

École Polytechnique de Montréal
Département de génie informatique et génie logiciel

INF3610
Systèmes Embarqués

Laboratoire 4
Introduction à l'accélération matérielle
par synthèse de haut niveau

Présenté à
Éva Terriault

Soumis le 22 avril 2018

Réponses aux questions:

Question 1:

Dans la situation actuelle (2 boucles imbriquées à 3 itérations/boucle), il serait idéal de dérouler les deux boucles. Quand on a affaire à de petites boucles, on gagne à les dérouler pour les paralléliser plutôt que de les pipeliner. On a pu remarquer justement cet avantage pour le `#pragma HLS unroll` sur `OneTo4`. Le fait qu'il n'y ait pas de lien entre les différentes opérations effectuées par chaque itération justifie encore plus notre choix de `pragma` car cela permet un déroulement percutant des 2 boucles imbriquées à 3 itérations.

Si on avait opté pour le pipelining des deux boucles, on n'aurait pas eu une bonne optimisation comparée au déroulement des boucles. En effet, la convolution n'est pas exécutée de manière régulière, ce qui engendrerait souvent un pipelining vide et réduirait même les performances par rapport au processus normal (sans `pragma`) comme observé dans le `#pragma HLS pipeline` sur `OneTo4` (voir Fig. 1.4).

Question 2:

On remarque que les conditions sont inverses à celles de la question 1. Ici on a affaire à une convolution appelée régulièrement et sans contrôle significatif en boucle. Il est donc préférable de pipeliner la boucle principale car la convolution est exécutée de manière régulière ici et on a une boucle qui couvre une image entière de résolution quand même appréciable. Le fait de la pipeliner permettra de réduire le délai de traitement entre chaque itération (chaque pixel de l'image). On voit l'efficacité du pipeline dans ce genre de situation, un exemple qui l'illustre bien est le `#pragma HLS pipeline` dans `IMG`. Par contre, ici faire un déroulement de la boucle engendrerait juste une optimisation négligeable comparée au pipelining car cela transformerait la boucle principale en plusieurs instances ce qui n'est pas avantageux dans ce cas. On s'en rend compte en regardant les résultats du `#pragma HLS unroll factor = 8` sur `IMG` (voir Fig. 1.3).

Question 3:

3a)

Le tableau ci-dessus illustre les différentes modifications qui ont été apportées au code du filtre sobel pour pouvoir améliorer les performances en termes d'images par seconde. Tout d'abord, nous avons modifié la signature de la fonction `sobel_operator()` pour qu'elle accepte un tableau 2D dans le but d'utiliser la cache personnalisée. L'implémentation de cette cache s'est basée sur celle mentionnée en bas de page dans l'énoncé. Une variable `lineBuffer` a été utilisée pour pouvoir reproduire le comportement de la cache. Comme mentionné dans l'énoncé, nous avons utilisé une cache de quatre lignes qui prenaient deux lignes au départ du traitement, puis qui continuaient à se remplir au fur et à mesure que la variable de sortie `out_pix` se remplissait. Notre première tentative a été d'abord de transférer l'image entière dans la cache, mais cela s'est révélé plus long que la première version matériel. Nous avons donc tenté de réduire le nombre de lignes que la cache pouvait accepter en obtenant une image correcte jusqu'à quatre lignes. Pour éviter les débordements lors de la manipulation de la cache, nous avons utilisé le modulo pour que les index de tableau soient compris dans le bon intervalle. Cette itération a diminué le temps d'exécution de manière significative.

Dans la deuxième itération, nous avons enlevé quatre boucles qui servaient à faire le traitement des bordures. Pour ce faire, une condition if sur les index i et j des deux boucles de parcours a été mise en place aux valeurs extrêmes (0, *IMG_WEIGHT* - 1, *IMG_WIDTH* - 1). La directive (*array partition*) a été utilisée pour le partitionnement du tableau 2D *lineBuffer* pour pouvoir augmenter le nombre de BRAM_18K utilisé afin de ne pas se limiter uniquement à deux entrées et sorties d'un BRAM_18K. Cela est appliquée sur uniquement une dimension de la cache. Cette modification n'a eu que de légers impacts sur la performance du module.

Dans la troisième itération, nous avons utilisé trois directives *pragma* pour réduire afin d'augmenter l'efficacité du module. Les directives *pipeline* et *loop_flatten off* ont été appliquées à la boucle avec index j comme indiqué dans l'énoncé au lieu de la boucle externe. Cette modification a permis de restreindre le nombre de ressources utilisées qui est entraîné par le déroulement automatique des boucles internes à la directive *pipeline*. De ce fait, le déroulement a été appliqué aux deux boucles présentes dans la fonction *soble_opertator()*. La directive *loop_flatten off* a été utilisé pour arrêter la combinaison d'autres boucles avec la boucle courante lors de la directive *pipeline*. Enfin, nous avons modifier les index pour qu'ils soient contenus dans des variables afin d'éviter les calculs redondants lors des accès aux différents tableaux manipulés. Cette modification a eu un effet notable sur le nombre de DSP48E utilisés par le module.

Ensuite dans la quatrième itération, nous avons utilisé une boucle pour automatiser le remplissage de l'union utilisée pour stocker les pixels traités. Nous avons remarqué que l'utilisation de *pipeline* ou d'*unroll* n'a pas d'effet, car cela est automatique avec la directive *pipeline* englobant cette boucle. On a remarqué seulement une différence en termes de nombre de LUT et de FF. Après toutes ces modifications nous sommes arrivés à avoir $1/0.042338 = 23.6194435$ images par seconde.

Tableau I: Utilisation des ressources matérielles et performances obtenues par le module Sobel

Exécutions du filtre	Temps d'exécution réel de l'affichage d'une image (secondes)	Temps d'exécution estimé par HLS (secondes)	BRAM_18K	LUT	FF	DSP48E
1. Exécution logicielle	2.941236	4.77653249	0	0	0	0
2. Première exécution matérielle	16.892871	4.77653249	4	2493	3121	2

<p>3. Modification de la signature de <code>sobel_operator</code> pour utiliser la cache personnalisé de taille 4x1920 avec la suggestion d'utiliser uniquement deux lignes en cache et une troisième en parallèle. Éliminer de ce fait la fonction <i>getVal()</i></p>	0.955695	0.95442988	8	3700	2751	2
<p>4. Traiter les bordures grâce une condition if sur les index i et j. Éliminer de ce fait quatre boucle for qui s'occupait du traitement des bordures. Et ajout de partitionnement sur la variable <i>lineBuffer</i></p>	0.766004	0.76746020	8	2278	2671	2

5. Ajout de pipeline et flatten off dans la boucle intérieur pour le traitement avec sobel_operator cache de taille 4x20. Enlever calcul d'index de cache et les stocker dans des variables appropriées.	0.042339	0.04175420	8	2646	3603	0
6. Création d'une boucle pour remplir le contenu de l'union (variable <i>tempPixel</i>) avec directive unroll.	0.042338	0.04175420	8	2618	3539	0

3b)

1- L'exécution logicielle produit un temps d'exécution plus petit que celui estimé par HLS. On peut expliquer cela par l'utilisation de la fonction *doSobelSW* au lieu de *doSobel()*. Il n'y a pas d'utilisation de matériel donc on devrait s'attendre à un résultat différent et plus rapide.

2-La première version matérielle donne un temps nettement différent de celui estimé étant donné que le code n'était pas réellement optimisé par des directives, cache ou autre méthode d'amélioration.

3-L'implémentation de la cache dans cette itération a contribué à rétrécir la différence entre le temps estimé et le d'exécution réel (différence notée dans les millièmes de seconde). On note aussi que la disparition de la fonction *getVal()* a nécessairement diminuer le temps d'exécution.

La cache diminue le délai entre les lectures et les écritures vu qu'on n'attend pas toutes les données de l'image pour faire les traitements.

4-Dans cette itération, nous pensons que le rapprochement entre les temps d'exécution s'expliquent principalement par le partitionnement du tableau *lineBuffer* qui permet de contourner les limitations imposées par les BRAM.

5-L'ajout des directives *pipeline* et *loop_flatten off* a permis d'optimiser l'utilisation des ressources de la carte, mais la différence est encore au niveau des millièmes de seconde. Il n'y a pas réellement d'amélioration en termes de rapprochement.

6-Cette itération ne comprend pas vraiment de différence notable par rapport à la dernière. Cependant, la différence en temps entre les deux itérations n'est pas significative, car nous avons observé quand même en moyenne un temps d'exécution matériel de 0.042339s. Il y a quand même une petite différence en termes de LUT et FF.

Question 4:

Il est à noter que chaque architecture (gros grains et grains fins) a ses avantages et inconvénients. Il faut donc les choisir judicieusement en fonction de la situation ou des contraintes auxquelles on fait face. L'architecture gros grains est plus adaptée lorsqu'on manipule des grosses tâches, elle permet ainsi de réduire la parallélisation de ces dernières et ainsi de moins les fractionner. Vu que dans ce lab, nous avons affaire à des tâches qui ne sont pas si grandes que ça, l'architecture à grains fins est celle qui pourra nous apporter les meilleures performances en comparaison avec celle à gros grains car elle permettra d'effectuer une parallélisation des tâches, parallélisation qui entraînerait une grande amélioration des performances. Certes, l'architecture à grains fixes permet une amélioration des performances mais peut malheureusement affaiblir celle-ci dans le cas où il y a un nombre élevé de tâches.

Question 5:

Le `#pragma HLS pipeline` réalise simultanément les instructions d'itérations différentes dans une boucle. Il permet donc en gros d'optimiser le délai de traitement entre chaque instruction ce qui réduit la latence tandis que le pipelining logiciel a une approche qui réside dans le fait de réduire au maximum les éventuelles liaisons entre les variables. Les deux méthodes permettent d'optimiser à leur façon les performances. Néanmoins, pour le pipelining logiciel, il permet certes de réduire les itérations d'instruction mais dans le cas d'une modification, il va falloir faire également des ajustements complexes pour adapter le code quand on opte pour le pipelining logiciel, tandis que le `#pragma HLS pipeline` pallie à ce problème de maintenance du code. Vu qu'il exécute juste les instructions en parallèle et de réduire la latence des instructions, il ne nécessite pas une réorganisation lors d'un changement dans le code comme le pipeline logiciel et peut être facilement désactivé en cas de besoin.

Questions supplémentaires:

a-/

Pour un dernier lab et surtout un lab de 15%, ce lab valait bien ce pourcentage. Nous l'avons trouvé plus long que le lab 2. En effet, il y avait beaucoup de choses à faire tel que inclure le filtre à un projet vivado permettant la sortie d'images sur le port HDMI du Zedboard, estimer les ressources matérielles et la performance du design ou encore valider un système avec une co-simulation logicielle/matérielle pour ne citer que certains. Il est indéniable que nous avons appris énormément de choses grâce à ce laboratoire. Néanmoins, nous avons des difficultés à exécuter et achever la partie SDK du lab et nous avons également redoublé d'efforts pour finir la dernière partie (optimisation) du laboratoire. Il pourrait être redonné la session prochaine mais en apportant un peu plus d'explication pour faciliter la démarche aux étudiants. Il faut surtout régler le problème de la mise à jour du SDK qui fait perdre énormément de temps pour rien.


b-/

Globalement, nous avons passé 40-45h environ sur ce laboratoire.

Annexe

Partie 1:


Étape 1

 **Loop**

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- IMG	16588800	16588800	8	-	-	2073600	no
+ OneTo4	4	4	1	-	-	4	no

Fig. 1.1: Informations sur l'exécution des deux boucles sans directive

Étape 2

 **Loop**

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- IMG	6220800	6220800	3	-	-	2073600	no

Fig. 1.2: Informations sur l'exécution de la boucle IMG
avec la directive unroll

Étape 3

i)

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- IMG	12960000	12960000	50	-	-	259200	no
+ OneTo4	4	4	1	-	-	4	no
+ OneTo4	4	4	1	-	-	4	no
+ OneTo4	4	4	1	-	-	4	no
+ OneTo4	4	4	1	-	-	4	no
+ OneTo4	4	4	1	-	-	4	no
+ OneTo4	4	4	1	-	-	4	no
+ OneTo4	4	4	1	-	-	4	no
+ OneTo4	4	4	1	-	-	4	no

Fig. 1.3: Informations sur l'exécution de la boucle IMG

avec la directive *unroll factor* = 8

ii)

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- IMG_OneTo4	8294402	8294402	4	1	1	8294400	yes

Fig. 1.4: Informations sur l'exécution de la boucle *OneTo4*

avec la directive *pipeline*

iii)

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- IMG	2073601	2073601	3	1	1	2073600	yes

Fig. 1.5: Informations sur l'exécution de la boucle *IMG* avec la directive *pipeline*

Partie 2:

Première itération logicielle et matérielle.

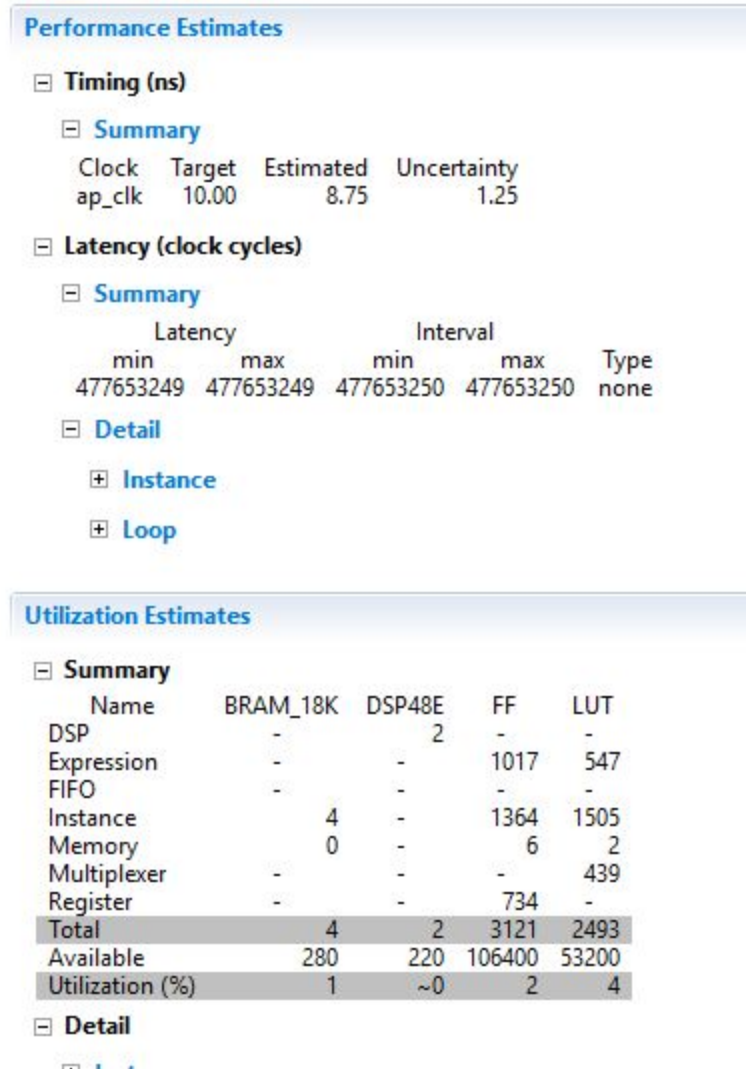


Fig. 2.1: Première itération logicielle et matérielle

2. Deuxième itération de la version matérielle

Performance Estimates

[-] Timing (ns)

[-] Summary

Clock

Target

Estimated

Uncertainty

ap_clk

10.00

8.75

1.25

[-] Latency (clock cycles)

[-] Summary

Latency

Interval

min

max

min

max

Type

95442988

95442988

95442989

95442989

none

[-] Detail

[+] Instance

[+] Loop

Utilization Estimates

[-] Summary

Name

BRAM_18K

DSP48E

FF

LUT

DSP

-

2

-

-

Expression

-

-

1700

850

FIFO

-

-

-

-

Instance

4

-

1136

1328

Memory

4

-

6

2

Multiplexer

-

-

-

571

Register

-

-

858

-

Total

8

2

3700

2751

Available

280

220

106400

53200

Utilization (%)

2

~0

3

5

Fig. 2.2: Deuxième itération de la version matérielle

3. Troisième itération de la version matérielle

Performance Estimates

[-] Timing (ns)

[-] Summary

Clock	Target	Estimated	Uncertainty	
ap_clk	10.00	8.75	1.25	

[-] Latency (clock cycles)

[-] Summary

Latency		Interval		Type
min	max	min	max	
6243620	76746020	6243621	76746021	none

[-] Detail

[+] Instance

[+] Loop

Utilization Estimates

[-] Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	2	-	-
Expression	-	-	894	524
FIFO	-	-	-	-
Instance	4	-	1221	1349
Memory	4	-	6	2
Multiplexer	-	-	-	403
Register	-	-	550	-
Total	8	2	2671	2278
Available	280	220	106400	53200
Utilization (%)	2	~0	2	4

[-] Detail

Fig. 2.3: Troisième itération de la version matérielle

4. Quatrième itération de la version matérielle

Performance Estimates

[-] Timing (ns)

[-] Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.75	1.25

[-] Latency (clock cycles)

[-] Summary

Latency		Interval		Type
min	max	min	max	
4175420	4175420	4175421	4175421	none

[-] Detail

+ Instance

+ Loop

Utilization Estimates

[-] Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	1088	643
FIFO	-	-	-	-
Instance	4	-	1816	1496
Memory	4	-	0	0
Multiplexer	-	-	-	411
Register	-	-	699	96
Total	8	0	3603	2646
Available	280	220	106400	53200
Utilization (%)	2	0	3	4

Fig. 2.4: Quatrième itération de la version matérielle

5. Cinquième itération de la version matérielle

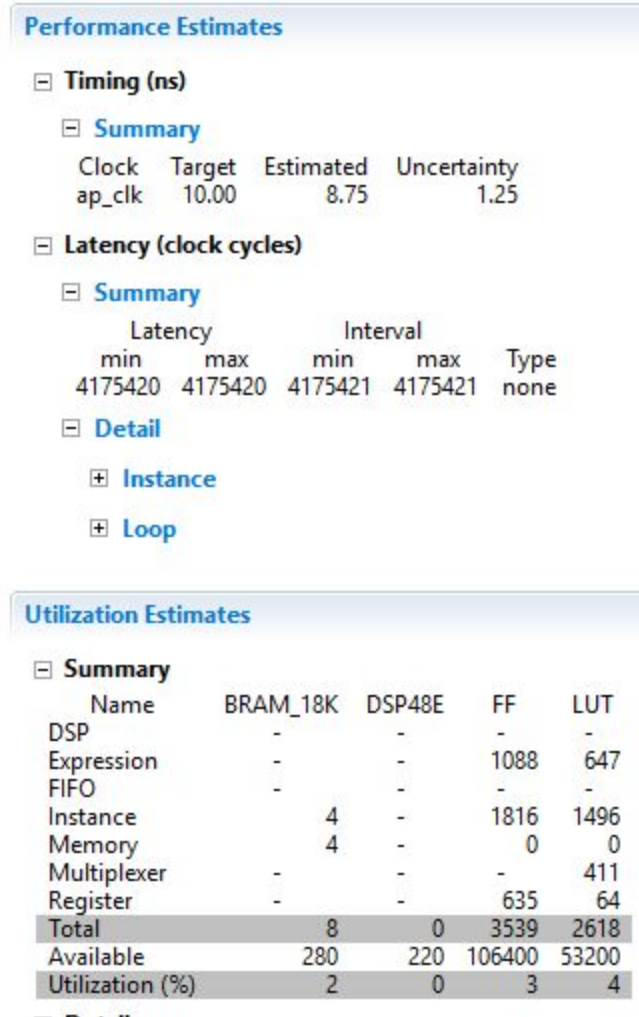


Fig. 2.5: Cinquième itération de la version matérielle