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 for image processing using High-Level Synthesis (HLS) with Xilinx Vitis HLS. The primary objective is to develop the `IMAGE_DIFF_POSTERIZE` kernel, which computes the absolute difference between two grayscale images and applies three-level posterization to visualize difference magnitudes. The lab demonstrates the HLS design flow from C++ specification through RTL generation, including performance analysis, functional verification via co-simulation, and systematic optimization using compiler directives. The accelerator achieves high throughput by exploiting data-level parallelism through wide memory interfaces (512-bit).

Problem Statement

Given two grayscale input images A and B of dimensions $HEIGHT \times WIDTH$ pixels, the goal is to compute a posterized difference image C where each output pixel is classified into one of three intensity levels based on the magnitude of the pixel-wise absolute difference.

Design Objectives

- Implement a functionally correct FPGA accelerator synthesizable from C++
- Achieve high throughput through memory bandwidth optimization and parallelism
- Minimize latency for typical image sizes (e.g., 256×256)
- Maintain reasonable resource utilization on target FPGA device
- Validate correctness through comprehensive testbench verification

2 Algorithm Design and Implementation

2.1 Algorithm Specification

The IMAGE_DIFF_POSTERIZE operation processes images represented as linearized 1D arrays in row-major order. For each pixel position i, the algorithm performs:

1. **Absolute Difference Calculation:**

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

where $A_i, B_i \in [0, 255]$ are 8-bit grayscale pixel values.

2. **Three-Level Posterization:** The output $C[i]$ is determined by comparing $D[i]$ against two threshold values:

$$C[i] = \begin{cases} 0 & \text{if } D[i] < \texttt{THRESH\_LOW} = 32 \\ 128 & \text{if } 32 \leq D[i] < \texttt{THRESH\_HIGH} = 96 \\ 255 & \text{if } D[i] \geq 96 \end{cases} \tag{1}$$

This mapping provides intuitive visualization:

- **Black (0)**: Negligible difference
- **Gray (128)**: Moderate difference
- **White (255)**: Significant difference

2.2 Data Types and Constants

The implementation uses types defined in image_defines.h:

```c
typedef uint8_t pixel_t;                // 8-bit grayscale pixel
typedef ap_uint<512> uint512_t;         // 512-bit wide data type

#define HEIGHT 256
#define WIDTH 256
#define IMAGE_SIZE (HEIGHT * WIDTH)     // 65536 pixels

#define THRESH_LOW 32
#define THRESH_HIGH 96
```

Listing 1: Core type definitions from image_defines.h

The ap_uint<512> type from Xilinx's arbitrary precision library enables efficient packing of 64 pixels ($512 \div 8 = 64$) into a single memory word, crucial for achieving high memory bandwidth utilization.

2.3 Testbench Structure

The testbench (tb_image_diff.cpp) performs comprehensive functional verification:

- 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 Baseline C Synthesis and Analysis

The baseline implementation represents the straightforward translation of the algorithm to hardware without optimization directives. This provides a reference point for evaluating the effectiveness of subsequent optimizations.

3.1 Baseline Architecture

The baseline design (present in commented form in `image_diff_baseline.cpp`) features:

- **Sequential Processing:** Single loop iterating over all `IMAGE_SIZE` pixels
- **Scalar Memory Access:** One 8-bit pixel read/write per iteration

Without pipelining directives, HLS schedules operations sequentially within each loop iteration, resulting in multi-cycle latency per pixel.

3.2 Synthesis Results

Table 1 summarizes the synthesis metrics for the baseline configuration:

Baseline Synthesis Results (256×256 Image)

Table 1: Baseline synthesis metrics without optimization directives.

Metric	Value
Target Clock Period	10.00 ns
Estimated Clock Period	**5.093 ns**
Worst-Case Latency (cycles)	**65,538**
Worst-Case Latency (time)	**655.38 µs**
Resource Utilization	
DSP48E	**0**
BRAM_18K	**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 C/RTL Co-Simulation

Co-simulation validates the RTL generated by HLS against the C++ golden model by executing the synthesized design in an HDL simulator (Verilog/VHDL) and comparing results with the testbench.

Co-Simulation Performance (Baseline)

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 for the baseline design.

Table 2: RTL co-simulation latency measurements.

Metric	**Value**
Total Execution Time [ms]	**0.655 ms**
Minimum Latency [cycles]	**65,536**
Average Latency [cycles]	**65,536**
Maximum Latency [cycles]	**65,536**
Verification Status	**PASS**

The reported latency of 65536 cycles from RTL co-simulation is consistent with the expected behavior of the baseline design (one iteration per pixel).

5 Performance Optimization

This section describes the systematic optimization process to achieve high-throughput image processing by exploiting memory bandwidth and computational parallelism.

5.1 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.

Micro-Architectural Rationale

The optimized kernel operates on 512-bit words (`uint512_t`), each containing 64 packed 8-bit pixels. The outer `Main_Loop` iterates over these chunks, resulting in:

$$\frac{IMAGE\_SIZE}{64} = \frac{256 \times 256}{64} = 1024$$

iterations for the target image size. By applying `#pragma HLS PIPELINE II=1` to this loop, the compiler schedules the design such that a new 64-pixel chunk is accepted every clock cycle in steady state. The reported pipeline depth of 4 cycles corresponds to the internal combinational stages required to extract pixels:

1. **Stage 1:** Read `chunk_A` and `chunk_B` from memory (AXI read)
2. **Stage 2:** Bit-slice all 64 pixels in parallel
3. **Stage 3:** Compute 64 absolute differences and evaluate thresholds
4. **Stage 4:** Pack 64 posterized pixels into `chunk_C`, write to memory (AXI write)

Therefore, the total latency of the main loop is well approximated by:

$$Latency \approx N + Depth \approx 1024 + 4 \approx 1028 \text{ cycles},$$

which matches the C-synthesis estimate. Any remaining gap between synthesis and RTL co-simulation is attributed to AXI transaction overheads and memory interface effects.

Applied Directives

- **`ap_uint<512>` data packing (`uint512_t`)**: The optimized implementation introduces a wide type which packs **64 grayscale pixels** of 8 bits each into a single memory element. As a result, each global memory read loads 64 pixels simultaneously, and each write stores 64 output pixels at once. This choice aligns naturally with the AXI data width and enables high memory bandwidth utilization with minimal packing/unpacking overhead.

- **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, to enable efficient 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 complement the `ap_uint<512>` packing strategy by transferring one full 64-pixel chunk per beat, ensuring that the highly parallel datapath is continuously fed and that results are written back efficiently.

- **`#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 sustaining continuous throughput across the 1024 chunk iterations for a 256×256 image.

- **`#pragma HLS UNROLL`**: Applied to the inner loop `Process_Loop`, which iterates over the 64 bytes inside each 512-bit word. Full unrolling creates 64 parallel processing lanes, so all 64 pixels in a chunk are processed in parallel within the same cycle. This transformation is the main source of spatial parallelism and is responsible for the substantial reduction in latency.

5.2 Synthesis Results: Optimized Design

Table 3 compares the baseline and optimized implementations for 256×256 images.

Baseline vs. Optimized Synthesis Results

Table 3: Performance and resource comparison.

Metric	Baseline	Optimized
Estimated Clock Period	5.093 ns	**7.300 ns**
Latency (cycles)	65,538	**1,190**
Total Execution Time	655.57 µs	**12.55 µs**
Resource Utilization		
DSP48E	0	**0**
BRAM_18K	0	**0**
Flip-Flops (FF)	37	**15,723**
LUTs	152	**29,638**

Execution Time Improvement:

$$\text{Speedup}_{\text{time}} = \frac{655.57\ \mu\text{s}}{12.55\ \mu\text{s}} \approx \mathbf{52.24}\times \quad (2)$$

This substantial speedup comes at the cost of increased resource utilization:

- **LUT increase:** $152 \rightarrow 29,638$ ($195\times$ growth)
  - 64 parallel datapaths for absolute difference and thresholding
  - Pipeline registers for 512-bit data paths
  - AXI interface control logic
- **FF increase:** $37 \rightarrow 15,723$ ($425\times$ growth)
  - Pipeline stage registers for 4-stage pipeline
  - AXI burst state machines and address generators
  - Data buffering for 512-bit transfers
- **Clock period degradation:** $5.093ns \rightarrow 7.300$ ns
  - Longer combinational paths due to 64-way parallelism
  - Routing congestion from wide datapaths

Despite the resource increase, absolute utilization remains low on modern FPGAs:

- Xilinx Alveo U200 (xcvu9p): ∼1.2M LUTs, ∼2.5M FFs
- This design: 7% LUT, 1% FF utilization of SLR
- Headroom for multiple kernels or larger image sizes

5.3 Scaling Analysis

To understand how the design scales with problem size, experiments were conducted by fixing $HEIGHT = 256$ and varying $WIDTH \in \{64, 128, 256, 512\}$.

Latency Scaling with Image Width

Table 4: Latency and execution time scaling with varying `WIDTH`.

WIDTH	Total Pixels	Latency [cycles]	Time [µs]
64	16,384	**326**	**3.91**
128	32,768	**614**	**6.79**
256	65,536	**1,190**	**12.55**
512	131,072	**2,342**	**24.01**

The latency scales **linearly** with the total number of pixels ($H \times W$).

5.4 Explored Optimizations with Limited Benefit

Beyond the final set of optimizations presented above, we experimented with additional techniques commonly used in Vitis HLS. For the specific workload of a 256×256 image, these approaches did not provide a meaningful performance improvement.

Non-beneficial or Marginal Optimizations

- **Aggressive AXI tuning (burst/outstanding):** We increased the maximum burst length and the number of outstanding transactions (e.g. read burst up to 64 and read outstanding up to 32). C-synthesis showed identical latency, while LUT usage increased by 30% due to larger state machines.
- **Task-level `DATAFLOW`:** We implemented a streaming version that separates the design into `Read_Inputs`, `Compute_Diff`, and `Write_Output` stages connected by `hls::stream`. Although this is a standard pattern for overlapping memory and computation, our baseline optimized kernel already achieves **II=1** on 512-bit chunks. As a result, DATAFLOW did not reduce end-to-end latency for this small image and slightly increased the measured total latency ($1190 \rightarrow 1193$ cycles), while FF utilization increased by 30% due to FIFO instantiation for streams.

6 Conclusion

This laboratory exercise successfully demonstrated the complete HLS design flow for FPGA-based image processing acceleration. Starting from a high-level C++ algorithmic description, the `IMAGE_DIFF_POSTERIZE` kernel was systematically optimized to achieve $\approx 52.2\times$ speedup over the baseline implementation.