# Boîte à outils pour la conception d'accélérateurs de traitement d'image

Stéphane Mancini

21 décembre 2018

## Table des matières

**1 Méthodologie** **2**

**2 Vue générale** **2**
2.1 Conception d'accélérateur de traitement d'image . . . . . 7
2.1.1 Référence algorithmique . . . . . 7
2.1.2 Implémentation virgule fixe et pré-HLS . . . . . 7
2.1.3 Code pour la HLS . . . . . 7
2.1.4 Emulation FPGA . . . . . 8
2.1.5 Validation sur FPGA et caméra . . . . . 8

**3 Images** **8**
3.1 Codage des images . . . . . 8
3.2 Stockage des images . . . . . 9
3.3 Format des images . . . . . 9
3.4 Accès aux images pour la validation HLS . . . . . 10
3.4.1 En C . . . . . 10
3.4.2 En Python . . . . . 11
3.5 Accès aux images dans le FPGA . . . . . 11
3.5.1 En SRAM du FGPA . . . . . 11
3.5.2 Génération d'un fichier coe . . . . . 12
3.5.3 Logiciel 'bare-metal' et images . . . . . 12
3.5.4 En logiciel sous Linux . . . . . 13
3.5.5 Accès aux images en DDR-SDRAM . . . . . 13

**4 Gestion de la virgule fixe** **14**
4.1 Gestion des images . . . . . 15

**5 Paramétrage des IP** **15**
5.1 Motivation . . . . . 15
5.2 Paramétrage de la virgule fixe . . . . . 16
5.2.1 Fonctions basiques . . . . . 16
5.2.2 Arithmétique . . . . . 17

**6 Capture vidéo sur Zybo** **18**

**7 Plateforme Zybo Caméra & Processing** **20**
7.1 Projet Vivado . . . . . 20
7.2 CatapultC . . . . . 20

**A Vivado** **22**

**B CatapultC** **22**

**C Zybo** **22**
C.1 Port USB/Série . . . . . 22

**D Génération de RAM** **22**

# 1 Méthodologie

# 2 Vue générale

La méthode de conception d'accélérateurs est une sous-partie de la méthode de conception de systèmes logiciels et matériels. On fera l'hypothèse que le partionnement logiciel/matériel a déjà été fait et on ne s'attachera qu'à la conception d'un accélérateur donné. Les étapes de conception décrites dans ce document sont des frontières 'théoriques', et, dans la réalité, on peut être amené à réaliser plusieurs étapes simultanément. L'intérêt de ce découpage est surtout de permettre une mise en perspective générale du projet et de se situer dans un flot de développement complexe.

Chaque étape est une spécialité en soi et l'optimisation globale d'un projet est assez difficile. Comme il est assez improbable de trouver la "meilleure" solution du premier coût (si tant est qu'il en exite une), dans un premier temps, l'objectif est de réaliser toutes les étapes le plus rapidement possible puis d'analyser les résultats pour ensuite optimiser le système. Les grandes étapes de la conception de l'accélérateur sont les suivantes

— **R**éférence algorithmique, figure 1
La référence algorithmique provient souvent d'une implémentation des équations mathématiques. De façon à s'affranchir de détails d'implantation, elle est le plus souvent en langage de 'haut niveau', comme matlab ou Python. Des langages commes C ou C++ sont possibles mais pas toujours souhaitables.

— **I**mplémentation virgule fixe, figure 2
L'artimétique virgule fixe est mise en place. Chaque valeur est représentée soit par

un nombre en virgule fixe soit par un entier. Dans un premier temps, un réglage grossier des précisions et dynamiques est suffisant pour valider l'implémentation.

— **I**mplémentation pour la HLS, figure 3

  En plus de la virgule fixe, les constructions algorithmiques tiennent compte des contraintes de la HLS :
  — Boucles bornées à bornes statiques
  — Allocation mémoire statique, c'est à dire d'allocation mémoire dynamique (création d'objet, allocation mémoire explicite). Ceci nécessite une identification de toutes les mémoire et le calcul de leur taille, qui sera basé sur le pire cas.
  — Détermination de toutes les constantes qui seraient des paramètres du code haut niveau (taille des tableaux, etc ...). Ces constantes peuvent être placées dans des fichiers de configuration sous forme de macros.
  — Interactions avec l'environnement
    — Accès aux mémoires et contraintes associées (simple ou double ports)
    — Passage de paramètres par variables ou mémoires
    — Synchronisation avec des flux de données
— **H**LS et réglage de l'architecture

  — Premiers tests de HLS
  — Analyse de performance et comparaison avec les performances prévues
  — Réglage des principaux paramètres architecturaux pour régler le compromis performance/surface : type de mémoire de chaque tableau interne/externe (SRAM simple ou double port, DFF), gestion des boucles (déroulement, pipeline), virgule fixe
— **V**érification fonctionnelle post-HLS, figure 4

  Le banc de test permet de vérifier que la HLS produit une architecture qui préserve la fonctionnalité.
— Synthèse logique
  Transformation du résultat de la HLS (RTL) en netlist
— **V**érification post synthèse logique, figures 6 5 7

  Théoriquement sous forme de simulation mais il est de temps en temps impossible de simuler le système et, dans ce cas, une émulation FPGA fait office de validation.
  Sur FPGA, l'accélérateur est connecté à son environnement (RAM, registres, FIFO) de façon à fournir des valeurs et stocker des résultats. Pour comparaison, il est également possible d'utiliser des mémoires de résultats produits aux étapes précédentes de façon à vérifier in-situ les résultats produits par l'accélérateur.

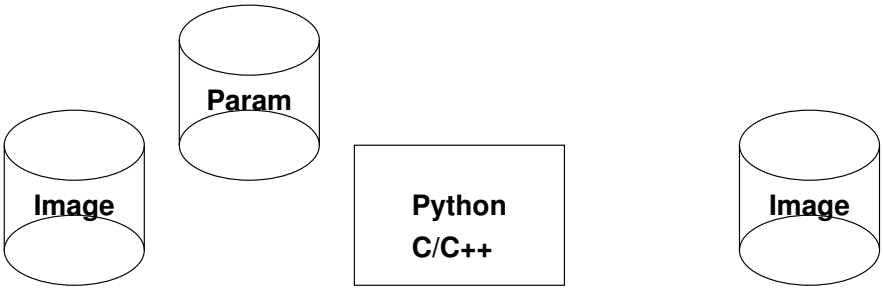


FIGURE 1 – Etape 1 : référence algorithmique

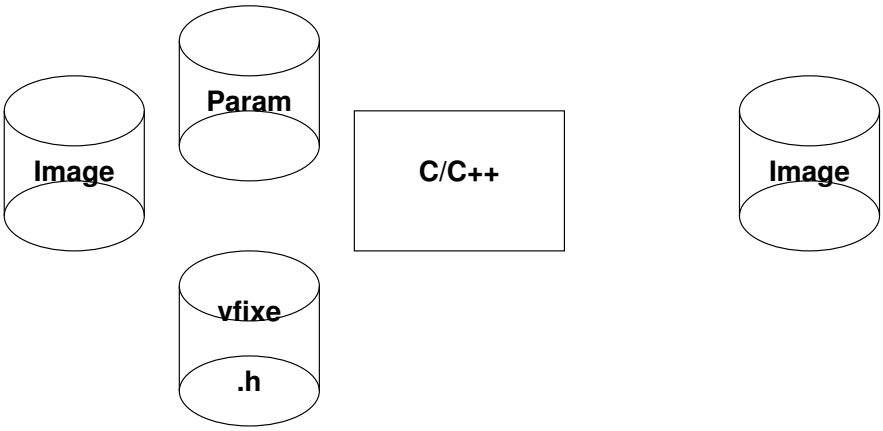


FIGURE 2 – Etape 2 : algorithme virgule fixe

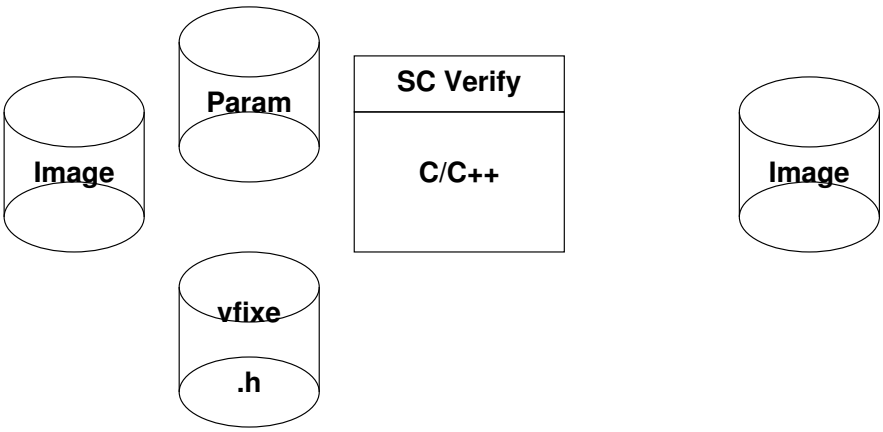


FIGURE 3 – Etape 3 : virgule fixe *pré*-HLS dans l'environnement SC-Verify

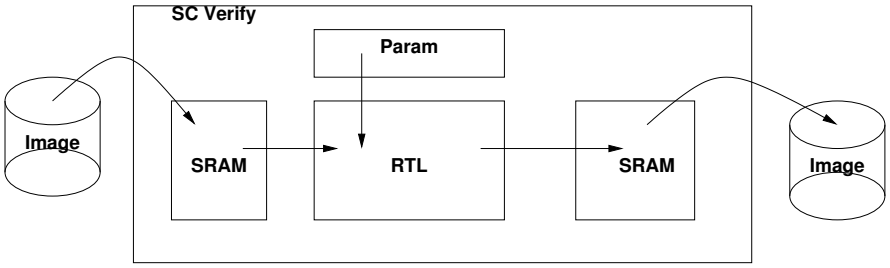


FIGURE 4 – Etape 4 : vérification *post*-HLS dans l'environnement SC-Verify

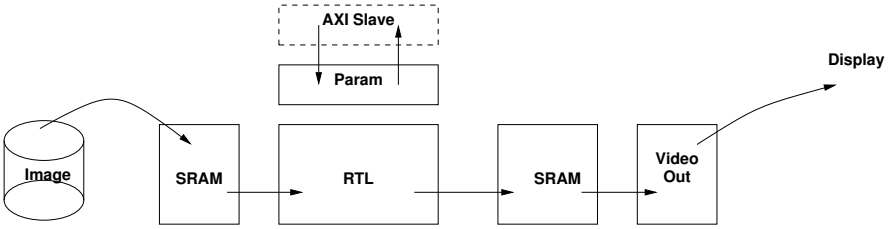


FIGURE 5 – Etape 5 : vérification par émulation ; Les données sont statiques (SRAM initialisée), les paramètres peuvent être statiques ou proviennent d'un SW par le bus AXI, les résultats sont affichés sur écran.

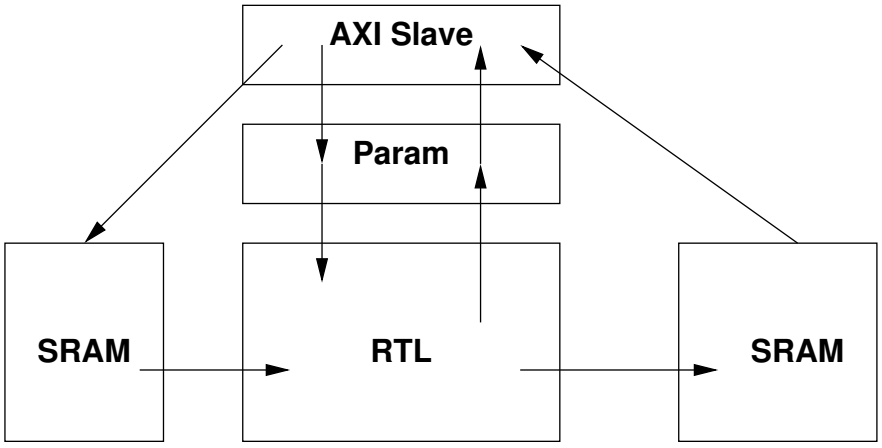


FIGURE 6 – Etape 6 : vérification par émulation dans un système HW/SW ; Les paramètres et les données viennent d'un logiciel (bare-metal par exemple), les résultats sont lus depuis le SW pour vérification.

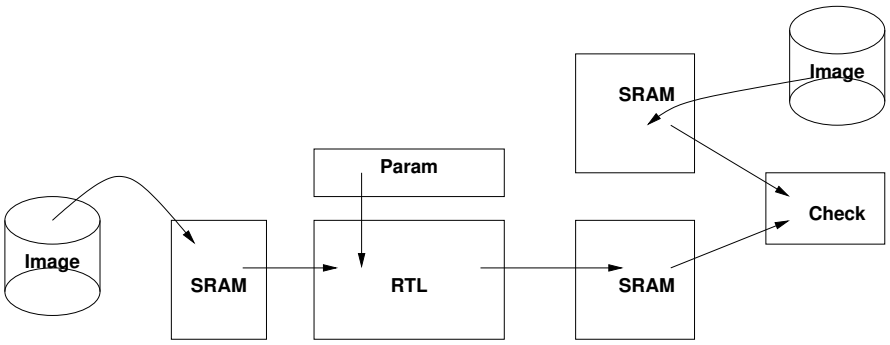


FIGURE 7 – Etape 6 bis : vérification par émulation dans un système HW/SW ; Les paramètres et les données sont statiques, les résultats sont comparés à des résultats attendus.

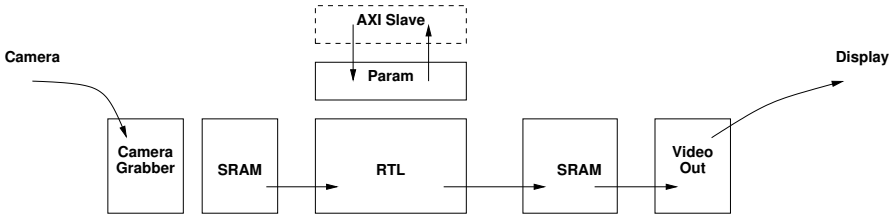


FIGURE 8 – Etape 8 : vérification par émulation ; Les données proviennent d'une caméra, les paramètres peuvent être statiques ou proviennent d'un SW par le bus AXI, les résultats sont affichés sur écran.

## 2.1 Conception d'accélérateur de traitement d'image

Concernant les accélérateurs de traitement d'image, la méthode mise en oeuvre est la suivante :

### 2.1.1 Référence algorithmique

Codée en Python. Les images d'entrée et sorties sont dans des fichiers, au format PGM. Les paramètres sont dans une donnée de type dictionnaire, facile à stocker dans un fichier

### 2.1.2 Implémentation virgule fixe et pré-HLS

Le code pour la HLS utilise la virgule fixe avec les type "AC types" de Mentor graphics. L'algorithme est vérifié par l'exécution du code compilé, sur un PC, sans HLS. Les paramètres de virgule fixe sont sous forme de macros, regroupés dans un fichier spécifique.

Une image est un tableau dont tous les éléments ont le même format de virgule fixe.

Les paramètres et précisions sont soit sous forme de macro lorsqu'ils sont statiques, soit en variable globale.

Les images à traiter sont dans des fichiers luts et écrits par l'intermédiaire de tableaux.

L'utilisation de la virgule fixe produit un 'bruit' de calcul et il est possible de mesurer l'écart à la référence algorithmique. Dans un premier temps, de façon à se focaliser sur la fonction, les paramètres de la virgule fixe sont assez large et permettent un fonctionnement de l'algorithme.

**R**emarque : Il est tentant d'avoir le même code pour la virgule flottante et la virgule fixe et de changer le type de données à l'aide de macros. A court terme cela est séduisant mais se révèle rapidement gênant car les codes sont très différents et les mélanger devient totalement contre-productif.

### 2.1.3 Code pour la HLS

Semblable au précédent, les mémoires étant identifiées et le maximum de paramètres sont statiques. Les interfaces au système sont identifiées :

— Registres de configuration
— Mémoire de données lues & écrites
— Mémoires pour les données intermédiaires

Pour le banc de test, les images d'entrée/sortie sont dans des fichiers luts/écrits depuis des tableaux.

La vérification post-HLS est incorporée à la HLS et l'outil compare automatiquement les résultats produits par le code RTL généré et les résultats de l'exécution du code C/C++ pour la HLS.

Après HLS, la synthèse logique RTL permet de vérifier les performances (timing) ainsi que le coût matériel.

### 2.1.4 Emulation FPGA

Les paramètres sont placés soit dans des registres ou bien des signaux constants. Dans le premier cas, un logiciel viendra régler les valeurs des registres.

Toutes les étapes avant l'émulation peuvent être réalisées à partir d'images dites 'de référence', qui serviront à faire tous les tests. Ainsi, il est possible de travailler sans caméra le plus longtemps possible.

Les images sont placées dans des SRAM, soit par initialisation de la SRAM, soit à l'aide d'un logiciel qui vient placer les données en SRAM.

**R**emarque : L'émulation apporte peu d'informations supplémentaires par rapport à la validation post HLS, car le placement/routage n'introduit (quasiment) pas d'erreurs fonctionnelles. L'intérêt peut être de vérifier plus rapidement des situations qui sont longues à reproduire par simulation. Inversement, la détection d'un bug nécessite de reproduire la situation en simulation, ce qui peut s'avérer difficile vu le peu d'observabilité sur FPGA.

### 2.1.5 Validation sur FPGA et caméra

Les images proviennent d'une caméra.

— Les images d'entrée sont placées en SRAM directement par l'entrée caméra. La sortie est une SRAM dont le contenu est affiché. Pour différentes raisons, il est plus simple d'utiliser des SRAM double-port. Un port est pour l'entrée (ou la sortie), l'autre pour l'accélérateur. Du point de vue de l'accélérateur, la mémoire est simple port.

— Les images sont placées en DDR-SDRAM, par exemple à l'aide de DMA. Cette solution est la plus réaliste mais est difficile à mettre en oeuvre.

# 3 Images

## 3.1 Codage des images

Une image est une matrice de pixels, chaque pixel étant soit codé sur plusieurs composantes couleur (RGB), soit simplement par une luminance. Typiquement les composantes sont des entiers sur 8 bits, 16 bits, ou autre. En général les componsantes ne sont pas de valeurs signées.

Il existe des codages plus complexes (par exemple le codage 4.2.2) et les composantes peuvent être autre chose que RGB (YUV, etc...) mais ce n'est pas le propos de cette boîte à outils.

En général les images sont représentées en mémoire de deux façons :

— Les composantes sont regoupées par pixels, dans l'ordre (R0, G0, B0, R1, G1, B1, etc...)

— Les composantes sont séparées (tous les Rn, puis tous les Gn, les Bn, etc ...)

Réaliser un traitement d'image consiste généralement à calculer chacun des pixels d'une image de sortie en fonction des pixels d'une image d'entrée. De façon à simplifier la

conception, nous ne traiterons que des images en luminance, sur une seule composante par pixel.

### 3.2 Stockage des images

En mémoire du calculateur, la matrice est stockée 'linéairement', c'est à dire que le pixel $(i, j)$ se trouve à l'adresse relative

$$@(i, j) = i + j * t_x$$

Avec $(t_x, t_y)$ les tailles horizontale et verticale de l'image.

Il existe d'autre schémas d'adressage que nous verrons au cours du projet.

Une image est stockées soit en SRAM soit en DDR-SDRAM. Les différences sont les suivantes :

- En DDR-SDRAM ; Pour accéder à un pixel il est nécessaire de passer par un bus système et par le contrôleur mémoire. Les DDR-SDRAM ont des débits élevés mais des latences élevées et ne sont efficaces que si l'on accède à plusieurs pixels dans des bursts. Il est nécessaire de mettre en place une mémoire de travail proche de l'accélérateur (mémoire cache, SPRAM, SRAM, etc. . . ), dans laquelle on recopiera des zones de l'image.
- En SRAM ; Dans ce cas, la SRAM fourni un mot mémoire par cycle d'horloge, le cycle suivant la présentation de l'adresse. Les SRAM sont très rapides mais de quantité réduite.

Pour les projets, pour des raisons de simplicité, nous privilégierons les SRAM en sachant qu'une implémentation réaliste serait plutôt un stockage en DDR-SDRAM avec un mécanisme de cache plus ou moins spécialisé.

### 3.3 Format des images

Pour la conception d'accélérateurs par HLS, il est donc possible de considérer une image comme un tableau de taille $t_x * t_y$ et d'y accéder en calculant l'adresse de façon appropriée.

Pour la validation du code C/C++, l'image est chargée en mémoire avant le lancement de l'accélérateur. A cette fin, l'image pourra être lue depuis un fichier.

Les formats de fichier les plus simples sont :

- `RAW` ; Le fichier n'a pas d'en-tête, les valeurs binaires des pixels sont les unes après les autres et les composantes entrelacées.
  L'entrelacement peut être :
  - Par pixel ; la séquence RGB répétée pour chaque pixel
    R0 G0 B0 R1 G1 B1
  - Par plan ; on trouve les composantes les unes après les autres
    R0 R1 B0 B1 G0 G1
- `PPM` ; Format simple à en-tête (voir `man ppm`), avec les données soit en texte ASCII soit en binaire, entrelacées par pixel. Les formats sont soit `PGM` (voir `man pgm`) pour les images en luminance (niveau de gris), soit `PPM` pour les images couleur.

Exemple d'image PGM de taille $(2, 2)$ :

```
P2
# Un commentaire
2 2
255
0 255 255 0
```

Ces types de fichier peuvent être produit de plusieurs façons. Par exemple :

- Logiciel `GIMP`, menu fichier->exporter->format `data`, `pgm` ou `ppm`
- Utilitaire `ImageMagick`, commande `convert`

  - `convert -monochrome -compress none image_entree.jpg image_sortie_ASCII.pgm`
  - `convert -monochrome image_entree.jpg image_sortie_BINAIRE.pgm`
  - `convert -monochrome image_entree.jpg GRAY:image_RAW_BINAIRE.raw`
  - `convert  image_entree.jpg RGB:image_RAW_BINAIRE_Couleur.raw`

Les utilitaires `ImageMagick` sont très efficaces et il est préférable de consulter leur documentation.

## 3.4 Accès aux images pour la validation HLS

Une fois l'image PGM ASCII formée, il est possible de la lire très simplement depuis n'importe quel langage, C ou Python. A l'inverse, l'écrire est aussi très facile. Les exemples suivant sont pour le format ASCII et peuvent être facilement transposés pour le format binaire.

### 3.4.1 En C

Exemple de code de lecture d'une image PGM ASCII (format `P2`) :

```
f=fopen("Mon_fichier.pgm", "r");
fscanf(f,"%s", format);
/* Lit la ligne qui contient la taille */
fgets(taille, 256, f);
/* mais passe les commentaires */
while (taille[0]=='#') fgets(taille, 256, f);
// lit la taille dans la chaine
sscanf(taille,"%d %d", &tx, &ty);
fscanf(f,"%s", temp);
for(int i=0;i<tx*ty*;i++)
  fscanf(f,"%d", &image[i]);
```

Dans cet exemple, on suppose que `image` est un pointeur sur un tableau d'entier préalablement alloué. Pour faire plus générique, on peut allouer l'image dynamiquement après avoir lut la taille. Il est possible de faire un peu plus élégant en C++.

### 3.4.2 En Python

Encore plus facile car il suffit de lire les lignes...

```
img=open('Mon_fichier.pgm');
format=img.readline()
line=img.readline()
while line[0] == '#' : line=img.readline()
(width,height) = [int(i) for i in img.readline().split()]
valmax=img.readline()
raster = []
for i in range(width*height):
    raster.append(int(img.read()))
img.close()
```

Il est possible de lire un fichier PGM binaire et le placer dans une variable de la façon suivante :

```
img=open('Mon_fichier.pgm');
format=img.readline()
line=img.readline()
while line[0] == '#' : line=img.readline()
(width,height) = [int(i) for i in img.readline().split()]
valmax=img.readline()
image_data=img.read(width*height)
img.close()
```

Pour lire un fichier RAW binaire :

```
img=open('Mon_fichier.pgm', 'b');
image_data_raw=img.read()
img.close()
```

L'écriture d'un fichier est le symétrique (voir documentation Python)

## 3.5 Accès aux images dans le FPGA

### 3.5.1 En SRAM du FGPA

Bien entendu, pour accéder aux pixels de l'image en SRAM il faut que la SRAM contienne les pixels. L'objectif est de valider le traitement sur la même image que celle utilisée pour la simulation, avant d'utiliser des images en provenance de la caméra. Il existe plusieurs façons d'émuler une image qui proviendrait d'une caméra :

- La SRAM est initialisée par les valeurs des pixels (raw)
  - La SRAM est initialisée à la création de l'unité SRAM. Il faut créer un fichier au format `coe`, qui est la liste des valeurs séparées par une virgule. Le fichier `coe` pourra être produit par un script Python ou tout simplement à partir

d'un fichier PGM. Le fichier au format `coe` sera utilisé à la création de la SRAM dans Vivado.

- Une entité VHDL de la SRAM est générée à partir de l'image, avec les valeurs des pixels. Ici, la 'SRAM' est un signal dont la valeur initiale est la liste des pixels. Un tel fichier pourra être généré par un script Python.
  Voir section D

— La SRAM est initialisée par le logiciel du processeur
Si l'on utilise une SRAM double port, l'un des ports peut être accédé par le processeur s'il est interfacé par une interface esclave sur le bus AXI. Le logiciel écrit la SRAM avec le contenu de l'image, ou bien la lit.

**A**ttention : Dans CatapultC, les paramètres de la SRAM incluent la polarité des signaux de contrôle : les 'read enable' et 'write enable' peuvent être actif sur niveau bas ou haut. Par défaut ils sont sur actifs sur niveau bas et vérifier si les IPs Xilinx/Altera sont compatibles.

### 3.5.2 Génération d'un fichier coe

En python, générer un fichier au format **c**oe est trivial.

```
print('memory_initialization_radix=10;')
print('memory_initialization_vector=')
print(',\n'.join(str(val) for val in image_data) + ';')
```

Vous pouvez aussi utiliser des base 2 ou 16, selon les besoins.

### 3.5.3 Logiciel 'bare-metal' et images

L'image peut être accédée depuis le logiciel (par exemple pour être copiée dans la SRAM de l'accélérateur) de plusieurs façon :

— Tableau initialisé
Un tableau est initialisé par les valeurs des pixels de l'image à l'aide d'une variable globale.
Le tableau déclaré est initialisé par les pixels. Un tel code peut être généré en Python à partir de l'image pgm/ppm.

— Objet initialisé
De façon à soulager le compilateur, un fichier objet qui contient l'image au format RAW est incorporé à l'édition de lien en ajoutant une section (`.input_data` par ex.). Ensuite, l'image est accédée par une variable qui pointe vers le symbole associé.

- Conversion du fichier image

```
$CROSS-objcopy -I binary -O elf32-littlearm -B arm \
               --rename-section .data=.input_data,alloc,load,readonly,data,contents  \
               image_RAW_BINARY.data image.o
```

en ayant affecté `$CROSS` par le préfixe du cross-compilateur

— Incorporation par linkerscript
Dans le linkerscript, ajouter une section avant la fin :

```
.input_data . : {
   . = ALIGN(64);
   _data_image_start = . ;
    image.o(.input_data)
   _data_image_end = . ;
} > memory_region
```

En prenant soin de remplacer les noms et de modifier `memory_region` selon l'entête du linker script (par exemple `ps7_ddr_0_S_AXI_BASEADDR` ou `ps7_ddr_0`)

— Accès depuis le C/C++
Déclarer une variable de même nom que le symbole dans la section ajoutée, puis utiliser un pointeur sur l'adresse de ce symbole.

```
extern unsigned int _data_image_start;
...
unsigned char *img=&_data_image_start;
...
p=img[x+tx*y]; // accès au pixel x,y dans l'image, tx doit prendre la taille hor
```

— Fichier
L'image est lue dans un fichier accessible (par exemple sur carte SD). Il sera possible d'utiliser un code similaire à la lecture d'une image pgm, mais en tenant compte du fait qu'il n'y a pas d'allocation mémoire dynamique en bare-metal.

En général lorsque l'image est incorporée dans le logiciel elle sera automatiquement placée en DDR-SDRAM. Bien que cela soit rarement nécessaire, il est possible de forcer son placement dans une zone particulière à l'aide du *linker script*.

### 3.5.4 En logiciel sous Linux

Les techniques précédentes sont utilisées mais la lecture de fichiers est encore plus simple. Le code utilisé pour la simulation peut être réutilisé.

### 3.5.5 Accès aux images en DDR-SDRAM

Il est possible de placer les images en mémoire DDR-SDRAM externe au FPGA. Dans ce cas, il faut mettre en place les mécanismes d'accès aux données depuis l'accélérateur. L'accélérateur charge ses données depuis la DDR-SDRAM de deux façons :

— Il dispose de son propre DMA
Dans cette situation, l'accélérateur se charge de demander les zones de données au DMA.

— Le DMA est externe à l'accélérateur
Un mécanisme de synchronisation HW/SW permet à l'accélérateur de 'réclamer'

au logiciel les données à charger. Le logiciel organise le transfert des données. Cette méthode est relativement souple mais peu efficace.

Dans les deux cas, il faudra veiller à l'alignement des données en mémoire. Par exemple, le premier pixel d'une image (ou d'une ligne) doit être au début d'un mot mémoire car le DMA ne peut pas commencer par une donnée qui n'est pas alignée. Pour aligner les données en mémoire il faudra forcer les adresses des données sur des multiples de la taille du bus de donnée.

Il est possible de placer des données en DDR-SDRAM à l'initialisation du SW, à l'aide du linker-script, et de les aligner. Voir la section précédente sur le logiciel en bare-metal.

L'accès aux données en DDR-SDRAM en présence de Linux pose de sérieux problèmes car le DMA fonctionne en adresses physiques alors que les applications utilisent des adresses virtuelles et les données des tableaux peuvent ne pas avoir des adresses physiques continues. Il faut réussir à gérer des données de plages mémoire physique contigües et cela nécessite l'utilisation de fonctions noyaux spécifiques, utilisable seulement par des *module* ou *driver*. Vu la durée des projets, cette technique est exclue.

# 4 Gestion de la virgule fixe

Pour résumer le document "Algorithmic C Datatypes" de MentorGraphics, le format des nombres en virgule fixe est

```
ac_fixed<W,I,sign>
```

avec `W` la taille totale et `I` le nombre de bits de la partie entière. `sign` indique si le nombre est signé `true` ou non-signé `false`.

Cette bibliothèque gère automatiquement les changements de type et conversion depuis/vers les nombres en virgule flottante. Les opérateurs arithmétiques sont 'surchargés' et gèrent automatiquement les décalages nécessaires pour réaliser des additions ou multiplication sur des nombres de formats différents. Il est également possible de réaliser des opérations de troncature/arrondi automatiquement, par déclaration des bons formats et affectations entre variables de formats différents.

Par exemple pour réaliser une troncature de `a` vers `b` :

```
ac_fixed<a_W, a_I, a_sign> a;
ac_fixed<b_W, b_I, b_sign> b;
b=a;
```

Par défaut, `b` est affecté par troncature (ou extension de zéro si `a` a moins de bits à droite de la virgule). Il est possible de spécifier le mode d'affectation avec deux paramètres supplémentaires du template (voir documentation).

De façon à gérer simplement la virgule fixe, l'expérience montre qu'il est préférable de gérer le paramétrage dans un fichier de configuration :

```
/* Fichier  imgproc_vfix_config.h /*
#define IMGPROC_A_PS 20, 10, true, AC_RND
```

```
/* Fichier  imgproc.c */
ac_fixed<IMGPROC_A_PS> a;
```

### 4.1 Gestion des images

Une image est un tableau et le format de la virgule fixe s'applique à tous les éléments du tableau :

```
/* Fichier  imgproc_vfix_config.h /*
#define IMGPROC_IMAGE_IN_SIZE_0 640
#define IMGPROC_IMAGE_IN_SIZE_1 480
#define IMGPROC_IMAGE_IN_P 20, 10, true, AC_RND

#define IMGPROC_IMAGE_IN_SIZE IMGPROC_IMAGE_IN_SIZE_0*IMGPROC_IMAGE_IN_SIZE_1


/* Fichier  imgproc.c /*
ac_fixed<IMGPROC_IMAGE_IN_P> image_in[IMAGE_IN_SIZE];
```

La HLS produira automatiquement une interface à une mémoire de taille `IMGPROC_IMAGE_IN_SIZE`, avec des mots mémoire de taile `IMGPROC_IMAGE_IN_P(W)`. Selon les besoins, cette mémoire pourra être simple ou double port.

**R**emarque : L'utilisation de préfixe dans les paramètres permet d'éviter les conflits entre IP. **R**emarque : L'utilisation du numéro d'axe (0, 1, etc ...) au lieu de leurs noms usuels (X, Y, etc ...), permet d'automatiser le nommage et de scripter la génération de paramètres

## 5 Paramétrage des IP

### 5.1 Motivation

Il est assez commun que le format d'une variable intermédiaire dépende de celui d'une autre variable. Par exemple, si on calcule $c = a * b + e$ et que l'on souhaite garder toute la précision de $a * b$ avant de faire la troncature pour produire $c$, on peut avoir :

```
/* Fichier  imgproc_vfix_config.h /*
#define IMGPROC_A_PS 20, 10, true
#define IMGPROC_B_PS 20, 10, true
#define IMGPROC_AB_PS 40, 10, true
#define IMGPROC_E_PS 10, 5, true

#define IMGPROC_C_PS 20, 10, true, AC_RND
...
/* Fichier  imgproc.c /*
ac_fixed<IMGPROC_A_PS> a;
```

```
ac_fixed<IMGPROC_B_PS> b;
ac_fixed<IMGPROC_E_PS> e;
...
ac_fixed<IMGPROC_AB_PS> ab;
ab=a*b;
c=ab+e;
```

Dans ce cas, la précision de `ab` dépend de celle de `a` et de `b`.

Il peut être utile d'écrire un script Python qui génère automatiquement la configuration de la virgule fixe en calculant automatiquement tout ce qu'il est possible de calculer, puis génèrer un fichier de configuration de la virgule fixe.

## 5.2 Paramétrage de la virgule fixe

### 5.2.1 Fonctions basiques

Les scripts Python `tpu_lib_param.py` et `tpu_lib_fixpoint.py` fournissent quelques fonctions pour faciliter la génération de paramètres des IP et automatiser le processus vu plus haut. Ce script permet de lire des fichiers de paramètres, de générer de nouveau paramètres par calcul, puis de générer un fichier de paramètres et les macros H associées. Par exemple, imaginons que l'on veuille générer une table de valeurs pour réaliser un calcul de cosinus. Le premier exemple calcule les valeurs de la table et règle la virgule fixe en fonction de l'amplitude des valeurs et de la précision attendue.

```
/* Calcul de la precision d'une table de cosinus */
import sys

from tpu_lib_param import *
from tpu_lib_fixpoint import *

# Calcul direct des parametres
n= 128                          # Nombre de valeurs en abscisse de la table
x= np.arange(0, np.pi, np.pi/n)  # Echantillonnage régulier par pas de pi/n
y= 3.3*np.cos(x)                # Les valeurs en ordonnées
# Déclare le format virgule fixe
p_cos= FixedPoint(sign='S')
# Calcule automatiquement le nombre de bits à droite et gauche de la virgule
# a partir des valeurs précédentes, pour une erreur de 1e-2
p_cos.range(y, 1e-2)


/* Generation des parametres */
toto= {}
toto['COS.N']= n
toto['COS.P']= p_cos
```

```
toto['COS.VAL']= y
toto['COS.VAL.INT']= p_cos.to_int(y)

param_to_H(toto, 'tab_cos_fixed.h')
```

Ceci génère une liste de macros dans le fichier `tab_cos_fixed.h`. Ces macros pourront être utilisées dans le code C de la façon suivante :

```
ac_fixed<COS_P_PS> tab_cos[COS_N] = {COS_VAL};
```

Une autre façon est de rendre ce code plus générique en indiquant les paramètres dans un fichier de configuration :

```
# Fichier cos.par
COS.N  : 128
COS.A  : 2.5
```

Ce fichier de configuration est lu puis sert à générer les valeurs

```
# Calcul  des parametres à partir d'un fichier de configuration
tutu={}
param_read(tutu, 'cos.par')
n= tutu['COS.N']
a= tutu['COS.A']
x= np.arange(0, np.pi, np.pi/n)
y= a*np.cos(x)
p= FixedPoint(sign='S')
p.range(y, 1e-2)
tutu['COS.VAL']= y
tutu['COS.VAL.INT']= p.to_int(y)
param_to_H(tutu, 'tab_cos_fixed_alt.h')
```

Le tableau généré s'utilise de la même façon que précédemment.

### 5.2.2 Arithmétique

Il est possible de calculer automatiquement les précisions suite à des opérations comme l'addition ou multiplication. Dans cet objectif, les opérateurs `+` et `-` sont surchargés :

— `s= a+b`, `s` est un `FixedPoint` de format $s_e = \max(a_e, b_e) + 1, \quad s_v = max(a_v, b_v)$
— `m= a*b`, `m` est un `FixedPoint` de format $m_e = a_e + b_e, \quad m_v = a_v + b_v$

# 6 Capture vidéo sur Zybo

Cette section donne les informations pour l'utilisation du projet *Vivado* (A) de référence `ZyboCAMGrabL.xpr`.

Le projet `ZyboCAMGrabL` permet d'accéder simplement au flux vidéo en provenance de la caméra. Le flux vidéo est stocké en *SRAM* interne au FPGA, directement, sans passer par un bus AXI. Cette solution a l'inconvénient de limiter la taille des images étant donnée la petite taille des SRAM mais a l'avantage d'être beaucoup plus simple qu'utiliser les bus AXI pour transférer l'image en mémoire *DDR SDRAM*.

Le projet `ZyboCAMGrabL` utilise le bloc *Vivado*
`Zynq/Zybo/ip_repo/axi_video_grabber_L_OV_Zybo`
Ce bloc est un module esclave sur le bus AXI et s'interface à la caméra et à la sortie VGA. Il contient :

- Un contrôleur pour décoder le flux vidéo
- Une memoire video double port accessible par le logiciel
  - Un port d'ecriture pour stocker l'image en mémoire
  - Un port de lecture pour accéder à l'image par logiciel
- une mémoire video double port pour affichage directe
  - Un port d'écriture du flux video
  - Un port de lecture par le controleur VGA
- Un contrôleur VGA pour affichage

La duplication des mémoires est nécessaire car il n'est pas possible d'utiliser une seule mémoire dans laquelle on pourrait lire l'image par le bus AXI et la lire pour l'affichage VGA. Ce schéma de base peut être modifié pour pouvoir écrire dans la mémoire d'affichage depuis le processeur. Dans ce cas le bloc `video_mem` dans `Zynq/Zybo/ip_repo/` peut être utilisé (non documenté). Il est aussi possible d'ajouter un overlay, c'est à dire une image qui s'affiche 'par dessus' l'affichage du flux vidéo. Le principe est que si un pixel de l'overlay est noir, on affiche le flux vidéo, sinon on affiche l'overlay (par exemple un cadre, etc ...).

La totalite du projet est en VHDL :

- Le plus haut niveau est :
  `VHDL/io_video/axi/axi_video_grabber_L_OV_Zybo_v1_0.vhd`
- L'interface entre la mémoire est le bus AXI est :
  `VHDL/io_video/axi/axi_video_grabber_L_OV_Zybo_v1_0_S00_AXI.vhd`
- L'interface entre tous les sous-modules du système :
  `VHDL/io_video/stream_processing/stream_L_capture_memory_display.vhd`

Cette architecture permet de capturer et afficher un flux vidéo de taille 320x240, en monochrome (noir & blanc). Il est possible d'avoir de la couleur, mais cela utiliser beaucoup plus de mémoire.

Pour utiliser le système, lancer *Vivado* (A) dans le repertoire du projet. Dans *'IP Integrator - Open Block Diagram'* pour voir la schématique du projet. Dans *Vivado*, *'File - Lauch SDK'*, ouvrir *Eclipse* pour lancer le code sur le processeur ARM. Ce logiciel sert à configurer la caméra et à accéder au flux video par logiciel.

Voir dans `camera_setup/camera_setup.c` pour un exemple d'accès à l'image vidéo.

Dans *Eclipse*, pour configurer le FPGA, *'Xilinx tools - Program FPGA - Program*

Pour lancer le logiciel, sélectionner le projet `camera_setup`, *'clic droit - Run As - Launch on Hardware'*

Pour observer la sortie du logiciel, voir C.1. Il est possible de modifier ce projet pour :

— Ajouter un bloc de traitement HW (voir 7)

— Paramétrer le bloc de traitement par des registres dont les valeurs proviennent du logiciel

— Ajouter des signaux de debug sur les LED et gérer les boutons poussoirs/switch

— Une autre possibilité est de concevoir un bloc de traitement purement HW, sans interface caméra ni VGA

# 7 Plateforme Zybo Caméra & Processing

## 7.1 Projet Vivado

Cette section décrit l'utilisation du projet Vivado (A) de référence `ZyboCAMVGAProc.xpr`. Le projet `ZyboCAMVGAProc` permet de connecter :

- Une caméra avec protocole parallèle RGB565
- Une mémoire vidéo d'entrée
- Une unité de traitement vidéo mémoire à mémoire, qui peut etre conçue à l'aide de *CatapultC*
- Une mémoire de sortie vidéo
- Une sortie vidéo VGA

Ce projet nécessite le processeur ARM car la caméra est configuré par un logiciel 'bare-metal' sur le processeur ARM du Zynq, par le protocole I2C.

La totalité du projet est en VHDL, seule la partie processeur est réalisée en schématique Vivado. Le projet Vivado intègre le VHDL dans le plus haut niveau de conception `design_1_wrapper`, accessible par l'onglet *'Project Manager - Hierarchy'*. Au cas où Vivado l'effacerait, il existe une copie de `design_1_wrapper.vhd` dans `VHDL/io_video/systems/Zybo_CAM_VGA_Proc_wrapper.vhd`

`design_1_wrapper` incorpore l'entité VHDL `Zybo_CAM_VGA_Proc` qui est dans `VHDL/io_video/boards/Zybo_CAM_VGA_Proc.vhd`

`Zybo_CAM_VGA_Proc` incorpore les contrôleurs caméra et VGA, ainsi que les mémoires vidéo et l'unité de traitement.

Pour insérer son propre module de traitement, il suffit de modifier `Zybo_CAM_VGA_Proc`, soit en renommant votre unité de traitement `imgproctest`, soit en remplaçant le nom `imgproctest` par celui de votre unité (ligne 181 et 332 de `Zybo_CAM_VGA_Proc.vhd`). Pour faire son propre projet *CatapultC*, voir B.

Pour utiliser le projet *Vivado*, une fois que tous les noms sont configurés, dans *Vivado* :

- *'Program and debug => generate Bitstream'*
- Brancher la carte Zybo sur le port USB et la sortie VGA
- *'Program and debug => Open Hardware Manager => Open Target'*
- *'Program and debug => Open Hardware Manager => Program Device (celui qui est proposé)'*

Puis dans *Vivado*, *'File - Launch SDK'* Cliquer le projet *'Camera setup'*, puis bouton droit *'Run As -> Launch on Hardware'*.

Bravo, il vous reste à debugger !!

Il est possible d'interagir avec le logiciel par le port USB/Série, voir C

## 7.2 CatapultC

Pour configurer *CatapultC*, voir B. Il existe un exemple de projet dans `tmp/CC/ImgProcTest`, `ImgProcTest.cpp` 'inverse' la luminosité des pixels au dessus de la bissectrice. Ce projet

écrit les pixels de sortie en lisant la mémoire vidéo d'entrée. Son interface correspond à celle des mémoires *Vivado*, sur 8 bit.

Attention : dans le projet *CatapultC*, au moment de configurer les mémoires, selon la version de *CatapultC*, ne pas oublier de régler les signaux de contrôle we (Write Enable) et re (Read Enable) actif sur niveau `1` (0 par défaut) [1].

A ce projet de base, vous pouvez

- Ajouter des signaux de controle start/done/ready depuis CatapultC *'Mapping -> Solution cocher start/done/ready'*.
  Ces signaux vous servirons à lancer l'unité lorsque vous en avez besoin. Il est possible de les connecter aux boutons poussoirs et switch de la carte
- Ajouter vos propres interface
  - Données et paramètres (en provenance de mémoires ou du logiciel)
  - Autres tableaux
  - controle (switch et boutons) ou debug (affichage sur LED)

Dans *'Tools - Set Options - Flow - Precision RTL '* décochez l'onglet *'Add IO Pads'* (très important!!), puis *'Apply & Save '*.

Avec *CatapultC*, faire la synthèse d'architecture (arrivez jusqu'à l'étape *RTL*). Dans la fenetre de gestion des fichier, dans l'onglet *'Synthesis'* Lancez *'Precision Synthesis'* en batch : *'Synthesize rtl.vhdl bouton droit - Launch Precision Batch'* Ceci produit une netlist au format `edf`, que vous trouverez à l'aide de la commande :

```
> find .  -name '*edf'
```

Comme *CatapultC* génère un nouveau répertoire à chaque modification, vérifiez bien que vous avez la bonne version.

Une fois le `edf` généré, dans *Vivado*, dans *'Project Manager/Hierarchy*, remplacer le `edf` par le nouveau en cliquant dessus *'clic droit - Replace File'*. Astuce : copiez le `edf` quelque part, le sélectionner par *Vivado*, puis l'écraser par chaque nouvelle version, ce qui évite de changer le projet *Vivado*. Puis, lancer à nouveau *'Generate Bitstream'* et vérifiez qu'il prend bien le nouveau `edf` (ou, par exemple, faire un projet tel que les LEDs indiquent le numéro de version).

Bravo, c'est terminé!!

1. L'interface graphique est capricieuse et n'hésitez pas à essayer plusieurs fois

# A Vivado

Pour pouvoir utiliser Vivado :
`> source /softslin/vivado_17.1/Vivado/2017.1/settings64.sh`

# B CatapultC

Pour faire son projet CatapultC copier `/tp-fmr/smancini/SLE/Projets/bash_mentor` ou `/tp-fmr/smancini/SEI_SoC_CNN/bash_mentor` puis `> source bash_mentor`. Catapult se lance par `> catapult`

# C Zybo

## C.1 Port USB/Série

Le processeur ARM de la Zybo peut être connecté à un PC par USB et il est possible d'interagir par une console texte sur le port USB/Série.

La commande :
`> minicom -D /dev/ttyUSB0 (ou /dev/ttyUSB1)` Ceci permet l'affichage des `printf` du code sur le processeur ARM. Il est possible d'interagir avec le code ARM en lisant le port USB/Série par des `scanf` dans le code du processeur ARM. Dans de cas `minicom` transmet sur le port USB/Série le texte saisi par l'utilisateur et le code ARM le "récupère" ensuite.

# D Génération de RAM

Exemple de code d'une SRAM générée avec un contenu initialisé. On ajoutera les bibliothèques adéquates et les termes entre $ seront remplacés par les noms ou listes de valeurs. Si besoin le type de `addr` sera adapté.

```
entity ${ram_name} is
  generic(
    --parameters size of the memory and width of the words
    --see in the manual for all possobilities
    -- total size of the memory is cellCount*wordSize
    cellCount : integer := ${ram_cc};  -- number of ram entries
    wordSize : integer := ${ram_ws};   -- size of ram data word
  );
  port( clk : in std_logic;
        addr : integer;
        din: in std_logic_VECTOR(wordSize-1 downto 0);--data in
        dout: out std_logic_VECTOR(wordSize-1 downto 0));--data out
end  ${ram_name};
```

```
architecture arch of  ${ram_name} is
  --the memory
  type ram_type is array (0 to cellCount-1) of std_logic_vector(wordSize-1 downto 0);
  signal ram : ram_type := (
    ${ram_data}  -- Memory content to be replaced by list of values
    );
  attribute block_ram : boolean;
  attribute block_ram of RAM : signal is TRUE;
begin
  portIO: process (clk)
  begin
    if (clk'event and clk =  '1' ) then
        dout <= ram(addr);
    end if;
  end process portIO;
end arch;
```