CSE237C Project 3 - Discrete Fourier Transform (DFT)

Henrik Larsson Hestnes

October 2021

1 Discrete Fourier Transform - Introduction

The purpose of this project was to implement and optimize a Discrete Fourier Transform implementation in Vitis. To calculate the throughput, the following equation has been utilized:

$$Throughput[Hz] = \frac{1}{latency\ of\ DFT-function[\#cycles] * estimated\ clock\ period[s]}$$

1.1 Baseline implementation

RMSE(R)	RMSE(I)	Throughput	BRAM	DSP	FF	LUT
0.041321460157633	0.030046040192246	38.9 Hz	2	45	5184	8852

This is the performance and resource usage of the baseline implementation. This baseline implementation will be used in the coming sections to compare different optimizations. This baseline is based on the DFT256, and every comparison will be of that DFT size unless specified else.

1.2 Optimization 1 - table lookup

RMSE(R)	RMSE(I)	Throughput	BRAM	DSP	FF	LUT
0.000082730904978	0.000134893096401	131 Hz	4	16	1480	2440

In this optimization, the table lookup from coefficients256.h was utilized instead of the cos() and sin() function calls. Since this optimization seems desirable in almost every aspect, the coming optimizations will be built on top of this.

1.3 Optimization 2 - separate input and output arrays

RMSE(R)	RMSE(I)	Throughput	BRAM	DSP	FF	LUT
0.000082730904978	0.000134893096401	131 Hz	2	16	1460	2337

In this "Optimization", the code and interface was rewritten so that the inputs and outputs are stored in separate arrays. As "Question 4" states, the next optimizations will be based on top of this.

1.4 Optimization 3 - array partitioning and pipelining

Throughput	BRAM	DSP	FF	LUT
414 Hz	2	5	964	1289

This optimization combines a cyclic array partitioning of every input and output array with a factor of 2, and also allows pipelining the loops.

1.5 Optimization dataflow

Throughput	BRAM	DSP	FF	LUT
209 Hz	8	10	1536	2514

In this optimization the code was restructured, to see the effect of exploiting the dataflow.

1.6 Optimization 4 - hls::stream

Throughput	BRAM	DSP	LUT	FF
131 Hz	4	16	1504	2595

In this optimization the code was changed to use hls::stream¡DTYPE¿ instead of DTYPE.

2 Q&A

2.1 Question 1

What changes would this code require if you were to use a custom CORDIC similar to what you designed for Project: CORDIC? Compared to a baseline code with HLS math functions for cos() and sin(), would changing the accuracy of your CORDIC core make the DFT

hardware resource usage change? How would it affect the performance? Note that you do not need to implement the CORDIC in your code, we are just asking you to discuss potential tradeoffs that would be possible if you used a CORDIC that you designed instead of the one from Xilinx.
To use a custom CORDIC in this code, line 22 and 23 of the baseline dft.cpp would had to be changed, and the calculation of w in line 17 can be removed. It is in line 22 and 23 the sine and cosine valuesm are being calculated, and the function call to these trigonometric functions would had to be changed with some function call to a custom function which calculates these values using the CORDIC principles. One would of course also need to implement a fully functioning function which calculates these CORDIC sine and cosine values.
The custom CORDIC core would of course be more configurable than the HLS math functions, which again would lead to the possibility of improving performance, resource usage and accuracy, probably at the cost of each other. This could however lead to improvement in both performance and resource usage compared to the Xilinx design, since the custom implementation will be very application specific, at the cost of actually having to design and optimize this custom CORDIC core.

2.2 Question 2

Rewrite the code to eliminate these math function calls (i.e. cos() and sin()) by utilizing a table lookup. How does this change the throughput and resource utilization? What happens to the table lookup when you change the size of your DFT?
As can be seen in section 1.2, Optimization 1 utilizes the table lookup instead of the sine and cosine from the baseline implementation. This gives a drastic 237% improvement of the throughput, which makes very much sense considering that the time demanding trigonometric functions now are replaced with a simple table lookup. You can also see that the resource usage goes down with a fair amount for every resource except BRAM. The reason for this is that the two sine and cosine tables will be stored in the BRAM, and we do not have to do the resource demanding calculation of the trigonometric values anymore.

SIZE	BRAM	DSP	FF	LUT
8	0	16	1617	2443
32	4	16	1447	2435
256	4	16	1480	2440
1024	8	16	1502	2435

What happens with the table lookup when you change the size of the DFT is that the table also changes size, and thus will be stored differently. When SIZE=8, you can see that the design uses no BRAMs, but has a higher usage of FFs, which is probably where the tables are stored. When SIZE=32 and

SIZE=256, the design uses 4 BRAMs, 1 BRAM per temp and coefficient array. Each BRAM can store up to 18000 bits, and since the four arrays per design uses $32 * 32 = 1024$ bits for SIZE=32, and $32 * 256 = 8192$ bits for SIZE=256, alle of these will fit into one BRAM. When SIZE=1024, each array will use $32 * 1024 = 32768$ bits, and will therefore need 2 BRAMs per array. That is why the design ends up using 8 BRAMs.

2.3 Question 3

Modify the DFT function interface so that the input and outputs are stored in separate arrays. How does this affect the optimizations that you can perform? How does it change the performance? And how does the resource usage change? Modify your testbench to accommodate this change to DFT interface. You should use this modified interface for the remaining questions.

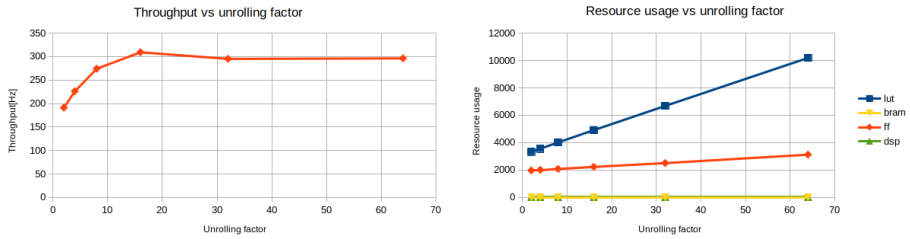
The biggest impact this has on the possibility of optimizations is that this implementation makes it possible to output the data as a stream, since we no longer have to wait for the whole matrix multiplication to be completed before making the output data available. This means that one potential optimization is to make an output stream that streams element per element, as soon as they are correct and available.

As can be seen by comparing section 1.3 to section 1.2, this optimization does not affect the throughput performance. The BRAM usage goes down from 4 to 2 due to the fact that the temporary arrays is removed, but other than that there is no significant change in resource usage. The biggest gain of this optimization is thus the possibility to stream the output as soon as it is available.

2.4 Question 4

Study the effects of loop unrolling and array partitioning on the performance and resource utilization. What is the relationship between array partitioning and loop unrolling? Does it help to perform one without the other? Plot the performance in terms of number of DFT operations per second (throughput) versus the unroll and array partitioning factor. Plot the same trend for resources (showing LUTs, FFs, DSP blocks, BRAMs). What is the general trend in both cases? Which design would you select? Why?

The amount of different options and combinations you have when optimizing through combining loop unrolling and array partitioning is enormous. To choose the loop unrolling factor and array partitioning factor in an arbitrary manner is therefor almost always a bad choice, unless it is your lucky day. The key of any optimization is to fully understand the code, and this applies very much here. To narrow down the huge amount of optimizations, this task will start by looking at unrolling the inner for loop of Optimization 3 with different factors, and then optimize with regards to array partition.



(a) Throughput (b) Resource usage

Figure 1: Comparison of performance with different loop unrolling factors and zero array partitioning

As can be seen from the plots above, the resource usage in fig. 1b is growing linearly with the increased unrolling factor, while the throughput saturates around an unrolling factor of around 16.
When it comes to array partitioning, there exist a lot of different optimizations such as block, cyclic and complete, which again can be used on the different arrays, so I really find it difficult to plot the trends. However, the resource usage of especially FF and LUT will go up with a bigger array partitioning factor, since the arrays will be stored here instead of in the BRAMs. The throughput will not necessarily go up, since the array partitioning almost always has to be combined with either pipelining or loop unrolling to improve the throughput.

My initial thought of combining a loop unrolling of the inner for loop with the same factor as the input array partitioning does not seem to give any wanted results when synthesizing it. After trying a significant amount of different optimizations with different loop unrolling factors, array partitioning and pipelining, the best optimization I found was to combine only a small array partitioning and pipelining. The result of this can be seen in section 1.4. I am sure that there is possible to get a better solution in terms of throughput by combining unrolling and partitioning, but considering the low resource usage and the respectable 414 Hz throughput I believe this solution becomes close to Pareto optimal.

EDIT: After finishing this project I read on Piazza that this might have something to do with a $unsafe\_math\_optimizations$ flag not being set in the compiler. Anyway, I did not have the time to do this all over again with the flag set so that might be the reason that unrolling the inner loop with the same factor as the input array was partitioned with did not work.

2.5 Question 5

Please read the dataflow section in the HLS User Guide pages 145-154, or the summary at this page, and apply dataflow pragma to your design to improve throughput. You may need to change your code

and make submodules so that it aligns with the task-level or function-level modularity that dataflow can exploit; an example of dataflow code is available here. How much improvement can you make with it? How does your dataflow design affect resource usage; how did it change compared to without dataflow? What about BRAM usage specifically? Please describe your architecture with figures on your report.

	Throughput	BRAM	DSP	FF	LUT
Optimized 2	131 Hz	2	16	1460	2337
Restructured without dataflow	105 Hz	8	5	1181	1913
Restructured with dataflow	209 Hz	8	10	1536	2514

As can be seen from the table above, there are without doubt possible to make improvements by exploiting the dataflow. The gain in throughput of the version with dataflow compared to the version without dataflow is huge, nearly 100%. Also compared to the "new baseline implementation", optimization 2, the gain in throughput is significant, 60%. With regards to the resource usage, the BRAM usage goes significantly up with the restructuring, due to the increase in internal arrays. The version with dataflow also uses twice as many DSPs as the version without dataflow.

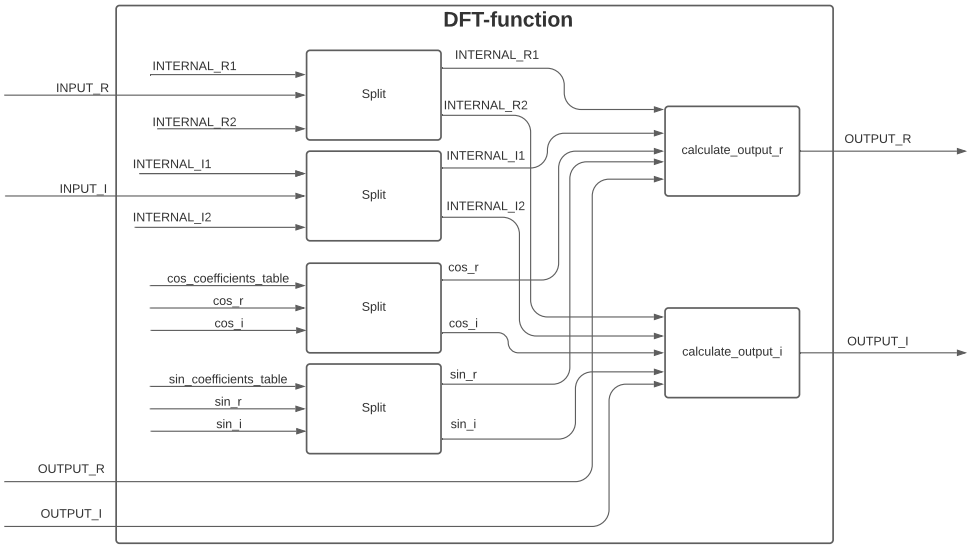


Figure 2: The architecture of the restructured function

2.6 Question 6

(Best architecture) Briefly describe your "best" architecture. In what way is it the best? What optimizations did you use to obtain this result? What is tradeoff you consider for the best architecture?
My "best" architecture architecture is an architecture combining loop pipelining and a simple cyclic array partitioning of factor 2. This architecture was found through a lot of trial and error, and gives a really good throughput considering the low resource usage. I believe this is a near Pareto optimal tradeoff between performance and resource usage, which yielded the following results.

	Throughput	BRAM	DSP	FF	LUT
Best 256	414 Hz	2	5	964	1289
Best 1024	26.2 Hz	4	5	976	1285

2.7 Question 7

(Bonus; streaming architecture) If you create a design using hls::stream, you will get bonus points for Project 3. We do not provide a testbench for this case since this is optional. You must write your own testbench because we expect you to change the function prototype from DTYPE to hls::stream. Please briefly describe what benefit you can achieve with hls::stream and why? NOTE: To get the extra credit, your design must pass Co-Simulation (not just C-Simulation). You can learn about hls::stream from the HLS User Guide page 216-225. An example of code with both hls::stream and dataflow is available (along with its testbench) here, and another here. In section 1.6, optimization 4, you can see the results from changing the architecture to hls::stream. As you can see there is not much of a change in the throughput nor resource usage, except for the two more BRAMs used since the whole input stream has to be saved in two new arrays. The great benefit you can with hsl::stream that cannot be seen from the table is that once a value is done computed and written to the stream, another part of the bigger program can use that value straight away by reading it off the stream. It thus does not have to wait for the complete function to be done computing, which makes it easier to pipeline a bigger program. This implementation passes the Co-Simulation.

3 PYNQ Demo

For the PYNQ Demo, some problem with the Vitis HLS tool did so that the ports in the RTL-code was not interpreted as streams, and thus did not have the TLAST port either. I spoke with professor Kastner about this problem today, 11.02.21. He said he could see that I understand what I have done, and could not immediately see where the problem was. His theory was that due to the new updates in Vitis 2021.1, the stream probably has to be implemented as an

hls::stream for the ports in the RTL-code to be interpreted as streams. Anyway, he told me to deliver the code as-is and explain this, and that I then will get graded on this task for the code I have written.