CSE 237C Project 3 Report

Aravind Seetharaman and Shrivatsan Rajagopalan

INTRODUCTION:

This report describes the implementation of multiple DFT architectures and the various optimizations that were carried out. This report also includes the observations that were made while carrying out these experiments and the results that were obtained in the process.

OBSERVATION AND RESULTS:

QUESTION 1: What changes would this code require if you were to use a custom CORDIC similar to what you designed for Project 2? 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.

RESPONSE TO QUESTION 1:

If we were to use a custom CORDIC we would be using it to implement the rotations for computing DFT. The sin and cos functions used in **dft_baseline** would be replaced by a call to the CORDIC module.

Yes, changing the accuracy of the CORDIC core will make the DFT hardware resource change. Increasing/Decreasing the accuracy of the CORDIC core is done by increasing/decreasing the number of iterations in the cordic loop and also by increasing/decreasing the number of bits for the decimal point. Therefore decreasing the accuracy will result in reduced usage of hardware resources whereas increasing CORDIC accuracy will increase the resource usage.

Therefore the number of iterations can be varied to achieve a tradeoff between accuracy and resource usage.

Also since the CORDIC implemented in Project 2 used fixed point variables instead of floating point variables (as is the case here), further reductions in resource usage can be achieved by making use of a custom CORDIC.

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 area? What happens to the table lookup when you change the size of your DFT?

RESPONSE TO QUESTION 2:

The cos() and sin() function calls were replaced by using the precomputed values of the cos and sin terms from the provided coefficients header file. (**dft256_optimized1)**

Throughput was calculated using the following equation:

Throughput (MHz) = 1000/ [(latency (clock cycles)) * (clock period (ns))]

TABLE 1

Architecture	Throughput (KHz)	BRAMs	DSP48s	FFs	LUTs
dft256_baseline	0.0256	18	199	12462	19175
dft256_optimized1	0.1208	4	16	1474	2498

The first major observation that was made by making use of this change is the considerable reduction in latency (and therefore an increase in throughput). This was observed because the complicated sin and cos functions computations which were calculated on floating point variables have been replaced with values from a precomputed look up table. This has also resulted in a big drop in the number DSP48s.

When we increase the size of the DFT, the size of the look up tables also increase.

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? What about the area results? Modify your testbench to accommodate this change to DFT interface. **You should use this modified interface for the remaining questions.**

RESPONSE TO QUESTION 3:

The testbench, the header and cpp files (**dft256_optimized2)** were modified such that the input and output variables were stored in separate arrays.

The dft function now takes in four arguments and the last two arguments are for the output variables.

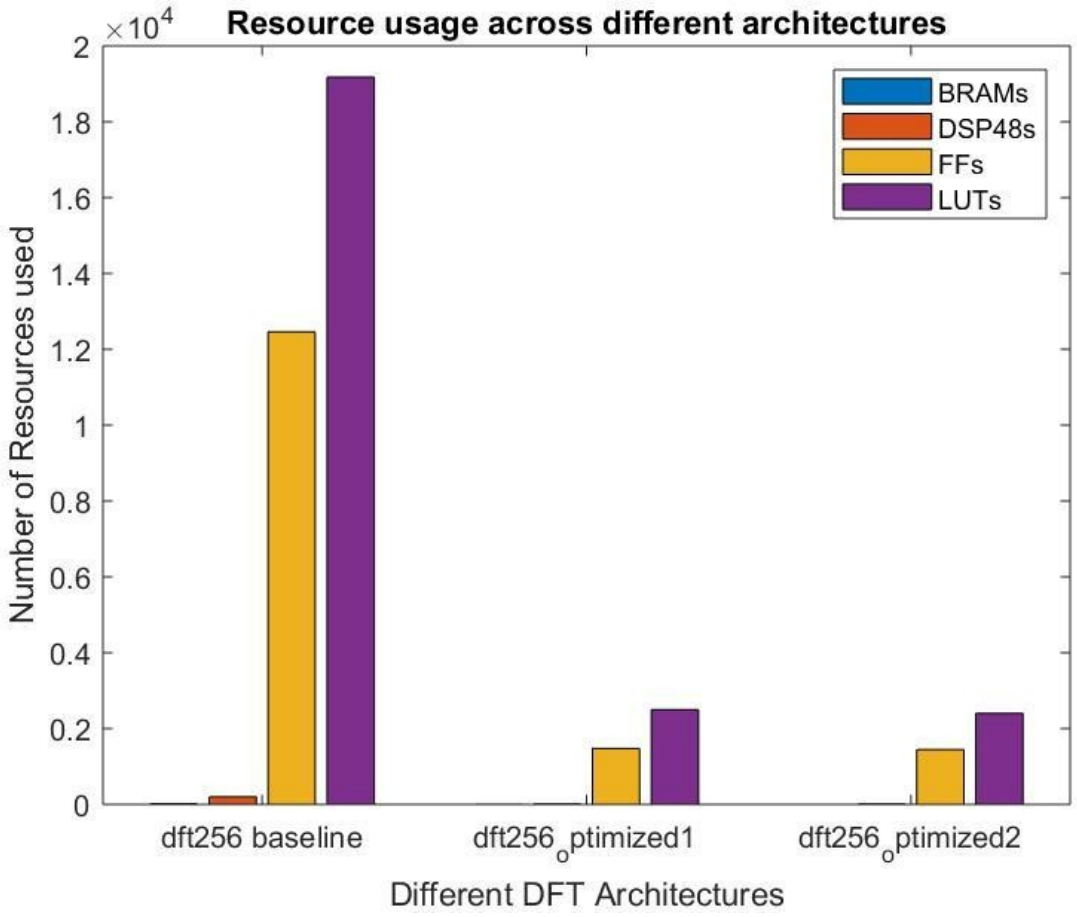
The observations that were made have been listed below in Table 2.

TABLE 2

Architecture	Throughput (KHz)	BRAMs	DSP48s	FFs	LUTs
dft256_baseline	0.0256	18	199	12462	19175
dft256_optimized1	0.1208	4	16	1474	2498
dft256_optimized2	0.1209	2	16	1445	2401

From the above table, it can be seen that the number of FFs and LUTs have reduced after this change. This is because of the removal of the last for loop which was used to transfer elements from the temporary variables back to the same input arrays.

FIG.1



Another change that occurred was the small reduction in the number of Block RAMs. This is because of not using the two temporary arrays any more.

QUESTION 4: Study the effects of loop unrolling and array partitioning on the performance and area. 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 matrix vector multiply operations per second (throughput) versus the unroll and array partitioning factor. Plot the same trend for area (showing LUTs, FFs, DSP blocks, BRAMs). What is the general trend in both cases? Which design would you select? Why?

RESPONSE TO QUESTION 4:

Loop unrolling and array partitioning are two important optimizations for DFT computation as this is a vector-matrix multiplication which is implemented using a for loop nested in another for loop.

Different factors were chosen for loop unrolling and array partitioning and the observations made from the results of synthesis are shown below.

dft256_optimized3 gives the results (and source, test files) for unrolling the inner loop with a factor of 128. The results obtained from other loop unroll factors have been tabulated.

dft256_optimized4 gives the results (and source, test files) for performing array partition (on the coefficient vectors) with a factor of 128. The results obtained from other array partition have been tabulated.

The following table (Table 3) shows the resources used and throughputs achieved for all the optimizations that have been performed.

TABLE 3

Architecture	Throughput (KHz)	BRAMs	DSP48s	FFs	LUTs
dft256_baseline	0.0256	18	199	12462	19175
Loop Unroll(factor =2)	0.1868	2	20	1932	3371
Loop Unroll(factor =16)	0.3139	2	20	2002	4731
Loop Unroll(factor =64)	0.3049	2	20	2252	9323
Loop Unroll(factor =128) (dft256_optimized3)	0.3067	2	20	2562	14767

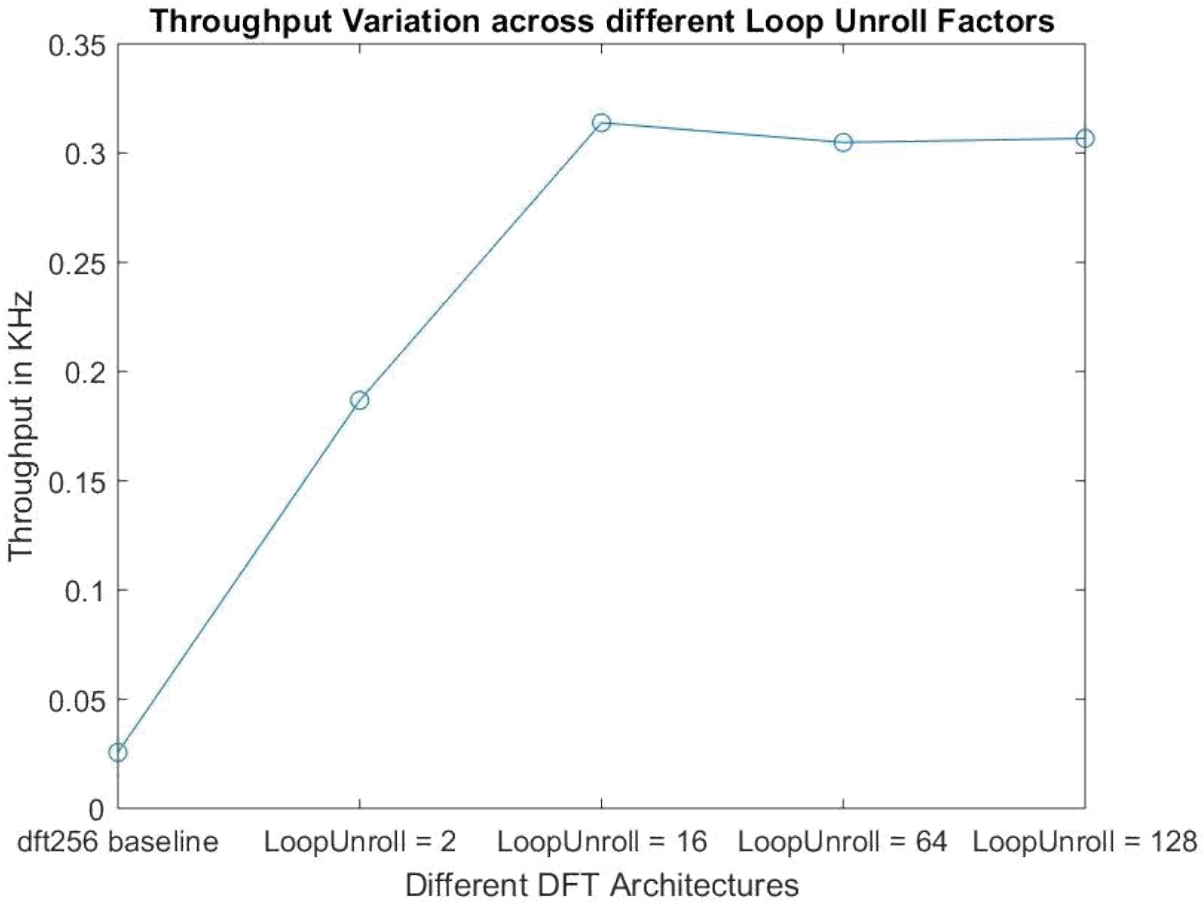


Fig. 2 Throughput Variation across different loop unroll factors

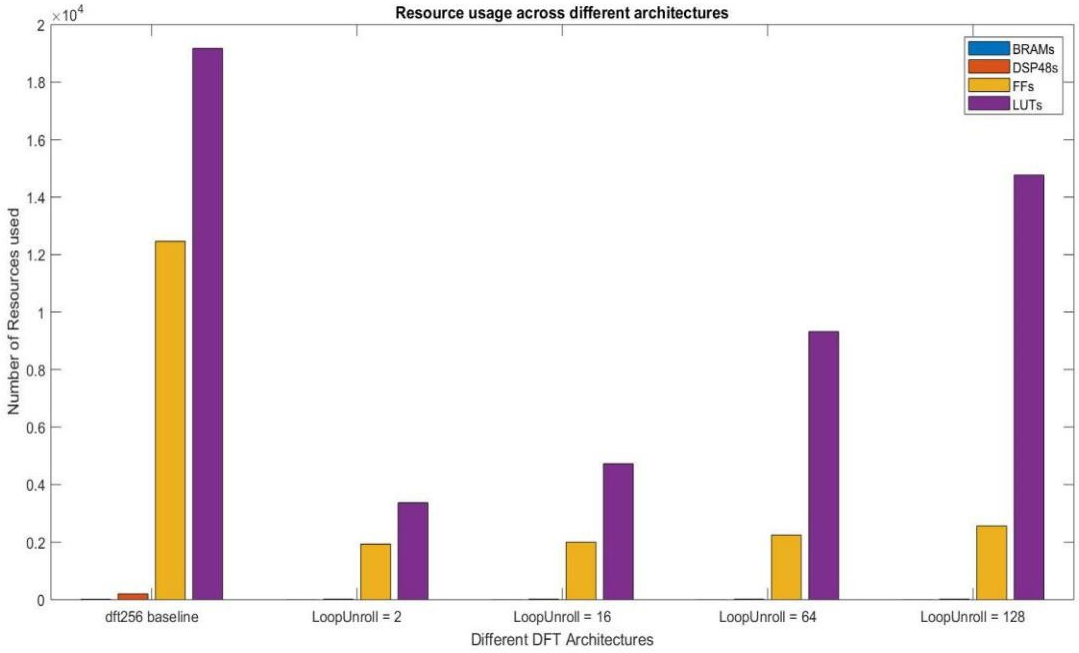


Fig. 3 Resources used for different DFT Architectures for block partition

From Table 3 and Fig. 2 it can be seen that the throughput initially increases as loop unrolling is introduced. However, we can see that the throughput saturates after increasing the loop unroll factor is increased beyond 16. The number of FFs and LUTs increases the unroll factor is increased but there is no increase in throughput after increasing the unroll factor above 16. Therefore if only loop unrolling is used, for this scenario a factor of 16 seems to be the optimum design in terms of throughput achieved and resources used.

TABLE 4

Architecture	Throughput (KHz)	BRAMs	DSP48s	FFs	LUTs
dft256_baseline	0.0256	18	199	12462	19175
Block Partition(factor = 2)	0.1235	4	16	1479	2456
Block Partition(factor = 16)	0.1235	0	16	2506	2778
Cyclic Partition(factor = 2)	0.07	4	8	945	5422
Cyclic Partition(factor = 16)	0.07	0	8	2028	6230
Array Partition + Loop Unroll	0.2877	0	36	6211	11333

Array partition on the coefficient matrix results in a notable increase in throughput when compared to the baseline design. However there does not seem to be any increase in throughput as the array partition factor is increased. The increase in resources can however be seen though.

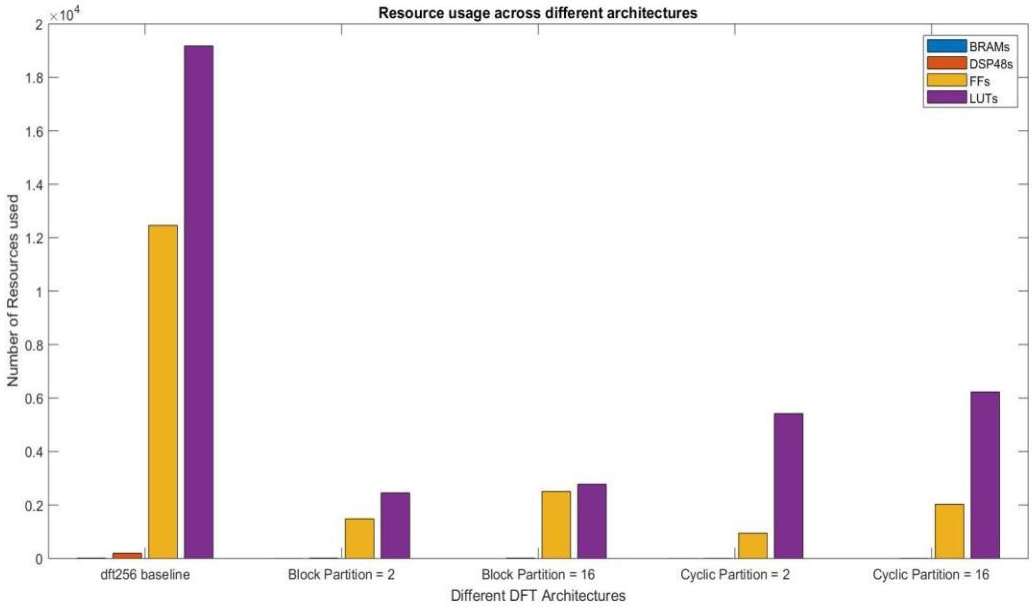


Fig. 4 Resource usage across DFT architectures for cyclic partition

As the array partition factor is increased from 2 to 16, the number of BRAMs goes to zero while FFs and LUTs are increased. This is because data is now partitioned into individual FFs instead of being stored in BRAMs. Block Partitioning seems to perform better (in terms of throughput achieved) than cyclic partition.

In the next design, (**dft256_optimized5**), loop unrolling was combined with pipelining the inner loop. The results of this architecture are discussed under the best design question

The next step that was attempted was the integration of array partition and loop unroll (**dft256_optimized6**). The results obtained from this step are presented in Table 4.

QUESTION 5: Please read dataflow section in the HLS user guide, and apply dataflow pragma to your design to improve throughput. You may need to change your code and make submodules. How much improvement can you make with it? How much does your design use resources? What about BRAM usage? Please describe your architecture with figures on your report. (Make sure to add dataflow pragma on your top function.)

RESPONSE TO QUESTION 5:

The dataflow pragma is used to implement task level pipelining. Therefore this requires the code to be restructured into submodules on which dataflow pipelining can be implemented. The code can be restructured into a set of functions that are called sequentially or a set of sequential for loops. The idea here is that while the first function or for loop is implementing, execution of the second function/for loop can start with partial results obtained from the first for loop.

Here, (**dft256_dataflow**) we have performed dataflow pipelining on the outer for loop. Instead of calling one inner for loop, the code has been changed to involve 2 for loops where coefficient calculation has been separated from the matrix-vector inner product calculation. In addition to this, to exploit pipelining, the second inner loop (second sub module) has been unrolled so that they can more efficiently make use of the coefficients obtained from the first sub module. The following table shows a comparison of resources used.

TABLE 5

Architecture	Throughput (KHz)	BRAMs	DSP48s	FFs	LUTs
dft256_baseline	0.0256	18	199	12462	19175
dft256_optimized1	0.1208	4	16	1474	2498
dft256_optimized2	0.1209	2	16	1445	2401
dft256_dataflow	0.232	4	20	2025	4115

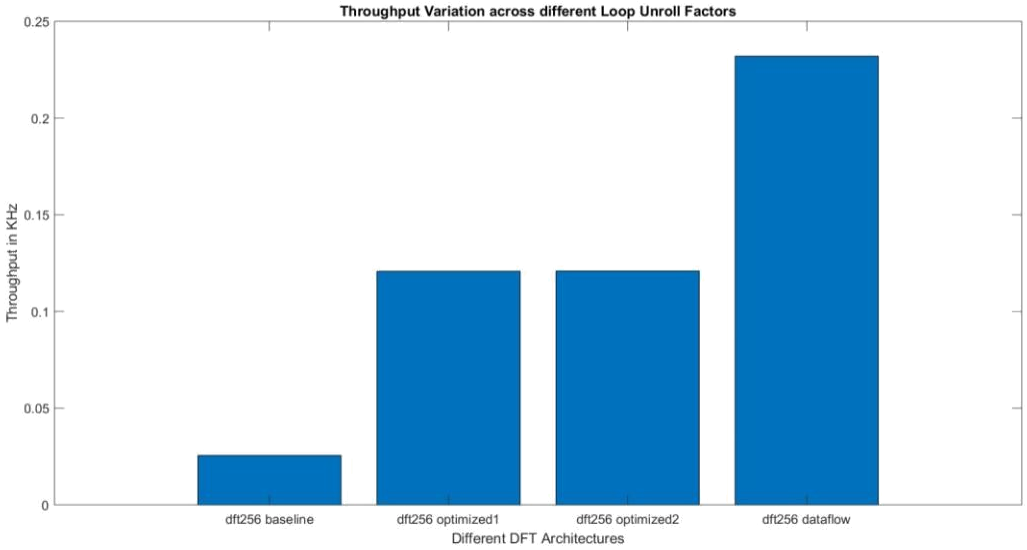


Fig. 5 Throughput variation across various Loop Unroll factors

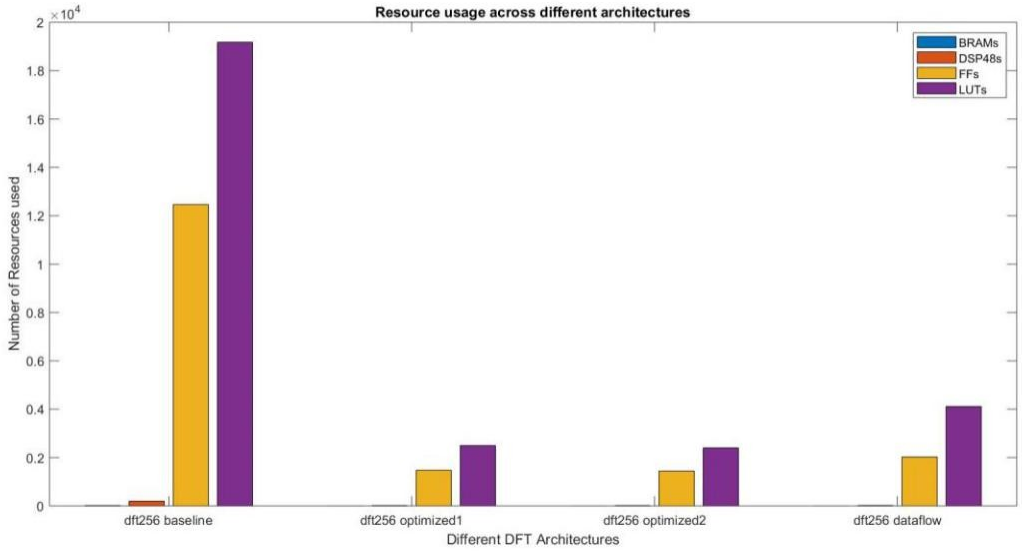


Fig. 6 Resource usage across DFT Architectures

We can see that making use of the dataflow pragma has improved the throughput but at the cost of increase in resources.

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 trade off you consider for the best architecture?

RESPONSE TO QUESTION 6:

In order to find a good architecture that achieves a good trade-off between resource usage and throughout, pipelining was considered. This is because the operations involved in DFT computation are sequential loads, multiplications followed by add and store operations. These individual blocks in the DFT chain can be pipelined to make more efficient use of the idle times of the various components. Pipelining has been carried out in conjunction with loop unrolling with a factor of 16 as it yielded the best results for us. (from Fig. 2)

TABLE 6

Architecture	Throughput (KHz)	BRAMs	DSP48s	FFs	LUTs
dft256_baseline	0.0256	18	199	12462	19175
dft256_optimized1	0.1208	4	16	1474	2498
dft256_optimized2	0.1209	2	16	1445	2401
Loop Unroll(factor =16)	0.3139	2	20	2002	4731
dft256_best	3.43	2	40	52153	35690
dft1024_best	0.0194	32	52	5294	10309

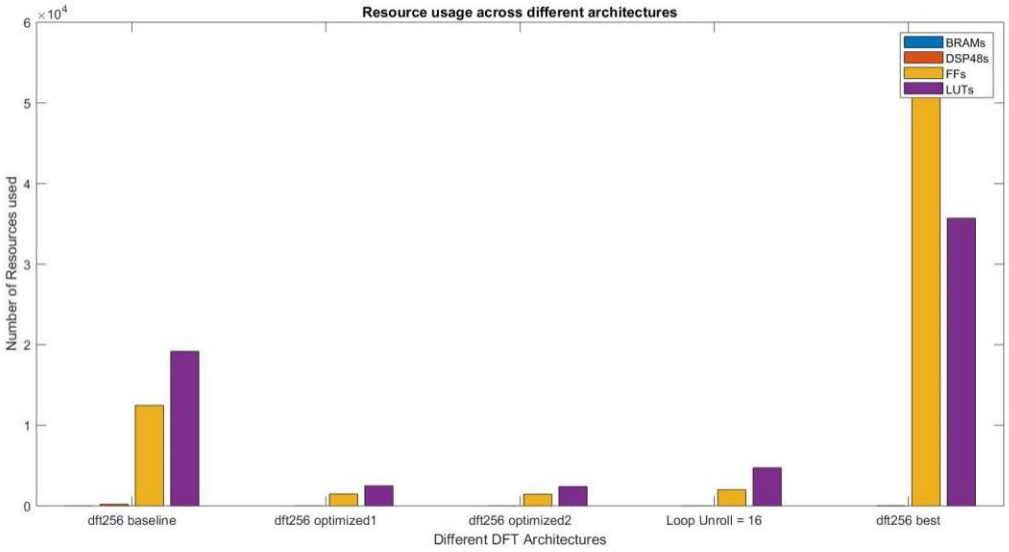


Fig. 7 Resource usage for Best Architecture

From Table 6 it can be seen that a combination of loop unrolling (factor = 16) and pipelining (**dft256_best**) has produced a considerable increase in throughput. As expected, the number of DSP48s, FFs and LUTs have also increased significantly. This has been chosen as the best design as the throughput increase is considerable when compared to earlier designs.

For implementing the 1024 point DFT (**dft1024_best**), loop unroll with factor 16 and array partition with factor 16 were the two pragmas that were used and the results obtained have been tabulated in Table 6.

QUESTION 7: (BONUS): If you create a design using hls::stream, you will get bonus points of Project 3. We do not provide any 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?

RESPONSE TO QUESTION 7:

The hls::stream pragma can be used whenever we require a FIFO structure for arrays, instead of storing entire arrays in blocks of RAM. In cases where we require sequential inputs/outputs from arrays, using hls::stream can lead to improved performance and lesser block RAMs when we use this pragma.

While implementing the hls::stream pragma in hardware, it is going to use a FIFO element of depth 1 for whichever variable we are using the pragma for. This means that if we are sequentially implementing our tasks, the depth of the FIFO element must be increased to hold the intermediate variables at the end of each task. Therefore it makes sense to use hls::stream with a dataflow pipelined architecture. This is because each intermediate result will be immediately fed into the next sub task and FIFOs with smaller depths can be used.

This architecture is expected to result in a decreased usage of memory resources. We have not implemented hls::stream but have only presented expected results for the same