

TP Vivado HLS

Objet du TP

Le but de ce TP est de développer sous Vivado HLS une implémentation matérielle efficace d'un filtre gaussien. Le filtre à implémenter correspond au noyau de convolution suivant :

$$K = \begin{pmatrix} 0.0608 & 0.1496 & 0.0608 \\ 0.1496 & 0.3680 & 0.1496 \\ 0.0608 & 0.1496 & 0.0608 \end{pmatrix}$$

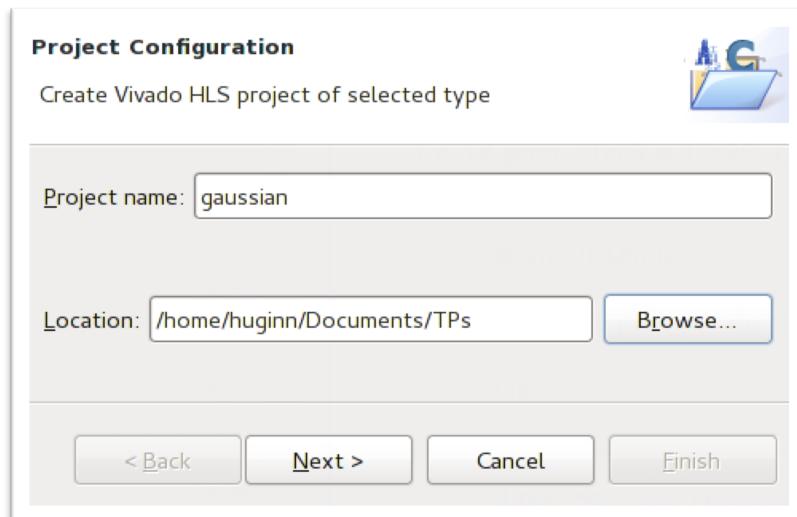
Le matériel généré devra pouvoir traiter 30 trames/s d'une résolution de 1920x1080 (1080p30).

Prise en main et première approche

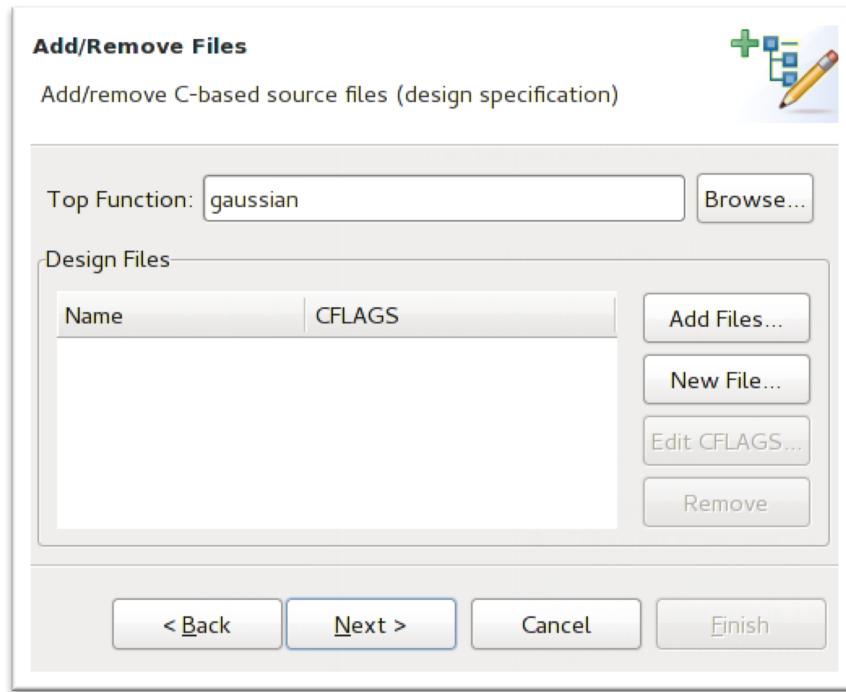
Création du projet

Pour créer le projet, suivez les étapes suivantes :

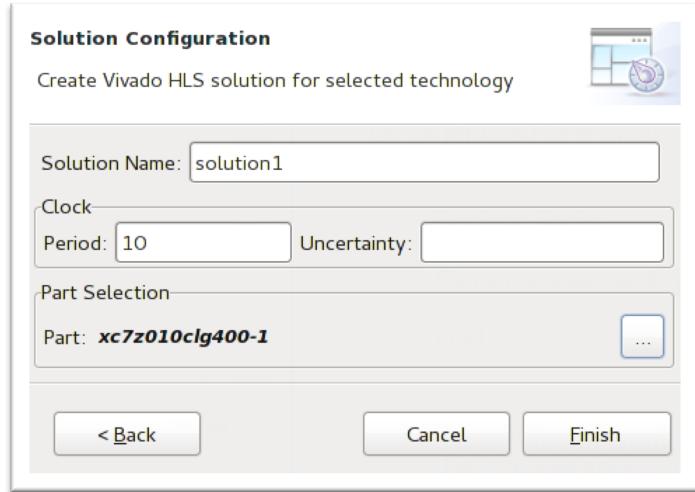
1. Décompressez l'archive **TP_Vivado_HLS.zip** contenant certaines librairies ainsi que les sources à utiliser au cours du projet.
 - a. Créez le répertoire de votre projet (par exemple `~/TPs/Vivado/`)
 - b. Copiez le répertoire **TP_Vivado_HLS\src** dans le répertoire du projet.
2. Démarrez Vivado HLS en utilisant le script **TP_Vivado_HLS\start_vivado**
 - a. `cd ~/.../TP_Vivado_HLS` (*remplacez les ... par le chemin menant au dossier décompressé*)
 - b. `chmod a+x start_vivado`
 - c. `./start_vivado`
3. Sélectionnez « **File -> New project...** ».
4. Entrez « **gaussian** » comme nom de projet. Sélectionnez l'emplacement où vous souhaitez créer le répertoire du projet, puis cliquez sur « **Next** ».



5. Le prochain écran vous permet de spécifier la fonction principale (« **top function** ») de votre design (celle dont vous souhaitez réaliser une implémentation matérielle).
 - a. Entrez simplement « **gaussian** » comme nom de fonction principale
 - b. Cliquez sur « **Add Files...** », sélectionnez tous les fichiers contenus dans le répertoire « **src** » copié précédemment, puis cliquez sur « **OK** ».



6. Le prochain écran permet d'ajouter des fichiers de test afin de valider la spécification logicielle. Afin de vous permettre de vérifier vos implémentations, un fichier de test a été préparé par nos soins. Cliquez sur « **Add Files...** » et sélectionnez le fichier « **TP_Vivado_HLS\test\test.cpp** ».
7. Le dernier écran vous permet de configurer le matériel utilisé pour la première « solution ». Outre une cible matérielle, une solution est caractérisée par l'ensemble des directives de compilation fournies par le concepteur. En général, plusieurs solutions sont évaluées au cours du cycle d'exploration jusqu'à ce que les contraintes de conception soient satisfaites.
 - a. Cliquez sur le bouton « ... » dans la zone « **Part Selection** » et entrez **xc7z010clg400-1** dans le champ recherche. Sélectionnez l'unique résultat, puis cliquez sur Ok.
 - b. Cliquez sur « **Finish** » pour terminer et sortir de l'assistant.



Ajout des fichiers sources

Le projet est maintenant créé avec ses fichiers sources et son fichier de test.

- Ouvrez le fichier « **gaussian.cpp** » en double-cliquant sur son nom dans l'explorateur :

```
#include "globals.h"

...

void gaussian(unsigned short in[H][W], unsigned short out[H][W]) {
#pragma HLS INTERFACE ap_bus port=in
#pragma HLS INTERFACE ap_memory port=out

    gaussian_3x3(in, out);

    // gaussian_sep(in, out);
    // gaussian_loopFusion(in, out);
    // gaussian_1line(in, out);
    // gaussian_scalarized(in, out);
    // gaussian_pipeline(in, out);
    // gaussian_caching(in, out);
    // gaussian_fixedPoint(in, out);
}
```

La fonction *gaussian(...)* est le point d'entrée de votre design. Pour l'instant, elle appelle simplement la fonction *gaussian_3x3(...)* avec les mêmes paramètres. La fonction *gaussian_3x3(...)* correspond à une implémentation naïve du filtre gaussien, que vous pouvez consulter. **Au cours du TP, vous remplacerez successivement cet appel par l'un de ceux commentés en dessous**, à mesure de vos implémentations. Le code de chaque fonction est à écrire dans le fichier de même nom.

Les deux lignes commençant par « **#pragma** » sont des **directives**. Vivado HLS possède un grand nombre de directives servant à guider la synthèse du HDL. Ces directives peuvent être placées :

- Soit directement dans le code source, sous la forme de **#pragmas**.
- Soit dans un fichier de directives propre à chaque solution.

En général, les directives d'interface (comme ici) sont placées dans le code source, car il s'agit d'une contrainte de conception qui ne varie pas d'une solution à l'autre. En revanche, pour les directives de compilation (par exemple : déroulage de boucles, pipelining...), l'utilisation du fichier de directives est plus appropriée car elles sont propres à une solution.

Validation

Avant de lancer la synthèse, il est judicieux de vérifier à **chaque fois** que votre implémentation est correcte. Pour ce faire, lancez simplement le test fourni en cliquant sur le bouton :



(Ou, alternativement : **Project > Run C Simulation**).

Si tout se passe bien, vous verrez « **test PASSED** » s'afficher dans la console.

Synthèse et rapport de synthèse

Notre spécification logicielle étant écrite, nous pouvons maintenant lancer la synthèse du matériel. Pour cela, cliquez simplement sur la flèche dans la barre d'outils:



La sortie de l'outil apparaît dans la console. Les warnings et messages importants apparaissent en **bleu**, les erreurs éventuelles en **rouge**. En cas de succès, un rapport de synthèse est généré.

Le rapport de synthèse comporte 4 volets :

- **General Information** : Rappelle la date et l'heure de synthèse et le matériel ciblé.
- **Performance Estimates** :
 - **Timing (ns)** : Temps de propagation estimé du design. Le timing doit être inférieur à la période de l'horloge, en tenant compte de l'incertitude fournie (le timing précis ne peut être connu à cette étape du flot de conception).
 - **Latency (clock cycles)** : Nombre de cycles nécessaires pour un calcul. Outre la latence, l'intervalle (« **interval** ») est aussi fourni : il s'agit du nombre de cycles entre deux calculs. Le sous-volet « **Detail** » fournit des informations sur la latence de chaque boucle et leur nombre d'itérations (« **Trip count** »).
- **Utilization Estimates** : Détaille les ressources utilisées par l'implémentation (LUTs, BRAMs, ...) et leur répartition, ainsi que les pourcentages d'utilisation.

- **Interface** : Liste les ports d'E/S générés pour l'horloge, les arguments de la fonction et (éventuellement) la valeur de retour. Le tableau donne pour chaque port le protocole utilisé et la correspondance avec le code C/CPP.

Outre le rapport de synthèse, la perspective « **Analysis** », accessible en haut à droite de votre environnement, fournit des informations détaillées sur les ressources utilisées et l'ordonnancement généré. **Vous êtes vivement encouragés à explorer ses possibilités.**

Questions

1. Quels sont la latence et l'intervalle d'initiation du matériel générés ? Avec une fréquence de 100 MHz, la contrainte de performance globale est-elle respectée ?
2. Essayez de réduire la période dans les paramètres de solution (**Solution > Settings... > Synthesis**). Jusqu'où pouvez-vous aller ? Quelle est l'impact sur la latence ? Comment l'expliquez-vous ?

Exploitation de la séparabilité

Séparabilité

Le filtre gaussien que l'on souhaite implémenter est *séparable* :

$$K = \begin{pmatrix} 0.0608 & 0.1496 & 0.0608 \\ 0.1496 & 0.3680 & 0.1496 \\ 0.0608 & 0.1496 & 0.0608 \end{pmatrix} = \begin{pmatrix} 0.2466 \\ 0.6067 \\ 0.2466 \end{pmatrix} (0.2466 \quad 0.6067 \quad 0.2466).$$

En conséquence, la convolution par K peut être remplacée par :

- une convolution par un vecteur colonne 3×1 ,
- suivie d'une convolution par un vecteur ligne 1×3 .

Questions

1. Implémentez cette approche sous la forme de deux nids de boucle dans le fichier « **gaussian_sep.cpp** », en stockant le résultat de la première convolution dans un tableau intermédiaire de la taille de l'image.
2. Lancez la synthèse (pensez à rétablir la période à 10ns dans les paramètres de la solution) et examinez le rapport. Le design obtenu est-il implantable sur la carte ? Pourquoi ? Comment l'expliquez-vous (rapports !), et quel lien faites-vous entre le code écrit et le matériel généré ?

Fusion de boucles

Dans l'approche précédente, l'utilisation des BRAMs est mauvaise. Les passes horizontale et verticale peuvent être effectuées conjointement en fusionnant les boucles, de sorte à minimiser l'empreinte mémoire.

Questions

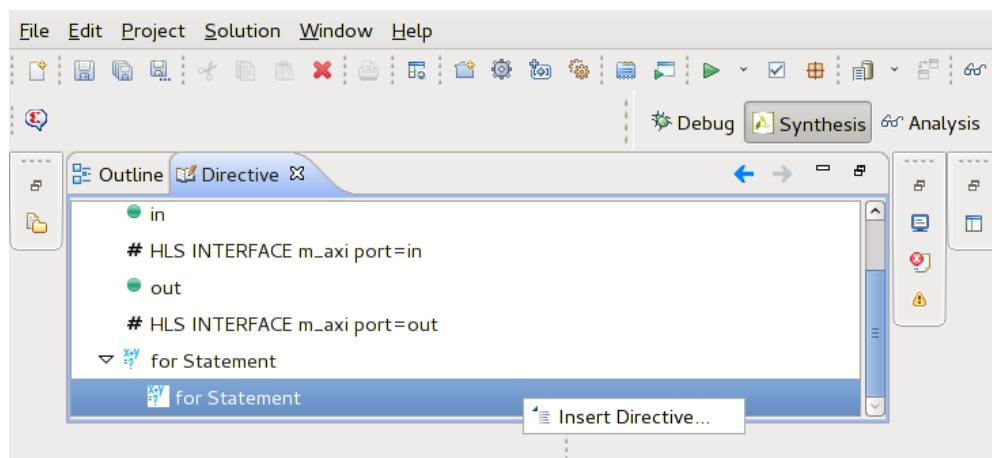
1. Procédez à la fusion des deux boucles en conservant pour l'instant le tableau intermédiaire (« **gaussian_loopFusion.cpp** »). L'empreinte mémoire a-t-elle changé ? Qu'en est-il de la latence ?
2. Reprenez votre implémentation en réduisant cette fois-ci la taille du tableau à une ligne (« **gaussian_1line.cpp** »). Votre design est-il maintenant implantable sur la carte ?
3. Procédez à une scalarisation avec vieillissement afin de conserver uniquement les résultats nécessaires lors du traitement d'une ligne (« **gaussian_scalarized.cpp** »). Combien de BRAMs utilisez-vous ? Que pouvez-vous en conclure ?
4. La contrainte de performances est-elle enfin atteinte ?

Pipeline de boucles

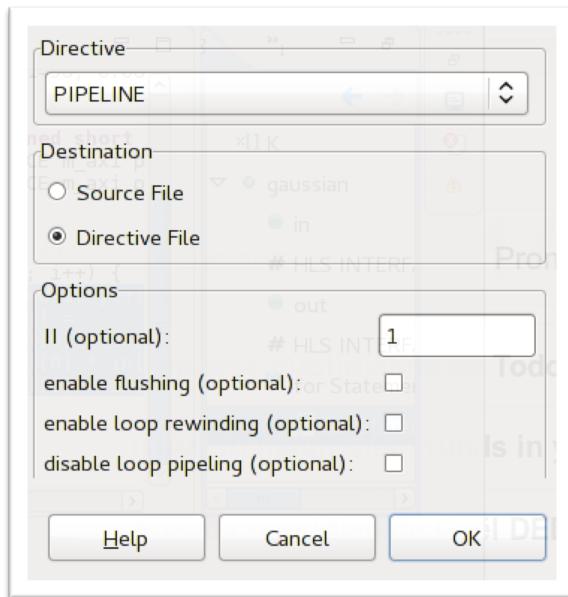
Afin de diminuer la latence globale, on peut *pipeliner* la boucle interne.

Questions

1. Dans l'onglet « **Directive** » (à côté de « **Outline** », à la droite de l'écran), faites un clic droit sur l'entrée correspondant à la boucle la plus interne, et sélectionnez « **Insert Directive...** » dans le menu contextuel.



2. Dans le dialogue, sélectionnez la directive PIPELINE et entrez un *Initiation Interval* (II) de 1.



3. Lancez la synthèse, et lisez les messages affichés dans la console. Le résultat est-il conforme à vos directives ? Comment l'expliquez-vous ? Vous pouvez vous aider de la perspective d'analyse (bouton « **Analysis** » en haut à droite).

Minimisation des accès mémoire

L'approche précédente n'est pas optimale en termes d'accès mémoire. En effet, chaque pixel de l'image d'entrée est lu plusieurs fois (combien ?).

Questions

1. Modifier votre implémentation afin de ne lire chaque donnée qu'une seule fois (« **gaussian_caching.cpp** »). On pourra utiliser un tableau de taille adéquate, soigneusement géré.
2. Synthétisez et analysez votre design. Quelle est l'influence sur le pipeline, et pourquoi ? La contrainte de performances semble-t-elle désormais respectée ? A quoi attribuez-vous ce paradoxe ?

Note : Pour copier des blocs de données il existe la fonction ***memcpy(...)*** dont le prototype est le suivant :

```
void *memcpy(void *dest, const void *src, size_t n);
```

Partitionnement

Le design précédent souffre d'une faiblesse : le « cache » utilisé est constitué d'une seule mémoire, ce qui limite le nombre d'accès simultanés.

Questions

1. Analysez l'ordonnancement de l'implémentation précédente et identifiez les zones où les accès mémoire restent problématiques.
2. Dans le volet « **Directive** », effectuez un clic droit sur la variable « **cache** » puis choisissez « **Insert directive...** ». Sélectionnez la directive « **ARRAY_PARTITION** », et entrez les paramètres suivants :
 - **Variable** : cache
 - **Type** : <vide>
 - **Factor** : <vide>
 - **Dimension** : 1
3. Relancez la synthèse. Que constatez-vous ?
4. En analysant soigneusement les rapports de synthèse, l'ordonnancement et les ressources utilisées, expliquez l'effet de la directive « **ARRAY_PARTITION** ».
5. Quel est la fréquence maximale atteignable sans risque de violation de timing ? Avez-vous enfin atteint votre but ?

Mise en œuvre en virgule fixe

Bien que précise et versatile, l'arithmétique flottante est gourmande en ressources (surface, consommation) et sa latence est importante. Dans les systèmes embarqués, on lui préfère souvent une mise en œuvre en **virgule fixe**, où le nombre de chiffres avant et après la virgule est prédéterminé.

Vivado HLS propose une bibliothèque (« **ap_fixed** »), basée sur les templates C++, permettant d'effectuer des calculs virgule fixe en précision arbitraire. Les valeurs virgule fixe sont déclarées comme suit :

```
ap_[u]fixed<W,I> valeur;
```

Où :

- Les types **ap_fixed** (resp. **ap_ufixed**) sont signés (resp. non signés).
- **W** représente le nombre de bits total (sans le bit de signe, le cas échéant).
- **I** représente le nombre de bits de la **partie entière**.
- Le nombre de bits de la partie fractionnaire est donc implicite et vaut **W-I**.

D'autres paramètres sont disponibles, comme le mode de quantification (arrondi ou troncature) où le comportement en cas d'overflow (saturation ou *wrap-around*) – nous n'aurons pas à les utiliser.

Questions

1. Transformez l'algorithme en virgule fixe (« **gaussian_fixedPoint.cpp** »). Tâchez de minimiser la taille de votre design en vérifiant, au moyen du test fourni, que votre implémentation respecte les contraintes de précision. Qu'en est-il des performances ? Jusqu'où pouvez-vous aller en modifiant la fréquence ?

Déroulage partiel

Pour réduire encore la latence, il est possible de dérouler partiellement la boucle interne.

Questions

1. En procédant comme pour le partitionnement de tableau, ajoutez une directive de déroulage partiel (directive « **UNROLL** », en spécifiant un facteur de déroulage). Quel est l'effet sur la latence ? Quel semble être le facteur optimal ?
2. Si la fréquence est fixée à 200 MHz et la contrainte de performances relevée à 60 trames/s, votre design satisfait-il à ces nouvelles exigences ?