ROB306 : Accélération matérielle sur FPGA de la multiplication de matrice

Meryl Claussmann
Iad ABDUL RAOUF
Ahmed Yassine HAMMAMI

***Abstract*—La multiplication des matrices est un outil mathématique qui possède plusieurs applications dans des domaines différents.**

***Index Terms*—FPGA, HLS, IP, AXI Stream, ZedBoard.**

I. Introduction

AUJOURD'HUI, par l'augmentation du nombre des données captés, et par le développement de l'intelligence artificielle, on a besoin d'effectuer le calcul matriciel pour des grandes dimensions. Ce calcul peut être implémenté dans des PCs, des centres de calcul ou des circuits embarqués. Par contre, cet outil mathématique possède un coût de calcul très élevé surtout pour des grandes dimensions, par conséquent, on doit trouver une méthode pour optimiser le code et diminuer le temps de calcul.

Dans ce projet, on va examiner les méthodes qui permettent d'optimiser la multiplication matricielle de plusieurs manières.

II. Evaluation sur PC

Dans cette partie, on va tester la multiplication des matrices en utilisant un PC et le compilateur GCC. Le PC utilisé dans cette partie possède les caractéristiques suivantes :

- OS : Windows 10 64 bits.
- CPU : Intel Core i7 8550-U
- Cache niveau L1 : 256 Ko
- Cache niveau L2 : 1 Mo
- Cache niveau L3 : 8 Mo
- RAM : 12 Go
- Fréquence du processeur : 1,99 GHz
- Turbo speed : 4 GHz

A. Algorithme naïf de multiplication des matrices

Dans un premier temps, on va tester un algorithme de multiplication des matrices qui calcule le produit matriciel avec la méthode traditionnelle.

Dans cet algorithme, on a utilisé des matrices dynamiques de taille N*N qui sont remplies respectivement avec les valeurs i et j, et à chaque fois, on fait varier la dimension des matrices afin d'évaluer les performances.

Résultat

On testé cet algorithme pour différents valeurs de N=Dimension des matrices et on a obtenu les résultats suivantes :

```c
int main()
{
    int i,j,k;
    struct timeval tv1, tv2;
    struct timezone tz;
    double elapsed;

    for (i= 0; i< N; i++)
        for (j= 0; j< N; j++)
    {
            A[i][j] = i;
            B[i][j] = j;
    }
    gettimeofday(&tv1, &tz);
    for (i = 0; i < N; ++i) {
        for (j = 0; j < N; ++j) {
            for (k = 0; k < N; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
```

Fig. 1. Multiplication des matrices avec la méthode traditionnelle séquentielle

Dimension des matrices	Temps en secondes
4	0.000002
8	0.000003
16	0.000041
32	0.000195
64	0.000857
128	0.007470
256	0.073643
512	0.689538
1024	6.007488
2048	178.533259

On remarque que le calcul prend un temps remarquable pour des des valeurs de N grands: Pour N=2048, le PC prends 178,533259 secondes pour effectuer le calcul ce qui peut être non-performant dans des algorithmes d'intelligences artificielle (détection d'image par exemple).

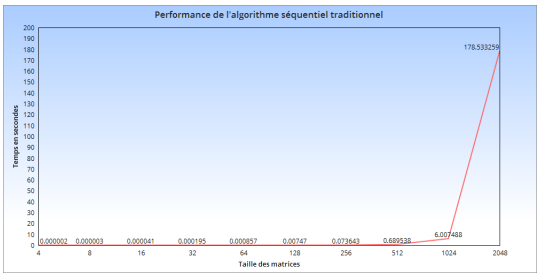


Fig. 2. Résultat pour l'algorithme séquentiel sans optimisation du code

B. Algorithme séquentiel optimisé

Dans cette partie, on va optimiser le code afin d'accélérer la façon de lire de la mémoire. Avec l'algorithme traditionnel, les matrices sont stockées dans la mémoire *heap*.

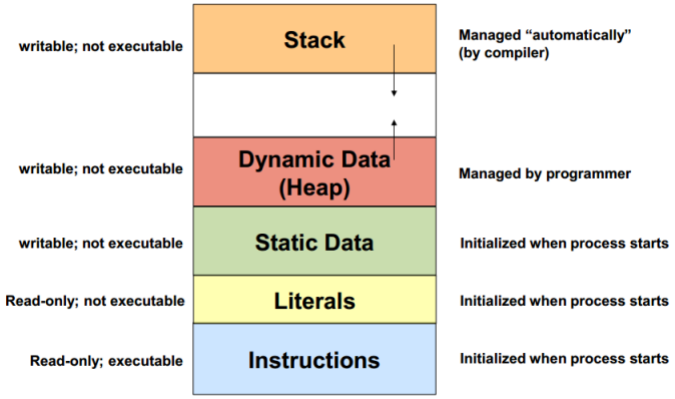


Fig. 3. Architecture du mémoire

Cette méthode possède des inconvénients, en effet, la lecture de cette case mémoire prend plus de temps, pour cela, on va stocker les matrices dans des tableau à une seule ligne N*N afin d'accélérer, d'une part, la lecture et l'écriture des valeurs des matrices et d'autre part, de réduire les sauts entre les cases mémoires. Ces nouveaux tableaux à une dimension seront stocker la mémoire *stack*

```
for(int o=0; o<N; o++){
    for(int f=0; f<N; f++){
        flatA[o * N + f] = A[o][f];
        flatB[f * N + o] = B[o][f];
    }
}


for(i=0; i<N; i++){
    iOff = i * N;
    for(j=0; j<N; j++){
        jOff = j * N;
        tot = 0;
        for(k=0; k<N; k++){
            tot += flatA[iOff + k] * flatB[jOff + k];
        }
        C[i][j] = tot;
    }
    }
```

Fig. 4. Multiplication des matrices avec la méthode séquentielle optimisée

Résultat

On testé cet algorithme pour différents valeurs de N=Dimension des matrices et on a obtenu les résultats suivantes :

Dimension des matrices	Temps en secondes
4	0.000002
8	0.000003
16	0.000031
32	0.000079
64	0.000610
128	0.005039
256	0.037529
512	0.288338
1024	2.320038
2048	18.778784

Pour des petites valeurs de N, on a obtenu les mêmes résultat avec l'algorithme traditionnel, mais pour N grandes, on a eu une grande différence entre le temps nécessaire : 18.778784s contre 178.533259s pour N=2048.

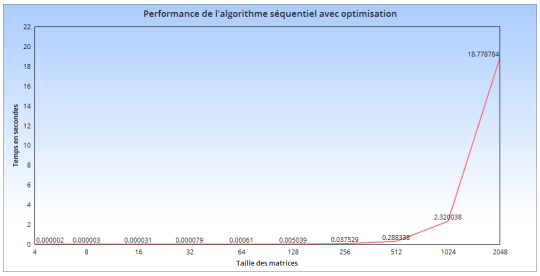


Fig. 5. Résultat pour l'algorithme avec la méthode séquentielle optimisée

C. Algorithme traditionnel avec parallélisation

Dans cette partie, on va analyser l'effet de la parallélisation des boucles *for* sur le temps d'exécution.

On va utiliser l'architecture OpenMp et on va ajouter des directives dans le code afin d'exécuter les boucles *for* sur les *threads* de l'ordinateur. Avec cette architecture, on peut créer n'importe quel nombre de *threads* pour exécuter le code, mais on va se contenter de tester notre algorithme sur 8 *threads*.

```
num_threads=omp_get_num_procs();
omp_set_num_threads(num_threads);
#pragma omp parallel for
for (i= 0; i< N; i++)
    for (j= 0; j< N; j++)
{
        A[i][j] = i;
        B[i][j] = j;
}

gettimeofday(&tv1, &tz);
#pragma omp parallel for private(i,j,k) shared(A,B,C)
for (i = 0; i < N; ++i) {
    for (j = 0; j < N; ++j) {
        for (k = 0; k < N; ++k) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Fig. 6. Multiplication des matrices avec la méthode traditionnelle parallélisée

Ici, nous avons utilisé les directives *pragma omp parallel* pour paralléliser la boucle *for* la plus externe. En utilisant cette instruction, le compilateur délègue des parties de la boucle *for* pour différents *threads*.

Résultat

On testé cet algorithme pour différents valeurs de N et on a obtenu les résultats suivantes :

Dimension des matrices	Temps en secondes
4	0.000049
8	0.000006
16	0.000014
32	0.000031
64	0.000208
128	0.002552
256	0.023951
512	0.182597
1024	1.678434
2048	43.449357

Pour des petites valeurs de N, on a obtenu résultats plus grands que la méthode traditionnelle. Ceci s'explique par le retards fait par les *threads* pour pouvoir collecter les valeurs calculées. Par contre, on observe une optimisation pour des valeurs de N grandes par rapport à l'algorithme séquentiel traditionnel, mais ces résultats sont plus grands que celles obtenues avec l'algorithme optimisés: 43.449357s contre 18.778784s pour la version séquentielle optimisées (N=2048).

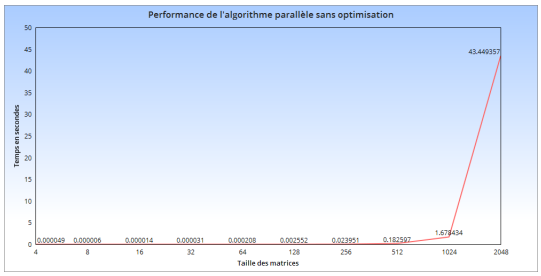


Fig. 7. Résultat pour l'algorithme avec la méthode traditionnelle parallélisée

D. Algorithme optimisé avec parallélisation

Dans cette partie, on va combiner l'algorithme optimisé ac=vec la parallélisation :
13

```
#pragma omp parallel for
for (i= 0; i< N; i++)
    for (j= 0; j< N; j++)
{
        A[i][j] = i;
        B[i][j] = j;
}

gettimeofday(&tv1, &tz);
#pragma omp parallel for
for(int o=0; o<N; o++){
    for(int f=0; f<N; f++){
        flatA[o * N + f] = A[o][f];
        flatB[f * N + o] = B[o][f];
    }
}


#pragma omp parallel for private(i,j,k,iOff,jOff,tot) shared(flatA,flatB,C)
for(i=0; i<N; i++){
    iOff = i * N;
    for(j=0; j<N; j++){
        jOff = j * N;
        tot = 0;
        for(k=0; k<N; k++){
            tot += flatA[iOff + k] * flatB[jOff + k];
        }
        C[i][j] = tot;
    }
    }
```

Fig. 8. Multiplication des matrices avec la méthode optimisée parallélisée

Résultat

Dimension des matrices	Temps en secondes
4	0.000008
8	0.000009
16	0.000010
32	0.000028
64	0.000157
128	0.001935
256	0.008259
512	0.069373
1024	0.512485
2048	4.855594

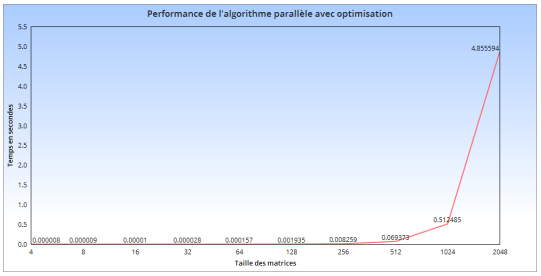


Fig. 9. Résultat pour l'algorithme avec la méthode optimisée parallélisée

E. Comparaison entre les différentes méthodes

On a analysé dans les sections précédentes les résultats de chaque algorithme. En effet, le meilleur algorithme qui donne de bonnes résultats est celui qui implémente la version optimisée parallélisée. Pour N = 2048, on a obtenu 4.855594s contre 178.533259s pour la version séquentielle traditionnelle (facteur d'accélération = 36,76). Pour l'algorithme traditionnel parallélisé, on obtenu de bonnes résultats seulement pour de grandes valeurs de N.

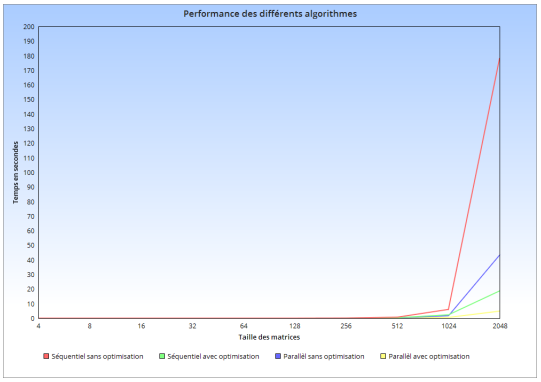


Fig. 10. Performance des différents algorithmes

Zoom pour N = 128

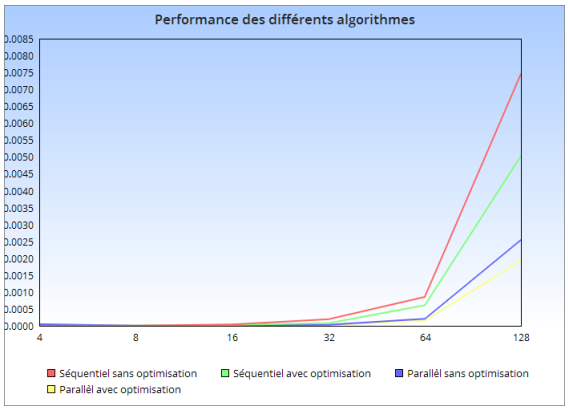


Fig. 11. Performance des différents algorithmes (*Zoom pour N= 128)*

III. Optimisations de l'ARM9 seul

Le but de cette partie est d'étudier les performances de l'ARM9 et de les comparer à celles du pc, ainsi que d'étudier

l'impact et les performances de diverses optimisations du processeur.

A. Basic Design sur Vivado - préliminaires

Afin de pouvoir mesurer les performances de l'ARM9, nous cherchons à faire tourner un code qui implémente la multiplication de matrices pour le processeur. pour cela, il faut tout d'abord construire un design Vivado qui symbolisera l'hardware, sur lequel on fera tourner le software codé en C dans SDK.

Le design choisi est un design basique, disponible sur le tutoriel Xilinx.

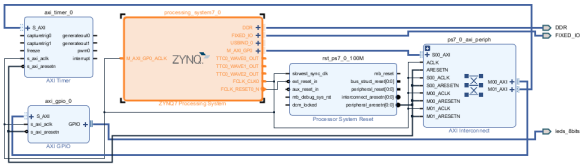


Fig. 12. Basic Design construit dans Vivado, qui servira de hardware support du software de SDK

B. Description de la mesure du temps

Dans un premier temps, afin de mesurer les temps d'exécution, la fonction `XTime` de Xilinx est utilisée, entre les lignes qui implémentent la multiplication de deux matrices d'entrée et produisent une matrice de sortie. L'implémentation des matrices (qui ne rentre pas dans la mesure du temps) se fait de manière aléatoire grâce à la fonction `rand` ; ceci permet au programme de ne pas "retenir" les valeurs implémentées et ainsi d'appréhender le calcul : les calculs doivent être fait entièrement chaque fois. Cependant, la précision de cette mesure du temps est à vérifier, c'est pourquoi nous allons également utiliser un autre outil de mesure du temps : le bloc ip AXI Timer de Vivado.

Les valeurs obtenues avec cette nouvelle mesure du temps seront comparées (pour le cas de l'algorithme naïf sans optimisation) à celles obtenues avec la technique initiale en guise de comparaison de l'efficacité de la technique initiale que nous avons utilisé pour les comparaisons et évaluations de performance qui vont suivre. Si ces valeurs sont proches, alors nous pourrons décréter que la technique initiale était assez efficace et que ses résultats sont justes. Si les valeurs sont trop éloignées, alors les valeurs absolues des résultats seront faussement estimées, mais par contre les variations resteront quant à elles correctes

En conclusion, après implémentation, la différence de précision est visible au centième : la première technique de mesure du temps, qui est celle qui a été utilisée, et donc assez précise.

On précise également que toutes les mesures de temps sont données en unités US, c'est-à-dire en microsecondes.

C. Implémentation du code naïf

Dans tout le projet, nous avons implémenté les matrices de tests dynamiquement, afin que le processeur recalcule chaque fois les temps d'exécution, et qu'il n'utilise pas des raccourcis de calculs dans le cas où les valeurs seraient toutes les mêmes.

Afin de commencer les tests de performance de l'ARM9, nous implémentons l'algorithme de multiplication de matrice naïf utilisé par l'ingénieur T1, mais adapté au logiciel SDK et ses contraintes. Ce code est disponible dans le fichier `sequentiel_sans_opti_meryl`.
L'algorithme calcule le produit matriciel avec la méthode traditionnelle.

Avec ce code c, nous obtenons les résultats suivants du temps d'exécution en fonction de la taille de la matrice :

N	uS (µs)
4	4,89
8	31,65
16	249,5
32	1921,63
64	15212,5
128	181219,32
256	1553405,84
512	18523384,78
1024	202535743,5
2048	1665042076,78

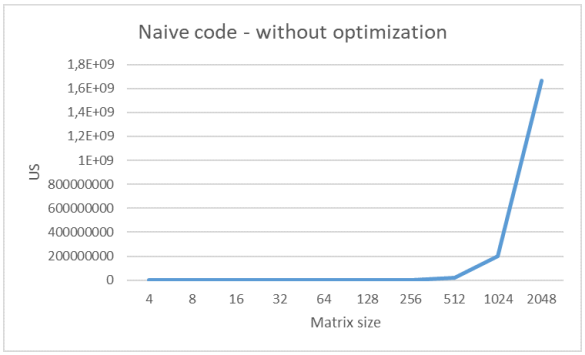


Fig. 13. Naive code - without optimization

L'augmentation du temps d'exécution avec la taille de la matrice est exponentiel, et atteint rapidement des temps non convenables. C'est pour cette raison que nous voulons optimiser les performances du processeur.
La première optimisation envisagée est celle du code c.

par ailleurs, en comparant ces performances avec celle du pc, il est possible de se rendre compte à quel point le pc est bien plus performant que le processeur : pour toutes les valeurs, nous avons un taux de variation de 99,99%, cela signifie que le pc est 100% plus performant que le processeur.

D. Optimisation du code

1) Description: La première optimisation envisagée est celle du code. Le but est d'accélérer la façon de lire de

la mémoire. Comme dit précédemment, avec l'algorithme traditionnel, les matrices sont stockées dans la mémoire *heap*. Cette méthode possède des inconvénients car la lecture de cette case mémoire prend plus de temps. pour y remédier, on va stocker les matrices dans des tableau à une ligne principale N*N afin d'accélérer, d'une part, la lecture et l'écriture des valeurs des matrices et d'autre part, de réduire les sauts entre les cases mémoires. Ces nouveaux tableaux à une dimension seront stocker la mémoire *stack*.
Ce code est disponible dans le fichier `sequentiel_opti_meryl`.

Grâce à cette optimisation, nous obtenons les performances suivantes :

N	uS (µs)
4	4,91
8	24,89
16	178,61
32	1285,21
64	9892,83
128	83084,73
256	669354,62
512	5423926,6
1024	43383792,71
2048	346372699,2

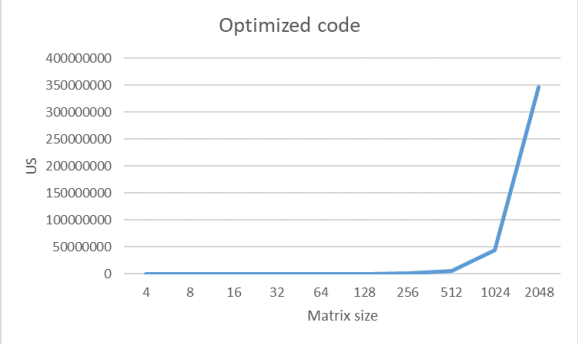


Fig. 14. Optimized code

Les performances semblent déjà bien meilleures : nous allons par la suite comparer et chiffrer la comparaison des performances du code naïf avec le code optimisé.

par ailleurs, en comparant ces performances avec celle du pc, il est possible de se rendre compte à quel point le pc est bien plus performant que le processeur : pour toutes les valeurs, nous avons un taux de variation de 99,99%, cela signifie que le pc est 100% plus performant que le processeur.

2) Comparaison du code naïf avec le code optimisé: Grâce aux résultats obtenus, nous pouvons chiffrer l'amélioration des performances :

N	Taux de variation (%)
4	0,4
8	21,4
16	28,4
32	33,1
64	35,0
128	54,2
256	56,9
512	70,7
1024	78,6
2048	79,1

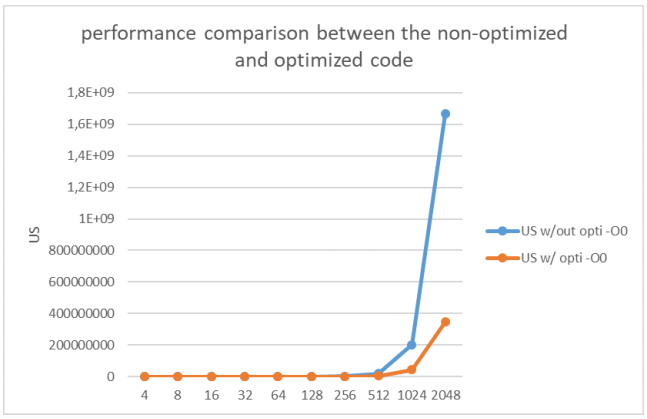


Fig. 15. Comparison between the non-optimized and optimized code performances

On remarque que pour une petite valeur de N (N=4), l'optimisation du code n'est pas si utile ; elle l'est à contrario indispensable dès que N atteint de grandes valeurs, au point de réduire par 2 le temps d'exécution dès que N=128, voire de quasiment réduire le temps de presque 100% pour des rès grandes valeurs. L'utilité de l'optimisation du code est donc non réfutable, voir même nécessaire.
Cette optimisation est donc validée.

E. Optimisation du compilateur - code naïf

Nous allons analyser les impacts d'une optimisation du compilateur, grâce aux optimisations suivantes :

- -O0 : pas d'optimisation activée, compile et construit l'environnement rapidement mais le code reste long à exécuter ; la taille du code ou l'utilisation de piles sont beaucoup plus long qu'avec n'importe quel autre mode d'optimisation. Le code généré est très proche du code source, mais plus de code est généré, dont du code inutilisé.
- -O1 : les optimisations du coeur du compilateur sont rendues possibles, ce mode est très utile pour le debug (mais la fidélité de l'information de débug est réduite), sans allongement inutile du code et donc une meilleure qualité du code qu'avec -O0. L'utilisation des piles est également améliorée.
- -O2 : il s'agit d'une meilleure optimisation que -O1 ; augmente la performance temporelle, mais dégrade la facilité d'usage du debug, et peut conduire à une augmentation de la taille du code par rapport à -01 ;

avec cette optimisation, il est possible que le compilateur génère des vecteurs d'instructions non désirables.

- -O3 : il s'agit d'une meilleure optimisation que -O2 ; ces optimisations recquièrent un temps d'analyse de compilation et de ressources beaucoup plus important ; le code généré voit ses performances s'accroitrent, qu'importe sa taille, ce qui peut donc conduire à une augmentation de sa taille ; l'activité de debug est également moins simple.
- -Os : réduit de manière équilibrée la taille du code et augmente la performance temporelle ; cette optimisation peut être très semblable à l'optimisation -O2 ou -O3. par rapport à -O3, il y a tout de même une réduction de la taille du code, et le debug n'est pas facilité contrairement à -O1.

Nous allons donc étudier si ces théories sont bien appliquées dans notre cas de figure, concernant les performances temporelles de l'ARM9 sur code naïf tout d'abord.

1) Optimisation -O1:

N	US (µs)
4	1,45
8	6,62
16	50,48
32	379,34
64	2953,31
128	83588,23
256	772429,01
512	15622142,2
1024	153152739,8
2048	1286822053

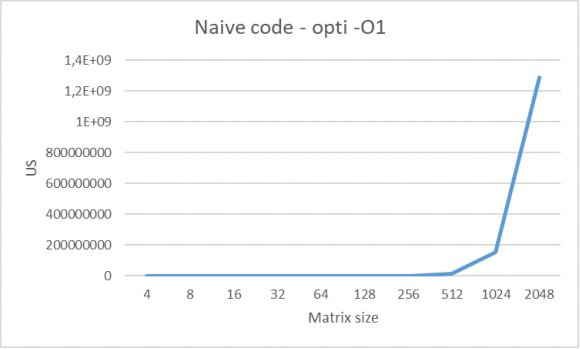


Fig. 16. Naive code - optimization -O1

N	Taux de variation (%) par rapport à -O0
4	70,3
8	79,1
16	79,8
32	80,3
64	80,6
128	53,9
256	50,3
512	15,7
1024	24,4
2048	22,7

On remarque une amélioration notable des performances de -01 par rapport à -O0, d'autant plus grande que la taille de la matrice est petite. Cette observation corrobore ce qui a été décrit pour cette optimisation plus haut.

2) Optimisation -O2:

N	US (µs)
4	1,12
8	4,61
16	29,91
32	243,14
64	1895,11
128	40934,21
256	306821,71
512	8312145,27
1024	77848660,66
2048	617728414,2

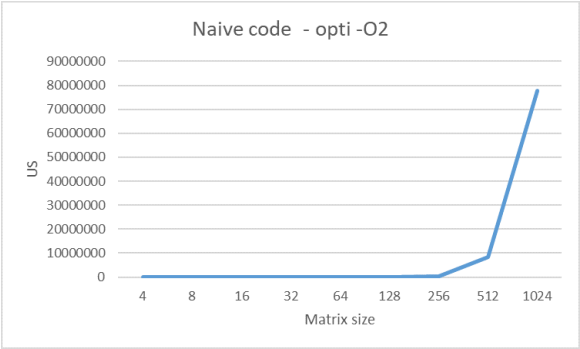


Fig. 17. Naive code - optimization -O2

N	Taux de variation (%) par rapport à -O0
4	77,1
8	85,4
16	88,0
32	87,3
64	87,5
128	77,4
256	80,2
512	55,1
1024	61,6
2048	62,9

N	Taux de variation (%) par rapport à -O1
4	22,8
8	30,4
16	40,7
32	35,9
64	35,8
128	51,0
256	60,3
512	46,8
1024	49,2
2048	52,0

On remarque une amélioration notable des performances de -02 par rapport à -O0, et qui ne varie pas autant avec la taille de la matrice que -O1. Cette observation corrobore ce qui a été décrit pour cette optimisation plus haut ; d'autant plus qu'on voit bien que -O2 est également meilleure que -O1 à 30-50%.

3) Optimisation -O3:

N	US (µs)
4	1,46
8	4,39
16	19,8
32	228,13
64	1740,99
128	18970,16
256	163938,99
512	1540050,27
1024	12925015,51
2048	110449133,4

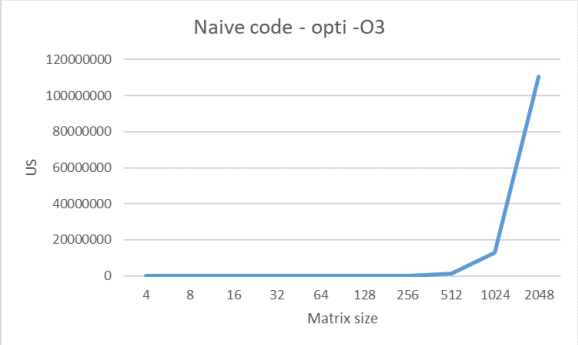


Fig. 18. Naive code - optimization -O3

N	Taux de variation (%) par rapport à -O0
4	70,1
8	86,1
16	92,0
32	88,1
64	88,6
128	89,5
256	89,4
512	91,7
1024	93,6
2048	93,4

N	Taux de variation (%) par rapport à -O2
4	-30,4
8	4,8
16	33,8
32	6,2
64	8,1
128	53,7
256	46,6
512	81,5
1024	83,4
2048	82,1

On remarque une amélioration notable des performances de -03 par rapport à -O0, et qui, contrairement aux autres, augmente avec la taille de la matrice. Cette observation corrobore ce qui a été décrit pour cette optimisation plus haut. Ses variations relatives par rapport à -O2 sont quant à elles plus contrastées, puisqu'on voit une baisse de performances pour des petites tailles de matrices (N=4), de légères améliorations jusqu'à N=64 puis des améliorations notables à partir de N=128, qui s'accroissent avec N. Encore une fois, cela confirme le fait que ces optimisations -O3 recquièrent un temps d'analyse de compilation et de ressources beaucoup plus important (ce qui se perd sur des petites tailles de matrices car alors le gain n'est plus assez important) mais le code généré voit ses performances s'accroitrent (notamment sur des grandes tailles de matrices).

4) Optimisation -Os:

N	US (µs)
4	1,29
8	4,75
16	31,03
32	256,97
64	1883,32
128	40947,9
256	329258,29
512	8340441,08
1024	77871488,72
2048	614123503,3

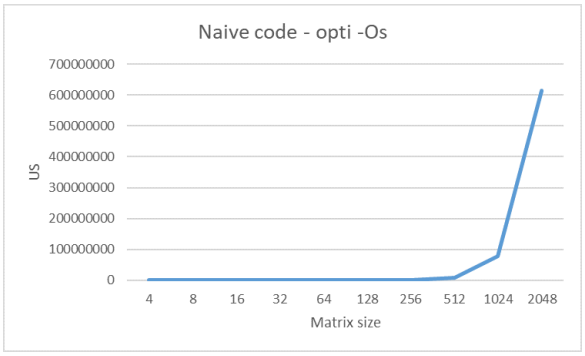


Fig. 19. Naive code - optimization -Os

N	Taux de variation (%) par rapport à -O0
4	73,6
8	85,0
16	87,6
32	86,6
64	87,6
128	77,4
256	78,8
512	55,0
1024	61,6
2048	63,1

N	Taux de variation (%) par rapport à -O2
4	-15,2
8	3,0
16	3,7
32	5,7
64	0,6
128	0,03
256	7,3
512	0,3
1024	0,03
2048	0,6

N	Taux de variation (%) par rapport à -O3
4	11,6
8	-8,2
16	-56,7
32	-12,6
64	-8,2
128	-115,9
256	-100,8
512	-441,6
1024	-502,5
2048	-456,0

Nous analysons le fait que cette optimisation présente toujours une amélioration de performance non négligeable par rapport à -O0, qui diminue de nouveau avec la taille de la matrice (en raison de la réduction de la taille du code, moins significatif quand la taille de la matrice augmente). Comme prévu, cette optimisation présente de faible variation avec l'optimisation -O2, et est bien moins performante que l'optimisation -O3 pour N¿=8.

5) Comparaison finale: Nous obtenons le graphe de conclusion suivant :

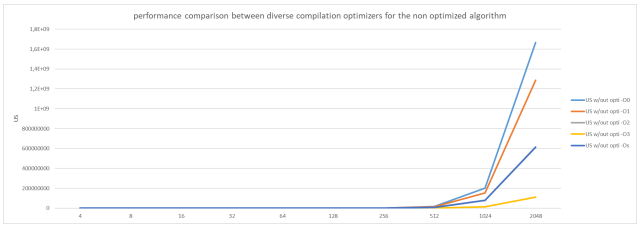


Fig. 20. performance comparison between the naive and optimized code

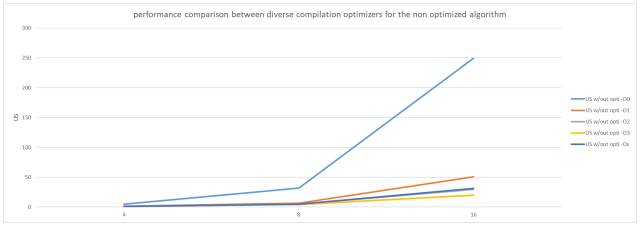


Fig. 21. Zoom for N=4, 8, 16

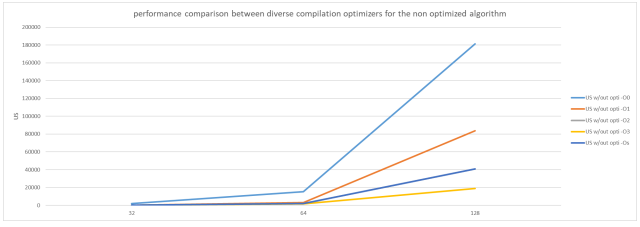


Fig. 22. Zoom for N=32, 64, 128

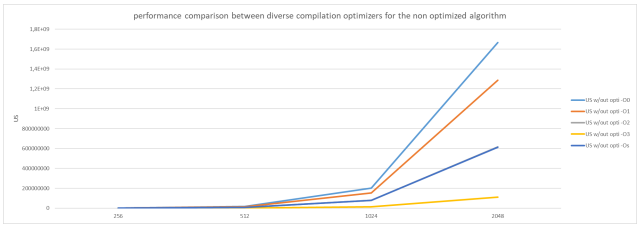


Fig. 23. Zoom for N=256, 512, 1024, 2048

F. Optimisation du compilateur - code optimisé

On a montré précédemment que le code optimisé présentait des performances bien meilleures que le code naïf. Nous nous attendons donc à avoir des performances encore meilleures dans cette section que dans la section précédente

1) Optimisation -O1:

N	US (µs)
4	1,92
8	6,12
16	34,03
32	251,13
64	1840,03
128	21140,35
256	180423,82
512	1650663,1
1024	13773234,55
2048	114557247,3

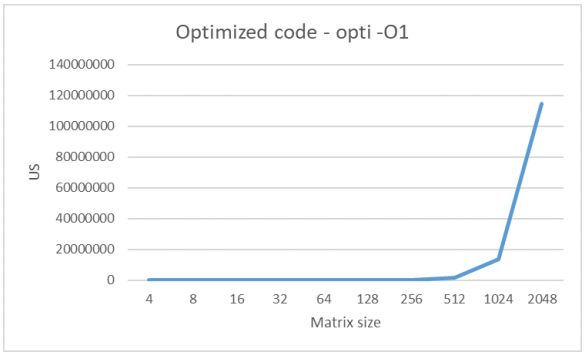


Fig. 24. Optimized code - optimization -O1

N	Taux de variation (%) par rapport à -O0
4	60,9
8	75,4
16	80,9
32	80,5
64	81,4
128	74,6
256	73,0
512	69,6
1024	68,3
2048	66,9

N	Taux de variation (%) par rapport à -O1 sans optimisation
4	-32,4
8	7,6
16	32,6
32	33,8
64	37,7
128	74,7
256	76,6
512	89,4
1024	91,0
2048	91,1

N	Taux de variation (%) par rapport à -O0
4	65,4
8	76,7
16	80,7
32	80,1
64	80,8
128	73,8
256	72,2
512	69,2
1024	68,0
2048	66,7

On remarque une amélioration notable des performances de -01 par rapport à -O0. Cette observation corrobore ce qui a été décrit pour cette optimisation plus haut. Comme annoncé, cet arrangement est également plus performant que l'arrangement sans optimisation du code, ce qui n'est pas étonnant puisque l'optimisation du code consiste à réduire la matrice à une colonne et ligne principales, et donc à gagner du temps sur les sauts de lignes : donc plus la taille est grande et plus le temps gagné sur les sauts de lignes est lui aussi important.

N	Taux de variation (%) par rapport à -O1
4	11,5
8	5,2
16	-1,3
32	-1,6
64	-3,5
128	-2,8
256	-3,0
512	-1,2
1024	-0,7
2048	-0,6

2) Optimisation -O2:

N	US (µs)
4	1,7
8	5,8
16	34,48
32	255,2
64	1904,33
128	21741,94
256	185835,41
512	1670842,75
1024	13866640,36
2048	115265540,5

N	Taux de variation (%) par rapport à -O2 sans optimisation
4	-51,8
8	-25,8
16	-15,3
32	-4,9
64	-0,49
128	46,9
256	39,4
512	79,9
1024	82,2
2048	81,3

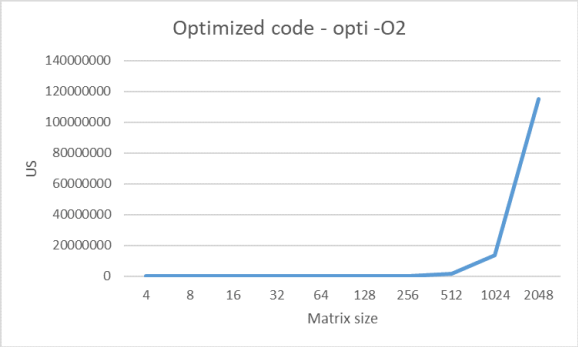


Fig. 25. Optimized code - optimization -O2

On remarque une amélioration notable des performances de -02 par rapport à -O0, et qui ne varie pas autant avec la taille de la matrice que -O1. Cette observation corrobore ce qui a été décrit pour cette optimisation plus haut et est similaire à celles des résultats obtenus pour le code non optimisé ; par contre un résultat plus surprenant est le fait que -O2 n'est pas bien plus performant que -O1, contrairement au cas avec le code non optimisé. Un autre résultat surprenant : pour des valeurs de N relativement petites (inférieures à 64), le code optimisé ne présente pas une amélioration des performances par rapport au code optimisé, au contraire ; ce qui n'était pas le résultat attendu : l'optimisation du code qui consiste à gagner du temps sur les sauts de ligne est donc vraiment visible sur les grandes valeurs de N.

3) Optimisation -O3:

N	US (µs)
4	1,25
8	4,13
16	21
32	234,68
64	1864,51
128	16568,09
256	144248,05
512	1436060
1024	11890627,6
2048	95590301,1

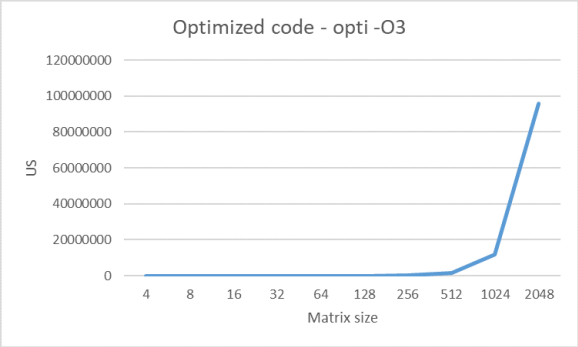


Fig. 26. Optimized code - optimization -O3

N	Taux de variation (%) par rapport à -O0
4	74,5
8	83,4
16	88,2
32	81,7
64	81,2
128	80,1
256	78,4
512	73,5
1024	72,6
2048	72,4

N	Taux de variation (%) par rapport à -O2
4	26,5
8	28,8
16	39,1
32	8,0
64	2,1
128	23,8
256	22,4
512	14,1
1024	14,3
2048	17,1

N	Taux de variation (%) par rapport à -O3 sans optimisation
4	14,4
8	5,9
16	-6,1
32	-2,9
64	-7,1
128	12,7
256	12
512	6,8
1024	8,0
2048	13,5

On remarque une amélioration notable des performances de -03 par rapport à -O0, et qui, contrairement aux autres, reste constant avec la taille de la matrice. Ses variations relatives par rapport à -O2 sont quant à elles plus contrastées, puisqu'on voit une augmentation de performances pour des petites tailles de matrices ($N_¡=4$), de légères pertes jusqu'à N=64 puis des améliorations notables à partir de N=128, qui s'accroissent avec N. Encore une fois, cela confirme le fait que ces optimisations -O3 recquièrent un temps d'analyse de compilation et de ressources beaucoup plus important (ce qui se perd sur des petites tailles de matrices car alors le gain n'est plus assez important) mais le code généré voit ses performances s'accroitrent (notamment sur des grandes tailles de matrices). Néanmoins, ce gain de performance est moins important qu'avec le code non optimisé, car alors le gain temporel avec la réduction des sauts de lignes a déjà été réalisé sur le code optimisé, donc il s'en ressent moins sur le compilateur. pour finir, dans ce cas, les variations avec le code non optimisé pour une compilation en -O3 ne sont pas non plus si signicantes, qui était ce à quoi nous aurions pu nous attendre au vu des résultats de comparaison pour l'option de compilation -O0. Ainsi, ajouter des optimisations de compilateur biaise les optimisations de code.

4) Optimisation -Os:

N	US (µs)
4	1,11
8	4,98
16	32,76
32	259,65
64	1982,87
128	22472,56
256	192294,26
512	1714233,72
1024	14220831,5
2048	118239987,9

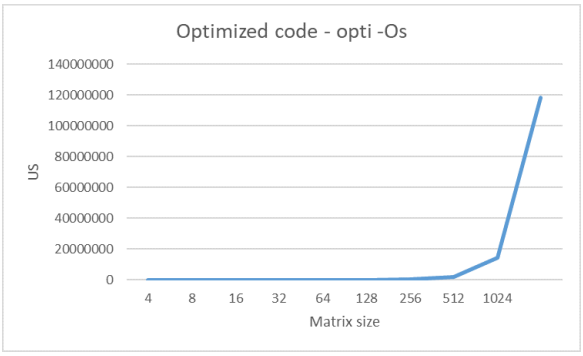


Fig. 27. Optimized code - optimization -Os

N	Taux de variation (%) par rapport à -O0
4	77,4
8	80
16	81,7
32	79,8
64	80,0
128	73,0
256	71,3
512	68,4
1024	67,2
2048	65,9

N	Taux de variation (%) par rapport à -O2
4	34,7
8	14,1
16	5,0
32	-1,7
64	-4,1
128	-3,4
256	-3,5
512	-2,6
1024	-2,6
2048	-2,6

N	Taux de variation (%) par rapport à -O3
4	11,2
8	-20,6
16	-56
32	-10,6
64	-6,3
128	-35,6
256	-33,3
512	-19,3
1024	-19,6
2048	-23,7

N	Taux de variation (%) par rapport à -O3 sans optimisation
4	14,0
8	-4,8
16	-5,6
32	-1,0
64	-5,3
128	45,1
256	41,6
512	79,4
1024	81,7
2048	80,7

Nous analysons le fait que cette optimisation présente toujours une amélioration de performance non négligeable par rapport à -O0, qui diminue de nouveau avec la taille de la matrice (en raison de la réduction de la taille du code, moins significatif quand la taille de la matrice augmente). Comme prévu, cette optimisation présente de faible variation avec l'optimisation -O2 (sauf pour N petit), et est bien moins performante que l'optimisation -O3 pour N¿=8, avec tout de même une moins grosse perte de performance que pour le code non optimisé. A l'inverse des précédents compilateurs, celui-ci présente un véritable avantage avec le code optimisé par rapport au code non optimisé avec ce compilateur, en particulier pour des grandes valeurs de matrices, et ce qui corrobore une fois de plus ce qui a été dit précédemment concernant l'optimisation du code et -Os.

5) Comparaison finale: Nous obtenons le graphe de conclusion suivant :

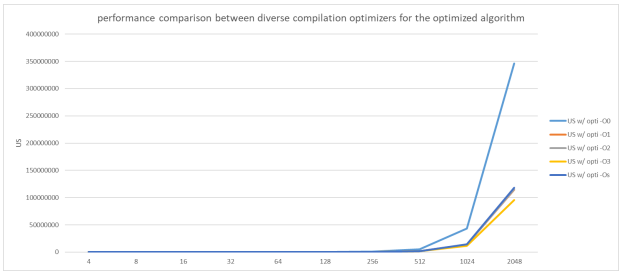


Fig. 28. Comparison between the naive and optimized code performances

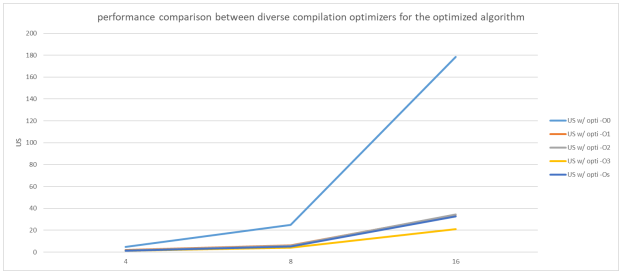


Fig. 29. Zoom for N=4, 8, 16

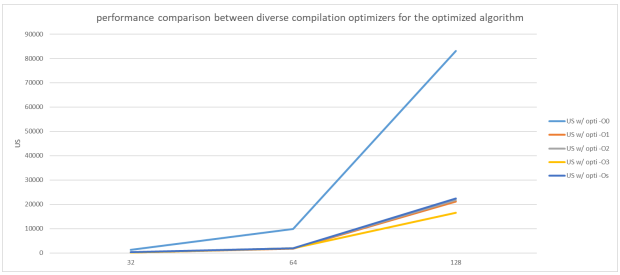


Fig. 30. Zoom for N=32, 64, 128

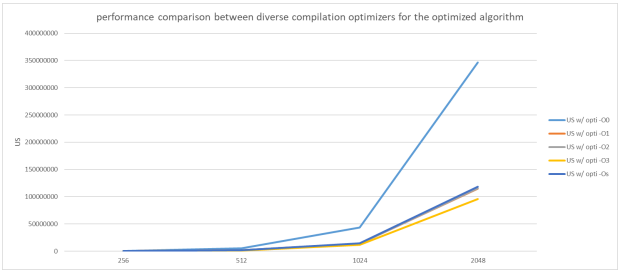


Fig. 31. Zoom for N=256, 512, 1024, 2048

G. Optimisation de la fréquence du compilateur

1) Description: Il est également possible d'aborder une autre manière d'optimiser l'arm9 : en faisant varier la fréquence du processeur. par défault, la fréquence du processeur est de 666MHz, valeur la plus optimisée possible. Nous allons dans cette partie étudier l'impact d'une diminution de cette valeur. Nous nous attendons à une augmentation des temps d'exécution, et donc à une baisse des performances ; mais le but va être de quantifier cette baisse de performance, pour étudier à quel point ce paramètre impacte les performances du processeur.
Nous allons diviser par 6 la fréquence du processeur, que nous instaurons à 111MHz.

Nous obtenons les résultats suivants :

N	US (µs)
4	28,93
8	190,64
16	1510,11
32	11634,76
64	92098,2
128	1097356,53
256	9404818,47
512	81589561,85
1024	714031658,5
2048	5817039951

2) Comparaison des résultats avec le code naïf: En chiffrant quantitativement les résultats obtenus avec un processeur de fréquence 6 fois moins grande, il est possible de comparer ces performances à celles du code naïf et de mettre en lumière l'importance du choix de la valeur de la fréquence du processeur :

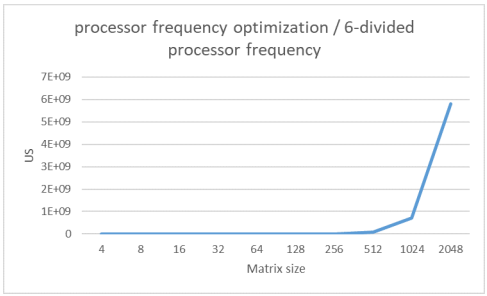


Fig. 32. performance optimization by dividing the processor frequency by 6

N	Taux de variation (%)
4	491,6
8	502,3
16	505,3
32	505,5
64	505,4
128	505,5
256	505,4
512	340,5
1024	252,5
2048	249,4

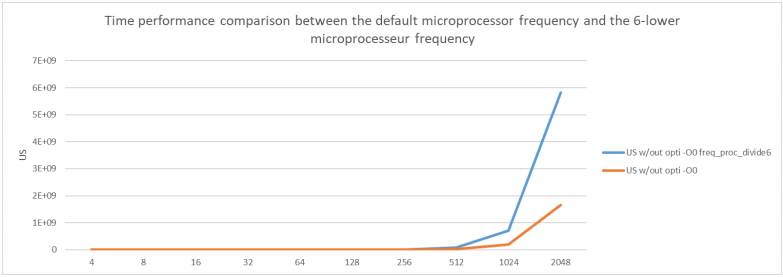


Fig. 33. performance comparison between the naive code and the 6-divided processor frequency optimization

On remarque les importantes valeurs des taux de variation : diviser la fréquence du processeur par 6 revient à augmenter de 500% les temps d'exécution, et donc de diminuer considérablement les performances de l'ARM9. L'importance de ce paramètre n'est pas négligeable, il s'agit d'un paramètre primordial à bien étudier et bien choisir pour optimiser le processeur.

NB : on remarque que plus la valeur de N est grande, et moins le choix de la fréquence du processeur a d'impact ; pour ce paramètre, manipuler de grandes tailles de matrices lisse l'influence du choix de la fréquence du processeur

H. Optimisation de l'utilisation du cache

Le cache est une fonctionnalité qui a pour but d'augmenter les performances temporelles d'un algorithme ; dans le cas d'accès à des chemins de mémoire répétés, stocker ces accès de données dans des tampons mémoires permettent d'y accéder plus rapidement notamment.

On distingue le *Data Cache* et le *Instruction Cache*.

De par cette analyse, nous nous attendons donc à une diminution des performances (et donc une augmentation des temps d'exécution) si ces caches sont désactivés. Nous allons également voir lequel de ces 2 caches a le plus d'impact en cas de désactivation

1) Remarques préliminaires: Avant d'aborder les résultats de l'optimisation par le cache, s'imposent quelques remarques quant aux fonctions que nous allons utiliser. Il en existe trois :

- Cache_Invalidate : ne change pas la mesure du temps d'exécution (cf. la sous section concernée)
- Cache_Disable : permet de désactiver le cache en question ; c'est la fonction que nous allons utiliser pour étudier l'impact du cache sur la mesure de performances
- Cache_Enable: permet d'activer le cache ; elle est activée par défaut (la mesure du temps d'exécution avec cette fonction est identique au cas où aucune fonction n'est indiquée ; cela a été vérifié par la mesure)

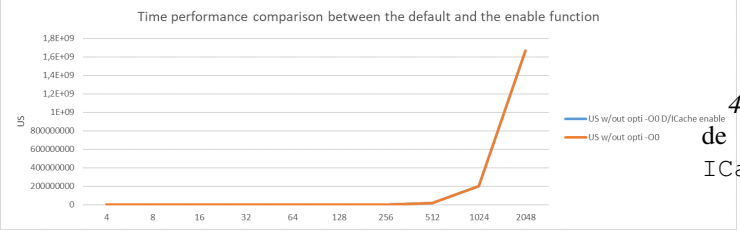


Fig. 34. Comparaison des performances avec utilisation de la fonction `enable` et par défaut - les performances sont identiques

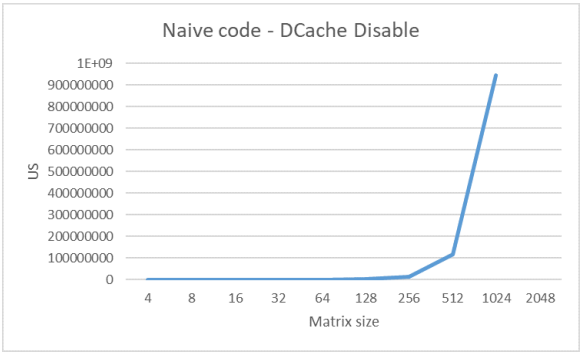
2) Cas de la fonction `Cache_Invalidate`: Comme dit précédemment, cette fonction n'affecte pas la mesure des temps d'exécution ; comme le prouve l'obtention des temps suivants :

N	US (µs) (DCache_Invalidate)	US (µs) (ICache_Invalidate)
4	5,06	5,04
8	31,97	31,94
16	253,34	253,33
32	2014,02	2013,99
64	15930,2	15928,24
128	181783,11	181769,42
256	1537808,24	1537835,89
512	18371476,34	18389108,22
1024	202192361,5	202234729,9
2048	1663167698	1662976043

Tout d'abord, on peut remarquer que les valeurs obtenues pour le DCache et le ICache sont semblables. De plus, les taux de variations avec les mesures du code naïf sans optimisation sont très faibles (de l'ordre de 0,1% à 4,8%) : ce n'est donc pas cette fonction que nous allons utiliser.

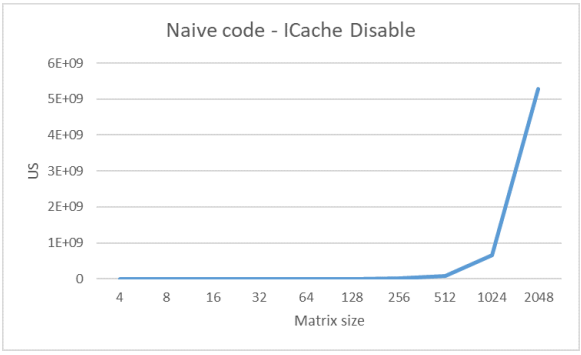
3) Influence du DCache: Nous allons étudier l'impact de la désactivation du DCache à l'aide de la fonction `DCache_Disable`.

N	US (µs)
4	64,57
8	474,74
16	3676,08
32	28622,29
64	228680,9
128	1838739,65
256	14783651,46
512	117287159,9
1024	944213556,8
2048	...



4) Influence du ICache: Nous allons étudier l'impact de la désactivation du ICache à l'aide de la fonction `ICache_Disable`.

N	US (µs)
4	47,56
8	336,16
16	2607,73
32	20378,39
64	161224,13
128	1290345,91
256	10305046,3
512	82711153,6
1024	662967185,7
2048	5288229432



5) Comparaison des performances avec le code naïf: Cas DCache_Disable

Lorsque le DCache est désactivé, nous obtenons les pertes en performance suivantes :

N	Taux de variation (%)
4	1220,4
8	1400,0
16	1373,4
32	1389,5
64	1403,2
128	914,6
256	851,7
512	533,2
1024	366,2
2048	...

Ainsi, retirer le DCache réduit considérablement les performances ; il s'agit donc d'un aspect non négligeable pour une bonne accélération matérielle. On remarque que plus la valeur de N augmente, et plus cette perte de performance est lissée néanmoins, sans pour autant devenir acceptable.

Cas ICache_Disable

Lorsque le DCache est désactivé, nous obtenons les pertes en performance suivantes :

N	Taux de variation (%)
4	872,6
8	962,1
16	945,2
32	960,5
64	959,8
128	612,0
256	563,4
512	346,5
1024	227,3
2048	217,6

Nous remarquons que comme pour le cas du DCache, plus la valeur de N augmente et moins le contraste de performances est grand entre un code où le ICache est activé et un code où il est désactivé. A remarquer également, les écarts de performance sont moins grands qu'en désactivant le DCache : ainsi, un DCache activé a un plus grand pouvoir d'accélération de performances qu'un ICache activé.

Mise en lien des 3 cas

Afin d'obtenir une comparaison plus visuelle, nous construisons les graphes suivants, qui permettent de corroborer ce qui a été dit ci-dessus : activer le DCache et le ICache permettent d'augmenter considérablement les performances temporelles, et le DCache a un plus grand pouvoir de performances que le ICache.

I. Conclusion

Ainsi, toutes ces analyses d'optimisations réalisées ont permis de voir le large spectre des possibles concernant l'optimisation de l'ARM9 (et bien plus encore). Quant au fait de déterminer la combinaison la plus performante, cela dépend entièrement de la taille de la matrice ; mais en considérant toutes les tailles possibles que l'on a étudiées, la meilleure combinaison est le code séquentiel optimisé avec l'optimisation du compilateur -O3, avec les I et D Cache activés et une fréquence de processeur maximale de 667MHz.

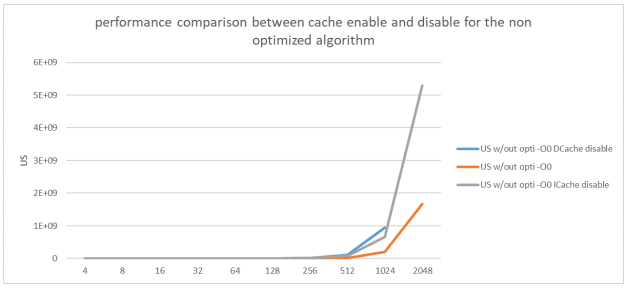


Fig. 35. Comparaison des performances pour les 3 cas

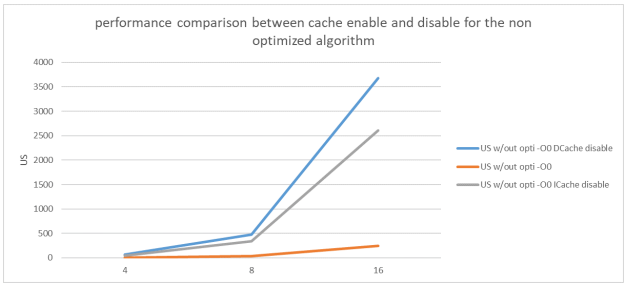


Fig. 36. Zoom pour N=4,8,16

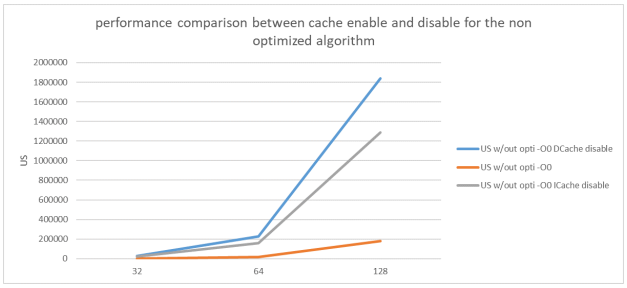


Fig. 37. Zoom pour N=32,64,128

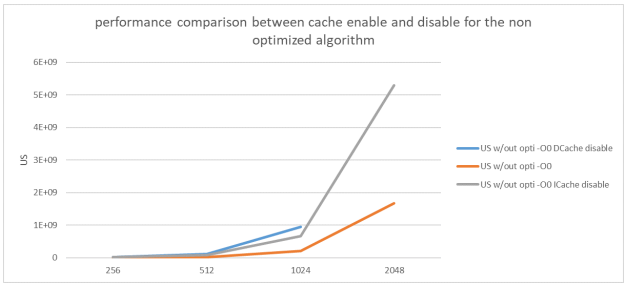


Fig. 38. Zoom pour N=256,512,1024,2048

N	Meilleure optimisation pour N	Temps US (µs)
4	Optimisé + -Os	1,11
8	Optimisé + -O3	4,13
16	Naïf + -O3	19,8
32	Naïf + -O3	228,13
64	Naïf + -O3	1740,99
128	Optimisé + -O3	16568
256	Optimisé + -O3	144248
512	Optimisé + -O3	1436060
1024	Optimisé + -O3	11800000
2048	Optimisé + -O3	95500000

IV. High Level Synthesis : IPs de multiplication matricielle

A. Le bloc C a accélérer

L'accélération matériel est faite sur l'algorithme naïf de la multiplication matricielle montré figure 39. Étant donné que chaque changement de la taille n des matrices $n \times n$ implique de faire une nouvelle IP, nous avons réalisé les IP pour $n = 4, 16, 64$ avec diverses niveaux d'optimisation ainsi que des interfaces en AXI4-Lite ou AXI4-Stream.

```
void matrixMultiplication1 (int A[n][n], int B[n][n], int C[n][n]){
  // calcul de C = A*B
    forLine:for(int i = 0; i < n; i++){
      forColl:for(int j = 0; j < n; j++){
          C[i][j] = 0;
          forMult:for(int k = 0; k < n; k++){
            C[i][j] += A[i][k] * B[k][j];
          }
        }
    }
}
```

Fig. 39. Algorithme de multiplication matricielle à accélérer

B. Directives d'optimisation des IPs en interface AXI4-Lite

Pour chaque taille de matrice n nous avons gardé une IP en version séquentielle simple non optimisé mais minimisant l'utilisation matériel, ainsi que notre meilleurs IPs en terme de performance c'est à dire en terme de minimisation de latence estimé par Vivado HLS. Ces deux IPs étant réalisé avec une interface AXI4-Lite pour le contrôle ainsi que pour les matrices en entrées.

- Cas n=4
  - une version basique sans directives
  - une version accéléré avec pipeline de la fonction entière et reshape total de A, B, C
- Cas n=16
  - une version basique sans directives
  - Les matrices sont trop grosses pour reprendre les directives du cas n=4. (utilisation matériel excessive). On effectue donc un pipeline de la boucle la plus extérieur et un reshape par bloc de A et B selon une dimension
- Cas n=64
  - une version basique sans directives
  - Là aussi les matrices sont trop grosses pour reprendre les directives du cas n=16. On effectue donc un pipeline de la boucle sur j et reshape par bloc de A et B selon une dimension

C. Directives d'optimisation des IPs en interface AXI4-Stream

De même que pour les IP en interface AXI4-Lite, pour chaque valeur de $n = 4, 16, 64$ nous avons concervé une IP basique sans optimisation, ainsi que notre IP la plus optimisé. Les directives propres à l'interface AXI4-Stream ont été ajoutées dans le code source et sont visible sur la figure 40. On constatera que le choix a été fait de n'utiliser qu'un port en entré pour charger les deux matrices A et B. Un autre choix possible aurait été de charger A et B simultanément via deux port afin d'augmenter la performance au prix d'une utilisation matérielle accrue.

- Cas n=4
  - une version basique sans directives
  - une version accéléré avec pipeline des lectures et écriture de A, B, et C ainsi qu'un pipeline de la boucle extérieur de la multiplication matricielle.
- Cas n=16
  - une version basique sans directives
  - Les matrices sont trop grosses pour reprendre toutes les directives du cas n=4. (utilisation matériel excessive). En particulier, le pipeline de la boucle extérieur de multiplication matricielle est remplacé par un pipeline sur la boucle immédiatememnt en dessous. Pour que la cible II = 1 du pipeline sur la multiplication matricielle soit respecté, il faut aussi insérer des directives de Reshape par blocs sur A et B
- Cas n=64
  - une version basique sans directives
  - Là aussi les matrices sont trop grosses pour reprendre les directives du cas n=16, il faut de nouveau les adapter.

```
#include <ap_axi_sdata.h>
#include <hls_stream.h>
#include "ap_int.h"
using namespace std;
#define SIZE 64
#define INPUT_SIZE SIZE*SIZE*2
#define OUTPUT_SIZE SIZE*SIZE
typedef ap_axiu<32,4,5,5> AXI_VALUE;

void matrixMultiplication1 (hls::stream<AXI_VALUE> &in_stream, hls::stream<AXI_VALUE> &out_stream){
#pragma HLS INTERFACE axis port=out_stream name=OUTPUT_STREAM
#pragma HLS INTERFACE axis port=in_stream name=INPUT_STREAM
#pragma HLS INTERFACE s_axilite port=return bundle=CONTROL_BUS
    int A[SIZE][SIZE];
    int B[SIZE][SIZE];
    int C[SIZE][SIZE];
    AXI_VALUE aValue;
    int k;
    // reconstruction A et B
    forLineA:for(int i = 0; i < SIZE; i++){
        forCollA:for(int j = 0; j < SIZE; j++){
            in_stream.read(aValue);
            union { unsigned int ival; int oval; } converter;
            converter.ival = aValue.data;
            A[i][j] = converter.oval;
        }
    }
    forLineB:for(int i = 0; i < SIZE; i++){
        forCollB:for(int j = 0; j < SIZE; j++){
            in_stream.read(aValue);
            union { unsigned int ival; int oval; } converter;
            converter.ival = aValue.data;
            B[i][j] = converter.oval;
        }
    }
    // calcul de C = A*B
    forLineC:for(int i = 0; i < SIZE; i++){
        forCollC:for(int j = 0; j < SIZE; j++){
            C[i][j] = 0;
            forMult:for(int k = 0; k < SIZE; k++){
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    // sortie C en stream
    forLineSortie:for(int i = 0; i < SIZE; i++){
        forCollSortie:for(int j = 0; j < SIZE; j++){
            union { unsigned int oval; int ival; } converter;
            converter.ival = C[i][j];
            aValue.data = converter.oval;
            aValue.last = (i*SIZE+j==OUTPUT_SIZE-1)? 1 : 0;
            aValue.strb = -1;
            aValue.keep = 15;
            aValue.user = 0;
            aValue.id = 0;
            aValue.dest = 0;
            out_stream.write(aValue);
        }
    }
}
```

Fig. 40. directive et code lié à l'interface en AXI4-Stream

D. Évaluation de Performance/Surface

Dans cette section, nous réalisons une étude comparé de la latence ainsi que de l'utilisation matériel des différentes IPs décrites aux sections précédentes. La figure 41 représentes les latences de chaque IP avec des axes logarithmiques afin d'éviter l'écrasement des courbes. On remarquera que la version la plus optimisé avec interface AXI4-Lite permet de gagner un facteur 100 sur la réduction de la latence. On remarquera aussi que étrangement, les IP avec interface AXI4-Stream sont annoncées plus lente par Vivado HLS que les IP en AXI4-Lite. Notre hypothèse est que Vivado HLS ne prend pas en compte le temps de transmission des données pour évaluer l'IP et donc que le temps estimé est optimiste par rapport au temps réel, en particulier pour l'interface en AXI4-Lite. De plus, dans le cas des interfaces en AXI4-Stream, les étapes de reconstruction des matrices A et B streamé ainsi que l'écriture de C dans la variable streamé en sortie se rajoutent et prennent du temps.

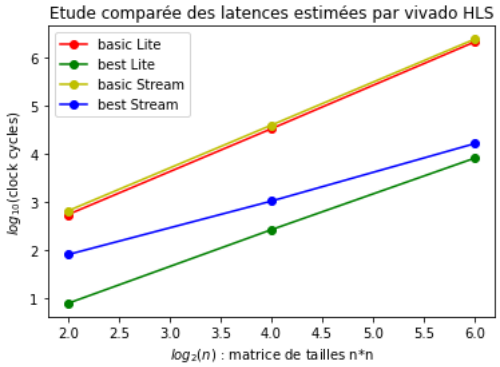


Fig. 41. Étude comparée des latences estimées par Vivado HLS et réprésentées sur une échelle logarithmique

Il est ensuite intéressant de quantifier l'utilisation matériel pour chacune des solutions. les figure 42, 43, 44, 45 représente respectivement le pourcentage d'utilisation matériel par l'IP des DSP48E, BRAM 18K, FF et LUT par rapport au total disponible sur la zedBoard. D'une manière générale, on remarquera que, comme attendu, les IPs otpimisées utilisent plus de matériel que les IPs basiques. En particulier les DSP48E, qui permettent de réaliser les opération arithmétiques sont très utilisés. On remarque aussi que les BRAM 18K, qui sont les blocs correspondant à la mémoire, sont, de manière prévisible, plus utilisé pour les grandes matrices.

E. Detail des performance pour les IP avec interface en AXI4-Stream

On s'interesse ici de savoir ce qui prend le plus de temps parmi les différentes opérations réalisés dans le cadre de l'IP avec interface en AXI4-Stream. Comme on pourra le remarquer sur la figure 51 montrant le détails de latence pour l'IP optimisé dans le cas $n = 64$, la latence du calcul matricielle ne compte que pour un quart de la latence total. En effet, les trois quarts restant proviennent de la lecture et de l'écriture des matrices en entré et sortie du stream. On en déduit donc qu'une piste d'amélioration de la performance

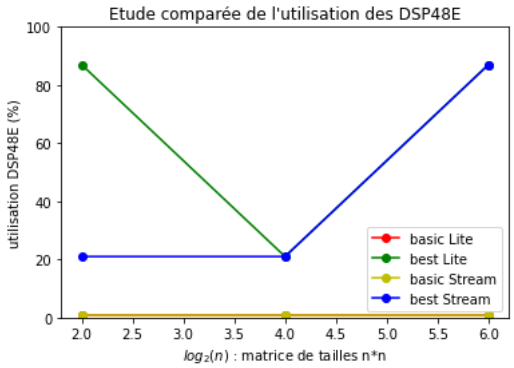


Fig. 42. Étude comparée de l'utilisation des DSP48E de l'IP en pourcentage du total disponible sur la ZedBoard

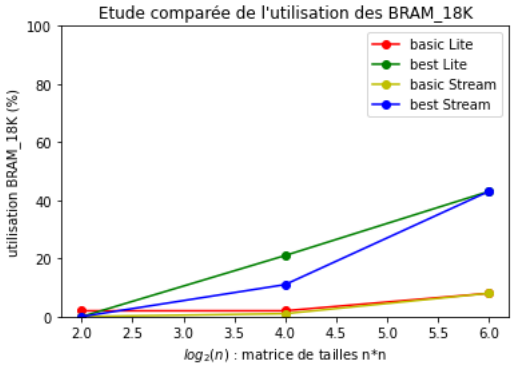


Fig. 43. Étude comparée de l'utilisation des BRAM 18K de l'IP en pourcentage du total disponible sur la ZedBoard

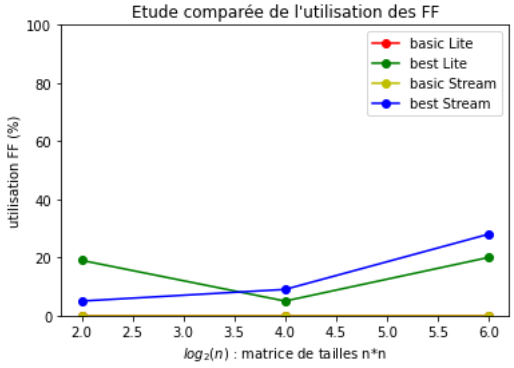


Fig. 44. Étude comparée de l'utilisation des FF de l'IP en pourcentage du total disponible sur la ZedBoard

serait de retravailler les interfaces. Notamment avec l'ajout d'un deuxième port d'entrer pour pouvoir lire A et B de manière simultanée.

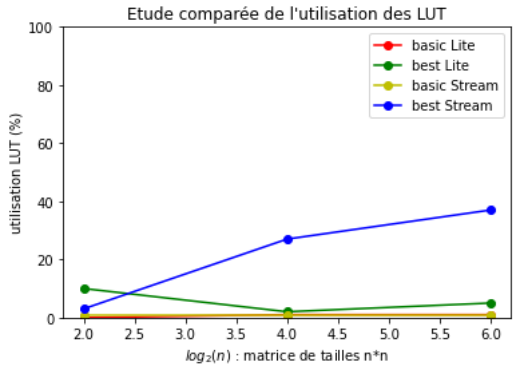
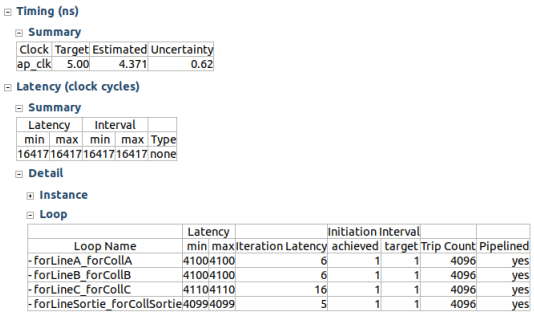


Fig. 45. Étude comparée de l'utilisation des LUT de l'IP en pourcentage du total disponible sur la ZedBoard



Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	5.00	4.371	0.62

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
16417	16417	16417	16417	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- forLineA_forCollA	41004100		6	1	1	4096	yes
- forLineB_forCollB	41004100		6	1	1	4096	yes
- forLineC_forCollC	41104110		16	1	1	4096	yes
- forLineSortie_forCollSortie	40994099		5	1	1	4096	yes

Fig. 46. Détails de la latences estimé par Vivado HLS de l'IP optimisée avec interface en AXI4-Stream pour des matrice de tailles 64×64. Le Calcul matricielle n'est responsable que d'un quart de la latence total

V. Intégration des IP sur la ZedBoard: Vivado et SDK

On va s'interreser dans cette section à intégrer les IPs développés en Vivado HLS sur la Zedboard et on va comparer les performances obtenues avec ces IPs, ARM9 et le PC.

A. Intégration des ip avec interface en AXI4-Lite

6 IP avec des interfaces en AXI4-Lite, correspondant aux tailles de matrices $n \times n$ avec $n = 4, 16, 64$ avec pour chaque taille une version basique et une version optimisé ont été réalisé via HLS et ont donc du être intégré.

1) Intégration coté Vivado: La première étape consiste à intégrer l'ip HLS de multiplication matricielle dans le bloc diagramme de Vivado comme montré figure 47

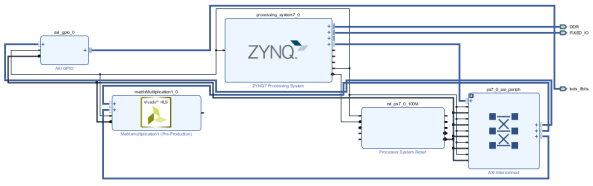


Fig. 47. Block diagrame pour une ip contenant le processing systeme et l'ip de multiplication en interface AXI4-Lite

2) Intégration coté SDK des IP sans directive d'optimisation: Il reste alors à adapter le code C pour que le processeur ARM9 interagisse avec l'IP. L'intégration s'est bien passé pour les IP basique. Les temps d'exécution obtenus pour la multiplication matricielle étaient alors de $1317\,\mu$s, $1548\,\mu$s, $5356\,\mu$s. Ce n'est pas très rapide comme attendu car l'IP est sans directive d'optimisation.

3) Intégration coté SDK des IP avec directive d'optimisation: L'intégration coté SDK s'est trouvé être ardue avec les IPs optimisées notamment à cause de la directive de reshape des matrice A,B et C causant un changement au niveau des interfaces de l'IP. Par exemple on peut voir sur la figure 48 le type associé à la matrice A et à fournir en entré de l'IP pour la multiplication matricielle 4×4. Cette interface a été obtenue après avoir utilisé la directive de reshape complet (option de dimension pris égale à 0). Il ne semble pas y avoir de documentation pour interpréter les valeurs à mettre dans chaque champs de cette structure à 16 éléments. Cependant après plusieurs test nous avons déterminé qu'ils correspondaient aux coefficients de la matrice stocké en colonne (et non en ligne comme supposé initialement). C'est à dire que les champs word_0, word_1, word_2, word_3 contiennent les coefficients de la première colonne de la matrice A. De même pour B et C. Nous avons donc réussit à intégrer l'IP et obtenue un temps d'exécution de 3.19 μs. Ce qui est bien meilleurs comme attendu, que le temps obtenue avec l'IP basique.

```
typedef struct {
    u32 word_0;
    u32 word_1;
    u32 word_2;
    u32 word_3;
    u32 word_4;
    u32 word_5;
    u32 word_6;
    u32 word_7;
    u32 word_8;
    u32 word_9;
    u32 word_10;
    u32 word_11;
    u32 word_12;
    u32 word_13;
    u32 word_14;
    u32 word_15;
} XMatrixmultiplication1_A;
```

Fig. 48. type de l'argument associé à A en entré de l'IP de multiplication matricielle avec directive de Reshape pour les matrices de taille 4×4 et 16×16

Puis nous avons tenté d'intégrer l'IP optimisé pour la multiplication de matrices de tailles 16×16. Nous avons alors découvert que le type des variables correspondant à A et B à l'interface de l'IP était le même que celui montré pour la multiplication matricielle 4×4 montré figure 48. Cette observation, est surprenante au premier abord, étant donné que les matrices 16×16 contiennent 256 éléments et non 16 comme semble l'indiquer le nombre de champ de cette structure. Cependant la directive de Reshape est différente. Pour $n = 16$, une directive par bloc selon une dimension (dim 2 pour A et dim 1 pour B) a été utilisé avec un facteur de 16 pour l'option des blocs. On devine donc que chaque champ de cette structure doit contenir 16 élément, probablement sous la forme d'un tableau. Cependant il n'est pas clair de savoir

si ces tableaux correspondent à des lignes ou à des colonnes. Il n'est pas non plus évident de savoir à quelles lignes ou colonnes correspondent chacun des champs. Les fonctions d'accès pour écrire et lire les variables aux interfaces étaient aussi différentes. Face à l'absence de documentation ainsi qu'à cause du planning serré, nous n'avons pas eu le temps de faire nos propre test pour deviner le fonctionnement des fonctions Xilinx associé à cette interface et nous n'avons donc pas réalisé l'intégration pour les IP optimisées avec interface AXI4-Lite pour la multiplication de matrices 16×16 et 64×64.

B. Intégration des ip avec interface en AXI4-Stream

Dans cette partie, on a utilisé un AXI DMA connecté avec les IPs développées en HLS! On a utilsé aussi un AXI Timer afin de déterminer le nombre de cycles nécessaires pour l'éxécution du code :

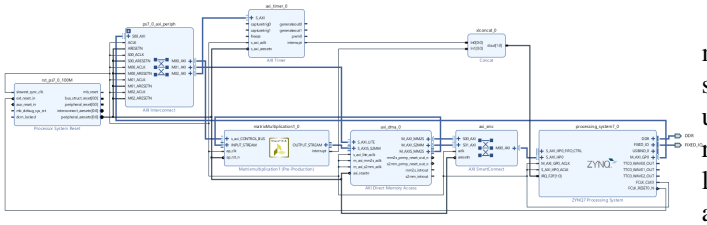


Fig. 49. Bloc Design avec AXI DMA et AXI Timer

Résultat : Temps en micro-secondes

Dimension des matrices	Version basique	Version pipeline
4	15.83	11.40
16	403.42	19.94
64	24039.44	32.697

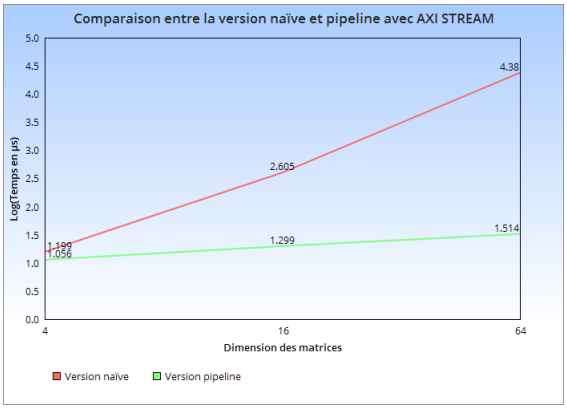


Fig. 50. Comparaison entre la version naïve et la version pipeline

Résultat : Nombre de cycles

Dimension des matrices	Version basique	Version pipeline
4	1479	1027
16	40182	1881
64	2401403	17233

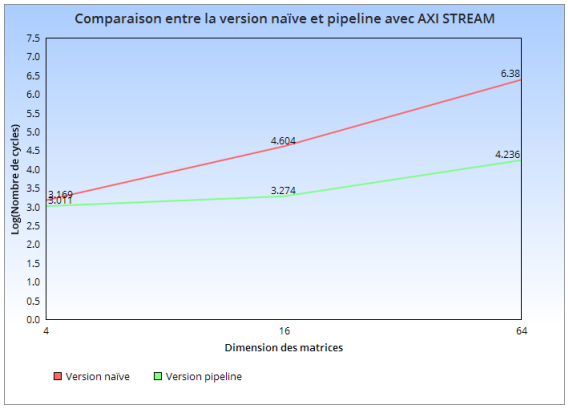


Fig. 51. Comparaison entre la version naïve et la version pipeline

VI. Etude de performance comparé entre les solutions PC, ARM9 et FPGA

On compare ici les meilleurs temps d'exécution pour la multiplication de matrice de taille $n \times n$, $n = 4, 16, 64$ obtenue sur le PC, sur le processeur ARM9 ainsi que sur le FPGA utilisants les IPs HLS optimisées. Pour la solution PC on rappelle que les meilleurs temps ont été obtenue à l'aide la librairie OpenMP tandis que les IPs retenus sont celle avec interface AXI4-Stream et les directives d'optimisation. La figure 52 affiche les temps d'exécution sur des échelles logarithmiques

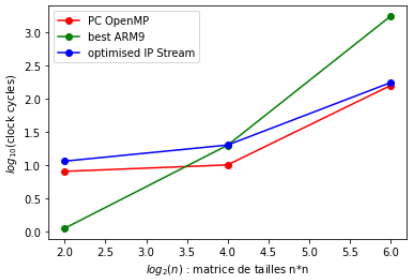


Fig. 52. comparaison de performance pour les meilleurs solutions sur PC, ARM9 et FPGA

On remarque que le PC avec les directives OpenMP reste meilleurs que l'accélération matériel FPGA même si on peut remarquer que les deux courbes se rejoignent pour $n = 64$. Il est très probable que le calcul sur FPGA soit plus performant sur de plus grandes matrices. Quant à la solution sur ARM9 elle moins bonne que les deux autres pour de grande matrices.

VII. Conclusion

Nous avons vu lors de se projet l'intérêt de l'accélération matériel sur FPGA permettant de contrôler finement la performance souhaité ainsi que la quantité de hardware à utiliser. Nous avons aussi constaté la puissance de l'outil Vivado pour la création de tels accélérateurs. Cependant malgré la qualité de ces outils, quelques difficulté persistent et il devient rapidement nécessaire d'obtenir une bonne maîtrise de ces logiciels ainsi que d'avoir des connaissances en électroniques.