ΕΘΝΙΚΟ ΜΕΤΣΟΒΕΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧ. ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
Τομέας Τεχνολογίας Υπολογιστών και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων

Σχεδιασμός Ενσωματωμένων Συστημάτων
9^{Ο} Εξάμηνο ΗΜΜΥ

4η ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ

Εργασία για High Level Synthesis σε FPGA:
Ανακατασκευή εικόνας με GAN

Ημ/νια Παράδοσης : 13/12/2024
Ομάδα 22
Παναγιώτης Μπέλσης AM : 03120874
Θεοδώρα Εξάρχου AM : 03120865

Ο σκοπός της άσκησης είναι να γίνει μελέτη γύρω από τον προγραμματισμό FPGA με High Level Synthesis (HLS). Η άσκηση αυτή παρουσιάζει την επιτάχυνση ενός Generative Adversarial Network (GAN) που έχει ως στόχο την ανακατασκευή ημιτελών εικόνων χειρόγραφων ψηφίων από το dataset MNIST (28x28 grayscale). Ο αλγόριθμος αφορά τον Generator του GAN, ο οποίος υλοποιείται σε C και δέχεται το πάνω μισό μιας εικόνας ως είσοδο για να προβλέψει το κάτω μισό.

Άσκηση 1. Performance and resources measurement

Η συνάρτηση η οποία εκτελείται στο hardware είναι η forward_propagation() από το αρχείο network.cpp που τρέχει το νευρωνικό δίκτυο, συγκεκριμένα ο generator. Το συγκεκριμένο νευρωνικό δίκτυο αποτελείται από 3 layers 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. Επιθυμούμε να πετύχουμε μεγιστοποίηση του αρχικού speedup με χρήση HLS pragmas στην Hardware υλοποίηση.

A) Αρχικά τρέχουμε το estimation της εφαρμογής χωρίς κανένα optimization (σειριακή εκτέλεση)
Το estimation είναι :

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

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	3	0	888
FIFO	-	-	-	-
Instance	-	0	268	677
Memory	33	-	66	21
Multiplexer	-	-	-	174
Register	-	-	558	-
Total	33	3	892	1760
Available	120	80	35200	17600
Utilization (%)	27	3	2	10

Για το Hardware απαιτούνται συνολικά **683.780** κύκλοι με το estimation.

Για τα Resource utilization estimates:

- DSP → Χρησιμοποιούνται 3 από τα 80 διαθέσιμα, με ποσοστό χρησιμοποίησης (3.75%).
  η χρήση των DSP παραμένει πολύ χαμηλή, με μόνο 3 από τα 80 διαθέσιμα DSP να αξιοποιούνται
- BRAM → Χρησιμοποιούνται 16 από τα 60 διαθέσιμα, με ποσοστό χρησιμοποίησης (26.67%).
- LUT → Χρησιμοποιούνται 1760 από τις 17,600 διαθέσιμες (10%).
- FF → Χρησιμοποιούνται 892 από τις 35,200 διαθέσιμες (2.53%).

Loop

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

Παρατηρώντας τον πίνακα των Loops, τα δύο loops με τη μεγαλύτερη Iteration Latency, Trip Count και συνολικό Latency είναι τα layer_1 και layer_3, λόγω της μεγάλης υπολογιστικής τους πολυπλοκότητας. Το layer_1 εκτελεί 11,760 επαναλήψεις (392*30) με πολλαπλασιασμούς και προσθέσεις, ενώ το layer_3 εκτελεί 19,600 επαναλήψεις (50*392). Στο layer_1 χρησιμοποιούμε σαν activation function την συνάρτηση ReLU() η οποία είναι πιό “ελαφριά” από την tanh() που χρησιμοποιείται στο layer_3. Συνεπώς το layer_3 είναι το πιό “βαρύ” υπολογιστικά. Τα τόσο υψηλά latency ιδίως σε αυτά τα 2 layer είναι αναμενόμενο.

B)
Μετά την ολοκλήρωση της δημιουργίας του bitstream και την εγγραφή του στην κάρτα SD, μεταφέρθηκε στο Zybo για εκτέλεση.

Αποτελέσματα που προκύπτουν από το zybo (κανένα optimization):

```
sh-4.3# ./lab4.elf
Starting dataset parsing...
Parsing finished...
Starting hardware calculations...
Hardware calculations finished.
Starting software calculations...
Software calculations finished.
Hardware cycles : 682918
Software cycles : 1475422
Speed-Up        : 2.16047
Saving results to output.txt...
```

Χωρίς καμία βελτιστοποίηση, στο zybo η εφαρμογή τρέχει σε **682.918 HW CC** με **speedup=2.16**
Το estimation (683.780 CC) ήταν αρκετά κοντά με την πραγματική εκτέλεση στο Zybo (682.918 CC).

Γ)

Σε αυτό το σημείο καλούμαστε να κάνουμε design space exploration για να βρούμε τις βελτιστοποιήσεις που θα επιταχύνουν σημαντικά τον αλγόριθμο. Θα πρέπει να δοκιμάσουμε διάφορα HLS pragmas και να εξετάσουμε τα αποτελέσματα του estimation που θα προκύψουν από την εφαρμογή τους. Μέσω αυτής της διαδικασίας, θα εντοπίσουμε τις βέλτιστες στρατηγικές για τη βελτίωση της απόδοσης του αλγορίθμου, με στόχο τη μείωση του χρόνου εκτέλεσης και την αύξηση της ταχύτητας.
Για να το πετύχουμε αυτό πρέπει να μειώσουμε το Iteration Latency, το Trip Count και το συνολικό Latency από τα Loops, και ειδικότερα από τα Layer 1 και Layer 3.

1η Βελτίωση (Full Pipelining) **speedup=120**

Κάνοντας **Full Pipelining** σε όλα τα loops, παρατηρούμε ότι η εκτέλεση επιταχύνεται σημαντικά. Συγκεκριμένα, από 683.780 CC στην εκτίμηση (estimation), το αποτέλεσμα πέφτει στα 12.031 CC. Στο Zybo, η εφαρμογή τρέχει σε 12.277 CC με **speedup= 120.173** γεγονός που δείχνει μια εξαιρετική βελτίωση στην ταχύτητα εκτέλεσης. Αυτή η σημαντική επιτάχυνση οφείλεται στην αποτελεσματική εφαρμογή του full pipelining σε όλα τα loops, το οποίο επιτρέπει την παράλληλη εκτέλεση των επαναλήψεων και μειώνει το latency.

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

HW accelerated (Estimated cycles)	12031

Resource utilization estimates for HW functions

Resource	Used	Total	% Utilization
DSP	80	80	100
BRAM	40	60	66,67
LUT	6354	17600	36,1
FF	10330	35200	29,35

```
sh-4.3# ./lab4.elf
Starting dataset parsing...
Parsing finished...
Starting hardware calculations...
Hardware calculations finished.
Starting software calculations...
Software calculations finished.
Hardware cycles  : 12277
Software cycles  : 1475362
Speed-Up         : 120.173
Saving results to output.txt...
```

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- read_input	394	394	4	1	1	392	yes
- layer_1	393	393	3	1	1	392	yes
- layer_1_act	30	30	1	1	1	30	yes
- layer_2	52	52	4	1	1	50	yes
- layer_3	400	400	10	1	1	392	yes

Όπως φαίνεται από το screenshot του Loop πετυχαίνουμε II=1 με το pipeline.

2η Βελτίωση (Pipelining + Loop Unrolling + Array Partitioning) **speedup=122**

Προσθέσαμε **Unroll** στα εσωτερικά loops, με factor που διαιρείται τέλεια με το μέγεθος κάθε loop και προσθέσαμε **Array Partitioning** για τους πίνακες xbuf[ ], layer_1_out[ ], layer_2_out[ ], παρατηρούμε μια μικρή βελτίωση στην απόδοση. Αυτή η βελτίωση είναι διακριτή στο estimation, καθώς τα αποτελέσματα είναι 11.823 CC, καλύτερη από το full pipelining. Στο Zybo, η εφαρμογή τρέχει σε 12.070 CC με **speedup = 122.25**.

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

HW accelerated (Estimated cycles)	11823

Resource utilization estimates for HW functions

Resource	Used	Total	% Utilization
DSP	80	80	100
BRAM	40	60	66,67
LUT	7358	17600	41,81
FF	9593	35200	27,25

```
sh-4.3# ./embedded_lab4_ex1.elf
Starting dataset parsing...
Parsing finished...
Starting hardware calculations...
Hardware calculations finished.
Starting software calculations...
Software calculations finished.
Hardware cycles : 12070
Software cycles : 1475560
Speed-Up        : 122.25
Saving results to output.txt...
sh-4.3#
```

Τελική Υλοποίηση **speedup=137**

Δοκιμάσαμε πολλές παραλλαγές και συνδυασμούς, εκ των οποίων αρκετές ήταν αποτυχημένες καθώς βγαίναμε συνήθως εκτός πόρων. Ορισμένες αλλαγές δεν άλλαζαν καθόλου το estimation, κάτι που οφείλεται στη δέσμευση που έχουμε θέσει στους DSP πόρους (το zybo έχει διαθέσιμα 80 DSP). Για παράδειγμα στο layer3 οποιοδήποτε unrolling μαγαλύτερο του 10 δεν έριχνε παραπάνω το latency. Δεν υπήρχαν διαθέσιμοι πόροι για να υποστηρίξουν ο,τι αλλαγή δοκιμάζαμε.

Έχουμε κάνει αλλαγές μόνο στη συνάρτηση `void forward_propagation(float *x, float *y){}` από το αρχείο network.cpp, στο network.h δεν έχει γίνει καμία αλλαγή.

Η υλοποίηση που μας έδωσε 137 speedup είναι εξής:

```
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];

#pragma HLS ARRAY_PARTITION variable=layer_1_out block factor=30 dim=1
#pragma HLS ARRAY_PARTITION variable=layer_2_out block factor=50 dim=1

        //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 II=1
                xbuf[i] = x[i];
        }

// Layer 1
        layer_1:
```

```
        for(int i=0; i<N1; i++)
        {
#pragma HLS PIPELINE II=1
#pragma HLS unroll factor=8
                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 unroll factor=30//itan30
                layer_1_out[i] = ReLU(layer_1_out[i]);
        }

// Layer 2
        layer_2:
        for(int i=0; i<M2; i++)
        {
#pragma HLS PIPELINE II=1
                l_quantized_type result = 0;
                for(int j=0; j<N2; j++)
                {
#pragma HLS PIPELINE II=1
                        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 II=1
                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();
        }
}
```

Το estimation είναι :

Details

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

HW accelerated (Estimated cycles)	10556

Resource utilization estimates for HW functions

Resource	Used	Total	% Utilization
DSP	80	80	100
BRAM	41	60	68,33
LUT	12226	17600	69,47
FF	9081	35200	25,8

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	6	4	1	49	yes
- layer_2	52	52	4	1	1	50	yes
- layer_3	400	400	10	1	1	392	yes

Για το Hardware απαιτούνται συνολικά **10.556** κύκλοι με το estimation.

Για τα Resource utilization estimates:

- DSP → Χρησιμοποιούνται όλα τα 80 διαθέσιμα
- BRAM → Χρησιμοποιούνται 41 από τα 60 διαθέσιμα, με ποσοστό χρησιμοποίησης (68.33%).
- LUT → Χρησιμοποιούνται 12226 από τις 17,600 διαθέσιμες (69.5%).
- FF → Χρησιμοποιούνται 9081 από τις 35,200 διαθέσιμες (25.3%).

Παρατηρούμε ότι έχουμε μεγαλύτερη δέσμευση των διαθέσιμων πόρων του zybo σε σχέση με τις προηγούμενες υλοποιήσεις. Αυτή η αύξηση είναι λογική, καθώς ο συνδυασμός όλων των #pragmas απαιτεί περισσότερους πόρους για να επιτευχθεί καλύτερη απόδοση.

Αποτελέσματα που προκύπτουν από το zybo (best optimization):

```
sh-4.3# ./lab4.elf
Starting dataset parsing...
Parsing finished...
Starting hardware calculations...
Hardware calculations finished.
Starting software calculations...
Software calculations finished.
Hardware cycles : 10770
Software cycles : 1476009
Speed-Up        : 137.048
Saving results to output.txt...
sh-4.3#
```

Στο zybo η εφαρμογή τρέχει σε **10.770 HW CC** με **speedup=137.048**
Το estimation (10.556 CC) ήταν αρκετά κοντά με την πραγματική εκτέλεση στο Zybo (10.770 CC).
Η τελική υλοποίηση μας έχει οδηγήσει σε σχεδον 63.5 φορές μεγαλύτερο speedup και αντίστοιχα Hardware cycles σε σχέση με την μη βελτιστοποιημένη εκτέλεση **.**

Δ)

Μέσω του HLS Report μπορούμε να δούμε οτι για τα loop καταφέραμε να μειώσουμε τα latency και τα trip count.

read input : Latency 1.568 → 395
Iteration Latency 4→ 5
Trip count 392 → 392

layer 1 : Latency 36.456 → 197
Iteration Latency 93→ 6
Trip count 392 → 49

layer 2 : Latency 4.600 → 52
Iteration Latency 92→ 4
Trip count 50 → 50

layer 3 : Latency 61.936 → 400
Iteration Latency 158→10
Trip count 392 → 392

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	6	4	1	49	yes
- layer_2	52	52	4	1	1	50	yes
- layer_3	400	400	10	1	1	392	yes

Η μεγαλύτερη αλλαγή έγινε στο latency του layer_1. Δοκιμάσαμε με πολλούς διαφορετικούς τρόπους να πετύχουμε παρόμοια μείωση και στο latency του layer_3, αλλά δυστυχώς δεν καταφέραμε ποτέ να το μειώσουμε κάτω από το 400. Στον παραπάνω κώδικα έχουμε βάλει unrolling με factor=50, για να γίνει unroll όλο το εσωτερικό loop. Ουσιαστικά όμως, ποτέ δεν πετυχαίνουμε κάτι τέτοιο, αφού λαμβάναμε ακριβώς ίδιο estimation και για factor=10, που σημαίνει ότι η αύξηση του παραλληλισμού περεταίρω έφτανε σε κορεσμό. Υποθέτουμε πως εξαντλούσαμε όλα τα DSP, με αποτέλεσμα να μην υπάρχουν επιπλέον διαθέσιμοι πολλαπλασιαστές για να γίνουν κι άλλοι πολλαπλασιασμοί παράλληλα.

Με τις αλλαγές που γίνονταν στο factor του unrolling, γίνονταν και αντίστοιχες αλλαγές στο partitioning των arrays για να κάνουν match τα blocks με το unrolling.

Σαν τελική ιδέα για την βελτιστοποίηση του speedup ήταν η αλλαγή του activation function tanh() σε κάποια λιγότερο απαιτητική υπολογιστικά που μας δίνει πάλι στο output αποτελέσματα στο [-1,1], όπως η arctan() ή softsign(). Αυτη η ιδέα εφαρμόστηκε μόνο σε estimation και μείωσε τους κύκλους, αλλά δεν εφαρμόστηκε στο zybo γιατι θεωρήσαμε ότι βγαίνει εκτός του ζητούμενου της άσκησης.

Για το Layer 1 παρατηρούμε ότι δεν έχει επιτευχθεί initiation achieved 1 , με αλλαγές καταφέραμε να το πετύχουμε με unroll factor 2 για το εξωτερικο loop στο layer 1 αλλά αυτό έχει μια μικρή επίπτωση στο speedup ,οποτε προτιμήσαμε να διατηρήσουμε υψηλό το speedup και να έχουμε initiation achieved 4.

	BRAM	DSP	FF	LUT	Bits P0	Bits P1	Bits P2	Banks/Depth	Words	W*Bits*Banks
forward_propagation	82	80	9081	12113						
I/O Ports(2)					64					
Instances(2)	0	0	268	677						
Memories(112)	82		273	232	1016			112	34276	310594
Expressions(870)	0	80	0	8437	9364	9241	1156			
-	0	0	0	78	16	78	0			
*	0	80	0	0	1131	1010	0			
+	0	0	0	6744	7771	7762	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	98	70	24	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(962)			8540		9383					
Channels(0)	0		0	0	0			0	0	0
Multiplexers(235)	0		0	2767	2762			0		

Στο Resource Profile λαμβάνουμε πληροφορίες σχετικά με την χρήση των πόρων του FPGA. Παρατηρούμε οτι στα είδη των μαθηματικών εκφράσεων του design αυτό που καταναλώνει όλα τα DSP είναι ο πολλαπλασιασμός, ενώ υπόλοιπες πράξεις (λογικές και αριθμητικές) αναλαμβάνουν τα LUT.

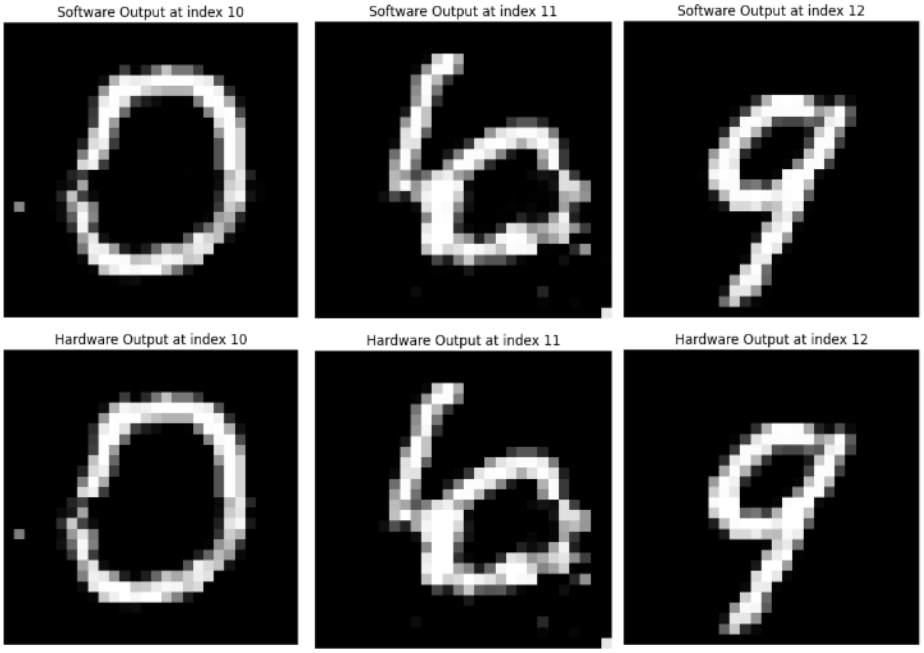
Γνωρίζουμε ότι ο πολλαπλασιασμός είναι μια υπολογιστικά απαιτητική πράξη. Με την προσθήκη των #pragmas επιθυμούμε παράλληλη εκτέλεση στο design μας και συνεπώς θέλουμε να γίνονται πολλοί πολλαπλασιασμοί παράλληλα για να αξιοποιηθούν όλα τα DSP.

Άσκηση 2. Quality measurement

Α)

Τρέξαμε τα source files για τον αλγόριθμο του generator ,στην συνέχεια μεταφέραμε το output.txt τρέξαμε το plot_output.ipynb από google collab. Παρακάτω παραθέτουμε τα αποτελέσματα που προκύπτουν για τους δεκαδικούς 0, 6 και 9 (idx: 10, 11, 12).

Για 8 bits ακρίβεια



Οι εικόνες του hardware και του software είναι πολύ παρόμοιες και η διαφορά δεν είναι αντιληπτή με το μάτι.Την καλύτερη απεικόνιση την έχει ο αριθμός 9 .Αυτό επιβεβαιώνεται και από το Max Pixel Error, το οποίο για 8 bits είναι πολύ χαμηλό.

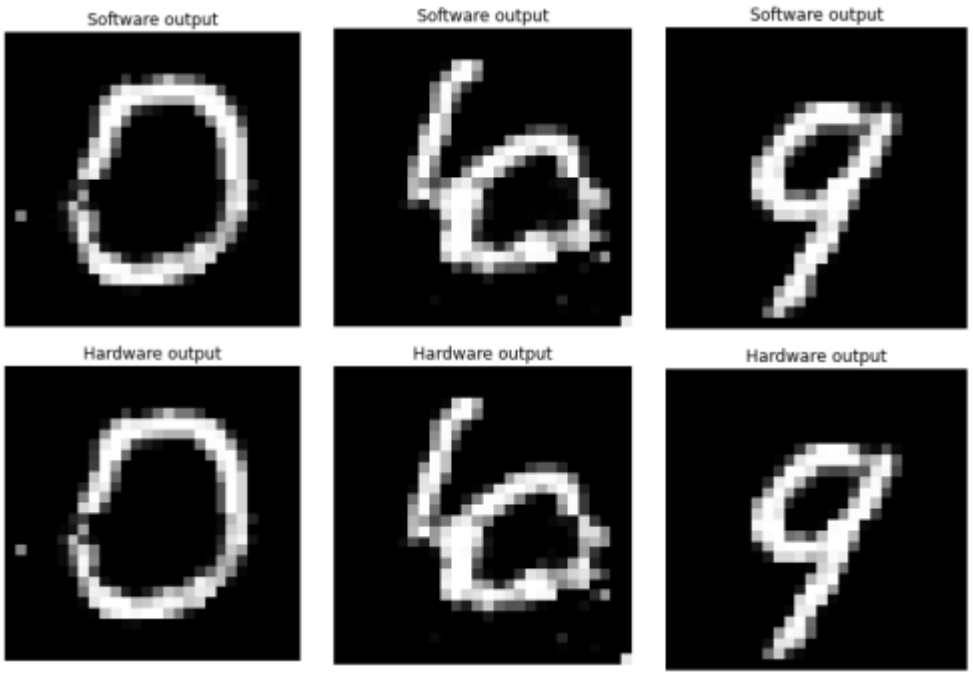
Index 10:
    Max pixel error: 16
    Peak Signal-to-Noise Ratio: 42.68
Index 11:
    Max pixel error: 17
    Peak Signal-to-Noise Ratio: 42.56
Index 12:
    Max pixel error: 13
    Peak Signal-to-Noise Ratio: 47.06

Max Pixel Error: Σχετικά χαμηλό, υποδεικνύει καλή ποιότητα αναπαραγωγής.
PSNR: Υψηλό πράγμα που δείχνει καλή ποιότητα εικόνας με μικρές διαφορές μεταξύ hardware και software.

Β)

Στην καλύτερη υλοποίηση που πετύχαμε χρησιμοποιούσαμε custom datatypes που έχουν 8bits δεκαδική ακρίβεια. Στην συνέχεια παράγουμε νέα bitstream αλλάζοντας μόνο την συγκεκριμένη ακρίβεια σε 4, 6, 10 bits και αποθηκεύουμε τα αντίστοιχα output που προκύπτουν, ώστε να τα φορτώσουμε στο notebook.

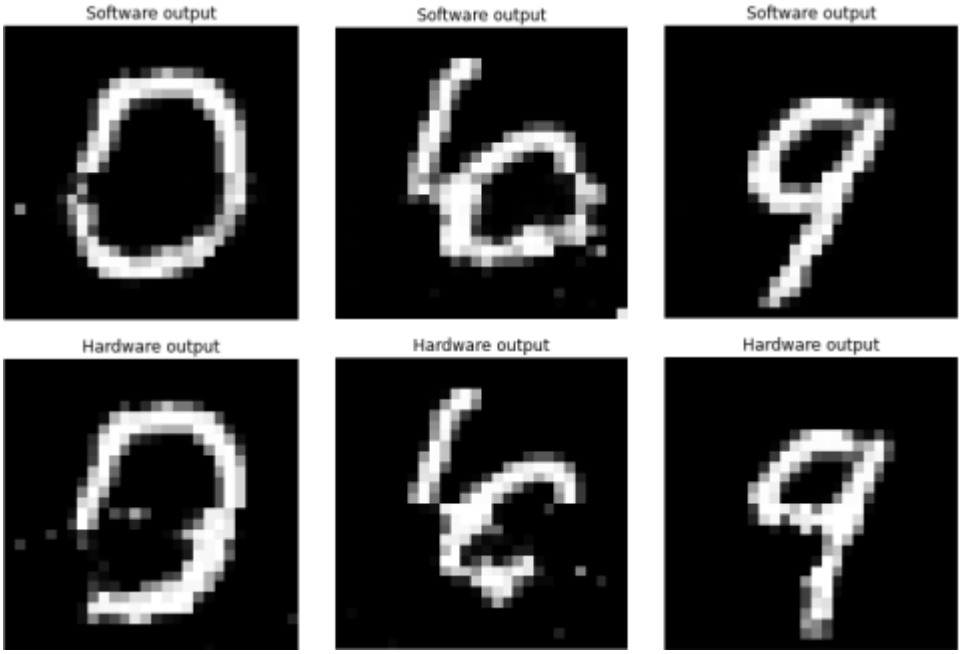
Για 10 bits



Index 10:
    Max pixel error: 5
    Peak Signal-to-Noise Ratio: 54.08
Index 11:
    Max pixel error: 5
    Peak Signal-to-Noise Ratio: 52.55
Index 12:
    Max pixel error: 4
    Peak Signal-to-Noise Ratio: 53.76

Max Pixel Error: Πολύ χαμηλό δείχνει εξαιρετική ακρίβεια.
PSNR: Πολύ υψηλό πράγμα που σημαίνει ότι η ποιότητα είναι πολύ καλή και οι διαφορές είναι ελάχιστες.

Για 4 bits



Index 10:
Max pixel error: 255
Peak Signal-to-Noise Ratio: 13.87

Index 11:
Max pixel error: 249
Peak Signal-to-Noise Ratio: 12.78

Index 12:
Max pixel error: 255
Peak Signal-to-Noise Ratio: 13.48

Max Pixel Error: Πολύ υψηλό, φτάνει το 255, δείχνει ότι υπάρχουν μεγάλες διαφορές στην αναπαραγωγή των εικόνων.
PSNR: Πολύ χαμηλό, υποδηλώνει ότι οι εικόνες δεν είναι καλής ποιότητας και εχουν πολλές παραμορφώσεις .Το οποίο επιβεβαιώνεται και οπτικά.

Γ)
Συμπέρασμα

Η ιδανική ακρίβεια bit για τα custom datatypes είναι όπως θα περιμέναμε τα 10bits, διότι υπάρχει μεγαλύτερη ακρίβεια στα βάρη που χρησιμοποιούμε σε κάθε layer. Επιπλέον, αφού τα τρέξουμε στο colab βλέπουμε ότι στα 10bits έχουμε το μεγαλύτερο PSNR και τα λιγότερα errors σε σχέση με τα υπόλοιπα bit (4, 8) για τα αντίστοιχα indexes. Η μετρική που προτιμάμε είναι το PSNR μπορεί να μας δώσει περισσότερη πληροφορία για όλη την εικόνα. Το max pixel error δεν δίνει σημαντική πληροφορία για την εικόνα αλλά για τη μέγιστο λάθος που αφορά μόνο ένα pixel. Το PSNR λαμβάνει υπόψη όλα τα pixel, αφού υπολογίζεται η μέση τιμή και έτσι επιτυγχάνει μια συνολική εκτίμηση της ποιότητας ανακατασκευής.