Image-Based Motion Estimation in a Stream Programming Language

by

Abdulbasier Aziz

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2007

Author . . . .
Department of Electrical Engineering and Computer Science
June 29, 2007

Certified by . . . .
Saman Amarasinghe
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Image-Based Motion Estimation in a Stream Programming Language

by

Abdulbasier Aziz

Submitted to the Department of Electrical Engineering and Computer Science
on June 29, 2007, in partial fulfillment of the
requirements for the degree of
Masters in Electrical Engineering and Computer Science

Abstract

In this thesis, a computer vision application was implemented in the StreamIt language. The reference application was an image-based motion estimation algorithm written in MATLAB. The intent of this thesis is to contribute a case study of StreamIt, an architecture-independent language designed for stream programming. In particular, it observes StreamIt's ability to preserve block structure and to handle otherwise complicated functionality in an elegant fashion, especially in comparison to the reference implementation, coded in MATLAB. Furthermore, we observe a runtime of the StreamIt-coded application 2% faster than that of MATLAB. Lastly, the potentials of boosted performance with parallelization of subroutines within the program are discussed.

Thesis Supervisor: Saman Amarasinghe
Title: Associate Professor

Acknowledgments

I am indebted greatly to those in the StreamIt group who helped guide me throughout this process. I am especially indebted to Bill Thies, whose patience, knowledge, and willingness to help went far beyond expectation. I also want to thank Arjuna Balasuriya whose work motivated this thesis, and Allyn Dimock for helping get the code through the compiler.

I further want to thank the other members of the StreamIt group, most notably Michael Gordon and David Zhang. I am especially grateful to my advisor, Saman Amarasinghe, for his encouragement and pulling through for me especially towards the end.

Thanks also to Selçuk Bayraktar who helped me get acquainted with LaTeX and with how not to be such a “huge nerd”.

Most importantly, I thank my Creator, the Almighty, from Whom I came and to Whom I return.

Contents

List of Figures

Chapter 1

Introduction

As applications in Machine Vision and Robotics advance in complexity and sophistication, so too do they advance in their need for faster rates of computation. The convergence of more precise and diverse sensor equipment, the advent of new techniques in image processing and artificial intelligence, and the increasing pervasiveness of embedded computing has effected a need for faster computation. Despite the advances in hardware and information processing algorithms, however, there still remain apt algorithms in the field that are either slow or foregone for alternatives because they cannot achieve real-time or sufficiently fast performance. Real-time image-based localization or even 3D reconstruction can have tremendous impact, especially when considering the increased pervasiveness of computing and image-capturing devices. Often, it is due to rate-limiting bottlenecks, executed on uni-processor architectures, which prevent these algorithms from achieving their maximum potential.

A feasible alternative that addresses this problem is to exploit the parallelism inherent to multi-core processors to free up computation-intensive tasks that act as a bottleneck within the running program. Furthermore, it is appropriate that a programmer trying to optimize such programs employ a streaming model for the sensor information that the robot or computer needs to process. The streaming model for programming entails that the programmer writes independent processing kernels and stitches them together by using explicit communication channels.

Although this model has existed for decades, and although many languages employ the notion of a stream as a programming abstraction, many of these languages have had their shortcomings. Despite their being often theoretically sound and elegant, they commonly lack a sufficient feature set and are too inflexible to adapt to a variety of different architectures. Hence, despite a program's being apt for a streaming model, programmers opt instead for a more general-purpose language like C or C++ [1]. Compilers as they exist are not "stream-conscious" and hence performance-critical blocks of code are optimized at the assembly level, and must be adapted for each architecture that exploits them. This process takes time and introduces new arenas in which to error. In particular, it is often not feasible for programmers in the field of robotics to be accustomed to such low-level optimization, and hence sometimes algorithms are entirely foregone for slower or otherwise compromised methods.

Furthermore, there is a complexity and messiness involved with coding a streaming algorithm in an imperative language. If the underlying algorithm lends itself to a streaming model, coding a sequential program is often tricky, and coding a parallelized version is heroic. The programmer is compelled to trade off productivity and programmability because such a language does not handle the underlying model well.

Enter the StreamIt language, developed at the MIT Computer Science and Artificial Intelligence Laboratory. The language is designed to facilitate the programming process for large applications appropriate for the streaming model, as well as their mapping to a wide set of architectures, including multicore architectures and workstation clusters. The StreamIt language has two goals.

1. Firstly, it must provide high-level abstractions that improve robustness programmer productivity while streaming.

2. Secondly, it ought to serve as a common language for multicore processors, and hence to enable a single program to execute in parallel across any number of available cores.

The intent of this thesis is to confirm the ability of the StreamIt language to achieve these aforementioned ends, and to do so by exposing the advantages it offers over the standard implementation of a vision-based program. In particular, we employ an image-based motion estimation algorithm, where the streaming items are the images themselves. The algorithm takes pairs of images and calculates the transformation between them by generating a homography matrix. Hence, this thesis serves as a case study to provide insight to the development of a streaming program based on this particular algorithm. We discuss the performance on a uniprocessor system, and parallel evaluation is currently in progress.

This thesis is organized as follows. Chapter 2 includes a description of the image-based motion estimation algorithm. Chapter 3 relates some background information and related work. Chapter 4 discusses the basics of the StreamIt programming language. The next chapter addresses the implementation of the algorithm in StreamIt. Chapter 6 concerns StreamIt programmability, and Chapter 7 closes with a brief discussion of our implementation's performance and future work.

Chapter 2

Image-Based Motion Estimation

The subject of motion estimation pertains to where one is and where one is going based on some sensor input. Robotic motion estimation-be it on land, in sea, or up in the air–could use input from inertial measurement units, pressure sensors, GPS, or vision among others. Using vision, one has a series of raw images as input, and must determine one's own motion based upon these images and the transitions between them. The images themselves are rich in information, but the efficient analysis of them requires intelligent algorithms if the robot wishes to act upon this information in real-time. Once the motion of the agent is estimated, one can use this information in turn for localization if an initial state is known.

Below is a description of an algorithm used for motion estimation for an autonomous underwater vehicle (AUV). The algorithm, presented by Kalyan and Balasuriya [4], estimates motion based on a series of images, and is particularly useful in underwater environments where GPS information is not available. Although their work also exploits internal measurement unit (IMU) and sonar data to obtain a precise measurement of the ego-motion, this case study focuses solely on the image-based motion estimation since the focus is not necessarily on the accuracy of location but rather the speed of computation. The algorithm also has, it should be noted, applicability beyond the underwater environment in that it is a generic vision algorithm.

The main part of the algorithm can be divided into three main sections, detailed in Figure 2-1. The first is feature extraction, the second is sharpening and feature correspondence matching, and the last is the implementation of the RANSAC algorithm. The first stage isolates those patches of each image that are most trackable, according to the renowned Kanade-Lucas-Tomasi (KLT) algorithm for feature detection [22]. Although Kalyan and Balasuriya employ a Harris corner detection algorithm, both are quite comparable [10], and the Kalyan and Balasuriya implementation has a shortcoming that necessitated the implementation of the KLT algorithm here. This shortcoming is the difficulty with which variable-sized arrays are dealt with, a difficulty more fully described in Section 5.2.2. The second stage takes patches of a given size around each feature and suggests a putative matching between the features of two frames. The last stage takes a random number of those matched pairs, calculates the homography matrix between them, and sees whether that matrix is appropriate considering all the pairs. This RANSAC stage determines an appropriate homography matrix, and then calculates the rotation and translation between the two successive frames. It should be noted that this algorithm does not account for a change of camera depth (assumed to be constant), but does take into account rotation and translation. The end goal is to generate a 3 x 3 fundamental matrix that characterizes both rotation and translation for every pair of images. Note that StreamIt nicely maps the conceptual block structure of this algorithm to the actual program itself.

All three stages provide opportunities for parallelization and optimization, but this section is expressly concerned with the science behind the algorithm itself.

2.1 Feature Extraction

The first stage concerns the extraction of “interest points” or “features”-patches of pixels of a given size. According to Tomasi and Kanade [22], a good feature is one that, by definition, is easily trackable from one frame and the next. There are a host of fairly comparable algorithms that can detect these features, but the Kanade-Lucas-Tomasi (KLT) algorithm is a quite appropriate method to employ for our purposes [10].

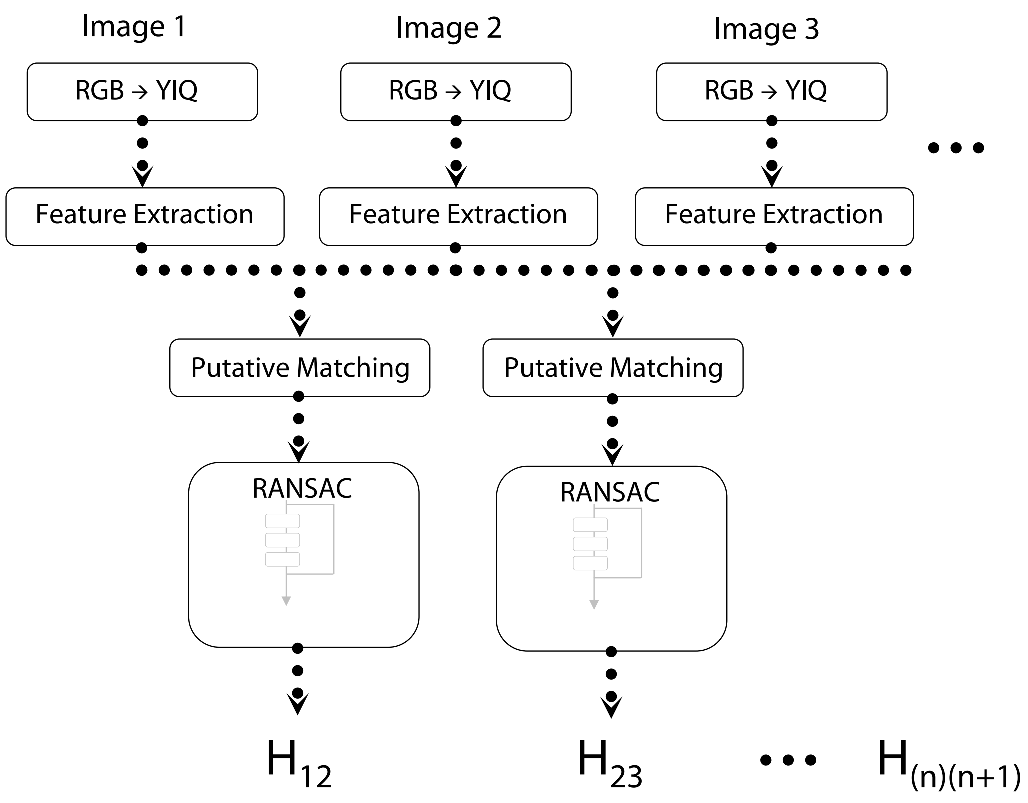


Figure 2-1: A Sample Block Diagram for the Image-Based Motion Estimation

As a camera moves, the patterns of image intensities change in a complex fashion. Any sequence of images can be characterized by a function of three variables $I(x, y, t)$, where the space variables x and y as well as the time variable t are discrete. Images taken at nearer time instants will be different from each other only slightly, and we imply this strong correlation by saying there are particular patterns that move in an image stream. We represent this by indicating a second point along that function $J(\mathbf{x}) = I(\mathbf{x}, \mathbf{y}, \mathbf{t} + \tau)$ such that $J(\mathbf{x}) = I(\mathbf{x} - \mathbf{d}) + n(\mathbf{x})$, where $\mathbf{d} =< \xi, \eta, 0 >$, where n is noise and $\mathbf{d}$ is the displacement vector.

What we try to track between successive frames is not pixels, since pixels change much due to noise and with respect to their neighboring pictures, but rather windows of pixels, which preserve texture quite well despite additions of noise.

When the displacement vector is very small, we can approximate the intensity function I by its Taylor Series approximation, and then truncate to the first-order term:

$$I(\mathbf{x} - \mathbf{d}) = I(\mathbf{x}) - \mathbf{gd} \tag{2.1}$$

where $\mathbf{g}$ is the gradient $< dI/dx, dI/dy >$

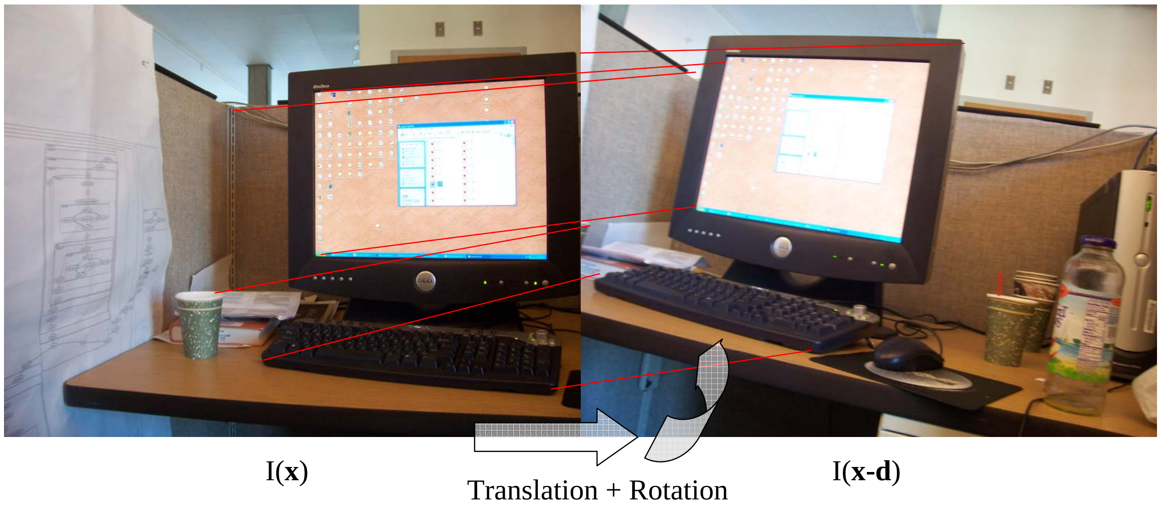


Figure 2-2: The Homography Matrix identifies rotations and translations between two frames

Now we can define a residue error

$$\epsilon = \sum_{W} [I(\mathbf{x}) - \mathbf{gd} - J(\mathbf{x})]^2 d\mathbf{x} = \sum_{W} (h - \mathbf{gd})^2 d\mathbf{x} \tag{2.2}$$

where $h = I - J$. To minimize this error, we take the derivative with respect to d and see the equation above equal to 0.

$$\sum_W (h - \mathbf{g}\mathbf{d})\mathbf{g}dA = 0 \tag{2.3}$$

Given that $(\mathbf{g} \cdot \mathbf{d})\mathbf{g} = (\mathbf{g} \cdot \mathbf{g}^T)\mathbf{d}$, we can say that

$$\sum_W (\mathbf{g} \cdot \mathbf{g}^T dA)\mathbf{d} = \sum_W \mathbf{h}\mathbf{g} \cdot dA \tag{2.4}$$

which can be written as

$$G\mathbf{d} = \mathbf{e} \tag{2.5}$$

where $G = \sum_W (\mathbf{g} \cdot \mathbf{g}^T dA)$, and $\mathbf{e} = \sum_W (I - J)\mathbf{g}dA$

As described by Tomasi and Kanade [22], it is the lesser of the eigenvalues of the 2 x 2 matrix G that we store for a particular pixel.

Once all the minor eigenvalues are determined for the part of the image under consideration, we order them. The number of best eigenvalues chosen is user-defined, and must be chosen such that all pixels' windows are in non-overlapping regions. Once this feature selection stage is completed, the program should have a list of the most trackable pixels, including their coordinates and minor eigenvalues.

2.2 Putative Matching

Now that two successive images each have their own list of features, we must see which features from the first image match with which of the second. The next stage involves matching these features between both frames that are maximally correlated with each other. That is, for every feature at (x_1, y_1) in the first image, the match with the highest neighborhood cross-correlation in image 2 is selected within a window centered at some (x_2, y_2). For each feature in image 2, the best match is similarly sought in image 1. On occasion, there will be the case in which a feature in one image is deemed a best neighbor by more than one feature in another image. In this case, we keep only the match with the highest cross-correlation.

Algebraically, if $I_k(x, y)$ and $I_{k+1}(x, y)$ are the two frames we're dealing with, and X_k represents the detected features in I_k, then the correlation measure we concern ourselves with is represented by the following:

$$C_m(X_{k,k+1}) = \frac{1}{\sqrt{\sigma^2_{(X_k)}\sigma^2_{(X_{k+1})}}} \sum_{i=-w}^{w} \sum_{j=-w}^{w} [I_k(x+i, y+j) - \bar{I}_k][I_{k+1}(x+i, y+j) - \bar{I}_{k+1}] \tag{2.6}$$

Here, σ^2 is the variance of a window's intensity and I is the patch's average intensity. Each pixel's intensity is normalized by this value $\bar{I}$, and in doing so we reduce the effects of ambient lighting.

2.3 RANSAC

The RANSAC stage is the final and most involved stage of the method, and furthermore introduces randomness to the output. Invoking the RANSAC (Random Sampling Consensus) stage implies that among the putative matches, there are some or many that do not in fact match.

The RANSAC algorithm is applied to the putatively matched set in order to estimate the homography between the frames and the correspondences which are consistent with that estimate, which we call "inliers". The RANSAC algorithm itself has a number of steps.

Each iteration of the RANSAC algorithm goes as follows. First, we generate a selection of a random 8 set of matches. We compute the homography of these 8 points, where the homography is a matrix H where x' = Hx, given x is the set of features in the first frame matched to x' in the second frame.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{2.7}$$

Objective: Compute the fundamental matrix between two images.
Algorithm

(i) Feature Extraction: Compute interest points in each image

(ii) Putative Correspondences: Compute a set of interest point matches based on proximity and similarity of their intensity neighborhood.

(iii) RANSAC robust estimation: Repeat for N samples, where N is determined adaptively.

  (a) Select a random sample of 8 correspondences and compute the fundamental matrix F.

  (b) Calculate the distance $d_\perp$ for each putative correspondence

  (c) Compute the number of inliers consistent with F by the number of correspondences for which $d_\perp <$ t pixels

  Choose the F with the largest number of inliers. In the case of ties, choose the solution that has the lowest standard deviation of inliers.

Figure 2-3: A summary of the RANSAC algorithm, adapted from Hartley and Zimmerman [13]

Computing a normalized homography matrix (or "fundamental matrix", a term used interchangeably) requires 8 points according to the algorithm described in Hartley and Zisserman, and it itself requires the normalization of the matches, an SVD calculation, and a denormalization to find the matrix H.

Then, the matrix is applied to the first set of points x, and the distance between the second set of points in the putative matches and the generated x' is determined. Whatever distances lie below the user-defined threshold, t, are deemed inliers. Whatever differences lie above t are discarded, and after several iterations, the number of which is calculated according to the procedure detailed on Hartley and Zisserman[13, p.105]], the best inliers are kept.

This procedure yields an acceptable set of inliers and respective homography, and at this point the RANSAC stage is complete. For a given pair of images, the homography is discovered and that information is printed or stored.

Chapter 3

Background and Related Work

Many computer vision and robotics applications come under the domain of stream programming, where a flow of data is processed in a similar way for a potentially infinite amount of time. The background work that pertains to this thesis lies in the areas of stream programming in general and the domain of computer vision and image processing particularly. The previous section discussed the very particular vision algorithm implemented in the StreamIt programming language, but here we turn to the other work of relevance to this thesis that has been done in the two aforementioned fields. In particular, we discuss Synchronous Data Flow, other streaming languages, and the work of Matt Drake who implemented an MPEG-2 Encoder in StreamIt.

3.1 Synchronous Data Flow

Synchronous Data Flow (SDF) graphs are often used for Digital Signal Processing applications in embedded systems design or other robotics applications. The SDF model has been employed in the Ptolemy program, the COSSAP system in Aachen, Germany, and for programming the Warp at Carnegie Mellon University [17].

Each node of an SDF graph represents a functional module or block that consumes a fixed number of inputs and produces a fixed number of outputs, both of which are known a priori. The arcs between nodes connote the communication paths between these modules. An SDF system is designed assuming some source with an infinite amount of data that enters the system, and an output at a destination that pushes an unlimited amount of data out. With each invocation of a module, a fixed number of input samples are consumed, and a fixed number of samples are produced. FIFO buffers handle all the data between modules. This SDF model permits communication between modules and also parallelization among non-dependent modules.

With regards to streaming programs, the SDF model offers four particular advantages:

(a) Parallelism: non-dependent modules can execute simultaneously on different cores of a multi-core processing architecture.

(b) Memory consumption: at compilation time, a measure of each FIFO buffer can be computed and minimized by choosing a schedule that accomplishes this

(c) Execution Time Analysis:due to static scheduling, one can do a worst-case execution time analysis, which is important for real-time applications. One can also compute latency with this analysis[1].

(d) Determinism: unlike shared-memory multi-threaded programs, programs written with SDF are deterministic and deadlock-free.

As we discuss in the next section, all three aspects of the SDF model are exploited in the StreamIt language and enable it to accelerate the execution of a vision algorithm like the one described. The addition of dynamic rates for the execution of certain modules does slightly complicate matters, but StreamIt scheduling is still beneficial for many reasons detailed later. [20]

3.2 Other Streaming Languages

StreamIt is just one of many streaming languages developed. The streaming abstraction has been an integral part of many programming languages, including data flow,

[1]Such an analysis is not yet included in StreamIt

CSP, synchronous and functional languages; see Stephens [32] for a [13] review. Synchronous languages which target embedded applications include Lustre [12], Esterel [5], Signal [11], Lucid [3], Stampede [18], and Lucid Synchrone [7]. Other languages of recent interest are Brook [6], Spidle [8], Cg [16], StreamC/KernelC [14], and Parallel Haskell [2]. The principle differences between StreamIt and these languages are

(a) StreamIt adopts the Synchronous Dataflow [15] model of computation (with added support for dynamic rates), which narrows the application class but enables aggressive optimizations such as linear state space analysis,

(b) StreamIt's support for a "peek" construct that inspects an item without consuming it from the channel,

(c) the single-input, single-output hierarchical structure that StreamIt imposes on the stream graph, and

(d) a "teleport messaging" feature for out-of-band communication [21].

3.3 MPEG-2 Encoding/Decoding

Another program of comparable size implemented in StreamIt is the MPEG-2 encoder and decoder designed by Matt Drake [9]. This thesis attested to StreamIt's ability to improve programmability and to offer portable, clean, and malleable code that also improves performance. Drake observes a dramatic reduction in the number of lines of code when converting a reference C-based MPEG-2 decoder, from 6,835 lines to 3,165. He also posits that StreamIt captures the modularity of the underlying algorithm better than the reference implementation. Although many of Drake's observations with respect to StreamIt hold for his case, translating from MATLAB had a different set of challenges that we describe further in Chapter 6.

Chapter 4

StreamIt Programming Language

StreamIt is an architecture-independent programming language intended for streaming applications. The language adopts the cyclo-static flow model of data, which is a generalization of synchronous data flow. StreamIt 2.1 also supports dynamic communications rates, which play an important role in this thesis. The programs written in this language can be modeled as graphs where each node represents some block of computation and the edges represent a FIFO-ordered communication of data over tapes. One goal of the language is to facilitate the developer's mapping of block-level functionality to relatively autonomous or stand-alone blocks of code.

Since the compiler assumes responsibility for determining layout and task scheduling, the implementation can become portable across a diversity of architectures. This way, the developer can concern himself with an efficient implementation without having to worry about the specific mappings to hardware and low-level optimizations.

4.1 Filters

The most basic unit of functionality in StreamIt is the filter. The basic task of the filter is to remove data items off the input tape that it is reading, process those data items, and output some relevant information back onto the tape. Although each filter has only one input channel and one output channel, it may also receive information from the outside world through portals and control message sending, which is described later in section 4.5.

Although each filter can be initialized with a block of code prior to its first execution, the work function implements its steady-state functionality and performs, as its name suggests, the real work. So long as there is sufficient data on the input channel, the work function fires. It "pops", or removes a certain number of data items upon every invocation, processes the information as it is coded to do, then "pushes", or adds a number of data items back onto its output channel. If the programmer wishes to view data further upstream on the input channel, he can "peek" a certain number of data items up without having to remove those items from the channel. All the rates for this pushing, popping, and peeking are explicitly stated in the beginning so that the compiler can later optimize the scheduling so as to allow the fastest flow of data down the stream. A sample filter looks like the example below, where the input channel presents the filter with RGB color values that it converts to NTSC color values:

```
float->float rgb2ntsc(){
   work pop 3 push 3 peek 3{
           push(peek(0) * 0.299 + peek(1) * 0.587 + peek(2) * 0.114);
           push(peek(0) * 0.596 - peek(1) * 0.275 - peek(2) * 0.321);
           push(peek(0) * 0.212 - peek(1) * 0.523 + peek(2) * 0.311);
           pop();pop();pop();
   }
}
```

Figure 4-1: RGB Values are converted to NTSC (YIQ) Values

Note that each filter must also have its input and output channel types declared as well, which are both floats in the example above. The filter, and StreamIt in general, can also operate on other data types including ints, booleans, complex numbers, structs of all types, and arrays of all types. Either the input or the output can be of a void type.

Helper functions may also be invoked within a work function or the init function, and can serve as an auxiliary block of computation. Structured much like a C++ or Java function, the helper function contains a name, list of parameters, return type, and body. In the feature extraction stage, we use a recursive quick-sort helper function to order the eigenvalues computed in the previous step. Naturally, the helper functions are limited to the filters in which they are defined.

4.2 Pipelines

The task of putting the pieces together, i.e. arranging the filters and how they communicate with each other, is encapsulated in the structuring of a pipeline. A pipeline, like a filter, has an input and output type. The filters must be arranged in linear fashion with the output type of a preceding filter agreeing with the input type of the filter that follows it. Filters are added with the `add` keyword, followed by the name of the filter and instances of the variables passed in to that filter. Note that since the filters are made at compile time, these variables must remain constant throughout the duration of the program.

```
float->float pipeline IFFT1D(int size, int zeropad){

      add pipeline IFFTKernel (size+zeropad){
            add Conjugate();
            add pipeline FFTKernel2(size+zeropad){
                  add pipeline FFTReorder(size+zeropad){
                        for (int i = 1; i< (n/2); i *= 2)
                              add FFTReorderSimple(n/i);
                  }

                  for (int j=2; j<=size+zeropad; j*=2)
                        add CombineDFT(j);
            }

            add Conjugate();

            add DivideByN(N);
      }
      add pipeline unZeroPad(size*2, zeropad*2){

            split roundrobin(size, zeropad);
                  add Identity<float>;
                  add float->void filter {work pop 1{pop();}}
            join roundrobin(1, 0);
      }
}
```

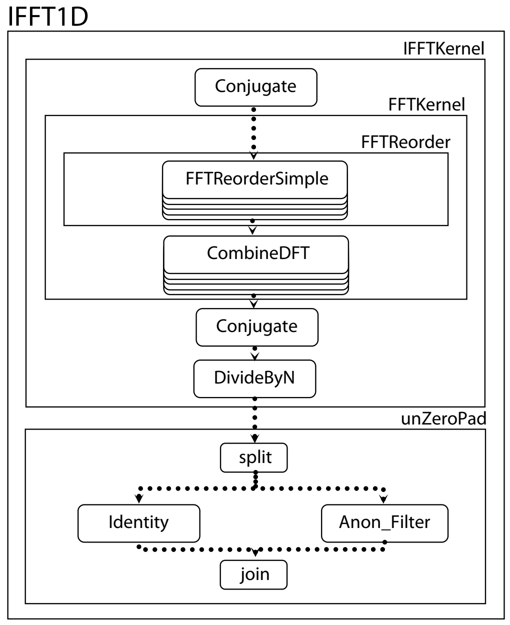


Figure 4-2: Inverse Fast Fourier Transform in StreamIt

4.3 Splitjoins

While filters and pipelines can be arranged serially within a program, any block arrangement that occurs in parallel is provided through the `splitjoin` statement. The `splitjoin` statement takes in one input and distributes it to a series of parallel streams, then recollects the output of those streams to be passed into the `splitjoin`'s output stream.

```
float->float splitjoin
 Transpose (int M, int N) {
    split roundrobin(1);
     for (int i = 0; i<N; i++){
        add Identity<float>;
     }
    join roundrobin(M);
 }
```

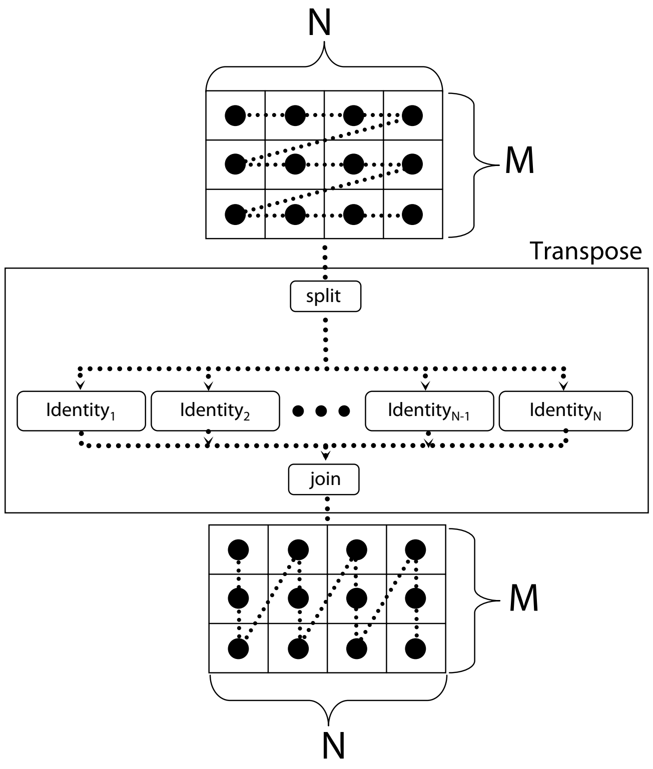


Figure 4-3: This filter takes a matrix in as a stream of `floats`, row by row, and outputs them column by column, thus transposing the matrix. Courtesy Bill Thies

There are a couple ways that the splitjoin gathers information from its input tape. If the splitjoin wishes to take one data item on its input channel and distribute that same datum to many child streams in parallel, it invokes a `split duplicate` command. If instead it wishes to distribute successive data on the input channel to the different child streams encompassed within it, it does the following. If one wished to take, say, the first 3 data items and push them into the first stream, and take the fourth data item and push it onto another stream, one would invoke the statement split roundrobin(3,1). Once the child streams are done processing, there is a join roundrobin() statement that acts much like the split roundrobin() statement, except that it performs the inverse operation, gathering data and pushing them onto the splitjoin's output tape.

4.4 Feedback Loops

Any iterative loop that requires the invocation of the same filter multiple times requires some form of feedback. The StreamIt `feedbackloop` accommodates iterative processes like the RANSAC loop that cannot be expressed within a filter but rather contain filters within. The forward path of the loop is defined with the statement `body`, and the feedback path is defined with the statement `loop`.

```
add featureListPair->featureListPair
      feedbackloop RANSAC {

      join roundrobin(1,1);
            body pipeline {


                  add selectIndices();


                  add makeFundMatrix();


                  add findInliers();
            }


            loop  FLPID_backloop;

      split duplicate;


}
```

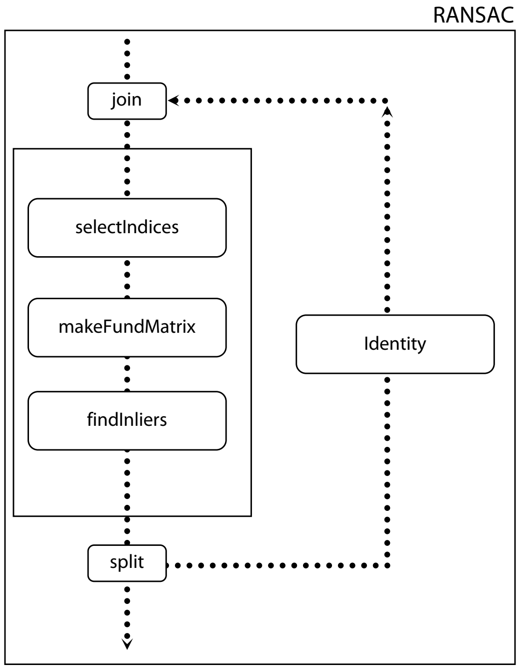


Figure 4-4: A simple `feedbackloop`, consisting of a joiner, splitter, forward path, and feedback path.

Feedback loops also have a specialized `push`-like statement, `enqueue`. Each enqueue statement pushes a single value on to the input to the joiner from the feedback path. There must be enough values enqueued to prevent deadlock of the loop components; enqueued values delay data from the feedback path.

4.5 Messaging

Although stream programming is well-suited to accommodate the normal flow of data, there exists a need often to communicate out-of-band signals or information from one filter to another outside of this regular flow. An example is a radio application where the listening frequency might change on occasion.

```
portal<selectIndices> feedbackOrInput;

add featureListPair->featureListPair
      feedbackloop RANSAC {

      join roundrobin(1,1);
            body pipeline {


                add selectIndices()
                        to feedbackOrInput;


                add makeFundMatrix();


                add findInliers(feedbackOrInput);
            }


            loop  FLPID_backloop;

      split duplicate;


}
```

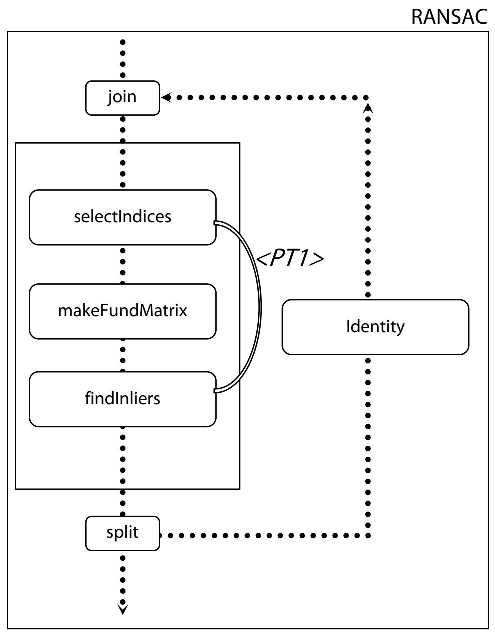


Figure 4-5: The same feedback loop from earlier is now capable of sending messages between the two filters `selectIndices` and `findInliers`.

Only filters (no pipelines) can receive messages, and they receive them through portals. Portals are a particular type of variable, defined within each pipeline, and associated with each message receiver type. They are defined with the `portal <receiverType>` statement. If one declares a portal as a local variable, and then passes that as a parameter to a child stream which adds a filter to the portal, the filter is also added to the portal in the original container. When a filter is created, it may be added to a portal by adding the keyword `to` and the portal variable to the end of the declaration.

4.6 Execution Model and Compilation

As mentioned before, each filter "fires" when it has a sufficient amount of data on its input tape. StreamIt has two states of execution for each filter: the initialization state, and the steady state. The steady state is achieved after the initialization state has primed the tapes so that peeking filters can execute their work functions once. A steady state is one that does not change the buffering in the channels. That is, at any iteration within the steady state, the amount of data on each channel should be invariant.

The StreamIt compiler takes care of deriving schedules for both initialization and steady states and also translates the StreamIt syntax. The output is a program in Java code, and this Java code can be run against a Java library or can be compiled into C code and linked with a runtime library. Because of the mapping of the code to the stream graph, the compiler can combine sequential filters, split filters apart, or even duplicate filters to run in parallel so as to optimize the runtime of the original program.

The Java library is especially convenient for debugging and testing, not to mention when invoking those functions of the compiler that have newly been added and might have bugs.

The StreamIt compiler defers low-level optimizations such as register allocation, code generation, and instruction scheduling to the existing compilers, and leaves the higher-level optimizations to itself. This approach allows for great deal of portability across architectures, in the sense that the compiled StreamIt code can run on so many machines [19].

Chapter 5

Motion Estimation Implementation

This chapter addresses the implementation of the motion estimation algorithm discussed in Chapter 2. The motion estimation algorithm, intended to find the homography matrix between two successive image frames, was originally coded by Kalyan and Balasuriya in MATLAB. In addition to translating the MATLAB code to StreamIt, we had to write some of the library functions that were invoked in MATLAB, like convolution and singular value decomposition. The full stream graph can be found in Appendix A.

5.1 Readying the stream

In practice, the program would begin with an input channel whose buffer is filled with integer values corresponding to the red, green, and blue values of each pixel in the image. We simply begin with an invocation of the StreamIt pre-defined `add FileReader <int>` statement to read values in from a binary (bin) file. The binary file, derived from a portable pixel map (ppm) file, simply contains a series of values for a 320 x 240 (width x height) image, and hence 320 x 240 x 3 values total for each image. Since the Kalyan code deals only with gray-scale images (the luminance of color images), we convert the RGB values to the YIQ color scheme, which contains one channel for luminance (Y), and two for chrominance (I and Q). Upon conversion, the pixels are then loaded into 2-D arrays of size 320 x 240, and pushed onto the input channel.

The user predefines the number of features to be extracted, the size of each feature window, and image height and width.

5.2 Feature extraction

The pipeline that contains the code corresponding to feature extraction is called `feature`, and contains within it a filter and another pipeline in series. This pipeline takes as input 3 matrices of size 320 x 240, and outputs both the image and a set of features whose size is user-defined.

```
float[320][240]->imageAndFeatures
pipeline feature( ... )
{
          add generateMinorEigs( ... );
          add chooseEigs( ... );
}

float[320][240]->imageAndFeatures pipeline chooseEigs( ... )
{
      portal<reMakeIAndF>bestFeatures;
      add splitjoin{
            split roundrobin(3,1);
                  add floatArrayIdentity();
                  add float[320][240]->float[320][240] pipeline {
                        add sortEigs( ... );
                        add findBest( ..., bestFeatures);
                  }
            join roundrobin(3,1);
      }
      add reMakeIAndF() to bestFeatures;
}
```

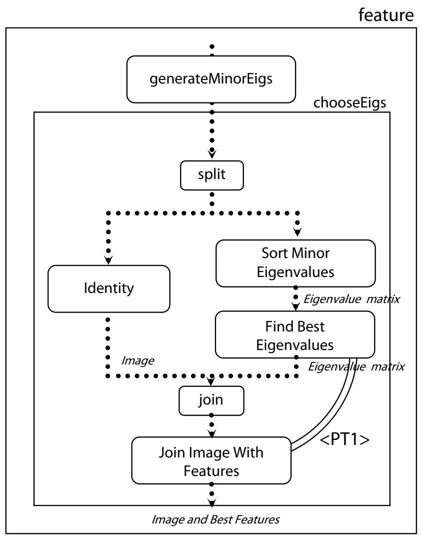


Figure 5-1: A high-level block diagram of the Feature Extraction stage

5.2.1 Generating Minor Eigenvalues

The filter `generateMinorEigs` takes the Y, I, and Q matrices from the input channel, and outputs those same matrices in addition to a matrix of equal size that contains all the minor eigenvalues. For each pixel, it computes the sum of the gradient values of the pixels within a square window outside that pixel. The size of this window can reasonably take a value between 3 and 19 or so. The gradient values computed are the gradient in the x direction squared (g_{xx}), the gradient in the y direction squared (g_{yy}), and the gradient in the x-y direction (g_{xy}). Once these values are summed across every pixel as per Equation 2.4, those gradient sums are arranged in a 2 x 2 matrix whose minor eigenvalue is calculated and stored.

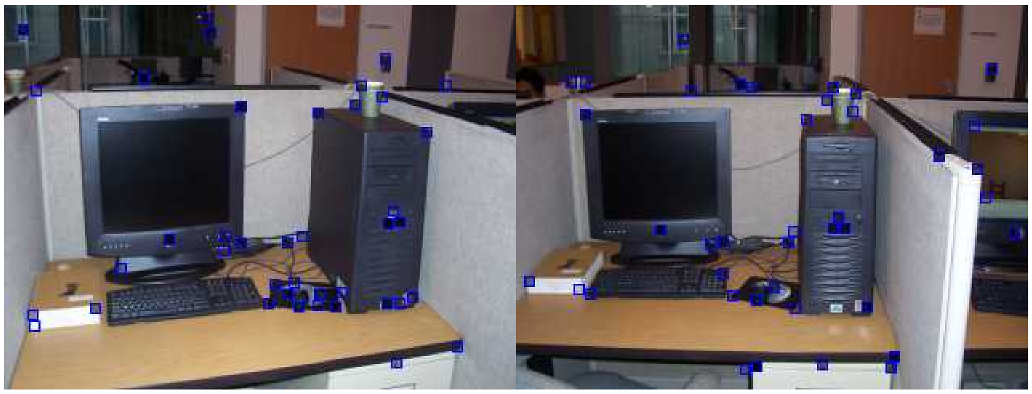
Executed as described, this would take on the order of 320 x 240 x w x w calculations, where w is the length of the window size. In order not to waste needlessly calculation time, we store the gradient values for the width of the image, w rows deep, and recalculate new image gradient sums by deleting the old values, and by replacing them with the new values.

5.2.2 Finding the Best Features

The next pipeline, taking the matrix of minor eigenvalues as the input, finds the highest valued minor eigenvalues in the matrix that are in non-overlapping windows. That is, the highest-valued pixels not within the windows of the higher-valued pixels before it are chosen. The number of features chosen is user-defined, and we found that 40 features or so are sufficient considering the processing needed in the correlation and RANSAC stages, though naturally this number varies based on the quality of the images.

This differs from the MATLAB code slightly. The Kalyan code uses a slightly different but comparable Harris Corner detection method to generate the minor eigenvalues, but once this matrix is generated, it simply chooses the values that lie beyond a certain threshold. While MATLAB can handle array sizes that change dynamically throughout the program, StreamIt must have its array sizes static, since space for these arrays are allocated at compile time. To tackle this problem, we assume a "virtual array", where an upper bound is set for an array size, but the array can virtually shrink so long as we keep a marker for the tail data item.

Once the matrix of minor eigenvalues is pushed to the next filter, they are ordered with a quick-sort and the top features in non-overlapping regions are selected. Those features, stored in a struct, are sent through a portal to a filter further downstream.



(a) Image 1 (b) Image 2

Figure 5-2: Two consecutive images with their 40 extracted features marked. The darker boxes are those with higher minor eigenvalues.

5.3 Sharpening and Correlation

The next major stage of the implementation involves the correlation of one image's feature set with another's. Before we do this, however, we essentially sharpen each image by subtracting a blurred form of the image from the image itself. We can blur an image by convolving the normal image with a blurring filter, more efficiently implemented with a 2-D Fast Fourier Transform (FFT).

5.3.1 2D Fast Fourier Transform

Before we perform the FFT on an image, we interleave its values with zeroes because the FFT filters work on complex numbers. We used the building block of an existing 1-D FFT StreamIt filter, which takes in on the input channel a series of floating point numbers (with alternate values equal to zero, and two parameters indicating the number of non-zero data items (size), and a zero-padding number that, when added to size, equals the next highest power of two. This 1-D FFT is computed with the Radix-2 algorithm, and hence works only on powers of two). The 2-D FFT runs a 1-D FFT on the rows of the image (where zeropadding also occurs), transposes those values, runs another 1-D FFT on the result, and finally transposes one more time to generate the final 2-D FFT output.

```
add splitjoin{
    split duplicate;
    add IAFID();
    add splitjoin{
       split duplicate;
            add Identity<float>;
            add float->float pipeline {
                  add splitjoin{
                        split roundrobin(1,0);
                        add FFT2( ... );
                        add pushAverager( ... );
                        join roundrobin(2);
                  };

                  add float->float filter multFFTs
                  {
                        float a, b, c, d;
                        work pop 4 push 2
                        {
                              a = pop(); b = pop();
                              c = pop(); d = pop();
                              push(a*c - b*d);
                              push(a*d + b*c);
                        }
                  };

                  add IFFT2( ... );
                  add takeInnerMatrix( ... );
                  }
      join roundrobin;
      };

      add float->float filter
      { work push 1 peek 2 pop 2
      {
            float temp = peek(0)-peek(1);
            push(temp);
            pop();
            pop();
      }
      };

      join roundrobin;
};
```

Figure 5-3: StreamIt code for sharpening an image

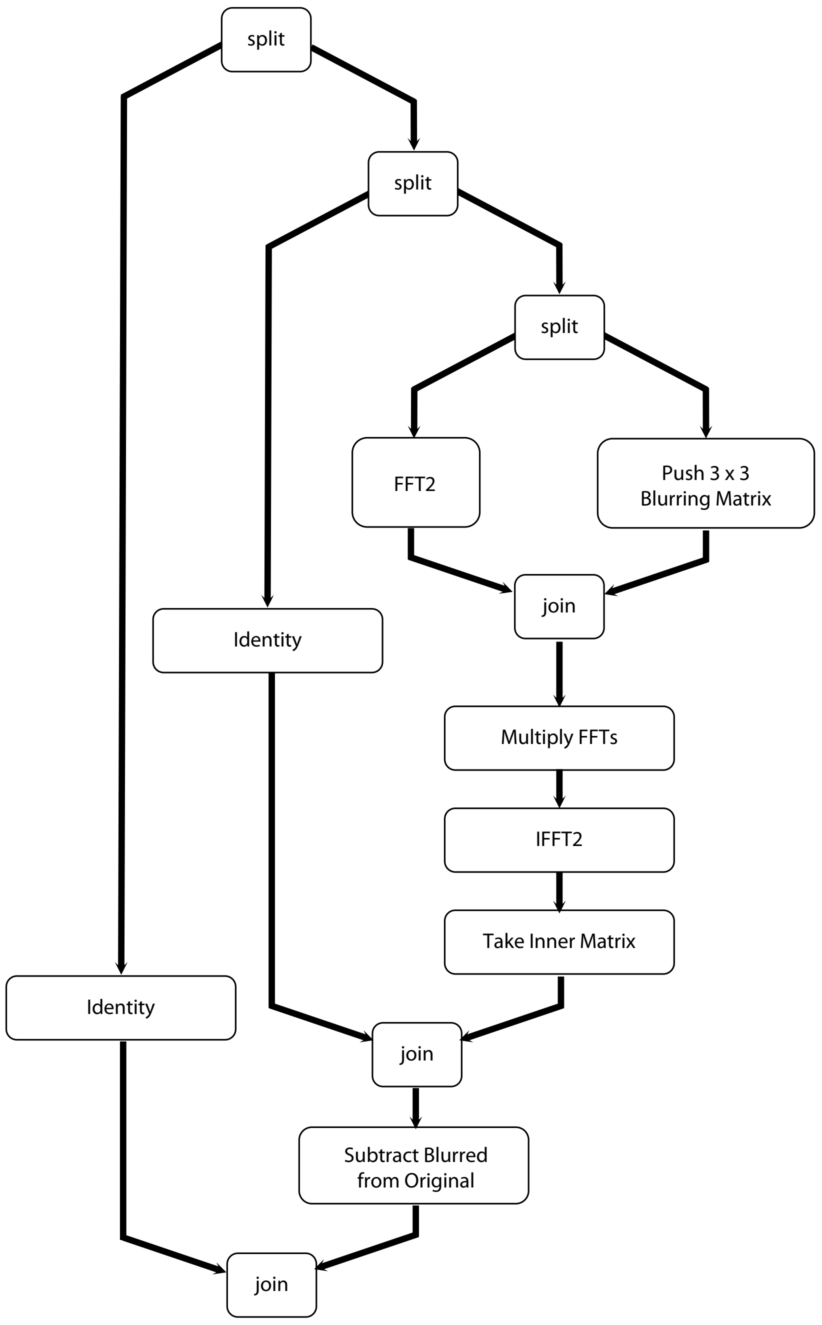
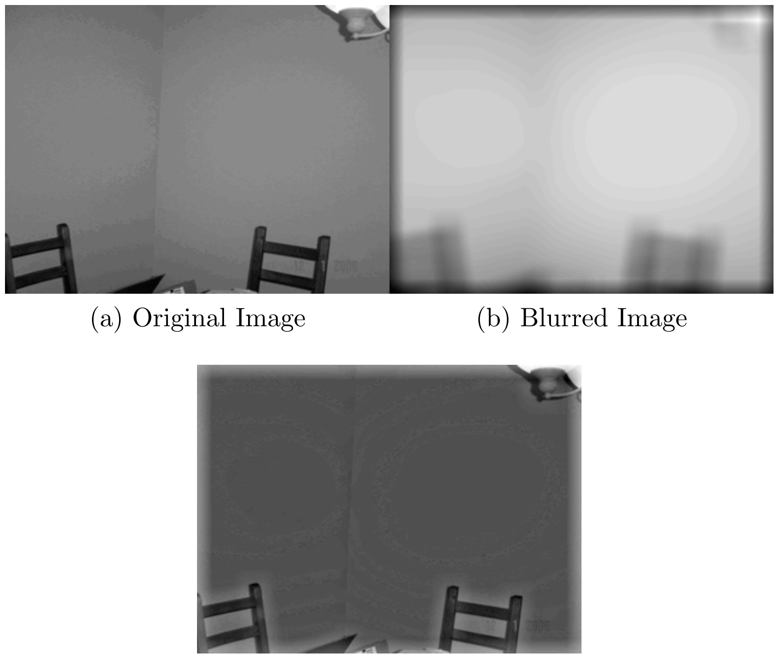


Figure 5-4: StreamIt graph for sharpening an image


(a) Original Image (b) Blurred Image

(c) Sharpened Image

Figure 5-5: The sharpened image is formed by subtracting the blurred image from the original

Once the 2-D FFTs of both the image and the blurring matrix are taken, we multiply one by the other, and we then take the 2-D IFFT (inverse FFT). Then, we take the inner part of the image with a size equal to the original (convolution adds the height and width of the blurring image to the original image). Now, we subtract the blurred image from the unblurred image, and the result is converted back into matrix form to be used by the correlation stage.

5.3.2 Finding Putative Matches

Once the image has been sharpened, and the feature sets of two successive images are ready, one can proceed to determine which features correlate best with which. As described by Equation 2.6, we do the following for each feature in the both images. We multiply every pixel in its window by every pixel in the window of another image's feature, and divide by the square root of the product of the standard deviations of each window. Each feature does this for every other feature in the other image. What results is one value for each of these iterations, such that if each image has, say, 50 features, all these values could be stored in a 50 x 50 matrix.

Having computed those values, we see which features “liked” each other most. That is, we find, for each feature r, which feature s from the other image it ranked highest. If s also ranked r as most highly correlated with it, then the two are a match that is kept. If s did not rank r as such, then s is a lone feature that is unmatched and discarded. All matched pairs are sent to the next stage.

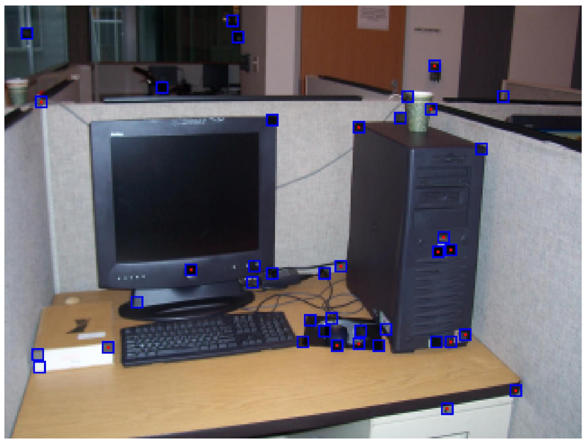
Since the image has been contorted through this process, in the sharpening step, we effectively save the image with a `splitjoin duplicate` before sharpening begins, and then after the join, we combine the old image with the new feature set, and discard the old image.

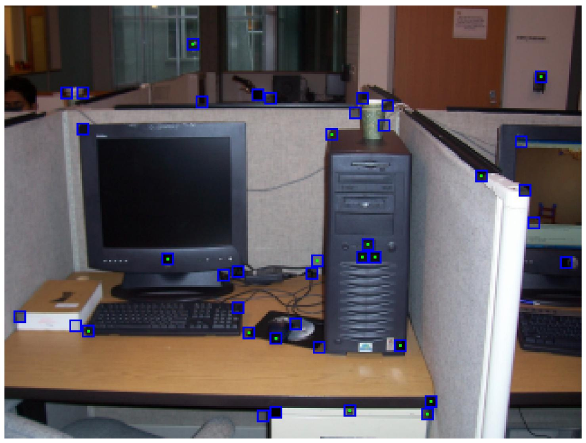
5.4 RANSAC

Given the putative matches from the correlation stage, the RANSAC stage finds matches that would generate a suitable homography matrix. The basic algorithm picks eight matches randomly, synthesizes a homography matrix (interchangeably referred to as a fundamental matrix) based on those eight matches, then finds how many of the putatively-matched features in one image are close to the other's after applying that matrix. If the sum of distances between the projected features in the second image and the actual features of the second image is small, RANSAC finishes, and we keep the most recent homography matrix.

One challenge RANSAC presents is its randomness. Because the outcome for each iteration of RANSAC is somewhat arbitrary, there are a random (though bounded) number of iterations it could run. This randomness necessitates the invocation of StreamIt's dynamic rates for filters.

Before we begin the main loop of the RANSAC stage, we normalize all the feature points by subtracting their mean and dividing by their standard deviation. This makes the matrix multiplication to come more manageable.


(a) Image 1


(b) Image 2

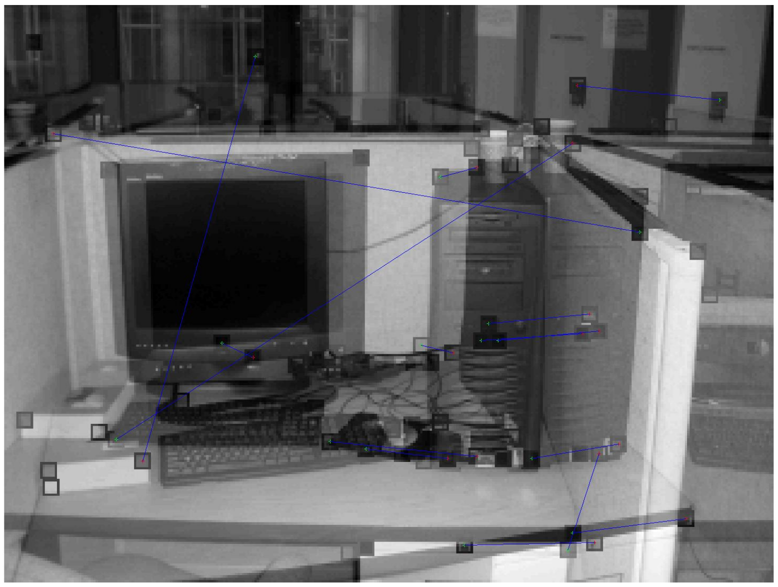

(c) Overlapped Images with Matches shown

Figure 5-6: Of the features marked by blue boxes, only those marked with dots in the middle are putative matches. The overlapped image indicates which putatively matched feature in one image matches with which feature from the other image.

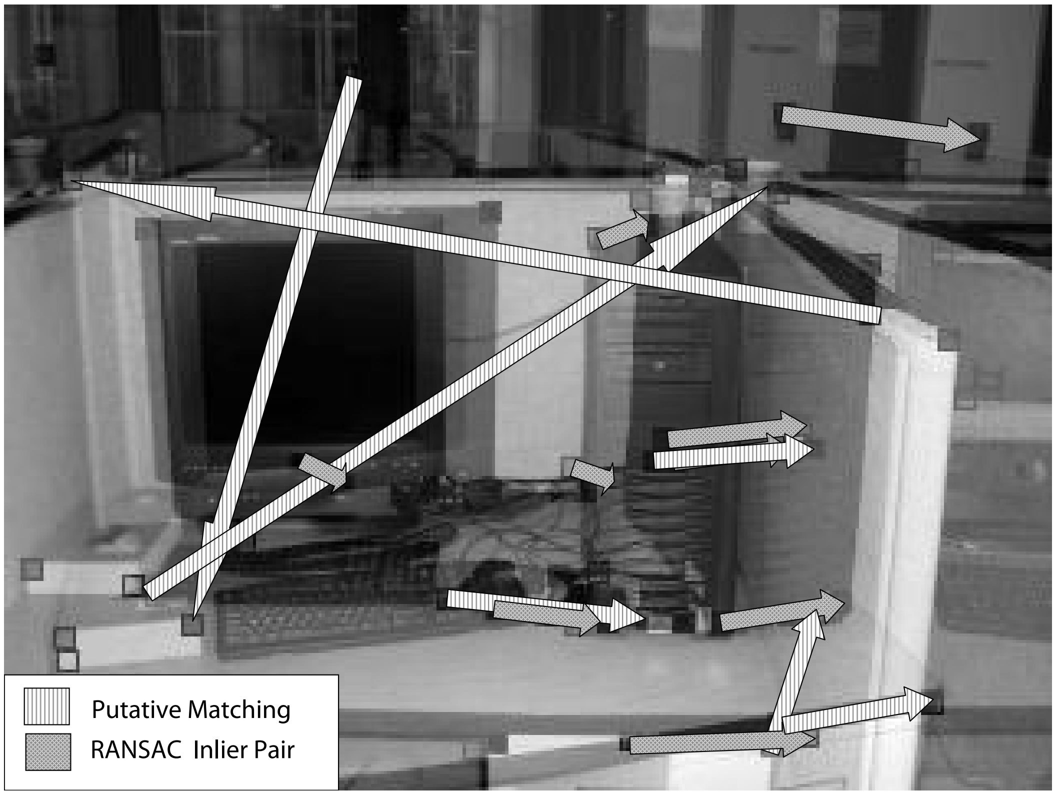


Figure 5-7: Two overlapped images, with 16 striped arrows showing the putative matches, and a set of 8 inliers with crosshatched arrows.

5.4.1 Main Feedback Loop

The RANSAC feedback loop must iterate until a suitable set of matches fit with a given homography matrix, i.e. a set of "inliers" must match. Because the exact number of iterations is unknown, we need to "gate off" any incoming data while the loop is running, and also prevent any data from being placed on the output channel. To do this, we simply place "gate" filters before and after the main RANSAC loop, and these filters push and pop one or zero elements based upon a message indicating RANSAC's completion. If RANSAC is still going, the input gate pops zero items and the output gate pushes zero items. If RANSAC has come to completion, the input gate pops one item and the output gate pushes one item.

Between these gates, however, is where RANSAC (outlined in Figure 2.3 on page

```
featureListPair->featureListPair pipeline ransac(){

      portal<selectIndices> feedbackOrInput;
      portal<gateOfEntry> pop1orNone;
      portal<gateOfExit> push1orNone;

      add normalizeFeaturePts();
      add gateOfEntry() to pop1orNone;

      add featureListPair->featureListPair feedbackloop{
            join roundrobin(1,1);
                  body pipeline {
                        add selectIndices() to feedbackOrInput;


                        add makeFundMatrix();


                        add findInliers(feedbackOrInput,
                                    pop1orNone, push1orNone);
                        }
                  loop  FLPID_backloop;


            split duplicate;
      }


      add gateOfExit() to push1orNone;
}
```

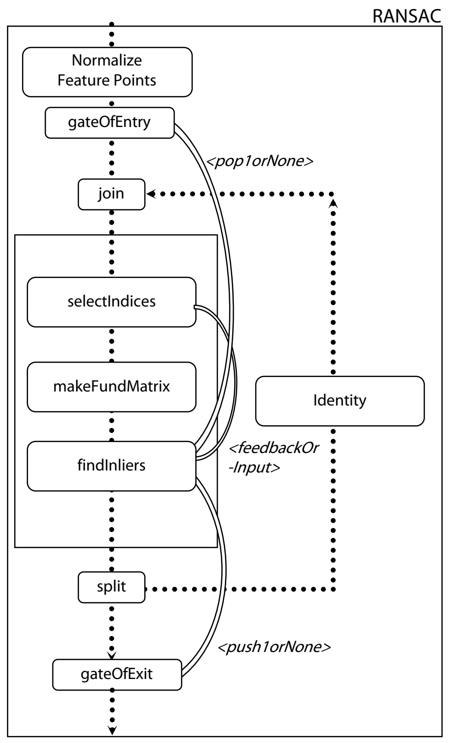


Figure 5-8: We use teleport messaging to indicate when the gates should "open" and when they should stay "closed".

21) is implemented. First, a random eight pairs of features of those putatively matched are chosen. Then, based on those eight pairs of features, we generate a fundamental matrix using singular value decomposition, in a process more detailed below. Lastly, we look to find, from among all the matched feature pairs, not just those randomly-chosen eight, but how well the matrix fits. That is, we apply the matrix to our first feature set, find where they project to on the second image, and find the distance between the projections and the second set of features. If enough of these pairs have a small enough distance between them, the matrix is chosen as the homography for that pair of images. All these operations occur in the body section of this feedback loop.

The `join` of the feedback loop takes the output of the process above on one hand, and the input from the entry gate. While closed, the entry gate continues to push the same datum it has been pushing, and the output of the body loop is used again at the start of the body loop, while the entry gate's outputted data item is discarded.

The above process represents an interesting means of implementing a `while` or `do while` loop that contains filters in StreamIt.

5.4.2 the Fundamental Matrix

Ideally, the fundamental matrix **F**, when multiplied by the first set of features in the first image should project them to the second set of features in the second image. Because the putative matches are not guaranteed to be matches of the same features, though, this fundamental matrix cannot be perfect. We use the eight features randomly selected in the filter prior to this one to generate the matrix as follows.

First, we normalize the eight points by subtracting their mean and dividing by their standard deviation. Then, we arrange their values in an 8 x 9 matrix to be evaluated by singular value decomposition. Once the 8 x 9 matrix has been decomposed into U, its left singular vectors, Σ, its singular value matrix, and V, its right singular vectors. Only the vector in V corresponding to the smallest singular value is kept, and it is reshaped into a 3 x 3 matrix, which then continues to another singular value decomposition. An output of this decomposition is a fundamental matrix.

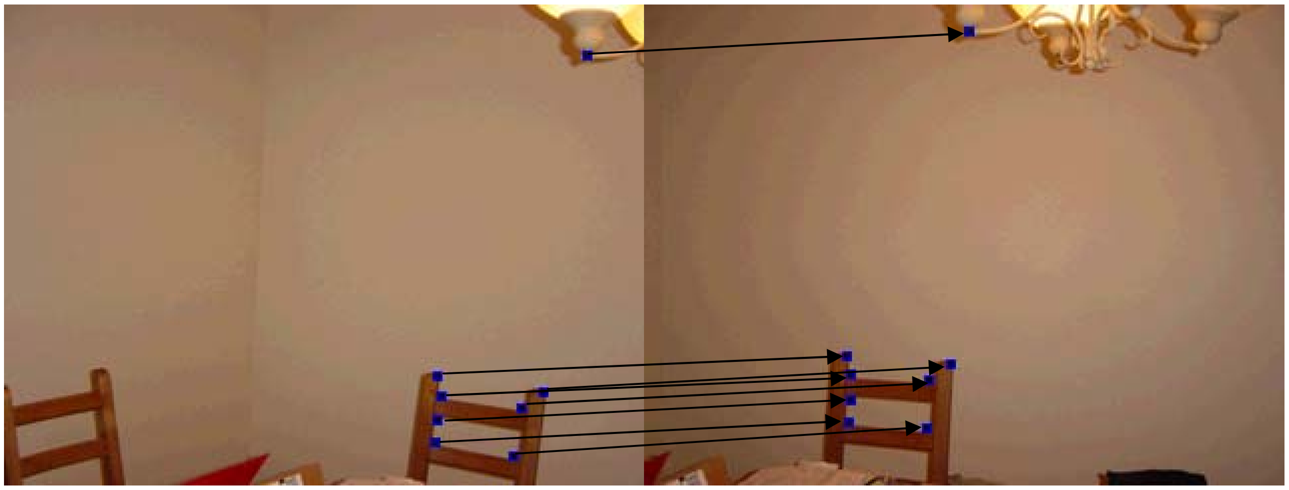
We then multiply the first image's set of features by this matrix, and see where they project on the second image relative to the existing second image set of features. We evaluate a first order approximation of the geometric error (Sampson Distance) of the fit of the fundamental matrix with respect to the set of matched points as RANSAC needs. We assume here that no calculations of the fundamental matrix result in a singular value decomposition degeneracy. A threshold of .001 for each match's distance determines whether or not a feature pair is deemed an "inlier". The RANSAC loop continues to run until the following equations hold:

$$N = \frac{log(1-0.99)}{log(1-f)} \tag{5.1}$$

$$N > trialcount \tag{5.2}$$

where *0.99* is the confidence there are no outliers, f is the fraction of samples putative matches that were inliers, and *trialcount* is the number of trials RANSAC has looped through.

Once RANSAC exits, there is a homographic matrix between the two images that can be stored or used immediately to find the rotation and translation between them.



(a) Images with Inliers marked

$$\begin{bmatrix} 0.00012 & 0.00005 & -0.03495 \\ -0.00006 & -0.00002 & 0.01694 \\ -0.02219 & -0.00933 & 6.65328 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} = \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix}$$

(b) $H\mathbf{x_1} = \mathbf{x_2}$

Figure 5-9: Two consecutive images with inliers marked by boxes. Note they match exactly between both images, and their homography matrix H is also included.

5.5 Preliminary Results

Results from preliminary runtime evaluations of StreamIt are encouraging. On a dual-processor Intel Xeon 2.20 GHz with 2 GB of RAM, our implementation ran 1.27x faster than the MATLAB reference implementation. On a different platform, a 3.6 GHz Pentium 4 with 3 GB of RAM, our implementation ran 1.45x faster. These speeds were attained after static-rate filters were fused, and after we unrolled loops by a factor of 2. This is quite impressive despite the lengthy quick-sort we used that the reference implementation did not, and an Fast Fourier Transform based on a radix-2 algorithm that is not as efficient as MATLAB's implementation of it. Although our implementation is running 2% faster than the reference implementation, this is sure to speed up more as one exploits more parallelism and maps the program to 4 or even 16-processor machines.

Chapter 6

StreamIt Programmability

6.1 StreamIt v. MATLAB

StreamIt is designed partly to obviate the low-level buffer management developers in streaming programming need to deal with. Although the previously designed MPEG-2 Encoder [9] in StreamIt was based on an existing C implementation, our Motion Estimation algorithm is based upon an existing MATLAB implementation. The MPEG-2 Encoder C implementation contained buffer management details that Drake was able to trim when converting to StreamIt. Resultantly, his code was shorter and preserved the block structure of the underlying algorithm quite well. In our case, we start with MATLAB code that has distinct advantages which do not lend themselves well to the stream programming model. Although StreamIt enhances performance, there are programming challenges presented by MATLAB code that this section discusses in detail. We also include proposed extensions to StreamIt that could enable some more productive and efficient coding from the developer's standpoint.

6.2 MATLAB

MATLAB lends itself well to applications in robotics and computer vision, and as its name (derived from MATrix LABoratory) suggests, cleanly manipulates vectors and matrices with a toolbox and libraries that support a vast array of operations and algorithms. There are two main advantages that the MATLAB implementation had over our implementation of Motion Estimation. The first pertains to dynamic typing, and the second deals with array manipulation.

MATLAB dynamically types its variables. Much like in Python, Ruby, ColdFusion, or other "duck typing"[1] languages, MATLAB does not require the user to type all its variables, and this stands in stark contrast to StreamIt where, as in C++, variables, filters, and pipelines are all typed. Although this does not present by any means an unreasonable amount of work for the programmer translating from MATLAB to StreamIt, it does present a challenge especially when declaring arrays whose sizes vary throughout the program. This is not to imply that a language is worse off with type information. To the contrary, the type information does ease the task of a programmer trying to understand what the program is actually doing. There is then a trade-off here between program design and program readability, and although MATLAB may be stronger in the former case, StreamIt is stronger in the latter.

Array manipulation more generally is perhaps the prime advantage MATLAB has over StreamIt when it comes to programmability. MATLAB's convenient array referencing provides for a quick-and-easy manipulation of arrays or parts of arrays. One can reference a sub-array or sub-matrix with one line of code in MATLAB, but in StreamIt such a reference requires a cumbersome for loop or nested for loops. Furthermore, matrix multiplication, transposition or inversion can be performed in one fell swoop, whereas in StreamIt, such operations could require the invocation of filters and splitjoins, where the filters need push and pop rates defined, and the splitjoins may require the input data to be of a particular form upon entry. But such shortcomings could be solved in StreamIt by simple syntactic transformations and a bigger StreamIt library.

The real challenge that StreamIt 2.1 poses, when it comes to programmability, is variable-sized arrays. In MATLAB, arrays of one size can be expanded or trimmed to another size without employing any new variable declarations. The current im-

[1]Duck typing is a principle of dynamic typing in which an object's interface and attributes determine valid semantics, rather than having the language enforce rules regarding inheritance or require type casting

plementation of StreamIt allocates memory at compile time, and hence necessitates a fixing of array sizes and does not allow for such expansions or reductions of the size. Although the SDF abstraction necessitates such a constraint, it still presents a challenge for converting from MATLAB. Take, for instance, the case where function parameters are of variable size in MATLAB. Often, the most natural expression for a MATLAB function is as a StreamIt filter. However, if that filter takes a variable number of data on the input channel, the programmer must reassess how that function ought to fit into the stream graph-either as a pipeline, within a filter, or as a helper function of a filter.

The common solution for the variable/fixed-sized array problem we found with the Motion Estimation implementation was to simply set an upper bound for the size of the array, and to declare the array with that upper bound. We had to keep another variable as a reference to the index of its tail element, but we were able to tackle this problem with this method. In the future, it would be straightforward to extend StreamIt to support dynamic memory allocation within a filter. Dynamically-allocated data could also be passed between filters by using a dynamic communication rate.

StreamIt's strength in programmability most notably manifests itself in the preservation of block structure. In MATLAB, data is stored in global buffers, and so the code hides the actual movement of data. It is easier to reason about independent modules rather than the function-based computation of traditional languages like MATLAB, and so StreamIt does make it easier to comprehend this block structure. Figure 6-1 demonstrates how StreamIt exposes the underlying structure, in this case more intuitively than MATLAB does.

6.3 Proposed Extensions

When programming in StreamIt, a common difficulty the programmer experiences pertains to the constraint whereby each channel can only carry one type of data. In our case, we were pushing images from one filter to another, but would also have

```
featureListPair->featureListPair
          pipeline makeFundMatrix()
{
    add normalizeIndices(s);
    add splitjoin{
          split duplicate;
          add FLPID();
          add pipeline{
               add makeConstraintMatrix(s);
               add svd1(s,9);//(s, 9);
               add reshapeVCol(s);
               add TransposeM(3,3);
               add svd2();//(3,3);
               add newF();
          }
      join roundrobin(1,1);
    }
    ...
    add denormalizeF();
}
```

```
function F = fundmatrix(varargin)
    ...
    [x1, T1] = normalise2dpts(x1);
    [x2, T2] = normalise2dpts(x2);

    % Make the constraint matrix
    A = [x2(1,:)'.*x1(1,:)'   x2(1,:)'.*x1(2,:)'  x2(1,:)' ...
         x2(2,:)'.*x1(1,:)'   x2(2,:)'.*x1(2,:)'  x2(2,:)' ...
         x1(1,:)'             x1(2,:)'            ones(npts,1) ];
    [U,D,V] = svd(A);
    % Extract fundamental matrix from the column of V
    % corresponding to smallest singular value.
    F = reshape(V(:,9),3,3);
    F = F';
    [U,D,V] = svd(F);
    F = U*diag([D(1,1) D(2,2) 0])*V';

    % Denormalise
    F = T2'*F*T1;
```

Figure 6-1: StreamIt v. MATLAB - StreamIt expresses computation in a more modular, hierarchical way with pipelines and filters.

metadata, like lists of features, which we would wish to send downstream to other filters. There were two options with respect to how to send that information. On the one hand, we could form a new data structure that we could push onto the output channel that would include both the image and the metadata. On the other hand, we could use teleport messaging to deliver that metadata to the intended receiver. Given that the former case was generally easier to program, we found ourselves pushing out data structures with a lot of metadata from filter to filter. Often, we need a filter to manipulate only some of the metadata and not all elements of these complex data structures, but end up pushing the entire data structure onto a pipeline that doesn't need all that information. Although this isn't problematic in terms of performance, when debugging with the Java library, pushing and popping large data structures can cause the program to take much longer to execute. A proposed extension for StreamIt is then to accommodate such a feature, where buffers between filters can take union data types. If the expected data at the input are all of one data type, the work function can proceed as normal. If the expected data at the input are of different types, then a switch statement at the beginning of the work statement could determine which it is, and then deal with it as expected.

Chapter 7

Conclusions

The fields of robotics and machine vision depend highly upon the speedy implementation of algorithms in those fields. Multicore processor architectures offer a unique opportunity to translate existing programs designed for uniprocessor systems into programs that can exploit the added resources these extra cores have to offer. StreamIt is a programming language that is well-poised to tackle the challenge of bridging the gap between new hardware capabilities and cutting-edge applications. This thesis offered a case study of the StreamIt programming language and its application to a Machine Vision algorithm. In particular, it discussed the implementation of an image-based motion estimation algorithm, and the process by which an existing MATLAB implementation was translated into the StreamIt language.

The Motion Estimation algorithm was written, unlike the MPEG-2 codec, with more lines of code than the reference implementation. However, MATLAB did invoke many library functions (svd, convolution, matrix multiplications, etc.) that we coded from scratch. Hence, as more benchmarks are added to the StreamIt library, the programmer will not need write nearly as much. Furthermore, the StreamIt code expressed the underlying block structure far better than the cluttered MATLAB code, and although data typing can become cumbersome for a programmer relative to the duck typing of MATLAB, a programmer reading the StreamIt code has a firmer understanding when becoming acquainted with the code. This presents a tradeoff that depends largely upon whose position one is in: the writer's, or the reader's.

Towards the end of confirming these findings, we implemented this motion estimation algorithm which used the vast majority of StreamIt's capabilities: feedback loops, messaging, and dynamic rates. The performance of this implementation, on a single processor architecture, was comparable to that of MATLAB.

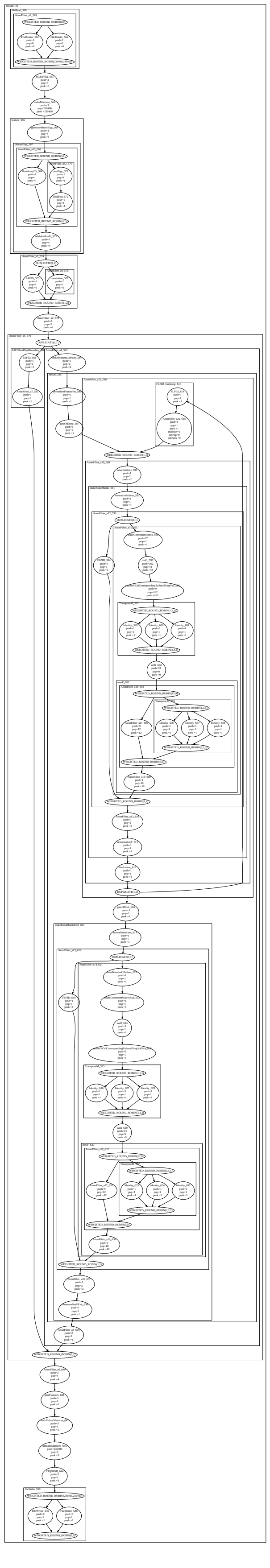
7.1 Future work

The existing Motion Estimation algorithm has great potential for faster performance. Executed on a uniprocessor architecture, the algorithm took an amount of time comparable to the MATLAB execution on a similar machine. Taking advantage of more parallelism, one could improve the performance tremendously. Luckily, the algorithm operates on independent pairs of consecutive images and hence can find homographies for these pairs of images in parallel, with the number of parallel branches depending upon the available number of processors. Beyond this, filters like the 2-D FFT can also be parallelized as well (we take 1-D FFTs of rows and columns, and these can be taken independently of one another). Beyond this, there exists the RANSAC loop which, if stalling, can be channeled to several processors.

Ultimately, the benefits of motion estimation can be leveraged for more involved applications. Mosaicing several images in realtime, realtime simultaneous localization and mapping (SLAM), and other robotics applications could make great use of StreamIt's power.

Appendix A

The Stream Graph for the StreamIt Motion Estimation Algorithm



Bibliography

[1] *International Conference on Compiler Construction*, Grenoble, France, April 2002.

[2] Shail Aditya, Arvind, Lennart Augustsson, Jan-Willem Maessen, and Rishiyur S. Nikhil. Semantics of pH: A Parallel Dialect of Haskell. In Paul Hudak, editor, *Proc. Haskell Workshop, La Jolla, CA USA*, pages 35–49, June 1995.

[3] E. Ashcroft and W. Wadge. Lucid. *a non procedural language with iteration. C. ACM*, 20(7), 1977.

[4] Kalyan B., Balasuriya A., Kondo H., Mak T., and Ura T. motion estimation and mapping by autonomous underwater vehicles in sea environments. *Proc. of IEEE Oceans*, 05(1), 2005.

[5] Gerard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[6] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Mike, and H. Pat. Brook for gpus: Stream computing on graphics hardware, 2004.

[7] P. Caspi and M. Pouzet. *Lucid Synchrone distribution.*

[8] C. Consel, H. Hamdi, L. Rveillre, L. Singaravelu, H. Yu, and C. Pu. Spidle: A dsl approach to specifying streaming applications, 2002.

[9] Matthew Drake. Stream programming for image and video compression. M.eng. thesis, Massachusetts Institute of Technology, Cambridge, MA, May 2006.

[10] Dirk Farin and Peter H.N. de With. Evaluation of a feature-based global-motion estimation system.

[11] T. Gautier, P. L. Guernic, and L. Besnard. Signal: A declarative language for synchronous programming of real-time systems. *Springer Verlag LNCS*, 274, 1987.

[12] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[13] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521623049, 2000.

[14] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable stream processors. *IEEE Computer*, pages 54–62.

[15] E. Lee and D. Messershmitt. *Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing*. IEEE Trans, 1987.

[16] W. Mark, S. Glanville, and K. Akeley. Cg: A system for programming graphics hardware in a c-like language, 2003.

[17] P. Murthy and E. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, 50(7):2064+, July 2002.

[18] Umakishore Ramachandran, Rishiyur S. Nikhil, James M. Rehg, Yavor Angelov, Arnab Paul, Sameer Adhikari, Kenneth M. Mackenzie, Nissim Harel, and Kathleen Knobe. Stampede: A cluster programming middleware for interactive stream-oriented applications. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1140–1154, 2003.

[19] Janis Sermulins. Cache optimizations for stream programs. M.eng. thesis, Massachusetts Institute of Technology, Cambridge, MA, May 2005.

[20] Dirk Stichling and Bernd Kleinjohann. CV-SDF - a model for real-time computer vision applications. In *WACV 2002: IEEE Workshop on Applications of Computer Vision*, Orlando, FL, USA, December 2002.

[21] William Thies, Michal Karczmarek, Janis Sermulins, Rodric Rabbah, and Saman Amarasinghe. Teleport messaging for distributed stream programs.

[22] Carlo Tomasi and Takeo Kanade. Detection and tracking of point features. Technical Report CMU-CS-91-132, Carnegie Mellon University, April 1991.