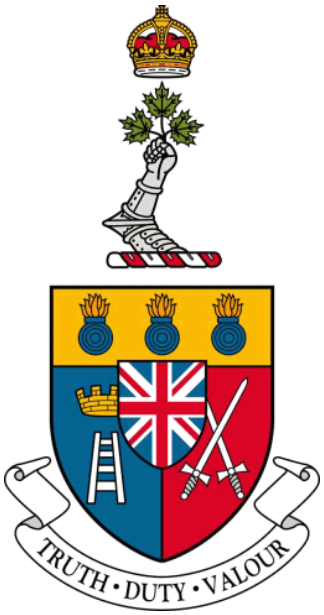
ROYAL MILITARY COLLEGE OF CANADA

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
EEE457 - COMPUTER ENGINEERING DESIGN PROJECT




# DID-07 – Detailed Design Document
# Parallelization of the Pixel Shiftmap Estimation Algorithm on a GPU

**Presented by:**
OCdt Jacob Wiebe
OCdt James Dolman

**Presented to:**
Capt. Adrien Lapointe
Dr. Don McGaughey

March 23, 2018

# TABLE OF CONTENTS

# TABLE OF FIGURES

# TABLE OF TABLES

# TABLE OF ABBREVIATIONS

| | |
|---|---|
| GPU | Graphics Processing Unit |
| ALPS | Algorithm for Parallel Shiftmapping |
| PSM | Pixel Shiftmap |
| CUDA | Compute Unified Device Architecture |
| SOR | Successive Over-Relaxation |
| OpenCV | Open Source Computer Vision Library |
| API | Application Programming Interface |

# 1 INTRODUCTION

This document describes the detailed models, design method, results, and issues forming the detailed design specifications of the project: Parallelization of the Pixel Shiftmap Estimation Algorithm on a graphical processing unit (GPU). This project, referred to as ALPS (Algorithm for Pixel Shiftmapping), implements the Brox optical flow estimation algorithm [1] using C++ and CUDA to interface with a GPU. ALPS employs parallel functionality to produce a pixel shiftmap faster than a current MATLAB implementation by Visesh Chari [2], within an acceptable level of error (see [3] for requirements). The project was decomposed into two phases: the sequential phase, in which a hierarchical design methodology was used, and the parallel phase, in which a incremental waterfall methodology was used.

The remainder of this document is decomposed into the following sections. Section 1 will identify the purpose of this document as well as explain this project and the Brox algorithm. Section 2 will discuss the detailed design of both serial and parallel phases. Section 3 will describe each module of the project. Section 4 will explain the design methodology in detail and the procedure taken. Section 5 will detail the implementation of the project. Section 6 will outline the testing methodology for validation of the results. Section 7 will present the results, including the experimental setup and performance of the program. Section 8 will analyze the requirements of the project in relation to its final results. Finally, section 9 will conclude the document.

## 1.1 Document Purpose

The purpose of the Detailed Design Document is broken into the 5 following subtopics:

a. to comprehensively explain the project's design including the architecture and process of execution;

b. to provide insight into the design methodology, design decisions, and challenges;

c. to present the results of the product in relation to its original requirements; and

d. to discuss the deviations from the originally planned project in terms of the design documents [3], [4], and [5], as well as justification for such deviations;

e. to comment on the success of the project and provide suggestions for future work.

## 1.2 Background

Images taken through the atmosphere at long ranges (greater than one kilometer) are warped and distorted due to random fluctuations in the index of refraction caused by atmospheric turbulence. The random warping effect on subsequent images presents, visually, an apparent motion within the image despite a stationary subject and fixed aperture. This can be corrected, or "dewarped" by mapping the displacement of pixels across multiple frames and shifting based on the result to generate one or multiple clarified images. See [3] for additional information regarding atmospheric distortion and a broader scope of image correction software.

The most computationally intensive process within the dewarping procedure is the estimation of a pixel shiftmap (PSM). To generalize, a PSM is a field of two-dimensional pixel displacement vectors that map the displacement of pixels from an image to their location in the following second image. The Brox algorithm [1] provides the basis for estimating a PSM that was used in this project.

To improve performance, the project program, Algorithm for Pixel Shiftmapping (ALPS), leverages the massively parallel capabilities of the GPU. Due to the goal of maximizing the performance boost derived from parallelism, the functionality to be parallelized was the area that was most computationally expensive. Through utilization of the MATLAB *profile* function on Chari's implementation, it was determined that the iterative solver module within the MATLAB code had the greatest CPU use. As such, this iterative solver was the focus of the parallelization phase of the project.

### 1.3 Brox Algorithm

A theoretical explanation of the Brox algorithm characteristics can be found in [4, §1.2.1]. The goal of the algorithm is to estimate, with minimal error, the optical flow of pixels between two images. This is represented as a PSM. The algorithm dictates the use of an iterative refinement of the PSM, modeled by the Gaussian Pyramid. To generalize Brox's computation of an estimated PSM between two images, it can be distilled into the following steps:

a. Conversion of images into gray value matrices

b. Gaussian pyramid decomposition of images into coarse matrices, further refining with each iteration

c. Calculation of energy values to be used to build a system of linear equations

d. Solving the resulting system of linear equations using an iterative solver method

e. Building the estimated PSM and returning to step b. for the next iteration (as applicable)

To improve accuracy and efficiency, the input images are first sampled at low resolution, the optical flow is calculated for each element, and applied as a starting point for the next higher resolution iteration. The optical flow field is estimated by solving a system of linear equations created by the generation an "energy function". This function represents the movement of pixels with an applied smoothing factor. Represented as the vector equation $A\vec{x} + \vec{b}$, optical flow field is solved for using an iterative method for solving a system of linear equations. The values solved for in $\vec{x}$ represent the displacement of pixels for a given level of the Gaussian pyramid.

There are many parameters which are required for execution such as the number of iterations in the pyramid, smoothing factors, etc. These values were optimized via trial and error; their affect on resultant accuracy depends greatly on characteristics of the input images. Parameters of particular importance are the number of levels of the Gaussian Pyramid, the maximum number of iterations for the solver module, and $\alpha$ and $\gamma$, the global smoothness factor and derivative weight, respectively. Increasing levels of the Gaussian Pyramid may improve accuracy due to more refinements of the image, at the cost of speed. The maximum number of iterations caps the

acceptable number of estimates above the error threshold before the system is deemed divergent. The $\alpha$ and $\gamma$ parameters are critical to accurately calculating the smooth energy field.

# 2 DESIGN

## 2.1 Overall Design

ALPS reads two images of equal dimensions, converts them into matrix representations and uses this data to calculate the estimated pixel shiftmap (PSM) between them. As illustrated in Fig. 1, the flow of the program is centered around the *optical flow* module. This is the heart of the program and its function can be described as bringing all the necessary components together to form a system of linear equations as presented in [1]. At each refinement of the Gaussian pyramid, the previous images are scaled to a higher resolution and the *optical flow* module is called. From *optical flow*, the math based modules such as *energy calculation* and *gaussian smooth* are used to determine the energy field and apply smoothing to it, respectively. Then, *matrix builder* produces a matrix in which each row can be solved linearly in the form of $A\vec{x} = \vec{b}$. In Chari's implementation and the sequential ALPS implementation, the Successive Over-relaxation (SOR) method is used to solve for the $\vec{x}$ vector. However, an iterative Jacobi algorithm was designed and implemented for the parallel solver module, replacing SOR in the second major version of ALPS. Due to the nature of the Jacobi solver code, it can be parallelized to solve all linear equations concurrently; see Section 5.1 for details on this implementation.

The architecture of each version of ALPS consists of eight modules. Seven headers which perform various mathematical operations and one main driver. As in Fig. 1, the program is to iterate through levels of the Gaussian pyramid with each level performing smoothing, construction of a system of equations and solving of this system using the successive over-relaxation or, depending on the version, the Jacobi method. At the completion of each level, the image resolution is refined and the cycle begins again. A detailed module diagram is included in Annex A and description of selected symbols used in this diagram are included in Annex B.

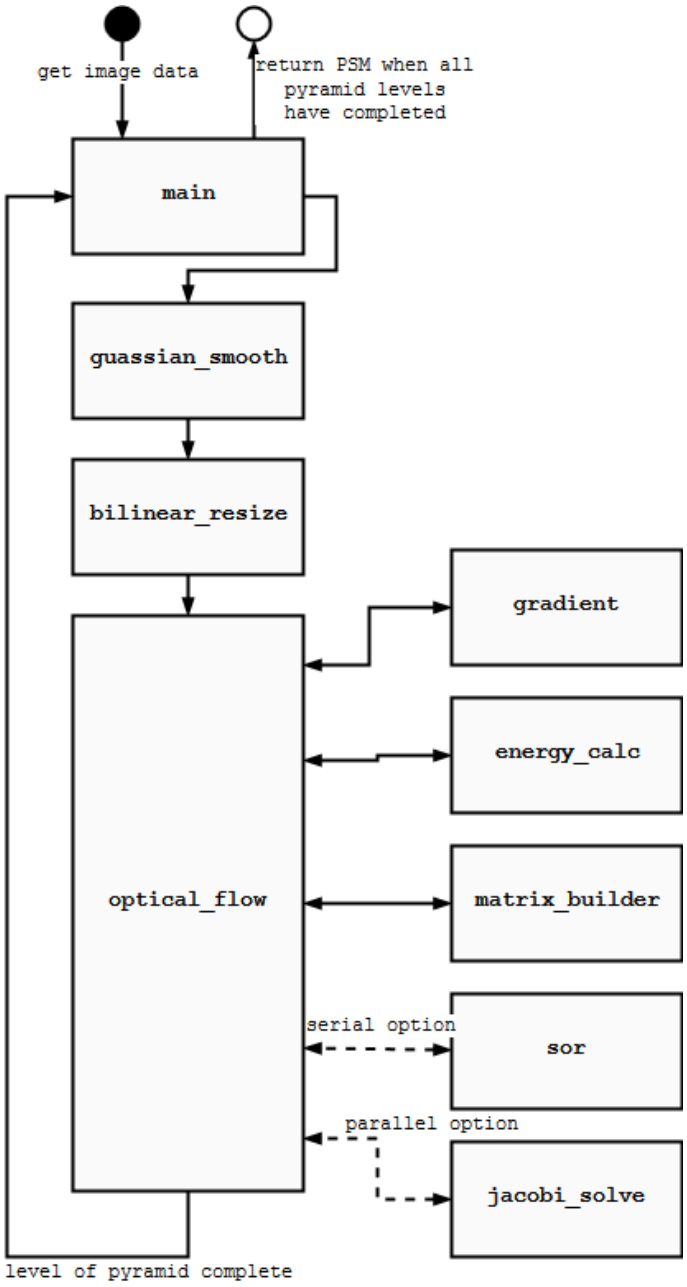



Fig. 1 - Architecture of ALPS

## 2.2 Parallel Design

The most intensive phase of this project was the parallelization. The goal, as defined by [3], was to run part of the program in parallel on a GPU with the intent of reducing computation time. Functionality was parallelized based on two criteria: sequential computation time and data independence. Functions and modules are to be prioritized based on their computation time. Code sections with longer times to completion provide a greater performance return if executed instead in parallel.

Areas of significant computation time in Chari's code were identified through a preliminary analysis with the MATLAB function *profile*. Upon completion of sequential ALPS, another profile test using Visual Studio's profiling tool was performed. The results showed that the SOR module in both implementations had the greatest completion times. As a result, the iterative solver module of was the focus of parallelization efforts. The SOR method was implemented for sequential ALPS, however, for the parallel versions, the Jacobi iterative method was chosen due to its suitability as a parallel algorithm.

One major strength of the Jacobi method is that computations for iterative estimates are independent for each equation in the system. The order in which theses equations are solved is therefore irrelevant from the perspective of the solution. This means that threads of execution in a parallel environment would only need to synchronize at the completion of each estimation. Equation (1) presents the algorithm used for the implementation of the Jacobi method, as provided in [6].

$$x_i^{(k)} = (b_i - \sum_{j \neq i} a_{i,j} x_j^{(k-1)})/a_{i,i} \tag{2.1}$$

This form of parallelization is not possible using other similar iterative solvers such as the Gauss-Seidel Method. This is because within the Gauss-Seidel method, the solutions of previous equations are necessary to calculate that of the current equation. In addition, the resulting solution will vary depending on the order in which the equations were solved [6]. This does not lend itself well to parallelism and was therefore ruled out as the method of choice. Since the SOR method is an extension of Gauss-Seidel, it also suffers the same drawback with regards to parallelism.

# 3 SOFTWARE MODULE DESCRIPTIONS

## 3.1 Main Module

The main module is the central procedure within the program. It initializes all constants for the program for easy modification of the algorithm parameters such as smoothness factor, levels of the Gaussian pyramid to be completed, and maximum number of SOR iterations to be performed before failure. It's second major function is to read image data from a file into a matrix of corresponding pixel brightness values. This is performed for both images in which the estimated PSM is to be determined from. The main module will then call a Gaussian smoothing function (see Section 3.2) on both image matrices before beginning the first iteration of the Gaussian Pyramid.

For each level of the pyramid, a series of derivatives are determined using the gradient module (see Section 3.4) and then the optical flow procedure is called. The optical flow module, outlined in Section 3.3, performs the steps outlined in [1] for solving for the change in displacement of pixels between the two images ($du$ in the $\vec{x}$ domain and $\vec{dv}$ in the $y$ domain) at a given resolution (level of the pyramid). With this data,the main module adds the change onto the existing displacements in each direction ($\vec{u}$ and $\vec{v}$) and re-samples the initial image matrices to the next

resolution. Finally, within a given resolution the intermediate shift maps ($\vec{u}$ and $\vec{v}$) are resized bilinearly (see Section 3.8). Once the algorithm has iterated through all levels of the pyramid (with each iteration refining the $\vec{u}$ and $\vec{v}$ PSMs) $\vec{u}$ and $\vec{v}$ are outputted completing the program's execution.

### 3.2 Gaussian Smoothing Module

The Gaussian smoothing module is called at each level of the Gaussian pyramid to "smooth" the image data. The result is less abrupt edges between pixels, ultimately preserving image patterns and producing a more accurate shift. A grid vector is applied to the provided image two of convolutions; a row-wise followed by a column-wise operation. The result of these two convolutions is a smoothed matrix representation of the image for the given scale factor. The smoothing estimates what the image would represent if it were down sampled to a lower resolution, in signal processing this is called decimation.

### 3.3 Optical Flow Module

The optical flow module can be thought of as the controller for the process of finding the optical flow field of a given level of the Gaussian pyramid. It produces the derivatives of the $\vec{u}$ and $\vec{v}$ vectors (i.e. the rate of change of the displacement of pixels in the $x$ and $y$ domain, respectively). This is done in the following steps. First a series of matrix derivatives are generated using the *gradient* function. With these variables, the energy model for the two images can be calculated as per the equations given in [1]. This module then calls the energy calculation module to generate the smooth energy at this level. All of this data contributes to the creation of the sparse matrix $A$ by the *matrix builder* function. Using $A$ in the equation $x = A^{-1} + b$ solves for $\vec{x}$ by calling the *SOR* or *Jacobi* module, depending on the version of ALPS being run. $\vec{x}$ holds the $\vec{du}$ and $\vec{dv}$ vectors; the solution to the system of linear equations at a given level of the pyramid. The sequence of calls performed by the optical flow module is presented visually in Fig. 1.

### 3.4 Gradient Module

Since Armadillo does not contain built-in capability to perform matrix gradient operations, a module to perform them was created for this project utilizing 2-dimensional convolutions in the $x$ and $y$ domain. This produces two Jacobian matrices, one with respect to $x$ and the other to $y$. This is used by the optical flow module to generate partial derivatives of matrices required by the Brox algorithm.

### 3.5 Energy Calculation Module

The energy calculation module is responsible for generating the "energy" of the pixel displacement field. This is an abstract field of values which serves to represent the "force" that would be needed to perform a shift of pixels given a series of weighting values. The goal, as

presented in [1], is to minimize this energy in order to produce the most accurate shift. A variational module is used to produce an energy based formulation. This is modeled by a linearized optical flow constraint coupled with an increasing concave function, $\Psi\ (s^2)$. The total energy can be expressed as:

$$E(u, v) = E_{Data} + \alpha E_{Smooth} \tag{3.1}$$

where $\alpha$ is a variable parameter. Data energy and smooth energy are expressed as:

$$E_{Data} = \int_{\Omega} \Psi(|I(x + w) - I(x)|^2 + \gamma|\nabla I(x + w) - \nabla I(x)|^2)dx \tag{3.2}$$

and

$$E_{Smooth} = \int_{\Omega} \Psi(|\nabla u|^2 + |\nabla v|^2)dx \tag{3.3}$$

respectively [1]. Equation (3.2) presents the first term as the quadratic deviations in pixel displacement and the second term acts as a quadratic regularizer to constraint the optical flow field to be continuous. The $\gamma$ parameter allows for the scaling of this constraint, effectively weighting the importance of smoothness [7]. Equation (3.3) is the model used for describing the spatial smoothness. In both energy equations, the $\Psi$ function is applied where $\Psi(s^2) = \sqrt{s^2 + \epsilon^2}$ ($\epsilon$ is $10^{-4}$ for the purposes of the algorithm). The calculation of the *data energy* is performed using a series of Jacobian matrices generated by *optical flow*. The minimum of the energy model can be found by solving the associated Euler-Lagrange equation. To simplify this model, the non-linear equations are approximated using a first order Taylor expansion [8].

### 3.6 Matrix Builder Module

The *matrix builder* is used to create the coefficient matrix $A$ and right hand side vector $\vec{b}$ for use in the equation $A\vec{x} = \vec{b}$. Each row within $A$ represents a linear equation. This matrix is built using the energy minimization model described in [1]. This procedure takes derivatives of the gray-values of the image at a particular Gaussian level, along with the last prediction of the optical flow (i.e. last $\vec{u}$ and $\vec{v}$) to build its coefficient matrix. The solution to this system of linear equations is the spacial gradient of the image (optical flow) that is minimized in terms of the "energy" field. $A$ is built with a diagonal of non-zero values which is significant because solution of each element in the diagonal is estimated later in the program based on its row-wise linear equation.

### 3.7 Iterative Solver

ALPS uses a different iterative solving method depending on the version chosen at runtime. The sequential version uses the SOR method or either can use the Jacobi method. Both solve a system of linear equations ($A\vec{x} = \vec{b}$) using successive iterations of the linear Euler-Lagrange equations provided by [1] to approximate a global minimum of the previously mentioned energy function. The resulting solution is the change in pixel displacement in the form of $\vec{du}$ and $\vec{dv}$ vectors. Each row of $A$ (the coefficient matrix calculated by *matrix builder*) represents a linear equation that is

solved for using an iterative method. The even rows of the solution vector, $\vec{x}$, are the values of $\vec{du}$ (the change in $x$-axis optical flow). Likewise, the odd rows of $\vec{x}$ represent the values of $\vec{dv}$ (the change in $y$-axis optical flow).

The iterative solvers initialize $\vec{x}$, to 0 during the first iteration. In subsequent iterations, the new $\vec{x}$ is generated row-wise. To find the new solution value, the previous solution to $\vec{x}$ is used in conjunction with the row of the coefficient matrix $A$ and vector $\vec{b}$. Upon completion of each iteration, the error of estimation between $\vec{x}_i$ and $\vec{x}_{i-1}$ is calculated. If it is below a certain threshold (i.e. trending towards convergence), then the function returns $\vec{x}$. If it does not reach convergence within the alloted number of iterations, an error state is returned.

The main difference between the SOR and Jacobi functions is that SOR uses values of $\vec{x}$ from previously solved equations within the iteration to determine the solution to the current equation. As such, the result of a particular iteration is dependent on the order in which the linear equations were solved. As detailed in Section 2.2, this does not perform well in a parallel context and therefore the Jacobi solver is used for the parallel version of this module.

**Algorithm 1** The Jacobi Method

**procedure** JACOBI BEGIN:$(A, b)$ ▷ The initial guess for $x^{(0)}$ is 0
  **for** $k = 1, 2, \ldots, maxIterations$ **do**
    **for** $i = 1, 2, \ldots, n$ **do**
      $sum = 0$
      **for** $j = 1, 2, \ldots, i-1$ **do**
        $sum = x_i + a_{ij}x_j^{(k-1)}$
      **end**
      $x_i = (b_i - sum)/(a_{i,i})$
      **end**
    **end**
  **return** $x$ ▷ check for convergence; continue if necessary

### 3.8 Bilinear Resizing Module

This module utilizes the OpenCV library (see Section 5.2.2) in order to perform bilinear resizing on image matrices. It simply converts an Armadillo type matrix (see Section 5.2.1) into a CV type matrix, calls the OpenCV resize function with a bilinear parameter and scale factor, and converts the data back into an Armadillo matrix before returning the result. This is used to create the next iteration of the Gaussian pyramid by sampling the original image at a particular resolution.

# 4 METHOD

## 4.1 Design Methodology

As initially presented in [4], the design and testing of this project is broken into two sections with different methodological approaches: the sequential ALPS implementation and the parallelization of the iterative solver.

Hierarchical design methodology is best suited for the sequential program due to the modular nature of the Brox algorithm. By first decomposing the algorithm into its logical modules, each module could be written and tested independently. Each module was decomposed into its smallest functional unit and tested for accuracy before integration into a larger module. Testing the modules involved comparing the outputs of the module in C++ to the outputs of the same logical part of Chari's implementation. The testing methodology used will be discussed in further detail in Section 6.1.

Once the modular sequential code was assembled and tested, a module for solving the system of equations was designed to run in a parallel environment. Once a working prototype was produced, subsequent updates were made to the code, further improving upon its performance through the addition of parallel capabilities. This follows the incremental waterfall methodology in which, upon the completion of each version of the code, the program is re-assessed and design components are added or improved upon. Each iteration allowed for more functionality to be delegated to the GPU, with an expected effect of a decrease in execution time. An explanation of the parallel implementation is found in Section 5.1.

## 4.2 Phase 1: Scoping and Definition of Requirements

The initial task for the project was determining its overall goal. This was to implement, with the use of a GPU, the Brox algorithm for accurate PSM estimation with improved performance over other available implementations. This would involve using the theory provided by [1] to perform a critical component of image correction (the PSM estimation) but not an entire image correction procedure. Two major implementation decisions were made with the intent of decreasing computation speed: the use of parallel programming on a GPU and the exclusive use of the C++ programming language.

## 4.3 Phase 2: Sequential Design

With the requirements in place, the next phase was to implement the Brox algorithm in C++. The first step for the design was to separate the algorithm detailed in [1] into logical modules that could be implemented in C++. This effort was aided by the MATLAB code [2]. Understanding how each module supported the procedure for PSM estimation was critical to the implementation of the program. As such, the majority of the project time was spent in this phase.

### 4.4 Phase 3: Parallel Design

Determining how, and what sections of the algorithm to parallelize was the next step. To maximize the time reduction, it was decided that parallelizing the slowest section of code in Chari's implementation would be most efficient, on the assumption that the C++ implementation would perform calculations similarly. As such, the method for solving the system of linear equations by SOR was the candidate module for parallelization. Using the form of sparse system of linear equations seen in [2], the Jacobi method was determined as suitable for parallel implementation. The reason for this was its avoidance of certain complex operations required by SOR such as matrix splitting and inversion, which do not apply well to a parallel environment.

### 4.5 Phase 4: Optimization and Final Design

The final phase of the project was involved refinement of the CUDA implementation and benchmarking of the final ALPS implementation. The parallel model was produced in a series of versions, each adding greater levels of parallelism, improving the end results. The testing of the two versions of ALPS (sequential and parallel) against the control (Chari's MATLAB implementation) is presented, and discussed in detail in Section 6.1 and Section 8, respectively.

## 5 IMPLEMENTATION

### 5.1 Parallel Module Implementation

The parallel implementation of the Jacobi solver employed the incremental waterfall approach. As such, the first prototype was designed to be simple and easy to debug. It consisted of running just the inner loop of the Jacobi solver in a CUDA kernel. Logically, this is described as: for each row in the $A$ matrix, all elements are checked in parallel for a non-zero value. The rows are executed one after another in sequence. Each row accesses an element of a sum vector for storing its local sum. Upon completion of all threads of execution the sums of all elements are combined and the particular row is solved for.

In the second version of parallel ALPS, more of the code was run on the GPU in a more complex environment with the effect of improved performance. This involved converting the a second sequential looping structure into a parallel architecture. All rows were run concurrently to solve for the entire $\vec{x}$ vector. This proved to be in fact be less complicated from a memory managing perspective than the first parallel version. This is due to the function no longer needing to synchronize the summation of a shared sum vector as all rows are completely independent. Instead, a local sum variable was used by each thread to perform the computations required to solve its small portion of the $\vec{x}$ vector.

For the parallel versions, the $A$ is in the form of a coordinate order sparse matrix. This employs a row and column vector to index a particular value. To effectively parallelize the row-wise iterations of linear systems in version two, the row-index vector was converted to compressed row

format. This allowed for greatly improved performance over the previous version due to the fact that threads are not wasted in checking every column for a match. Instead, since the values in the row matrix are indexes to values in the column vector, these conditions are done away with and every thread contributes to the calculation of the solution.

In larger images, the number of rows in the resulting $A$ matrix are greater than the number of threads available in a CUDA block. To solve this issue, threads are allocated to the maximum number within a block and excess rows are allocated to threads on subsequent blocks.

## 5.2 Libraries and External Software

This project employs the the Armadillo C++ linear algebra library, the OpenCV library and the CUDA developer toolkit. Each serves a critical role within the project, as discussed in this section but also carry drawbacks.

### 5.2.1 Armadillo C++ Linear Algebra Library

To facilitate the use of complex matrix calculations and manipulations, the Armadillo C++ library is implemented in all but the parallel Jacobi solver module. This enables a host of useful features such as indexing, submatrix views, splitting, element-wise operations, etc. These are completed using Armadillo's wrapper of the BLAS (Basic Linear Algebra Subprograms) library. Armadillo also simplifies the storage of matrix and vector types used extensively in ALPS. Finally, the readability of Armadillo supported matrix operations is greatly simplified. Table 2.1 shows the functionality that the Armadillo library provides to ALPS.

TABLE: 2.1 - DESCRIPTIONS OF IMPORTANT FUNCTIONS FROM THE ARMADILLO LIBRARY

| Operation | Functional Description |
|---|---|
| Convolution | returns the 2D convolution of two matrices |
| Shed | remove a row or column from a matrix |
| All/Any | tests condition against all elements of a matrix |
| Find | returns the row and column indices of the nonzero entries a matrix |
| Norm | returns the maximum singular value of a matrix |
| Solve | solves a dense system of linear equations |

### 5.2.2 OpenCV Library

OpenCV was used to translate images from their standard format (.png, .jpg, .bmp, etc.) into a corresponding matrix in which each pixel is given a value based on its grayscale brightness. Additionally, since Armadillo does not have a bilinear resizing function, OpenCV is used to perform resizing for the levels of the Gaussian pyramid.

### 5.2.3 CUDA Developer Toolkit

CUDA allows for GPU interfacing and processing. The project goal to improve performance of the Brox algorithm means that GPU interfacing is critical. Leveraging GPU capabilities allows for thousands of threads executing concurrently. In the case of this project, threads are assigned to operate on elements of a matrix independently of one another while accessing a shared memory space. In addition to the standard API, parallel ALPS employs the cuBLAS to perform normalization of vectors and cuSPARSE to convert a coordinate sparse matrix to a compressed sparse row format.

CUDA handles the identification of the GPU device in addition to allocating memory, copying data to the device, separating resources into logical blocks and threads and many other essential functions for GPU parallel processing. Using CUDA's library functions included within its API, memory is effectively managed for sending and receiving data.

## 6 VERIFICATION AND VALIDATION

To validate the outputs of the program, whether it be a sequential or parallel version, the procedure is identical. The program is run with inputs of an image with a known warp applied and a control image. If the resulting PSM is within $10^{-8}$, the program is deemed to be accurate to the extent required. This project involved an extensive amount of testing which needed to be supported by a sound testing methodology in line with the overall design methodology. The sequential ALPS was built and tested in modules before assembling for a final validation. The parallel versions of ALPS were tested with incremental additions to GPU usage.

### 6.1 Testing Methodology - Sequential

Each component of the sequential code was tested independently for the correct outputs. Each module requires correctness and accuracy from the previous modules in execution in order to produce a result within the project's required margin of error. Once all modules were verified they could be combined into the larger "linker modules". This design methodology allowed for a fine level of control due to the reduced scope. Additionally, it maximized the project team's output as both members could work independently on separate modules with minimal slack time. From a technical perspective the hierarchical decomposition of the logical code modules allowed for the isolation of faults within the code before the size of the program hindered this effort.

The majority of the time spent on this project was in the process of verification and validation. Due to a host of workspace issues and library linking issues near the start of the sequential implementation phase, most of the C++ code was written without testing for build or runtime errors. Although not ideal, this strategy was needed in order to continue making progress towards a runnable program while debugging the library and workspace issues. As a result of this challenge, testing of sequential ALPS required some modification from the hierarchal testing methodology proposed in [4]. Instead of coding small logical pieces of code and debugging them before continuing, most of the code was written before any testing took place. An efficient

methodology was devised for testing this code, a variation of the hierarchical design methodology, it can be described as “decomposition testing”. This strategy follows the flow outlined in Fig. 2.

To elaborate, each module is tested against Chari’s MATLAB inputs and outputs, extracted from the MATLAB debugger. The same inputs (typically matrices) were fed into the same logical portions of code in Chari’s implementation and ALPS, then the outputs were compared. If they were equal within a reasonable margin of error, it was determined that the module functions correctly. If they did not match or there were other problems running the code, the testing scope was narrowed and the outputs were compared again with identical inputs. This methodology saved debugging time as creating test inputs for each small code block was only needed if all larger modules failed to produce the correct outputs. Once all modules had been tested individually, they could be added together to ensure small errors did not propagate through the program.

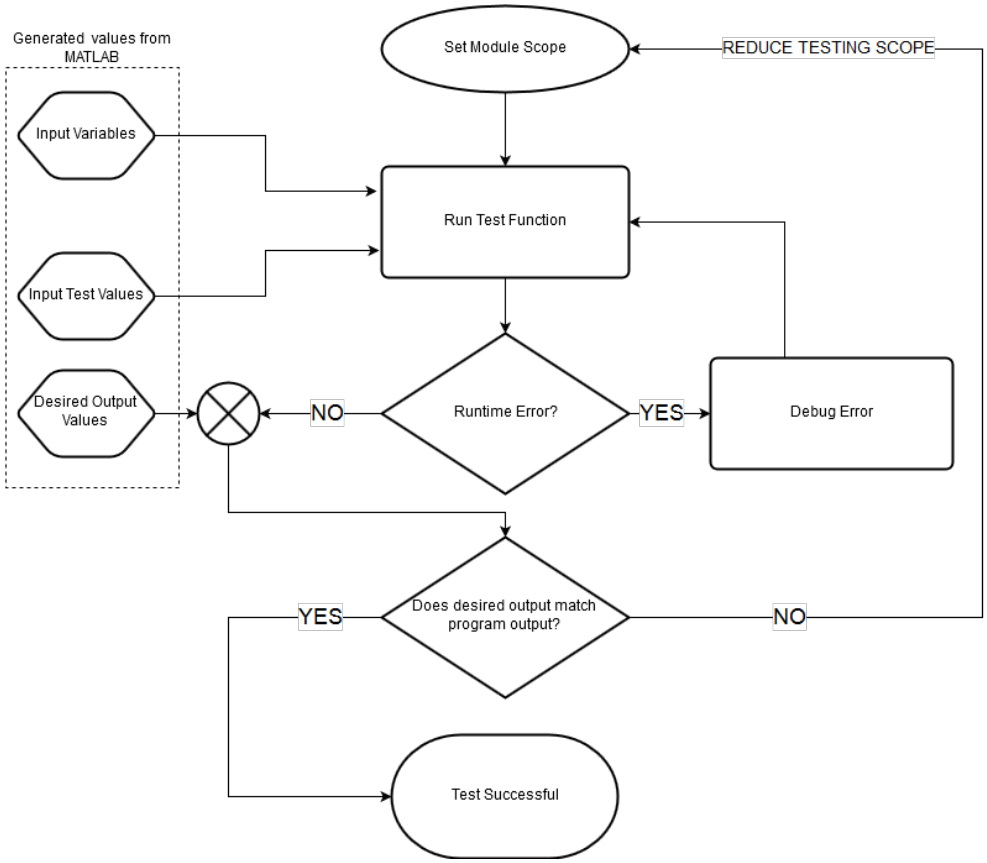



Fig. 2 - Hierarchical testing strategy

## 6.2 Testing Methodology - Parallel

The parallel implementation was tested in much the same way as the sequential modules. Since the first iteration of the parallel program only involved replacing the solver, the Jacobi solver module could be easily integrated into the modular program design and tested in the same environment as the SOR module. However, testing the CUDA code offered a unique set of

challenges. Particularly, dealing with allocating memory and referencing pointers is not something dealt with in the project until the point of testing CUDA code. An iterative approach was taken in order to debug random value returns from the kernel call. For this the serial Jacobi module was copied into the CUDA file and initialized to use pointer references instead of local variables. Once that was verified, the same code was attempted to run from the GPU in a single thread to isolate the source of the problem. Then, more threads could be added. Just as with the design methodology, the testing process incrementally added code into the parallel environment, testing at each step.

# 7 RESULTS

This program was required to produce an accurate PSM estimation in less time than Chari's matlab implementation. This leads to two major characteristics when examining the results of the project: speed of execution and accuracy of results.

## 7.1 Experimental Setup

To verify the accuracy of the estimated PSM, a series of test images were used as inputs. Since the program takes two images as inputs to determine the movement of pixels, each of four tests involved one control image and one image with a known warp applied. The estimated optical flow can then be compared against the applied warp to determine the accuracy of the program. A speed test of the grayscale image was then compared to the benchmark of Chari's implementation. Subsections 7.1.1 to 7.1.3 present the four levels of inputs used. The output parameters that were tested for were based on the requirements outlined in [3]. They are as follows:

a. accuracy of results relative to the known warp;

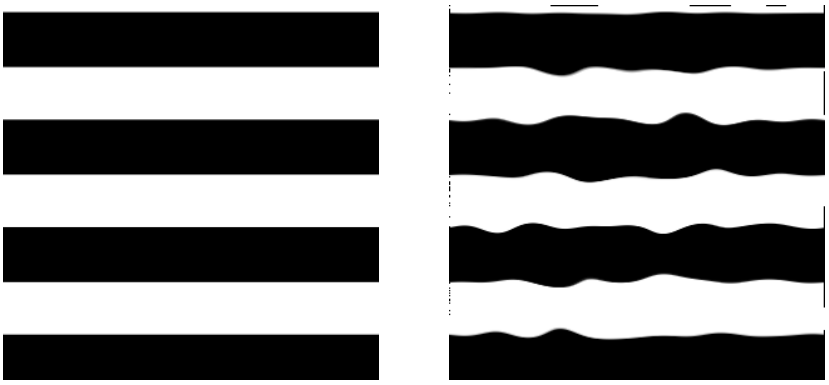
b. total runtime of sequential-c program (using C++ SOR module);

c. total runtime of program (using C++ serial Jacobi method); and

d. total runtime of program (using CUDA parallel Jacobi method and optimizations).

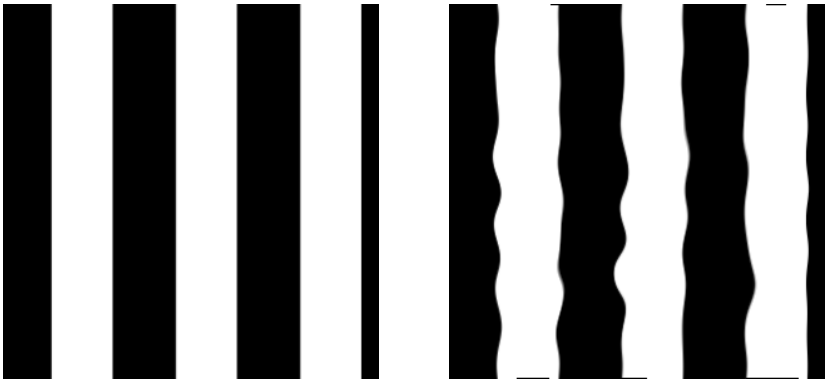
### 7.1.1 Vertically and Horizontally Warped Test Images

The first two test scenarios involved verifying the $x$ and $y$ displacement vectors independently. The first set of inputs (Fig. 3) included an image warped in the $x$-axis only, the second (Fig. 4) contained an image warped in the $y$ axis only. Using these inputs, a potential problem in the final estimated PSM could be isolated to either the $x$ or $y$ domain.



(a) Control (b) Warped Image

Fig. 3 - Horizontally warped test image and control

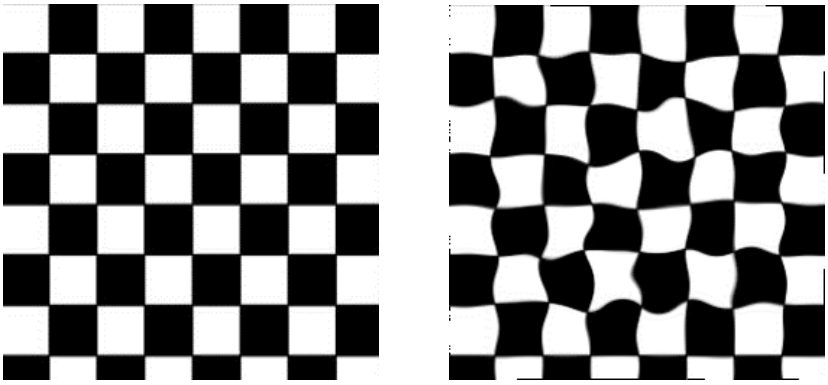


(a) Control (b) Warped Image

Fig. 4 - Vertically warped test image and control

### 7.1.2 Two-Dimensionally Warped Test Images

To further validate the accuracy of the previous vertical and horizontal warp tests, a checkerboard image warped in both $x$ and $y$ domains (Fig. 5) was used to test both concurrently.



(a) Control (b) Warped Image

Fig. 5 - 2-Dimensionally warped test image and control

#### 7.1.3 Full Grayscale Test Image

With a pure black and white image with well defined lines such as in Fig. 5, calculating the displacement of pixels may be simplified due to the distinction along object edges. To test robustly, more complex images with less defined object edges, such as that in Fig. 6 were used.


(a) Control (b) Warped Image

Fig. 6 - 2-Dimensionally warped grayscale test image and control

### 7.2 Presentation of Results

#### 7.2.1 Accuracy

The accuracy of a given version of ALPS is determined based on the difference between the PSM that it produces and the PSM produced by the MATLAB code. Due to an issue with the resizing module, the exact interpolation used by the MATLAB code could not be replicated. As such, error caused by the discrepancy between the the resize method used in MATLAB and that used in ALPS propagated through each iteration of the Gaussian

pyramid, leading to an error ranging from 0.001% to 20%. It should be noted, however, that all modules apart from the resizing module produce the expected results within an accuracy of 0.1% mean squared error.

### 7.2.2 Speed

The profile results of the two parallel ALPS versions is presented in Fig. 7. At the time of writing, the parallel implementations are not yet polished enough to be accurately profiled for speed.

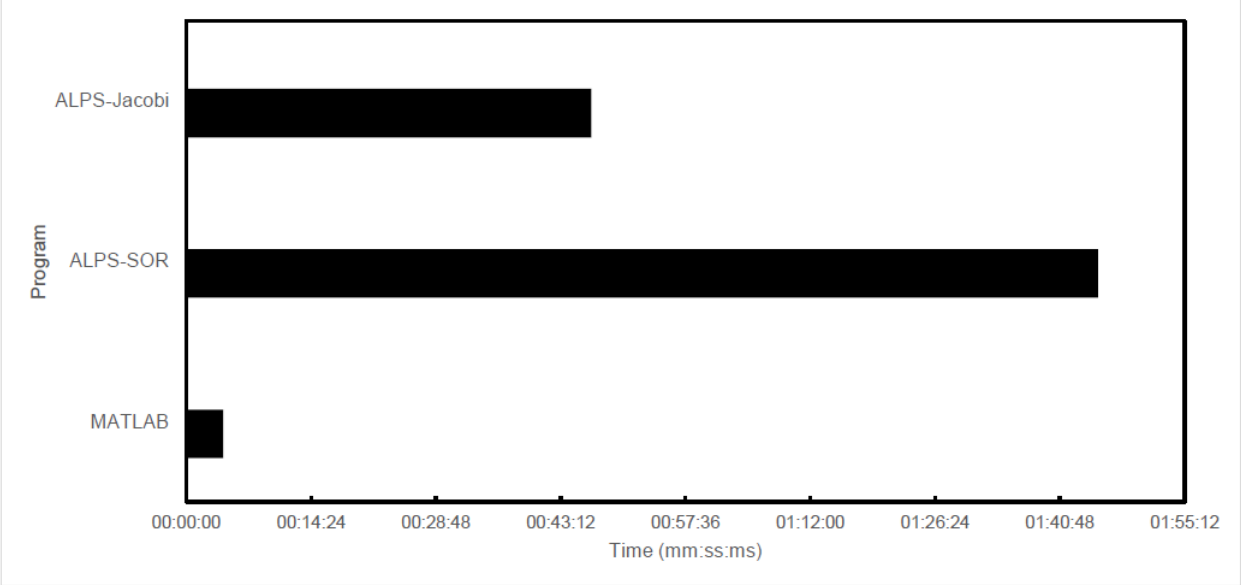



Fig. 7 - Average Execution Time of Sequential Brox Algorithm Implementations

## 8 DISCUSSION

The results of the project ultimately did not meet the goal at the time of writing, however, the modules themselves perform the necessary matrix operations with appropriate accuracy. Additionally, the design of this project lays the groundwork for a parallel implementation of Jacobi method capable of solving the system of linear equations provided by the Brox algorithm. Table 8.1 presents the requirements stated in [3] and how the project results compare.

TABLE: 8.1 - DESCRIPTION OF REQUIREMENTS

| Requirement | Description of Requirement | Result | Comment |
|---|---|---|---|
| FR-01 | A reference image and an image distorted with a known pixel shift will be used for the programs inputs | Satisfied | |
| FR-02 | The program will produce a pixel shift map in the form of two arrays: the $x$ component of the shift and the $y$ component of the shift | Satisfied | |
| PR-01 | The pixel shift map will be generated, at a maximum, in equal time as a benchmark of the original pixel shift map algorithm written in MATLAB | Not Satisfied | |
| PR-02 | The converted Brox algorithm must have a mean squared error of less than 10% | Partially Satisfied | All but one module meets the requirement |
| IR-01 | The program will interface with a GPU in order to perform calculations in parallel | | |
| IR-02 | NVIDIAs CUDA Toolkit shall provide the interface for sending commands and data between the programmer and GPU | Satisfied | |
| IR-03 | This program shall have an expert level user interface (i.e.the user will run an executable file in the command line) | Satisfied | |
| ImpR-01 | The product shall be written entirely in the C++ programming language | Satisfied | |
| ImpR-02 | Development of the parallel algorithm shall be accomplished using NVIDIAs CUDA parallel computing platform | Satisfied | |
| ImpR-03 | The program parallelization will be implemented on, and developed for, the Nvidia Quadro K2000D | Satisfied | |
| ImpR-04 | The benchmark of the Brox Optical Flow MATLAB code will be run on a Intel Xeon E5430 CPU | Satisfied | |
| ImpR-05 | The program will convert 64-bit variables used by MATLAB into 8-bit variables for use in C++ | Abandoned | double precision is required for accurate results |
| SR-01 | The first functional prototype of initial Brox Optical Flow Computation Algorithm translation into C++ shall be available for testing no later than January 8, 2018 | Not Satisfied | Functional prototype took much longer than expected |

Overall, most of the requirements were met, however, the critical requirements were not met to the level desired. This should not undplay the strong basis that this project sets for future implementations of the Brox algorithm in C++. ALPS provides accurate and efficient code modules for performing the complex matrix and mathematical operations required by the Brox algorithm. Additionally, a detailed design and an implementation of a CUDA supported Jacobi solver provides all of the tools needed to see this project to completion.

## 8.1 Unresolved Issues

From Table 6.1 there are some requirements that were not satisfied. Subsections 8.1.1 to 8.1.5 detail the unresolved issues of the project and provide recommendations for meeting these requirements.

### 8.1.1 Speed Requirement

The sequential versions of ALPS did not, and were not expected to, meet the speed requirement of PR-01. The parallel versions were both implemented but still contained runtime errors with certain inputs at the time of writing. The first version produces correct results when given certain inputs but not those given in the experimental setup. Since an average speed cannot be deduced for this version, it does not meet the requirement. The second version was completed with all CUDA memory accesses but at the time of writing it cannot accurately solve a system of linear equations. Having re-scoped the project focus more of the parallel implementation phase, would have potentially allowed for the completion of the parallel versions to the level in which they were designed.

### 8.1.2 Floating to Fixed Point Conversion

The requirement to perform a conversion from floating point to fixed point was removed from the project around the mid-point of its time-line. Double precision was required for many calculations within the program to retain accuracy of results. As a result, the 8 bit fixed-point number requirement was abandoned for the final implementation.

### 8.1.3 Scheduling

The main objective of this project was the implementation of PSM estimation procedure in parallel on a GPU. Unfortunately, the majority of the time spent on the project did not involve CUDA. This was due to an underestimation concerning the time required to manually write a C++ program to implement the Brox algorithm starting from a MATLAB implementation. Near the beginning of the project when the requirements were defined, this did not appear to be as difficult a task. It was assumed that Chari's MATLAB implementation would translate very smoothly into C++ with the help of a linear algebra library. This was not the case and resulted in many schedule adjustments due to unexpected

bugs, configuration issues, library incompatibilities and time spent learning new software. Requirement SR-01 was not met due to these unexpected delays. Having a better idea of the scope of the project and the tools available would have helped reduce these delays.

### 8.1.4 Library Issues

An issue that took a large amount of time away from the project goal was library linking issues, incompatibilities between libraries and library limitations.

A drawback to using these libraries is their lack of compatibility. CUDA cannot use Armadillo data-types for representing objects such as vectors and matrices. Additionally, CUDA code cannot perform Armadillo linear algebra operations. Fortunately, Armadillo provides a function to create a pointer to memory from its objects to simplify the conversion between Armadillo and standard C types. With these pointers and the sizes of the armadillo objects, standard arrays were defined and used in place of Armadillo's vectors.

There were some challenges that arose through using the CUDA interface. Firstly, as identified in section 5.2, the Armadillo library is not compatible with CUDA, therefore, Armadillo data had to be converted to standard C when passed to the CUDA module. Second, is the number of threads and blocks available and how that affects shared memory. Shared memory is allocated by block and 1024 threads can be run on a given block with the version of CUDA and GPU used. This means if more than 1024 threads should be ran at once, the shared memory space must be partitioned, further complicating the interface. A final consideration is bandwidth of the GPU. The performance improvement of executing multiple threads on the GPU over executing a single thread locally must compensate for the additional time required to transfer data to and from the GPU. Minimizing the bandwidth used while interfacing with the GPU is therefore critical to improving the performance of the algorithm.

OpenCV is required by ALPS to read image files as inputs. This returns a CV-type matrix and, similar to the conversion from Armadillo type to standard type, pointers to memory are used to convert back and forth between Armadillo and OpenCV types. Either matrix type for example, can be initialized from pre-existing data in memory given a pointer and dimensions of the matrix. This conversion is not trivial such as simply casting to another unit. Much time was spent studying the differences in the way both libraries represented Matrix data in memory so that the project could ensure accurate matrix conversions between the two libraries.

### 8.1.5 Image Resize Error

ALPs cannot resize images in the same manor as the MATLAB program. The resizing in ALPS is performed by an OpenCV function, resize(). However, it does not produce the same output as the MATLAB imresize() function due to differences in their methods of interpolation. The MATLAB function resizes using a *bilinear interpolation* method which differs from that offered in the OpenCV library, introducing an error of 5 - 10 % with each

call. Through experimentation, a replacement interpolation method for OpenCV was identified using *pixel area relation*. This interpolation reduced to 2-5 % with each call. This gap in resizing introduces error into the matrices that propagates rapidly as a resizing is performed multiple times in each level of the program.

# 9 CONCLUSION

ALPS successfully implemented the majority of the Brox algorithm into C++. The various components of the Brox algorithm were identified and the greater structure of the algorithm was decomposed into modules. These modules were successfully converted into C++ and hierarchically tested for compliance with their MATLAB counterparts. The C++ implementation was then incrementally built in versions leading to multiple releases of ALPS which followed the projects design and implementation methodology.

In addition, this project developed an iterative linear system solver that allowed for the Brox algorithm to be exploited by parallel computing methods. This was due to the fact that the project effectively identified the section of the Brox algorithm that spent the most time on the CPU; this being the SOR module.

ALPS has had limited success in maintaining accuracy with the MATLAB implementation due to inconsistencies with interpolations, however, it offers a modular architecture for which the Brox algorithm can be implemented. Finally, the parallel module has reaches beyond this algorithm and has the potential to be used to solve linear systems in a multitude of applications.

# A COMPLETE ARCHITECTURE OF ALPS

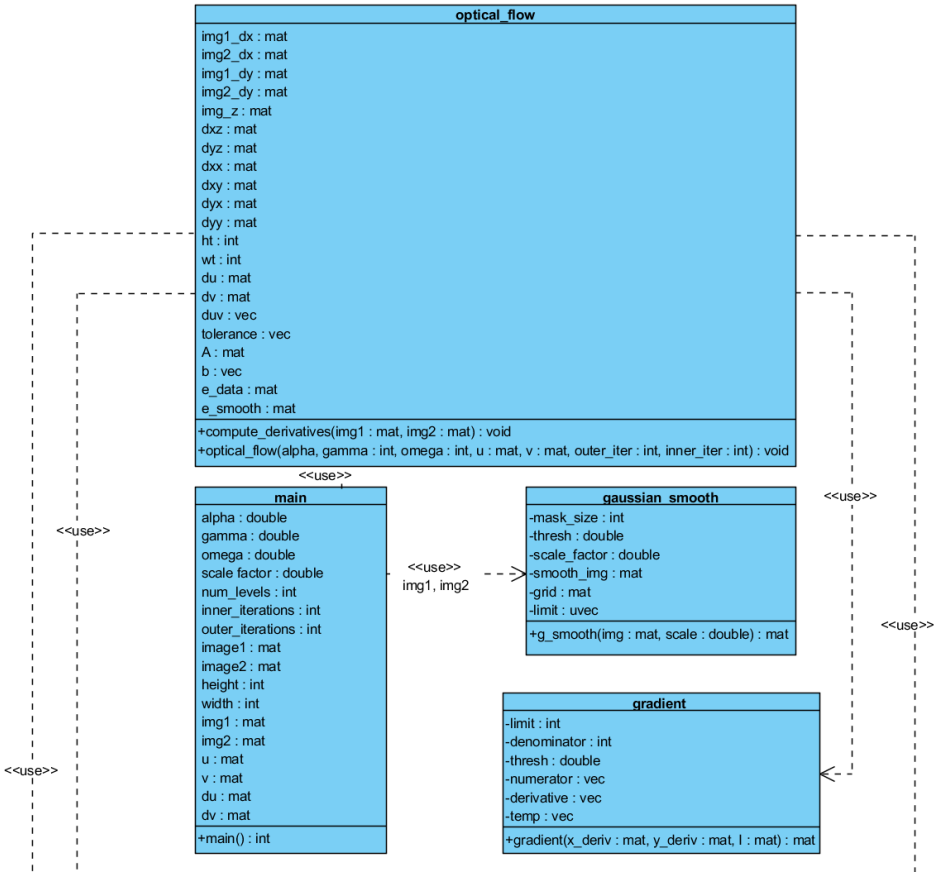



Fig. 8 - Architecture of ALPS (Part 1)

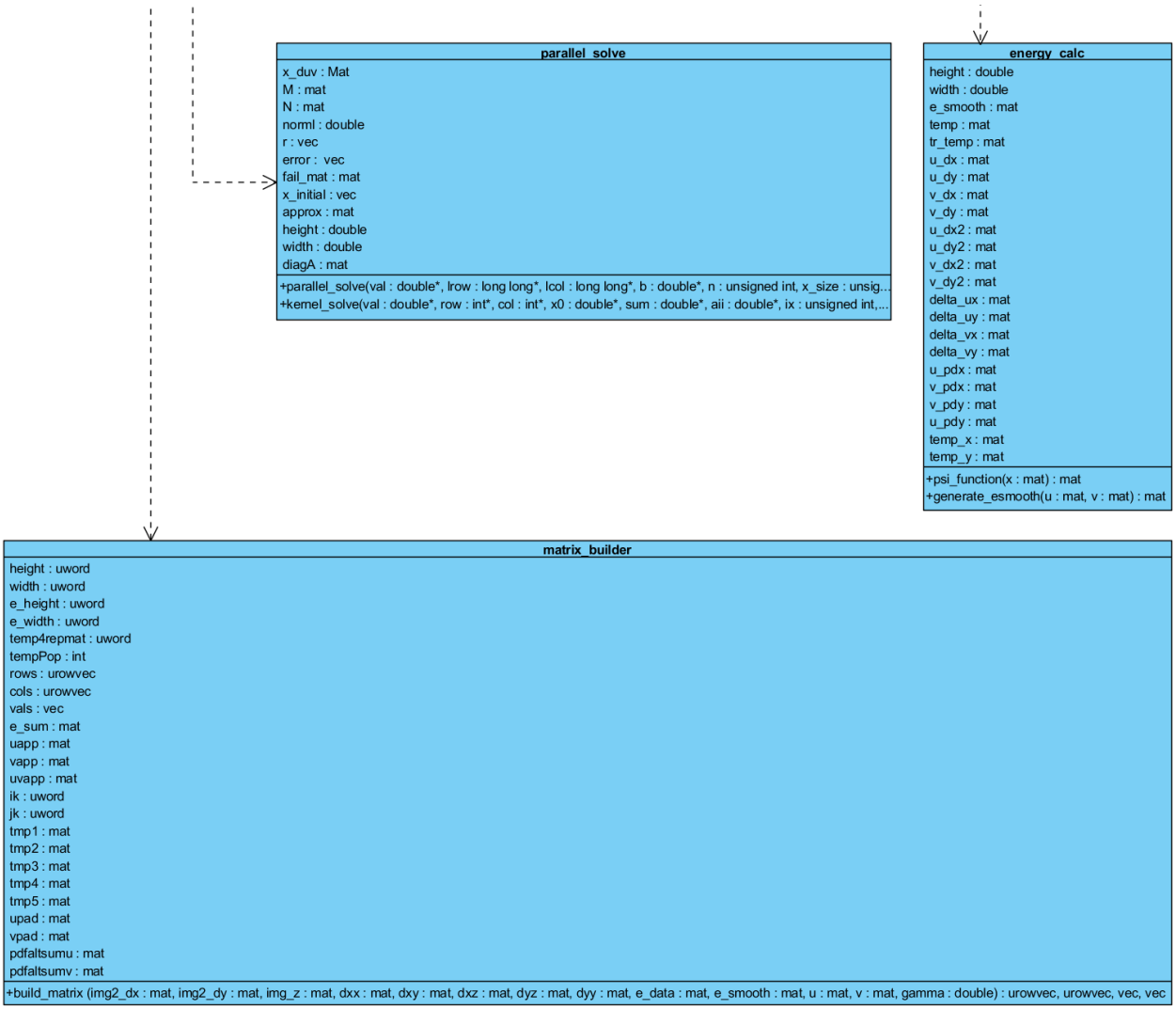



Fig. 9 - Architecture of ALPS (Part 2)

# B DESCRIPTION OF SELECTED SYMBOLS

Table B serves to support the architecture diagram presented in Appendix A by providing a description of some important symbols used.

TABLE B - DESCRIPTION OF SELECTED SYMBOLS (DRAFT VERSION)

| Symbol | Description |
|---|---|
| alpha | global smoothness parameter |
| gamma | weight given to derivatives in energy functions |
| omega | real relaxation scalar used in the SOR algorithm |
| num_levels | number of iterations of the guassian pyramid |
| inner_iterations | maximum number of iterations that the iterative solver will run |
| outer_iterations | maximum number of solver attempts if convergence is not found |
| u | change in pixel displacement in the $\vec{x}$ axis |
| v | change in pixel displacement in the $y$ axis |
| img_z | difference between gray-value matrix of images 1 and 2 |
| img_dx | partial derivative of the gray-value matrix of image 1 with respect to $\vec{x}$ |
| img_dxz | difference between the "$\partial_x$" of the gray-value matrix of images 1 and 2 |
| du | derivative of the change in pixel displacement in the $\vec{x}$ axis |
| duv | solution vector to the matrix system of linear equations |
| e_smooth | smooth energy matrix |
| e_data | data energy matrix |

# REFERENCES

[1] T. Brox and A. Bruhn, "High accuracy optical flow estimation based on a theory for warping," *8th European Conference on Computer Vision*, vol. 322, pp. 25–36, 2004.

[2] V. Chari, "High accuracy optical flow," 2009. [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/17500-high-accuracy-optical-flow. [Accessed: 14-Sep-2017]

[3] J. Wiebe and J. Dolman, "DID-03 - Statement of requirements: Parallelization of the pixel shiftmap estimation algorithm on a GPU," 2017.

[4] J. Wiebe and J. Dolman, "DID-04 - Preliminary design document: Parallelization of the pixel shiftmap estimation algorithm on a GPU," 2017.

[5] J. Wiebe and J. Dolman, "DID-06 - Schedule update: Parallelization of the pixel shiftmap estimation algorithm on a GPU," 2018.

[6] R. Barrett and M. Berry, *Templates for the solution of linear systems: building blocks for iterative methods*. Society for Industrial and Applied Mathematics, 1994, pp. 9–11.

[7] I. Cohen, "Nonlinear variational method for optical flow computation," vol. 1, 1993, pp. 523–530.

[8] J. Sànchez, N. Monzón and A. Salgado, "Robust Optical Flow Estimation," *Image Processing On Line*, vol. 2013, pp. 242–261, 2014.

[9] C. Sanderson and R. Curtin, "Armadillo: a template-based C++ library for linear algebra," *Journal of Open Source Software*, vol. 1, p. 26, 2016.