Fast Algorithmic Tracking: Optimizing KCF

Justin Davis & Andrew Depke

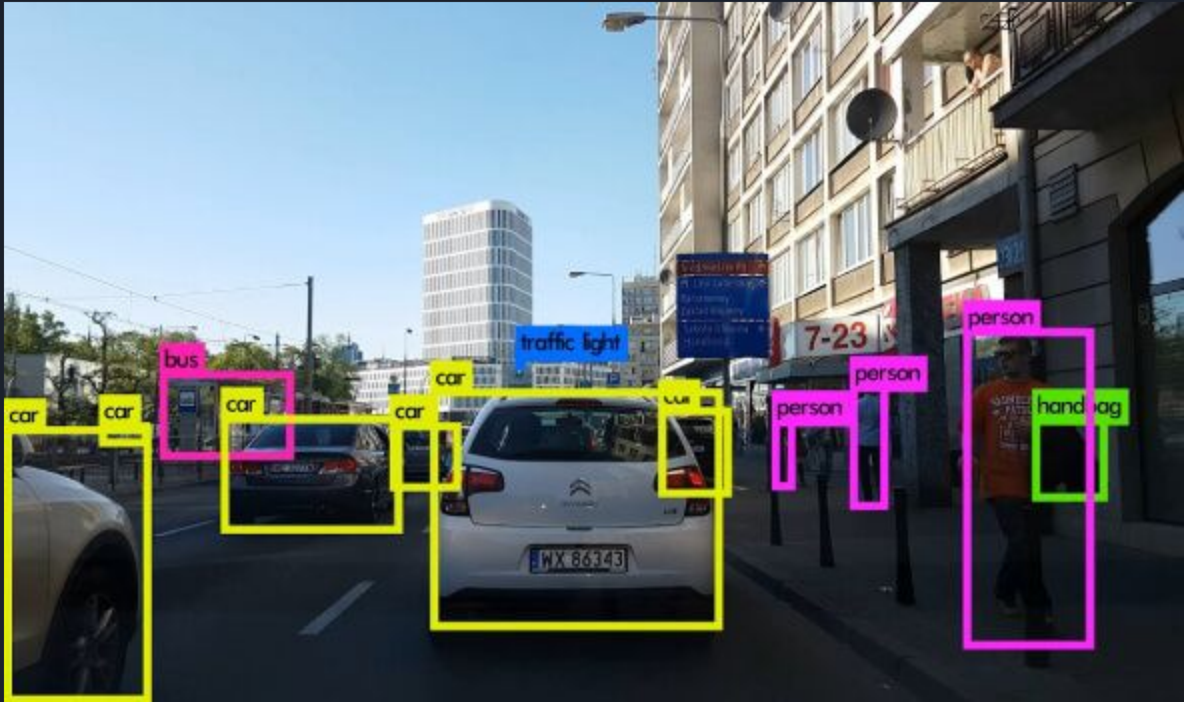
Introduction

What is object tracking?

- Creating labels and/or bounding boxes around targets of interest
- Continuous tracking of targets throughout movement

Why is it useful?

- Autonomous driving
- Doorbell notifications
- Snapchat filters



Background

Two general approaches to object tracking:

Traditional:

- Often simpler algorithms, cheaper detection and tracking
- Usual not very versatile (not multi-object, not scale or rotation invariant)

Neural networks:

- Deep convolutional networks, expensive inference for high accuracy
- Multi-object and highly versatile tracking
- Capable of higher accuracy than traditional algorithms

Motivation

Why not just use DNNs?

- Edge devices
  - Neural networks are hardware demanding
  - We might not even have a GPU, neural nets infeasible
  - Battery life?
- Low latency
  - High accuracy tracking neural nets are composed of many layers… long inference time
- Predictable tracking
  - If our targets are consistent and uniform, we don’t need the versatility of CNNs

Implementation Background

- OpenCV
  - Industry standard for computer vision tasks
  - Open source modules, allows us to create our own module and seamlessly integrate
  - Provides built-in tooling for working with hardware acceleration
- CUDA
  - Nvidia’s platform for hardware acceleration on the GPU
  - Closely integrated with host C++ code, straight forward to write and deploy kernels
- OpenMP
  - Powerful CPU multithreading with minimal work from the user
  - Mark up loops and let the framework’s scheduling handle threading for us

Our Approach

Various implementations of the KCF algorithm:

Baseline

Baseline + OpenCL FFTs (OpenCV built-in)

Baseline + CUDA

Baseline + OpenMP

Baseline + OpenMP + FFTW Just-in-time FFTs

Baseline + CUDA + OpenMP

Optimizations

= CPU Multithreaded

= GPU Accelerated

Example Optimization

Time, IPC, BranchMiss, CacheMiss

```
                                  /* Convert BGR to ColorNames
                                   */
 0.02%  0.00               @11    void FastTracker::extractCN(Mat patch_data, Mat & cnFeatures) const {
                           @6       Vec3b & pixel = patch_data.at<Vec3b>(0,0);
                                    unsigned index;

                           @7       if(cnFeatures.type() != CV_32FC(10))
                           @22        cnFeatures = Mat::zeros(patch_data.rows,patch_data.cols,CV_32FC(10));

 0.04%  0.00               @9       for(int i=0;i<patch_data.rows;i++){
 1.58%  1.58   7.6%   3.9% @7         for(int j=0;j<patch_data.cols;j++){
 3.09%  1.07          7.2% @11          pixel=patch_data.at<Vec3b>(i,j);
41.42%  0.44          100% @43          index=(unsigned)(floor((float)pixel[2]/8)+32*floor((float)pixel[1]/8)+32*32*floor((float)pixel[0]/8));

                                        //copy the values
15.81%  1.44   100%  41.2% @5           for(int _k=0;_k<10;_k++){
38.05%  1.33         92.8% @25            cnFeatures.at<Vec<float,10> >(i,j)[_k]=ColorNames[index][_k];
                                        }
                                      }
                                    }

        0.00               @9     }
```

Example Optimization

Time, IPC, BranchMiss, CacheMiss

```
                                665         if(cnFeatures.type() != CV_32FC(10))
                                666           cnFeatures = Mat::zeros(patch_data.rows,patch_data.cols,CV_32FC(10));
                                667
                                668         // perform the loops in parallel using OpenMP
                                669         // and collapsed the loops to avoid overhead
                                670
                                671         // for loop variables
                                672         const int batch_size = 2;
                                673         const int patch_area = patch_data.rows*patch_data.cols;
 0.01%                    0.5%  674  @22    #pragma omp parallel for
                                675  @30    for(int idx=0;idx<patch_area;idx+=batch_size){
 1.48%  1.68  100%        0.5%  676  @9       for(int offset=0; offset<batch_size; offset++){
 0.72%  1.03  14.7%       0.2%  677  @10        if(idx+offset >= patch_area) break;
 0.54%  0.27                    678  @4         int h = idx+offset;
10.81%  0.80             10.0%  679  @7         int i = h/patch_data.cols;
 3.34%  0.12              4.8%  680  @7         int j = h%patch_data.cols;
 1.85%  0.62              3.2%  681  @12        Vec3b pixel = patch_data.at<Vec3b>(i,j);
36.18%  0.80              100%  682  @43        unsigned index=(unsigned)(floor((float)pixel[2]/8)+32*floor((float)pixel[1]/8)+32*32*floor((float)pixel[0]/8));
                                683             // for(int k=0;k<10;k++)
                                684             //    cnFeatures.at<Vec<float,10> >(i,j)[k] = ColorNames[index][k];
                                685             //auto t = cnFeatures.at<Vec<float, 10> >(i, j);
 1.03%  0.00              0.6%  686  @3         Vec<float, 10> t;
 4.01%  1.00             16.7%  687  @15        t[0] = ColorNames[index][0];
 3.03%  1.58              8.1%  688  @15        t[1] = ColorNames[index][1];
 3.22%  1.87              7.7%  689  @15        t[2] = ColorNames[index][2];
 3.84%  1.57              7.7%  690  @15        t[3] = ColorNames[index][3];
 4.10%  1.38              7.9%  691  @15        t[4] = ColorNames[index][4];
 3.91%  1.36              5.1%  692  @15        t[5] = ColorNames[index][5];
 3.51%  1.64              3.4%  693  @15        t[6] = ColorNames[index][6];
 3.81%  1.83              2.3%  694  @15        t[7] = ColorNames[index][7];
 3.73%  1.63              1.6%  695  @15        t[8] = ColorNames[index][8];
 4.25%  1.47              2.0%  696  @15        t[9] = ColorNames[index][9];
 6.64%  0.98              3.6%  697  @17        cnFeatures.at<Vec<float,10> >(i,j) = t;
                                698           }
```

Evaluation

We ran our models on a collection of videos and sampled the following metrics:

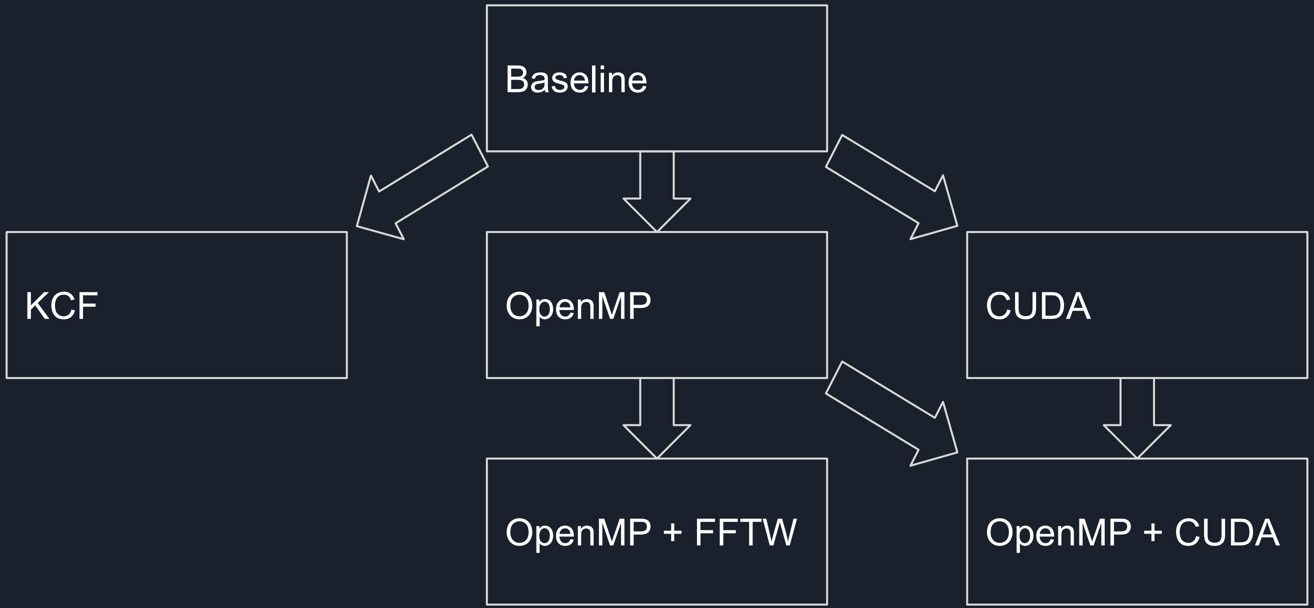
- Initialization time (ms)
- Tracking time (per frame) (ms)

Used special tracking dataset called Got10K

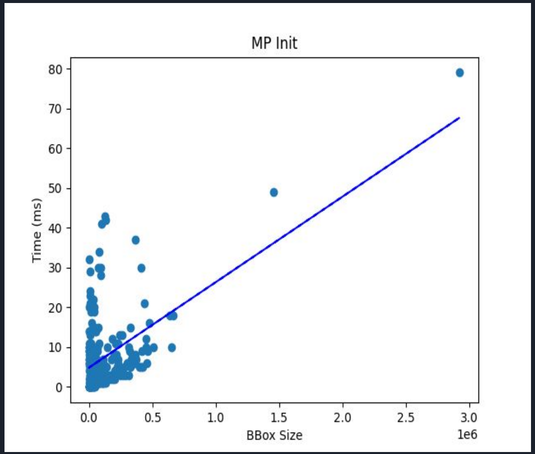
This was run on the following hardware:

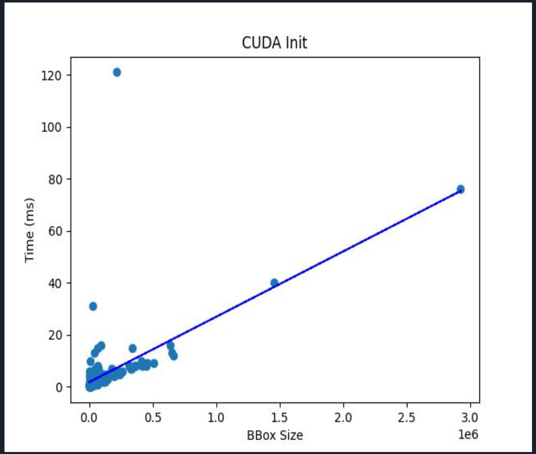
- CPU: i7-7700k @ 4.5GHz
- RAM: 16GB 3600MHz
- GPU: Nvidia GTX 1080
- OS: Ubuntu 20.04 LTS
- All standard programs still open such as Spotify, Chrome, Slack, etc..
    - (which accounts for some variance)

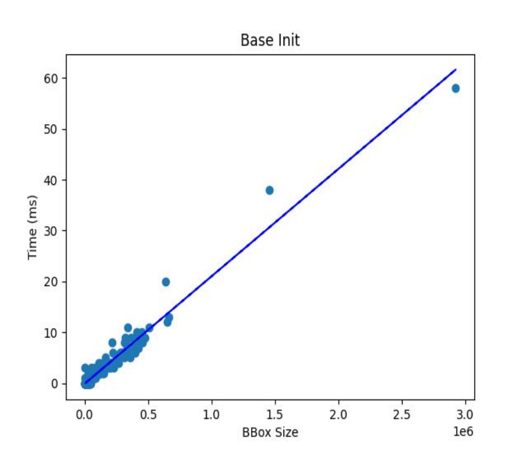
Class Hierarchy

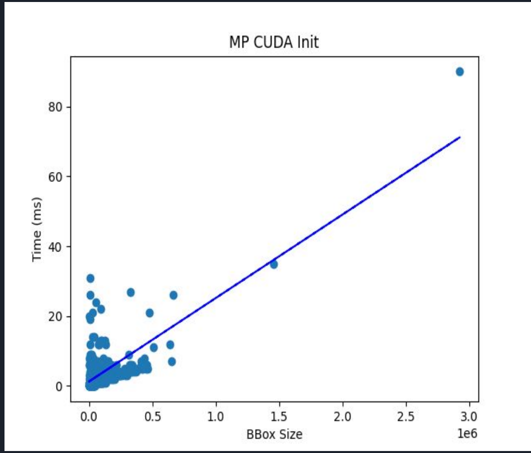


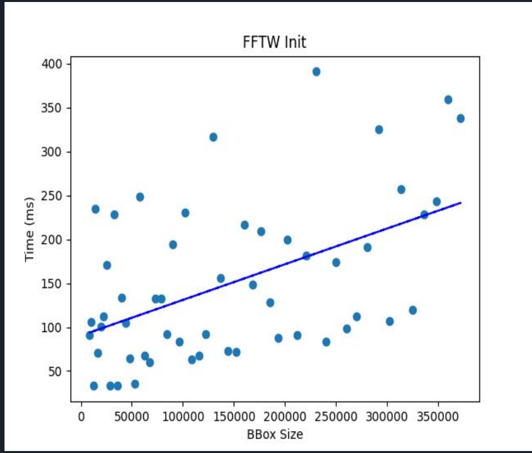
Initialization Results



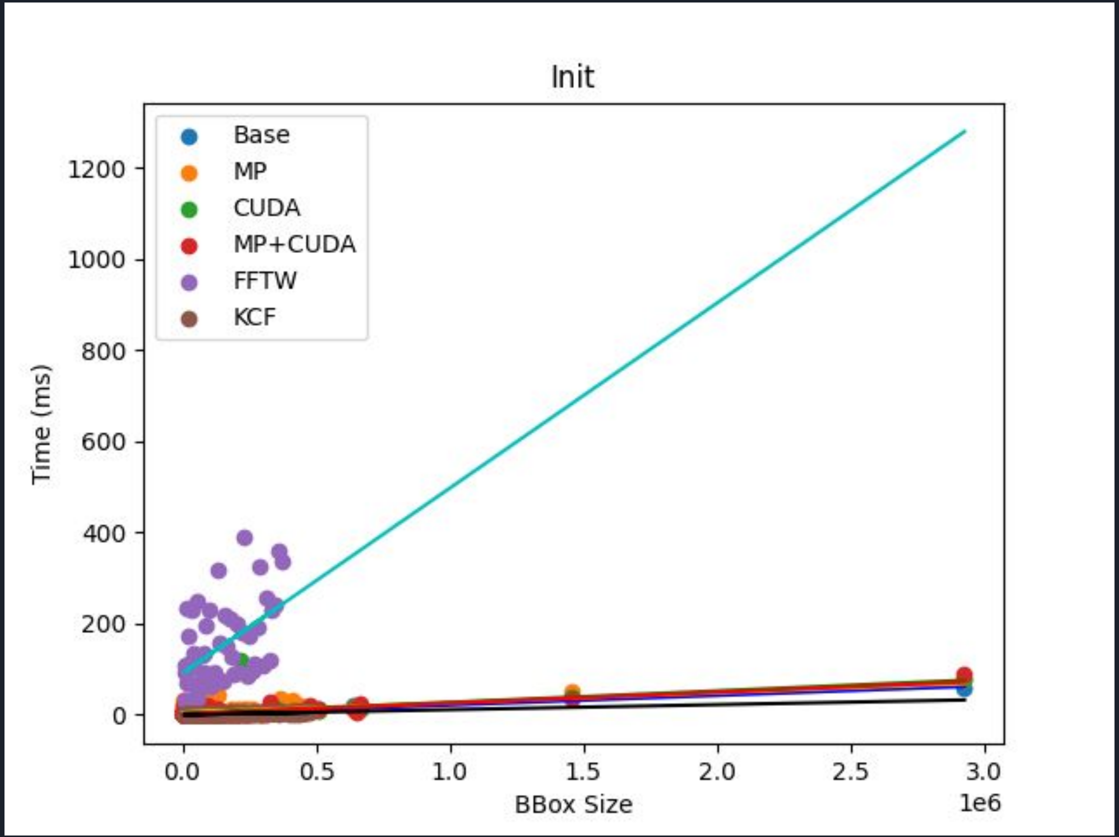




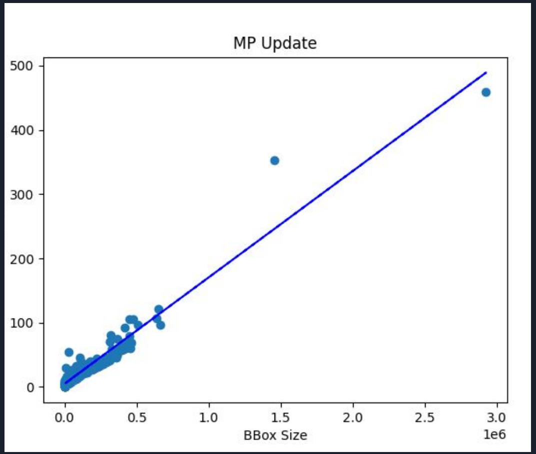


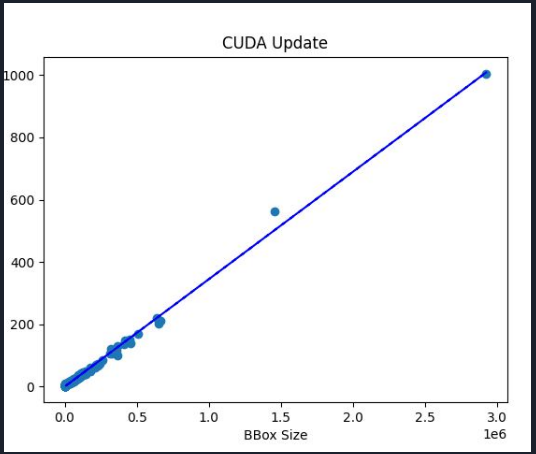


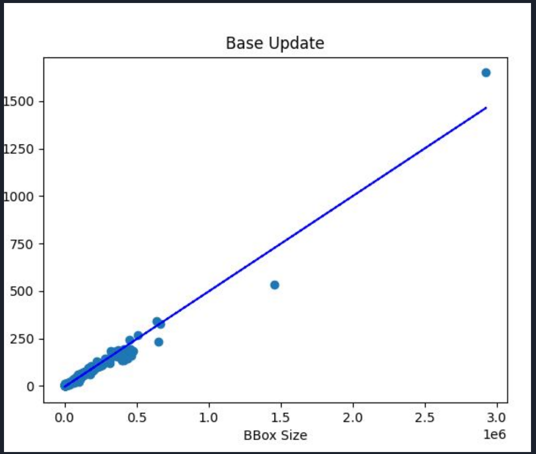
Aggregated Initialization Results

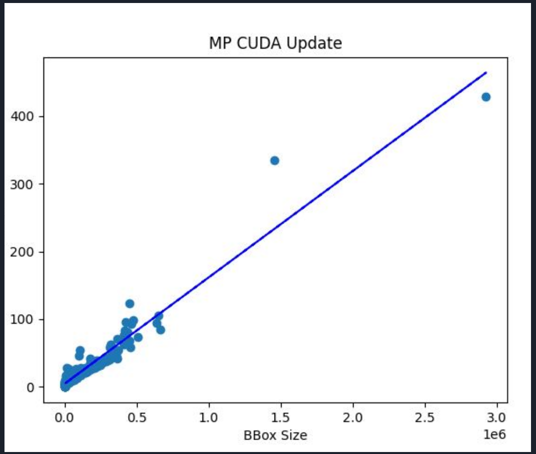


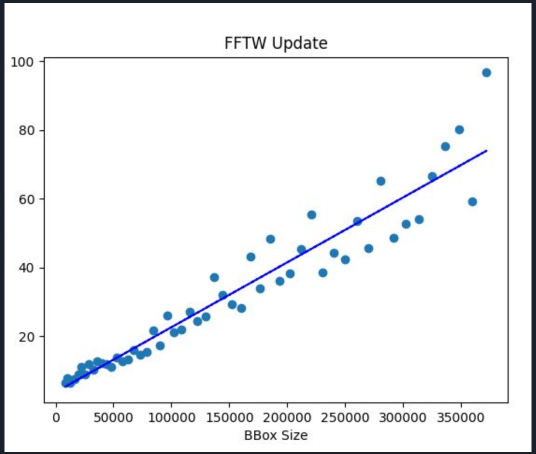
Update Results



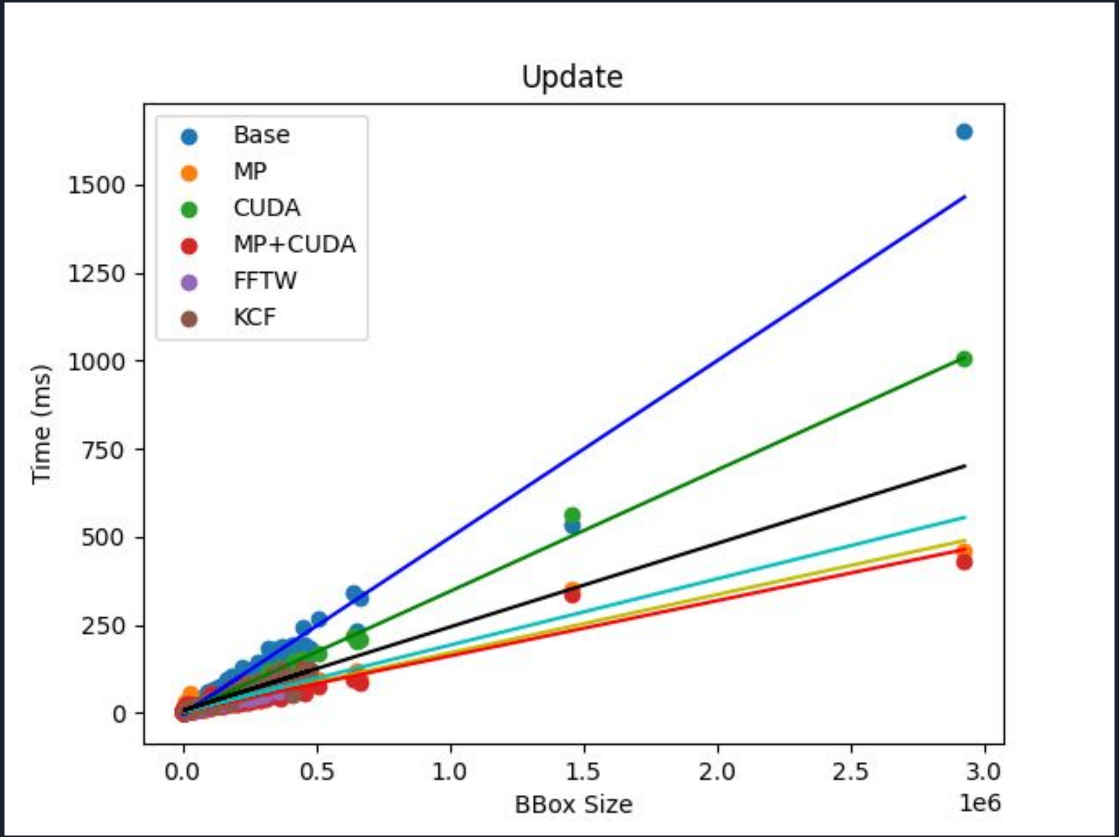








Aggregated Update Results



Tabular Results

	Baseline	MP	CUDA	MP + FFTW	MP + CUDA
Init Speedup	0.42	0.42	0.30	**0.01**	**0.65**
Update Speedup	**0.67**	**1.59**	0.88	1.29	1.58
Max Update Speedup	**1.19**	2.06	1.44	1.65	**2.35**
Min Update Speedup	**0.39**	**0.76**	0.47	0.72	0.54

Conclusion

Takeaways

1. 5 Different tracker versions were made outside of OpenCV
2. 2 Trackers showed promising optimization results compared to the OpenCV optimized version
3. Performance increases significantly came from multithreading and CPU optimizations rather than GPU leveraging

Thank you!