CSCI580: Project Proposal

Andrew Depke, Justin Davis

ACM Reference Format:
Andrew Depke, Justin Davis. 2022. CSCI580: Project Proposal. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 PROBLEM DESCRIPTION

Many modern object detection algorithms are massive convolutional neural networks that demand high-powered hardware and significant computational time to detect and track objects from an image feed. In many cases, the high precision of deep neural network trackers is unnecessary and even a bottleneck, so we can turn to traditional algorithmic-based tracking instead.

These algorithmic-based tracking methods are traditionally implemented using sequential algorithms on CPUs. While their time per frame is already far improved from that of deep neural network-based trackers, their accuracy suffers. Despite this accuracy decrease, algorithmic trackers should not be discarded since they can be used to "fill in the gaps" between inferences of a higher accuracy tracker.

Deep neural network trackers are replacing traditional algorithmic trackers at an increasing rate. However, for many applications, the increased accuracy, occlusion detection, and failure detection are not needed or add unneeded complexity. Many examples where this is the case are edge, internet-of-things devices, and industrial applications. This could be present in an application where the controlled tracking of fast-moving objects is occurring. An example is a high-speed seed sorter. These sorters let the seeds fall through an illuminating chamber and use compressed air to separate rejects from the stream. There is a small window of time for the machine to analyze the seeds with a camera and compute how to remove them. Having high-speed tracking would allow more accurate rejections to occur. The sorter or other similar applications could benefit highly from having access to higher speed tracking algorithms versus the higher accuracy deep learning-powered ones.

2 CHALLENGE AND INTEREST

The challenge around this problem can be split into a three part scheme:

- **Parallelization**
  These algorithmic trackers have traditionally been implemented and designed to be run on a single CPU core. There is a multitude of trackers available through OpenCV, but all of which implement a sequential algorithm for acquiring the result. Thus, the input data and/or algorithm will have to be rethought from a parallel standpoint for the chosen tracker which we will implement.
  This transformation from a sequential to a parallel algorithm can pose significant challenges should there be a sufficient overlap of tasks. For the KCF tracking algorithm, the process can be divided into three stages.
  - **Image filtering**
    This includes Gaussian filters, fast Fourier transforms, convolutions, and other common image processing techniques. While many of these techniques have either a parallel or GPU implementation, they are all stand-alone and we would seek to combine them into one continuous parallel process or a single GPU kernel.
  - **Computing the circulate matrix and online learning of the model**
    The model which KCF follows uses principal component analysis and some ridge regressions to compute a representation of the object/region to be tracked. These methods as implemented and used are cost-intensive and thus the first frame of tracking incurs a much higher cost than the remainder of the tracker. This gives another disadvantage compared to deep learning trackers which have a relatively constant run time no matter which frame is being processed.
  - **Condensing, updating, and comparing features**
    Algorithmic trackers rely on data structures being updated in place. The methods utilized in this algorithm rely on "condensing" features computed from each new frame. These features are condensed one by one and end up being processed up to **four** times for each new frame where the feature set can be more than the number of pixels in the region of interest. Updating and comparing the features face a similar dilemma where each pixel is handled individually all sequentially.
- **CUDA Kernels**
  To leverage the compute resources available through the graphics processor, we'll need to write custom CUDA kernels. The KCF algorithm implementation in OpenCV is already composed of some GPU-accelerated kernels, but many if not all of these will be insufficient due to OpenCV using CPU-readback of the outputs. In a performance-critical environment, we cannot afford to idle while waiting for data to be marshaled across the system's bus into the main memory. Instead, we can keep the data local to the GPU in DRAM using custom kernels and not perform any readback until the objects have been detected. Unfortunately, the algorithm cannot be implemented with just a single kernel, since multiple passes occur, which change the shape of the resulting data. However, we expect little to no impact from this considering that the data will not be moved out of video memory during transitions of these passes.

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

- **Correctness**
  The purpose of using an algorithmic replacement for object detection over a more traditional neural network is to achieve similar detection accuracy while expending far fewer computational resources. Our approach will have no value if it's unable to detect objects with arbitrary and possibly changing scales with sufficient accuracy. We hope to achieve similar detection and tracking capabilities as algorithms such as KCF.
  Our goal for correctness is to maintain a $+/-1$ pixel margin for each of our bounding box coordinates relative to the OpenCV implementation. This will allow us to incur minor type casting and floating point precision inaccuracies while still maintaining high accuracy.

Despite this three-stage scheme for optimization, the core tracking algorithms all rely on a two-stage process. They are fed an initial bounding box and frame and use online learning to compute the reference point. Then they are fed a new frame and find the previous box in a new frame. This simple API/interface of init and update allows easy testing and gives lots of freedom for the underlying algorithm implementation.

3 OUR APPROACH

(1) **Feature extraction and fusion**
Before we can execute the KCF algorithm, we first need to create the input data to feed it. For this, the RGB video feed is segmented into a series of still frames, where we can execute tracking on each frame, or at a reduced frequency if desired. Using a given video frame, we first need to use a variety of color space conversions to aid gradient and general feature extraction in the following steps. The image will be converted to a histogram of gradients (HOG), color naming (CN), and hue-saturation-intensity (HSV) each having 31 + 1, 11, and 3 channels respectively. Using these three color planes, we can fuse them into a single 45-channel (exclude the + 1 zero channel from HOG) image matrix ready to be processed by the following stages.

(2) **Online-learning**
This stage involves a multitude of steps all of which can be individually optimized and include many operations which can be processed on either a CPU or GPU. The online-learning involves training a regularized least squares (RLS) classifier whenever a new target is given to the tracking algorithm. There is a high initial cost to training this classifier for the first frame, but altering this classifier for each additional frame to be processed incurs a much lower cost. The adjustment of this classifier is decided by a given learning rate. This learning can be fixed or variable, but regardless the learning step computation time remains the same.

(3) **Occlusion detection**
When an object is detected as being occluded, we want to discard any back-training that might propagate and introduce inaccuracies in the detection model. To detect if a tracked object is occluded, we can measure the bounding box of the past-threshold pixels after generating the response map, and if the box has significantly exceeded the bounds of the previous frame's box for the same object, we can declare the object occluded. Once an object has been marked as occluded, we can trivially prevent the model from learning on it by setting the learning rate to 0 for that frame.

(4) **Bounding box prediction**
To create the bounding box of the tracked object, we can generate a cluster from the response map to see where the greatest response values are grouped together. The cluster center defines the center of the target, so we can construct the extents of the box by encapsulating the surrounding region of the cluster. If there are no significant clusters present in the response map, then no object is detected for that frame.

(5) **Model updating**
After an object has been successfully detected in the image frame, we need to continuously update the detection model to account for any changes in color, shape, orientation, or scale throughout the video feed. The RLS model and the features are updated and condensed at the end of each frame processing. These operations can easily be parallelized or even removed/simplified to reduce overhead outside of the core algorithm.

4 TOOLS WE WILL USE

Our implementation will be built using OpenCV 4.6.0, leveraging CUDA 11.7 or newer. OpenCV is an industry standard for general computer vision tasks and provides an exhaustive collection of tooling to build our solution. OpenCV can be compiled from source with the CUDA extension, which allows us to seamlessly integrate and execute our GPU kernels with minimal changes to the source code. Our choice for using nVidia's CUDA platform for GPU acceleration is largely based on the near-native support for it in OpenCV as well as industry preference is almost entirely biased towards nVidia's hardware. Hardware-agnostic GPU acceleration APIs are available as well, notably OpenCL or Vulkan compute, but we believe that implementing our solution in such environments will only reduce the ease of adoption as well as create unnecessary friction for interfacing with the OpenCV library. Utilizing both OpenCV and CUDA also allows us to have access to a wide array of resources and debugging techniques which may not be present with other tools.

The language we will be implementing in will be C++ targeting a minimum standard of C++17. This gives us access to modern tooling and standard library changes, whilst also allowing almost any device to compile and run our version of an algorithmic tracker.

5 EVALUATION STRATEGY

- **General Evaluation Strategy**
  Our baseline to compare with is the CPU-only version of the scale-adaptive KCF algorithm. To compare the two implementations, we will set up a testing method similar to the testing methodology used in the original KCF paper but utilizing a different dataset. The chosen bench-marking method will be the Got10K tracking data set and benchmarking framework. This framework is a common evaluation

tool used by many state-of-the-art trackers per their original papers.

- **Algorithm Performance Comparison**
  We aim to benchmark OpenCV's KCF, a deep learning tracker based on a siamese network architecture, such as SiamFC, and our implementation of scale-adaptive KCF. The OpenCV implementation is chosen since it implements a single-threaded CPU version, and this implemented is widely available and used. The KCF tracker is also regarded as one, if not the, most accurate tracking algorithms available within the OpenCV library. Since many OpenCV trackers are inter-operable and share the same interface, an arbitrary number of traditional tracking algorithms can be used.
- **Hardware Utilized**
  Each of these trackers will be evaluated on an Nvidia Xavier NX embedded system. The reasoning behind this decision is that tracking performance matters more on embedded systems (such as robotics, edge, and IOT) than it matters in data center or server domains where ample GPU power is available for the deep learning trackers. While the Xavier NX still has a GPU, it is not state-of-the-art and has significant bottlenecks compared to desktop-style GPUs. The Xavier NX has access to 384 CUDA cores and 48 tensor cores, whilst a desktop-style GPU of the same architecture from 2017 has access to 5120 CUDA cores and 640 tensor cores. The Xavier NX has a similarly powered CPU with a quad-core ARM processor on board. The limitations of these processors will be able to adequately showcase any performance gains in applicable use cases over the sequential implementations or deep learning-based trackers.
- **Expectations and Constraints**
  We expect to achieve nearly identical detection and tracking performance when compared with the CPU-only version of scale-adaptive KCF, with negligible loss due to floating point precision limitations. We also expect to achieve significantly improved precision and accuracy when compared to the non-scale-adaptive variation of the KCF algorithm, even when tracked objects are not changing in scale throughout the tracking process, considering that our approach performs additional template matching and leverages occlusion detection steps.

6 TENTATIVE TIMELINE

We intend to begin the project by parallelizing the existing CPU-based implementation, while also porting from OpenCV version 2 to 4. This effort will more accurately allow us to not only measure the real benefit of transitioning the algorithm to the GPU but will also ensure iterative correctness as we restructure the algorithm to process in parallel instead of sequentially. For this milestone, we believe we can port the full algorithm by **November 1st**.

Once the algorithm is correctly running in a distributed fashion on the CPU, we'll begin writing CUDA kernels for each independent pass of the algorithm, keeping the overall structure as similar as possible. This milestone is considerably more effort and will require learning significant amounts of CUDA, so we believe this component will not be finished until **November 13th**.

After the full algorithm has been ported to a series of sequential CUDA kernels, we'll begin merging the kernels whenever possible to avoid dispatch overheads and any costs related to data marshaling. We don't believe it's practical to try and merge the full algorithm into a single kernel due to the control flow structure as well as changes in the data size throughout the different stages, but many stages such as sequential dense and Gaussian, or the four FFT passes, should be able to be merged. We expect kernel merging to be a smaller task than creating the kernels themselves, so this work should be finished by **November 20th**.

If work is completed early there are a few stretch goals which could be achieved. Since the Xavier NX is a shared memory system, optimizations could be made to operations and data structures to operate without any memory transfers. Another optimization could be creating a custom data structure, instead of the OpenCV mat, to increase locality and reduce memory copying. This data structure could also have certain operations implemented directly on itself such as feature compression.

Date	Milestone
November 1	Porting to OpenCV 4, parallel CPU-based
November 13	CUDA kernels
November 18	Progress report and prototype
November 20	Merged kernels
December 10	Final report