De La Salle University, Manila
College of Computer Studies

Term 1, A.Y. 2025-2026

CSC612M (Advanced Computer Architecture)

INTEGRATING PROJECT MILESTONE #3

Implementing Template-Matching Object Tracking in SIMT using CUDA

GROUP 3

Guillermo, Juliana Isabelle A.

Ty, Darryl Johnson T.

Ughoc, Daniella A.

November 2025

I. Introduction

This document presents Milestone #3 of our Integrating Project for the Template-Matching Object Tracking system using SIMT and CUDA parallelization. Building upon the multi-resolution C implementations (240p, 480p, 720p) and CUDA work completed in Milestone #2, this milestone focuses on optimizing and comparing different parallel execution strategies.

To reiterate, the dataset used in our project came from Kaggle. The dataset comprised of 130 MP4 videos of drivers exhibiting various head movements. The resolution of the videos in the original dataset is HD or 720p (1280 pixels x 720 pixels). For our project, we arbitrarily chose one video from the dataset and created additional versions of it in 240p and 480p resolutions by using an online video resizer tool. After obtaining the resized videos, we proceeded to extract the image frames in JPG format using an online video frame extractor tool. As such, the resolutions of our input frames are 240p (426x240), 480p (854x480), and 720p (1280x720). From the extracted frames, we only used 8 frames as input since using all frames would result in a long execution time. The template image was obtained by cropping the region of interest from the first frame input. Then, the output of the program are the processed frames in JPG format and the coordinates of the best match location for each of the images.

To achieve optimal GPU performance, we implemented different parallelization techniques including shared memory optimization, and tilling strategies. In order to compare execution times between sequential and parallel versions at all three resolutions, we thoroughly profiled and benchmarked these implementations using NVIDIA tools (nvprof and nsys). This document includes our parallel algorithm implementations, detailed performance benchmarks, execution time comparisons, and analysis of optimization trade-offs.

II. Object Tracking in CUDA - Parallel Algorithms Implemented

A. Grayscale Conversion Mapping

The algorithm applied to parallelize the RGB to grayscale conversion function is what Mattson would call Loop Parallelism (also known as Mapping) in the book Patterns for Parallel Programming. Loop Parallelism describes a program structure where a loop iterates over a set of data where the iterations are executed concurrently. In the color_to_grayscale_kernel function, where the GPU equivalent of color_to_grayscale from the C program is done, the set of data in this loop is the pixels in the image. Data parallelism is also done since multiple threads are dispatched to perform the same operation on different pixels in parallel.

In this function, each pixel, represented by coordinates (x, y) is handled by a separate CUDA thread. The kernel computes the thread's x and y values from blockIdx, blockdim, and threadIdx. It then reads the RGB values from the colored image, and then applies the luminance formula of gray=0.299R+0.587G+0.114B. It then writes the calculated grayscale value to gray_img[y * width + x].

Sequential C

```c
void color_to_grayscale(const unsigned char *color_img, unsigned char *gray_img, int width, int height, int channels) {
    //loop through each row of the image
    for (int y = 0; y < height; ++y) {
        //loop through each pixel(column)
        for (int x = 0; x < width; ++x) {
            //color_idx is the index of the red component of the current pixel
            int color_idx = (y * width + x) * channels;
            unsigned char r = color_img[color_idx];
            unsigned char g = color_img[color_idx + 1];
            unsigned char b = color_img[color_idx + 2];

            double gray_val = 0.299f * r + 0.587f * g + 0.114f * b;

            //unsigned char ranges from 0-255
            gray_img[y * width + x] = (unsigned char)gray_val;
        }
    }
}
```

CUDA

```c
__global__
void color_to_grayscale_kernel(const unsigned char *color_img, unsigned char *gray_img, int width, int height, int channels) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < width && y < height) {
        //index for the gray_img
        int gray_idx = y * width + x;
        //index for color_img
        int color_idx = gray_idx * channels;

        unsigned char r = color_img[color_idx];
        unsigned char g = color_img[color_idx + 1];
        unsigned char b = color_img[color_idx + 2];

        gray_img[gray_idx] = (unsigned char)(0.299f * r + 0.587f * g + 0.114f * b);
    }
}
```

Figure 1. Grayscale conversion implementation in C and CUDA.

B. Maximum Similarity Reduction Kernel

The *find_max_kernel* turns the usual process of scanning for the highest value in the similarity map into a parallel one. Instead of letting one thread go through the map step by step, every thread reads one value and places it in shared memory. Threads inside the same block then compare the values until only the block's largest value remains.

After this step, each block writes its result to a smaller output array. A second pass repeats the same process but only on the block results, this time using just one block since the data is already much smaller. This then produces the global maximum similarity value and its index, which makes this much faster because many threads help with the search instead of only one.

Figure 2. Reduction step implementation in C and CUDA.

C. Shared Memory Template Optimization

In the first version of the matching kernel, each thread read template pixels directly from global memory whenever it needed them, which meant the same values were fetched many times across many threads. In the shared memory version, threads work together to copy the template into a shared array once at the start of the computation.

After all threads sync, every comparison uses this shared copy instead of going back to global memory. This reduces repeated memory access and lets the GPU spend more time computing and less time waiting for memory transfers.

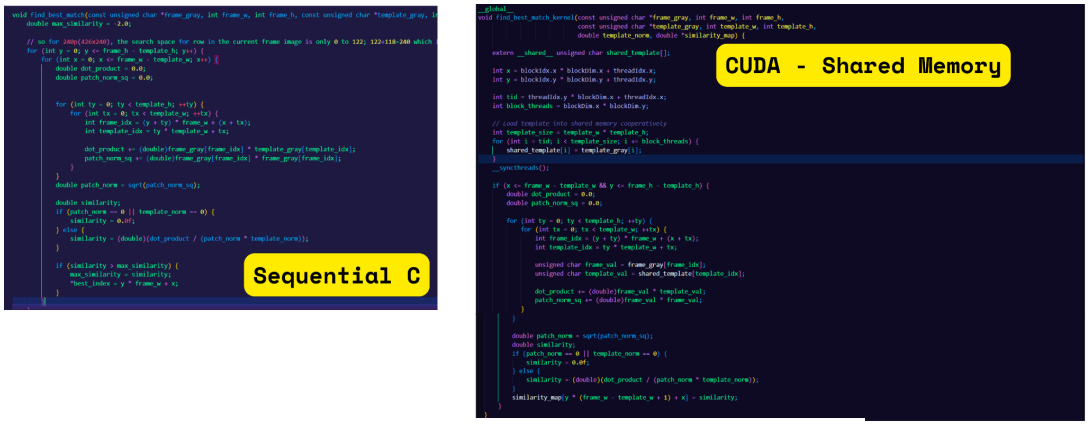

Figure 3. Implementation of Shared Memory in CUDA.

D. Tiled Template Processing

When the template does not fit well in shared memory, the tiled kernel breaks it into smaller blocks. The threads load one tile at a time into the shared memory, sync, and use that tile to update their similarity calculations. Once they finish with one tile, they move to the next until the entire template is covered.

This approach would keep memory usage manageable and would avoid loading the entire template at once. It would also help the GPU reuse data efficiently and handle larger templates and higher resolutions without running out of shared memory.

CUDA 720p without Tilling

```
__global__
void find_best_match_kernel(const unsigned char *frame_gray, int frame_w, int frame_h,
                            const unsigned char *template_gray, int template_w, int template_h,
                            double template_norm, double *similarity_map) {

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

    if (x <= frame_w - template_w && y <= frame_h - template_h) {
        double dot_product = 0.0;
        double patch_norm_sq = 0.0;

        for (int ty = 0; ty < template_h; ++ty) {
            for (int tx = 0; tx < template_w; ++tx) {
                int frame_idx = (y + ty) * frame_w + (x + tx);
                int template_idx = ty * template_w + tx;

                unsigned char frame_val = frame_gray[frame_idx];
                unsigned char template_val = template_gray[template_idx];

                dot_product += (double)frame_val * template_val;
                patch_norm_sq += (double)frame_val * frame_val;
            }
```

CUDA 720p with Tilling

```
__global__
void find_best_match_tiled_kernel(const unsigned char *frame_gray, int frame_w, int frame_h, const unsigned char *template_gray, int template_w, int

    __shared__ unsigned char s_template_tile[TILE_DIM * TILE_DIM];

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

    int tid = threadIdx.y * blockDim.x + threadIdx.x;
    int num_threads = blockDim.x * blockDim.y;

    double dot_product = 0.0;
    double patch_norm_sq = 0.0;

    bool active = (x <= frame_w - template_w && y <= frame_h - template_h);

    for (int ty_start = 0; ty_start < template_h; ty_start += TILE_DIM) {
        for (int tx_start = 0; tx_start < template_w; tx_start += TILE_DIM) {

            int tile_pixels = TILE_DIM * TILE_DIM;

            for (int i = tid; i < tile_pixels; i += num_threads) {
                int t_local_y = i / TILE_DIM;
                int t_local_x = i % TILE_DIM;

                int t_global_y = ty_start + t_local_y;
                int t_global_x = tx_start + t_local_x;

                if (t_global_y < template_h && t_global_x < template_w) {
                    s_template_tile[i] = template_gray[t_global_y * template_w + t_global_x];
                } else {
                    s_template_tile[i] = 0;
                }
            }
            __syncthreads();
```

Figure 4. Implementation of Tilling in CUDA.

III. Sequential vs Parallel Program Comparison

A. 240p

Sequential C Version

The C sequential version of this program calculates the map pattern in color_to_grayscale, as well as the cosine similarity scores to find the single highest number and its location in find_best_match. In color_to_grayscale, it iterates through

the image array index by index sequentially. As such, it is completely order dependent in terms of its execution time.

For the object tracking logic in find_best_match, the C program uses a sequential Sliding Window algorithm. It calculates the cosine similarity for one patch, and then immediately compares that score against the current maximum. This is what is known as an Integrated Linear Search, where the calculation of the score and the decision of finding the maximum are done in the same loop. Because of this, it creates a data dependency where the final result is not known until the very last iteration is complete.

Parallel CUDA Version

In the CUDA version of the program, it does things in parallel instead of sequentially. In the grayscale conversion function called color_to_grayscale_kernel, a unique Thread ID is assigned to every pixel coordinate (x, y). All threads execute the luminance conversion simultaneously. With this approach, the operation is independent, as the time complexity is the time required to convert a single pixel instead of the sum of the time to convert all pixels.

For template matching, the logic is split into two distinct kernels. There is the similarity map generation, where the find_best_match_kernel launches thousands of threads simultaneously, where each thread calculates for the cosine similarity score for a specific patch instead of using a sliding window. Each thread writes its result to a unique index in the similarity map.

To find the best match, the find_max_kernel has threads cooperating to compare values in pairs within shared memory (for example, Thread 0 compared index 0 with index 128) until the global maximum is found. This approach is much faster than doing N number of comparisons one after the other.

B. 480p

Sequential C Version

The sequential C implementation remains identical to the 240p version, relying largely on the Sliding Window algorithm and the Integrated Linear Search. However, the distinction lies in the computational load. With the images' resolutions and template sizes growing much larger than the 240p version, the number of operations also scale. Since the C code relies on only a single thread, it is also significantly slower. The execution time increases linearly with the number of pixels, making the nature of the sequential loops a very large bottleneck at this resolution.

Parallel CUDA Version

The CUDA implementation also uses the one thread per pixel approach used in 240. It explores two parallel strategies to handle the larger data points: a shared memory version and a global memory version.

The shared memory variant utilizes cooperative loading. In this design, threads within a block pause execution to collectively load the template into the GPU's on-chip shared memory cache. Theoretically, this is designed to reduce the pressure on memory bandwidth. However, this approach is shown to have overhead due to synchronization, as threads must wait at certain points, at __syncthreads(), to ensure that the data is loaded before any other computations can proceed.

The global memory variant (without shared memory) proved to be faster in this execution. It relies on implicit caching, wherein instead of manually managing the cache, every thread reads directly from global memory. While this theoretically increases bandwidth pressure,

C. 720p

Sequential C Version

The sequential C version follows the same ideas used in the 240p and 480p versions, but the amount of work done is much larger. The program still uses nested loops for grayscale conversion and the Sliding Window algorithm for template matching. In the grayscale function, it walks through the frame pixel by pixel then converting each one into grayscale in a fixed and ordered way.

Template matching would again use an Integrated Linear Search. At every valid position in the frame, it computes the cosine similarity for that patch, and right after that, it checks if the current score would be better than the saved maximum. The number of comparisons would also grow a lot because the frame is much bigger and the template is also larger. The CPU must do all cosine similarity calculations one by one, and it must finish the very last iteration before knowing the best match. This makes the sequential approach extremely slow at 720p, since it relies on just one thread doing millions of operations back to back.

Parallel CUDA Version

The CUDA version changes this by distributing the heavy work to thousands of threads. In the grayscale kernel, each pixel gets its own thread, so the luminance conversion happens at the same time instead of being done in order. This removes the bottleneck that could be seen in the C version.

The CUDA version for template matching uses many threads to compute similarity scores in parallel. A single thread is assigned to each patch in the frame, which effectively does away with the necessity of a sliding window loop. All these scores are kept in a shared similarity map. To get the maximum score, the program applies a reduction kernel, where threads compare values in shared memory in pairs, which then reduces the search time as the number of active threads diminishes.

At 720p, this is combined with optional shared memory tiling to handle larger templates. The tiled version allows threads to load small parts of the template into shared memory, work on them, and then move to the next tile. The CUDA version handles the large workload much faster, as the GPU performs the core operations in parallel while the CPU version has to do everything in a long chain or sequence.

IV. Results and Execution Time Comparison

The table below shows the comparison of the execution time of the C and CUDA programs for the different image resolutions. It can be seen that for all image sizes, the CUDA Version is significantly faster and outperforms the C version.

Resolution	Execution Time (ms)	CUDA Global Memory Kernel Execution Time (ms)	CUDA Shared Memory Kernel Execution Time (ms)
240p	37536.527	6.526	6.717
480p	494972.636	77.565	79.362
720p	2691901.432	401.239	414.591

A. C Results

240p	Starting 240p C object tracker... Frame 0: Best match found at (x=138, y=46) Frame 1: Best match found at (x=147, y=49) Frame 2: Best match found at (x=160, y=47) Frame 3: Best match found at (x=180, y=52) Frame 4: Best match found at (x=170, y=51) Frame 5: Best match found at (x=151, y=47) Frame 6: Best match found at (x=129, y=50) Frame 7: Best match found at (x=125, y=51) Kernel Execution Time: 37536.527000 milliseconds Number of Recorded Kernel Calls: 9

480p	Starting 480p C object tracker... Frame 0: Best match found at (x=289, y=90) Frame 1: Best match found at (x=300, y=106) Frame 2: Best match found at (x=329, y=105) Frame 3: Best match found at (x=372, y=114) Frame 4: Best match found at (x=351, y=112) Frame 5: Best match found at (x=312, y=103) Frame 6: Best match found at (x=268, y=107) Frame 7: Best match found at (x=261, y=110) Kernel Execution Time: 494972.636000 milliseconds Number of Recorded Kernel Calls: 9
720p	Starting 720p C object tracker... Frame 0: Best match found at (x=426, y=131) Frame 1: Best match found at (x=453, y=141) Frame 2: Best match found at (x=493, y=132) Frame 3: Best match found at (x=556, y=148) Frame 4: Best match found at (x=526, y=144) Frame 5: Best match found at (x=468, y=131) Frame 6: Best match found at (x=401, y=139) Frame 7: Best match found at (x=389, y=143) Kernel Execution Time: 2691901.432000 milliseconds Number of Recorded Kernel Calls: 9

(a)

(b)

(c)

Figure 5. Sample output of the C program for the first-frame results of (a) 240p, (b) 480p and (c) 720p resolutions.

B. CUDA Results (Global Memory)

240p

```
Starting CUDA 240p object tracker (NO SHARED TEMPLATE VERSION)
Template loaded successfully (87 x 118).
Frame 0: Best match found at (x=138, y=46) with similarity: 0.999748
Frame 1: Best match found at (x=147, y=49) with similarity: 0.976051
Frame 2: Best match found at (x=160, y=47) with similarity: 0.978532
Frame 3: Best match found at (x=180, y=52) with similarity: 0.945356
Frame 4: Best match found at (x=170, y=51) with similarity: 0.973051
Frame 5: Best match found at (x=151, y=47) with similarity: 0.976266
Frame 6: Best match found at (x=129, y=50) with similarity: 0.963636
Frame 7: Best match found at (x=125, y=51) with similarity: 0.971881
```

```
==380894== Profiling application: ./CUDA_240p_NO_SHARED_TEMP
==380894== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   90.75%  6.7168ms         8  839.60us  838.26us  840.50us  find_best_match_kernel
(unsigned char const *, int, int, unsigned char const *, int, int, double, double*)
                   7.40%  547.56us        17  32.209us  1.9200us  78.081us  [CUDA memcpy HtoD]
                   0.92%  68.291us        33  2.0690us  1.3440us  4.5440us  [CUDA memcpy DtoH]
                   0.62%  45.635us        16  2.8520us  2.5920us  3.2640us  find_max_kernel(double
const *, int, double*, int*)
                   0.31%  22.881us         9  2.5420us  2.4320us  3.1040us  color_to_grayscale_ker
nel(unsigned char const *, unsigned char*, int, int, int)
      API calls:   97.21%  853.35ms        58  14.713ms  5.6070us  850.97ms  cudaMalloc
                   1.42%  12.428ms        42  295.89us  17.152us  844.13us  cudaMemcpy
                   0.50%  4.3693ms         8  546.17us  351.07us  1.1632ms  cudaMemcpy2D
                   0.42%  3.6673ms        33  111.13us  9.8540us  1.6281ms  cudaLaunchKernel
                   0.41%  3.5864ms        58  61.835us  6.5330us  1.3082ms  cudaFree
                   0.03%  297.42us       114  2.6080us     104ns  118.79us  cuDeviceGetAttribute
                   0.01%  87.945us         1  87.945us  87.945us  87.945us  cuDeviceGetName
                   0.00%  15.317us        33     464ns     121ns  1.8820us  cudaPeekAtLastError
                   0.00%  15.163us         1  15.163us  15.163us  15.163us  cuDeviceTotalMem
                   0.00%  11.205us         1  11.205us  11.205us  11.205us  cuDeviceGetPCIBusId
                   0.00%  6.5570us         3  2.1850us     413ns  4.1120us  cuDeviceGetCount
                   0.00%  2.8720us         1  2.8720us  2.8720us  2.8720us  cuModuleGetLoadingMode
                   0.00%  1.6750us         2     837ns     234ns  1.4410us  cuDeviceGet
                   0.00%     208ns         1     208ns     208ns     208ns  cuDeviceGetUuid
```

480p

```
Starting CUDA 480p object tracker (NO SHARED TEMPLATE VERSION)
Template loaded successfully (150 x 232).
Frame 0: Best match found at (x=289, y=90) with similarity: 0.999850
Frame 1: Best match found at (x=300, y=106) with similarity: 0.973546
Frame 2: Best match found at (x=329, y=105) with similarity: 0.972979
Frame 3: Best match found at (x=372, y=114) with similarity: 0.948270
Frame 4: Best match found at (x=351, y=112) with similarity: 0.975688
Frame 5: Best match found at (x=312, y=103) with similarity: 0.976127
Frame 6: Best match found at (x=268, y=107) with similarity: 0.963581
Frame 7: Best match found at (x=261, y=110) with similarity: 0.970502
```

```
==381056== Profiling application: ./CUDA_480p
==381056== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   94.99%  73.919ms         8  9.2399ms  9.2268ms  9.2618ms  find_best_match_kernel
(unsigned char const *, int, int, unsigned char const *, int, int, double, double*)
                   4.68%  3.6457ms        17  214.45us  5.4400us  648.20us  [CUDA memcpy HtoD]
                   0.16%  124.10us        25  4.9630us  1.3760us  13.217us  [CUDA memcpy DtoH]
                   0.11%  88.836us        24  3.7010us  2.6560us  5.6320us  find_max_kernel(double
const *, int const *, int, double*, int*, bool)
                   0.05%  40.322us         9  4.4800us  2.8480us  4.9600us  color_to_grayscale_ker
nel(unsigned char const *, unsigned char*, int, int, int)
      API calls:   89.39%  976.02ms        74  13.189ms  5.8530us  962.81ms  cudaMalloc
                   7.81%  85.303ms        74  1.1527ms  6.4060us  9.2648ms  cudaFree
                   1.59%  17.341ms        34  510.03us  19.486us  2.0559ms  cudaMemcpy
                   0.61%  6.6829ms         8  835.36us  452.28us  1.0095ms  cudaMemcpy2D
                   0.47%  5.1494ms        41  125.60us  10.169us  2.6812ms  cudaLaunchKernel
                   0.09%  928.85us       114  8.1470us     131ns  469.63us  cuDeviceGetAttribute
                   0.02%  220.11us         1  220.11us  220.11us  220.11us  cuDeviceGetName
                   0.01%  66.254us         1  66.254us  66.254us  66.254us  cuDeviceTotalMem
                   0.01%  60.772us        41  1.4820us     128ns  36.738us  cudaPeekAtLastError
                   0.00%  28.540us         1  28.540us  28.540us  28.540us  cuDeviceGetPCIBusId
                   0.00%  11.299us         3  3.7660us     350ns  10.548us  cuDeviceGetCount
                   0.00%  7.3560us         2  3.6780us  1.1860us  6.1700us  cuDeviceGet
                   0.00%  1.6760us         1  1.6760us  1.6760us  1.6760us  cuDeviceGetUuid
                   0.00%     937ns         1     937ns     937ns     937ns  cuModuleGetLoadingMode
```

720p

```
Starting CUDA 720p object tracker (NO TILLING VERSION)
Template loaded successfully (251 x 351).
Frame 0: Best match found at (x=426, y=131) with similarity: 0.999854
Frame 1: Best match found at (x=453, y=141) with similarity: 0.976094
Frame 2: Best match found at (x=493, y=132) with similarity: 0.977169
Frame 3: Best match found at (x=556, y=148) with similarity: 0.944861
Frame 4: Best match found at (x=526, y=144) with similarity: 0.974873
Frame 5: Best match found at (x=468, y=131) with similarity: 0.974679
Frame 6: Best match found at (x=401, y=139) with similarity: 0.964025
Frame 7: Best match found at (x=389, y=143) with similarity: 0.970891
==381136== Profiling application: ./CUDA_720p
==381136== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   95.94%  385.34ms         8  48.168ms  46.533ms  48.714ms  find_best_match_kernel
(unsigned char const *, int, int, unsigned char const *, int, int, double, double*)
                   3.96%  15.899ms        17  935.25us  16.609us  2.4455ms  [CUDA memcpy HtoD]
                   0.06%  236.65us        25  9.4650us  1.5040us  25.793us  [CUDA memcpy DtoH]
                   0.03%  113.76us        24  4.7400us  2.8160us  8.8960us  find_max_kernel(double
const *, int const *, int, double*, int*, bool)
                   0.02%  72.545us         9  8.0600us  3.4880us  8.9930us  color_to_grayscale_ker
nel(unsigned char const *, unsigned char*, int, int, int)
      API calls:   55.96%  750.94ms        74  10.148ms  4.9260us  726.57ms  cudaMalloc
                   40.89%  548.70ms        74  7.4149ms  6.3680us  147.62ms  cudaFree
                   2.27%  30.496ms        34  896.95us  21.698us  4.0134ms  cudaMemcpy
                   0.48%  6.3811ms         8  797.63us  574.57us  1.2376ms  cudaMemcpy2D
                   0.34%  4.5273ms        41  110.42us  9.1920us  1.7712ms  cudaLaunchKernel
                   0.05%  645.80us       114  5.6640us     128ns  241.06us  cuDeviceGetAttribute
                   0.02%  213.28us         1  213.28us  213.28us  213.28us  cuDeviceGetName
                   0.00%  36.174us        41     882ns     123ns  20.795us  cudaPeekAtLastError
                   0.00%  29.241us         1  29.241us  29.241us  29.241us  cuDeviceTotalMem
                   0.00%  23.329us         1  23.329us  23.329us  23.329us  cuDeviceGetPCIBusId
                   0.00%  8.6460us         3  2.8820us     195ns  7.9910us  cuDeviceGetCount
                   0.00%  4.7630us         2  2.3810us     188ns  4.5750us  cuDeviceGet
                   0.00%  1.4090us         1  1.4090us  1.4090us  1.4090us  cuModuleGetLoadingMode
                   0.00%     459ns         1     459ns     459ns     459ns  cuDeviceGetUuid
```

(a)

(b)

(c)

Figure 6. Sample output of the CUDA-Global Memory program for the first-frame results of (a) 240p, (b) 480p and (c) 720p resolutions.

C. CUDA Results (Shared Memory)

240p

```
Starting CUDA 240p object tracker (WITH SHARED TEMPLATE VERSION)
Template loaded successfully (87 x 118).
Frame 0: Best match found at (x=138, y=46) with similarity: 0.999748
Frame 1: Best match found at (x=147, y=49) with similarity: 0.976051
Frame 2: Best match found at (x=160, y=47) with similarity: 0.978532
Frame 3: Best match found at (x=180, y=52) with similarity: 0.945356
Frame 4: Best match found at (x=170, y=51) with similarity: 0.973051
Frame 5: Best match found at (x=151, y=47) with similarity: 0.976266
Frame 6: Best match found at (x=129, y=50) with similarity: 0.963636
Frame 7: Best match found at (x=125, y=51) with similarity: 0.971881
```

```
==380834== Profiling application: ./CUDA_240p
==380834== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   90.74%  6.5255ms         8  815.69us  815.28us  816.15us  find_best_match_kernel
(unsigned char const *, int, int, unsigned char const *, int, int, double, double*)
                   7.35%  528.23us        17  31.072us  1.9200us  79.554us  [CUDA memcpy HtoD]
                   0.95%  68.577us        33  2.0780us  1.3440us  4.0960us  [CUDA memcpy DtoH]
                   0.63%  45.601us        16  2.8500us  2.5920us  3.2330us  find_max_kernel(double
const *, int, double*, int*)
                   0.32%  23.265us         9  2.5850us  2.4320us  3.0400us  color_to_grayscale_ker
nel(unsigned char const *, unsigned char*, int, int, int)
      API calls:   97.77%  978.68ms        58  16.874ms  5.4040us  976.17ms  cudaMalloc
                   1.15%  11.470ms        42  273.11us  16.535us  833.80us  cudaMemcpy
                   0.39%  3.9304ms        33  119.10us  9.1750us  1.7863ms  cudaLaunchKernel
                   0.36%  3.6141ms         8  451.77us  318.12us  651.49us  cudaMemcpy2D
                   0.25%  2.4747ms        58  42.668us  6.3720us  618.82us  cudaFree
                   0.05%  513.39us       114  4.5030us     166ns  202.94us  cuDeviceGetAttribute
                   0.01%  142.50us         1  142.50us  142.50us  142.50us  cuDeviceGetName
                   0.01%  72.340us        33  2.1920us     152ns  57.176us  cudaPeekAtLastError
                   0.00%  21.834us         1  21.834us  21.834us  21.834us  cuDeviceTotalMem
                   0.00%  15.618us         1  15.618us  15.618us  15.618us  cuDeviceGetPCIBusId
                   0.00%  8.3540us         2  4.1770us     311ns  8.0430us  cuDeviceGet
                   0.00%  7.3660us         3  2.4550us     218ns  4.9810us  cuDeviceGetCount
                   0.00%  1.5160us         1  1.5160us  1.5160us  1.5160us  cuModuleGetLoadingMode
                   0.00%     622ns         1     622ns     622ns     622ns  cuDeviceGetUuid
```

480p

```
Starting CUDA 480p object tracker (SHARED TEMPLATE VERSION)
Template loaded successfully (150 x 232).
Frame 0: Best match found at (x=289, y=90) with similarity: 0.999850
Frame 1: Best match found at (x=300, y=106) with similarity: 0.973546
Frame 2: Best match found at (x=329, y=105) with similarity: 0.972979
Frame 3: Best match found at (x=372, y=114) with similarity: 0.948270
Frame 4: Best match found at (x=351, y=112) with similarity: 0.975688
Frame 5: Best match found at (x=312, y=103) with similarity: 0.976127
Frame 6: Best match found at (x=268, y=107) with similarity: 0.963581
Frame 7: Best match found at (x=261, y=110) with similarity: 0.970502
```

```
==380970== Profiling application: ./CUDA_480p_with_SHARED
==380970== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   96.83%  77.072ms         8  9.6341ms  9.6301ms  9.6361ms  find_best_match_kernel
(unsigned char const *, int, int, unsigned char const *, int, int, double, double*)
                   2.88%  2.2897ms        17  134.69us  5.2160us  353.96us  [CUDA memcpy HtoD]
                   0.13%  103.94us        25  4.1570us  1.3120us  11.808us  [CUDA memcpy DtoH]
                   0.11%  88.675us        24  3.6940us  2.6560us  5.6960us  find_max_kernel(double
const *, int const *, int, double*, int*, bool)
                   0.05%  40.833us         9  4.5370us  3.1360us  4.9600us  color_to_grayscale_ker
nel(unsigned char const *, unsigned char*, int, int, int)
      API calls:   89.62%  909.44ms        74  12.290ms  4.2150us  899.74ms  cudaMalloc
                   8.31%  84.284ms        74  1.1390ms  4.9140us  9.6199ms  cudaFree
                   1.32%  13.382ms        34  393.58us  17.153us  1.6590ms  cudaMemcpy
                   0.38%  3.8262ms         8  478.28us  344.28us  849.52us  cudaMemcpy2D
                   0.32%  3.2033ms        41  78.129us  8.2500us  1.3460ms  cudaLaunchKernel
                   0.05%  491.17us       114  4.3080us     175ns  220.86us  cuDeviceGetAttribute
                   0.01%  135.20us         1  135.20us  135.20us  135.20us  cuDeviceGetName
                   0.00%  25.071us         1  25.071us  25.071us  25.071us  cuDeviceTotalMem
                   0.00%  18.585us        41     453ns      92ns  2.7820us  cudaPeekAtLastError
                   0.00%  8.9580us         1  8.9580us  8.9580us  8.9580us  cuDeviceGetPCIBusId
                   0.00%  5.8370us         2  2.9180us     223ns  5.6140us  cuDeviceGet
                   0.00%  4.3990us         3  1.4660us     190ns  3.8630us  cuDeviceGetCount
                   0.00%  1.3510us         1  1.3510us  1.3510us  1.3510us  cuModuleGetLoadingMode
                   0.00%     339ns         1     339ns     339ns     339ns  cuDeviceGetUuid
```

720p

```
Starting CUDA object tracker (TILED SHARED MEMORY)...
Frame 0: Best match (x=426, y=131), Sim: 0.999854
Frame 1: Best match (x=453, y=141), Sim: 0.976094
Frame 2: Best match (x=493, y=132), Sim: 0.977169
Frame 3: Best match (x=556, y=148), Sim: 0.944861
Frame 4: Best match (x=526, y=144), Sim: 0.974873
Frame 5: Best match (x=468, y=131), Sim: 0.974679
Frame 6: Best match (x=401, y=139), Sim: 0.964025
Frame 7: Best match (x=389, y=143), Sim: 0.970891
Total GPU processing time: 407.675000 ms
```

```
==381262== Profiling application: ./CUDA_720p_tiled
==381262== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   96.24%  399.43ms         8  49.929ms  47.910ms  56.750ms  find_best_match_tiled_
kernel(unsigned char const *, int, int, unsigned char const *, int, int, double, double*)
                   3.65%  15.161ms        17  891.83us  15.040us  2.4343ms  [CUDA memcpy HtoD]
                   0.06%  236.10us        25  9.4430us  1.5050us  27.360us  [CUDA memcpy DtoH]
                   0.03%  143.27us        24  5.9690us  3.5200us  13.185us  find_max_kernel(double
const *, int const *, int, double*, int*, bool)
                   0.02%  78.720us         9  8.7460us  3.4560us  11.648us  color_to_grayscale_ker
nel(unsigned char const *, unsigned char*, int, int, int)
      API calls:   70.35%  1.07197s        74  14.486ms  5.2500us  1.05742s  cudaMalloc
                   27.11%  413.09ms        74  5.5822ms  5.7770us  56.778ms  cudaFree
                    1.82%  27.762ms        34  816.52us  21.398us  3.7236ms  cudaMemcpy
                    0.39%  6.0124ms         8  751.55us  451.27us  1.1110ms  cudaMemcpy2D
                    0.27%  4.1116ms        41  100.28us  7.5970us  1.9296ms  cudaLaunchKernel
                    0.04%  632.88us       114  5.5510us     127ns  209.75us  cuDeviceGetAttribute
                    0.01%  167.57us         1  167.57us  167.57us  167.57us  cuDeviceGetName
                    0.00%  33.948us        16  2.1210us     141ns  17.346us  cudaPeekAtLastError
                    0.00%  26.236us         1  26.236us  26.236us  26.236us  cuDeviceTotalMem
                    0.00%  20.607us         1  20.607us  20.607us  20.607us  cuDeviceGetPCIBusId
                    0.00%  8.1690us         3  2.7230us     253ns  7.5300us  cuDeviceGetCount
                    0.00%  8.0670us         2  4.0330us     271ns  7.7960us  cuDeviceGet
                    0.00%  1.7070us         1  1.7070us  1.7070us  1.7070us  cuModuleGetLoadingMode
                    0.00%  1.0830us         1  1.0830us  1.0830us  1.0830us  cuDeviceGetUuid
```

(a)

(b)

(c)

Figure 7. Sample output of the CUDA-Shared Memory program for the first-frame results of (a) 240p, (b) 480p and (c) 720p resolutions.

V. Discussion

A. Parallelism Speedup

The most obvious result is the large speed-up from the C program to the CUDA program. As the C program iterates through the image sequentially through nested loops, the time complexity grows by much more as image size also grows. With the CUDA program, the GPU processes thousands of pixels simultaneously. This explains why the gap widens as the resolution increases. The GPU has enough resources to perform extra work, while the CPU can only use one thread, hence why the C program is so slow.

	Speed Up Compared to C		
	240p	**480p**	**720p**
CUDA - Global Memory	5587.88x	6381.39x	6708.97x
CUDA - Shared Memory	5751.84x	6236.90x	6492.91x

B. Shared Memory vs Global Memory

The effect of using shared memory in the kernel average runtime are inconsistent for the different image resolution. In the 240p program, the shared memory was faster (6.717 ms vs 6.526 ms), while in the 480p and 720p versions, the shared memory (tiled version in the 720p variant) was actually slightly slower than the global memory version.

In the 480p implementation, the shared memory version is slower because the code attempts to load the entire template into shared memory. The find_best_match_kernel declares a dynamic shared memory array to hold the whole template:

```
extern _shared_ unsigned char shared_template[];

int template_size = template_w * template_h;
for (int i = tid; i < template_size; i +=
block_threads) {
    shared_template[i] = template_gray[i];
}
```

The bottleneck comes from the size of the template. Most CUDA architectures have around 48KB or 64KB of shared memory per streaming multiprocessor. With this resolution size, the template is estimated to be around 34KB if each pixel is 1 byte, much larger than the 10KB size of the 240p template. The NVIDIA CUDA Best Practices Guide states that thread instructions are executed sequentially, and executing other warps (a group of 32 threads) when one is blocked is the only way to keep the hardware occupied. Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of active warps. With this, if shared memory usage on a block is relatively high, the amount of blocks that can be on a multiprocessor is reduced.

In the 480p version, the shared memory variant allocating 34KB of shared memory becomes a problem as the streaming multiprocessor can only hold 1 block at a time. To make the GPU fast, it needs to run multiple blocks of threads on a single core at the same time. This allows the GPU to switch to a new block if the current one is waiting for memory (also known as Latency Hiding). If the GPU streaming multiprocessor has a limit of 64KB of shared memory, the GPU can only fit 1 block per streaming multiprocessor since 64KB/34KB = 1.8. As we cannot have a fraction of a block, it gets rounded down to 1 block. The program slows down due to this single block, as when that block waits for data, the hardware sits idle.

In contrast, the global memory version uses virtually no shared memory. This allows the streaming multiprocessor to hold many active blocks simultaneously. Even though accessing global memory is slower per instruction, the GPU does not actively show this delay as it switches to other blocks while waiting. The shared memory loses this parallelism, resulting in slower overall performance.

In the 240p version, the shared memory usage per block is determined by its template size, 87 x 118 bytes, which is around 10KB per block. Since the shared memory is assumed to have 64KB per streaming multiprocessor, it can accommodate approximately 6 blocks (64KB / 10KB 6.4). As this is more than the blocks that the 480p version can fit, the streaming multiprocessor maintains high occupancy. When the active warp stalls on a global memory read, the scheduler has a pool of other warps for the remaining 5 blocks to execute, and performs context switching. This is most likely why the shared memory version is faster for only this resolution variant, as well as the fact that shared memory is physically closer to the processor cores and offers higher bandwidth and lower latency normally.

The shared memory (or tiled) version for 720p was also slower than the global memory approach for the CUDA program. This is due to the heavy instruction overhead required to manage the tiles and synchronization barriers, which then outweighs the benefits of caching. The tiled approach here was used, as the template for 720p is not able to fit in the 64KB shared memory (88KB template size), so in order to see if the program would benefit from shared memory, this was the approach used.

The function find_best_match_kernel introduces nested loops to manage tiles of size 32.

```
for (int ty_start = 0; ty_start < template_h; ty_start
+= TILE_DIM) {
      for (int tx_start = 0; tx_start < template_w;
tx_start += TILE_DIM) {
        ...
        __syncthreads();
        ...
         int frame_idx = (y + ty_start + ty) * frame_w
+ (x + tx_start + tx);
        ...
        __syncthreads();
    }
}
```

The tiled algorithm requires breaking the large template into small 32×32 chunks. To ensure that all threads have finished loading a tile before reading it, the code must use a “barrier”, which is _syncthreads(). This function waits until all threads in the block have reached the synchronization point. Since this function was placed inside a double loop, this forces the hardware to stop and align every single thread, thousands of times per frame.

According to Appendix B.24 of the NVIDIA CUDA C++ Programming Guide, __syncthreads() acts as a barrier that "waits until all threads in the thread block have reached this point." Due to this, the hardware cannot hide latency very well because it is constantly forced to idle and wait for the barrier to resolve inside every tile iteration.

The NVIDIA CUDA C++ Best Practices Guide in the section on memory optimizations says that the complexity of managing shared memory can outweigh

the benefits due to the "instruction overhead associated with address arithmetic." In the tiled code, the program must calculate complex global indices (y + ty_start + ty) * frame_w + (x + tx_start + tx) for every pixel operation.

The global memory version, on the other hand, is faster because it avoids these overheads entirely. It relies on the GPU's built-in L2 Cache to handle data reuse automatically. Since the global memory version accesses data in a linear pattern, the hardware cache is efficient enough to provide high performance without the extra overhead computation cost.

VI. Conclusion

This project was able to shorten the execution time of an object-tracking program written in C by implementing it in parallel with CUDA. Across all image resolutions, the CUDA implementation outperformed the C version with speed-up of 5751.84x for 240p, 6381.39x for 480p, and 6708.97x for 720p. CUDA made the improvement possible by sending out multiple threads to execute independent tasks such as grayscale conversion, sliding-window patch cosine similarity evaluation, and parallel reduction for maximum-similarity detection.

Meanwhile, using further parallelization techniques such as shared memory and tilling can have variable effects. Their impact depends heavily on factors such as template size, shared memory size availability, synchronization overhead, and the resulting occupancy of the GPU. As such, careful consideration and evaluation must be done before these optimizations should be incorporated in the CUDA implemented. All in all, the results of our project shows the considerable potential of using CUDA and GPU processors for accelerating computationally intensive workloads in the field of computer vision.

VII. References

AlHumaidan, B., Alghofaily, S., Al Qahtani, M., Oudah, S., & Nagy, N. (2024). Parallel image processing: Taking grayscale conversion using OpenMP as an example. *Journal of Computer and Communications*, 12, 1-10. https://doi.org/10.4236/jcc.2024.122001

NVIDIA Corporation. (n.d.). *CUDA C++ best practices guide* (Version 13.0). Retrieved from https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/

NVIDIA Corporation. (n.d.). *CUDA C++ programming guide* (Version 13.0). Retrieved from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Sahilkamdar. (2025, November 6). *Color image to greyscale image using data parallelism.* Medium. https://medium.com/@sahilkamdar24/color-image-to-greyscale-image-using-data-parallelism-ada45c1c8ada