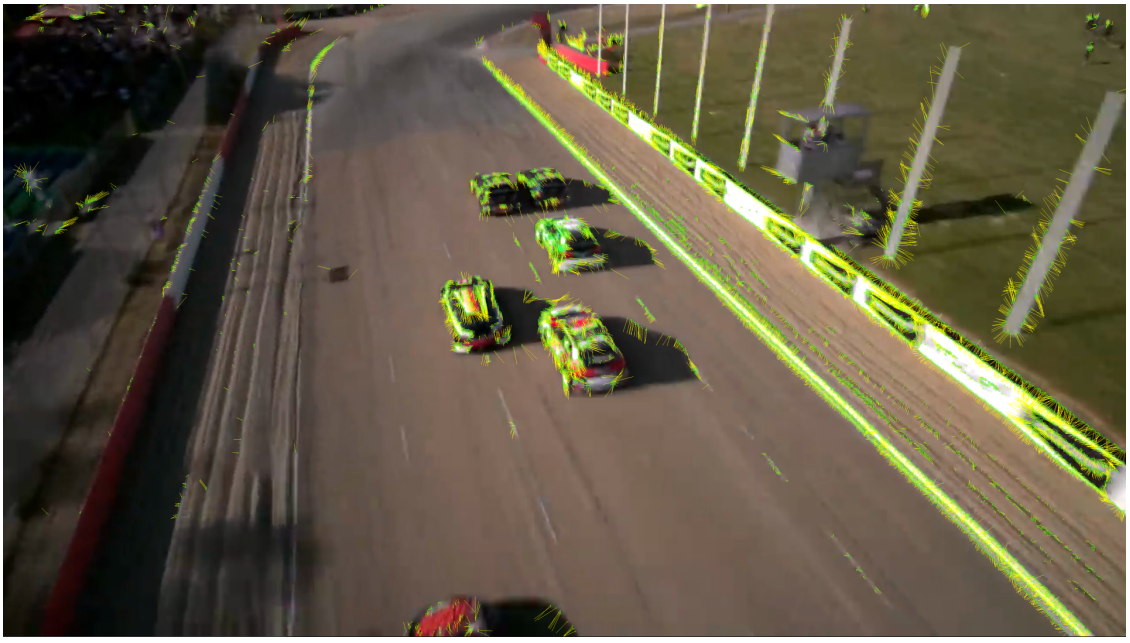
15-418 Final Report

Jessica Dai, Charlotte Wang
jdai2@andrew.cmu.edu, crw2@andrew.cmu.edu
May 6, 2024

1 Summary

We implemented a parallel Kanade-Lucas-Tomasi (KLT) feature tracker on CPU using OpenMP with Morevac and Harris corner detection algorithms. We compared the results of parallelism on a variety of high quality videos.



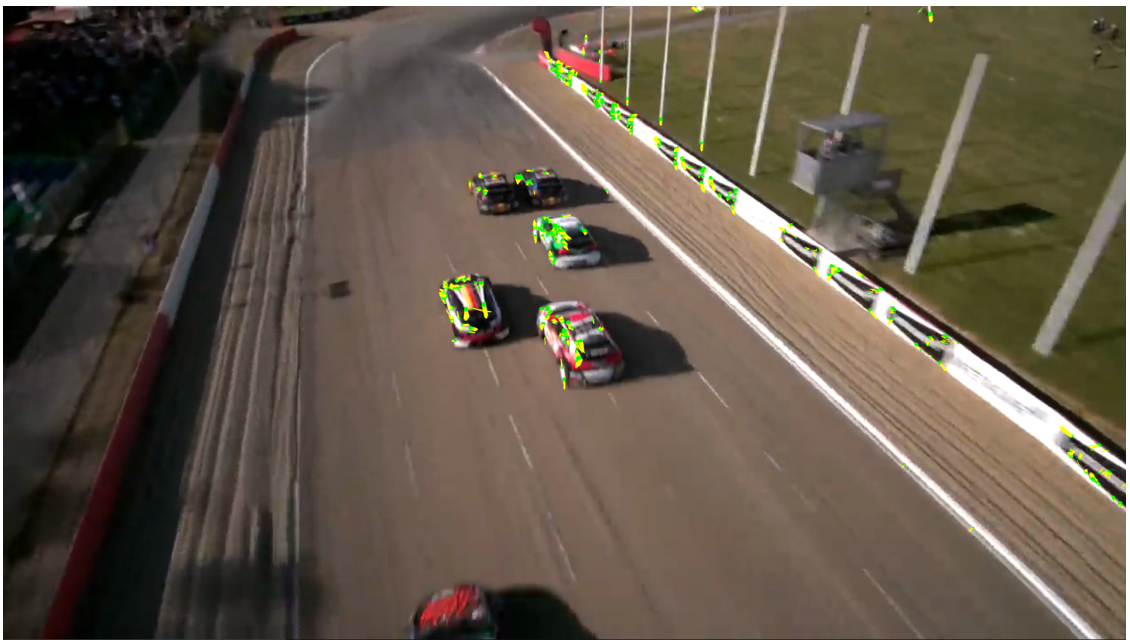


Figure 1: Example Optical Flow Result

2 Background

2.1 What is Optical Flow?

Our project is focused on computing **optical flow**. Optical flow is an umbrella term in computer vision for algorithms that describe the movement of pixels in a series of images(i.e. a video). It comes in 2 varieties, dense and sparse. For dense, this movement is calculated at every point of the image. For sparse, the movement is calculated on a set of "interesting points". This project implements **sparse** optical flow.

This is an important operation to parallelize because the goal is to have **sparse optical flow in real time**. We parallelized a tracking algorithm with that as the goal, and strategized accordingly. This has several implications on the **limits** and ways we should parallelize.

- No Parallelization across frames(since we won't have future frames in real time)
- Extremely short time frames. A real-time video of even 20 fps means we need to process each in less than 50ms ideally. This puts strains on how overhead the strategies can have.

The reason why we chose sparse was because it has a more interesting workload. Dense means that there is the exact same operation on every single pixel, so we decided the workload wasn't as interesting as sparse, where the **set of interesting points can vary in cardinality, spatial concentration**...etc. Furthermore, the list of interesting points is also another sub-part of the algorithm that can be parallelized. Thus, the big picture of our algorithm looks like:

For each previous frame(`prevImg`) and current frame(`curImg`)

1. Find a set of "interesting points" in `prevImg`
2. Find the velocities of the "interesting points" from `prevImg` to `curImg`

Jessica Dai, Charlotte Wang
jessicad1@andrew.cmu.edu, crw2@andrew.cmu.edu

Specifically, we implemented the **Kanade-Lucas-Tomasi (KLT) feature tracker.** This means the set of interesting points is found by a corner detection algorithm(we tried 2). Further, it makes an assumption that the velocities of points close to each other should be the same. This lets us find the velocities with a minimal amount of linear algebra.

2.2 Finding Interesting Points AKA Corner Detection

Corner detection is used to detect corners in an image in order for the KLT algorithm to track. We implemented two main methods for corner detection: Moravec's algorithm and Harris' algorithm.

For both, the inputs were an image (openCV float matrix and vector float form), an area of pixels defined in terms of integers "xarea" and "yarea", an integer threshold value, and a boolean "verbose" for printing out resulting information.

The result was a list of corners of vector pair form.

The main data structures involved are the data structure for the image (openCV matrix, float vector) and corners list (pair vector). The key operations performed on the image is reading from it multiple times and calculating values based on the entries read. These values are necessary for determining if a certain region is a corner.

The part that is computationally expensive for corner detection is doing the computations on the pixels in the neighboring regions and in the area. The gradient computation for Harris' detection can be very costly since it uses derivative filters (Sobel filters) and convolves the image with these filters. The structure tensor computation, SSD, and Harris' response function are all computationally expensive.

The parallelization for corner detection is across the areas of pixels we are testing over. Since each area's computations are independent of another, we can parallelize over this work.

2.2.1 Moravec's Corner Detection

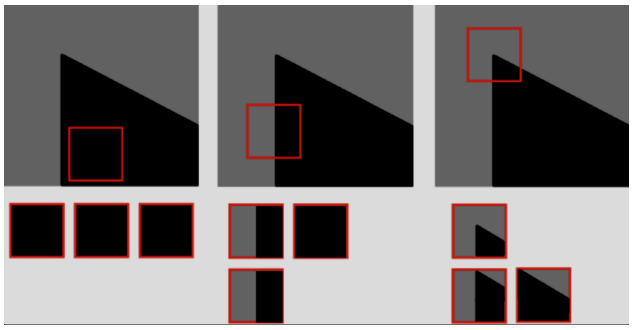


Figure 2: Moravec's Principle: (Left to Right) small movements of the sub-image (highlighted in red) don't result in any alteration; when examining a contour, alterations are noticeable solely in one direction; whereas at a corner, notable changes occur in every direction. [1]

Moravec's corner detection algorithm is one of the earliest corner detection algorithms and defines a corner as a point with low self-similarity. The algorithm examines areas of pixels within the image to detect potential corners. It does this by assessing the similarity of an area of pixels with neighboring patches (in all four directions) that overlap significantly. See Figure 1. This comparison gauges similarity through the sum of squared differences (SSD) between the corresponding pixels

of these patches. A reduced SSD value signifies greater similarity between patches. For each area, we then test to see if it is a corner if it is a local maxima. We do this by taking the minimum SSD of the four neighboring areas (directions) and then checking if this value is greater than the threshold value. If it is, than we have found a corner.

2.2.2 Harris' Corner Detection

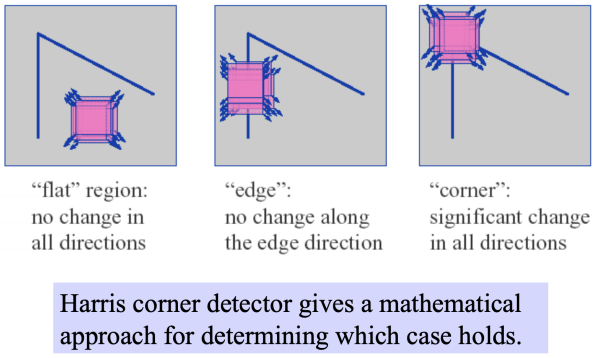


Figure 3: Harris' algorithm compares in all directions rather than only the four for Moravac's [2]

Harris' algorithm provides more accurate corner detection than Morevac's. Harris first smooths the image with a Gaussian filter and then calculates the gradients of the image using a derivative filter (Sobel filter) in both the horizontal and vertical directions. It then computes the covariance matrix using these gradients. Finally, the Harris response function calculates the eigenvalues: the determinant of the area minus the weighted square of the trace. Then, eigenvalues with high Harris response values (larger than threshold) are considered corners.

2.3 Finding Velocities - Lucas-Kanade

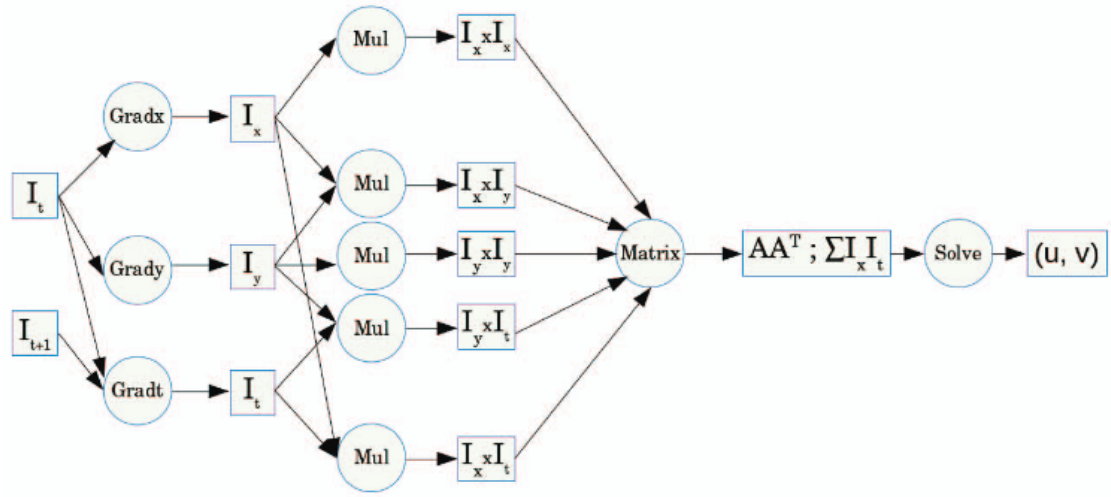
The algorithm for Lucas-Kanade assumes approximate constant velocity v_x and v_y over a small area of the image. Thus, given the Intensity of the current frame as I and next frame as H, we have:

$$H(x, y) = I(x + u, y + v).$$

At **EACH POINT OF INTEREST**, the velocity can thus can estimated by the solution to the following equation. The derivation is via the first order Taylor approximation of the previous equation.

$$\begin{bmatrix} \sum I^2 x & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} \sum I_x I_t \\ \sum I_y I_t \end{bmatrix}$$

The sum in the equation above is over a square surround the point being processed, the size of this square is a parameter for this equation, we'll call this k.
I_x, I_y and I_t are the derivatives at the pixel for the direction x, y and t(time). These gradients can be calculated via the Sobel filter centered at the pixel.
The equation is 2×2, thus it can be solved in closed form with Kramer's method.



The **pseudo code** for processing a SINGLE POINT is the following:

- For each point inside the k by k square surround the pixel:
  - Calculate I_x, I_y, I_t via the sobel filter (convolution)
  - Calculate I_x^2, I_y^2, $I_x y$, $I_x I_t$, $I y I_t$(all arithmetic).
- Sum the quantities in the second line above over the k by k square
- Construct the matrix equation above and solve it
- Save the velocity

We found **2 main approaches to parallelism** here, and explored both approaches with results in the following sections:

1. Parallel over points of interest
2. Parallel over k by k square surrounding pixel

3 Approach

Our approach toward parallelizing KLT is **openMP on CPU**. Because Gates machines did not have openCV installed(which we used for dataloading and visualization purposes), all experiments were run on our own devices, which is a `11th Gen Intel(R) Core(TM) i7-11800H` with **8 cores and 16 cpus** (via hyperthreading).

This approach was mainly chosen for its novelty [3]. There are many public instances of available code for KLT with CUDA on GPU. It is in fact part of the CUDA vision library. Thus, we decided to not re-create the wheel, and explore a CPU-based approach for parallelizing KLT. Although CPUs are not as commonly used for such tasks, it is still a useful endeavor, as devices running such an algo in real time may not always have a GPU available to them.

Our appraoch varied over time(as seen in the section below), but the common approach is that rather than parallelizing each step of corner detection/Lucas Kanade seperatly, we tried to re-write both as a **series of operations surround a single pixel/corner**, and parallelizing over that instead.

This approach is quite far removed from the usual way of writing computer vision algorithms, which is calling a series of general vectorized operations from computer vision libraries / matrix libraries such as `opencv` and `numpy`. Not only would that be much easier to optimized(just replace each convolve with a for loop), it would also not yield as good result because it will have **low arithmetic intensity** relative to the amount of data access needed. Our approach will make sure that all operations surround a single pixel/neighborhood is processed in one go, thus decreasing the amount of memory access there is in the algo, and providing higher levels of parallelism.

3.1 Iterations

Many of the following sections will contain speedup graphs. The speedup is split into three portions, LK speedup, corners speedup(whichever algo used), and total time speedup(including init time).

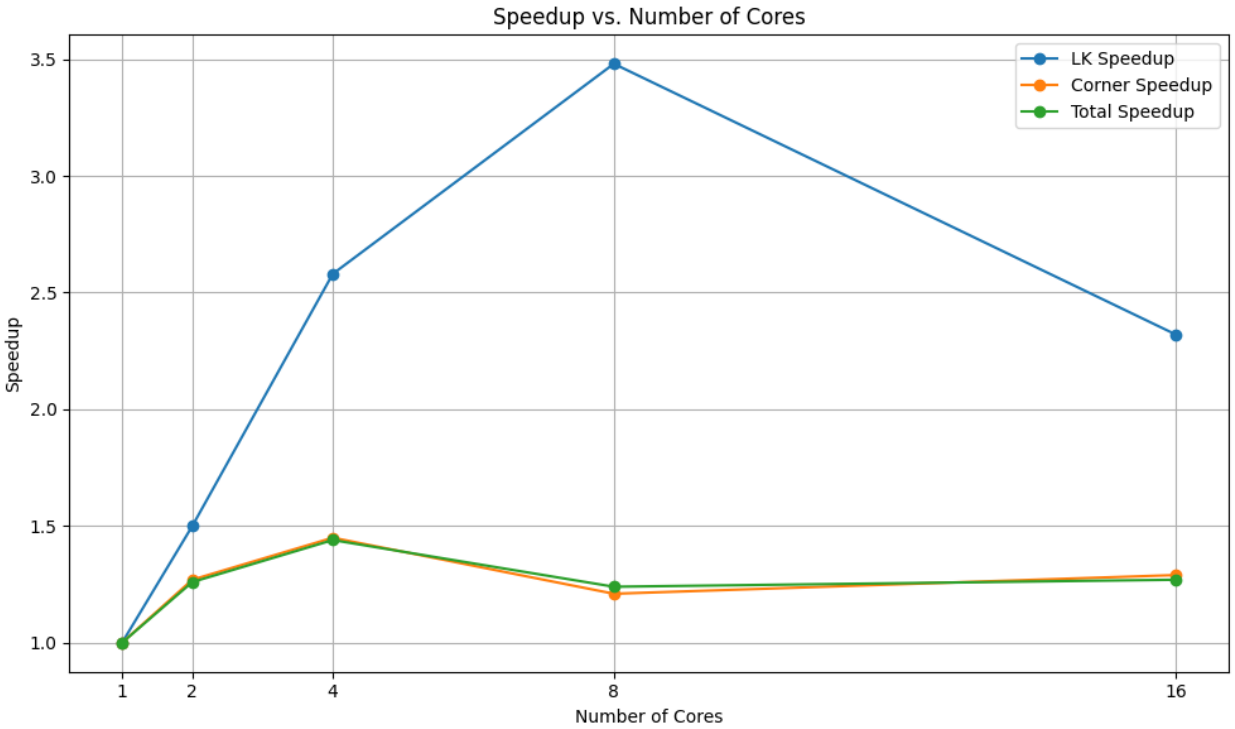
3.1.1 Initial Approach: Sequential Algorithm

Given the difficulty of implementing a computer vision algorithm with many steps from scratch(and more importantly no opencv function as a crutch), we started off with the most "from scratch" C++ implementation of KLT we could locate on the internet [4]. However, this approach came with several issues:

- Locating such code took a significant amount of time. Almost everyone(including the code we started with), relied on libraries for a good part of the implementation. We didn't want this since we wished to be able to parallelize and edit the functions.
- The code we located, as we found out later, is actually not only **inefficient**, but also **incorrect**, and we spend a good amount of time fixing it. The code used Moravec's method for corner detection albeit incorrectly (did not properly compute SSD and reduced values over 2 dimensions instead of 4).

Table 1: Speedup Over 1 Core

NumThreads	LK Time	LK Speed	Corner Time	Corner Speed	Total Time	Total Speed
1	284	1	2562	1	2984	1
2	188.67	1.50	2019.33	1.27	2376.07	1.26
4	109.73	2.58	1772.00	1.45	2065.40	1.44
8	81.53	3.48	2112.33	1.21	2404.33	1.24
16	122	2.32	1982	1.29	2344	1.27



Above is the speedup graph from a naive parallelization of our initial code. A `#pragma omp for` was simply added over the for loop going over the x-axis in corner detection and the for loop going over all corners in lk.

We identified 2 main reasons for its lackluster speedup and performance:

- Excessive Memory Allocation - The code started with had several unneeded memory allocation that made the code slightly clearer, but perform much much worse.
- Overhead of using `cv::Mat` instead of native C++ data structure. It has several nice properties, like being thread-safe. However, we are really using the image for read access only, thus these overheads is just wasting time.

After we removed these problem, we found that corner detection and lucas-kanade processed much faster and parallelized much better.

3.1.2 Fix 1: Does Batching Help? Hmm... Not really

As per usual before we do any useful work, let's play around with **openMP schedules**. We ran the algorithm on the following settings for both LK and Corners(the two are in 2 separate for loops):

- static, 1
- static, 16
- static, 32
- dynamic, 32

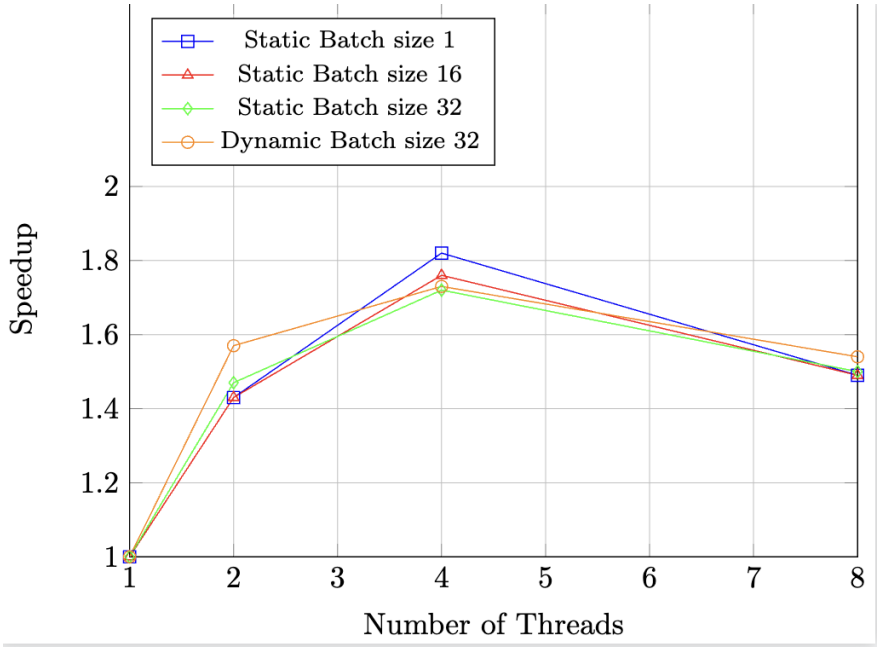


Figure 4: Thread Number vs Average Corner Detection Speedup

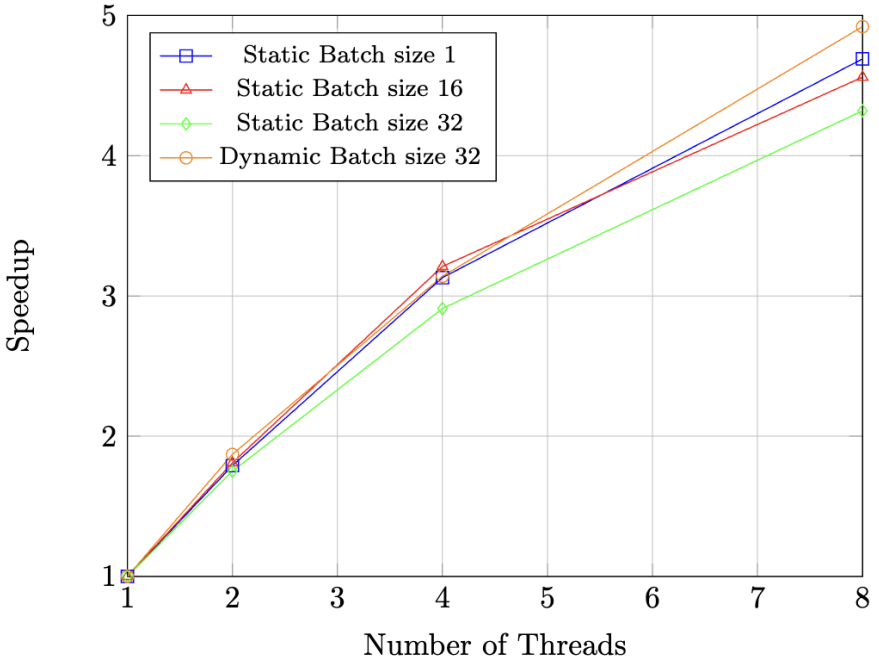


Figure 5: Thread Number vs KLT Speedup

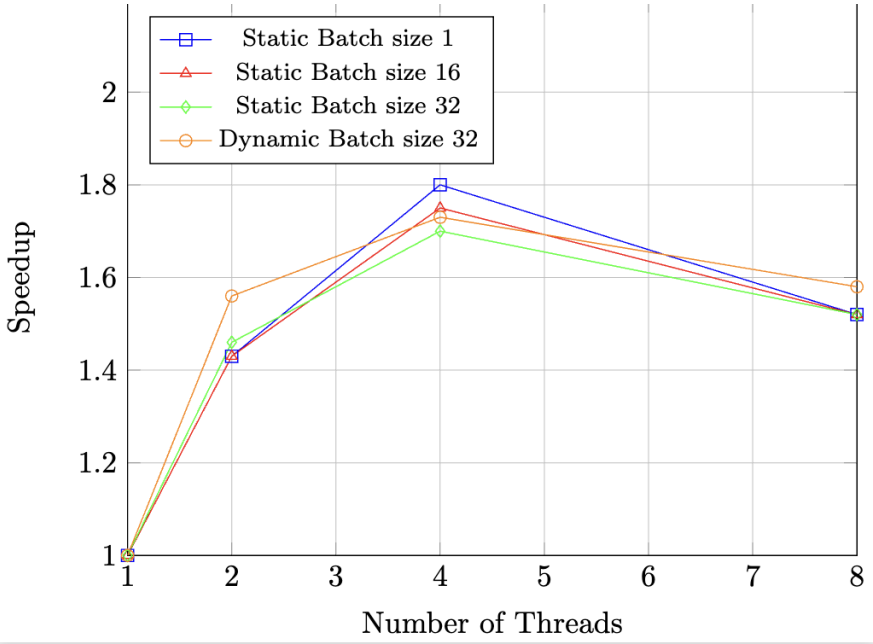


Figure 6: Thread Number vs Total Time Speedup

We see here that increasing batching size does not really help with increasing speedup for detecting corners, Lucas Kanade, or overall time. We see very similar speedups among all the batches. This is most likely due to the fact that in this implementation, the unnecessary memory allocations and overhead overshadows what benefits batching can give.

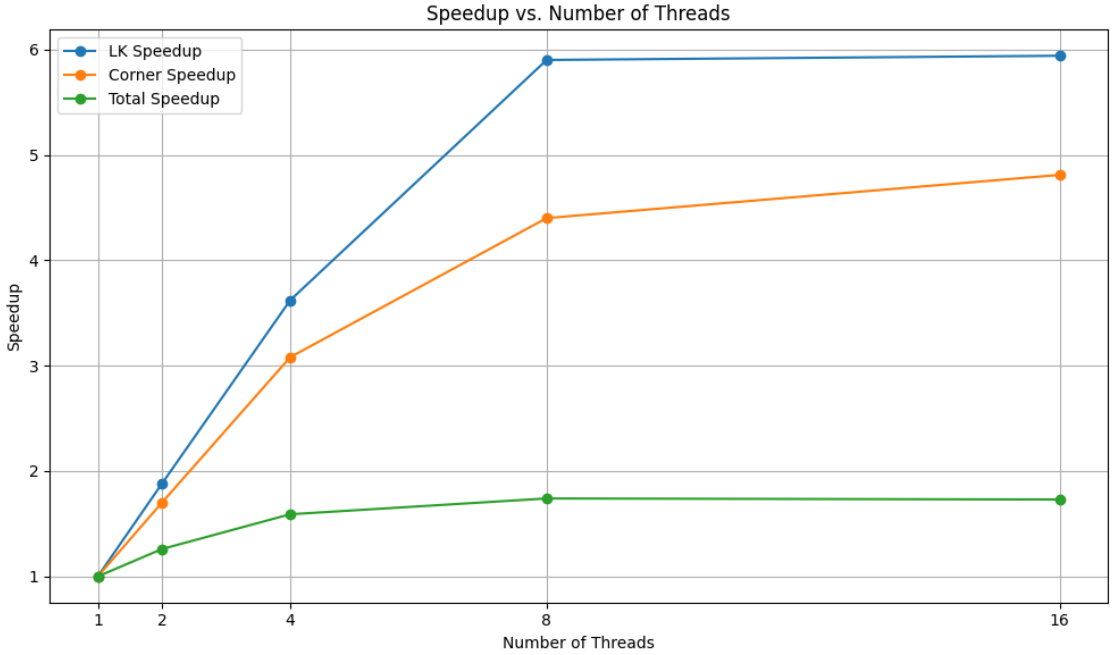
Unrelated to batching, one thing we can see here is how overhead for Corner Detection severely impacts performance (and total time performance). As described earlier, this is likely due to the significant overhead of copying matrices leading to excessive memory allocation that is actually unnecessary (we will fix this issue).

3.1.3 Fix 2: Change Image to `vector<vector<float>>`

Given that the code was too wasteful of computation to optimized property, we manage to remove all the unnecessary memory allocations and changed out image data type from `cv::Mat` to `vector<vector<float>>` in an attempt to better the locality of our program. This shall serve as our primary base for optimization for the following few iterations.

Table 2: Speedup Over 1 Core

#Threads	LK Time	Corner Time	Total Time	LK Speed	Corner Speed	Total Speed	Cache Miss
1	121.27	37.53	312.27	1.00	1.00	1.00	55.65
2	64.53	22.00	247.80	1.88	1.70	1.26	50.47
4	33.53	12.20	196.40	3.62	3.08	1.59	43.43
8	20.53	8.53	179.47	5.90	4.40	1.74	42.14
16	20.40	7.80	180.67	5.94	4.81	1.73	34.76



As we can see in the performance data above. Both parts of the algorithm is now running much faster than before. Further, for both LK and Corner, the **speedup is now monotonically increasing**. Moreover, the corner detection is now faster than the LK for once, meaning that our main algorithm is actually taking more our the time for once. The reason for performance with the extra memory allocations is likely that those memory allocations were all copying sections of the images over, and with multiple threads sharing a single image object, this meant that there was high contention for it, leading to low speedup of our first attempt.

However, we still see **bad speedup in terms of total time**, as it is still below 2 with 16 threads. This is mostly because it takes a siginificant amount of time(100ms ish) for openCV to open and load in a PNG image. As the focus of our project is parallelizing the algorithm, and not parallelizing data-loading(which is interesting in its own right), we will mostly focus on the optimization of Corner and LK time. However, for real implementation, this is an important notice that data loading also needs speedup for scaling.

3.1.4 Adding Harris Corner Detection

At this stage, we also thought it would be interesting to add a different corner detection algorithm, the Harris' Corner Detector, which is more widely used in computer vision. See the Background section for more details.

Harris' corner detector is much more computationally intensive than Moravec's due to the fact that it checks pixels in every direction rather than only four. Indeed, Harris checks 25 more pixels for every pixel that Moravec's checks with.

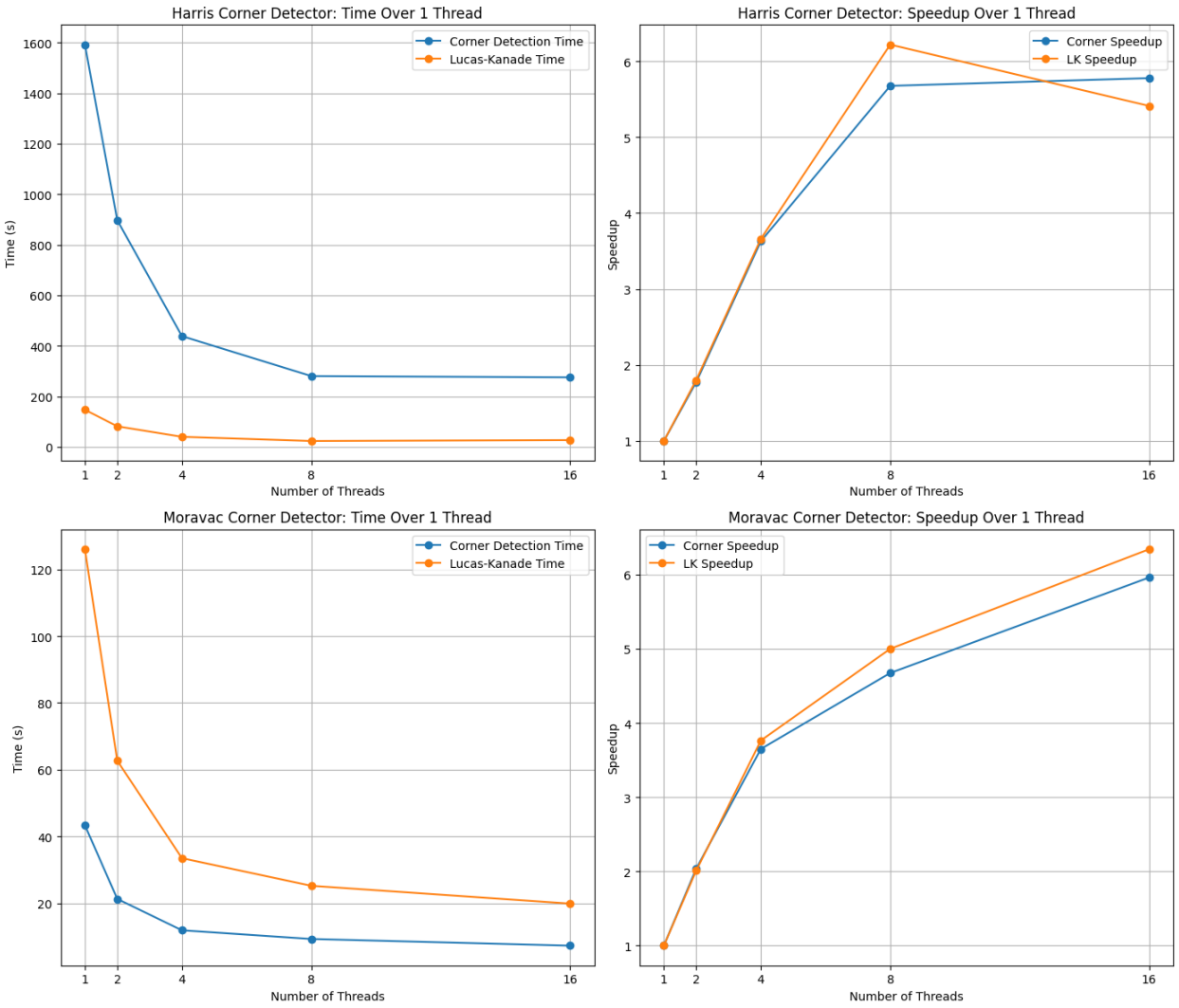
The final results and comparison of the 2 corner detection algorithms look like the following:

Table 3: Harris Corner Detector: Time and Speedup Over 1 Thread

# Threads	Corner Time (s)	LK Time (s)	Total Time (s)	Corner Speedup	LK Speedup
1	1591.87	147.20	1739.07	1.00	1.00
2	897.13	81.73	978.86	1.78	1.80
4	438.53	40.20	478.73	3.63	3.66
8	280.47	23.67	304.13	5.67	6.23
16	275.53	27.20	302.73	5.77	5.41

Table 4: Moravac Corner Detector: Time and Speedup Over 1 Thread

# Threads	Corner Time (s)	LK Time (s)	Total Time (s)	Corner Speedup	LK Speedup
1	43.33	126.00	169.33	1.00	1.00
2	21.27	62.67	83.94	2.04	2.01
4	11.87	33.47	45.34	3.65	3.77
8	9.27	25.20	34.47	4.68	5.00
16	7.27	19.87	27.14	5.96	6.34



Analysis:

From the graphs above, we see how **Harris' algorithm takes a significant amount of time**. Harris' algorithm is much more computationally intensive and thus takes more time than LK and Moravec's. The overall speedups given from Harris' and Moravec's are relatively similar but the computational time is significantly different.

For Harris' we see a dip in speedup between 8 threads and 16 threads. We do not see this issue for Moravec's. This is likely do to the fact that we do not have true hyper-threading and there is also higher contention for Harris'. For Harris' we have more contention due to more memory accesses/reads. As mentioned above, Harris' has approximately 25x the number of comparisons as Moravec's. Thus, this contention hurts speedup when there are many threads.

3.1.5 All under One `#pragma omp` Umbrella

We wondered if memory access would slightly improve if we put both the corner detector and the ensuing flow calculation under a single for loop. Instead of:

- Iterate over all pixels, pick out corner pixels
- Iterate over all corners, calculate flow

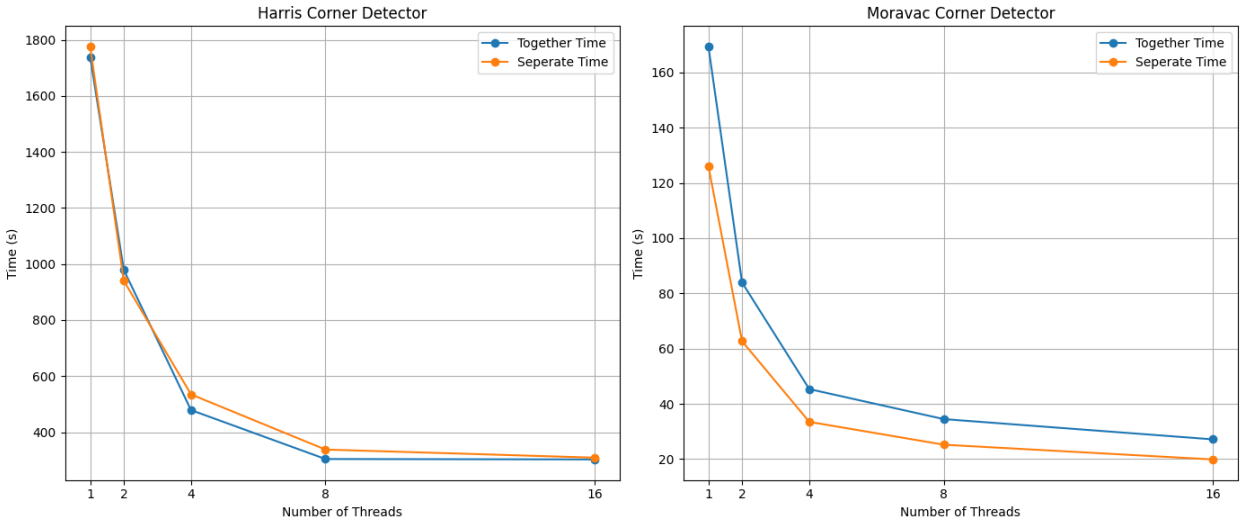
We do:

- Iterate over all pixels, check if corner, calculate flow if corner

We implemented a "together" for both Morevac's and Harris' corner detection algorithms. Below are the results of running our experiments and a comparison with the separate versions.

Table 5: Lucas-Kanade Time and Speedup Over 1 Core

Number of Threads	Harris LK Time (s)	Moravac LK Time (s)	Harris Speedup	Moravac Speedup
1	1775.27	182.40	1.00	1.00
2	940.20	97.47	1.89	1.87
4	534.73	61.87	3.32	2.95
8	337.87	55.47	5.25	3.29
16	309.07	42.20	5.74	4.32



We found that putting the corner detector together with the flow calculation did not actually grant us better computational times. This is likely due to workload imbalance in the cases where something is a corner and where something isn't. How our code is structured is that there is an if statement that checks if a corner is a pixel. If it is a pixel, then it will calculate flow; if not, it will not perform this calculation. Thus, when we are parallelizing over the pixels, there is an imbalance of work done per thread depending on the case it's in.

In the Harris case, we do not see much of a difference in the combined versus separate times. This is likely due to the fact that Harris corner detection is already very computationally intensive before

it reaches the if statement. Thus, the work that occurs depending on the if statement does not have much of an impact on time. Thus, we see a very similar result between separate and together.

For the Moravec case, we see that together takes longer than separate over all threads. This is likely due to the fact that Moravec is much less computationally intensive than Harris and is thus greatly impacted by the branching in the code. Thus, there is workload imbalance and thus the LK time will be worse overall.

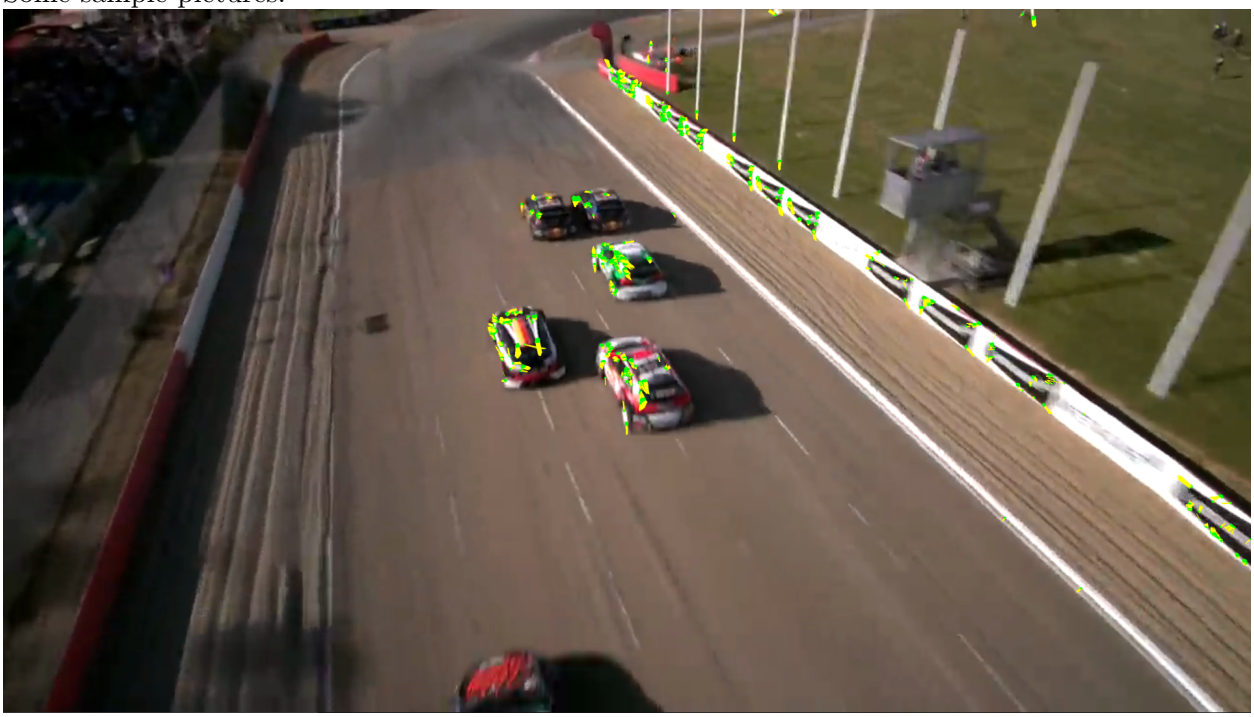
4 Results

These will be results and experiment regarding the **FINAL** iteration of our algorithm. A lot of the analysis regarding what worked and what different work and why are in the **APPROACH** section for clarity. Much of the information related to why speedup isn't perfect...etc is in that section.

Same as previous section, we measured our performance as the speedup time of LK, Corners and Total Time.

I think overall we are pretty successful in our project. The algorithm, given a proper workload, generate a reasonable level of parallelism given the short time-frame it is to run. If given a fast image loader as well, I think I real-time LK algo is well within reach(Morevac Corner + LK time total to less than 50 ms on 16 threads), which was the goal of this project.

Some sample pictures:



4.1 Experimental Setup

The following are the common denominator for the experimental setup:

- CPU(8 core, 16 cpu), as described in approach
- Size of input(for all in approach section) and also below if not explicitly mentioned: 4096 × 1023 images, 16 frames(15 pairs). All times are taken as an average of all pairs.
- The parameters for the corner detectors are in the source code. They are held constant unless otherwise specified. They were tuned by hand to result in a good amount of corners(30K) to track.

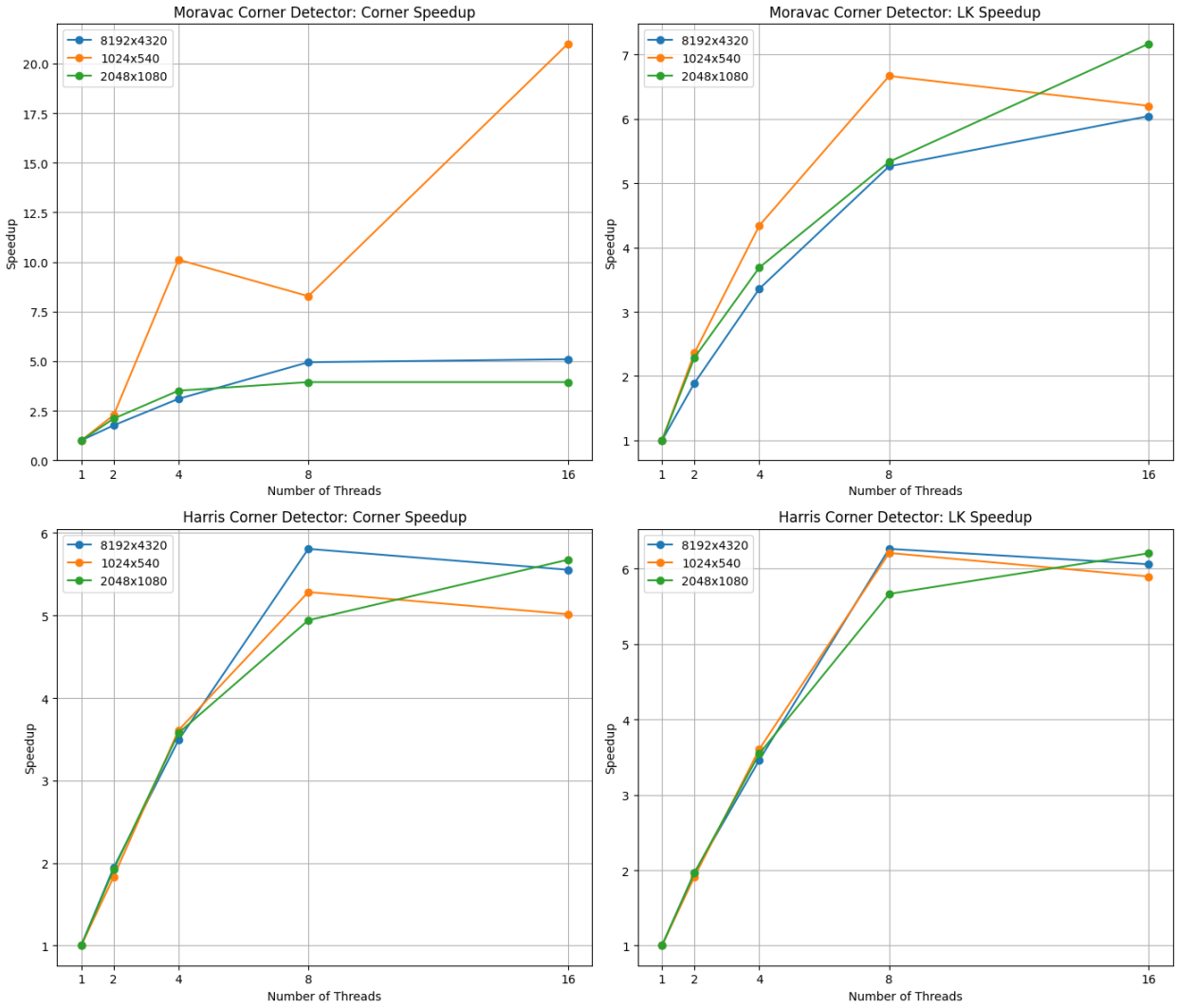
4.2 Final Speedup Graph

Refer to "Adding Harris Corner Detector" section of the Approach section for speedup graph, table and analysis. We deem that our final implementation. Further, we note that speedup is calculated against the 1-thread openMP version. We would test it against a "real" sequential version, but it doesn't currently exist since all normal c++ impl of it are vectorized library functions.

The following experiments are done on this final implementation. They are experiments in how the image size(which is almost directly indictive of workload), can change the parallelism behavior of the program.

4.3 Experiment: Image Size

This experiment is regarding the impact of the size of an image to the speedup of our algorithm. We expected that larger images will parallelize better, as there is more work to parallelize over.



The actual results raise several interesting points:

- The 2048 is consistent worse in speedup almost than the 8K picture. This makes sense as per before

Jessica Dai, Charlotte Wang
jessicad1@andrew.cmu.edu, crw2@andrew.cmu.edu

- The 1024 performance is interesting. In many cases the speedup is actual better than the 4K, in Moravac, the speedup was superlinear. We predict that this is because the image becomes small enough that it starts fitting in a cache level of a processor, meaning much faster access times than before. Further we note that the params for corner detection that is tuned on the larger images may not work great for smaller size images(since most of them define convolution windows that calculations range over). Some more visual inspection may be needed to figure params that are suitable for all.

4.4 Analysis: Why Isn't Speedup Super Super Super Perfect T-T

I think there are 2 main reasons that the speedup is not perfect in our algorithm.

- **Lower Locality than Expected** I think our algorithm could be improved on its locality more. We changed it to a 2D vector so that the locality of it could be more clear to us. However, we have run cache miss analysis on the program that see misses of up to 50 percent in some settings. This points to that perhaps the C++ double vector may not be located at the same place, and we are actually performing some kind of vector chasing. In addition, it also means that when we load one row, we load, or try to load, the entire row, when really, we want to load local squares.
  It would have been really interesting to try out more image representations, such as a 1D array(rows appended together), or may even 5 by 5 squares appended together. These are settings which would make certain convolutions much more efficient in data retrieval.

- **Low Time frame** As before, when optimized well, LK runs on a much lower amount of time, in terms of 10-100ms, for a whole image. This is in direct contrast to much of the workload we have done in this class, which are mostly in terms of much longer time-frames. This means that high overhead strategies would not work for this kind of algorithm. Thus, there is a limited amount of things we can really do.

4.4.1 Regions for Improvement

We have split up our algorithm as a corner detector and a flow calculator. Thus, all along, we can analyze exactly which one takes more time and is parallelizing much better. That being said, I think the Harris corner detection part of the algorithm could be a focus for improvement, given that it actually takes much longer than Lucas-Kanade, and we should, make the slowest part of our code run faster. I think the data locality ideas mentioned in the section above could really help with those ideas.

5 Appendix

5.1 Work Distribution

Thread 0: Charlotte Wang - 50% - LK Optimization, Experiment(Batches), Locating Starting Code Thread 1: Jessica Dai - 50% - Corner Detector Optimization, Experiments(Image), Locating Dataset

Both wrote up report equally.

5.2 References

References

[1] V. Mazet, "Corner detection," https://vincmazet.github.io/bip/detection/corners.html.

[2] C. M. University, "Harris corner detector," https://www.cs.cmu.edu/ 16385/s17/Slides, 2017, accessed: May 5, 2024.

[3] F. Liu, H. Jiang, and X. Li, "Understanding memory performance of gpgpus for large-scale image processing," *2018 IEEE International Conference on Big Data (Big Data)*, pp. 3127–3132, 2018.

[4] H. Acharya, "Opflow: A low code workflow management system for running scientific experiments," https://github.com/hrishioa/OpFlow, 2015.