# Laboratoire 2

## Accélérateurs Zynq SoC pour la multiplication conçus avec Vivado HLS et intégrés dans un système logiciel/matériel

Les objectifs de ce laboratoire sont :

1) Réaliser la validation et l'exploration d'architectures à l'aide de HLS Vivado pour un coprocesseur de multiplication de matrices s'exécutant sur FPGA float, half-precision float et short (16 bits).
2) Ajouter à ces coprocesseurs une interface avec le DMA (Direct Memory Access) et connecter le tout à une DDR via un bus rapide (HP)
3) Exporter le coprocesseur dans la librairie de Vivado sous forme d'un IP
4) Concevoir avec Vivado le design du système incluant le coprocesseur, le DMA et le ARM et autre périphérique
5) Programmer le DMA à partir d'un processeur ARM
6) Comparer différentes architectures d'accélérateur avec une version identique du calcul matriciel s'exécutant sur le ARM, ce qui représente une première expérience de codesign logiciel/matériel sur FPGA.
7) En résumé de 1) à 6), bien comprendre l'approche de conception de bas vers le haut utilisé proposé par les outils de Vivado et Vivado HLS

Évidemment, vous remarquerez qu'il y a plusieurs activités au menu, mais pour accélérer le travail plusieurs étapes sont déjà réalisées, de sorte qu'il ne vous reste qu'à faire l'assemblage. En particulier, vous verrez que l'on aura recours à des fichiers script pour accélérer la conception des de certains étapes, ce qui est aussi une pratique industrielle courante.

Le travail se fera en 4 étapes. L'étape 1 se fera dans la partie 1 alors que les étapes 2 à 4 se feront dans la partie 2.

## Étape 1 Synthèse HLS et optimisation

Le code de ce calcul de multiplication matricielle est donné à la figure 0.

Dans le répertoire *architecture_exploration*, nous allons faire l'exploration architecturale sur 4 **architectures** qui se trouvent respectivement dans les répertoires de code de départ:

1) *hls_architecture1/architecture/exploration* et
2) *hls_architecture2/architecture/exploration*
3) *hls_architecture3/architecture/exploration*
4) *hls_architecture4/architecture/exploration*

```
void matrixmul(int A[N][M], int B[M][P], int AB[N][P]) {
    #pragma HLS ARRAY_RESHAPE variable=A complete dim=2
    #pragma HLS ARRAY_RESHAPE variable=B complete dim=1
    /* for each row and column of AB */
    row: for(int i = 0; i < N; ++i) {
        col: for(int j = 0; j < P; ++j) {
            #pragma HLS PIPELINE II=1
            /* compute (AB)i,j */
            int ABij = 0;
            product: for(int k = 0; k < M; ++k) {
                ABij += A[i][k] * B[k][j];
            }
            AB[i][j] = ABij;
        }
    }
}
```

**Figure 0 Multiplication matricielle**

Dans notre cas aura *L1*, *L2* et *L3* à la place de *row*, *col* et *product* et également les matrices seront carrées (N=M=P). De plus on utilisera le ARRAY_PARTITION plutôt que ARRAY_RESHAPE.

### Objectifs de cette 1ère étape

Dans cette première étape, l'exploration no 1 peut être vue comme un tutorial. Elle est très similaire au matériel présenté en classe dans le bloc2, mais au lieu de travailler sur des matrices de type float (32 bits) de 4x4, ce sera des float 42x42. Puis, à l'exploration no 2, vous ferez un travail similaire à l'exploration 1 mais avec des matrices de type half float (16 bits). À l'exploration no 3, vous verrez une manière différente du DSP (p. 13, bloc 2, chap. 3) pour représenter le half float. Plus précisément, nous verrons qu'il est possible de forcer la non-utilisation des DSP en remplaçant par des LUT et FF. Finalement, à l'exploration no 4, vous travaillerez avec des matrices entières 16 bits avec et sans DSP.

De ces 4 architectures, nous travaillerons avec des DSP. De ces explorations nous sélectionnerons certaines **solutions** que vous allez par la suite implémenter sur votre carte. En effet, une fois qu'on aura déterminé les bons paramètres, ces solutions seront appliquées à l'**étape 2** avec en plus, l'ajout de l'interface (*stream*) pour fonctionner avec le DMA. Ce sera également là que nous générerons un certain nombre de IPs par solution qui pourra ensuite être instancié à l'**étape 3** dans Vivado. Finalement à partir de Vivado, nous allons générer le bitsream pour ensuite utiliser SDK pour effectuer un test de performance. Cette dernière partie avec SDK représente l'**étape 4**.

### 1) *Exploration architecturale no 1* pour une multiplication de matrices de type float (32 bits) avec DSP (très similaire è l'exemple du bloc 2 du chap. 3)

On s'intéresse ici au calcul float 32 bits, un des calculs les plus demandant pour un FPGA. Le code de départ se trouve dans *hls_architecture_1/architecture_exploration*.

- **Réalisez la solution 1.1 (inspirez-vous de la solution 1 du bloc 2 (page 29) sauf que dans ce dernier c'est pour une 4x4 plutôt que 42x42)**

Simulez votre code avec l'aide du testbench pour vous assurez que la fonctionalité est bonne. Si c'est le cas, vous devriez obtenir ce qui suit dans la console :

```
Matrixes identical ... Test successful!
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
Finished C simulation.
```

Ensuite, trouvez une solution qui minimise l'utilisation des ressources du FPGA pour des matrices de dimension 42 x 42 (voir lignes 9 et 10 dans *mult.h*).

- **Réalisez la solution 1.2**

Toujours pour des matrices de dimension 42x42 et à l'aide du pragma *hls pipeline*, **maximisez** l'utilisation des ressources du FPGA tout en offrant une latence **minimale**. Ici, maximisez les ressources veut dire environ 95% de ressources pour un ou plusieurs parmi BRAM, DSP, FF et LUT.
***Suggestion : inspirez-vous de la solution 4 du bloc 2 (p. 37)***

- **Réalisez la solution 1.3**

Toujours pour des matrices de dimension 42x42 et à l'aide du pragma *hls pipeline*, **coupez de moitié** l'utilisation des ressources du FPGA tout en offrant une latence **minimale**. Ici, coupez de moitié les ressources veut dire inférieur à 45% des ressources pour les ressources BRAM, DSP, FF et LUT.
***Suggestion : Faire varier II (2, 3, etc.) avec array_partition option complete ou soit mettre II=1 et faire varier les block factor de array partition (ça veut dire couper de moitié le nombre de DSP et en arrondissant à l'entier supérieur lorsque non divisible par 2 (p. 38)***

- **Réalisez la solution 1.4**

Déterminez la dimension maximale (DIM et SIZE dans *mmult.h*) pour un calcul de produit matriciel sur des float, avec un pragma *hls pipeline* situé juste au-dessus de L2, qui **maximise** l'utilisation des ressources du FPGA tout en offrant une latence **minimale**, évidemment tout en respectant le 95% (comme à la solution 1.2).

***Suggestion : inspirez-vous de la solution 7 du bloc 2 (pp. 50 à 52). Ici, comme 42x42 va faire explosez les ressources disponibles sur votre FPGA et qu'une 4x4 n'utilise pas complètement les***

***ressources (p. 52 du bloc 2), trouvez la bonne dimension qui utilisera autour de 95% entre 4x4 et 42x42***

Dans le rapport, vous devrez expliquer votre démarche. Vous devrez donner un tableau du sommaire de la latence (Figure 1) et du détail des boucles (Figure 2), ainsi que le sommaire des ressources utilisées (Figure 3) pour les solutions 1.1 à 1.4.

**Latency (clock cycles)**

**Summary**

| Latency | | Interval | | |
|---|---|---|---|---|
| min | max | min | max | Type |
| | | | | |

**Figure 1. Latence pour la multiplication complète**

**Detail**

**Instance**

**Loop**

| | Latency | | | Initiation Interval | | | |
|---|---|---|---|---|---|---|---|
| Loop Name | min | max | Iteration Latency | achieved | target | Trip Count | Pipelined |
| | | | | | | | |

**Figure 2. Latence au niveau des boucles (il peut y avoir plus qu'une ligne)**

**Utilization Estimates**

**Summary**

| Name | BRAM_18K | DSP48E | FF | LUT |
|---|---|---|---|---|
| DSP | | | | |
| Expression | | | | |
| FIFO | | | | |
| Instance | | | | |
| Memory | | | | |
| Multiplexer | | | | |
| Register | | | | |
| Total | | | | |
| Available | 280 | 220 | 106400 | 53200 |
| Utilization (%) | | | | |

**Figure 3 Ressource requises**

Questions de synthèse pour les solutions 1.1 à 1.4 :

Q1.1) Que fait le testbench?
Q1.2) Combien de ressources DSP sont requises pour la solution 1.1
Q1.3) Combien de ressources DSP sont requises pour la solution 1.2.

Q1.4) Combien de ressources DSP sont requises pour la solution 1.3.
Q1.5) Combien de ressources DSP sont requises pour la solution 1.4.
Q1.6) Comment déterminer le nombre de ressources DSP requises **par multiplication/accumulation** avec un float?
Q1.7) Comparez la complexité (avec notation O) en temps d'exécution et ressources DPS entre les solutions 1.1, 1.2, 1.3 et 1.4. Expliquez.
Q1.8) À la solution 1.2, pour le pragma HLS ARRAY_PARTITION, proposez une valeur de *block factor* plutôt qu'un *complete* qui donnera un résultat similaire au niveau débit, temps d'exécution et ressources. Justifiez.
Q1.9) À la solution 1.3, pour le pragma HLS ARRAY_PARTITION, proposez une valeur de block factor plutôt qu'un *complete* qui donnera un résultat similaire au niveau débit, temps d'exécution et ressources. Justifiez.
Q1.10) Combien de lignes de codes VHDL demande l'implémentation de la solution 1.2?

### 2) *Exploration architecturale no 2 pour une multiplication de matrices de type half float (16 bits) avec DSP*

En informatique, la demi-précision (parfois appelée FP16) est un format binaire informatique à virgule flottante qui occupe 16 bits (2 octets au lieu de 4 octets tel le float) dans la mémoire de l'ordinateur. Il est destiné au stockage de valeurs à virgule flottante dans des applications où une plus grande précision n'est pas indispensable, en particulier les images d'infographie et les réseaux de neurones.

Le code de départ se trouve dans *hls_architecture_2/architecture_exploration*

- **Réalisez la solution 2.1**

Trouvez une solution qui minimise l'utilisation des ressources du FPGA pour des matrices de dimension 42 x 42 (voir lignes 9 et 10 dans mult.h).

- **Réalisez la solution 2.2**

Déterminez si la dimension maximale (DIM dans *mmult.h*) qui maximise l'utilisation des ressources du FPGA tout en offrant une latence minimale est toujours 42 (solution 1.1)? Si non déterminez cette nouvelle valeur de DIM. Suggestion : procédez par recherche binaire et respectez le 95% (comme à la solution 1.2). Bien Justifiez la valeur finale obtenue de DIM.

- **Réalisez la solution 2.3**

Pour des matrices de dimension trouvez en 2.2 et à l'aide du pragma *hls pipeline*, **coupez de moitié** l'utilisation des ressources du FPGA tout en offrant une latence **minimale**. Ici, coupez de moitié les ressources veut dire aux environs de 45% des ressources pour un ou plusieurs parmi BRAM, DSP, FF et LUT. Vous pouvez faire varier II ou utilisez des block factor (comme à la Q1.9) ou encore les deux à la fois.

*Tous comme pour la solution 1, donner un tableau du sommaire de la latence (Figure 1) et du détail des boucles (Figure 2), ainsi que le sommaire des ressources utilisées (Figure 3) pour les solutions 2.1 à 2.3.*

Q2.1) Combien de ressources DSP sont requises pour la solution 2.1?
Q2.2) Combien de ressources DSP sont requises pour la solution 2.2?
Q2.3) Combien de ressources DSP sont requises pour la solution 2.3?
Q2.4) Combien de ressources DSP requises **par multiplication/accumulation** avec un *half float*?

### 3) *Exploration architecturale no 3 pour une multiplication de matrices de type half float (16 bits) sans DSP*

Le code de départ se trouve dans *hls_architecture_3/architecture_exploration*. Remarque importante : par rapport aux explorations no 1 et no 2, ici on a modifié le code des 3 boucles par celui de la figure 4.

```
mat_type temp;
L1:for (int ia = 0; ia < DIM; ++ia)
      L2:for (int ib = 0; ib < DIM; ++ib)
            {
                  mat_type sum = 0;
                  L3:for (int id = 0; id < DIM; ++id){
                        temp = a[ia][id] * b[id][ib];
                        sum = sum + temp;
                  }
                  out[ia][ib] = sum;
            };

            return;
```

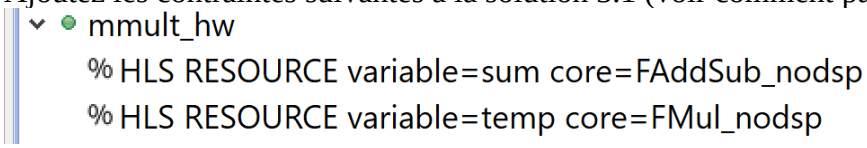
**Figure 4**

- **Réalisez la solution 3.1**

  Ajoutez les pragmas pour avoir la solution 2.2. Mais avant inclure dans *mult.h* la dimension (DIM) trouvée en 2.2. Également, simulez pour vous assurez du bon fonctionnement, vu les changements de code de la figure 4. Une fois ceci complété, synthétisez.

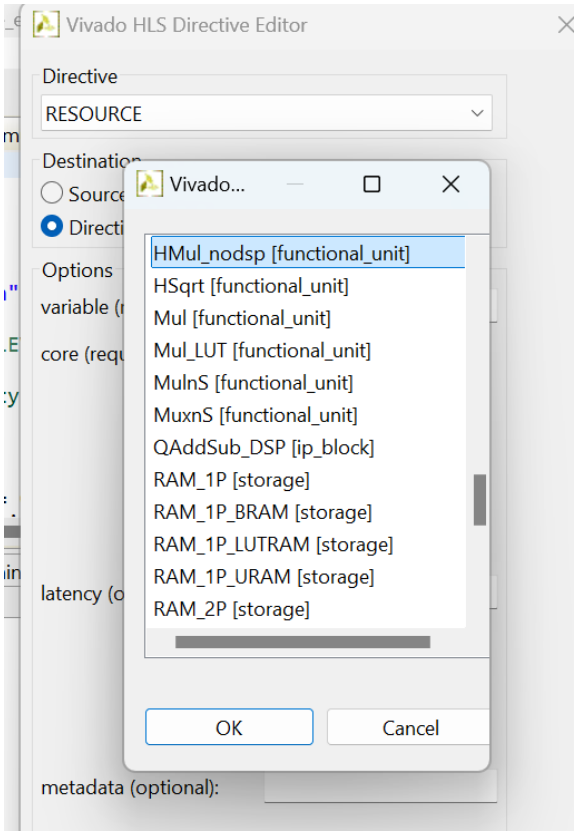
- **Réalisez la solution 3.2**

  Ajoutez les contraintes suivantes à la solution 3.1 (voir comment page suivante):

  et resynthétisez.

Augmentez ou diminuiez DIM pour exploiter au maximum 95% de ressources pour un ou plusieurs parmi BRAM, DSP, FF et LUT.




- **Réalisez la solution 3.3**

À partir de la solution 3.2, coupez de moitié le débit en mettant II=2 et aussi en mettant le bon block factor sur le *array_partition* des variables a et b.

*Tous comme pour la solution 1, donner un tableau du sommaire de la latence (Figure 1) et du détail des boucles (Figure 2), ainsi que le sommaire des ressources utilisées (Figure 3) pour les solutions 2.1 à 2.3.*

Q3.1) Expliquez le rôle des pragmas de ressources utilisées à la solution 3.2
Q3.2) Justifiez l'utilisation du code de la figure 4 pour les pragmas de ressource utilisé à la solution 3.2
*Q3.3)* Y a-t-il ici un avantage d'utiliser aucun dsp? Justifiez.
*Q3.4)* Contrairement aux solution 1 et 2 (sections 1 et 2 resp. plus haut) est-ce que ici II=2 implique la moitié des ressources? Justifiez.

## *Exploration architecturale no 4 pour une multiplication de matrices de type short (16 bits)*

Le code de départ se trouve dans *hls_architecture_4/architecture_exploration*

Le 8 et 16 bits sont souvent utilisés dans les réseaux de neurones. En effet, on entraine le réseau avec des floats 32 bits sur GPU pour le calcul des poids et lorsqu'on souhaite embarquer ce réseau de neurones dans un FPGA (par exemple dans une voiture ou encore un satellite), on quantifie sur 16 ou 8 bits (et même parfois sur 4 bits) sans trop perdre de précision. Si on multiplie 16 bits par 16 bits, on aura besoin de mettre le résultat sur 32 bits. Vous devriez donc observer un gain en ressource.

- **Réalisez la solution 4.1**

Trouvez une solution qui minimise l'utilisation des ressources du FPGA pour des matrices de dimension 42 x 42.

- **Réalisez la solution 4.2**

Déterminez si la dimension maximale (DIM dans *mmult.h*) qui maximise l'utilisation des ressources du FPGA tout en offrant une latence minimale est toujours 42 (solution 1.1)? Si non déterminez cette nouvelle valeur de DIM. Suggestion : procédez par recherche binaire et respectez le 95% (comme à la solution 1.2). Bien Justifiez la valeur finale obtenue de DIM.

- **Réalisez la solution 4.3**

À partir de la solution 4.2, coupez de moitié le débit en mettant II=2 et aussi en mettant le bon block factor sur le *array_partition* des variables a et b.

- **Réalisez la solution 4.2.1**

Refaire la solution 4.2, mais en y ajoutant les pragma ressources :

```
mmult_hw
    % HLS RESOURCE variable=sum core=AddSubnS
    % HLS RESOURCE variable=temp core=Mul_LUT
```

Déterminez si la dimension maximale (DIM dans *mmult.h*) qui maximise l'utilisation des ressources du FPGA tout en offrant une latence minimale est toujours celle trouvée en 4.2? Si non déterminez cette nouvelle valeur de DIM.

- **Réalisez la solution 4.3.1**

Refaire la solution 4.3, mais en y ajoutant les pragma ressources suivants

```
mmult_hw
    % HLS RESOURCE variable=sum core=AddSubnS
    % HLS RESOURCE variable=temp core=Mul_LUT
```

et en utilisant le DIM trouvez en 4.2.1.

*Tous comme pour la solution 1, donner un tableau du sommaire de la latence (Figure 1) et du détail des boucles (Figure 2), ainsi que le sommaire des ressources utilisées (Figure 3) pour les solutions 2.1 à 2.3.*

Q4.1) Expliquez le rôle des pragmas de ressources utilisées aux section 4.2.1 et 4.3.1
Q4.2) Y a-t-il ici un avantage d'utiliser aucun dsp? Justifiez.
Q4.3) Contrairement aux solution 1 et 2 (sections 1 et 2 resp. plus haut) est-ce que ici II=2 implique la moitié des ressources? Justifiez.

N.B. La réponse aux questions de ces 4 manipulations ne sera qu'à remettre avec le rapport final du lab 2 début décembre. Il n'y a donc pas de rapport à remettre dans 2 semaines.

## Étape 2 Ajout des interfaces et productions de 3 IPs

On ne pourra faire l'implémentation de toutes les solutions explorées à l'étape 1 car nous allons manquer de temps… Le but final (étape 4) sera de comparer un certain nombre de ces solutions avec des équivalences en logiciel sur le ARM Cortex A9 de votre carte PYNQ. En plus de faire cette comparaison d'accélération entre le logiciel et le matériel, nous allons aussi comparer entre eux les solutions afin de voir l'impact d'utiliser un type de donnée plutôt qu'un autre.

| **Synthèse** | **Nom de l'accélérateur** | **À partir de quelle solution** | **Dans quel répertoire du code de départ appliqué les optimisations de la solution** |
|---|---|---|---|
| No 1 | hls_accel | Solution 2.2<br>DIM = 54 (II=1) | hls_architecture2/impl_sol2_2 |
| No 2 | hls_accel | Solution 3.3<br>DIM = 54 (II=2)<br>(utilisez le bon blocking factor) | hls_architecture3/impl_sol3_3 |
| No 3 | hls_accel | Solution 4.2<br>DIM = 200<br>(II=1)<br>(utilisez le bon blocking factor) | hls_architecture4/impl_sol4_2 |

**Tableau 1 Notez qu'on peut garder le même nom d'accélérateur (hls_accel) puisqu'à l'étape 3 nous allons créer 3 projets Vivado**

Pour chaque solution du tableau 1, nous allons appliquer à la top function *HLS_accel* (situé dans les répertoires ***implementation*** du code de départ) les mêmes dimensions et optimisations faites dans l'étape 1 (et résumé au tableau 1) sur *mmult_hw* tout en lui ajoutant aussi l'interface (*AXI stream*) pour ainsi fonctionner avec le DMA. J'expliquerai en classe le concept de AXI stream et de DMA. C'est également ici que nous allons générer un IP qui sera par la suite instancié dans Vivado à la prochaine étape (Étape 3).

Insérez les directives (pragma) de la solution dans le fichier *run_hls_script.tcl* (juste après la ligne 17). Pour faire simplement un *copy and paste*, vous trouverez ces directives dans *directives.tcl* (sous l'onglet *constraints* en bas de la solution).

**Attention :** assurez-vous d'avoir installé la patch pour le problème de débordement (overflow) du 1er janvier 2022 qui fait planter la commande de *export_design* pour créer un IP. Pour cela suivez les directives sur le lien suivant avec Python version 2.7.5 : https://support.xilinx.com/s/article/76960?language=en_US

Vous pouvez maintenant exécuter sous Vivado HLS 2018.3 Command Prompt. Cela peut prendre plusieurs minutes d'exécution. En effet, la commande la plus lente qui est *export_design -evaluate verilog -format ip_catalog* réalise la synthèse de votre solution mais en y incluant les interfaces. Le premier résultat (synthèse No 1 du tableau 1) va se retrouver dans le répertoire *…\hls_architecture2\architecture_impl_sol2_2\hls_wrapped_mmult_prj\solution2_2\impl\ip* et sera utilisé à l'étape 3.

***Attention : Si parfois cette commande ne termine pas normalement (avec error) et que la patch est bien installée, c'est fort probablement parce que le path à partir duquel vous exécuter la commande, combiné à la longueur du sous-répertoire où Xilinx va mettre le IP, excède 256 caractères. Il vous faut alors raccourcir le path et recommencer[1].***

Vous remarquerez que contrairement au code de l'exploration architecturale, ici des templates (pour passer différentes dimensions et types) sont utilisés pour la fonction *mmult_hw*. Cette dernière est appelée de la fonction *wrapped_mmult_hw* (aussi en template) qui elle ajoute les interfaces. Finalement, *wrapped_mmult_hw* est appelée de la top function *HLS_accel* de Vivado HLS. C'est d'ailleurs précisément *HLS_accel* qui sera aussi le nom de notre accélérateur dans la librairie Vivado. Le résultat est présenté à la figure 6 avec la mise en place de 2 interfaces de type AXI stream à travers la fonction *pop_stream()* qui charge les 2 matrices en provenance de la mémoire via le DMA et la fonction *push_stream()* qui retourne en mémoire la matrice résultante. Nous analyserons en classe plus en détail cette architecture.

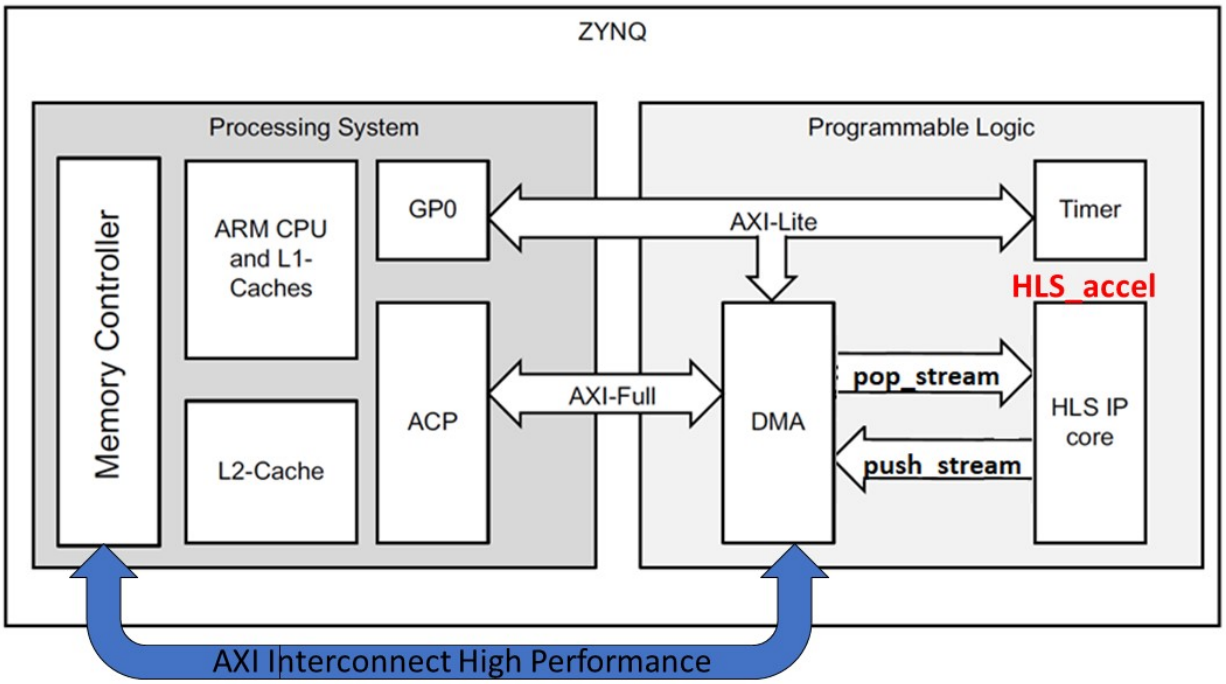



**Figure 6. Configuration du xc7z020 (va être vue en classe).**

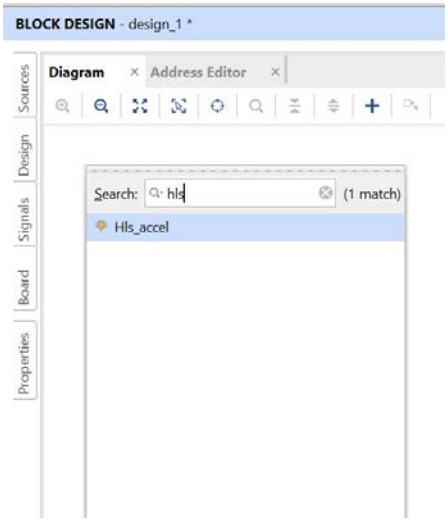
[1] Dans mon cas je travaille dans C:\Lab2\aut2024\hls_architecture2\architecture_impl_sol2_2\... (similaire pour hls_architecture3 et hls_architecture4) et je n'ai jamais ce problème.

## Étape 3 Conception du système dans Vivado

Ici **vous devrez d'abord créer 3 projets** Vivado (dans 3 répertoires différents c'est-à-dire *projet_sol2_2*, *projet_sol3_3* et *projet_sol4_2*) dans lequel on fera l'importation du *hls_accel* respectif.

Pour importer un IP dans le *projet_sol2_2* (ce sera similaire pour les 2 autres projets) et pour ensuite créer votre système, faites les commandes suivantes :

i. Allez dans Vivado_projet1 et créez un projet Vivado comme pour le tutorial Vivado du lab 1 (en sélectionnant votre carte (PYNQ Z2) comme board).
ii. Une fois dans Vivado, dans le menu de droite (PROJECT MANAGER) cliquer sur IP Catalog qui va s'ouvrir à droite de Project Summary.
iii. Puis cliquez avec le bouton de droite sur VIVADO Repository et sélectionnez Add Repository
iv. Cherchez le répertoire *…\hls_architecture2\architecture_impl_sol2_2\hls_wrapped_mmult_prj\solution2_2\impl\ip*
v. Une fois sélectionner le fichier *xilinx_com_hls_HLS_accel_1_0.zip* sera dézipper et deviendra un IP localisé dans *User Repository* de votre projet Vivado.
vi. Dans le menu de droite (PROJECT MANAGER) tapez Create Block Design (comme pour le lab 1!)
vii. Dans Diagram cliquez sur + puis tapez *HLS_accel*. Vous devriez voir apparaître votre accélérateur *HLS_accel0* (ce qui confirme qu'il est bien importé). **Attention!** Ne pas l'instancier dans votre design car il le sera automatiquement à la prochaine étape.




viii. Puisque le système dans lequel on va utiliser *HSL_accel0* est un peu complexe (utilisation non seulement du *ARM* et de *HSL_accel0*, mais aussi de DMA et de Timer) et que l'on

manque de temps, on va le créer avec un fichier *.tcl* et j'expliquerai les composant en classe :

- À partir du code de départ, copiez dans votre répertoire de travail (e.g. C:/Lab2 le fichier *Stream16bits_1copro.tcl.*
- Allez dans la console Tcl (fenêtre du bas dans Vivado) et tapez cd C:\Lab2
- Tapez la commande *source Stream16bits_1copro.tcl*.

Encore une fois, nous examinerons ensemble le détail du design résultant quand nous aborderons le DMA et l'interconnexion AXI.

ix. Dans la synthèse No3 du tableau 1, si la taille est grande est supérieur à 160*160, cette dernière risque d'être supérieur au tampon que le DMA utilise pour transférer, qui est de $2^{14}$ (soit 16384). Assurez-vous bien que le champs *Width of Buffer Lenght Register* en puissance de 2 est plus grand que DIM*DIM*sizeof(short) ou prenez le maximum à 26…

**x.** À partir avec votre expérience de Vivado au lab 1, vous êtes en terrain connu! Poursuivez avec la création du wrapper VHDL, du bitstream, l'Export Hardware en incluant le bitstream et le démarrage de SDK avec LaunchSDK . Nous poursuivrons avec SDK à l'étape 4.

Répétez la même série de commandes (i à x) pour importer le IP de *hls_architecture1/implementation_sol_2_2* mais :

- en spécifiant respectivement à l'étape iv (ci-haut) les paths : …*hls_architecture3\architecture_impl_sol3_3\hls_wrapped_mmult_prj\solution3_3\impl\ip* puis à l'étape viii utilisez le fichier script *Stream16bits_2copro.tcl.*

Répétez la même série de commandes (i à x) pour importer le IP de *hls_architecture1/implementation_sol_3_2* mais :

- en spécifiant respectivement à l'étape v (ci-haut) les paths : …*hls_architecture4\architecture_impl_sol4_2\hls_wrapped_mmult_prj\solution4_2\impl\ip* puis à l'étape viii utilisez le fichier script *Stream16bits_copro.tcl.*

**Source de problème possible avec l'export hardware au point x de la page précédente :**
Assurez-vous, une fois le démarrage de SDK avec *Launch SDK*, que dans le répertoire *sdk/design_1_wrapper_hw_platform_0* du projet Vivado il y a bel et des répertoires concernant *ps7_init*. Si ce n'est pas le cas, inutile d'aller plus loin dans SDK, ça ne fonctionnera pas. En effet, vous devez cocher les 4 premières lignes de *Target Setup* comme illustré à la figure 7 (et comme pour le lab 1). Or si le répertoire *sdk/design_1_wrapper_hw_platform_0* est incomplet, il sera impossible de cocher *Run ps7_init* et *Run ps7_post_config*. Voici ce que je recommande alors :

- Détruire la branche sdk dans le projet Vivado
- Refaire les points viii à x des pages 10 et 11 sans oublier de recréer le wrapper à la toute fin
- Si ça ne fonctionne pas, m'envoyer un message.

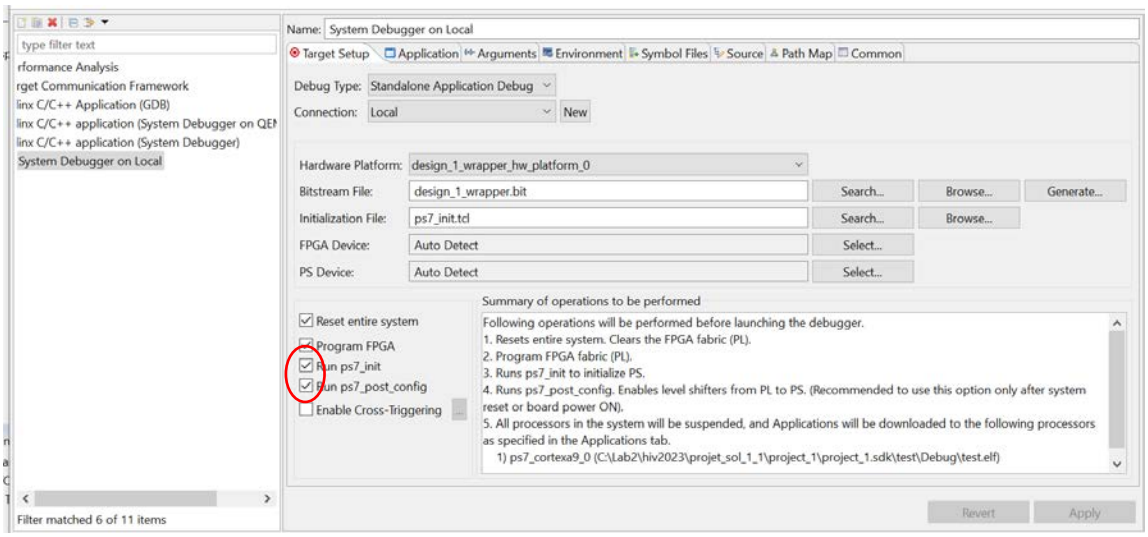



**Figure 7 Si tout fonctionne normalement vous devez pouvoir cocher *Run ps7_init* et *Run ps7_post_config.* Sinon, l'exécution ne fonctionnera pas.**

## Étape 4 Conception de la partie logiciel du système avec SDK et tests de performance

### De manière générale, ce qu'on vous demande de faire

Pour chaque projet Vivado déjà créé dans le code départ (*projet_sol2_2*, *projet_sol3_3* et *projet_sol4_1*), vous allez créer un SDK. L'objectif ici est triple :

i. Valider votre implémentation dans un **test d'intégration fonctionnelle** plutôt qu'unitaire. En effet, à l'étape1 (partie 1) vous avez fait de l'exploration architecturale de différentes solutions et il était possible de simuler avant. Cette simulation était sur la multiplication seulement donc elle constitue un **test unitaire fonctionnelle**. Puis aux étapes 2 et 3, le fait d'avoir ensuite synthétisé 3 solutions tour à tour dans un IP (*hls_accel*), d'avoir intégré l'interface au DMA à chaque IP, et finalement d'avoir intégré chaque IP dans un système avec Vivado constitue l'**intégration système** sur lequel il faut maintenant faire le **test d'intégration**. Pour faire ce test d'intégration dans SDK, le processeur (p.e. core 0) va construire deux matrice A et B de dimension DIM x DIM, va demander au DMA[2] de transférer A et B à *hls_accel* à travers l'interconnexion du FPGA (i.e. l'interconnexion AXI sur canal HP qu'on va bientôt voir en classe), puis le processeur va attendre que le DMA lui envoie le résultat du produit matriciel de A X B (qui est mis dans C) quand celui-ci sera terminé sur le FPGA. Puis, core 0 va faire à son tour le même calcul matriciel (A X B) sur sa propre architecture RISC et va comparer sa matrice résultante C avec celle obtenue par *hls_accel*. Si tous les éléments des 2 matrices résultantes sont les mêmes, on dira que le test d'intégration fonctionnelle passe. Le code qui va servir pour ces tests est dans le répertoire *Test_dePerformance_sur_ARM* dans le code départ.

[2] Le DMA sera vu en classe et il a été intégré dans le fichier script utiliser pour construire votre système à l'étape 3.

Notez finalement que le processeur aurait pu faire son calcul en même temps que *hls_accel* (ce qui représente vraiment l'intérêt d'avoir un DMA), mais ici, pour simplifier les choses, le processeur bloque tant que le DMA n'a pas retourné la matrice C.

ii. Valider votre implémentation dans un premier **test de performance**. Alors que le test d'intégration fonctionnelle vérifie uniquement la fonctionnalité (p.e. sans tenir compte du temps d'exécution), le test de performance va chercher à voir si le calcul fait sur *hls_accel* est plus rapide que le calcul fait sur le processeur. En plus du DMA, on va ici utiliser une minuterie[3] dont la fonctionnalité est décrite ci-après en italique. Prenez le temps de regarder le *main.c* pour repérer ce test de performance. Repérez d'abord le bloc qui fait le calcul du temps sur le processeur (trace *running Matrix Mult in SW*), puis même chose avec le calcul sur *hls_accel* (trace *call HW accelerator*). Dans les 2 cas, le temps courant (en nombre de cycles) de la minuterie est pris avant et après le calcul et on a qu'è faire la différence. La minuterie roule à 100 MHz et qu'on répète 100 fois le test (pour avoir une bonne moyenne) on peut déterminer le temps en microsecondes pour 1 multiplication DIM X DIM en divisant par 100*100[4].

***Fonctionnalité de la minuterie (timer)***
*Cette minuterie compteur de 32 bits de Vivado roule à 100 MHz donc incrémente 100 000 000 fois par seconde. Si vous le laissez compter 42 secondes (4 200 000 000) il sera proche du débordement ($2^{32}$). Tout ceci pour dire qu'il faut faire plusieurs fois le test (ceci afin de faire une moyenne représentative) mais il faut éviter de dépasser ce 42 secondes. Évidemment, on a pris soin de vérifier qu'avec 100 tests vous ne dépasserez pas ces 42 secondes (autant que half que short), mais il est bon de connaitre cette information…*

iii. Un dernier test qui s'inscrit aussi dans la catégorie **test de performance** est de vérifier l'hypothèse suivante (uniquement projet_sol_3_3):

*Pour une même dimension de matrice (DIM x DIM) et pour un même type de coefficient de matrice (half float ou fp16), 2 calculs de multiplication de matrices avec II = 2 effectués en parallèle (1 sur core 0 et 1 sur core 1), prendront moins de temps que 2 calculs consécutifs (séquentiels) avec II = 1 sur le même cœurs (e.g. core 0).*

Ce test s'applique uniquement à la synthèse No 2 du tableau 1 (solution 3_3). Plus précisément, pour cette solution, aux étapes 2 et 3, vous avez créé un IP avec II = 2 que vous avez instanciés 2 fois (via le fichier tcl/tk fourni) dans Vivado. Si vous examinez bien ce schéma Vivado (Figure 8), vous observerez 2 instances de *hls_accel* (*hls_accel 0* et *hls_accel1*) 2 DMA (*axi_dma_1* et *axi_dma_2*) et 2 minuteries (*axi_timer_1* et *axi_timer_2*) . L'idée est de faire rouler en parallèle deux calculs de multiplication matricielle. Un calcul matriciel se fera sur le *core 0* (donc le processeur pilotera *hls_accel_0*, *axi_dma_1* et *axi_timer_1*) et un deuxième calcul se fera en parallèle sur *core*

[3] La minuterie a aussi été intégrée dans le fichier script utiliser pour construire votre système à l'étape 3
[4] Attention si vous diminuez le nombre de tests assurez-vous ne pas trainer des fractions car on utilise des entiers (le float étant plus compliqué à imprimer sur la console de SDK).

*1* (donc le processeur pilotera *hls_accel_1*, *axi_dma_2* et *axi_timer_2*). Ici, le défi est d'assurer cette parallélisation du calcul. Comment? En synchronisant le départ des cores comme vous l'avez si bien fait dans la partie 3 du lab 1. Plus précisément via la DDR pour partager une zone mémoire entre les cores (via BASEADDR = 0x3000000) donc avec :

```
volatile uint32_t* req = (uint32_t*) (BASEADDR +0x4);
volatile uint32_t* ack = (uint32_t*) (BASEADDR +0x0);
volatile uint32_t* core1_begin_time = (uint32_t*) (BASEADDR +0x8);
volatile uint32_t* core1_end_time = (uint32_t*) (BASEADDR +0x0C);
```

De là, core 1 **va démarrer le premier**, faire son calcul en logiciel, mais il devra attendre le signal de core 0 pour repartir et compléter le calcul sur FPGA via *hls_accel_1*.

De son côté, core 0 **va partir en deuxième**, faire son calcul en logiciel, puis juste avant de faire son calcul sur FPGA via *hls_accel_0* et va donner le signal à core 1 pour que lui aussi démarre son calcul sur FPGA.

Finalement, après avoir effectué son calcul FPGA, core 0 devra attendre le signal de core 1 qui indique qu'il peut récupérer les temps de calcul de core 1 (*core1_begin_time* et *core1_end_time*). Pourquoi ces 2 valeurs? Elles vont nous permettre de vous assurer de la durée totale des 2 calculs. En fait, core 0 va prendre le minimum entre core 0 et core 1 des 2 temps de départ et le maximum des 2 temps de fin (voir lignes 249 de *main.c* des solutions 1.3 et 2.3). Puis les 2 cores pourront terminer leur exécution.

## Plus en détail ce que vous devrez faire et quels sont les résultats à produire

Avant de regarder le détail pour manipuler SDK à travers vos 3 projets, voici ce que vous **devez-faire et quels résultats devez-vous sortir des points i, ii et iii des 2 pages précédentes?** Les voici énumérés :

a. Construire le projet SDK de **projet_sol2_2**, le compiler et exécuter. Faites une capture d'écran dans SDK (similaire è la Figure 9). Cette capture devra apparaitre dans votre rapport. Gardez bien en tête le temps total avec 1 core (dernière trace affichée à la figure 9).
b. Construire le projet SDK de **projet_sol3_3**, compiler et exécuter. (Rappelez-vous ici la synchronisation avec req et ack et que core 1 est maitre. Il faut donc partir dans le bon ordre les cores et ne pas faire deux reset.) Faites une capture d'écran dans SDK (similaire à la figure 10). Cette capture devra apparaitre dans votre rapport. Puis comparez le temps total pour core 0 et core 1 (dernière trace affichée) avec 2 fois celui de l'étape a. À partir de là, vous êtes en mesure de dire si l'hypothèse de la page 16 est valide.
c. Construire le projet SDK de projetsol4_2, le compiler et exécuter. Faites une capture d'écran dans SDK (similaire à la Figure 9). Cette capture devra apparaitre dans votre rapport. Gardez bien en tête le temps total avec 1 core.

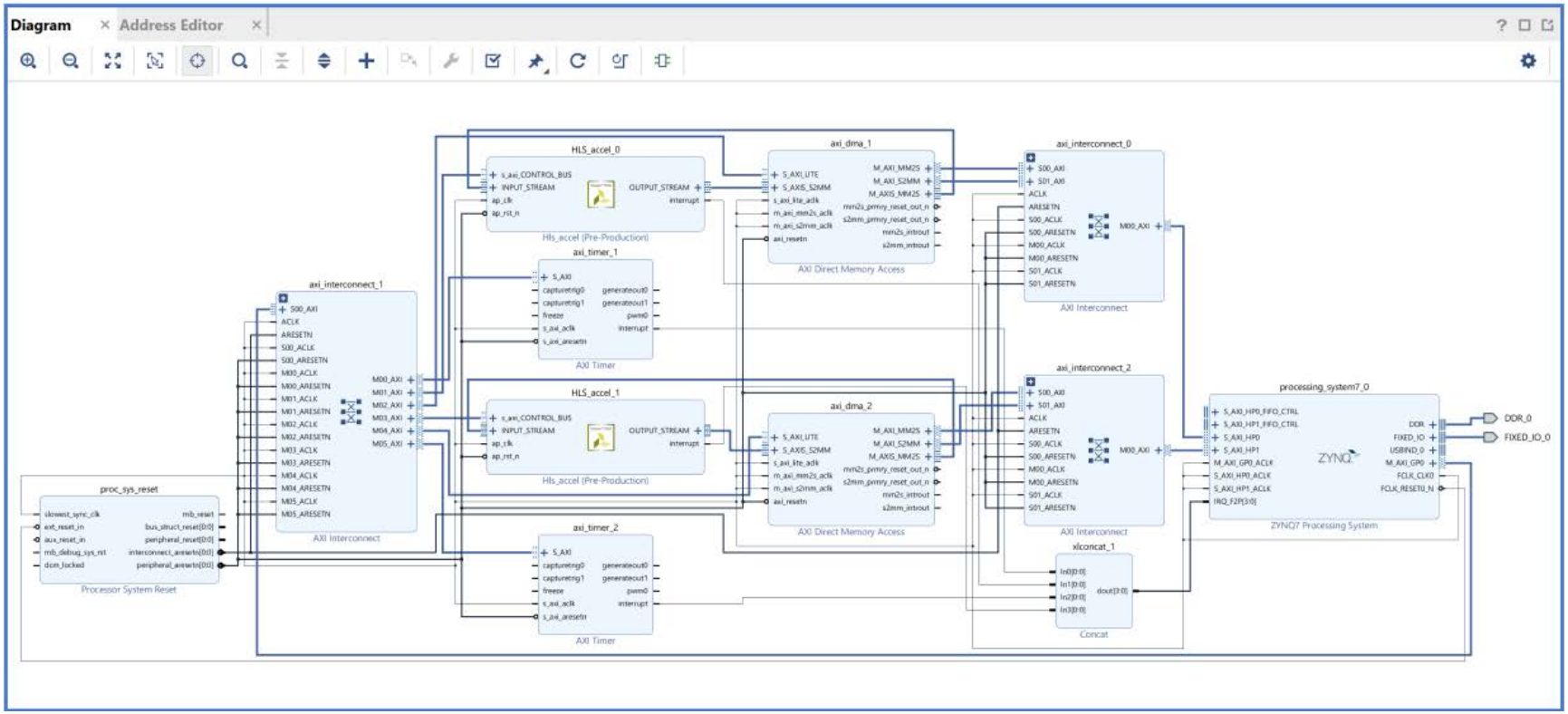



**Figure 8 projet_sol3_3 dans Vivado avec intégration de 2 IPs hls_accel**

```
Connected to COM4 at 115200
************************************************************
 _fp16 54 x 54  MATRIX MULT -> AXI DMA -> ARM ACP
 XAPP1170 redesigned with Vivado + HLS + IP Integrator 2015.3
************************************************************
DMA Init done
Loop time for 1024 iterations is 1369 cycles
Running Matrix Mult in SW

Total run time for SW on Processor is 3837762 cycles over 1024 tests.

Cache cleared

Total run time for AXI DMA + HW accelerator is 12626 cycles over 1024 tests
Acceleration factor:

ERROR: results mismatch: 2 times

Temps total avec 1 core:       microsecondes pour 1 multiplication de matrices 54 x 54
```

**Figure 9 Exemple de trace obtenu (certaines valeurs ont été enlevées)**

```
************************************************************************
 _fp16 MATRIX MULT II = 2-> AXI DMA -> ARM AXI HP   Core 1
 Matrix 54 x 54
 Number de test effectues  10
 Inspire de l'application XAPP1170 de Xilinx
************************************************************************
DMA Init done
Core 1: loop time for 10 iterations is 18 cycles
Core 1: Running Matrix Mult in SW

Core 1: Total run time for SW  is 38495 microseconds over 10 tests.
************************************************************************
 _fp16 MATRIX MULT II = 2-> AXI DMA -> ARM AXI HP Core 0
 Matrix 54 x 54
 Number de test effectues  100
 Inspire de l'application XAPP1170 de Xilinx
************************************************************************
DMA Init done
Core 0: loop time for 100 iterations is 124 cycles
Core 0: running Matrix Mult in SW

Core 0: total run time for SW on Processor is 38494 microseconds over 100 tests.
Core 0: call HW accelerator


Cacache cared


Core 1: end of HW accelerator 588653
Core 1: total run time for AXI DMA + HW accelerator is 588 microseconds over 10 tests
Core 1: acceleration factor:

Core 1:  results mismatch: 2 times

 accelerator 1962224
Core 0: total run time for AXI DMA + HW accelerator is 196 microseconds over 100 tests
Core 0: acceleration factor:

Core 0 ERROR: results mismatch: 2 times

Temps total pour core 0 et core 1:       microsecondes pour 2 multiplications de matrices 54 x 54 en parallele
```

**Figure 10 Exemple de trace obtenu (certaines valeurs ont été enlevées)**

## Détails dans SDK

Ici vous êtes en terrain connu puisqu'il s'agit de manipuler SDK. Vous travaillerez toutefois en standalone (*bare metal* ou encore *foreground/background*) plutôt qu'avec uC/OSIII. Le standalone est plus simple (figure 11) mais on n'a pas le concept de tâches ou thread, c'est pour cela qu'on doit devra faire le test sur 2 cores pour *projet_sol3_3*.

**Pour construire les projets *projet_sol2_2* et *projet_sol4_2 :***

Utilisez *Standalone* pour OS Platform quand vous créez votre projet *test* (Figure 11). Vous n'avez pas à créer et à initialiser le BSP, il sera créé automatiquement de BSP avec l'option Create New (Figure 11) et il contient tout ce qu'il faut (contrairement uC/OS-III, vous n'avez rien à configurer). Puis, quand vous créez votre application, vous verrez qu'un *hello world* vient s'ajouter dans le projet *test*. Tout comme au lab 1, il permet de voir que votre BSP fonctionne bien. Une fois tester vous détruisez *helloworld.c* et ajoutez les 3 fichiers qui se trouve dans le répertoire *Test_de_performance_sur_ARM* du code de départ de la manière suivante :

- pour le projetsol_2_2, prendre dans *Test_de_performance_sur_ARM/Test_pour_half_float/Projet_sol2_2*
- pour le projet_sol_3_2, prendre dans *Test_de_performance_sur_ARM/Test_pour_short/Projet_sol4_2*

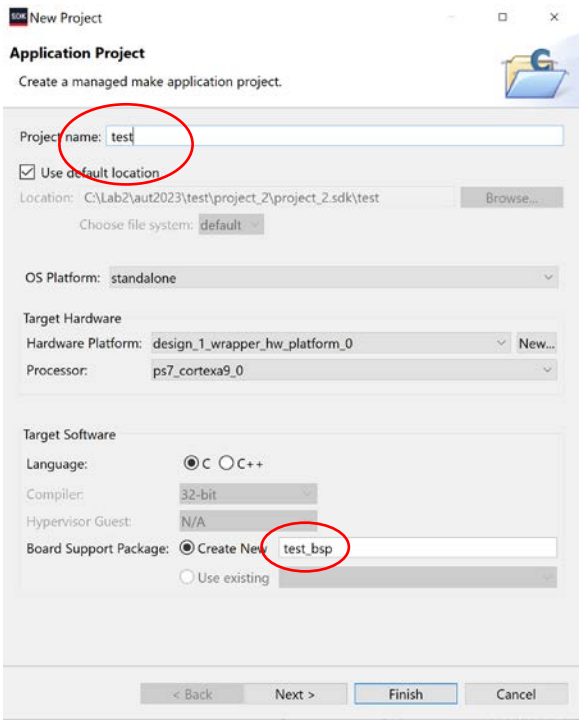



**Figure 11 Création du projet test avec un OS standalone plutôt que uC/OS-III**

Aussi, pour chaque projet SDK, n'oubliez pas de modifier le fichier *lsscript.ld* du répertoire source afin d'augmenter la mémoire c.-à-d. le heap Size à 0xFF00 et la stack à 0xFF00.

Finalement, retenez que pour *projet_sol2_2* qui utilise un half float 16, il manque une librairie quand pour le test software sur ARM. **Il faut donc activer la librairie du half precision (fp16) :**

1) Cliquez sur le bouton de droite sur test dans la fenêtre de gauche d'édition (figure 12) puis faire glisser sur C/C++ Build Settings

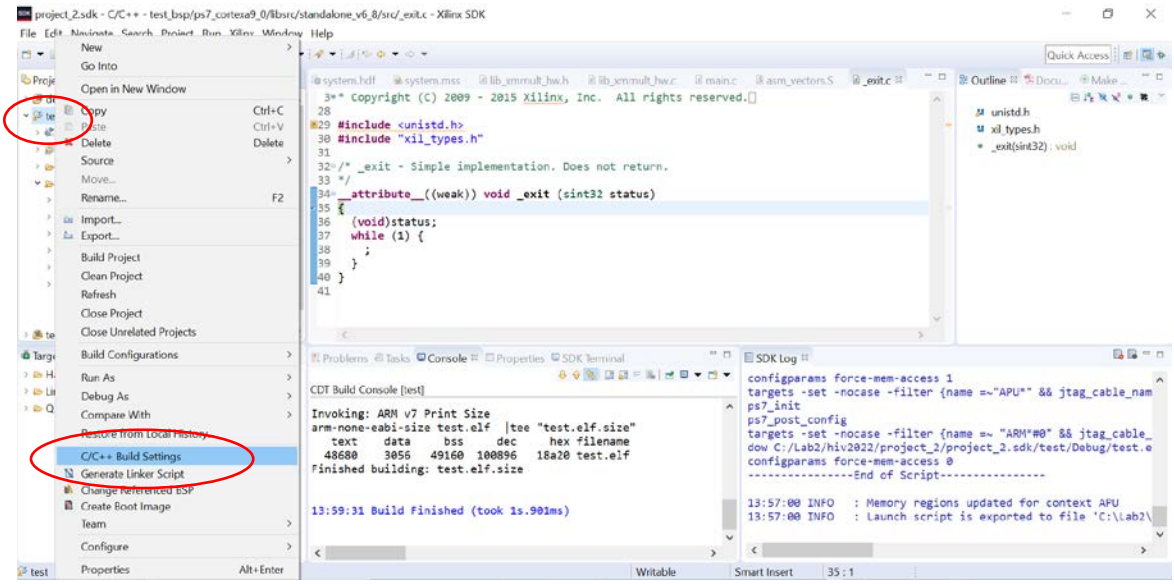



**Figure 12 Comment ajouter une librairie à la compilation dans SDK**

2) Dans la fenêtre qui apparaitra (Figure 13) cliquez sur Miscellaneous puis ajoutez -mfp16-format=ieee au commande du makefile. Faites finalement Apply et OK.

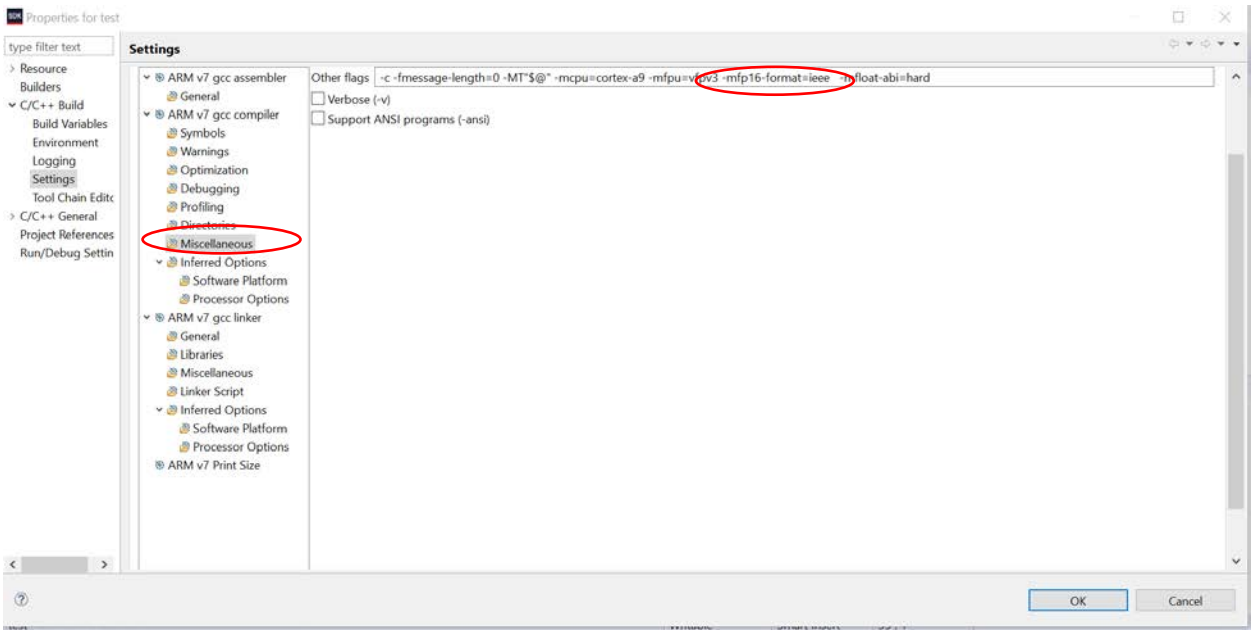



**Figure 13 Comment ajouter une librairie à la compilation dans SDK (suite)**

À partir de là vous pouvez compléter les test pour ***projet_sol2_2*** **et** ***projet_sol4_2***

**Pour construire le projet *projet_sol3_3 (similaire au lab 1 partie 3)*:**

Toujours avec *Standalone* pour OS Platform, créez votre application mais cette fois vous devez créer une application par core et donc créer 2 BSP (core 0 et core 1). Ceci est illustré à la figure 12. Des *hello world* viennent s'ajouter. Pour chaque core, détruisez sur les *helloworld.c* et ajoutez les 3 fichiers qui se trouve dans le répertoire *Test_de_performance_sur_ARM* du code de départ de la manière suivante :

- pour le projet_sol_2_2, prendre dans *Test_de_performance_sur_ARM/Test_pour_half_float/Projet_sol3_3* et

Aussi, n'oubliez pas de modifier le fichier *lsscript.ld* du répertoire source afin d'augmenter la mémoire c.-à-d. le heap Size à 0xFF00 et la stack à 0xFF00.

Et tout comme pour le ***projet_sol2_2*** **il faut activer la librairie du half precision (fp16) (figures 12 et 13).**

Finalement, on vous recommande ici de d'abord faire exécuter indépendamment sur chaque core en cliquant un seul core à la fois (Figure 14). Mais avant, mettre en commentaire les 2 while(!*ack), sinon voua allez rester bloqué. Si tout fonctionne bien sur chaque coeur, faites la synchronisation demandée (comme au lab 1 partie 3) simplement en enlevant les commentaires devant les while de chaque core. Pour démarrer les 2 cores souvenez-vous que vous devez faire un *Reset processor* uniquement sur le core 1 et pas sur le core 0 (ou vice versa).

Dernier point à faire attention! Comme il est possible que vous ayez plusieurs SDK ouverts, les résultats d'une fenêtre de SDK peuvent s'afficher dans une autre fenêtre de SDK. Ne soyez donc pas surpris si rien ne s'affiche : c'est soit votre terminal qui n'est pas branché ou soit qu'il s'affiche dans une autre fenêtre…

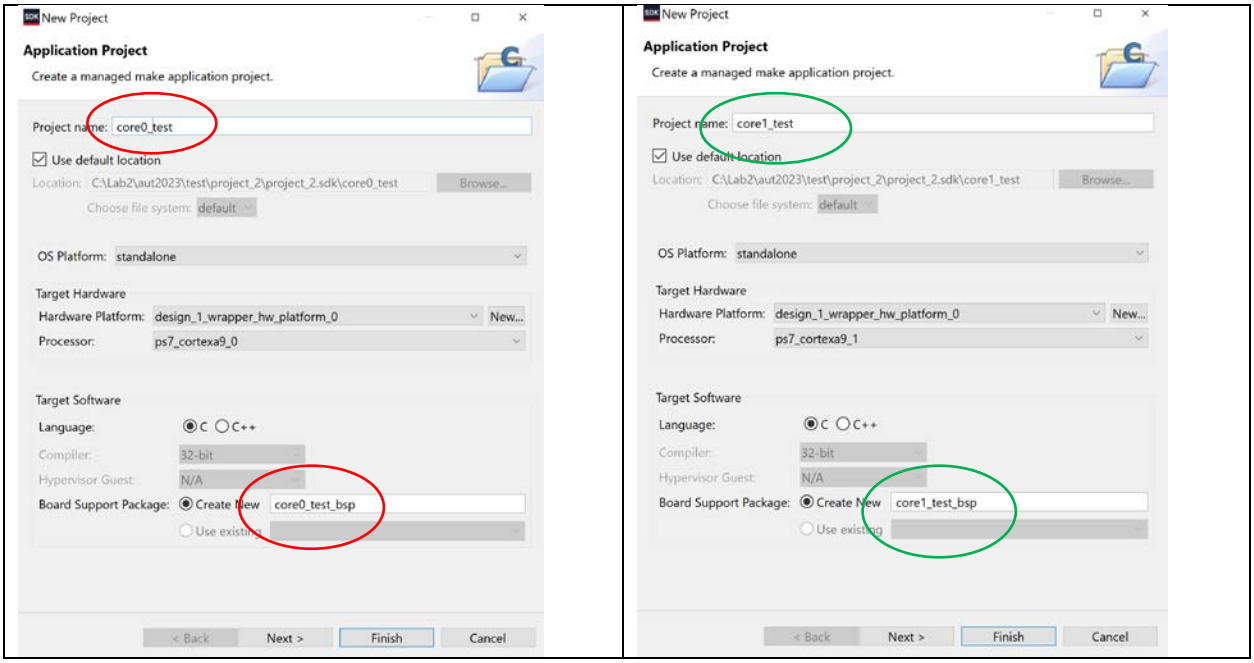



**Figure 14 Création du projet test avec un OS standalone plutôt que uC/OS-III**

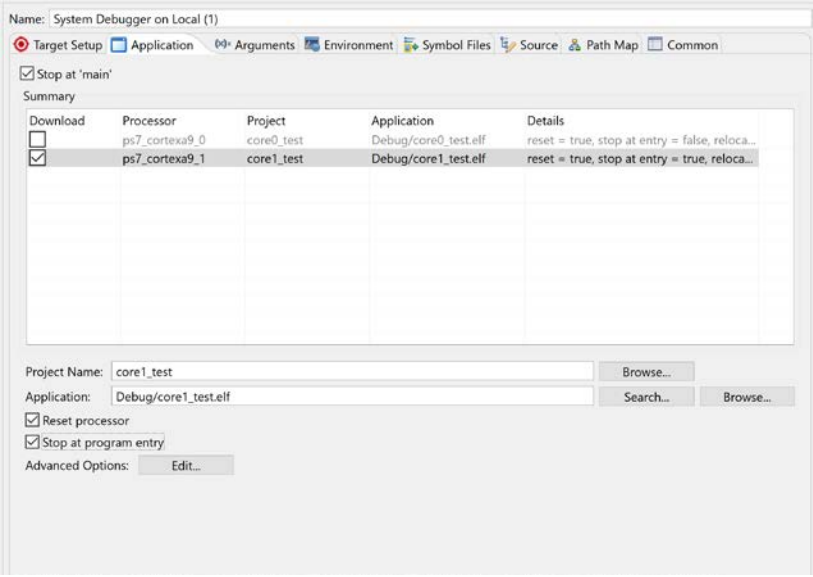



**Figure 15 Exemple si on souhaite tester uniquement le core 1 avant de mettre la synchronisation**

**Questions et barème**

**Q1S** Concernant le projet_sol2_2 (Synthèse No1 Tableau 1):

a) Que peut-on conclure des 2 erreurs de comparaison entre logiciel et matériel pour le projet_sol2_2?

b) L'accélération du matériel par rapport au logiciel est-elle significative? Anticipiez-vous une telle accélération à la suite de la section ***Exploration architecturale no 2*** ? Justifiez.

**Q2S** Concernant le projet_sol3_3 (Synthèse No2 Tableau 1):

a) Démarrez Vivado et ouvrez le projet *projet_sol3_3*. Dans le bas de la colonne de gauche, ouvrez l'onglet *IMPLEMENTATION* puis *Open Implemented Design* puis cliquez sur Report Utilization. Vous devriez voir apparaitre la boite de la figure 16 sur laquelle vous cliquez OK. Puis vous verrez apparaitre le tableau de la figure 17 après avoir cliquez sur l'onglet design_1_i. En observant HLS_accel_0 et HLS_accel_1, vous verrez qu'il y a un certain nombre de BRAM. Si vous ouvrez l'onglet HLS_accel_0 y a-t-il une correspondance entre le nombre de BRAM et les pragma HLS ARRAY_PARTITION utilisés à l'Exploration architecturale no 3 (solution 3_3)? Justifiez.

b) Pourquoi ce nombre de BRAM observé dans Vivado n'était pas comptabilisé dans votre rapport de synthèse de Vivado_HLS (sous Summary)?

c) En observant juste les ressources BRAM, aurait-on pu mettre 3 instances de hls_accel dans le projet_sol3_3? Justifiez.

d) En observant juste les ressources de l'ensemble de la puce SoC 7020, aurait-on pu mettre 3 instances de *hls_accel* dans le projet_sol3_3? Justifiez.

e) Sachant que j'ai une puce de la série Zynq-7000 SoC (chap. 1, p37) dans laquelle appartient la puce 7020 avec assez de ressources pour mettre plus de 4 instances sur de hls_accel. Qu'elle est alors ma limite en terme du bus AXI. Observez bien votre design (Fig 8). Justifiez.

f) Finalement, est-ce que l'hypothèse de la page 16 est vérifié? Justifiez.

**Q3S** Concernant le projet_sol4_2 (Synthèse No3 Tableau 1). Si le DIM obtenu lors de ***Exploration architecturale no 4*** solution 4_2 est différent de celui proposé au Tableau 1 (DIM=200), expliquez pourquoi ce choix de 200 qu'on peut voir comme une valeur limite. Suggestion : allez consultez les ressources comme pour la question Q2S a).

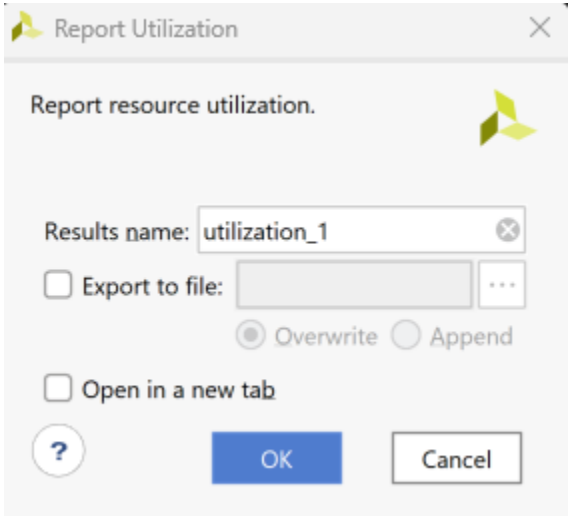



**Figure 16**

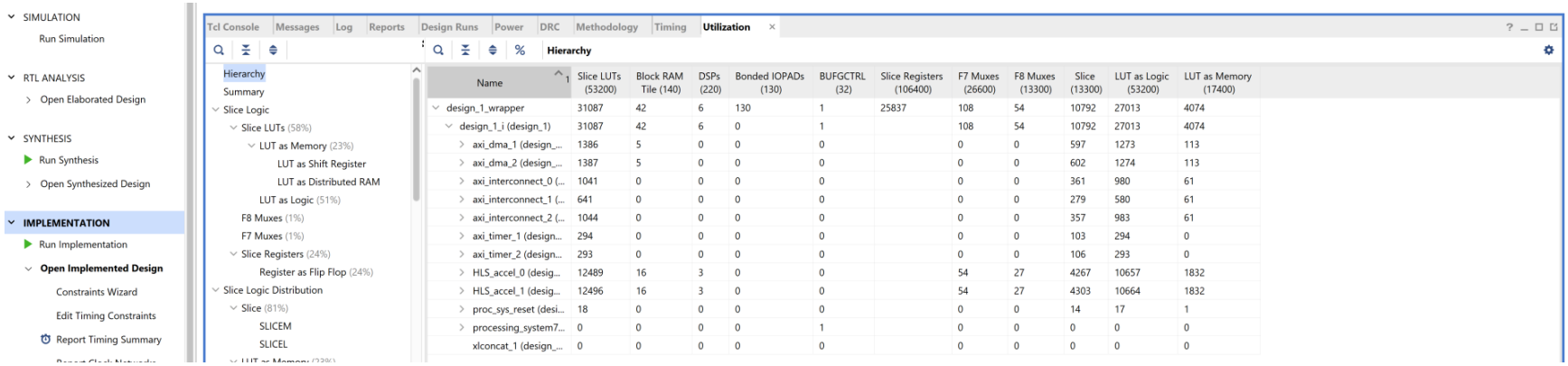



**Figure 17**

**Barême et évaluation**

| Questions | Notes | Commentaires |
|---|---|---|
| Exploration architecturale no 1 et questions | 1 | |
| Exploration architecturale no 2 et questions | 1 | |
| Exploration architecturale no 3 et questions | 2 | |
| Exploration architecturale no 4 et questions | 1 | |
| | | |
| Réponse à QS1 | 1 | |
| Réponse à QS2 | 3 | |
| Réponse à QS3 | 1.5 | |
| | | |
| Évaluation au lab | 5.5 | |
| Total | 16 | |

L'évaluation au lab (en présentiel) d'une durée de 15 mins max aura lieu le 4 décembre.
L'équipe complète doit être présente. La remise du lab sera le 6 décembre avant minuit pour tous les groupes.