
Evaluating Performance of Two Implementations of the Shi & Tomasi Feature Tracker

Michael O. McCracken

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
mike@cs.ucsd.edu

Abstract

We discuss the performance characteristics of two implementations of a feature selection and tracking algorithm most recently described by Shi & Tomasi [1], and the opportunities for improving performance using parallelism. One implementation is a demonstration prototype written in Matlab, and the other, KLT, is a publicly available library written in C. Each algorithm is profiled for bottlenecks and it is found that the Matlab version has limited opportunity for parallelism because as a prototype, it is limited in scope and does not do enough independent work. The C library KLT performs much more work, including feature selection and multiple feature tracking. It holds promise for parallelism, but no results are available because it was discovered late in the project.

1 Introduction

Feature selection and tracking are the two major problems in many object tracking schemes. In early tracking work, features have been selected based on intuitive descriptions of feature quality. The feature tracker presented in [1] by Shi and Tomasi, an extension of previous work by Tomasi and Kanade in [2], approaches the selection of features in a way that is optimal by construction with respect to the accompanying tracking algorithm. The algorithm for feature selection involves evaluating every window of four pixels in the image frame for a texturedness property. Texturedness is explained in detail in [1], and involves computing the eigenvalues of the 2×2 gradient matrix at a given window. Because the computation of texturedness is completely independent for each window being examined, this seems to be a natural candidate for acceleration using parallelism.

Shi and Tomasi's tracking algorithm is based on an earlier tracker due to Tomasi and Kanade ([2]), which itself uses techniques developed earlier in [3] by Lucas and Kanade. Tracking in the Kanade-Lucas-Tomasi algorithm is accomplished by finding the parameters that minimize a dissimilarity measurement between feature windows that are related by a pure translation motion model. This is another source of potential parallelism, because each calculation of these parameters is independent.

Finally, [1] introduced the concept of using an affine motion model to evaluate feature

quality across frames. This monitoring allows the identification of features that do not correspond to features in the real world, such as depth illusions and edges of highlights. This evaluation process is similarly an independent calculation that only involves matched feature windows, and contains potential parallelism.

In section 2, I discuss reproducing existing results using two implementations of the KLT tracker, and give some details about both implementations. Section 3 contains results from performance profiling and examination of the code of each implementation, including comments on the performance significance of some design decisions. Section 4 has comments regarding the potential parallelization opportunities specific to each implementation. Section 5 discusses some related work that is relevant to performance improvement. Finally, section 6 contains conclusions and possible future work.

2 Implementations and Replication

This section introduces the two examined implementations of the Kanade-Lucas-Tomasi tracking algorithm, the first of which is implemented in Matlab, and was the focus of this performance evaluation work initially. Further research revealed another implementation in C of the tracker. This implementation is due originally to Birchfeld, and is documented in [4]. Each implementation includes example code and images, and it is therefore possible to verify previous results. This was accomplished by running the examples and examining the results in comparison to reference values, when available.

2.1 Matlab implementation

The Matlab implementation is from Jianbo Shi, and is a demonstration of measuring dissimilarity using affine models. It supports the major contribution from [1]. This implementation was available in the UCSD vision lab, and was the original focus of this project. It contains two examples, one which uses a simulated image feature and a deformation of that feature. The example calculates the affine transform that relates the two versions of the feature. The second example uses a real image, and asks the user to choose a feature window. Once the window is chosen, the example treats the same center point as the center of the feature window in a second image, a subsequent frame of a panning motion, and calculates the affine transform as before. Both examples were evaluated, and the results were as expected. The simulated example had a pre-set expected value of error that should result from the affine transform estimation process, and this was easily verifiable.

2.2 KLT - C implementation

The KLT library [4] is a publicly available implementation of the overall tracking method due to Kanade, Lucas and Tomasi. KLT stands for Kanade-Lucas-Tomasi. It includes library routines for image I/O, feature selection based on the technique from Shi and Tomasi [1], and image tracking using the translation motion model. It does not include the feature quality monitoring work from [1] that uses the affine motion model.

This implementation includes five example programs that involve various uses of the library, including demonstrating the I/O features and an optimization. The output files are easily examined to note the features and the success of the tracker. The code appears to perform as expected, and outputs reasonable feature locations and tracking results.

As a side note about this implementation, it is necessary to hand-edit the source code and Makefile in the standard distribution of the library to successfully compile KLT on both Mac OS X and AIX. The instructions in the README file were sufficient to guide this editing. I did not discover this library until late in the project, due to delays in completely

understanding the limitations of the Matlab code I started with. Those delays were caused mainly by unfamiliarity with both the material and the Matlab programming environment.

3 Performance Evaluation

The implementations were evaluated using an Apple Powerbook with a 1.25 GHz G4 processor and 512 MB of RAM, and the NPACI sponsored IBM SP system¹ Blue Horizon, which has 375 MHz Power3 processors. Execution profiling was used to find out where the majority of the running time was spent in each example. This should in general be the first refuge of the performance programmer, as even well-reasoned assumptions about the most expensive operations can be wrong.

3.1 Matlab implementation

The Matlab implementation was examined on the Blue Horizon. Matlab provides a built-in profiling facility, which is enabled using the command `profile on`. Subsequent commands are profiled until the command `profile off` is issued. A report can then be viewed by typing `profile report`. This brings up an HTML file that displays function call times, number of calls and shows the lines of code that took the most time in each function. An interactive GUI profile viewer is available in later versions of Matlab, and can be started with the command `profile viewer`.

The example that used a synthetic image was profiled to avoid measuring the user's selection time. The total time to run the example was 1.33 seconds. This is not necessarily long enough to give reliable results from a profiling timer. However, the timing resolution of the Matlab timer is not easily discovered. Because of that, being unclear about how to scale the input parameters to the algorithm to force it to compute longer without requiring larger images, and inexperience with Matlab, the profiling results were taken as is. I believe that even if the times are not completely accurate for the shortest functions, the overall proportions are correct.

Out of the total 1.33 seconds, the majority was spent in the `compute_AD` function, which is responsible for finding the affine transform. However, the child function `iter_AD` takes up most of the time in `compute_AD`. We need to continue to trace child calls until we find a function that has no bottleneck. Following the function call stack a few levels deeper, we find that the function `find_AD` takes up 29% of the runtime, but has no clear bottleneck. It has many calls to `trapz`, an intrinsic Matlab function that approximates an integral.

It appears that as a serial implementation, this version is relatively efficient. However, keep in mind that it is only performing the core calculation once - on one preselected feature window, for one frame. The limited scope of this example code hampers the performance evaluation of Matlab as a platform for the tracker.

3.2 KLT - C implementation

The KLT library was compiled using the `gcc` compiler, which made it easy to generate profiling data. Using the `-pg` command line option compiles profiling support into the executable. When the instrumented executable is run, it produces a profile data file named `gmon.out`, which can then be interpreted using the `gprof` program. This program produces a hierarchical listing of the time spent in each procedure, including the procedures called by it. This listing is sorted by percentage of overall runtime, and allows a full view of the performance behavior of the program. Other useful statistics, such as number of calls to a procedure, are also reported.

¹<http://npaci.edu/BlueHorizon/>

The KLT example programs were examined on the Powerbook G4, because `gprof` was not working on Blue Horizon. The example program `example3` was examined in most detail. This program finds the 150 best features in an image and tracks them through the next two images. This example enables an optimization in the library that reduces the number of times the image data is copied. The running time of the example averaged 1.5 seconds.

In the case of the KLT example programs, the profiling confirmed that the feature selection procedure was called once, and the feature tracking procedure was called once per feature returned by the selection procedure, for each pair of images being tracked across. In the feature selection procedure, the majority of the time (41.6% overall) is spent in a function that performs a convolution, which is called from a utility function that computes gradients. The convolution function itself is not amenable to parallelization, but because it consists of creating a copy of the image using floating point numbers for each pixel, every time it is called, and only mostly simple calculations otherwise, it is very likely that some caching and better memory management would reduce this bottleneck and allow more improvement from parallelism.

4 Opportunities for Parallelization

4.1 Matlab

A sign that would indicate opportunity for parallelization is a loop or other iterative structure with few dependencies that is performed many times. The calls to `trapz` do not qualify for this, and in fact, neither do the calls to `compute_AD`. The scope of the demonstration code is such that very little parallelizable work is done. No automatic feature selection, tracking, or evaluation across frames is done in either example program, and the code does not implement those algorithms. This leaves little independent work, because the majority of potential parallelism was to be found in selection and tracking of multiple independent features.

In the project proposal, NetSolve [5] was cited as an example of a current, well-maintained parallel math library that interfaces with Matlab. Unfortunately, such libraries do not match well with the kind of computation that is done in this application. Libraries like NetSolve are worthwhile when one needs to perform linear algebra on very large data structures. Client programs can call a parallel version of a standard function and the library will transparently partition the data and launch a parallel computation on available machines. The library will not help if the algorithm requires custom parallelization. The Matlab tracking program under evaluation does make many calls that NetSolve has parallel implementations of, but they are all performed on very small matrices, which would mean that using NetSolve's version would expand runtime greatly.

There do exist general purpose Matlab parallel interfaces, which would allow custom parallelization. A survey of them is available [6] but from personal experience, the available ones are not stable or well-maintained. Matlab's creators, Mathworks do not provide an official parallel interface, and do not plan to. Therefore, because of the architecture of Matlab, and the majority of calculations being performed on small matrices, the Matlab demonstration of the Shi & Tomasi algorithm does not hold much promise for parallelization.

4.2 C

The C language KLT tracker has much more potential for parallelization. Because it implements the entire algorithm, it iterates over many feature windows during selection and tracks each feature across as many images as are necessary. A promising approach to parallelization would be to separate the image into sections and have multiple processors

handle the image selection and tracking. Because it is written in C, the stable, industry standard MPI parallel communication libraries can be used to perform custom task-based parallelization, as opposed to simply editing calls to linear algebra subroutines.

There are two problems with the above approach, however. First, the code as written stores a great deal in global state. If it were required to scale much past the size of a single shared-memory computer (usually 4 to 8 processors, as on a node of Blue Horizon), the code would require significant rewriting to efficiently share the global state information across the interconnect network of a larger system. Second, the example programs again run so quickly that parallelization might have a detrimental effect. However, the examples only track a small number of features in a moderate sized image - there is some evidence that applications may soon require more computation. For example, see [7].

A consequence of the library being written in C is that a potential low-effort parallelization opportunity, the use of OpenMP constructs, was not available. OpenMP is a standard for Fortran programs which allows parallelization of loop structures through the use of simple code annotations and a special compiler. This can be a very easy way to achieve performance gains on small processor counts where it is easy for a compiler to efficiently distribute data. However, parallelization using MPI does seem promising for large applications. However, because the existence of the KLT library was unknown until very late in this project, no results are available to show.

5 Related Work

Aliaga et al [7] use a custom version of the Kanade-Lucas-Tomasi algorithm for large (1024×1024). The reasons given were that the performance was not good enough for their application. They cited multiple copying and expensive conversions done for each frame as reasons. Their custom version uses specific knowledge of their application to improve performance, as well as utilizing multiple processors. It is not explained how their parallelization is done.

There has been some work in parallel feature tracking, including a parallel spline-based feature tracker by Szeliski et. al [8]. Implementation details of the Szeliski tracker are not clear, especially with respect to parallelism. They state that their tracker is parallel, but do not count it as a contribution or use that fact in comparison to other techniques, and provide no further details.

In [9], Benedetti and Perona present a real-time feature detection system that relies on reconfigurable hardware for high performance. Their special hardware setup allows them to exploit very low-level parallelism in the computations that would be unprofitable to attempt with a networked supercomputer or grid system. This is due to the high communication speed present in their hardware, where individual functional units read data from a single high-speed bus.

6 Conclusion

During the course of this project I have learned a great deal about implementing vision algorithms. There is a clear tradeoff between ease of implementation and practicality. The Matlab implementation generates a clear demonstration, is short and readable, and clearly reflects the mathematics involved. The C implementation was clearly much more work, and while well-written and clear, is still more difficult to read. Because of the state of parallel tools for the environment, the choice to use Matlab also limits one's options for performance improvements if an algorithm is deemed worthy of production use.

It is unfortunate that I didn't find the KLT C library sooner, because the relative incom-

pleteness of the Matlab version limited its value for the intended focus of this project. This project also underscores a fact about parallel computing - that it is not necessarily an improvement always. Many calculations do not map well to parallel computers, and it is very important to carefully analyze the actual performance of programs that do have an intuitive parallelization, because it may be that the true bottleneck is elsewhere.

References

- [1] Jianbo Shi and Carlo Tomasi. Good features to track. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR'94)*, Seattle, June 1994.
- [2] C. Tomasi and T. Kanade. Detection and tracking of point features. Technical Report CMU-CS-91132, Carnegie-Mellon University, 1991.
- [3] B. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, pages 674–679, Vancouver, 1981.
- [4] Stan Birchfield. KLT tracker library manual. <http://vision.stanford.edu/~birch/klt/>.
- [5] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4.1. Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, June 2002.
- [6] MIT. Parallel matlab survey. <http://supertech.lcs.mit.edu/~cly/survey.html>.
- [7] Daniel G. Aliaga, Dimah Yanovsky, Thomas Funkhouser, and Ingrid Carlbom. Interactive image-based rendering using feature globalization.
- [8] Richard Szeliski, Sing Bing Kang, and Heung-Yeung Shum. A parallel feature tracker for extended image sequences. Technical Report CRL 95/2, Digital Equipment Corporation Cambridge Research Lab, 1995.
- [9] Arrigo Benedetti and Petro Perona. Real-time 2-d feature detection on a reconfigurable computer. In *CVPR*, pages 586–593, 1998.

A Matlab Profile Listing

The Matlab profile listing is an HTML page that is attached separately.

B C KLT Profile Listing

This is an abbreviated listing of the output from `gprof` for the most time-consuming functions in the KLT tracker, defined as those that consume at least 10% of the total runtime.

index	%time	self	descendents	called/total	parents	
				called+self called/total	name	index children
		0.00	0.93	1/1	__start	[2]
[1]	92.1	0.00	0.93	1	_main	[1]
		0.00	0.69	2/2	_KLTTrackFeatures	[3]
		0.00	0.23	1/1	_KLTSelectGoodFeatures	[8]
		0.00	0.01	3/3	_KLTWriteFeatureListToPPM	[22]

		0.00	0.00	3/3	_pgmReadFile [70]
		0.00	0.00	3/3	_KLTStoreFeatureList [65]
		0.00	0.00	2/2	_KLTWriteFeatureTable [73]
		0.00	0.00	1/1	_KLTCreateTrackingContext [78]
		0.00	0.00	1/1	_KLTCreateFeatureList [76]
		0.00	0.00	1/1	_KLTCreateFeatureTable [77]

					<spontaneous>
[2]	92.1	0.00	0.93		__start [2]
		0.00	0.93	1/1	_main [1]

[3]	67.8	0.00	0.69	2/2	_main [1]
		0.00	0.69	2	_KLTTrackFeatures [3]
		0.00	0.36	6/7	__KLTComputeGradients [5]
		0.00	0.13	580/580	__trackFeature [11]
		0.00	0.09	3/7	__KLTComputeSmoothedImage [10]
		0.00	0.09	3/3	__KLTComputePyramid [12]
		0.01	0.00	3/4	__KLTToFloatImage [19]
		0.00	0.00	290/290	__outOfBounds [54]
		0.00	0.00	9/9	__KLTCreatePyramid [60]
		0.00	0.00	6/6	__KLTFreePyramid [63]
		0.00	0.00	5/51	__KLTCreateFloatImage [56]
		0.00	0.00	5/45	__KLTFreeFloatImage [57]
		0.00	0.00	4/8	_KLTCountRemainingFeatures [61]
		0.00	0.00	3/5	__KLTComputeSmoothSigma [64]

[4]	62.4	0.00	0.21	7/21	__KLTComputeSmoothedImage [10]
		0.00	0.42	14/21	__KLTComputeGradients [5]
		0.00	0.63	21	__convolveSeparate [4]
		0.32	0.00	21/21	__convolveImageHoriz [6]
		0.31	0.00	21/21	__convolveImageVert [7]
		0.00	0.00	21/51	__KLTCreateFloatImage [56]
		0.00	0.00	21/45	__KLTFreeFloatImage [57]

[5]	41.6	0.00	0.06	1/7	__KLTSelectGoodFeatures [9]
		0.00	0.36	6/7	_KLTTrackFeatures [3]
		0.00	0.42	7	__KLTComputeGradients [5]
		0.00	0.42	14/21	__convolveSeparate [4]
		0.00	0.00	4/13	__computeKernels [58]

[6]	31.7	0.32	0.00	21/21	__convolveSeparate [4]
		0.32	0.00	21	__convolveImageHoriz [6]

[7]	30.7	0.31	0.00	21/21	__convolveSeparate [4]
		0.31	0.00	21	__convolveImageVert [7]

[8]	23.3	0.00	0.23	1/1	_main [1]
		0.00	0.23	1	_KLTSelectGoodFeatures [8]
		0.10	0.14	1/1	__KLTSelectGoodFeatures [9]
		0.00	0.00	1/8	_KLTCountRemainingFeatures [61]

[9]	23.3	0.10	0.14	1/1	_KLTSelectGoodFeatures [8]
		0.10	0.14	1	__KLTSelectGoodFeatures [9]
		0.00	0.06	1/7	__KLTComputeGradients [5]
		0.00	0.04	1/1	__sortPointList [17]
		0.00	0.03	1/7	__KLTComputeSmoothedImage [10]
		0.01	0.00	1/4	__KLTTToFloatImage [19]
		0.00	0.00	52224/52224	__minEigenvalue [46]
		0.00	0.00	4/51	__KLTCreateFloatImage [56]
		0.00	0.00	4/45	__KLTFreeFloatImage [57]
		0.00	0.00	1/5	__KLTComputeSmoothSigma [64]
		0.00	0.00	1/1	__enforceMinimumDistance [81]

[10]	20.8	0.00	0.03	1/7	__KLTSelectGoodFeatures [9]
		0.00	0.09	3/7	_KLTTrackFeatures [3]
		0.00	0.09	3/7	__KLTComputePyramid [12]
		0.00	0.21	7	__KLTComputeSmoothedImage [10]
		0.00	0.21	7/21	__convolveSeparate [4]
		0.00	0.00	7/13	__computeKernels [58]

[11]	12.9	0.00	0.13	580/580	_KLTTrackFeatures [3]
		0.00	0.13	580	__trackFeature [11]
		0.05	0.02	2099/2099	__computeIntensityDifference [13]
		0.02	0.04	1519/1519	__computeGradientSum [15]
		0.00	0.00	1740/1740	__allocateFloatWindow [47]
		0.00	0.00	1519/1519	__compute2by2GradientMatrix [49]
		0.00	0.00	1519/1519	__compute2by1ErrorVector [48]
		0.00	0.00	1519/1519	__solveEquation [50]
		0.00	0.00	580/580	__sumAbsFloatWindow [51]