ROB306 - Accélérateur Matériel avec HLS

COELHO RUBACK Arthur GUAN Xinyuan SANTOS SANO Matheus ZHANG Chunyu

***Abstract*—Accélération de multiplication matricielle en utilisant une plateforme SoC FPGA Xiling avec un dual core ARM avec comparaison de performance entre PC, ARM et FPGA.**

***Index Terms*—multiplication de matrice, accélération matérielle, FPGA, HLS, Xilinx, ARM**

I. Introduction

La multiplication matricielle est une opération essentielle pour la computation, mathématique, physique, ingénierie et la plus partie des domaines. Dans une partie significative des cas elle a des besoins de vitesse ou consommation énergétique assez évidentes. Dans ces cas, la FPGA semble être une solution viable vu sa capacité de calcul et efficacité énergétique. Pour accomplir cette tache il système doit être mis en place pour communiquer avec la partie logique et transférer les donnés pour le calcul à bonne cadence et bonne ordre. Dans le cadre de ce travail la multiplication matricielle a été effectué sur un data-set standardisé sur PC (x86_64), ARM (single et dual-core) et FPGA. Pour la partie FPGA un programme baremetal a été rajouté sur l'ARM pour exécuter le transfer de donnés, sa récupération et faire le timing de l'opération. Les donnés choisis sont les entiers de 32 bits.

A. Optimisations communes à toutes parties

1) Cache: En enregistrant les matrices dans la mémoire en mode ligne, comme est souvent le cas, on a changé l'ordre de multiplication typique (i,j,k) pour une autre qui profite plus de la relation de proximité des donnés de la cache: (i,k,j).

2) Options compilateur: Le temps de compilation a été augmenté pour réduire le temps d'exécution du programme. Cela se donne grâce aux flags "O1", "O2" et "O3" qui permettent au compilateur de réordonnancer les instructions pour augmenter l'efficacité de l'exécution du programme. Elles ont aussi la désavantage de empêcher le debug et donc ont été utilisés que dans la version finale.

II. Performance de PC

A. Environnement de Test

La configuration de l'environnement est dans Tableau I.

TABLE I
La configuration de l'environnement

Processeur	Intel(R) Core(TM) i7-12700H de 12ème génération
Fréquence	2,30 GHz
Mémoire	16,0 Go de RAM
Système d'exploitation	Windows 11
Compilateur	GNU GCC Compiler

B. Impact des niveaux d'optimisation du compilateur

Nous avons choisi quatre niveaux d'optimisation: O0, O1, O2 et O3. Le rapport d'accélération à chaque niveau d'optimisation sont illustrés dans Fig. 1. Nous pouvons tirer les conclusions suivantes :

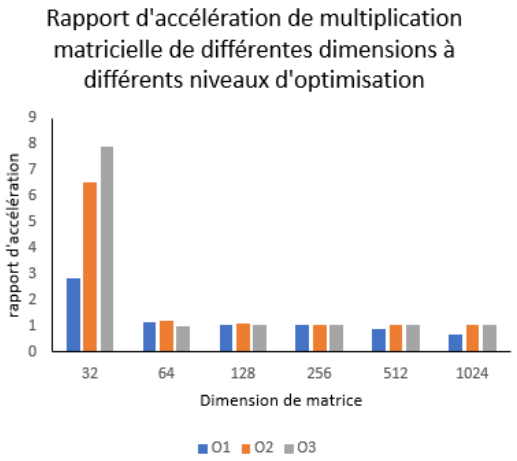

Fig. 1. Rapport d'accélération à différents niveaux d'optimisation

1) Lorsque la dimension de la matrice est de 32, O1, O2 et O3 ont l'effet d'accélération le plus évident sur la matrice.
2) Lorsque la dimension de la matrice est supérieure à 64, les taux d'accélération de tous les niveaux d'optimisation (O1, O2, O3) sont relativement proches, et se rapprochent progressivement de 1.

C. Impact de la parallélisation

Nous explorons l'influence du nombre de threads parallèles. Le rapport d'accélération en utilisant différents nombres de threads est dans Fig. 2, nous pouvons conclure ce qui suit :

1) À mesure que le nombre de threads augmente, l'accélération de la multiplication matricielle augmente généralement.
2) Lorsque la dimension de la matrice est grande, le nombre de threads améliore le taux d'accélération de manière particulièrement significative.
3) Lorsque la dimension de la matrice est petite, l'accélération peut diminuer à mesure que le nombre de threads augmente.

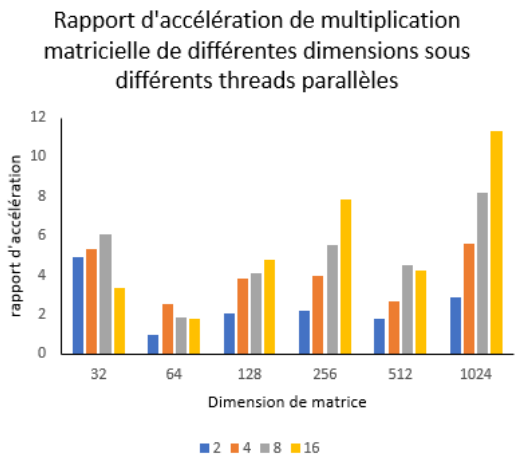

Fig. 2. Rapport d'accélération sous différents threads parallèles

D. Efficacité de l'algorithme de Strassen

Considérons maintenant les avantages potentiels de l'algorithme de Strassen pour accélérer la multiplication matricielle. De nos observations dans Fig. 3, nous pouvons tirer les conclusions suivantes :

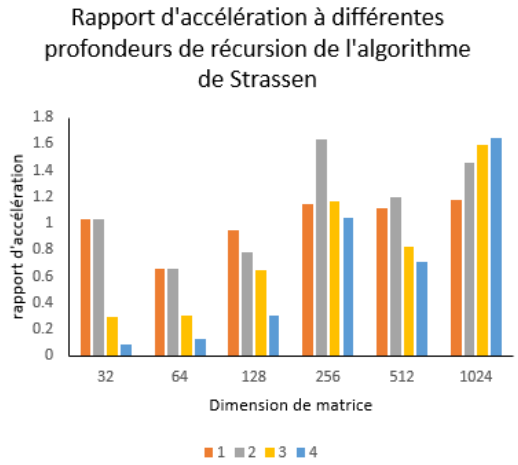

Fig. 3. Rapport d'accélération à différentes profondeurs de récursion

1) Lorsque la dimension de la matrice est petite, le rapport d'accélération à toutes les profondeurs de récursion est inférieur à 1, ce qui signifie que par rapport à l'algorithme traditionnel, l'algorithme de Strassen est moins efficace dans cette dimension.
2) À mesure que la dimension de la matrice augmente, les avantages de l'algorithme deviennent progressivement évidents. Lorsque les dimensions de la matrice sont 256 et 512, la profondeur de récursion optimale est de 2. Lorsque la dimension de la matrice est de 1024, à mesure que la profondeur de récursion augmente, le taux d'accélération augmente progressivement.

E. Impact de SIMD (Instruction unique, multiples données)

La technologie SIMD permet d'exécuter une seule instruction sur plusieurs éléments de données simultanément. Figure 4 donne le rapport d'accélération avec SIMD. À partir de ces données, nous pouvons tirer les conclusions suivantes :

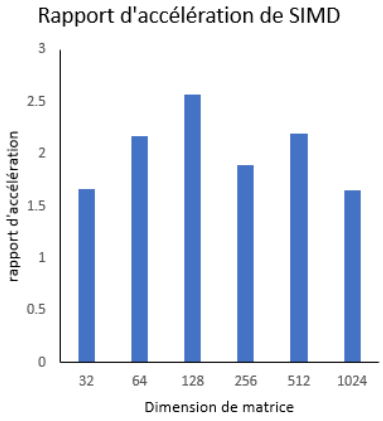

Fig. 4. Rapport d'accélération de SIMD

1) La multiplication matricielle accélérée à l'aide de SIMD a un rapport d'accélération supérieur à 1 dans toutes les dimensions de la matrice, ce qui indique que l'optimisation de la multiplication matricielle par SIMD est efficace.
2) Lorsque les dimensions de la matrice sont 64 et 128, SIMD a le taux d'accélération le plus élevé, proche de 2,5.
3) À mesure que la dimension de la matrice augmente, le taux d'accélération de SIMD diminue progressivement, mais même à la dimension de 1024, le taux d'accélération dépasse 1,5.

F. Impact de changer l'ordre des boucles

En tenant compte du fait que changer l'ordre des boucles de deuxième et troisième couche permet d'accéder à des adresses mémoire consécutives de B, cela optimise l'utilisation du cache et réduit le temps d'accès à la mémoire. De nos observations dans Fig. 5, nous pouvons tirer les conclusions suivantes :

1) Lorsque la dimension de la matrice est de 32, le taux d'accélération est le plus élevé, proche de 1,6.
2) À mesure que la dimension de la matrice augmente, le taux d'accélération diminue progressivement.
3) Lorsque la dimension de la matrice est de 1024, l'accélération est la plus faible, mais toujours légèrement supérieure à 1.

III. Processeur ARM9

Après maximiser les performances du calcul matriciel sur le PC, on essaye de maximiser les performances du calcul matriciel en utilisant le processeur ARM9. A noter que toutes les matrices ont été lues depuis la carte SD: la matrice A,

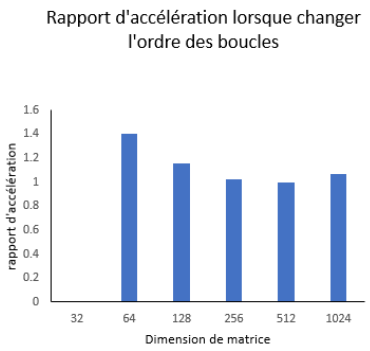

Fig. 5. Rapport d'accélération lorsque changer l'ordre des boucles

la matrice B et la matrice CGolden qui sert à vérifier que le calcul de la multiplication AxB est correct.

Dans un premier temps, la multiplication des matrices a été réalisée sans aucune optimisation, c'est-à-dire en single core, sans optimisation du compilateur et avec le cache désactivé. Il est important de noter que la cache est déjà activée par défault lors de la création d'un projet dans Vivado. Il faut donc désactiver la cache à l'aide de la fonction *Xil_DCacheDisable()*.

Le processeur accède à la mémoire cache beaucoup plus rapidement qu'il n'accède à la mémoire RAM. Cela se traduit par des temps d'exécution nettement inférieurs lorsque la mémoire cache est activée, par rapport à une exécution sans mémoire cache.

Une fois la mémoire cache activée, il est possible de maximiser ses avantages en réorganisant l'ordre des boucles de manière à ce que les éléments des deux tableaux soient accédées par ligne, étant donné que les éléments sont stockés en mémoire de cette manière. Par conséquent, en remplaçant l'ordre des boucles de **IJK** à **IKJ**, la mémoire cache est utilisée plus efficacement, ce qui entraîne une réduction du temps d'exécution de la multiplication matricielle.

Jusqu'alors, aucune optimisation du compilateur (-O0) n'avait été appliquée. Par conséquent, l'optimisation maximale du compilateur (-O3) a été mise en œuvre, ce qui a entraîné une réduction notable du temps d'exécution de la multiplication. Ainsi, avec le cache activé et exploité au maximum en réordonnant les boucles, ainsi que l'optimisation maximale du compilateur (-O3), on a amélioré considérablement la performance de la multiplication matricielle sur un seul cœur (single core). Désormais, pour réduire encore le temps d'exécution de près de la moitié, la multiplication peut être parallélisée sur deux cœurs (dual core).

Pour paralléliser la multiplication, deux cœurs de traitement ont été utilisés (Processeur 0 et Processeur 1). Chacun de ces cœurs est responsable du calcul de la moitié du produit de multiplication de la matrice (matrice de sortie C). Cette parallélisation a été implémentée avec la méthode "**attente active**" (busy waiting), dans laquelle les deux cœurs communiquent à l'aide de drapeaux, qui sont partagés via la mémoire partagée BRAM:

1) Dans un premier temps, le Processeur 0 lit les matrices de la carte SD, tandis que le Processeur 1 attend l'autorisation de lire les matrices de la carte SD. Quand le Processeur 0 termine la lecture, il écrit le flag **START_READ** dans la BRAM;
2) lorsqu'il lit ce flag, le Processeur 1 commence à lire les matrices sur la carte SD, tandis que le Processeur 0 attend le flag **STOP_READ**. Quand le Processeur 1 termine la lecture des matrices, il écrit le flag **STOP_READ** dans la BRAM et commence à calculer la moitié de la multiplication. Lorsque le Processeur 0 lit le flag **STOP_READ** dans la BRAM, il commence l'autre moitié de la multiplication;
3) lorsqu'il términe de multiplier la matrice, le Processeur 1 écrit le flag **STOP_MULT** dans la BRAM. Ainsi, si le Processeur 0 termine la multiplication avant le Processeur 1, le Processeur 0 attend la lecture de le flag **STOP_MULT**;
4) enfin, après que le Processeur 0 termine sa partie de la multiplication et lit le flag **STOP_MULT** (signalant que le Processeur 1 a également terminé sa multiplication), le Processeur 0 calcule le temps de multiplication et la multiplication est terminée.

Une fois la multiplication matricelle est parallélisée, le temps de'exécution obtenu est environ deux fois moins long que dans le cas d'un seul cœur. Dans la Figure 6, vous pouvez vérifier le temps d'exécution de la multiplication matricielle avec une optimisation maximale pour les monocœurs et les doubles cœurs. En comparant les deux, on constate que le temps d'exécution du dual core est environ la moitié du temps obtenu avec le single core. Ce fait est logique puisque le nombre de couleurs a doublé.

En résumé, la performance maximale a été obtenue pour le cas à double cœur avec le niveau d'optimisation maximal du compilateur (-03), avec le cache activé et avec le réordonnancement des boucles (boucle **IKJ**) pour mieux utiliser le cache.

IV. Design FPGA avec Vivado HLS

A. Designs AXI_Lite

Pour réaliser une base de référence, nous utilisons un algorithme naif de multiplication des matrices avec 3 boucles. La figure suivante montre l'évaluation de latence et l'utilisations du circuit de ce design pour toutes les tailles de matrices.

Selon Fig. 7, la latence accroît proportionnellement avec le nombre d'éléments des matrices. Le design utilise toujours 0 BRAM, 3 DSP, de 193 à 246 FF et de 250 à 310 LUT. Le circuit n'est pas vraiment utilisé car aucune optimisation n'est appliquée.

Pour améliorer la performance de l'algorithme, il faut utiliser plus de circuit pour paralléliser les calculs. La première méthode que nous avons proposé pour optimiser la performance est d'utiliser l'option "pipeline". Le tableau II montre la latence et l'utilisation du circuit en appliquant "pipeline" à

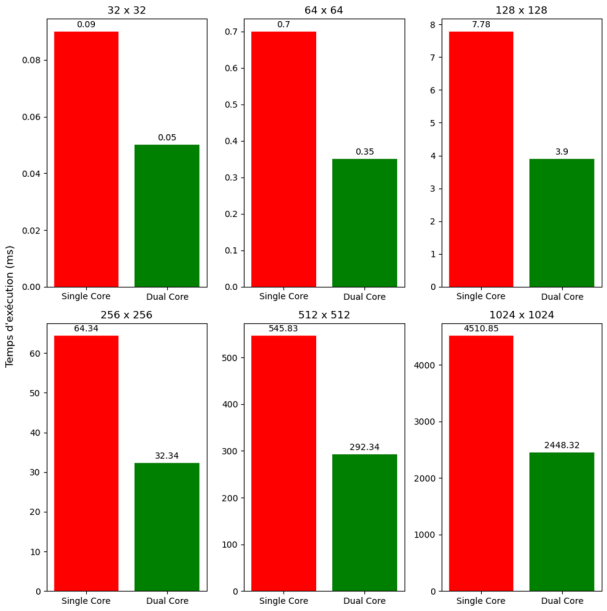

Fig. 6. Temps d'exécution pour toutes les tailles de matrice en single coeur (rouge) et dual coeur (vert).

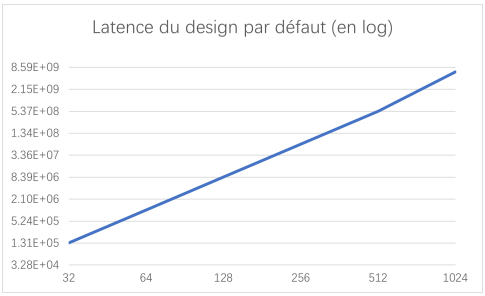

Fig. 7. Latence du design par défaut

chaque boucles pour le design de multiplication des matrices de taille 32*32.

TABLE II
PERFORMANCE ET UTILISATION DU CIRCUIT DU DESIGN AXI_LITE AVEC PIPELINE

Boucle	Latence	BRAM	DSP	FF	LUT
loop_a	32897	6	432	13771	15268
loop_b	32773	6	3	908	1850
loop_c	32771	6	3	751	804

Selon les résultats, dans le cadre avec le "pipeline" seul, la parallélisation de boucle loop_c est la meilleure. Cependant, la latence des trois solutions sont toujours la même. Nous avons trouvé le problème de dépendance de données entre deux lignes de A ou deux colonnes de B. Pour résoudre ce problème, on a utilisé "array_reshape" pour qu'on puisse paralléliser les opérations sur les différentes lignes ou colonnes. Le tableau III montre les résultats d'optimisation avec "array_reshape".

Selon les résultats, l'optimisation sur loop_b est donc la meilleure dans ce cas.

TABLE III
PERFORMANCE ET UTILISATION DU CIRCUIT DU DESIGN AXI_LITE AVEC PIPELINE ET ARRAY_RESHAPE

Boucle	Latence	BRAM	DSP	FF	LUT
loop_a	1030	116	282	10295	10389
loop_b	1030	116	96	4033	2225
loop_c	32775	116	3	4932	13772

Puisqu'on a 3 meilleures solutions dans leur niveau d'optimisation, on peut tracer une courbe Pareto pour voir si on peut équilibrer la latence et l'utilisation du circuit. Fig. 8 montre la courbe Pareto des trois designs.

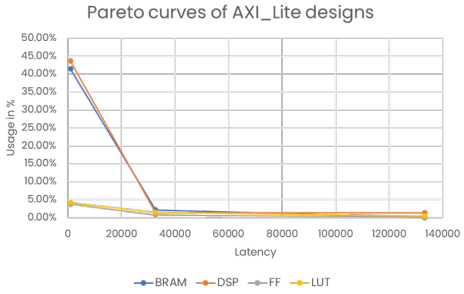

Fig. 8. Courbe Pareto des designs AXI_Lite

Les courbes sont concaves, qui signifie si on veut diminuer la latence, on doit forcément augmenter l'utilisation des 4 composants du circuit. La courbe est bien une courbe Pareto.

B. Designs AXI_Stream

Pour les matrices de taille 128*128, le design AXI_Lite a déjà surchargé le circuit. Pour que les optimisations puissent être appliquées sur les grandes matrices, nous proposons d'utiliser le port AXI_Stream, qui est plus adapté aux grandes matrices. Pour résoudre la dépendance de données, nous proposons d'utiliser un buffer dans l'IP pour stocker les matrices A et B. Cependant, l'utilisation de DSP a beaucoup restreint le niveau de parallélisation, le design a utilisé tous les DSPs à taille 128*128. Pour aller à plus grandes tailles jusqu'à 1024*1024, on propose de séparer les matrices A et B par les blocs de taille 64*64 et de faire la multiplication par blocs. Comme dans la dernière partie, on trace une courbe Pareto pour les cas naif, "pipeline" seul et "pipeline + array_reshape" pour les matrices de taille 32*32. Fig. 9 montre la courbe.

Selon les résultats, on peut toujours équilibrer la latence et l'utilisation du circuits en choisissant les différentes options d'optimisation.

C. Comparaison

Pour montrer la performance de notre accélérateur, nous faisons une comparaison de latence de nos designs AXI_Stream avec le design naif.

Pour montrer mieux l'accélération par rapport à la base de référence, on trace une autre courbe de taux d'accélération.

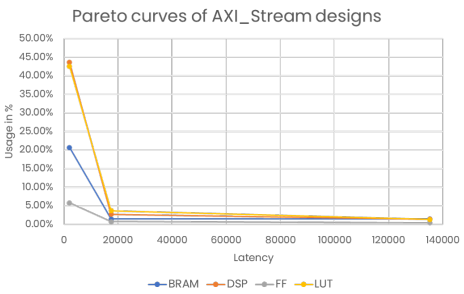

Fig. 9. Courbe Pareto des designs AXI_Stream

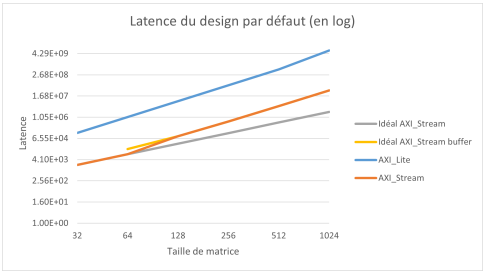

Fig. 10. Comparaison de latence entre le design naif et le design AXI_Stream

Selon Fig. 11, notre accélérateur matériel arrive à accélérer la multiplication matricielle jusqu'à 186 fois avec les matrices de taille 1024*1024.

V. Architecture système

Le système SoC FPGA-ARM a été construit en utilisant 3 DMA's pour fournir les donnés au module IP et est representé su la Vivado dans l'image 12.

Un autre détail important d'implementation a été le réordonnancèrent de d'estocage des donnés matriciels lors que

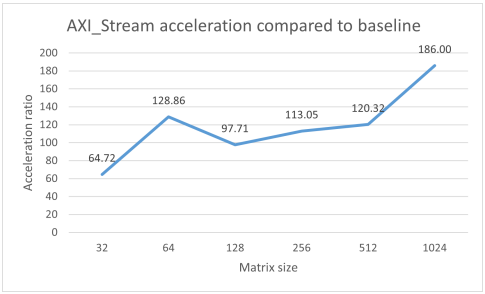

Fig. 11. Taux d'accélération des designs AXI_Stream par rapport au design naif

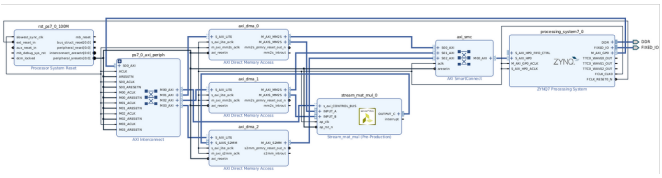

Fig. 12. Architecture système FPGA

la multiplication des gros matrices par blocs. Les blocs on été stockés sequentielment pour minimiser le nombre d'appels aux transferts DMA.

VI. Conclusion

Un graphe comparative de performance de toutes les systèmes presentés jusqu'au moment est trouvable sur la figure 13. Il faut noter que pour PC, ARM et FPGA on dénote la meilleure performance obtenu avec toutes les optimisations et parallélisation. En supplément un test a été mené avec une GPU (Tesla T4) disponible sur la plateforme Notebooks Google et programmé en utilisant "pycuda" et testé avec le même data-set de toutes les autres.

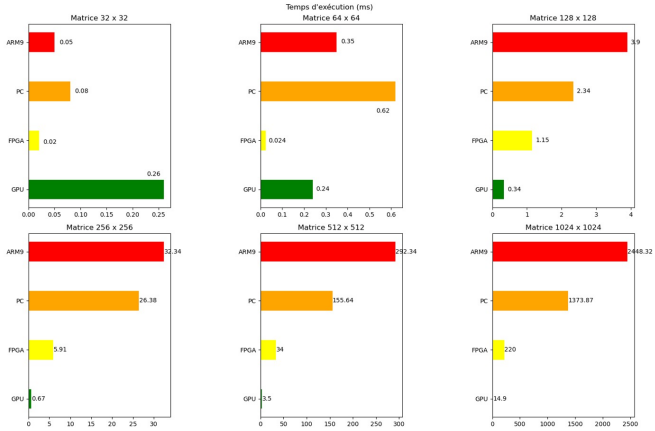

Fig. 13. Comparaison des meilleures performances de chaque système.

Comme montré sur la figure 13 la différence de temps d'exécution est notable plutôt sur des matrices de taille importantes.

La vue les auteurs sur le meilleur choix selon les méthodes proposés est dépendante d'une série de facteurs, notamment: temps disponible pour développement, contrainte de temps, taille souhaité et consommation énergétique. Pour les petites matrices toutes les solutions sont viables, on privilégie donc celles qui sont plus simples a implémenter, notamment le PC. Sur le scénario de temps de développement réduit, on privilégie le PC et la GPU vu que la plupart des développeurs sont capables de faire du software pour ces plateformes. Pour une contrainte de temps forte, la FPGA et la GPU sont les meilleures options. Finalement dans le cadre d'un système embarquée avec forte limitation énergétique (consommations ou dissipation) la FPGA est la meilleure solution.