Github repo: https://github.com/MasterYI814/WES237C

Q1:

The input x is placed in register shift_reg with the value of x feeding into shift_reg every loop. The FIR filter coefficients are placed in a register c and a multiply and accumulate function will perform every time the FIR function is called. So we have control of the bitwidths of the shift_reg, the coefficients register and the accumulated result. Knowing the largest and smallest value possible in the coefficients and input data, we can find the appropriate bitwidth. The range of coefficients is +/-11, so I choose ap_int<5>. The range of input data is +/-100 so I choose ap_int<8>. And finally, the bitwidth of the accumulated result is calculated as 8+5+8+1=22 bit. The 8 and 5 are the bitwidth of the multiplication, the second 8 is for adding it 128 times and the 1 is for the sign bit. Depending on the input data, we can further minimize the bitwidth of shift_reg and acc, but this is a safe call based on the information we have.

The result shows an improvement from 541kHz to 575kHz with a reduction of BRAM, DSP, FF, LUT usage. I think the throughput improvement is not as large as the resource reduction improvement in this optimization.

Q2:

As we increase the II in for loop we see the latency per cycle increases from 134 to 516. This makes sense as II is defined as the number of clock cycles until the next iteration of the loop can start. The larger the II, more cycles take to run before the next iteration. As we increase the II past 4, we see the Final II stays at 4 and the latency cycles do not increase any more. This makes sense because the number of cycles it takes to finish two reads, one multiply and one addition is 4 cycles. So any II set larger than 4 will not be valid. To calculate the II for a general loop I will need to find the operations and see how long it takes to run them sequentially and that will be my maximum II.

Q3:

The original FIR implementation uses an if/else control inside the loop and it is unnecessary because one of the conditions is only run one time. This is a waste of resources. So I change the structure of the for loop. I move the i=0 condition out of the for loop. By doing this, I keep the latency cycle and throughput is now 1.14MHz. I gain in decreasing the number of FF and LUT used. FF decrease from 262 to 113 and LUT decrease from 306 to 234.

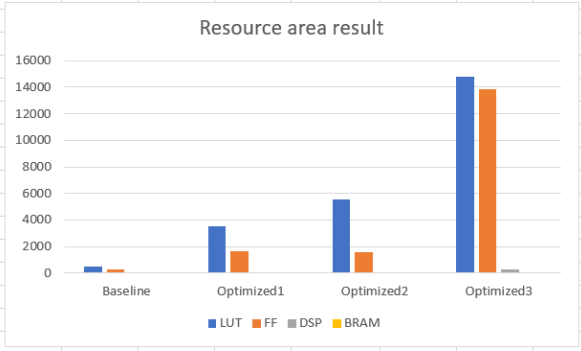
Q4:

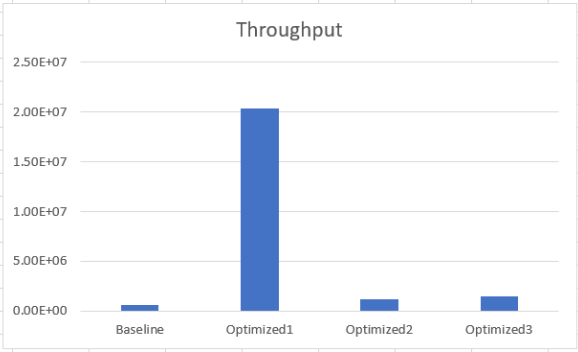
I was able to partition the TDL and MAC operations in the same for loop to two for loops. And I use HLS unroll factor = 8 to achieve a throughput of 2.03MHz. When I started to go for a higher unroll factor I see a slack factor is introduced in the synthesis report. We see a drastic increase in the number of DSP, FF and LUT used and this is expected because we are applying parallelism to the TDL and MAC and the logic/delay units should increase with that. We achieve to increase the clock speed and reduce latency in this part with a drastic increase in resource size.

Q5:
Cyclic partitioning creates smaller arrays by interleaving elements from the original array. The array is partitioned cyclically by putting one element into each new array before coming back to the first array to repeat the cycle until the array is fully partitioned. Block partitioning creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into N equal blocks. Complete partitioning decomposes the array into individual elements. For a one-dimensional array, this corresponds to resolving a memory into individual registers.
By default, using a complete array partition will decimate the array into individual registers. it is used when the entire array is accessed at once or when resource constraints dictate that only a single memory block can be used. As a result, we see BRAM is used when using complete. When I experiment with cyclic and block partitioning, I found out that they seem to yield the same result, in terms of cycle time, cycle number and resource. Compared to complete partitioning, we are not using BRAM anymore and the number of FF used increased slightly. But the big decrease is in the number of cycles. (it decreases from 119 to 7) making the throughput 23.7MHz. Because I have used pipeline and parallelism in my code before, using a cyclic partition increases memory bandwidth and makes faster access time.

Question 6:
My approach to a high throughput design is to fully unroll all the for loops (we know as a fact that the filter is 128 tap) and in addition I add cyclic array partitioning to it. The for loop is splitted into two to take advantage of code hoisting. Variable bandwidth is also used to find the smallest possible variable size. I am able to achieve an estimated cycle time of 7.013ns and latency cycles of 7, giving a final throughput of 20.37MHz.

	LUT	FF	DSP	BRAM	Throughput
Baseline	461	240	1	1	5.73E+05
Optimized1	3535	1661	12	0	2.04E+07
Optimized2	5509	1544	12	1	1.17E+06
Optimized3	14798	13838	304	0	1.46E+06





The final result shows that optimized1 has the best throughput at 20MHz and it does not use as much resource compared to other architectures.

Here is a screenshot of the optimized1 design synthesis summary.