

Polytechnique Montréal

INF3610 Laboratoire 4 partie 2

Exploration de l'espace de solution logiciel/matériel

Frédéric Fortier - Arnaud Desaulty
04/04/2016

Table des matières

Introduction.....	2
I. Modifications notables depuis la partie 1.....	2
1. Modifications à apporter à votre projet existant.....	2
2. Images de tailles différentes	3
II. Démarrage de la simulation Simtek	3
III. Modes de communication	4
1. ModuleRead/ModuleWrite.....	4
2. Test des communications – Travail à effectuer	8
IV. Accélération du coprocesseur de calcul matriciel : de la simulation à la mise en œuvre	9
1. Synthèse de haut niveau versus synthèse logique	9
2. Génération d'un projet Vivado à partir de Simtek.....	10
3. Analyse des résultats produits par GenX en vue d'une implémentation ou d'une analyse de performance.....	15
a. <i>Fichier system.mhs</i>	16
b. Exécutable system.xmp (permet de faire la mise en œuvre sur la carte ZedBoard) ..	17
c. Fichiers report de Vivado HLS (permet de faire l'analyse de performance).....	17
V. Rendu	20

Introduction

Lors de la partie 1 du laboratoire, nous avons eu l'occasion de commencer à travailler sur le développement d'un filtre d'image. Il a fallu développer ce filtre et s'assurer de son bon fonctionnement sur le logiciel SpaceStudio. Pour ce faire, vous avez testé votre code à l'aide du simulateur UTF Elix.

Pour cette deuxième partie de laboratoire, nous allons procéder à une exploration succincte de l'espace de solutions logicielles/matérielles. Il s'agira dans un premier temps de tester différents modes de communications entre le contrôleur logiciel et le filtre matériel et de faire une analyse de performances de ces différentes solutions. Dans un second temps, il vous sera demandé de synthétiser le filtre de Sobel (et celui d'RGB -> niveau de gris) et de procéder à quelques optimisations sommaires afin de dégager des compromis latence/coût.

I. Modifications notables depuis la partie 1

1. Modifications à apporter à votre projet existant

Un nouveau projet SpaceStudio est fourni avec la partie 2 et ce code fourni a été légèrement modifié par rapport à celui donné pour la partie 1 (pour pouvoir fonctionner correctement en simulation Simtek, présentée à la section II). Ainsi, pour éviter les problèmes, assurez-vous de bien suivre les points suivants :

- Utilisez le projet de la partie 2, pas de la partie 1
- Récupérez tel quel le code des modules de la partie 1, sauf pour le fichier `BitmapRW.cpp` et pour les fichiers `Sobel.{cpp,h}`. Dans `BitmapRW`, vous devrez reprendre la partie « À compléter » que vous aviez faite dans la partie 1 pour la remettre à l'endroit marqué « À compléter » sur la partie 2. Quant au module Sobel, des modifications ont dû être apportées pour permettre la synthèse de haut niveau (que nous verrons au point IV.2) et impliquent de passer explicitement les tableaux de luminance et résultats à la fonction `sobel_operator` plutôt que de passer par une variable membre.
- Tel qu'illustré à la figure 1, sélectionnez la configuration Simtek « **linux** », puis allez dans *Tools/Preferences/linux/Zynq Simulation Arguments* et sélectionnez le script *fsinject.sh* (situé dans `Sobel/import`) à exécuter durant la création du système de fichiers.

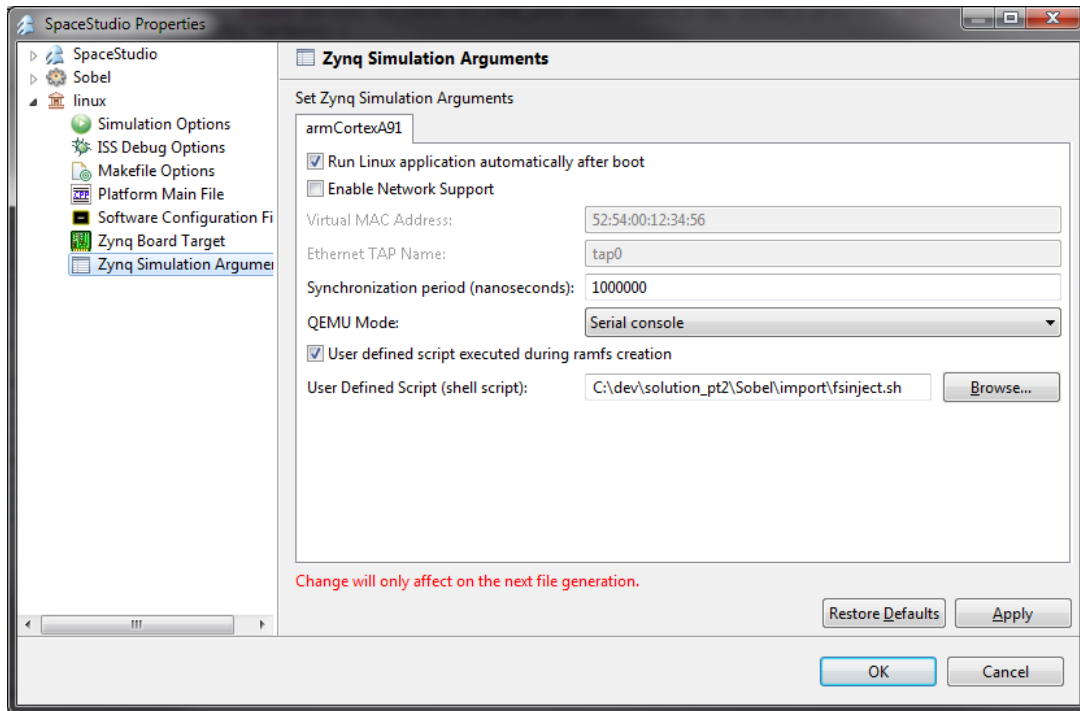


Figure 1

2. Images de tailles différentes

Des images de tailles différentes ont été fournies pour permettre de calculer les temps d'exécutions en fonction de la taille des données. Pour supporter des images de tailles différentes, vous devrez donc :

- Sélectionner les bons fichiers à ouvrir dans BitmapRW
- Modifier la taille de l'image dans le fichier ApplicationDefinitions.h du projet.
- Régénérer (**avec clear file cache coché**) et recompiler (rebuild all) le projet.

II. Démarrage de la simulation Simtek

Pour pouvoir estimer les performances de notre solution, il va falloir utiliser la technologie de simulation Simtek que vous avez pu voir lors du tutorial JPEG du laboratoire 1. Vous utiliserez la configuration *linux* déjà présente dans votre projet. La métrique à laquelle nous allons nous intéresser pour ce laboratoire est **le temps de simulation**. Pour que votre temps de simulation sous Simtek ait une signification, il faut procéder à plusieurs modifications sur votre code :

1. Vous devez être en mode monitoring, pour pouvoir produire les données d'usage de bus et des différentes tâches.
2. Il faut que la simulation s'arrête une fois le filtre terminé, c'est-à-dire quand le résultat est revenu vers le module *BitmapRW*, sans vérifier que celui-ci est bon (le code fourni s'en assure déjà si vous êtes en monitoring).

3. Ne laissez pas d’affichage dans votre code.

Notez que l’affichage en Simtek n’apparaîtra pas dans la console à moins d’appeler explicitement la fonction *wait(1)* qui force un *flush* de stdout. Cependant, comme la fonction *wait* affecte le temps de simulation (de même qu’un *SpacePrint*), les fonctions *waitIfNotMonitoring* et *SpacePrintIfNotMonitoring* ont été ajoutées (dans le fichier *ApplicationDefinitions.h*) et vous permettront d’écrire un code qui affichera correctement en mode *Release* (au détriment de l’exactitude des temps de simulation) et n’affichera rien mais aura un temps représentatif en mode *Monitoring*.

Une fois ces modifications effectuées, vous pouvez alors générer puis reconstruire votre projet. Des erreurs peuvent apparaître lors de ces deux étapes si vous n’avez pas coché la case *clear file cache* lors de la génération.

Il est possible que la première exécution soit dérangée par le pare feu Windows qui va vous indiquer que l’émulateur linux tente d’accéder au réseau. Toutefois, relancer l’exécution ne doit pas faire réapparaître ce pop-up.

III. Modes de communication

Comme expliqué précédemment dans la partie 1 du laboratoire, le canal de base de communication intermodule dans SpaceLib (l’API de SpaceStudio) est la FIFO. Quand vous utilisez les fonctions *moduleRead* et *moduleWrite* dans votre code, la génération automatique de code va instancier les files de communications associées à ces appels. Toutefois, l’API de SpaceStudio vous permet d’utiliser d’autres canaux qui peuvent accélérer l’exécution dans certains cas de figures.

1. ModuleRead/ModuleWrite

Par défaut, une paire d’appels *ModuleRead/ModuleWrite* va générer une file de communication entre les deux modules impliqués. Cette file va par défaut faire une taille de 64 mots de 32 bits. Il est cependant possible de modifier ces paramètres via le menu d’analyse des communications représenté dans la barre d’accès rapide par une loupe à l’extrême droite de la barre.

Tel qu’illustré à la figure 2, une fois l’analyse des communications terminée, une fenêtre s’ouvre et vous permet de sélectionner un canal particulier pour l’éditer :

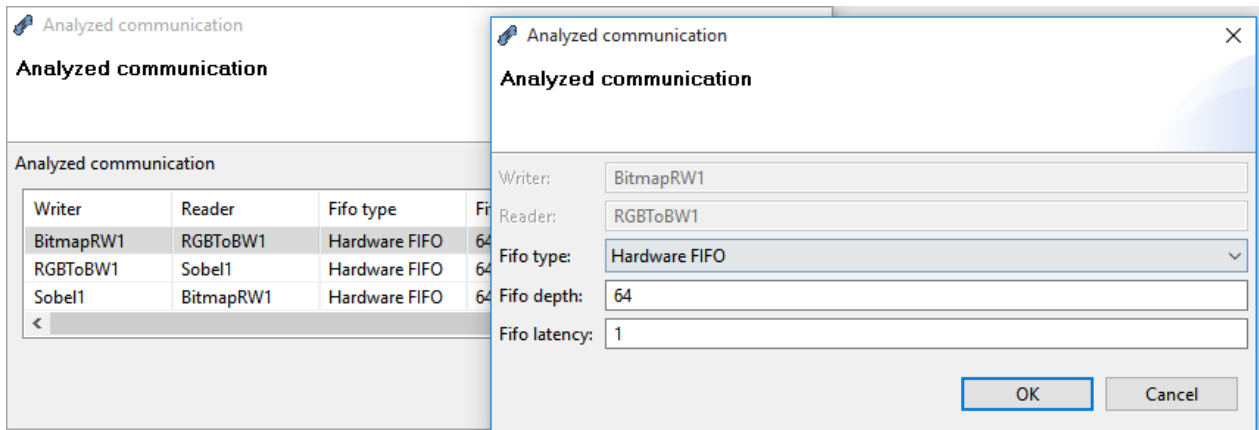


Figure 2

Dans ce menu, vous pouvez éditer le type de communication par FIFO de ce canal particulier et décider d'utiliser soit une *Hardware FIFO* soit le *DMA* instancié sur le FPGA. Vous pouvez, de plus, éditer la taille de la FIFO (notez que ce paramètre n'aura d'influence que si vous choisissez *Hardware FIFO* en tant que type de FIFO). Pour mieux expliquer comment se fait la transformation de FIFO HW à DMA, examinons les figures 3 et 4 provenant de la documentation de SpaceStudio:

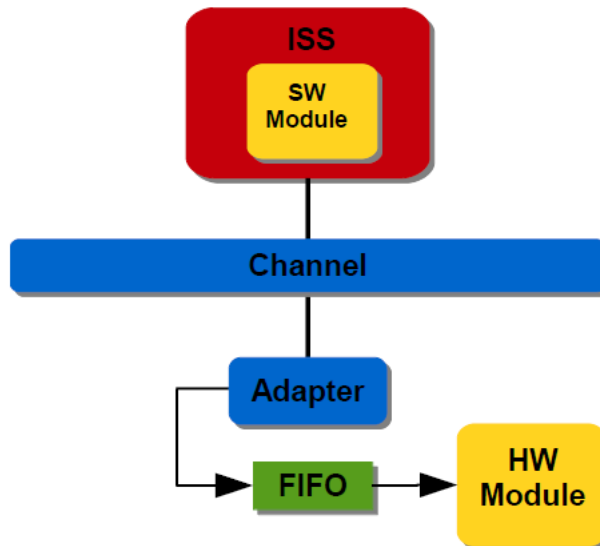


Figure 3 : Lecture du coprocesseur (HW) à travers un FIFO

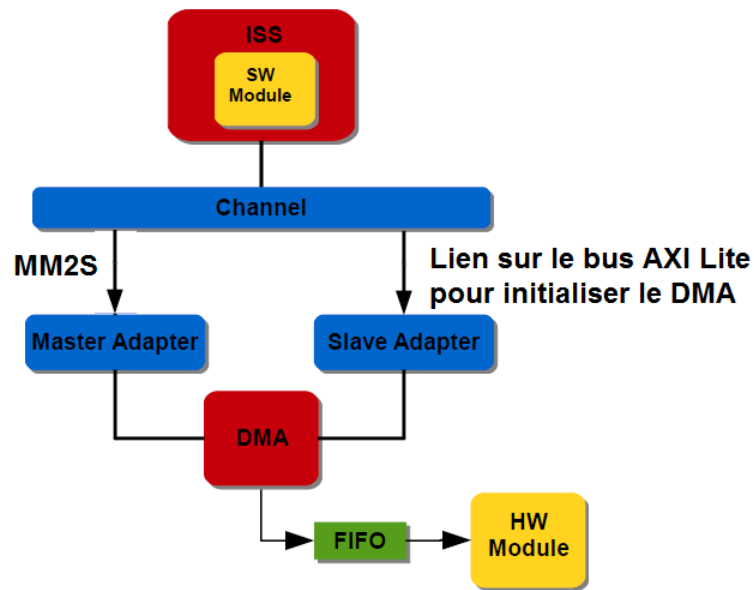


Figure 4 : Lecture du coprocesseur (HW) via le DMA (MM2S pour memory2stream)

La première figure réalise une communication entre une tâche logicielle (producteur) et matérielle (consommateur) à travers le FIFO, c'est donc le ARM qui est responsable de produire les données à partir de sa mémoire. Dans la figure suivante, on a également un modèle producteur/consommateur, mais c'est le DMA qui ira chercher les données en mémoire (c'est pourquoi on a un master adapter). Le processeur est donc libéré de ce travail (et peut faire autre chose). Le lien MM2S assure donc les communications entre mémoire et DMA. Pour le Zynq, le DMA peut supporter 3 sources de mémoire différentes. Notez également que du côté droit on a un slave adapter car le processeur qui est alors le maitre initialise le DMA via un bus AXI Lite¹. Donc tout comme nous l'avons vu en classe avec le bus OPB, le DMA a à la fois un adaptateur maitre et un adaptateur esclave.

Également si on fait des écritures du module HW vers la mémoire via DMA, on aura le flot illustré à la figure 5.

Notez finalement que le DMA peut simultanément supporter stream2memory et memory2stream, mais 2 adaptateurs maitres sont alors requis.

La Figure 6 présente le bloc diagramme bloc du DMA provenant de la librairie Xilinx (donc programmé sous FPGA). Notez bien sur cette figure, les losanges qui illustrent les connexions des figures 4 et 5.

¹ Nous verrons en classe ce qu'est le bus AXI Lite.

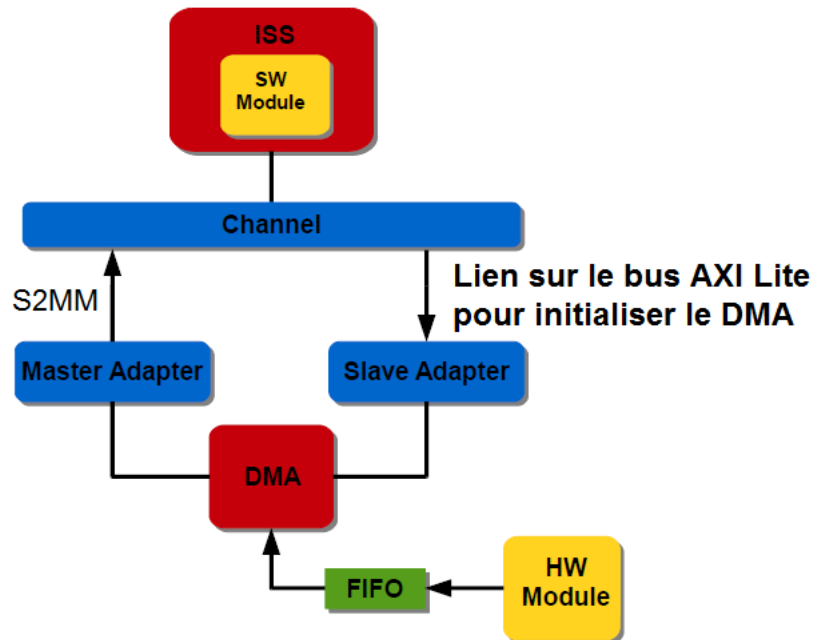


Figure 5 : Écriture du coprocesseur (HW) via le DMA (S2MM pour stream2memory)

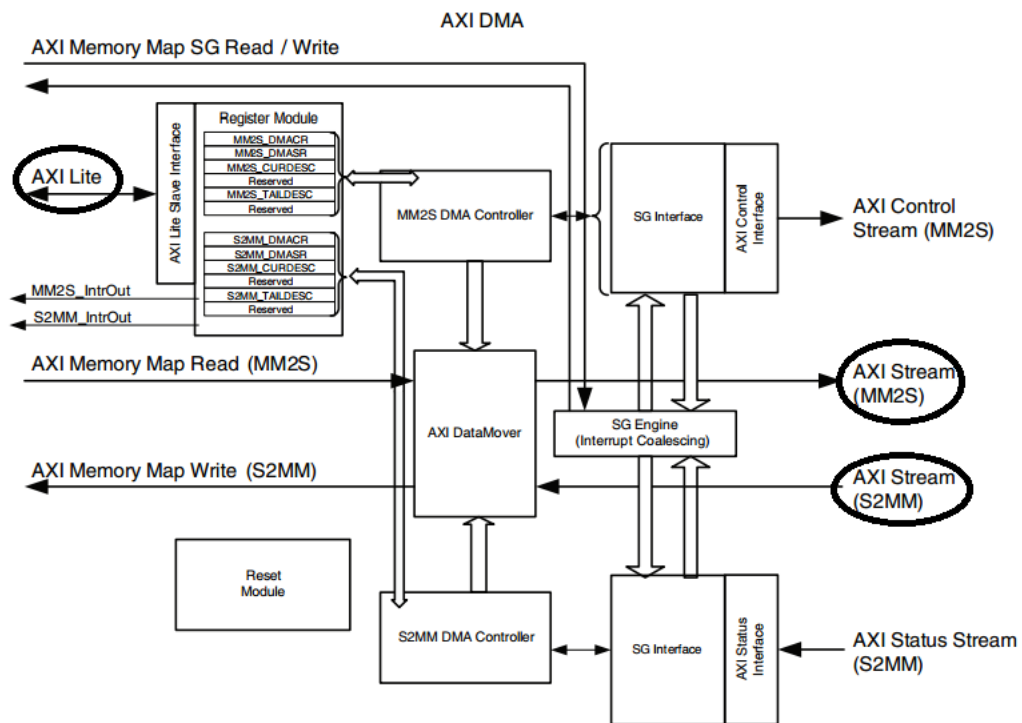


Schéma général du DMA utilisé dans ce laboratoire (source : http://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf)

Pour le labo, nous vous demanderons d'utiliser dans un premier temps les FIFO hardware, puis le DMA.



Encart technique : DMA ou Direct Memory Access

Le DMA est un composant matériel présent dans de nombreux systèmes informatiques pouvant prendre en charge les transactions mémoire (read et write) qu'un processeur aurait dû effectuer. Cela a deux effets principaux : la libération du processeur qui peut alors effectuer d'autres tâches le temps du transfert et l'accélération du transport des données grâce au traitement par le DMA, qui est optimisé pour les transferts mémoire.

Pour une introduction au DMA, voir aussi :

<http://www.embedded.com/electronics-blogs/beginner-s-corner/4024879/Introduction-to-direct-memory-access>

2. Test des communications – Travail à effectuer

Il vous est demandé d'effectuer les tests de performances décrits dans le tableau ci-dessous suivant 2 configurations : la première est **la communication par Hardware FIFO** et la seconde en **remplaçant le type de FIFO par DMA**. Vous ferez ces tests sur des images de tailles différentes, qui sont fournies. La procédure pour modifier l'image chargée dans la simulation est décrite à la section II.3

	36x36	100x100	500x500	1920x1080
Config 1				
Config 2				

Tableau 1 Estimation des temps de communication

Notez que les temps de simulations peuvent être assez longs (de l'ordre de la dizaine de minutes pour une image 1920x1080).

Une fois ces valeurs recueillies, **fournissez un graphique du temps d'exécution en fonction du nombre d'éléments à traiter** lors des communications. Dessinez les courbes des 2 configurations sur le même graphique. Fournissez vos premières observations quant aux différents modes de communications.

Question 1. Faites une génération en *monitoring* des configs 1 et 2 pour l'image 1920x1080 et analysez l'utilisation processeur dans les deux cas (via l'utilitaire *Monitoring Explorer*). Qu'observez-vous ? Cela vous semble-t-il cohérent avec les modes de communication utilisés ?

Question 2 Vous attendiez-vous à une plus grande différence entre les résultats des deux modes et si oui, comment expliquez-vous que cela ne soit pas le cas ?

IV. Accélération du coprocesseur de calcul matriciel : de la simulation à la mise en œuvre

Nous avons vu en classe qu'en déroulant les boucles, il est possible d'accélérer le calcul d'une opération sur processeur superscalaire comme le Cortex A9, à condition d'avoir un bon compilateur. Lorsque l'on porte une fonction du Cortex A9 vers du matériel (ici FPGA), il est également possible de procéder à cette opération de déroulement de boucles dans un but d'accélération. Si la fonction s'y prête bien, le gain est en général plus important que sur le superscalaire. Pourquoi? C'est ce que nous allons observer dans la suite de cette section à l'aide de la plateforme de simulation SpaceStudio.

Dans un premier temps nous allons rapidement² décrire ce qu'est la Synthèse de Haut Niveau (SHN, HLS pour *High Level Synthesis* en anglais) et la différencier de la synthèse logique (SL). Ensuite nous verrons qu'à partir de SpaceStudio, il est possible de générer une vue complète du code exécutable (partie logicielle) et RTL (partie coprocesseur) incluant donc système complet (filtre de Sobel). De cette vue, on peut: 1) utiliser Vivado pour une mise en œuvre (implémentation) du filtre de Sobel sur FPGA (carte Zedboard) et/ou 2) faire l'analyse des performances de la mise en œuvre de votre configuration afin d'améliorer la solution (ex. par rapport au cahier de charge). Étant donné le temps disponible pour ce laboratoire, nous allons expérimenter la partie 2). La partie 1) serait fort intéressante à réaliser mais elle risque de demander trop de temps (considérant qu'une synthèse avec Vivado peut prendre plusieurs heures). Toutefois, le cours INF8500 vous permettra d'expérimenter cette partie.

1. Synthèse de haut niveau versus synthèse logique

Vous avez expérimenté en INF1500 et INF3500 les produits de Aldec (Active-HDL) qui permettent la synthèse logique. Cette dernière est un processus par lequel une description du circuit, généralement au niveau de transfert de registre (RTL), est transformée en une mise en œuvre de la conception en termes de portes logiques, typiquement par un logiciel. Ces outils de synthèse (et bien d'autres) génèrent des flux binaires pour dispositifs logiques programmables tels que les FPGA, tandis que d'autres visent la création d'ASIC.

Alors que la synthèse logique utilise une description de niveau RTL, la synthèse de haut niveau (SHN) fonctionne à un niveau d'abstraction plus élevé, à commencer par une description algorithmique dans un langage de haut niveau tels que SystemC et ANSI C / C++. La synthèse de SHN, parfois appelée la synthèse comportementale (par opposition à structurelle pour la synthèse logique), est donc un processus de conception automatisé qui interprète une description algorithmique d'un comportement souhaité et crée le matériel numérique pour mettre en œuvre

² Une introduction sera aussi donnée dans le cours INF3610, mais pour approfondir vos connaissances sur le fonctionnement de la synthèse de haut niveau, suivre le cours INF8500.

ce comportement. La sortie d'un outil SHN est une description SystemC, Verilog ou VHDL au niveau RTL, qui peut ensuite être utilisé en entrée de la synthèse logique. La synthèse logique n'est pas appelé à disparaître, mais on risque de voir de plus en plus de SHN au cours des prochaines années dans le domaine de la conception des systèmes embarqués. (Vous pouvez comparer la synthèse logique et la SHN, respectivement à l'assembleur et au langage C++. L'assembleur fait toujours partie de la chaîne de compilation logiciel, mais le concepteur interagit d'abord avec le C++.)

2. Génération d'un projet Vivado à partir de Simtek

Dans SpaceStudio (niveau Simtek), il est possible d'invoquer GenX (*Generate Xilinx*) pour mettre en œuvre la réalisation du système (le filtre) sur Zedboard. SpaceStudio génère le code logiciel applicatif qui roulera sur le Cortex A9, il génère les appels systèmes Linux (tout en choisissant le bon portage (BSP, etc.)), les drivers (aussi nommé firmware) et le bootloader. Par contre SpaceStudio ne fait pas de SHN pour la génération RTL du code applicatif que l'on désire mettre en matériel (comme c'est le cas ici pour le filtre de Sobel) mais il est possible d'invoquer un outil de SHN pour faire ce travail. Ici l'outil qui sera invoqué est Vivado HLS³.

Voici donc les étapes pour passer d'une configuration Simtek à un projet Vivado en passant par Vivado HLS :

1. **Pour toute la section III. Vous utiliserez la configuration 2 (DMA) de la section II et une image de 100x100**
2. Mettre active votre configuration Simtek : cliquez sur **Architecture/Configuration** → **Simtek Configurations** → < nom de la configuration >
3. Allez dans *Tools>Preferences* et cochez la case *EDA is Enabled* dans *GenX>EDA>ISE14.4*. Profitez pour modifier les différents paths pour que ceux-ci correspondent aux paths de la machine (*normalement C:/Logiciels/...*).
4. Toujours dans *Tools>Preferences*, cochez la case *HLS is enabled* dans *HLS>Vivado HLS*
5. Dans cette configuration assurez-vous d'avoir mis les étiquettes L1 et L2 devant les deux boucles appelant *sobel_operator* pour chaque pixel, ainsi que l'étiquette L3 et L4 devant chaque boucle for de votre filtre de Sobel (nous en aurons besoin plus tard) :

```
L1:for (int i = 1; i < IMG_HEIGHT - 1; ++i) {  
L2: for (int j = 1; j < IMG_WIDTH -1; ++j) {
```

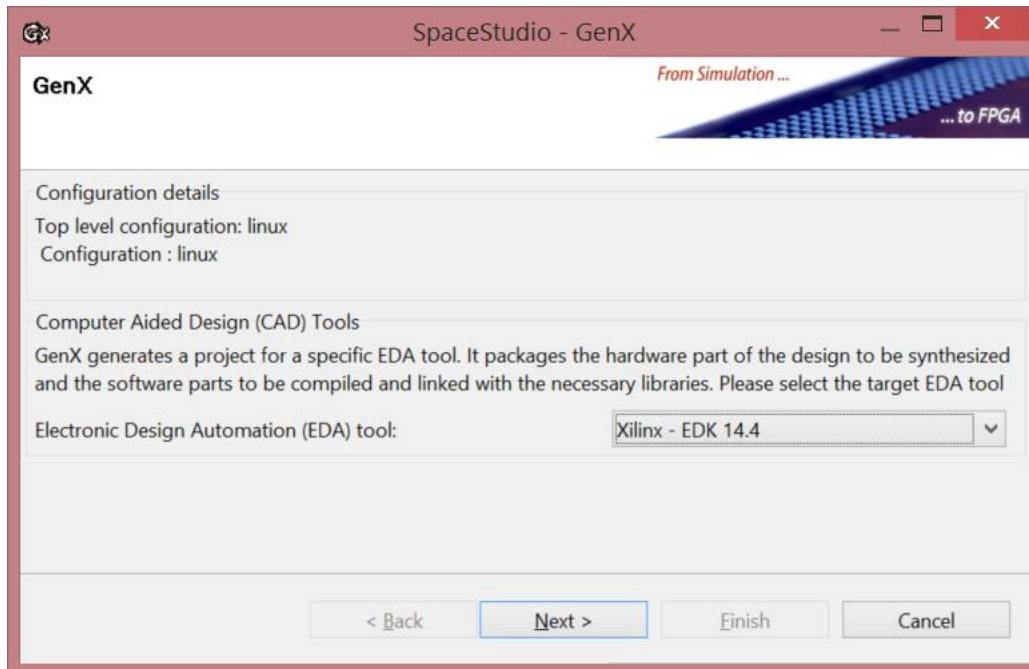
Code 1 Boucles d'appel de sobel_operator

```
L3: for(int i = 0; i < 3; i++){  
L4: for(int j = 0; j < 3; j++){
```

Code 2 Boucles nécessaires au filtre de Sobel sur un pixel

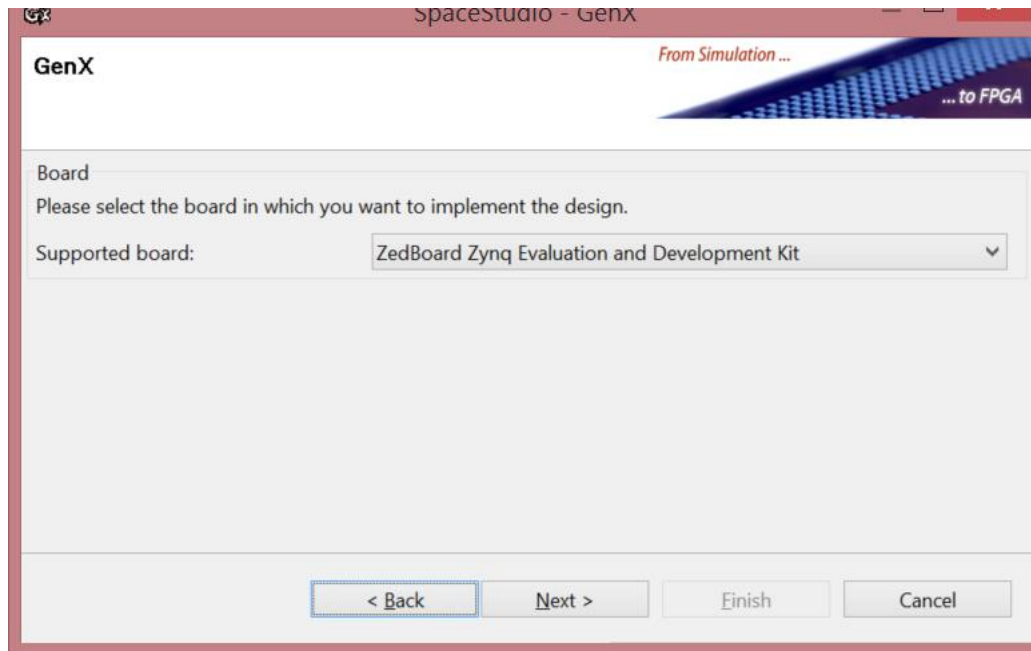
³ Encore une fois, dans ce laboratoire Vivado HLS sera utilisée pour démontrer la possibilité de passer du C++/SystemC au RTL, mais si vous désirez en savoir davantage sur cet outils et son fonctionnement, suivre le cours INF8500.

6. De plus, assurez-vous que toute la mémoire de vos modules Sobel et RGBToBW est **en allocation statique** car le synthétiseur ne peut synthétiser des variables allouées dynamiquement (il faut que la taille des variables soit connue à la compilation)
7. Cliquez sur **Architecture/Configuration → Hw Section → GenX**
Ici la génération est appelée afin d'analyser les différents composants du système dont principalement les communications. Ensuite la fenêtre suivante devrait apparaître :



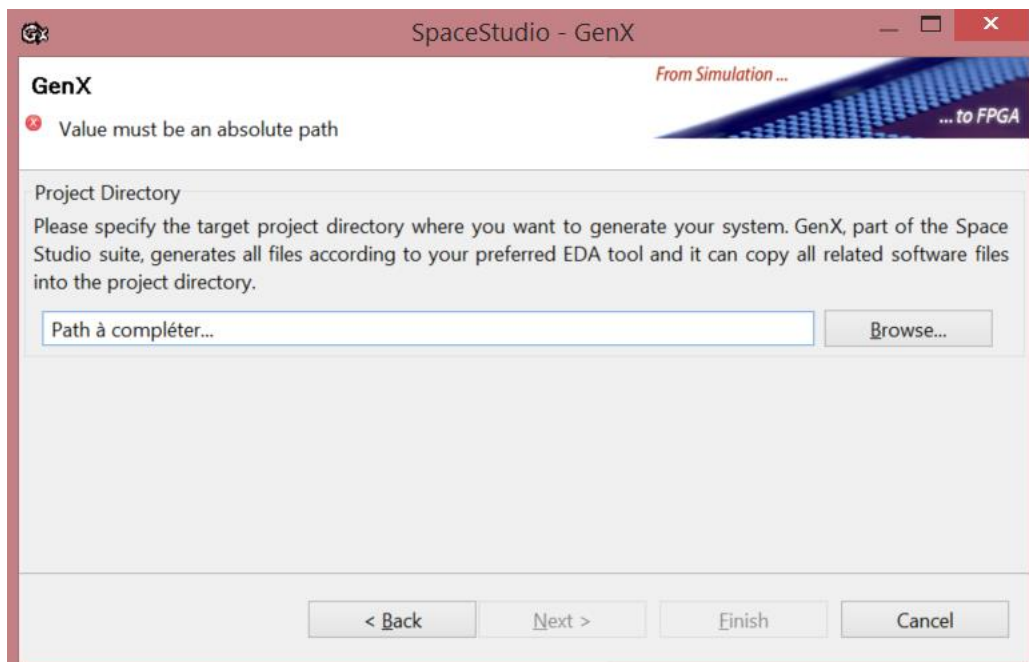
Il s'agit donc de sélectionner la version d'EDK qui sera utilisée pour la génération. Choisissez EDK 14.4 et cliquez sur **next**.

8. Une deuxième fenêtre apparaît alors :



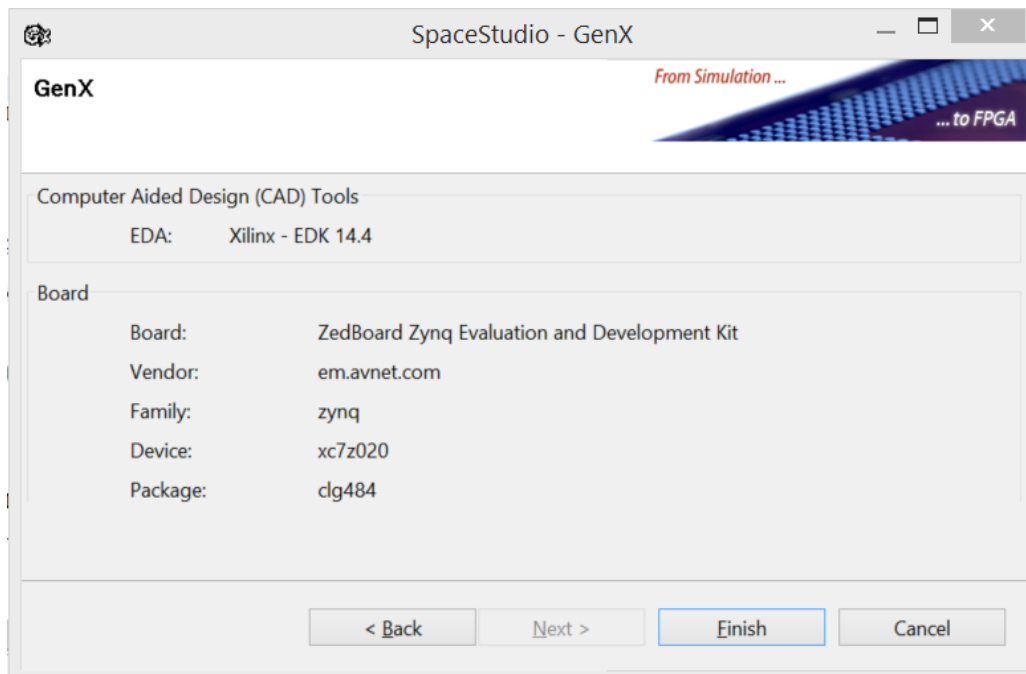
Ici vous devez choisir la carte ciblée, sélectionnez **Zedboard Zynq Evaluation and Development Kit**, puis faites **Next**.

9. Vous devez ensuite indiquer le répertoire où sera déposé le projet :



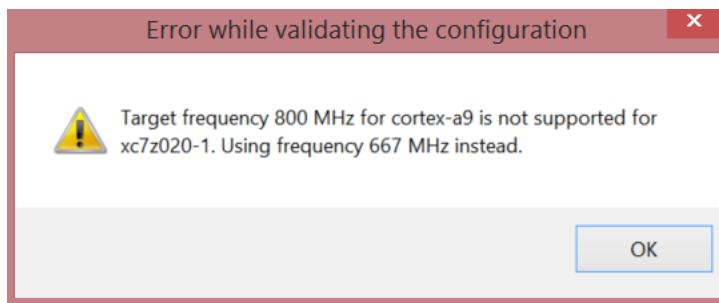
Ici créez un répertoire implémentation à l'intérieur de votre projet, puis cliquez sur Next.

L'outil vous fera alors un résumé des paramètres sélectionnés :



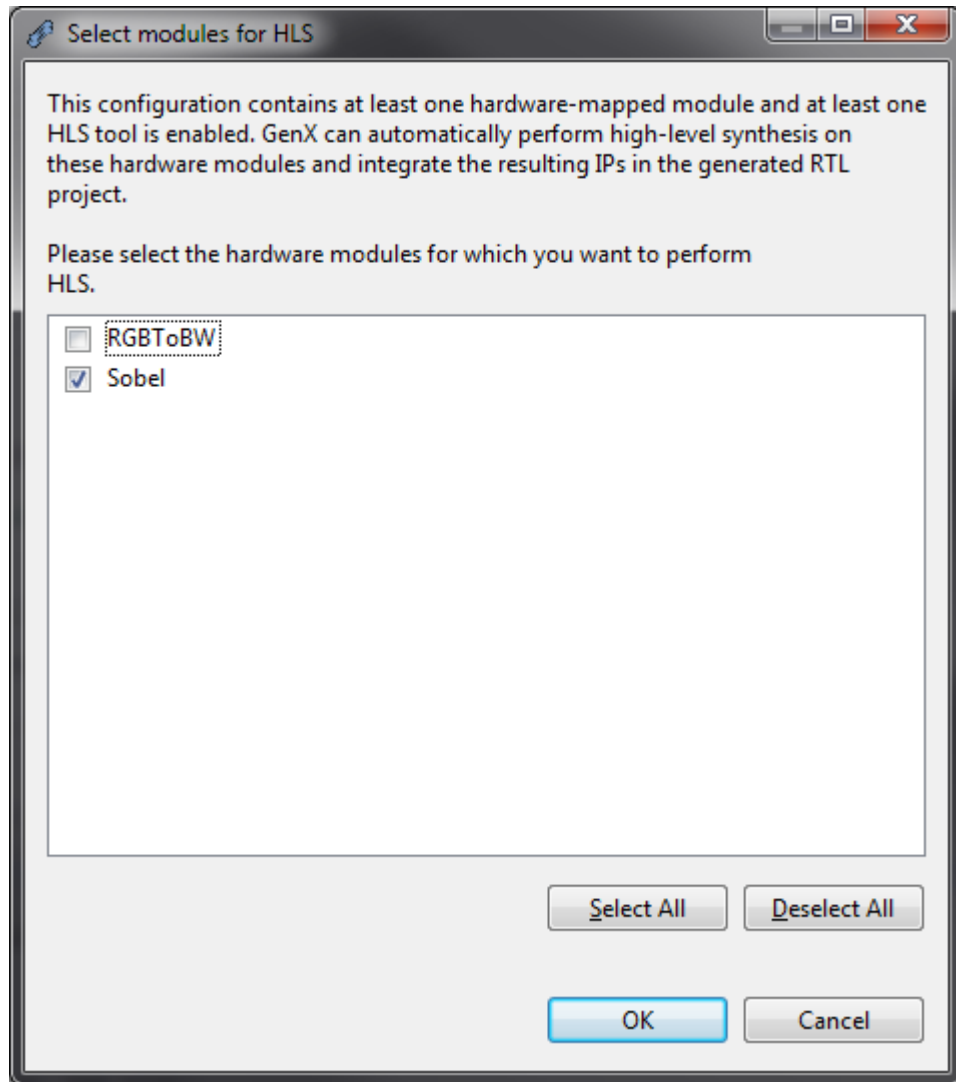
Cliquez sur Finish.

10. Un warning apparaîtra :



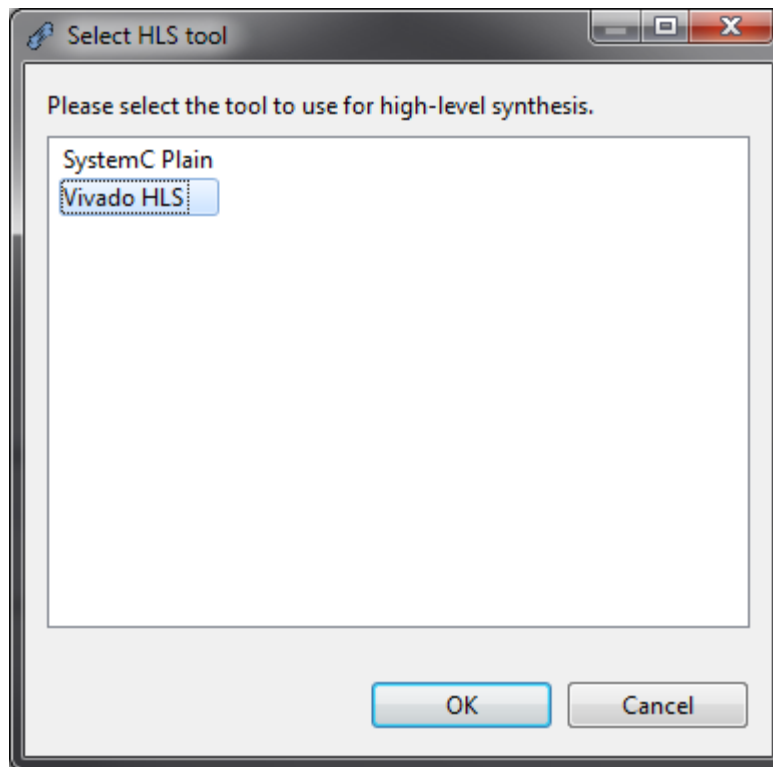
Cliquez simplement sur OK.

À partir de ce moment, pour chaque composant du système, GenX cherche un équivalent dans la librairie CoreGen de Xilinx de chaque composante du système (processeur ARM et ses périphériques, BRAM, etc.). Toutes les composantes pour lesquelles il n'y a pas de correspondance vont apparaître dans la fenêtre suivante :



11. Dans cette configuration on a 2 modules qui sont assignés comme matériel (coprocesseur). Ce que nous indique cette figure est que ces 2 modules n'ont pas de correspondance dans la librairie mais que GenX a trouvé un outil SHN sur l'ordinateur pour synthétiser ces blocs. Pour cette section de laboratoire, nous ne nous intéressons qu'à *Sobel* et nous ne sélectionnerons donc que ce module pour indiquer à GenX de procéder à la synthèse SHN (l'expérimentation des étapes qui suivent sur le module *RGBToBW* est laissé en exercice mais ne sera pas évalué). Cliquez ensuite **OK**.
12. La fenêtre suivante confirme que Vivado HLS a bel et bien été trouvé⁴ :

⁴ GenX supporte aussi d'autres outils de SHN tel que CatapultC. Si cette dernière avait été disponible sur votre machine, GenX aurait offert le choix entre les deux.

















Cliquez simplement OK.

À partir de là, GenX appelle Vivado HLS et lui demande de synthétiser *Sobel*. Si tout va bien, cette synthèse devrait se terminer par un **Done**.

3. Analyse des résultats produits par GenX en vue d'une implémentation ou d'une analyse de performance

Dans un premier temps, nous allons regarder le contenu du projet Vivado généré par GenX. Ouvrez une fenêtre et allez dans le répertoire *implementation* (étape 9 de la procédure pour GenX). Vous devriez observer les fichiers suivants :

Nom	Modifié le	Type	Taille
 _xps	2016-03-26 18:50	Dossier de fichiers	
 data	2016-03-26 18:50	Dossier de fichiers	
 etc	2016-03-26 19:03	Dossier de fichiers	
 hls	2016-03-26 18:06	Dossier de fichiers	
 implementation	2016-03-26 18:50	Dossier de fichiers	
 pcores	2016-03-26 18:05	Dossier de fichiers	
 Sobel_armCortexA91	2016-03-26 18:05	Dossier de fichiers	
 ps_clock_registers.log	2016-03-26 18:50	Document texte	4 Ko
 system.log	2016-03-26 19:03	Document texte	1 Ko
 system.make	2016-03-26 18:50	Fichier MAKE	8 Ko
 system.mhs	2016-03-26 20:25	Fichier MHS	25 Ko
 system.mss	2016-03-26 20:25	Fichier MSS	2 Ko
 system.xmp	2016-03-26 20:25	Xilinx Platform Stu...	1 Ko
 system_incl.make	2016-03-26 18:50	Fichier MAKE	6 Ko

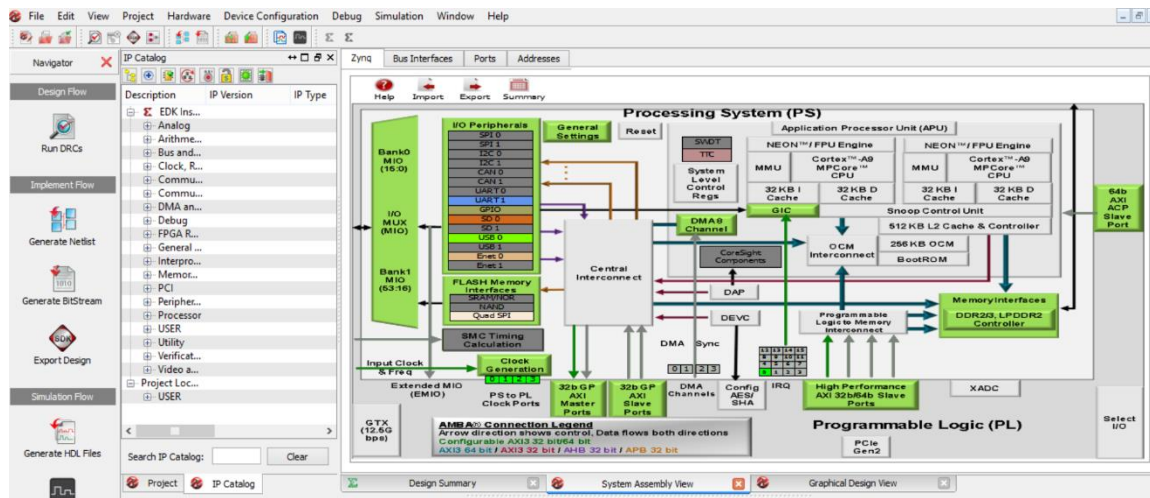
a. *Fichier system.mhs*

Dans ce fichier vous retrouverez une version détaillée (RTL) du main.cpp de Simtek. Plus précisément vous y retrouverez les composantes telles que le DMA. Prenez le temps de trouver cette composante et répondez à la question suivante :

Question 3. Basé sur le schéma général du DMA (Section I.1), indiquez les connexions de AXI Lite aux DMA.

b. Exécutable `system.xmp` (permet de faire la mise en œuvre sur la carte ZedBoard⁵)

Cet exécutable permet de démarrer Xilinx Platform Studio. La fenêtre de départ devrait être la suivante :



Si l'on désirait créer le *bitstream*, mettre ce dernier dans une mémoire flash et finalement de démarrer la carte Zedboard, il faudrait travailler à partir de cette fenêtre (en invoquant au départ *Generate BitStream*).

c. Fichiers `report` de Vivado HLS (permet de faire l'analyse de performance)

Cette dernière partie va vous permettre de faire l'analyse des performances de la mise en œuvre de votre configuration afin d'améliorer votre solution. D'abord allez dans le répertoire `hls\Sobel\Sobel\solution1\syn`

Vous trouverez 4 sous-répertoires : *report*, *systemc*, *verilog* et *vhdl*. Les 3 derniers contiennent la description RTL du module *Sobel* en SystemC, Verilog et VHDL. Il s'agit donc de la sortie de Vivado HLS. Par exemple si vous ouvrez le fichier *Sobel.cpp* du répertoire *systemc* vous verrez une description SystemC RTL générée automatiquement (un peu cryptique...). Notez l'utilisation de `SC_METHOD` (et non de `SC_THREAD`). De même si vous ouvrez le fichier *Sobel_thread.vhd* du répertoire *vhdl*, vous trouverez une description RTL VHDL de *Sobel_thread*.

Le dernier répertoire est le répertoire *report*. Il s'agit en fait du rapport de synthèse effectué par Vivado HLS. Allez donc dans ce répertoire. Le fichier qui nous intéresse est le fichier *Sobel_thread_csynth.rpt*. Ouvrez-le. Dans la section *Performance Estimates* de ce fichier vous devriez trouver le nombre d'itérations de chaque boucle de *Sobel*. Ici nous nous intéressons au calcul de L1, L2, L3 et L4. Vous devriez avoir quelque chose qui ressemble à ça (cas d'une image 100 x 100):

⁵ Cette sous-section qui correspond à l'implémentation sur carte à titre d'informative et est approfondie dans le cours INF8500 et le projet intégrateur INF3995.

```

+ L1:
  * Trip count: 98
  * Latency:    441980
+ L2:
  * Trip count: 98
  * Latency:    4508
+ L3:
  * Trip count: 3
  * Latency:    42
+ L4:
  * Trip count: 3
  * Latency:    12

```

Ce que nous indique cette partie du rapport est qu’après synthèse sur FPGA (Zedboard), pour effectuer un calcul complet **du cœur** d’un filtre Sobel 3x3 sur une image 100x100 (en ignorant les bords de l’image), il faudra 441980⁶ cycles. Si on se concentre sur les deux boucles du filtre sur un pixel (L3 et L4), on observe que l’application du filtre sur un pixel demande 12 cycles (42/3). La question est la suivante pourrait-on faire mieux? La réponse est évidemment oui. Une façon de faire est de demander à Vivado HLS un déroulement de boucles. Si aucune dépendance de données n’existe (aléas de données vus en classe), Vivado HLS pourrait le faire. Mais il faut le lui indiquer par le biais d’une directive. Cette directive peut prendre la forme d’un *pragma*. Vous allez donc retourner à la section III.2 (étape 2)⁷ et ajouter les pragma suivant dans Sobel :

```

L3: for(int i = 0; i < 3; i++){
#pragma HLS unroll
  L4: for(int j = 0; j < 3; j++){
#pragma HLS unroll

```

Ces *pragmas* indiqueront à Vivado HLS de dérouler complètement les boucle L3 et L4. Donc la multiplication par un filtre 3x3 se fera complètement en parallèle. Mais il faudra toutefois accumuler les résultats de ces multiplications (ceci peut demander un certain délai car dans ce cas d’accumulation, il y a dépendances...).

Refaites les étapes 1) à 13) de la section III.2 mais **ATTENTION, sauvez le projet généré par Genx (étape 9) dans *implementation2***. Ainsi vous pourrez comparer les fichiers *Sobel_thread_csynth.rpt* du répertoire *report*, avec ou sans déroulement de boucle. Lorsque vous ferez cette comparaison vous verrez que les boucles L3 et L4 ont disparu. Ceci est normal, puisqu’elles sont parallélisés.

⁶ Ce résultat peut varier légèrement selon votre implémentation

⁷ Pour sauver du temps, vous pouvez aussi simplement ouvrir `hls\Sobel\Sobel\vivado_hls.app` et ajouter ces pragmas au code généré par SpaceStudio, mais nous ne détaillerons pas plus cette procédure ici (elle est cependant assez simple).

Finalement, refaites une dernière fois les étapes 1) à 13) de la section III.2 mais en remplaçant le déroulage des boucles L3 et L4 par une directive de *pipelining* à l'intérieur de la boucle L2, comme ceci.

```
L2: for (int j = 1; j < IMAGE_WIDTH - 1; ++j) {  
    #pragma HLS pipeline
```

Vous remarquerez encore une fois sur dans le fichier rapport l'absence des boucles L3 et L4, puisque celles-ci auront complètement été complètement « aplaties » par le processus de *pipelining*.

Répondez aux questions suivantes :

Question 4. Donnez les accélérations par rapport à la solution sans pragma de déroulement de boucles ou de pipeline.

Question 5. Ces accélérations correspondent-elles à ce que vous vous attendiez? Que pouvez-vous conclure?

Question 6 (Bonus). Dépassez le x9 d'accélération sans dérouler L1 ou L2 ou modifier l'algorithme (suggestion: regardez les directives HLS présentées en cours).

Finalement, nous allons terminer cette section en améliorant la précision de la simulation Simtek de votre configuration. Vous aurez sans doute remarqué qu'il n'y a aucun délai dans le module de Sobel (dans Simtek). En effet, si l'on n'indique pas de délai à SpaceStudio, il considère que l'exécution est instantanée du point de vue simulation. Vous allez donc pour chaque configuration (avec et sans déroulement de boucle) ajoutez une annotation de temps venant du rapport Vivado HLS.

Pour le cas sans déroulement de boucle, on a vu que 441982⁸ cycles sont requis pour une image 100x100. Vous allez donc ajouter l'instruction *computeFor(441982);* après l'application du filtre sur l'image (juste avant le ModuleWrite vers BITMAPRW_ID). Pour réduire l'impact de la lecture et du décodage de l'image bitmap sur vos résultats, mettez aussi votre code envoyant l'image et recevant l'image filtrée (dans BitmapRW, de l'envoi de RGB à RGBToBW à la réception dans de Y) dans une *for loop* s'exécutant 50 fois. Refaites une simulation complète de votre configuration (i.e. Generate, Re-Build all et Execute). Prenez en note la valeur indiquée à *Simulation has ended @...*

Refaites cette simulation en remplaçant la valeur du paramètre passé à *computeFor()* par vos temps en déroulant les boucles L3 et L4 et en pipelinant le contenu de L2.

⁸ Remplacez par le nombre de cycles qu'il faut à votre implémentation

Question 7. Comparez le temps de simulation avec et sans déroulage de boucle. Expliquez la différence obtenue par rapport aux résultats de la question 4.

Question 8. Proposez (sans l'implémenter) une manière d'augmenter significativement les performances obtenues et le gain relatif avec/sans déroulage de boucle, qui réduirait aussi les ressources matérielles utilisées.

V. Rendu

Pour ce laboratoire, nous vous demandons de remettre, une fois par groupe de deux, un rapport contenant les tableaux et graphiques à compléter, une brève analyse de ceux-ci, ainsi que la réponse aux questions 1 à 8. Soyez précis dans vos réponses car la pondération sera fortement axée sur l'analyse de vos résultats. N'hésitez à inclure des graphiques supplémentaires si ceux-là vous semblent pertinents. Remettez aussi le code source de vos modules.