et de l'ARM9. Ce résultat était déjà attendu une fois que le circuit FPGA a été programmé de manière à mieux répondre à l'application souhaitée, dans notre cas la multiplication des matrices.

CONCLUSION GÉNÉRALE

a) : Afin de trouver le meilleur compromis de performance-coût pour l'application choisie, 3 configurations différentes ont été étudiées. La fonction de multiplication des matrices a été appliquée, optimisée et évaluée dans un processeur PC généraliste, un processeur embarqué ARM9 et un accélérateur matériel FPGA.

Afin de rechercher le meilleur gain, différentes étapes ont été réalisées pour chaque configuration. Premièrement, pour les performances sur PC, différents paramètres d'optimisation et méthodes de multiplication ont été étudiées. Ensuite, concernant les tests effectués en ARM9, les performances ont été évaluées en faisant varier les options de compilation, les différentes méthodes de multiplication et l'utilisation ou non du cache pour les instructions et les données. Enfin, l'évaluation du circuit FPGA a été basée sur l'analyse de différentes solutions pour lesquelles différentes directives ont été appliquées, telles que le pipeline, la partition et le reshape.

Après avoir effectué la comparaison des performances entre les trois configurations, il a été possible de constater la supériorité du circuit FPGA sur les autres. Cela souligne l'importance d'utiliser un circuit reprogrammable pour des applications spécifiques puisqu'il peut être configuré pour mieux s'adapter à l'application.