GPU IMPLEMENTATION OF VIDEO ANALYTICS ALGORITHMS FOR AERIAL IMAGING

A Thesis presented to

the Faculty of the Graduate School

at the University of Missouri

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

---

by

EVAN TETERS

Dr. Kannappan Palaniappan, Thesis Supervisor

MAY 2023

The undersigned, appointed by the Dean of the Graduate School, have examined the dissertation entitled:

GPU IMPLEMENTATION OF VIDEO ANALYTICS ALGORITHMS FOR AERIAL IMAGING

presented by Evan Teters,

a candidate for the degree of Master of Science and hereby certify that, in their opinion, it is worthy of acceptance.

Dr. Kannappan Palaniappan

Dr. Hadi AliAkbarpour

Dr. Michael Byrne

ACKNOWLEDGMENTS

While my Masters took a year full-time and 3 years all-said, it was the culmination of my undergrad work that started all the way back in the Spring of 2017. Many thanks to Dr. Palaniappan and Dr. Hadi for their willingness to work with me back then, and all the way till now. Their guidance and help has been invaluable, and I wouldn't have accomplished without them. Thanks of course also go to the many members of our lab who helped me along the way! Thanks to Drs. Frasier, Shizeng, and Yao for their depth of expertise in this field. Thanks to fellow students Jaired Collins and Tim Krock, for their feedback on this thesis and unnumerable other times they were able to help me with whatever problems I was having with this work. Of course, significant gratitude to PhD candidate and now Doctor Rumana Aktar, whose impressive dissertation made this work possible. Thanks also to Nafis Ahmed who worked with me on speeding up NCC and doing rigorous testing. This work would not have gotten as far as it did without Dardo Kleiner, whose significant expertise with Gstreamer saved this project at several times when I thought I had reached an immovable obstacle. And of course thanks and love to my friends and family, for their patience and support as I spent countless nights on the work that culminated in this thesis.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

ABSTRACT

This work examines several algorithms that together make up parts of an image processing pipeline called Video Mosaicing and Summarization (VMZ). This pipeline takes as input geospatial or biomedical videos and produces large stitched-together frames (mosaics) of the video's subject.

The content of these videos presents numerous challenges, such as poor lighting and a rapidly changing scene. The algorithms of VMZ were chosen carefully to address these challenges.

With the output of VMZ, numerous tasks can be done. Stabilized imagery allows for easier object tracking, and the mosaics allow a quick understanding of the scene. These use-cases with aerial imagery are even more valuable when considered from the edge, where they can be applied as a drone is collecting the data. When executing video analytics algorithms, one of the most important metrics for real-life use is performance. All the accuracy in the world does not guarantee usefulness if the algorithms cannot provide that accuracy in a timely and actionable manner.

Thus the goal of this work is to explore means and tools to implement video analytics algorithms, particularly the ones that make up the VMZ pipeline, on GPU devices–making them faster and more available for real-time use. This work presents four algorithms that have been converted to make use of the GPU in the GStreamer environment on NVIDIA GPUs. With GStreamer these algorithms are easily modular and lend themselves well to experimentation and real-life use even in pipelines beyond VMZ.

Chapter 1

Introduction

1.1 Problem Statement

As edge devices become more powerful, there is an opportunity to take advantage of that power. With the onboard GPUs available, current workflows can potentially be moved from ground stations to the edge, to increase real-time capabilities with video analytics algorithms. With aerial imagery there is a special need for this, thanks to the large size of images and depth of algorithms needed to handle complex problems. Aerial imagery can come with a variety of problems that need solving. To deal with these problems we can deploy parallel devices that relieve the ground station to focus on processing and taking action based on information received from the algorithms run on these devices..

There is also a need for these real-time algorithms to be modular. This makes it easier for edge-to-ground processing because the outputs are passed generically. This also makes it easier for quick experimentation and real-time updates, because modules can be swapped in and out or configured differently. GStreamer provides a video-pipelining API for doing just this, and makes the GPU algorithms developed

much more useful.

The algorithms in this work were chosen for their ability to be parallelized, and their usefulness in aerial image processing. Most of them can be found in the pipeline for Video Summarization and Mosaicing, and many of the examples throughought will make reference to this use case.

1.2 Challenges

When collecting aerial imagery, there are a variety of possible problems due to the environment and medium. The lens can become dirty or a lack of careful preparation can cause a loss of focus. Lighting effects can cause image artifacts and a lack of light can reduce the amount of information the sensor can pick up, reducing image quality. All of these conditions degrade the quality of the images themselves. Also, the aerial platform will likely be shaking in the air. Without compensating for this, this provides further challenge to assumptions that could otherwise be made about stable video. We also have to account for the overall movement of the platform since frames will likely be observing the same scene from multiple angles.

When the use case with aerial imagery involves object detection or tracking, there is also movement relative to the ground that must be considered. The target can be moving on its own, potentially growing larger or smaller relative to each frame. It can also move into shadows or behind objects, causing difficulty with occlusion. When using feature detection, some important features may not be visible at certain angles of the camera or even movements of an object.

Aerial imagery presents a variety of challenges on its own, but so too does embedded device GPU programming. The device has limited space and resources. Memory leaks can significantly slow the system down. Many edge devices focus on low power GPU and CPU architectures such as ARM, which come with occasional limitations

as well. Also, there is an increased focus on using GPU programming due to a limited capability of the CPU.

1.3 Literature Review

To the best of our knowledge, there is little existing work focused on using GStreamer for aerial applications. However there is a large body of work focused on UAV based on-board processing with low-power embedded systems. Vega et al. present a hierarchical method to split Video Summarization tasks on the device and on the ground to minimize the impact of air-to-ground communication latency. They show that video summarization solely on the device or on the ground underperforms their proposed hierarchical split. They test on the VIRAT image dataset (VIRAT is explained more in chapter 5.1) with a Jetson TK1, which achieves a maximum of 42 frames per second, with an average closer to 10 [1]. This shows that porting modules of VMZ to GStreamer will help improve performance, because they can easily be split across devices if needed.

There is also a significant corpus of work dedicated to video summarization. An early work by Brown and Lowe developed image panoramics using invariant SIFT features and matching, as well as RANSAC for homography estimation. They also use bundle adjustment as the sequence continues to reduce the accumulation of error, and finally they apply blending algorithms for cleaner panoramics. This showed best results on imagery of stationary scenes from a camera rotation point [2]. Goncalves et al. developed an automatic algorithm for image-to-image registration [3]. Trinh et al. developed a video summarization pipeline in 2012 that used feature detection and matching to generate homographies [4]. Tao et al. developed a graph-based greedy algorithm that took advantage of temporal continuity to group frames together as smaller panoramics and then connect them with cross-group relationships. This

was a fairly slow pipeline, taking 15 minutes for 48 images [5]. In 2015 Viguier et al. developed a more robust basis for a pipeline that generates mini-mosaics as an intermediate step to final video summarization between these mini-mosaics [6]. The reliance on fast, low-level image based algorithms ensured better performance and runtime. This work was continued by Rumana Aktar, culminating in her 2022 dissertation on video summarization and mosaicing (VMZ) [7] that will be explored more in the next section, 1.4.

Hadi et al. described a method using bundle adjustment that takes advantage of available camera metadata to determine the homography transformation to the ground [8]. Thus, traditional image matching means are avoided and the speed and accuracy is high. Unfortunately, the availability of this metadata cannot be relied upon. Also, the bundle adjustment requires significant preprocessing before application. Still, Teters et al. built upon this method and demonstrated how quickly embedded systems could accomplish image registration and warping [9], laying early foundation for this work.

There is some literature on speeding up and improving the features detection and matching used in this work. Lewis [10] proposed a method to speed NCC up using the pre-computed integral histogram, which many standard implementations (including the ones we use) take advantage of. Briechle and Hanebeck demonstrated NCC for template matching that achieves less computation by approximating the numerator [11]. In 2009, Yoo and Han introduced a significantly faster signal-processing based method that takes advantage of logic operations to compute NCC without any multiplies [12]. This method is more sensitive to noise, however. Gangodkar et al., parallelized the Fast NCC (which is based on pre-computed sum-tables to mitigate the computational complexity of conventional NCC) on CUDA-based GPU[13]. NCC is parallelized across FPGA's by Wang and Wang, although the implementation is hardware-specific [14]. They do show the benefit of parallelizing the computation,

as our GPU implementation does. In work on a tracker by Jakob Santner et al., they showed good results by also using the OpenCV NCC for the static part of their tracker, as it gives strong cues when the target reappears [15]. They do not analyze or parallelize the NCC, however. Arunagiri and Jaloma demonstrated that for stereo matching, a NCC based cost function is fastest and consumes less energy when implemented using integers rather than floating point numbers on the GPU [16]. They do not try other methods of parallelization or focus specifically on NCC, however. Fouda and Ragab parallelized the NCC by using OpenMP [17] for shared-memory systems, but did not explore the parallelization on GPUs, in contrast to our work.

The foundational ASIFT paper by Morel and Yu [18] is important for our work. The simulated rotation and translation aids in matching aerial reference frames, as discussed in chapter 3.3. Another work by Wang et al. explores an image registration algorithm that uses ASIFT to do an initial registration followed by adaptive normalized cross-correlation to refine the ASIFT feature points. While this reduces the RMSE of matching points, they find a significant (undisclosed) performance decrease due to the increased complexity of running NCC for each match, even for small images. In [19] the authors propose a multi-stage matching methodology that uses RANSAC to find geometric transformations with SIFT points. The idea is similar to ASIFT, and the results show better RMSE performance. However, this method is more complex, and the code is not available in open domain.

1.4 Video Mosaicing and Summarization

This work adapts the following algorithms to GPU: structure tensor feature detection, normalized cross-correlation, affine-invariant SIFT, and georegistration. They are robust methods appropriate for aerial imagery. Taken together, the algorithms make up parts of the Video Mosaicing and Summarization (VMZ) pipeline as seen in Figure

1.3. This graphic is taken from [7] where it is demonstrated how VMZ can take an aerial video as input and produce a stitched-together composite representation of the area. It uses the algorithms listed above as well as some additional logic to tie them all together.

VMZ uses the concept of "reference frames" to analyze the video sequence. At the beginning, the first frame is marked as the first reference frame. This is used as a base frame to build the first mini-mosaic. Subsequent frames are then compared against this reference frame. Feature points on both the reference and the current frame are matched to find the transformation required to warp the current frame to the plane of the reference frame. Then, a blending algorithm is used to ensure the mosaic results are uniform and without any of the original image boundaries. When a tranformation warps an image outside the predefined bounds of a mini-mosaic, the current frame is set as the next reference frame for further processing. This is now part of the current mini-mosaic, and robust image matching is used to match reference frames together to achieve transformations of the current frames with respect to the original base frame. A different criteria will define when it is time to start generating a new mini mosaic with a different base frame (reference frame 1).

Figure 1.2 depicts the reference frame system, and Figure 1.3 shows the entire VMZ pipeline, highlighted our work with a purple box. It also shows that canvas estimation is not intended to be fully functional as part of this work. We simply write the mini-mosaics to a large black frame. Also, only warping is implemented, not blending. Finally, it shows that this work still has further improvements to be made for the Current-to-Reference calculations, as they still generate a certain amount of error which results in erroneous mini-mosaics.

Previously, VMZ has been implemented in Matlab and Python. This work attempts to recreate the components of VMZ with C++ to get closer to real-time processing and enable GPU computing on embedded devices. To run on the target

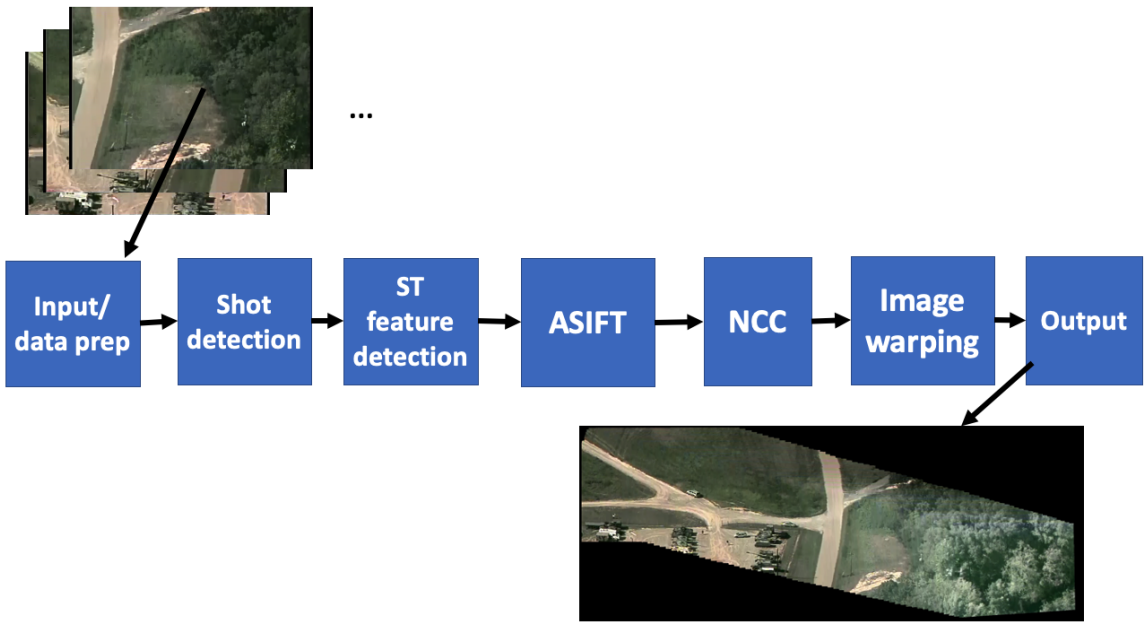


Figure 1.1: This figure shows the process of breaking a video into shots to be processed into mini-mosaics. For each shot, there is a base frame and several reference frames used for matching and warping.

device, our focus was the smaller subset of algorithms as shown in Figure 1.1 which fit into the modules shown in Figure 1.3.

1.5 Contributions

- The first contribution of this work is to accomplish a method for writing GPU-based modules of GStreamer that run on NVIDIA devices.
- A fast GPU-enabled implementation of the structure tensor algorithm that also exposes homogeneous points of the image as features.
- Four modular programs that accomplish video processing algorithms (ST, NCC, Georegistration, and ASIFT) while using NPP, OpenCV, and other libraries.
- Results with these algorithms showing how real-time GPU processing is possible on embedded devices.

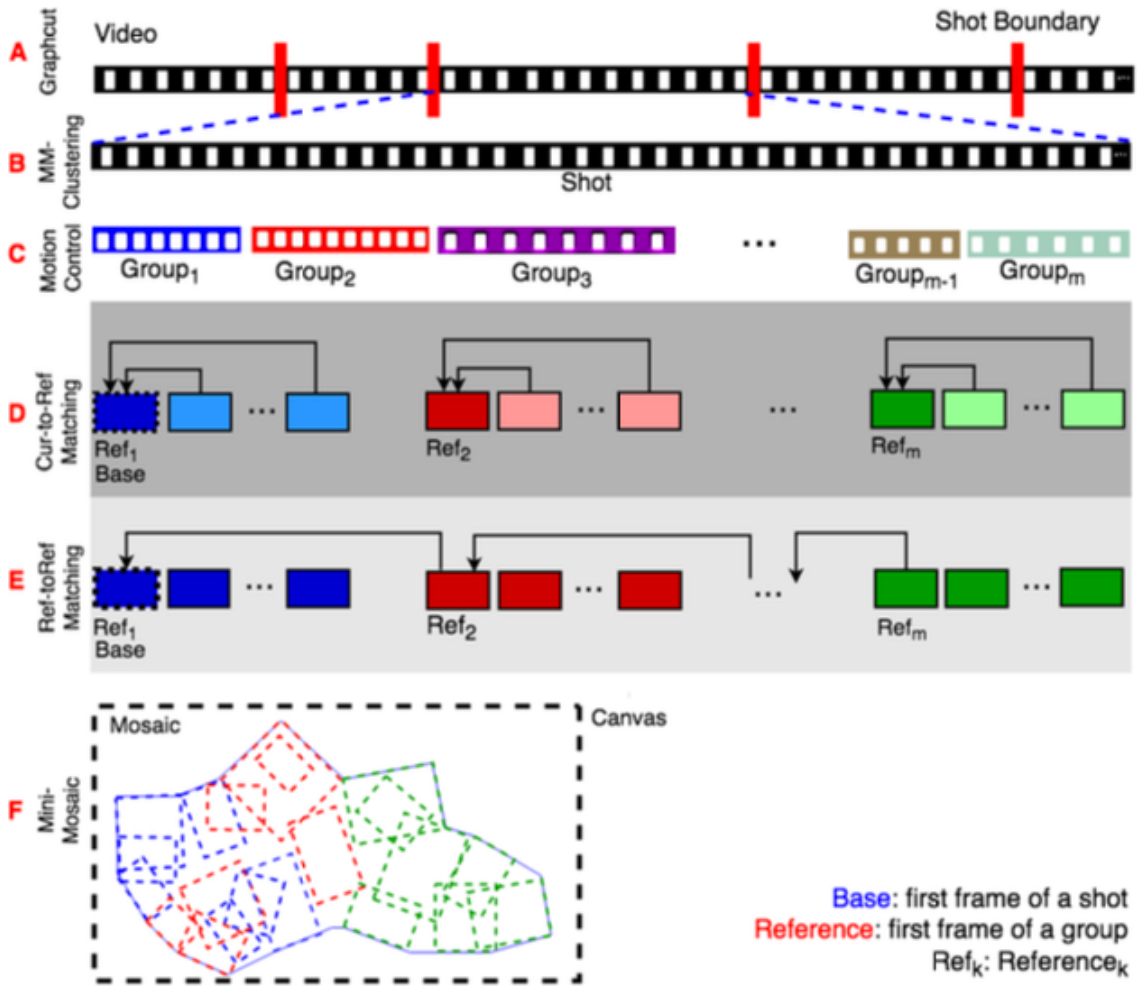


Figure 1.2: This shows each of the algorithms worked on for this thesis, and how they can fit together to produce mosaics. This pipeline is called Video Mosaicing and Summarization [7]

- Implements a GPU based integral histogram and weighted integral histogram with state-of-the-art performance on embedded systems.

1.6 Organization

The rest of this thesis is organized as follows: Chapter 2 discusses GStreamer and why it is the basis for the modules presented in this thesis. Chapter 3 delves into the image feature and matching algorithms that were rebuilt for GPUs by this work, as

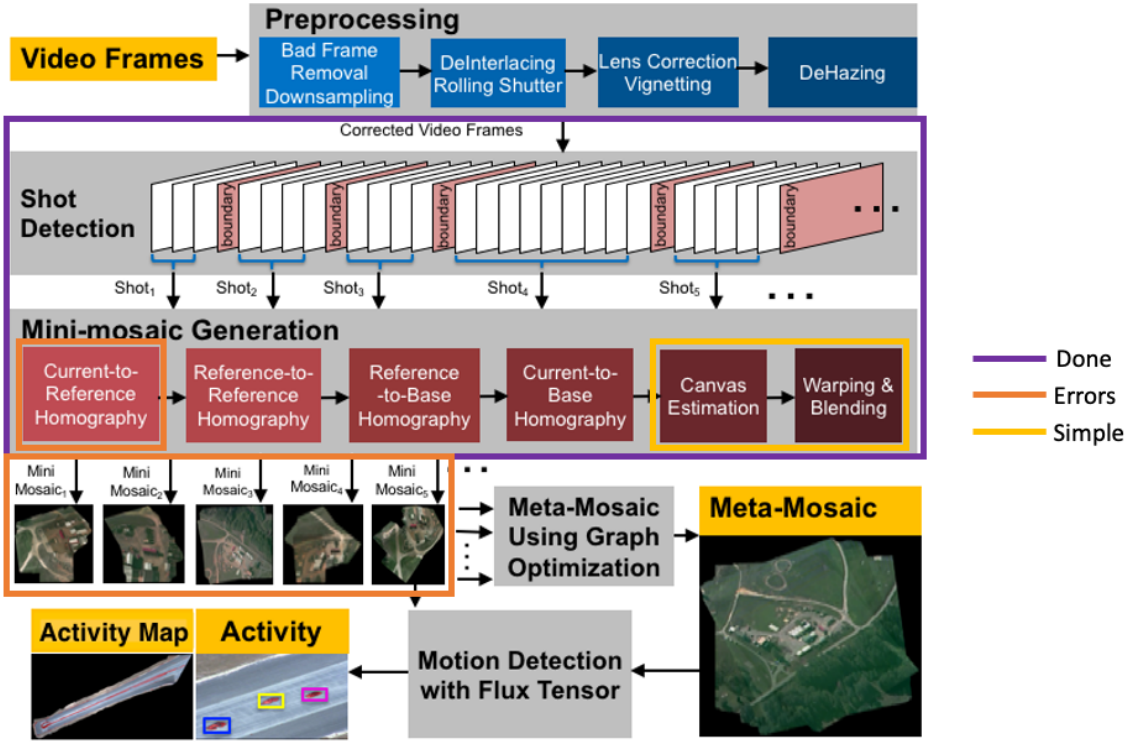


Figure 1.3: This shows the original diagram from the dissertation here [7] and where this work recreates the functionality (circled in purple). Also circled in yellow are two features with limited or basic functionality compared to the original work. Circled in orange represents where there are errors, or improvement is needed.

well implementation details of each that allow them to be performant and modular. Chapter 4 explains how the image feature detection and matching can be used to accomplish image registration and warping, or transformations that map images to a common plane. Chapter 5 enumerates experimental results of each module and of the modules working together. It shows how the results of VMZ can be accomplished with these basic building blocks. Chapter 6 details PyVMZ, a Python implementation of VMZ that has been modified to work with the NCC module from this thesis, followed by conclusions and future scope of our work in Chapter 7.

Chapter 2

GStreamer and NVIDIA DeepStream Frameworks

The implementations of the VMZ modules in this work are built in Deepstream, which is a Gstreamer library built by NVIDIA. To better understand the modules, this chapter explains Gstreamer and the Gstreamer library Deepstream.

2.1 GStreamer

GStreamer is a multi-media pipelining framework that allows the user to construct modular programs that can be tied together to accomplish tasks by each acting on data that is fed through modular elements in sequence. Typically this looks like Figure 2.1. A source element provides the data, usually a data stream like a video. A filter, or a number of filters, applies changes to the data. Then a sink element performs some kind of output operation, like writing to an output GUI window or writing to files.

The modularity has tremendous benefits for making experimentation and debugging easier. For experimentation, elements can easily be moved around or param-

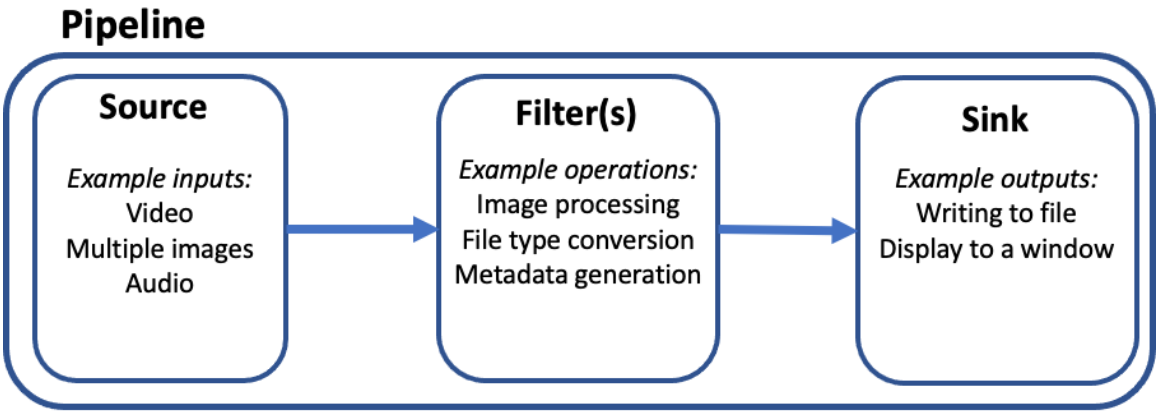


Figure 2.1: A demonstration of the different kinds of GStreamer elements and how they fit together. The source acts as input, the filters all act on the input, and the sink acts as output.

eterized in different ways to get different effects. For debugging, the engineer can separate out modules and verify the output with controlled input.

It supports a wide range of audio and video formats, and allows for the format to change from input to output along a pipeline. But for the purposes of this work, we are mostly concerned with video-in, video-out where the true output is saved to files as part of one of the elements.

In GStreamer, programs that make up the above blocks of the diagram are called elements, and they are the core of the framework. An application, or pipeline, consists of many elements chained in sequence, sometimes even branching off and chained in parallel. GStreamer is inherently multi-threaded, and elements in an application are constantly negotiating over when to run.

An element has pads which represent an element's input and output. There are source (output) and sink (input) pads. An element is represented by what its pads are capable of, and elements can only be linked up if the output pad capabilities match with the input capabilities of the next one. Shortened to caps, GStreamer refers to this process as caps negotiation.

To find an element's caps, the command gst-inspect-1.0 can be used from the

terminal. It will display documentation associated with an element, its input and outputs caps, and also show the arguments of element.

With GStreamer, a programmer can accomplish a wide array of tasks with just the base GStreamer elements. However, for more custom behavior, the programmer can write their own elements as well. After writing an element, its details can be queried on the command-line with gst-inspect-1.0.

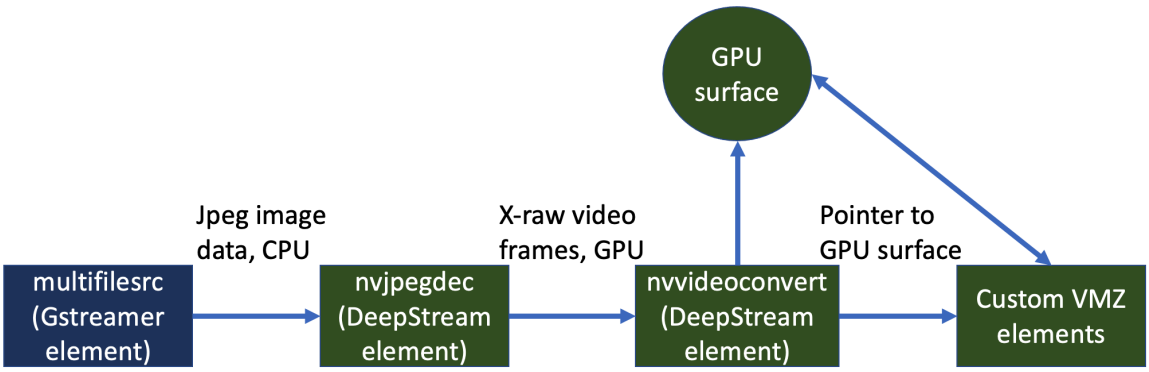
In general, data flows through the application one element at a time, in one direction. This is represented by buffers of data, and events.

In this work, the algorithms implemented are using a framework built on top of GStreamer called DeepStream. DeepStream is built by NVIDIA, and is a great example of the customizability and flexibility of GStreamer.

2.2 DeepStream

GStreamer is especially useful in this work's use-case because of NVIDIA's effort putting together a set of elements collectively labeled DeepStream. These elements are made specifically to run on certain NVIDIA GPUs and embedded devices in AI pipelines. Some DeepStream elements have the capability to run AI models, but that is not what I am seeking to use for my purposes. Here, I am interested in the supporting DeepStream elements, that allow for reading and working with video on embedded devices while efficiently transferring GPU buffers to other elements. With this, I can make my own modular elements that work with GPU data.

This makes it possible to run optimizations on the VMZ pipeline that involve the GPU in a much more efficient manner. I can also take advantage of another set of NVIDIA APIs, the NVIDIA Performance Primitives (NPP) to ensure primitive operations in these algorithms are being performed as efficiently as possible in these elements, without ever needing to transfer image data off the GPU.



DeepStream Element	nvjpegdec	nvvideoconvert	Custom VMZ elements
Function	Decode jpeg input for DeepStream, Outputs x-raw video frames in CUDA memory	Call nvbufsurftransform to generate a GPU surface to work with	Un-map GPU surface pointer, Do work (VMZ operations) with GPU buffer

Figure 2.2: The DeepStream elements generate GPU surface buffer and pass the pointer along to be used by subsquent elements. The elements created for this work are the the final box in the diagram, using that pointer to reference the GPU data quickly and efficiently.

DeepStream elements work just like GStreamer elements because they are GStreamer elements. The programmer can still use regular GStreamer elements to help with saving output, setting up parallel streams, and anything else once processed by DeepStream-specific elements.

2.2.1 DeepStream Foundational Code Layer

NVIDIA has set up the modules in such a way that they pass around pointers to an optimized GPU surface that their modules are built to work with. Because we want to work with them as regular GPU image buffers, we have to do some conversion like so:

With this frame and its pitch, the image processing modules for real-time processing can be written with standard means. These modules make use of custom GPU kernels in combination with some NPP functions as well as OpenCV functions.

Algorithm 1 How to load in the GPU buffer for CUDA

Input: inbuf, the standard GStreamer data buffer
gst_buffer_map (inbuf, &in_map_info, GST_MAP_READWRITE)
surface = reinterpret_cast<NvBufSurface *>(in_map_info.data);
egl_frame = cudaGraphicsResourceGetMappedEglFrame (cudaGraphicsEGLRegisterImage (surface)) // full details not shown
frame = static_cast<Npp8u *>(egl_frame.frame.pPitch[0].ptr);
Start processing with CUDA device pointer **frame**.

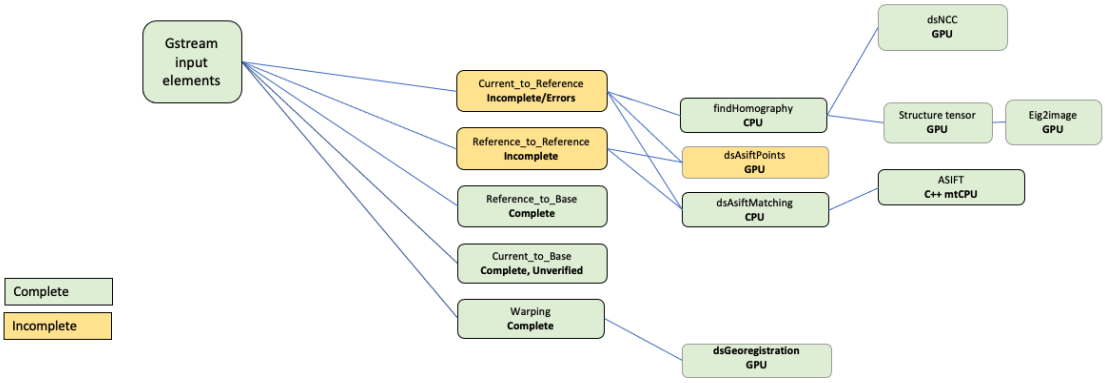


Figure 2.3: The architecture and progress of the Deepstream implementation of VMZ.

2.2.2 DeepStream Progress

Using Deepstream and the methods detailed above, we have built out the VMZ pipeline as a number of modules that will be described in the following chapter. Diagram 2.3 shows the status of all these modules. Some are not fully operational, hence the entire pipeline does not generate the same results as VMZ yet.

Chapter 3

Features for Image Matching

The base for VMZ, as well as many other pipelines and algorithms, is feature detection and matching. These operations are foundational and incredibly important, and can be very time consuming if not careful. Luckily they are typically highly parallelizable and can be optimized on GPU. The following sections discuss ST, NCC, and ASIFT, and their implementations for this work.

3.1 Structure Tensor

The structure tensor feature detector is a powerful feature detector that makes use of the gradient values of an image. It provides information about corners and edges. In this work we use the eigenvalues of the structure tensor response to achieve a more precise descriptor of feature points. With this modification, it is excellent for denoting the distinctiveness of an area. By extension, it can also be used to indicate homogeneous areas (continuous areas with a low response).

Where Ix and Iy are the gradients in the horizontal and vertical directions of structure tensor and are defined as

$$J_{2D}(x,y) = \Delta I(\Delta I)^T = \begin{bmatrix} I_x^2 & I_x I_y \\ I_y I_x & I_y^2 \end{bmatrix}. \tag{3.1}$$

Note that we also apply a Gaussian to smooth it:

$$J_{2D}(x,y) = G_\sigma * \Delta I(\Delta I)^T. \tag{3.2}$$

For VMZ, we also use a custom variant of the ST that takes advantage of feature blocks, which are specifically sized regions across the image within which we extract features to ensure an even spread of features across the image. This helps matching stability later on. It also allows us to identify regions of the image without strong feature blocks, and identify those as homogeneous regions, which are themselves points that can be used for matching.

It is parallelizable because we are performing operations across the image in the same way that are not dependent on other results. In this work, each atomic mathematical operation is done with NPP. The extraction of feature points and homogeneous points is parallelized on custom GPU kernels to efficiently sort through large amounts of data in the image.

We use the eigenvalues of the ST to identify distinctive regions of the image. To obtain λ_1 and λ_2 we do

$$
\begin{aligned}
T &= I_{xx} + I_{yy} \\
D &= I_{xx} * I_{yy} - I_{xy}^2 \\
A &= T/2 \\
B &= \sqrt{T^2/4 - D} \\
\lambda_1 &= A + B \\
\lambda_2 &= A - B.
\end{aligned}
\tag{3.3}
$$

For this work, this was implemented with several NPP functions to perform each of the operations listed in the Equation block 3.1. For example, to calculate the determinant we did:

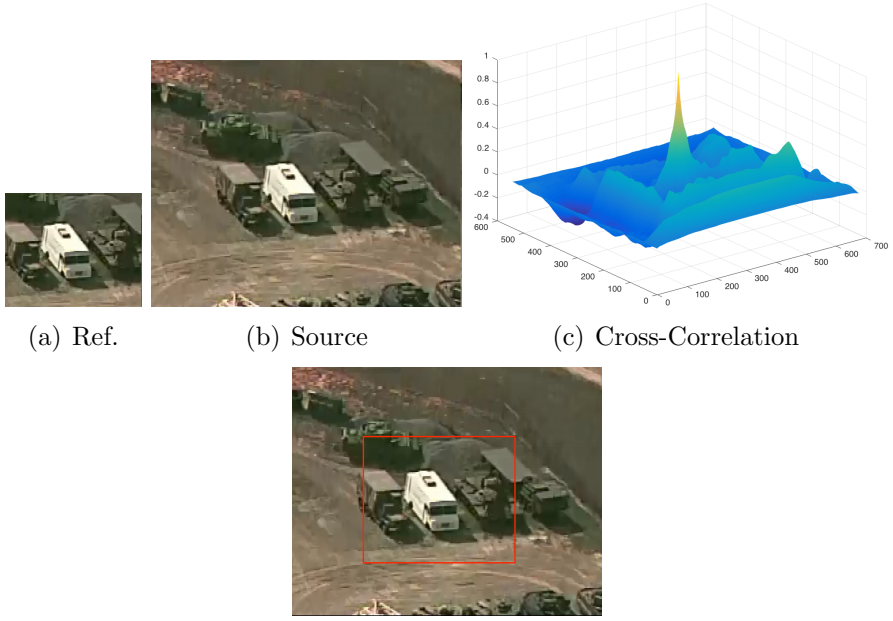
$$
\begin{aligned}
B_1 &= \text{nppiMul\_32f\_C1R}(I_{xx}, I_{yy}) \\
B_2 &= \text{nppiMul\_32f\_C1R}(I_{xy}, I_{xy}) \\
D &= \text{nppiSub\_32f\_C1R}(B_1 - B_2),
\end{aligned}
\tag{3.4}
$$

where B_1 and B_2 are intermediate buffers. To be efficient we used these buffers repeatedly discarding values that aren't needed.

With λ_1 as the main signal used for our purposes, we then find the maxima of the image, constrained by the feature blocks of size 40x40, limiting our responses to one maxima per block. Responses below 0.5 are discarded from this, allowing homogeneous blocks when there are 2x2 regions of feature blocks without any maxima. The homogeneous point is recorded at the top left coordinate of the block.

3.2 Normalized Cross-Correlation

Normalized cross-correlation (NCC) is a popular measure of similarity between two images or image blocks. It is less sensitive to absolute intensity changes than other methods, but is quite expensive to compute. Because of this, we typically try to limit the area of an image being evaluated to find a good match. The NCC can be applied as a sliding window across an area where the maximum of the response map is used to represent the matching point.



(a) Ref. (b) Source (c) Cross-Correlation

(d) Template on source

Figure 3.1: Example of NCC computation in our mosaicing algorithm for a template/reference block with source/current image: (a) template/reference block, (b) source/current image, (c) cross-correlation between template block and source block, and (d) template is matched and highlighted on source image.

The NCC between a template (or reference) image block $T(B, \mathbf{x}, t-k)$ from the frame at time $(t-k)$ and search (or current/source) image block $I(B, \mathbf{x}, t)$ from the frame at time t, is defined as

$$\gamma(B, \Delta\mathbf{x}_B, t) =$$
$$\frac{\sum_{\mathbf{x}\in B}[T(B(\mathbf{x}), t-k) - \mu_T(B(\mathbf{x}), t-k)][I(B(\mathbf{x}+\Delta\mathbf{x}_B), t) - \mu_I(B(\mathbf{x}+\Delta\mathbf{x}_B), t)]}{\sqrt{\sum_{\mathbf{x}\in B}[T(B(\mathbf{x}), t-k) - \mu_T(B(\mathbf{x}), t-k)]^2 \sum_{\mathbf{x}\in B}[I(B(\mathbf{x}+\Delta\mathbf{x}_B), t) - \mu_I(B(\mathbf{x}+\Delta\mathbf{x}_B), t)]^2}} \tag{3.5}$$

where $\mu_B(B, t-k) = \langle I(B, \mathbf{x}, t-k)\rangle$ and $\mu_B(B, t) = \langle I(B, \mathbf{x}+\Delta, t)\rangle$ are the local intensity means in the target and template image regions, respectively. An example of NCC computation and template matching is presented in Figure 3.1.

We perform the NCC in a sliding window fashion on an area in order to determine the best match for a given larger current block and smaller reference block. Because of this sliding window computation, even when we narrow the current/source block down to an area around detected features, we have a great number of multiplies and adds. Element-wise multiplication $[I(X + \Delta X, t-k) - \mu_{t-k}][I(X, t) - \mu_t]$ results in $size(I)$ multiplies, and the sum is another $size(I)$ adds. On the bottom, the square roots are both $size(I)$ multiplies, and the sums are another two. In totality, this is $6 * size(I)$ operations per time we compute the NCC. A typical size of the reference is 40x40, so 1600 pixels. To sweep over a current frame of size 142x142 pixels, we would need to do 103 NCC operations with no out-of-frame padding. This results in $6 * 1600 * 103 = 988,800$ operations for one block. Our main pipeline is roughly 8000 frames with about 120 blocks per frame, amounting to a total of 1,067,904,000,000 operations (not to mention that 8000 frames have to be loaded, although these are fairly small - 720x480 pixels). This can be optimized, with the use of integral image, and the Fourier transform. These help in employing optimization techniques which brings down the linear computation required for each of the sum operation to almost constant time and allows for the algorithm to be much faster with far fewer operations. The calculation of NCC is a serial portion of the program that Amdahl's

law says is the limit of the parallelization. Our goal with parallelism is to allow many of these NCC operations to happen concurrently, or in the case of the current GPU implementation, allow a single NCC calculation itself to use multiple threads.

Our GStreamer implementation of NCC receives ROI metadata on the GStreamer metadata bus to indicate where it should perform the NCC. This is passed forward from the structure tensor GStreamer element and indicates where feature points were found with the region around them that should be searched for matches. After computing the NCC with `nppiCrossCorrValid_Norm_8u32f_C1R`, it passes metadata forward that represents the matching points, to be used by other elements like the warping and georegistration element. Figure 3.2 shows some example debugging output from our GStreamer element that shows the visualization of the NCC source, template, and matching response map, as well as the chosen point for matching.

3.3 Affine Invariant SIFT

Scale-invariant feature transform (SIFT) is an important feature detection algorithm in computer vision. In it, keypoints are obtained from a difference of Gaussians method, and points are invariant to location, scale and rotation.

ASIFT is a robust application of SIFT along two additional camera axes and also defines a matching framework using the extended set of features that result from the algorithm. The algorithm accomplishes this by rotating and tilting the image with a predetermined set of transformations to simulate movement along the camera's longitudinal and lateral axes. See figure 3.3 from [18] that illustrates this concept.

The predetermined ASIFT tilt (t) and rotation (ϕ) parameters are important to optimal performance and efficiency in the algorithm. They determine what tilt and rotation angle are applied to each image before SIFT matching is applied in each iteration of the algorithm. Morel et al. determine this in [18] as $\Delta t = \sqrt{2}$ and

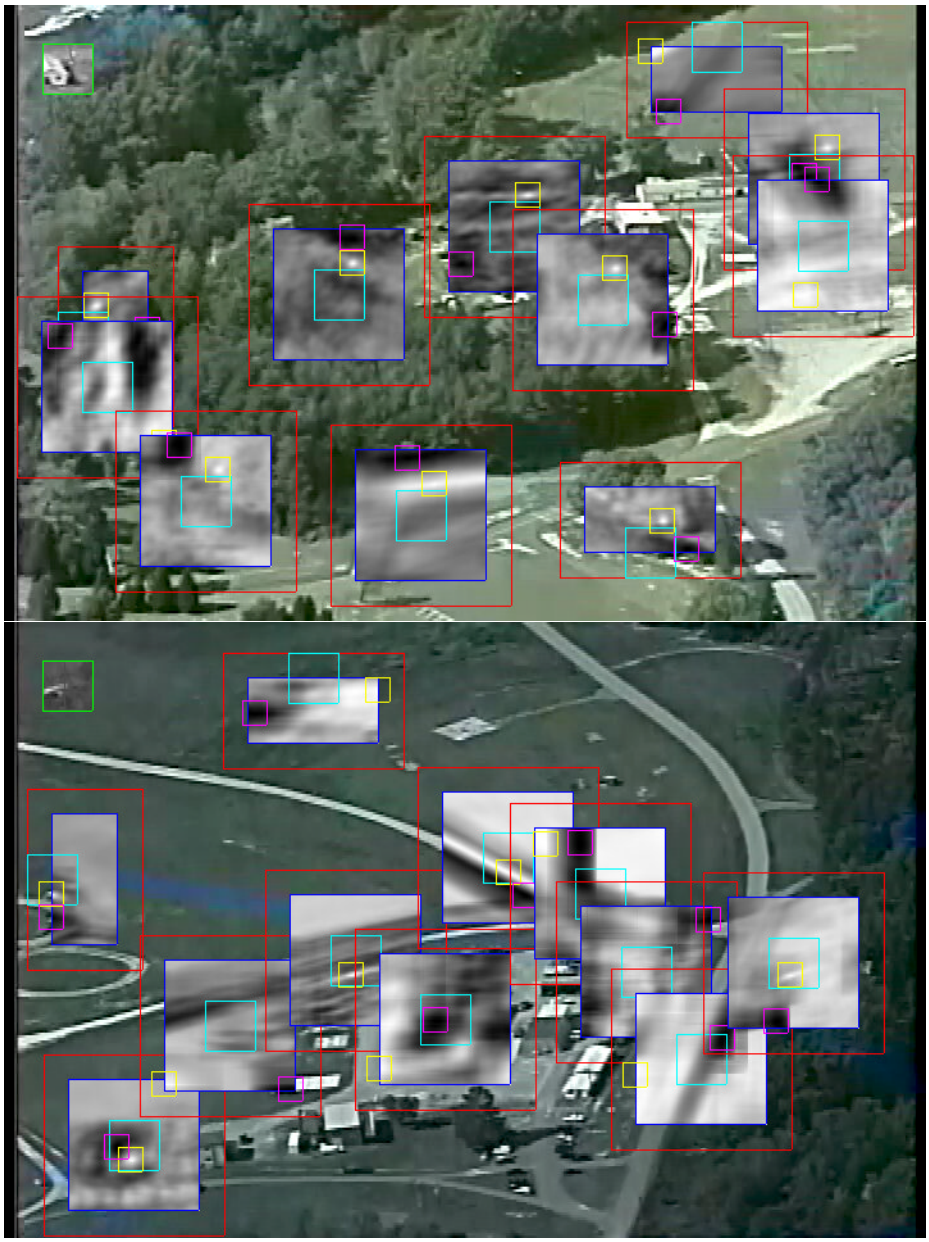


Figure 3.2: This shows NCC in action on some sample frames from the VMZ sequence. The red square shows the entire region of the current frame that will be searched. The Cyan square in the center shows the region that is cut out of the reference frame to be compared to the current frame. The Blue square shows the valid area of NCC response from applying NCC in a sliding window fashion (cyan square sliding across the red square.) Finally, the yellow and pink squares represent the max and min of the NCC response, respectively. The center pixel of the square region is where that point is located.

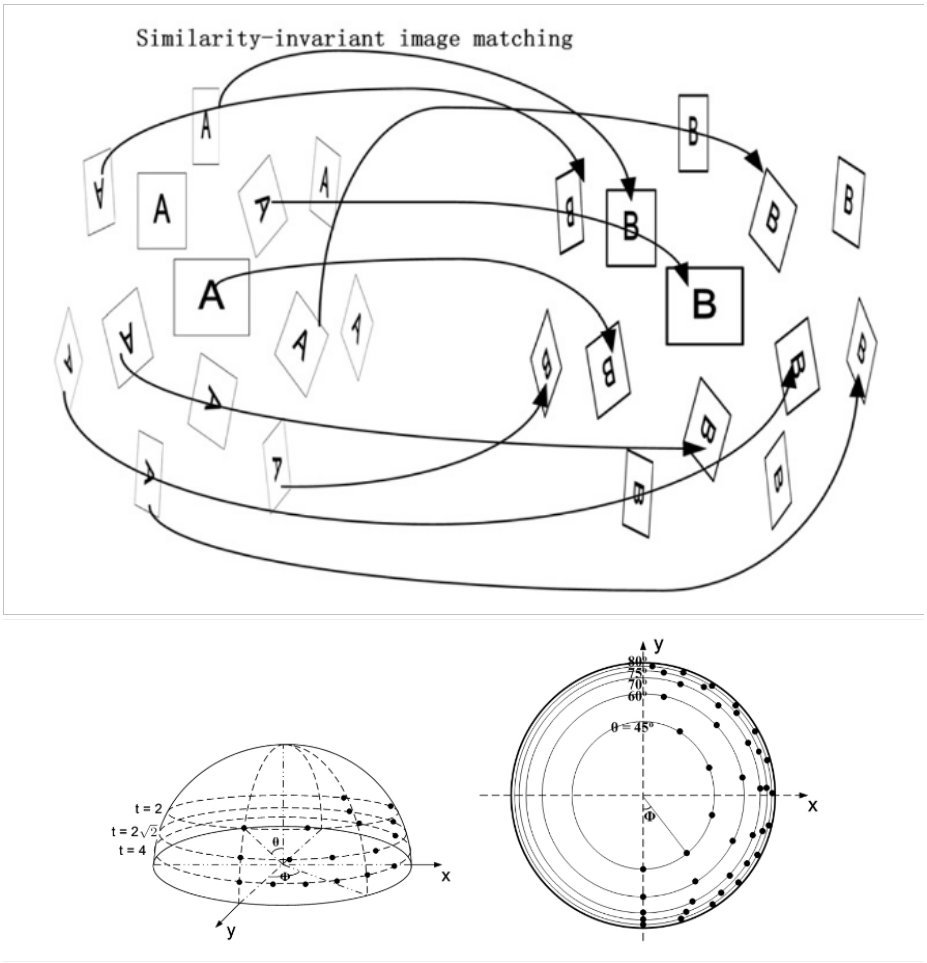


Figure 3.3: Figure credit to [18]. The general illustration, top, shows how two images are compared against each other in a number of different configurations of rotation and tilt. The bottom two figures show the sampling method for the parameters. $\Delta t = \sqrt{2}$ and $\Delta\phi = \frac{72^\circ}{t}$, two experimentally obtained parameters determined in [18] that decrease the sampling rate as the tilt and rotation become more extreme.

$\Delta\phi = \frac{72^\circ}{t}$. These decrease the sampling rate geometrically as the tilt increases and the simulation is becoming less like the original images.

SIFT covers 4 of 6 affine parameters by normalizing rotations and translations, and simulating all zoom outs of the image [18]. ASIFT simulates the remaining two parameters by simulating the camera optical axis directional change and applying SIFT along those axes. [18] shows that this is mathematically affine invariant. Finally, ASIFT avoids being prohibitively expensive by sampling along the affine space.

This sampling is done along a geometric series, sampling more frequently at higher

latitude and longitude changes. This is because a stronger distortion will result in more drastic image changes and more potential errors.

ASIFT samples 13.5 times the area and feature space as SIFT. While this results in a computational increase, we limit the use of it only to match reference frames, which occurs every 40 frames or so. Thus the performance impact is kept minimal.

The additional invariance that ASIFT provides is very powerful for matching two images far apart in a video, because it is particularly robust to geometric changes. This is why it is used in VMZ for matching reference frames.

To accomplish this on the Xavier, we have transported and converted existing SIFT and ASIFT code to see similar results in near-real time. The SIFT features and the image warps for ASIFT are all accomplished on the GPU, while some of the driver code and image matching are still accomplished on the CPU.

3.4 Integral Image & Integral Histogram for Fast NCC and Fast Feature Extraction

Many computer vision and image processing algorithms require finding the sum of a rectangular area of the image. Evaluating the sum of an area takes $O(N^2)$ when implemented naively. However, with the integral image this can be reduced to constant time O(1).

The integral image is a preprocessing step where we precompute sum values for each pixel in a buffer that is the same size as the original image. In this buffer, each (x, y) position represents the sum of the pixels from $(0, 0)$ to (x, y) in a rectangular region. With these sums, simple arithmetic can achieve any desired sum.

The generation of the integral image, for an image of width M and height N, takes O(MxN) but can be improved further with parallel processing.

The integral histogram is an extension of the integral image idea. Instead of

storing the sum, a histogram is maintained for each pixel of the image to represent the rectangular region from $(0, 0)$ in to (x, y).

Image feature extraction uses the integral image to normalize images with the sum when needed, like in the normalized cross-correlation as mentioned in Chapter 3.2. The integral histogram makes collecting histogram-based features easier. These are robust to translation and particularly invariant to orientation. Spatially weighting the features also addresses a shortcoming of histogram-based features, which is that they do not naturally account for spatial information.

We have developed a method for generating the integral image while utilizing the GPU as well as for generating multi-scale spatially weighted integral histograms in GPU. The weighted histogram problem has not been adressed with integral histograms, to the best of our knowledge. The GPU implementation of both of the non-weighted and weighted case are very important, because of the high computational cost of integral histograms in general.

To the best of our knowledge, our two integral histogram GPU algorithms are extremely performant. They are faster than other parallel methods [20] [21].

To generate non-weighted integral histograms, we use parallel prefix-sums [20]. This is an established GPU programming pattern that can be understood as a way to efficiently perform sums in parallel and bring results together afterwards. In our previous work, we explored two versions of the scan-transpose-scan implementation of generation parallel-prefix sums, and we found that our single-scan method was performing slightly better by utilizing the GPU. The single-scan-transpose-scan method is shown in 3.4.

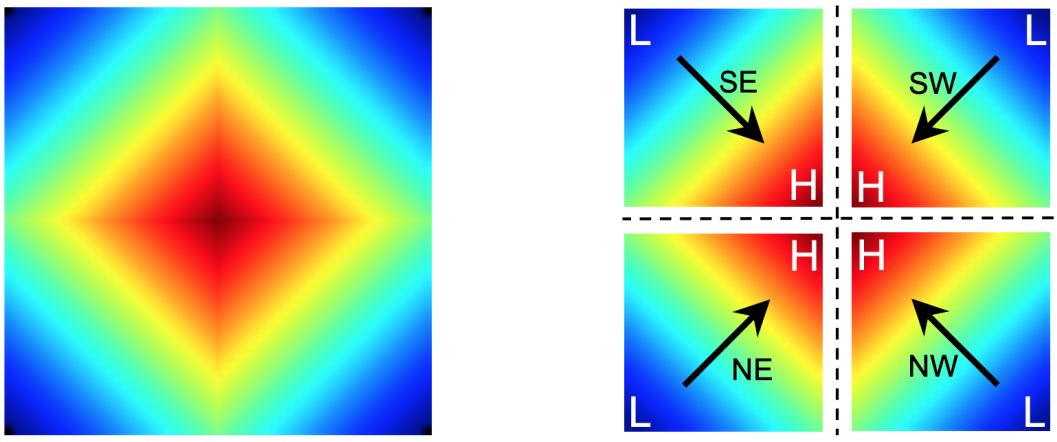
To generate the weighted integral histogram, we decompose the Manhattan spatial filter and fragment the region of interest into four parts.

Figure 3.4 shows how we decompose the Manhattan filter into four parts to compute weighted integral histograms. Then Figure 3.5 shows how we can then compute

Algorithm 2 The implementation of GIH-Single-STS from [20]

Input: Image I of size hw, number of bins b
Output: Integral histogram tensor IH of size b h w

1: **Initialize IH**
   IH $\leftarrow 0$
   $\mathbf{IH}(\mathbf{I}(\mathbf{w},\mathbf{h}),\mathbf{w},\mathbf{h}) \leftarrow 1$
2: **for** all $b \times h$ blocks in parallel **do**
3: //horizontal cumulative sums
4: **Prescan(IH)**
5: **end for**
6: //transpose the histogram tensor
   $IH^T \leftarrow$ 3D_Transpose(IH)
7: for all $b \times w$ blocks in parallel **do**
   //vertical cumulative sums
   Prescan(IH^T)
8: **end for**



(a) Manhattan

(b) 4-Directional Independant Weights

Figure 3.4: A representation of breaking an image's Manhattan filter into a 4x4 grid. The weights increase linearly as the pixels get closer to the center.

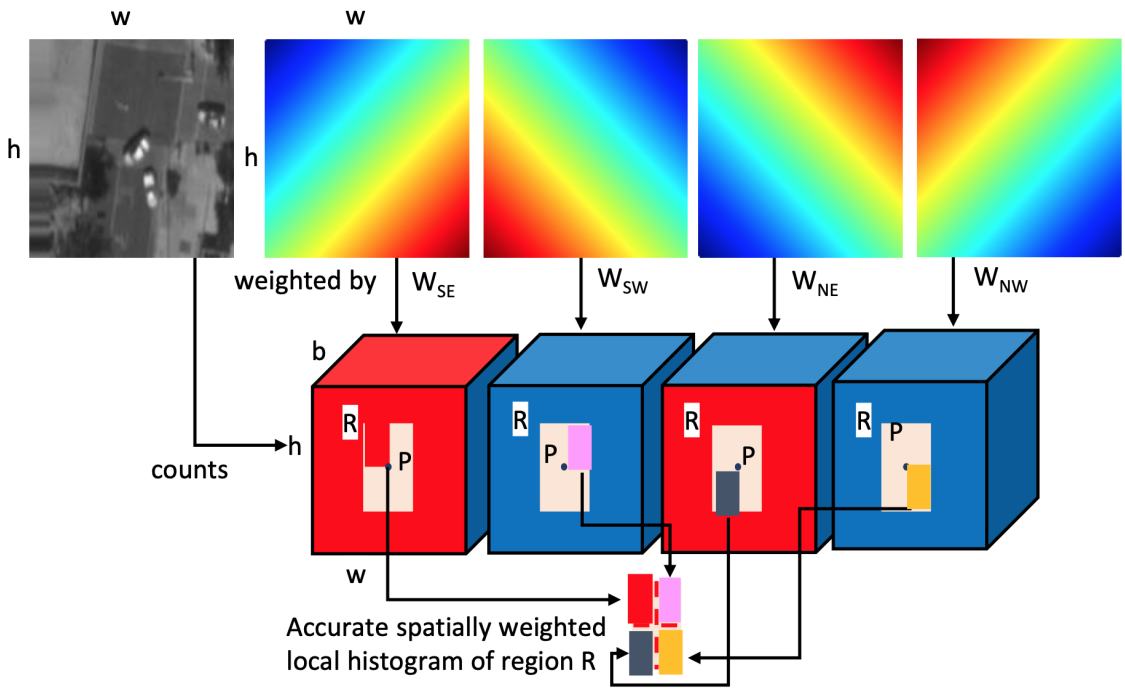


Figure 3.5: The decomposed Manhattan filter applied to an roi from an image. The weights are applied in parallel and then combined to form the full histogram for the image.

the integral histogram of any region of interest by also splitting it into four parts. The rectilinear nature of the Manhattan spatial filter allows us to apply translation from the center point to appropriately compensate the weighting of any computation in generating the integral histogram of any ROI.

Chapter 4

Image Registration and Warping

Aerial georegistration is important when working with aerial imagery. Aerial georegistration normalizes all frames in a video to a common plane, such that the features and geometry of the images align. Figure 4.1 demonstrates this concept, as well as the risk of parallax. This allows for applications that involve understanding of a large area, such as disaster response. In these cases, it could be very useful to have real-time transformations, which this work accomplishes.

The homography needed to perform the georegistration can be obtained in a number of ways. Most often, feature point matching is used. In this case we obtain these from ASIFT feature matching or OpenCV feature matching done via RANSAC. The matches are used to find the best homography to accomplish a transformation that minimizes the distance between the matches after transformation.

Usually the first image in the sequence is chosen as a reference frame for the warping, and subsequent frames are transformed to that plane.

Given that the image transform is applying the same function to each pixel, it is easy to see how multiple pixel transforms can happen at the same time. Registration is thus a parallelizable problem, and this is likely what the NPP warping function `nppiWarpPerspective_8u_AC4R` uses. Unfortunately, NVIDIA does not publish the

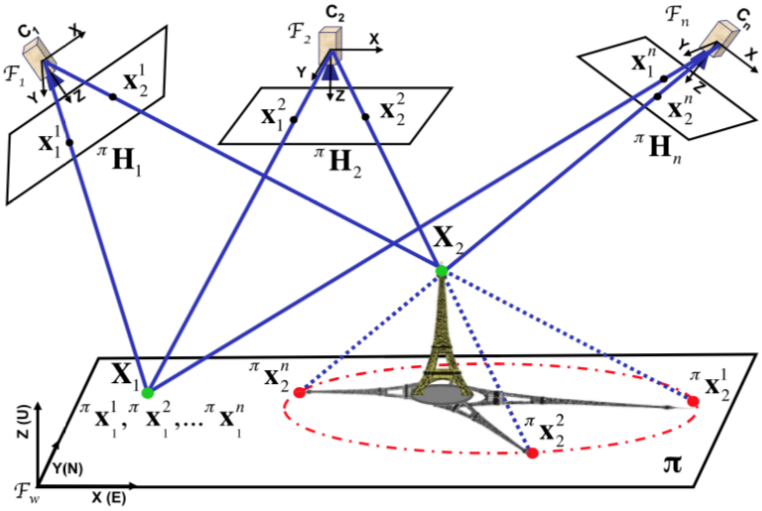


Figure 4.1: A scene and its ground plane π. It is observed by n aerial cameras. ${}^{\pi}\mathbf{H}_i$ represents the homography transformation between the image plane of C_i and the plane π. For an on-the-plane 3-D point such as $\mathbf{X}_1$, its homographic transformations from the camera image planes onto π will all merge together and coincide to $\mathbf{X}_1$ (the green point). However, for an off-the-plane point such as $\mathbf{X}_2$, its homographic transformations will be spread out (see the red points). These red points are spurious and induced by the parallax. [8]

source code for these, so we treat the GPU programming side of this warping function as a black box.

Our GStreamer element for georegistration receives points from some downstream element that is doing image matching. It reads these points from the GStreamer metadata buffer and then uses them for the OpenCV function `findMatches`. The result of this is the homography that we use on the NPP function to warp the image, `nppiWarpPerspective_8u_AC4R`. We then write it to a larger file. If the resulting warped image bounds moves too much or goes outside the bounds of the larger image frame, we declare this frame a reference frame and send a metadata signal back to the beginning of the pipeline to respond to changes. This is done by firing a click event at an expected (x, y) coordinate in the frame, which is the only way to pass signals to downstream elements in GStreamer. In our case the structure tensor element takes this as a signal to generate new reference frame ST points.

Chapter 5

Experimental Results

In this chapter, the results for each algorithm's translation to GStreamer and GPU programming will be shown. For more detailed code diagrams or a more in-depth discussion of the remaining bugs in the DeepStream VMZ pipeline, see Appendix A and B.

5.1 Data

The primary dataset for results was aerial imagery from flight 2, tape 1-6 of the Video And Image Retrieval And Analysis Tool (Henceforth VIRAT) dataset [22]. It is an aerial imagery dataset well suited to the expected use cases of the VMZ pipeline, and it is the primary dataset for the main VMZ work [7], as well as many aerial registration and summarization works [4]. It is approximately 5 minutes long at 30 frames per second, 9290 frames and the resolution is 720x480. It captures many of the problems encountered with aerial imagery: Changing scene, changing structure, camera movement, occasional lens effects.

Examples of frames from VIRAT dataset and mosaics are presented in Figure 5.1

Figure 5.1: A few random frames from VIRAT sequence, 09152008flight2tape1_6. The sequence contains 9290 frames and each frame is of dimension 720x480.

and Figure 5.2, respectively.

5.2 Hardware

All results for the GStreamer modules were run on an NVIDIA Jetson Xavier AGX. This device is very small with very low power consumption at 40W. For comparison, standard desktop power consumption typically ranges from 250W-1000W. Its uses a GPU running NVIDIA Volta architecture. It contains 512 NVIDIA CUDA Cores across 64 tensor cores. It is a powerful embedded device for edge computing and suits the needs of various aerial image process algorithms, especially when GPU programming can be used like with the algorithms chosen.

5.3 Structure Tensor Experimental Results

Figure 5.3 shows the structure tensor response obtained from applying the operator as described in Chapter 3.1.

Figure 5.4 shows an example of where from that response we pull the feature maxima. The points are well spread out because of our feature block method. In areas where there were no feature points within a certain region, we show the homogeneous point in green.

Figure 5.2: Example mosaics from VIRAT sequence, 09152008flight2tape1_6. Number of frames in each mosaic is 82 (top) and 248 (bottom).

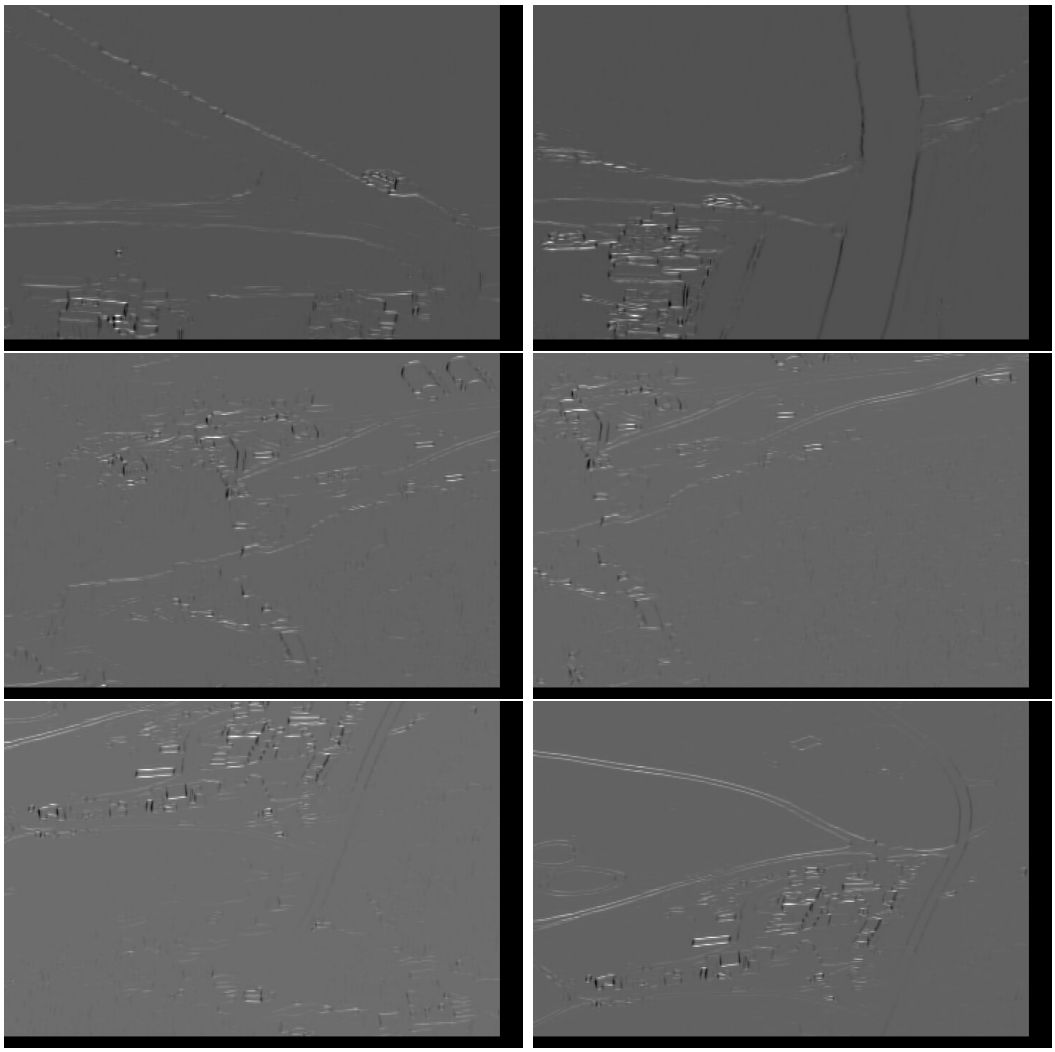


Figure 5.3: The structure tensor response, visualized in grayscale. The images are shifted such that there is no black border on the top or the left.

Figure 5.4: Boxes drawn around ST points for different frames of the sequence. Green squares represent homogeneous points

On the Jetson Xavier with VIRAT images sized 720x480, this module runs on average in 34.63 milliseconds. This corresponds to 28.9 frames-per-second operation. This is much faster than other operations in the pipeline like NCC and ASIFT. Structure tensor is a straightforward mathematical operation and can be done very fast with the GPU operations being used.

5.4 NCC Experimental Results

While NCC was reimplemented in GStreamer as part of this work, as mentioned in Chapter 3.2, this work used `nppiCrossCorrValid_Norm_8u32f_C1R`. This function was evaluated separately with experiments run with the help of Ahmad et al. [23]

For timing analysis of our methods with respect to the original MATLAB implementation, we selected three main factors. As stated previously, our dataset consists of 8131 image files, which contains a total of 933,538 blocks. The factors we have chosen for our timing analysis are, $\mathbf{Total-Time}$, the time required to calculate NCC matching for all of the 8131 images, $\mathbf{Time-Per-Block}$, the average time required to calculate NCC matching over a single block, and $\mathbf{Time-Per-Frame}$, the average time required to calculate NCC matching over a source image, searching for the template image in it. It can also be described as the average time required to calculate all of the blocks in a single image file.

For running our CPU based multi-threaded implementations, we have used a **Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz server** with **14 cores with Hyper-Threading**, so **28 threads**, and **252 GB of RAM**.

To experiment on the GPU, we tested with two different GPUs with varying amounts of streaming multiprocessors (SMs) and CUDA Cores. First was a GeForce GTX-1080 GPU. This has 20 SMs containing a total of 2560 CUDA Cores. Second was a Tesla V100, with 5120 CUDA Cores across 80 SMs. Because of increased

interest in performing computation on the edge, we also tested on a NVIDIA Jetson Xavier AGX.

We also tried our muti-threaded code on the Xavier. It contains 8 CPU cores in a Carmel ARM v9.2 CPU which has a memory bandwidth of 137 GB/s.

For our two separate multi-threaded CPU implementation, in the Table 5.1, we have only shown the results for the implementation that used the C++ standard threading library. As we will also show, the timing differences between the two implementations, **C++ standard threads** and **OpenMP**, tends to be minimal as the number of threads increases. We have also tested on threads numbering from 1 to 27, but for the timing comparison, we have selected to show only the results of four instances, and included the timing results when running on a single thread, and when running on 7, 14 and 27 threads concurrently. From the selected set of timing for different number of threads, the actual timing trend for all of these threads can be easily seen. The top row of Figure 5.5 describes the timing comparison for all of the thread configurations.

It is evident from Table 5.1 that different threading libraries, like **C++ standard threads** and **OpenMP**, for multi-threaded implementation, performs very similar and has little effect on the total time for running NCC computations as the number of threads increases. From here on, we will only focus on the multi-threaded implementation with the **C++ standard threads** library.

Table 5.1 shows that switching to OpenCV, from MATLAB, gives some speedup straightaway, with no parallelism added. Further, it shows that we have a speedup as we increase the number of threads, but the speed increase is not linear as the thread count increases. Finally, it shows that there is a speedup when using the GPU, but that speedup is more conservative than expected. This is because the NPP API does not take advantage of the full breadth of parallelism available, since it only parallelizes the given single NCC operation. The V100 shows a much better improvement, likely

Timing comparison between the different methods				
NCC Computation Method	**Total-Time (s)**	**Time-Per-Block (ms)**	**Time-Per-Frame (ms)**	**Speed-Up[1]**
MATLAB	12374.7	14.1	1522.0	1.0
C++ CPU - Threads = 1	3168.9	3.4	423.8	3.8
OpenMP - Threads = 1	2791.3	3.0	343.3	4.4
C++ CPU - Threads = 7	822.3	0.9	99.7	15.0
OpenMP - Threads = 7	810.7	0.9	99.7	15.3
C++ CPU - Threads = 14	497.1	0.5	60.4	24.9
OpenMP - Threads = 14	489.5	0.5	60.2	25.4
C++ CPU - Threads = 27	313.7	0.3	39.2	39.5
OpenMP - Threads = 27	319.2	0.3	39.3	38.8
Xavier CPU - Threads = 8	446.4	0.5	54.9	27.7
Xavier GPU	1167.0	1.3	143.5	10.6
GTX-1080 GPU	854.4	0.9	103.6	14.5
V-100 GPU	152.5	0.2	18.7	81.1

Table 5.1: This compares the performance at a high level, then at increasingly granular levels. Time-Per-Block for multi-threaded CPU implementation is the observed time calculated from the total time and number of blocks. *Speed-up is shown with respect to total time, but all columns show about the same results. The blue color uses OpenCV, the green color shows results for NPP to do GPU computation. Performance statistics reflect a real application workload with variable block sizes (majority of search blocks are 142x142), template block sizes (majority of search blocks are 40x40), an average of 120 blocks per frame, and 8132 frames.

due to better hardware specification.

We computed the difference of the NCC results from the original MATLAB implementation with our multi-threaded CPU, and the GPU implementations using NPP. The correlation matrix result for the GPU implementation using NPP is very different from what we get from MATLAB and the multi-threaded CPU implementation using OpenCV, while the MATLAB result and the result from our multi-threaded implementation using OpenCV is quite similar. Below, we measured the peak points in the NCC correlation matrix from MATLAB and OpenCV, which is essentially the best point where the reference block matches in the search window. An exact match is defined where (X-Peak, Y-Peak) is exactly same in MATLAB and OpenCV, and they

[1]Speed-up with respect to MATLAB implementation

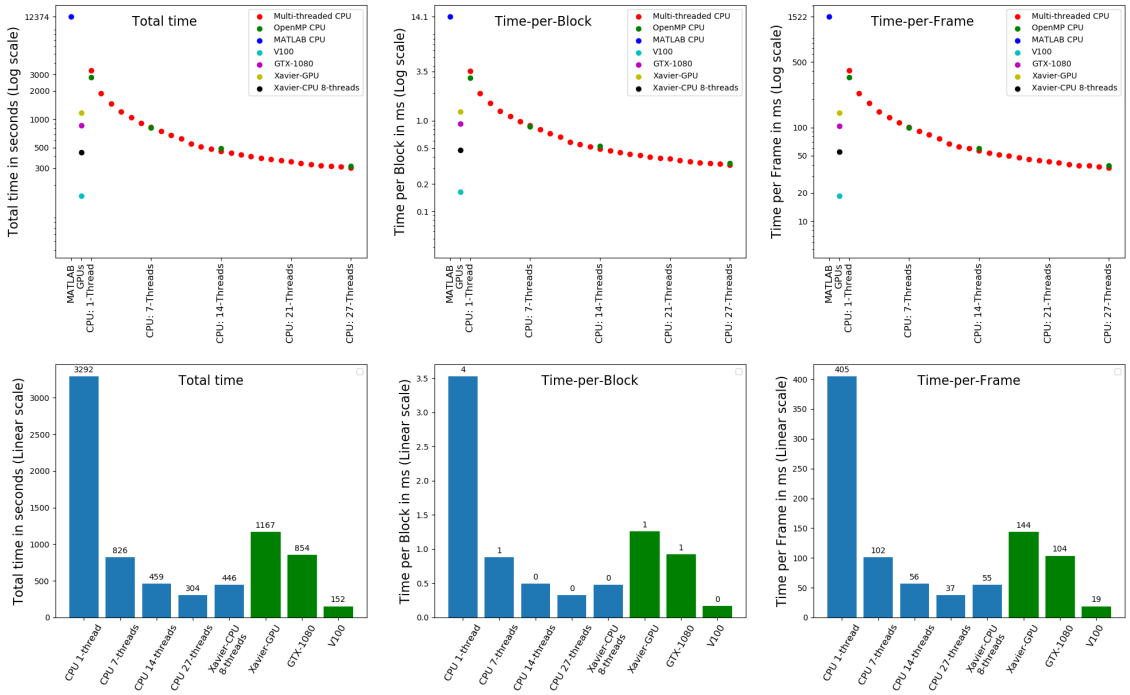


Figure 5.5: The timing comparison between all of our methods. The first column of plots are for the factor Total-Time (for all frames), second column of plots are for Time-Per-Block, and the third ones are for Time-Per-Frame. All of the plots in the first row are in Log scale for the Y (time) axis for including the MATLAB timing while keeping all of the other method timings distinguishable. The second row of plots excludes the MATLAB results, and are in linear scale. In each of the plot, GPU results are shown under a single X-axis value, as well as the MATLAB result for the first row of plots, and the rest of the X-axis values, from 1 to 27, denote the number of threads used for the CPU implementation.

have an Euclidean distance of zero. In our tests, we have seen that around **94.9%** of blocks per frame has an exact match or zero Euclidean distance. Including closely matching block-pairs with less than or equal to 4 pixels euclidean distance increases the percentage agreement between the MATLAB and multi-threaded OpenCV NCC peak results. In our experiment, we got **99.37%** of blocks with a close match or less than or equal to 4 pixel radius Euclidean distance. Thus, only **0.63%** of blocks have error greater than 4 pixels. In our experiment, we pick blocks (structure tensor (ST) feature points) which are far from image edges. The distance between ST points should satisfy the following condition, where $edge\_distance_{ST}$ refers to the minimum distance of a ST point from image edge, and SW_d and $temp_d$ stands for size of search block and template block, respectively:

$$edge\_distance_{ST} \geq \frac{SW_d}{2} + temp_d. \tag{5.1}$$

For a comparison of the OpenCV and MATLAB results to the significantly non-matching results from NPP, it is best seen in Figure 5.7. Here it is clear that for these example images the correlation responses are quite different. NPP does not find values that are nearly as high except for where the peak is, which could actually be considered a more ideal and less noisy NCC map. Still, because our implementation for mosaicing is based on the accuracy of MATLAB, we evaluate where the differences in the three algorithms come from.

We have thoroughly checked the correlation matrix for all of our implementations and libraries used by NCC methods. And while OpenCV is open-source, and their implementation details can be seen directly from their code base, both MATLAB and NPP's NCC implementation details cannot be checked due to their closed-source nature. We have summarized that the difference in NCC correlation matrix and the peaks might be caused for multiple reasons.

1. For OpenCV's Cross-Correlation operation, using its `matchTemplate` func-

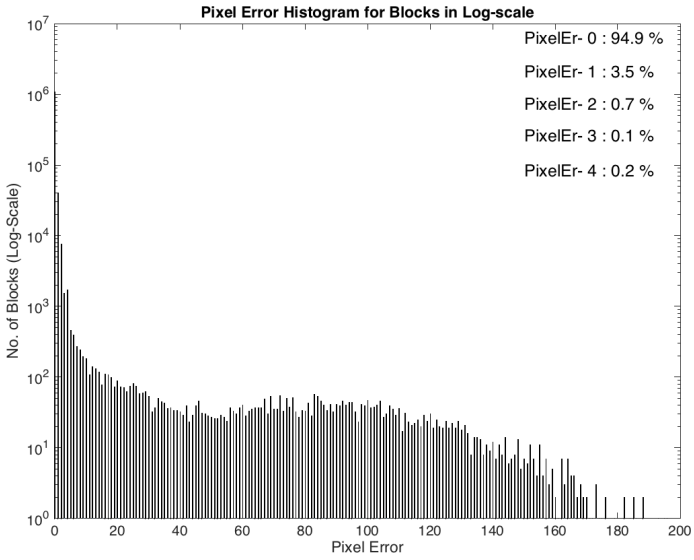


Figure 5.6: Error histogram for comparing (X-Peak, Y-Peak)-location of matched point in MATLAB and our Multi-Threaded NCC implementation using OpenCV. X-axis represents pixel error between same block peak in MATLAB and OpenCV where y-axis stands for number of blocks with that error. From this graph, we can see that around 94.9% of blocks have an exact match, i.e., zero error between them. As the error increases, number of blocks drops significantly. For example, only 3.5%, 0.7%, 0.1% and 0.2% of blocks have pixel error of 1, 2, 3 and 4 respectively. Only 0.63% of blocks have error greater than 4 pixels.

tion, had 6 different methods for calculating the correlation matrix. We have selected the method $'CV\_TM\_CCOEFF\_NORMED'$, which is one of the normalized methods available, and the equation for this method is given in the documentation. Internally, this method uses an integral-image and Fourier-transformation for optimization. But for MATLAB's and NPP's NCC operation, using `normxcorr2` and `nppiCrossCorrValid_NormLevel_32f_C1R`, respectively, don't give any implementation details, and work as a black box, so we could not be sure of the exact method, implementation and optimization techniques being used in its case.

2. The peak finding algorithm used for MATLAB after calculating the correlation matrix was selecting the peak in a column-wise order, where in OpenCV/C++ it was finding the peak in the correlation matrix in a row-wise order. NPP chooses the tied max at the top left of the image. This only creates a mismatch if there are multiple peaks in the correlation matrix with the same maximum value.

3. Checking the correlation matrix from both OpenCV/C++ and MATLAB's NCC computation, it was evident that the size of the correlation matrix for MATLAB is larger in both dimension than the correlation matrix for OpenCV. We found out that this change initially occurs in the Search-Window, where the `normxcorr2` function of MATLAB automatically applies a padding so that it can also calculate the correlation in the border regions of the search-window. And this padding in the search-window causes the computed correlation matrix to be larger in size as well. The `matchTemplate` function of OpenCV doesn't apply any kind of padding while performing the NCC calculation. The dimensions for correlation-matrix can be computed from the search-window and template sizes, where CM_d is the Correlation-Matrix size, SW_d is the Search-Window size, and $Temp_d$ is the template block size (assuming square blocks),

$$CM_d = (SW_d - Temp_d) + 1. \tag{5.2}$$

This Correlation-Matrix size computation is same for both MATLAB and OpenCV. But the Search-Window (SW) size for MATLAB differs from OpenCV as the additional padding in applied on it. For MATLAB, the Search-Window size increases by following this equation, where SWP_d is the Search-Window size with the added padding,

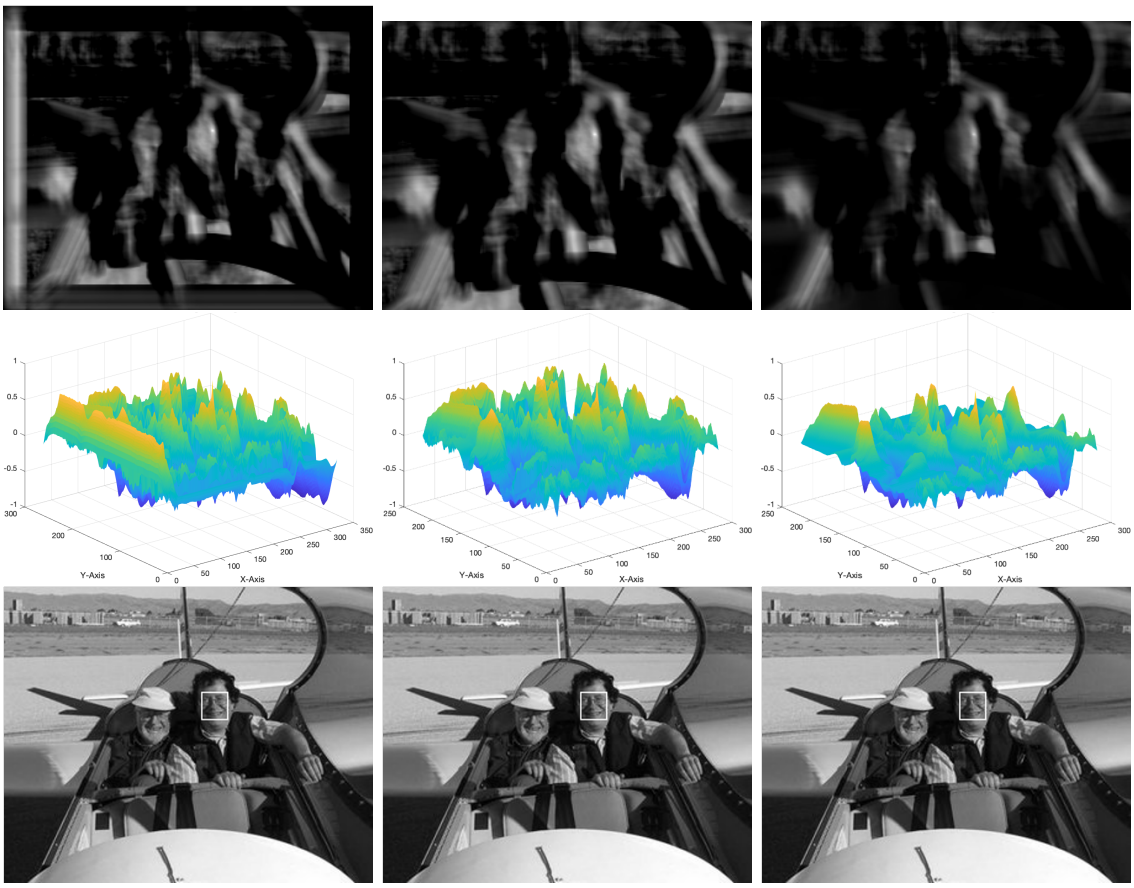
$$SWP_d = SW_d + (2 * (Temp_d - 1)). \tag{5.3}$$

It can be seen from Figure 5.7 that the correlation-matrix for MATLAB is larger than the other two methods and contains border effects, which is the reason for the zero-padding added to the Search-Window. There are different variations of the NPP function we used that use"Full" mode, which allow us control over when 0 padding is used–thus it is not an issue for NPP.

4. As for the difference between OpenCV/MATLAB and the NPP API used, we have found a few major contributing factors. First, precision and some normalization seem to be responsible for reducing the size and number of peaks, this is evident from the correlation matrices and corresponding 3D response maps shown in Figure 5.7. Because the implementation is hidden from us, we cannot be sure, but it seems to be a regional normalization. We found this to be proven by NCC performed with templates taken from the image itself. The peaks were never exactly one, likely because they were averaged with the scores around them.

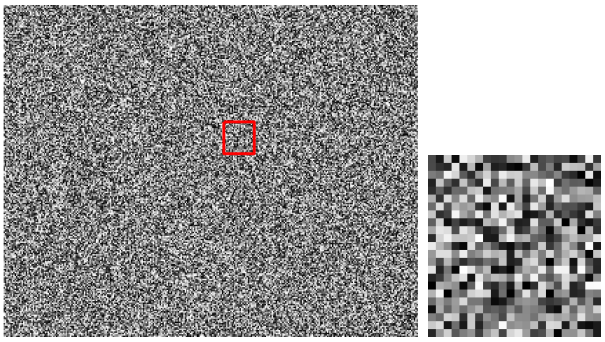
5. The more significant difference is a lack of instability handling in NPP. Because the NCC algorithm divides by the variance, there can be issues when a homogeneous region is taken as the source or template, this is illustrated in Figure 5.8 and Figure 5.9. Divide by zero or divide by very small amounts causes overflow or other related problems in the output. This can be handled by discarding bad values or by regularizing, adding a constant to the numerator and denominator. OpenCV and MATLAB

(a) Test source image and template

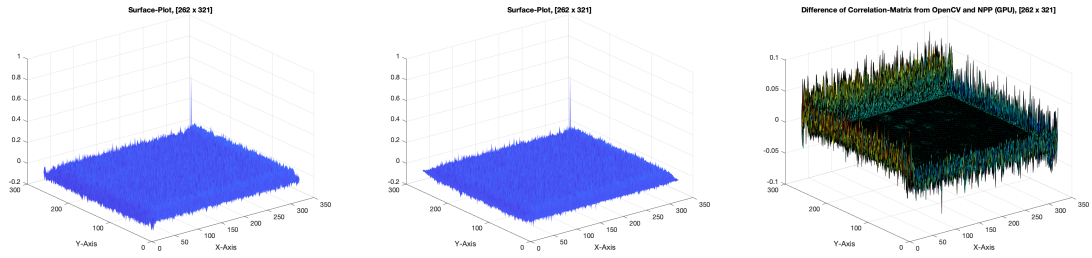


(b) MATLAB (c) OpenCV-C++ (d) NPP (GPU)

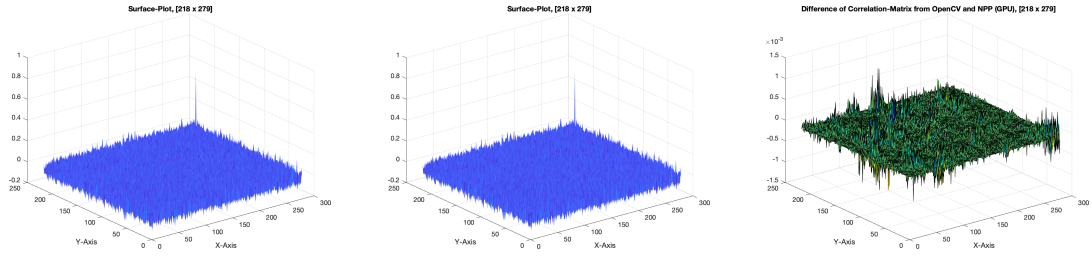
Figure 5.7: This figure illustrates the correlation-matrix and peak comparison between all of our methods. The first row contains the source image and the template, which is to be searched within the source image. The next rows show the correlation matrix and template matched to the source image for each method. The MATLAB result in Row 2 includes the padding which is the default behavior.



(a) Template marked in Source (b) Template

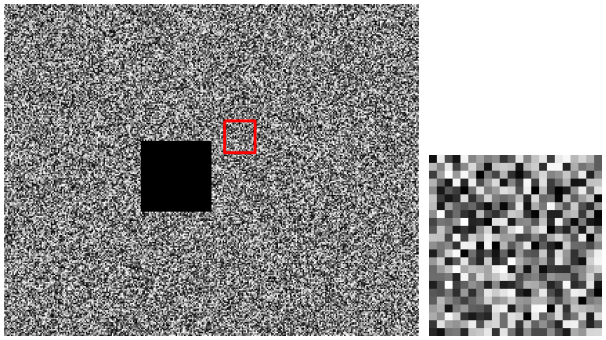


(c) OpenCV Correlation Matrix (SW Padded) (d) NPP Correlation Matrix (SW Padded) (e) Difference (SW padded)

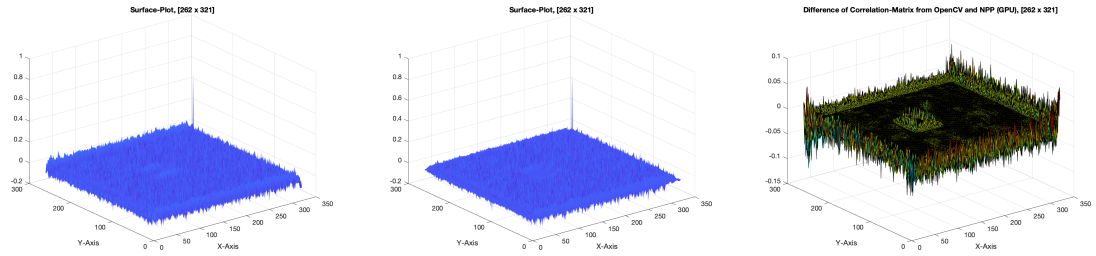


(f) OpenCV Correlation Matrix (g) NPP Correlation Matrix (h) Difference (no SW padded)

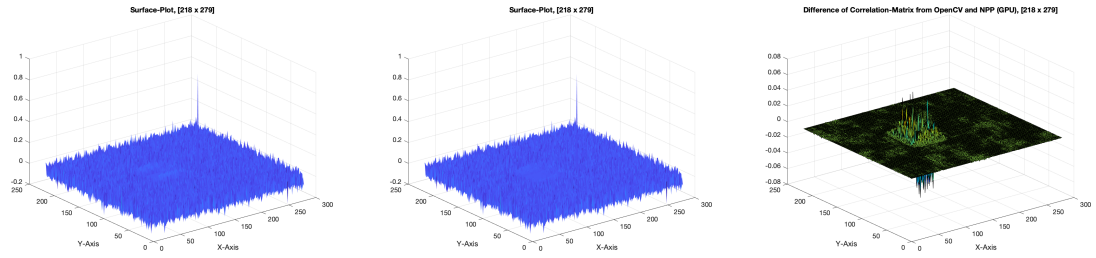
Figure 5.8: For demonstrating the differences in the results for NPP from both MATLAB and Multi-Threaded CPU implementation using OpenCV, we have generated a random image as a source image of dimension (300x240), within which we will search our template-patch, and selected a patch from it as the template of dimension (22x23). For computing the difference map, we have taken the correlation matrix from both OpenCV and NPP, and for the first image we can see that most of the differences are in the zero padded region, which is homogeneous, and in the rest of the image where variance is high, the difference is very small. This is also apparent from the second difference map, as there is no homogeneous region due to no padding, all of the differences are in the 10^{-3} region.



(a) Template marked in Source (b) Template



(c) OpenCV Correlation Matrix (SW Padded) (d) NPP Correlation Matrix (SW Padded) (e) Difference (SW padded)



(f) OpenCV Correlation Matrix (g) NPP Correlation Matrix (h) Difference (no SW padded)

Figure 5.9: We have also generated a random image with a flat patch (homogeneous) inside it as a source image of dimension (300x240), within which we will search our template-patch, and selected a patch from it as the template of dimension (22x23). Similar to the previous figure 5.8, for the difference map, we have taken the correlation matrix from both OpenCV and NPP, and for the first image we can see the differences are in the zero padded region as well as in the patch with flat region, which is homogeneous, and in the rest of the image where variance is high, the difference is very small. This is also apparent from the second difference map, as the only homogeneous region there is the flat patch and the difference is high in that region, and for all other region, the differences are in the 10^{-3} region.

implementations handle these cases, while NPP does not. This makes it unreliable, as in natural imagery we will use many homogeneous regions and the issue cannot be ignored. Luckily, the OpenCV GPU matchTemplate algorithm handles these cases, so we still have a GPU option.

From the speedup results, we can conclude that parallelization results in a significant speed improvement for NCC. Parallelizing across all the template matching needed for a pair of frames, as in the CPU multi-threaded implementation, brought us up to a 40x improvement, although this requires a large number of threads. The performance increase does not scale linearly with threads, as it would seem some overhead prevents this.

Parallelizing the NCC operation itself, as in the GPU implementation, also saw great improvements. Per NCC operation, we saw a 10-80x improvement depending on the hardware. It is much harder to draw any conclusions about what this improvement scales with, as we did not have enough GPUs of similar hardware to test with.

We also saw good performance on the Jetson Xavier AGX, where there is potential to pre-calculate NCC scores for mosaicing on the edge. Although note that generally higher performance might be desired when compared to multi-threaded methods. It is desirable to develop a GPU implementation that parallelizes across NCC operations, rather than within operations as in NPP. This might be more efficient on the Xavier.

Furthermore, the data in Figure 5.6 shows that our NCC implementations are not pixel perfect identical, and there are many possible reasons for that which are still being investigated, as evident from the above discussion.

5.5 ASIFT Experimental results

5.5.1 ASIFT code issues

ASIFT still needs to be further debugged as there is a crash when using the library SiftGPU on the Xavier. This could be due to a number of potential issues. To do the transformation, the GPU image data is being rotated first with `nppiRotate_8u_AC4R`. It is then going through a simulated tilt with a simple vertical shrink affine warp performed with `nppiWarpAffine_8u_AC4R`. The library SiftGPU [24] is then being used to calculate Sift points on the GPU image. It is somewhere in here that an "illegal memory access" error is encountered. Because it is being treated as an API, or a black box, the inputs should be examined carefully. See Appendix figure A.4 and section B.3 for more detail on the code architecture and possible solutions. The image step has been validated to still be the step of the GPU buffer, while the width and height are the new size of the ROI given the two transforms applied. These should be correct, but we also experimented with other inputs with no result. It is also possible that obscure properties of the GPU buffer obtained from DeepStream are causing memory errors when they would otherwise be unexpected–in which case a domain expert will be needed to find where the problem lies. Finally, the SiftGPU library we use is actually a custom edit that allows the GPU buffer to be passed in rather than generated by the library when given a filename. Thus it is difficult to be certain if the GPU buffer we pass in violates certain assumptions of the library. More exploration of the library's code is required.

5.5.2 Partial ASIFT Results

With only rotations in ASIFT working, we tested with images far apart in the VIRAT sequence and were still able to obtain quality matches. In Figure 5.10 it is shown how

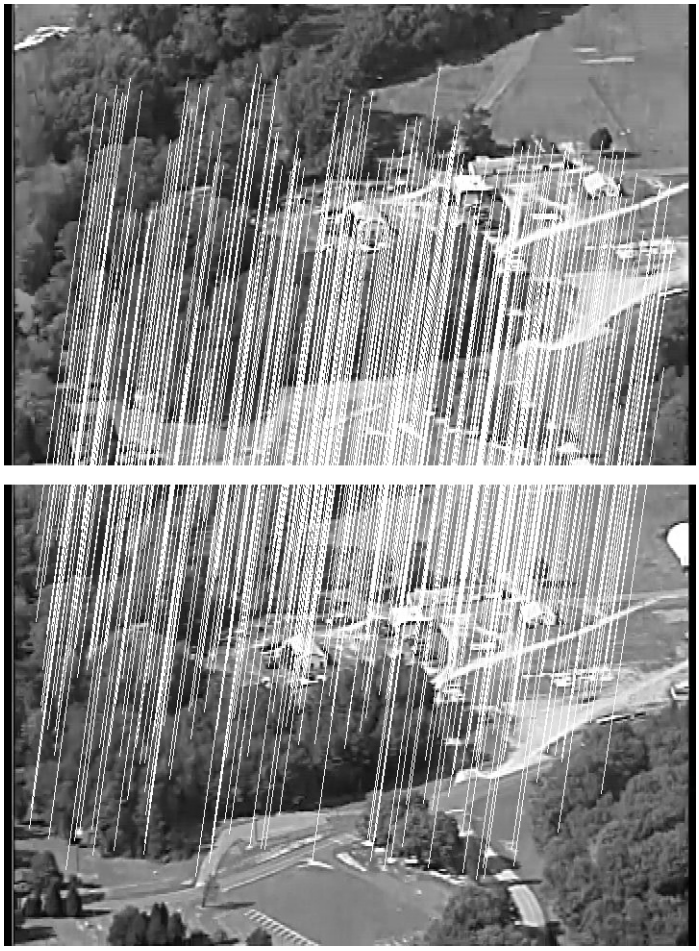


Figure 5.10: Frame 1 and Frame 17 of the VIRAT sequence, matched with ASIFT. It is clear how ASIFT accounts for the camera's movement.

the algorithm handles aerial imagery far apart in a sequence, like reference frames in VMZ.

5.6 Integral Image Experimental results

As mentioned in chapter 3.4, we have developed two methods to compute the Integral histogram. While the prescan-transpose-scan method generates the non-weighted integral histogram and the tiling-diagonal-scan solution generates the weighted histogram, they are still comparable methods. The spatial decomposition of the weighted histogram method still works in the GPU to generate a standard histogram.

These methods can both be used with images of different sizes and by choosing different bin sizes, so we use these as the main two criteria to compare these methods.

Figure 5.11 shows the result of these experiments. In general, the computation time is very fast, proving to be a very useful application on the Jetson Xavier.

It is clear that the second proposed method, the tiling solution that also generates a weighted integral histogram, performs better in all cases. It also scales better as image size and bin size increases. This is because the scan solution is less parallelizable as image size increases, due to having to parallelize longer rows and columns. The filtering method employed in the spatial computation is more paralleizable with image size because the region computations work well with GPU paradigms.

5.7 Georegistration Experimental Results

5.7.1 Georegistration Issues

The georegistration module is responsible for much of the driver code because driver logic determines when warping is done. It is in this driver logic, and likely somewhere in one of the implementations before the georegistration module, that a bug appears to be present. As shown in 5.12, the final warps quickly drift and have faulty transformations far earlier than they should. This could be because of errors in the structure tensor points, or in the NCC matches. That being said, nothing appears to be wrong with the base functionality of this module itself. When fed groundtruth homographies it gives the same output at standard VMZ, as shown in figure 5.13.

5.7.2 Georegistration Results

We used the Georegistration module in conjunction with the other, incomplete modules, to generate an attempt at registrations. Figure 5.12 shows those results. Com-

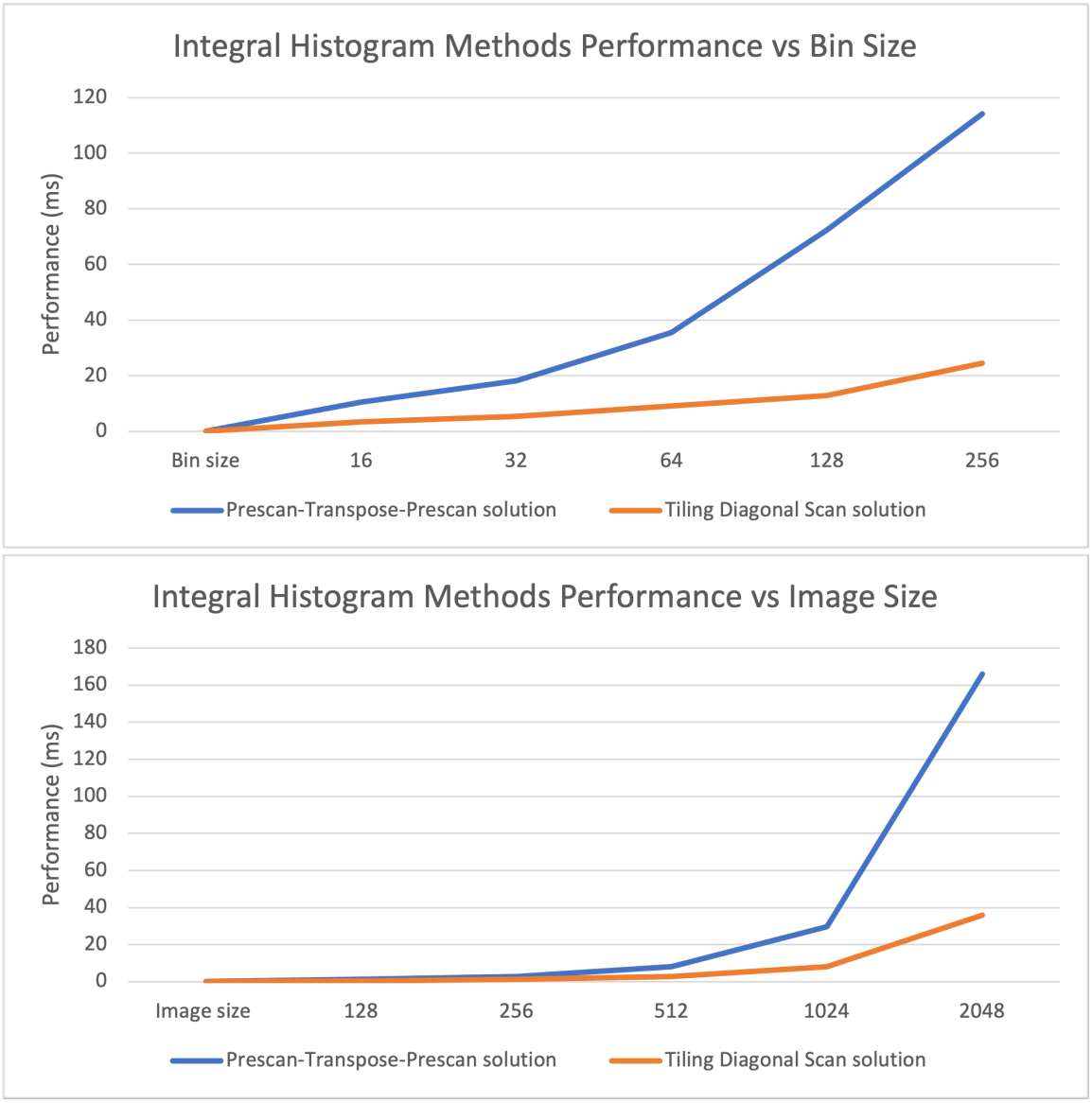


Figure 5.11: Comparing the performance of the two integral histogram methods. In the first case where bin size is varied, the images are 512x512. In the second, bin size is kept constant at 16. The tiling diagonal scan solution (using the weighted integral histogram) consistently performs better, and scales better as image size and bin size increase. All tests were run on the NVIDIA Jetson Xavier

pare this to figure 5.13 which were generated with the groundtruth homographies from the original VMZ code. These figures show there are errors in the other VMZ modules, but that the georegistration module works well when fed the correct input. For a more detailed discussion on the possible causes of this problem, see the code diagrams and discussion in Appendix A and B.

These georegistration results were generated with the module taking an average of 27.64 milliseconds to run on the Xavier with the VIRAT 720x480 images. This is about 36.2 frames-per-second. It is quite fast because the main operation is the GPU-enabled `nppiWarpPerspective_8u_AC4R` and there is not much extra complexity.

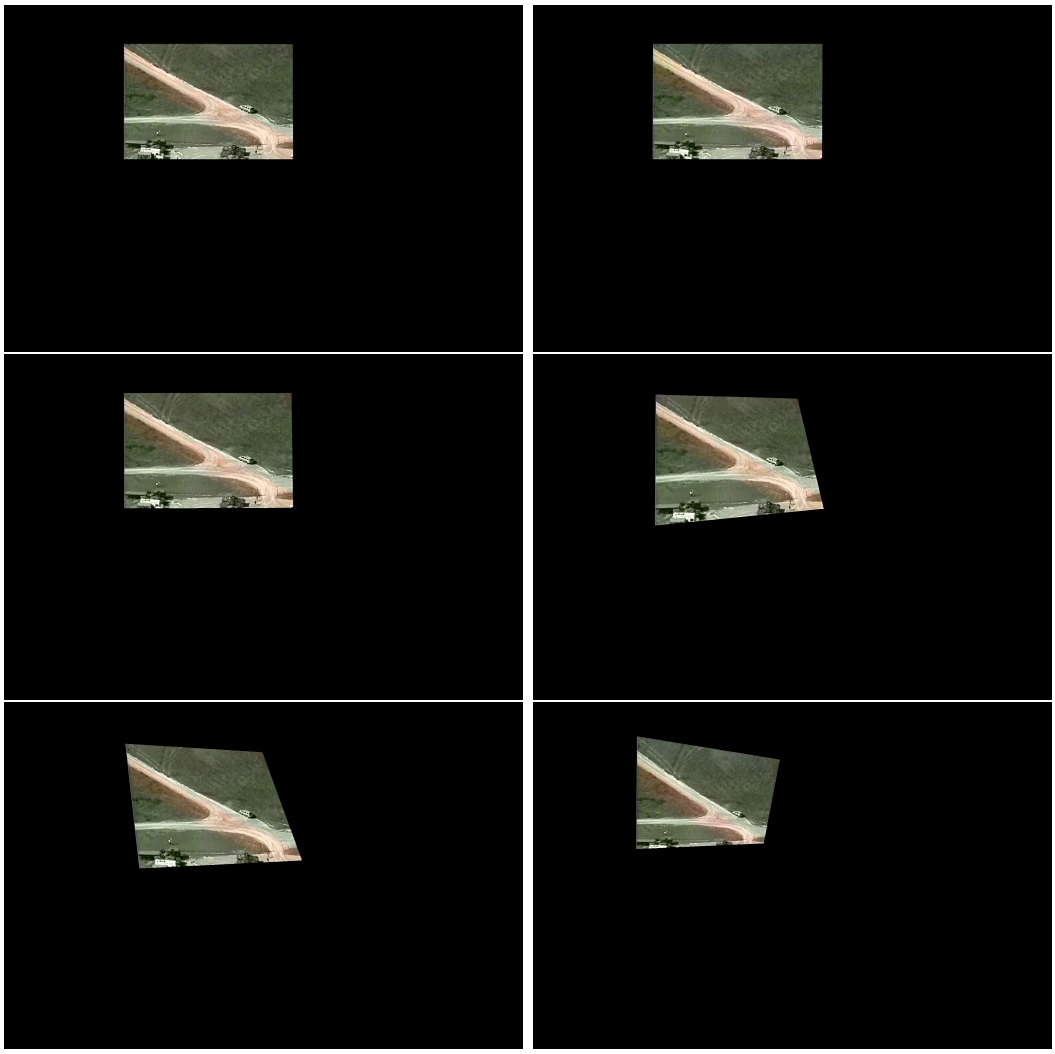


Figure 5.12: The warped results of the VMZ pipeline made up of the GStreamer elements presented here. The final element does georegistration, resulting in these images drawn to a large canvas. The top left image is frame 9, followed by 10-14 left-to-right and top-to-bottom. The first 11 frames of the sequence are warped reasonably well, maintaning a good representation of the true plane of the scene. At frame 12, the warps start to show errors and only get worse as the sequence goes on.

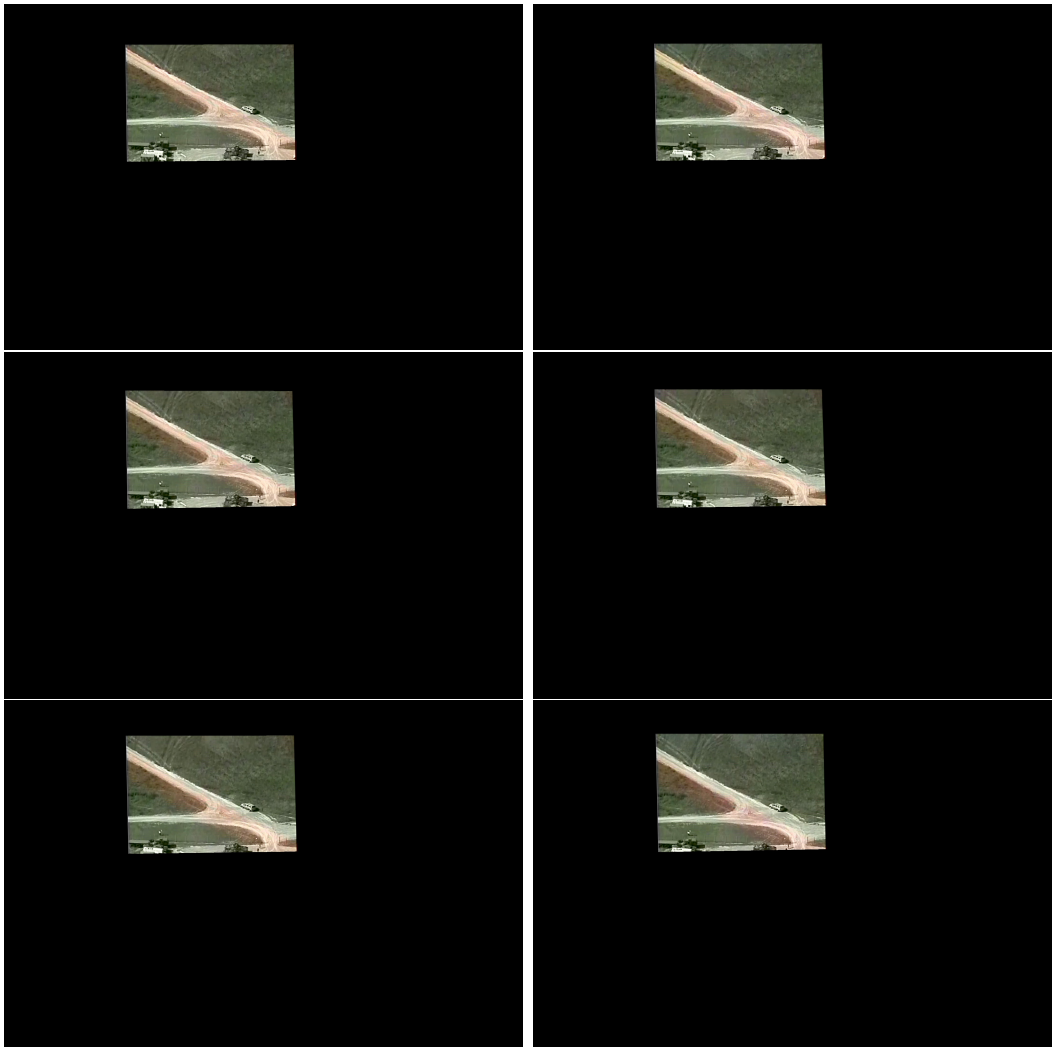


Figure 5.13: Here are the results of the GStreamer element when fed the groundtruth homography values for each frame. Because the warps look identical the true VMZ results, we at least know that the warping itself is not broken in this module. Rather, the results in figure 5.12 must either be a result of a previous module having a bug, or the VMZ logic in this module having an error.

Chapter 6

PyVMZ

6.1 PyVMZ

We have developed a Python codebase that runs VMZ with multi-threaded CPU modules, called PyVMZ. To increase the performance of this with powerful GPUs, we developed a version that uses the NPP GPU function we explored further in Chapters 3.2 and 5.4. To illustrate the modules of PyVMZ and their implementation details, see Figure 6.1. While mostly in Python, some of the depedencies are written in C++ for performance and access to certain APIs.

PyVMZ reproduces output of the original Matlab example. See Figure 6.2 for an example of this on the first shot of the VIRAT dataset, which consists of frames 1 through 247.

Table 6.1 shows the results of running this on a server cluster and on the Jetson Xavier. With capable GPUs, it approximately doubled the speed of VMZ. With the Jetson Xavier, the speed was slower but not significantly so. The nature of the code integration made certain code optimization impossible–the image data had to be copied onto the GPU for each NCC operation. This takes a large percent of the

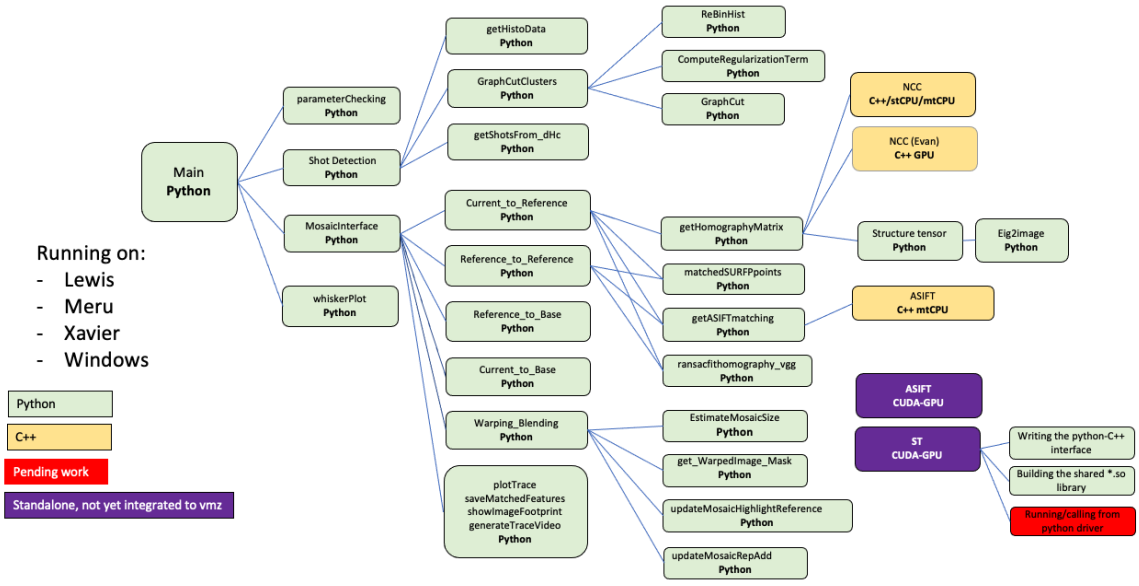


Figure 6.1: This diagram shows the code components of PyVMZ, starting from main.py, where the driver logic calls every other function referenced. In the case of NCC the two options can be seen, the multithreaded version (mtCPU) and the GPU version. In the bottom right are the GPU modules that have not yet been integrated into this pipeline.

time, so if a tighter integration were possible, with fewer GPU copies, VMZ would be even faster.

Each implementation is described as the following:

- **Matlab VMZ (Optimized):** The final version of VMZ in Matlab with mex-file integration for the C++ NCC & ASIFT code. Thus, it uses the same NCC implementation at PyVMZ (Multithreaded), below. Run with 20 threads for NCC & ASIFT.

- **PyVMZ (Multithreaded):** Original PyVMZ, previously the fastest implementation of VMZ on the VIRAT benchmark dataset. Written in PyVMZ with C++, multithreaded ASIFT and NCC. Run with 20 threads for NCC & ASIFT.

- **PyVMZ NCC GPU:** The contribution of this thesis, a version of PyVMZ using the GPU-enabled NCC from the Gstreamer implementation. This was run on the V100 GPU for best performance. Run with 20 threads for ASIFT.

Figure 6.2: The top image shows the original minimosaic generated from the first shot of the VIRAT data sequence. The bottom image shows PyVMZ output.

VMZ & PyVMZ Timing Results			
Version	**Overall Time (h:mm:ss)**	**FPS**	**Speed-Up**
Matlab VMZ (Optimized)	1:19:06	1.96	1.0
PyVMZ (Multithreaded)	0:51:36	3.01	1.53
PyVMZ NCC GPU	**00:36.3**	**4.20**	**2.14**
PyVMZ (Multithreaded) Xavier	1:40:57	1.53	0.78
PyVMZ NCC GPU Xavier	1:45:19	1.47	0.75

Table 6.1: Preexisting VMZ Matlab and PyVMZ code compared to new timings with a GPU module for NCC, as well as timings on the Xavier. The GPU module improves performance when run on a strong GPU, and maintained similar performance when run on the Xavier. This timing was done on the VIRAT dataset.

- **PyVMZ (Multithreaded) Xavier:** The same as PyVMZ (Multithreaded), run on the Jetson Xavier as an embedded device comparison point. Run with 20 threads for NCC & ASIFT.

- **PyVMZ NCC GPU Xavier:** The same as PyVMZ NCC GPU, run on the Jetson Xavier as an embedded device comparison point. Run with 20 threads for ASIFT.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The contributions of this work are as follows:

We have made significant progress towards porting the VMZ algorithm to embedded NVIDIA devices. Individually, we can perform the major steps of the VMZ algorithm. Performance of these modules proves to be near-real-time. We have identified the modules with errors and possible causes for next steps to be taken with the DeepStream modules. We have implemented parts of the underlying VMZ driver logic, identified why that is difficult in a streaming environment, and which parts need improvement.

It was shown that GStreamer is a powerful tool for image processing applications on embedded devices, although the programming model requires a significant change in how one thinks about driver and main logic of a program, especially when trying to maintain modularity.

We also developed an efficient GPU-based weighted Integral histogram method that can be used for matching problems.

We made significant contributions to PyVMZ, a VMZ implementation in Python. Our implementation using a GPU implementation of NCC achieves the fastest execution time seen by VMZ on the full VIRAT dataset.

7.2 Future Work

Here are future tasks to broaden and continue the impact of this work.

Our DeepStream implementation of VMZ needs changes to be complete, and there is plenty of room for improvement. The ASIFT algorithm needs simulated tilts on GPU to be fully completed. This involves solving the code out-of-memory access issue present in either our DeepStream implementation or the library we used for the SIFT feature detector. The driver logic that implements the Current-to-Reference calculations needs further evaluation to find where in the multiple stages of the pipeline there are problems preventing the code from generating results similar to VMZ. If we could abstract the driver logic out of the modules somehow, it would likely be much easier to debug. Further stages of VMZ, such as the pre-processing steps and the blending steps, could also be implemented on the Xavier for the GStreamer pipeline. It would also be ideal to decouple the DeepStream GPU-buffer access logic from the modules. If we did this, the pipeline could work with other Gstreamer GPU pipelines, not just pipelines with Deepstream managing the input and output.

With more modularity, we could also try other kinds of matching, like deep learning matching. This could provide more generalizability, and run very well on the hardware of the Jetson Xavier which has been optimized for machine learning.

Testing on an actual drone would prove very valuable, and we could even try to determine the optimal split between modules run on-device and on the ground station to quickly produce desired results.

More modules of PyVMZ could be integrated with GPU code. If we could find a

way to share GPU buffers and do less reallocation of frames, the NCC module and others could be significantly faster. Improving PyVMZ has the benefit of being able to run the code in more places and being easier for programmers to edit.

Of course, one of the major goals of VMZ is the generation of a meta-mosaic, and that is still future work. This will likely involve matching between the mini-mosaics, which could involve a deep learning-enabled method, or a robust hand-crafted feature approach similar to ASIFT used here.

Appendix A

Deepstream VMZ Code Diagrams

A.1 Overall Architecture and Logic

Diagram A.1 demonstrates the 4 VMZ Deepstream modules, how they work together, and the major logical components and decision trees for each.

A.2 Individual Deepstream Module Code Diagrams

Figures A.2, A.3, A.4, and A.5 show how each Individual module (the dark blue blocks of figure A.1) operates at a more detailed level. It also shows where some of the bugs still exist.

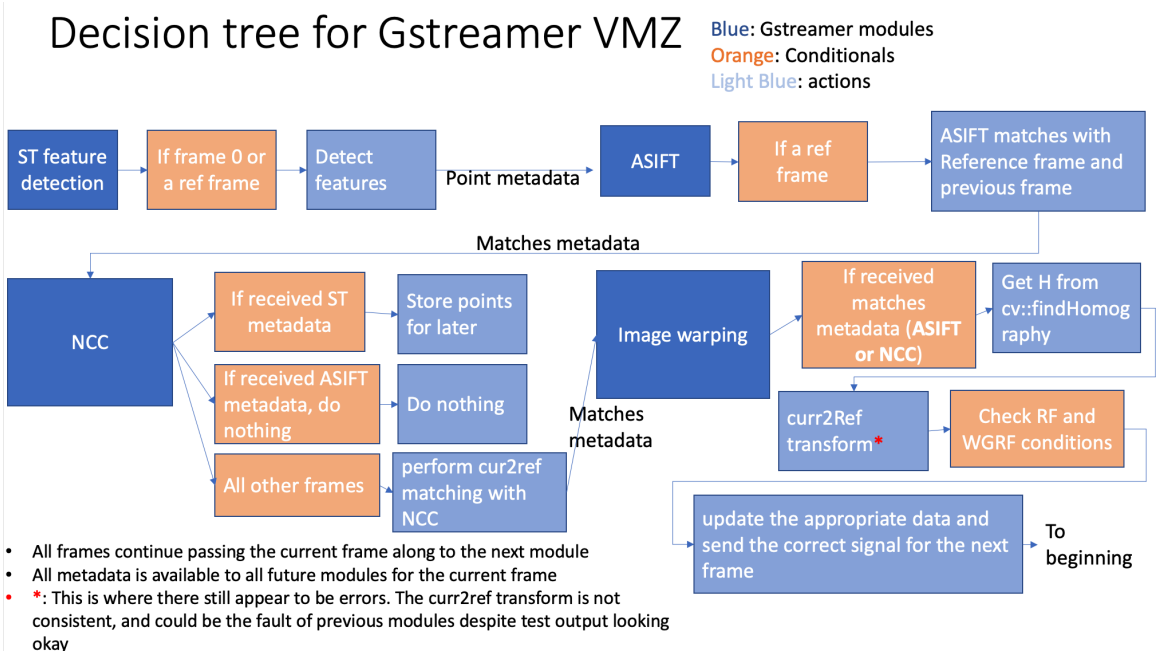


Figure A.1: This graph shows how the modules work together to perform VMZ, with their logical and conditional components. This shows where the current-to-reference logic is being done, which is still estimating the erroneous homography values.

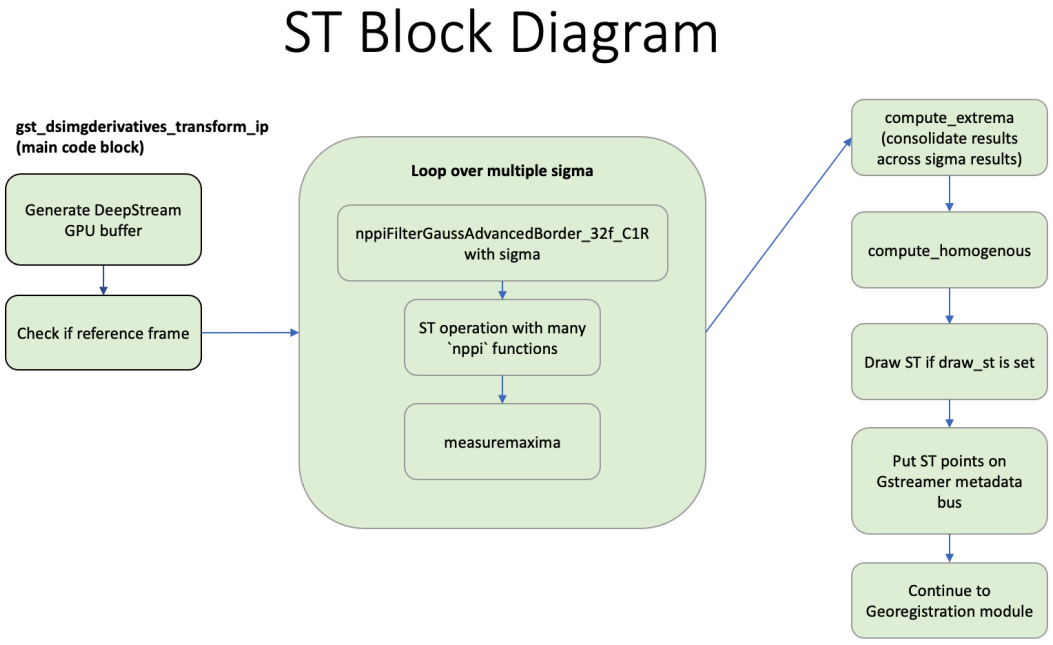


Figure A.2: This diagram represents the logical components of the structure tensor DeepStream element. It is a standard impplementation, using multiple sigma values for the Gaussian to ensure a good response.

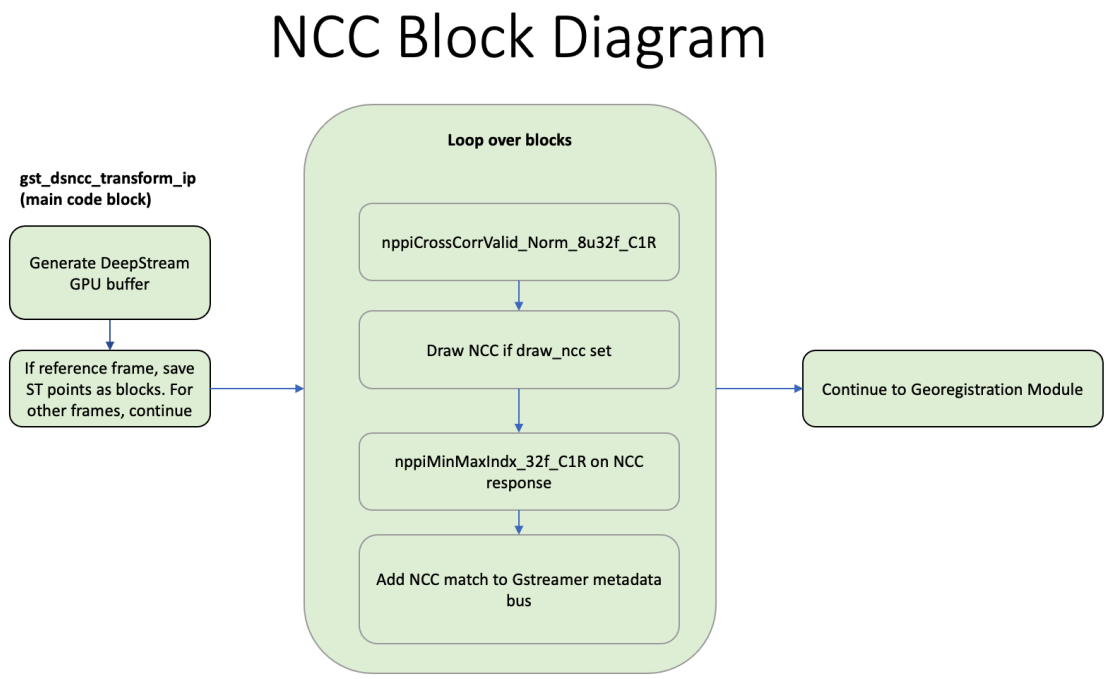


Figure A.3: This diagram shows the logical components of the NCC DeepStream element.

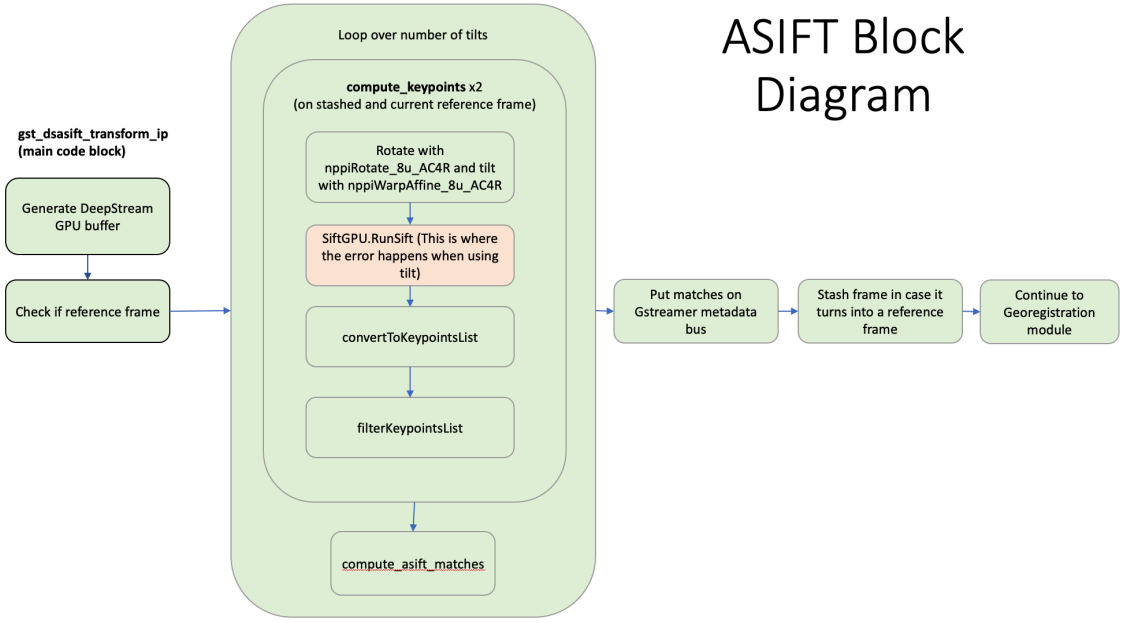


Figure A.4: This diagram shows how the ASIFT code module operates. Recall that ASIFT is incomplete because of an error when performing tilts. Here you can see the operations that lead up to that, and where calling the SiftGPU returns this error.

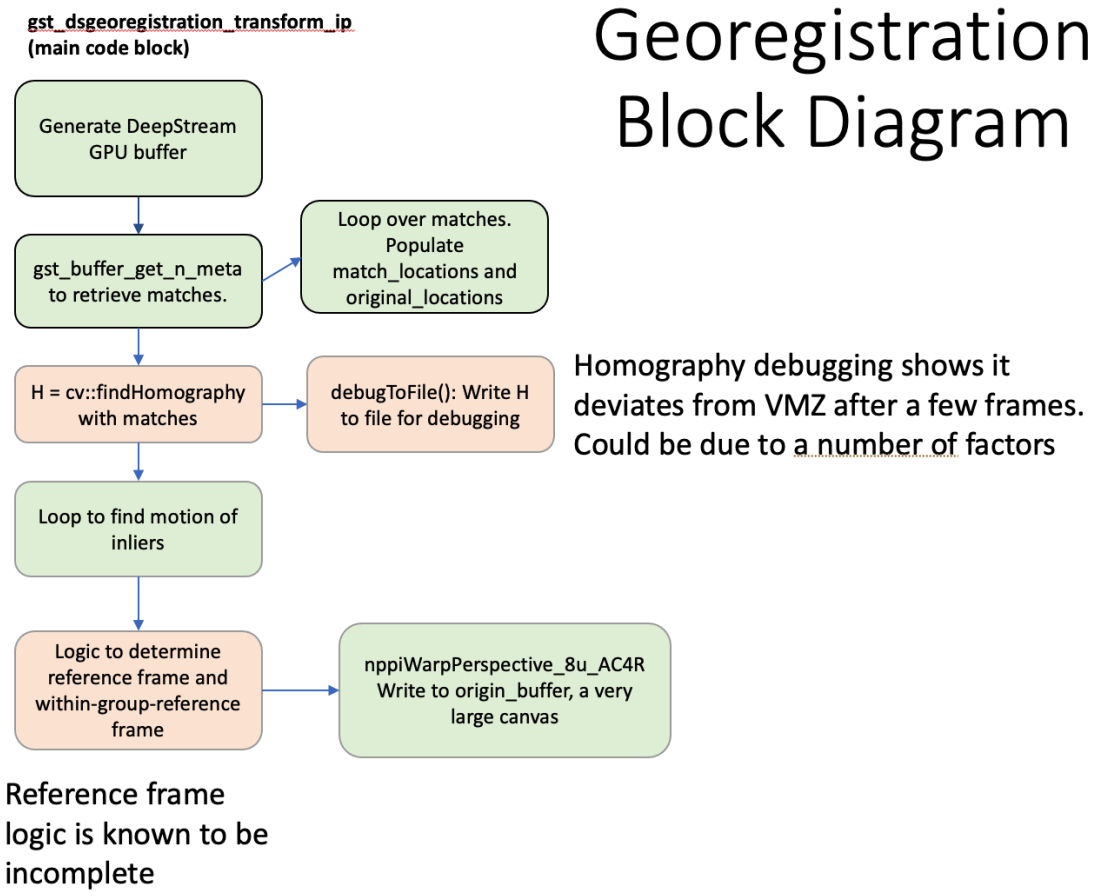


Figure A.5: This diagram shows the logic of the Georegistration DeepStream element. This shows where the homography is generated from matches retreived from other elements in the pipeline. The accuracy of this homography is dependent on the input into this element, but here is where to look for the homography errors discussed in 5.7.

Appendix B

Deepstream VMZ Code Problems

B.1 Code Profiling

Gstreamer and NVIDIA code profiling interaction is not well documented. NVIDIA's code profiling tool 'nvprof' does not work with the Gstreamer elements as far as we could tell. As such, we were unable to determine exactly what was taking the bulk of the time for elements that could have their timing improved, like the structure tensor element. One reason for the modules slower speed could be the surrounding Gstreamer setup and teardown logic, especially because of the steps involved in extracting the GPU buffer from Deepstream every time. Someone trying to speed up these modules might first looking at isolating the modules from Deepstream so that the setup and teardown only happens once at the beginning and end of the pipeline, rather than during every module.

B.2 Homography Estimation Errors

For the broken Current-to-Reference homography estimation, future debugging should focus on the inputs to the georegistration module. This would be the structure tensor module and the normalized cross correlation module. NCC has been tested extensively and validated against the VMZ version of NCC, so is unlikely to be giving faulty matches. It is likely then a fault in the structure tensor logic. While the output of the ST module appears correct, an in-depth analysis could be made, in a way that compares the ST detections and homogonenous regions of VMZ to Deepstream VMZ. The difficulty here is that the NPP functions will likely give slight variations when compared to VMZ CPU methods, so results will need to be within a margin of error (similar to the evaluation done on NCC).

Also worth noting for future corrections to the pipeline is the exact frame where errors begin to show. Figure 5.12 indicates that it is frame 12 where homographies begin to deviate in a significantly erroneous way. When examining the homography values for frame 10-12, one will notice a significant change for H[2][1]: -57.9377, -40.9728, 45.6832. There is a flip in the sign. It is worth evaluating ST and NCC on frame 12 to confirm there is nothing significantly different about frames 12 and beyond that is causing an edge case. The interesting thing is that this is not accumulating error, it is deterministic and is always happening this way when comparing frames 12 and beyond with frame 1 as the reference frame.

Another possible fix to the ST module involves using another ST library, mentioned in the next section.

B.3 ASIFT Tilt

The ASIFT memory access problem, at first glance, seems like it should be easy to determine how to fix. However, all standard debugging for this kind of problem has

not solved the issue. As shown in A.4, the error is happening in the SiftGPU library after the Affine function is applied to generate a tilt. The NPP rotate function is working fine here, it is only the tilt causing the issue. The output has been written to a file (although this and the image boundary being used could be verified again), and the affine shift is being applied correctly. When combined with the rotate API, it is also appearing correctly. The input to SiftGPU includes the image stride (number of bytes in a row of the image), and those have been validated as well. From the outside, it is unclear what is responsible for the error in the SiftGPU library. Future debugging will need to follow the path of the GPU data in the SiftGPU library. It is worth nothing we have made minor changes to SiftGPU to allow direct GPU pointer input, where it previously required CPU data or a filename.

It is also worth noting that ASIFT was the most difficult module to convert and has a large surface for possible errors. ASIFT was converted by taking existing code from the authors of ASIFT that was designed with and uses multi-threaded CPU operations, including for SIFT. The redesign for the GPU implementation in this thesis tries to recreate the overarching logic that loops over rotations and tilts, using the GPU operations mentioned from NPP in 5.5. The SIFT CPU implementation has been replaced with SiftGPU, a third-party library that may not give the exact same output as the ASIFT SIFT(CPU) library. This output is then used with the matching code from ASIFT, which remains on CPU because it was quite complex to account for all of the rotations and tilts.

B.4 Other Code Options

Another tactic to solving the code problems would be to use different libraries for the modules. Another option for structure tensor is the structure tensor GPU implementation used by VB3D, a fast and verified implementation. The main hurdle

to implementation here is a build system problem. Gstreamer and the Deepstream components use Makefiles and the basic make system to build each shared library. VB3D ST is built with CMake, so an attempt will need to be made to convert the Deepstream modules to use CMake.

To fix the ASIFT problem, we might be able to replace the NPP tilt API with our own GPU kernel, so we have more control over what is happening to the GPU buffer. There is also an option to switch the ASIFT code to a CPU implementation, like the implementation used by existing VMZ. This will not be trivial to implement with the Deepstream module because of dependencies on parallelizing via OpenMP. Also, it will result in a significant slowdown because frames will need to be copied to the CPU every time ASIFT is performed. Still, this should result in a working Reference-to-reference component.

Bibliography

[1] Augusto Vega, Chung-Ching Lin, Karthik Swaminathan, Alper Buyuktosunoglu, Sharathchandra Pankanti, and Pradip Bose. Resilient, UAV-embedded real-time computing. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pages 736–739, New York City, NY, USA, October 2015. IEEE.

[2] Matthew Brown and David G. Lowe. Automatic Panoramic Image Stitching using Invariant Features. *International Journal of Computer Vision*, 74(1):59–73, August 2007.

[3] Hernni Goncalves, Lus Corte-Real, and Jos A. Goncalves. Automatic Image Registration Through Image Segmentation and SIFT. *IEEE Transactions on Geoscience and Remote Sensing*, 49(7):2589–2600, July 2011. Conference Name: IEEE Transactions on Geoscience and Remote Sensing.

[4] Hoang Trinh, Jun Li, Sachiko Miyazawa, Juan Moreno, and Sharath Pankanti. Efficient UAV video event summarization. In *Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012)*, pages 2226–2229, November 2012. ISSN: 1051-4651.

[5] Tao Yang, Jing Li, Jingyi Yu, Sibing Wang, and Yanning Zhang. Diverse Scene Stitching from a Large-Scale Aerial Video Dataset. *Remote Sensing*, 7(6):6932–6949, June 2015. Number: 6 Publisher: Multidisciplinary Digital Publishing Institute.

[6] Raphael Viguier, Chung Ching Lin, Hadi AliAkbarpour, Filiz Bunyak, Sharathchandra Pankanti, Guna Seetharaman, and Kannappan Palaniappan. Automatic Video Content Summarization Using Geospatial Mosaics of Aerial Imagery. In *2015 IEEE International Symposium on Multimedia (ISM)*, pages 249–253, December 2015.

[7] Rumana Aktar. *Video Summarization Using Mosaicing and Activity Maps for Aerial and Biomedical Imagery*. PhD thesis, University of Missouri–Columbia, July 2022.

[8] Hadi AliAkbarpour, Kannappan Palaniappan, and Guna Seetharaman. Parallax-Tolerant Aerial Image Georegistration and Efficient Camera Pose Refinement-tWithout Piecewise Homographies. *IEEE Transactions on Geoscience and Remote Sensing*, 55(8):4618–4637, August 2017. Conference Name: IEEE Transactions on Geoscience and Remote Sensing.

[9] Evan Teters, Kannappan Palaniappan, Guna Seetharaman, and Hadi AliAkbarpour. Real-time geoprojection and stabilization on airborne GPU-enabled embedded systems. In Kannappan Palaniappan, Gunasekaran Seetharaman, and Peter J. Doucette, editors, *Geospatial Informatics, Motion Imagery, and Network Analytics VIII*, page 17, Orlando, United States, April 2018. SPIE.

[10] J.P. Lewis. Fast Template Matching. *Vis. Interface*, 95, November 1994.

[11] Kai Briechle and Uwe D Hanebeck. Template matching using fast normalized cross correlation. In *Optical Pattern Recognition XII*, volume 4387, pages 95–102. International Society for Optics and Photonics, 2001.

[12] Jae-Chern Yoo and Tae Hee Han. Fast normalized cross-correlation. *Circuits, Systems and Signal Processing*, 28(6):819, 2009. Publisher: Springer.

[13] Durgaprasad Gangodkar, Sachin Gupta, Gurbinder Singh Gill, Padam Kumar, and Ankush Mittal. Efficient Variable Size Template Matching Using Fast Normalized Cross Correlation on Multicore Processors. In P. Santhi Thilagam, Alwyn Roshan Pais, K. Chandrasekaran, and N. Balakrishnan, editors, *Advanced Computing, Networking and Security*, pages 218–227, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[14] Xiaotao Wang and Xingbo Wang. FPGA-based parallel architectures for normalized cross-correlation. In *IEEE Int. Conf. on Information Science and Engineering*, pages 225–229, 2009.

[15] Jakob Santner, Christian Leistner, Amir Saffari, Thomas Pock, and Horst Bischof. PROST: Parallel robust online simple tracking. In *IEEE Conf. on Computer Vision and Pattern Recognition*, pages 723–730, 2010.

[16] Sarala Arunagiri and Jaime Jaloma. Parallel GPGPU stereo matching with an energy-efficient cost function based on normalized cross correlation. In *Image Processing: Algorithms and Systems XI*, volume 8655, page 86550X. International Society for Optics and Photonics, 2013.

[17] Y. Fouda and K. Ragab. An efficient implementation of normalized cross-correlation image matching based on pyramid. In *IEEE International Joint Conference on Awareness Science and Technology Ubi-Media Computing (iCAST 2013 UMEDIA 2013)*, pages 98–103, 2013.

[18] Jean-Michel Morel and Guoshen Yu. ASIFT: A New Framework for Fully Affine Invariant Image Comparison. *SIAM Journal on Imaging Sciences*, 2(2):438–469, January 2009.

[19] Yang Huachao, Zhang Shubi, and Wang Yongbo. Robust and Precise Registration of Oblique Images Based on Scale-Invariant Feature Transformation Algo-

rithm. *IEEE Geoscience and Remote Sensing Letters*, 9(4):783–787, July 2012. Conference Name: IEEE Geoscience and Remote Sensing Letters.

[20] Mahdieh Poostchi, Kannappan Palaniappan, Filiz Bunyak, and Guna Seetharaman. Realtime motion detection based on the spatio-temporal median filter using GPU integral histograms. In *Proceedings of the Eighth Indian Conference on Computer Vision, Graphics and Image Processing - ICVGIP '12*, pages 1–8, Mumbai, India, 2012. ACM Press.

[21] Mahdieh Poostchi, Kannappan Palaniappan, Da Li, Michela Becchi, Filiz Bunyak, and Guna Seetharaman. Fast integral histogram computations on GPU for real-time video analytics. *arXiv preprint arXiv:1711.01919*, 2017.

[22] Sangmin Oh, Anthony Hoogs, Amitha Perera, Naresh Cuntoor, Chia-Chih Chen, Jong Taek Lee, Saurajit Mukherjee, J. K. Aggarwal, Hyungtae Lee, Larry Davis, Eran Swears, Xioyang Wang, Qiang Ji, Kishore Reddy, Mubarak Shah, Carl Vondrick, Hamed Pirsiavash, Deva Ramanan, Jenny Yuen, Antonio Torralba, Bi Song, Anesco Fong, Amit Roy-Chowdhury, and Mita Desai. A large-scale benchmark dataset for event recognition in surveillance video. In *CVPR 2011*, pages 3153–3160, June 2011. ISSN: 1063-6919.

[23] Nafis Ahmed, Evan Teters, Rumana Aktar, Ismail Akturk, Guna Seetharaman, and Kannappan Palaniappan. GPU and multi-threaded CPU enabled normalized cross correlation. In *Geospatial Informatics X*, volume 11398, page 1139805. SPIE, April 2020.

[24] Changchang Wu. SiftGPU : A GPU Implementation of Scale Invariant Feature Transform (SIFT). 2007.

VITA

I am Evan Teters, born in Springfield, Missouri. I received my Bachelors in Computer Science from the University of Missouri-Columbia in Spring 2019. I started my Masters in Computer Science in the Fall of 2019. After a year of working on my Masters full-time, I took a position as a Mobile developer for Quarkworks, Inc, where I would later become a Mobile Team Lead. Currently I work as Android Team Lead at Handshake.