CSCI580: Project Milestone Update

Andrew Depke, Justin Davis

ACM Reference Format:
Andrew Depke, Justin Davis. 2022. CSCI580: Project Milestone Update. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In this paper, we present a prototype GPU-accelerated and scale-invariant object tracking algorithm. This algorithm is not a new development by itself, but our novelty stems from enabling real-time use of this algorithm when graphics accelerator hardware is available.

Before discussing the prototype we've built, we must first explain some of the driving decisions in our design and how they've impacted our prototype. First, we've decided against using a neural network for our object tracking method largely due to the performance trade-off demanded. While modern neural network tracking solutions can be significantly more accurate and adaptable, they often draw far more computation requirements than traditional algorithmic approaches. For many edge devices, running a neural network with any sufficient amount of accuracy is simply infeasible, so using other approaches is a requirement. It's vital to note that just because many edge devices are more on-par with smartphone in terms of computation potential, many devices still have dedicated video accelerators and have the option of using multi-threading to aid with performance. This was reflected in our prototype design, since we have models designed specifically for edge devices with only CPU multi-threading, as well as models that can leverage CUDA capabilities if present on the device.

We chose to implement the scale-invariant KCF algorithm, which is more computational demanding than the baseline KCF algorithm, largely due to what we believe is a necessary improvement in runtime tracking adaptability. The cost of the additional scale-invariance enables the algorithm to track objects moving along the Z axis, where the Z axis in this case is along the camera's sight vector. Without this change, tracking will be lost if the scale changes from the initial patch size. If consistent tracking along this additional axis is not a necessity for a given case, then the additional overhead of the scale-invariance can be removed without too much effort, while still maintaining the benefits of the parallelized components of the baseline algorithm.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 THE PROTOTYPE

We present an implementation of the scale-invariant kernelized correlation filter (KCF) algorithm for object tracking. For our in-development prototype, we've created a custom module for OpenCV 4.5.0, leveraging OpenCV's built-in CUDA tooling.

The module was derived from the original implementation present within OpenCV's contrib modules. This implementation was heavily tied in with the OpenCV build system and we decoupled this implementation so it can be compiled independently only depending on OpenCV's public API. This original decoupled version is what we use refer to as the baseline, standard, or std implementation.

Our current prototype adds OpenMP as a dependency for allowing full use of a CPU. This differs from OpenCV's original implementation (before decoupling) which utilized many pre-compile stages for a wide variety of CPU targets and extensive use of OpenCL to multi-threading.

Tools used:

(1) OpenCV - 4.5.5
(2) CUDA 11.7
(3) Ubuntu 20.04
(4) Tracy C++ Profiler
(5) Python 3.8 with matplotlib, numpy, and OpenCV 4.6

3 CHALLENGES

Initially porting the algorithm to modern OpenCV was luckily fairly straightforward, the hardest component being setting up the build system. To solve this, we created per-platform installation scripts that would fetch versioned content for OpenCV as well as the CUDA modules we use. With these working, we could simply call $FindPackage(OpenCV)$ in our CMake script to pull all includes and link properly. We have a similar process for using OpenMP.

We faced some harder challenges while optimizing however, largely related to using CUDA kernels. While OpenCV's CUDA module does provide some prebuilt kernels for parts of our algorithm, some of these kernels are not usable out of the box. An example of this is the discrete fast-fourier kernel provided, since this provides a merged-plane output of the complex and real number spaces, but we're only interested in the complex number space for our algorithm. In theory we can split the planes as a post-processing step, but this is an additional hurdle to overcome. We believe this difficulty mainly comes from the fact that OpenCV calls cuFFT which is a library that Nvidia develops. Since these two libraries are not necessarily inter operable it is no surprise we had difficulties using these tools out of the box. Many of the modules we were planning to utilize from OpenCV are also part of their contrib package which are labeled as may not work.

We hope to overcome this issue by writing calls directly to cuFFT ourselves or by duplicating and modifying the OpenCV API for accessing that library. We get valid outputs using the cuda::dft functions, but all of our bounding boxes become shifted by 200 pixels to the left. Whilst we could have done a performance analysis and

Metric	Baseline	Optimized
Mean (ms)	17.28	8.68
Median (ms)	16.9	8.65
Min time (ms)	15.52	3.83
Total time (ms)	3400	1710

Figure 1: Performance measurements for ExtractCN

predicated an expected speedup, we felt that would be disingenuous since our results would not actually be valid.

Some other challenges we faced are getting the actual source code for certain OpenCV functions in order to make external optimized versions. Some examples of these include the fft, ifft, fft2, and mulSpectrums functions. While OpenCV is open source, it is such a large library that it is difficult to nail down what you can actually remove from the build structure and compile independently.

4 EVALUATION

For evaluating our optimization efforts, we used [Tracy], a hybrid instrumentation and sampling profiler for the CPU and GPU. Using this tool, we were able to add manual zone markup to our modified function calls to get accurate timings around every function, as well as gather hardware performance counter samples at a fixed polling interval. Using the combination of these two metrics we retrieved accurate and fine-grained measurements for the impacts of our changes. In the following figures displaying code breakdowns, the left-most column of numbers is broken down into the following metrics: % time spent, instructions per cycle (IPC), branch misprediction rate, and cache hit rate. These columns are color-coded as a visual aid as well.

In this section we'll break down the fundamental steps of the KCF algorithm and examine where optimizations and restructuring took place.

(1) Extracting color names
Before the tracking algorithm can begin image processing, we first need to generate the color names feature data plane. To do this, the traditional implementation iterates through all pixels in the source image and looks up the corresponding color name value in a very large precomputed lookup table. The following table shows our performance measurements before and after optimization, as well as a further breakdown and analysis of the microcode.
How we achieved an approximately 2x performance improvement as seen in 1 is discussed by analyzing the algorithm changes. When initially tackling this function, we were able to trivially parallelize looping over all of the pixels in the image by giving each thread a collection of rows to iterate over independently. This was a strong improvement, but we were still suffering from large cache misses due to the size of the source image. For this, we introduced a batching system where multiple adjacent pixels could be processed per outer-loop, improving cache locality. Interestingly, we found that a batch size of just 2 was optimal after extensive testing, despite causing the branch predictor to always miss on the inner *for* loop as seen in 3, line 676, third column (100% miss rate). Our final optimization for this function was to tackle the 100% branch-predictor miss rate and the 90%+ cache miss rate in 2, lines 570 and 571. To fix this, we could simply hand-unroll the loop such that the *k* iterator variable would not be evicted from cache everytime the massive *ColorNames* table would get loaded into cache. This optimization removed all branching and improved per-line cache hit rates by over 9 times. Even though our total instructions increased, the lack of control flow implies that all instructions can be perfectly linearly prefetched, resulting in a large overall improvement.

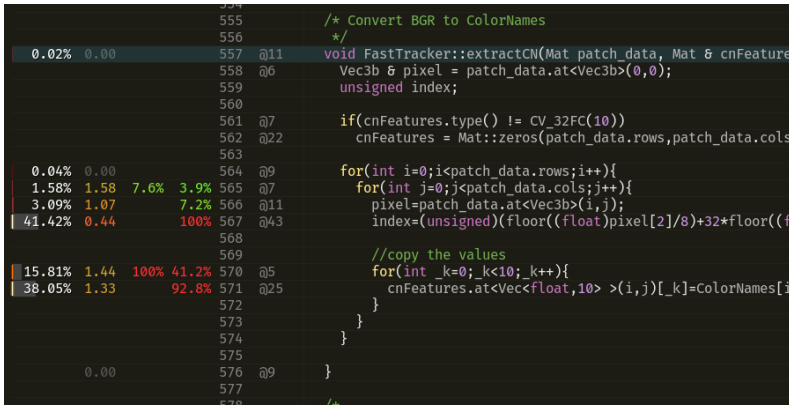

Figure 2: Image Showing Profiling of ExtractCN

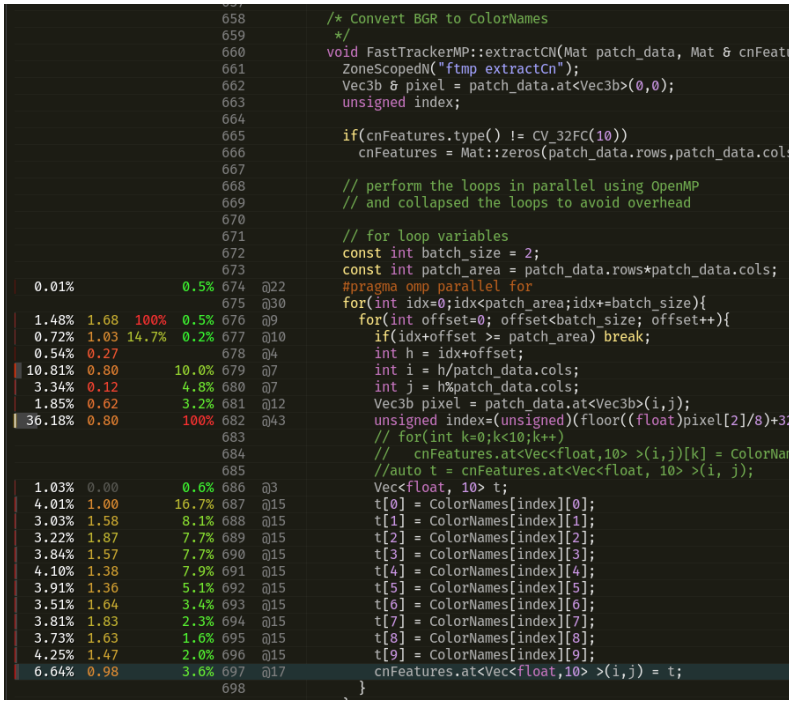

Figure 3: Image showing Profiling of the Optimized verison of extractCN

(2) Dense Gaussian kernel
The dense Gaussian kernel stage is composed of several fast-fourier transforms, as well as scaling, summing, and an inverse fast-fourier transform. To optimize this component of the algorithm, we first looked at multi-threading the discrete FFTs, however we found that the existing baseline implementation luckily already provides a CPU-optimized implementation. We were able to leverage the CPU-parallelized version, for our CPU-only model, but we'll have to likely implement our own CUDA kernel for this function for our GPU-accelerated model of the algorithm. We next looked at parallelizing the thresholding step used after the inverse FFT step. For thresholding, all pixels in the image are iterated, setting any pixels that are negative to just be 0. Our baseline implementation does this sequentially, but we were able to improve this by using OpenMP to parallelize the nested loop to be split among our thread pool. This was a trivial performance improvement since each iteration of the loop is entirely independent and can be binned into threads easily.
(3) Updating the projection matrix
For the most part this part of the algorithm is fairly quick, however there is one wide matrix multiplication that is responsible for a large amount of execution time. Updating the projection matrix involves taking the PCA data from a prior stage and multiplying it with its own transpose, then scaling the resulting matrix by $\frac{1.0}{rows \times columns}$. The matrix multiplication is where most of the time of the function is spent, so we optimized this be introducing a window-based parallel matrix multiplication algorithm as discussed in lecture. The basic idea is to take a chunk of the rows of the first matrix, as well as a chunk of the columns of the second matrix and multiply them, creating a resulting submatrix that can later be merged with other submatrices, which are evaluated in parallel, which allows for a cache-optimized dense matrix multiplication. We also extended this trick to the scaling step that happens immediately afterwards, where we could process rows and columns of the input matrix in batches which would would apply the scaling multiplication to. Since this could be trivially parallelized with OpenMP, the projection matrix update function could be significantly optimized when compared to the baseline implementation.

For our final evaluations and testing the performance of our optimized algorithm, we utilized the Got-10K data set [1]. This data set focuses on "in-the-wild" objects so it typically provides a good performance analysis for algorithm accuracy. We chose to utilize this data set since it is readily available and has a variety of image and bounding box sizes for us to benchmark. We also chose this over a synthetic data set of solid color images, since the algorithm can fail and it can help us measure accuracy at a later date. We utilized the testing portion of the Got-10K data set, which contains 180 videos of various lengths and resolutions. This data set is distributed as folders of jpegs with a starting groundtruth bounding box. We transformed the data into folders of .avi video files to ease reading the data.

For our evaluation workflow we first read in all .avi video paths. Then we sort them so we read them in ascending order. Then each video is read and each version of the tracker is run on the frame, with timing results done with the C++17 std::chrono library. We store all the timing results for each frame of each video in text files which we then read in with a Python script. Our script takes all the data and stores it into a map with the key being the bounding box size and the data being the time per frame. Once that data is processed we run some statistics on percentage differences between the standard and optimized times.

Overall the results are tabulated in 4.

Metric	Improvement
Initialization Time	x0.49
Update Time	x1.36
Max Update Speedup	x2.33
Min Update Speedup	x0.07
Median Speedup	x1.32

As can be seen our optimized version can significantly speedup the time to update per frame with an average of 1.36 times. However, this comes at a downside of effectively doubling the initialization time. This was deemed acceptable since setting up threads, allocating memory, and other factors occur in the init method which handles initialization. Since a typical tracking scenario has many updates for a single init this becomes increasingly inconsequential. This becomes clear when the graphs of initialization time and update time relative to bounding box size show up.

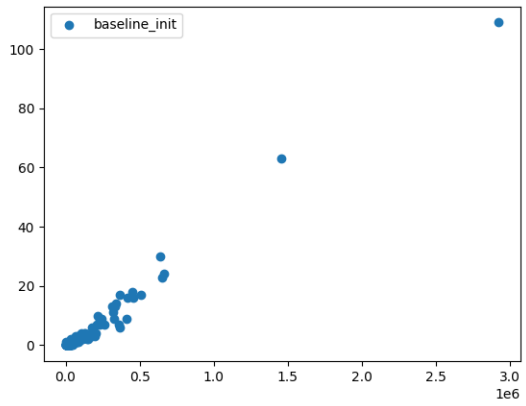

Figure 4: Graph of baseline init time vs. bounding box size

Once analyzing the figures we realized there were certain trends present which showcase when our multi-threaded optimizations function best.

(1) When the bounding box area is greater than $0.1 * 10^6$, 10000, or a $100x100$ bounding box, we can use the multi-threaded implementation without any worries. Although often we would want to use the multi-threaded versions for IoT scenarios anyways since we have continuously running tracking which would allow the thread initialization to be a split cost between many instances.
(2) Our optimizations begin to be more beneficial as the number of pixels increases. For the majority of the data set, smaller

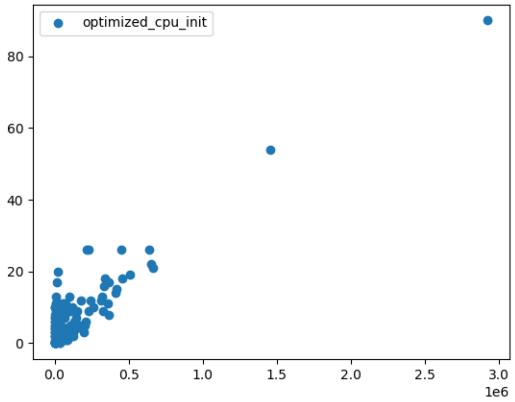

Figure 5: Graph of optimized init time vs. bounding box size

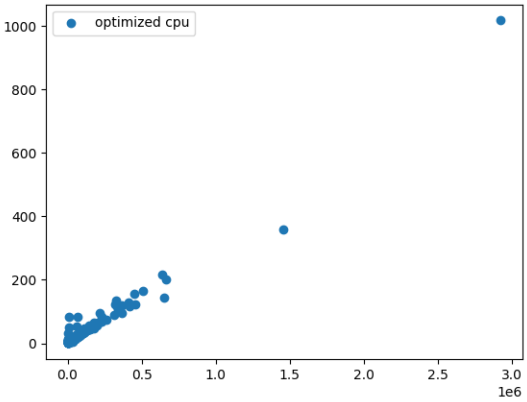

Figure 7: Graph of optimized update time vs. bounding box size

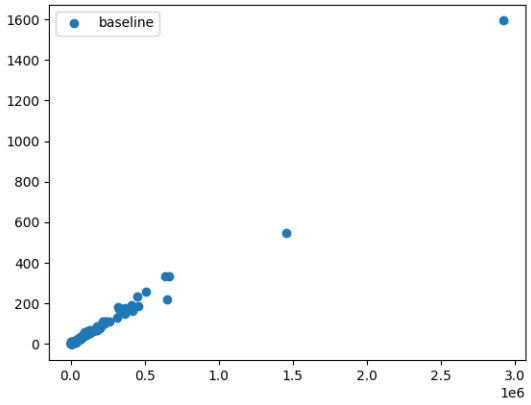

Figure 6: Baseline of update time vs. bounding box size

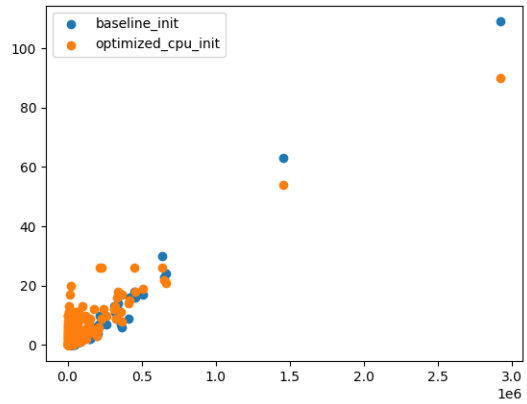

Figure 8: Graph of baseline and optimized init time vs. bounding box size

bounding boxes are present which mean other optimizations from multi-threading may be more beneficial (such as vector operations). However for large bounding boxes, our optimizations become significant. For the largest size bounding box we see a 1.6x speedup and for the second largest we see a 1.57x speedup. We believe that our optimizations would therefore yield 1.5x speedup for the majority of tracking scenarios at high resolution in IoT scenarios. Since many IoT scenarios are also indoors we believe that larger bounding boxes would also appear more often, whereas the Got-10K dataset is mostly outdoor footage.

(3) Time complexity of the algorithm is not changed. The current bounding factor of the KCF tracker is the fast Fourier transform which has a time complexity of $O(nlogn)$. For our scenarios this becomes, $O(m * n * log(m * n)$ for an mxn image. However, we can see from the trend present in figure 9 that each version has the same time complexity, but we managed to significantly reduce locality overhead and utilize the CPU better.

5 WHAT'S NEXT

While our prototype implementation has some significant performance improvements over the baseline, we are still looking to further improve our model. There is a considerable amount of the algorithm that has not yet been ported to CUDA kernels, and there are places where keeping the data on the GPU between sequential passes instead of downloading back to the CPU would be highly beneficial. One of our big hotspots we want to replace with a CUDA kernel is the non-inverted discrete fast-fourier algorithm used multiple times in the dense Gaussian stage of the algorithm. Unfortunately OpenCV's existing CUDA-accelerated FFT ($libcufft$) does not allow for separated-plane output of the real and complex number spaces for the DFT result. We could solve this by downloading

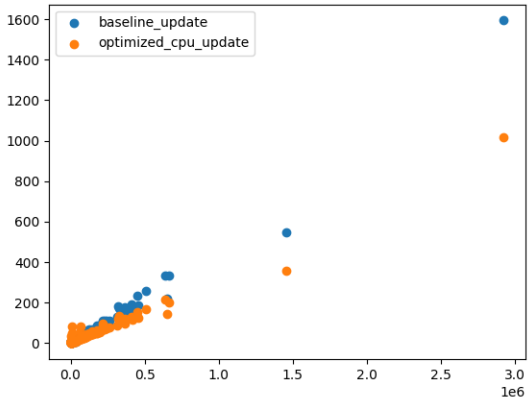

Figure 9: Graph of baseline and optimized update time vs. bounding box size

the two-plane output and isolating the complex number space that we're interested in, or modify the kernel to perform the split on the GPU and reduce the total data transfer required for the operation.

Another hotspot for optimization is our lack of kernel merging so far, which is causing a non-insignificant amount of execution time to be spent waiting for data to be sent or arrive over the PCIe bus. One of the big challenges with merging kernels is the control flow present in the algorithm, which GPUs are not proficient at handling. In addition, on some edge devices, such as the Xavier NX we profile on, the memory architecture is flattened and shared between the CPU and the GPU, so merging kernels would have almost no if any impact at all.

We can also focus on improving locality and introducing parallelism to some of the more 'mundane' sections of the algorithm. One such area is where all feature layers are iterated over a series of times to check for errors. If those sections were able to be ran in parallel then it would help reduce the run time for each update even further. From our profiling it would appear that a 1.25x speedup in some function calls can be achieved.

Another focus could be identifying portions of memory copying which could be reduced. Many of the OpenCV implementations utilize many memory copy operations to ensure that no data is operated on twice or effected by more than a single function. Identifying these memory copies and eliminating them ahead of time could significantly speedup the algorithms run time for all bounding box sizes. We would also like to focus on improving the fast fourier transform and inverse fast fourier transform for the CPU. We believe that a multi-threaded CPU version would be beneficial for bounding boxes below a certain size where the data transfer overhead is too great for GPU workloads.

REFERENCES

[1] Lianghua Huang, Xin Zhao, and Kaiqi Huang. GOT-10k: A large high-diversity benchmark for generic object tracking in the wild. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(5):1562–1577, may 2021.