Optical Flow Analysis Using the Lucas-Kanade Algorithm

Sophie Smith (sophiesm) and Urvi Agrawal (urvia)

15-418 Final Project, Fall 2019

Summary

For this project, we parallelized the Lucas-Kanade Algorithm for optical flow and object tracking across image frames. We evaluated different approaches to parallelism using CUDA, Open MP, and MPI on the GHC Cluster machines. We found approximately 9x speedup with our data-parallel shared address space implementation, 3x speedup with our asynchronous message-passing implementation, and 31x speedup with the CUDA approach.

Background

Optical flow analysis aims to track the movement of a specific image region over time and is commonly applied to object tracking applications in computer vision. Based on a sequence of images, the goal is to compute a vector tracking the movement of individual pixels from one image frame to the next. The Lucas-Kanade algorithm is a popular approach for optical flow analysis which operates under the assumption that the flow of a given pixel in an image is approximately constant in the direct neighborhood of this pixel across consecutive frames. Given this assumption, this allows the use of a least-squares estimate to match an image template containing selected pixels from some previous frame to the updated positions of these pixels in the next consecutive frame.

Despite these assumptions, this algorithm accurately tracks pixel movement and resultantly is used across many areas of computer vision, including object tracking, image alignment, and general optical flow applications.

The goal of this project is to explore the parallelization of the Lucas-Kanade method for optical flow analysis using the CUDA platform, Open MP library, and MPI library. We aim to determine which model is best suited for the parallelization of this application and our selected workload, as well as analyze the ways in which we need to refactor our implementation to best specialize the algorithm to each framework.

Inputs, Outputs, and Data Structures for Lucas-Kanade Algorithm

The Lucas-Kanade algorithm operations on consecutive image frames to track the movement of selected image coordinates across consecutive frames. The input to the algorithm is the set of coordinates corresponding to the tracked set of image pixels. For every coordinate *i*, this is represented through a pair of doubles (x_i, y_i) stored across two equal-length arrays. Across most of our implementations, we initialized 1800 coordinates centralized around the license plate of our selected frame sequence (see image below), although our point quantity and location varied throughout the different benchmarks and implementations we performed.

Additionally, the algorithm operates given a sequence of image frames as input. For our analysis, we selected a grayscale video featuring the movement of cars across a road. The pixel values were obtained using a Matlab script and stored in textfiles. Through our implementation, we loaded the pixel values to represent images as two-dimensional arrays of doubles.

Figure 1. Sample frame of video used in tracking application.

Throughout each iteration, we kept temporary local arrays of modifications to the frames, including warping and smoothing, described later in this paper. We also kept intermediate updates to pixel locations for computing the flow of each pixel across each given frame.

The outputs from this algorithm are the updated coordinates representing the new location of each pixel throughout the progression of the provided frames. We modeled these results with in-place updates to the coordinate arrays.

Algorithmic Description

Due to the flexibility and assumptions of the Lucas-Kanade algorithm, we had the opportunity to iterate on multiple implementations of the algorithm to find the version best suited for our applications and workload. Our first approach was based on a Matlab implementation of the Lucas-Kanade tracking algorithm implemented as part of a different course at CMU. This is the most formal variant of the algorithm and the most precise implementation in terms of tracking accuracy. As it iterates across the consecutive frames in the image, the following steps are computed on pairs of frames to determine pixel updates. For each pixel to update, the updates are performed in a convergence loop which continues iterating until the norm of the pixel movement reaches a small constant:

```
perform linear interpolation on both image frames
compute error image and gradient in x, y directions on all frames
compute the Hessian using the Jacobian and gradient
compute pixel difference using Hessian inverse, Jacobian, error image
update x, y coordinates by the computed difference
```

The challenge with this implementation is the heavy emphasis on matrix applications. To maintain precision in tracking, there's a significant amount of matrix dependent computation and interpolation. The challenge with this is for efficient implementation, a matrix or specialized computer vision library is essential. Thus, we decided to experiment with configuring our workspace with OpenCV, however, due to space limitations and complications with the configuration process, we found this to be too challenging. Therefore, we attempted to implement this algorithm without library operations by simulating matrices and matrix operations with multi-dimensional arrays. Without libraries, we found no easy approach to implementing linear interpolation and therefore needed to readjust our approach to focus on a different implementation of the algorithm.

Next, we tried a significantly more simplified implementation of the Lucas-Kanade algorithm. Due to the assumption by Lucas-Kanade that flow remains constant in the direct neighborhood of a pixel, we determined that the computationally heavy calculations including applying kernels and warping images could be approximated with subtraction and multiplication of a small neighborhood around the pixel of interest. The simplified algorithm operates to determine pixel updates by the following steps inside the convergence loop:

```
inside convergence loop across every pixel:
	for 5x5 neighborhood around the pixel:
		approximate error, gradient by subtraction of neighboring values
		compute an approximate Hessian using these values

	update pixel coordinates by computing determinant of Hessian
```

We fully implemented and tested this approach, but then realized that upon integrating parallelization to this implementation, the arithmetic intensity was too low for effective parallelism with a Shared Address Space or MPI model. We performed too much oversimplification on the computation such that performance was significantly decreased due to the overhead of spawning threads. Therefore, to combat this problem, we decided to implement a sequential algorithm with a complexity midway between the two previously attempted implementations to use with these models. The steps for our finalized algorithm are the following:

```
inside convergence loop across every pixel:
	compute gradient and reduce noise
	for neighborhood around pixel:
		access computed error, gradient
		compute approximate Hessian using values

	update pixel coordinates based on Hessian determinant
```

Workload Analysis

To better understand the aspects of the algorithm most beneficial to parallelize, we analyzed the amount of time spent in each major part of the algorithm. Comparing across the three major components of the algorithm (loading and initializing frames, smoothing and preprocessing images, performing the convergence loop for pixel updates), we found that barely any fraction of time is spent in the former two steps. Seen in Figure 2, we found it most logical to focus all optimization efforts on the direct application of the Lucas-Kanade update function.

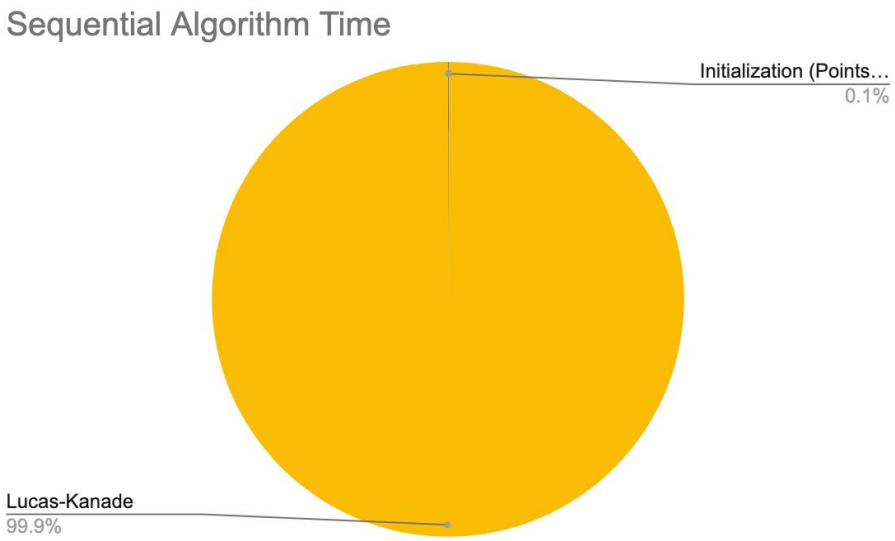

Figure 2. Workload distribution of sequential algorithm.

To determine the intricacies of the workload, we analyzed the breakdown of time spent within the main Lucas-Kanade computation. Inside this function, we find a better distribution of work. For our analysis, we chose to focus our parallelism approach on the neighbor computations, i.e. computing movement across the neighboring region for a pixel to determine given flow for each pixel. We found this component to be less trivially parallelizable due to the workload imbalances of the implementation. Therefore, it would provide a more challenging task for us to implement and optimize.

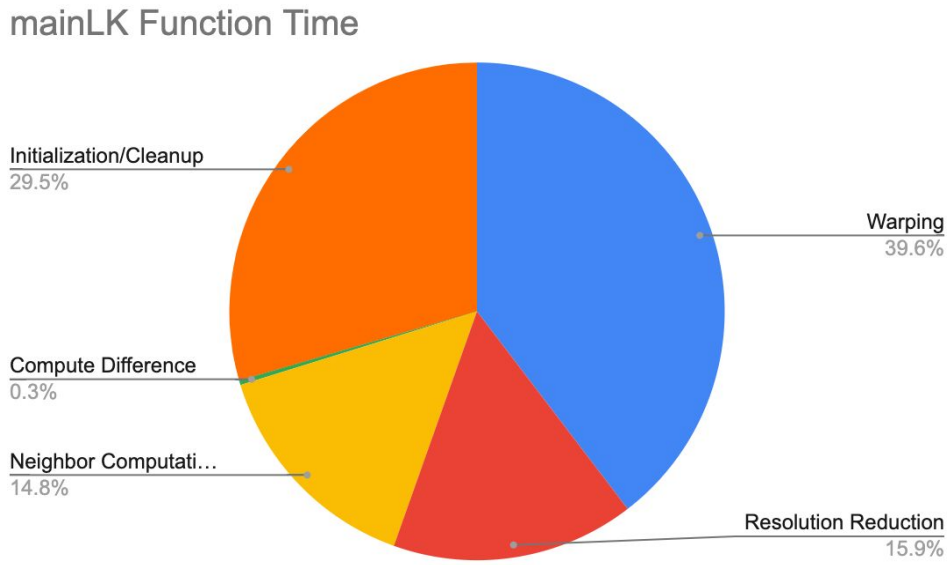

Figure 3. Workload distribution of the main Lucas-Kanade computation function.

Workload Challenges and Dependencies

The Lucas-Kanade algorithm is sequential in nature which limits the amount of parallelism we can achieve. The algorithm operates by starting with selected points in the initial image frame. Then, pairs of consecutive image frames, the above process is applied to determine the movement of the pixel from the former frame to the latter. Using the updated location values from this pair, the next consecutive frame is analyzed to determine movement across this incremental step. Thus, each frame is dependent on the pixel coordinate updates from the previous frame and there is no way to parallelize across frames without significant approximations of pixel movement patterns and a subsequent loss of accuracy. Thus, it's more realistic to focus on parallelizing the computation on consecutive frames rather than the pipelined process as a whole.

The first challenge for parallelism within the inner computation is with image preprocessing. The resolution reduction step is used to eliminate noise in the case where the movement of previous pixel values might have been too large to be considered accurate. This step, achieved by applying a smoothing kernel to each image coordinate, has dependencies on neighboring coordinates. This is implemented using an in-place update of coordinate values, therefore this implements a strict ordering of smoothing kernel applications. To parallelize this step would require significant stalling to wait upon the update of dependent pixels. Thus, it would essentially serialize the algorithm because of the order we're able to iterate across the image. Due to this, we found it impractical to attempt to parallelize this step.

The second opportunity and challenge for parallelism comes from the workload imbalance across each iteration of updating pixel coordinates. While each pixel update can occur

independently of updates to other pixels, to complete the entire update of each pixel requires individual pixel updates to finish. As described above, the flow of a given pixel is computed through a convergence loop to determine the optimal movement of the coordinate. Shown in Figure 4, there is a significant amount of variation in the average number of iterations required until convergence. Despite setting an upper bound on the number of iterations, we still found varied performances. Each individual iteration requires a non-trivial amount of time, leading to the pixel updates to vary vastly in completion time. Thus, the emphasis of our project was exploring different dynamic scheduling and load balancing techniques in the different parallelism frameworks to limit overhead and optimize speedup.

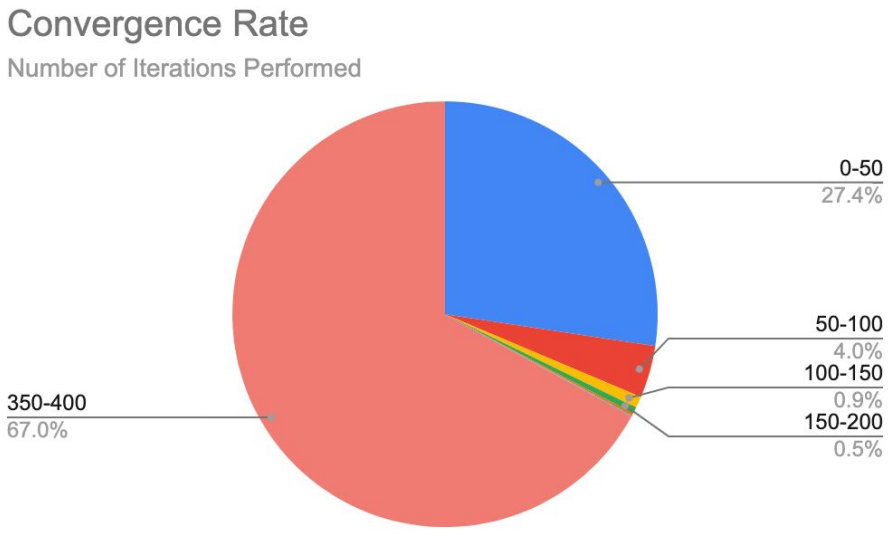

Figure 4. Distribution of number of iterations required until convergence.

Experimental Baseline

For each analysis, we benchmarked against a single-threaded implementation of the Lucas-Kanade algorithm. Due to the varied parallelization models which we implemented, each benchmark required slight refactoring of the implementation. Therefore, the baseline for each model was compared against a sequential model of the algorithm refactored identically to the parallel model. This ensured a consistent comparison of performance across sequential and parallel timing measurements.

Approaches and Results

Shared Address Space

Described earlier, initial attempts at parallelism were applied to the oversimplified Lucas-Kanade implementation. Due to the overhead associated with this approach from the low arithmetic intensity, we rewrote the sequential implementation in favor of a more computationally expensive algorithm. This modified algorithm, also described above, was then optimized by both data-parallel and task-parallel approaches using Open MP.

Data Parallelism

The main convergence loop for each individual image pixel has no dependencies on the updates made to other pixels in the same frame. Thus, it is relatively intuitive to pursue a data-parallel approach to speedup. For each pixel, we compute the optical flow by following the same set of computations: image gradient, reduction of resolution and convergence loop. Therefore, we can distribute the workload by assigning each thread a given number of pixels to compute updates over. As there was no complexity due to synchronization with this approach, the main challenge of parallelization was due to load distribution. From the previous algorithm analysis, we saw that the convergence loop took varying amounts of time to complete. Since processing the next frame in the pipeline is dependent on the completion of all pixel updates, we needed to ensure that the threads finished computing these updates at approximately the same time to limit the amount of idle time threads encounter. Static scheduling is insufficient to accomplish this balance as there is no pattern to determine which pixels will converge slower. This is all dependent on previous values, neighboring intensities, and the current frame. Thus, for our project, we focused on experimenting with dynamic scheduling techniques.

To distribute work, we used the Open MP parallel for construct in the section where we iterate across all pixels and perform the convergence loop on each. On smaller frames and pixel counts, we experimented with different chunk sizes and scheduling techniques. We found moderate improvement by modifying the chunk size to limit the amount of communication and synchronization necessary for threads to access the dynamic task queue. The main benefit, as hypothesized above, was from the selection of dynamic scheduling rather than static scheduling. Selecting a dynamic schedule with a chunk size of 8, we eliminated a fair amount of overhead with threads accessing the work queue and were able to moderately balance the drastically varying convergence rates of pixels.

Results

We computed the speedup of our implementation by measuring the total execution time across an implementation of tracking 1800 pixels across the progression of 180 consecutive image frames. The speedup was computed in regards to our single-threaded reference implementation. Shown in the below plot, the optimal speedup achieved was approximately 9x on 16 threads.

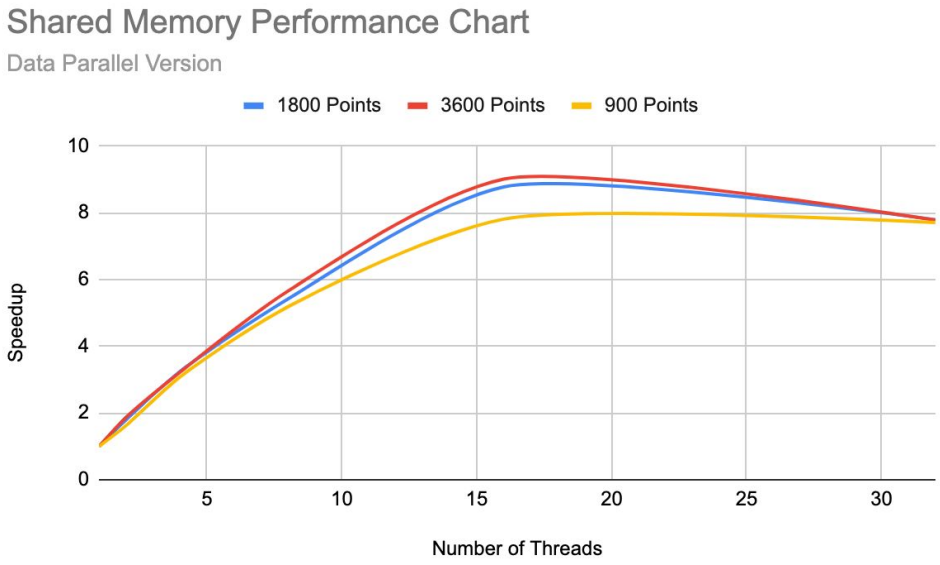

Figure 5. Speedup observed for data-parallel shared address space model.

Analysis

Despite implementing a more arithmetically intense version of the Lucas-Kanade algorithm, we still found there to be a less-than-ideal amount of work for each thread to complete. Due to this, the overhead of spawning threads to handle computation across all pixels and coordinating access among these threads to the work queue decreased the amount of relative speedup we could achieve. It is also likely that our implementation failed to achieve perfect load balancing in regards to the number of iterations until convergence. With unequal load balancing, some threads may have less work than others and be left idle waiting until the next frame iteration of the algorithm when they will be assigned more work. In the future, it would be interesting to experiment more with different scheduling techniques and different implementations of the convergence step to analyze the impact of this on achievable speedup.

From the plot above, we also observe no major impact on speedup due to problem size. As we modify the problem size, i.e. the number of pixels we compute, there is only a slight adjustment in the speedup achieved. This is because by increasing the number of pixels, we only have more work which the threads must divide amongst themselves to compute. If we had access to larger sized frames, then it would be interesting to explore the impact of this on speedup. Larger images would require a longer amount of time for all frame preprocessing including computing the gradient and reducing noise. This could potentially improve speedup by assigning more work per thread per pixel.

Task Parallelism

The previous sequential workload analysis indicated that besides the convergence loop, a significant amount of time is spent in the preprocessing steps per pixel. That is, computing the gradient across the image and preprocessing the frames to remove noise. Contrary to our

strictly data-parallel approach explored previously, we chose to experiment with parallelizing the preprocessing steps through task-based parallelism.

All preprocessing steps must complete before the convergence loop to compute optical flow can occur for each pixel. Therefore, rather than including preprocessing inside the loop body for each pixel computation, as our initial sequential implementation did, we refactored our code to create two separate loops across all pixels. The first loop completed all preprocessing tasks and the second loop completed the convergence updates for each given pixel. Additionally, we have to restructure our representation of a pixel for ease of storing all preprocessed results since now they occur at a separate time than the pixel-specific coordinate updates. The impact of this is the requirement of additional memory for representing each value. Overall, this approach is well-suited for shared address space parallelism as no additional communication between threads is necessary to share the results of preprocessing for a coordinate to the thread computing the pixel update for the given coordinate.

The five tasks which every thread may complete are computing the error image, computing the gradient in the x-direction, computing the gradient in the y-direction, and if the previous updates are large enough, reducing the resolution of the next and previous frames. Each of these tasks is dependent on the current pixel coordinate for update amounts and whether the task will be completed. Using the Open MP tasks construct, we experimented with have separate threads handle each of these tasks. Additionally, we experimented with different granularities of tasks, i.e. rather than separating each function call into a separate task, we explored the impact of different groups of functions under one thread. In the results below, we consider the smallest granularity as the case where each function is its own task. Medium granularity groups computing the error image with the noise reduction (since noise reduction is not guaranteed to be computed for each pixel). And largest granularity is all gradient and error computations in one task and noise reduction in a separate task. Because pixel updates are dependent on this preprocessing step, we had to synchronize execution and ensure all preprocessing completed prior to the computation of optical flow on any given pixel. In theory, we could have begun computing optical flow on the pixels whose values had been preprocessed, but the additional overhead with scheduling this approach and fewer resources dedicated to our task computation would most likely decrease performance overall.

Results

First, we analyzed the performance of the smallest granularity model across different thread numbers, finding approximately equally good performance across 16 and 32 threads. Shown below is our comparison of different granularity across these same thread numbers. We observe approximately equal results. The baseline sequential model was also refactored based on similar principles for a more accurate comparison.

The below graph displays the speedup results for our hybrid task- and data-parallel approach (as we kept the parallel for loop across the convergence loop). We measured speedup in comparison to a single-threaded sequential model also refactored with the same modifications

we made to this implementation. We found optimal performance with the highest granularity implementation, but overall worse speedup than achieved with our strictly data-parallel implementation.

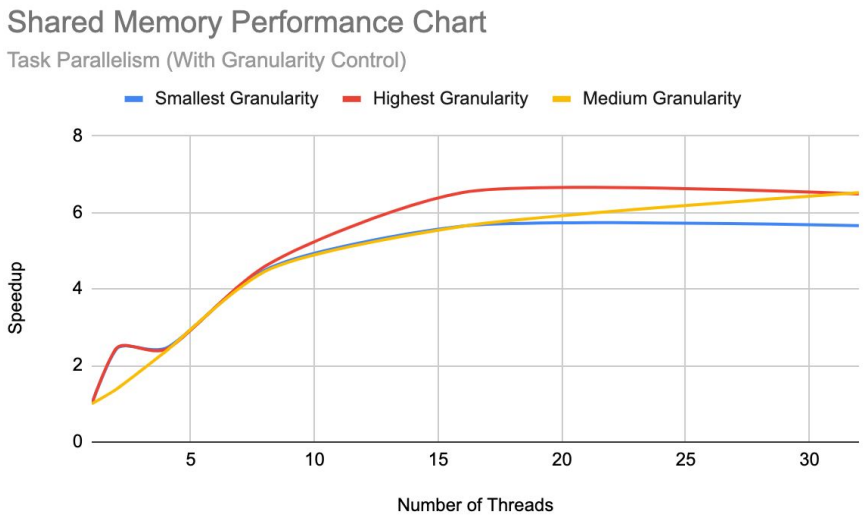

Figure 6. Speedup observed for task-parallel shared address space model.

Analysis

We find through these trials that task-based parallelism performs worse than the strictly data-parallel approach. Through an analysis of our model, this intuitively makes sense as by refactoring our code to first preprocess all images and then perform pixel updates, we needed to ensure all tasks completed before continuing execution. Thus, the waiting period necessary and the additional overhead to spawn threads to handle these relatively limited computations provides less benefit than ideal. Potentially, if the tasks were all computationally more expensive, we would have found better performance. Ultimately, the task-based parallelism still presents moderately good speedup in comparison to our sequential implementation.

Across the task-based parallel approach, we found some benefit to having the largest granularity of tasks. Although there is not much variation achieved with the different granularity tasks, we do observe slight differences. In the smaller granularity cases, the amount of work per task is not enough to justify the overhead of threads accessing the task queue. The synchronization across accesses and waiting time on spawning more tasks decreases the benefit since the amount of time in each task is relatively smaller. With more granularity, each thread computes more work per task and therefore overall will have fewer accesses to the shared task queue. This lessens the negative benefits due to coordinating the distribution of tasks among the threads.

Overall, it makes sense if rather than preprocessing all images prior to updating pixel coordinates, we handle the preprocessing at the same time the given pixel will be updated. Firstly, it's reasonable with respect to the amount of memory used in the program. By handling

all preprocessing initially, we need to store the results of these computations so that they can be later accessed by the threads while performing the convergence loop to update pixel coordinates. As described with the implementation, this was handled by creating a struct for each pixel which contains preprocessed images. Thus, rather than the lifetime of a preprocessed image being only inside the convergence loop for its respective pixel, the allocated space for the pixel exists across the entire implementation. So, while it saves time in terms of allocating and freeing memory often, it requires us to consistently have more memory allocated than necessary at a given amount of time. Additionally, the overhead with creating tasks, adding these tasks to the task queue and assigning threads to handle each lessens the benefit of task-based parallelism due to the associated overhead.

Message Passing

As the majority of our project is dependent on efficiently scheduling the convergence loop to limit idle threads and optimize load balancing, we chose to explore a message-passing implementation using MPI so that we could have more freedom with scheduling mechanisms. Optimization with shared address space using Open MP is fairly limited due to the high-level constructs applicable, so there wasn't much freedom in terms of work delegation with this approach. Through delegating a master thread to handle scheduling and work assignments in our MPI implementation, we were able to explore different scheduling approaches and granularities of messages to determine the impact on overall speedup.

Synchronous Model

To begin restructuring the implementation for message passing, we found it easier to start by implementing synchronous sends and receives. While we expected to observe worse performance with this model due to the unnecessary waiting periods, we still found it valuable to observe the influence various factors had on the performance of this model.

For this implementation, we associated the master thread (thread index of 0) with the task of work scheduling. This thread handles dynamically providing the pixels to each thread when the thread is willing to accept work. The general structure is that the master thread iterates across all worker thread indices and sends a segment of pixels to each worker. Then, the workers handle computations on these values. Once complete, all worker threads send the results back to the master thread where the master places the updated coordinates into the array of accumulated results.

The problem with this implementation, however, is due to these differing convergence rates of pixels. With the structure of this implementation, the master thread waits on receiving results from the workers through a series of synchronous receives. Through iterating over each of the thread indices, the master calls receive on each and blocks until the receive instruction has completed. Thus, even if a later thread index converges faster and is willing to accept work earlier than some previous index, the master must wait on receiving results from the earlier indices before receiving and subsequently assigning work to later indices which may in practice converge faster.

Asynchronous Model

To alleviate this problem of load balancing with the synchronous model and to enable sending more work almost immediately after the worker thread has completed its set of computations, we explored using asynchronous send and receive instructions.

The segment of the code where we implement this dynamic scheduling on the master thread is contained in the Appendix. In brief, however, to accomplish this implementation we needed to maintain two arrays of MPI Request* structs, one for send instructions and one for receive instructions, in order to determine when the respective asynchronous calls finished. At the beginning of each iteration, the master thread sends all messages to every thread which is willing to accept work. That is the threads whose previous send entry is either NULL or completed. Using the MPI_Test instruction, we are able to identify whether an instruction has completed. Then, the master thread iterates across all threads once again and initiates a receive instruction from each so long as the previous receive instruction is either NULL or completed. By implementing these cases, we ensure that the master thread won't execute out of order in terms of trying to receive or send with threads that haven't yet completed their assigned work. This logic is implementing through continuously looping to ensure instructions are accomplished whenever applicable.

In terms of worker scheduling, each worker thread begins by receiving a range of coordinates from the master thread. We follow this asynchronous receive by an MPI_Wait instruction which essentially simulates a synchronous receive. This is necessary as the computations can't begin until the coordinates have been received. After performing all necessary updates, the thread initiates an asynchronous send of results back to the master, returning to the beginning and waiting to receive the next set of work to complete.

As discussed previously, we experiment with different granularities with respect to the number of coordinates sent. This is explored to analyze the impact of fewer sends and receives from the master thread to each worker. The results for various combinations tested are displayed below.

Results

Below we find a plot comparing the speedup achieved by varying numbers of processors across both asynchronous and synchronous execution models. To determine the effect of granularity of message size, we plotted results at a variety of message sizes, i.e. number of pixel coordinates sent at one period of time.

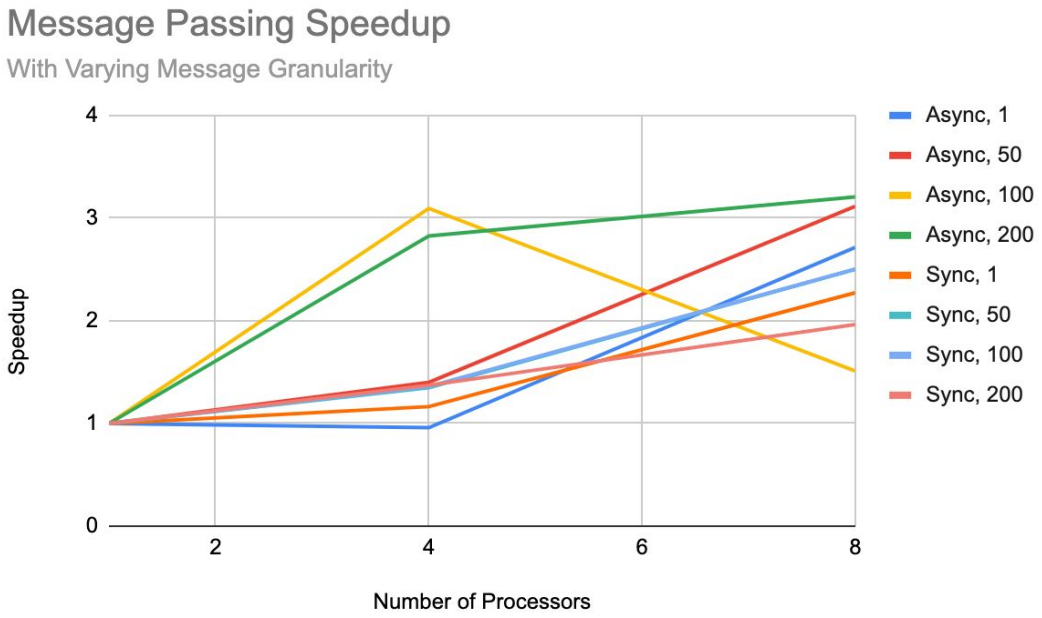

Figure 6. Speedup observed for message-passing implementations.

Analysis

As expected, we found better performance with the asynchronous message passing implementation over the synchronous approach. This is because the synchronous approach had unnecessary waiting by the master thread on receiving results from the workers which lessened all opportunities for load balancing. By handling the receive instructions in a strict ordering across the threads, we guarantee that all sends and receives to or from this worker thread must happen in this order, regardless of the rate the pixels assigned to each worker thread converge. With the asynchronous model, since we don't wait on the completion of any request and instead use the MPI_Test and MPI_Wait instructions to maintain a rough ordering, we are better able to accommodate these different convergence rates.

We also found better results across the larger granularity models, specifically the implementation in which we send 100-200 pixel coordinates per message. With a larger message size, we limit the number of times each worker thread needs to wait on receiving work from the master and needs to communicate for receiving the initial pixel coordinates to handle updates on, as well as sending the results back to the master. With all of these steps, there is time dedication to obtaining these values, reporting updates and waiting on the completion of instructions and assignment of new work which is not contributing to solving the entire parallelization across each pixel. Thus, it's expected to observe these improved results based on granularity.

Overall, our message-passing implementation did perform worse than the comparable shared address space approach. This can be attributed to the fact that shared address space doesn't

involve the communication overhead which is associated with message passing. Each computation is associated with the initial receiving of values and the end reporting of results, and each of these steps requires a non-trivial amount of time. Shared address space doesn't encounter any of these costs which helps explain the improved speedup we observed in that model. Additionally, with the shared address space approach, we didn't need to dedicate a thread to handle the scheduling, so we were able to have more workers to divide the pixel computations across.

CUDA

Since the main Lucas-Kanade update section can be run on each individual pixel independently of the neighboring values after the initial preprocessing, there is a high potential for data parallelism here. So we implemented a parallel version of the algorithm on the CUDA platform to be able to better exploit the fact that we have a lot of pixels that can all be updated in parallel using CUDA kernels.

The CUDA version of the algorithm is based on the initial simpler sequential code since it was started before we updated the algorithm. A big issue we faced with this implementation was how memory intensive the update step is. At a time, the update step requires space for 3 5x5 double arrays, 2 2x2 double arrays, and more smaller arrays to keep track of all the information for a single pixel. When we compute this step in parallel using a kernel for an entire thread block, this places a lot of strain on local device memory. In fact, when we first tried launching a kernel with 900 threads in order to keep a high thread count, the launch failed because it completely exhausted the register space on the multiprocessor. There isn't really an algorithmic way to fix this as all of those matrix computations are critical to the Lucas-Kanade update.

The two approaches we tried to successfully launch a kernel are dynamically allocating memory from inside the kernel using malloc and decreasing the number of threads per block. Both of these methods help with significantly reducing the load on the registers while executing the kernel. The issue, however, is that both of these approaches significantly lower speedup. Having a lower number of threads per block is inefficient because it doesn't effectively utilize the available resources and dynamic memory allocation from inside the kernel is significantly slower than statically allocated arrays because addressing the heap from within the kernel. We tried converting all of our local arrays in the kernel to dynamically allocated and that gave us extremely slow execution and was definitely not an efficient fix. We also experimented with different thread counts and found that our kernel can only be successfully launched with 200 threads per block when all of the arrays statically allocated. While this is a very low thread count compared to the maximum of 1024 threads that is possible, it was about 10x faster than the dynamic memory allocation fix that we tried. We tried some midway approaches with a mix of static and dynamic allocations in order to increase the number of threads per block, but the overhead of dynamic allocation from within the kernel is simply too high and hence we didn't get good results with a mixed approach.

Our final CUDA implementation consists of a data-parallel approach where we parallelize the image smoothing preprocessing step over the entire image and the update step across the pixels we're tracking. For explicit dimensions, with our sample license plate image, the smoothing kernel works with 900 threads per block and 384 such blocks to cover the 720 x 480 image and the Lucas-Kanade update kernel works with only 200 threads per block and 6 such blocks to cover the 1800 pixels in the license plate bounding box that we're tracking.

A small difference between the CUDA implementation and all of the other implementations is that we don't preprocess the images sequentially here. The other parallel versions simply do the image smoothing in place just like the sequential algorithm, which works out because in terms of total execution time, a very small fraction of time is spent preprocessing and it's not worth parallelizing that part with the overhead of thread creation and message passing. For CUDA, we tried in place smoothing and using a smoothing kernel parallelized across all the pixels in the entire image. It didn't change our results much since it is such a small part of the overall execution time so we simply kept the kernel implementation. Although the final output varies slightly with the kernel because the smoothing isn't as perfect, it doesn't affect the overall position of the bounding box by much and isn't a significant error.

Results

The CUDA version gives us ~31x speedup from the initial sequential version. This is by far the best speedup we've had from all of our parallel implementations simply because of how powerful it is to run data-parallel code on a GPU versus a CPU. There aren't any plots for this version because there aren't multiple configurations that we can test this with.

Analysis

Although this is our best speedup, it is still extremely limited due to memory intensity of the update kernel. Being limited to 200 threads per block means that our resources remain severely underutilized. We could perhaps explore a way to further simplify the matrix computation but this is already adapted from a simplified sequential version. Further diluting the math might lead to bad results. Another thought was to come up with a pipelined version of the convergence loop so that we could divide up the work per pixel amongst the different threads and communicate via shared memory data and explicit sync barriers but any attempts to parallelize a convergence loop significantly skew the results because it is an inherently sequential process.

While load balancing is still an issue because all of the pixels take varying amounts of iterations to converge, dynamic load balancing isn't an option with CUDA since we very explicitly divide up the work into fixed blocks and grids.

Overall, even with all of the constraints that the CUDA platform presents in terms of memory management and load balancing, it is still very well suited for highly data-parallel algorithms as seen when comparing speedups.

Comparative Analysis

Each of the models we explored has a unique approach to distributing the workload which affected both the algorithmic approach we implemented and our approach to parallelism. We found the shared address space to be better suited for the algorithm than message passing because of the small amount of communication necessary between threads in comparison to communicating large arrays between them in message passing. Although, the initial shared address space model had less arithmetic intensity, leading to poor performance. Due to this, we had to restructure the algorithm to implement a more computationally expensive flow algorithm. Message-passing allowed more freedom in terms of scheduling to accommodate the different convergence rates, however, the overhead with sending data and dedicating a thread to handle the dynamic schedule decreased overall performance. Although CUDA was very well suited for a data-parallel approach, it placed great limitations in terms of memory management in device memory and the number of threads per block to offset the large amount of local memory required by each thread.

Conclusions

Below we display a table of the best speedup results achieved with the models we implemented.

Implementation	**Speedup**	**Threads**
Message Passing - Asynchronous	~3.1x	8
Message Passing - Synchronous	~2.5x	8
Shared Memory Data Parallel Model	~9x	16
Shared Memory Task Parallel Model	~5.6x	16
CUDA	~31x	200 per block

Throughout the project, we determined limitations with the algorithm we decided to parallelize in terms of the opportunities for speedup. For optical flow analysis, there is a fixed order in which we need to handle computations on each frame, thus, leaving us only with the chance to parallelize per-frame computations unless we wanted to experience a loss of accuracy. Due to this, we aimed to identify opportunities within each frame to improve performance. The limitations of this algorithm are the lack of work for computing the updated coordinate of each pixel. Each step in this process was relatively short, so we found little benefit from parallelizing any of the image pre-processing or attempting approaches such as task-based parallelism. Rather, our approach for the project became more focused on how we could effectively schedule the updates to each pixel coordinate due to a large amount of variation in convergence rates. With the different models listed above, we analyzed different effects of dynamic work scheduling in order to better offset different pixel update rates.

References

Baker, S. & Matthews, I. International Journal of Computer Vision (2004) 56: 221. https://doi.org/10.1023/B:VISI.0000011205.11775.fd

Garcia-Dopico, A., Pedraza, J.L., Nieto, M. et al. Parallelization of the optical flow computation in sequences from moving cameras. J Image Video Proc 2014, 18 (2014) doi:10.1186/1687-5281-2014-18

Lakhtakia, Akhlesh, and Raúl J Martín-Palma. *Engineered Biomimicry*. Elsevier, 2013.

Plyer, Aurelien & Le Besnerais, Guy & Champagnat, Frederic. (2014). Massively Parallel Lucas Kanade Optical Flow for Real-Time Video Processing Applications. Journal of Real-Time Image Processing. 11. 1-18. 10.1007/s11554-014-0423-0.

Rojas, Ra'ul. “Lucas-Kanade in a Nutshell.” *Freie Universität Berlin*, www.inf.fu-berlin.de/inst/ag-ki/rojas_home/documents/tutorials/Lucas-Kanade2.pdf.

Work Distribution

The workload was distributed approximately 50-50.

Appendix

Implementation of asynchronous work scheduling on the master thread for reference -

```
for (int i = 1; i < numThreads; i++) {
	int flag = 0;
	if (sendsComplete[i] != NULL) {
		MPI_Test(&sendsComplete[i], &flag, &status);
	}
	if (sendsComplete[i] != NULL && flag == false) continue;
	pixel toSend[GRANULARITY];
	for (int j = 0; j < GRANULARITY; j++) {
		if (currPt + j < PTS_COUNT) {
			toSend[j] = *pixels[currPt + j];
		} else {
			toSend[j] = *pixels[0];
			toSend[j].index = PTS_COUNT;
		}
	}
	MPI_Isend(toSend, GRANULARITY, PIXEL_TYPE, i, tag, MPI_COMM_WORLD, &sendsComplete[i]);
	currPt += GRANULARITY;
}

for (int i = 1; i < numThreads; i++) {
	int flag = 0;
	if (recvsComplete[i] != NULL) {
		MPI_Test(&recvsComplete[i], &flag, &status);
	}
	if (recvsComplete[i] != NULL && flag == false) continue;
	pixel toReceive[GRANULARITY];
```

```
        MPI_Irecv(toReceive, GRANULARITY, PIXEL_TYPE, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD,
        &recvsComplete[i]);

        for (int j = 0; j < GRANULARITY; j++) {
                int idx = round(toReceive[j].index);
                if (idx < PTS_COUNT) {
                        pixels[idx]->x = toReceive[j].x; pixels[idx]->y = toReceive[j].y;
                }
        }
}
```