ECE415: High Performance Computing Systems
Lab4 - CUDA Implementation and Performance Optimization of Clahe Algorithm

Contents

Introduction

In this lab we implement and then try to optimize the Contrast Limited Adaptive Histogram Equalization (CLAHE) algorithm on the GPU using CUDA. We start from the reference CPU implementation, get a simple correct GPU version running and then apply a series of optimizations that target the main performance bottlenecks that we observed.

0 CPU to GPU code

The original CPU implementation of CLAHE is organized in two main stages:

1. **Tile-wise histogram computation and LUT generation.**
   The image is partitioned into tiles of size `TILE_SIZE` $\times$ `TILE_SIZE`. For each tile we:
   - compute the 256-bin histogram
   - apply clipping using `CLIP_LIMIT` and redistribute the excess
   - compute the cumulative distribution function (CDF) and build a 256 entry LUT

   All LUTs are stored in a large 3D array of size `grid_h * grid_w * 256`
2. **Bilinear interpolation for each pixel**
   For each pixel (x, y), we:
   - locate the four neighboring tiles (top-left, top-right, bottom-left, bottom-right)
   - read the corresponding LUT values for the pixel intensity from these tiles
   - perform bilinear interpolation based on the pixel position relative to the tile centers
   - write the final value to the output image

On the GPU we preserve exactly this two phase structure but map it onto two CUDA kernels:

- `lut_init`: one thread per tile and computing the histogram and LUT of that tile
- `bilinear_interp`: one thread per pixel, performing bilinear interpolation and writing the output image

The host code is responsible for file I/O, device memory allocation, data transfers, kernel launches and time and throughput measurement.

0.1 Host Device Separation

The original helper functions `read_pgm`, `write_pgm` and `free_pgm` are host side functions (they perform file I/O and CPU memory management). In our GPU version we keep their logic identical to the reference CPU code and simply annotate them with `__host__` to make explicit that they are not executed on the device. Aside from this annotation we do not modify their implementations.

0.2 Data Structures on the GPU

On the CPU, the image is represented with the following structure:

```
typedef struct {
    int w;
    int h;
    unsigned char *img;
} PGM_IMG;
```

We reuse the same struct type on the GPU side, but we only allocate the `img` pointer on the device. The width and height are passed to the kernels as separate integer arguments.
On the host we have:

```
PGM_IMG h_img_in, h_img_out;
```

and we create the device side counterparts for the image data:

```
PGM_IMG d_img_in, d_img_out;
int *d_all_luts;
```

The corresponding memory allocations and host device transfers are:

```
cudaMalloc((void **)&d_img_in.img, height * width * sizeof(unsigned char));
cudaMalloc((void **)&d_img_out.img, height * width * sizeof(unsigned char));
cudaMalloc((void **)&d_all_luts, grid_h * grid_w * 256 * sizeof(int));

cudaMemcpy(d_img_in.img, h_img_in.img, height * width * sizeof(unsigned char),
    cudaMemcpyHostToDevice);

// ... kernel launches ...

cudaMemcpy(h_img_out.img, d_img_out.img, height * width * sizeof(unsigned char),
            cudaMemcpyDeviceToHost);
```

We also inlcude macros and error checking around `cudaMalloc` and `cudaMemcpy` to handle failures (freeing any allocated host or device memory and calling `cudaDeviceReset()` in case of error).

0.3 Kernel 1: Tile-wise LUT Initialization (`lut_init`)

In the sequential implementation tile processing is performed by nested loops over the tile indices:
CPU (original):

```
for (ty = 0; ty < grid_h; ++ty) {
    for (tx = 0; tx < grid_w; ++tx) {
        x_start = tx * TILE_SIZE;
        y_start = ty * TILE_SIZE;

        actual_tile_w = (x_start + TILE_SIZE > w) ? (w - x_start) : TILE_SIZE;
        actual_tile_h = (y_start + TILE_SIZE > h) ? (h - y_start) : TILE_SIZE;

        current_lut_ptr = &all_luts[(ty * grid_w + tx) * 256];

        compute_histogram(img_in.img, w, h,
                          x_start, y_start,
                          actual_tile_w, actual_tile_h,
                          current_lut_ptr);
    }
}
```

On the GPU we parallelize this loop by assigning **one CUDA thread per tile**. We launch a two dimensional grid in tile space. Inside the kernel we compute the tile indices from the CUDA block and thread indices and then call a device version of `compute_histogram`.
GPU: one thread per tile (`lut_init` kernel):

```
__device__ void compute_histogram(unsigned char* data,
                                  int w, int h,
                                  int start_x, int start_y,
                                  int tile_w, int tile_h,
                                  int* lut) {
    int hist[256] = {0};
    int x, y, i, avg_inc, val;
    int excess = 0, cdf = 0;
    int total_pixels = tile_w * tile_h;

    // Build histogram
    for (y = start_y; y < start_y + tile_h; ++y) {
        for (x = start_x; x < start_x + tile_w; ++x) {
            if (x < w && y < h) {
                hist[data[y * w + x]]++;
            }
        }
    }

    // Clip histogram
    for (i = 0; i < 256; ++i) {
        if (hist[i] > CLIP_LIMIT) {
            excess += (hist[i] - CLIP_LIMIT);
            hist[i] = CLIP_LIMIT;
        }
    }

    // Redistribute excess
    avg_inc = excess / 256;
    for (i = 0; i < 256; ++i) {
        hist[i] += avg_inc;
    }

    // Compute CDF and LUT
    for (i = 0; i < 256; ++i) {
        cdf += hist[i];
        val = (int)((float)cdf * 255.0f / total_pixels + 0.5f);
        if (val > 255) val = 255;
        lut[i] = val;
    }
}

__global__ void lut_init(int *d_all_luts,
                         unsigned char *d_img_in,
                         int height, int width) {
    int grid_w = (width  + TILE_SIZE - 1) / TILE_SIZE;
    int grid_h = (height + TILE_SIZE - 1) / TILE_SIZE;

    int tx = threadIdx.x + blockIdx.x * blockDim.x;
    int ty = threadIdx.y + blockIdx.y * blockDim.y;

    if (tx >= grid_w || ty >= grid_h) return;

    int x_start = tx * TILE_SIZE;
    int y_start = ty * TILE_SIZE;

    int actual_tile_w = (x_start + TILE_SIZE > width)
                        ? (width  - x_start)
                        : TILE_SIZE;
    int actual_tile_h = (y_start + TILE_SIZE > height)
                        ? (height - y_start)
                        : TILE_SIZE;

    int *current_lut_ptr = &d_all_luts[(ty * grid_w + tx) * 256];

    compute_histogram(d_img_in, width, height,
                      x_start, y_start,
                      actual_tile_w, actual_tile_h,
                      current_lut_ptr);
}
```

At this first stage we keep the histogram computation inside each tile sequential within a single thread (no shared memory or tile parallelism yet). However we process many tiles in parallel across the GPU.

0.4 Kernel 2: Per-pixel Bilinear Interpolation (`bilinear_interp`)

The second stage of the CPU code iterates over all pixels and performs bilinear interpolation of the four neighboring LUT values:

CPU (original):

```
for (y = 0; y < h; ++y) {
    for (x = 0; x < w; ++x) {

        ty_f = (float)y / TILE_SIZE - 0.5f;
        tx_f = (float)x / TILE_SIZE - 0.5f;

        y1 = (int)floor(ty_f);
        x1 = (int)floor(tx_f);
        y2 = y1 + 1;
        x2 = x1 + 1;

        y_weight = ty_f - y1;
        x_weight = tx_f - x1;

        if (x1 < 0) x1 = 0;
        if (x2 >= grid_w) x2 = grid_w - 1;
        if (y1 < 0) y1 = 0;
        if (y2 >= grid_h) y2 = grid_h - 1;

        val = img_in.img[y * w + x];

        tl = all_luts[(y1 * grid_w + x1) * 256 + val];
        tr = all_luts[(y1 * grid_w + x2) * 256 + val];
        bl = all_luts[(y2 * grid_w + x1) * 256 + val];
        br = all_luts[(y2 * grid_w + x2) * 256 + val];

        top = tl * (1.0f - x_weight) + tr * x_weight;
        bot = bl * (1.0f - x_weight) + br * x_weight;
        final_val = top * (1.0f - y_weight) + bot * y_weight;

        img_out.img[y * w + x] = (unsigned char)(final_val + 0.5f);
    }
}
```

On the GPU this is a **one thread per pixel** pattern. We launch a 2D grid in image space and each thread computes the output value of a single pixel:

GPU: one thread per pixel (`bilinear_interp` kernel):

```
__global__ void bilinear_interp(unsigned char *d_img_in,
                                unsigned char *d_img_out,
                                int *all_luts,
                                int height, int width) {
    int grid_w = (width  + TILE_SIZE - 1) / TILE_SIZE;
    int grid_h = (height + TILE_SIZE - 1) / TILE_SIZE;

    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;

    if (x >= width || y >= height) return;

    float ty_f = (float)y / TILE_SIZE - 0.5f;
    float tx_f = (float)x / TILE_SIZE - 0.5f;

    int y1 = (int)floor(ty_f);
    int x1 = (int)floor(tx_f);
    int y2 = y1 + 1;
    int x2 = x1 + 1;

    float y_weight = ty_f - y1;
    float x_weight = tx_f - x1;

    if (x1 < 0)        x1 = 0;
    if (x2 >= grid_w)  x2 = grid_w - 1;
    if (y1 < 0)        y1 = 0;
    if (y2 >= grid_h)  y2 = grid_h - 1;

    int val = d_img_in[y * width + x];

    int tl = all_luts[(y1 * grid_w + x1) * 256 + val];
    int tr = all_luts[(y1 * grid_w + x2) * 256 + val];
    int bl = all_luts[(y2 * grid_w + x1) * 256 + val];
    int br = all_luts[(y2 * grid_w + x2) * 256 + val];

    float top = tl * (1.0f - x_weight) + tr * x_weight;
    float bot = bl * (1.0f - x_weight) + br * x_weight;
    float final_val = top * (1.0f - y_weight) + bot * y_weight;

    d_img_out[y * width + x] = (unsigned char)(final_val + 0.5f);
}
```

This kernel is almost a direct translation of the sequential per pixel code with the nested `for` loops replaced by the 2D CUDA thread grid.

0.5 Grid and Block Configuration

For both kernels we use a two dimensional block configuration of:

```
#define BLOCK_DIM 16  // 16 x 16 = 256 threads per block
```

The reasoning behind this is that our GPU supports a maximum of 49,152 bytes of shared memory per thread block (as found in deviceQuery). For our implementation if we used a 32×32 block size, we would have 1024 threads per block. Since each thread has a private histogram of 256 integers (1024 bytes) the total memory required per block would be:

$$1024 \text{ threads} \times 256 \text{ bins} \times 4 \text{ bytes/bin} \approx 1,048,576 \text{ bytes (1 MB)}$$

This vastly exceeds the 48 KB limit and therefore we restrict the block size to 16×16 (256 threads).

For the LUT initialization (tile space) we configure:

```
dim3 block_dim(MAX_BLOCK_LENGTH, MAX_BLOCK_LENGTH);
dim3 grid_dim_lut((grid_w + MAX_BLOCK_LENGTH - 1) / MAX_BLOCK_LENGTH,
                  (grid_h + MAX_BLOCK_LENGTH - 1) / MAX_BLOCK_LENGTH);

lut_init<<<grid_dim_lut, block_dim>>>(d_all_luts, d_img_in.img, height, width);
```

For the bilinear interpolation we configure:

```
dim3 grid_dim_bilinear((width  + MAX_BLOCK_LENGTH - 1) / MAX_BLOCK_LENGTH,
                       (height + MAX_BLOCK_LENGTH - 1) / MAX_BLOCK_LENGTH);

bilinear_interp<<<grid_dim_bilinear, block_dim>>>(d_img_in.img, d_img_out.img,
                                                  d_all_luts, height, width);
```

0.6 Performance

Below are the graphs showing the time and throughput of the kernels in comparison to the CPU execution:

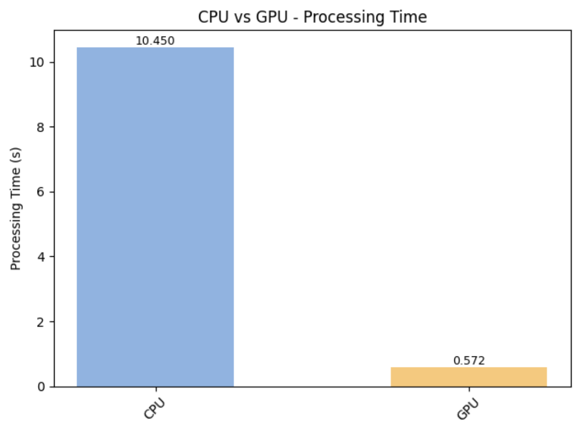


Figure 1: CPU vs GPU - Processing Time

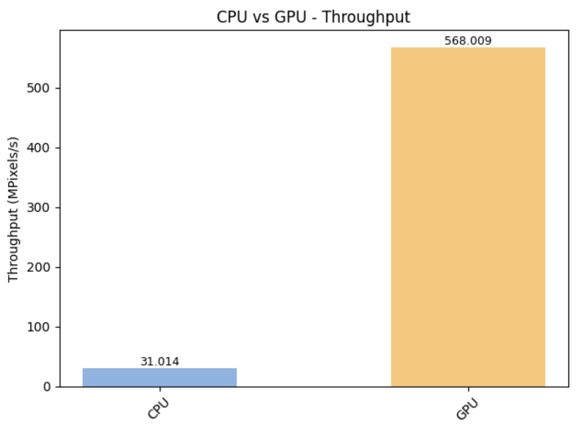


Figure 2: CPU vs GPU - Throughput

1 Optimization 1 - Shared Histogram - Failed

In our first optimization we focus on parallelizing the histogram computation within each tile by using a shared histogram in `shared` memory. In the baseline GPU version each tile's histogram and LUT were computed sequentially by a single thread (one thread per tile). Here we assign a full CUDA block to each tile (TILE SIZE of 16x16) and all threads in the block cooperate to build the tile histogram in parallel.

1.1 Reasoning

The baseline kernel `lut_init` used one thread per tile which leaves a lot of potential parallelism unused: for a $TILE\_SIZE \times TILE\_SIZE$ tile we have up to $TILE\_SIZE^2$ pixels but only one thread processing them. Since histogram computation is the most expensive part of CLAHE we want to:

- assigng multiple threads to a tile for better parallelism
- keep the histogram in shared memory to reduce global memory traffic and have faster memory accesses
- use atomic operations on shared memory to safely update the shared histogram

1.2 Changes in the Host Code

On the host side the structure of the program remains the same (load image, allocate device memory, copy input, launch kernels, copy output, measure time). The main difference is that we now launch one **block** per tile instead of one **thread** per tile.

In the previous implementation we computed the grid for `lut_init` as:

```
blocks_x = (grid_w + MAX_BLOCK_LENGTH - 1) / MAX_BLOCK_LENGTH;
blocks_y = (grid_h + MAX_BLOCK_LENGTH - 1) / MAX_BLOCK_LENGTH;
dim3 block_dim(MAX_BLOCK_LENGTH, MAX_BLOCK_LENGTH);
dim3 grid_dim_lut(blocks_x, blocks_y);
```

Listing 1: Baseline grid configuration for LUT initialization

In the shared histogram version we set:

```
printf("Max Block dims (%d, %d)\n", BLOCK_DIM, BLOCK_DIM);
printf("Lut_init grid dims (%d, %d)\n", grid_w, grid_h);
dim3 block_dim(BLOCK_DIM, BLOCK_DIM);
dim3 grid_dim_lut(grid_w, grid_h);

// for bilinear interpolation kernel, one pixel per thread
blocks_x = (width + BLOCK_DIM - 1) / BLOCK_DIM;
blocks_y = (height + BLOCK_DIM - 1) / BLOCK_DIM;
printf("bilinear_interp grid dims (%d, %d)\n", blocks_x, blocks_y);
dim3 grid_dim_bilinear(blocks_x, blocks_y);
```

Listing 2: Shared-histogram grid configuration

Now each block corresponds to exactly one tile (`grid_dim_lut = (grid_w, grid_h)`) and each block has `BLOCK_DIM` × `BLOCK_DIM` threads that cooperate to process that tile.

The rest of `main` (error checking, memory allocation, `cudaMemcpy`, timing, and the `bilinear_interp` launch) remains unchanged in structure so we don't include them here.

1.3 Shared Histogram Kernel

The main change is in the `lut_init` kernel. We fuse the previous `compute_histogram` into `lut_init` to avoid an extra function call and then instead of computing the histogram in a private local array inside a single thread we now:

- keep a single 256 bin histogram in `__shared__` memory per block
- have all threads in the block cooperatively build this histogram using `atomicAdd` on the shared array
- perform clipping, excess redistribution and LUT computation on this shared histogram.

Shared histogram `lut_init` kernel:

```
__global__ void lut_init(int *d_all_luts, unsigned char *d_img_in,
                         int height, int width) {
    int grid_w, grid_h;
    int tx, ty, x_start, y_start, actual_tile_w, actual_tile_h;
    int *current_lut_ptr;
    __shared__ int hist[256];
    __shared__ int excess;
    int x, y, i, avg_inc, val;
    int cdf = 0, total_pixels;
    int tid, total_threads;

    // Calculate grid dimensions
    grid_w = (width  + TILE_SIZE - 1) / TILE_SIZE;
    grid_h = (height + TILE_SIZE - 1) / TILE_SIZE;

    // Each block corresponds to a single tile (tx, ty)
    tx = blockIdx.x;
    ty = blockIdx.y;

    if (tx >= grid_w || ty >= grid_h) return;

    // Tile origin
    x_start = tx * TILE_SIZE;
    y_start = ty * TILE_SIZE;

    // Handle boundary tiles that might be smaller than TILE_SIZE
    actual_tile_w = (x_start + TILE_SIZE > width)  ? (width  - x_start) : TILE_SIZE;
    actual_tile_h = (y_start + TILE_SIZE > height) ? (height - y_start) : TILE_SIZE;

    // Pointer to the specific 256-entry LUT for this tile
    current_lut_ptr = &d_all_luts[(ty * grid_w + tx) * 256];

    total_pixels = actual_tile_w * actual_tile_h;

    // Thread ID within the block and total number of threads
    tid = threadIdx.y * blockDim.x + threadIdx.x;
    total_threads = BLOCK_DIM * BLOCK_DIM;

    // Initialize shared variables
    if (tid == 0) excess = 0;
    if (tid < 256) hist[tid] = 0;

    __syncthreads();

    // Build histogram cooperatively: each thread processes multiple pixels
    for (i = tid; i < total_pixels; i += total_threads) {
        x = x_start + (i % actual_tile_w);
        y = y_start + (i / actual_tile_w);
        // Boundary check mostly for the right/bottom edge tiles
        if (x < width && y < height) {
            unsigned char val_local = d_img_in[y * width + x];
            atomicAdd(&hist[val_local], 1);
        }
    }
    __syncthreads();

    // Clip histogram and accumulate excess
    if (tid < 256 && hist[tid] > CLIP_LIMIT) {
        atomicAdd(&excess, hist[tid] - CLIP_LIMIT);
        hist[tid] = CLIP_LIMIT;
    }
    __syncthreads();

    // Redistribute excess uniformly
    avg_inc = excess / 256;
    if (tid < 256) {
        hist[tid] += avg_inc;
    }
    __syncthreads();

    // Compute CDF & LUT (we let a single thread do the prefix sum)
    if (tid == 0) {
        for (i = 0; i < 256; ++i) {
            cdf += hist[i];
            val = (int)((float)cdf * 255.0f / total_pixels + 0.5f);
            if (val > 255)
                val = 255;
            current_lut_ptr[i] = val;
        }
    }
}
```

Listing 3: Fused lut and histogram CUDA kernel with shared histogram per tile

Compared to the original device side `compute_histogram` the main differences are:

- The histogram is now declared as `__shared__ int hist[256]` so it is shared by all threads in the block.
- Histogram updates use `atomicAdd` on shared memory to avoid race conditions when multiple threads update the same bin.
- The histogram initialization and clipping are done cooperatively, using `tid < 256` so that only the first 256 threads touch the 256 bins.

- The CDF and LUT computation is still done by a single thread (`tid == 0`) to keep the prefix sum simple in this optimization step.(we will address this in the next optimization)

The `bilinear_interp` kernel remains unchanged functionally and is identical to the one in the baseline GPU implementation, since this optimization targets only the histogram/LUT generation phase.

1.4 Performace

We consider this optimization failed due to the performance downgrade compared to the baseline implementation but solving bank conflicts and contention will "unlock" more throughput.

Below are the graphs showing the effect of this optimization on the time and throughput in comparison to the previous version:

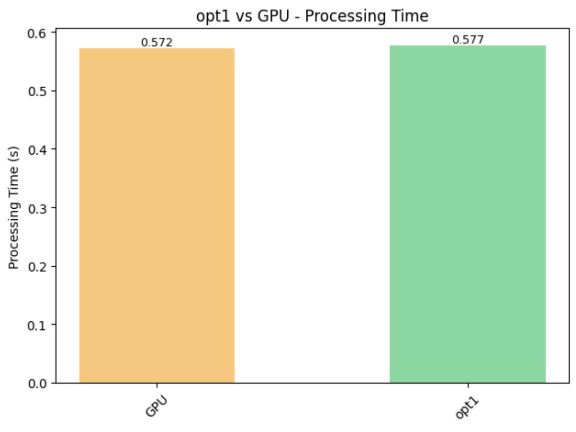


Figure 3: opt1 vs GPU - Processing Time

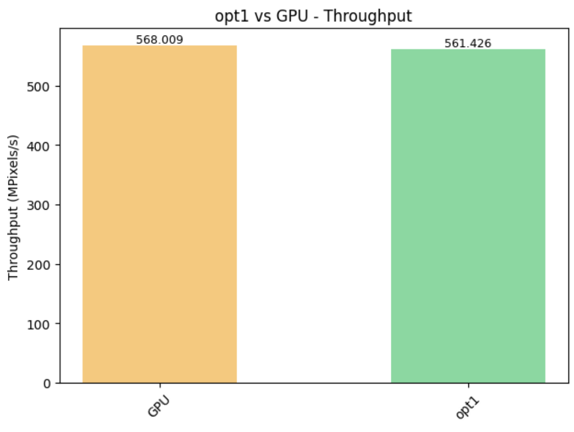


Figure 4: opt1 vs GPU - Throughput

Below are the graphs showing the effect of this optimization on the time and throughput in comparison to all previous versions:

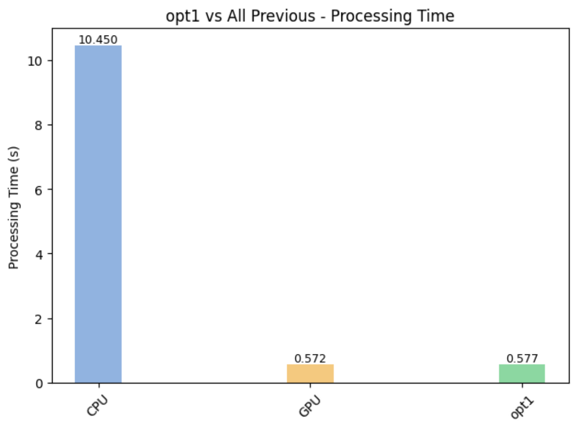


Figure 5: opt1 vs All Previous - Processing Time

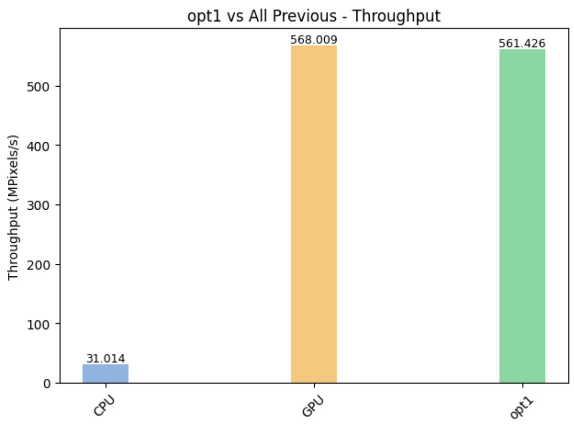


Figure 6: opt1 vs All Previous - Throughput

2 Optimization 2 - Split Shared Histogram

In the previous optimization we used a single shared histogram `__shared__ int hist[256]` per block and all threads in the block updated it cooperatively with `atomicAdd` in shared memory. But it suffered from two major issues:

- contention on the same 256 histogram bins
- shared memory bank conflicts when many threads in a warp hit nearby or identical bins.

In this optimization we split the block's threads into several *teams* each team maintaining its own private histogram in shared memory. We then reduce (sum) these per team histograms into a final shared histogram `hist[256]` before clipping and prefix sum. This reduces both contention and bank conflicts.

2.1 Previous Shared Histogram Kernel

In the previous `lut_init` kernel:

- One block per tile: `blockIdx.(x,y) = (tx,ty)`.
- All threads in the block share a single histogram:

```
__shared__ int hist[256];
__shared__ int excess;

tid = threadIdx.y * blockDim.x + threadIdx.x;
// initialize first 256 threads
if (tid == 0) excess = 0;
if (tid < 256) hist[tid] = 0;
__syncthreads();

```

- Each thread processes a strided subset of the tile's pixels and calls `atomicAdd` on `hist[val]`:

```
for (i = tid; i < total_pixels; i += total_threads) {
    x = x_start + (i % actual_tile_w);
    y = y_start + (i / actual_tile_w);
    if (x < width && y < height) {
        unsigned char val_local = d_img_in[y * width + x];
        atomicAdd(&hist[val_local], 1);
    }
}

```

This makes all threads compete on the same 256 shared memory locations. For images many pixels have similar intensities so many threads in a warp often attempt to update the same or nearby bins at the same time. This causes:

- serialization of the atomic operations
- frequent bank conflicts because multiple threads try to access the shame bank.

2.2 Bank Conflicts in the Previous Design

On our GPU shared memory is organized into:
$N_{\text{banks}} = 32$ banks. Each 32 bit word is assigned to a bank according to its address index:

$$\text{bank}(i) = i \bmod N_{\text{banks}}$$

Our histogram `hist` has 256 32 bit integers:

$$\texttt{hist[0]}, \texttt{hist[1]}, \ldots, \texttt{hist[255]}.$$

The bin index `val` (pixel intensity) determines which bank is accessed:

$$\text{bank}(\texttt{val}) = \texttt{val} \bmod 32.$$

Inside a warp suppose we have 32 threads indexed by $t = 0, 1, \ldots, 31$ each updating some bin `hist[val_t]`:

$$\text{bank}_t = \text{bank}(\texttt{val}_t) = \texttt{val}_t \bmod 32.$$

- If many threads see similar intensities (for example 100) then:

$$\text{bank}_t \approx 100 \bmod 32 = 4,$$

so most threads in the warp try to access bank 4 simultaneously.

- Even if intensities differ but are close, e.g. $\texttt{val}_t = 100, 101, \ldots, 131$, then:

$$\text{bank}_t = (100 + t) \bmod 32,$$

which still causes multiple threads to collide on the same bank within the warp.

Whenever two or more threads in a warp access different addresses in the same bank the accesses are serialized. Since all histogram updates go through the same `hist[256]` array this means that almost every atomic update has a chance of running into a bank conflict.

2.3 Logical Changes in Optimization 2

To reduce this contention and the probability of bank conflicts we add **thread teams** and multiple private histograms in shared memory. The key changes in the new `lut_init` kernel are:

1. We allocate a 2D array of histograms in shared memory:

```
__shared__ int priv_hist[THREAD_TEAMS][256];
__shared__ int hist[256];
__shared__ int excess;

```

   where THREAD_TEAMS is the number of teams per block and each team has its own 256 bin histogram.

2. We still have BLOCK_DIM * BLOCK_DIM threads per block. Each thread has a global linear thread ID:

```
tid = threadIdx.y * blockDim.x + threadIdx.x;
total_pixels = actual_tile_w * actual_tile_h;

```

3. All teams and the final histogram are initialized cooperatively:

```
if (tid == 0)
    excess = 0;

for (i = 0; i < THREAD_TEAMS; i++)
    priv_hist[i][tid] = 0;  // each bin across all teams

hist[tid] = 0;
__syncthreads();

```

4. Each thread is assigned to a team by dividing its `tid`:

```
int hist_idx = tid / THREADS_PER_TEAM;

```

   where:

$$\texttt{THREADS\_PER\_TEAM} = \frac{\texttt{TOTAL\_THREADS}}{\texttt{THREAD\_TEAMS}}.$$

   Hence,

$$\texttt{hist\_idx} \in \{0, 1, \ldots, \texttt{THREAD\_TEAMS} - 1\}.$$

5. In the histogram building loop each thread now updates its team's private histogram:

```
// Build Histogram
for (i = tid; i < actual_tile_h * actual_tile_w; i += TOTAL_THREADS) {
    x = x_start + (i % actual_tile_w);
    y = y_start + (i / actual_tile_w);
    if (x < width && y < height) {
        unsigned char val_local = d_img_in[y * width + x];
        atomicAdd(&(priv_hist[hist_idx][val_local]), 1);
    }
}
__syncthreads();

```

6. After all teams have filled their private histograms we reduce them into a single shared histogram `hist[256]`:

```
for (i = 0; i < THREAD_TEAMS; i++) {
    hist[tid] += priv_hist[i][tid];
}
__syncthreads();

```

   Then we perform clipping, excess redistribution and CDF/LUT computation as before using `hist[256]`.

2.4 Why This Reduces Bank Conflicts and Contention

Each private histogram `priv_hist[t][256]` is laid out in memory as a contiguous segment of 256 integers. We can think its indexing as:

$$\mathrm{addr}(t, b) = t \cdot 256 + b,$$

where t is the team index and b is the bin index ($0 \leq b < 256$). The corresponding bank index is:

$$\mathrm{bank}(t, b) = \mathrm{addr}(t, b) \bmod 32 = (t \cdot 256 + b) \bmod 32.$$

Because 256 is a multiple of 32, we have:

$$256 \bmod 32 = 0 \Rightarrow \mathrm{bank}(t, b) = b \bmod 32.$$

Thus for a fixed bin b all teams access the same bank as in the single histogram case. However the critical difference is that now:

- At any given moment threads in a warp are distributed across different teams and different pixels, so the pattern of (t, b) pairs is more scattered.
- Each `priv_hist[t]` is only being updated by a subset of the threads (its own team) and not by all threads in the block. This reduces the number of concurrent accesses to the same bank and the same bin.

The total number of threads per block as `TOTAL_THREADS` and the number of teams as `THREAD_TEAMS` then each team has:

$$\mathtt{THREADS\_PER\_TEAM} = \frac{\mathtt{TOTAL\_THREADS}}{\mathtt{THREAD\_TEAMS}}$$

Now each team handles only a fraction of the pixels and updates only its private 256 bins so we divide the contention and the probability of conflicting accesses by roughly `THREAD_TEAMS` which are 4.
We also have to mention that we experimented with the number of THREAD_TEAMS and these were the results:

Table 1: Effect of number of thread teams on performance and divergence

THREAD_TEAMS	**Throughput (MPixels/s)**	**Divergence (%)**
2	570	18
4	577	16
8	485	47
16	432	17
32	330	12
64	*Note: 64 teams exceed shared memory capacity*	

2.5 Remaining Sequential Parts

After the reduction:

- Clipping and excess computation on `hist[256]` is still done cooperatively using atomic updates to `excess` (once per bin).
- The CDF and LUT computation is still carried out by a single thread (`tid == 0`) as in the previous optimization. This step is only 256 iterations and is not the bottleneck.

2.6 Performance

Below are the graphs showing the effect of this optimization on the time and throughput in comparison to the previous version:

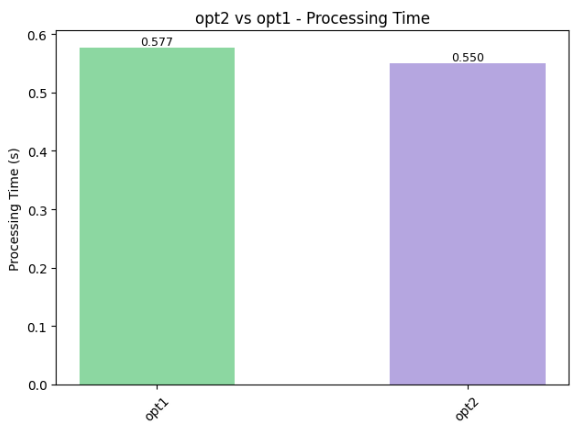


Figure 7: opt2 vs opt1 - Processing Time

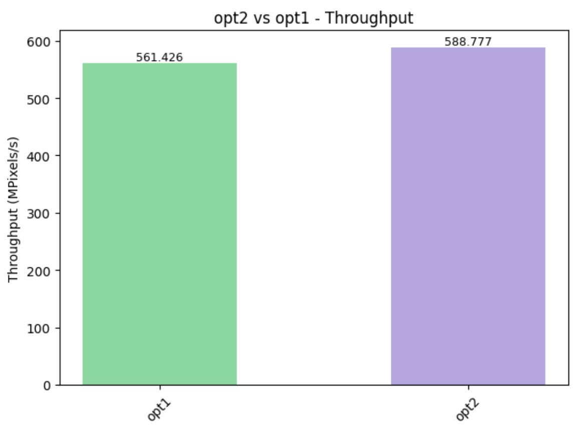


Figure 8: opt2 vs opt1 - Throughput

Below are the graphs showing the effect of this optimization on the time and throughput in comparison to all previous versions:

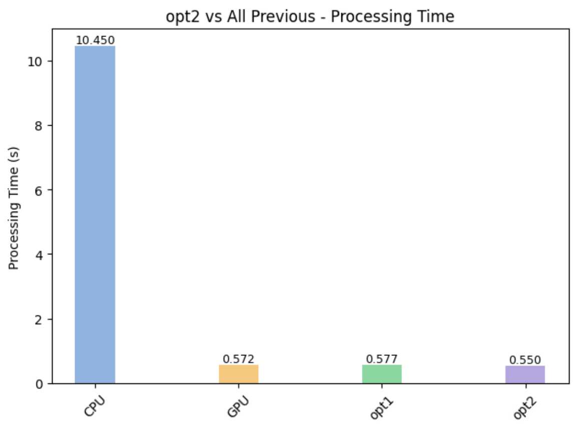


Figure 9: opt2 vs All Previous - Processing Time

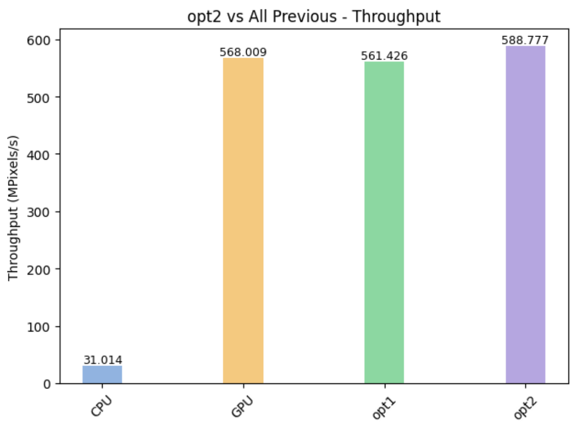


Figure 10: opt2 vs All Previous - Throughput

3 Optimization 3 - Shared Histogram and Prefix Sum

In the third optimization we try to speed up the CDF computation during LUT construction. Previously we used a simple serial prefix sum in each block, which leaves 255 threads idle while one thread loops over the 256 bins. Since every tile needs this CDF, this becomes an serial bottleneck. Here we replace that loop with a parallel scan in shared memory so that all threads in the block do work.

3.1 Reasoning

After building and clipping the histogram (Optimizations 1–2) each tile holds a 256 element array.

$$\mathrm{cdf}[i] = \sum_{k=0}^{i} \mathrm{hist}[k]$$

In the baseline GPU implementation we computed this using a simple loop and a single thread per tile:

```c
// Compute CDF & LUT
if (tid == 0) {
    for (i = 0; i < 256; ++i) {
        cdf += hist[i];
        // Calculate equalized value
        val = (int)((float)cdf * 255.0f / total_pixels + 0.5f);
        if (val > 255)
            val = 255;
        current_lut_ptr[i] = val;
    }
}
```

This creates a serial bottleneck inside each block, wasting 255 threads which remain idle. Since CLAHE involves hundreds or thousands of tiles, this adds up quickly. Using shared memory and a parallel prefix sum fixes this bottleneck.

We choose the **work-efficient parallel scan algorithm**, ruduce + reverse step because:

- It is better suited for shared memory
- Avoids the loop carried dependency that is in the serial CDF computation.
- No warp divergence.

Reduction Phase

During reduction phase we build a reduction tree. At step d, each thread with index i performs:

$$\text{if } i \bmod 2^{d+1} = 2^{d+1} - 1 \text{ then} \qquad A[i] = A[i] + A[i - 2^d].$$

The final element $A[n-1]$ contains the total sum.

Reverse Phase

Before reverse we set the last element to zero:

$$A[n-1] = 0.$$

At step d (in reverse order), each thread with index i that satisfies the same condition performs:

$$\begin{aligned} t &= A[i - 2^d], \\ A[i - 2^d] &= A[i], \\ A[i] &= A[i] + t. \end{aligned}$$

This distributes the prefix sums back down the tree.

Example for an Array of Size 8

Let the input array be:

$$[1, 2, 3, 4, 5, 6, 7, 8].$$

Reduction:

$$\begin{gathered} [1, 2, 3, 4,\ 5, 6, 7, 8] \\ [1, 2, 3, 4,\ 5, 6, 7, 15] \\ [1, 2, 3, 4,\ 5, 6, 7, 15] \\ [1, 2, 3, 4,\ 5, 6, 7, 36] \end{gathered}$$

Set last to zero:

$$[1, 2, 3, 4, 5, 6, 7, 0]$$

Reverse:

$$[1,2,3,4,5,6,7,0]$$
$$[1,2,3,4,5,6,7,15]$$
$$[1,2,3,4,5,6,10,15]$$
$$[1,2,3,4,5,6,10,36]$$

The final exclusive prefix sum is:

$$[0,1,3,6,10,15,21,28].$$

Number of Threads

Since each thread in step d processes a pair of elements, the number of active threads is:

$$\frac{n}{2^{d+1}} \quad \Rightarrow \quad \text{maximum is } \frac{n}{2}.$$

3.2 Implementation in CUDA

First each thread loads one histogram value into the shared array temp. The reduce phase performs a tree based parallel accumulation where threads at increasing strides add partial sums toward the end of the array. Once the reduction completes the last element is set to zero which converts the scan into an exclusive prefix sum.

In the reverse phase the algorithm walks back down the tree exchanging values and creates prefix sums to all positions. After this step temp contains the exclusive scan of the histogram.

Since the CLAHE CDF requires an inclusive prefix sum, each thread adds its original histogram value hist[tid] to the exclusive result. The final cumulative value is written back to temp[tid] and then normalized to form the corresponding LUT entry.

```
 __shared__ int temp[256];

temp[tid] = hist[tid];
__syncthreads();

// reduce phase
for (stride = 1; stride < 256; stride <<= 1) {
    idx = (tid + 1) * stride * 2 - 1;
    if (idx < 256)
        temp[idx] += temp[idx - stride];
        __syncthreads();
}

// Set last element to zero for
if (tid == 255)
    temp[255] = 0;
__syncthreads();

// reverse phase
for (stride = 128; stride > 0; stride >>= 1) {
    idx = (tid + 1) * stride * 2 - 1;
    if (idx < 256) {
        swap_val = temp[idx - stride];
        temp[idx - stride] = temp[idx];
        temp[idx] += swap_val;
    }
    __syncthreads();
}

cdf = temp[tid] + hist[tid];
temp[tid] = cdf;
__syncthreads();

// Store LUT
if (tid < 256) {
    val = (int)((float)temp[tid] * 255.0f / total_pixels + 0.5f);
    if (val > 255) val = 255;
    current_lut_ptr[tid] = val;
}
```

3.3 Performance

Below are the graphs showing the effect of this optimization on the time and throughput in comparison to the previous version:

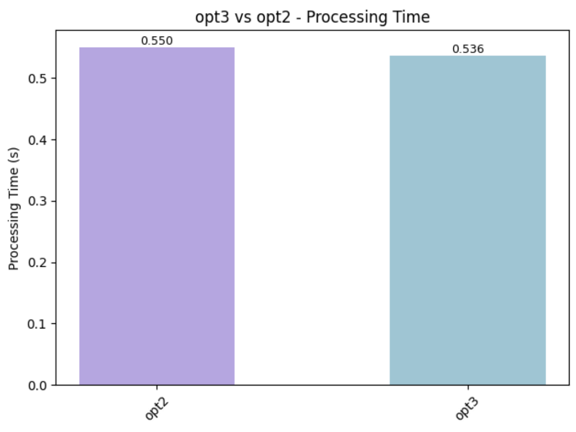


Figure 11: opt3 vs opt2 - Processing Time

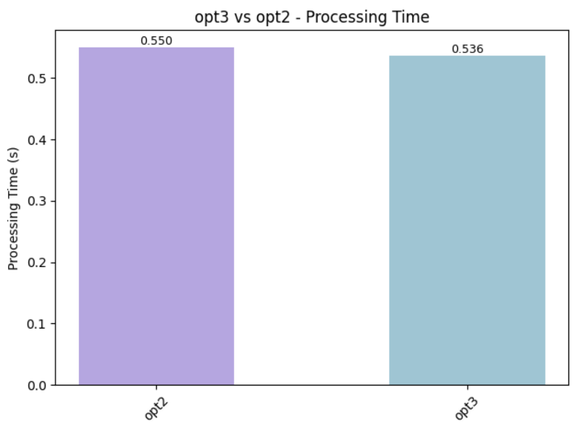


Figure 12: opt3 vs opt2 - Throughput

Below are the graphs showing the effect of this optimization on the time and throughput in comparison to all previous versions:

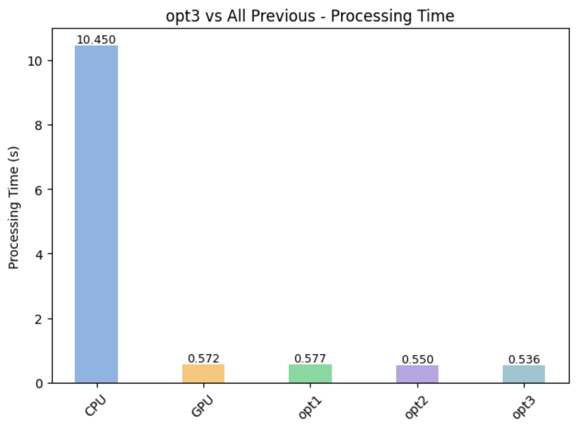


Figure 13: opt3 vs All Previous - Processing Time

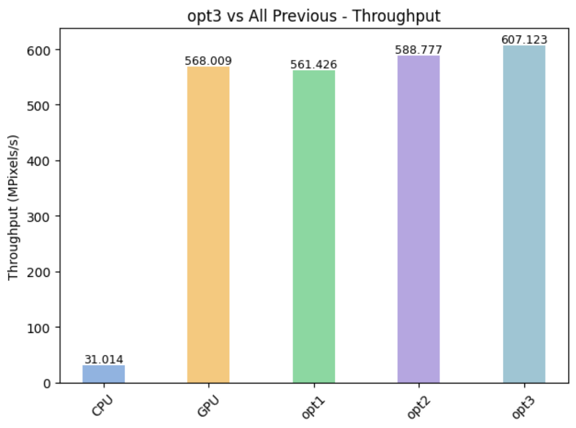


Figure 14: opt3 vs All Previous - Throughput

4 Optimization 4 - Prefix Sum with Padding

After we added the parallel prefix sum in Optimization 3 to compute the CDF on the GPU we observed that the scan phase itself can suffer from shared memory bank conflicts. In this optimization we keep the same overall structure as in Optimization 3 (thread teams, private histograms, reduction into a single shared histogram) but we modify the prefix sum implementation to use a padde shared array in order to reduce bank conflicts during the up sweep and down sweep phases.

4.1 Bank Conflicts in the Unpadded Scan

To compute the cumulative distribution function (CDF) for each tile on the GPU we use a parallel prefix sum over the 256 bin histogram stored in shared memory. After we have built and clipped the histogram in `hist` we copy it into a temporary shared array and perform an in place scan:

```
__shared__ int hist[256];
__shared__ int temp[256];

temp[tid] = hist[tid];
__syncthreads();

// reduce phase
    for (stride = 1; stride < 256; stride <<= 1) {
        idx = (tid + 1) * stride * 2 - 1;
        if (idx < 256)
            temp[idx] += temp[idx - stride];
        __syncthreads();
    }

    // Set last element to zero for
    if (tid == 255)
        temp[255] = 0;
    __syncthreads();

    // reverse phase
    for (stride = 128; stride > 0; stride >>= 1) {
        idx = (tid + 1) * stride * 2 - 1;
        if (idx < 256) {
            swap_val = temp[idx - stride];
            temp[idx - stride] = temp[idx];
            temp[idx] += swap_val;
        }
        __syncthreads();
    }

    cdf = temp[tid] + hist[tid];
    temp[tid] = cdf;
    __syncthreads();

    // Store LUT
    if (tid < 256) {
        val = (int)((float)temp[tid] * 255.0f / total_pixels + 0.5f);
        if (val > 255) val = 255;
        current_lut_ptr[tid] = val;
    }
}
```

Each iteration of the reduce phase combines partial sums separated by a stride `offset`. For a given `offset`, thread `tid` computes:

$$\texttt{idx} = (\texttt{tid} + 1) \cdot 2 \cdot \texttt{stride} - 1$$

and if `idx < 256` updates

$$\texttt{temp[idx]} \leftarrow \texttt{temp[idx]} + \texttt{temp[idx - stride]}.$$

In the final iteration `temp` holds the total sum at the last element.

In the reverse we propagate prefix sums back through the tree. For the same index pattern `idx` each active thread swaps and adds:

$$\begin{aligned} t &= \texttt{temp[idx - stride]}, \\ \texttt{temp[idx - stride]} &= \texttt{temp[idx]}, \\ \texttt{temp[idx]} &= \texttt{temp[idx]} + t. \end{aligned}$$

This transforms the reduced array into an exclusive prefix sum. Finally we convert it to an inclusive CDF by adding back the original histogram bin:

$$\texttt{cdf[tid]} = \texttt{temp[tid]} + \texttt{hist[tid]}.$$

Each thread then maps its bin's CDF entry to a LUT value in the range $[0, 255]$ and writes it to the LUT of the respective tile. This implementation already parallelizes the CDF computation across 256 threads but the access pattern to `temp[idx]` and `temp[idx - stride]` during the reduce and reverse can cause multiple threads in a warp to hit the same shared memory bank and this is leading to bank conflicts.

We have 32 shared memory banks and each `int` in `temp[256]` occupies one word in shared memory. The hardware assigns consecutive `temp` elements to consecutive banks in this pattern: the first 32 elements go to banks 0–31, the next 32 elements again go to banks 0–31, and so on. In other words, elements that are 32 positions apart in the array always end up in the same bank.

Each thread repeatedly reads and writes two positions in `temp`: `idx` and `idx - stride`. The value of `idx` is computed from the thread index and the current stride `stride` inside the loop. For a given `stride` the active threads in a warp will touch a specific pattern of indices in `temp`. Because the scan doubles the stride at each step (1, 2, 4, 8, . . .) these indices are regularly spaced in memory and are often separated by distances like 2, 4, 8, 16, 32, 64, . . .

Whenever two indices differ by a multiple of 32 they are stored in the same bank. This means that in some of the scan steps several threads in the same warp end up reading or writing different locations that share the same bank which is a bank conflict. The hardware must then serialize those accesses which is exactly what we mean by a bank conflict. Since the scan runs several such steps in both the reduce and reverse phases these conflicts appear repeatedly and slow down the prefix-sum computation.

4.2 Optimization 4: Padded Prefix Sum

To reduce these conflicts we keep exactly the same scan logic but change how we lay out the temporary array in shared memory. We introduce a padded index:

```
#define PAD(i)  ( (i) + ((i) >> 5) )
```

and allocate a slightly larger shared array:

```
__shared__ int temp[256 + 8];
```

The macro `PAD(i)` adds one extra element after every 32 logical entries since:

$$\left\lfloor \frac{i}{32} \right\rfloor = i >> 5.$$

So logical element i is stored at physical index

$$\text{index}(i) = i + \left\lfloor \frac{i}{32} \right\rfloor$$

which means that each block of 32 logical elements occupies 33 positions in shared memory. This way every 32 elements do not warp back to the same bank but with each iteration they offset by 1 from the previous one and hence we avoid bank conflicts.

We now load and access the scan array using this padded indexing:

```
// Load with padding
temp[PAD(tid)] = hist[tid];
__syncthreads();

// reduction phase
for (stride = 1; stride < 256; stride <<= 1) {
    idx = (tid + 1) * stride * 2 - 1;
    if (idx < 256)
        temp[PAD(idx)] += temp[PAD(idx - stride)];
    __syncthreads();
}

// Set last element to zero for exclusive scan
if (tid == 255)
    temp[PAD(255)] = 0;
__syncthreads();

// reverse phase
for (stride = 128; stride > 0; stride >>= 1) {
    int idx = (tid + 1) * stride * 2 - 1;
    if (idx < 256) {
        int t = temp[PAD(idx - stride)];
        temp[PAD(idx - stride)] = temp[PAD(idx)];
        temp[PAD(idx)] += t;
    }
    __syncthreads();
}

```

```
// Inclusive conversion
cdf = temp[PAD(tid)] + hist[tid];
__syncthreads();

// LUT write
val = (int)((float)cdf * 255.0f / total_pixels + 0.5f);
current_lut_ptr[tid] = (val > 255 ? 255 : val);
```

4.3 Why the Padding Reduces Bank Conflicts

Shared memory is organized in 32 banks and our `int` array is laid out so that consecutive elements go to consecutive banks. In the unpadded case the first 32 elements of `temp` use banks 0–31, the next 32 elements again use banks 0–31 and so on. Any two indices that are 32 positions apart end up in the same bank.

The scan accesses `temp` in a pattern: in each iteration active threads read and write pairs of entries separated by a power of two. As the stride doubles (1, 2, 4, 8, . . .) the indices accessed by a warp are separated by 32, 64, 96, etc. positions. In the unpadded `temp[256]`, these distances map threads back onto the same subset of banks which forces the hardware to serialize many of those accesses and causes bank conflicts.
With the padded layout every group of 32 logical elements is shifted by one extra position in shared memory. The first 32 logical elements still occupy banks 0–31, but the next 32 elements are shifted by one bank, the next by two banks, and so on. Indices that used to lie exactly 32 positions apart (and thus in the same bank) no longer line up in such a simple repeating pattern.

The access pattern of the scan (with strides 1, 2, 4, 8, . . .) is spread more evenly across the 32 banks instead of concentrating many accesses on the same few banks. This reduces how often multiple threads in a warp contend for the same bank and lowers the amount of serialization during the prefix sum computation.

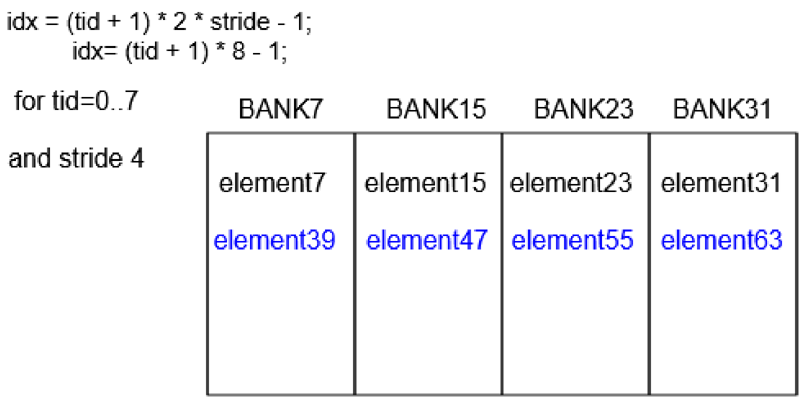
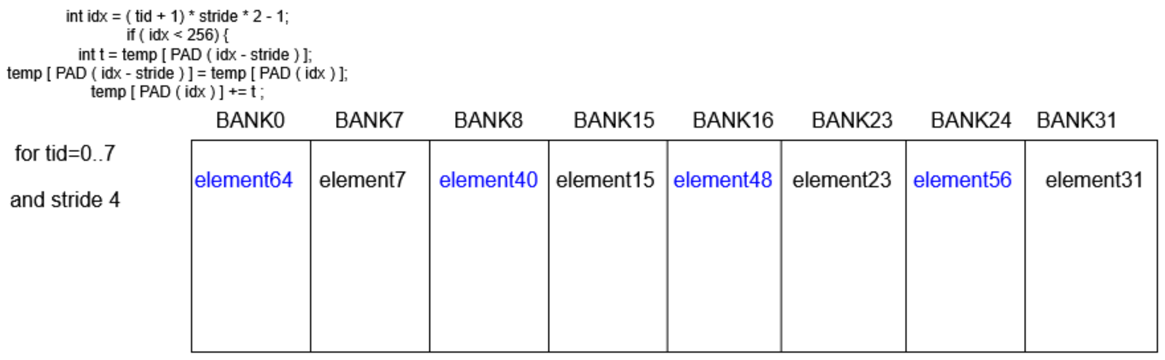


Figure 15: Bank contents in upadded version



```
int idx = ( tid + 1) * stride * 2 - 1;
if ( idx < 256) {
int t = temp [ PAD ( idx - stride ) ];
temp [ PAD ( idx - stride ) ] = temp [ PAD ( idx ) ];
temp [ PAD ( idx ) ] += t ;
```

Figure 16: Bank contents in padded version

We show one reduce iteration with stride = 4 for threads tid = 0..7. Each thread updates a logical index idx = (tid + 1) * stride * 2 - 1, which in this step corresponds to indices 7, 15, 23, 31, 39, 47, 55, and 63. In the unpadded layout these indices are mapped to banks 7, 15, 23, 31, 7, 15, 23, and 31, so pairs of threads (0,4), (1,5), (2,6), and (3,7) access the same bank. This creates four conflicts that the hardware must serialize. In the padded layout we store element i at PAD(i) = i + (i » 5) which shifts indices 39, 47, 55, and 63 to physical positions 40, 48, 56, and 64. As a result the same logical indices now hit banks 7, 15, 23, 31, 8, 16, 24, and 0 so each thread in the example touches a different bank.

4.4 Performance

Below are the graphs showing the effect of this optimization on the time and throughput in comparison to the previous version:

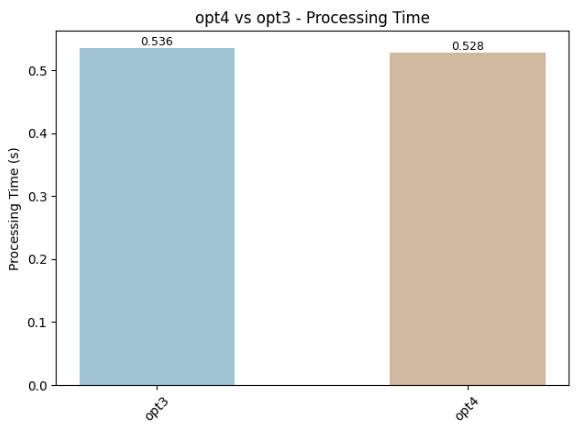


Figure 17: opt4 vs opt3 - Processing Time

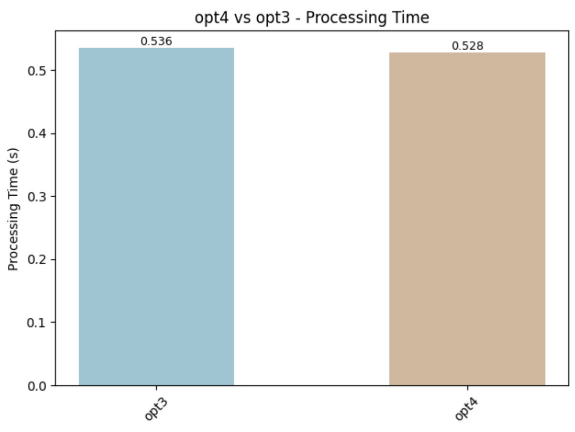


Figure 18: opt4 vs opt3 - Throughput

Below are the graphs showing the effect of this optimization on the time and throughput in comparison to all previous versions:

5 Streams

Next we looked at the host–device transfers. In this optimization we:

- Pinned host memory for the input and output images.
- A CUDA stream to issue asynchronous memory copies and kernel launches.

On the input side, `read_pgm` previously used a regular `malloc`:

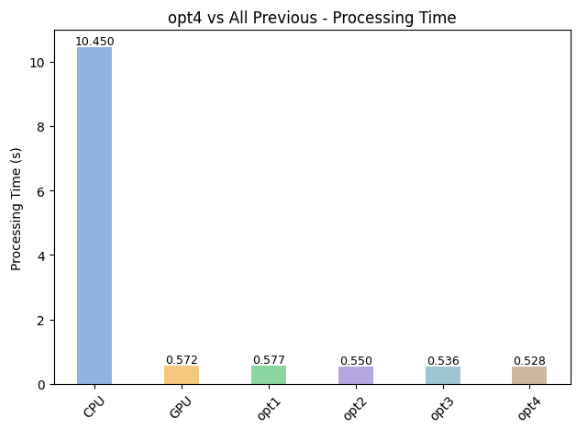


Figure 19: opt4 vs All Previous - Processing Time

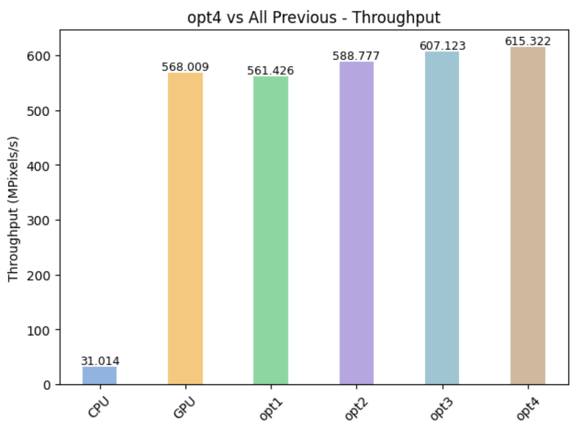


Figure 20: opt4 vs All Previous - Throughput

```
result.img = (unsigned char *)malloc(result.w * result.h
                                     * sizeof(unsigned char));
```

Listing 4: Original pageable host allocation in read$_pgm$

In Optimization 5 we replace this with a pinned allocation using `cudaMallocHost` which allows the runtime to perform true asynchronous DMA transfers:

```
cudaError_t err_runtime = cudaMallocHost((void**)&result.img,
                                         result.w * result.h
                                         * sizeof(unsigned char));
if (err_runtime != cudaSuccess) {
    cudaDeviceReset();
    exit(1);
}
```

Listing 5: Pinned host allocation in read$_pgm$

Similarly in `main` we no longer allocate the output image with `malloc` but instead use a pinned buffer:

```
h_img_out.w = width;
h_img_out.h = height;

// Previously: h_img_out.img = (unsigned char*)malloc(...);
cudaError_t error_out_h = cudaMallocHost((void**)&h_img_out.img,
                                         height * width
                                         * sizeof(unsigned char));
```

Listing 6: Pinned host allocation for output image

The second change is adding a CUDA stream and the conversion of all transfers to asynchronous calls. In the baseline version we used synchronous copies and default stream launches:

```
// Synchronous copy to device
cudaMemcpy(d_img_in.img, h_img_in.img,
           height * width * sizeof(unsigned char),
           cudaMemcpyHostToDevice);

// Kernels in default stream
lut_init<<<grid_dim_lut,      block_dim>>>(...);
bilinear_interp<<<grid_dim_bilinear, block_dim>>>(...);

// Synchronous copy back to host
cudaMemcpy(h_img_out.img, d_img_out.img,
           height * width * sizeof(unsigned char),
           cudaMemcpyDeviceToHost);
```

Listing 7: Baseline synchronous H2D/D2H transfers and kernel launches

Instead we create a stream `s1`, use `cudaMemcpyAsync` for both directions and launch the kernels in that stream:

```
// Create a CUDA stream
cudaStream_t s1;
cudaError_t error_runtime = cudaStreamCreate(&s1);
CHECK_CUDA_FAIL(error_runtime);

// Asynchronous host-to-device copy in stream s1
error_runtime = cudaMemcpyAsync(d_img_in.img, h_img_in.img,
                                height * width * sizeof(unsigned char),
                                cudaMemcpyHostToDevice, s1);
CHECK_CUDA_FAIL(error_runtime);

// Launch kernels in stream s1
lut_init<<<grid_dim_lut,      block_dim, 0, s1>>>(d_all_luts, d_img_in.img,
                                                  height, width,
                                                  grid_w, grid_h);
error_runtime = cudaGetLastError();
CHECK_CUDA_FAIL(error_runtime);

bilinear_interp<<<grid_dim_bilinear, block_dim, 0, s1>>>(
    d_img_in.img, d_img_out.img, d_all_luts, height, width);
error_runtime = cudaGetLastError();
CHECK_CUDA_FAIL(error_runtime);

// Asynchronous device-to-host copy in stream s1
error_runtime = cudaMemcpyAsync(h_img_out.img, d_img_out.img,
                                height * width * sizeof(unsigned char),
                                cudaMemcpyDeviceToHost, s1);
CHECK_CUDA_FAIL(error_runtime);

// Wait for all work in s1 to finish
error_runtime = cudaStreamSynchronize(s1);
CHECK_CUDA_FAIL(error_runtime);

cudaStreamDestroy(s1);
```
Listing 8: Asynchronous transfers and kernel launches in a CUDA stream

5.1 Performance

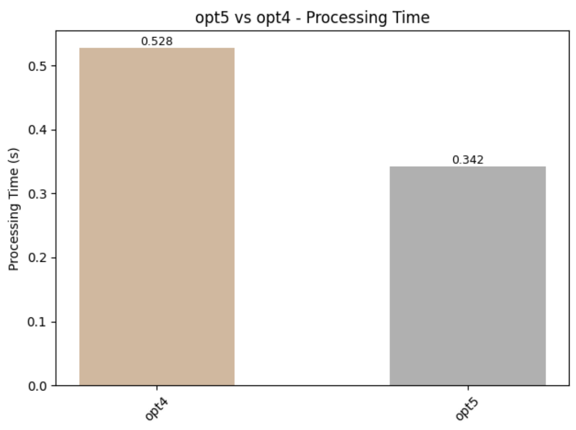


Figure 21: opt5 vs opt4 - Processing Time

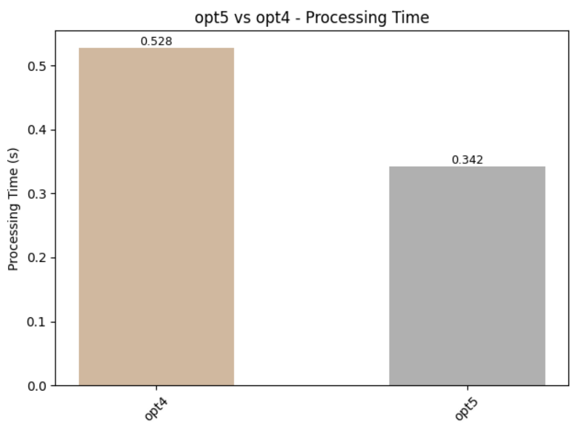


Figure 22: opt5 vs opt4 - Throughput

Below are the graphs showing the effect of this optimization on the time and throughput in comparison to all previous versions:

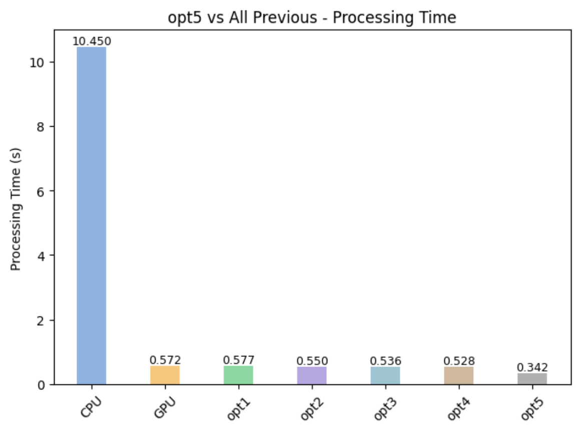


Figure 23: opt5 vs All Previous - Processing Time

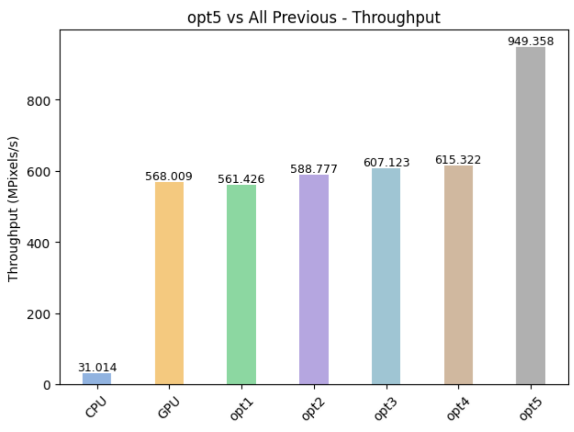


Figure 24: opt5 vs All Previous - Throughput

6 Branch Removal, Precomputation, and Memory-Access Optimizations

Finally we apply optimizations like removing branches that cause warp divergence, precomputing values that would otherwise be recalculated thousands of times and giving the compiler additional information about memory access patterns using `const` and `__restrict__`.

6.1 bilinear_interp kernel

Here we replaced the branches that introduced warp divergence:

```
if (x1 < 0)
    x1 = 0;
if (x2 >= grid_w)
    x2 = grid_w - 1;
if (y1 < 0)
    y1 = 0;
if (y2 >= grid_h)
    y2 = grid_h - 1;
```

with these with branch-free expressions using max() and min():

```
y1 = max(0, min(y1, grid_h - 1));
y2 = max(0, min(y2, grid_h - 1));
x1 = max(0, min(x1, grid_w - 1));
x2 = max(0, min(x2, grid_w - 1));
```

We also precompute the divsion used to convert pixel coordinates into tile-space coordinates.

```
ty_f = (float)y / TILE_SIZE - 0.5f;
tx_f = (float)x / TILE_SIZE - 0.5f;
```

We replaced this with:

```
const float inv_tile = 1.0f / TILE_SIZE;
ty_f = y * inv_tile - 0.5f;
tx_f = x * inv_tile - 0.5f;
```

Additionally, we precompute frequently used index expressions:

```
int img_idx = y * width + x;

int lut_row1 = y1 * grid_w * 256;
int lut_row2 = y2 * grid_w * 256;
int lut_x1   = x1 * 256;
int lut_x2   = x2 * 256;

tl = all_luts[lut_row1 + lut_x1 + val];
tr = all_luts[lut_row1 + lut_x2 + val];
bl = all_luts[lut_row2 + lut_x1 + val];
```

```
br = all_luts[lut_row2 + lut_x2 + val];

...
d_img_out[img_idx] = (unsigned char)(final_val + 0.5f);
```

We also pass grid_w and grid_h, as arguments to the kernel:

```
grid_w = (width + TILE_SIZE - 1) / TILE_SIZE;
grid_h = (height + TILE_SIZE - 1) / TILE_SIZE;
```

Passing them as parameters avoids redundant computation across thousands of threads.
Finally, we mark read-only arrays as const, and pointers accessed uniquely as __restrict__, which enables the compiler to apply additional optimizations.

```
__global__ void bilinear_interp(unsigned char *d_img_in, unsigned char *d_img_out, int *
    all_luts, int height, int width);
```

6.2 lut_init kernel

Here we also replaced branches with equivalent branch-free expressions in order to avoid divergence inside a warp. For example, in the histogram clipping stage we originally had:For example, in the histogram clipping stage we originally had:

```
if (hist[tid] > CLIP_LIMIT) {
    atomicAdd(&excess, (hist[tid] - CLIP_LIMIT));
    hist[tid] = CLIP_LIMIT;
}
```

This was replaced with the following branch-free form:

```
int excess_local = max(0, hist[tid] - CLIP_LIMIT);
atomicAdd(&excess, excess_local);
hist[tid] = min(hist[tid], CLIP_LIMIT);
```

The logic is identical, but every thread now executes the same instructions.
Similarly, we replaced the ternary operator used to clamp LUT output values:

```
current_lut_ptr[tid] = (val > 255 ? 255 : val);
```

with its equivalent branch-free expression:

```
current_lut_ptr[tid] = min(val, 255);
```

Next, we replaced the boundary-handling branches when computing the actual tile width and height on the image edges

```
actual_tile_w = (x_start + TILE_SIZE > width) ? (width - x_start) : TILE_SIZE;
actual_tile_h = (y_start + TILE_SIZE > height) ? (height - y_start) : TILE_SIZE;
```

with a branch-free version:

```
actual_tile_h = min(TILE_SIZE, height - y_start);
```

Finally, we added const and __restrict__ to input arrays to enable better memory-access optimizations:

```
__global__ void lut_init(int *d_all_luts, const unsigned char *__restrict__ d_img_in,
    int height, int width, int grid_w, int grid_h)
```

6.3 Performance

Below are the graphs showing the effect of this optimization on the time and throughput in comparison to all previous versions:

7 Note

For all the performance benchmarking we used the image `senator.pgm` and compiler flag -04

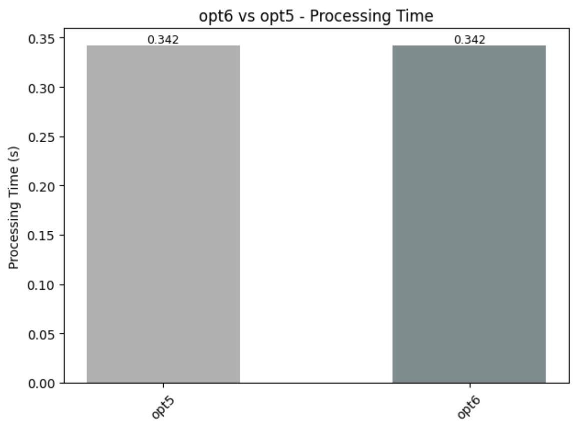


Figure 25: opt6 vs opt5 - Processing Time

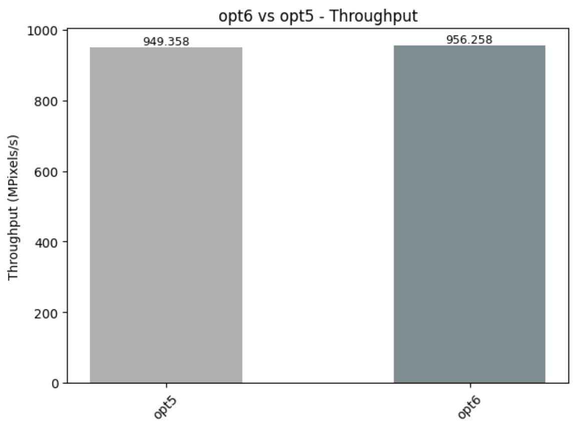


Figure 26: opt6 vs opt5 - Throughput

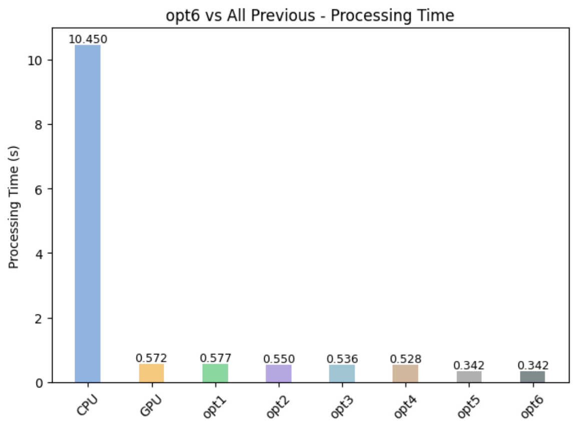


Figure 27: opt6 vs All Previous - Processing Time

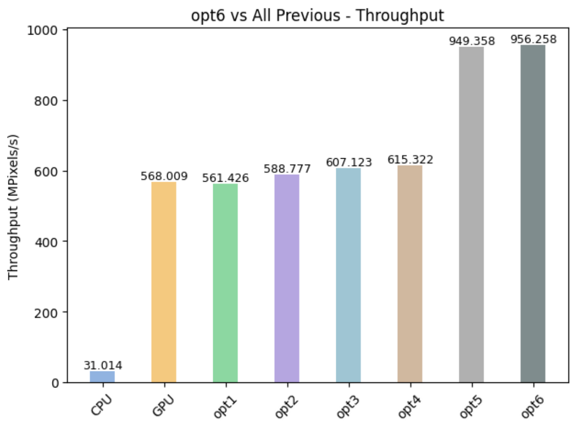


Figure 28: opt6 vs All Previous - Throughput