Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Artificial Intelligence and Systems Engineering

Heterogeneous SoC Systems
(BMEVIMIMA25)

Homework

Jedla Martin (DEC4F6)

December 27, 2025

Contents

1 Scalar implementation in C

1.1 Algorithm Description

The scalar implementation acts as the golden reference for verifying the correctness of all subsequent hardware and vectorized versions. The algorithm performs 2D median filtering on an RGB image using a 5×5 sliding window.

To ensure bit-exact consistency across CPU, GPU, and FPGA platforms, the sorting step is implemented using the **Batcher Odd-Even Merge Sort** network rather than a standard library sort (like `std::sort`). This sorting network consists of a fixed sequence of Compare-and-Swap (CAS) operations, which is deterministic and identical to the structure used in the hardware implementation.

The process for each pixel is as follows:

1. **Window Extraction:** A 5×5 window is extracted for each color channel (R, G, B).
2. **Padding:** Boundary pixels are handled by reading valid image data where possible and zero-padding (black pixels) where the window overhangs the image edge.
3. **Sorting:** The extracted values are passed to the `batcher_sort_5x5` function, which sorts the array in-place using the hardcoded network.
4. **Selection:** The median value (index 12) is selected and written to the output.

1.2 Source Code

The code below shows the main filter function `median2d_c_batcher`. For brevity, the implementation of the `batcher_sort_5x5` function (which contains the unrolled sorting network) is omitted, but it is called for every 5×5 window.

```
void median2d_c_batcher(int32_t imgHeight,
                        int32_t imgWidth,
                        int32_t imgWidthF,
                        uint8_t* imgSrcExt,
                        uint8_t* imgDst) {
#pragma omp parallel for
    for(int row = 0; row < imgHeight; ++row) {
        for(int col = 0; col < imgWidth; ++col) {
            uint8_t buffer[M_HEIGHT * M_WIDTH * 3] = {0};
            for(size_t frow = 0; frow < M_HEIGHT; ++frow) {
                for(size_t fcol = 0; fcol < M_WIDTH; ++fcol) {
                    // r
                    buffer[(frow * M_WIDTH + fcol) * 3] =
                        imgSrcExt[((row + frow) * imgWidthF + col + fcol) * 3];

                    // g
                    buffer[(frow * M_WIDTH + fcol) * 3 + 1] =
                        imgSrcExt[((row + frow) * imgWidthF + col + fcol) * 3 +
                                  1];
```

```
                    // b
                    buffer[(frow * M_WIDTH + fcol) * 3 + 2] =
                        imgSrcExt[((row + frow) * imgWidthF + col + fcol) * 3 +
                                  2];
                }
            }
            batcher_sort_5x5(buffer);
            imgDst[(row * imgWidth + col) * 3] =
                buffer[(M_HEIGHT * M_WIDTH) / 2 * 3];
            imgDst[(row * imgWidth + col) * 3 + 1] =
                buffer[(M_HEIGHT * M_WIDTH) / 2 * 3 + 1];
            imgDst[(row * imgWidth + col) * 3 + 2] =
                buffer[(M_HEIGHT * M_WIDTH) / 2 * 3 + 2];
        }
    }
}
```

2 Vectorized, multi-threaded implementation

2.1 Algorithm Description

The goal of this task is to optimize the median filter using Data Level Parallelism (SIMD) and Thread Level Parallelism. The target architectures are **AVX2** (for AMD64) and **NEON** (for ARM64).

2.1.1 Why Batcher Sort for Vectorization?

The primary challenge in vectorizing a median filter is that standard sorting algorithms (like Quicksort) rely on conditional branching. Branching is highly inefficient in a SIMD context because all lanes in a vector register must execute the same instruction; divergence forces serialization.

The **Batcher Odd-Even Merge Sort** network is ideal for vectorization because it is **oblivious** to the data values. It consists of a fixed, pre-determined sequence of comparison operations. This allows us to map the "compare-and-swap" operation directly to vector hardware instructions (`min` and `max`) without any conditional jumps.

2.1.2 SIMD Parallelization Strategy

To achieve high throughput, we do not sort a single 5×5 window inside one vector register. Instead, we utilize the vector registers to process multiple neighboring pixels simultaneously.

For a vector width of N (e.g., $N = 32$ bytes for AVX2, $N = 16$ bytes for NEON), the algorithm works as follows:

1. **Data Loading:** We load the image data such that a single vector register V_k contains the pixel value at window position k for N different windows simultaneously.
   - V_0 holds the top-left pixel for windows centered at columns $c, c+1, \ldots, c+N-1$.
   - V_1 holds the next pixel (offset by 1) for these same N windows.

2. **Parallel Execution:** When we apply a single Batcher step (e.g., compare V_0 and V_1), we are effectively performing that comparison for N overlapping windows at the same time.

3. **Throughput:** With one iteration of the sorting network, we calculate the median for N output pixels. Theoretically, this speeds up the calculation by a factor of N compared to the scalar approach.

2.2 Algorithm Visualization

The figure below illustrates this strategy. Each vertical bar represents a SIMD register holding values from neighboring image positions. The sorting network is applied "vertically" across these registers.

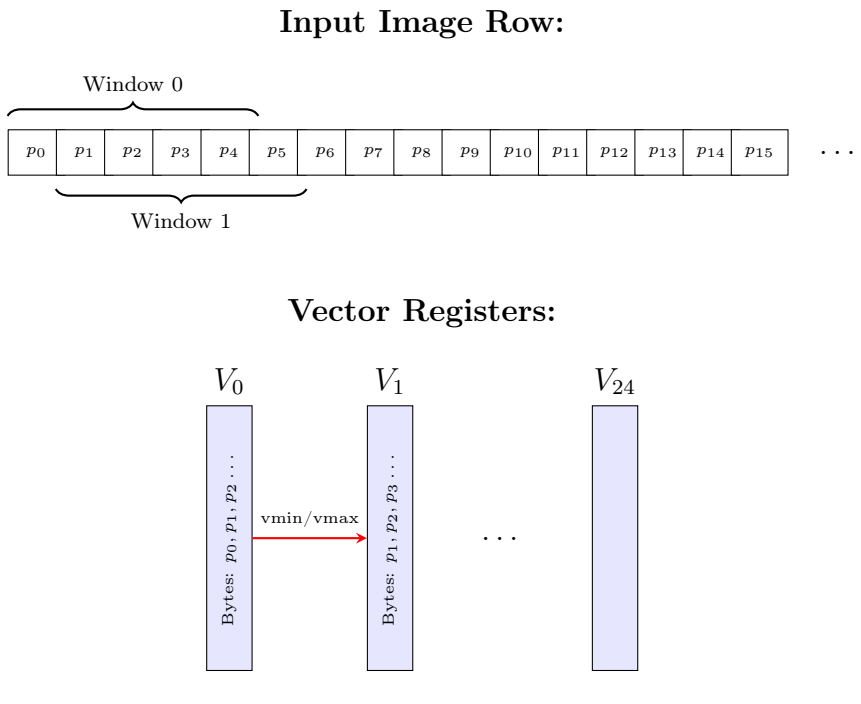


Figure 1: Visual representation of the Vectorized Batcher Algorithm. Each vector register holds data for 32 (AVX) or 16 (NEON) separate filter windows, enabling parallel sorting.

2.3 AMD64 Implementation (AVX2)

The AVX2 implementation processes 32 output pixels per iteration using 256-bit registers.

A limitation of the AVX2 instruction set in this context is the lack of flexible, low-latency byte-level shuffle instructions across registers (like NEON's `ext`). Therefore, we rely on **unaligned loads** (`_mm256_loadu_si256`) to fetch the 25 shifted versions of the window data directly from memory. While unaligned loads are relatively efficient on modern x86 CPUs, they still incur a higher cache/memory bandwidth cost compared to aligned loads with in-register shuffling.

```
void median2d_avx_uload(int32_t imgHeight,
```

```
                        int32_t imgWidth,
                        int32_t imgWidthF,
                        uint8_t* imgSrcExt,
                        uint8_t* imgDst) {
    #pragma omp parallel for
    for(int32_t row = 0; row < imgHeight; ++row) {
        __m256i window[WIN_LENGTH * WIN_HEIGHT];
        for(int32_t col = 0; col < imgWidth * 3; col += 32) {
            // Load the window: 256 bit => 32 * uint8_t => 10 * rgb r g
    #pragma GCC unroll WIN_HEIGHT
            for(uint32_t i = 0; i < WIN_HEIGHT; ++i) {
    #pragma GCC unroll WIN_LENGTH
                for(uint32_t j = 0; j < WIN_LENGTH; ++j) {
                    window[i * WIN_LENGTH + j] =
                        _mm256_loadu_si256(reinterpret_cast<const __m256i*>(
                            imgSrcExt + (row + i) * imgWidthF * 3 + col +
                            j * 3));
                }
            }

            batcher_sort_5x5(window);

            _mm256_storeu_si256(
                reinterpret_cast<__m256i*>(imgDst + row * imgWidth * 3 + col),
                window[12]);
        }
    }
}
```

2.4 ARM64 Implementation (NEON)

The NEON implementation processes 16 output pixels per iteration using 128-bit registers (`uint8x16_t`).

Unlike AVX2, the NEON instruction set includes powerful **vector extract** instructions (`vextq_u8`) which can construct a new vector from two existing vectors by shifting bytes. This allows us to optimize the memory access pattern significantly:

1. We can perform **aligned loads** (or fewer, larger block loads) to fetch a continuous chunk of image data into registers.
2. We then use `vextq_u8}` to "shuffle" or slide this data within the registers to generate the necessary window offsets.
3. This avoids the penalty of multiple unaligned loads and reduces the total number of load instructions issued to the memory unit.

Below are two versions: one using standard unaligned loads (`median2d_neon_uload`) and the optimized shuffle version (`median2d_neon_sh`).

```
void median2d_neon_uload(int32_t imgHeight,
                         int32_t imgWidth,
```

```
                        int32_t imgWidthF,
                        uint8_t* imgSrcExt,
                        uint8_t* imgDst) {
    #pragma omp parallel for
    for(int32_t row = 0; row < imgHeight; ++row) {
        for(int32_t col = 0; col < (imgWidth * 3); col += 16) {
            uint8x16_t window[WIN_LENGTH * WIN_HEIGHT] = {};

    // Load window
    #pragma GCC unroll WIN_HEIGHT
            for(uint32_t i = 0; i < WIN_HEIGHT; ++i) {
    #pragma GCC unroll WIN_LENGTH
                for(uint32_t j = 0; j < WIN_LENGTH; ++j) {
                    window[i * WIN_LENGTH + j] = vld1q_u8(
                        imgSrcExt + (row + i) * imgWidthF * 3 + col + j * 3);
                }
            }

            batcher_sort_5x5(window);

            // Store median value (13th element in sorted 25 elements)
            vst1q_u8(imgDst + row * imgWidth * 3 + col, window[12]);
        }
    }
}
```

```
void median2d_neon_sh(int32_t imgHeight,
                      int32_t imgWidth,
                      int32_t imgWidthF,
                      uint8_t* imgSrcExt,
                      uint8_t* imgDst) {
    #pragma omp parallel for
    for(int32_t row = 0; row < imgHeight; ++row) {
        uint8x16_t window[WIN_LENGTH * WIN_HEIGHT];
        uint8x16_t nullVec = vdupq_n_u8(0);
        for(int32_t col = 0; col < (imgWidth * 3); col += 16) {
    // Load window
    #pragma GCC unroll WIN_HEIGHT
            for(uint32_t i = 0; i < WIN_HEIGHT; ++i) {
                uint8x16_t row_data[2];
                // 0 - 15
                row_data[0] =
                    vld1q_u8(imgSrcExt + (row + i) * imgWidthF * 3 + col);
                // 12 - 27
                row_data[1] =
                    vld1q_u8(imgSrcExt + (row + i) * imgWidthF * 3 + col + 12);
                window[i * WIN_HEIGHT + 0] = row_data[0];
                window[i * WIN_HEIGHT + 4] = row_data[1];

                // 16 - 27
                row_data[1] = vextq_u8(row_data[1], nullVec, 4);
```

```
                window[i * WIN_HEIGHT + 1] =
                    vextq_u8(row_data[0], row_data[1], 3);
                window[i * WIN_HEIGHT + 2] =
                    vextq_u8(row_data[0], row_data[1], 2 * 3);
                window[i * WIN_HEIGHT + 3] =
                    vextq_u8(row_data[0], row_data[1], 3 * 3);
            }

            batcher_sort_5x5(window);

            // Store median value (13th element in sorted 25 elements)
            vst1q_u8(imgDst + row * imgWidth * 3 + col, window[12]);
        }
    }
}
```

3 GPU OpenCL implementation

3.1 Overview

In this task, the median filter was implemented on a GPU using OpenCL. The massively parallel architecture of the GPU allows for processing thousands of pixels simultaneously. However, the performance is heavily dependent on how effectively the memory hierarchy (Global, Local, and Private memory) is utilized.

Several kernels were developed to explore different memory strategies:

1. **Global Memory Scalar:** Naive implementation reading directly from global memory.
2. **Global Memory Vectorized:** Attempt to use SIMD vectors (`uchar16`) for sorting.
3. **Local Memory Tiling:** Using explicit caching of pixel blocks in fast local memory to minimize global bandwidth.
4. **Data Types:** Comparison between ‘float‘ and ‘int‘ implementations to target specific ALU capabilities.

3.2 Global Memory Implementations

3.2.1 Scalar Implementation

This kernel assigns one work-item (thread) per color component of a pixel. Each thread reads the 25 neighboring pixels directly from the off-chip Global Memory. While simple, this approach suffers from high latency and redundant memory traffic, as neighboring threads load the same overlapping pixel data multiple times.

```
__kernel void median2d_kernel_gl(
        int imgWidth,
        int imgWidthF,
        __global unsigned char* gInput,
```

```
        __global unsigned char* gOutput)
{
    // 1 thread calculates 1 component

    int row = get_global_id(1);
    int col = get_global_id(0);

    //if (col >= imgWidth * 3 || row >= get_global_size(1)) return;

    unsigned char window[25] = {0};

    int src_offset = (row * imgWidthF * 3) + col;

    for(int i = 0; i < 5; ++i){
        for(int j = 0; j < 5; ++j){
            int rd_offset = src_offset + (i * imgWidthF * 3) + (j * 3);
            window[i * 5 + j] = gInput[rd_offset];
        }
    }

    batcher5x5_uchar(window);

    int wr_offset = (row * imgWidth * 3) + col;
    gOutput[wr_offset] = window[12];
}
```

3.2.2 Vectorized Implementation Issues

An attempt was made to vectorize this global memory kernel using `uchar16` or `uchar8` vectors to process multiple pixels per thread (similar to the CPU AVX approach).

However, this approach proved to be **highly inefficient** on the GPU due to **register spilling**. To implement the sorting network for vectors, the compiler must keep 25 distinct vector variables alive simultaneously.

- A single `uchar16` variable occupies 16 bytes.
- 25 such variables require $25 \times 16 = 400$ bytes of register space per thread, plus overhead for intermediate calculations.

This exceeds the physical register file limit per thread on most GPU architectures. Consequently, the compiler "spills" variables into Global memory (acting as slow RAM), which destroys performance and limits the number of active wavefronts/warps (occupancy). Therefore, the scalar or tiled scalar approach is preferred on GPUs.

3.3 Local Memory Optimization

To solve the bandwidth bottleneck, we utilize **Local Memory** (Shared Memory). The image is processed in tiles (work-groups).

1. **Collaborative Loading:** Threads in a work-group cooperatively load a 16×16 tile of pixels plus the necessary "halo" (boundary pixels) into `__local` memory.

2. **Synchronization:** A `barrier(CLK_LOCAL_MEM_FENCE)` ensures all data is loaded.
3. **Processing:** Threads read the 5×5 window from the fast local memory cache instead of global memory.

3.3.1 Local Memory with Float Storage

In this variant, pixel data is cast to `float` in local memory. This is often done because some GPU ALUs are optimized for floating-point operations, or to avoid bank conflicts associated with smaller byte-sized accesses.

```
__kernel void median2d_kernel_sh_uchar_float(
        int imgWidth,
        int imgWidthF,
        __global unsigned char* gInput,
        __global unsigned char* gOutput)
{
    // 1 thread calculates 1 component

    int row = get_global_id(1);
    int col = get_global_id(0);

    // 20x20x1 window
    __local float data[20][20];

    int rgb_offset = (get_group_id(1) * get_num_groups(0) + get_group_id(0)) % 3;
    int wg_offset =  (get_group_id(1) * get_local_size(1) * imgWidthF * 3) +
    ↪  (get_group_id(0) / 3 * get_local_size(0) * 3 + rgb_offset);
    int loc_row = get_local_id(1);
    int loc_col = get_local_id(0);


    // 1 thread loads 2 bytes
    #pragma unroll
    for(int i = 0; i < 2; ++i){
        int win_idx = (loc_row * get_local_size(0)) + (loc_col) + i *
        ↪  get_local_size(1) * get_local_size(0);
        int win_row = win_idx / 20;
        int win_col = win_idx % 20;

        int rd_offset = wg_offset + (win_row * imgWidthF * 3) + (win_col * 3);

        if(win_row < 20 && win_col < 20){
            data[win_row][win_col] = (float)gInput[rd_offset];
        }
    }

    barrier(CLK_LOCAL_MEM_FENCE);

    float window[25] = {0};
    #pragma unroll
    for(int i = 0; i < 5; ++i){
        #pragma unroll
```

```
        for(int j = 0; j < 5; ++j){
            window[i * 5 + j] = data[loc_row + i][loc_col + j];
        }
    }

    batcher5x5_float(window);


    int wr_offset = (row * imgWidth * 3) + (get_group_id(0) / 3 * get_local_size(0)
    ↪  * 3 + loc_col * 3 + rgb_offset);
    gOutput[wr_offset] = (unsigned char) window[12];


}
```

3.3.2 Local Memory with Vectorized Types (float3/int3)

To improve memory throughput further, we can use vectorized loads (`vload3`) to read full RGB pixels at once. This reduces the number of memory instructions issued.

A critical optimization in this kernel is the specific choice of **3-component vectors** (`float3`/`int3`) over standard power-of-two vectors:

- **Bank Conflicts:** In Local Memory architecture, accessing data with power-of-two strides (e.g., `float4`, `float8`, or `float16`) often causes multiple threads to access the same memory bank simultaneously, leading to serialization of requests (bank conflicts).
- **Conflict Avoidance:** By using `float3`, the data access pattern creates a stride that naturally offsets threads across different memory banks. This prevents the bank conflicts that would occur with `float4`, thereby maximizing the effective Local Memory bandwidth.

```
__kernel void median2d_kernel_sh_uchar_float3(
        int imgWidth,
        int imgWidthF,
        __global unsigned char* gInput,
        __global unsigned char* gOutput)
{
    // 1 thread calculates 3 component (not 8 or 16 because of bank conflict)

    int row = get_global_id(1);
    int col = get_global_id(0);

    if (col * 3 >= imgWidth * 3) return;

    // 20x20x1 window
    __local float data[20 * 20 * 3];

    int wg_offset =  (get_group_id(1) * get_local_size(1) * imgWidthF * 3) +
    ↪  (get_group_id(0) * get_local_size(0) * 3);
    int loc_row = get_local_id(1);
```

```
    int loc_col = get_local_id(0);


    // 1 thread loads 2 bytes
    #pragma unroll
    for(int i = 0; i < 2; ++i){
        int win_idx = (loc_row * get_local_size(0)) + (loc_col) + i *
        ↪  get_local_size(1) * get_local_size(0);
        int win_row = win_idx / 20;
        int win_col = win_idx % 20;

        int rd_offset = wg_offset + (win_row * imgWidthF * 3) + (win_col * 3);

        if(win_row < 20 && win_col < 20){
            int local_idx = (win_row * 20 * 3) + (win_col * 3);
            vstore3(convert_float3(vload3(0, gInput + rd_offset)), 0, data +
            ↪  local_idx);
        }
    }

    barrier(CLK_LOCAL_MEM_FENCE);

    float3 window[25] = {0};
    #pragma unroll
    for(int i = 0; i < 5; ++i){
        #pragma unroll
        for(int j = 0; j < 5; ++j){
            window[i * 5 + j] = vload3(0, data + ((loc_row + i) * 20 * 3) + (loc_col
            ↪  * 3 + j * 3));
        }
    }

    batcher5x5_float3(window);


    int wr_offset = (row * imgWidth * 3) + (col * 3);
    vstore3(convert_uchar3(window[12]), 0, gOutput + wr_offset);

}
```

3.4 Implementation Notes

3.4.1 Integer vs. Float ALUs

The kernels presented above were also implemented using `int` and `int3` types instead of `float`. The source code is identical (replacing `float` with `int` and `convert_float3` with `convert_int3`). Modern GPUs often have separate ALUs for integer and floating-point arithmetic. Implementing both versions allows benchmarking to determine which pipeline offers better performance on the specific hardware used.

3.4.2 Memory Alignment Warning for Float3

A critical detail when working with OpenCL vectors is memory alignment regarding 3-component vectors.

- When allocating an array of `float3` (e.g., `__local float3 data[...]`), the OpenCL compiler implicitly aligns elements to 16 bytes (the size of `float4`), effectively padding the 4th component with zeros.
- This padding is applied silently without compiler warnings. If the developer assumes standard 12-byte packing (4 bytes $\times$ 3 components) and calculates indices accordingly, the memory offsets will be incorrect.
- This behavior creates significant potential for bugs that are difficult to diagnose. A developer unaware of this implicit alignment will likely only discover the discrepancy through extensive debugging of memory access patterns. To avoid this ambiguity, the implementation uses a flat scalar `float` array and manual pointer arithmetic.

4 Vitis HLS C implementation

4.1 Architecture Overview

For the FPGA implementation, the requirement is to process the video stream in real-time (1 pixel per clock cycle, `II=1`). Unlike CPU/GPU implementations where the entire image is accessible in memory, the hardware must process pixels as they arrive sequentially.

To achieve this efficiently, a **Line Buffer** architecture is used (Figure 2):

- **Buffering:** Four line buffers (BRAM) store the previous rows of the image.
- **Column Access:** As a new pixel arrives, a 5×1 column is read simultaneously from the buffers.
- **Sliding Window:** This column updates a 5×5 register array (shifting left).
- **Sorting:** The window contents are sorted by the combinational Batcher network.

Control Signal Synchronization Strategy

A critical design trade-off was made regarding the synchronization signals (HS, VS, DE). Ideally, to align the output pixel (which geometrically corresponds to the center of the 5×5 window) perfectly with the control signals, the signals should be delayed to match the algorithmic group delay (buffering them for 4 rows). However, buffering 1-bit control signals for multiple video lines would consume a significant amount of on-chip memory (BRAM) for very little gain.

Therefore, this implementation delays the control signals to match the fixed pipeline computation latency. The exact delay value was derived from the HLS synthesis report, which indicated a total latency of 75ns. Given the clock period of 5ns, the required delay is calculated as:

$$\text{Delay Cycles} = \frac{75\text{ns}}{5\text{ns}} = 15 \text{ cycles}$$

Consequently, every control signal is delayed by 15 cycles (represented as `SIGNAL_DELAY` in the code logic relative to the specific pipeline stage). As a result, the output image is spatially shifted relative to the synchronization frame, but this shift is imperceptible in real-time video streaming and saves significant FPGA memory resources.

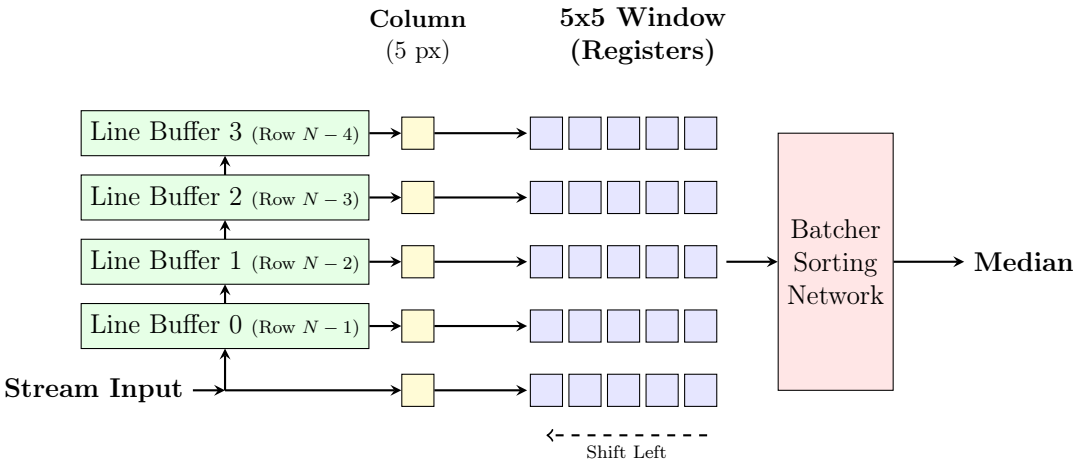


Figure 2: Line Buffer Architecture for Streaming Image Processing. Data moves from Line Buffers → Column Vector → Sliding Window → Sorter.

4.2 Source Code

The hardware kernel implementation uses static variables to infer BRAM line buffers and registers that persist between clock cycles.

```
void median2d_hw(uint8_ct* r_in, uint8_ct* g_in, uint8_ct* b_in, uint1_ct* hs_in,
↪  uint1_ct* vs_in, uint1_ct* de_in, uint8_ct* r_out, uint8_ct* g_out, uint8_ct*
↪  b_out, uint1_ct* hs_out, uint1_ct* vs_out, uint1_ct* de_out){
#pragma HLS INTERFACE mode=ap_none port=return

#pragma HLS INTERFACE mode=ap_none port=r_in
#pragma HLS INTERFACE mode=ap_none port=g_in
#pragma HLS INTERFACE mode=ap_none port=b_in
#pragma HLS INTERFACE mode=ap_none port=hs_in
#pragma HLS INTERFACE mode=ap_none port=vs_in
#pragma HLS INTERFACE mode=ap_none port=de_in

#pragma HLS INTERFACE mode=ap_none port=r_out
#pragma HLS INTERFACE mode=ap_none port=g_out
#pragma HLS INTERFACE mode=ap_none port=b_out
#pragma HLS INTERFACE mode=ap_none port=hs_out
#pragma HLS INTERFACE mode=ap_none port=vs_out
#pragma HLS INTERFACE mode=ap_none port=de_out

#pragma HLS PIPELINE II=1

    static uint8_ct line_buf[3][WIN_H - 1][BUF_L];
    #pragma HLS BIND_STORAGE variable=line_buf type=ram_2p
    #pragma HLS ARRAY_PARTITION variable=line_buf dim=1 type=complete
    #pragma HLS ARRAY_PARTITION variable=line_buf dim=2 type=complete
```

```
uint8_ct cur_pixel[3];
cur_pixel[0] = *r_in;
cur_pixel[1] = *g_in;
cur_pixel[2] = *b_in;

// Variable to check edge of hs
static uint1_ct hs_edge[2];
hs_edge[0] = hs_edge[1];
hs_edge[1] = *hs_in;

// Current column of the frame - latest index of circular buffer
static uint14_ct curr_col;
if(hs_edge[0] == 0 && hs_edge[1] == 1){
    curr_col = 0;
} else{
    curr_col++;
}


// Window to calculate the median value of
static uint8_ct window[3][WIN_H][WIN_L];
#pragma HLS ARRAY_PARTITION variable=window type=complete

// Shifting the window left by 1 column every iteration. Delay: 1
for(int rgb = 2; rgb >= 0; --rgb){
#pragma HLS UNROLL
    for(int x = WIN_H - 1; x >= 0; --x){
        for(int y = 0; y < WIN_L - 1; ++y){
            window[rgb][x][y] = window[rgb][x][y + 1];
        }
    }
}
// Putting the new column in
for(int rgb = 2; rgb >= 0; --rgb){
#pragma HLS UNROLL
    // The new pixel
    window[rgb][WIN_H - 1][WIN_L - 1] = cur_pixel[rgb];
    for(int x = WIN_H - 1 - 1; x >= 0; --x){
        window[rgb][x][WIN_L - 1] = line_buf[rgb][x][curr_col];
    }
}

// Putting the new data in the buffer
for(int rgb = 2; rgb >= 0; --rgb){
#pragma HLS UNROLL
    for(int x = 0; x < WIN_H - 1 - 1; ++x){
        line_buf[rgb][x][curr_col] = line_buf[rgb][x + 1][curr_col];
    }
    line_buf[rgb][WIN_H - 1 - 1][curr_col] = cur_pixel[rgb];
}

// Copying the window to a temp vairable to be able to sort it
```

```
    uint8_ct window_cpy[3][WIN_H * WIN_L];
    #pragma HLS ARRAY_PARTITION variable=window_cpy dim=0 type=complete
    for(int rgb = 2; rgb >=0; --rgb){
    #pragma HLS UNROLL
        for(int x = WIN_H - 1; x >= 0; --x){
            for(int y = WIN_L - 1; y>=0; --y){
                window_cpy[rgb][x * WIN_H + y] = window[rgb][x][y];
            }
        }
    }
    batcher5x5(window_cpy[0]);
    batcher5x5(window_cpy[1]);
    batcher5x5(window_cpy[2]);

    *r_out = window_cpy[0][WIN_H * WIN_L / 2];
    *g_out = window_cpy[1][WIN_H * WIN_L / 2];
    *b_out = window_cpy[2][WIN_H * WIN_L / 2];


    // Delaying the control signals
    static uint1_ct de_dl[PIXEL_DELAY];
    static uint1_ct hs_dl[PIXEL_DELAY];
    static uint1_ct vs_dl[PIXEL_DELAY];

    *de_out = de_dl[0];
    *hs_out = hs_dl[0];
    *vs_out = vs_dl[0];

    for(int i = 0; i < PIXEL_DELAY; ++i){
    #pragma HLS UNROLL
        de_dl[i] = (i == PIXEL_DELAY - 1) ? *de_in : de_dl[i + 1];
        hs_dl[i] = (i == PIXEL_DELAY - 1) ? *hs_in : hs_dl[i + 1];
        vs_dl[i] = (i == PIXEL_DELAY - 1) ? *vs_in : vs_dl[i + 1];
    }
}
```

4.3 Testbench

The testbench verifies the logic by comparing the HW model output against a standard software reference.

4.3.1 Latency: Simulation vs. Hardware

A key challenge in verifying pipelined hardware in C is the difference in latency behavior:

- **Misalignment in C-Sim:** The C++ function computes the pixel data instantly (0 cycles), but the control signals (HS, VS, DE) are explicitly delayed by 15 cycles in the code logic. This results in a temporal mismatch where valid data appears at cycle T, but the validity flag (`de_out`) appears at $T + 15$.
- **Compensation Strategy:** To resolve this, the Testbench utilizes a `std::deque` (queue) as a delay line. The instant data results are pushed into the queue, and

popped only after 15 cycles, perfectly aligning the data with the delayed control signals for verification.

```
#define TEST_WIDTH  64
#define TEST_HEIGHT 32

// Hardware Latency parameters for 5x5 window
// The explicit delay you added in HW for signals
constexpr int SIGNAL_DELAY = 15;
// Number of lines to skip at the start (Line buffer fill time for 5x5)
constexpr int SKIP_LINES = 2;

// Function Declaration
void median2d_hw(uint8_ct* r_in, uint8_ct* g_in, uint8_ct* b_in,
                 uint1_ct* hs_in, uint1_ct* vs_in, uint1_ct* de_in,
                 uint8_ct* r_out, uint8_ct* g_out, uint8_ct* b_out,
                 uint1_ct* hs_out, uint1_ct* vs_out, uint1_ct* de_out);

// Software Reference
uint8_t median_ref(const std::vector<uint8_t>& img, int x, int y, int w, int h) {
    std::vector<uint8_t> window;
    for (int wy = -2; wy <= 2; ++wy) {
        for (int wx = -2; wx <= 2; ++wx) {
            int cx = x + wx; int cy = y + wy;
            // Mirroring/Clamping boundary handling
            if (cx < 0) cx = 0; if (cx >= w) cx = w - 1;
            if (cy < 0) cy = 0; if (cy >= h) cy = h - 1;
            window.push_back(img[cy * w + cx]);
        }
    }
    std::sort(window.begin(), window.end());
    return window[12];
}

int main() {
    const int num_pixels = TEST_WIDTH * TEST_HEIGHT;
    std::vector<uint8_t> src_img(num_pixels);
    std::vector<uint8_t> hw_result(num_pixels);
    std::vector<uint8_t> sw_result(num_pixels);

    // 1. Generate Input Image
    std::cout << "[TestBench] Generating Input Image..." << std::endl;
    srand(123); // Fixed seed for reproducibility
    for (int i = 0; i < num_pixels; ++i) {
        uint8_t val = (i * 4) % 256;
        if ((rand() % 20) == 0) val = (rand() % 2) ? 0 : 255; // Add salt & pepper
        ↪  noise
        src_img[i] = val;
    }

    // 2. Hardware Simulation
    std::cout << "[TestBench] Running Hardware Simulation..." << std::endl;
    uint8_ct r_in = 0, g_in = 0, b_in = 0;
```

```
    uint1_ct hs_in = 0, vs_in = 0, de_in = 0;
    uint8_ct r_out, g_out, b_out;
    uint1_ct hs_out, vs_out, de_out;

    // Queue to delay the C-simulation data to match the Control Signal Delay
    std::deque<uint8_t> r_delay_q;

    // Run enough cycles to flush the pipeline
    int total_cycles = (TEST_WIDTH + 10) * (TEST_HEIGHT + 5);
    int out_idx = 0;

    for (int i = 0; i < total_cycles; ++i) {
        // --- Input Generation ---
        int frame_x = i % (TEST_WIDTH + 10);
        int frame_y = i / (TEST_WIDTH + 10);
        bool active_video = (frame_x < TEST_WIDTH) && (frame_y < TEST_HEIGHT);

        if (active_video) {
            r_in = src_img[frame_y * TEST_WIDTH + frame_x];
            de_in = 1; hs_in = 1;
        } else {
            r_in = 0; de_in = 0; hs_in = 0;
        }
        vs_in = (frame_y < TEST_HEIGHT) ? 1 : 0;

        // --- IP Call ---
        median2d_hw(&r_in, &g_in, &b_in, &hs_in, &vs_in, &de_in,
                    &r_out, &g_out, &b_out, &hs_out, &vs_out, &de_out);

        // --- C-Sim Compensation Logic ---

        // 1. Push current "instant" result into the delay line
        r_delay_q.push_back((uint8_t)r_out);

        // 2. Pop the "aligned" result (from SIGNAL_DELAY cycles ago)
        uint8_t r_aligned = 0;
        if (r_delay_q.size() > SIGNAL_DELAY) {
            r_aligned = r_delay_q.front();
            r_delay_q.pop_front();
        }

        // 3. Capture result based on VALID signal
        // Since de_out is delayed by the IP, and r_aligned is delayed by the queue,
        // they are now perfectly synchronized.
        if (de_out == 1 && out_idx < num_pixels) {
            hw_result[out_idx++] = r_aligned;
        }
    }

    // 3. Software Reference Calculation
    std::cout << "[TestBench] Running Software Reference..." << std::endl;
    for (int y = 0; y < TEST_HEIGHT; ++y) {
        for (int x = 0; x < TEST_WIDTH; ++x) {
```

```
            sw_result[y * TEST_WIDTH + x] = median_ref(src_img, x, y, TEST_WIDTH,
            ↪  TEST_HEIGHT);
        }
    }

    // 4. Verify Results
    std::cout << "[TestBench] Verifying Results..." << std::endl;
    int errors = 0;

    // Group delay for 5x5 window (Center is at index 2)
    const int GROUP_DELAY_X = 2;
    const int GROUP_DELAY_Y = 2;

    // We verify the "valid" center part of the image
    // START: y = 2 because rows 0,1 outputs correspond to negative window
    ↪  coordinates (padding)
    // STOP:  y < H-2 because the HW stream ends before flushing the very last
    ↪  pixels of the image
    for (int y = GROUP_DELAY_Y; y < TEST_HEIGHT - GROUP_DELAY_Y; ++y) {
        for (int x = GROUP_DELAY_X; x < TEST_WIDTH - GROUP_DELAY_X; ++x) {

            // 1. Software Index (The pixel we want to verify)
            int sw_idx = y * TEST_WIDTH + x;

            // 2. Hardware Index (Where this pixel is located in the output stream)
            // The HW output for Center(x,y) appears when Input is at (y+2, x+2)
            // Linear Index = (Row + DelayY) * Width + (Col + DelayX)
            int hw_idx = (y + GROUP_DELAY_Y) * TEST_WIDTH + (x + GROUP_DELAY_X);

            // Boundary check: ensure we don't read past the captured HW buffer
            if (hw_idx >= num_pixels) continue;

            uint8_t hw_val = hw_result[hw_idx];
            uint8_t sw_val = sw_result[sw_idx];

            if (hw_val != sw_val) {
                std::cout << "Error at (x=" << x << ", y=" << y << ") "
                          << "SW Idx: " << sw_idx << " HW Idx: " << hw_idx
                          << " Exp: " << (int)sw_val
                          << " Got: " << (int)hw_val << std::endl;
                errors++;
                if(errors > 20) break;
            }
        }
        if(errors > 20) break;
    }

    if (errors == 0) {
        std::cout << "\n*** TEST PASSED! ***" << std::endl;
        return 0;
    } else {
        std::cout << "\n*** TEST FAILED with " << errors << " errors. ***" <<
        ↪  std::endl;
```

```
        return 1;
    }
}
```

4.4 Integration and Resource Results

After synthesis, the IP was integrated into Vivado (Figure 3). The resource utilization reports are stacked below in Figure 4.

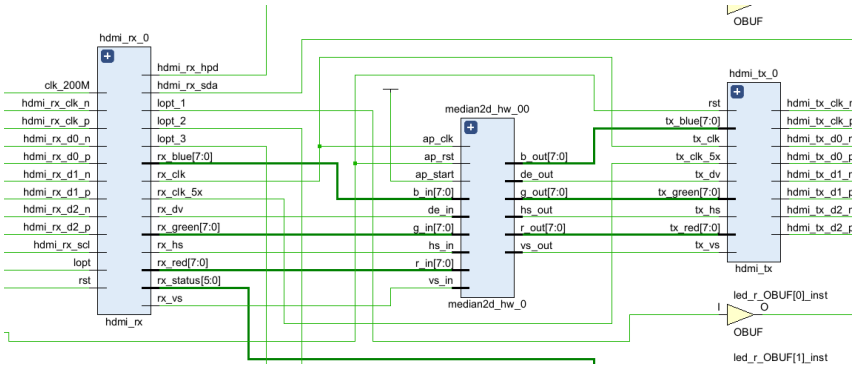
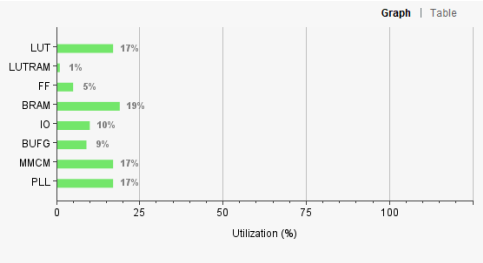


Figure 3: Synthesized Median Filter IP block integrated into the Vivado design.

MODULES & LOOPS	LATENCY(NS)	INTERVAL	PIPELINED	STRUCTURE	BRAM	DSP	FF	LUT	URAM
median2d_hw	75.000	1	yes	function	48	0	5 149	14 695	0

(a) HLS Synthesis Resource Estimates (LUT, FF, BRAM, DSP).



(b) Final Post-Implementation FPGA Utilization.

Figure 4: Resource utilization reports for the Median Filter hardware.

5 Summary

This documentation details the implementation of a 5×5 median filter across heterogeneous computing platforms, ranging from general-purpose CPUs to GPUs and FPGAs. The primary objective was to optimize the image processing pipeline by leveraging the specific architectural advantages of each hardware target while maintaining bit-exact consistency with a scalar reference.

To ensure efficient execution on parallel architectures (SIMD, GPU, FPGA), the **Batcher Odd-Even Merge Sort** algorithm was selected for all implementations. Its branchless, comparator-network structure allows for deterministic execution paths, which is critical for vectorization and hardware pipelining.

The work is divided into four main implementations:

- **Scalar Reference (CPU):** A standard C/C++ implementation using the Batcher network serves as the golden reference for correctness verification.
- **Vectorized Implementation (CPU):** This version exploits Data Level Parallelism using **AVX2** (for x86) and **NEON** (for ARM) instruction sets. By mapping the sorting network's comparisons to vector min/max instructions, the system sorts multiple pixel windows simultaneously, significantly increasing throughput.
- **GPU Acceleration (OpenCL):** The algorithm was ported to the GPU using OpenCL. The implementation overcomes global memory bandwidth bottlenecks by using **Local Memory tiling**, where work-groups cooperatively load image blocks into on-chip cache. Optimization analysis highlighted that using flat scalar arrays is preferable to vector types (like `uchar16`) to avoid register spilling and implicit padding issues.
- **Hardware Acceleration (Vitis HLS):** A fully pipelined streaming architecture was developed for FPGA. Using a **Line Buffer** design, the kernel processes incoming pixels in real-time with an Initiation Interval of 1 (`II=1`). The design explicitly handles the algorithmic and pipeline latency differences between the high-level C simulation and the synthesized hardware signals.

Source Code Availability

The complete source code, including the scalar reference, vectorized kernels, OpenCL definitions, and Vitis HLS projects, is available at the following repository:

`https://github.com/jedlamartin/heterogen_soc_hf`