Département de génie informatique et génie logiciel

INF3610

Systèmes embarqués

Laboratoire 4

Introduction à l'accélération matérielle par synthèse de haut niveau

Soumis par

Félix Boulet, #1788287

Giuseppe La Barbera, #1799919

le 6 décembre 2017

Partie 1

Note de la latence, etc. pour « questions initiales »

Loop

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

Avec #pragma HLS unroll dans OneTo4 :

Loop

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

Avec #pragma HLS unroll factor=8 dans IMG :

Loop

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

Avec #pragma HLS pipeline dans OneTo4 :

Loop

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

Avec #pragma HLS pipeline dans IMG :

Loop

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

Question 1

Il faudrait dérouler les deux boucles. Il est utile de dérouler une petite boucle imbriquée de ce genre, qui ne fait que rendre 2D une certaine opération (qu'on aurait aussi bien pu indexer en 1D), car celle-ci a avantage à être parallélisée. Il n'y a d'ailleurs pas d'interdépendance entre les différents calculs, puisque chaque itération de la boucle affecte seulement un unique résultat

dans la matrice 3x3. On peut observer l'accélération qu'amène cette méthode dans les résultats du *#pragma HLS unroll* sur OneTo4, cela rendant possible un déroulement efficace des boucles imbriquées. Il ne serait pas bénéfique de plutôt pipeliner l'opération, puisque l'on s'attend à un mode d'opération irrégulier. On se retrouverait donc souvent avec un pipeline vide, ce qui résulte en un avantage minime par rapport à une implémentation sans déroulement ni pipeline (on peut l'observer, supposant un pipeline plein toutefois, dans les résultats du *#pragma HLS pipeline* sur OneTo4).

Question 2

Il faudrait pipeliner la boucle principale. En effet, pipeliner une opération régulière de grande envergure (comme le passage sur chaque pixel d'une image de grande taille) permet de réduire le délai de traitement entre chaque pixel à un seul cycle, une fois le premier traité. Cela amène une grande accélération, comme on peut le voir dans les résultats du *#pragma HLS pipeline* dans IMG (qui pipeline implicitement la boucle interne en plus de la boucle principale). En revanche, dérouler la boucle principale aurait un impact minime sur l'accélération du système, puisque la boucle interne s'en retrouverait simplement multipliée en de nombreuses instances. On peut observer ce comportement dans les résultats du *#pragma HLS unroll factor=8* sur IMG.

Question 3a

Tableau

Modification	Temps réel par image (sec)	Temps estimé par image (sec)	BRAM	DSP	FF	LUT
1. Version logicielle	3,1292	4,9145	-	-	-	-
2. Première version matérielle	17,146	4,9145	4	2	2659	2687
3. Changement pour que *sobel_operator* utilise la 3e signature (tableau 2D) et retrait de l'union pour des « << » et « | ».	3,1698	1,2650	4	2	2282	2020
4. Ajout d'une cache à 4 lignes (*linebuffer*) non optimisée (sans partitionnement)	1,7355	1,0784	8	2	2943	2468
5. Utilisation de *#pragma HLS unroll* dans *sobel_operator* (les deux sous-boucles) et *#pragma HLS ARRAY_PARTITION variable=lineBuffer complete dim=1* pour partitionner la cache	0,9029	0,2490	8	0	3325	2594
6. Utilisation de #pragma HLS pipeline dans la boucle principale passant sur la dimension *j* (donc qui parcourt une ligne de l'image). Cela a pour effet de faire	0,2276	0,0418	8	0	3989	3049

	implicitement le *unroll* de *sobel_operator* qui avait été implémenté auparavant.						
7.	**Optimisation des transactions mémoire pour permettre des rafales (*burst*) en lecture/écriture. Pour parvenir à cela, les conditions ont été retravaillées et le chargement de la cache est devenu continu (2 lignes chargées initialement, la 3e se charge pendant le traitement de la ligne 0 qui sera toute noire, et le traitement commence au chargement de la 4e ligne, pour une exécution continue).**	0,0423	0,0418	8	0	3496	2586

Explications

1 → 2 : La version logicielle s'exécute plus rapidement que la version matérielle initiale. Même en meilleur cas, cette dernière resterait plus lente que l'exécution logicielle (4,9145 secondes par image, en théorie). Cela veut dire que la version matérielle initiale n'est réellement pas optimisée pour être implémentée sur un FPGA.

2 → 3 : Le fait de passer à un tableau 2D pour l'exécution de l'opérateur Sobel permet de réduire la dépendance à une fonction comme *getVal*, qui nécessite des calculs coûteux en temps pour aller chercher les indices des pixels nécessaires au calcul. Le fait de passer d'une union à des shifts et opérations logiques booléennes n'a qu'un effet minime sur la performance, mais permet d'obtenir un comportement précis plutôt qu'une version équivalente créée par le synthétiseur. Cela évite également la conversion de types. Accélération réelle : ~ 5,4x. Accélération théorique : ~ 3.9x.

3 → 4 : Ajouter une mémoire cache permet de lire des données et de les emmagasiner à même le module, évitant ainsi de faire de nombreuses transactions mémoire avec la DDR (qui sont très exigeantes en termes de temps). Définir un tableau à même une fonction dans HLS l'implémente comme une cache en BRAM à l'intérieur du module. Bien sûr, à ce point-ci, la cache n'est pas optimisée et la fonction *sobel_operator* ne peut en lire qu'une seule donnée à la fois, donc le facteur d'accélération reste sensiblement restreint (moins de 2x en termes de performance réelle). Accélération réelle : ~ 1,8x. Accélération théorique : ~ 1,2x.

4 → 5 : Dérouler l'opérateur Sobel pour qu'il puisse effectuer (en théorie) l'ensemble des opérations nécessaire pour transformer un pixel (donc sur 9 pixels environnants) en un seul cycle d'horloge donne un bon gain de performance. Il a été nécessaire d'également partitionner la cache pour permettre au module de travailler de la sorte (pouvoir lire plus d'une donnée en cache à la fois). En théorie, puisque la cache a été divisée en 4 lignes distinctes, 3 données pourraient être lues et traitées en même temps (car 3 lignes de cache sont utilisées lors du calcul). Accélération réelle : ~ 1,9x. Accélération théorique : ~ 4,3x.

5 → 6 : Pipeliner le traitement d'une ligne complète de l'image permet d'accélérer énormément le traitement. En effet, en faisant cela, on permet au système de traiter en théorie 1 pixel par cycle d'horloge (une fois le premier passé). Toutefois, dans notre cas, nous ne sommes pas parvenus à atteindre un *initiation interval (II)* de 1 sur le pipeline, mais seulement de 2. Cela veut donc dire qu'en théorie, chaque pixel prendra 2 cycles d'horloge pour être traité. Cela est dû au fait que la mémoire cache n'est pas structurée de sorte à permettre une telle rapidité d'exécution (*sobel_operator* se fait en plus d'un cycle puisqu'il ne peut pas lire 9 valeurs à la fois). En revanche, on obtient tout de même la vitesse requise (20+ FPS), en théorie toutefois (avoir une *II* de 1 aurait permis 40+ FPS). Il reste donc à optimiser les accès mémoire pour obtenir la même performance en réel. Accélération réelle : ~ 4,0x. Accélération théorique : ~ 6,0x.

6 → 7 : En éliminant des conditions (*if*) inutiles, tel qu'expliqué dans le tableau, on obtient des lectures et écritures en rafales lors de la lecture des lignes de cache, lors de leur passage dans *sobel_operator* et lors de l'écriture des résultats. Cette accélération mémoire permet d'obtenir la performance désirée. Accélération réelle : ~ 5,4x. Accélération théorique : Aucune.

Question 3b :

1. Les valeurs réelles et théoriques sont différentes. En effet, il ne devrait y avoir aucun lien entre celles-ci puisque le filtre est exécuté en logiciel et que l'estimation est faite pour le matériel.
2. Les valeurs réelles et théoriques sont extrêmement différentes. Cela est dû au fait que les accès mémoire à la DDR prennent énormément de temps (12,23 secondes de plus par rapport à l'estimation théorique). Cette version de l'implémentation matérielle n'est pas du tout optimisée, donc elle provoque des lectures de 1 pixel à la fois, et des écritures de 1 pixel à la fois.
3. Les valeurs diffèrent, mais beaucoup moins qu'à la précédente (un peu moins de 2 secondes). Cela est dû au fait que la transformation de *sobel_operator* pour que la fonction utilise un tableau 2D permet l'utilisation de *bursts* dans certaines circonstances : les accès mémoire sont donc plus uniformes et moins de temps est perdu à lire/écrire des données.
4. Les valeurs diffèrent encore, mais moins qu'à l'itération précédente (~ 0,7 seconde). Cela est dû au fait qu'on charge les pixels dans une cache pour réduire les accès mémoire à même le module (*sobel_operator* lit ses valeurs à partir d'une BRAM plutôt que d'aller les chercher en mémoire principale). On note une amélioration notable de la performance réelle (on passe à plus de 0.5 FPS).
5. Ici, les valeurs diffèrent du même temps qu'à l'itération précédente (~ 0,7 seconde). Bien qu'on ait réussi à améliorer la performance théorique en déroulant la boucle interne de *sobel_operator* et en partitionnant la cache, cela ne réduit pas la latence produite par les lectures en mémoire principale. La performance réelle ne fait donc que s'améliorer de la même manière que la performance théorique.
6. Les valeurs diffèrent beaucoup moins qu'à l'itération précédente (~ 0,2 seconde). Le fait de pipeliner le traitement sur une ligne de l'image a permis d'activer certaines lectures en rafales (notamment celle du chargement d'une nouvelle ligne de cache), puisque le

synthétiseur est en mesure de détecter qu'on traite une ligne entière de l'image à la fois. On note donc une amélioration très marquée de la performance réelle (près de 5 FPS).

7. Cette dernière itération est la seule qui montre une performance similaire entre la valeur réelle et la valeur théorique (~ 0,0005 seconde de différence). Cela est dû au fait que nous ayons réussi à activer les lectures et écritures en rafales pour toutes les opérations nécessitant un accès à la mémoire. La boucle de traitement est structurée de manière à ce que la ligne de cache puisse être lue avec une latence égale ou inférieure au pipeline de traitement (du temps qu'un pixel entre dans le pipeline jusqu'à ce qu'il en sorte) en raison de son mode d'action continu et avec peu de branchements (conditions).

Question 4

Travailler avec la synthèse de haut niveau a l'avantage de permettre de tester le code en logiciel rapidement pour observer son comportement avant de l'implémenter sur une carte FPGA. Lorsqu'on travaille directement en VHDL/Verilog, les simulations que l'on peut effectuer sont excessivement lentes et servent plutôt à vérifier l'état des signaux ou à valider le comportement avec un *testbench*. Aussi, pour observer le comportement de notre algorithme avec des périphériques ou autres artifices (le lecteur de carte SD dans notre cas, par exemple), il faut synthétiser et implémenter le code VHDL/Verilog sur la carte FPGA, ce qui prend beaucoup de temps par rapport à une exécution logicielle sur le processeur ARM du Zynq, par exemple. Bref, même si on est un ninja du VHDL/Verilog, qui code aussi rapidement qu'en C++, on perd quand même du temps en simulation et en implémentation par rapport à l'utilisation de HLS.

Questions supplémentaires

a. Nous l'avons bien aimé. Il nous permet d'apprendre l'ensemble du flot de conception sur une carte Zynq, de la création du design avec le PS7 jusqu'à l'implémentation HLS, en passant par l'utilisation du SDK pour développer des programmes qui vont rouler sur le ARM. Un point qui serait à améliorer serait le tutoriel pour la création du design sur Vivado : ce n'était pas toujours clair quelles étaient les horloges à changer, et il y avait certaines inconstances dans les images placées dans le tutoriel (par exemple, un module qui se nommait 76M_RST…truc devient à moment donné 100M_RST… truc, ce qui peut porter à confusion puisqu'on nous demande de comparer notre design avec cette dernière image). Il serait aussi utile de faire en sorte que le driver HDMI fonctionne quand on fait simplement un nouveau lancement de notre logiciel à partir du SDK, si possible (sans devoir relancer le bitstream et avoir à reset la carte une fois sur deux). Le laboratoire devrait définitivement être redonné la session prochaine, l'effet final est impressionnant et on est fier de le montrer à nos amis.

b. Pas tant que ça. Les deux périodes plus 9-10 heures en dehors du lab, pour atteindre la performance donnant le plus de points (une demi-heure à la maison à changer du code, 2h à tenter de mettre ce code sur la carte FPGA sans que les portes de l'enfer ne s'ouvrent sous mes pieds, 5-6 heures après les labs, et 1h30 sur le rapport).

Annexe

Résultats pour l'implémentation de première partie (logicielle) :

Loop

Loop Name	Latency min	Latency max	Iteration Latency	Initiation Interval achieved	Initiation Interval target	Trip Count	Pipelined
- Loop 1	18664560	491445360	17282 ~ 455042	-	-	1080	no
+ Loop 1.1	17280	455040	9 ~ 237	-	-	1920	no
++ Loop 1.1.1	222	222	74	-	-	3	no
+++ Loop 1.1.1.1	72	72	24	-	-	3	no
++ Loop 1.1.2	4	4	1	-	-	4	no

Performance : ~ 3.1292 secondes/frame → ~ 0.32 FPS

Résultats pour première exécution matérielle :

Temps : ~ 17.1455 secondes/frame

Selon Vivado HLS estimation : 4.91445362 secondes / frame

DSP = 2 (~0%), BRAM = 4 (1%), FF=2659 (2%), LUT=2687 (5%)

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.75	1.25

Latency (clock cycles)

Summary

Latency min	Latency max	Interval min	Interval max	Type
18664561	491445361	18664562	491445362	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	2	-	-
Expression	-	-	807	881
FIFO	-	-	-	-
Instance	4	-	1364	1505
Memory	0	-	6	2
Multiplexer	-	-	-	299
Register	-	-	482	-
Total	4	2	2659	2687
Available	280	220	106400	53200
Utilization (%)	1	~0	2	5

Après utilisation de la 3^{e} signature de fonction et enlevé l'union (utilisation de shifts)

Performance : 3.1698 secondes / frame = 0.315477 FPS

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
4154761	126497161	4154762	126497162	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	2	-	-
Expression	-	-	739	424
FIFO	-	-	-	-
Instance	4	-	1136	1328
Memory	0	-	6	2
Multiplexer	-	-	-	266
Register	-	-	401	-
Total	4	2	2282	2020
Available	280	220	106400	53200
Utilization (%)	1	~0	2	3

Avec cache à 4 lignes, de base :

Performance : ~ 1.7355 secondes / frame = 0.57617 FPS

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
18676109	107840909	18676110	107840910	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	2	-	-
Expression	-	-	1185	718
FIFO	-	-	-	-
Instance	4	-	1136	1328
Memory	4	-	6	2
Multiplexer	-	-	-	420
Register	-	-	616	-
Total	8	2	2943	2468
Available	280	220	106400	53200
Utilization (%)	2	~0	2	4

Avec unroll dans sobel_operator et ARRAY_PARTITION (complete dim=1) de la cache :

Performance : 0.9029 seconde / frame = 1.1075 FPS

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
18676109	24896909	18676110	24896910	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	1116	665
FIFO	-	-	-	-
Instance	4	-	1646	1454
Memory	4	-	0	0
Multiplexer	-	-	-	475
Register	-	-	563	-
Total	8	0	3325	2594
Available	280	220	106400	53200
Utilization (%)	2	0	3	4

Avec pipeline dans la boucle passant sur la ligne de cache :

Performance : ~ 0.22761 seconde/frame = 4.39348 FPS

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.15	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
4180349	4180349	4180350	4180350	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- LoadCache	11547	11547	3849	-	-	3	no
+ LoadLine	3840	3840	2	-	-	1920	no
- Loop 2	4168800	4168800	3860	-	-	1080	no
+ Loop 2.1	3857	3857	20	2	1	1920	yes

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	1116	699
FIFO	-	-	-	-
Instance	4	-	1888	1736
Memory	4	-	0	0
Multiplexer	-	-	-	422
Register	-	-	985	192
Total	8	0	3989	3049
Available	280	220	106400	53200
Utilization (%)	2	0	3	5

Avec optimisation des transactions mémoire pour permettre des burst en lecture/écriture :

Performance : ~ 0.042339 seconde / frame = 23.6189 FPS ☺

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
4175420	4175420	4175421	4175421	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	1088	638
FIFO	-	-	-	-
Instance	4	-	1816	1496
Memory	4	-	0	0
Multiplexer	-	-	-	420
Register	-	-	592	32
Total	8	0	3496	2586
Available	280	220	106400	53200
Utilization (%)	2	0	3	4