Nathaniel
Mosk

Discrete Fourier Transform

The Discrete Fourier Transform is a mathematical tool used to find the resonant frequencies over a subset of frequencies within the sample frequency bandwidth.

An input of a sampled signal in time is the input to the transform and the output is a set of frequencies and their relative strength.

Baseline Implementation

We have a simple program that will help us calculate the DFT for a given sampled signal.

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn} \quad \textbf{(Eq.1)}$$

The program implements this equation (taken from the wikipedia page for DFT). The input samples span n from 0 to N-1 and the frequencies we are interested in span k, integer multiples of size 1/N.

There is a loop that iterates through the frequencies and then another loop inside that sums up our energies across samples.

Sine and Cosine

To calculate the complex exponential, we can use euler identities and the sine and cosine functions.

When we do this we know that HLS and Vivado will implicitly use a cordic to calculate those values.

If we were to implement a custom cordic inside the program, it probably would not perform any better than the automatic cordic that HLS provides.

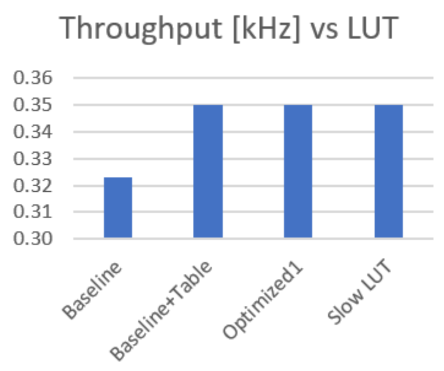
Look Up Table

One way to optimize our program is to remove the calculation of the sine and cosine values. By adding a look up table with those values, we can have those values on standby.

Implementing this was tricky because the program has to iterate through two dimensions at a time, and our LUT is only one dimension. Luckily we know that the trigonometric values are symmetric about 2pi. So we can find the modulus of i*j and use that sine or cosine value.

Another thing we can do is add an output array so that we can avoid having to loop back through our sample values to replace them.

Frankly, I expected the implementation of the LUT to have a bigger effect on the throughput.



The chart above shows that there was improvement, however it was only 7% faster by removing the sine and cosine. Example optimized 1 is the LUT with the output array, baseline & table is just the LUT, and the slow LUT represents when I had the program generate a LUT.

Adding the output array had virtually no effect on the throughput.

Loop Considerations

Every time the program loops, we are using up overhead to count the next iteration. If we can unroll the loop, we can avoid those calculations.

Another thing we can do is pipeline the operation such that every new sample we get we can start a new cycle of calculations.

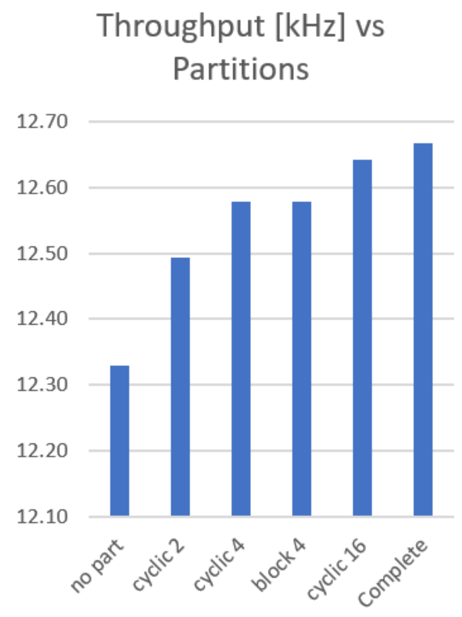
By getting started on the next cycle of calculations, we can take advantage of doing parallel operations and making the most of our resources.

I found good results by pipelining the program.

Once the program is pipelined we can partition the memory of the input samples so that the values can be more readily accessed by the processor.

The below graph shows that the more partitioned the array is, the higher throughput the program has. All the experiments were implemented on a program with #HLS pipeline ii=32 on the outside loop and ii=3 on the inside loop.

Nathaniel
Mosk



Dataflow

If we were able to break the program into functions or modules, we could use dataflow pragma to arrange the functions in a streamline manner.

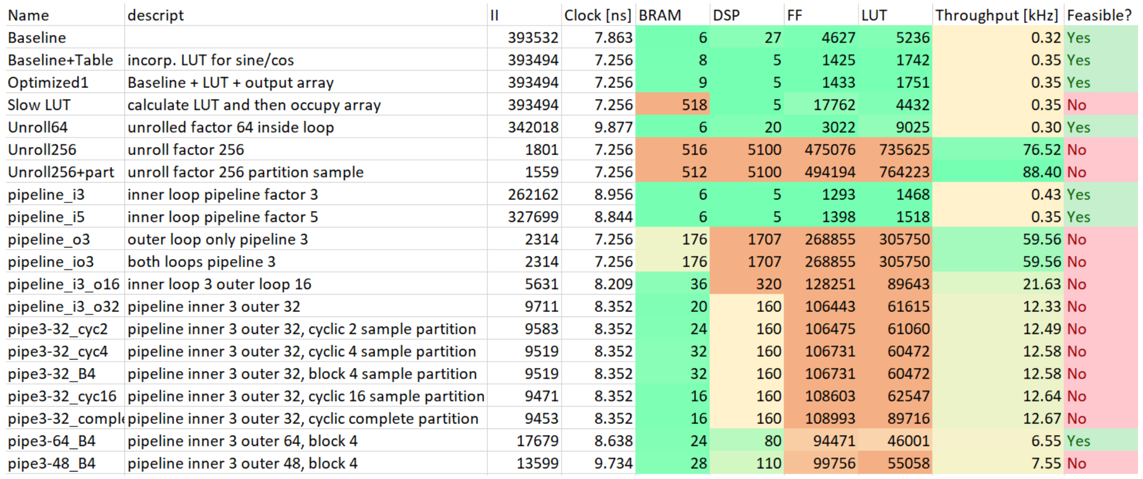
This program is really simple and essentially just has a double loop. I couldn't think of a way to make this into modules. That's probably the reason why adding the dataflow pragma didn't result in any improvement.

Best Architecture

At the 256 sample DFT level, I was able to get a feasible design with a ii=64 pipeline. Anything lower took up too many resources.

However, when I went up to the 1024 sample DFT I couldn't get the pipelining to work and still use low enough resources to be feasible.

Nathaniel
Mosk



Name	descript	II	Clock [ns]	BRAM	DSP	FF	LUT	Throughput [kHz]	Feasible?
Baseline		393532	7.863	6	27	4627	5236	0.32	Yes
Baseline+Table	incorp. LUT for sine/cos	393494	7.256	8	5	1425	1742	0.35	Yes
Optimized1	Baseline + LUT + output array	393494	7.256	9	5	1433	1751	0.35	Yes
Slow LUT	calculate LUT and then occupy array	393494	7.256	518	5	17762	4432	0.35	No
Unroll64	unrolled factor 64 inside loop	342018	9.877	6	20	3022	9025	0.30	Yes
Unroll256	unroll factor 256	1801	7.256	516	5100	475076	735625	76.52	No
Unroll256+part	unroll factor 256 partition sample	1559	7.256	512	5100	494194	764223	88.40	No
pipeline_i3	inner loop pipeline factor 3	262162	8.956	6	5	1293	1468	0.43	Yes
pipeline_i5	inner loop pipeline factor 5	327699	8.844	6	5	1398	1518	0.35	Yes
pipeline_o3	outer loop only pipeline 3	2314	7.256	176	1707	268855	305750	59.56	No
pipeline_io3	both loops pipeline 3	2314	7.256	176	1707	268855	305750	59.56	No
pipeline_i3_o16	inner loop 3 outer loop 16	5631	8.209	36	320	128251	89643	21.63	No
pipeline_i3_o32	pipeline inner 3 outer 32	9711	8.352	20	160	106443	61615	12.33	No
pipe3-32_cyc2	pipeline inner 3 outer 32, cyclic 2 sample partition	9583	8.352	24	160	106475	61060	12.49	No
pipe3-32_cyc4	pipeline inner 3 outer 32, cyclic 4 sample partition	9519	8.352	32	160	106731	60472	12.58	No
pipe3-32_B4	pipeline inner 3 outer 32, block 4 sample partition	9519	8.352	32	160	106731	60472	12.58	No
pipe3-32_cyc16	pipeline inner 3 outer 32, cyclic 16 sample partition	9471	8.352	16	160	108603	62547	12.64	No
pipe3-32_compl	pipeline inner 3 outer 32, cyclic complete partition	9453	8.352	16	160	108993	89716	12.67	No
pipe3-64_B4	pipeline inner 3 outer 64, block 4	17679	8.638	24	80	94471	46001	6.55	Yes
pipe3-48_B4	pipeline inner 3 outer 48, block 4	13599	9.734	28	110	99756	55058	7.55	No

See above, only pipe3-64_B4 has low enough resources to be feasible.

You can see that as we have more partitions, we have more flip flops and LUTs taken up.

Streaming Interface

By streaming our samples we can probably make the program better. I couldn't figure out how to do this but I made some good infrastructure by following the examples.