Σχεδίαση Συστημάτων Υλικού - Λογισμικού

Εργαστήριο 2ο

Ονοματεπώνυμο	AEM	E-mail
Στασινός Αλκιβιάδης	9214	astasinos@ece.auth.gr
Κοσέογλου Σωκράτης	8837	sokrkose@ece.auth.gr

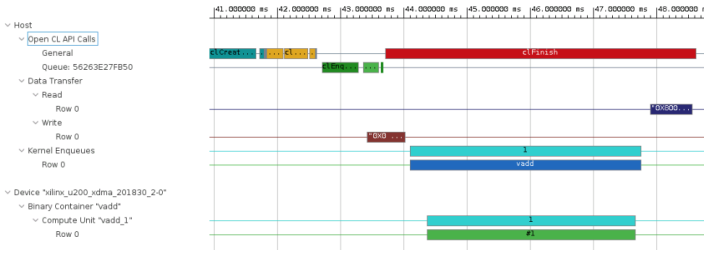
Ερώτημα 1.

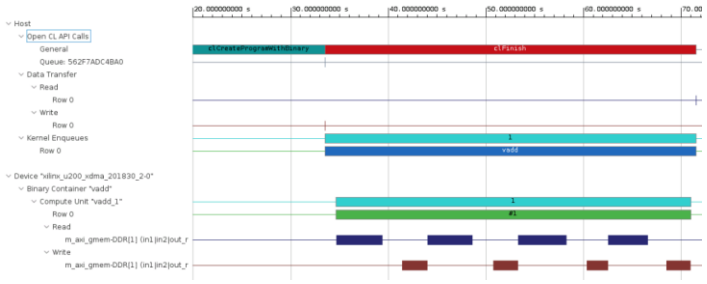
Αρχικά, εκτελέσαμε **Software Emulation** και **Hardware Emulation** στο Vitis το παράδειγμα **vadd** από τις έτοιμες βιβλιοθήκες της **Xilinx** έτσι ώστε να κατανοήσουμε καλύτερα το **Vitis**. Επίσης, έγινε μια μελέτη τόσο του κώδικα **host.cpp** που είναι γραμμένος σε γλώσσα **OpenCL**, όσο και του κώδικα **vadd.cpp** που είναι σε **C++**. Στην συνέχεια ακολουθεί μια περιληπτική ανάλυση του κάθε κώδικα καθώς και τα αποτελέσματα των emulations.

Ο κώδικας **host.cpp** ο οποίος προορίζεται για να τρέξει στην CPU, αρχικά ορίζει **4 vectors**, 2 vectors εισόδου και 2 vectors με τα αθροίσματα που θα υλοποιηθούν σε Software και Hardware αντίστοιχα. **Αρχικοποιεί**, με τυχαίες τιμές τους πίνακες και **εκτελεί το Software άθροισμα** των πινάκων. Στην συνέχεια, ψάχνει να βρει **διαθέσιμη συσκευή Alveo**, ώστε να της **αναθέσει το binary file** που θα δημιουργήσουμε. Τέλος, αντιγράφει τα vectors στην **Global Memory**(που είναι προσβάσιμη και από την CPU και από το FPGA), καλεί τον **Kernel**, **συγκρίνει** τα αποτελέσματα του Hardware με αυτά του Software και εκτυπώνει **TEST PASSED** εάν εκτελέστηκε σωστά.

Ο κώδικας **vadd.cpp** φτιάχνει κάποια **interfaces**, έπειτα δημιουργεί στην **BRAM** 3 πίνακες, 2 πίνακες εισόδου και 1 πίνακα όπου θα έχει τις επιμέρους προσθέσεις των κελιών. Στην συνέχεια, **αντιγράφει** του πίνακες από την **Global Memory** στην **BRAM** με τεχνικές *pipelining*, **υλοποιεί τις προσθέσεις των κελιών** και τέλος **αντιγράφει** τον πίνακα πίσω στην **Global Memory**.

Στην συνέχεια, αφού κάνουμε **Build** και **Run**, βλέπουμε τα αποτελέσματα του **Software Emulator** (αριστερή εικόνα) και του **Hardware Emulator** (δεξιά εικόνα) αντίστοιχα. Αυτά που αξίζει να σημειωθούν είναι το data transfer από την **DRAM** του host στην **Global Memory** μέσω τεχνικών **DMA**, όπως φαίνεται στην τρίτη και τέταρτη σειρά της αριστερής εικόνας. Ενώ, στην δεξιά εικόνα, στις δύο τελευταίες σειρές, μπορούμε να δούμε την μεταφορά δεδομένων από την **Global Memory** στην **BRAM του FPGA**, μέσω των **AXI Interfaces**.





Ερώτημα 2.

Σε αυτό το ερώτημα, αλλάξαμε τον κώδικα του **host.cpp** έτσι ώστε να υλοποιεί τον software πολλαπλασιασμό δύο δισδιάστατων πινάκων και να καλεί τον **kernel** που δημιουργήσαμε σε C++ ο οποίος υλοποιεί τον πολλαπλασιασμό σε Hardware.

Αρχικά, όπως και στο host.cpp του προηγούμενου ερωτήματος δημιουργήσαμε στην **DRAM** του host 4 vectors με διαστάσεις **n*m** και **m*p** για τα vectors εισόδου και **n*p** για τα vectors πολλαπλασιασμών. Στην συνέχεια εκτελέστηκε ο **software πολλαπλασιασμός** των vectors, ενώ μηδενίσαμε όλα τα στοιχεία του vector που θα

αποθηκεύσει τα στοιχεία του Hardware πολλαπλασιασμού. Ακόμα, όπως και προηγουμένως ο host ψάχνει για διαθέσιμα **Alveo Boards** για να του αναθέσει σαν task το **binary αρχείο** που θα δημιουργήσουμε. Τέλος, αντιγράφει τα vectors από την **DRAM** στην **Global Memory**, καλεί τον **kernel** και **συγκρίνει** τα αποτελέσματα του Hardware με αυτά του Software και εκτυπώνει **TEST PASSED** εάν εκτελέστηκε σωστά.

```
67  std::vector<int, aligned_allocator<int>> source_in1(n*m);
68  std::vector<int, aligned_allocator<int>> source_in2(m*p);
69  std::vector<int, aligned_allocator<int>> source_hw_results(n*p);
70  std::vector<int, aligned_allocator<int>> source_sw_results(n*p);
71
72  // Create the test data
73  std::generate(source_in1.begin(), source_in1.end(), std::rand);
74  std::generate(source_in2.begin(), source_in2.end(), std::rand);
75
76  int result;
77  for(int i = 0; i < n; i++) {
78      for(int j = 0; j < p; j++){
79          result  = 0;
80          for(int k = 0; k < m; k++) {
81              result += source_in1[i*m + k] * source_in2[k*p + j];
82          }
83          source_sw_results[i*p + j] = result;
84          source_hw_results[i*p + j] = 0;
85      }
86  }
```

```
154   bool match = true;
155   for (int i = 0; i < n; i++) {
156       for(int j = 0; j < p ; j++){
157     if (source_hw_results[i*p +j] != source_sw_results[i*p +j]) {
158       std::cout << "Error: Result mismatch" << std::endl;
159       std::cout << "i = " << i << " CPU result = " << source_sw_results[i*p +j]
160                 << " Device result = " << source_hw_results[i*p +j] << std::endl;
161       match = false;
162       break;
163     }}
164   }
165
166   std::cout << "TEST " << (match ? "PASSED" : "FAILED") << std::endl;
167   return (match ? EXIT_SUCCESS : EXIT_FAILURE);
168 }
```

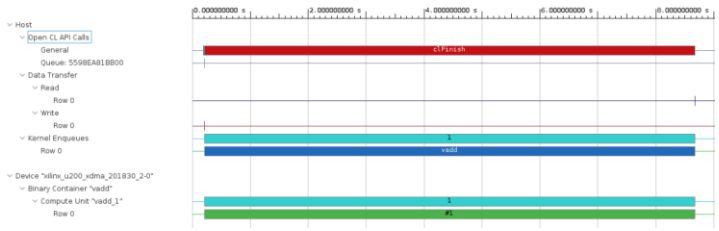
Έπειτα, ο **kernel** που δημιουργήσαμε, ορίζει τα **AXI Interfaces**, δημιουργεί δύο πίνακες στην **BRAM** τους οποίους κάνει *ARRAY PARTITION* και αντιγράφει τους δυο πίνακες εισόδου από την **Global Memory** στην **BRAM**. Τέλος, **εκτελεί τον πολλαπλασιασμό** των πινάκων με μεθόδους *Pipeline* και *Loop Unrolling* και αντιγράφει τα αποτελέσματα στην **Global Memory**, ώστε να μπορούν να γίνουν access από τον host.

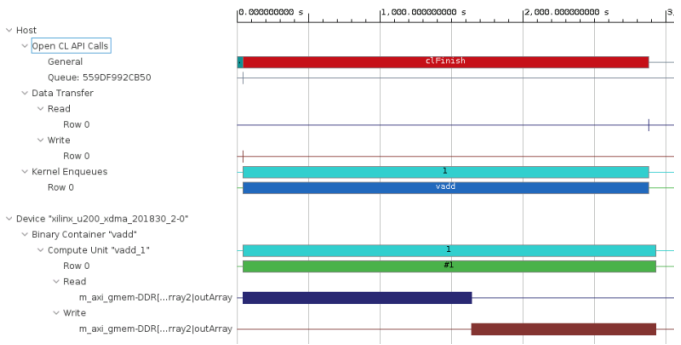
```
37    #pragma HLS INTERFACE m_axi port = inArray1 offset = slave bundle = gmem
38    #pragma HLS INTERFACE m_axi port = inArray2 offset = slave bundle = gmem
39    #pragma HLS INTERFACE m_axi port = outArray offset = slave bundle = gmem
40    #pragma HLS INTERFACE s_axilite port = inArray1 bundle = control
41    #pragma HLS INTERFACE s_axilite port = inArray2 bundle = control
42    #pragma HLS INTERFACE s_axilite port = outArray bundle = control
43    #pragma HLS INTERFACE s_axilite port = return bundle = control
44
45    ap_int<32> BRAM_in1[n][m];
46    ap_int<32> BRAM_in2[m][p];
47
48    #pragma HLS ARRAY_PARTITION variable=BRAM_in1 cyclic factor=m_iter dim=2
49    #pragma HLS ARRAY_PARTITION variable=BRAM_in2 cyclic factor=m_iter dim=1
50
51    ARRAY_COPY1(inArray1,BRAM_in1);
52    ARRAY_COPY2(inArray2,BRAM_in2);
```

```
56     for(int i = 0; i < n; i++){
57     #pragma HLS loop_tripcount min=n_iter max=n_iter
58
59         for(int j = 0; j < p; j++) {
60         #pragma HLS loop_tripcount min=p_iter max=p_iter
61         #pragma HLS UNROLL factor=2
62         #pragma HLS PIPELINE II=1
63
64             result = 0;
65     mainloop: for(int k = 0; k < m; k++){
66                 #pragma HLS loop_tripcount min=m_iter max=m_iter
67
68                 result += BRAM_in1[i][k] * BRAM_in2[k][j];
69             }
70             outArray[i*p + j] = result;
71         }
72     }
```

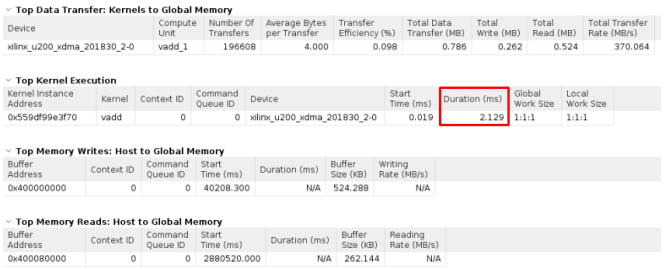
```
10  void ARRAY_COPY1(ap_int<32> *inArray1,ap_int<32> A[n][m]){
11      for(int i = 0; i < n; i++){
12          #pragma HLS loop_tripcount min=n_iter max=n_iter
13              for(int j = 0; j < m; j++){
14                  #pragma HLS loop_tripcount min=m_iter max=m_iter
15                  #pragma HLS UNROLL factor=2
16                  #pragma HLS PIPELINE II=1
17                  A[i][j] = inArray1[i*m + j];
18              }
19          }
20  }
21
22  void ARRAY_COPY2(ap_int<32> *inArray2,ap_int<32> B[m][p]){
23      for(int i = 0; i < m; i++){
24          #pragma HLS loop_tripcount min=m_iter max=m_iter
25              for(int j = 0; j < p; j++){
26                  #pragma HLS loop_tripcount min=p_iter max=p_iter
27                  #pragma HLS UNROLL factor=2
28                  #pragma HLS PIPELINE II=1
29                  B[i][j] = inArray2[i*p + j];
30              }
31          }
32  }
```

Τέλος, στις παρακάτω εικόνες βλέπουμε τα αποτελέσματα των **Software Simulation** (αριστερή εικόνα) και **Hardware Simulation** (δεξιά εικόνα).





Τέλος, βλέπουμε ότι ο **χρόνος εκτέλεσης** του kernel σε Hardware Emulator είναι **2.126ms**, συνεπώς γνωρίζουμε ότι η υλοποίηση στο Alveo Board θα χρειαστεί λιγότερο από 2.126ms, δεδομένου ότι ο Hardware Emulator δίνει πάντοτε το worst case scenario.



Top Data Transfer: Kernels to Global Memory

Device	Compute Unit	Number Of Transfers	Average Bytes per Transfer	Transfer Efficiency (%)	Total Data Transfer (MB)	Total Write (MB)	Total Read (MB)	Total Transfer Rate (MB/s)
xilinx_u200_xdma_201830_2-0	vadd_1	196608	4.000	0.098	0.786	0.262	0.524	370.064

Top Kernel Execution

Kernel Instance Address	Kernel	Context ID	Command Queue ID	Device	Start Time (ms)	Duration (ms)	Global Work Size	Local Work Size
0x559df99e3f70	vadd	0	0	xilinx_u200_xdma_201830_2-0	0.019	2.129	1:1:1	1:1:1

Top Memory Writes: Host to Global Memory

Buffer Address	Context ID	Command Queue ID	Start Time (ms)	Duration (ms)	Buffer Size (KB)	Writing Rate (MB/s)
0x400000000	0	0	40208.300	N/A	524.288	N/A

Top Memory Reads: Host to Global Memory

Buffer Address	Context ID	Command Queue ID	Start Time (ms)	Duration (ms)	Buffer Size (KB)	Reading Rate (MB/s)
0x400080000	0	0	2880520.000	N/A	262.144	N/A