Parallel Video Object Tracker: Final Project Report

Eric Li, Yanxin Jiang

December 2024

1 SUMMARY

We implemented a high-performance video object tracker in `C++` and `CUDA` using Normalized Cross-Correlation. Our system achieves approximately 22.5 FPS on input video through shared and constant memory optimizations, demonstrating a 2.5× speedup over our naive CUDA baseline (~9.0 FPS). The tracker features a robust lost-frame recovery algorithm that intelligently switches between search strategies based on tracking confidence. Additionally, we implemented an interactive mode that enables users to select tracking templates from input videos and visualize the tracking process in real-time performance metrics.

2 BACKGROUND

2.1 Algorithm Overview

Video object tracking is a computer vision technique designed to identify and follow a specific object across consecutive frames in a video sequence. The process typically begins by defining a reference image of the tracking object, extracts its representation, and then searches for the best matching region in each subsequent frame.

Our implementation utilizes Normalized Cross-Correlation (NCC) to match a template against candidate regions. As shown in previous work [1], NCC is a popular measure of similarity between two images or image blocks. In our project, it measures the similarity between a template image T and a candidate patch F_{patch} within the frame. The zero-mean normalized cross-correlation formula is defined as:

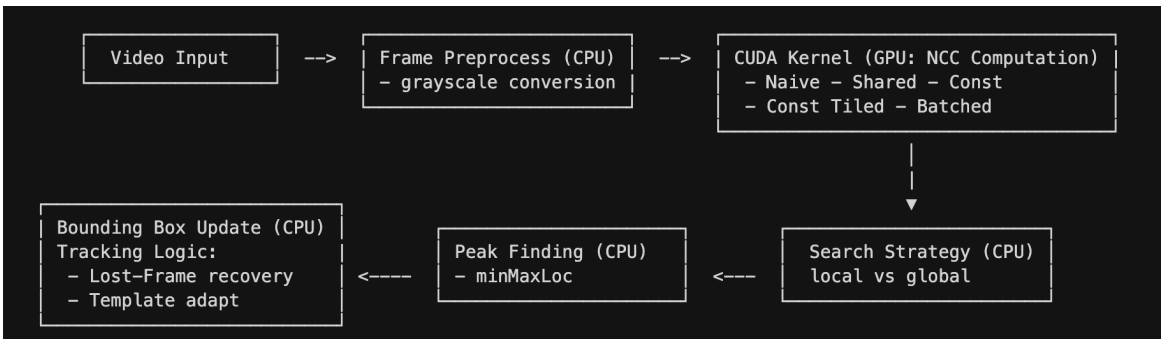
$$NCC(T, F_{patch}) = \frac{\sum[(T_i - \bar{T}) \times (F_i - \bar{F})]}{N \times \sigma_T \times \sigma_F}$$

where $\bar{T}$ and $\bar{F}$ represent the mean values of the template and frame patch respectively and σ_T and σ_F represent their standard deviations. N is the total number of template pixels (template width $\times$ height). Our output is an NCC 2D map in which each index has the output range of $[-1.0, 1.0]$, where 1.0 indicates a perfect match. We used NCC for our project as it is illumination invariant and contract invariant, which effectively handle brightness change and allow it to work with varying contrast level.

2.2 System Implementation and Workflow

The system requires four key data structures: Frame Image, Template, NCC Map, and Bounding Box. Frame Image is a grayscale float matrix `CV_32FC1` with values normalized between 0.0 and 1.0 stored in row-major order. Template is a stored float image of the tracked object, extracted from a user-selected Region of Interest(RIO) in the first/selected frame and adaptively updated with a learning rate of 0.1 during tracking. NCC Map is the system output, which stores similarity scores for each possible template position. Its dimensions corresponds to the search area (($frameW - templW + 1) \times (frameH - templH + 1)$). Bounding box maintains the current tracking position of the object $(x, y, width, height)$. It is updated based on the best NCC match or maintained during low-confidence intervals.

The tracking algorithm is implemented frame-by-frame. First, the input frame is converted to a grayscale float format. Next, the NCC map is computed in parallel on the GPU, where similarity scores are calculated and stored for every possible template position on NCC map. After the computation, the system then determine which search strategy to use. Normally we choose a local search, which is restricted to a 60-pixel radius around the previous position for performance speedup. If the object is lost (may exit the frame), we use Global search instead to check the entire frame. We then need to find the maximum NCC value within the selected region to identify the position of the candidate object. If the confidence exceeds the threshold, the bounding box is updated and the template is adapted by blending a new high-confidence patch (10% new, 90% old). Conversely, if confidence is low, the previous bounding box is retained, and a lost-frame counter is incremented. If the lost frame counter exceeds 50, the lost frame triggers a switch to global search.



2.3 Complexity and Parallelism Analysis

The computation of template matching is intensive with a time complexity of

$$\underbrace{(W \times H)}_{\text{Number of Positions}} \times \underbrace{(w \times h)}_{\text{Work per Position}} = O(W \times H \times w \times h)$$

for a full-frame search. For a 640×480 frame and 50×50 template, there are $254,721$ independent correlation computations required for each frame. Local search reduces this cost to $O(R^2 \times w \times h)$, where R is the search radius. The space complexity for the NCC map is proportional to the number of valid search positions, which is $O((W - w + 1) \times (H - h + 1))$.

As the correlation computation for each pixel is independent of the others, the workload is pretty suitable for parallel execution, allowing thousands of positions to be computed simultaneously. The computation also demonstrates the great spatial and temporal locality as adjacent threads access overlapping regions of the frame data and the template is reused across all threads for a given frame. The high arithmetic intensity makes the algorithm amenable to SIMD execution on GPU warps.

However, the overall speedup and performance is still limited by some bottlenecks. The primary one is the memory bandwidth limitation due to redundant reads of overlapping frame windows, global memory access overheads in basic implementation, and the latency of transferring great frame data (~1.2 MB per frame) between the host and device.

3 APPROACH

3.1 Technologies Used

- **Language**: C++, CUDA
- **APIs**: CUDA Runtime API, OpenCV
- **Build System**: CMake

- **Target Machines**: GHC cluster machines with NVIDIA GPUs
- **Compiler**: NVCC (CUDA 11.7) with GCC 11

3.2 System Architecture

The system architecture is composed of three primary components: the Video I/O, the NCC Computation, and the Tracking Logic. The Video I/O handles frame loading and visualization using OpenCV. The core processing is the NCC Computation Layer, where we can perform the template matching operations on the GPU through parallel CUDA kernels. Finally, the Tracking Logic is implemented on the CPU; it determines the search window strategy, executes the peak finding algorithm via `minMaxLoc`, updates the bounding box, and handles the lost-frame recovery logic. This hybrid design assigns the intensive data-parallel computations to the GPU, while the sequential logic remains on the host.

3.3 Mapping to Parallel Hardware

3.3.1 Thread-to-Computation Mapping

To map the problem to parallel hardware, each CUDA thread computes the NCC score for one output pixel position. It maps the 2D output space directly to a 2D grid of thread blocks. For the naive and shared memory kernel, we used a block size of 16×16 (256 threads), while the constant memory kernels used 32×8 blocks to better align with warp execution and constant cache access patterns.

```
Grid Layout:
- outW = frameH-templH+1); outH = frameH-templH+1
- Block size: 256 threads per block
- Grid size: ceil(outW/block.x) x ceil(outH/block.y) blocks
- Total threads: outW x outH (one per NCC output pixel)
Thread (ox, oy) computes:
- NCC score at template position (ox, oy), stored at ncc_map[oy * outW + ox]
```

3.3.2 Memory Hierarchy Mapping

We utilized the GPU's memory hierarchy by placing frequently used template data in Constant or Shared Memory, while keeping the larger frame data in Global Memory and accessing it with coalesced reads whenever possible.

Global Memory:

- Frame data `dev_frame[frameH * frameW]`: Read by all threads
- Template data `dev_templ[templH * templW]`: Read by all threads (baseline GPU kernel)
- Output NCC map `dev_out[outH * outW]`: Written by each thread

Shared Memory (optimized kernels):

- Shared Template `shared_templ[templW * templH]`: Loaded once per block (shared kernel)
- Frame tile `shared_frame[tileW * tileH]`: Loaded by block threads (tiled kernel)

Constant Memory (optimized kernels):

- Template: `templ_const[MAX_TEMPL_PIXELS]` : Cached, Read-only, and Broadcast to all threads
- Size limit: 4096 pixels (16KB)

3.4 CUDA Kernel Implementations And Optimization Iterations

Our development process involves several optimizations. We began with a `Naive Kernel` which simply mapped threads to pixels. Despite the correctness, it had a memory bottleneck due to redundant global memory reads.

We then attempted a `Shared Memory` approach, caching the template within the block. We got some performance as the access to the template now is within the block, which improve the memory access latency. However, the performance did not improve greatly as it was limited by shared memory capacity. Frame data is still read from global memory and template size now is limited by the shared memory size.

We also tried `Batch` approach in which multiple frames were processed together as a single batch. However, this strategy did not perform well due to increased memory overhead and reduced parallel efficiency. Template updates during batch processing caused inconsistency as NCC maps could compute with old template, but template may update in the middle of batch.

Then, we designed `Const` kernel which copy the template data into the constant memory. It works well as no need to access global memory for template but has the similar issue to `shared` kernel. To optimize it, we merge the methods of `shared` and `Const` kernels, which is our final optimal solution `Tiled Constant Memory` kernel. This method stores the template in constant memory and cooperatively loads frame tiles into shared memory, maximizing bandwidth usage.

In addition to the optimization in GPU implementation, We also modified the original serial algorithm

by introducing a "Lost-Frame" state machine, which switches between a local search for speed and a global search for recovery based on tracking confidence.

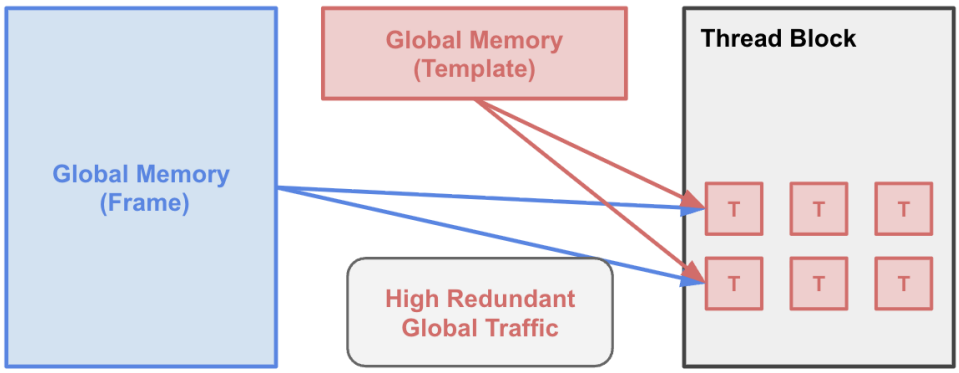
Per-thread computation for all kernels:

1. Compute frame patch mean and std in the first pass over template-sized region)
2. Compute covariance with template in the second pass)
3. Normalize to get NCC value and store it into NCC map

```
for (dy = 0; dy < templH; dy++) :
    idx = (oy + dy) * frameW + ox
    for (dx = 0; dx < templW; dx++) :
        calculate sum and sum of square via frame[idx + dx]
calculate frame mean and std via sum and ssq
for (dy = 0; dy < templH; dy++) :
    for (dx = 0; dx < templW; dx++) :
        cov += (frame_val - frame_mean) * (templ_val - templMean);
ncc = cov / (std * templStd * N);
```

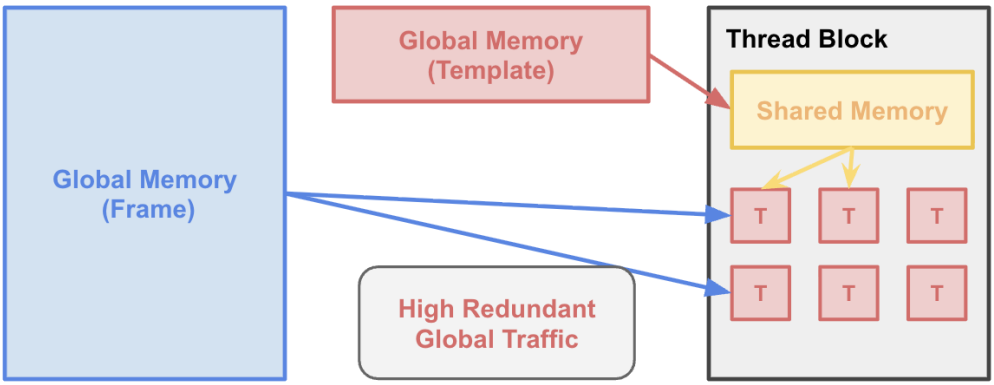
3.4.1 Naive CUDA Kernel (`nccKernelNaive`)

Naive Kernel: This is our GPU baseline implementation. Each thread reads the template directly from global memory for every pixel computation. While it is simple and straightforward, it suffers by severe memory bandwidth limitations without caching or coalesced memory access patterns. For every read, template is read from global memory, which greatly increase the cost of memory access.



3.4.2 Shared Memory Kernel (`nccKernelShared`)

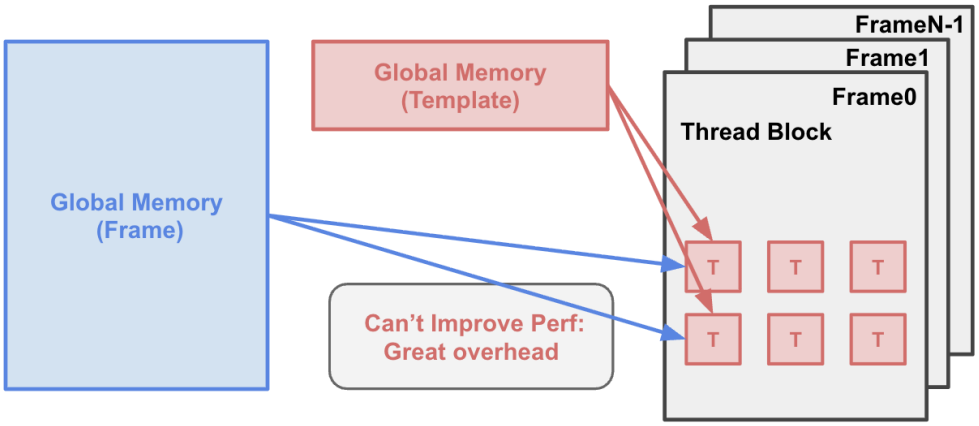
To relief bandwidth pressure, this kernel caches the template in shared memory. Threads within a block cooperatively load the template once, reducing global memory traffic. Each thread loads `templSize / numThreads` elements. After loading the template, we use `\_\_syncthreads()`, to ensure all threads see loaded template. In this case, the template is read from shared memory instead of global, which should improve speedup. While it reduces some global memory traffic, it still needs to read frame from global memory. Additionally, shared memory size is limited for each block so that large template may not fit in shared memory.



3.4.3 Batched Kernel (`nccKernelNaiveBatched`)

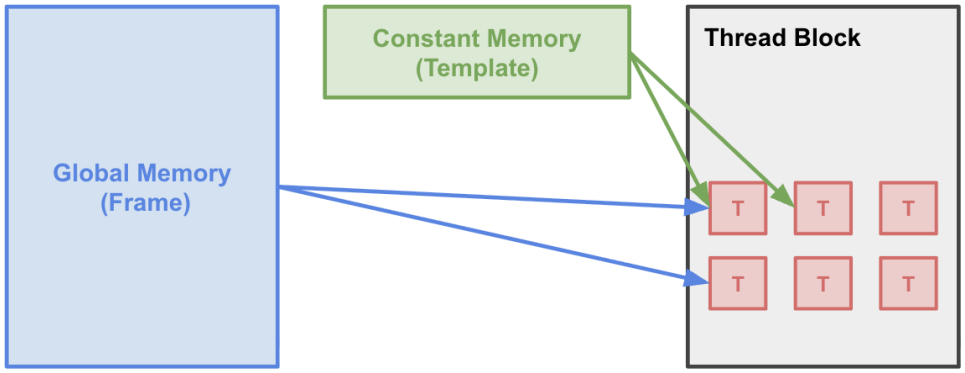
In batched kernel, it processes multiple frames concurrently using a 3D grid decomposition. The grid z-dimension `blockIdx.z` maps to the frame index, enabling the kernel to process `numFrames` simultaneously. Input frames are stored sequentially in global memory, and each thread computes a pointer to its assigned frame using the appropriate batch-index offset.

The batching strategy maximizes instruction throughput by keeping the GPU heavily loaded and amortizes the kernel-launch overhead and host-device transfer latency across the entire batch. However, batching introduces high input latency as the system must buffer a full batch before processing. It also increases global memory usage since multiple full-resolution frame should store on the device simultaneously. Additionally, it restricts template adaptation within a batch, as the template need to remain fixed across all frames to maintain consistency.



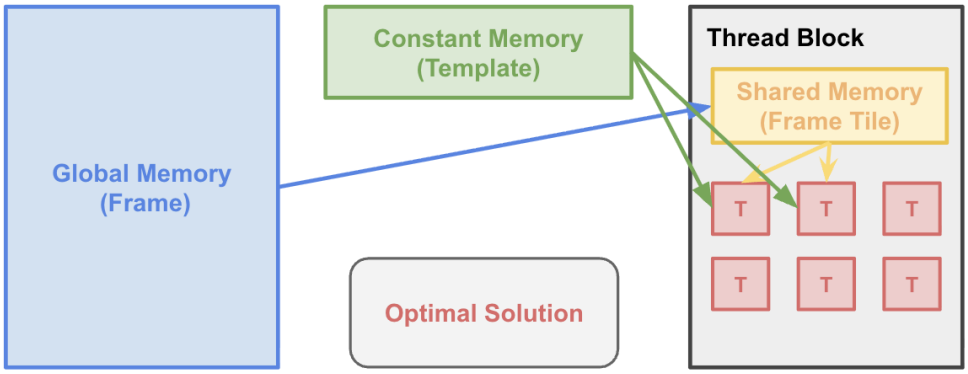
3.4.4 Constant Memory Kernel (`nccKernelConst`)

In this method, template is stored in `__constant__` memory through `cudaMemcpyToSymbol`. This allows the template to be cached and efficiently broadcast to all threads in a warp. The constant cache is faster and cached across warps. It eliminates the shared memory usage for the template, freeing it up for other data. However, the constant memory size is limit so we can only select template with the size limit of 4096 pixels (16KB). Furthermore, frame data is still in global memory.



3.4.5 Tiled Constant Memory Kernel (`nccKernelConstTiled`)

This is our highest-performance method to solve the memory bottlenecks. It combines the advantages of the previous approaches by storing the template in constant memory and caching tiles of the frame image in the shared memory. Threads load the frame tile required by the block, ensuring that all memory access for the computation comes from the fast on-chip shared memory. In tiled constant memory kernel, the system first load frame tile into shared memory; Then use `___syncthreads` to ensure that all threads finish loading the frame tile within the block; Compute NCC using shared frame tile and constant template and store output into the NCC map.



Memory Optimization: Cache both template and frame tile

- Template: constant memory which is read-only and cached
- Frame tile: shared memory which is loaded cooperatively within thread blocks
- Tile size: `(blockDim.x + templW - 1)` × `(blockDim.y + templH - 1)`

```
extern __shared__ float shared_frame[];
for (i = tid; i < tileSize; i += numThreads) :
    ty = i / tileW; tx = i % tileW;
    fy = tileY + ty; fx = tileX + tx;
    if (fy >= 0 && fy < frameH && fx >= 0 && fx < frameW):  Copy frame to shared_frame[i]
    else: shared_frame[i] = 0.0f;
__syncthreads();
```

Within the constant memory limit, this algorithm has the best performance for high arithmetic intensity workload and solve the memory bottlenecks.

3.5 Lost-Frame Recovery Algorithm

To handle object occlusions and exist/re-entries, in CPU tracking logic, we add a lost-frame recovery algorithm. By default, the system does the Local Search, searching only a 60-pixel radius around the last know position to maximize the speedup. For this mode, we assume that object moves slowly between frames. If the tracking confidence falls below a threshold (0.4) for a period of 50 consecutive frames, or if the bounding box drifts outside the frame, the system switches to Global Search, which scans the entire frame to recover the object. Once confidence reaches the high-confidence threshold (> 0.6), the system converts back to Local Search.

3.5.1 Lost-Frame State Machine

```
State: Local Search
  ↓    (low confidence for 50 frames OR bounding box outside frame)
State: Global Search
  ↓    (Reach high-confidence threshold)
State: Local Search (reset counter)
```

3.6 Template Adaptation

To handle appearance changes during tracking, the template is adaptively updated. When a match is found with strong confidence (> 0.7), the current template is blended with the new frame patch using an exponential moving average with a learning rate of 0.1. This allows the track to adapt to gradual changes without the risk of drifting too easily due to noise.

$$templ_{new} = (1 - LR) \times templ_{old} + LR \times frame_{patch}$$

Issue: This adaptive update does not work when the target has large geometric change, such as in-plane rotation. For example, in one of our input videos, a nail appears horizontally in the initial template but later becomes vertically oriented. This change is too abrupt for the exponential moving average to capture, causing the partially updated template to become invalid. The issue is not an 'improper' learning rate. Rather, NCC itself is inherently insensitive to large rotational or structural variations. The tracker may fail even though the adaptive update mechanism behaves as designed.

3.7 Baselines

To contextualize the performance and robustness of our CUDA NCC tracker, we implemented and evaluated two additional baselines: (1) a CPU-based OpenCV tracker using CSRT, and (2) a GPU baseline using pure optical flow (Farneback) on CUDA. These baselines illustrate the contrasting characteristics of traditional CPU tracking pipelines and generic GPU motion-estimation methods.

3.7.1 Opencv CPU

Our first baseline uses OpenCV's built-in `TrackerCSRT` implementation, a well-established correlation filter tracker designed for accuracy and robustness. The workflow includes ROI selection, tracker initialization, per-frame update, drawing, and video writing. Timing is measured for decoding, tracking, drawing, and write operations.

The CPU baseline is highly stable, even under mild occlusions, illumination change, and moderate motion. CSRT incorporates spatial reliability maps and uses features such as like color histograms, making it more tolerant to noise than raw optical flow or raw template matching. It can track object moving very fast, turning to new angles frequently, and adapt changes quickly.

However, this robustness comes at a cost: The processing time for CPU to compute is relatively high, and CPU will become a bottleneck, especially when decoding and writing frames concurrently.

Overall, the CPU baseline provides a strong accuracy reference but achieves only modest throughput.

3.7.2 GPU with Simple Pure Optical Flow Algorithm

The second baseline leverages `cv::cuda::FarnebackOpticalFlow` to estimate dense optical flow between consecutive frames. After computing per-pixel displacements on the GPU, we extract the flow vectors inside the bounding box and use the median motion to update the object location. This simulates a simplistic "track by motion" algorithm.

This baseline is expected to exhibits unstable and inconsistent tracking performance, as pure optical flow tracking is extremely unstable and cannot tolerate changes in object of angle / speed / or even mildest changes. It only estimates pixel motion but does not ensure that the motion inside a bounding box corresponds to the actual tracked object.

On the running time side, despite running on the GPU, dense optical flow requires great memory bandwidth and produces large flow fields each frame. Our measurements show that the flow compuation took quite a lot of time, and harming throughput.

Thus, the pure optical-flow GPU baseline is fast in theory but unstable in practice, and often produces worse accuracy than both the CPU baseline and our NCC approach. More importantly, it has high overhead and low throughput overall.

4 Result

We are running our test with short video from the following website:

`https://www.pexels.com/video/mini-cooper-on-highway-along-the-desert-1572547/`

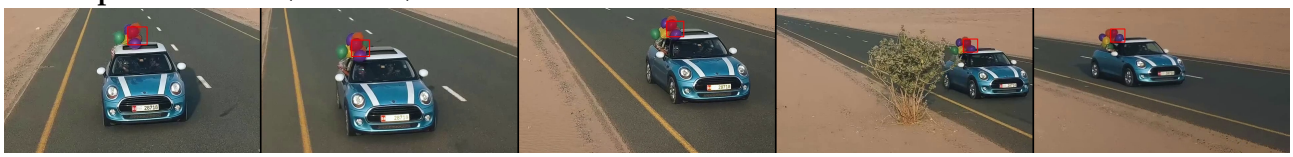
We are running our test by selecting an area on the balloons to track. The video includes scene of a car driving smoothly, with angle, speed, size changing in the view of camera (camera angle changing), and a scene passing behind a bush that blocks the sight of our target for around 0.5 second (common

lose track happen here). We present our performance review and analysis below:

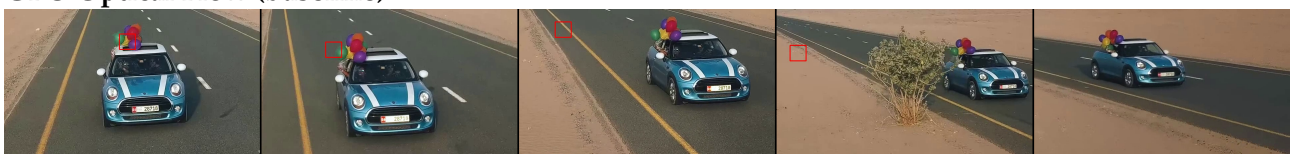
4.1 Performance in Tracking Quality

Here we show the tracking performance of frame. This video consists of 476 frames in total, and we select frames `50, 150, 250, 350, 450` to present.

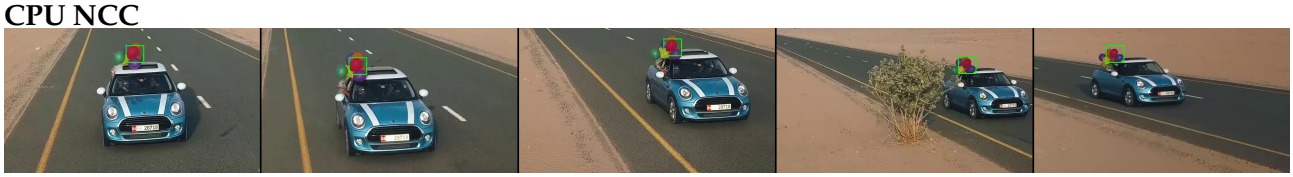
CPU OpenCV CSRT (Baseline)



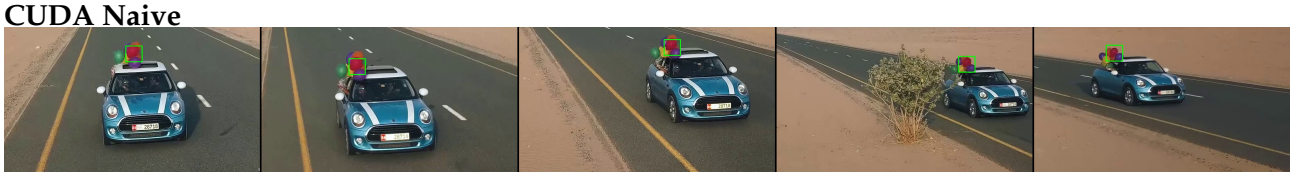
GPU Optical Flow (baseline)



CPU NCC

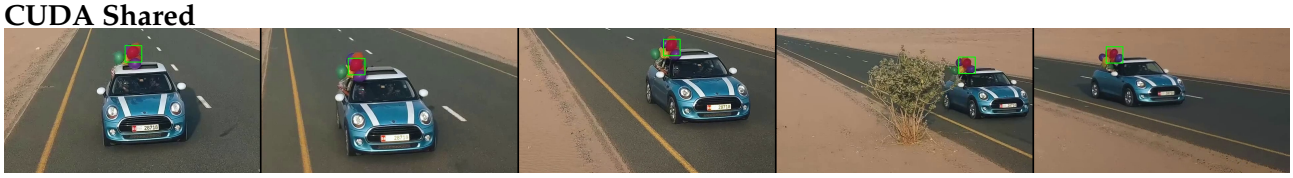


CUDA Naive

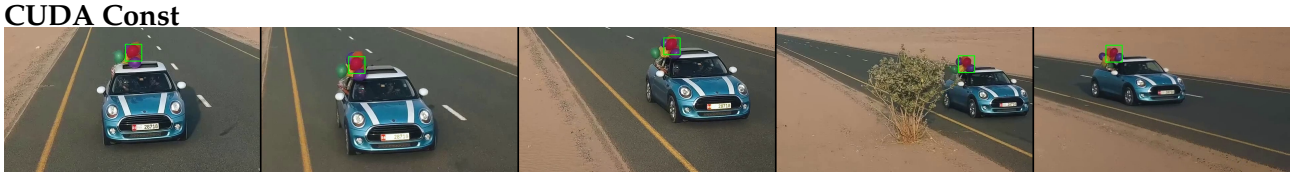


CUDA Naive Batched (4)

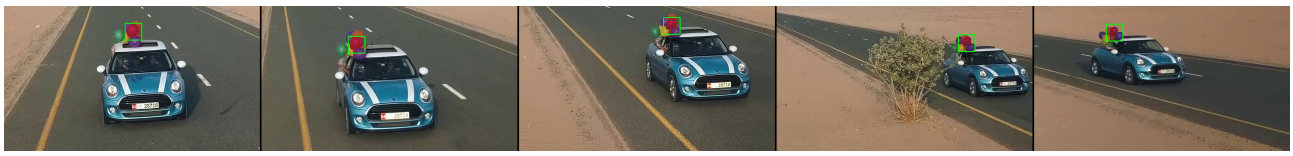
CUDA Shared



CUDA Const



CUDA Const Tiled



Observe that CPU OpenCV CSRT version tracks most smoothly and reliably. This is expected as it has been developed and refined for years. Our another baseline, GPU Optical Flow, however, performs badly and start the lose track from very early stage.

All other NCC versions (Our implementations) performs well in tracking stability, whereas instability occurs in some scenes, they generally tracks well. Most lose track and instability occur from the vehicle goes behind the bush (a little prior to frame `350`). Despite risk, our NCC implementation successfully handled most cases and re-tracked the balloon after the tree bush (as shown in frame `350`). Even though the stability generally cannot compare to the OpenCV built in, which uses CSRT and FFT, they still performed satisfactory in these cases.

Notice that our batched version is running with batch=4, it means it sacrifised FPS in box drawing and tracking, as it update tracking box every 4 frames (causing lag and further instability).

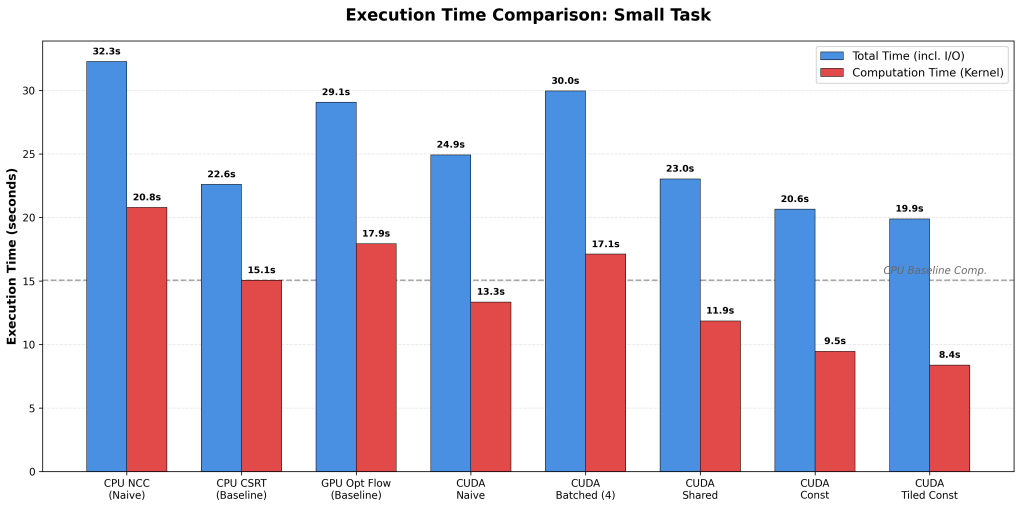
For detailed video reviews, please refer to

`https://github.com/askEric0/Parallel-Video-Object-Tracker/tree/main/output`

4.2 Performance in Execution Time

Table 1: Summary of Total Time and Computation Time, Small Task

Method	**Total Time (sec)**	**Computation Time (sec)**
CPU OpenCV CSRT (baseline)	22.5979	15.0740
GPU Optical Flow (baseline)	29.0744	17.9328
CPU NCC	32.2753	20.787
CUDA Naive	24.9275	13.3503
CUDA Naive Batched (4)	29.9643	17.1203
CUDA Shared	23.0411	11.8647
CUDA Const	20.6547	9.4693
CUDA Const Tiled	19.8930	8.3923



4.3 Speedup and Performance Analysis

The execution times in Table 1 reveal a clear progression of performance as we move from the CPU implementation to increasingly optimized CUDA kernels. The CPU NCC version requires 32.27 seconds to process the 476-frame video (14.7 FPS), establishing the baseline for our speedup analysis. Most of GPU implementations exceed the CPU throughput, but their performance varies depending on memory access patterns and kernel design.

4.3.1 Naive CUDA Kernel.

The naive GPU kernel reduces the total runtime from 32.27 seconds to 24.93 seconds, yielding a speedup of approximately 1.29× over CPU NCC. More importantly, its computation time drops from 20.78 to 13.35 seconds (1.55× speedup). This improvement comes purely from massive thread-level parallelism, despite the kernel still suffering from severe memory redundancy.

4.3.2 CUDA Shared-Memory Kernel.

Moving the template into shared memory further reduces computation time to 11.86 seconds (1.75× faster than CPU NCC and further 1.12× faster than the naive kernel). The reduction reflects fewer global memory transactions per block, leading to a increase in efficiency of memory access.

4.3.3 CUDA Constant-Memory Kernel.

Caching the template in constant memory improves performance greatly: computation time drops to 9.47 seconds (2.19× faster than CPU NCC and further 1.40× faster than naive CUDA version). Since the template is broadcast efficiently by the constant cache, all threads benefit from reduced memory latency without consuming the limited shared memory resource.

4.3.4 CUDA Constant-Memory Tiled Kernel (Best).

The final optimized kernel incorporates both constant-memory template storage and shared-memory tiling for the frame. This provides the largest reduction in global memory access. The computation time falls to 8.39 seconds, which corresponds to a 2.48× speedup over CPU NCC and a 1.59× speedup over the naive GPU kernel.

4.3.5 CUDA Batched Kernel.

Although batching can improve throughput in large offline workloads, the batched kernel is slower than the naive kernel for small videos. The total time increases to 29.96 seconds. This indicates that: batching introduces additional memory overhead, increases working-set size per launch, and breaks the per-frame template update logic. The fact is that there is no memory reuses between frames, and it is not realizable to calculate multiple frames together to improve speed up.

4.3.6 Comparison to External Baselines.

The well-developed CPU OpenCV CSRT tracker achieves 21.06 FPS, outperforming our CPU NCC baseline but still slower than our two fastest GPU kernels. The GPU optical-flow baseline, while running on CUDA, performs poorly both in tracking quality and computational efficiency (computation time: 17.93 seconds). Farneback optical flow produces large flow fields every frame, leading to high bandwidth consumption and degraded performance compared to our optimized NCC kernels.

Across all measurements, the optimized `const_tiled` variant is the best, reducing computation time by nearly a factor of three compared to the CPU implementation and by 37% compared to the naive GPU version. The const cache broadcast and memory tiling increase the speed of execution to the most extend in these cases and, at the same time, maintained a high-quality tracking performance.

4.4 What Limited Speedup?

Our speedup was primarily limited by Memory Bandwidth and Data Transfer Latency.

1. **Memory Bandwidth:** In the Naive kernel, overlapping template positions caused massive redundant reads from global memory as adjacent threads read overlapping regions. For a 50×50 template, each output position reads 2500 frame pixel. Such overlapping reads greatly increase the memory traffic. This is generalizable for all implemenations: even the optimized kernels remain partially constrained by frame loads, since frame data is much larger than on-chip memories and must be fetched repeatedly.

2. **Host-Device Transfer:** As shown in Table 1, the gap between Computation Time and Total Time is large. Even for a small task, it still needs to transfer about hundreds frames, which create a great bottleneck that kernel optimization cannot solve. This gap reflects unavoidable host–device synchronization, frame uploads, NCC-map downloads (when needed), and kernel launch latencies. For a 476-frame video, the PCIe transfers dominate a nontrivial portion of the timeline. Unlike kernel optimizations, data-transfer costs do not scale with thread count and remain a fundamental bottleneck for small or medium-sized videos.

3. **I/O** For similar reason video loading, frame parsing, frame writing, I/O consumed a lot in the total time, leading to total time to be more than twice as high as computation time.

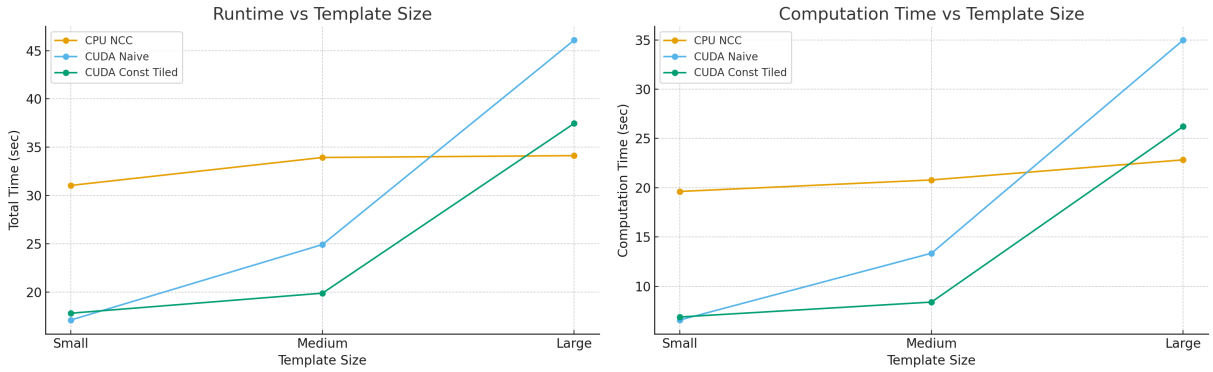
4. **Batching Overhead:** The Batched Kernel was slower for the small task than the Naive Kernel. The latency of buffering multiple frames and managing a 3D grid could outweighed the throughput benefits. It will definitely create larger global memory footprint, and higher kernel-launch latency. More importantly, it is sacrifising tracking stability as it updates box every batchsize frames.

4.5 Problem Size

To study the sensitivity of running time with problem size, we change the problemsize and monitor execution time again.

Table 2: Effect of Template Size on CPU, CUDA Naive, and CUDA Const-Tiled Performance

Method	Template Size	Total Time (s)	Computation Time (s)
CPU NCC	Small	31.0477	19.6193
	Medium	33.9384	20.7870
	Large	34.1315	22.8273
CUDA Naive	Small	17.1132	6.5718
	Medium	24.9275	13.3503
	Large	46.0901	34.9799
CUDA Const Tiled	Small	17.8173	6.8794
	Medium	19.8930	8.3923
	Large	37.4635	26.2133



Overall, shrinking the template size improves performance for all implementations by reducing both arithmetic work and memory traffic. The CPU sees a smallest runtime decrease, while the GPU kernels benefit much more strongly. Among the GPU variants, the naive kernel is extremely sensitive to template size and loses most of its advantage at large templates, whereas the `const_tiled` kernel maintains good speedups for small and medium templates and degrades more gracefully as the problem size grows.

For NCC template matching, theoretically, the increase of problem size, will directly also increase the running time. Each NCC evaluation requires computing the mean, variance, and covariance over a patch of size $w \times h$. Doubling either template dimension (e.g., from 50×50 to 100×100) quadruples the per-thread computation, since the patch area grows quadratically. That is because, each NCC evaluation requires reading $w \times h$ pixels from the frame. Larger templates increase the total memory bandwidth requirement, pushing the kernel closer to the device's memory-bandwidth ceiling. This makes the kernel more memory-bound and reduces potential speedup. Large templates will also reduce the effect of shared memory operations, or even may exceed the required shared memory bound, which will lead to performance drop.

4.6 Deeper Analysis of Execution Time Breakdown

In our optimal design **CUDA Tiled Constant** kernel, the total time is **19.89 seconds** while the computation time is **8.39 seconds**. The difference in total and computation time reveals that our application is no longer computation-bound, but rather system-bound due to the initialization and data transfer overhead.

Kernel Execution (~42%): The actual NCC computation on the GPU only consumes 8.39 seconds. Compared with the baseline kernel, this confirms that our memory optimization through tiling and constant cache successfully reduced the computation latency.

System Overhead (~58%): The remaining 11.50 seconds are used for the non-computation operations,

which includes the data transfer between Host and Device and Video I/O and prepossessing. Each input video must be decoded into individual frames, and transferring these large frame buffers to the GPU introduces a significant serial bottleneck. Similarly, transferring the NCC map back to the host for peak detection and bounding-box updates adds additional overhead. These operations on CPU side could then dominate the runtime.

Therefore, the sequential logic on CPU side becomes the bottleneck of this system. To improve the overall system, we could use CUDA streams so that the NCC computation of the current frame and the memory reading for the next frame can be implemented simultaneously so that we could overlap the memory transfer with the computation, using similar pipelining logic to hide the transfer overhead. Furthermore, we could do peak finding in parallel to reduce the sequential logic in the overall system.

4.7 Machine Target Analysis

GPU was the right choice. NCC computation could benefit greatly from data parallelism ($O(W \times H)$ independent tasks). A sequential CPU implementation cannot approach the throughput of thousands of concurrent GPU threads, especially given the high arithmetic intensity of template matching. For Scalability, As resolution increases, the compute workload would grow dramatically while the PCIe transfer grows linearly. Thus, the GPU becomes increasingly efficient for large videos. However, for very small or low-resolution videos, the CPU baseline has the similar performance to GPU's since the initialization and transfer overhead is much smaller.

5 List of Work

Credit: 50%-50%.

Week 1:

1. Set up project structure, build system, and OpenCV I/O + visualization with CPU baseline track for reference (Yanxin)
2. Write a naive CUDA kernel for template matching and verify correctness with visualization. (Eric)

Week 2:

1. Add batched frame processing to test optimization. (Yanxin)

2. Add shared memory to test optimization. (Eric)

Week 3:

- Applied constant memory and shared-memory tiling to GPU kernel. (Eric & Yanxin)
- Add interactive mode that visualize the tracking process in real-time performance metrics. (Eric & Yanxin)

Week 4

- Clean up and finalize the code for the final report. (Eric & Yanxin)

6 Summary

In this project, we developed a high-performance video object tracker based on Normalized Cross-Correlation and systematically optimized it across multiple stages of the GPU memory hierarchy. Starting from a naive CUDA implementation, we progressively introduced shared memory caching, constant-memory broadcasting, and finally a tiled constant-memory design that significantly reduced redundant global memory traffic. These optimizations achieved up to a 2.5× speedup over the naive baseline and showed near real-time performance at $\sim$ 24 FPS, making them capable of produce streaming tracking. At the same time, they maintaining good tracking quality. Through comparisons with CPU-based CSRT and GPU optical-flow baselines, we demonstrated that purpose-built GPU kernels can outperform both traditional CPU trackers and general motion-estimation algorithms in throughput. The NCC algorithm surpassed the pure optical flow can be concluded. However, OpenCV CPU CSRT is well developed and polished for years, making them still on the top of tracking quality.

We also experimented with the batching idealization where it brings overhead on small tasks, which becomes one blocking factor of speed up. The Limiting factors include: Host-Device Transfer, Memory Bandwidth Limits, and I/O operations. We successfully identified following problems and gradually improve the kernel design, resulting a satisfactory result.

References

[1] N. Ahmed, E. Teters, R. Aktar, I. Akturk, G. Seetharaman, and K. Palaniappan, "Gpu and multi-threaded cpu enabled fast normalized cross-correlation," contributed equally.