Final Project Report
FPGA Acceleration for Lucas-Kanade Optical Flow

Himanshu Chaudhary, Adithya Ramanujam, and Ketaki Jain.

I. Introduction

OPTICAL Flow technique estimates direction of apparent motion of points of interest in frames of a video sequence. Motion estimation techniques like optical flow are used in Computer Vision, Stereo vision, Robotics, Object tracking, motion detection, camera stabilization and obstacle avoidance applications. Refer Figure 1 and 2 for examples of Optical flow in action. Figure 1 shows the motion vectors overlayed over an frame of vehicles on a highway showing the direction of motion of the vehicles in the frame. Figure 2 shows the flow vectors of players walking along the game field. These motion vectors are converted into a color map which is also shown in the figure. This color map is used to identify regions of interest in the frames. i.e - It shows the parts of the image which are moving.

The Lucas-Kanade method [1] is a gradient-based algorithm involving computation of a 2D motion map by calculating gradients at every pixel of the frame. We are implementing the dense optical flow version of this algorithm which computes the motion all pixels in the image. Algorithms like Lucas-Kanade have a large volume of mathematical operations, are computationally heavy, and hardware intensive. Optical flow is used in a lot of time-sensitive applications and thus, needs to meet real-time guarantees.

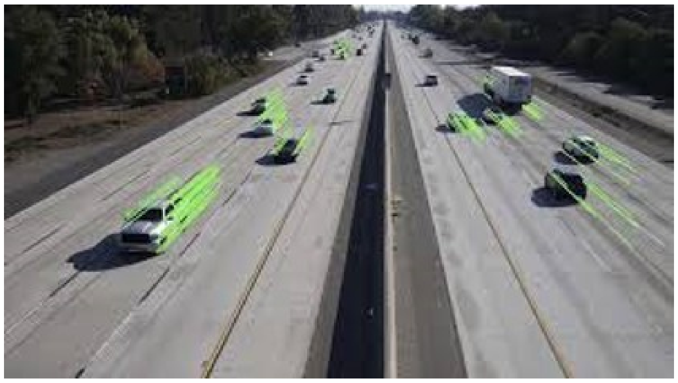


Fig. 1. Optical Flow in Action

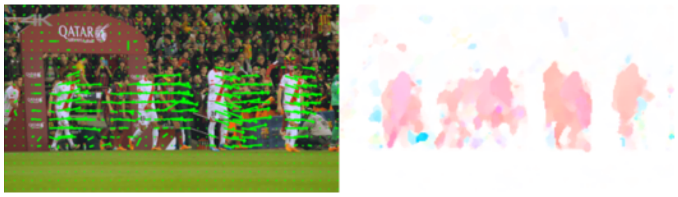


Fig. 2. Optical Flow Color Map

II. Problem Description

Through our project, we first analyze the working of the the Lucas-Kanade algorithm by implementing it in Python, generate a reference C code, and then use HLS to design an FPGA accelerator using Vitis HLS tool for the Lucas-Kanade algorithm. We perform Design space exploration to identify the best techniques to optimize for latency and resource utilizations. We then deploy our synthesized model to Pynq Z2 FPGA. Ensuring functional correctness, we compare the performance between the reference C code and the optimized code in terms of latency, repeatability and resource utilization. We also compare the speedup obtained on FPGA implementation over CPU implementation of the unoptimized C code.

III. Related Work

Works like Mahalingam et al. [2] discussed a detailed hardware pipelined dataflow allowing us to identify the different sections/blocks we would need to implement in our C implementation. Refer Figure 3 for the basic algorithm flow for Lucas Kanade. This work also discussed innovative methods to use floating point arithmetic to speed up the algorithm. Works like Seong et al. [3] also highlighted different bottlenecks involved in the Lucas Kanade algorithm. The MAC operations involved in the Least Square Matrix creation step and the division operation in the velocity compute and thresholding stages are generally the performance bottlenecks for FPGA implementations. In addition to these, the large number of memory operations makes Lucas Kanade hard to implement for real-time applications. Seon et al. [3] used an innovative interleaved subsampling and reconstruction technique to store image frames to reduce the amount of memory storage and increase speed of memory access by reducing the total amount of storage required. Gultekin et al. [4] used DMA controllers to speed up memory access and reduce the number of I/O operations. Xing et al. [5] and Allaoui et al. [6] introduce Simulink HDL implementations to design Lucas Kanade accelerators. These use a graphical interface as a HLS language to design FPGA accelerators. The major breakthrough was locating a application note by Xilinx [7], presenting a HLS-based implementation of the Lucas-Kanade algorithm. We used this as our starting point, and the developed the reference C code from the provided skeleton code as a reference.

Department of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, 30332.

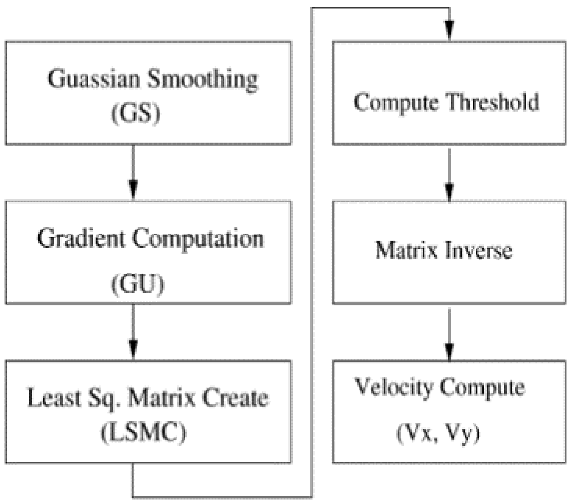

Guassian Smoothing (GS)
Gradient Computation (GU)
Least Sq. Matrix Create (LSMC)
Compute Threshold
Matrix Inverse
Velocity Compute (Vx, Vy)

Fig. 3. Optical Flow Algorithm Steps

IV. System implementation

We used 2 input frames for testing both the C and Python implementations. Refer figure 4 for the 2 input frames used. We run the optical flow algorithm in both implementations and plot the flow vectors computed as a color map, overlay it over frame 2 and warp the frame 2 by reversing the flow to create a motion compensated image to compare with frame 1 and calculate metrics like PSNR. (Peak Signal to noise ratio)

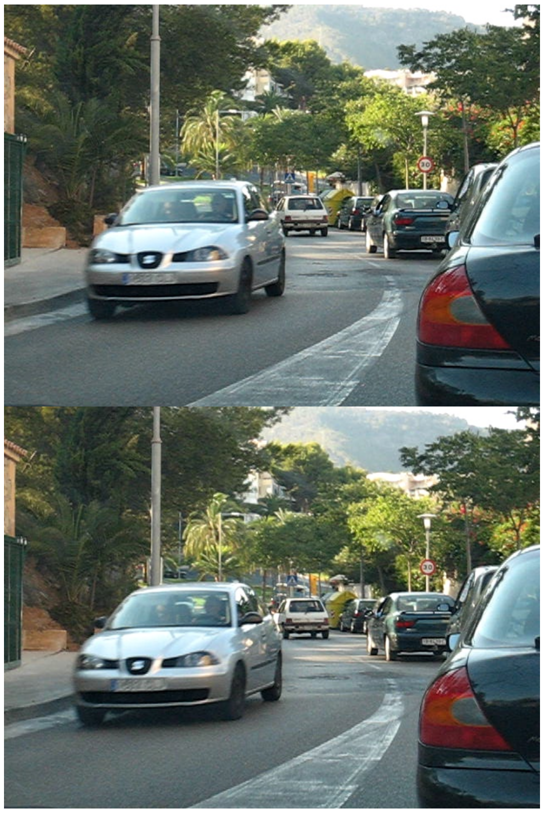


Fig. 4. Input Frames

A. Python implementation and testbench

We used Python with OpenCV libraries to create a optical flow implementation which we could use to get a ground truth for the input image frames.This served as a good model for visual inspection and observing the Lucas Kanade algorithm in action. OpenCV has an implementation of Lucas Kanade which is optimized using Shi Tomasi Algorithm to perform sparse optical flow calculations. This is done as Lucas Kanade by itself is very computationally expensive. Shi Tomasi is a Harris- Corner detection algorithm which extracts features of interest within the image over which Lucas Kanade is run.

Our problem statement is to perform dense optical flow measurement where we calculate the flow for all pixels within a image. OpenCV uses Farneback method [8] which is a much less computationally expensive algorithm for performing this calculation. We have tuned the Farneback implementation to best match the parameters used in our Lucas Kanade implementation and develop a ground truth image for optical flow for visual comparison. Refer figure 5 for the output of our python implementation. The 3 figures are the color map corresponding to the computed flow vectors, color map overlayed over the input frame and the warped (motion compensated) frame generated by reversing the flow vectors and adding it to the second input frame.

In addition, we also used MATLAB to create a test-bench to generate the optical flow color map outputs based on the flow vectors calculated using our reference C code. We used this to plot and evaluate results generated by the C code into a color map.

B. Reference C implementation

The reference design from Xilinx demonstrates implementation of Lucas-Kanade algorithm with Vivado HLS tool. The idea of our project is to realize real-time implementation of Lucas-Kanade algorithm on FPGA with Vitis HLS tool, thus this seemed to be a good starting point. We use the C code implementation of the algorithm as part of the provided solution as a reference to compare our HLS implementation in terms of speedup, correctness and resource utilization.

Since the reference code is incompatible with the Vitis HLS tool, as a first step, we converted this code into a normal C implementation by removing any tool dependencies so that it could be compiled with standard C/C++ compilers in a local environment.

Next, we tested the algorithm implementation on the input frames and produced the flow vectors which we passed into our python test-bench to generate color maps. Figure 6 shows the execution capture of this C reference code of Lucas-Kanade algorithm. The code also computes the PSNR value after performing the motion compensation to indicate the accuracy and correctness of Lucas-Kanade optical flow algorithm. As shown in Figure 6, the PSNR value is around 18dB which is similar to the PSNR value reported by the Xilinx reference design.[7] Also, the code has been updated to save the computed flow vectors and motion compensated frames on the disk in the form of text files. These text files are further used to import in Python environment for plotting of these vectors which helps

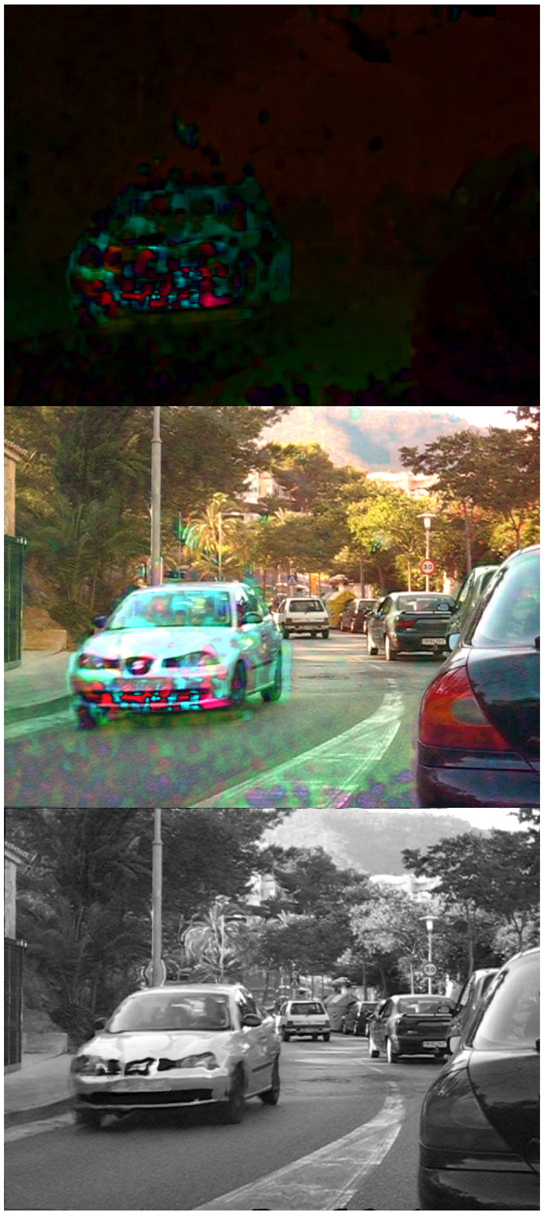


Fig. 5. Python Implementation Results

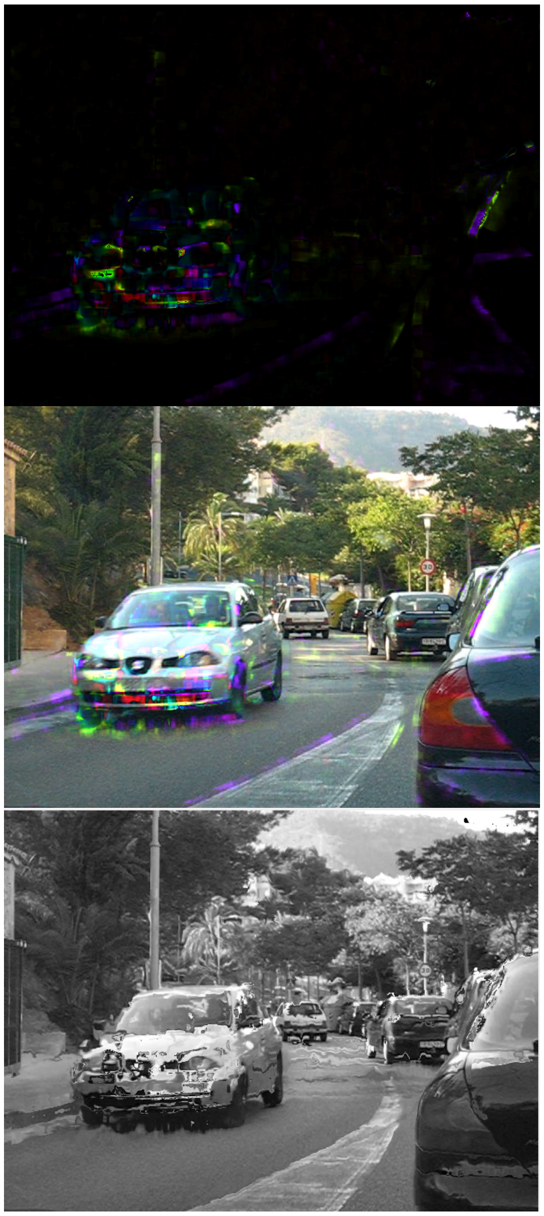
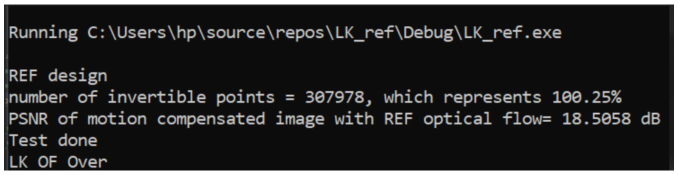


Fig. 7. C Implementation Results

with visual inspection of the algorithm correctness. Figure 7 shows the results of the C implementation namely the flow map, overlayed flow vectors over the input frame and the motion compensated frame.



```
Running C:\Users\hp\source\repos\LK_ref\Debug\LK_ref.exe

REF design
number of invertible points = 307978, which represents 100.25%
PSNR of motion compensated image with REF optical flow= 18.5058 dB
Test done
LK OF Over
```

Fig. 6. Execution capture of Lucas-Kanade C reference implementation

C. HLS Synthesis

We reworked the reference C implementation, to create our own FPGA-friendly version of the LK algorithm using Vitis HLS tool. The major modifications were along the lines of:

- Removal of variables in loop conditions
- DRAM to BRAM allocation for faster access
- Static allocation of arrays and avoiding variables where possible
- Simplifying loops and fixing loop bounds to get fixed latency values

We also reduced the size of our input images (100x100) to match it with the constraints of Pynq Z2.

D. Optimizations

Our reference implementation consists of a series of blocks - Isotropic filtering of the images, derivative calculations in the spatial and temporal dimensions, integration filter to generate Least square matrix coefficients and computation of flow vectors. These 4 steps are sequential and cannot be parallelized directly. The filter and derivative blocks use a kernel size of 5x5 while the integration block uses a integration window of 11x11 making optimization quite challenging. In order to accelerate this implementation, we looked at exploiting parallelism within each block independently and then pipeline the blocks. We also identified that the integration block is the bottleneck. Thus, we decided to put most of our optimization effort on that block. In general, we observed that the individual blocks are limited mainly by I/O operations. Due to resource limitations on the FPGA, unrolling and partitioning of the memory could not be done to an excessive degree. This made us focus on techniques like Dataflow and pipeline optimizations in order to obtain speedup without increasing resource utilization.

1) Optimizing Isotropic Filter – Dataflow: The Isotropic filter block performs blurring on both the image frames. The inputs to this function is the input image and a buffer to hold the filtered image. The 2 filtering operations are independent and hence, we could perform dataflow optimization here to effectively halve the latency.

2) Optimizing Derivative blocks – Dataflow: The derivative block consists of 3 functions. 2 of the functions takes the 1st input frame as input and calculates the derivative in the 2 spatial directions (Horizontal and Vertical). The inputs to these functions are the input image and a buffer to hold the filtered image. The third function computes the temporal derivative by taking both input frames as input. There is no direct dependency between the 3 functions except that some of the inputs are common. We created duplicates of the input buffer and used dataflow optimization to improve the performance of this block.

3) Optimizing Integration block – Dataflow: The integration block is run for combinations of the derivative block outputs to generate 5 coefficients for every pixel. This is essentially 5 MAC operations. This block is the bottleneck of the algorithm implementation. In order to optimize this block, we created copies of the input buffers to this block and apply the dataflow optimization. This greatly sped up memory access and gave a very good speedup. In addition, we also added loop tiling, partitioning and unroll optimizations to a limited degree and maximized speedup obtained from this block.

4) Optimizing Compute Vectors block – Loop Tiling: The compute vector block takes the outputs of the integration block and performs a matrix inversion operation. In order to optimize this block, we used loop tiling, partitioning and unroll optimizations.

E. FPGA deployment

We exported our final optimized design to Vivado, and successfully generated the bitstream and hardware setup file. This was then uploaded to a Pynq-Z2 FPGA. This required us to create a Jupyter Notebook testbench. It was observed that the FPGA executed the function in **84.75msec** which is very close to the expected latency from our synthesis results.

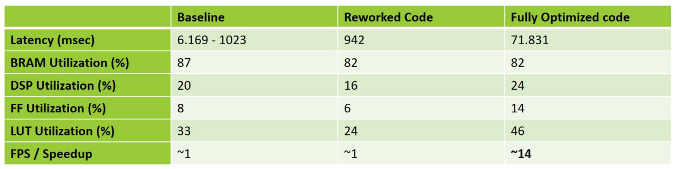
V. Results

A. Latency and Resource Utilization

Figure 8 shows the comparison of latency and resource utilization of 3 implementations:

- Baseline Implementation (Reference C code CPU implementation)
- Reworked implementation (Reworked code to make it FPGA friendly)
- Optimized implementation

Through the results, it can be seen that we managed to get a **14x** speedup without changing the resource utilization of the FPGA. Also it's important to highlight that there was not much increase in resource utilization with the high gain in speedup. Another interesting result was that the reworked implementation was able to give a msec improvement without adding any HLS optimizations.



	Baseline	Reworked Code	Fully Optimized code
Latency (msec)	6.169 - 1023	942	71.831
BRAM Utilization (%)	87	82	82
DSP Utilization (%)	20	16	24
FF Utilization (%)	8	6	14
LUT Utilization (%)	33	24	46
FPS / Speedup	~1	~1	**~14**

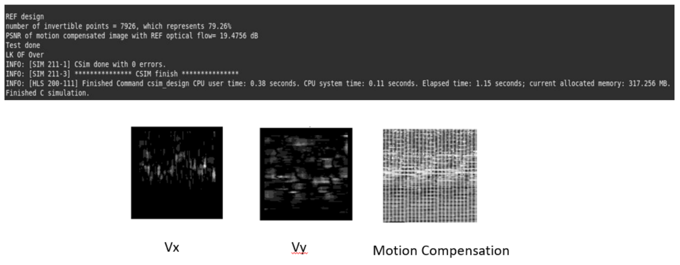
Fig. 8. Comparison of latency and resource utilization

B. Functional Correctness

To check the functional correctness, we used MATLAB to get a visual color map of our LK algorithm generated outputs. The outputs obtained are the flow vectors, and the motion-compensated image using the flow vectors generated. By taking mean square difference between the input and the motion-compensated image, we calculate the PSNR (Peak Signal to Noise Ratio) which gives a metric to analyse how well the optical flow was measured. In order to check the functional correctness of our optimized implementation, we run the same set of inputs on our reference C code and find the difference of the outputs generated by the reference implement ion and our optimized implementation. The absence of any remainder values indicate correct functionality. Figure shows the outputs of our FPGA implementation namely the flow vectors in the horizontal direction (Vx), flow vectors in the vertical direction (Vy), and the motion compensated image. The PSNR obtained was 19.156dB.

For the 100x100 image size, we found the outputs to have a significant noise. This is because our window size (11x11) is comparable to the image size (100x100) introducing a lot of noise. We verified this was functionally correct by comparing this output to the output generated by our reference C implementation for the same inputs.

We did not choose to reduce our window size as reducing this would have lead to even worse performance due to the aperture effect. This essentially keeps a minimum bound on window size and hence, the only way to improve accuracy of the optical flow calculation is to increase the image resoultion (input size). Our original input was 480x600 image which was resized into a 100x100 image. Due to the resource limitations on the Pynq Z2 FPGA, this was the maximum input size for which we could synthesize the model.



```
REF design
number of invertible points = 7926, which represents 79.26%
PSNR of motion compensated image with REF optical flow= 19.4756 dB
Test done
LK OF Over
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
INFO: [HLS 200-111] Finished Command csim_design CPU user time: 0.38 seconds. CPU system time: 0.11 seconds. Elapsed time: 1.15 seconds; current allocated memory: 317.256 MB.
Finished C simulation.
```

Fig. 9. Flow vectors and motion compensated output

VI. Future Work

The future work for this project could include dealing with higher resolution images and evaluating the effectiveness of our optimized implementation of the algorithm on a high resourced FPGA. Also, it would be interesting to explore the variable fixed point arithmetic for this implementation instead of relying on floating point as it enables an opportunity of further speedup which loosing much on the accuracy. And there are some other HLS optimization techniques that could be explored to further improve the throughput of algorithm.These include burst mode, wider bus transactions, streaming buffers and adoption of filter coefficients as in powers of 2.

References

[1] B. D. 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 - Volume 2*, IJCAI'81, (San Francisco, CA, USA), p. 674–679, Morgan Kaufmann Publishers Inc., 1981.

[2] V. Mahalingam, K. Bhattacharya, N. Ranganathan, H. Chakravarthula, R. R. Murphy, and K. S. Pratt, "A vlsi architecture and algorithm for lucas–kanade-based optical flow computation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 1, pp. 29–38, 2010.

[3] H.-S. Seong, C. E. Rhee, and H.-J. Lee, "A novel hardware architecture of the lucas–kanade optical flow for reduced frame memory access," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 26, no. 6, pp. 1187–1199, 2016.

[4] G. K. Gultekin and A. Saranli, "An fpga based high performance optical flow hardware design for computer vision applications," *Microprocessors and Microsystems*, vol. 37, no. 3, pp. 270–286, 2013.

[5] X. Xing, Y. Yongjie, and X. Huang, "Real-time object tracking based on optical flow," in *2021 International Conference on Computer, Control and Robotics (ICCCR)*, pp. 315–318, 2021.

[6] R. Allaoui, H. H. Mouane, Z. Asrih, S. Mars, I. El Hajjouji, and A. El mourabit, "Fpga-based implementation of optical flow algorithm," in *2017 International Conference on Electrical and Information Technologies (ICEIT)*, pp. 1–5, 2017.

[7] D. Bagni, P. Kannan, and S. Neuendorffe, Feb 2017.

[8] G. Farnebäck, "Two-frame motion estimation based on polynomial expansion," in *SCIA*, 2003.