CS 338 Final Project: Calculating Optical Flow with the Lucas-Kanade Method using CUDA C

Dzung Pham

December 9, 2019

I. Introduction

Optical flow is the description of the motion of pixels between two consecutive scenes. In 2D, a flow is often represented as a vector with a horizontal and vertical component (figure 1), from which the direction and magnitude of the motion can be easily derived. The idea of optical flow was introduced in as early as 1950 by American psychologist James Gibson [1] to describe how humans and animals perceive motion. Today, optical flow has numerous important applications in fields such as robotics, object tracking, and action recognition.

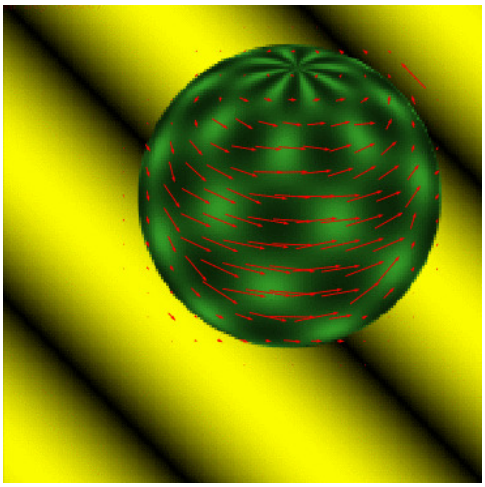

Figure 1: *Quiver plot of a sphere rotating right*

One of the earliest methods for calculating optical flow is the Lucas-Kanade algorithm, which was first introduced in 1981 [2]. Lucas-Kanade is a differential method that makes use of first-order spatial and temporal derivatives. Besides the *brightness constancy assumption*, which assumes pixels don't change their color as they move, Lucas-Kanade also assumes that pixels within a small neighborhood move in the same way. With these assumptions, Lucas-Kanade solves for the flow of each pixel using the Least Square method.

The data parallel nature of the algorithm makes it a perfect fit for parallelizing on a graphics processing unit (GPU). The platform of choice for this final project is CUDA – a programming framework for NVIDIA GPUs. Two important optimizations have been implemented: shared memory with tiling and pitched 2D memory. Varying block sizes and different tiling strategies have been experimented with to determine their impact on the execution time.

II. The Lucas-Kanade Method

1. Brightness Constancy Assumption

As mentioned earlier in the introduction, the algorithm makes the assumption that pixels don't change their color when they move. More mathematically, let $f(x, y, t)$ be a function that outputs

the value of a pixel at location (x, y) at time t, and suppose that at time $t + \Delta t$, the pixel has moved to location $(x + \Delta x, y + \Delta y)$. Then, the brightness constancy assumption is equivalent to the following equation:

$$f(x, y, t) = f(x + \Delta x, y + \Delta y, t + \Delta t)$$

We can approximate the right-hand side using the Taylor series expansion to the first order term, yielding:

$$f(x, y, t) = f(x, y, t) + \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial y}\Delta y + \frac{\partial f}{\partial t}\Delta t$$

Let $u = \Delta x$ be the horizontal flow component, $v = \Delta y$ be the vertical flow component, and $\Delta t = 1$ represent a discrete time step. In addition, let f_x, f_y and f_t represents the partial derivatives. Substituting these terms into the equation, we have:

$$f(x, y, t) = f(x, y, t) + f_x u + f_y v + f_t$$

or more compactly:

$$f_x u + f_y v = -f_t$$

2. Constant Flow Within Neighborhood

If we only have one brightness constancy equation for each pixel, then we would not be able to calculate the flow since there are two variables that need to be solved for. Thus, to deal with this problem, the next important assumption made by Lucas-Kanade is that for each pixel p, the pixels in the neighborhood around p will move in the same direction and with the same magnitude as p does (figure 2). For simplicity, let the neighborhood be an $s \times s$ square centered at p, where s is an odd integer ≥ 3. Then, what we end up with is a system of s^2 equations for the pixels in the neighborhood:

$$\begin{aligned} f_{x_1} u + f_{y_1} v &= -f_{t_1} \\ f_{x_2} u + f_{y_2} v &= -f_{t_2} \\ &\vdots \\ f_{x_{s^2}} u + f_{y_{s^2}} v &= -f_{t_{s^2}} \end{aligned}$$

Since this is an overdetermined system of equations ($s^2 > 2$), it is very unlikely that there exists u and v that satisfy each equation. However, it's possible to find u and v that minimize the error as much as possible using a technique called the *Least Square Method*.

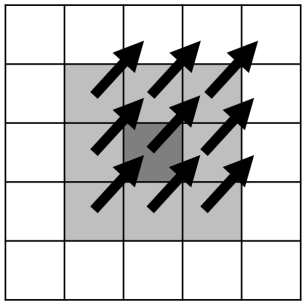

Figure 2: *Example of constant flow in a square of size 3*

3. The Least Square Method

Let's define the following matrices:

$$\mathbf{A} = \begin{bmatrix} f_{x_1} & f_{y_1} \\ f_{x_2} & f_{y_2} \\ \vdots & \vdots \\ f_{x_{s^2}} & f_{y_{s^2}} \end{bmatrix}, \ \mathbf{v} = \begin{bmatrix} u \\ v \end{bmatrix}, \ \mathbf{b} = \begin{bmatrix} -f_{t_1} \\ -f_{t_2} \\ \vdots \\ -f_{t_{s^2}} \end{bmatrix}$$

Then, the system of equations can be written as:

$$\mathbf{Av} = \mathbf{b}$$

To get $\mathbf{v}$, we could multiply both sides by the inverse of $\mathbf{A}$ at the front. However, $\mathbf{A}$ is not a square matrix, and so its inverse doesn't exist. This problem can be easily dealt with by first multiplying both sides by $\mathbf{A}^T$ (the transpose of $\mathbf{A}$), yielding:

$$\mathbf{A}^T\mathbf{Av} = \mathbf{A}^T\mathbf{b}$$

Now, since $\mathbf{A}^T\mathbf{A}$ is a square matrix, we can isolate $\mathbf{v}$ by multiplying both sides by $(\mathbf{A}^T\mathbf{A})^{-1}$, giving us:

$$\mathbf{v} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b}$$

Note that $\mathbf{A}^T\mathbf{A}$ must be invertible in order for the method to work. To check for this, we can

calculate the determinant of $\mathbf{A}^T\mathbf{A}$ and check if it's zero or not. If it's zero, then the matrix is singular and thus not invertible. Another way to check is calculate the eigenvalues of the matrix and make sure that they are all greater than zero or some threshold τ to reduce noise.

We now have:

$$\mathbf{A}^T\mathbf{A} = \begin{bmatrix} \sum f_{x_i}^2 & \sum f_{x_i} f_{y_i} \\ \sum f_{x_i} f_{y_i} & \sum f_{y_i}^2 \end{bmatrix}, \ \mathbf{A}^T\mathbf{b} = \begin{bmatrix} -\sum f_{x_i} f_{t_i} \\ -\sum f_{y_i} f_{t_i} \end{bmatrix}$$

The determinant of $\mathbf{A}^T\mathbf{A}$ is:

$$\det(\mathbf{A}^T\mathbf{A}) = (\sum f_{x_i}^2)(\sum f_{y_i}^2) - (\sum f_{x_i} f_{y_i})^2$$

and its inverse is:

$$(\mathbf{A}^T\mathbf{A})^{-1} = \frac{1}{\det(\mathbf{A}^T\mathbf{A})} \begin{bmatrix} \sum f_{y_i}^2 & -\sum f_{x_i} f_{y_i} \\ -\sum f_{x_i} f_{y_i} & \sum f_{x_i}^2 \end{bmatrix}$$

4. Taking Derivatives

The last pieces needed for the algorithm are the derivatives f_x, f_y and f_t. But what does it mean to take the derivative of an image with respect to coordinates and time? From the original definition of derivative, we have:

$$f_x = \frac{\partial f}{\partial x} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x, y, t) - f(x, y, t)}{\Delta x}$$
$$f_y = \frac{\partial f}{\partial y} = \lim_{\Delta y \to 0} \frac{f(x, y + \Delta y, t) - f(x, y, t)}{\Delta y}$$
$$f_t = \frac{\partial f}{\partial t} = \lim_{\Delta x \to 0} \frac{f(x, y, t + \Delta t) - f(x, y, t)}{\Delta t}$$

Since the domain of our function is discrete, we have $\Delta x = \Delta y = \Delta t = 1$. Thus, the derivatives above simplify to:

$$f_x = f(x + 1, y, t) - f(x, y, t)$$
$$f_y = f(x, y + 1, t) - f(x, y, t)$$
$$f_t = f(x, y, t + 1) - f(x, y, t)$$

The first two equations is just one of many different ways to calculate spatial derivatives. Less noisy discrete derivatives include the Sobel, Laplace and Prewitt operator, which simply convolve the neighborhood around each pixel with a small 3×3 or 5×5 matrix. For this project, the 3×3 Prewitt operator is the operator of choice:

$$f_x : \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}, \ f_y : \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

5. The Final Algorithm

To summarize, given 2 grayscaled images I and I' of size $h \times w$ and an odd window size $s \geq 3$, the steps of our Lucas-Kanade optical flow algorithm are as follow:

1. Initialize f_x, f_y and f_t to be $h \times w$ arrays for the derivatives
2. Calculate the entries of f_x and f_y by convolving with the Prewitt operator
3. Calculate the entries of f_t by taking the difference of I' and I
4. Initialize u and v to be $h \times w$ arrays for the horizontal and vertical flow components
5. For each pixel, look at the square of size s centered at the pixel, and calculate $\mathbf{A}^T\mathbf{A}$. If it's invertible (or its eigenvalues are all greater than a threshold τ), then calculate the flow $(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b}$ and assign to u and v
6. Return u and v

Note that the derivatives and flows at the edge of the image ($s/2$ pixels from the boundaries) can be simply set to 0 for simplicity.

III. Parallelizing with CUDA

This section describes the various implementation choices as well as optimization strategies for the Lucas-Kanade algorithm. For more detailed info on GPU and CUDA, visit NVIDIA's programming guide at: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

1. A Simple Implementation

Step 2, 3 and 5 of the algorithms can be easily parallelized with a GPU by simply assigning the work for each pixel to a GPU thread. Step 2 and 3 can be combined into a single derivative kernel, while step 5 will need to be done in a separate flow kernel because it depends on the results of step 2 and 3. (Very recently, with the appropriate GPU and CUDA, threads can cooperate on a grid-wide level, thus allowing all the steps to be combined into a single kernel and eliminating the need to launch separate kernels.)

The CPU will take care of reading, gray-scaling and linearizing the inputs, as well as visualizing and writing the outputs. In addition, the CPU will also handle step 2 and 3 since the computation per pixel for those steps is too low for the GPU to be able to provide any significant speed-up over the CPU, and the bottleneck lies mainly in step 5. (The decision to let the CPU take care of calculating the derivatives was made after implementing and profiling the derivatives GPU kernels.)

Since the derivative and flow steps both follow the convolution pattern, a tiling strategy with shared memory can be leveraged to maximize computation per memory read. In addition, because the data are 2D images, pitched memory can also be used to provide better memory alignment and consequently more efficient memory access. Due to the low amount of work for the derivative kernels, these optimization strategies are only applied to the flow kernel, which has varying convolution window size. The next two sections will describe the two strategies in more details.

2. Shared Memory with Tiling

Due to the high cost of reading from the global memory of a GPU, if each thread works independently of one another, then a pixel that is shared by multiple threads will have to be read into memory multiple times, one for each thread, thereby lowering the number of computations per memory. To alleviate this issue, all threads in a block can cooperatively load all memory needed into the GPU's shared memory, which provides much faster access than global memory does.

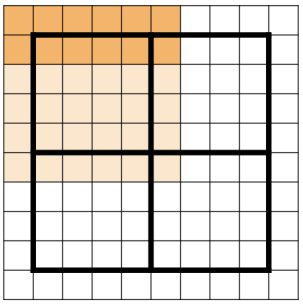

Figure 3: *Example of tiling on a* 10×10 *image with window size* 3 *and* 4×4 *output block. The bold black borders indicate the blocks. The light orange region are the memory needed to compute the top left block. The dark orange region indicates the tile.*

To be more concrete, consider the following example (figure 3): suppose the inputs are 10×10 images, the window size is 3 and the original block size in the simple implementation is 4×4. Then, the pixels needed for each block is a 6×6 square around each block. We can declare the amount of shared memory needed for each block to be $6 \times 6 \times$ size of a float $\times$ 3, where the number 3 comes from the fact that we have f_x, f_y and f_t. We could then let the blocks be 6×6 instead of 4×4 so that each thread can load the memory needed at their respective location. Once the memory is fully loaded, the threads can then calculate the flow.

A problem arises when the window size is so large that the number of threads in the above scheme exceeds the maximum number of threads per block. In this situation, the square block can be replaced with a smaller rectangular tile. Because 2D arrays are stored in row-major order, the tile

should have width equal to the square block's side, while its height should be smaller. Then, to load all the memory needed in the block, we will repeatedly load the pixels covered by the tile starting from the top and shift the tile downward appropriately so that we can load the next tile until we have covered the entire block. Then, starting from the top again and in the same manner, we can calculate the flow for each pixel in the tile. This tiling approach essentially assigns the work of multiple pixel to each thread, thus trading some parallelism for more efficient memory access.

To summarize and generalize the tiling strategy, given output block size b, window size s and maximum number of threads per block m, the steps involved are:

1. If $(b+s-1)^2 \leq m$, then let the block dimension be $(b+s-1) \times (b+s-1)$, else let it be $(\lfloor m/(b+s-1) \rfloor) \times (b+s-1)$. The number of tiles t will then be $b+s-1$ divided by the height of the block.

2. For $i = 0$ to t, load the pixels covered by the i^{th} tile.

3. For $i = 0$ to t, calculate the flows for the pixels covered by the i^{th} tile.

3. 2D Memory Allocation with Pitch

Currently when using 2D (or 3D) data with a GPU, we need to flatten/linearize the data into a single contiguous array. If we flatten the data "as-is", then issues with improper memory alignment might arise, since global memory must be accessed in 32, 64 or 128-byte transactions, and the memory must be aligned. Consider the following contrived example (figure 4): a 3×3 array of single-precision floats (4 bytes each), and memory transaction of 16 bytes. When the array is linearized, the second row would be spread across two 16-byte segments, and so when we try to access the contents of the second

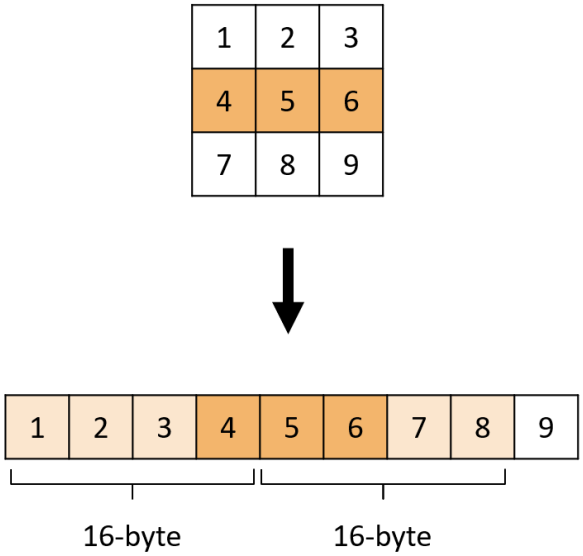

Figure 4: *Example of a 3×3 array of single-precision floats flattened into a 1D array without pitch. The second row (dark orange) is spread across two 16-byte segments (light orange)*

row, two separate memory transactions must be issued. This makes memory coalescing difficult, thereby reducing throughput.

To prevent this issue, we can instead pad each row of the 2D array so that the alignment requirement is met. Consider the above example again with 3×3 array padded (figure 5). Since the memory transaction is 16 bytes each, each array would be padded with 4 bytes at the end, giving the array a new width called 'pitch'. Now, when we access the second row, only one memory transaction is needed since the second row can fit entirely in a 16-byte segment.

Allocating and moving 2D array with pitch can be easily achieved by using the `cudaMallocPitch` and `cudaMemcpy2D` instead of `cudaMalloc` and `cudaMemcpy`. Note that when accessing entries of the padded linearized 2D array, instead of multiplying the row index by the original width of the array, we will have multiply it by the returned pitch from

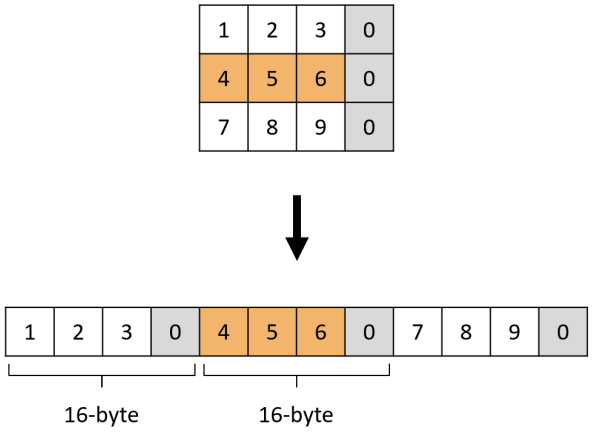

Figure 5: *Example of a* 3×3 *array of single-precision floats padded at the end of each row (grey) and flattened into a 1D array. The second row (dark orange) is completely contained in the second 16-byte segment*

`cudaMallocPitch`.

IV. Performance Results

This section evaluates the speed of the GPU implementation on various image and window sizes and with different optimizations. The data was gathered on a machine equipped with an I7-8750H chip and an NVIDIA GeForce GTX 1070, both running at stock speed. All results for the GPU were collected by NVIDIA's profiler `nvprof`, which allows for very precise timing of GPU kernels and CUDA API calls. Results for the CPU were collected using the standard library `time.h`. Note that only the time for calculating the flow is measured. In other words, memory allocation/release, flattening/grayscaling input, taking derivative, visualization, etc. are not taken into consideration because flow calculation is the most dominant factor. Each measurement is averaged over 10 runs.

1. Comparing Different Optimizations and Configurations for GPU

The execution time and speed-up over the simple GPU kernel with no optimization were measured

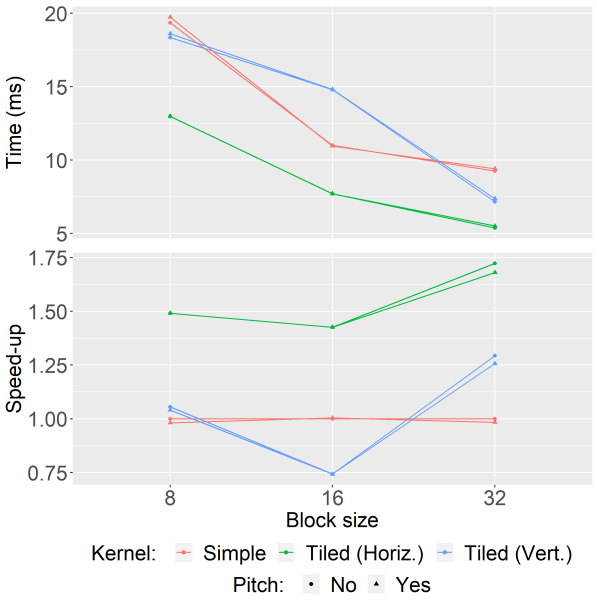

Figure 6: *Average time (top) and speed-up over simple GPU kernel with block size* 8 *(bottom) for each block size and optimization on the same* 1033×1033 *image and window size* 25

with 3 different block sizes, namely $8, 16, 32$, for no optimization, tiled, and pitched kernels. Vertical tiling (tile has height larger than width and moves from left to right) was also implemented and tested with to see how much worse it is compared to horizontal tiling and no optimization. The inputs were two 1033×1033 images of the sphere rotating right earlier seen in figure 1, with window size 25. (The odd dimension 1033×1033 was chosen specifically to cause memory misalignment issue and allow the the pitch kernel to do work.) The results are plotted in figure 6. For reference, the CPU took between 900-1000 ms to compute the flow, which is about 45-50 times slower than the worst time for GPU.

Overall, as block size increases, execution time and speed-up improve, which is not surprising. Horizontal tiling performs the best, with a speed up of about 1.3-1.75 compared to the no optimization kernel. Interestingly, not only does pitch not help,

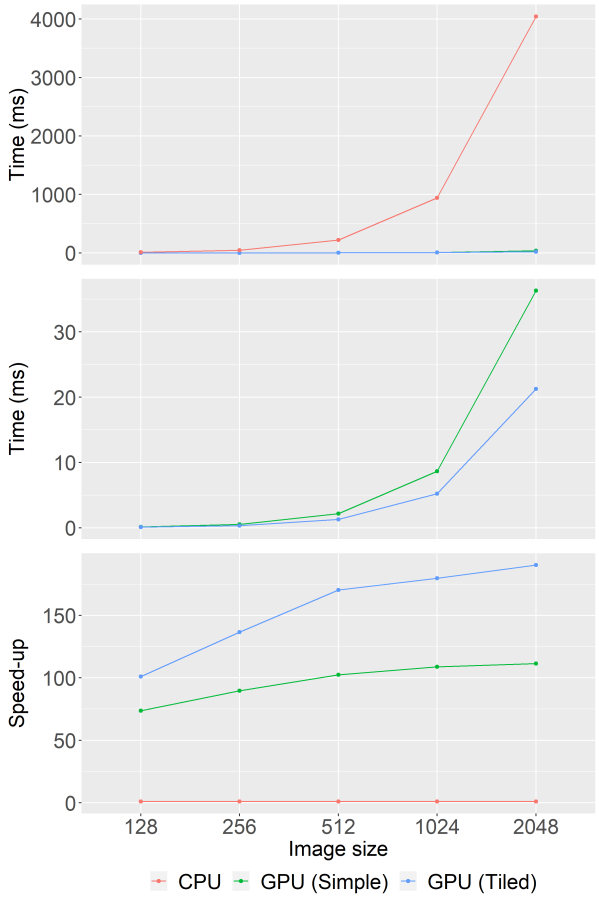

Figure 7: *Average time (top and middle) and speed-up over CPU (bottom) for CPU kernel, simple GPU kernel and tiled GPU kernel at each image size*

but it also slightly worsens the time and speed-up, which however is within the margin of error. The fact that it doesn't contribute to any gain is most likely due to the compute capability of the GPU tested on (6.1 for GTX 1070). It is suspected that with an older architecture/lower compute capability, the gain will be more evident. Lastly, vertical tiling only provides a small speed-up of around 1.25 at block size 32, while not really helping or even worsening the performance at smaller block sizes. This experiment emphasizes that without understanding the underlying architecture, improper tiling is as bad as no optimization.

2. Image Size Scaling

To see how CPU and the different GPU kernels scale with image size, the execution time and speed-up over CPU were measured on the sphere image with window size 25 and at varying dimensions: 128×128, 256×256, 512×512, 1024×1024, 2048×2048. Thus, the size of the image multiplies by 4 at each dimension step. The GPU kernels were run with 32×32 block since this configuration is the best. The results are plotted in figure 7. Pitch kernels are not included due to their lack of difference from the non-pitch kernels.

As expected, the CPU time scaled the worst, then comes the simple GPU kernel, and finally the tiled GPU kernel. The GPU kernels achieve speed-ups between 75 to nearly 200 times the CPU, and the speed-ups only increase with image dimension, with growth slowing down slightly at higher dimensions. The execution times for all of the kernels seem to multiplies by 4 as the image size also multiplies by 4. Furthermore, the time gap between the simple GPU kernel and the tiled GPU kernel appears to enlarge as image dimension increases.

3. Window Size Scaling

Lastly, to determine how CPU and GPU scale with window size, the execution time and speed-up over CPU were measured on the 1024×1024 sphere image at various window size from 5 up to 33 – the maximum window size possible for 32×32 blocks. The results are plotted in figure 8.

Unsurprisingly, the CPU is still the worst, then comes the simple GPU kernel, then the tiled GPU kernel. The time gap between the kernels only widens as window size increases. Speed-ups over CPU on the whole decrease with window size for both GPU kernels, but are still at least 100 times and 150 times faster for the simple GPU and tiled GPU kernel, respectively.

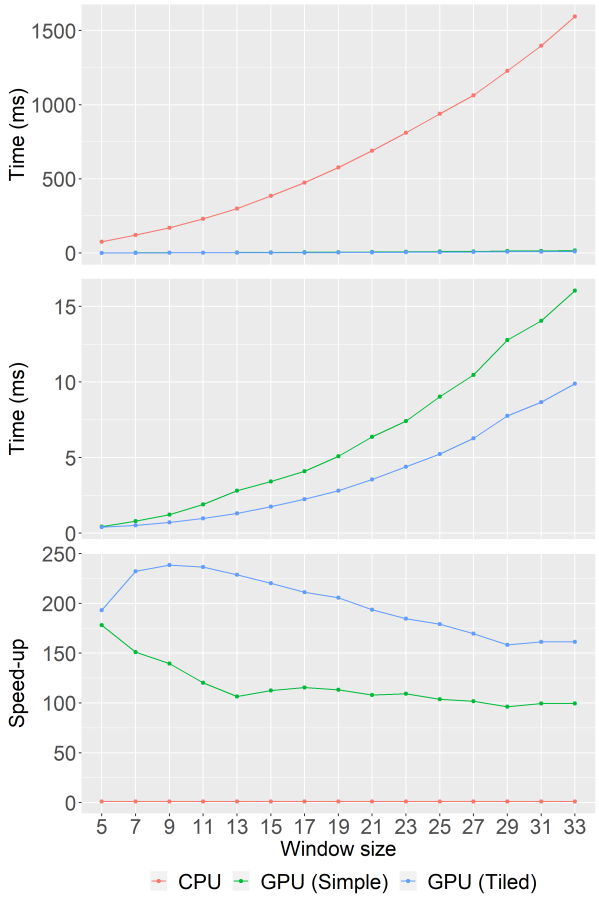

Figure 8: *Average time (top and middle) and speed-up over CPU (bottom) for CPU kernel, simple GPU kernel and tiled GPU kernel at each window size for the same* 1024×1024 *image*

Interestingly, the speed-up trend for the tiled GPU kernel is markedly different from that of the simple GPU kernel. Between window size 5 and 13, the speed-up for the simple GPU kernel decreases fairly rapidly from 175 down to around 100, while the tiled GPU kernel actually sees increased speed-up from a bit less than 200 to nearly 250 with window size from 5 to 9. As window size rises all the way up to 29, the tiled kernel experiences a gradual decrease in speed-up, while the simple kernel has only a slight drop. From 29 to 33, the speed-up seems to stop decreasing for both kernels.

V. Visualization

Optical flow can be visualized with quiver plots like in figure 1, or with HSV images. The most intuitive option is quiver plots, since they can immediately convey the direction and magnitude of the flow in a clear manner. However, due to the lack of easily usable plotting packages in C, HSV image will be used instead to visualize the flow. This section describes the specific steps needed to convert the optical flow to pixel value and to interpret the resulting visualization. Since this is not an essential part of the Lucas-Kanade algorithm, the CPU is used to handle the conversion of flow (HSV) to RGB.

HSV (figure 9) stands for Hue-Saturation-Value. It is a different representation of RGB color model that was designed to more closely match up with the way humans perceive color attributes. Hue is the colors arranged on a radial disk, saturation can be thought of as how much white paint is in the mix, and value can be thought of as how much black paint is in the mix. All of these attributes are in the $[0, 1]$ range, with the exception of hue, which can be rescaled to be in the range $[0, 360)$ to represent the degree of the radial color disk.

The direction of the optical flow can now be represented by Hue, while the magnitude of the flow can be represented by Value. Saturation can simply be set to 1. The specific formulas for getting Hue and Value from a flow (u, v) are:

$$H = \arctan\frac{v}{u},\ V = \sqrt{u^2 + v^2}$$

Once we got Hue and Value, we can now convert it to an RGB value. There are multiple conversion methods, some are self-explanatory but lengthy, while others are short but non-intuitive. To make the code lightweight, the following optimized method is used for input H and V between 0 and 1 and $S = 1$ (based on `http://www.chilliant.`

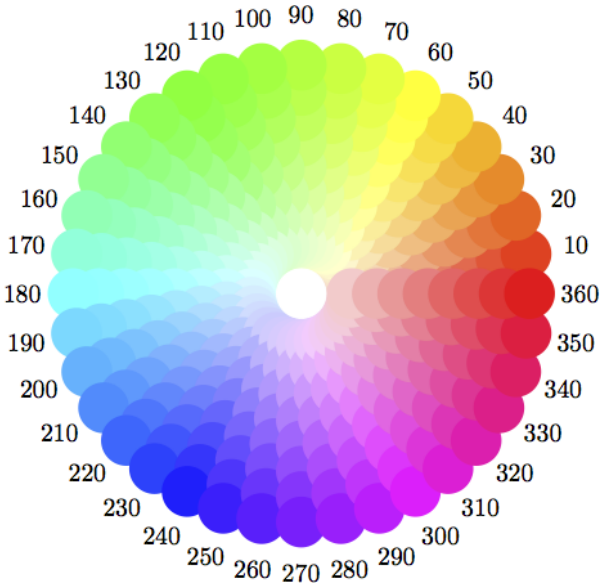

Figure 9: *The HSV color wheel*

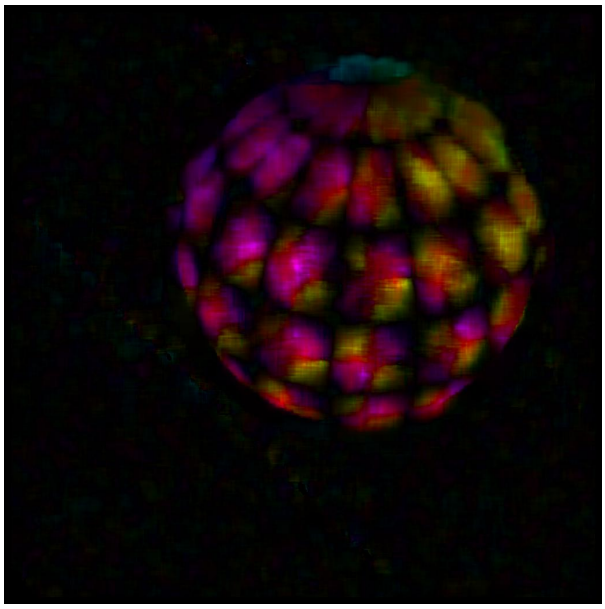

Figure 10: *HSV visualization of the optical flow of a sphere rotating right (window size = 25)*

com/rgb2hsv.html):

- $R \leftarrow \max(\min(|H * 6 - 3| - 1, 1), 0) \times V$
- $G \leftarrow \max(\min(2 - |H * 6 - 2|, 1), 0) \times V$
- $B \leftarrow \max(\min(2 - |H * 6 - 4|, 1), 0) \times V$

Figure 10 shows an example of the HSV visualization of the optical flow for the sphere rotating right in figure 1. The majority of the sphere has pink, red, and yellow colors, which, referencing the color wheel in figure 9, indicates movement to the right. Outside of the sphere is mostly black color, which indicates no/little movement.

VI. Numerical Considerations

To avoid overflow when calculating derivatives and optical flows, each pixel of the input images is scaled to be within the $[0 - 1]$ range by dividing by 255. Thus, the CPU and GPU must deal with floating point numbers. Due to the differences in the underlying architecture between CPU and GPU, there can be small divergences in the results calculated by the two devices. It has been observed that the resulting visualization pixels can differ, but only by at most one. As such, when testing the GPU's results against the CPU's results, a small tolerance of 1 for the difference is used. In addition, results from certain builtin math functions can also differ. For example, the `sqrtf` function used for taking square root of floats, on very small positive input, can give `nan` on the GPU, while on the CPU, it gives an actual number. Thus, robust handling of the results needs to be implemented. Lastly, floating point equality comparison might not always give the expected answer. This is done when checking if the determinant of $\mathbf{A}^T\mathbf{A}$ is 0 or not, so to deal with this case specifically, it's better to check if the absolute value of the number of interest is less than some reasonably small threshold (like 1e-5).

VII. Conclusion and Future Work

To sum up, implementing the Lucas-Kanade method on GPU with CUDA achieves a significant speed-up over CPU. Shared memory with tiling manages to boost the speed even further. 2D memory allocation with pitch doesn't seem to help with high compute capability devices.

The Lucas-Kanade version implemented is a single-pass algorithm, which only allows for detecting small local pixel motions. To enable large global motion detection, a strategy called Pyramidal Lucas-Kanade can be used instead [3]. The method recursively computes the flow for a shrunk version of the input and uses the resulting flow as guess for computing the flow of the larger version. Implementing this would be a helpful exercise for understanding optical flow, but would not require any different parallelizing strategy. Therefore, it will be left for the future. The implementation in this paper can be found at: `https://github.com/vietdzung/lucas_kanade_cuda`.

References

[1] Gibson, J. (1950). The Perception of the Visual World. Houghton Mifflin. 1

[2] Lucas, B. and Kanade, T. (1981). An Iterative Image Registration Technique with an Application to Stereo Vision. Proceedings of Imaging Understanding Workshop, pp. 121-130. 1

[3] Bouguet, J.-Y. (1999). Pyramidal Implementation of the Lucas Kanade Feature Tracker Description of the algorithm. Intel Corporation. 10