



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΙΣΤΩΝ

ΣΧΕΔΙΑΣΜΟΣ ΕΝΣΩΜΑΤΩΜΕΝΩΝ ΣΥΣΤΗΜΑΤΩΝ – 7^ο ΕΞΑΜΗΝΟ

Μαρκέτος Νικόδημος – ΑΜ:03117095

Τζομάκα Αφροδίτη – ΑΜ: 03117107

4^η ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ

Άσκηση 1 – Performance and resources measurement

A.

Details

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

B. Αφού μεταφέραμε τα απαραίτητα αρχεία sd_card και bitstream που παρήγαγε το Xilinx SDx Project και τρέξαμε την εφαρμογή στο board παίρνουμε τα εξής αποτελέσματα:

```
Starting dataset parsing...
Parsing finished...
Starting hardware calculations...
Hardware calculations finished.
Starting software calculations...
Software calculations finished.
Hardware cycles : 683060
Software cycles : 1476351
Speed-Up : 2.1613
Saving results to output.txt...
```

Παρατηρούμε ότι οι hw κύκλοι συμφωνούν με το estimation που πήραμε προηγουμένως ενώ έχουμε speed-up ≈ 2 .

Γ. Προσθέτοντας στον κώδικα κάποια HLS pragmas και τρέχοντας ξανά την εφαρμογή στο zybo παίρνουμε τα εξής αποτελέσματα:

```
Starting dataset parsing...
Parsing finished...
Starting hardware calculations...
Hardware calculations finished.
Starting software calculations...
Software calculations finished.
Hardware cycles : 12473
Software cycles : 1475898
Speed-Up : 118.327
Saving results to output.txt...
```

Παρατηρούμε ότι τα αποτελέσματα που δίνουν τα optimizations του HLS προσφέρουν πολύ ικανοποιητικό speed up κάνοντας τον κώδικα 118 φορές πιο γρήγορο(ο βελτιστοποιημένος κώδικας φαίνεται στην επόμενη σελίδα).

Δ. Παρακάτω φαίνεται το latency report που αναφέρεται στην optimized υλοποίησή μας:

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	393	393	3	1	1	392	yes
- layer_1_act	60	60	2	-	-	30	no
- layer_2	52	52	4	1	1	50	yes
- layer_3	400	400	10	1	1	392	yes

Παρατηρούμε ότι το μεγαλύτερο latency έχει το layer_3, αποτέλεσμα που οφείλεται στις συναρτήσεις tanh και to_float στο εξωτερικό loop. Αυτό φαίνεται και στο μεγαλύτερο iteration latency που βλέπουμε στο report.

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
forward_propagation	-	1309	1310	-	-
read_input	yes	395	1	5	392
layer_1	yes	393	1	3	392
layer_1_act	no	60	-	2	30
layer_2	yes	52	1	4	50
layer_3	yes	400	1	10	392

Η υλοποίησή μας, όπως φαίνεται και παραπάνω, δεν είναι fully pipelined. Πιο συγκεκριμένα στο layer_1_act δεν εφαρμόσαμε pipeline καθώς καλεί μια υπό-συνάρτηση, την ReLu, η οποία δεν είναι η ίδια pipelined. Επομένως δεν θα βλέπαμε κάποια ιδιαίτερη καλυτέρευση στην απόδοση της εφαρμογής αλλά αντιθέτως ίσως και να αποτελούσε ελαττωματικό παράγοντα όσον αφορά στην κατανάλωση πόρων (καταχωρητών).

Προχωρώντας στην ανάλυση των μαθηματικών εκφράσεων που περιέχει το design παίρνουμε τα εξής δεδομένα:

	BRAM [^]	DSP	FF	LUT	Bits P0	Bits P1	Bits P2	Banks/Depth	Words	W*Bits*Banks
▼ forward_propagation	81	80	6978	6601						
> I/O Ports(2)					64					
> Instances(4)	0	0	268	983						
> Memories(139)	81		833	316	1286			139	34276	310594
▼ Σ Expressions(423)	0	80	0	3646	4468	4382	694			
> -	0	0	0	78	16	78	0			
> *	0	80	0	0	1112	999	0			
> +	0	0	0	2483	2962	2930	0			
> and	0	0	0	5	5	5	0			
> ashr	0	0	0	161	54	54	0			
> icmp	0	0	0	83	223	81	0			
> or	0	0	0	6	6	6	0			
> select	0	0	0	721	45	180	694			
> shl	0	0	0	88	32	32	0			
> xor	0	0	0	21	13	17	0			
> Registers(468)			5877		6447					
Channels(0)	0		0	0	0			0	0	0
> Multiplexers(136)	0		0	1656	1651			0		

Βλέπουμε πως τα περισσότερα DSPs απαιτεί, όπως περιμέναμε, η πράξη του πολλαπλασιασμού.

Βελτιστοποιημένος Κώδικας – network.cpp

```
#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
#pragma HLS ARRAY_PARTITION variable=xbuf cyclic factor=28
#pragma HLS ARRAY_PARTITION variable=layer_1_out
#pragma HLS ARRAY_PARTITION variable=layer_2_out

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
        for(int j=0; j<M1; j++)
        {
            #pragma HLS UNROLL factor=15
                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++)
{
    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=15
                l_quantized_type term = layer_1_out[j] * W2[j][i];
                result += term;
        }
        layer_2_out[i] = ReLU(result);
}

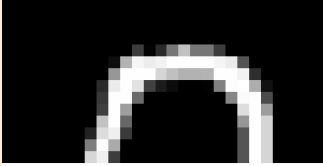
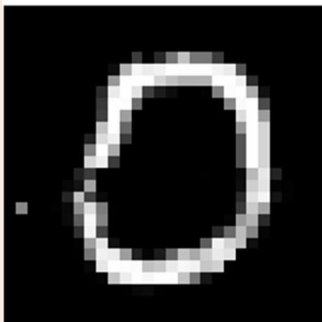
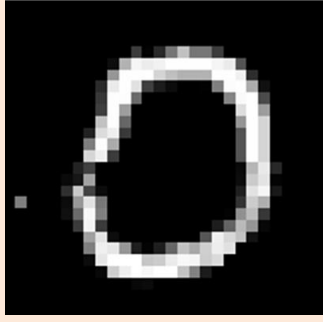
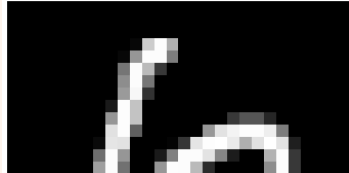

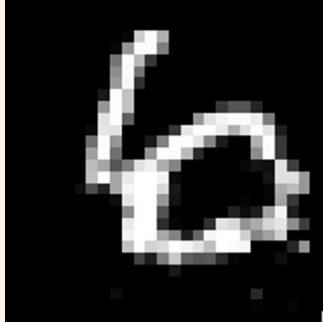



// Layer 3
layer_3:
for(int i=0; i<M3; i++)
{
    #pragma HLS PIPELINE
        l_quantized_type result = 0;
        for(int j=0; j<N3; j++)
        {
            #pragma HLS UNROLL factor=25
                l_quantized_type term = layer_2_out[j] * W3[j][i];
                result += term;
        }
        y[i] = tanh(result).to_float();
}
}

```

Άσκηση 2 - Quality measurement

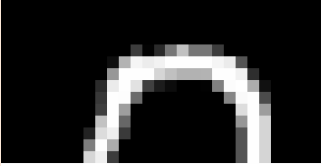
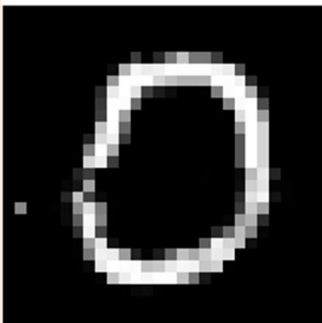




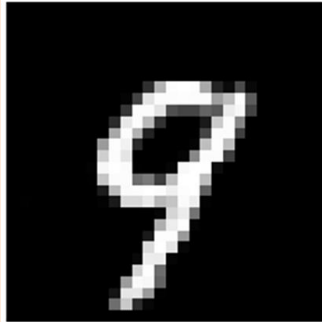
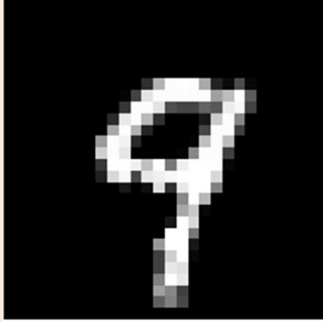
A. Τρέξαμε το plot_output.ipynb στο google collab online και πήραμε τις παρακάτω combined εικόνες:

➤ 8-bit decimal precision:

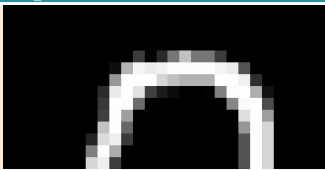
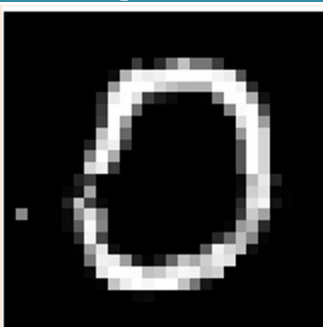
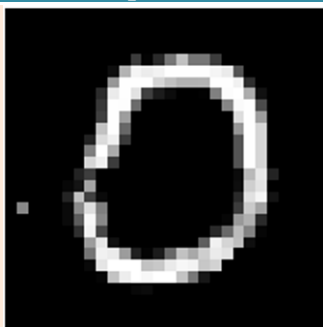
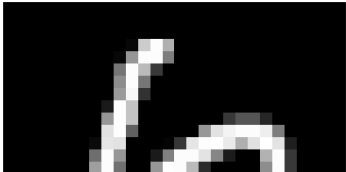

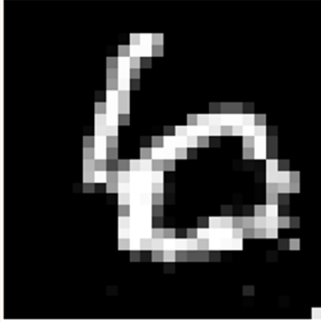

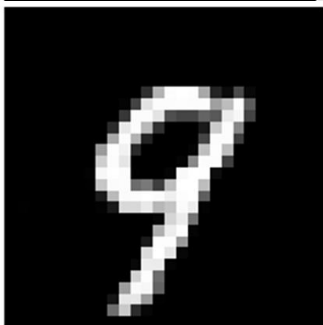
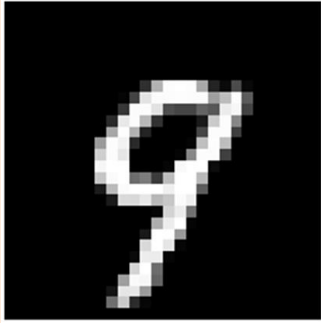
<i>idx</i>	<i>Input</i>	<i>SW Output</i>	<i>HW Output</i>
10			
11			
12			

B. Τρέχοντας ξανά όπως πριν αλλά αυτήν την φορά μεταβάλλοντας τα bit δεκαδικής ακρίβειας των datatypes από 8 σε 4 και ύστερα σε 10 παίρνουμε τα παρακάτω αποτελέσματα:

➤ *4-bits decimal precision:*

<i>idx</i>	<i>Input</i>	<i>SW Output</i>	<i>HW Output</i>
10			
11			
12			

➤ 10-bits decimal precision:

<i>idx</i>	<i>Input</i>	<i>SW Output</i>	<i>HW Output</i>
10			
11			
12			

Εκείνο που παρατηρούμε είναι και το θεωρητικά αναμενόμενο. Πιο συγκεκριμένα βλέπουμε ότι για μικρότερη ακρίβεια δεκαδικών ψηφίων (4 bits) οι αριθμοί δεν είναι αρκετά καθαροί ενώ για μεγαλύτερη (από τα 8 bit και ύστερα) έχουμε ικανοποιητικές εικόνες.

Γ. Από τα αποτελέσματα των υπολογισμών προέκυψαν τα ακόλουθα:

➤ *Index = 10:*

Precision	Max Pixel Error	Peak Signal-to-Noise Ratio
4 bits	255	14.052
8 bits	16	42.682
10 bits	5	54.085

➤ *Index = 11:*

Precision	Max Pixel Error	Peak Signal-to-Noise Ratio
4 bits	249	14.635
8 bits	17	42.570
10 bits	5	52.556

➤ *Index = 12:*

Precision	Max Pixel Error	Peak Signal-to-Noise Ratio
4 bits	255	13.526
8 bits	13	47.065
10 bits	4	53.770

Εν προκειμένω εκείνο που μας ενδιαφέρει είναι η ακρίβεια των εικόνων οπότε θα εστιάσουμε στο Peak Signal-to-Noise Ratio ως μετρική. Συνεπώς σύμφωνα με αυτήν την σύμβαση καταλήγουμε στο ότι η ιδανική για την περίπτωση μας ακρίβεια είναι εκείνη των 10 bits (μεγαλύτερο Peak Signal-to-Noise Ratio).

Το Max Pixel Error αναφέρεται συγκεκριμένα σε ένα pixel και την μέγιστη απόκλιση αυτού. Επομένως θεωρείται ενδεικτικό της γενικής «καθαρότητας» της εικόνας.