

4η Εργαστηριακή Άσκηση για το μάθημα

Σχεδιασμός Ενσωματωμένων Συστημάτων

Ομάδα 29

Μέλη : Αριστοτέλης Γρίβας

Σάββας Λεβεντικίδης

Εργαστηριακή Άσκηση :

Ο σκοπός της άσκησης είναι να γίνει μελέτη γύρω από τον προγραμματισμό FPGA με High Level Synthesis (HLS). Πρόκειται για optimization και επιτάχυνση C κώδικα για να τρέξει τελικώς στο hardware του Xilinx Zybo FPGA. Η εφαρμογή που θα μελετηθεί για επιτάχυνση θα σχετίζεται με νευρωνικά δίκτυα και ειδικότερα τα Generative Adversarial Networks (GANs). Η εφαρμογή αφορά την ανακατασκευή εικόνων. Συγκεκριμένα, δέχεται ως input το πάνω μισό των εικόνων (που απεικονίζουν αριθμούς στην περίπτωση μας), και καλεί το νευρωνικό για να κάνει generate/predict το υπόλοιπο μισό της εικόνας. Ο κώδικας που εκτελεί την εφαρμογή GAN δίνεται έτοιμος στο εργαστηριακό υλικό, και εμείς καλούμαστε να δούμε τη συμπεριφορά του σε SW και HW όσον αφορά σε κατανάλωση πόρων και σε απόδοση αποτελεσμάτων, και στη συνέχεια να τον βελτιστοποιήσουμε.

(Για τα ερωτήματα που ακολουθούν παραθέτουμε τους κώδικες των network.cpp, tanh.h και network.h στους αντίστοιχους φακέλους).

Άσκηση 1. Performance and resources measurement

A)

Αφού εισάγουμε τα source και header files, που δίνονται από το εργαστηριακό υλικό, στο SDSoC, ορίζουμε τη συνάρτηση `forward_propagation` ως HW function και καταγράφουμε το report με τα estimated resources, μέσω της επιλογής “Estimate Performance”.

Οι απαιτούμενοι κύκλοι για την εκτέλεση της εφαρμογής καθώς και οι πόροι που χρειάστηκαν για την εκτέλεση δίνονται παρακάτω :

Estimations:

Details

Performance estimates for 'forward_propagation in main.cp ...

HW accelerated (Estimated cyc	683780
-------------------------------	--------

Resource utilization estimates for HW functions

Resource	Used	Total	% Utilization
DSP	3	80	3,75
BRAM	16	60	26,67
LUT	1760	17600	10
FF	892	35200	2,53

Επιπλέον, καταγράφουμε τα αποτελέσματα του HLS report για κάθε loop όπως φαίνεται παρακάτω :

Loop performance Estimation:

Detail

Instance

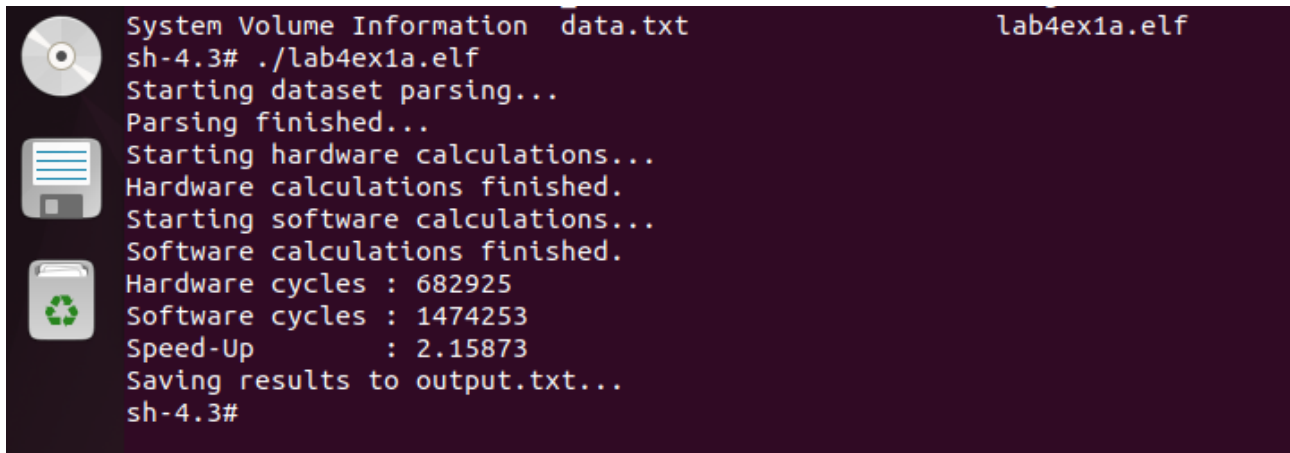
N/A

Loop

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- read_input	1568	1568	4	-	-	392	no
- layer_1	36456	36456	93	-	-	392	no
+ layer_1.1	90	90	3	-	-	30	no
- layer_1_act	60	60	2	-	-	30	no
- layer_2	4600	4600	92	-	-	50	no
+ layer_2.1	90	90	3	-	-	30	no
- layer_3	61936	61936	158	-	-	392	no
+ layer_3.1	150	150	3	-	-	50	no

B)

Στη συνέχεια, απενεργοποιούμε το “estimate performance” από τα options, και ενεργοποιούμε τις λειτουργίες “generate bitstream” και “generate SD card image”. Έτσι, παράγουμε το bitstream καθώς και τα αρχεία τα οποία θα χρειαστούν για να τρέξει η εφαρμογή στο zybo. Αφού περάσουμε τα αρχεία στην sd card (δεν ξεχνάμε το data.txt file που περιέχει το input dataset), κάνουμε τη σύνδεση με το zybo μέσω του minicom, κάνουμε boot το board και εκτελούμε το αρχείο .elf της κάρτας sd (βρίσκεται στο /mnt). Τα αποτελέσματα που παίρνουμε είναι τα ακόλουθα :



```
System Volume Information  data.txt  lab4ex1a.elf
sh-4.3# ./lab4ex1a.elf
Starting dataset parsing...
Parsing finished...
Starting hardware calculations...
Hardware calculations finished.
Starting software calculations...
Software calculations finished.
Hardware cycles : 682925
Software cycles : 1474253
Speed-Up       : 2.15873
Saving results to output.txt...
sh-4.3#
```

Όπως βλέπουμε, οι κύκλοι που εκτελούνται στο Hardware είναι αρκετά κοντά στην εκτίμηση που έγινε με το SDSoc, ενώ το speedup σε σχέση με την SW εκτέλεση στον ARM είναι ίση με 2.15873.

Γ)

Στο σημείο αυτό, θα δοκιμάσουμε διάφορα optimizations, με σκοπό να μειώσουμε τους απαιτούμενους κύκλους της εφαρμογής, πετυχαίνοντας γρηγορότερη εκτέλεση. Έτσι δοκιμάζουμε διάφορα HLS pragmas και παρατηρούμε τα αντίστοιχα performance estimations.

- Αρχίζουμε, με **το #pragma HLS pipeline II= 1** , το οποίο επιχειρεί να πετύχει το βέλτιστο δυνατό pipeline, αρχίζοντας τη δρομολόγηση της νέας επανάληψης ενός βρόχου στον επόμενο κύκλο ρολογιού. Προφανώς για τα μονά loops, η δήλωση του directive γίνεται αμέσως μετά το for statement. Όσον αφορά,ωστόσο, στα διπλά loops, πειραματιστήκαμε τόσο στα εσωτερικά όσο και στα εξωτερικά loops, βάζοντας τα directives σε όλες τις πιθανές θέσεις. Αφού τρέξαμε τα estimations, καταλήξαμε στο συμπέρασμα πως οι βέλτιστες θέσεις για τη δήλωση του directive είναι αμέσως μετά το for statement του εξωτερικού loop για όλα τα διπλά loops (layer1,layer2,layer3). (Προφανώς, δοκιμάσαμε να βάλουμε και στα εσωτερικά και στα εξωτερικά loops ταυτόχρονα, χωρίς να έχουμε κάποια βελτίωση στην απόδοση)

Τα αποτελέσματα που πήραμε είναι τα εξής :

Estimations for pipeline II = 1:

Details

Performance estimates for 'forward_propagation in main.cp ...

HW accelerated (Estimated cyc	86183
-------------------------------	-------

Resource utilization estimates for HW functions

Resource	Used	Total	% Utilization
DSP	80	80	100
BRAM	41	60	68,33
LUT	6037	17600	34,3
FF	6754	35200	19,19

Loop performance for pipeline II = 1:

▣ Latency (clock cycles)

▣ Summary

Latency		Interval		Type
min	max	min	max	
12687	12687	12688	12688	none

▣ Detail

▣ Instance

▣ Loop

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- read_input	395	395	5	1	1	392	yes
- layer_1	11760	11760	30	30	1	392	yes
- layer_1_act	30	30	2	1	1	30	yes
- layer_2	52	52	4	1	1	50	yes
- layer_3	400	400	10	1	1	392	yes

Παρατηρώντας τους κύκλους που απαιτούνται αυτή τη φορά, καταλαβαίνουμε πως έχουμε μεγάλη βελτίωση σε σύγκριση με την αρχική έκδοση του κώδικα. Αυτό, όπως βλέπουμε και από το loop performance, οφείλεται στο γεγονός πως τα περισσότερα loops είναι πλέον pipelined (II = 1), και έχουν πολύ μικρότερο latency σε σύγκριση με πριν. Ωστόσο, το layer_1 loop, δεν είναι πλήρως pipelined καθώς έχει II = 30. Επομένως, συνεχίζουμε τα optimizations, ώστε να πετύχουμε pipelined σχεδίαση. (Σημειώνουμε πως οι απαιτούμενοι πόροι λόγω των pragmas έχουν αυξηθεί).

- Δοκιμάζουμε, λοιπόν, τα loop unroll directives **#pragma HLS unroll factor=X**, τα οποία στοχεύουν στη μείωση των επαναλήψεων στο εκάστοτε loop, και στην παραλληλοποίηση της εκτέλεσης. Τονίζουμε σε αυτό το σημείο, πως δοκιμάσαμε το συγκεκριμένο directive, τόσο στα εσωτερικά, όσο στα εξωτερικά loops, και για διάφορες τιμές του unroll factor (από 2 έως το max επιτρεπόμενο). Καταλήγουμε στα εξής :

- *layer_1: loop unroll factor = 2 στο εξωτερικό loop*
loop unroll factor = max = 30 στο εσωτερικό loop
- *layer_2: loop unroll factor = max = 30 στο εσωτερικό loop*
- *layer_3: loop unroll factor = max = 50 στο εσωτερικό loop*
- *layer_1_act: loop unroll factor = max = 30 (single loop)*

Τρέχοντας και πάλι τα estimations έχουμε τα εξής αποτελέσματα :

Estimations for pipeline 1 and loop unroll:

Details

Performance estimates for 'forward_propagation in main.cp ...

HW accelerated (Estimated cyc	10718
-------------------------------	-------

Resource utilization estimates for HW functions

Resource	Used	Total	% Utilization
DSP	80	80	100
BRAM	41	60	68,33
LUT	7587	17600	43,11
FF	5996	35200	17,03

Loop performance for pipeline 1 and loop unroll:

Summary

Latency		Interval		Type
min	max	min	max	
1077	1077	1078	1078	none

Detail

Instance

Loop

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- read_input	395	395	5	1	1	392	yes
- layer_1	197	197	3	1	1	196	yes
- layer_2	52	52	4	1	1	50	yes
- layer_3	400	400	10	1	1	392	yes

Παρατηρούμε πως οι κύκλοι έχουν βελτιωθεί και πάλι σε αρκετά μεγάλο βαθμό, και η υλοποίησή μας είναι πλέον πλήρως pipelined. Ωστόσο, έχουμε στη διάθεσή μας και άλλα directives που δεν έχουμε χρησιμοποιήσει, τα οποία θα μπορούσαν να βοηθήσουν στη μείωση των κύκλων. Συνεπώς, συνεχίζουμε να πειραματιζόμαστε για optimizations.

- Επιχειρούμε να χρησιμοποιήσουμε το **directive #pragma HLS partition variable=<variable> <block, cyclic, complete> factor=<int> dim=<int>** στους πίνακες που χρησιμοποιούμε, σπάζοντάς τους σε μικρότερους πίνακες, ώστε να αυξήσουμε τα input/output ports και να βελτιώσουμε το memory bandwidth. Καταλήξαμε σε partitions με :

- Partition του πίνακα *xbuf* με *block factor = 8*
- Partition του πίνακα *layer_1_out* με *block factor = 10*
- Partition του πίνακα *layer_2_out* με *block factor = 10*

Όπως θα δούμε, έχουμε μία βελτίωση με τη χρήση του συγκεκριμένου directive, αλλά είναι αρκετά μικρή (λιγότερη από 100 κύκλους βελτίωση).

- Επίσης, επιχειρήσαμε να χρησιμοποιήσουμε το directive **#pragma HLS allocation instances=<list> limit=<value> <type>**, το οποίο αποσκοπεί στον περιορισμό της χρήσης πόρων από τις συναρτήσεις που χρησιμοποιούνται, θέτοντας σε αυτές ένα άνω όριο στον αριθμό των επαναλήψεων που θα πραγματοποιήσουν. Ωστόσο, δεν παρατηρήθηκε κάποια αύξηση στην απόδοση και, συνεπώς, δε χρησιμοποιήθηκε.

Για τον optimal κώδικα, έχουμε τα εξής αποτελέσματα :

Estimations for pipeline 1 ,loop unroll and partition:

Details

Performance estimates for 'forward_propagation in main.cp ...

HW accelerated (Estimated cyc	10660
-------------------------------	-------

Resource utilization estimates for HW functions

Resource	Used	Total	% Utilization
DSP	80	80	100
BRAM	40	60	66,67
LUT	8253	17600	46,89
FF	6672	35200	18,95

Βλέπουμε για ακόμη μία φορά μία βελτίωση στους κύκλους, αν και είναι μικρή. Τονίζουμε, πως, οποιοδήποτε επιπλέον optimization μας οδηγούσε είτε σε υπέρβαση των διαθέσιμων πόρων, είτε στα ίδια αποτελέσματα, καμία δηλαδή βελτίωση.

Τρέχοντας τον τελικό και βελτιωμένο κώδικα στο zybo, παίρνουμε τις παρακάτω πληροφορίες καθώς και το αντίστοιχο output.txt που θα χρησιμοποιήσουμε στην άσκηση 2:

```
sh-4.3# ./lab4ex1opt.elf
Starting dataset parsing...
Parsing finished...
Starting hardware calculations...
Hardware calculations finished.
Starting software calculations...
Software calculations finished.
Hardware cycles : 10934
Software cycles : 1475554
Speed-Up       : 134.951
Saving results to output.txt...
```

Και πάλι βλέπουμε πως οι HW κύκλοι είναι αρκετά κοντά στην εκτίμηση που έκανε το SDSoC, ενώ το speed-up αυτή τη φορά έχει αυξηθεί σε πολύ μεγάλο βαθμό και είναι ίσο με περίπου 135.

Δ)

Τέλος, παρατηρούμε και το HLS report που παρήχθη για τον optimized κώδικα. Έχουμε τις εξής πληροφορίες :

Loop performance for pipeline 1 ,loop unroll and partition:

Latency (clock cycles)

Summary

Latency		Interval		Type
min	max	min	max	
1068	1068	1069	1069	none

Detail

Instance

Loop

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- read_input	395	395	5	1	1	392	yes
- layer_1	210	210	16	1	1	196	yes
- layer_2	52	52	4	1	1	50	yes
- layer_3	400	400	10	1	1	392	yes

Βλέπουμε και πάλι, πως έχουμε μία πλήρως pipelined σχεδίαση, ενώ τα latencies σε όλα τα loops έχουν ελαχιστοποιηθεί. Το πιο μεγάλο από αυτά ανήκει στο loop layer_3, και αυτό είναι λογικό, καθώς αυτό το loop έχει τις περισσότερες επαναλήψεις (trip count = 392), μαζί με το loop read_input, το οποίο, ωστόσο, δεν έχει όσο σύνθετες πράξεις όσο το loop layer_3.

Τέλος,κατευθυνόμαστε στην καρτέλα Resource profile του HLS report και καταγράφουμε τα είδη των μαθηματικών εκφράσεων (expressions) του design :

	BRAM	DSP	FF	LUT	Bits P0	Bits P1	Bits P2	Banks/Depth	Words	W*Bits*Banks
▼ forward_propagation	81	80	6672	8158						
> I/O Ports(2)					64					
> Instances(6)	0	0	700	1149						
> Memories(129)	81		753	316	1246			129	34326	311394
▼ Σ Expressions(511)	0	80	0	4736	5121	5038	1158			
> -	0	0	0	78	16	78	0			
> *	0	80	0	0	1118	1004	0			
> +	0	0	0	3101	3579	3558	0			
> and	0	0	0	5	5	5	0			
> ashr	0	0	0	161	54	54	0			
> icmp	0	0	0	81	215	77	0			
> or	0	0	0	19	15	7	0			
> select	0	0	0	1182	74	206	1158			
> shl	0	0	0	88	32	32	0			
> xor	0	0	0	21	13	17	0			
> Registers(527)			5219		5963					
Channels(0)	0		0	0	0			0	0	0
> Multiplexers(168)	0		0	1957	1952			0		

Η έκφραση που χρειάζεται τα πιο πολλά DSP είναι ο πολλαπλασιασμός (*), και αυτό οφείλεται πιθανότατα στα optimizations που κάναμε, τα οποία οδήγησαν στην ανάγκη για περισσότερους και ταυτόχρονους πολλαπλασιασμούς στην υλοποίηση.

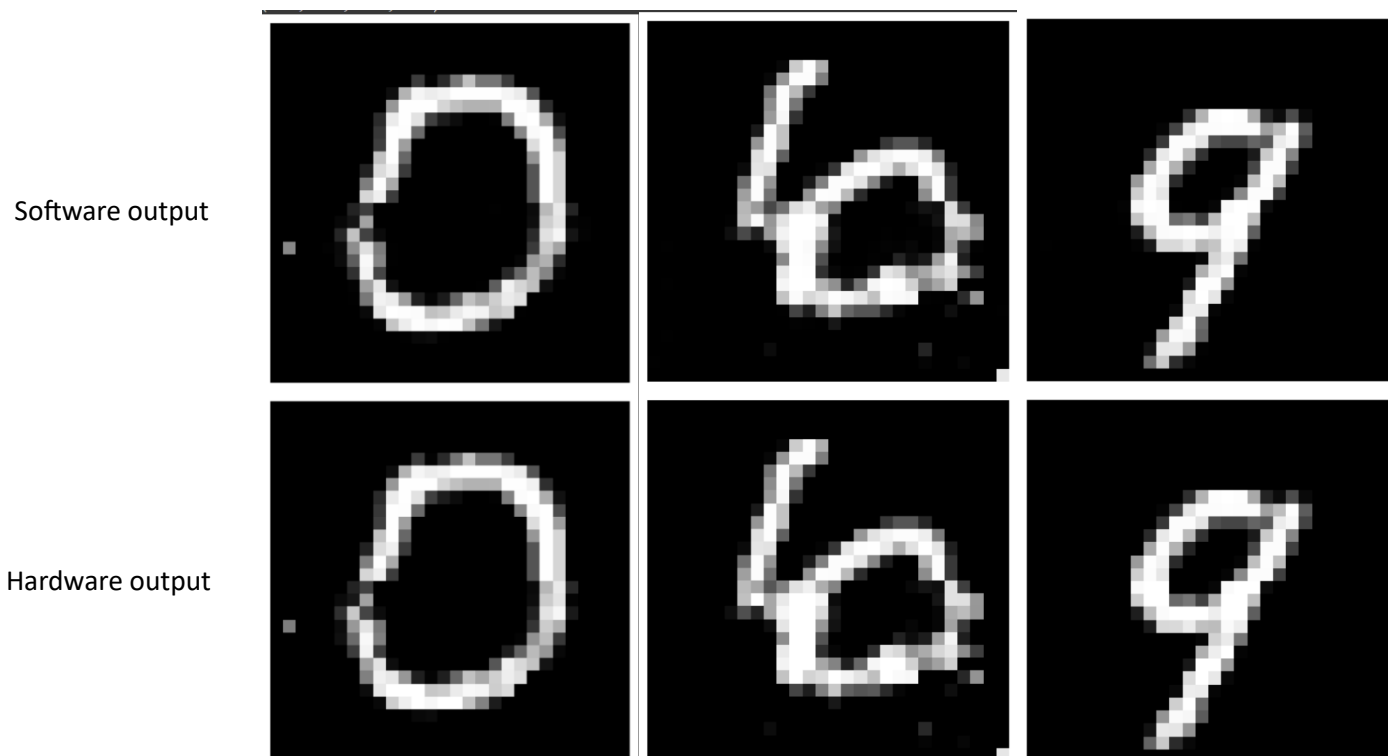
Άσκηση 2. Quality measurement

Σε αυτό το σημείο καλούμαστε να μετρήσουμε την ποιότητα ανακατασκευής των εικόνων μέσω του κώδικα που μας δίνεται, στο juryter notebook. Μέσω αυτού θα κάνουμε combine τις μισές εικόνες που δόθηκαν σαν input στο data.txt με τις μισές εικόνες από το output.txt που παρήγαγε το SW αλλά και το HW.

1)

Αφού έχουμε πάρει τα αποτελέσματα του SW και HW στο αρχείο output.txt από το zybo, στην άσκηση 1, είμαστε έτοιμοι να τρέξουμε το plot_output.ipynb (από google collab online), και να εμφανίσουμε τις combined εικόνες από SW και HW για idx: 10, 11, 12 . Η τιμή του idx συμβολίζει τον αριθμό τον οποίο θα κάνει generate η εφαρμογή. Δηλαδή, για idx = 10 εμφανίζεται ο αριθμός 0, για idx = 11 εμφανίζεται ο αριθμός 6 και για idx = 12 εμφανίζεται ο αριθμός 9. Τα αποτελέσματα που εμφανίζονται είναι τα εξής :

8 bits:



Παρατηρούμε πως με γυμνό μάτι δεν είναι εμφανής κάποια διαφορά μεταξύ των δύο εικόνων για κάθε idx, συνεπώς η ανακατασκευή της εικόνας είναι αρκετά

ακριβής. Ωστόσο, όπως θα δούμε στο ερώτημα 3 αυτής της άσκησης, όπου διερευνούμε τις λεπτομέρειες ανακατασκευής, υπάρχουν ορισμένες παράμετροι που χαρακτηρίζουν ακριβέστερα την ακρίβεια αυτή και την τελική απόδοση, όπως το max pixel error και το Peak Signal-to-Noise Ratio (psnr).

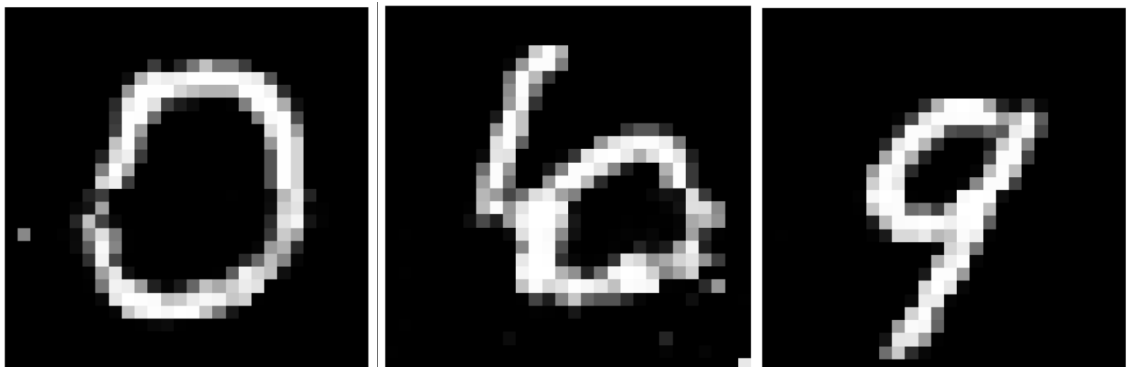
2)

Στο σημείο αυτό, καλούμαστε να αλλάξουμε τις τιμές για τα ορίσματα των datatypes που χρησιμοποιεί η υλοποίηση μου μας δόθηκε. Αλλάζουμε, λοιπόν, τις τιμές των BITS και BITS_EXP στο αρχείο network.h όπου $BITS_EXP = 2^{(BITS+2)}$ και φτιάχνουμε νέα designs με bits 4 και 10 (bits αρχικής υλοποίησης = 8). Επίσης, δεν ξεχνάμε να αλλάξουμε τις pre-computed τιμές της Tanh στο αρχείο tanh.h για κάθε τιμή των bits. Αφού τρέξουμε και αυτές τις υλοποιήσεις στο zybo, παίρνουμε τα αρχεία output.txt και μέσω του jupyter τρέχουμε και πάλι το plot_output.ipynb.

Τονίζουμε πως για bits = 10, η optimized υλοποίηση είχε μεγάλη χρήση πόρων, επομένως, δεν μπορούσε να τρέξει στο zybo. Συνεπώς, και μόνο για bits = 10, χρησιμοποιήσαμε την unoptimized υλοποίηση, παρόλο που τα αποτελέσματα δεν επηρεάζονται από την επιλογή αυτή. Τα αποτελέσματα που παίρνουμε ακολουθούν παρακάτω :

4 Bits

Software output

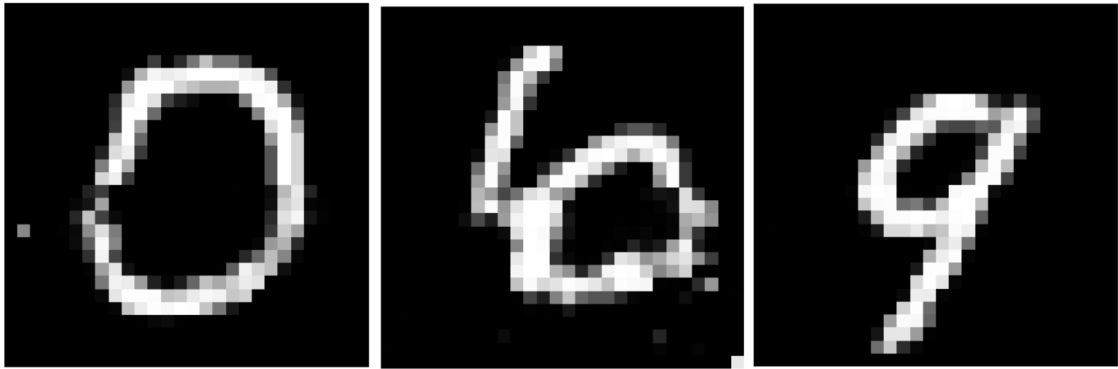


Hardware output

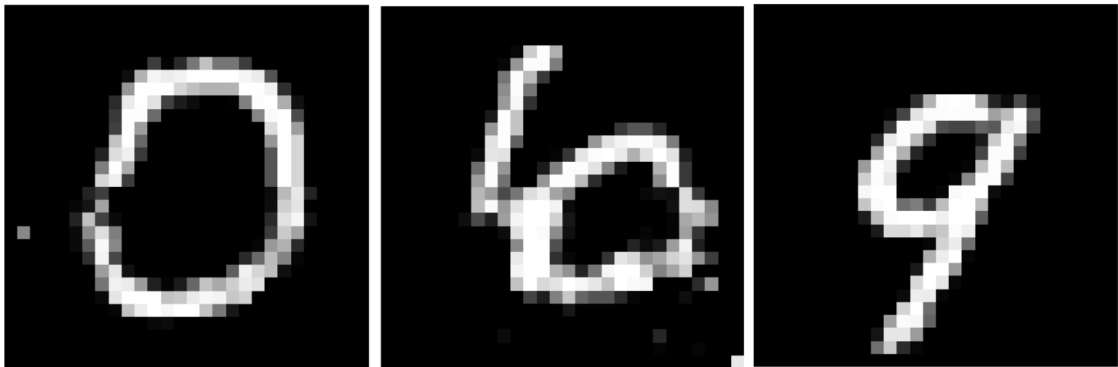


10 Bits

Software output



Hardware output



Παρατηρούμε τα εξής:

- Για bits = 10, οι εικόνες μεταξύ SW και HW είναι πανομοιότυπες και σε σύγκριση με την υλοποίηση των 8 bits, δεν μπορούμε οπτικά να διακρίνουμε κάποια έντονη διαφορά.

- Για bits = 4, οι διαφορές μεταξύ των εικόνων σε SW και HW είναι ιδιαίτερα εμφανής, με τις δεύτερες να είναι πολύ πιο ανακριβείς σε σχέση με αυτές των 8 και 10 bits. Αυτό είναι απολύτως λογικό, και ευθύνεται στο γεγονός πως λόγω έλλειψης των απαιτούμενων bits για την αναπαράσταση μιας συγκεκριμένης απόχρωσης του γκρι (το οποίο είναι μία ενδιάμεση τιμή μεταξύ άσπρου και μαύρου, δηλαδή τιμή στο σύνολο $(0,1)$), αποδίδει σε ορισμένα pixels τις ακραίες τιμές, δηλαδή 0 ή 1. Έτσι, όπως φαίνεται εύκολα, αν κοιτάξουμε προς το αριστερό μέρος στο νούμερο 0 στο SW όπου έχουμε μία “ασθενή” απόχρωση του γκρι (φαίνεται σα να τείνει να γίνει μαύρο), λόγω της έλλειψης bits, η υλοποίηση αποφασίζει να βάλει μαύρο χρώμα, έτσι στο σημείο αυτό κόβεται ο αριθμός μας στο HW. Στην αντίθετη περίπτωση, στο δεξί μέρος όπου η απόχρωση του γκρι τείνει προς το άσπρο, η

υλοποίηση μας αντίστοιχα αποφασίζει να βάλει στο σημείο αυτό άσπρα pixels στο HW.

3)

Στο jupyter notebook μετριέται η ποιότητα ανακατασκευής των εικόνων του HW σε σχέση με το SW. Τα αποτελέσματα για το max pixel error και το Peak Signal-to-Noise Ratio (psnr) για τις διάφορες τιμές των bits που δοκιμάσαμε, είναι αυτά τα οποία θα μας δώσουν περισσότερες πληροφορίες με στόχο την επιλογή της βέλτιστης υλοποίησης, και καταγράφονται παρακάτω:

8 Bits

```
Max pixel error: 16  
Peak Signal-to-Noise Ratio: 42.6822168370888
```

idx = 10

```
Max pixel error: 17  
Peak Signal-to-Noise Ratio: 42.56993337396983
```

idx = 11

```
Max pixel error: 13  
Peak Signal-to-Noise Ratio: 47.065287020211215
```

idx = 12

4 Bits

```
Max pixel error: 255  
Peak Signal-to-Noise Ratio: 14.051663945384881
```

idx = 10

```
Max pixel error: 249  
Peak Signal-to-Noise Ratio: 14.634988266184102
```

idx = 11

```
Max pixel error: 255  
Peak Signal-to-Noise Ratio: 13.525831164368576
```

idx = 12

10 Bits

```
Max pixel error: 5  
Peak Signal-to-Noise Ratio: 54.08543347142643
```

idx = 10

```
Max pixel error: 5  
Peak Signal-to-Noise Ratio: 52.556099880280584
```

idx = 11

```
Max pixel error: 4  
Peak Signal-to-Noise Ratio: 53.76982650203158
```

idx = 12

Στο σημείο αυτό είναι ανάγκη να αναλύσουμε τη σημασία της κάθε παραμέτρου :

Max pixel Error:

Η παράμετρος αυτή μας δίνει την πληροφορία σχετικά με τη μεγαλύτερη απόκλιση ενός μεμονομένου pixel στη μία εικόνα, σε σύγκριση με το αντίστοιχο pixel στην άλλη εικόνα. Για την παράμετρο αυτή προφανώς, προτιμούμε όσο το δυνατόν μικρότερη τιμή.

Peak Signal-to-Noise Ratio (psnr):

Ο όρος αυτός εκφράζει την αναλογία μεταξύ της μέγιστης δυνατής ισχύος ενός σήματος και της ισχύος του αλλοιωτικού θορύβου που επηρεάζει την πιστότητα της αναπαράστασής του. Η μεταβλητή αυτή, σε αντίθεση με το max pixel Error, λαμβάνει υπόψη ολόκληρη την κατανομή και όχι ένα μεμονομένο pixel. Συνεπώς, είναι περισσότερο αντιπροσωπευτική για την ακρίβεια της εφαρμογής, σε σύγκριση με την προηγούμενη. Για την παράμετρο αυτή, προτιμούμε όσο το δυνατόν μεγαλύτερη τιμή.

Σχολιασμός αποτελεσμάτων :

Από τα παραπάνω αποτελέσματα, καταλαβαίνουμε πως και οι δύο παράμετροι βελτιώνονται όσο περισσότερα bits χρησιμοποιούμε για την υλοποίησή μας. Ωστόσο, παρατηρούμε πως το ποσοστό βελτίωσης από 8 σε 10 bits, είναι αρκετά μικρότερο σε σύγκριση με το ποσοστό βελτίωσης από 4 σε 8 bits. Αυτό, εξ' αλλου, είναι κατανοητό και από τα οπτικά αποτελέσματα που πήραμε, καθώς για 4 bits είχαμε πολύ χειρότερες εικόνες σε σύγκριση με τα 8 bits.

Όσον αφορά στο ποια τεχνική θα προτιμούσαμε, αυτό είναι ένα ερώτημα στο οποίο η απάντηση δεν είναι ξεκάθαρη. Θυμίζουμε πως για την υλοποίηση με 10 bits, οι απαιτούμενοι πόροι της optimized έκδοσης του κώδικα ήταν τόσοι πολλοί, που υπερέβαιναν τα όρια. Στο άλλο άκρο, για 4 bits, ενώ οι πόροι αρκούσαν για τον optimized κώδικα, τα αποτελέσματα της εφαρμογής που πήραμε δεν ήταν ιδιαίτερα ακριβή. Τέλος, για τα 8 bits, είχαμε μία ενδιάμεση λύση στα δύο αυτά προβλήματα, καθώς οι πόροι για τον optimized κώδικα ήταν επαρκείς και τα αποτελέσματα της εφαρμογής είχαν αρκετή ακρίβεια.

Έχοντας τα παραπάνω υπόψη, κατανοούμε πως η βέλτιστη τεχνική μέτρησης έγκυται στην εκάστοτε εφαρμογή που υλοποιούμε και στους συμβιβασμούς που μπορούμε να κάνουμε.