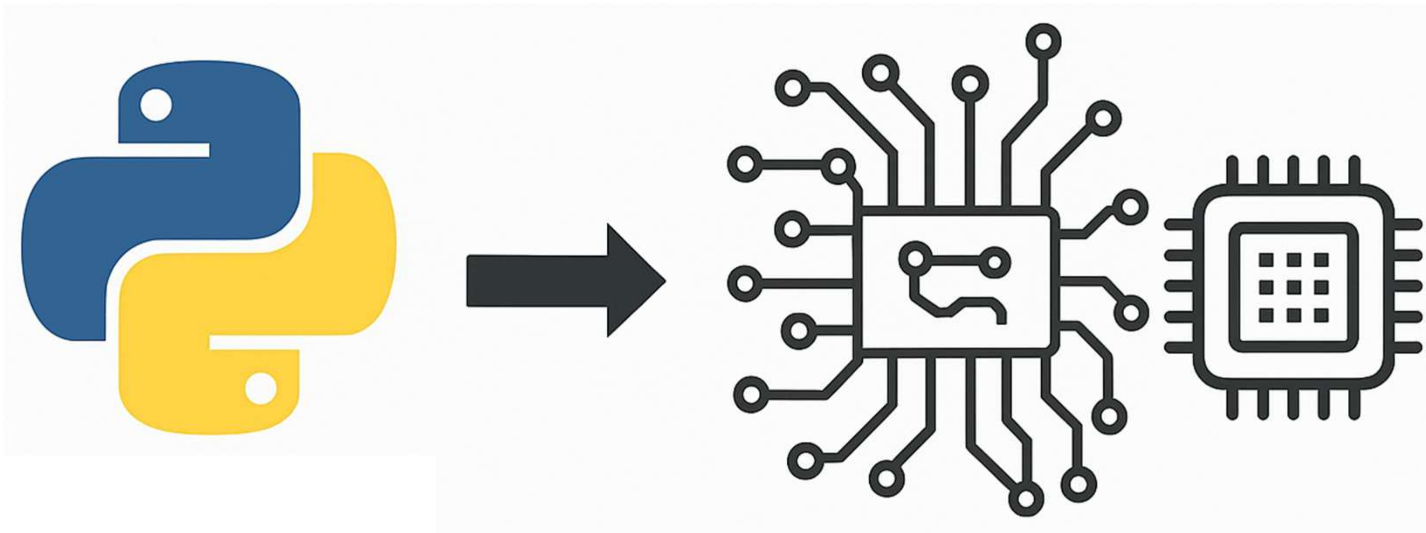
Framework de conversion Python -> VHDL



Geoffrey GALBIT

IP2I Lyon

Présentation du flow

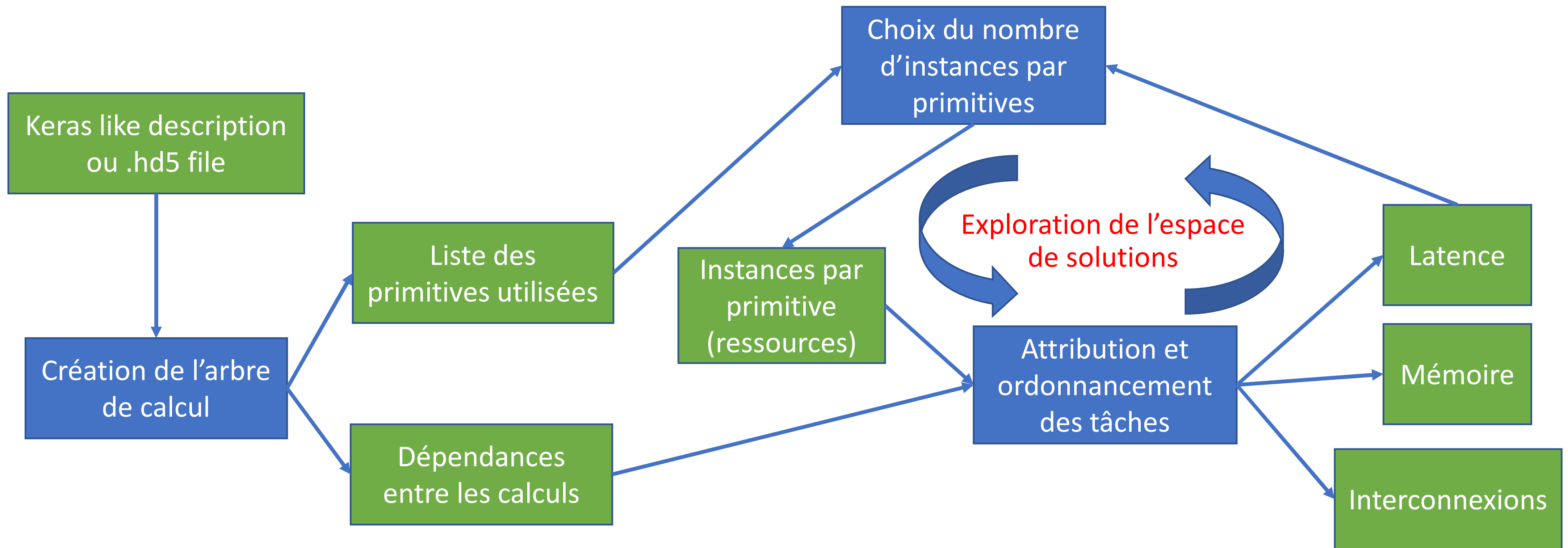
Dev SW:
- Adaptation lib pour NN choisi
- Prise en compte des param HW
- Entrainement avec dynamique réelle
- Evaluation perf et implémentatibilité

Dev passerelle SW/FW:
- Lib hdl
- Script transfo NN -> rtl
- Simulation netlist -> validation perf
- Synthèse -> valider estim resources

Dev FW fixe:
- Choix cible & host architecture
- DAQ drivers

Scientific Data
Topology and parameters definition
Software training
Neural Network
Performance evaluation
Implementation feasibility
HDL Lib
Rtl generation
Netlist Simulation
Synthesis & Implementation
Host Architecture
FPGA
DAQ Drivers
FPGA related software
Efficiency measurement

Dimensionnement du re-use



Création de l’arbre de calcul

- Entrées : Description Keras like ou import depuis un fichier hdf5

Ces descriptions sont au niveau «layer»

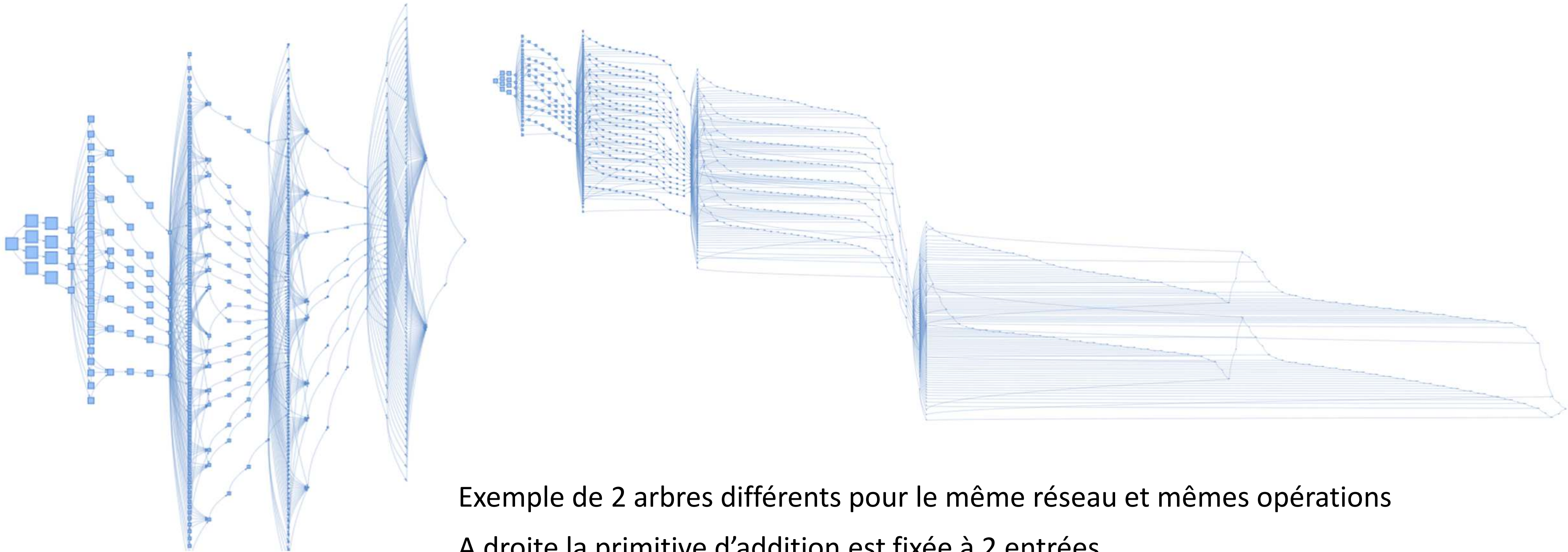
La lib utilise donc des fonctions Python qui mappent les fonctions Keras aux primitives de la library vhdl

```
def addConv1D(lnConv1DInputId, filters, kernel_size, layer_name = "UNNAMED"):

  lnConv1DOutputId = []

  for nFilter in range(filters):

    ##Kernel instanciation
    lnKerOutputId = []

    #For each input flows of the filter
    for nInputId in lnConv1DInputId:

      #Get the input flow size (the output flow of the input node)
      inputFlowSize = ddNode[nInputId]["OutputFlowSize"]
      outputFlowSize = inputFlowSize-(kernel_size-1)

      nKerId = addNode({"Type":"Kernel1D, "InputFlowSize":inputFlowSize,"OutputFlowSize": outputFlowSize,"layer_name":layer_name,
"kernel_size":kernel_size})
      oProcessingTree.add_node(nKerId)
      oProcessingTree.add_edge(nInputId, nKerId)
      lnKerOutputId.append(nKerId)

    lnAdderOutputId = addReductionAdder(lnKerOutputId, layer_name=layer_name)

    lnConv1DOutputId.append(lnAdderOutputId[0])

  return lnConv1DOutputId
```

Au final, cela crée un graph orienté où chaque node est une primitive de calcul, chaque edge est un dataflow

Création de l’arbre de calcul

Les fonctions «Keras like» permettent de définir la granularité des primitives générées



Exemple de 2 arbres différents pour le même réseau et mêmes opérations

A droite la primitive d’addition est fixée à 2 entrées.

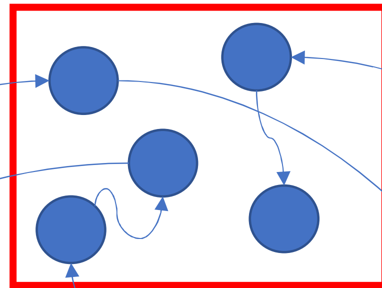
A gauche il existe une primitive pour chaque nombre possible d’entrée (ces primitives sont donc différentes et ne sont pas interchangeables).

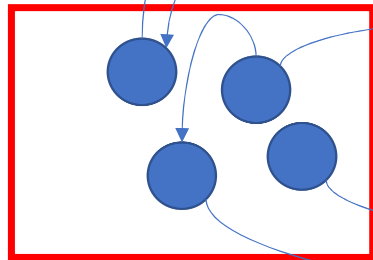
Affectation et ordonnancement des tâches

Une tâche correspond à la transformation d'un ou plusieurs dataflow d'entrée en un dataflow de sortie qui s'opère dans une processing unit.

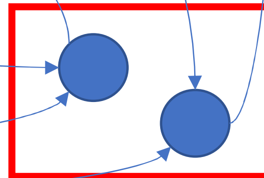
L'affectation est donc triviale dans le cas du design minimal (1 seule instance de chaque primitive).

Processing unit type B

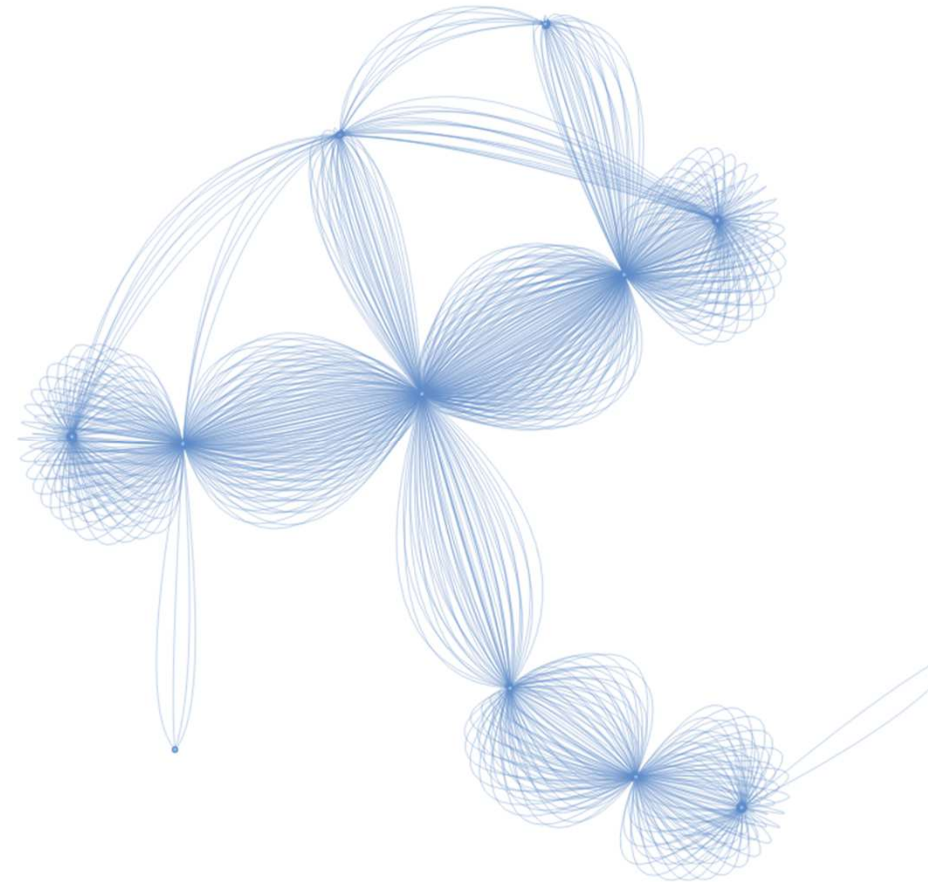




Processing unit type A

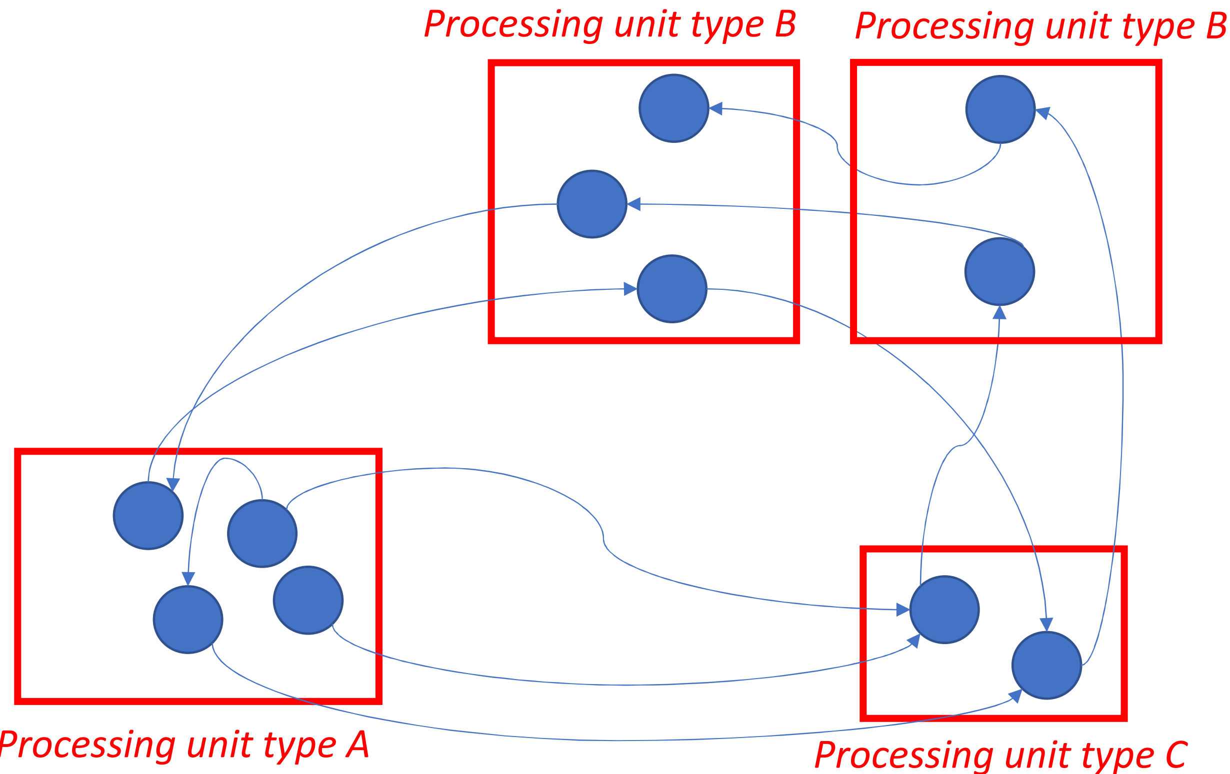


Processing unit type C



Affectation et ordonnancement des tâches

Lorsque plusieurs instances d'une primitive sont disponibles, l'algorithme d'affectation doit séparer les tâches intelligemment lors de l'ordonnancement (le but est ici de minimiser la latence du système).



Exemples de règles d'optimisation basées sur le graph:

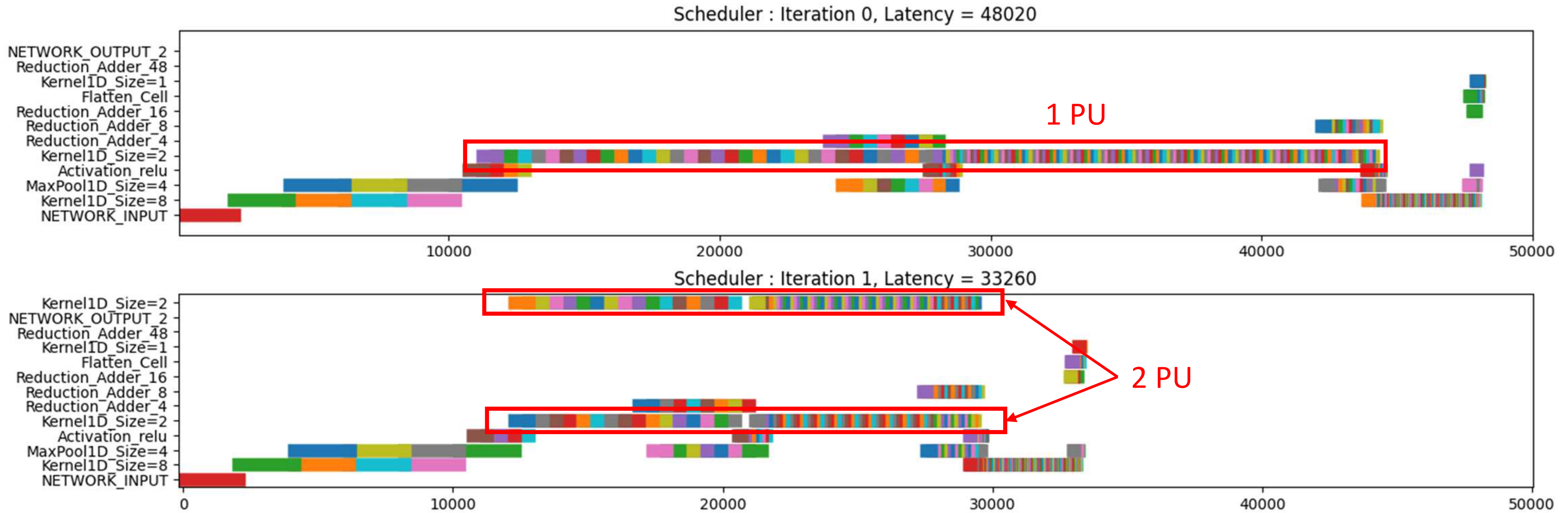
2 noeuds ont plus de chance d'être séparés si :

- Ils reçoivent le même dataflow d'entrée (possibilité de synchroniser l'envoi et de paralléliser le calcul)
- Ils sont voisins directs dans l'arbre de calcul (possibilité de pipeliner les calculs)
- Ils nourrissent le même noeud

Affectation et ordonnancement des tâches

L'affectation optimal dépend également de l'ordonnancement temporel des tâches

Représentation temporel du gain de duplication d'une processing unit



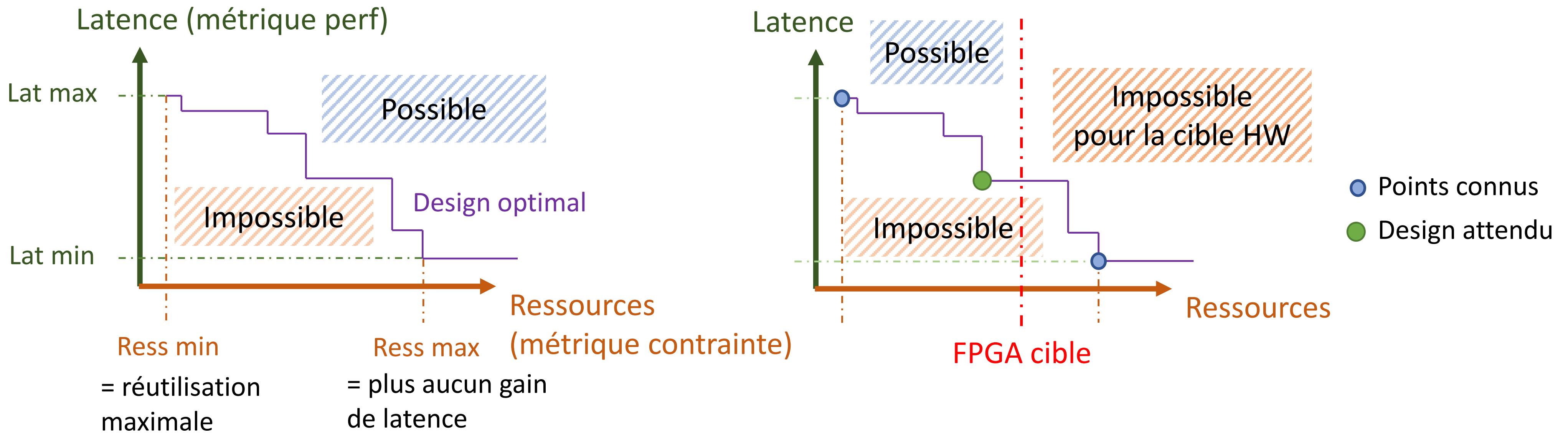
L'algorithme actuellement implémenté est plutôt basique mais donne de bons résultats sur le cas test

Exploration de l’espace de solutions

TradeOff entre différentes grandeurs :

- Ressources disponibles (DSP, mémoire, routage …) ← Dépends du FPGA et du système hôte
- Latence
- Précision
- Puissance
- …

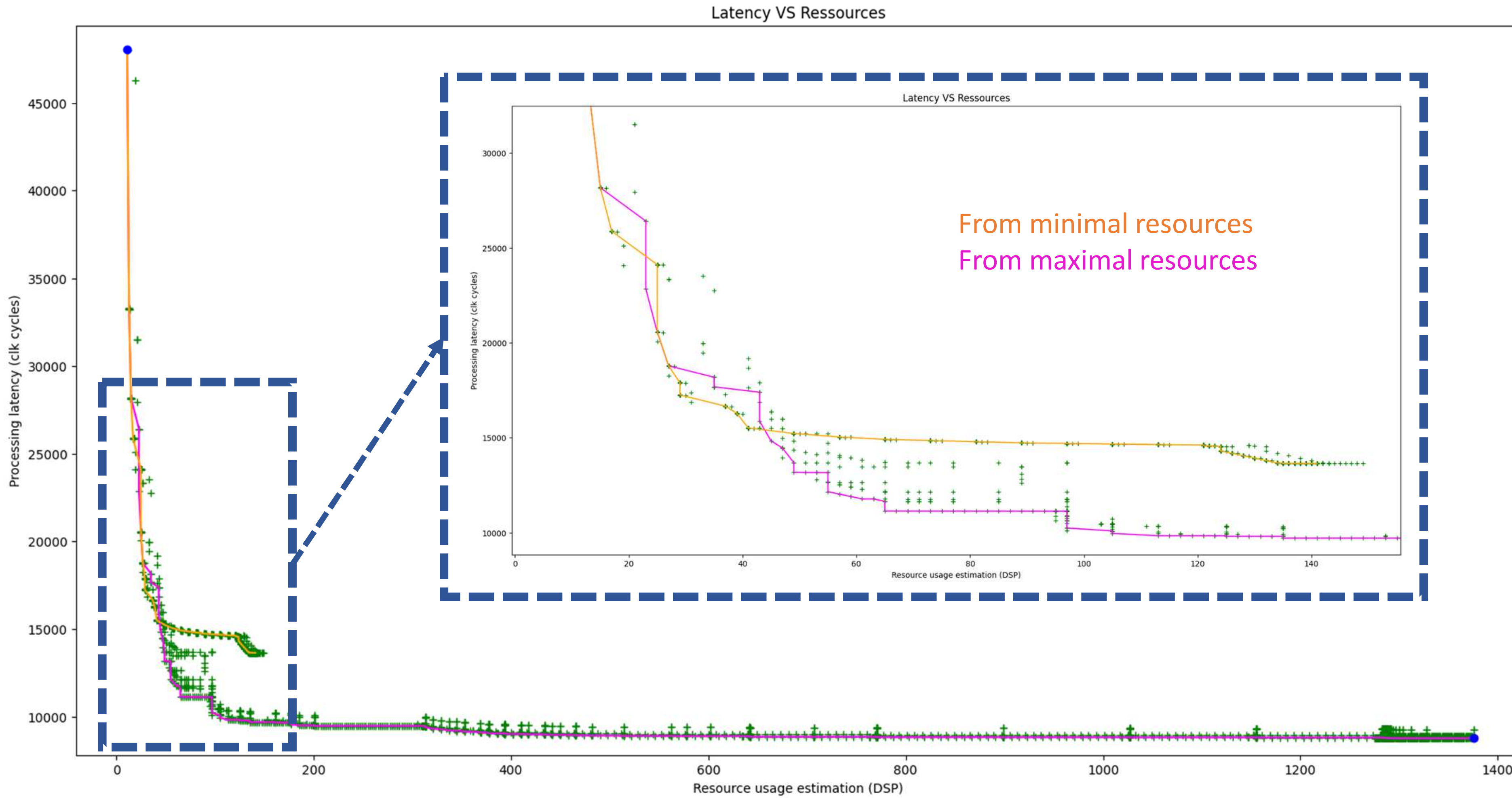
(Latence, Précision, Puissance, … :) Contraintes extérieures



Approche choisie pour converger vers le design cible : Partir du design minimal (latence max) et en itérer en dupliquant le module de calcul permettant le meilleur gain en latence (valeur facile à estimer).

Exploration de l’espace de solutions

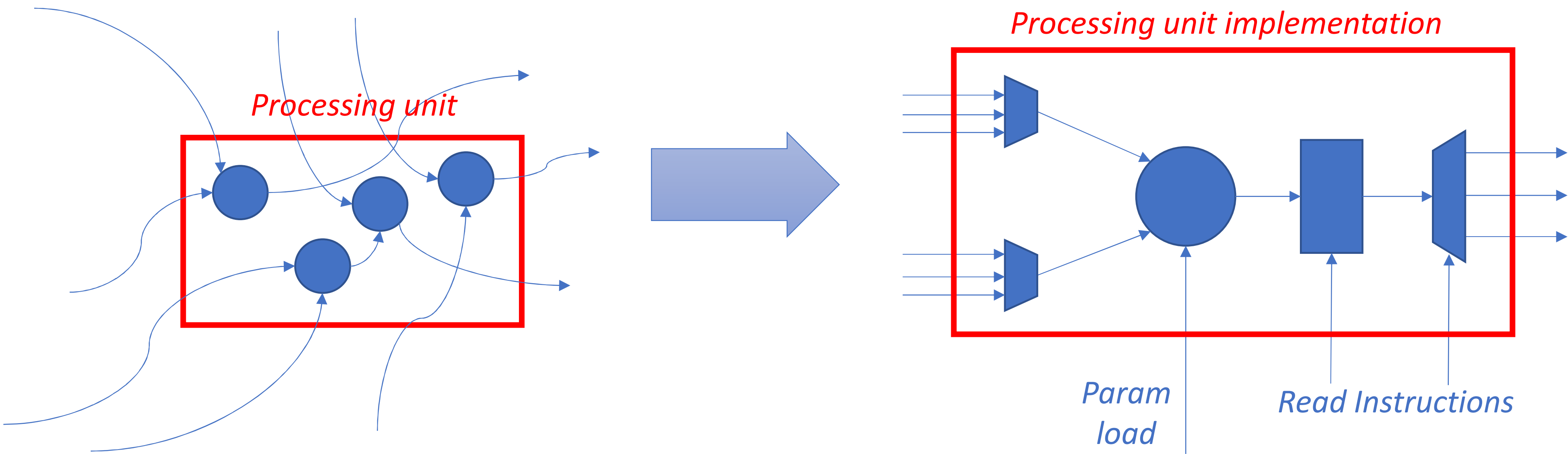
Compromis ressources/latence (algorithme glouton, aucune optimisation)



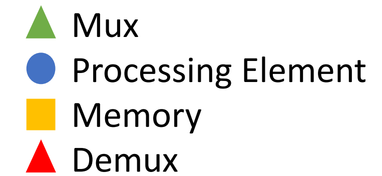
Préparation du design à synthétiser

Chaque processing unit est remplacée par un bloc contenant:

- Des MUX pour gérer le routage en entrée (1 par entrée du composant)
- L'élément de calcul
- Une mémoire qui stocke les dataflows générés
- Un DEMUX pour gérer le routage en sortie



Etapes suivantes



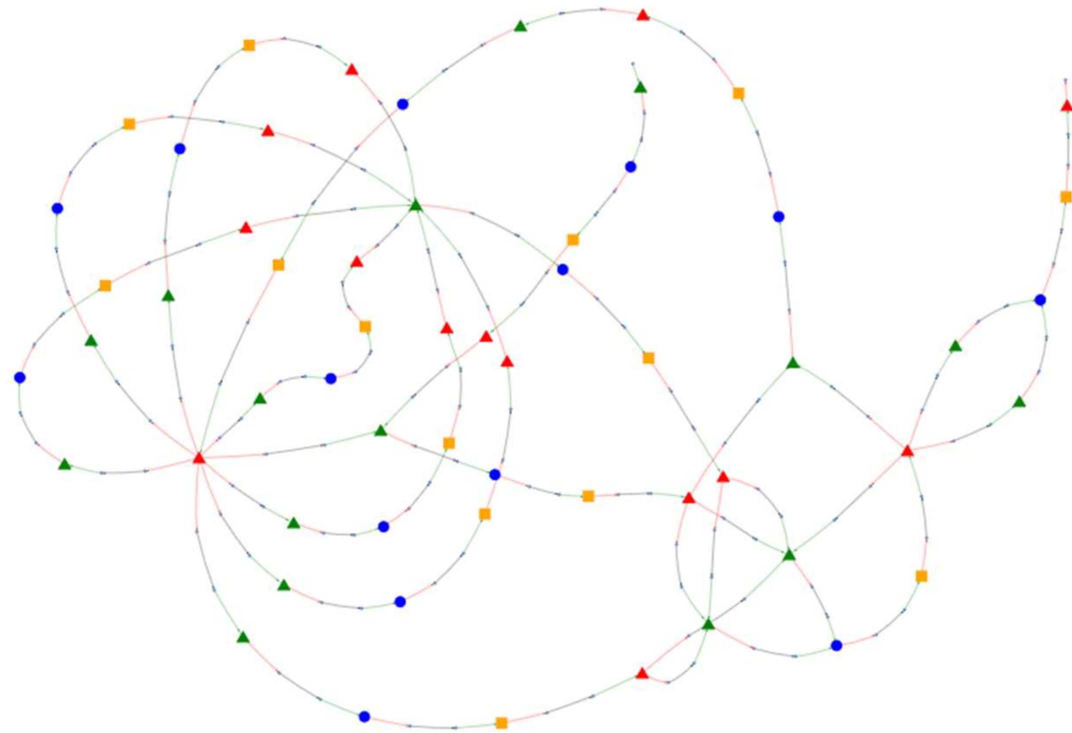
D'après l'attribution des nodes aux PU et le séquençage des tâches:

Etude des besoins en mémoire (pour chaque mémoire de sortie de PU)
-> Permet de dimensionner les mémoires en fonction du nombre de dataflow générés par une processing unit

Etude des besoins en routage entre les PU
-> Permet d'identifier les différents modes de routage

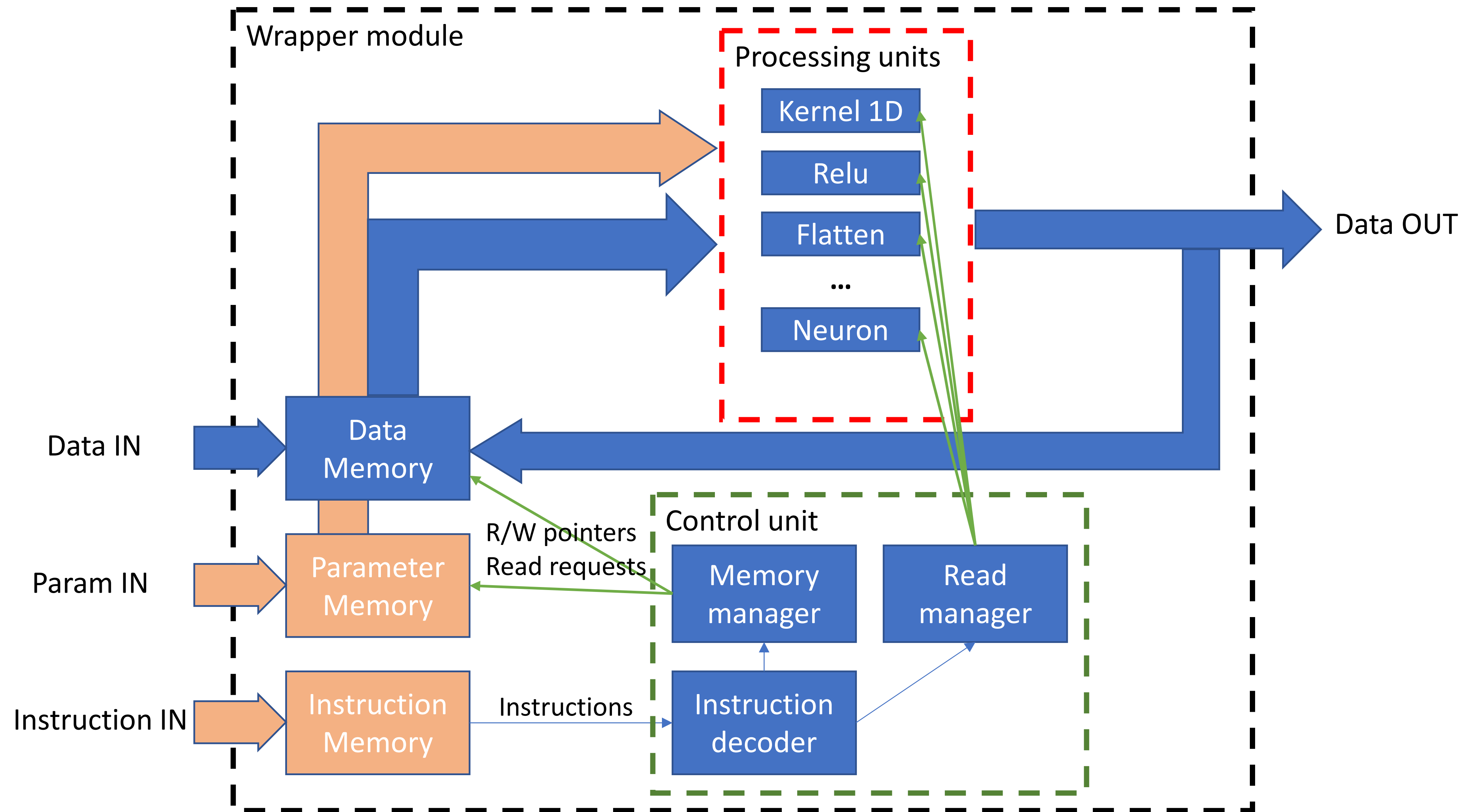
Génération des instructions avec le séquençage de:

- L'envoi des paramètres au PU (si nécessaire)
- Des ordres d'envoi des dataflows



Design généré pour l'implémentation minimale (11 PU)

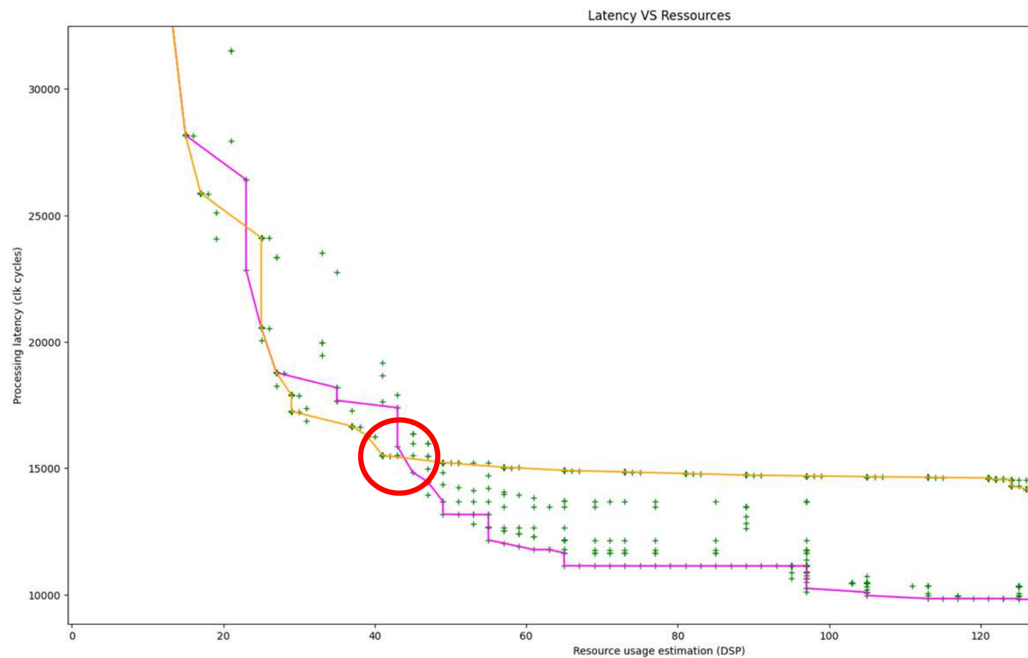
Full design



Résultats

Choix de la configuration:

- Exploration de l’espace en Latence VS DSP depuis le design minimal
- Critère d’arrêt : latence inférieure à 16k cycles

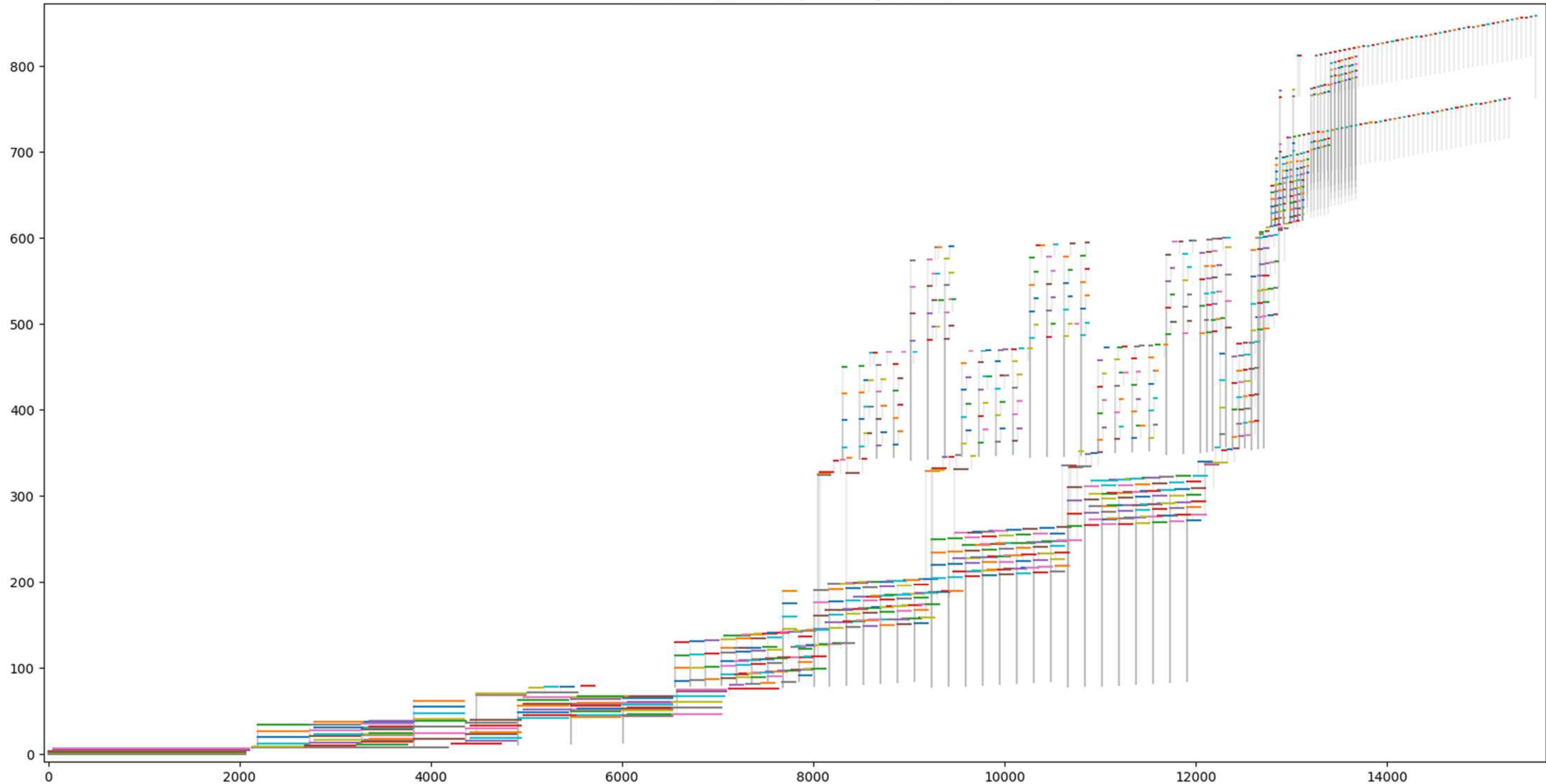


31 processing units:
NETWORK_INPUT x 1
Kernel1D_kernel_size=8 x 4
MaxPool1D_pool_size=4 x 3
Activation_Relu x 1
Kernel1D_kernel_size=2 x 4
Reduction_Adder_nIn=2 x 7
Flatten_Cell_0/6 x 1
Flatten_Cell_1/6 x 1
Flatten_Cell_2/6 x 1
Flatten_Cell_3/6 x 1
Flatten_Cell_4/6 x 1
Flatten_Cell_5/6 x 1
Kernel1D_kernel_size=1 x 4
NETWORK_OUTPUT_nIn=2 x 1

Résultats

Résultat du scheduling :

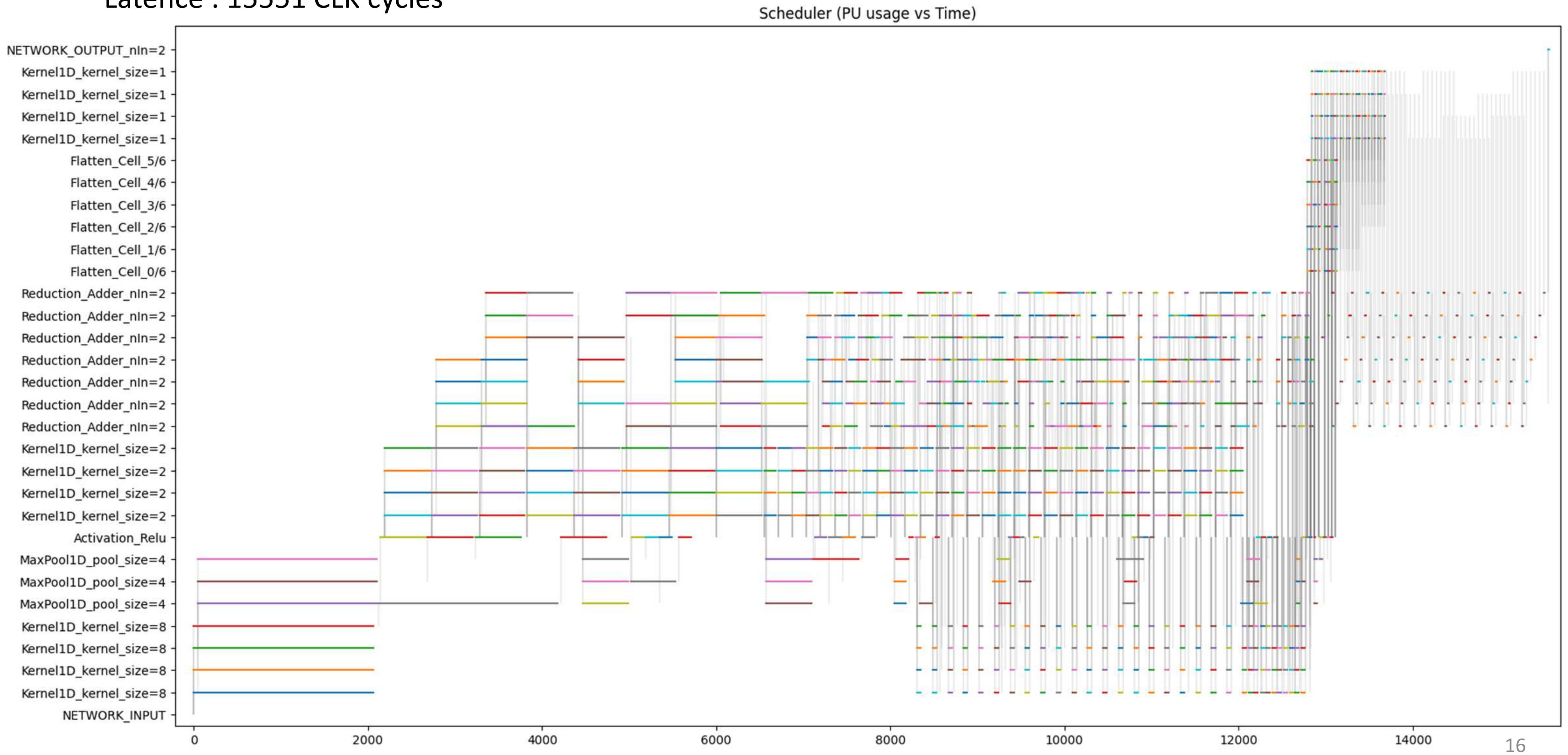
Latence : 15551 CLK cycles



Résultats

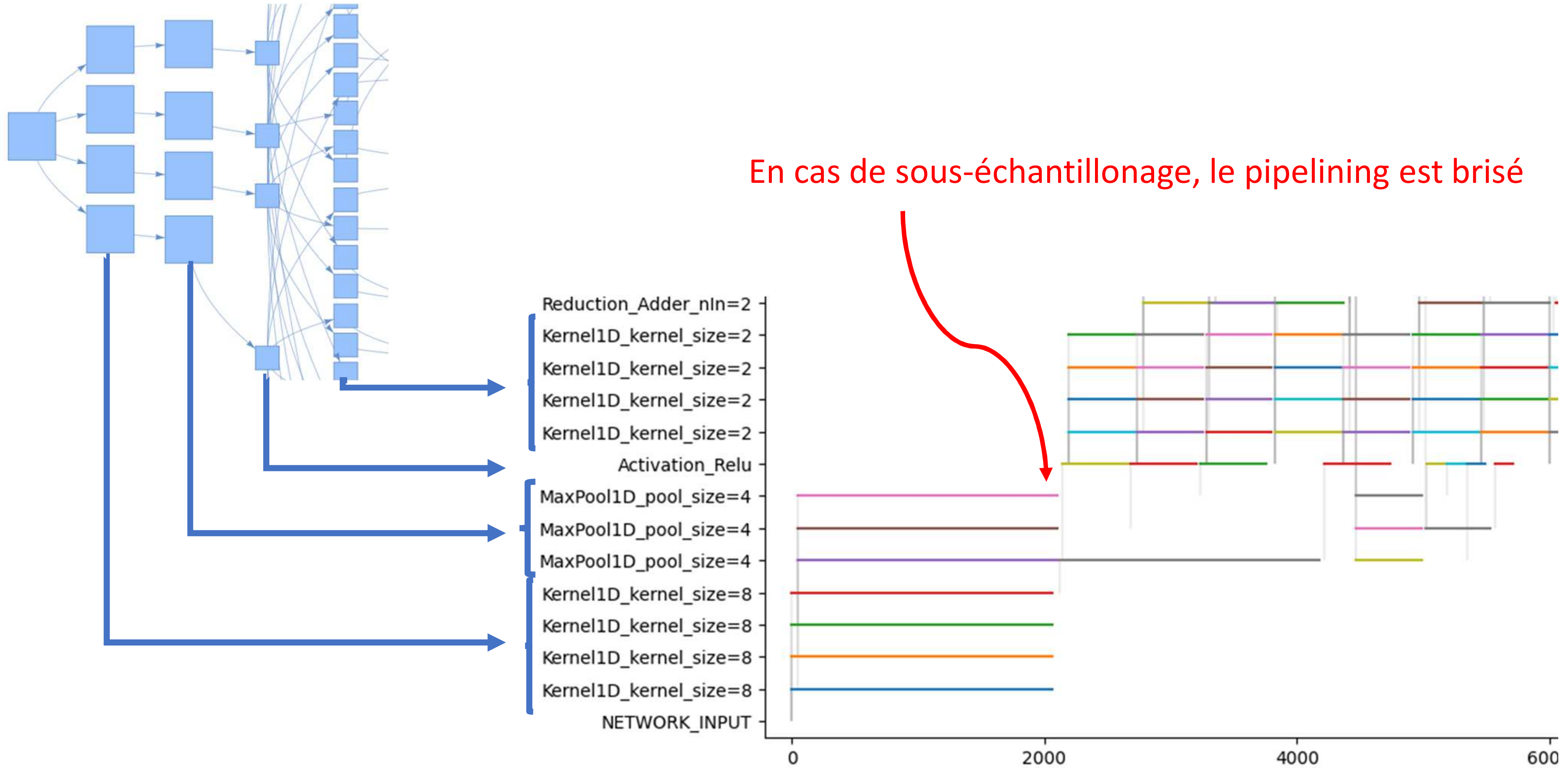
Résultat du scheduling :

Latence : 15551 CLK cycles



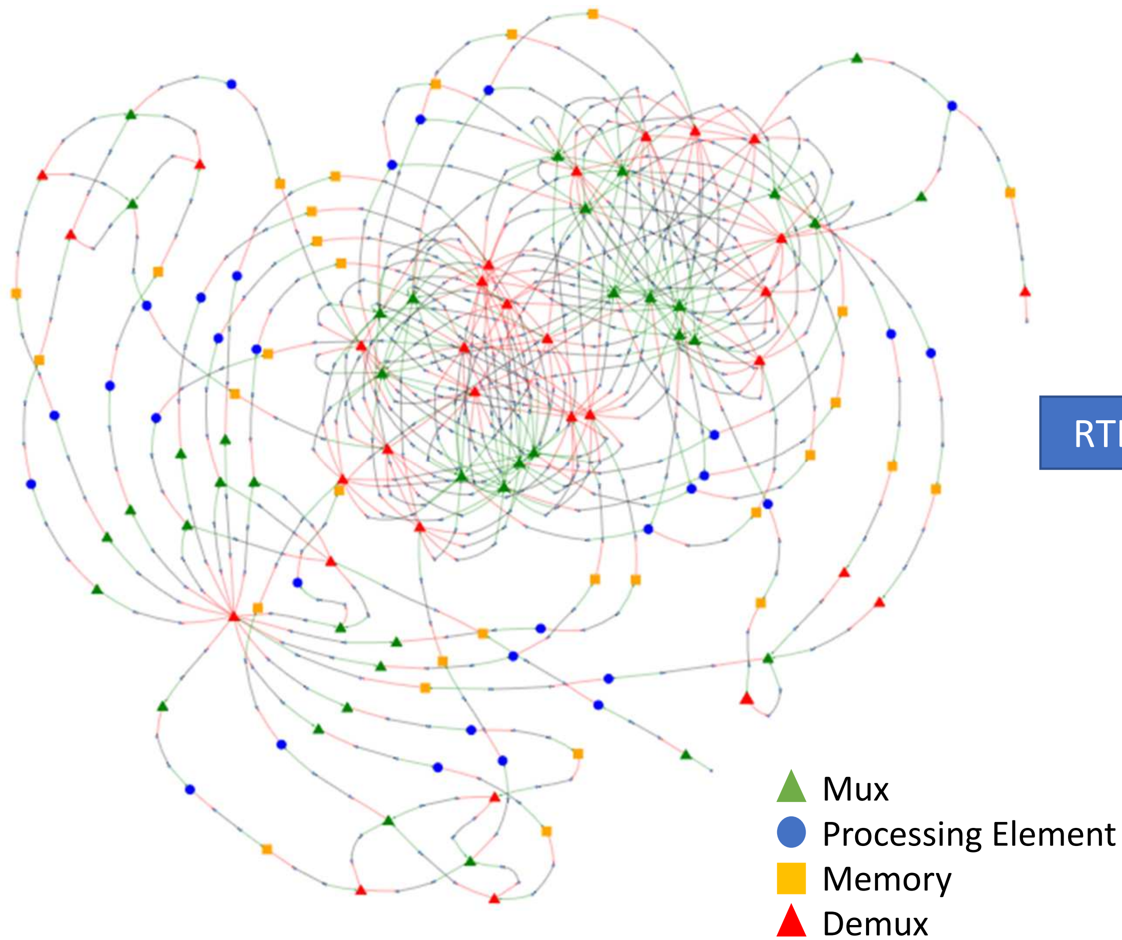
Résultats

Résultat du scheduling :

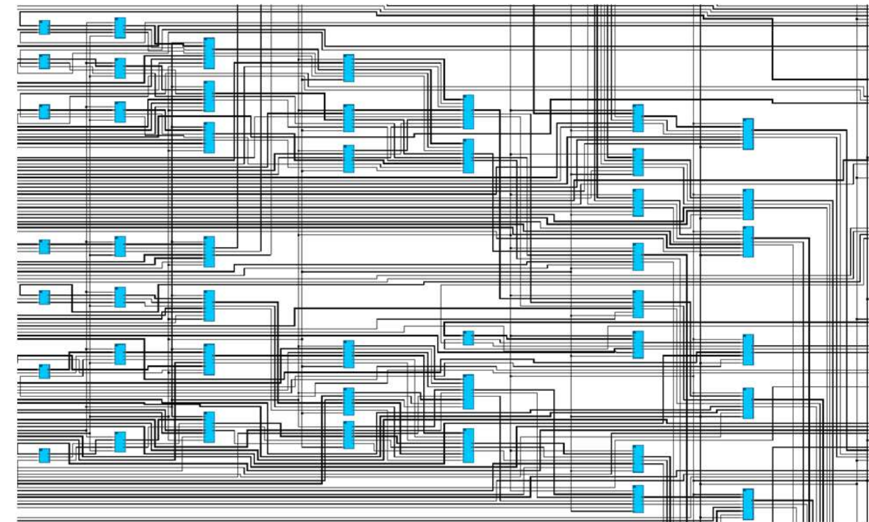


Résultats

Résultat de la transformation en graph synthétisable:



RTL gen



Résultats

Génération de paramètres pour la control unit:

Time *Instr_type* *PU_Id* *Param_value*

```
0 PARAM 1 11111111111111111111111100110100
...
8 PARAM 1 00000000000000000000001001101111
9 PARAM 2 00000000000000000000000101111010
...
17 PARAM 2 11111111111111111111111101010110
18 PARAM 3 00000000000000000000000100110100
...
26 PARAM 3 00000000000000000000000101110100
27 PARAM 4 11111111111111111111111010111111
...
35 PARAM 4 11111111111111111111111100011110
36 INSTR 0 0 2047 15
86 INSTR 1 0 2040 1
87 INSTR 2 0 2040 1
88 INSTR 3 0 2040 1
2164 INSTR 4 0 2040 1
2179 INSTR 5 0 509 1
2229 PARAM 9 00000000000000000000000101000100
2230 PARAM 9 00000000000000000000000110100000
2231 PARAM 9 00000000000000000000000101000101
2232 PARAM 10 11111111111111111111111011110011
2233 PARAM 10 00000000000000000000000001001111
...
```

Rd_Start_Addr

Rd_Stop_Addr

Demux_Mask

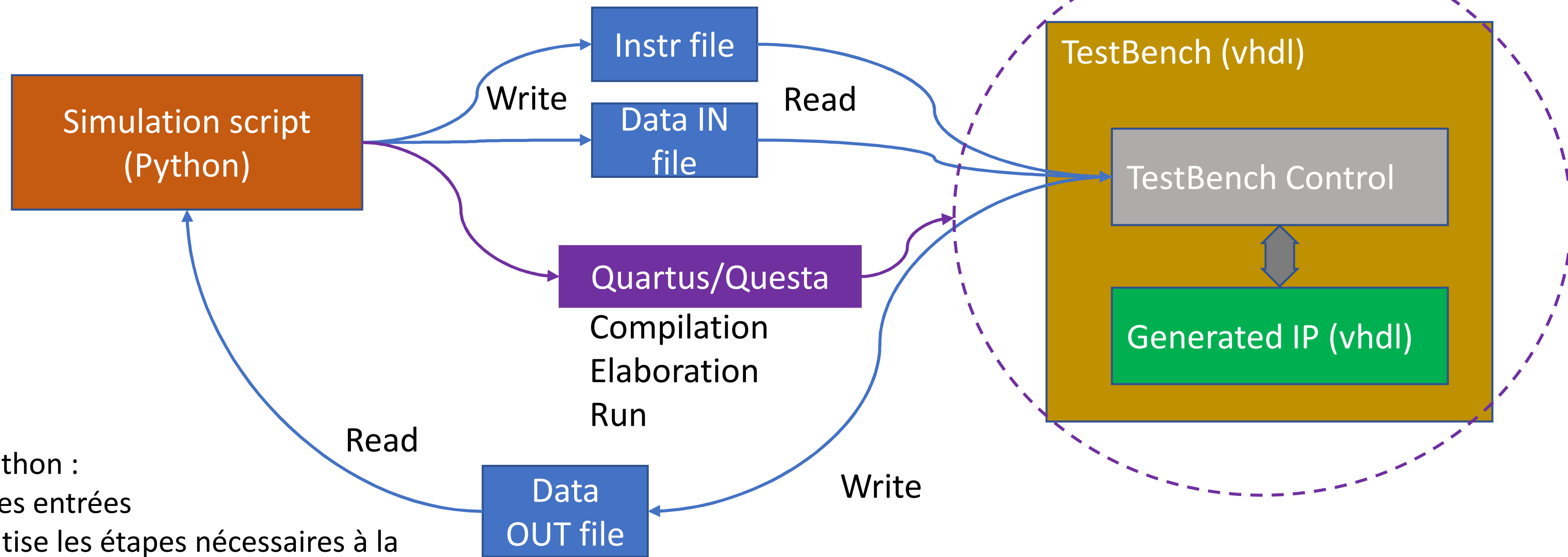
- Load Param to PU 1
- Load Param to PU 2
- Load Param to PU 3
- Load Param to PU 4

Load Param to Kernel_1D

- Launch Input Data read to Kernel_1D
- Launch Kernel_1D out read to MaxPool modules

Choix de design:
Un seul bus d'instruction provenant de la control unit (une instruction par CLK cycle) cela introduit un léger shift temporel des tâches

RTL Simulation



Le script Python :

- Fourni les entrées
- Automatise les étapes nécessaires à la simulation
- Récupère les résultats et les analyse (comparaison avec calcul soft)

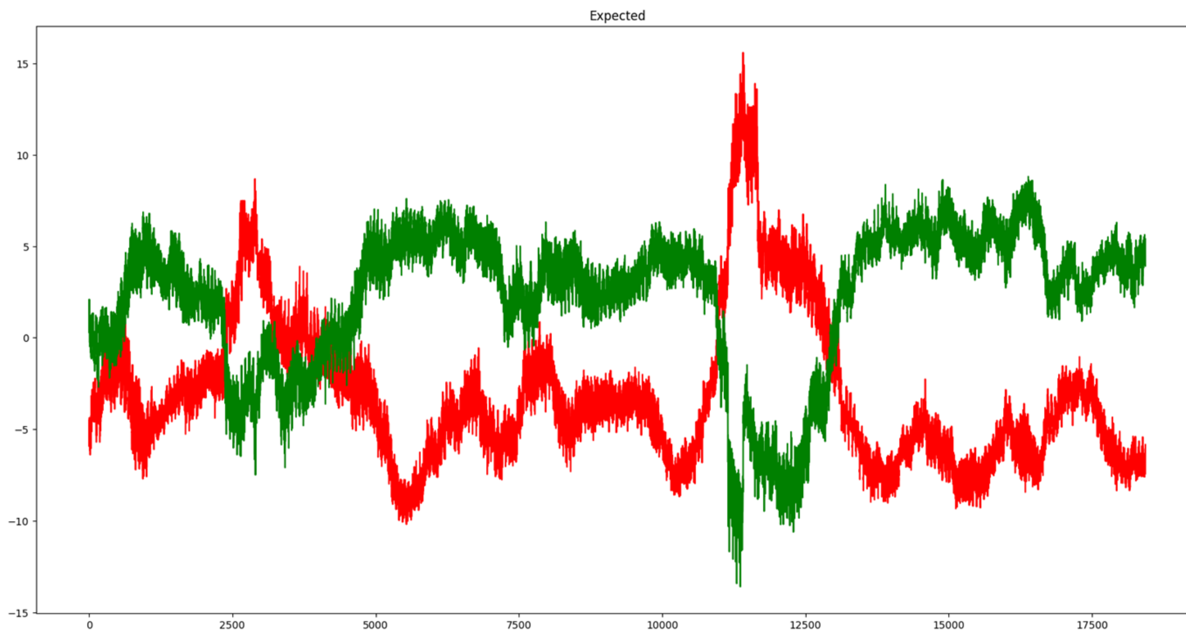
Le TestBench :

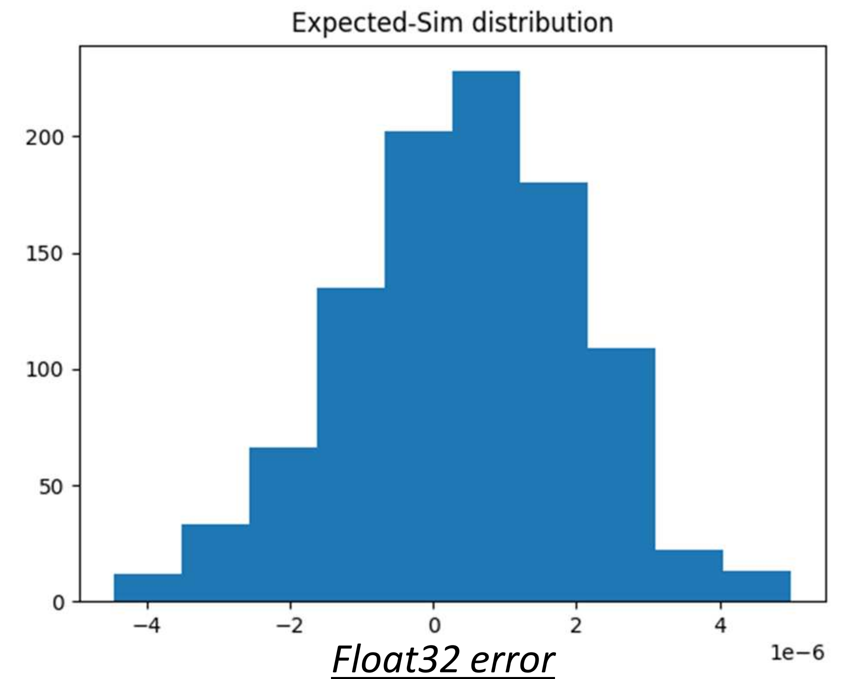
- Génère l'horloge
- Instancie l'IP générée
- Séquence le test
- Fait des accès aux fichiers

Résultats

Simulation RTL du design généré :

Implémentation float 32bits

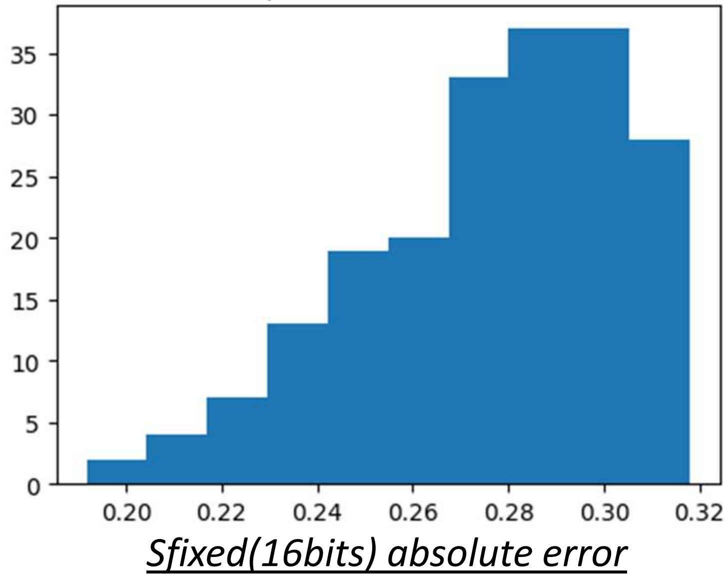




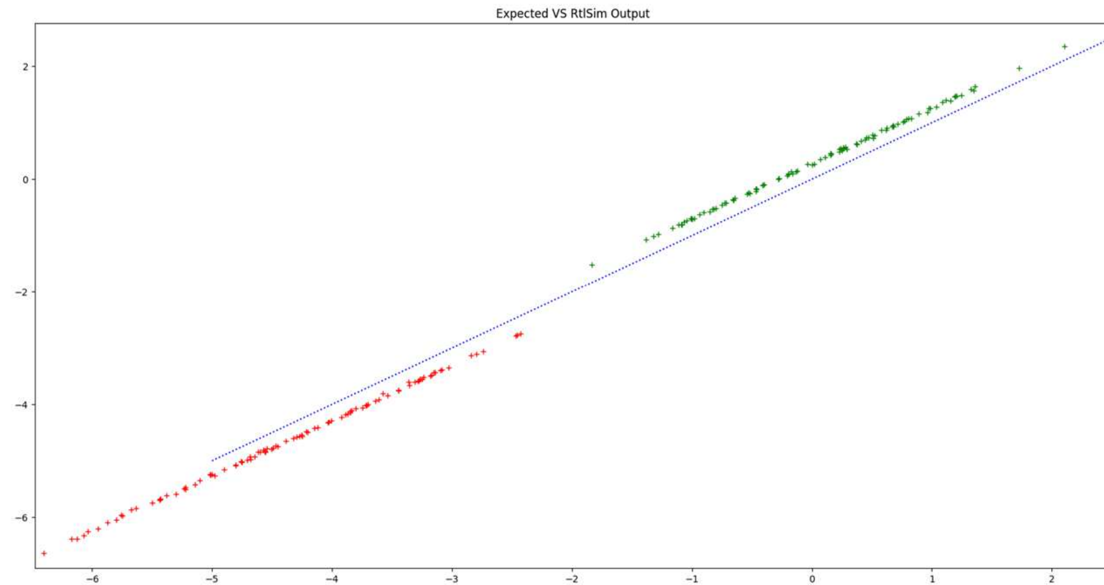
Résultats

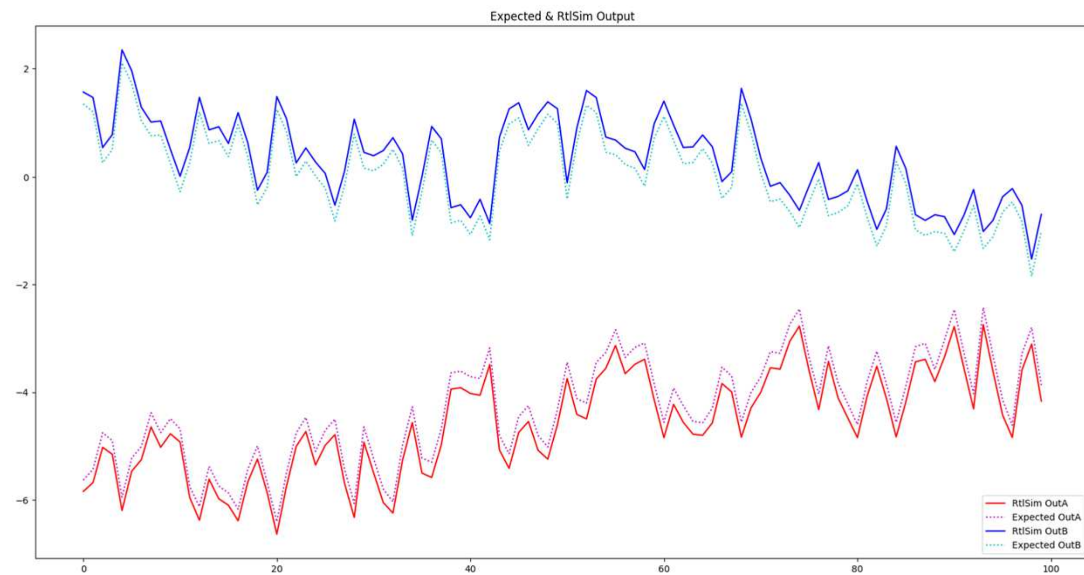
Simulation RTL du design généré :

Implémentation sfixed 16bits (ipart=6bits)



Sfixed(16bits) absolute error





Aucun réentrainement n'a été effectué:

- Troncature des données d'entrées et paramètres
- Aucun arrondi au niveau des opérations

Résultats

Resources

	Resource	Usage
1	Estimate of Logic utilization (ALMs needed)	7010
2		
3	Combinational ALUT usage for logic	7009
4		
5	Dedicated logic registers	11758
6		
7	I/O pins	4131
8	Total MLAB memory bits	0
9	Total block memory bits	3023680
10		
11	Total DSP Blocks	44
12		
13	Maximum fan-out node	CLK_IN
14	Maximum fan-out	13425
15	Total fan-out	104443
16	Average fan-out	4.24

Table 4. Maximum Resource Counts for Cyclone V E Devices

Resource		Member Code				
		A2	A4	A5	A7	A9
Logic Elements (LE) (K)		25	49	77	150	301
ALM		9,430	18,480	29,080	56,480	113,560
Register		37,736	73,920	116,320	225,920	454,240
Memory (Kb)	M10K	1,760	3,080	4,460	6,860	12,200
	MLAB	196	303	424	836	1,717
Variable-precision DSP Block		25	66	150	156	342
18 x 18 Multiplier		50	132	300	312	684
PLL		4	4	6	7	8
GPIO		224	224	240	480	480
LVDS	Transmitter	56	56	60	120	120
	Receiver	56	56	60	120	120
Hard Memory Controller		1	1	2	2	2

FPGA, Altera, 5CEBA4F17C8N, Cyclone V, 49000 Portes, 49000 Cellules, 18480 Blocs, FBGA 256.

Code commande RS: **830-3565** | Référence fabricant: **5CEBA4F17C8N** | Marque: Altera

Sous-total (1 unité)*

69,70 € HT

83,64 € TTC

Conclusions et next steps

- Le framework peut générer automatiquement le design du cas test
- La simulation comportementale du design valide les résultats
- La performance atteignable en virgule fixe pour ce cas rend inutile l'utilisation des float32
- Cet outil est utile pour adapter rapidement des réseaux de calcul sur des cibles limitées

Next steps

- Réaliser l'implémentation du système complet (partie auto-générée + wrapper) sur un Cyclone V
- Optimisation de la gestion de la mémoire des processing units
- Finaliser l'interface d'ajout de modules dans la library rtl
- Prise en charge d'autres types de layers définis par Keras

Backup

Library hdl

Pour chaque composant de la lib:

Un fichier contenant la description RTL

Définition de l'entité (paramètres génériques et ports)

```
entity Kernel1D is
  Generic (
    KERNEL_SIZE : natural := 8
  );
  Port (
    CLK_IN     : in std_logic;
    RESET_IN   : in std_logic;
    PARAM_IN   : in data_bus_array_type(0 to 0);
    DATA_IN    : in data_bus_array_type(0 downto 0);
    DATA_OUT   : out data_bus_array_type(0 downto 0)
  );
end Kernel1D;
```

Paramètres génériques RTL -> fixé lors de l'appel du composant

Bus de paramètres dynamiques (optionnel)
Bus d'entrée (possiblement plusieurs)
Bus de sortie (1 par composant)

Library hdl

Pour chaque composant de la lib:

Un fichier contenant la description RTL

Définition de l'architecture (fonctionnalités internes du composant)

```
architecture Stratix10Dsp of Kernel1D is
…
begin
…
  -- DSP Chain instanciation ---
  DSP_GEN:
  for i in 1 to KERNEL_SIZE-1 generate
    Inst_DSP : NativeFloatingPointDSP
    port map (
      clk0     => CLK_IN,
      ena      => '1',
      clr0     => RESET_IN,
      result   => Result_arr(i),
      chainin  => chainout_arr(i-1),
      ay       => DSPin_arr(i),
      az       => Param_Arr(i+1),
      chainout => chainout_arr(i)
    );
  end generate;
…
end Stratix10Dsp;
```

Plusieurs architecture sont définissables
(la sélection se fait lors de l'instanciation du composant)

Peut instancier d'autres composants de la lib

La description de la fonctionnalité doit s'adapter à
la généricité de l'entité

Library hdl

Pour chaque composant de la lib:

Un fichier contenant une fonction Python décrivant le comportement du composant

```
def evalKernel1D(in_data, dyn_param, generic):

  out = []
  for i in range(0, int((len(in_data) - generic[“KERNEL_SIZE”] )) + 1):
    out.append(np.sum(in_data[i:i + generic[“KERNEL_SIZE”]] * dyn_param))

return out
```

Cette fonction sera utilisé pour les tests unitaires

Library hdl

Un fichier doit indexer tous les fichiers appartenant à la lib

```
{
…

    “Kernel1D": {
        “rtlFilePath": “../libRtlSrc/Kernel1D.vhd",
        “pythonFilePath”: “../libPySrc/Kernel1D.py”,
        “n_input” : 1,
        “has_param”: True,
        “pipeline_latency”: 10,
        “generic": [“Name”:“KERNEL SIZE]
    }

    “MaxPool1D": {
        “rtlFilePath":…
…

}
```

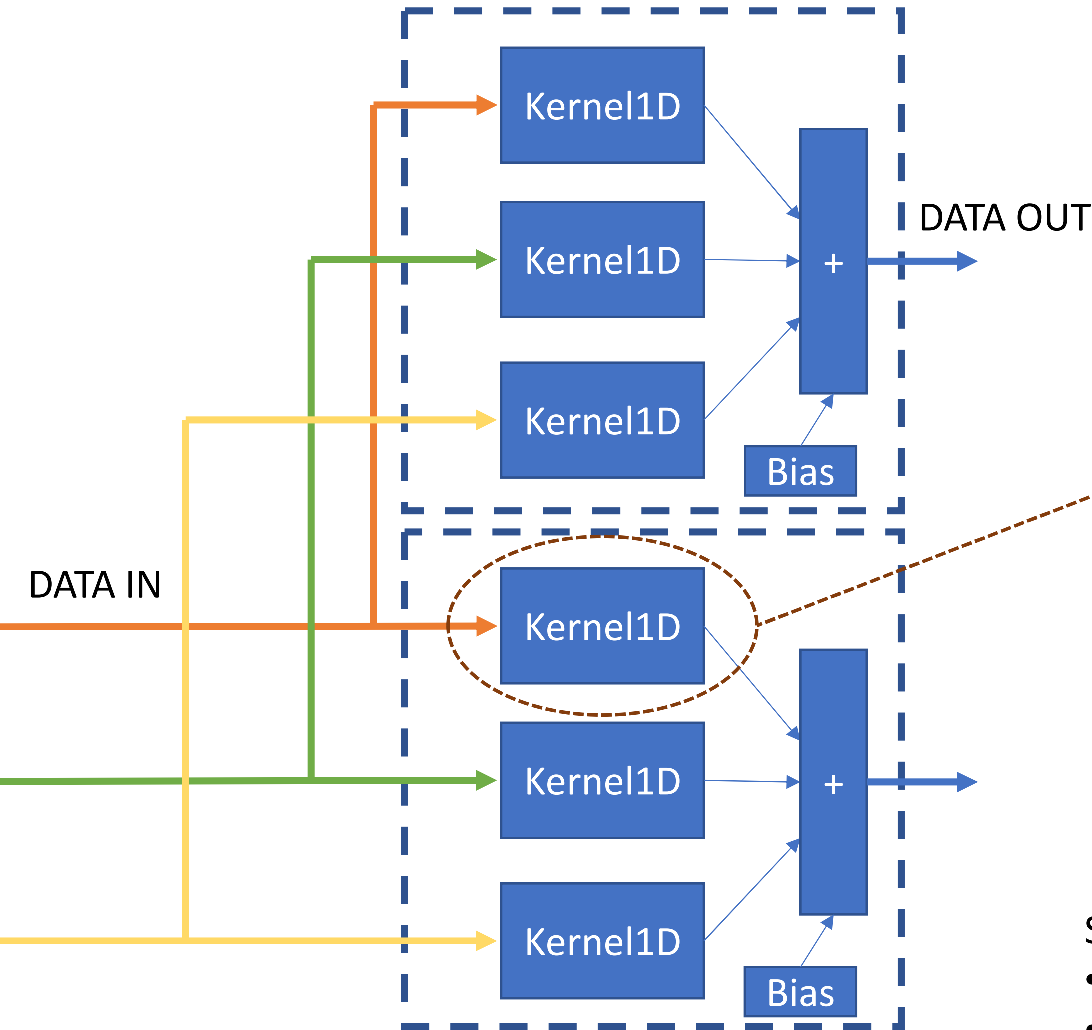
Library hdl

Un fichier contenant les fonctions python “Keras like“ et qui va construire le processing tree en utilisant les composants de la lib

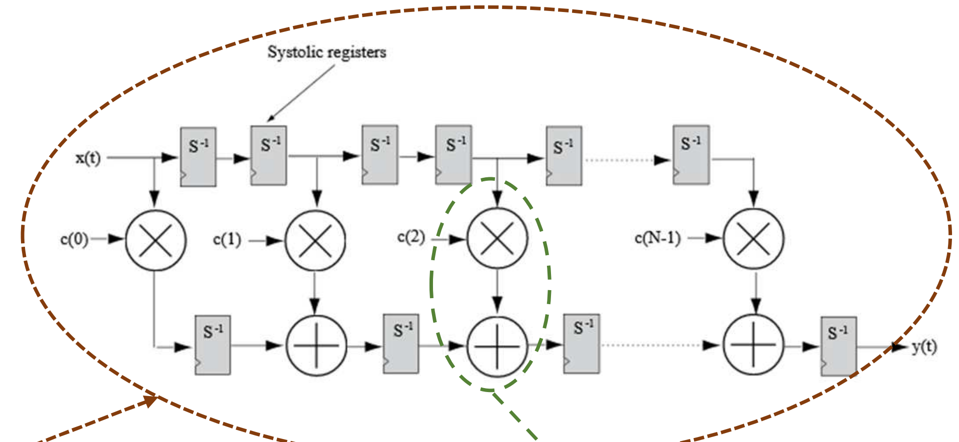
```
def addConv1D(lnConv1DInputId, filters, kernel_size, layer_name = "UNNAMED"):

  lnConv1DOutputId = []

  for nFilter in range(filters):

    ##Kernel instanciation
    lnKerOutputId = []

    #For each input flows of the filter
    for nInputId in lnConv1DInputId:

      #Get the input flow size (the output flow of the input node)
      inputFlowSize = ddNode[nInputId]["OutputFlowSize"]
      outputFlowSize = inputFlowSize-(kernel_size-1)

      nKerId = addNode({"Type":"Kernel1D, "InputFlowSize":inputFlowSize,"OutputFlowSize":
outputFlowSize,"layer_name":layer_name, "kernel_size":kernel_size})
      oProcessingTree.add_node(nKerId)
      oProcessingTree.add_edge(nInputId, nKerId)
      lnKerOutputId.append(nKerId)

    lnAdderOutputId = addReductionAdder(lnKerOutputId, layer_name=layer_name)

    lnConv1DOutputId.append(lnAdderOutputId[0])

  return lnConv1DOutputId
```

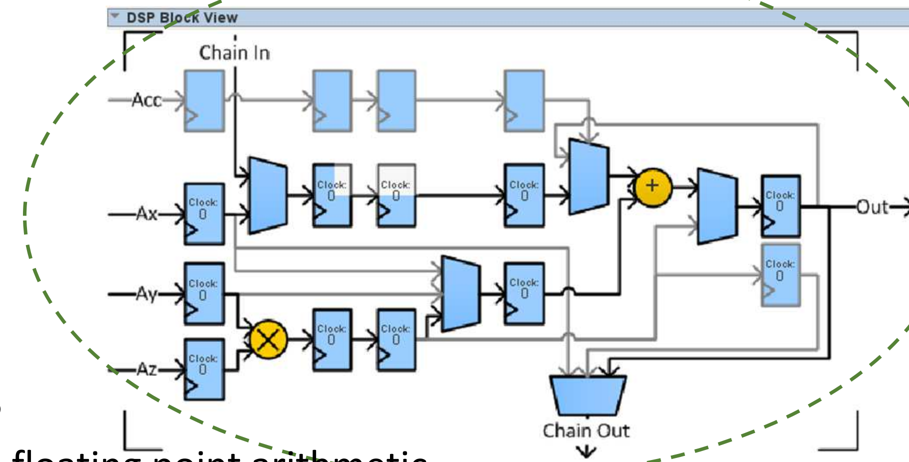
Library HDL - description des composants nécessaires

Conv1D_Layer (N_INPUTS => 3, N_FILTERS => 2)



Kernel 1D module = FIR filter

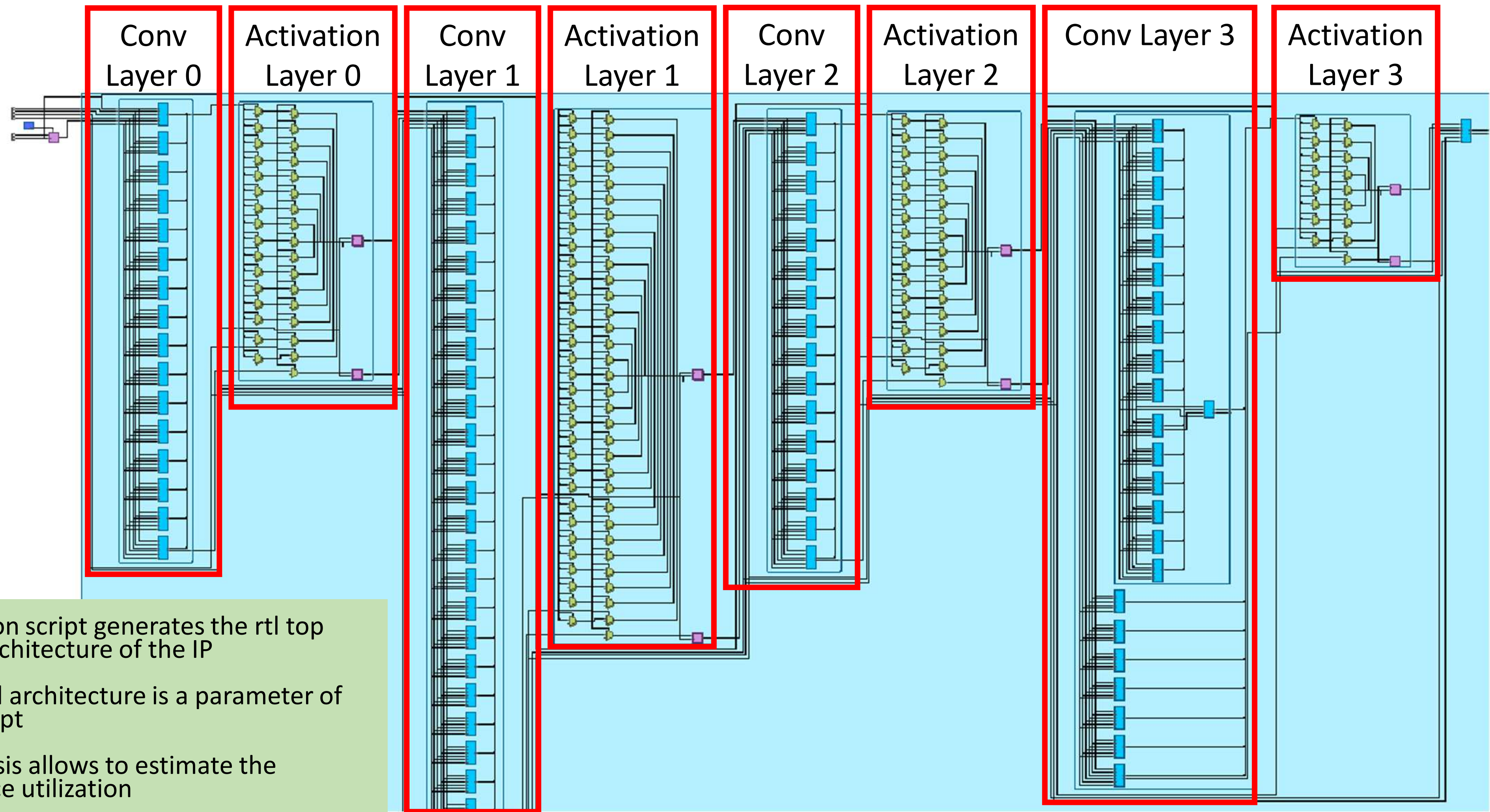




Stratix 10 DSP

- Native 32b floating point arithmetic
- Up to 4000 units available

RTL Generation script



A python script generates the rtl top level architecture of the IP

Desired architecture is a parameter of the script

Synthesis allows to estimate the resource utilization

Fully unrolled architecture (no reusability) -> only small design can fit in FPGA