



ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ
ΥΠΟΛΟΓΙΣΤΩΝ

Σχεδίαση Συστημάτων Υλικού - Λογισμικού

Εργαστήριο 3: Εξοικείωση με το εργαλείο Vitis

Από τους φοιτητές:
Σωτήριος Καραντζούλης 10952
Κωνσταντίνος Γκαρίπης 10924

Διδάσκων: Ιωάννης Παπαευσταθίου
Υπεύθυνοι Εργαστηρίου: Α. Αθανασιάδης - Δ. Καρανάσος

January 12, 2026

Περιεχόμενα

1	Εισαγωγή	3
2	Kernel	3
2.1	Source code	3
2.2	Λειτουργικότητα	5
2.3	Βελτιστοποίηση	6
2.3.1	Axi Interface	6
2.3.2	Array partition	7
2.3.3	Dataflow	7
2.3.4	Λοιπές βελτιστοποιήσεις	9
3	Αποτελέσματα	9
3.1	Δεδομένα	9
3.2	Speedup	9

1 Εισαγωγή

Σκοπός του εργαστήριου 3 ήταν η υλοποίηση της προηγούμενης λειτουργικότητας του kernel αλλά με την πλήρη αξιοποίηση του axi bus. Πιο συγκεκριμένα, στα προηγούμενα εργαστήρια ο kernel έκανε προσβάσεις στην μνήμη για κάθε pixel που χρειαζόταν. Ωστόσο, το bus του πρωτοκόλου axi επιτρέπει την μεταφορά έως και 512 bit σε ένα access στην RAM. Έτσι, ο kernel μπορεί να επεξεργάζεται περισσότερα pixel, μειώνοντας παράλληλα τα χρονικά κοστοβόρα accesses στην μνήμη.

Επιπλέον επιτάχυνση επιτυγχάνεται και με την χρήση διαφορετικών bundles στο axi για την επίτευξη παράλληλης ανάγνωσης και εγγραφής από και προς την μνήμη.

2 Kernel

2.1 Source code

```
1 #include <stdint.h>
2 #include <ap_int.h>
3 #include <hls_stream.h>
4
5 // Original Image
6 #define WIDTH 128
7 #define HEIGHT 128
8
9 // Transaction Definition
10 #define PIXEL_SIZE 8
11 #if WIDTH <= 64
12     #define AXI_WIDTH_BITS WIDTH*PIXEL_SIZE // Data width of Memory Access in bits per cycle
13 #else
14     #define AXI_WIDTH_BITS 512 // Data width of Memory Access in bits per cycle
15 #endif
16 #define AXI_WIDTH_BYTES (AXI_WIDTH_BITS / PIXEL_SIZE)
17
18 #if WIDTH <= 64
19     #define BUFFER_WIDTH_CHUNKS 1
20 #else
21     #define BUFFER_WIDTH_CHUNKS 2
22 #endif
23 #define BUFFER_WIDTH_BYTES (BUFFER_WIDTH_CHUNKS * AXI_WIDTH_BYTES)
24 #define V_LIMIT 256
25
26 const unsigned int h_steps = (WIDTH - BUFFER_WIDTH_BYTES) / (AXI_WIDTH_BYTES) + 1;
27
28 // Type Definitions
29 typedef ap_uint<AXI_WIDTH_BITS> uint512_dt;
30 typedef ap_uint<PIXEL_SIZE> pixel_t;
31
32 // Helper Functions
33 pixel_t Compare(pixel_t A, pixel_t B);
34
35 const unsigned int c_size = BUFFER_WIDTH_BYTES;
36 const unsigned int c_len = HEIGHT*WIDTH / c_size;
37
38 // MAIN
39 extern "C" {
40     void IMAGE_DIFF_POSTERIZE(const uint512_dt *in_A, const uint512_dt *in_B, uint512_dt *out)
41     {
42         // INTERFACE DIRECTIVES
43         #pragma HLS INTERFACE m_axi port = in_A offset = slave bundle = gmem0
44         #pragma HLS INTERFACE m_axi port = in_B offset = slave bundle = gmem1
45         #pragma HLS INTERFACE m_axi port = out offset = slave bundle = gmem2
46         #pragma HLS INTERFACE s_axilite port = in_A bundle = control
47         #pragma HLS INTERFACE s_axilite port = in_B bundle = control
48         #pragma HLS INTERFACE s_axilite port = out bundle = control
49         #pragma HLS INTERFACE s_axilite port = return bundle = control
50
51         // Local Buffers
52         pixel_t Prior_chunk_1[BUFFER_WIDTH_BYTES];
53         pixel_t Prior_chunk_2[BUFFER_WIDTH_BYTES];
54         pixel_t Filtered_chunk[BUFFER_WIDTH_BYTES];
55         pixel_t inter_pixels[2][V_LIMIT];
56
57         #pragma HLS ARRAY_PARTITION variable=inter_pixels complete dim=1
58         #pragma HLS ARRAY_PARTITION variable=Prior_chunk_1 complete
59         #pragma HLS ARRAY_PARTITION variable=Prior_chunk_2 complete
60         #pragma HLS ARRAY_PARTITION variable=Filtered_chunk complete
61         #pragma HLS ARRAY_PARTITION variable=inter_pixels complete
62
63         // Stream Declaration
64         hls::stream<uint512_dt> stream_G;
65
66         // Calculate number of vertical steps
```

```

68     const int v_steps = (HEIGHT % V_LIMIT) ? (HEIGHT / V_LIMIT) + 1 : (HEIGHT / V_LIMIT);
69
70     for (int v_step = 0; v_step < v_steps; v_step++){
71
72         // Handle row range for this vertical step
73         int ref_row = (v_step == 0) ? 0 : v_step * V_LIMIT - 2;
74         unsigned int last_row = (v_step + 1) * V_LIMIT;
75
76         // Clamp to image height
77         if (last_row > HEIGHT) last_row = HEIGHT;
78
79         // Special case for single step to include last row and start from 0
80         if (v_steps == 1) {
81             last_row = HEIGHT;
82             ref_row = 0;
83         }
84
85
86         // Shift the buffer horizontally
87         for (int h_step = 0; h_step < h_steps; h_step++) {
88
89             // Stream to connect Stage 1 of Comparison and Stage 2 of filtering
90             #pragma HLS DATAFLOW
91             #pragma HLS STREAM variable=stream_G depth = 100
92
93             int curr_buf = h_step % 2;
94             int prev_buf = 1 - curr_buf;
95
96             // Loop the buffer and compare over all image rows
97             for (int row = ref_row; row < last_row; row++) {
98                 unsigned int href_point = h_step * AXI_WIDTH_BYTES;
99                 unsigned int ref_point = (row*WIDTH + href_point)/AXI_WIDTH_BYTES;
100
101
102                 for (int i = 0; i < BUFFER_WIDTH_CHUNKS; i++) {
103                     #pragma HLS PIPELINE II=1
104
105                     uint512_dt val1 = in_A[ref_point + i];
106                     uint512_dt val2 = in_B[ref_point + i];
107                     uint512_dt res_G;
108
109                     // Compute Compare (Posteriorize) on all AXI PIXELS (64 elements) in parallel
110                     for (int v = 0; v < AXI_WIDTH_BYTES; v++) {
111                         #pragma HLS UNROLL
112
113                         // Unpack
114                         pixel_t p1 = val1.range(PIXEL_SIZE * (v + 1) - 1, v * PIXEL_SIZE);
115                         pixel_t p2 = val2.range(PIXEL_SIZE * (v + 1) - 1, v * PIXEL_SIZE);
116
117                         // Compare
118                         pixel_t g_val = Compare(p1, p2);
119
120                         // Re-pack
121                         res_G.range(PIXEL_SIZE * (v + 1) - 1, v * PIXEL_SIZE) = g_val;
122                     }
123
124                     // Push G result to the stream for the next stage
125                     stream_G.write(res_G);
126                 }
127             }
128
129             for(int k=0; k<BUFFER_WIDTH_BYTES; k++) {
130                 #pragma HLS UNROLL
131                 Prior_chunk_1[k] = 0;
132                 Prior_chunk_2[k] = 0;
133             }
134
135             // Filter the buffer over rows and output
136             for (int row = ref_row; row < last_row; row++) {
137
138                 // Declaring here for hinting a temporary storage
139                 pixel_t Current_chunk[BUFFER_WIDTH_BYTES];
140                 #pragma HLS ARRAY_PARTITION variable=Current_chunk complete
141
142                 unsigned int href_point = h_step * AXI_WIDTH_BYTES;
143
144                 // Row chunk unpacking
145                 for (int chunk = 0; chunk < BUFFER_WIDTH_CHUNKS; chunk++) {
146                     #pragma HLS PIPELINE II=1
147
148                     uint512_dt temp_val = stream_G.read();
149
150                     for (int v = 0; v < AXI_WIDTH_BYTES; v++) {
151                         #pragma HLS UNROLL
152                         Current_chunk[chunk*AXI_WIDTH_BYTES + v] = temp_val.range(PIXEL_SIZE * (v + 1) - 1, v *
PIXEL_SIZE);
153                     }
154                 }
155
156                 if (row >= ref_row + 2) { // All lines available, normal operation
157
158                     // Filtering Logic
159                     if(href_point == 0){
160                         // First pixel of the chunk at the left border
161                         Filtered_chunk[0] = 0;
162                     }
163                     else{
164                         if(WIDTH >128)
165                             Filtered_chunk[0] = inter_pixels[prev_buf][row - 2 - ref_row];

```

```

166     }
167
168     for (int col = 1; col < BUFFER_WIDTH_BYTES; col++) {
169         #pragma HLS UNROLL
170
171         if (col == BUFFER_WIDTH_BYTES - 1){
172             Filtered_chunk[col] = 0;
173         }
174         else {
175             int16_t temp_filter = 5 * Prior_chunk_1[col]           // Center pixel
176                             - Current_chunk[col]               // Down pixel
177                             - Prior_chunk_2[col]               // Up pixel
178                             - Prior_chunk_1[col - 1]           // Left pixel
179                             - Prior_chunk_1[col + 1];           // Right pixel
180
181             // Clamping the result to [0, 255]
182             pixel_t filtered_pixel = (pixel_t) (temp_filter < 0 ? 0 : (temp_filter > 255 ? 255 :
temp_filter));
183
184             // Write back to output buffer
185             Filtered_chunk[col] = filtered_pixel;
186         }
187     }
188
189     // Middle pixel of the chunk
190     if(WIDTH > 128)
191         inter_pixels[curr_buf][row - 2 - ref_row] = Filtered_chunk[BUFFER_WIDTH_BYTES/
BUFFER_WIDTH_CHUNKS];
192
193     // Write the filtered chunk to output stream
194     uint512_dt out_val;
195     unsigned int write_row_offset = ((row - 1) * WIDTH + href_point) / AXI_WIDTH_BYTES;
196
197     for (int chunk = 0; chunk < BUFFER_WIDTH_CHUNKS; chunk++) {
198         #pragma HLS PIPELINE II=1
199         // Output packing
200         for (int v = 0; v < AXI_WIDTH_BYTES; v++) {
201             #pragma HLS UNROLL
202             out_val.range(PIXEL_SIZE * (v + 1) - 1, v * PIXEL_SIZE) = Filtered_chunk[chunk*
AXI_WIDTH_BYTES + v];
203         }
204
205         out[write_row_offset + chunk] = out_val;
206     }
207 }
208
209 // Refresh the Chunks
210 for (int v = 0; v < BUFFER_WIDTH_BYTES; v++) {
211     #pragma HLS UNROLL
212     Prior_chunk_2[v] = Prior_chunk_1[v];
213     Prior_chunk_1[v] = Current_chunk[v];
214 }
215 }
216 }
217 }
218
219 // Make the first and last row zeros
220 for(int chunk=0; chunk < WIDTH/AXI_WIDTH_BYTES; chunk++){
221     out[chunk] = 0;
222     out[(HEIGHT-1)*(WIDTH/AXI_WIDTH_BYTES) + chunk] = 0;
223 }
224 }
225 }
226
227
228 /* Compare Helper Function
229 - Input : 2 uint8_t numbers
230 - Output : Quantized absolute difference
231 */
232 pixel_t Compare(pixel_t A, pixel_t B){
233     pixel_t C;
234     int16_t temp_d = (int16_t) A - (int16_t) B;
235
236     constexpr uint8_t T1 = 32;
237     constexpr uint8_t T2 = 96;
238
239     // Note: Since temp_d is unsigned (uint8_t), it can never be < 0.
240     // This logic performs a modular subtraction check.
241     uint8_t D = (temp_d < 0) ? -temp_d : temp_d;
242
243     if(D < T1) C = (pixel_t) 0;
244     else if(D < T2) C = (pixel_t) 128;
245     else C = (pixel_t) 255;
246
247     return C;
248 }

```

Listing 1: Source code του Kernel

2.2 Λειτουργικότητα

Η βασική λειτουργικότητα αυτού του kernel είναι πως ζητάει από τον host πακέτα των 512 bit (64 pixel) ανά γραμμή εικόνας, τα επεξεργάζεται και έπειτα τα αποθηκεύει πίσω στην

RAM. Πιο συγκεκριμένα, ο kernel θα ζητήσει BUFFER_WIDTH_CHUNKS πακέτα των 512 bit από τις πρώτες γραμμές των A και B εικόνων, θα εφαρμόσει την Compare απευθείας στην λήψη τους και όταν έχει αρκετές γραμμές (τουλάχιστον 3) έτοιμες θα εφαρμόσει και το φίλτρο. Έπειτα οι δύο πιο πρόσφατες γραμμές αποθηκεύονται και ζητείται η επόμενη κυλώντας έτσι κατα μήκος του ύψους (HEIGHT) του πίνακα. Αφότου παραλάβει και την τελευταία γραμμή μετακινεί το παράθυρο στην οριζόντια διάταξη (WIDTH) και η διαδικασία επαναλαμβάνεται για τις εναπομείνουσες στήλες.

Η Βασική δυσκολία της ανάπτυξης του κώδικα έγκειται στην απόκτηση και εγγραφή πάντοτε λέξεων των 512, απο και προς τις ευθυγραμμισμένες ανα 512 θέσεις μνήμης. Συγκεκριμένα, λόγω αυτού, γίνεται πολυπλοκότερη η εγγραφή και ανάγνωση των στοιχείων στα σύνορα της εικόνας, απόρροια της προϋπόθεσης που θέτει το φίλτρο για την γνώση των γειτονικών στοιχείων. Για να αντιμετωπιστεί το πρόβλημα το παράθυρο των 128 pixel μετατοπίζεται κάθε φορά κατα 64 pixel για να καλύψει τις ανάγκες του παλαιού δεξιού συνόρου. Η επιλογή αυτή σπαταλά χρόνο εκτέλεσης. Ακόμη κατά την εγγραφή, δεν υπάρχει η πληροφορία για το πρώτο στοιχείο (αριστερό σύνορο) και η οποία στο απλό AXI πρωτόκολλο δεν μπορεί να αγνοήσει το ήδη γραμμένο πρώτο πίξελ (γίνεται με το flag TKEEP στο AXIS) οπότε υπάρχει μνήμη που θυμάται το αριστερό σύνορο (inter_pixels). Και αυτή η επιλογή με την σειρά της οδηγεί σε μειονέκτημα utilisation. Να σημειωθεί ότι για την αποφυγή εξάρτησης απο το μέγεθος το πίνακα άρα πιθανού overutilisation, ο κώδικας επιτρέπει το κατακόρυφο roll των cached τμημάτων γραμμών μέχρι ένα μέγιστο όριο V_LIMIT. Στην συνέχεια ολοκληρώνει τα οριζόντια βήματα και ξεκινά ξανά απο την γραμμή που σταμάτησε. Το όριο αυτό επιλέγεται αρκετά μεγάλο (256 στον συγκεκριμένο κώδικα) μιας και αφορά στην αποθήκευση μόνο μιας στήλης από πίξελ που δεν οδηγεί γρήγορα σε αυξημένη κατανάλωση πόρων.

Για διατηρηθεί η ίδια λογική, εάν το δοσμένο WIDTH είναι μικρότερο των 64 pixel τότε το AXI_WIDTH_BYTES γίνεται ίσο με το WIDTH. Έτσι με ένα beat του axi ο kernel λαμβάνει μία ολόκληρη γραμμή pixel. Τα μεγέθη αυτά είναι αρκετά μικρά ώστε να έχουμε ικανοποιητικό χρόνο εκτέλεσης και να μην τίθενται θέματα overutilisation.

2.3 Βελτιστοποίηση

Χρησιμοποιήθηκαν διάφορες τεχνικές με pragmas αλλά και με σωστή αξιοποίηση του bandwidth στο axi bus ώστε να επιτευχθεί ικανοποιητική βελτιστοποίηση.

2.3.1 Axi Interface

Οι είσοδοι in_A και in_B και η έξοδος out είναι πλέον τύπου uint512_dt ώστε ο kernel να αξιοποιεί πλήρως μέσω του AXI bus το μέγιστο throughput της DDR (512 bits/cycle). Έτσι τα accesses στην μνήμη μειώνονται δραματικά και άρα και το execution time.

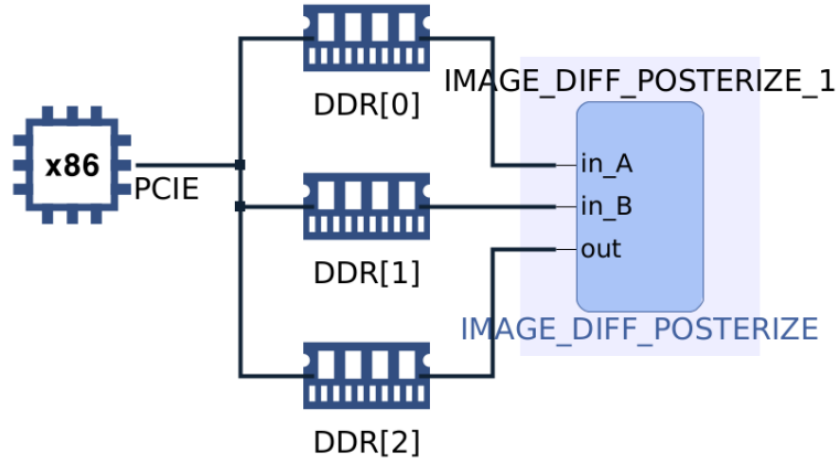
Επιπλέον τα in_A και in_B και out τοποθετήθηκαν σε διαφορετικά bundles ώστε να υπάρχει παράλληλη ανάγνωση και εγγραφή προς βέλτιστη εξυπηρέτηση του DATAFLOW.

```

1 void IMAGE_DIFF_POSTERIZE(const uint512_dt *in_A, const uint512_dt *in_B, uint512_dt *out, unsigned int size)
2 {
3     // INTERFACE DIRECTIVES
4     #pragma HLS INTERFACE m_axi port = in_A offset = slave bundle = gmem0
5     #pragma HLS INTERFACE m_axi port = in_B offset = slave bundle = gmem1
6     #pragma HLS INTERFACE m_axi port = out offset = slave bundle = gmem2
7     #pragma HLS INTERFACE s_axilite port = in_A bundle = control
8     #pragma HLS INTERFACE s_axilite port = in_B bundle = control
9     #pragma HLS INTERFACE s_axilite port = out bundle = control
10    #pragma HLS INTERFACE s_axilite port = return bundle = control
11
12    [...]

```

Listing 2: Axi Interface pragmas



Σχήμα 1: System Diagram

2.3.2 Array partition

Οι πίνακες που χρησιμοποιούνται για την αποθήκευση των πακέτων ανά γραμμή έγιναν complete partition ώστε να είναι εφικτή η εγγραφή στοιχείων σε οποιοδήποτε θέση ταυτόχρονα. Δόθηκε ιδιαίτερη προσοχή στην αποφυγή αποθήκευσης σε μπαuffer μεμονομένων pixel που θα προκαλούσε bottleneck στο stream των 512 bits/cycle. Έτσι περιορίζουμε την αποθήκευση pixel μόνο στην περίπτωση της υλοποίησης μνήμης για το φίλτρο όπου το complete partition εξασφαλίζει την σωστή ροή με ανάγνωση μεγάλων πακέτων άμεσα.

```

1  [...]
2
3  pixel_t Prior_chunk_1[BUFFER_WIDTH_BYTES];
4  pixel_t Prior_chunk_2[BUFFER_WIDTH_BYTES];
5  pixel_t Filtered_chunk[BUFFER_WIDTH_BYTES];
6  pixel_t inter_pixels[2][HEIGHT];
7
8  #pragma HLS ARRAY_PARTITION variable=inter_pixels complete dim=1
9  #pragma HLS ARRAY_PARTITION variable=Prior_chunk_1 complete
10 #pragma HLS ARRAY_PARTITION variable=Prior_chunk_2 complete
11 #pragma HLS ARRAY_PARTITION variable=Filtered_chunk complete
12 #pragma HLS ARRAY_PARTITION variable=inter_pixels complete
13
14  [...]

```

Listing 3: Array partition pragmas

2.3.3 Dataflow

Χρησιμοποιήθηκε το pragma Dataflow ώστε μόλις τα δεδομένα γίνουν διαθέσιμα από το στάδιο Compare να αξιοποιούνται απευθείας στο filtering στάδιο. Στάδια καλούμε τις διακριτές δομές επανάληψης που διαδέχονται η μία την άλλη σειριακά, μετά το πέρας των επαναλήψεων τους, απο την σκοπιά του software. Για αυτόν τον σκοπό, χρησιμοποιήθηκε και το pragma STREAM στην μεταβλητή stream_G ώστε να επικοινωνούν σωστά και παράλληλα στον χρόνο τα δύο στάδια.

Αυτή η μέθοδος επιτυγχάνει δραματική επιτάχυνση στο execution time καθώς τα στάδια loading, επεξεργασίας και output είναι "pipelined", οδηγώντας τα δεδομένα σε μία συνεχή ροή (stream) απο την απόκτηση έως την επιστροφή τους στην DDR.

```

1  [...]
2
3  // Loop the buffer and compare over all image rows
4  for (int row = 0; row < HEIGHT; row++) {
5      unsigned int href_point = h_step * AXI_WIDTH_BYTES;
6      unsigned int ref_point = (row*WIDTH + href_point)/AXI_WIDTH_BYTES;

```

```

7
8
9      for (int i = 0; i < BUFFER_WIDTH_CHUNKS; i++) {
10         #pragma HLS PIPELINE II=1
11
12         uint512_dt val1 = in_A[ref_point + i];
13         uint512_dt val2 = in_B[ref_point + i];
14         uint512_dt res_G;
15
16         // Compute G(in1, in2) on all AXI PIXELS (64 elements) in parallel
17         for (int v = 0; v < AXI_WIDTH_BYTES; v++) {
18             #pragma HLS UNROLL
19
20             // Unpack
21             pixel_t p1 = val1.range(PIXEL_SIZE * (v + 1) - 1, v * PIXEL_SIZE);
22             pixel_t p2 = val2.range(PIXEL_SIZE * (v + 1) - 1, v * PIXEL_SIZE);
23
24             // Compare
25             pixel_t g_val = Compare(p1, p2);
26
27             // Re-pack
28             res_G.range(PIXEL_SIZE * (v + 1) - 1, v * PIXEL_SIZE) = g_val;
29         }
30
31         // Push G result to the stream for the next stage
32         stream_G[row * BUFFER_WIDTH_CHUNKS + i] = res_G;
33     }
34 }
35
36 [...]

```

Listing 4: Compare stage

```

1      [...]
2      for (int row = 0; row < HEIGHT; row++) {
3
4          // Declaring here for hinting a temporary storage
5          pixel_t Current_chunk[BUFFER_WIDTH_BYTES];
6          #pragma HLS ARRAY_PARTITION variable=Current_chunk complete
7
8          unsigned int href_point = h_step * AXI_WIDTH_BYTES;
9          // Row chunk unpacking
10         for (int chunk = 0; chunk < BUFFER_WIDTH_CHUNKS; chunk++) {
11             #pragma HLS PIPELINE II=1
12             uint512_dt temp_val = stream_G[row * BUFFER_WIDTH_CHUNKS + chunk];
13             for (int v = 0; v < AXI_WIDTH_BYTES; v++) {
14                 #pragma HLS UNROLL
15                 Current_chunk[chunk*AXI_WIDTH_BYTES + v] = temp_val.range(PIXEL_SIZE * (v + 1) - 1, v *
PIXEL_SIZE);
16             }
17         }
18
19         if (row >= 2) { // All lines available, normal operation
20
21             // Filtering Logic
22             if (href_point == 0) {
23                 // First pixel of the chunk at the left border
24                 Filtered_chunk[0] = 0;
25             }
26             else {
27                 Filtered_chunk[0] = inter_pixels[prev_buf][row - 1];
28             }
29
30             for (int col = 1; col < BUFFER_WIDTH_BYTES; col++) {
31                 #pragma HLS UNROLL
32
33                 if (col == BUFFER_WIDTH_BYTES - 1) {
34                     Filtered_chunk[col] = 0;
35                 }
36                 else {
37                     int16_t temp_filter = 5 * Prior_chunk_1[col] // Center pixel
38                     - Current_chunk[col] // Down pixel
39                     - Prior_chunk_2[col] // Up pixel
40                     - Prior_chunk_1[col - 1] // Left pixel
41                     - Prior_chunk_1[col + 1]; // Right pixel
42
43                     // Clamping the result to [0, 255]
44                     pixel_t filtered_pixel = (pixel_t) (temp_filter < 0 ? 0 : (temp_filter > 255 ? 255 :
temp_filter));
45
46                     // Write back to output buffer
47                     Filtered_chunk[col] = filtered_pixel;
48                 }
49             }
50         }
51
52         // Middle pixel of the chunk
53         inter_pixels[curr_buf][row - 1] = Filtered_chunk[BUFFER_WIDTH_BYTES/BUFFER_WIDTH_CHUNKS];
54
55         // Write the filtered chunk to output stream
56         uint512_dt out_val;
57         unsigned int write_row_offset = ((row - 1) * WIDTH + href_point) / AXI_WIDTH_BYTES;
58
59         for (int chunk = 0; chunk < BUFFER_WIDTH_CHUNKS; chunk++) {
60             #pragma HLS PIPELINE II=1
61             // Output packing
62             for (int v = 0; v < AXI_WIDTH_BYTES; v++) {
63                 #pragma HLS UNROLL

```



```

64         out_val.range(PIXEL_SIZE * (v + 1) - 1, v * PIXEL_SIZE) = Filtered_chunk[chunk*
        AXI_WIDTH_BYTES + v];
65     }
66
67     out[write_row_offset + chunk] = out_val;
68     [...]

```

Listing 5: Filter stage

2.3.4 Λοιπές βελτιστοποιήσεις

Όπου ήταν εφικτό χρησιμοποιήθηκε loop unrolling ή pipeline ώστε οι εγγραφές σε πίνακες ή αναγνώσεις από τοπικούς πίνακες να επιταχυνθεί.

3 Αποτελέσματα

3.1 Δεδομένα

Μετρήθηκε το execution time για διάφορα μεγέθη πίνακα.

Πίνακας 1: Avg Execution Time (ms)

HEIGHT	WIDTH	Execution time
64	64	0.024
128	128	0.038
256	256	0.206
512	256	0.402

Επιπλέον βλέπουμε εμφανή βελτίωση στο transfer efficiency συγκριτικά με τα αποτελέσματα του προηγούμενου εργαστηρίου (1.563% σε όλες τις περιπτώσεις).

Πίνακας 2: Transfer Efficiency

HEIGHT	WIDTH	Transfer Efficiency (%)
64	64	1.563
128	128	3.077
256	256	3.109
512	256	3.125

3.2 Speedup

Για λόγους σύγκρισης παρατίθενται και οι μετρήσεις του προηγούμενου εργαστηρίου:

Πίνακας 3: Avg Execution Time previous kernel (ms)

HEIGHT	WIDTH	Single Bundle
64	64	0.103
128	128	0.377
256	256	1.432

Για τον πίνακα μεγέθους 128 πετυχαίνουμε speedup:

$$speedup = \frac{0.377}{0.038} = 9.92$$

Δηλαδή η ταχύτητα σχεδόν δεκαπλασιάστηκε.