Computer Architecture Report

Lab 1: H/W accelerator design using Vitis HLS

Fraidakis Ioannis

November 2025

1 Introduction

This laboratory exercise focuses on the design and implementation of a hardware accelerator using High-Level Synthesis (HLS) with Xilinx Vitis. The primary objective is to develop an image processing module, specifically the `IMAGE_DIFF_POSTERIZE` kernel. This accelerator computes the absolute difference between two input images and applies a posterization thresholding logic to generate a simplified output image. The lab explores the HLS design flow, including C++ synthesis, baseline performance analysis, co-simulation, and performance optimization using pragmas to exploit parallelism and improve throughput.

2 Question 1: Design

The core functionality of the accelerator is to process two input matrices (images) A and B of dimensions $HEIGHT \times WIDTH$. The system calculates the absolute difference D for every pixel and maps it to specific intensity levels based on predefined thresholds.

2.1 Algorithmic Logic

The operation is performed pixel-wise for indices (i, j):

1. **Difference Calculation:** Calculate the absolute difference:

$$D = |A[i][j] - B[i][j]|$$

2. **Posterization Thresholding:** The output $C[i][j]$ is determined by:

$$C[i][j] = \begin{cases} 0 & \text{if } D < 32 \quad \text{(Black)} \\ 128 & \text{if } 32 \leq D < 96 \quad \text{(Gray)} \\ 255 & \text{if } D \geq 96 \quad \text{(White)} \end{cases}$$

In the HLS implementation, the two input images and the output image are represented as 1D arrays (or equivalent pointers) in memory.

2.2 Testbench Structure

The C/C++ testbench is responsible for:

- Generating or loading input images A and B (e.g. synthetic patterns).
- Executing pure software reference implementation to produce a golden output C_{ref}.
- Invoking the HLS function `IMAGE_DIFF_POSTERIZE` to produce C_{hls}.
- Comparing C_{hls} with C_{ref} and reporting mismatches.

3 Question 2: Baseline Synthesis

For the baseline implementation, the function is synthesized without any directives. This provides a reference design point in terms of latency and resource utilization.

The main synthesis metrics reported by Vitis HLS for $HEIGHT = WIDTH = 256$ are summarized in Table 1. The estimated clock period refers to the clock period constraint achieved during synthesis, while the worst-case latency corresponds to the maximum number of clock cycles required to process one 256×256 image pair.

Baseline Synthesis Results (256x256)

Table 1: Baseline (unoptimized) synthesis results for `IMAGE_DIFF_POSTERIZE`.

Metric	Value
Target Clock Period	10.00 ns
Estimated Clock	**5.093 ns**
Worst-Case Latency (cycles)	**65538**
Worst-Case Latency (time)	**0.655 ms**
Resource Utilization	
DSP48E	**0**
BRAM	**0**
Flip-Flops (FF)	**37**
LUTs	**152**

We observe that the baseline configuration already meets a fairly tight clock period (around 5.09 ns, better than the 10 ns target), but the latency is on the order of the total

number of pixels: roughly one clock cycle per pixel, plus control overhead. The resource usage is negligible relative to the available resources of the target device, as only a very small number of LUTs and flip-flops are used, and no DSPs or BRAMs are inferred.

4 Question 3 – Co-Simulation

To validate the functional correctness of the generated RTL, C/RTL co-simulation is executed in Vitis HLS. The same testbench used for C-simulation is reused, but now the HLS-generated RTL is simulated at the register-transfer level.

Co-Simulation Results

The co-simulation confirms that the RTL implementation is bit-accurate with respect to the software reference. No mismatches were observed between C_{hls} and C_{ref} for the tested input images. Table 2 summarizes the performance-related statistics reported by the co-simulation environment for the baseline design.

Table 2: Co-simulation performance summary (Baseline).

Metric	Value
Total Execution Time [ms]	**0.655**
Minimum Latency [cycles]	**65536**
Average Latency [cycles]	**65536**
Maximum Latency [cycles]	**65536**

The reported latency of 65536 cycles from RTL co-simulation is consistent with the expected behavior of the baseline design (one iteration per pixel, plus negligible control overhead) and matches closely the synthesis estimates.

5 Question 4: Optimization

This section details the steps taken to improve the performance of the accelerator, focusing on loop latency and throughput.

5.1 Scaling Analysis with Varying Width

In order to study the scaling behavior of the accelerator, experiments were performed by fixing $HEIGHT = 256$ and varying $WIDTH$ in the set $\{64, 128, 256, 512\}$. For each configuration, the design was synthesized and the corresponding latency and execution time were measured. Table 3 summarizes the results.

Latency and Execution Time Scaling (HEIGHT = 256)

Table 3: Latency and execution time scaling with varying `WIDTH`.

WIDTH	Latency [cycles]	Execution Time [ms]
64	**326**	**3905**
128	**614**	**6785**
256	**1190**	**12545**
512	**2342**	**24065**

Observations:

- The latency scales **linearly** with the total number of pixels ($N \times M$).
- Without pipelining, the tool executes iterations sequentially. As the width doubles, the execution time effectively doubles.
- This indicates an algorithmic complexity of $O(H \times W)$, which is expected for pixel-wise operations.

5.2 Optimization Directives

To accelerate the design, the implementation utilizes wide memory access types (`uint512_t`) to load 64 pixels simultaneously, combined with specific HLS pragmas.

Applied Directives

- **`#pragma HLS PIPELINE II=1`**: Applied to the `Main_Loop` to enforce an Initiation Interval (II) of 1. This ensures that the hardware initiates the processing of a new 512-bit data chunk (containing 64 pixels) every clock cycle, significantly increasing instruction-level parallelism and memory throughput.
- **`#pragma HLS UNROLL`**: Applied to the inner loop `Process_Loop`, which runs over the 64 bytes inside each 512-bit word. Full unrolling of this loop creates 64 parallel processing lanes, so all 64 pixels in a chunk are processed in the same cycle. This transformation is the main source of spatial parallelism and is responsible for the significant reduction in latency.
- **Interface directives (`m_axi`, `s_axilite`)**: The input (A, B) and output (C) ports are mapped to AXI4-Master interfaces (`m_axi`) with 512-bit data width and depth `IMAGE_SIZE/64`, each on a separate bundle (`gmemA`, `gmemB`, `gmemC`), to enable burst-mode access to off-chip global memory. The control signals are mapped to an AXI4-Lite interface (`s_axilite`) for interaction with the host. These directives enable burst transfers of 64 pixels per memory access, ensuring that the highly parallel datapath is kept fed with data and that results are written back efficiently, even though they do not directly change the core computational latency.

5.3 Best Implementation Results

The table below summarizes the performance of the final optimized design for the 256 × 256 resolution, compared to the baseline (unoptimized) design. The data are taken from the `solution_baseline` and `solution_accelerated` synthesis reports.

Optimized Design Results

Metric	Baseline	Optimized
Estimated Clock	5.093 ns	**7.300 ns**
Latency (cycles)	65536	**1190**
Total Execution Time	655.57 μs	**12.55 μs**
DSP48E	0	**0**
BRAM	0	**0**
FF	37	**15723**
LUT	152	**29638**

Speed-up Calculation:

$$Speedup = \frac{Latency_{Baseline}}{Latency_{Optimized}} = \frac{655.57}{12.55} \approx \mathbf{52.24}\times$$

From Table 1 and the optimized results, we see that the total execution time drops from 655.57 μs to just 12.55 μs, at the cost of a significant increase in LUT and FF usage. Nevertheless, the absolute utilization remains low with respect to the capacity of the `xcu200` device (about 7% LUTs and 1% FFs of SLR), making this trade-off very favorable for throughput-oriented applications.

6 Conclusion

In this laboratory exercise, a hardware accelerator for image differencing and posterization was specified, implemented, and evaluated using Vitis HLS. Starting from a straightforward C/C++ description of the algorithm, a baseline hardware implementation was obtained, and its performance and resource usage were characterized.

Subsequently, HLS directives such as `PIPELINE` and `UNROLL` were applied to expose additional parallelism and increase effective memory bandwidth. The optimized design achieved a speed-up of approximately 62.9× in terms of cycle latency (from 65538 cycles down to 1042 cycles), with still modest resource utilization on the target FPGA.

Overall, the lab demonstrates how high-level synthesis can be used to explore performance/area trade-offs systematically and highlights the importance of memory architecture and loop transformations in achieving high throughput.