Σχεδιασμός Ενσωματωμένων Συστημάτων

4η Εργαστηριακή Άσκηση

Εργασία για High Level Synthesis σε FPGA

Νικολέτα-Μαρκέλα Ηλιακοπούλου
03116111

Φοίβος-Ευστράτιος Καλεμκερής
03116010

1 Εφαρμογή: Ανακατασκευή εικόνας με GAN

Η εφαρμογή αφορά την ανακατασκευή εικόνων από το MNIST dataset (28x28 χειρόγραφοι grayscale αριθμοί). Συγκεκριμένα, έχουμε το generator νευρωνικό μοντέλο από το GAN με την υλοποίηση του σε C μαζί με τα trained βάρη. Η εφαρμογή δέχεται ως input το πάνω μισό των αριθμών και καλεί το νευρωνικό για να κάνει generate/predict το υπόλοιπο μισό της εικόνας. Μετά την εκτέλεση φαίνεται ο μέσος χρόνος εκτέλεσης ανά εικόνα σε κύκλους για SW και HW και το speed-up. Σκοπός της άσκησης είναι η μεγιστοποίηση του speed-up με HLS optimizations στην HW function, αλλά και η αξιολόγηση της ποιότητας ανακατασκευής των εικόνων.

Για την εφαρμογή των HLS optimizations, καθώς και την εκτίμηση πόρων και επίδοσης θα χρησιμοποιήσουμε το εργαλείο SDSoC.

2 Performance and resources measurement

Από τον κώδικα που μας δίνεται, η συνάρτηση forward_propagation είναι ο κώδικας C που τρέχει το νευρωνικό, συγκεκριμένα ο generator. Αποτελείται από διάφορα layers που τρέχουν το ένα μετά το άλλο (dense⊠relu⊠dense⊠relu⊠dense⊠tanh) με σκοπό να παράξουν το κάτω μισό της εικόνας. Ουσιαστικά πρόκειται για πολ/σμούς matrix-vector το οποίο είναι μια διαδικασία με πολλές πράξεις, αργή για CPU.
Στο πρώτο μέρος της άσκησης θα δοκιμάσουμε να τρέξουμε την εν λόγω συνάρτηση στο Zybo FPGA, εφαρμόζοντας σε αυτή HLS βελτιστοποιήσεις με σκοπό την επιτάχυνσή της.

2.1 Εκτέλεση

1. Αρχικά δοκιμάσαμε να θέσουμε τη συνάρτηση forward_propagation ως HW function και να κάνουμε estimate performance χωρίς να κάνουμε οποιοδήποτε optimization. Προκειμένου να γίνει αυτό, απενεργοποιήσαμε την παραγωγή bitstream και SD card image και ενεργοποιήσαμε την επιλογή εκτίμησης επίδοσης. Ύστερα, κάναμε build και καταγράψαμε το report που δείχνει τα estimated resources και cycles της hardware function:

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

HW accelerated (Estimated cycles)	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

Σχήμα 1: Εκτίμηση αριθμού κύκλων και απαιτούμενων πόρων για τον αρχικό κώδικα

2. Στη συνέχεια, απενεργοποιήσαμε την επιλογή εκτίμησης επίδοσης, ενεργοποιήσαμε τις επιλογές για παραγωγή bitstream και SD card image και ξανακάναμε build. Ακολούθως, μεταφέραμε με scp στο zybo το φάκελο sd_card, όπως παράχθηκε εντός του Debug, καθώς και το αρχείο data.txt που περιλαμβάνει την είσοδο του αλγορίθμου. Αντικαταστήσαμε το περιεχόμενο του directory /mnt του zybo με το περιεχόμενο της sd_card και κάναμε reboot. Έπειτα τρέχοντας το εκτελέσιμο (*./esd_lab4.elf*) εντός του /mnt λαμβάνουμε τα παρακάτω αποτελέσματα:

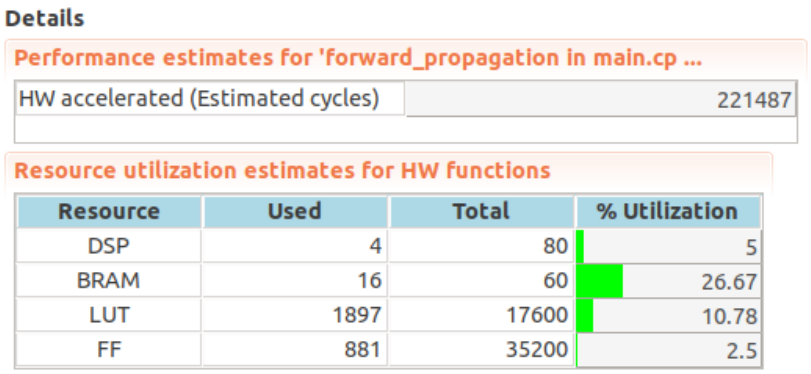
```
Hardware cycles : 682945
Software cycles : 1473208
Speed-Up        : 2.15714
```

Σχήμα 2: Αριθμός Hardware και Software κύκλων κατά την εκτέλεση στο Zybo

Όπως βλέπουμε, ο πραγματικός αριθμός των κύκλων στο hardware είναι πολύ κοντά στην πρόβλεψη που πραγματοποίησε το SDSoc. Επιπλέον, παρατηρούμε ότι ακόμα και χωρίς την εφαρμογή οποιασδήποτε βελτιστοποίησης, με την εκτέλεση της συνάρτησης στο FPGA, έχουμε speed-up ίσο με 2.15714.

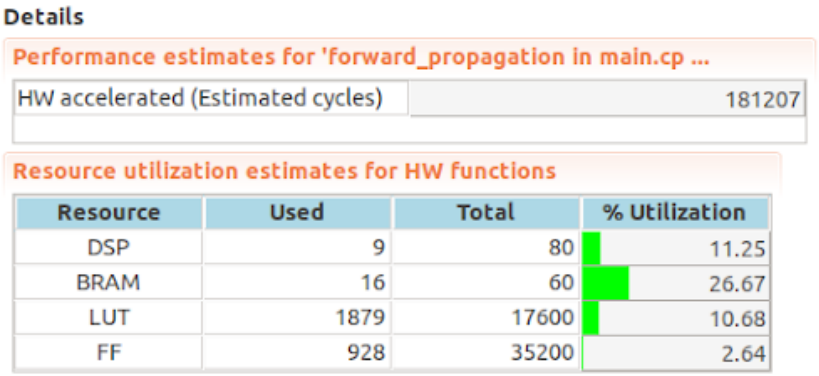
3. Σε αυτό το σημείο κληθήκαμε να κάνουμε design space exploration για να βρούμε τα απαιτούμενα optimizations, ώστε να επιταχυνθεί σημαντικά ο αλγόριθμος. Δοκιμάσαμε ειδικότερα τα προτεινόμενα HLS pragmas προκειμένου να γίνει pipelined ο αλγόριθμος και να επιταχυνθεί η εκτέλεση των loops.

   - Ξεκινήσαμε, όπως και στην προτεινόμενη μεθοδολογία βελτιστοποίησης με τον προσδιορισμό του trip count για κάθε loop. Φυσικά, αυτό είναι σταθερό για κάθε loop, αφού τα N1, M1, N2, M2, N3 και M3 είναι σταθερές ορισμένες στο network.h. Θέσαμε, συνεπώς, το avg tripcount ίσο με το εκάστοτε όριο του βρόχου. Όπως αναμέναμε, η αλλαγή αυτή δεν επέφερε καμιά αλλαγή στην επίδοση.
   - Στη συνέχεια, αρχίσαμε να προσθέτουμε σταδιακά pipeline pragmas στα loops. Ειδικότερα, χρησιμοποιήσαμε την οδηγία #pragma HLS PIPELINE, η οποία προσπαθεί by default να εφαρμόσει το καλύτερο δυνατό pipeline, εκκινώντας από το ιδανικό σενάριο δρομολόγησης νέας επανάληψης σε κάθε κύκλο ρολογιού (initiation interval = 1).
   - Προσθέτοντας την οδηγία #pragma HLS PIPELINE στα εσωτερικά nested, αλλά και στα απλά loops λάβαμε αισθητά καλύτερη εκτίμηση επίδοσης:



Σχήμα 3: Pipeline στα απλά και εσωτερικά loops

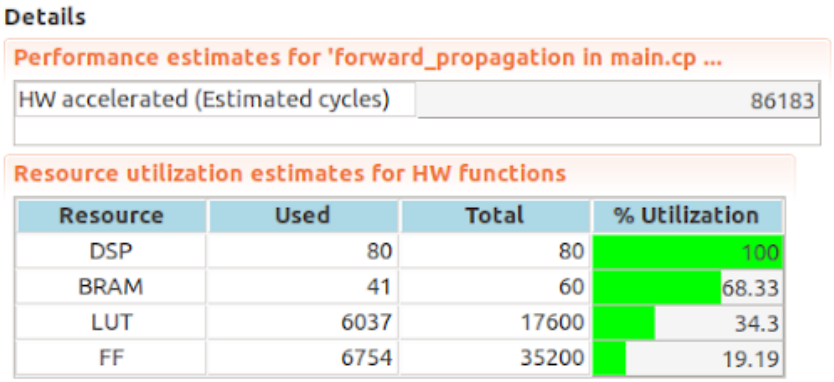
   - Στη συνέχεια, δοκιμάσαμε την προσθήκη της οδηγίας #pragma HLS unroll factor=2 στα εσωτερικά loops των διπλών βρόχων, λαμβάνοντας σημαντική βελτίωση της επίδοσης:



Σχήμα 4: Unroll factor=2 στα εσωτερικά loops

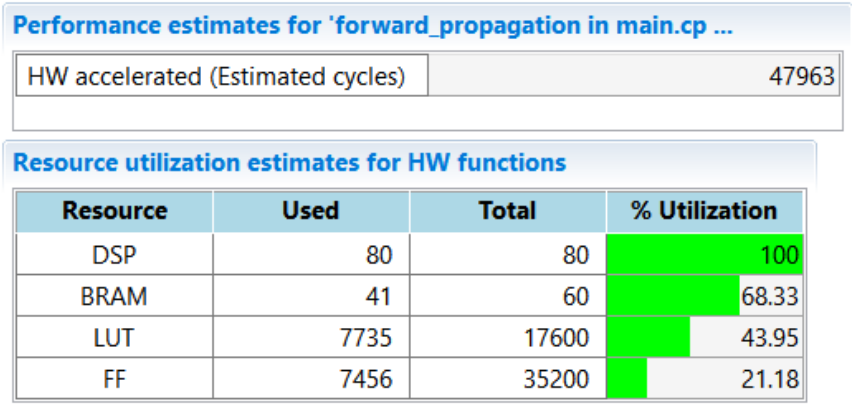
Μεγαλύτερο unroll factor στα εσωτερικά loops δεν επέφερε καμία βελτίωση. Από την άλλη, η εφαρμογή του unroll και στα απλά loops, όπως το read_input και το layer_1_act όχι μόνο δε βελτίωσε την επίδοση, αλλά προκάλεσε αύξηση των απαιτούμενων πόρων, οπότε απορρίφθηκε.

- Εφαρμόσαμε, ύστερα, pipeline και στα εξωτερικά loops των τριών layers. Η αλλαγή αυτή υπήρξε κρίσημη για τη συνέχεια της βελτιστοποίησης, αφού μείωσε τους κύκλους κατά περισσότερο από 50%:



Σχήμα 5: Pipeline και στα εξωτερικά loops

- Έπειτα, δοκιμάσαμε την εφαρμογή unroll και στα εξωτερικά loops. Όπως διαπιστώθηκε, για τα επίπεδα 2 και 3 αυτό είναι ανέφικτο, αφού ακόμα και για unroll factor 2, οι απαιτήσεις σε BRAM υπερβαίνουν κατά 4 φορές τους διαθέσιμους πόρους. Ωστόσο, στο εξωτερικό loop του επιπέδου 1, εφαρμογή unroll factor 2 υποτετραπλασίασε τον εκτιμώμενο αριθμό κύκλων:



Σχήμα 6: Unroll factor 2 στο εξωτερικό loop του layer 1

- Η παραπάνω αλλαγή έδειξε ότι με το ξετύλιγμα του εξωτερικού loop του layer 1 βρισκόμαστε στο σωστό δρόμο, επομένως δοκιμάσαμε μεγαλύτερα unroll factors. Πράγματι, διαπιστώσαμε ότι αύξηση του unroll factor στο εξωτερικό loop του layer 1 προκαλεί σταθερή μείωση στον αριθμό κύκλων, αλλά και αύξηση της χρησιμοποίησης πόρων. Συνεπώς επιλέξαμε το μεγαλύτερο unroll factor που δεν προκαλέι υπέρβαση του διαθέσιμου αριθμού LUTs, συγκεκριμένα το 8 (Για unroll factor είχαμε περαιτέρω μείωση των κύκλων, αλλά απαιτούνταν 102% χρησιμοποίηση LUTs και συνεπώς απορρίφθηκε).

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

HW accelerated (Estimated cycles)	24076

Resource utilization estimates for HW functions

Resource	Used	Total	% Utilization
DSP	80	80	100
BRAM	41	60	68.33
LUT	12870	17600	73.12
FF	10968	35200	31.16

Σχήμα 7: Unroll factor 8 στο εξωτερικό loop του layer 1

Θεωρώντας εδώ ότι έχουμε πετύχει μια σηματνική επιτάχυνση, ελέγξαμε το HLS Report για να λάβουμε περισσότερες πληροφορίες για το design μας.

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- read_input	395	395	5	1	1	392	yes
- layer_1	2205	2205	45	45	1	49	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

Σχήμα 8: Pretty-Optimised design: Loop Details

Παρατηρώντας τα Loop Details, όπως παρουσιάζονται παραπάνω, αναδεικνύεται ένα σημαντικό ζήτημα. Το πρόγραμμά μας, δεν είναι fully pipelined, αφού με το συνδυασμό unroll και pipeline που έχουμε χρησιμοποιήσει, καθίσταται αδύνατη η μέγιστη αξιοποίηση του pipeline στο layer 1, με αποτέλεσμα εκεί να έχουμε initiation interval ίσο με την καθυστέρηση μιας ολόκληρης επανάληψης (αντί για 1 που επιθυμούμε).
Προβληματισμένοι από την παρατήρηση αυτή δοκιμάζουμε μια διαφορετική προσέγγιση, με στόχο να έχουμε ένα fully pipelined πρόγραμμα. Ειδικότερα, αποφασίζουμε να χρησιμοποιήσουμε το μέγιστο unroll factor στις εσωτερικές λούπες και να εφαρμόσουμε pipeline μόνο στις εξωτερικές, για όλα τα επίπεδα. Η αλλαγή αυτή αποδείχτηκε κρίσιμη, καθώς όπως φαίνεται και παρακάτω έριξε τους εκτιμώμενους κύκλους κάτω απ' το μισό.

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

HW accelerated (Estimated cycles)	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

Σχήμα 9: Μέγιστο unroll factor στις εσωτερικές, Pipeline στις εξωτερικές λούπες

Στην τελευταία αυτή βελτιστοποίηση έχει χρησιμοποιηθεί και unroll factor 2 στον εξωτερικό βρόχο του επιπέδου 1, γεγονός που γλιτώνει περίπου 1000 κύκλους.

Στο σημείο αυτό, έχοντας πετύχει σύμφωνα με τις προβλέψεις του SDSoc μια σημαντική επιτάχυνση του αλγορίθμου αποφασίσαμε να παράξουμε το bitstream, προκειμένου να δοκιμάσουμε τον κώδικα πάνω στο zybo.

```
Hardware cycles : 11001
Software cycles : 1476847
Speed-Up        : 134.247
```

Σχήμα 10: Αριθμός Hardware και Software κύκλων κατά την εκτέλεση στο Zybo

Όπως παρατηρούμε, ο αριθμός των κύκλων που προκύπτει κατά την εκτέλεση στο zybo είναι και πάλι πολύ κοντά στην πρόβλεψη του SDSoc. Βλέπουμε ότι μετά την εφαρμογή των βελτιστοποιήσεων στον κώδικα επιτυγχάνουμε speedup 134.247 στο hardware σε σχέση με το software. Ο αριθμός αυτός είναι πάνω από 60 φορές μεγαλύτερος από την επιτάχυνση που παρατηρήθηκε στο hardware για τον αρχικό κώδικα. Αυτό επιτεύχθηκε με την εκτεταμμένη αξιοποίηση του pipeline και την εφαρμογή μετασχηματισμών βρόχων.

4. Τέλος, παρατηρήσαμε όσα σημειώνονται στο HLS Report του SDSoc για τη βελτιστοποιημένη έκδοση του κώδικα. Ειδικότερα, καταγράψαμε τα latency details για κάθε loop:

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- 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

Σχήμα 11: Optimised design: Loop Details

Παρατηρούμε ότι σύμφωνα με τον πίνακα η υλοποίησή μας είναι πλήρως pipelined, ενώ το πρόβλημα που εντοπίσαμε προηγουμένως, έχει εξαλειφθεί. Συγκεκριμένα, σε όλα τα επίπεδα πετυχαίνουμε το ιδανικό initiation interval, ίσο με 1.
Έπειτα, από το Resource Profile της ίδιας αναφοράς μπορούμε να λάβουμε πληροφορίες σχετικά με τη χρησιμοποίηση πόρων του FPGA, όπως προέκυψε από τη σύνθεση του design. Μας ενδιαφέρουν κυρίως οι πόροι που απαιτούνται για κάθε μαθηματική έκφραση, όπως παρουσιάζονται στον ακόλουθο πίνακα.

	BRAM	DSP	FF	LUT	Bits P0	Bits P1	Bits P2	Banks/Depth	Words	W*Bits*Banks
forward_propagation	82	80	5996	7503						
I/O Ports(2)					64					
Instances(2)	0	0	268	677						
Memories(113)	82		305	245	1032			113	34326	311394
Expressions(504)	0	80	0	4669	5062	4987	1156			
-	0	0	0	78	16	78	0			
*	0	80	0	0	1113	1004	0			
+	0	0	0	3054	3541	3530	0			
and	0	0	0	5	5	5	0			
ashr	0	0	0	161	54	54	0			
icmp	0	0	0	75	200	68	0			
or	0	0	0	20	16	8	0			
select	0	0	0	1167	72	191	1156			
shl	0	0	0	88	32	32	0			
xor	0	0	0	21	13	17	0			
Registers(524)			5423		6192					
Channels(0)	0		0	0	0			0	0	0
Multiplexers(135)	0		0	1912	1781			0		

Σχήμα 12: Optimised design: Resource Profile

Όπως προκύπτει από τα παραπάνω, η μοναδική έκφραση που απαιτεί DSP είναι ο πολλαπλασιασμός (*) που χρησιμοποιεί συγκεκριμένα 80 DSPs. Αξίζει να σημειώσουμε ότι ο αριθμός αυτός αυξήθηκε κατά πολύ σε σχέση με τον απαιτούμενο για την αρχική υλοποίηση (3). Αυτό οφείλεται στο γεγονός ότι λόγω των μετασχηματισμών που εφαρμόσαμε, περισσότερες πράξεις εκτελούνται ταυτόχρονα και υπάρχει, επομένως, ανάγκη για περισσότερες μονάδες εκτέλεσης υπολογισμών.

Άλλωστε, παρόμοια αύξηση απαιτήσεων παρατηρήθηκε και για τους υπόλοιπους πόρους. Χαρακτηριστικά, η τελική υλοποίηση απαιτεί περίπου 4 φορές περισσότερα LUTs, 2.5 φορές περισσότερη BRAM και εξαιτίας της προσθήκης του pipeline, περίπου 7 φορές περισσότερα flip-flop.

Παραθέτουμε, τέλος, τον πλήρη κώδικα, όπως προέκυψε μετά τις παραπάνω βελτιστοποιήσεις:

```
#include "network.h"
#include "weight_definitions.h"
#include "tanh.h"

l_quantized_type ReLU(l_quantized_type res)
{
  if (res < 0)
    return 0;

  return res;
}

l_quantized_type tanh(l_quantized_type res)
{
  if (res >= 2)
    return 1;
  else if (res < -2)
    return -1;
  else
  {
    ap_int <BITS+2> i = res.range();      //prepare result to match tanh value
    return tanh_vals[(BITS_EXP/2) + i.to_int()];
  }
}

void forward_propagation(float *x, float *y)
{
  quantized_type xbuf[N1];
  l_quantized_type layer_1_out[M1];
  l_quantized_type layer_2_out[M2];


  //limit resources to max DSP number of Zybo - do not change
  #pragma HLS ALLOCATION instances=mul limit=80 operation

  read_input:
  for (int i=0; i<N1; i++)
  {
    #pragma HLS PIPELINE
    xbuf[i] = x[i];
  }

  // Layer 1
  layer_1:
  for(int i=0; i<N1; i++)
  {
    #pragma HLS PIPELINE
    #pragma HLS unroll factor=2
    for(int j=0; j<M1; j++)
    {
      #pragma HLS unroll factor=30
      l_quantized_type last = (i==0) ? (l_quantized_type) 0 : layer_1_out[j];
      quantized_type term = xbuf[i] * W1[i][j];
      layer_1_out[j] = last + term;
    }
  }
  layer_1_act:
  for(int i=0; i<M1; i++)
  {
    #pragma HLS PIPELINE
    #pragma HLS unroll factor=30
```

```
    layer_1_out[i] = ReLU(layer_1_out[i]);
  }

  // Layer 2
  layer_2:
  for(int i=0; i<M2; i++)
  {
    #pragma HLS PIPELINE
    l_quantized_type result = 0;
    for(int j=0; j<N2; j++)
    {
      #pragma HLS unroll factor=30
      l_quantized_type term = layer_1_out[j] * W2[j][i];
      result += term;
    }
    layer_2_out[i] = ReLU(result);
  }

  // Layer 3
  layer_3:
  for(int i=0; i<M3; i++)
  {
    #pragma HLS PIPELINE
    l_quantized_type result = 0;
    for(int j=0; j<N3; j++)
    {
      #pragma HLS unroll factor=50
      l_quantized_type term = layer_2_out[j] * W3[j][i];
      result += term;
    }
    y[i] = tanh(result).to_float();
  }
}
```

3 Quality measurement

Στο μέρος αυτό της άσκησης καλούμαστε να μετρήσουμε τη ποιότητα ανακατασκευής των εικόνων μέσω του jupyter notebook που δίνεται. 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.

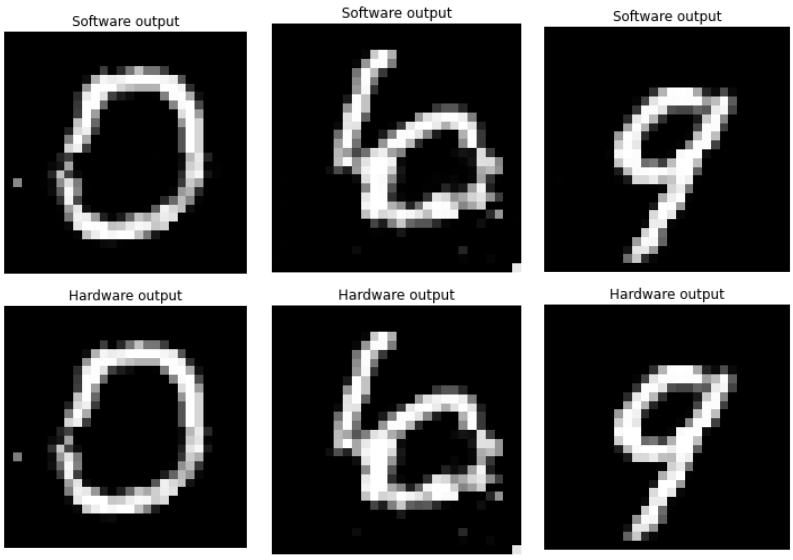
3.1 Εκτέλεση

3.1.1 Σύγκριση εικόνων SW-HW

Ανακατασκευάζουμε τις εικόνες για idx = 10, 11, 12, δηλαδή για τα νούμερα 0, 6 και 9. Σε αυτό το κομμάτι έχουμε ως δεδομένα τιμές tanh με 8-bit δεκαδική ακρίβεια, για το output layer του GAN. Για την έξόδο του SW και του HW από το zybo χρησιμοποιούμε τον βελτιστοποιημένο κώδικα για την παραγωγή του bitstream, παρόλο που το αποτέλεσμα είναι το ίδιο ανεξαρτήτως βελτιστοποίησης. Στο Σχήμα 13 παρατίθενται τα αποτελέσματα.
Παρατηρούμε πως οι εικόνες που παράγονται από το SW και το HW για συγκεκριμένο idx φαίνονται ίδιες με γυμνό μάτι. Βέβαια, όπως θα δούμε και στη συνέχεια, υπάρχουν μικρές διαφορές σε μερικά pixel. Σε γενικές γραμμές η ανακατασκευή του 2ου μισού των εικόνων επιτυγχάνεται με μεγάλη ακρίβεια.

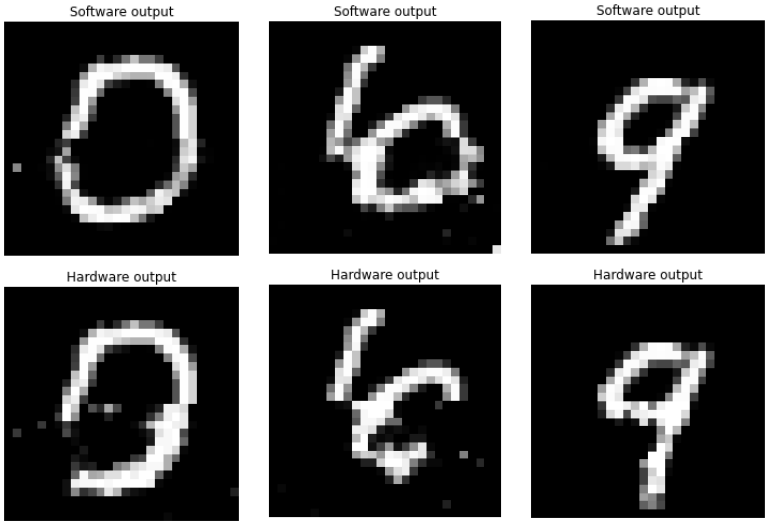
3.1.2 Ανακατασκευή εικόνων με διαφορετική ακρίβεια

Στη συνέχεια, καλούμαστε να πειραματιστούμε με την ακρίβεια των tanh values. Παράγουμε το αντίστοιχο bitstream για 4 και 10 bits ακρίβεια, αντίστοιχα. Σημειώνουμε, εδώ, πως στην περίπτωση των 10 bits δεν τρέξαμε τον βελτιστοποιημένο κώδικα, καθώς οι διαθέσιμοι πόροι (DSP) δεν επαρκούσαν. Τα αποτελέσματα των εξόδων παρουσιάζονται στα Σχήματα 14, 15.



(α′) idx=10 (β′) idx=11 (γ′) idx=12

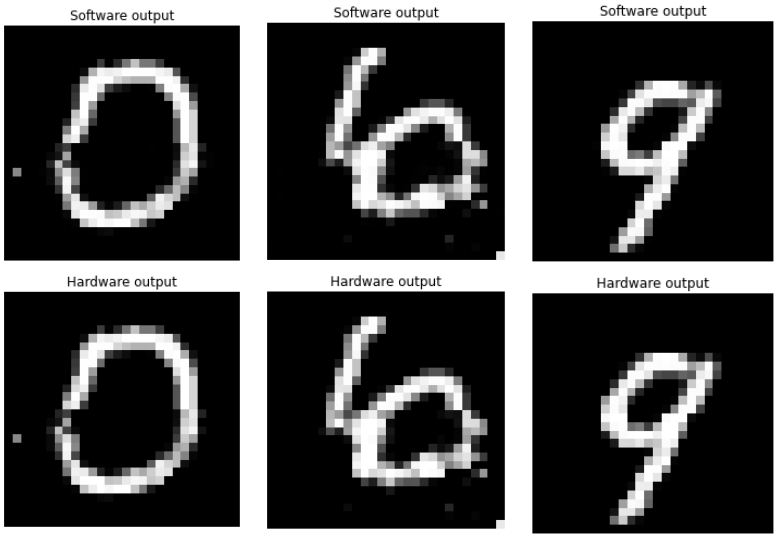
Σχήμα 13: tanh accuracy: 8 bits



(α′) idx=10 (β′) idx=11 (γ′) idx=12

Σχήμα 14: tanh accuracy: 4 bits

Για ακρίβεια tanh 4 bits, τα αποτελέσματα του SW και του HW διαφέρουν, με τα δεύτερα να είναι αισθητά πιο ανακριβή. Η μεγαλύτερη σχετική ακρίβεια σημειώνεται για idx=12 (αριθμός 9). Αυτό συμβαίνει αφού η εικόνα του SW αποτελείται κυρίως από pixel με τιμές πολύ κοντά στο 0 ή στο 1 (άσπρα, μαύρα και ελάχιστα - ενδιάμεσα - γκρι pixel). Ακόμα και χωρίς μεγάλη ακρίβεια τα pixel θα πάρουν τιμές "των άκρων" στο HW διατηρώντας κάποιο βαθμό ομοιότητας. Στους άλλους δύο αριθμούς παρατηρούμε τις μεγαλύτερες διαφορές στις περιοχές που εμφανίζονται ομάδες με ενδιάμεσες τιμές στο 0 και το 1 ("γκρι") pixel. Και σε αυτή την περίπτωση τα pixel τείνουν να πάρουν ακριανές τιμές δημιουργώντας ασυνέχειες.



(α′) idx=10 (β′) idx=11 (γ′) idx=12

Σχήμα 15: tanh accuracy: 10 bits

Στην περίπτωση ακριβείας της τάξης των 10 bit τα αποτελέσματα SW-HW μοιάζουν πανομοιότυπα στις εικόνες. Με το μάτι δεν παρατηρούμε αισθητή διαφορά σε σχέση με τις εικόνες που δημιουργήθηκαν με ακρίβεια 8 bit στο activation layer(tanh).

3.1.3 Ποιότητα ανακατασκευής των εικόνων

Όπως προκύπτει από τα παραπάνω, παρόλο που μπορούμε με το μάτι να έχουμε μια εικόνα σχετικά με την ποιότητα ανακατασκευής, είναι δύσκολο να εντοπίσουμε τις μικρές διαφορές που καθιστούν τελικά μια υλοποίηση καλύτερη της άλλης. Για να μετρήσουμε με ακρίβεια την ποιότητα ανακατασκευής των εικόνων χρησιμοποιούμε τις μετρικές Max Pixel Error και Peak Signal-to-Noise Ratio (PSNR). Παρακάτω, παρατίθενται οι διάφορες τιμές των μετρικών αυτών για όλους τους συνδυασμούς (εικόνων)-(ακρίβειας στο activation layer του GAN).

idx	**10**	**11**	**12**
Max pixel error	255	249	255
Peak Signal-to-Noise Ratio	14.051663945384881	14.634988266184102	13.525831164368576

Πίνακας 1: Μετρικές επίδοσης ανακατασκευής εικόνων tanh accuracy: 4 bits

idx	**10**	**11**	**12**
Max pixel error	16	17	13
Peak Signal-to-Noise Ratio	42.6822168370888	42.56993337396983	47.065287020211215

Πίνακας 2: Μετρικές επίδοσης ανακατασκευής εικόνων tanh accuracy: 8 bits

idx	**10**	**11**	**12**
Max pixel error	5	5	4
Peak Signal-to-Noise Ratio	54.08543347142643	52.556099880280584	53.76982650203158

Πίνακας 3: Μετρικές επίδοσης ανακατασκευής εικόνων tanh accuracy: 10 bits

Τα παραπάνω αποτελέσματα συμφωνούν σε γενικές γραμμές με τα συμπεράσματα που προέκυψαν από την εποπτική παρατήρηση των εικόνων. Ανάμεσα στις δύο μετρικές, προτιμούμε την **Peak Signal-to-Noise Ratio**, η οποία εκφράζει το λόγο της μέγιστης ισχύος ωφέλιμου σήματος (εδώ 255) προς την ισχύ του

θορύβου, η οποία υπολογίζεται ως η τετραγωνική ρίζα του μέσου των τετραγώνων της απόκλισης για κάθε pixel. Όπως γίνεται αντιληπτό, η μετρική αυτή, λαμβάνει υπόψη ολόκληρη την κατανομή, αφού βασίζεται στο μέσο όρο των αποκλίσεων.
Το max pixel error, από την άλλη, εκφράζει τη μέγιστη απόκλιση που εμφανίζεται μεταξυ των δύο εικόνων σε ένα pixel. Η συγκεκριμένη μετρική μπορεί να είναι παραπλανητική, αφού ενδέχεται κατά την ανακατασκευή των pixel να προκύψουν outliers, δηλαδή μεμονωμένα pixel με πολύ διαφορετική τιμή από την αναμενόμενη, τα οποία όμως δεν επηρεάζουν την κατανομή στο σύνολό της. Ωστόσο, για την ανακατασκευή εικόνων μας ενδιαφέρει πρωτίστως η παραγωγή ενός συνολικά καλού αποτελέσματος και, επομένως, εστιάζουμε στην "μέση" ποιότητα της παραχθείσας εικόνας. Μια διαφορετική μετρική που θα μπορούσε να χρησιμοποιηθεί αντί του max pixel error είναι το mean squared error (MSE), το οποίο χρησιμοποιείται στον παρονομαστή του PSNR και λαμβάνει υπόψη το σύνολο της κατανομής, εκφράζοντας τη μέση τετραγωνική απόκλιση της κατανομής.

Όσον αφορά την ακρίβεια στο activation layer (tanh) παρατηρούμε πως, όσο μεγαλύτερη είναι αυτή, τόσο καλύτερη είναι η ανακατασκευή των εικόνων, όπως προκύπτει από τους συγκεντρωτικούς πίνακες παραπάνω. Ειδικότερα, είναι σαφές ότι για το μεγαλύτερο διαθέσιμο αριθμό bits, έχουμε την ποιοτικότερη ανακατασκευή εικόνας, όπως υποδεικνύουν οι τιμές των 2 μετρικών. Έτσι, αν μας ενδιαφέρει μόνο το τελικό αποτέλεσμα στην εικόνα, τότε θα διαλέξουμε, φυσικά, τα **10 bits**, προκειμένου να πετύχουμε τη μεγαλύτερη ακρίβεια (PSNR $>$ 50 και το μικρότερο MPE).
Ωστόσο, στο σημείο αυτό, αναδεικνύεται ένα trade-off. Πιο συγκεκριμένα, η χρήση μεγαλύτερης ακρίβειας συνεπάγεται μεγαλύτερη χρησιμοποίηση πόρων. Επιχειρώντας να τρέξουμε τον βελτιστοποιημένο κώδικά μας μέσω του SDx για την παραγωγή του bitstream, διαπιστώσαμε πως για τα 10 bits οι διαθέσιμοι πόροι του συστήματος δεν επαρκούν (απαιτούνται περισσότεροι DSPs).
Το γεγονός αυτό σημαίνει ότι είτε θα πρέπει να επαναλάβουμε τη διαδικασία βελτιστοποίησης για τον νέο αριθμό bit στο tanh, είτε να αρκεστούμε σε ένα πρόγραμμα κατώτερων χρονικών επιδόσεων, προκειμένου να πετύχουμε μεγαλύτερη ακρίβεια ανακατασκευής.
Φαίνεται, ως εκ τούτου, ότι υπάρχει ένα σημαντικό trade-off μεταξύ ποιότητας ανακατασκευής εικόνων και επίδοσης κώδικα. Σε αυτή την περίπτωση επικεντρωνόμαστε στον παράγοντα που μας ενδιαφέρει περισσότερο για κάθε εφαρμογή και επιλέγουμε ανάλογα την ακρίβεια και τα optimizations. Συχνά, η επιλογή μπορεί να είναι ένας συμβιβασμός μεταξύ των δύο παραγόντων, με χαρακτηριστικό παράδειγμα την περίπτωση των 8 bit που επιτυγχάνει εξαιρετική επίδοση- όπως αναλύθηκε στο πρώτο μέρος της άσκησης- αλλά και ιδιαίτερα ικανοποιητική ποιότητα ανακατασκευής.

Σημείωση: Στο φάκελο του project μπορούν να βρεθούν- εκτός από την αναφορά- το σύνολο του C source code για την ανακατασκευή της εικόνας με το βελτιστοποιημένο network.cpp, το αρχείο data.txt που περιέχει τις μισές εικόνες, καθώς και το notebook όπου παρουσιάζονται τα αποτελέσματα της ανακατασκευής που αναλύσαμε προηγουμένως.