Nate Mosk

WES 237C - Project 1: FPGA optimization with a FIR filter

FPGAs are like a middle ground between a software program and an application specific integrated circuit. They allow us “reprogram” a chip to do our bidding.

Vitis is a Xilinx tool that allows us to convert C/C++ code into verilog or VHDL for use on an FPGA. This practice is known as High Level Synthesis. Vitis allows us to synthesize C code and get metrics on performance, including clock period and initiation interval.

Vitis has features that allow for optimization of our design, which is important so we can create devices that work well. Using an FIR filter as an example, we will discuss some basic design optimizations such as loop unrolling, bitwidth optimization, pipelining, and memory partitioning.

Our basic FIR filter will convolve 128 filter coefficients with data as it is streamed into our filter.

The way we will measure performance is via throughput. This is “the number of times that the fir function can be invoked per second. Since we are giving the fir function one sample for each invocation, it is equivalent to the samples/second that this filter could handle.”

For our 128-tap FIR filter, our baseline throughput is 1.07 MHz, close to a million samples per second. Let’s see if we can improve it with optimization.

1. Bitwidth

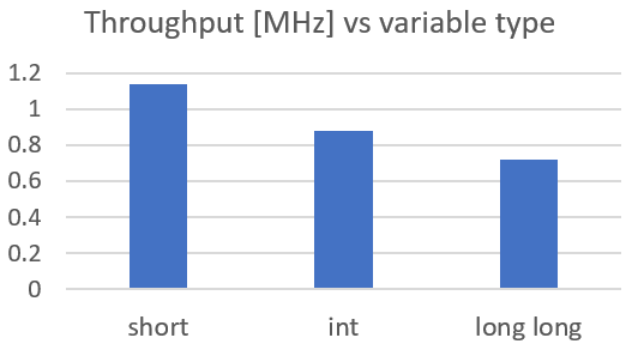
An int has 32 bits while a short has 16 bits. Long long types have 64 bits! Luckily we do not need to go to 9 gazillion to match our golden output.

By handling variables that have less bits, we can save on processing time (at the expense of precision). The least amount of regular bits we can use is 16 or short. Using an 8 bit char type only allows us -128 to 127, and we need bigger numbers than that to use our filter as intended with the inputs provided.

To implement, simply declare our variables as shorts. E.g. short acc = 0;

Using a shorter bitwidth improves the throughput of the filter. It’s worth noting that the interval cycle count for each of these implementations was similar, but the clock time for the long long type was 2x longer than the short type implementation. This makes sense because it’s the same number of calculations or steps, but the calculation of a long long number is going to take much longer because it is an additional 48 bits to handle.

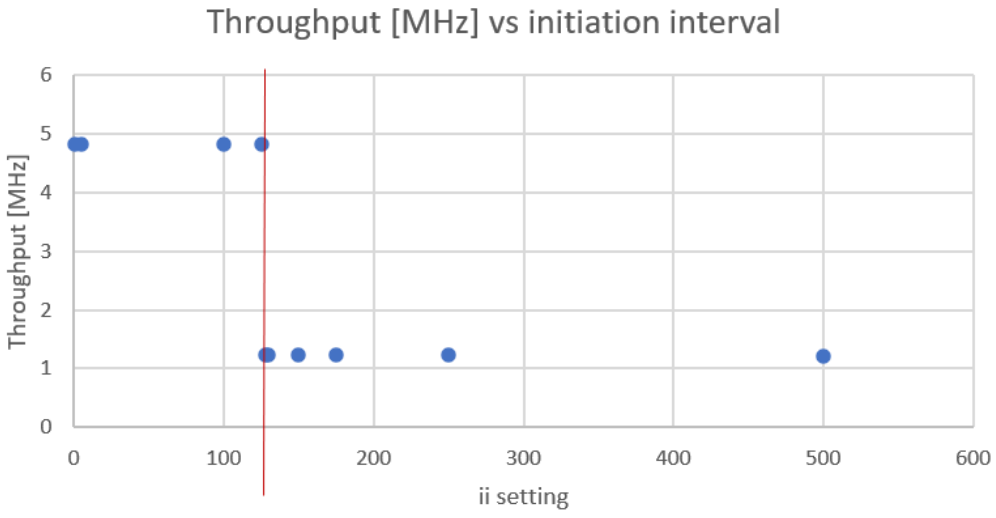
See results below show that using a short is the fastest type. We can potentially play with other types using ap_int tool in vitis that allows us to do an arbitrary portion of bits, but this simple test of short vs long shows the concept well.



2. Pipelining

By using the #pragma HLS pipeline ii = X setting, we can direct vitis to optimize the IP for initiation interval. For the 128 tap filter, we have very high throughput when we take advantage of the filter's known number of MACs (128 multiply accumulates). When we tell vitis to start the next computation before the last one is finished, we can save a lot of time. This is known as Pipelining. It makes sense that once we set the ii to 128 or greater, our throughput takes a big hit because initiating at 129 cycles is basically the same as not optimizing at all in this case. See below on the chart, at the redline for ii = 128, throughput goes from almost 5 MHz down to 1 MHz.

Results:
Throughput w/pipeline @ <128 = 4.81 MHz
Throughput w/pipeline @ >128 = 1.23 MHz or less



3. Code Hoisting

Following along with our textbook, there is originally an if statement embedded in the loop. Since we know that eventually we will get to the 0th entry in our filter, we can remove the if statement asking if i = 0. We put the initial mult-acc after the for loop and thereby hoist the if statement out of the program.

Funnily enough, this only improved latency cycle count from 135 baseline to 134 optimized. Once we add back in pipelining, it goes to 30, and there's no difference really.

I will keep the code this way though because it is cleaner and more simple.

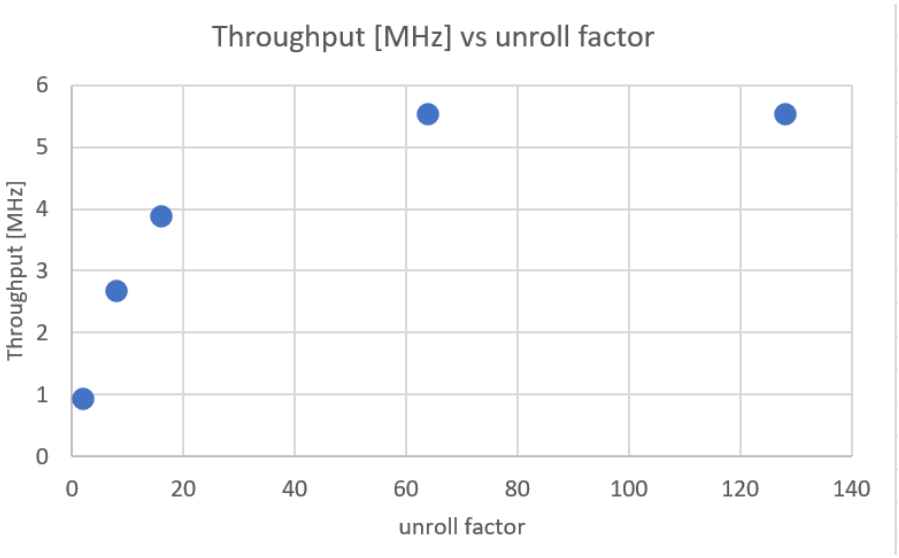
4. Loop Partitioning

By breaking the shift loop and the MAC loop into separate pieces of code, we can allow these to run in parallel. We can also take advantage of loop unrolling.

I used the command #pragma HLS unroll factor=16+ to unroll the loops by large factors. Believe it or not unrolling by a factor of 2 actually slowed down the code.

Testing out different values we can see improvement until the loop is almost completely unrolled. We are able to completely unroll the loop and vitis actually doesn't even recognize that there are two loops when we set the unroll factor to 128. It only sees the top level function, fir.

I'm going to save the version of 32 factor unroll because I think we are harnessing a good amount of optimization without losing the actual loop. We get it done in 4 big chunks.



Note that this only works when the register of filter values can be accessed separately. We need complete memory partitioning. When we fail to partition the memory, the loop partition actually is

a bad move. It causes throughput to tank because we are doubling the amount of looping we have to do.

5. Memory Partitioning

By doing memory partitioning, we allow vitis to cut up our variable holding our filter data and store it in multiple pieces. The advantage is that we can access more entries at the same time, and use them in parallel computations.
Block memory cuts the memory into blocks, cyclic memory partitions the memory and will rotate through entries, kind of like RAID storage. Using #pragma HLS ARRAY_PARTITION type=complete, we can completely partition all the array entries for complete random access.

When testing the different uses, they're throughput does not change very much.

However, when the memory partition is used in combination with the loop partition and unrolling, we see great effects in the throughput. This would make sense because by allowing the memory to be accessed completely and doing an unrolled loop, we can basically do 128 computations at the same time rather than having to wait and read memory while running through the loops.

6. Best Case Design

That's the principle which we use in the best case design.

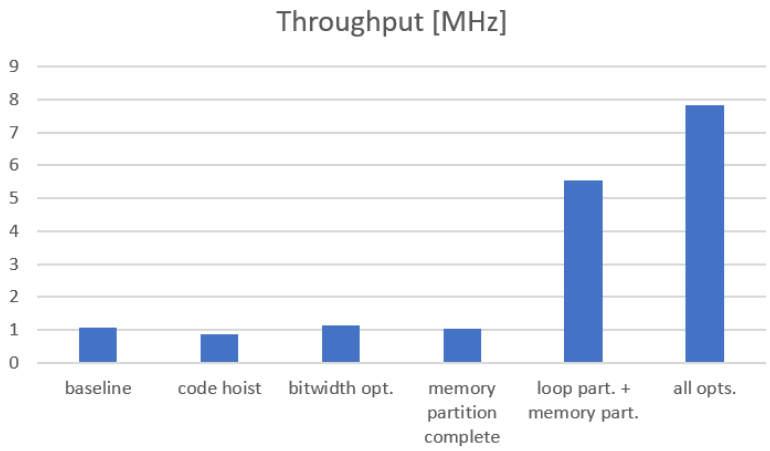
We have:

- Bitwidth reduction by defining our variables as type *short*, we don't need the extra precision and time to process
- loop unrolling with the partitioned loops to allow for parallelism
    - #pragma HLS unroll factor=64
- In tandem with memory partition
    - #pragma HLS ARRAY_PARTITION variable=my_reg type=complete
- On top of that, we can set up a pipeline with initiation interval
    - #pragma HLS pipeline ii=110
- And of course we hoisted the conditional statement out of the code and didn't introduce any other unnecessary steps

This allowed for a throughput of 7.8 MHz, our best performance yet. The graph below is a survey of our various results with different code optimizations.

Clearly it is worth using optimizations to get better performance on our FIR filter as we were able to do about 8x more samples per second, basically an order of magnitude.

These types of optimizations will work for other applications than this simple FIR filter.



For reference, here's our baseline compared to our best code. You can match the bullet points above to the code to follow along with each improvement we made.

Baseline from book:

```
void fir(data_t *y, data_t x) {
    coef_t c[N] = {
        53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53
    };
    static
     data_t shift_reg[N];
    acc_t acc;
    int i;

    acc = 0;
    Shift_Accum_Loop:
    for (i = N - 1; i >= 0; i--) {
        if (i == 0) {
            acc += x * c[0];
            shift_reg[0] = x;
        } else {
            shift_reg[i] = shift_reg[i - 1];
            acc += shift_reg[i] * c[i];
        }
    }
    *y = acc;
}
```

Best code:

```
#include "fir.h"

void fir (
  data_t *y,
  data_t x
  )
{
#pragma HLS pipeline ii=110
    coef_t c[N] = {10, 11, 11, 8, 3, -3, -8, -11, -11, -10, -10,
    // Write your code here
    static data_t my_reg[N];
#pragma HLS ARRAY_PARTITION variable=my_reg type=complete
    acc_t acc;
    int i;
    acc = 0;
    for (i = N-1; i > 0; i--){
#pragma HLS unroll factor=64
            my_reg[i] = my_reg[i - 1];
        }
    for (i = N-1; i > 0; i--){
#pragma HLS unroll factor=64
            acc += my_reg[i] * c[i];
        }
        acc += x * c[0];
        my_reg[0] = x;
    *y = acc;

}
```

Note: After writing this report I realized one fatal flaw …. You can see great gains by setting ii = 1 once you have done the memory partition and loop unroll.

I was able to get up to 86 MHz of throughput !!!

This makes sense because when I was originally testing out the pipelining, I didn't have memory partition set up yet, so probably it could not take advantage of the II setting.