Computer Architecture

Laboratory Report

Lab 2: Image Processing Accelerator

with Sharpening Filter

Author

Fraidakis Ioannis

December 2025

Abstract

This report presents the design and optimization of an FPGA-based image processing accelerator implementing absolute difference, posterization, and sharpening filter operations. Three architectural variants were developed: **V1** (sequential with 2D buffers), **V2** (sequential with line buffers), and **V3** (dataflow streaming). Key findings include a **64× theoretical speedup** from V1 to V3 through dataflow parallelism, and identification of memory bandwidth as the primary bottleneck. Multi-bank DDR mapping via `connectivity.cfg` reduced V3 execution time from 11 μs to **5 μs**, doubling transfer rates from 8.9 to 18.9 GB/s.

Contents

1 Introduction

This laboratory exercise extends the previous lab by adding a 3×3 sharpening filter to the image processing pipeline. The accelerator now performs three operations in sequence:

1. **Absolute Difference**: $D[i][j] = |A[i][j] - B[i][j]|$
2. **Posterization**: Map D to discrete levels (0, 128, or 255)
3. **Sharpen Filter**: Apply a Laplacian-based 3×3 convolution kernel

Additionally, three different architectural implementations (V1, V2, V3) were developed to explore optimization strategies for throughput and resource utilization, analyzing trade-offs between memory access patterns and dataflow parallelism.

2 Algorithm Description

2.1 Processing Pipeline

The image processing pipeline consists of three stages:

1. **Difference Calculation:** For each pixel (i, j):

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

2. **Posterization Thresholding:**

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

3. **Sharpen Filter:** Apply Laplacian-based 3×3 kernel:

$$K = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

The convolution output is computed as:

$$C[i][j] = 5 \cdot P[i][j] - P[i-1][j] - P[i+1][j] - P[i][j-1] - P[i][j+1]$$

The result is then clamped to the valid pixel range:

$$C[i][j] = \min(255, \max(0, C[i][j]))$$

2.2 Memory Interface

All implementations use 512-bit wide AXI master interfaces (64 pixels per memory transaction). This aligns with the DDR memory bus width and maximizes memory bandwidth utilization, enabling efficient burst transfers to and from global memory.

3 Implementation Versions

This section describes the three accelerator implementations and architectural differences.

Version 1: Sequential Three-Stage Pipeline

File: `src_hw/accelerated_v1.cpp`

Architecture Overview:

- Three sequential stages that execute one after another
- Uses 2D local BRAM buffers (`C_tmp[HEIGHT][PADDED_WIDTH]`, `C_filt`)
- Full-frame buffering (entire image stored in on-chip memory)

Stage Execution:

1. **Stage 1 (Posterize):** Read 512-bit chunks from A/B, compute absolute difference, apply posterization (3-level quantization), store results to `C_tmp` buffer
2. **Stage 2 (Filter):** Pixel-by-pixel filtering with direct 2D array access
3. **Stage 3 (Pack):** Read from `C_filt`, pack into 512-bit words, write to DDR

Throughput Analysis:

- **Stage 1 & 3:** Process 64 pixels/cycle (limited by 512-bit memory interface)
- **Stage 2:** Processes only 1 pixel/cycle — *this is the performance bottleneck*
- **Total Latency:** $\approx$ `TOTAL_CHUNKS` + `HEIGHT` $\times$ `WIDTH` + `TOTAL_CHUNKS` cycles

Key Optimizations:

- `#pragma HLS ARRAY_PARTITION variable=C_tmp cyclic factor=3 dim=1`
  Enables parallel access to north/center/south rows in Filter stage
- `#pragma HLS ARRAY_PARTITION variable=C_tmp cyclic factor=64 dim=2`
  Enables 64-pixel parallel access in Posterize stage
- `#pragma HLS PIPELINE II=1` on all major loops
- `#pragma HLS UNROLL` for inner 64-pixel processing

Trade-offs:

- ✓ Simple to implement and debug — good starting point and reference baseline
- ✓ Pixel-by-pixel filtering avoids line buffer complexity (unlike V2/V3)
- ✗ Requires substantial BRAM for full-frame buffering
- ✗ Stages execute sequentially (no overlap)

Version 2: Sequential with Line Buffers

File: `src_hw/accelerated_v2.cpp`

Architecture Overview:

- Three sequential stages (same as V1) but with optimized filter stage
- Uses 1D packed 512-bit chunk buffers (`uint512_t C_tmp[TOTAL_CHUNKS]`) instead of V1's 2D pixel arrays
- Filter stage uses line buffers and sliding window for memory efficiency

Throughput Analysis:

- **All Stages:** Process 64 pixels/cycle (II=1 for 512-bit chunks)
- **Filter Latency:** Extended loop (`+CHUNKS_PER_ROW+1` iterations) to fill and drain the sliding window, due to convolution
- **Key Improvement:** Eliminates the 1-pixel/cycle bottleneck from V1 filtering

Filter Stage Implementation:

- **Line Buffers:** `uint512_t lb[2][CHUNKS_PER_ROW]`
  Stores 2 rows of 512-bit chunks for vertical neighbor access
- **Sliding Window:** `uint512_t win[3][3]`
  3×3 grid of chunks — center chunk `win[1][1]` contains pixels being filtered

West/East Neighbor Access: Handles chunk boundaries explicitly:

- West neighbor at $k = 0$: Access `win[1][0].range(511, 504)`
- East neighbor at $k = 63$: Access `win[1][2].range(7, 0)`

Key Optimizations:

- `#pragma HLS ARRAY_PARTITION variable=lb complete dim=1`
  Both line buffer rows accessible in parallel
- `#pragma HLS ARRAY_PARTITION variable=win complete dim=0`
  Entire 3×3 sliding window accessible simultaneously
- `#pragma HLS PIPELINE II=1` on all major loops

Trade-offs:

- ✓ 64× faster filter stage compared to V1's pixel-by-pixel approach
- ✓ Less BRAM usage (no array partitioning needed unlike V1)
- ✗ Complex sliding window logic with chunk boundary handling
- ✗ Still sequential execution (stages do not overlap)

Version 3: Dataflow Streaming Architecture (Ultra-Optimized)

File: `src_hw/accelerated_v3.cpp`

Architecture Overview:

- **True dataflow** with `#pragma HLS DATAFLOW`
- Three concurrent stages connected via `hls::stream<uint512_t>`
- Pipeline parallelism: all stages execute simultaneously on different data chunks

Stage Functions:

1. `compute_diff_wide()`: Reads A/B, computes posterized difference, writes to `stream_post`
2. `apply_filter_wide()`: Reads from `stream_post`, applies sharpen filter, writes to `stream_filt`
3. `write_result_wide()`: Reads from `stream_filt`, writes to output memory C

Stream Configuration:

```
hls::stream<uint512_t> stream_post("s_post");
hls::stream<uint512_t> stream_filt("s_filt");
#pragma HLS STREAM variable=stream_post depth=16
#pragma HLS STREAM variable=stream_filt depth=16
```

Expected Performance:

- **Throughput:** 64 pixels per clock cycle
- **Latency:** Theoretical minimum of 1,024 cycles (256×256 pixels $\div$ 64 pixels/cycle)

Key Optimizations:

- `#pragma HLS DATAFLOW`
  Enables concurrent execution: Total latency = max(Stage Latencies) instead of $\sum$
- `#pragma HLS PIPELINE II=1` inside each function
  Achieves single-cycle throughput per 512-bit chunk
- `#pragma HLS STREAM variable=... depth=16`
  Provides buffering slack to prevent producer-consumer stalls

Trade-offs:

- ✅ Highest throughput due to overlapped stage execution
- ✅ Memory-efficient streaming (no full-frame buffering)
- ❌ Most complex implementation

4 HLS Synthesis Results

Vitis HLS Synthesis Comparison (256×256 Image)

Table 1: Latency and resource utilization comparison across versions.

Metric	V1	V2	V3
Target Clock Period (ns)	10.00	10.00	10.00
Estimated Clock (ns)	**7.30**	**7.30**	**7.30**
Latency (cycles)	**67632**	**3104**	**1048**
Resource Utilization			
DSP48E	**18**	**0**	**0**
BRAM_18K	**256**	**30**	**0**
Flip-Flops (FF)	**21841**	**19876**	**22975**
LUTs	**54593**	**38816**	**40869**

Loop-Level Latency Breakdown

The Vitis HLS Performance & Resource Estimates provide detailed loop-level latency analysis, revealing the bottlenecks in each implementation.

Table 2: V1 loop-level breakdown: Sequential three-stage pipeline.

Module / Loop	Latency (cycles)	% of Total
`IMAGE_DIFF_POSTERIZE` (Top)	**67,632**	100%
`Pipeline_Posterize_Main_Loop`	1,027	1.5%
`Pipeline_Filter_Row_Filter_Col`	**65,551**	**97.0%**
`Pipeline_Pack_Main_Loop`	1,027	1.5%

V1 Bottleneck Analysis:

The filter stage (`Pipeline_Filter_Row_Filter_Col`) accounts for **97%** of total latency. This is because:

- The nested loop processes **one pixel per cycle** (II=1 on inner loop only)
- Total filter cycles = HEIGHT × WIDTH = 256 × 256 = 65,536 cycles

Optimization Opportunity: The posterize and pack stages already leverage 512-bit parallelism (64 pixels/cycle). Extending this to the filter stage could reduce it from 65,536 to just 1,024 cycles

Table 3: V2 loop-level breakdown: Sequential with line buffer optimization.

Module / Loop	Latency (cycles)	% of Total
`IMAGE_DIFF_POSTERIZE` (Top)	**3,104**	100%
`Pipeline_Posterize_Loop`	1,027	33.2%
`Pipeline_Filter_Loop`	1,042	33.6%
`Pipeline_Write_Loop`	1,027	33.2%

V2 Optimization Strategy:

By switching to line buffers and a sliding window approach, V2 achieves:

- **63× faster** filter stage (1,032 vs 65,551 cycles)
- Balanced stage latencies: Posterize (1,027), Filter (1,042), Write (1,027)

Optimization Opportunity: With nearly identical stage latencies (∼1,030 cycles each), the three stages could run *concurrently* using HLS streams and `DATAFLOW`. This would reduce total latency from $\sum = 3,104$ cycles to $\max = 1,042$ cycles, a **3× improvement**. The key change is replacing BRAM intermediate buffers with `stream` FIFOs and refactoring each stage into a separate function.

Table 4: V3 module-level breakdown: Dataflow streaming architecture.

Module / Function	Latency (cycles)
`IMAGE_DIFF_POSTERIZE` (Top)	**1,049**
`compute_diff_wide`	1,035
`apply_filter_wide`	**1,042**
`write_result_wide`	1,035

V3 Dataflow Parallelism:

With `#pragma HLS DATAFLOW`, all three functions execute **concurrently**. Instead of waiting for one stage to complete before starting the next, stages process different chunks simultaneously.

The total latency equals the **slowest stage** (1,042 cycles for `apply_filter_wide`) plus pipeline fill/drain overhead (∼7 cycles), achieving **1,049 cycles**, only **2.4% above** the theoretical minimum of 1,024 cycles for a 256×256 image.

Key Insight: Balanced stage latencies are critical for dataflow efficiency. If one stage were significantly slower, it would bottleneck the entire pipeline.

5 Vitis IDE Execution Results

The following results are obtained from running Emulation-HW using Vitis IDE.

5.1 Kernel Execution Statistics

Kernels & Compute Units → Kernel Execution

Table 5: Kernel execution metrics from Vitis IDE.

Metric	V1	V2	V3
Kernel Name	IMAGE_DIFF	IMAGE_DIFF	IMAGE_DIFF
Clock Frequency (MHz)	300	300	300
Compute Unit Time (μs)	**230**	**15**	**11**
Kernel Execution Time (μs)	233	22	16

Source for clock frequency: `hw_link` → `Emulation-HW` → `binary_container` → `Link Summary`. Target: 300 MHz, Estimated: **411 MHz**.

Compute Unit Time vs. Kernel Execution Time:

- **Compute Unit (CU) Time:** Measures the time the hardware compute unit is actively executing, from when the CU starts processing to when it signals completion. This reflects the *pure hardware execution time* including any memory stalls within the CU.
- **Kernel Execution Time:** Measures the total time from when the host issues the kernel launch (e.g., `clEnqueueNDRangeKernel`) to when the kernel completes. This includes CU Time *plus* software overhead (eg kernel setup).

The difference represents the runtime/launch overhead, which is $\approx$ *constant* regardless of kernel complexity. For faster kernels, it becomes a larger fraction of total time.

5.2 Kernel Data Transfers

Kernel Data Transfers → Top Kernel Data Transfer

Table 6: Top kernel data transfer statistics.

Metric	V1	V2	V3
Total Transfers	**64**	**64**	**64**
Transfer Rate (MB/s)	**9595.3**	**9595.3**	**8868.2**
Avg Transfer Size (Bytes)	**1024**	**1024**	**1024**
Link Utilization (%)	**25**	**25**	**25**

5.2.1 Understanding Kernel Transfer Metrics

Mechanism: AXI Burst Inference

The discrepancy between the code's 64-byte read and the profiler's 1024-byte transfer is due to **Burst Inference**. The HLS compiler analyzes the access pattern in the loop:

```
for (int i = 0; i < TOTAL_CHUNKS; i++) { ... A[i] ... }
```

Because memory addresses are accessed sequentially $(i, i+1, i+2 \dots)$ inside a pipelined loop (II=1), the memory controller groups multiple logical requests into a single physical transaction to amortize control overhead.

Deriving the 1024-Byte Transfer Size

The AXI interface uses a *Burst Length* of 16 beats to saturate bandwidth:

- **1 Beat (Logical Read):** Defined by `uint512_t` = 512 bits = **64 Bytes**.
- **1 Burst (Physical Tx):** The controller coalesces 16 beats per transaction.
- **Calculation:** 16 beats × 64 bytes/beat = **1024** Bytes.

Verification against Profiler Data:

Total Image Size: 256×256 pixels $= 65,536$ Bytes (64 KB)
Total Transfers: $65,536$ Bytes $\div$ 1024 Bytes/Burst = **64** Transfers

Why V1 & V2 have same Transfer Rates and higher than V3?

V1 and V2 share the same **sequential execution model**:

1. Read all input data (A, B) → store in local BRAM
2. Process (posterize, filter)
3. Write all output (C) to DDR

This creates **non-overlapping, sequential memory bursts** that fully utilize the AXI bus during each phase, achieving maximum burst efficiency (**9595.3 MB/s**).

V3 uses `#pragma HLS DATAFLOW`, making all stages run **concurrently**:

- `compute_diff_wide` is **reading** from A/B
- `write_result_wide` is **writing** to C
- Both compete for the **same memory controller** simultaneously

This **memory contention** reduces the effective transfer rate to **8868.2 MB/s**. However, the trade-off is worthwhile, since V3's overlapped execution still achieves the **fastest execution time (11 μs)** .

Transfer Efficiency = 25%:
Vitis calculates transfer efficiency using this formula:

$$\text{Efficiency} = \frac{\text{Average Bytes}}{\min(\text{Memory Byte Width} \times 256, 4096)}$$

For V3:

- Memory Byte Width: $512 \div 8 = 64$ bytes
- Memory Width $\times$ 256: $64 \times 256 = 16384$ bytes
- $\min(16384, 4096) = \mathbf{4096}$ bytes (DDR page size limit)
- Efficiency: $1024 \div 4096 = \mathbf{25}\%$

5.3 Host Data Transfers

Host Data Transfer → Host Transfer

Table 7: Host-to-device and device-to-host transfer statistics (same for all versions).

Metric	Host to Device	Device to Host	Unit
Transfer Count	**1**	**1**	–
Transfer Rate	**3.151**	**3.135**	MB/s
Avg Transfer Size	**131,072**	**65,536**	Bytes

Understanding Host vs Kernel Transfer Rate Difference

Dramatic Rate Difference:

Host ↔ Device:	∼3 MB/s	(PCIe path)
Kernel ↔ DDR:	∼9000 MB/s	(On-chip AXI path)
Ratio:	∼3000×	slower for host transfers

Why Such a Large Difference?

1. **Hardware Emulation vs Real Hardware:** The ∼3 MB/s rate is characteristic of `hw_emu` mode, where PCIe transfers are *simulated in software*. On real FPGA hardware, PCIe Gen3 x16 achieves 8–12 GB/s.
2. **Different Memory Paths:**
   - **Kernel ↔ DDR:** Uses on-chip 512-bit wide AXI bus directly connected to DDR memory controller
   - **Host ↔ Device:** Traverses PCIe bus with DMA engines

Host Transfer Size Breakdown:

Host → Device (WRITE):	**131072 Bytes**
Buffer A (input image 1):	65536 Bytes
Buffer B (input image 2):	65536 Bytes
Device → Host (READ):	**65536 Bytes**
Buffer C (output image):	65536 Bytes

Buffer size = TOTAL_CHUNKS × PIXELS_CHUNK = 1024 × 64 = 65536 Bytes

6 Performance Comparison Summary

Overall Performance Comparison

Table 8: Summary comparison of all three implementations.

Characteristic	V1	V2	V3
Architecture	Sequential	Sequential	Dataflow
Buffer Type	2D Arrays	512-bit Chunks	Streams
Filter Implementation	Direct 2D	Line Buffer	Line Buffer
Stage Parallelism	None	None	Full Overlap
HLS Latency (cycles)	**67632**	**3104**	**1048**
FPGA Exec Time (μs)	**230**	**15**	**11**
BRAM Usage	High	Medium	Low
Implementation Complexity	Low	Medium	High

6.1 Theoretical Speed-up Calculation

V3 vs V1 Speed-up:

$$\text{Speedup} = \frac{\text{Latency}_{V1}}{\text{Latency}_{V3}} = \frac{67632}{1048} \approx \mathbf{64.5}\times$$

V3 vs V2 Speed-up:

$$\text{Speedup} = \frac{\text{Latency}_{V2}}{\text{Latency}_{V3}} = \frac{3104}{1048} \approx \mathbf{2.96}\times$$

6.2 Theoretical vs. Actual Performance Analysis

V3 Performance Discrepancy: Compute-Bound vs Memory-Bound

HLS Latency:	~1,050 cycles
FPGA Frequency:	300 MHz
Theoretical Time:	$1050 \div 300$ MHz $\approx$ **3.5** μs
Measured Time:	**11** μs
Discrepancy:	~3× slower than expected

Root Cause: Memory Bandwidth Bottleneck

The V3 kernel is **memory-bound**, not compute-bound. The discrepancy arises because the kernel's memory demands exceed what a single DDR bank can provide.

Alveo U200 Memory Specifications

Memory Type:	DDR4
Configuration:	4 banks × 16 GB (64 GB Total)
Max Data Rate:	2400 MT/s (MegaTransfers/second)
Total Bandwidth (All 4 Banks):	77 GB/s

Peak Rate Per Channel:

$$\text{Bandwidth} = 2400 \text{ MT/s} \times 8 \text{ bytes} = 19,200 \text{ MB/s} = \mathbf{19.2} \text{ GB/s}$$

Source: AMD Alveo U200 Data Sheet (DS962)

Kernel Bandwidth Demand vs. Hardware Supply

V3 Architecture Demand:
With `#pragma HLS DATAFLOW`, all three stages execute **simultaneously**:

- `compute_diff_wide`: Reading from `gmemA` (Image A)
- `compute_diff_wide`: Reading from `gmemB` (Image B)
- `write_result_wide`: Writing to `gmemC` (Image C)

Each AXI port is 512-bits wide. At 300 MHz:

$$\text{Bandwidth per port} = \frac{512 \text{ bits}}{8} \times 300 \text{ MHz} = \mathbf{19.2} \text{ GB/s}$$

Total Demand:

$$3 \text{ ports} \times 19.2 \text{ GB/s} = \mathbf{57.6} \text{ GB/s}$$

Hardware Supply: In the default configuration, all three AXI bundles (`gmemA`, `gmemB`, `gmemC`) are mapped to a **single DDR bank**.

The "Traffic Jam" Result:

$$\text{Ratio} = \frac{\text{Demand}}{\text{Supply}} = \frac{57.6 \text{ GB/s}}{19.2 \text{ GB/s}} = \mathbf{3}\times$$

$$\text{Actual Time} = 3.5\ \mu s \times 3 \approx \mathbf{10.5}\ \mu\mathbf{s}$$

This aligns closely with the observed **11** μ**s** execution time.

Optimization Path:

To achieve the theoretical ~3.5 μs execution time on real hardware, map each AXI bundle to a **separate physical DDR bank**:

```
# connectivity.cfg
[connectivity]
sp=IMAGE_DIFF_POSTERIZE_1.A:DDR[0]
sp=IMAGE_DIFF_POSTERIZE_1.B:DDR[1]
sp=IMAGE_DIFF_POSTERIZE_1.C:DDR[2]
```

This eliminates memory contention by providing 3 × 19.2 GB/s = 57.6 GB/s of parallel bandwidth, matching the kernel's demand.

Verified: V3 Performance After Multi-Bank DDR Mapping

After applying the `connectivity.cfg` configuration to map each AXI bundle to a separate DDR bank:

Metric	Before (Single Bank)	After (Multi-Bank)
Clock Frequency	300 MHz	300 MHz
Compute Unit Time	**11** μ**s**	**5** μ**s**
Kernel Execution Time	16 μs	9 μs
Transfer Rate (MB/s)	8868.2	**18886.5**

Key Observations:

- **2.2× faster CU time:** Eliminating memory contention reduced compute unit time from 11 μs to **5** μ**s**, approaching the theoretical 3.5 μs.
- **2.1× higher transfer rate:** Kernel transfer rate increased from 8868.2 MB/s to **18886.5 MB/s**, nearly saturating the per-bank bandwidth.
- **Host transfers unchanged:** Host-to-device and device-to-host rates remain ~3 MB/s (PCIe emulation limited).

The remaining gap between 5 μs and theoretical 3.5 μs is attributed to AXI protocol overhead and pipeline fill/drain latency.

V2 Performance Analysis: Partial Bottleneck

Theoretical: $3104 \div 300$ MHz $\approx$ **10.3** μs
Measured: **15** μs (50% slower)

Root Cause: Stage 1 reads A and B simultaneously, demanding 38.4 GB/s but limited to 19.2 GB/s. This 2× throttling on Stage 1 alone explains the gap:

$$\text{Total} = \underbrace{6.8\ \mu s}_{\text{Stage 1 (2×)}} + \underbrace{3.5\ \mu s}_{\text{Stage 2}} + \underbrace{3.4\ \mu s}_{\text{Stage 3}} = 13.7\ \mu s \approx \mathbf{15}\ \mu s$$

V2 is only partially bottlenecked because Stages 2-3 operate within bandwidth limits.

7 Conclusion

This laboratory exercise explored three architectural approaches for implementing an image processing accelerator with difference, posterization, and sharpening:

- **V1 (Sequential + 2D Buffers):** Simplest implementation with pixel-by-pixel filter access. The 1-pixel/cycle filter stage creates a **97% bottleneck**, resulting in 230 μs execution time.
- **V2 (Sequential + Line Buffers):** Achieves 64 pixels/cycle in all stages via sliding window. Stage 1's simultaneous A/B reads cause **2× memory contention** (10.3 μs → 15 μs).
- **V3 (Dataflow + Streams):** Maximum parallelism with concurrent stage execution. Default single-bank config caused **3× contention** (11 μs); multi-bank DDR mapping achieved **5** μs.

Key Finding: Memory Optimization Verified

Both V2 and V3 are **memory-bound**. Mapping AXI bundles to separate DDR banks via `connectivity.cfg` eliminated contention for V3:

Version	Theoretical	Single-Bank	Multi-Bank
V2	10.3 μs	15 μs	–
V3	3.5 μs	11 μs	**5** μs

V3's transfer rate doubled from 8868 MB/s to 18886 MB/s with multi-bank mapping.

Reference: For more details on interpreting the Vitis profile summary, see the official AMD documentation: Interpreting the Profile Summary.

A System Architecture Diagrams

This appendix presents the V3 dataflow kernel system diagrams generated by Vitis, illustrating the memory mapping configurations that affect kernel performance.

A.1 Single-Bank DDR Configuration (Default)

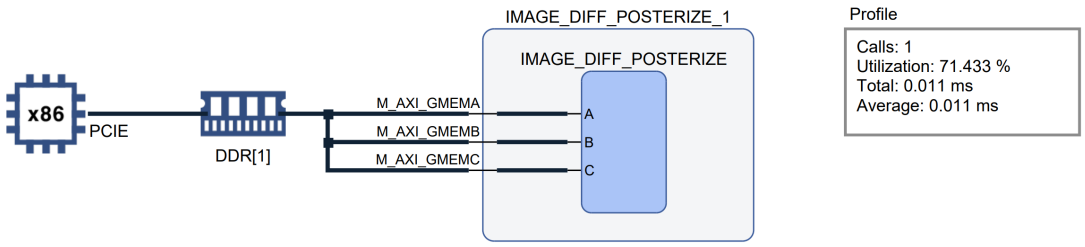

Figure 1: V3 kernel system diagram with default memory mapping. All three AXI master ports (`gmemA`, `gmemB`, `gmemC`) are mapped to a single DDR bank, creating a memory bandwidth bottleneck.

A.2 Multi-Bank DDR Configuration (Optimized)

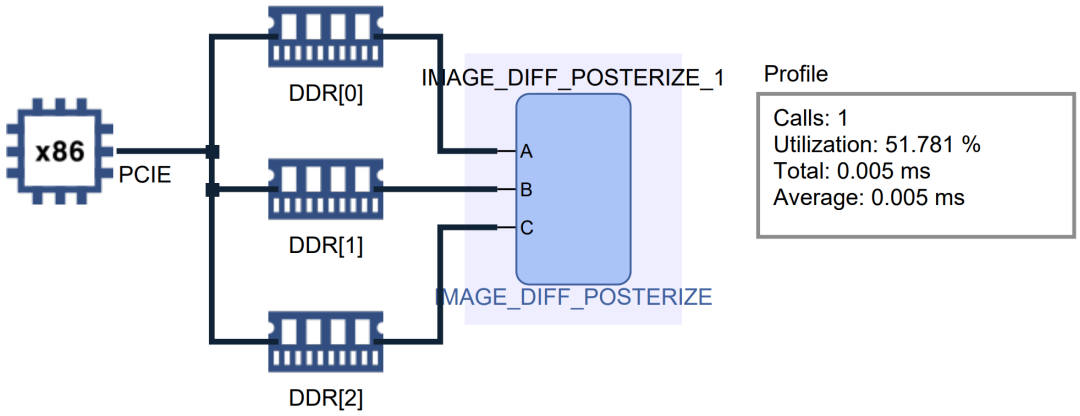

Figure 2: V3 kernel system diagram with multi-bank DDR mapping via `connectivity.cfg`. Each AXI master port is mapped to a separate DDR bank, eliminating memory contention.