

HLS Component Development

Using the Vitis Unified IDE 2024.1

Lab-3: Performance Improvement vs Resource Utilization

Mar-2025

Abstract

This lab introduces various techniques and directives which can be used in Vitis HLS to improve design performance as well as area and resource utilization. The design under consideration performs discrete cosine transformation (DCT) on an 8x8 block of data.

Objectives

After completing this lab, you will be able to:

- Add directives in your design
- Understand the effect of PIPELINE directive
- Understand the effect DATAFLOW directive on throughput
- Using ALLOCATION to limit the resources consumption


Introduction

This design implements a discrete cosine transformation (DCT), and it is provided as C source. The DCT algorithm expresses a finite sequence of data points in terms of a sum of different frequency cosine functions. DCT is commonly used to separate an image into spectral sub-bands. Like the FFT or DFT, there are many iterations performed in the algorithm.

Lab Steps

Step 1: Create the HLS component

In this step, you will launch the Vitis Unified IDE tool and create a new HLS component based on a provided DCT design.

- a. Click the *Vitis Unified IDE 2024.1* icon () from the desktop to launch the tool.
- b. Once the tool is launched, from the Vitis Components window, click the **Open Workspace** link.
- c. Browse to your desired location e.g. `/home/indi/MnRA/Lab3`
- d. Click **Ok** to open the new workspace. The tool relauches using the new workspace.
- e. Once the tool relauches, create the HLS component from File menu.
- f. Enter `DCT_NoOpt` in the **Component name** field.
- g. Add the source file (`dct.c`) from your desired location (e.g. `/home/indi/MnRA/Lab3/src/dct.c`).
- h. Chose Top function as `dct(short*, short*)`

- i. Add test bench files (`dct_test.c`, `in.dat`, `out.golden.dat`) as shown in figure below.

Create HLS Component - Empty HLS Component

Name and Location > Configuration File > **Source Files** > Hardware > Settings > Summary

Add Source Files

Specify design files, test bench files and flags for your component. You can also skip this step now and add sources later

DESIGN FILES		
FILE(S)	CFLAGS	CSIMFLAGS
Flags common to all files		
/home/indi/MnRA/Lab3/src/dct.c		

Top Function: [Browse](#)

TEST BENCH FILES	
FILE/FOLDER(S)	CFLAGS
Flags common to all files	
/home/indi/MnRA/Lab3/src/dct_test.c	
/home/indi/MnRA/Lab3/src/in.dat	
/home/indi/MnRA/Lab3/src/out.golden.dat	

[Cancel](#) [Back](#) [Next](#)

- j. Use **xczu7ev-ffvc1156-2-e** as part from the list.
- k. Enter **10** for the **clock** setting (1).
- l. Select **Vivado IP Flow Target** from the **flow_target** drop-down list (2).

1.1 Explore and understand DCT algorithm

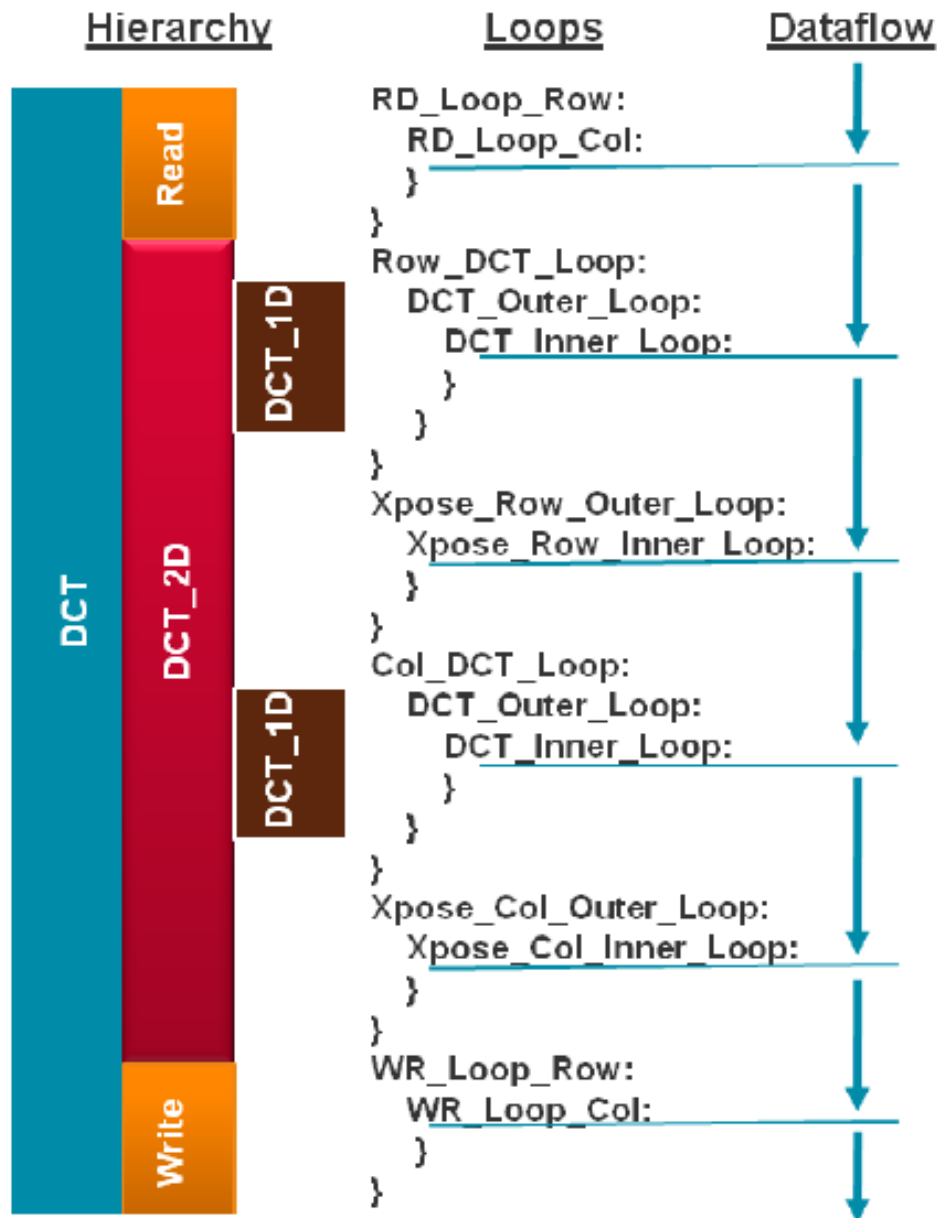
Double-click on the `dct.c` under the source folder to open its content in the information pane.

The top-level function `dct`, is defined at line 78. The DCT function leverages a 2D DCT algorithm by first processing each row of the input array via a 1D DCT, then processing the columns of the resulting array through the same 1D DCT. It calls the `read_data`, `dct_2d`, and `write_data` functions.

The `read_data` function is defined at line 54 and consists of two loops: `RD_Loop_Row` and `RD_Loop_Col`. The `write_data` function is defined at line 66 and consists of two loops to perform writing the result. The `dct_2d` function, defined at line 23, calls the `dct_1d` function and performs transpose.

Finally, the `dct_1d` function, defined in line 4, uses `dct_coeff_table` and performs the required function by implementing a basic iterative form of the 1D Type-II DCT algorithm.

The following figure shows the function hierarchy on the left-hand side, the loops in the order they are executed, and the flow of data on the right-hand side.



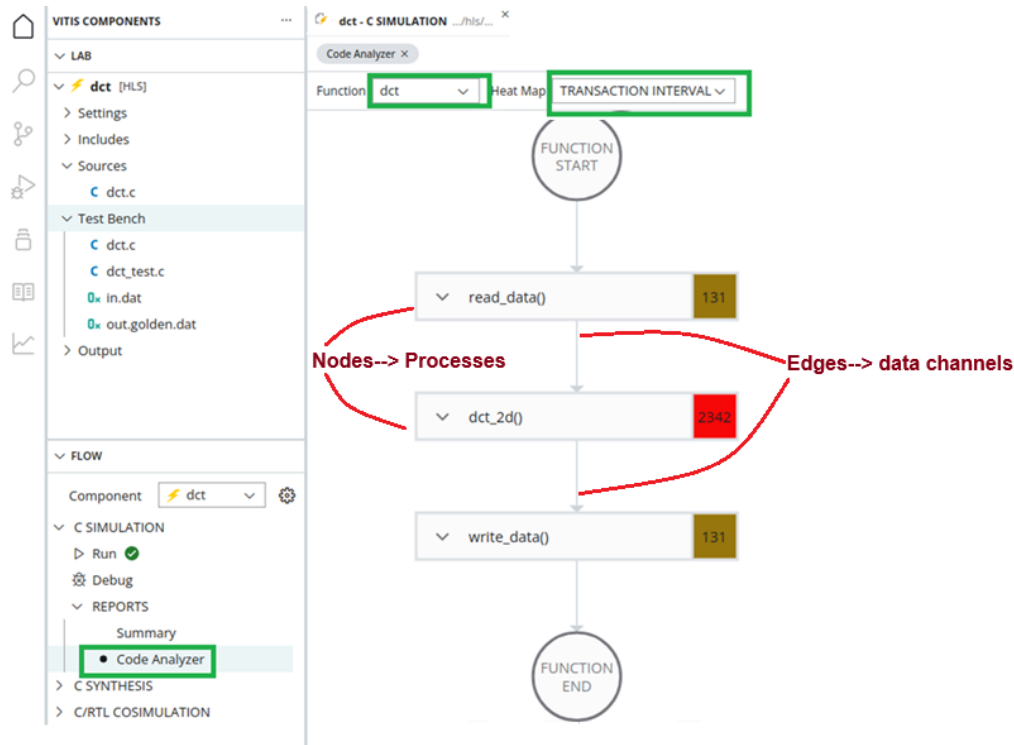
1.2 Simulate the design under C simulation and Code Analyzer.

- From the **Flow** view, hover over **C Simulation** and click the **Settings** icon (gear symbol) that appears.
- The **hls_config.cfg** configuration file opens in the Information pane.
- Select **csim.code_analyzer** to enable Vitis HLS Code Analyzer.
- Click **Run** under **C Simulation** from the **Flow** view.
- Observe the simulation log in the Output tab. Look for the **Results are good** and "C-simulation finished successfully" message at the end.

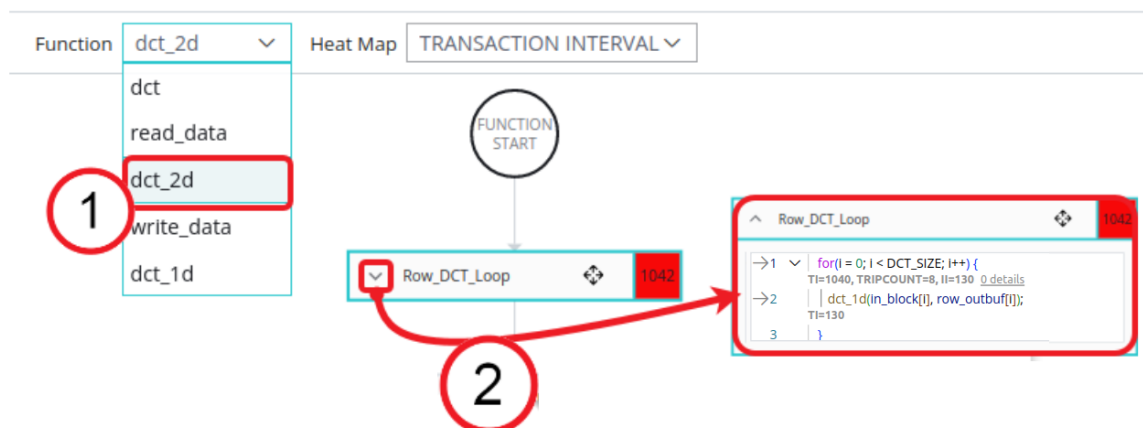
1.3 Analyze the code using Code Analyzer.

- Open **Code Analyzer** from Flow View under C Simulation > Reports > Code Analyzer.

- The Code Analyzer report initially displays a graph of the processes and channels defined by the top-level function of the component.
- With this graph view, nodes represent dataflow processes with their names being function names or loops labels (unnamed loops are named **Process #N**).
- Estimated transaction intervals (TI) are also presented next to the node name.
- Edges are the communication channels extracted from the variables of the design. You can see their names, the volume of data and the average throughout, expressed by default in bits per second, both estimated from the C test bench run.



- For the `read_data()` node, click the down arrow to show the code.
The performance analysis of Code Analyzer is overlaid in the code snippet. Here, this is shown after line 1.
If there are multiple lines, the overlaying happens in between lines of code.
- Select `dct_2d` from the Function drop-down list (1)
- For the `Row_DCT_Loop()` click the down arrow to show the code (2).



e. Observe the following:

- Code Analyzer estimates the performance of the innermost line first.
- The transaction interval is $TI = \text{TRIPCOUNT} * II$, so for `ROW_DCT_Loop()`, you get $TI = 8 * 130 = 1040$.
- You can repeat the investigations hierarchically up into the parent loop: the sum of all $TI(s)$ of all the statements in the loop body is computed.

Note about performance analysis numbers:

Performance analysis numbers are shown at the beginning of a loop, within its curly braces , or right after a function call site.

Note about reported trip count values:

- If loop bounds are compile-time constants, they are reported in Code Analyzer.
- Otherwise, if a trip count pragma is provided, this value is used by Code Analyzer.
- Otherwise, the trip count value is measured when running the C test bench.

In any case, the reported trip counts account for the loop unrolling: loop trip counts are divided by unrolling factors. Fully unrolled loops will always have a reported trip count of 1.

The code analyzer also guides you about ant performance bottlenecks.

- Select `dct_1d` from the Function drop-down list (1).
- Select **Performance Guidance** from the Heat Map drop-down list (2).
- For the `DCT_Outer_Loop()` click the down arrow to show the code (3).
- Under Line-1 at the end of TI information, click on the 1 detail. This will open guidance pane (5).



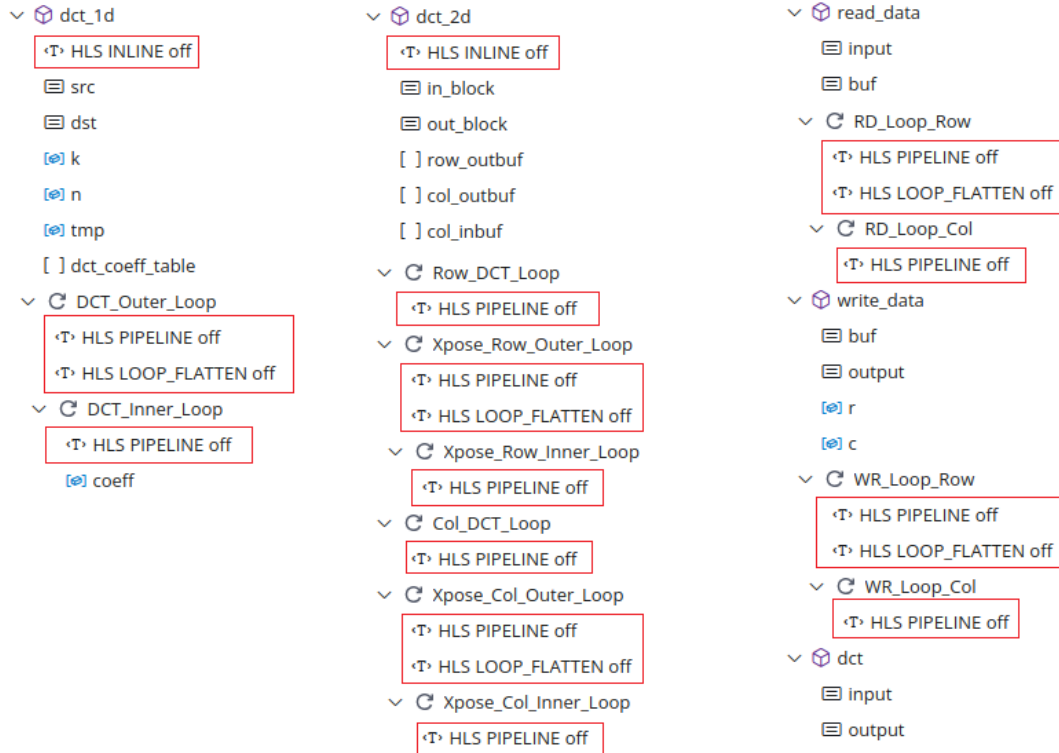
The tool has provided you with information that higher Initiation Interval (II) of this loop is due to cyclic dependence on variable `tmp`, `src` and `dst`. Such guidance helps you identify possible areas where some optimizations are required to be performed during Synthesis of the design for better performance.

Step 2: Synthesize and Analyze the Design

Next, you will synthesize the design without any optimization. The tool can automatically identify basic optimizations to enhance the performance of the design. These optimizations are applied during the Synthesis phase, which alters the behavior of the code flow. To understand the default behavior of the code after synthesis, it is advisable to first disable these default optimizations. You can apply or disable optimizations using "HLS Directives" view or by adding into configuration file (`hls.config.cfg`).

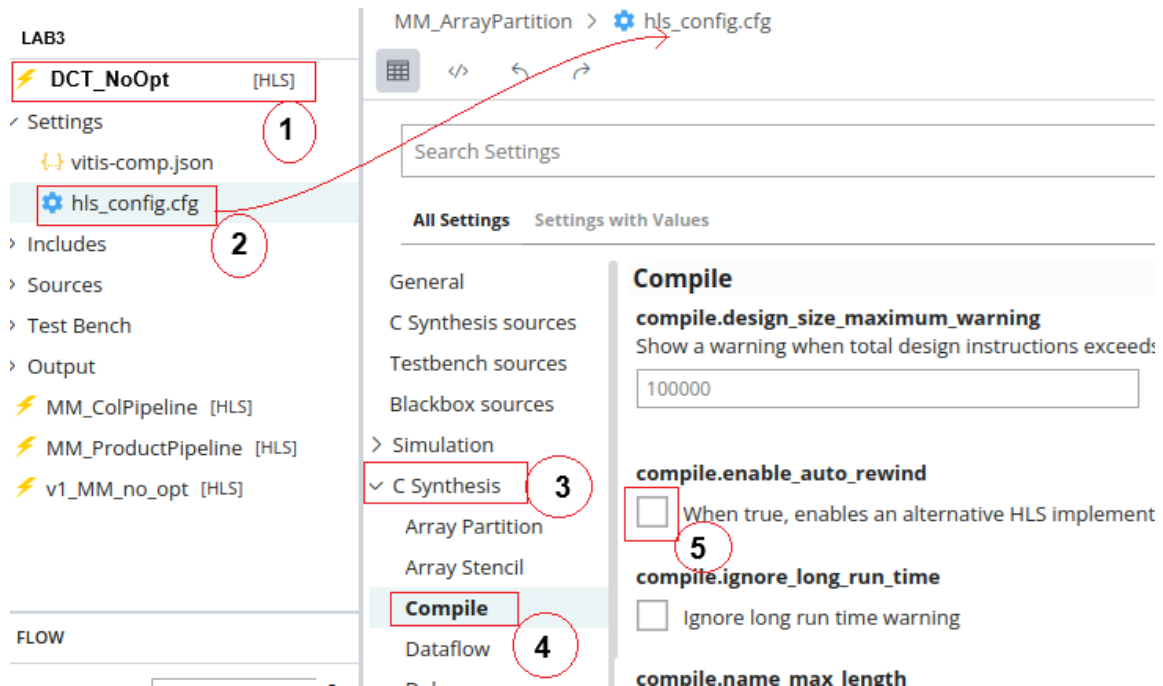
- Ensure the C source code `dct.c` is open/ visible in the Information pane.
- On the rightmost auxiliary pane, click on **HLS Directives**.
- Disable the INLINE optimizations on `dct_1d` and `dct_2d` functions.
- Similarly, disable PIPELINE for the following loops:
`DCT_Outer_Loop`, `DCT_Inner_Loop`, `Row_DCT_Loop`, `Xpose_Row_Outer_Loop`, `Xpose_Row_Inner_Loop`, `Col_DCT_Loop`, `Xpose_Col_Outer_Loop`, `Xpose_Col_Inner_Loop`, `RD_Loop_Row`, `RD_Loop_Col`, `WR_Loop_Row`, `WR_Loop_Col`.
- Similarly, disable loop flattening (LOOP_FLATTEN) for the following loops: `DCT_Outer_Loop`, `Xpose_Row_Outer_Loop`, `Xpose_Col_Outer_Loop`, `RD_Loop_Row`, `WR_Loop_Row`

HLS Directives pane should like below:



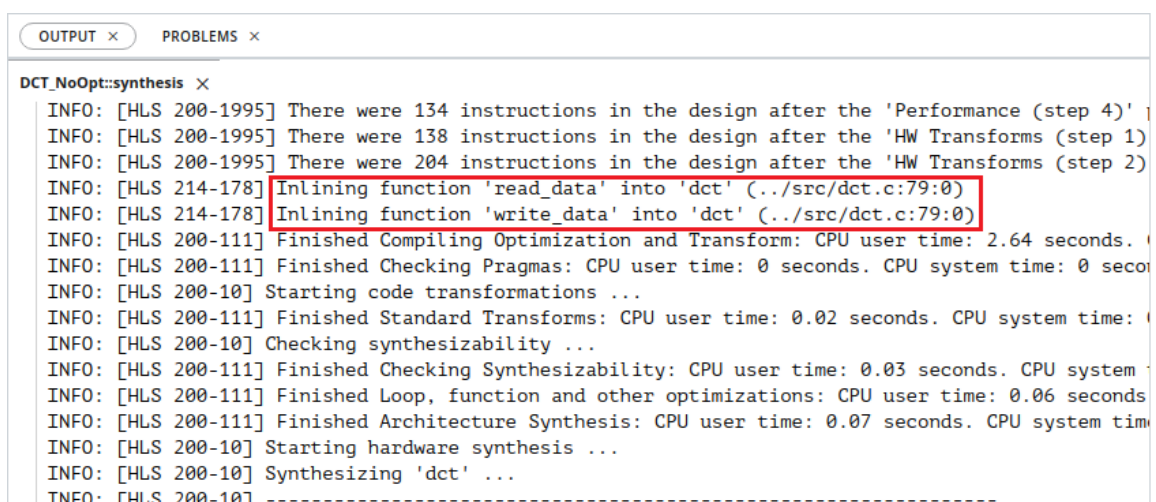
Since the tool auto-rewind the top-level loop to continually feed inner pipelined loop. To see the affect without auto-rewind option perform following steps.

- f. Open the configurations file `hls_config.cfg` under Settings in Vitis component view.
- g. Now under C Synthesis > Compile, **un-check** the box `compile.enable_auto_rewind`.



- h. Click the **Run** under **C SYNTHESIS** in the **Flow** Navigator to synthesize the design to RTL.

When synthesis is completed, several report files will become accessible in Vitis Component view under Synthesis Report section i.e. `DCT_NoOpt > Output > syn > report`. Note that the Synthesis Report section in the Explorer view only shows `dct_1d_csynth.rpt`, `dct_2d_csynth.rpt`, and `dct_csynth.rpt` entries. The `read_data` and `write_data` functions reports are not listed. This is because these two functions are **inlined**. Verify this by scrolling up into the Output view.



The Synthesis Report shows the performance and resource estimates as well as estimated latency in the design. Keep in mind that the design is not optimized. Open `dct_1d_csynth.rpt` and `dct_2d_csynth.rpt` files. The report for `dct_2d` clearly indicates that most of these design cycles (5716) are spent doing the row and column DCTs. Also the `dct_1d` report indicates that the latency is 337 clock cycles $((40+2)*8+1)$.

Step 3: Apply PIPELINE optimization directive

3.1 Create New Implementation of Component

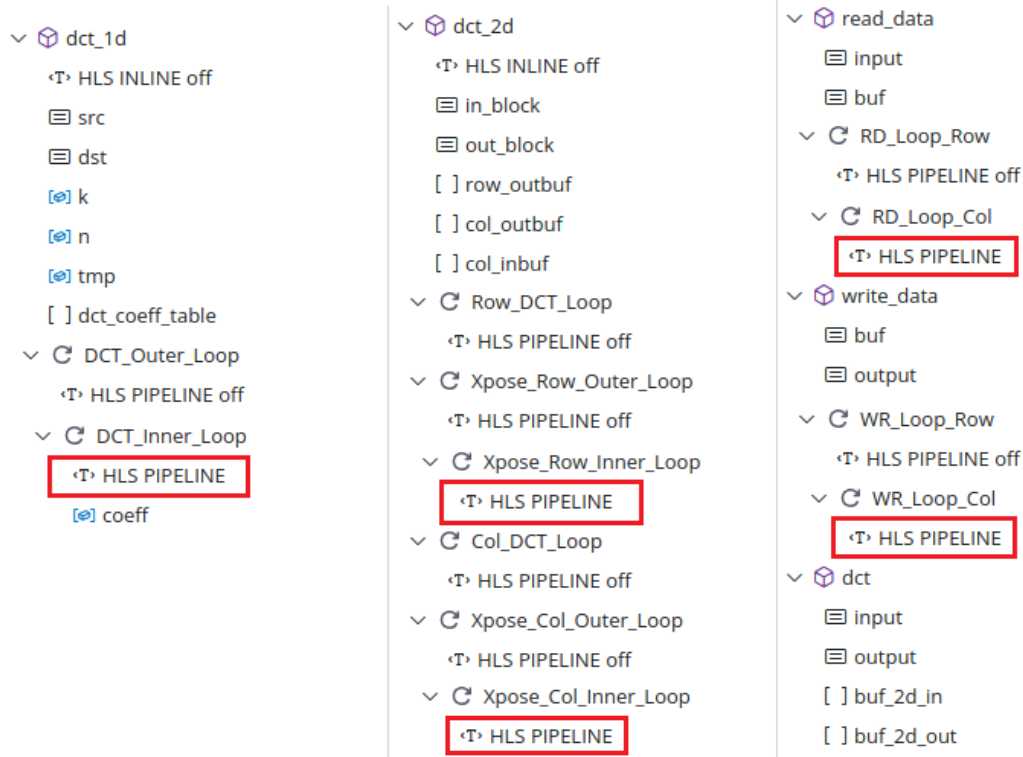
- a. Create a new implementation by Cloning the component `DCT_NoOpt`.
- b. Right click on the component name `DCT_NoOpt` in Vitis Component pane.
- c. Select **Clone Component**.
- d. Type the component name in `DCT_Pipeline` in the dialogue that appeared.

A copy of the component will be created with all the same configuration settings as the original component, including the clock, part, and optimization directives.

3.2 Applying Optimization Directives

Apply the PIPELINE directive to `DCT_Inner_Loop`, `Xpose_Row_Inner_Loop`, `Xpose_Col_Inner_Loop`, `RD_Loop_Col`, and `WR_Loop_Col` as below:

- a. Ensure the C source code `DCT.c` is visible in the Information pane.
- b. On the rightmost auxiliary pane, click on **HLS Directives**.
- c. Now, Select `DCT_Inner_Loop` loop.
- d. Hover over the PIPELINE directive and click on edit (pencil) symbol.
- e. Un-check the box in-front of `off(optional)`.
- f. Leave II (Initiation Interval) blank as Vitis HLS will try for an II=1, one new input every clock cycle.
- g. Keep all other options as default and click Ok.
- h. Repeat same process for `Xpose_Row_Inner_Loop`, `Xpose_Col_Inner_Loop`, `RD_Loop_Col`, and `WR_Loop_Col`.
- i. Similarly delete `LOOP_Flatten off` directive from the following loops:
`DCT_Outer_Loop`, `Xpose_Row_Outer_Loop`, `Xpose_Col_Outer_Loop`, `RD_Loop_Row`, `WR_Loop_Row`.
To delete these directives, in the HLS Directives tab, hover over the text `HLS LOOP_FLATTEN off` and click on the cross (X) symbol.
HLS Directives pane should like below:



- j. Click the Run under C Synthesis in the Flow Navigator to synthesize the design to RTL. Once the Synthesis is complete compare the performance (Interval and Latency, clock) and resource (FF, LUT, DSP, BRAM etc.) consumption of both implementations.
- k. Under the **View** menu select **HLS Compare Reports** to compare both implementations i.e. DCT_NoOpt and DCT.Pipeline.

HLS Compare Reports ×		
	⚡ DCT_NoOpt	⚡ DCT_Pipeline
✓Component Information (5)		
Part	xczu7eg-ffvc1156-2-e	xczu7eg-ffvc1156-2-e
Path	/home/indi/MnRA/Lab3/DCT_NoOpt	/home/indi/MnRA/Lab3/DCT_Pipeline
csynth clock target (ns)	10	10
csynth clock uncertainty (ns)	2.7	2.7
impl clock target (ns)		
✓Component Metrics (3)		
✓C SYNTHESIS (11)	✓	✓
Clock Target (ns)	10	10
Clock Uncertainty (ns)	2.7	2.7
Clock Achieved (ns)	2.474	3.633
Interval (cycles)	6008	1404
Latency (cycles)	6007	1403
Latency (ns)	60070	14030
BRAM	5	5
DSP	1	1
FF	228	395
LUT	898	1270
URAM	0	0

Performance parameters

Resource Consumption

Observe the following:

- Latency is reduced from 6007 to 1403 clock cycles due to pipelining.
- Interval is reduced from 6008 to 1404 clock cycles due to pipelining.
- FFs and LUTs utilization has increased due to pipelined stages.
- BRAM and DSP48E remained same.

3.3 Analyze your design

Open the synthesis report (csynth.rpt).

In the **Performance & Resource Estimation** expand the **dct** entry and observe the **RD_Loop_Row_RD_Loop_Col** and **WR_Loop_Row_WR_Loop_Col** entries. These are two nested loops flattened and given the new names formed by appending inner loop name to the outer loop name.

Similarly, expand the **dct_2d** entry and observe **Xpose_Row_Outer_Loop_Xpose_Row_Inner_Loop** and **Xpose_Col_Outer_Loop_Xpose_Col_Inner_Loop** entries. These are again two nested loops flattened and given the new names formed by appending inner loop name to the outer loop name.

You can also verify this by looking in the Console view message.

```

OUTPUT × PROBLEMS ×
DCT_Pipeline::synthesis ×
INFO: [HLS 200-111] Finished Loop, function and other optimizations: CPU user time: 0.06 seconds. CPU system time: 0.03 seconds. Elapsed time: 0.09
INFO: [HLS 200-2061] Successfully converted nested loops 'Xpose_Row_Outer_Loop'(..src/dct.c:37:4) and 'Xpose_Row_Inner_Loop'(..src/dct.c:39:7) in
INFO: [HLS 200-2061] Successfully converted nested loops 'Xpose_Col_Outer_Loop'(..src/dct.c:48:4) and 'Xpose_Col_Inner_Loop'(..src/dct.c:50:7) in
INFO: [HLS 200-2061] Successfully converted nested loops 'DCT_Outer_Loop'(..src/dct.c:13:4) and 'DCT_Inner_Loop'(..src/dct.c:15:7) in function 'dct' into
INFO: [HLS 200-2061] Successfully converted nested loops 'RD_Loop_Row'(..src/dct.c:59:4) and 'RD_Loop_Col'(..src/dct.c:61:7) in function 'dct' into
INFO: [HLS 200-2061] Successfully converted nested loops 'WR_Loop_Row'(..src/dct.c:71:4) and 'WR_Loop_Col'(..src/dct.c:73:7) in function 'dct' into
INFO: [XFORM 203-541] Flattening a loop nest 'Xpose_Row_Outer_Loop'(..src/dct.c:37:4) in function 'dct_2d'.
INFO: [XFORM 203-541] Flattening a loop nest 'Xpose_Col_Outer_Loop'(..src/dct.c:48:4) in function 'dct_2d'.
INFO: [XFORM 203-541] Flattening a loop nest 'DCT_Outer_Loop'(..src/dct.c:13:4) in function 'dct_1d'.
INFO: [XFORM 203-541] Flattening a loop nest 'RD_Loop_Row'(..src/dct.c:59:4) in function 'dct'.
INFO: [XFORM 203-541] Flattening a loop nest 'WR_Loop_Row'(..src/dct.c:71:4) in function 'dct'.
INFO: [HLS 200-111] Finished Architecture Synthesis: CPU user time: 0.08 seconds. CPU system time: 0.01 seconds. Elapsed time: 0.1 seconds; current
INFO: [HLS 200-10] Starting hardware synthesis ...

```

From the report, it can also be noticed that most of the latency is introduced by the `dct_2d` function which is 1270 out of total 1403 clock cycles.

Expand the `dct_2d` and `dct_1d` entry, notice that there still hierarchy exists in the module, i.e. it is not flattened the loops (`Row_DCT_Loop` and `Col_DCT_Loop`) are not merged with their respective inner loops. These loops cannot be merged because they are not perfect due to nontrivial logic. Both loops call `dct_1d` function before moving to their respective inner loops.

Step 4: Pipelining DCT_Outer_Loop of `dct_1d` function --> Unrolling DCT_Inner_Loop

Notice from Line-17 of `dct.c` code that `dct_1d` performs multiply and accumulate operations (Line-17: `tmp += src[n] * coeff;`).

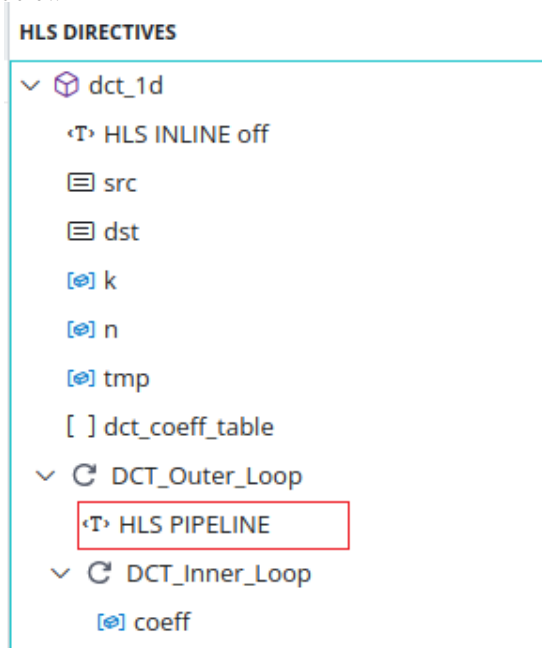
To improve the performance further we can unroll `DCT_Inner_Loop` to run multiple copies of multiply and add operations in parallel. For this move the `PIPELINE` `PIPELINE` directive from inner loop (`DCT_Inner_Loop`) to the outer loop (`DCT_Outer_Loop` of `dct_1d`). Pipelining the outer loop will result in complete unrolling of inner loop.

4.1 Create New Implementation of Component

- Create a new implementation by Cloning the component `DCT_Pipeline`.
- Right click on the component name `DCT_Pipeline` in Vitis Component pane.
- Select **Clone Component**.
- Type the component name in `DCT_PipelineOuter` in the dialogue that appeared.

4.2 Applying Optimization Directives

Remove the `PIPELINE` directive to `DCT_Inner_Loop`, and apply `PIPELINE` directive to `DCT_Outer_Loop` as below:-



4.3 Synthesize and Analyze the Design

- Click the Run under C Synthesis in the Flow Navigator to synthesize the design to RTL.
- Once the Synthesis is complete compare the performance (Interval and Latency, clock) and resource (FF, LUT, DSP, BRAM etc.) consumption of both implementations.
- Under the **View** menu select **HLS Compare Reports** to compare both implementations i.e. DCT_NoOpt, DCT_Pipeline and DCT_PipelineOuter.

	⚡ DCT_NoOpt	⚡ DCT_Pipeline	⚡ DCT_PipelineOuter
Component Information (5)			
Part	xczu7eg-ffvc1156-2-e	xczu7eg-ffvc1156-2-e	xczu7eg-ffvc1156-2-e
Path	/home/indi/MnRA/Lab3/DCT_NoOpt	/home/indi/MnRA/Lab3/DCT_Pipeline	/home/indi/MnRA/Lab3/DCT_PipelineOuter
C SYNTHESIS (11)	✓	✓	✓
Clock Target (ns)	10	10	10
Clock Uncertainty (ns)	2.7	2.7	2.7
Clock Achieved (ns)	2.474	3.633	3.59
Interval (cycles)	6008	1404	604
Latency (cycles)	6007	1403	603
Latency (ns)	60070	14030	6030
BRAM	5	5	5
DSP	1	1	8
FF	228	395	787
LUT	898	1270	1237
URAM	0	0	0

Observe the following:

- Latency has further reduced from 1403 to 603 clock cycles due parallel execution of `tmp += src[n] * coeff` operation.
- Similarly, Interval is reduced from 1404 to 604 clock cycles.
- FFs and LUTs utilization has further increased to accommodate parallel operations.
- DSP48E utilization is also increased from 1 to 8. Since the `DCT_Inner_Loop` completely unrolled, creating 8 (`DCT_SIZE = 8`) copies of `tmp += src[n] * coeff`. This resulted into 8x parallel multiplication operations, requiring 8 DSP48E.

Expand `dct_2d` in the Synthesis report under **Performance & Resource Estimates**, then expand `dct_1d2` till final stage i.e. `DCT_Outer_Loop`. Observe that the pipeline initiation interval (II) is one (1) cycle. This is a bit un-expected, as the function has to perform 8 reads. The reason that there is an II of 1 is because the coefficient table was automatically partitioned, resulting in 8 separate ROMs: this helped reduce the latency by keeping the unrolled computation loop fed.

Step 5: Apply DATAFLOW Directive

To improve throughput further we can apply DATAFLOW directive to top-level function `dct`.

5.1 Create New Implementation and Synthesis

- Create a new implementation by cloning the previous `DCT_PipelineOuter` and name it as `DCT_Dataflow`.
- Open `dct.c` code and go to HLS Directive pane.
- Select the function `dct` in the Directive pane and click on plus (+) symbol.

- d. Select DATAFLOW directive from drop down and add to Config file.
- e. Synthesis the new implementation.

5.2 Analyze the Design

When the synthesis is completed, open Synthesis report and observe the `dataflow` type pipeline at `dct` function.

Observe that throughput or interval is reduced further from 604 to 471.

- The Dataflow pipeline throughput indicates the number of clock cycles between each set of inputs reads (interval parameter). If this value is less than the design latency it indicates the design can start processing new inputs before the current input data are output.
- Note that the dataflow is only supported for the functions and loops at the top-level, not those which are down through the design hierarchy. Only loops and functions exposed at the toplevel of the design will get benefit from dataflow optimization.

In the synthesis report, look at the **Performance & Resource Estimates**. Observe that most of the latency and interval (throughput) is caused by the `dct_2d` function. The interval of the top-level function `dct`, is less than the sum of the intervals of the `read_data`, `dct_2d`, and `write_data` functions indicating that they operate in parallel and `dct_2d` is the limiting factor.

It can be seen that `dct_2d` is not completely operating in parallel as `Row_DCT_Loop` and `Col_DCT_Loop` were not pipelined. One of the limitations of the DATAFLOW optimization is that it only works on top-level loops and functions.

One way to have the blocks in `dct_2d` operate in parallel would be to pipeline the entire function. This however would unroll all the loops and can sometimes lead to a large area increase.

An alternative is to raise these loops up to the top-level of hierarchy, where DATAFLOW optimization can be applied, by removing the `dct_2d` hierarchy, i.e. inline the `dct_2d` function.

▼ Performance & Resource Estimates

⌵

⌴

☰

🧩

⏸

⚠

%

🔍

☒ Modules
☒ Loops
☒ Hide empty columns

MODULES & LOOPS	LATENCY(CYCLES)	LATENCY(NS)	ITERATION LATENCY	INTERVAL	TRIP COUNT	PIPELINED	BRAM	DSP	FF	LUT	URAM
<div> <div>⌵</div> <div>🧩</div> <div>dct (3)</div> </div>	604	6.040E3	-	471	-	dataflow	6	8	787	1241	0
<div> <div>></div> <div>●</div> <div>read_data (1)</div> </div>	66	660.000	-	64	-	loop auto-rew	0	0	26	172	0
<div> <div>⌵</div> <div>●</div> <div>dct_2d (4)</div> </div>	470	4.700E3	-	470	-	no	3	8	735	895	0
<div> <div>></div> <div>●</div> <div>Row_DCT_Loop (1)</div> </div>	168	1.680E3	21	-	8	no	-	-	-	-	-
<div> <div>🔗</div> <div>Row_Outer_Loop_Row_Inner_Loop</div> </div>	64	640.000	2	1	64	yes	-	-	-	-	-
<div> <div>●</div> <div>Col_DCT_Loop</div> </div>	168	1.680E3	21	-	8	no	-	-	-	-	-
<div> <div>🔗</div> <div>Col_Outer_Loop_Col_Inner_Loop</div> </div>	64	640.000	2	1	64	yes	-	-	-	-	-
<div> <div>></div> <div>●</div> <div>write_data (1)</div> </div>	66	660.000	-	64	-	loop auto-rew	0	0	26	172	0

Step 6: Apply INLINE Directive

Remember that we have turned the inlining Off for the functions `dct_2d` and `dct_1d`. We have to activate the INLINE on these two functions so that they can be merged in their calling loops and raised to top-level of hierarchy where DATAFLOW optimization can be applied.

6.1 Create New Implementation and Synthesis

- a. Create a new implementation by cloning the previous `DCT_Dataflow` and name it as `DCT_Inline`.
- b. Open `dct.c` code and go to HLS Directive pane.
- c. Select the function `dct_1d` in the Directive pane, hover over INLINE directive and click on edit (pencil) symbol.
- d. Un-check the box in-front of `off` (optional).

- e. Repeat above two steps for `dct_2d` function.
- f. Synthesis the new implementation.

6.2 Analyze the Design

When the synthesis is completed, open Synthesis report.

- Observe that the latency reduced from 604 to 409 clock cycles, and the Dataflow pipeline throughput drastically reduced from 471 to 64 clock cycles.
- The inlining of `dct_1d` function to the loops from which it is called, allows the loop nest to be flattened automatically. As a result the loops (`Row_DCT_Loop`, `Col_DCT_Loop`) are now flattened.
- Note also that the DSP48E usage has doubled (from 8 to 16). This is because, previously a single instance of `dct_1d` was used to do both row and column processing; now that the row and column loops are executing concurrently, this can no longer be the case and two copies of `dct_1d` are required: Vitis HLS will seek to minimize the number of clock cycles, even if it means increasing the area.
- Observe that the `dct_2d` entry is now replaced with `Loop_Row_DCT_Loop_proc`, `Loop_Xpose_Row_Outer_Loop_proc`, `Loop_Col_DCT_Loop_proc`, and `Loop_Xpose_Col_Outer_Loop_proc` since the `dct_2d` function is inlined.
- Also observe that all the functions are operating in parallel, yielding the top-level function interval (throughput) of 64 clock cycles.

Performance & Resource Estimates

<input checked="" type="checkbox"/> Modules <input checked="" type="checkbox"/> Loops <input checked="" type="checkbox"/> Hide empty columns											
MODULES & LOOPS	LATENCY(CYCLES)	LATENCY(NS)	ITERATION LATE	INTERVAL	TRIP COU	PIPELINED	BRAM	DSP	FF	LUT	URAM
▼ dct (6)	409	4.090E3	-	64	-	dataflow	3	16	1138	1705	0
> read_data (1)	66	660.000	-	64	-	loop auto-rew	0	0	27	172	0
> Loop_Row_DCT_Loop_proc (1)	70	700.000	-	64	-	loop auto-rew	0	8	380	361	0
> Loop_Xpose_Row_Outer_Loop_proc (1)	66	660.000	-	64	-	loop auto-rew	0	0	27	172	0
> Loop_Col_DCT_Loop_proc (1)	70	700.000	-	64	-	loop auto-rew	0	8	380	361	0
> Loop_Xpose_Col_Outer_Loop_proc (1)	66	660.000	-	64	-	loop auto-rew	0	0	26	185	0
> write_data (1)	66	660.000	-	64	-	loop auto-rew	0	0	26	172	0

Step 7: Apply ALLOCATION Directive

Although the previous implementation improves the throughput, it also consumes 16 DSPs. This is because the `dct_1d` function is inlined twice and consumes 8 DSPs per instance. We can limit the use of resources by using ALLOCATION. With help of ALLOCATION directive we can limit the amount of DSPs used by limiting the amount of multiplications.

7.1 Create New Implementation and Synthesis

- a. Create a new implementation by cloning the previous `DCT_Inline` and name it as `DCT_Allocation`.
- b. Open `dct.c` code and go to HLS Directive pane.
- c. Select the loop `DCT_Outer_Loop` in the Directive pane, and click on plus (+) symbol.
- d. Select ALLOCATION directive from drop-down list.
- e. Change the last filed `type` (optional) to `operation`.
- f. In the `instances` (required) field select `mul`.

g. Set the limit to 2 in `limit (required)` field as shown below

h. Synthesis the new implementation.

When the synthesis is completed, open Synthesis report. Notice that the DSP usage is decreased to 4 DSPs only. On the other hand, the latency (785) and throughput (256) have increased.

Performance & Resource Estimates											
<div> ⌵ ⌶ ≡ 🧩 ⚙️ ⚠️ % </div> <div> <input checked="" type="checkbox"/> Modules <input checked="" type="checkbox"/> Loops <input checked="" type="checkbox"/> Hide empty columns </div>											
MODULES & LOOPS	LATENCY(CYCLES)	LATENCY(NS)	ITERATION LAT	INTERVAL	TRIP COU	PIPELINED	BRAM	DSP	FF	LUT	URAM
<div>🧩 dct (6)</div> <div> <div>></div> <div>● read_data (1)</div> </div> <div> <div>></div> <div>● Loop_Row_DCT_Loop_proc (1)</div> </div> <div> <div>></div> <div>● Loop_Xpose_Row_Outer_Loop_proc (1)</div> </div> <div> <div>></div> <div>● Loop_Col_DCT_Loop_proc (1)</div> </div> <div> <div>></div> <div>● Loop_Xpose_Col_Outer_Loop_proc (1)</div> </div> <div> <div>></div> <div>● write_data (1)</div> </div>	785	7.850E3	-	256	-	dataflow	3	4	1350	2153	0
	66	660.000	-	64	-	loop auto-rew	0	0	27	172	0
	258	2.580E3	-	256	-	loop auto-rew	0	2	486	585	0
	66	660.000	-	64	-	loop auto-rew	0	0	27	172	0
	258	2.580E3	-	256	-	loop auto-rew	0	2	486	585	0
	66	660.000	-	64	-	loop auto-rew	0	0	26	185	0
	66	660.000	-	64	-	loop auto-rew	0	0	26	172	0

Additional resources and guides

If you want to know more about the topics from this lab or other optimizations techniques, the best starting point is Vitis High-Level Synthesis User Guide (UG1399).

The following chapters are the most interesting:

- HLS Programmers Guide
- Creating Vitis HLS Components
- HLS Optimization Directives
- HLS Pragmas (gives an overview of ALL the pragmas and options for the pragmas)
- Arbitrary Precision Data Types Library

Summary

In this tutorial, you learned:

- How pipelining of loops affects performance and resource usage.
- How to apply DATAFLOW directive and its effects on performance and resource consumption.

- How `INLINE` helps `DATAFLOW` optimization.
- How resources can be limited using `ALLOCATION` directive.