Laboratoire 2 – Partie 2

Un accélérateur Zynq pour la multiplication point flottante conçu avec Vivado HLS et intégré dans un système

Ce laboratoire s'inspire d'un *Application Note* de Xilinx (XAPP1170) conçu pour s'exécuter sur la Zynq ZC702 et le porte sur la Cora Z7. 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
2) Ajouter à ce coprocesseur une interface avec le DMA (Direct Memory Access)
3) Comprendre le concept de DMA
4) Connecter ce DMA à une mémoire DDR via un bus rapide (HP)
5) Programmer le DMA à partir du ARM
6) Exporter le coprocesseur dans la librairie de Vivado
7) Concevoir avec Vivado le design du système incluant le coprocesseur, le DMA et le ARM et autre périphériques
8) 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.

Évidemment, vous remarquerez qu'il y a plusieurs activités au menu, mais pour accélérer le travail plusieurs étapes sont déjà préconçues, de sorte qu'il ne vous reste qu'à faire l'assemblage. En particulier, vous verrez que l'on a recours à des fichiers script pour accélérer la conception des étapes 1), 6) et 7), ce qui est aussi une pratique industrielle courante.

Le travail se fera en 4 étapes. L'étape 1, qui consiste à explorer et optimiser des architectures sera la plus longue. Les étapes 2, 3 et 4 vous enseigneront une approche de conception de bas vers le haut utilisée dans l'industrie pour concevoir un système.

Ce laboratoire s'inspire d'un *Application Note* de Xilinx fourni avec le code de départ et on vous y référera au besoin. Dans ce même *Application Note*, nous avons ajouté des blocs (en français donc facile à repérer). Ces notes (souvent surligner) sont des adaptations pour l'utilisation de la Cora Z7 (plutôt que la ZC702) ou des éclaircissements.

Dans ce qui suit, voici donc **les 4 étapes** à réaliser :

Étape 1 Synthèse HLS et optimisation

Avant de débuter, assurez-vous de bien avoir complété le chap. 7, lab 1 du tutoriel. Le calcul matriciel étant sur des matrices 4x4 dans le tutoriel, nous utiliserons des matrices de plus grandes dimensions, dont celles de départ qui est 32x32.

Dans le répertoire *architecture_exploration*, nous allons explorer **3 architectures** qui se trouvent respectivement dans les répertoires de code de départ :

1) *hls_design1/architecture_exploration*,
2) *hls_design2/architecture_exploration* et
3) *hls_design3/architecture_exploration*.

Soit une forme générale du code sur laquelle nous allons travailler :

```
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 1 Multiplication matricielle

N.B.

i. Pragma *ARRAY_RESHAPE* est équivalent à *HLS ARRAY_PARTITION*, c.-à-d. qu'on aurait pu aussi utiliser : *HLS ARRAY_PARTITION variable A complete dim=2* et *HLS ARRAY_PARTITION variable B complete dim=1*

ii. 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).

À travers ces 3 explorations sur un code similaire, nous souhaitons valider l'hypothèse suivante:

Hypothèse de départ *En insérant une directive de pipeline à la boucle for col avec une demande de débit II=1 (c.-à-d. qu'on désire produire un résultat à tous les cycles), le résultat est que la boucle for la plus interne (product) est entièrement déroulée*[1] *et pipelinée et nous nous attendons à ce que le circuit résultant inclue à peu près M opérateurs de multiplication/addition et qu'il est une latence d'environ N * P cycles. Autrement, dit si N=M=P on aura pour la multiplication de matrice une complexité de* $O(N^2)$ *plutôt que* $O(N^3)$ *mais avec une complexité en ressources de O(N).*

[1] Si on en avait eu plus qu'une, ils auraient tous été déroulées.

Exploration no 1 dans hls/hls_design1/architecture_exploration

Dans ce répertoire, il existe 4 fichiers : *mmult.h*, *mmult_accel.cpp*, *mmult_test.cpp* et *run_hls_script_explore.tcl*

mmult_test.cpp va permettre de faire la simulation et comparer chaque solution obtenue de l'exploration avec une version s'exécutant sur votre ordinateur (équivalent à un scoreboard de testbench). Cette comparaison n'est uniquement que pour un niveau fonctionnel.

mmult.h et *mmult_accel.cpp* contiennent la fonction *mmult_hw* qui est en faites le *top function* dans HLS Vivado. On va donc y explorer des solutions.

Pour vous aider à comprendre le but de cette exploration et son importance, retenez pour l'instant qu'une fois qu'on aura déterminé les bons paramètres de notre architecture (en l'occurrence ici la meilleure solution pipelinée avec le débit maximum possible), nous les appliquerons ensuite à un autre *top function HLS_accel* (dans le répertoire *implementation*), qui lui reprendra l'optimisation de *mmult_hw* mais lui ajoutera aussi l'interface (*stream*) pour fonctionner avec le DMA[2]. Ce sera également là que nous générerons un IP qui pourra ensuite être instancié dans Vivado.

Finalement, *run_hls_script_explore.tcl* vous permet de bâtir directement une solution 0 sans optimisation. Pour cela, démarrez *Vivado HLS 2018.3 Command Prompt*, placez-vous dans le répertoire courant (par ex., cd …\Lab2\hls_design1\architecture_exploration) et exécuter le script avec la commande *hls_vivado -f run_hls_script_explore.tcl*. Vous pouvez ensuite fermer la fenêtre et démarrer Vivado HLS 2018.3 (version GUI) et ouvrir le projet *mmult_hw*.

À partir de là trouver la meilleure solution pipelinée (HLS pipeline) appliquée à la boucle L2, permettant le meilleur débit et donc la meilleure latence[3] pour une implémentation sur la CoraZ7. Deux approches sont possibles (valider les 2 approches):

i) Une fois la boucle sur laquelle appliquer le pipeline, briser complètement la matrice A et la matrice B et minimiser le débit en commençant à II=1 tout en vous assurant que vous respectez les ressources de la carte Cora Z7. Si les contraintes de ressources ne sont pas respectées, augmenter II à 2 puis resynthétiser. Et ainsi de suite…

ii) Une fois la boucle sur laquelle appliquer le pipeline, fixer le débit à sa meilleure valeur c.-à-d. II=1 et chercher à maximiser le nombre de blocs en commençant par 16 (pourquoi 16 et pas 32?) tout en vous assurant que vous respectez les ressources de la carte Cora Z7. Si les contraintes de ressources ne sont pas respectées, diminuer le nombre de blocs (vous pouvez faire une recherche binaire) puis resynthétiser. Et ainsi de suite…

[2] On aurait pu faire l'exploration architecturale directement dans *HLS_accel* mais *mmult_hw* n'ayant pas d'interface, ce sera plus rapide et plus simple de travailler avec *mmult_hw*

[3] La partie 1 du lab 2 (chapiter 7, lab 2) devrait vous avoir inspirer...

Question 1 : Dans un cas comme dans l'autre (i ou ii), la dimension que vous choisirez pour A n'est pas la même que pour B. Référez à la figure 1 et justifiez le choix des dimensions ainsi que les valeurs associées à ses dimensions (en indiquant bien pourquoi commencer avec 16 et non 32).

Question 2 : Justifiez pourquoi i) est équivalent à ii). Suggestion : utilisez *schedule viewers* dans l'analyse de Vivado HLS et comparez les 2 solutions obtenues.

Question 3 : En examinant le rapport de synthèse de la solution retenue (section *loop*), ainsi que les ressources utilisées, peut-on conclure à ce stade-ci quelque chose sur la complexité de notre hypothèse de départ?

Question 4 : Créer une dernière solution avec les directives suivantes (attention le pipeline est cette fois dans L1):

```
set_directive_array_partition -type complete -dim 2 "mmult_hw" a
set_directive_array_partition -type complete -dim 1 "mmult_hw" b
set_directive_pipeline -II 1 "mmult_hw/L1"
```

Ici bien que vous observiez un dépassement des ressources pour la CoraZ7 (et probablement aussi pour une Cora Z10 et un Zedboard), selon notre hypothèse, on devrait observer une complexité en latence d'exécution de O(N) et de O(N^3) en ressources? Justifiez. Si ce n'est pas le cas, justifiez le problème?

Finalement, retenez vos solutions pour l'évaluation et la meilleure solution pour l'étape 2.

Exploration no 2 dans hls_design2/architecture_exploration

L'exploration architecturale peut consister en l'ajout de pragma. Mais également, puisqu'on travaille au niveau algorithmique, elle peut aussi consister à modifier l'algorithme. C'est ce que nous allons faire dans cette section.

Peu importe le choix de l'approche choisie précédemment (i ou ii), vous devriez avoir constaté que pour un calcul de multiplication de matrices de 32 par 32 à réaliser avec la Cora Z7, il est impossible de briser complètement les matrices (avec *block* = 16 blocs ou *complete*) tout en ayant un débit II=1. Pour vos connaissances générales, sachez qu'on aurait pu le faire avec une Zedboard, mais la Cora Z7 (et même Cora Z10) est trop petite…

Nous allons donc tenter d'explorer un compromis pour tenter de prendre moins de ressources. Nous allons explorer une approche de type *diviser pour régner* tel qu'illustré à la Figure 2. Vous devez modifier *mmult_accel.cpp* afin de réaliser 4 multiplications (c.-à-d., pour Q1, Q2, Q3 et Q4 tel qu'illustré à la fig. 2):

- Vous remarquerez (ligne 9 de mmult_accel.cpp) qu'un 4^e a été ajouté à la fonction (quadrant),
- Et que l'appel de mult_hw se fera 4 fois (ligne 48 de mmult_test.cpp)
- Complétez le code à la ligne 13 qui fixe les paramètres des boucles avant l'exécution du calcul matriciel.

À partir de là trouver la meilleure solution pipelinée (HLS pipeline) sur L2, permettant le meilleur débit et donc la meilleure latence. Cette fois, vous pouvez vous restreindre à une ou l'autre des 2 approche (i ou ii de la section précédente).

Remarque. Puisque vous allez ajouter du code, assurez-vous de bien simuler votre résultat avant de le synthétiser. Il serait bête de synthétiser et amener dans Vivado et SDK une version boguée!

Question 5 : En comparant les latences d'exécution, ainsi que les ressources, entre votre meilleure résultat et celui obtenu à l'exploration 1, les valeurs du rapport de synthèse changent-elles significativement. Pourquoi?

Finalement, retenez cette solution pour l'évaluation et pour l'étape 2.

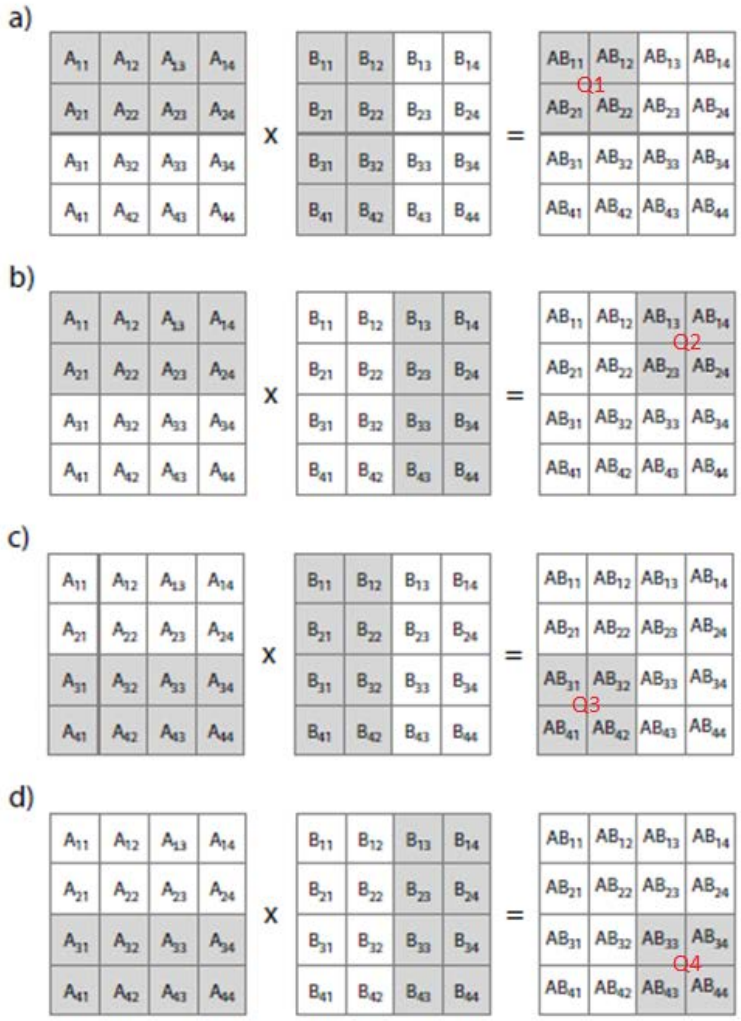

Figure 7.3: One possible blocked decomposition of the matrix multiplication of two 4 × 4 matrices The entire **AB** product is decomposed into four matrix multiply operations operating on a 2 × block of **A** and a 4 × 2 block of **B**.

Figure 2

Exploration no 3 dans hls_design3/architecture_exploration

En supposant une matrice carrée (NxN) et en utilisant toujours une directive HLS pipeline dans col (L2) comme à la Fig. 1, on vous demande ici de déterminer par exploration architecturale la valeur de N (plus précisément la variable DIM à la ligne 7 de *mmult_test.cpp*) afin d'avoir II=1 et un partitionnement en blocs de N/2.

Finalement, retenez cette solution pour l'évaluation et pour l'étape 2.

Étape 2 Ajout des interfaces et productions de 3 IPs

Tel qu'expliqués à la section 1.1), nous allons maintenant appliquer à la *top function HLS_accel* (situé dans les répertoires ***implementation*** du code de départ) les optimisations faites dans l'étape 1 sur *mmult_hw* tout en lui ajoutant aussi l'interface (*stream*) pour ainsi fonctionner avec le DMA. Ce sera également là que nous allons va générer un IP qui sera par la suite instancié dans Vivado (Étape 3).

Remarque. *Si vous désirez mieux comprendre ce que vous faites, il est conseillé de lire les pages 10 à 13 du Application Note, en vous assurant de commencer à comprendre les figures 7 et 8 (nous y reviendrons aussi en classe).*

Exploration no 1 dans hls_design1/implementation

Utilisez le fichier *run_hls_script.tcl* et insérez les directives de la meilleure solution obtenue à l'étape 1. Puis exécuter *hls_vivado -f run_hls_script.tcl* 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 bas niveau (comme en INF3500) de votre solution. Le résultat, va se trouver dans le répertoire *implementation/hls_wrapped_mmult_prj/solution0/impl/ip* et sera utilisé à l'Étape 3.

Vous remarquerez que contrairement au code de l'exploration architectural, 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* sera aussi le nom de notre accélérateur dans la librairie Vivado.

Exploration no 2 dans hls_design2/implementation

Ici vous devrez faire la même chose qu'au premier paragraphe de la section précédente, mais en l'appliquant au répertoire *hls_design2/implementation*. Mais avant tout, vous devez aussi insérer votre code pour le traitement des quadrants dans *mmult_hw* (à la ligne 48 de *mmult.h*). Notez aussi que l'appel de la fonction mmult_hw (4 fois) (à la ligne 152 du même fichier) est bel et bien la !

Exploration no 3 dans hls_design3/implémentation

Ici vous devrez faire la même chose qu'au premier paragraphe de la section précédente, mais en l'appliquant au répertoire *hls_design3/implementation*. Mais avant tout, insérez la dimension trouvée à l'étape 1 à la ligne 9 de *mmult.h*.

Étape 3 Conception du système dans Vivado

Ici **vous devrez d'abord créer 3 projets** Vivado[4] (dans 3 répertoires différents c'est-à-dire Vivado_project1, Vivado_project2 et Vivado_project3).

Ensuite, pour chaque projet, vous devrez d'abord importer votre IP généré à l'étape 2 :

- Pour le projet 1, ce sera celui dans *hls_design1/implementation/…*
- pour le projet 2, ce sera celui dans *hls_design2/implementation/…*, et
- pour le projet 3 ce sera celui de *hls_design3/implementation/...*

Pour importer le IP (de hls_design1/implementation de l'étape 2) dans le projet1 et 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 CoraZ7 comme board).
ii. Une fois dans Vivado, dans le menu de droite (PROJECT MANAGER) tapez IP Catalog
iii. Vous verrez alors un onglet *IP Catalog* apparaitre à côté de *Project Summary*. Cliquez sur cet onglet.
iv. Puis à droite de *Search*, cliquez sur bouton de droite et sélectionnez *Add Repository*
v. Cherchez le répertoire *hls_design1/implementation/hls_wrapped_mmult_prj/solution0/impl/ip*
vi. 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.
vii. Dans le menu de droite (PROJECT MANAGER) tapez Create Block Design et ajouter tout de suite un wrapper à design (comme pour le lab 1!)
viii. Dans Diagram cliquez sur + puis tapez *HLS_accel*. Vous devriez voir apparaître votre accélérateur *HLS_accel0* (ce qui confirme qu'il est bien importé)
ix. Mais 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* (comme en INF3500!). Tapez donc dans la console Tcl Console (en bas) la commande *source design_HP.tcl* (le fichier se trouve dans le code de départ).
x. À partir de la poursuivre à l'étape 3 de la page 19 du *Application Note* et ce jusqu'à la page 23 (avec votre expérience de Vivado au lab 1, vous êtes en terrain connu!).

Répétez la même série de commandes pour importer le IP de *hls_design2/implementation* et *hls_design3/implementation* mais en spécifiant respectivement à l'étape v (ci-haut) les paths :
hls_design2/implementation/hls_wrapped_mmult_prj/solution0/impl/ip et
hls_design3/implementation/hls_wrapped_mmult_prj/solution0/impl/ip

[4] Et non 3 design dans un même Vivado

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

Pour chaque projet Vivado vous allez créer un SDK dans lequel vous pourrez comparer les performances de la solution obtenue à l'étape 1 (explorations no 1 à no 3) avec le ARM (Cortex A9). Alors qu'à l'étape 1 la simulation fonctionnelle se comparait avec une exécution sur votre machine (via *mmult_test.cpp*), ici on parle davantage d'un test de performance avec le Cortex A9.

Bref, lequel sera le plus vite? À vous de valider!

Ici vous êtes en terrain connu puisqu'il s'agit de manipuler SDK. Vous travaillerez toutefois en standalone (bare metal) plutôt qu'avec uC/OSIII (c'est encore plus simple!). Pour cela, utilisez *Standalone* pour OS Platform quand vous créer votre BSP. Puis, quand vous créer votre application, vous verrez qu'un hello world vient s'ajouter. Tout comme au lab 1, il permet de voir que votre BSP fonctionne bien. Une fois tester vous remplacerez *helloworld.c* par les 3 fichiers qui se trouve dans le répertoire *arm_sw* du code de départ (attention de ne pas détruire le *main.c* et les fichiers *platform.h*, *platform.c* et *platform_config.h*). 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 0x3000 (au besoin, voir page 29 du Application Note).

Par conséquent, pour chaque projet Vivado vous allez créer un SDK comme indiqué au paragraphe précédent. Prenez le temps de bien comprendre le code et voyez l'accélération que votre coprocesseur fournit.

Attention! Comme il est possible que vous ayez 3 SDK ouverts, assurez-vous qu'il n'y a qu'un seul SDK terminal d'ouvert et une seule exécution à la fois!

Et pour terminer, voici quelques questions qui complèteront ce laboratoire :

Question 6 : Décrivez comment se fait le test performance? En particulier, décrivez le rôle du *axi_timer* dans votre design, son fonctionnement et sa comparaisons avec le *fit_timer* du lab 1?

Question 7 : Décrivez le rôle du DMA et justifier le choix d'une interconnexion HP (plutôt qu'une interconnexion GP).

Question 8 : Donnez un *print screen* de chaque exécution sur SDK. Est-ce que l'accélération est au rendez-vous pour chaque résultat de l'étape 1?

Question 9 : Lorsque vous comparez les résultats des projets 1 et 2, sont-ils conformes à votre réponse donnée à la question 5.

Évaluation et remise du rapport

Deux séances d'évaluation auront lieu aux alentours du 20 avril. Vous devrez alors me montrer les 3 solutions de l'étape 1 et leur implémentation finale sur Vivado/SDK. Ça se fera par prise de rendez-vous. Je vous donnerai les plages et les détails.

Vous devrez aussi rendre un fichier .zip des répertoires hls avec les réponses aux 9 questions au plus tard le 20 avril (pas de rapport).