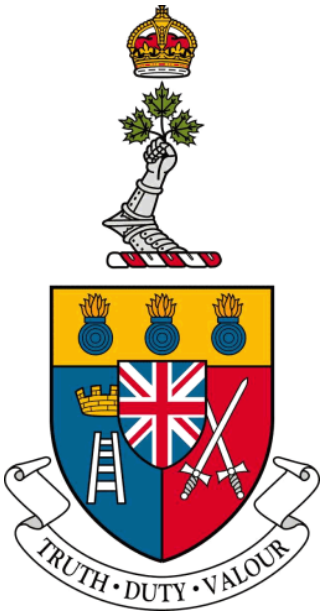
ROYAL MILITARY COLLEGE OF CANADA

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



DID-04 – Preliminary Design Specifications
Parallelization of the Pixel Shiftmap Estimation Algorithm on a GPU

Presented by:
OCdt Jacob Wiebe
OCdt James Dolman

Presented to:
Capt. A. Lapointe
Dr. D. McGaughey

November 24, 2017

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

TABLE OF ABBREVIATIONS

GPU	Graphics Processing Unit
PSM	Pixel Shiftmap
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
RGB	Red Green Blue
SOR	Successive Over-Relaxation

1 INTRODUCTION

This document outlines the initial models, design decisions, equipment, schedule and risks forming the preliminary design specifications of the project: Parallelization of the Pixel Shiftmap Estimation Algorithm on a GPU. The project will implement Brox's optical flow estimation algorithm [1] using C++ and CUDA to interface with a GPU. The program will employ parallel functionality to produce results faster than a current MATLAB implementation by Visesh Chari [2], with an acceptable level of error (see DID-03 [3] for requirements). This will be accomplished using three techniques:

1. employing fixed-point numbers instead of commonly used floating-point numbers
2. partitioning data structures to be processed in parallel
3. parallelizing iterative functionality in the algorithm

A hierarchical design methodology will be used for the design and testing of the first phase of the project: the sequential C++ implementation of Brox's algorithm. This will be built from the reverse engineering of Chari's MATLAB code. An incremental waterfall design methodology will be employed for the second phase: program parallelization. This will produce multiple prototypes with increasing parallel functionality

The reasons for hierarchical methodology for the first phase is due to the modular nature of the program design. 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. As such, each module will be decomposed into its smallest functional unit and tested for accuracy before integration into a larger module.

For the parallelization phase of the project, incremental waterfall design better suits the problem of parallelizing functionality in the context of this project. Incrementally, functionality within the sequential program will be implemented in parallel. With each iteration, more of the code will be delegated to the GPU, with an expected effect of a decrease in execution time.

The breakdown of this document is as follows: section 1 will identify the purpose of this document as well as expand on the background of the project. Section 2 will detail our preliminary design by first providing an overview of the project's logical modules. The subsequent subsections within section 2 include fixed to floating-point conversion, parallel design, and verification and validation. Section 3 will identify the necessary equipment. Section 4 will discuss scheduling. Section 5 will investigate the risks and issues with development, and provide solutions to compensate. Section 6 will conclude the document.

1.1 Document Purpose

The purpose of the Preliminary Design Specification document is broken into the 3 following subtopics:

a. to highlight the overall system architecture by detailing the key modules required;

b. to provide groundwork for the project's detailed design phase through additional background information and design decisions; and

c. to outline the project's schedule to organize the necessary tasks and deadlines.

1.2 Background

The use of long range optics results in distortion of images due inconsistencies in the atmosphere. The random warping effect on subsequent images presents, visually, an apparent motion of objects despite a stationary subject and aperture. This can be corrected, or "dewarped" by mapping the displacement of pixels across multiple frames and shifting based on the result to generate a single, clarified image. 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 PSM. To generalize, a PSM is a field of two-dimensional pixel displacement vectors from subsequent frames aimed at a particular region of interest. The algorithm detailed in [1] provides the basis for estimating a PSM that will be used in this project. The remainder of this section will discuss the characteristics of the Brox algorithm and how parallel programming can be applied to improve results of a software implementation of this algorithm.

1.2.1 Brox Algorithm Characteristics

The intent of the Brox algorithm is to calculate, with minimal error, an estimated PSM between two images. The vectors that make up a PSM are the optical flows between two corresponding gray values in two images, shortly separated in time. In order to perform the calculation accurately, the Brox algorithm provides the following constraints/assumptions: [1]

a. Gray value constancy assumption

b. Piecewise smooth optical flow field

c. Course-to-fine warping strategy

Like many optical flow methods, the Brox algorithm holds the brightness constancy assumption. This assumption states that the gray value of corresponding pixels in two consecutive frames should be the same. The mathematical representation of this assumption as presented in [1] is:

$$I(x, y, t) = I(x + u, y + v, t + 1) \tag{1.1}$$

The issue with this assumption is that it exposes the result to image degradation vulnerabilities. This is because the gray value constancy assumption will inevitability be violated in a field setting by the inherent noise of a camera sensor, as well as changes in illumination of the subject. To compensate, the Brox algorithm allows for small variations in the gray value of pixels to ensure that the calculated displacement not greatly affected by

these changes. This is accomplished by using the gradient of the image gray value, as seen in

$$\nabla I(x, y, t) = \nabla I(x + u, y + v, t + 1) \tag{1.2}$$

where $\nabla = (\partial_x, \partial_y)$.

Another image degradation vulnerability due to this assumption is motion ambiguity. Motion ambiguity occurs when an untextured object, one with uniform gray value across its area, cannot be distinguished when viewed through an aperture with a relatively small viewing area. Edge information is necessary to observe displacement in uniform gray values as individual pixels cannot be distinguished since they all hold the same gray value [4]. Therefore, if an untextured object moves within an image, the motion within the object cannot be detected without more information. This is refereed to as the aperture problem and it is illustrated in Fig. 1, where the ring is the field of view of the aperture. In the first two images, the movement of the untextured object cannot be determined. In the right most image, however, there is sufficient edge information to distinguish the displacement of the object.

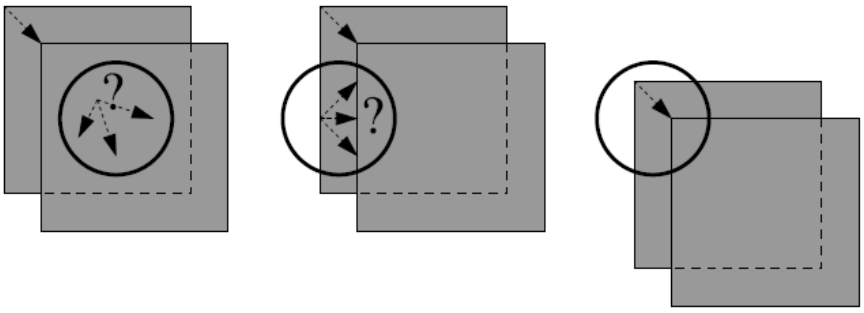


Fig. 1 - Illustration of the aperture problem as a consequence of motion ambiguity [4]

To mitigate the aperture problem, the Brox algorithm introduces a smoothness assumption by requiring a *piecewise smooth* optical flow field [1]. This smoothing adds more information to the image to combat untextured object areas by taking into account surrounding pixels. Additionally, if optical flow calculations do not take into account interaction between neighbouring pixels, problems occur in areas where there are discontinuities in the optical flow field. A piecewise smooth optical flow field reduces these problems by smoothing the discrepancies between the displacement vectors of pixels in a given area.

The calculation of an estimated PSM of high resolution images is a challenge if not decomposed. Fig. 2 illustrates how an image is decomposed in a Guassian pyramid with levels of increasing resolution. Using this, a complex image can be reduced so that the apparent global minimum is solved simply. At the higher levels of the pyramid, high frequency image details, represented as local minimums, are filtered out. As the resolution increases, there is an estimated location for a global minimum from the previous iteration which allows for a more efficient and accurate solution of the current iteration.

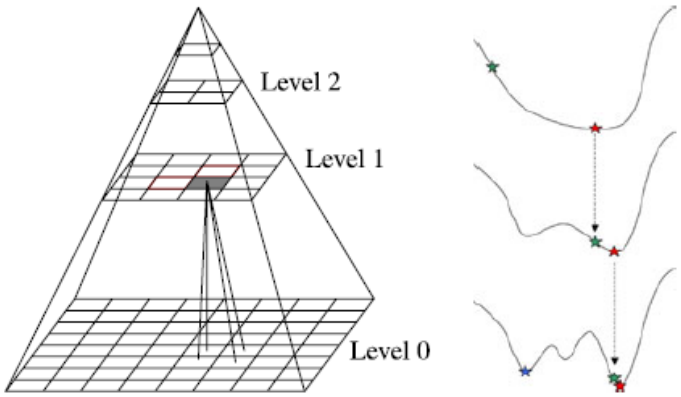


Fig. 2 - Pyramid decomposition of an image [4]

1.2.2 Parallel Programming

Discussing parallel programming and GPU function will set the grounds for the design of the second phase of this project. The advantages of parallel computing was discussed in general terms in [3]. The goal of parallel processing with regards to this project is the improvement it offers to processing speed. With the use of a GPU, hundreds of processes can be run concurrently. Parallel programing is the design and implementation of a program that will execute parallel processes. This involves the decomposition of tasks into smaller operations which can be executed in parallel.

A major issue with the parallelization of this project is the dependency of data. The core structure involves multiple loops, each with iterations requiring the previous output to refine numerical approximations. It is therefore not feasible to run all iterations of the sequential program concurrently. Instead, more complex methods must be considered. For the purpose of this project, two major parallel design aspects are presented. First, with some modifications to the sequential algorithm design, there is the potential for running multiple iterations of certain calculations concurrently. Second, the extensive number of matrix manipulations used can allow for partitioning of matrices into multiple smaller matrices, in which operations can be performed on in parallel.

This project will compound both strategies mentioned to design the parallelization of the Successive Over-Relaxation algorithm using the Red/Black method as explained in section 2.4. This method for solving dense matrix linear equations allows for an iterative estimation process to be run as multiple processes concurrently. It uses both a time approach, in which a sequential loop is run as multiple threads, and a spatial approach, in which the matrix is split into red and black elements which can be operated on independently.

Given the purpose of this project; to increase the speed of estimating a PSM, optimization of the parallel program must be considered. Since, optimization is the final stage of the project, (grouped into the testing and evaluation, see section 4) it will only be covered briefly in this document, and in much more detail in DID-07. In general, a greater number

of processing units available, the more efficient the program becomes. In order to optimize performance of a parallel program, the tasks should be evenly spread across all cores and made as simple as possible. An additional factor that will vastly improve results is the floating-point to fixed number conversion, explained in section 2.3.

2 DESIGN SECTIONS

2.1 Overall Architecture

The parallelized C++ implementation of the Brox algorithm will consist of a series of code modules. These can be divided into core modules and supporting modules. Core modules will sequentially step through the phases of the algorithm, calling most of the supporting modules. For example, the core modules will handle reading images and gathering the necessary variables (partial derivatives, energy, smoothness, etc.) of to solve Brox's formulae. The supporting modules will chiefly perform mathematical functions on matrices. Due to the restraint of working in the C++ programming language, there are many built in functions in MATLAB which must be imported from C++ libraries to achieve the same functionality. Most mathematical matrix operations will be imported from the Armadillo linear algebra library (see Table 2.2 for a complete list of functions that will be used). For reading images, the OpenCV2 library will be used. Fig. 3 shows the a proposed architecture of the sequential program based on Chari's MATLAB implementation. It provides a logical blueprint for the sequential C++ program and loosely represents files and methods that will be involved. The symbols used are defined in Table 2.1. The functional descriptions of the modules in Fig. 3 are described in section 2.2.

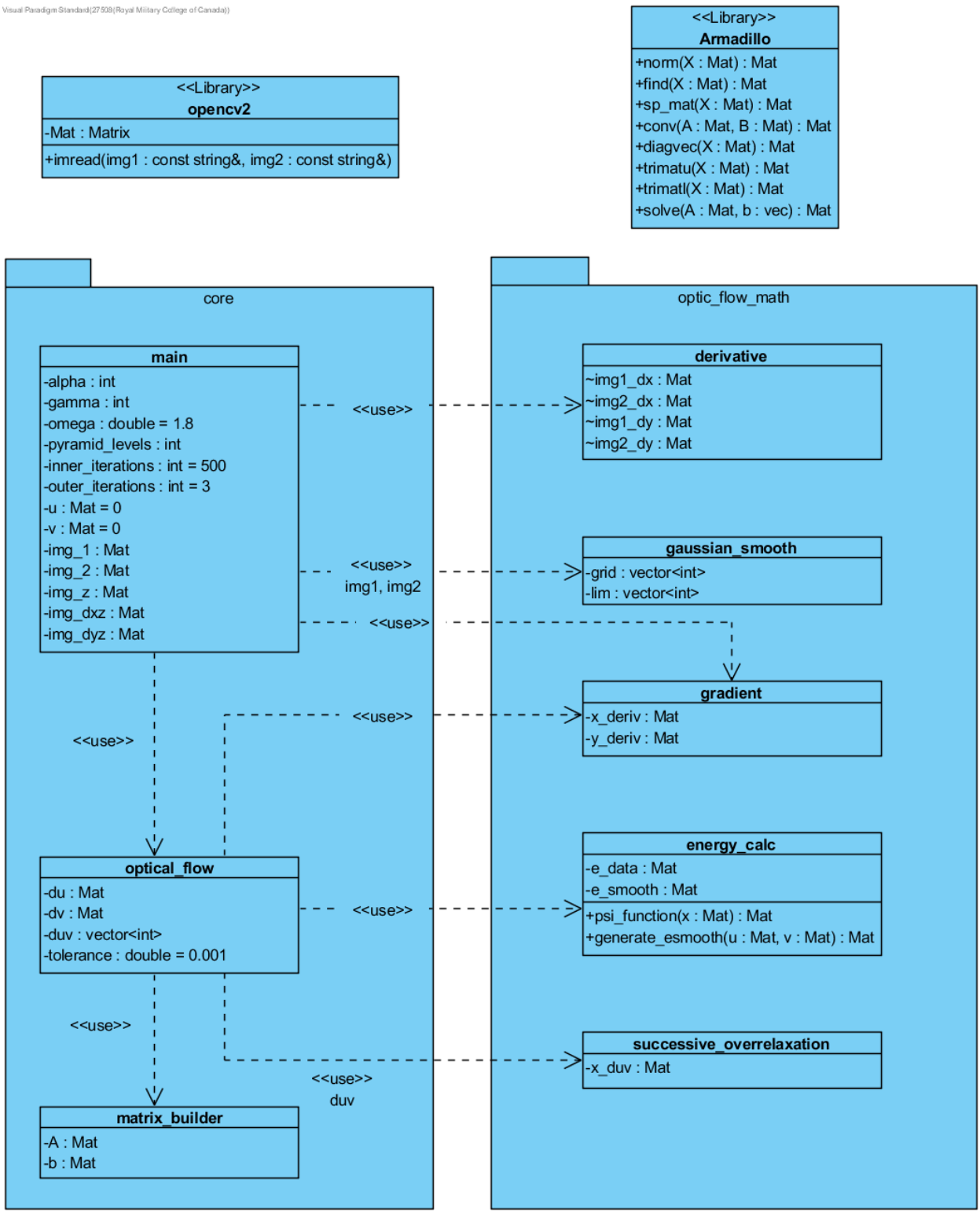


Fig. 3 - Logical architecture of the sequential C++ implementation of the Brox Algorithm

TABLE: 2.1 - DESCRIPTION OF SELECTED SYMBOLS

Symbol	Description
alpha	global smoothness parameter
gamma	weight given to derivatives in energy functions
omega	real relaxation scalar used in the SOR algorithm
pyramid_levels	number of iterations of the guassian pyramid
sor_ iter_in	maximum number of iterations SOR will run
sor_ iter_out	maximum number of SOR attempts if convergence is not found
u	change in pixel displacement in the x axis
v	change in pixel displacement in the y axis
img_1	gray-value matrix of image 1
img_2	gray-value matrix of image 2
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 x
img_dxz	difference between the "∂_x" of the gray-value matrix of images 1 and 2
du	derivative of the change in pixel displacement in the x axis
duv	vector containing the SOR algorithm solution to a system of linear equations
e_smooth	smooth energy matrix
e_data	data energy matrix
grid	smoothed image vector

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

d. Solving the resulting system of linear equations using SOR

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

To provide more clarity into the call sequence of the sequential C++ implementation, Fig. 4 presents an activity diagram modeling the major actions of the program. Moving from top to bottom roughly represents the order in which functions are called. The items in Fig. 4 represent logical modules and are not necessarily coupled to functions or classes.

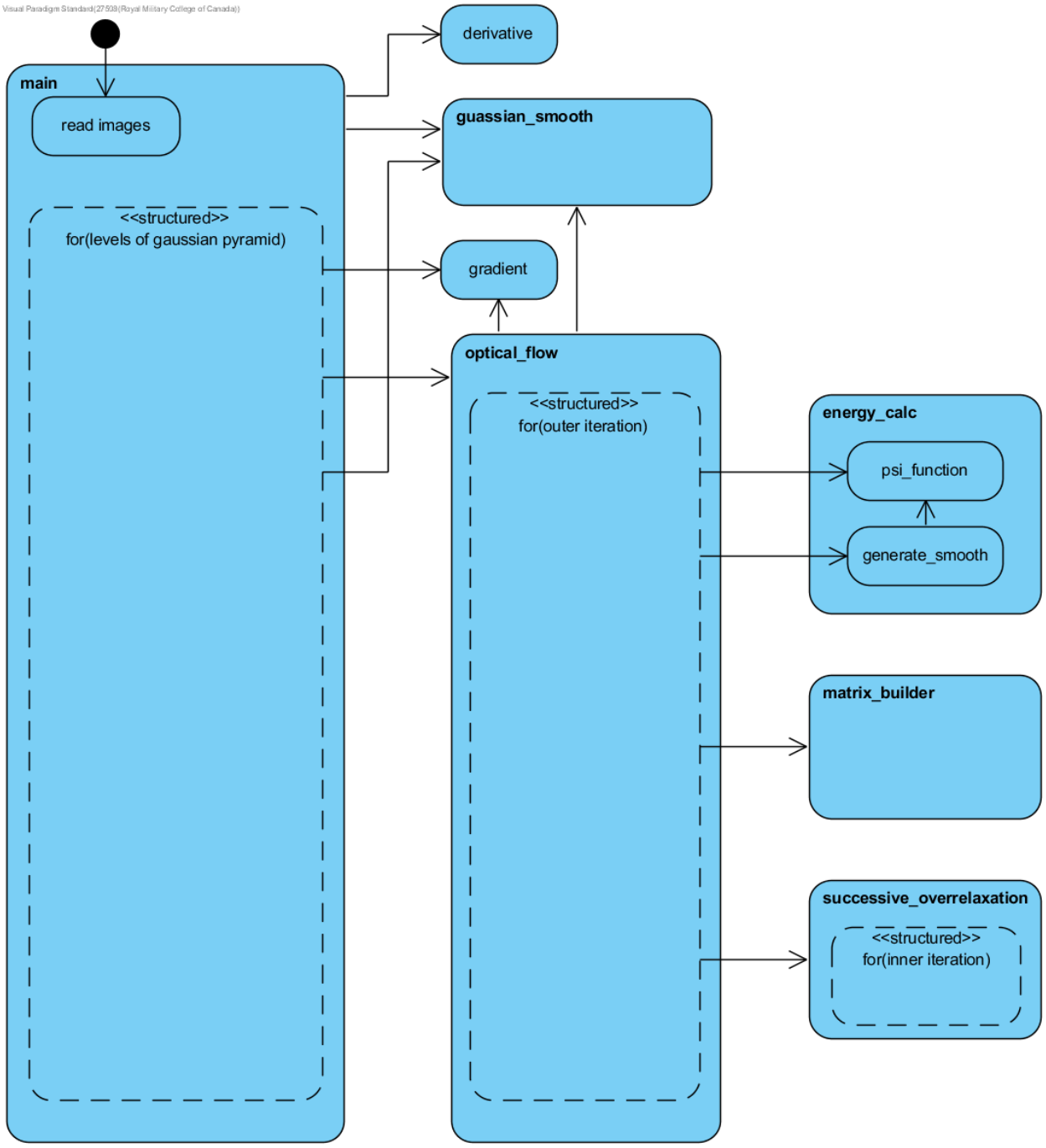


Fig. 4 - Logical activity diagram of sequential C++ implementation of the Brox Algorithm

2.2 Software Module Descriptions

2.2.1 Main Module

The main module will run sequentially and transform two given images into an estimated PSM based on the Chari's implementation of Brox's Algorithm. It chiefly serves to perform initialization of image matrices, displacement matrices (which will form the basis for the PSM) and various input parameters. Additionally, it will perform the consolidation of all matrix data and output the estimated PSM. The input parameters, alpha, gamma, omega as well as the number of levels for the Gaussian pyramid and the number of SOR iterations can be specified to balance speed and quality. To accurately compare execution time between versions, these parameters will remain constant between all prototypes of the sequential and parallel programs.

2.2.2 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 will produce the derivatives of the u and v vectors (i.e. the rate of change of the displacement of pixels in the x and y plane, respectively). All subsequent modules will be executed from this module.

2.2.3 Energy Calculation Module

Calculating the energy of the pixel displacement field is critical to the Brox algorithm. 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:

$$E(u, v) = E_{Data} + \alpha E_{Smooth} \tag{2.1}$$

where α is a variable parameter. Data energy and smooth energy are expressed as:

$$E_{Data} = \int_{\Omega} \Psi(|I(x + w) - I(x)|^2 + \gamma|\nabla I(x + w) - \nabla I(x)|^2)dx \tag{2.2}$$

and

$$E_{Smooth} = \int_{\Omega} \Psi(|\nabla u|^2 + |\nabla v|^2)dx \tag{2.3}$$

respectively [1]. Equation (2.2) 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 γ parameter allows for the scaling of this constraint, effectively weighting the importance of smoothness [5]. Equation (2.3) is the model used for describing the spatial smoothness. In both energy equations, the Ψ function is applied where $\Psi(s^2) = \sqrt{s^2 + \epsilon^2}$ (ϵ is 0.0001 for the purposes of the algorithm). These energy formulas form a system of linear equations. In C++, the Armadillo library will be used to compute divergence through the used of convolutions.

2.2.4 Successive Over-Relaxation Module

Two common classes of methods for solving a linear system of equations are direct and iterative methods. Direct solving methods tend to complete faster than their iterative counterparts, at the cost of reduced precision due to round-off errors. In addition to precision, iterative solving methods have another advantage: they tend to be less memory intensive due to their fundamentally simple calculations [6]. This is particularly useful for a GPU interface as the reduction of memory usage equates to less overhead, ultimately improving execution speed. See section 2.4 for more details regarding the use of memory in parallel computation.

In Chari's program, the SOR Algorithm is used for solving a dense system of linear equations. This algorithm uses successive iterations of the linear Euler-Lagrange equations provided by [1] to approximate a global minimum of the previously mentioned energy function. A template for the SOR algorithm provided [7], is referenced in Chari's implementation. The C++ implementation will employ this reference in conjunction with the MATLAB code to build the SOR module. The Armadillo library will be essential to this module as it requires multiple matrix operations. The C++ implementation will produce a solution in the form of a vector (duv), in which odd indexes are the elements of du and even indexes are the elements of dv.

2.2.5 Math Modules

The MATLAB implementation of the Brox Algorithm calls many built in MATLAB functions which perform complex mathematical calculations. The default C++ libraries do not provide most of these operations. Fortunately, the Armadillo library has a C++ function for many built-in MATLAB functions on matrices used by the existing Chari implementation. Table 2.1 shows the functionality that will be imported from the Armadillo library for the C++ implementation [8].

TABLE: 2.2 - COMPARISON OF MATH FUNCTIONS FROM THE ARMADILLO LIBRARY

Operation	Functional Description
Convolution	returns the 2D convolution of two matrices
Find	returns the row and column indices of the nonzero entries a matrix
Sparse	removes (by squeezing) zeros from a matrix
Norm	returns the maximum singular value of a matrix
Diagonal	returns a column vector of the main diagonal elements of a matrix
Triangular-Upper	returns the elements of a matrix on and above a certain diagonal line
Triangular-Lower	returns the elements of a matrix on and below a certain diagonal line
Solve	returns X, where "$A \times X = B$" is a dense system of linear equations

The gradient function is grouped with the math module but is not contained in the Armadillo library. Instead, it will be manually built, reverse engineering the gradient function in Chari's implementation.

2.3 Floating-Point to Fixed Point-Conversion

One design challenge as required by [3] is to use fixed point numbers while interfacing with the GPU. The purpose of this requirement is to reduce the amount of data processing, compared to using floating-point numbers, therefore improving overall performance. The MATLAB implementation uses the default type of double for the elements in all matrices. This is represented by 64 bits. The C++ implementation will represent matrix data in 16 bit fixed-point integers. Simply converting the integer value of a floating-point number into 16 bits would result in grave precision loss. Therefore, the design will employ the Q number format in order to represent decimal values as fixed-point integers. Q4.11 notation will represent one signed bit, four integer bits and eleven fractional bits giving a number range of -16.0 to 15.9995, inclusive, with a resolution of $\frac{1}{2048}$.

The major drawback of this design decision is that the precision of the matrix data has been reduced from its floating-point number representation. Two assumptions mitigate this effect on data accuracy. First, the integer portion of pixel displacement is assumed to always be within the range that can be represented with four bits (approximately -16 to 16). Secondly, the effect of precision greater than $\frac{1}{2048}$ is considered negligible in for the purposes of this algorithm.

Despite the reduced precision of using fixed-point in place of floating-point numbers, the trade-off is significantly less data being transfered to, and operated on by the GPU. For example, if the PSM is being estimated using two 512×512 pixel images the total data size of the PSM will be 525KB using 16 bit fixed-point numbers. This is compared to a floating-point PSM of 2098KB.

2.4 Parallel Design

The most intensive phase of this project is the parallelization. The goal, as defined by [3], is to run part of the program in parallel on a GPU with the intent of reducing computation time. Functionality will be parallelized based on two criteria, sequential computation time and data independence. Functions and modules are to be prioritized based on their sequential computation time. Longer times to completion in sequence provide a greater performance return if executed in parallel. Areas of significant computation time where identified through analysis through a preliminary test using the MATLAB function *profile* on Chari's MATLAB program.

2.4.1 Identification of Computational Bottlenecks

Using the MATLAB program in a *profile* session, the time to completion of each method was measured. More specifically, the time that a given method actively spent executing and also the time that it spend passively executing via child functions that it called was extracted.

Profile Summary

Generated 16-Nov-2017 15:37:36 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
optic_flow_brox	1	35.402 s	0.355 s	
resolutionProcess_brox	25	26.838 s	5.609 s	
sor	74	17.393 s	16.988 s	
imshow	144	7.245 s	3.896 s	
constructMatrix_brox	74	1.833 s	1.781 s	

Fig. 5 - Top level *profile* summary of *optic_flow_brox* in MATLAB implementation

One can see that a large amount of time was spent in *resolutionProcess_brox* and also the function it calls: *sor*. From this preliminary test of the MATLAB code, the method that uses the most CPU time, by a great margin, is *sor*. As discussed in 2.2.4, this method solves a system of linear equations, returning the vectors u and v. When iterations of a loop do not require information from their previous iteration, it is simple to call all iterations at the same time to be run on the GPU. However, the challenge with decomposing this method, as to be run in parallel, is that each iteration of its looping structure is dependent on variables from the previous iteration.

2.4.2 Successive Over-Relaxation Parallelization

The SOR algorithm does not uphold data independence in its original from, however, this can be solved using the red/black method. In general terms, the red/black SOR method solves iteratively through the "red" and "black" values, having no dependency between the two sets. This first involves labeling each element in the matrix with alternating red and black logical values. For a row i and column j, red elements are those in which "$i + j$" is even and black elements are where "$i + j$" is odd [6]. Fig. 5 displays how red and black elements are arranged.

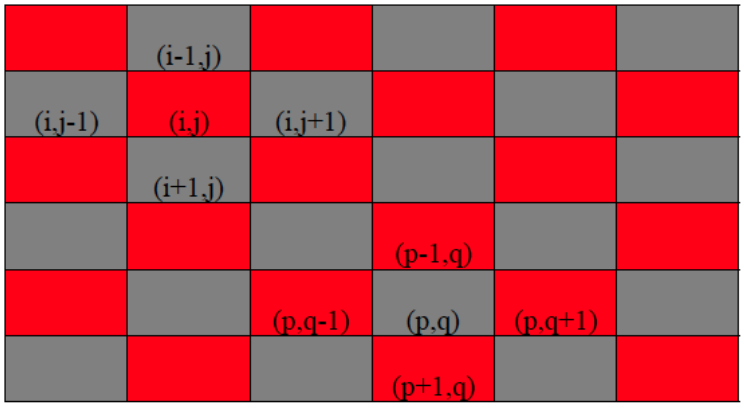


Fig. 6 - Arrangement of red and black elements in a matrix [6]

The colour sets are split into their own matrices as displayed in Fig. 7. Matrix reordering can take significant time to complete on large matrices, however, [9] states that the performance improvements due to the avoidance of sparse matrix accesses will outweigh reordering time.

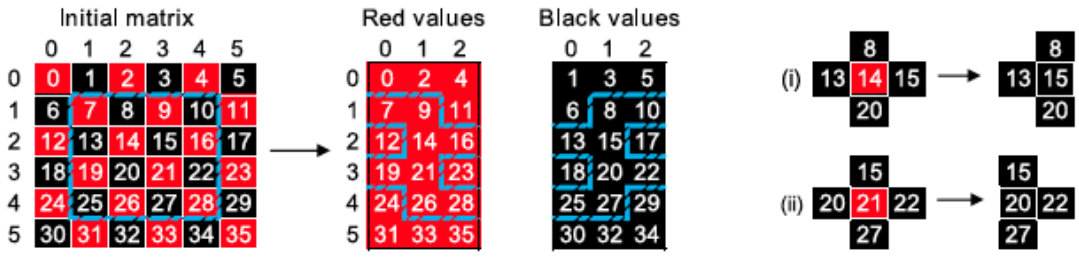


Fig. 7 - Separation of red and black matrix elements [9]

Multiple threads can be run within each colour set. Each iteration involves either the calculation of all black or all red elements. Once both sets have finished, they are synchronized by the controller and started again. This will continue until the result is below a certain error margin, at which point the algorithm is complete.

2.5 Verification and Validation

There are two major phases to this project: the implementation of sequential C++ code with fixed-point data representation and the parallelization on the GPU. Since they are very different design process, they will be tested in different ways, with the emphasis being on the design heavy parallelization.

For the sequential program, a hierarchical design and verification process will be implemented. Functionality will be tested as unique functions first, then implemented into the larger project. Logical modules shown in Fig. 3 will be tested with known inputs to verify their outputs. To verify output correctness, control matrices will be used as inputs for both C++ and MATLAB. If the outputs match, the module is producing the desired results. As the modules become more complicated, test values can be run on Chari's MATLAB files (which often include many functions), to verify the outputs of the sequential C++ implementation of the same logic. Due to the conversion of matrix elements from floating-point to fixed-point using Q notation, the validation of each module's accuracy is essential. The output of a function may be correct when ran once, however, after 500 iterations of rounding errors the result could be vastly inaccurate. Once individual module accuracy has been assured through the method mentioned previously, the accuracy of these modules operating in conjunction must be tested. This is to verify that there is no error propagation that develops through the program; that total program accuracy is still within the requirements.

For the second phase, the parallelization of C++ code, an incremental waterfall method will be employed. As discussed in section 2.4, the functions or lines of code with the greatest execution

time that are not restricted by dependent data will be prioritized for parallelization. From the preliminary analysis of Chari's MATLAB code, there are few independent regions; many modules are iterative and rely on the results of previous iterations. However, there are methods, such as the red/black method seen in 2.4, where independent regions can be created. As mentioned in section 2.4.1, an area of key interest for this project is the SOR module. In the current MATLAB implementation, it the most computationally intensive module and thus greatly affects the algorithm's performance. Therefore, it is logical to run this module in parallel as it will provide the most performance gains over other modules in the Brox algorithm. After an implementation such as this, more parallel functionality can be added to further optimize performance, or precision, in future releases.

To test for correctness in a estimated PSM, control image sets will be used. Each set has an original image and one that has been distorted by a known warp. Because the applied warp is known, the optical flow and resulting PSM are also known. With this information, an estimated PSM outputted from the program can be compared to the known PSM to test the mean square error. Fig. 8 shows an original image and its resultant warped image. The estimated PSM should accurately approximate (within 10% mean squared error) the displacement of pixels between the images.

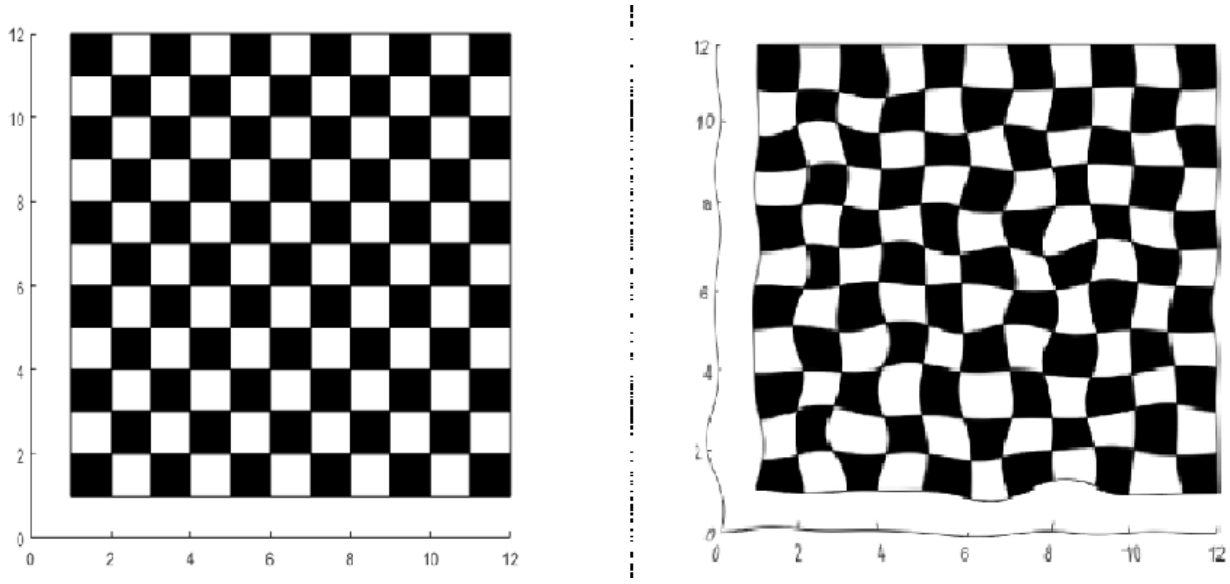


Fig. 8 - Control image and image with applied known warp

With a pure black and white image with well defined lines such as in Fig. 8, calculating the displacement of pixels is simplified because the gray value is binary, hence no ambiguities. To test robustly, more complex images, such as that in Fig. 9 will be used.

Fig. 9 - Gray scale control image and known warp

3 EQUIPMENT IDENTIFICATION

For this project, the hardware equipment (GPU and work station) were procured through the RMCC ECE department equipment stores. Visual Studio is part of the university's licensed software and is included with the default configuration of the workstation. The C++ libraries required, as mentioned in 2.1, are the Armadillo C++ linear algebra library and the OpenCV library. Both are available for free online. The CUDA Toolkit will provide the tools for interfacing with the GPU, handling the SOR parallelization. It is also available for free online. The equipment used in this project is summarized in Fig. 3.1.

TABLE: 3.1 - PARTS LIST

Part Description	Part/Version Number	Expected Arrival Date
Dell Precision T7400 Work Station	BWEWJ4X_6	26 Sep 17
NVIDIA Quadro K6000 GPU	VCQK6000-PB	26 Sep 17
Microsoft Visual Studio 2010	9.0.30319.1	26 Sep 17
Armadillo C++ Library	8.200.2	16 Nov 17
OpenCV	3.3.1	16 Nov 17
CUDA Toolkit	9.0.176	26 Nov 17

4 SCHEDULING

The project schedule is broken down into four phases:

a. Production of design documentation and analysis of algorithm

b. Serial C++ implementation of algorithm

c. Parallelization of program

d. Testing and evaluation

The first phase, and a portion of the second phase, will occur in the fall semester. A clear picture of the sequential C++ program's architecture along with a plan for implementation issues will be available by the time of the DID-05 presentations. The remaining phases are reserved for the winter semester, when the sequential program will be tested and debugged, and the parallel program will be interfaced with the GPU for operations to be run in parallel using CUDA. The schedule's main tasks are derived from a blend of these four phases and the document deadline schedule. An "hours to completion" strategy is used in which each subtask is given an estimated number of hours to complete. Tasks are aligned on the final submission date and progress back, each task scheduled from its respective deadline. Based on a number of working hours per week, completion dates can be estimated. Additional time can be allocated if tasks along the critical path do not meet the deliverable deadlines.

In Fig. 10, black bars represent groups of tasks, green diamond are deadlines, red bars are tasks on the critical path, and blue bars are other tasks. The dates to the right of the elements are the scheduled completion dates.

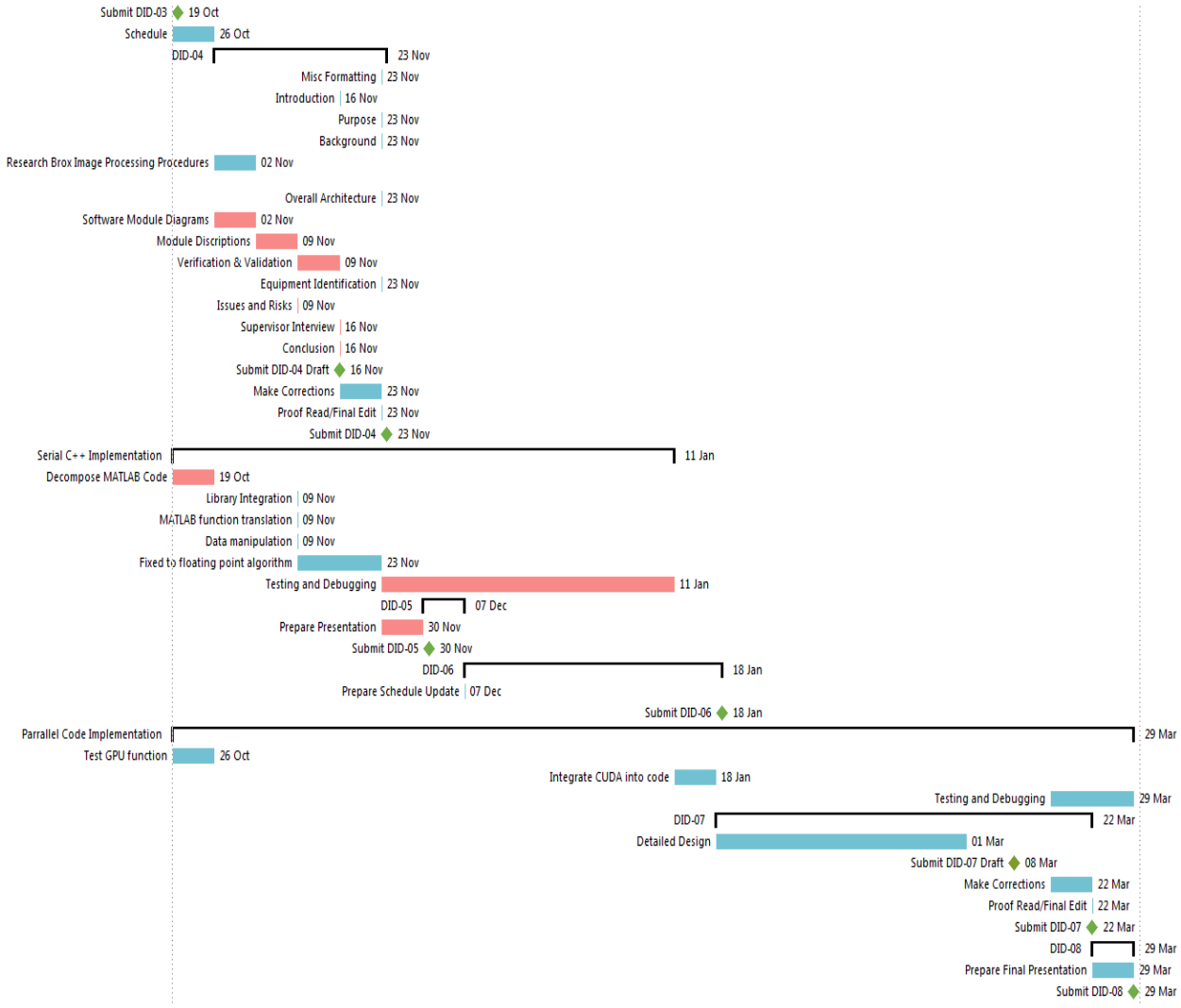


Fig. 10 - Gantt Chart of Preliminary Project Schedule

5 Unresolved Issues and Risks

This section explains some of the assumptions made at this phase in the design process, as well as the potential risks that could arise.

As required by [3], the error margin of the PSM must be less than 10% relative to a control. Due to the reduction of precision from using 16 bit fixed-point numbers, there is a risk of rounding errors compounding to produce a larger than acceptable error. This is mitigated in the hierarchical design described in section 2.5. To summarize, by testing each function before implementing it into its larger module, larger errors can be isolated to their respective function. In the scenario where a re-assessment of the function causing a large error yields no solution, it is likely that the number of bits used to represent numbers will need to be increased. An additional byte can be allocated to all matrix elements allowing for a Q4.19 notation; a resolution of $\frac{1}{131072}$.

This is not an ideal solution due to the resulting increase in data processing time. However, a PSM within a 10% margin of error is of greater priority than the processing time of the program, so long as it continues to meet its speed requirement.

The level of understanding of the Brox algorithm at this point in the project is assumed to be sufficient for design and implement a parallelization of this algorithm. At the time of the preliminary design, the current depth of knowledge is appropriate for the sequential C++ implementation, however, a greater understanding of the mathematics involved in the algorithm may lead to a state where the functionality to be parallelized must be changed. Take the red/black SOR method for example, although implemented on a GPU previously as seen [9], there is a possibility that this method is incompatible within the Brox algorithm. In this unlikely situation, other avenues for parallelism would need to be explored.

At this point in the project, it is assumed that the level of complexity of instructions send to the NVIDIA K6000 GPU are such that it can execute them without error. However, until the parallelization of the red/black SOR method is implemented, this cannot be verified. Modifications may need to be made to the parallel computation model in order to accommodate the limit of instruction complexity set by the GPU. This would involve the CPU performing the more complex calculations, matrix inversions for example, before sending lower level instructions to the GPU.

An important note provided by [9] is that the limiting factor of the computation speed of the red-black SOR method is the memory bandwidth capability of the GPU, not the computational throughput. Testing will reveal if that is also the case for this project given its discrepancy in hardware and software implementation from the research. It is anticipated that the use of fixed point numbers will significantly reduce the memory bandwidth when interfacing with the GPU, decreasing the likeliness of this issue. However, similar to the problem of complex calculations, modifications may need to be made to the design in order to reduce bandwidth to efficiently make use of the GPU's capability.

6 CONCLUSION

This document has outlined the logical procedure and architecture of the program, as well as the equipment, schedule, and risks of the project. It links to the next deliverable, the detailed design document, by providing the foundation of the project's design. This foundation comprises of the foundational decisions that we have made thus far, such as software modules to be implemented, handling of units to reduce the memory, and identification of parallelization opportunities within the Brox algorithm. In addition, the project schedule provides information on how tasks within the project will be accomplished including the creation and submission of the detailed design document.

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] A. Wedel and D. Cremers, *Stereo Scene Flow for 3D Motion Analysis*. London: Springer-Verlag, 2011.

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

[6] S. Mittal, "A study of successive over-relaxation (SOR) method parallelization over modern hpc languages," *International Journal of High Performance Computing and Networking*, vol. 7, pp. 292–298, 2014.

[7] 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.

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

[9] E. Konstantinidis and Y. Cotronis, "Accelerating the red/black SOR method using GPUs with CUDA," vol. 13, 2011, pp. 589–598.