AN FPGA IMPLEMENTATION OF
LARGE-SCALE IMAGE ORTHORECTIFICATION

Thesis
Submitted to
The School of Engineering of the
UNIVERSITY OF DAYTON

In Partial Fulfillment of the Requirements for
The Degree of
Master of Science in Electrical Engineering

By
Daniel Alan Shaffer
Dayton, Ohio
May, 2018
AN FPGA IMPLEMENTATION OF LARGE-SCALE IMAGE ORTHORECTIFICATION

Name: Shaffer, Daniel Alan

APPROVED BY:

__________________________________________  ______________________________________
Eric J. Balster, Ph.D.                        Frank A. Scarpino, Ph.D.
Advisor Committee Chairman                  Committee Member
Associate Professor, Electrical and        Professor Emeritus, Electrical and
Computer Engineering                        Computer Engineering

__________________________________________
Tarek M. Taha, Ph.D.
Committee Member
Professor, Electrical and Computer
Engineering

__________________________________________  ______________________________________
Robert J. Wilkens, Ph.D., P.E.                 Eddy M. Rojas, Ph.D., M.A., P.E.
Associate Dean for Research and Innovation    Dean, School of Engineering
Professor
School of Engineering
ABSTRACT

AN FPGA IMPLEMENTATION OF LARGE-SCALE IMAGE ORTHORECTIFICATION

Name: Shaffer, Daniel Alan
University of Dayton

Advisor: Dr. Eric J. Balster

This thesis presents a hardware implementation of an image orthorectification process using back-projection. Image orthorectification is integral to effective analysis and exploitation of aerial imagery and is often one of the largest processing bottlenecks. As imaging sensors grow in pixel count and associated target footprint, the image orthorectification process requires an associated increase in compute capability. In order to support size, weight, and power (SWaP) constrained processing environments, such as on-board systems for unmanned aerial vehicles (UAVs), efficient and scalable solutions must be developed. Moreover, in surveillance applications minimizing latency is paramount. This thesis presents an integer-based high performance Field Programmable Gate Array (FPGA) implementation of a back-projection algorithm for orthorectification. A 2.6x speedup is achieved over software processing with an associated 23x reduction in total energy per image.
To my parents, who got me started on this path.

And to Gayle, who gave me the motivation to finish it.
ACKNOWLEDGMENTS

I would like to thank Kerry Hill and the ADDA Facility at AFRL for their support in this effort.
TABLE OF CONTENTS

ABSTRACT .................................................................................. iii
DEDICATION ............................................................................... iv
ACKNOWLEDGMENTS ................................................................. v
LIST OF FIGURES ....................................................................... vii
LIST OF TABLES ......................................................................... ix
CHAPTER I. INTRODUCTION ...................................................... 1

CHAPTER II. BACK-PROJECTION ALGORITHM DESCRIPTION ........ 4

CHAPTER III. FIXED-POINT BACK-PROJECTION ALGORITHM DESCRIPTION . 10

CHAPTER IV. FPGA IMPLEMENTATION OF BACK-PROJECTION ......... 12
  4.1 Selection ............................................................................. 12
  4.2 Selection Painting Buffer ................................................... 15
  4.3 Painting .............................................................................. 16
    4.3.1 Painting Math Sub-Module ............................................. 17
  4.4 Caching System ................................................................. 19
    4.4.1 Min Max Sub-Module .................................................... 20
    4.4.2 Cache Filler Sub-Module ............................................... 22
    4.4.3 Quad Cache Sub-Module ............................................... 23
    4.4.4 Cache Translator Sub-Module ....................................... 23

CHAPTER V. RESULTS ............................................................... 26

CHAPTER VI. CONCLUSION ........................................................ 39

BIBLIOGRAPHY .......................................................................... 41
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Orthorectification Process</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Flight Pattern</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>Example DEM with Interpolated Pixels</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Image as Captured from Camera Source</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>Projection Geometry</td>
<td>6</td>
</tr>
<tr>
<td>2.4</td>
<td>Example of Combining Orthorectified Images</td>
<td>8</td>
</tr>
<tr>
<td>2.5</td>
<td>Example of Overlaying Orthorectified Image onto a Map</td>
<td>9</td>
</tr>
<tr>
<td>4.1</td>
<td>Back-Projection System Overview</td>
<td>13</td>
</tr>
<tr>
<td>4.2</td>
<td>Selection Module Data Flow</td>
<td>14</td>
</tr>
<tr>
<td>4.3</td>
<td>Selection Painting Buffer Module Data Flow</td>
<td>16</td>
</tr>
<tr>
<td>4.4</td>
<td>Painting Module Data Flow</td>
<td>17</td>
</tr>
<tr>
<td>4.5</td>
<td>Cache System Data Flow</td>
<td>19</td>
</tr>
<tr>
<td>4.6</td>
<td>Example Window with Proper Min/Max $i/j$ Called Out</td>
<td>21</td>
</tr>
<tr>
<td>4.7</td>
<td>64-Bit Aligned Cache Filler</td>
<td>22</td>
</tr>
<tr>
<td>4.8</td>
<td>DDR and Cache Storage Methodology</td>
<td>24</td>
</tr>
<tr>
<td>4.9</td>
<td>Cache Translator Example</td>
<td>25</td>
</tr>
<tr>
<td>5.1</td>
<td>Floating-Point Software Projected Image</td>
<td>29</td>
</tr>
<tr>
<td>5.2</td>
<td>Software and Hardware Integer Projected Images</td>
<td>29</td>
</tr>
<tr>
<td>5.3</td>
<td>Floating-Point and Integer Difference Image (Contrast and Brightness Enhanced)</td>
<td>30</td>
</tr>
</tbody>
</table>
LIST OF TABLES

3.1 Integer Variables and Scale Factors ........................................... 11
4.1 Camera Parameters for Selection Module ....................................... 13
4.2 Weights for Input Pixels ............................................................... 19
5.1 SWaP Results .............................................................................. 28
5.2 Integer Results Accuracy .............................................................. 28
5.3 Hardware Resource Usage on Stratix IV FPGA as Reported by Altera Compiler .................. 35
CHAPTER I
INTRODUCTION

As image sensors grow in pixel count and information density, it becomes increasingly practical to use them for more tasks. Comparing photographs taken of the same area over a range of time allows for the measurement of changes in traffic patterns, sand dune migration [1], forest disturbances [2], and many other purposes [3]. However, for multiple image sets taken over different time periods to align properly they must be coregistered and orthorectified [4–6]. Orthorectification is the process of North-orienting and aligning an image to an Earth coordinate system. Orthorectification is accomplished by using known information about the camera angle and using a Digital Elevation Map (DEM) to describe the local altitude of the terrain being photographed [7]. Orthorectification involves rotating and stretching an image as seen in Figure 1.1a and in Figure 1.1b. The amount of required stretching and rotation of the image depends on the position of the image capture vehicle.
in relation to the center of the image plane. Figure 1.2 provides a pictorial view of this relationship. One of the most common orthorectification techniques is back-projection, which works by projecting an Earth coordinate backwards into the image space [7].

While higher pixel densities have allowed images to be more useful, it also causes an increase in the required processing time for orthorectification. Several methods have been proposed to decrease the processing time of the back-projection algorithm. The use of general purpose graphical processing units (GPGPUs) has proven effective with wide area electro-optical imagery [8]. The development of an integer based back-projection algorithm [7] has led to an orthorectification method which works well with field programmable gate arrays (FPGA) which have lower SWaP constraints than GPGPUs and which also free up the CPU for other processing tasks.

This thesis describes an efficient implementation of an integer based back-projection algorithm using a Stratix-IV FPGA [9] on a GiDEL PROCe-IV PCIe board [10]. The FPGA back-projection
solution consists of three simultaneous instances of the algorithm per FPGA, resulting in a 2.6x improvement in speed and a 23x reduction in power over the original software based solution.

Following the Introduction, the back-projection algorithm is described in Chapter II. Chapter III describes how the basics of this algorithm are implemented in the FPGA, including a section with more details into a module designed to optimize memory access. Finally, the Results and Conclusion summarize how this algorithm fits on an actual FPGA.
CHAPTER II
BACK-PROJECTION ALGORITHM DESCRIPTION

The back-projection algorithm that is implemented in an FPGA is described in [7]. It is summarized here for completeness. A DEM is used as the output canvas for the projected image. Figure 2.1 shows the DEM with interpolated pixels. This figure shows an image overlaying the DEM containing two stick figures, Jim (on the left) and Bob (on the right) who are lying prone on the ground. As shown in Figure 2.1, the image has no distortion or warping. Accurate measurements can be made of the distance between Jim and Bob in the image, as well as the size of their various features.

Figure 2.1: Example DEM with Interpolated Pixels
The larger points in the DEM \((x, y)\) are the original DEM points, and the small pixels \((\chi, \gamma)\) are the bi-linearly interpolated positions. The altitude at each non-interpolated DEM point is referenced as \(Z(x, y)\). In this figure, an interpolated pixel location \((x', y')\) is highlighted which is distance \(\delta_E\) and \(\delta_N\) from the origin of the figure. The values of \(\delta_E\) and \(\delta_N\) are given by Equation 2.1,

\[
\delta_E = \frac{\Delta E}{I}, \quad \delta_N = \frac{\Delta N}{I}
\]

where \(\Delta E\) and \(\Delta N\) are the distance between non-interpolated DEM posts in the easting and northing directions, respectively, and \(I\) is the number of interpolation steps between DEM posts chosen to critically sample the DEM.

Figure 2.2 shows the image as captured from the camera source in the air. As shown in Figure
2.2, the image is distorted. With Figure 2.1 used as a reference, we can see that Jim and Bob are rotated about an unknown point in the image. Also, Jim and Bob’s heights are now different from what they were in Figure 2.1, and thus no longer the same size as each other. Figure 2.2 has a grid with an even sampling shown, but Jim and Bob are now located on different pixels from Figure 2.1. The pixels for this figure are referenced as $P(i,j)$, where $P()$ is the pixel value at each column $i$ and row $j$. The distortion of the images is due to the uneven sampling of the ground scene in the original image space. In this figure, we can see that the same point $(x',y')$ is highlighted as was called out in Figure 2.1. We observe that this point does not lie directly on the pixel grid with even sampling like it did in the previous figure. Instead, it lies between $(i,j)$, $(i+1,j)$, $(i,j+1)$, and $(i+1,j+1)$.

Figure 2.3 shows that the camera focal point $(X_c,Y_c,Z_c)$, the point $P(i,j)$, and the ground point $(X,Y,Z)$ are all collinear. The values of $i$ and $j$ are calculated using the collinearity equations shown in Equation 2.2 as seen in [7], where $f$ is the focal length of the camera, $m_{\bullet}$ are the elements of the homography matrix, and $X_c$, $Y_c$, and $Z_c$ are the longitude, latitude, and altitude of the focal
point of the camera.

\[ i = -f \frac{m_{11}(X-X_c)+m_{12}(Y-Y_c)+m_{13}(Z-Z_c)}{m_{31}(X-X_c)+m_{32}(Y-Y_c)+m_{33}(Z-Z_c)} \]  
\[ j = -f \frac{m_{21}(X-X_c)+m_{22}(Y-Y_c)+m_{23}(Z-Z_c)}{m_{31}(X-X_c)+m_{32}(Y-Y_c)+m_{33}(Z-Z_c)} \]  

These collinearity equations use the slope between each interpolated pixel in the DEM and the point-source location of the focal point of the imaging system to calculate the value of \((i, j)\). The homography matrix \(M\) is defined by Equation 2.3

\[
M = \begin{bmatrix}
m_{11} & m_{12} & m_{13} \\
m_{21} & m_{22} & m_{23} \\
m_{31} & m_{32} & m_{33}
\end{bmatrix} = R_\omega R_\phi R_\kappa
\]  

from [7]. \(R_\omega\), \(R_\phi\), and \(R_\kappa\) are the rotation matrices about the X-axis, Y-axis, and Z-axis, respectively, with \(R_\bullet\) being defined by Equation 2.4.

\[
R_\omega = \begin{bmatrix}
1 & 0 & 0 \\
0 & \cos(\omega) & \sin(\omega) \\
0 & -\sin(\omega) & \cos(\omega)
\end{bmatrix}
\]

\[
R_\phi = \begin{bmatrix}
\cos(\phi) & 0 & -\sin(\phi) \\
0 & 1 & 0 \\
\sin(\phi) & 0 & \cos(\phi)
\end{bmatrix}
\]  

\[
R_\kappa = \begin{bmatrix}
\cos(\kappa) & -\sin(\kappa) & 0 \\
\sin(\kappa) & \cos(\kappa) & 0 \\
0 & 0 & 1
\end{bmatrix}
\]  

If we use \(m'_\bullet = -fm_\bullet\) to replace the variables in Equation 2.2, we can simplify it to Equation 2.5.

\[ i = \frac{m'_{11}D_E+m'_{12}D_N+m'_{13}D_A}{m_{31}D_E+m_{32}D_N+m_{33}D_A} \]  
\[ j = \frac{m'_{21}D_E+m'_{22}D_N+m'_{23}D_A}{m_{31}D_E+m_{32}D_N+m_{33}D_A} \]  

Next we set the numerators of Equation 2.5 to equal \(i_n\) and \(j_n\) respectively and the denominator equal to \(r_d\), which allows for replacing Equation 2.5 with Equation 2.6

\[ i[x', y'] = \frac{i_n[x', y']}{r_d[x', y']} \]  
\[ j[x', y'] = \frac{j_n[x', y']}{r_d[x', y']} \]  

where the numerators \(i_n\) and \(j_n\) are described in Equation 2.7

\[ i_n[x', y'] = m'_{11}D_E[x'] + m'_{12}D_N[y'] + m'_{13}D_A[x', y'] \]  
\[ j_n[x', y'] = m'_{21}D_E[x'] + m'_{22}D_N[y'] + m'_{23}D_A[x', y'] \]
and the denominator in Equation 2.8 as done in [7].

\[ r_d[x', y'] = m_{31}D_E[x'] + m_{32}D_N[y'] + m_{33}D_A[x', y'] \] (2.8)

Figure 2.4 from [11] shows an example of how the image captured from the plane as seen in (a) must be rotated and warped as seen in (b) to properly orthorectify the image for the combination of several images into a single larger image (c). Figure 2.5 from [12] is an example of an image which has been back-projected being placed on to a map.
Figure 2.5: Example of Overlaying Orthorectified Image onto a Map
CHAPTER III

FIXED-POINT BACK-PROJECTION ALGORITHM DESCRIPTION

For increased computation speed, the back-projection algorithm from [7] is implemented using fixed-point integers rather than floating-point variables. The integerized version of the algorithm from [7] is summarized in the following section.

To preserve precision, each floating-point variable is multiplied by a constant scaling factor when converting it to fixed-precision. To allow for the use of bit shifting when performing the multiplication, the constant is chosen to be a power of 2 as shown in Equation 3.1

\[
\hat{F} = \lfloor 2^\lambda F \rfloor
\]  

(3.1)

where \( \lambda \) is the scale factor, \( F \) is the floating-point representation of a variable, and \( \hat{F} \) is the fixed-point representation of the same variable. Choosing too small of a scale factor \( \lambda \) will result in a loss of accuracy in the fixed-point result, while too large of a factor will result in overflow. The back-projection method implemented by [7] scales all of the variables used in the back-projection process by a factor of either \( \lambda_1 \) or \( \lambda_2 \). The variables used by the process are described in Table 3.1 which is taken from [7].

Based on analysis done in [7], the last row of the homography matrix (\( m_{3\bullet} \)) requires a larger scale-factor than the rest of the variables used in back-projection, so it has a unique scale factor of \( \lambda_2 \). In [7], three variables are used to quantify the distance from an interpolated point on the DEM grid to the focal point of the camera, \( \hat{D}_E \), \( \hat{D}_N \), and \( \hat{D}_A \). These values are calculated using Equation
Table 3.1: Integer Variables and Scale Factors

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Scale Factor</th>
<th>Integer Variable Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m_{11}'$, $m_{12}'$, $m_{13}'$</td>
<td>$\lambda_1$</td>
<td>$\hat{m}<em>{11}'$, $\hat{m}</em>{12}'$, $\hat{m}_{13}'$</td>
</tr>
<tr>
<td>$m_{21}'$, $m_{22}'$, $m_{23}'$</td>
<td>$\lambda_1$</td>
<td>$\hat{m}<em>{21}'$, $\hat{m}</em>{22}'$, $\hat{m}_{23}'$</td>
</tr>
<tr>
<td>$m_{31}'$, $m_{32}'$, $m_{33}'$</td>
<td>$\lambda_2$</td>
<td>$m_{31}$, $m_{32}$, $m_{33}$</td>
</tr>
<tr>
<td>$\delta_E$, $\delta_N$</td>
<td>$\lambda_1$</td>
<td>$\delta_E$, $\delta_N$</td>
</tr>
<tr>
<td>$X_c$, $Y_c$, $Z_c$</td>
<td>$\lambda_1$</td>
<td>$X_c$, $Y_c$, $Z_c$</td>
</tr>
<tr>
<td>$X_0$, $Y_0$</td>
<td>$\lambda_1$</td>
<td>$X_0$, $Y_0$</td>
</tr>
<tr>
<td>$\zeta[x', y']$</td>
<td>$\lambda_1$</td>
<td>$\zeta[x', y']$</td>
</tr>
</tbody>
</table>

3.2.

$$\hat{D}_E[x'] = X_0 + x'\delta_E - \hat{X}_c$$

$$\hat{D}_N[y'] = Y_0 + y'\delta_N - \hat{Y}_c$$

(3.2)

$$\hat{D}_A[x', y'] = \zeta[x', y'] - \hat{Z}_c$$

The integerized versions of Equation 2.7 and Equation 2.8 becomes that shown in Equation 3.3.

$$\hat{i}_n[x', y'] = \left[ \frac{\hat{m}_{11}'\hat{D}_E[x'] + \hat{m}_{12}'\hat{D}_N[y'] + \hat{m}_{13}'\hat{D}_A[x', y']}{2(\lambda_1 - \lambda_2)} \right]$$

$$\hat{j}_n[x', y'] = \left[ \frac{\hat{m}_{21}'\hat{D}_E[x'] + \hat{m}_{22}'\hat{D}_N[y'] + \hat{m}_{23}'\hat{D}_A[x', y']}{2(\lambda_1 - \lambda_2)} \right]$$

(3.3)

$$\hat{r}_d[x', y'] = \left[ \frac{\hat{m}_{31}'\hat{D}_E[x'] + \hat{m}_{32}'\hat{D}_N[y'] + \hat{m}_{33}'\hat{D}_A[x', y']}{2\lambda_2} \right]$$

The values for $\lambda_1$ and $\lambda_2$ in Equation 3.3 are scaled by [7] to avoid overflow and to achieve a high precision. Using the fixed-point integer approximation of $\hat{r}_d[x', y']$, the inversion of this variable is calculated by [7], leading to an integerized calculation for $i$ and $j$ shown by Equation 3.4

$$\hat{i}[x', y'] = \left[ \frac{\hat{i}_n[x', y']}{2^{\lambda_2}} \right]$$

(3.4)

$$\hat{j}[x', y'] = \left[ \frac{\hat{j}_n[x', y']}{2^{\lambda_2}} \right]$$

and Equation 3.5.

$$i[x', y'] \approx \left[ \frac{\hat{i}[x', y']}{2^{\lambda_2}} \right]$$

(3.5)

$$j[x', y'] \approx \left[ \frac{\hat{j}[x', y']}{2^{\lambda_2}} \right]$$
CHAPTER IV
FPGA IMPLEMENTATION OF BACK-PROJECTION

In the proposed implementation, the back-projection algorithm is initially split into two main processing modules, the selection module and the painting module. The selection module handles the interpolation of the DEM, the solving of the collinearity equation, and the calculation of the weights used for the four input pixels in bi-linear interpolation. The painting module is designed to retrieve the four input pixels which surround \( p(x', y') \) from memory, multiply them by their respective weights, and then sum them into one output pixel \( P(x', y') \). Due to the limitations of SDRAM memory controllers a simple random access into memory by the painting module to retrieve these four pixels incurs too large of a latency, and a new memory access method is necessary. Thus two extra modules are created, an on-chip cache to speedup memory access and an on-chip first-in first-out register array (FIFO) to hold the values coming out of the selection module while the cache is being filled. Figure 4.1 shows the data paths through the hardware system.

4.1 Selection

The selection module is at the beginning of the chain of hardware modules used to process an image and starts the processing by loading input data from software. Figure 4.2 shows a more detailed picture of the sub-modules in the Selection module. The module loads two main sets of data, a DEM for the area looking to be imaged, and a set of parameters based on the camera model used to capture the input image. The DEM is sized to contain the desired canvas size of the output image once scaled by the DEM interpolation factor. The input parameters are based on a model for
the camera which was used to capture the input images, and are calculated from such information as the angle the camera is at during image acquisition, which direction the camera is pointing, and the zoom of the camera. The camera model parameters are shown in Table 4.1.

Table 4.1: Camera Parameters for Selection Module

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Code Variable Name</th>
<th>Variable Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W_{m1}, H_{m1}$</td>
<td>$W_{m1}, H_{m1}$</td>
<td>Width and height of input image</td>
</tr>
<tr>
<td>$x_0, y_0$</td>
<td>$x_0, y_0$</td>
<td>Starting row and column indices in DEM</td>
</tr>
<tr>
<td>$x_{nd}, y_{nd}$</td>
<td>$x_{nd}, y_{nd}$</td>
<td>Ending row and column indices in DEM</td>
</tr>
<tr>
<td>$I_p$</td>
<td>$I_p$</td>
<td>Interpolation factor in DEM</td>
</tr>
<tr>
<td>$MinX, MinY$</td>
<td>$MinX, MinY$</td>
<td>Integerized location of first pixel in DEM grid</td>
</tr>
<tr>
<td>$gsd_0, gsd_1$</td>
<td>$gsd_0, gsd_1$</td>
<td>Integerized GSD in northing and easting directions</td>
</tr>
<tr>
<td>$X_c, Y_c, Z_c$</td>
<td>$iX_c, iY_c, iZ_c$</td>
<td>Integerized location of camera</td>
</tr>
<tr>
<td>$m_{1\ldots33}$</td>
<td>$im_{1\ldots33}$</td>
<td>Integerized homography matrix</td>
</tr>
</tbody>
</table>
Next the selection module begins interpolating points in the DEM and building an output canvas \( P() \) where the weighted pixels can be painted. This step begins by registering the index corners of the DEM \((x_0, y_0)\) and \((x_{nd}, y_{nd})\). A process begins counting in raster order across the \( x \) index starting at \( x_0 \) and going until it reaches \( x_{nd} \), at which point it increments its \( y \) index and resets the \( x \) index. Once it counts from \((x_0, y_0)\) to \((x_{nd}, y_{nd})\), it has finished processing an entire DEM and raises a ‘completed’ flag.

Each increment of the \( x \) index invokes the next interpolation of a set of points between \( x \) and \( x + 1 \), and between \( y \) and \( y + 1 \) with a step size of \( \delta_E \) and \( \delta_N \) respectively (see Equation 2.1). The final output image resolution is determined at this time by multiplying the number of steps between each \( x \) or \( y \) by the interpolation size \( Ip \).

Finally, the index addresses \((i, j)\) of the pixels in the original image \( p() \) required by each point \((x', y')\) in the output canvas \( P() \) are calculated as well as the weights for each input pixel. The outer loop module computes the initial weights and address \((i, j)\) which map to a DEM location.

Figure 4.2: Selection Module Data Flow
Next, the inner loop module computes the incremental weights and addresses which map to the DEM locations \((\chi, \gamma)\), between \((x, y)\) and \((x + 1, y)\).

### 4.2 Selection Painting Buffer

As the selection module begins writing out selected \((i, j)\) addresses and weights, they are read by the next module called the selection painting buffer which writes these values to a series of round-robin filled FIFOs. Figure 4.3 shows the data flow through the FIFOs in this module. As each 64 bit data word is written into the module, it is split into two separate data words of size 32 bit. The purpose of this is two-fold. First, a 32 bit interface is used to attempt to allow the data path to meet high clock speeds. Often when using larger data paths, Altera FPGAs will require a slower clock and will have a harder time meeting timing. Second, the data word is split to allow for easier reading when used in future modules since some later modules only use one portion of the input word for their unique calculations.

These FIFOs are designed to hold the maximum number of words which the selection module can output for a single DEM grid, so that an entire DEM can be held by a single FIFO. When a FIFO is full, the selection painting buffer automatically advances to the next empty FIFO to promote concurrent processing between the selection module and the painting module. The number of FIFOs is chosen to match the number of parallel caches in the cache module to ensure as little resource starving as possible.

For compatibility, this module is designed to use either the default Altera FIFO IP or to use a custom FIFO. The selection painting buffer module is able to handle a stall either when the input is waiting on data, or when the next module is unable to accept any more data. A conflict-prevent
process was created to help prevent an error when the internal FIFOs are either completely empty or completely full. It monitors the boundary conditions and handles them automatically.

![Diagram of Selection Painting Buffer Module Data Flow]

Figure 4.3: Selection Painting Buffer Module Data Flow

4.3 Painting

The painting module is made up of three sub-modules: a pixel requester, a delay module, and the painting math module. Figure 4.4 shows the data flow through these modules.

Because the selection module outputs just the upper-left pixel address \((i, j)\) required from the input image, the pixel requester is used to calculate the address of the other three addresses which surround the pixel \((x', y')\) to be mapped. These pixels are \((i + 1, j)\), \((i, j + 1)\), and \((i + 1, j + 1)\). The step from an address \(i\) to \(i + 1\) is just an addition of 1, but the step from \(j\) to \(j + 1\) is done by adding in the width of the input image in pixels. This value is currently hardcoded, but for a particular camera model the width of the input image is typically static, so this value can be passed into the hardware with the initial parameters at load time.
The delay module is a simple block which delays the pixel weights (coefficients) coming out of the selection painting buffer and going into the painting math module. This is done to allow the pixel requester enough clocks to retrieve the input pixel values from memory so that the weights and the pixels line up appropriately. Due to the limits of DDR random access speeds, the pixel requester submits an address to the cache system described later in this paper for the pixel values needed by the painting math sub-module rather than directly reading them out of memory itself.

4.3.1 Painting Math Sub-Module

The painting math module reads a group of four input pixel values from the pixel requester sub-module as well as the corresponding weights for the pixels from the delay module. Next the painting math sub-module performs a bi-linear interpolation on these four pixels using Equation 4.1 to output a single $P(X, Y)$ pixel.

$$
P(X, Y) = R_0 \cdot p(i, j) + R_1 \cdot p(i + 1, j) + R_2 \cdot p(i, j + 1) + R_3 \cdot p(i + 1, j + 1)
$$

Figure 4.4: Painting Module Data Flow
As seen in Equation 4.1, the value of $P(X, Y)$ is a sum of the four pixels surrounding $(x', y')$ multiplied by a weight factor. To compute the weights, bi-linear interpolation starts with Equation 4.2 as found in [13]

$$f(x, y) = \sum_{i=0}^{1} \sum_{j=0}^{1} a_{ij} x^i y^j = a_{00} + a_{10} x + a_{01} y + a_{11} xy$$  \hfill (4.2)$$

where the values for $a_{00} \ldots a_{11}$ are found using Equation 4.3

$$a_{00} = f(0, 0),$$
$$a_{10} = f(1, 0) - f(0, 0),$$
$$a_{01} = f(0, 1) - f(0, 0),$$
$$a_{11} = f(1, 1) + f(0, 0) - (f(1, 0) + f(0, 1)).$$  \hfill (4.3)$$

Equation 4.2 can be expanded to

$$f(x, y) = f(0, 0) + (f(1, 0) - f(0, 0)) x + (f(0, 1) - f(0, 0)) y + (f(1, 1) + f(0, 0) - (f(1, 0) + f(0, 1))) xy.$$  \hfill (4.4)$$

Replacing $f()$ with $p()$

$$f(0, 0) = p(i, j),$$
$$f(1, 0) = p(i + 1, j),$$
$$f(0, 1) = p(i, j + 1),$$
$$f(1, 1) = p(i + 1, j + 1)$$  \hfill (4.5)$$

and expanding the equations yields

$$P(X, Y) = p(i, j) + p(i + 1, j) x - p(i, j) x + p(i, j + 1) y - p(i, j) y + p(i + 1, j + 1) xy + p(i, j) xy - p(i + 1, j) xy - p(i, j + 1) xy.$$  \hfill (4.6)$$

Grouping the factors by $p()$ leads to

$$P(X, Y) = (1 - x - y + xy) \cdot p(i, j) + (x - xy) \cdot p(i + 1, j) + (y - xy) \cdot p(i, j + 1) + (xy) \cdot p(i + 1, j + 1).$$  \hfill (4.7)$$

Table 4.2 shows how the values of $R_{0, \ldots, 3}$ are replaced with $x$ and $y$. Since retrieving the value of the pixels surrounding $(x', y')$ from DDR memory was determined to be prohibitively slow, a faster method of retrieving these pixels was needed.
Table 4.2: Weights for Input Pixels

<table>
<thead>
<tr>
<th>R_{0...3}</th>
<th>Reference</th>
<th>Variable Name</th>
<th>Code Variable Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R_0</td>
<td>(1 - x - y + xy)</td>
<td>( F_{RAX _MULT} - F_{ix} - F_{iy}M_{ix}I_{iy} )</td>
</tr>
<tr>
<td></td>
<td>R_1</td>
<td>(x - xy)</td>
<td>( F_{ix} - I_{ix}I_{iy} )</td>
</tr>
<tr>
<td></td>
<td>R_2</td>
<td>(y - xy)</td>
<td>( F_{iy}M_{ix}I_{iy} )</td>
</tr>
<tr>
<td></td>
<td>R_3</td>
<td>(xy)</td>
<td>( I_{ix}I_{iy} )</td>
</tr>
</tbody>
</table>

4.4 Caching System

To increase the speed of the random accesses to DDR memory, the cache system module was designed and is composed of several key components. Figure 4.5 provides a detailed view of the data path through the cache system and its sub-modules. The first component in this system is the min/max calculator, which determines which input pixels will be needed by each DEM grid so that the cache can be filled with all the necessary input data for each grid. Next in the chain is the cache filler, which contains the interface to the DDR memory and which handles reading all of the
necessary data from external memory and passing it to the caches. The final module in the flow is the quad cache, which is actually a group of four BRAM modules which were combined to act as one cache. The cache translator is designed to act as an intermediary between the cache system and the painting system, and is used to translate physical memory address requests from the painting module into the relative addresses needed by the quad cache module. The cache system module which these are all sub-modules of handles communication between them.

4.4.1 Min Max Sub-Module

The purpose of the min/max module is to determine which area of memory containing the input image the caches will need to load in order to operate on each DEM grid in the output image. It is best to think of each image stored in memory as if it were a two-dimensional array. Since the Selection module operates on a single DEM grid at a time, we can draw a virtual box around a 2-D slice of memory from the input image which will be used to interpolate the output image. This 2-D array will change in size and shape as we work our way through different portions of the output image due to the nature of the image capture process. Since random access into memory to retrieve input image pixel values is relatively slow and since the size of the window of input memory constantly changes, we optimize the data retrieved dynamically from on-board DDR and we load it into on-chip M9K resources.

The min/max module processes a single pixel address request per clock as each request comes out of the Selection module. When the first address arrives, its relative location in the input image \((i, j)\) is calculated using Equation 4.8 and saved as both the minimum and maximum \(i\) as well as the minimum and maximum \(j\).

\[
\begin{align*}
i &= \text{Data}_{\text{in}} \mod W_{\text{image}} \\
j &= \left\lfloor \frac{\text{Data}_{\text{in}}}{W_{\text{image}}} \right\rfloor
\end{align*}
\]
When a second address arrives, it can now be used to build a virtual box which encompasses both requests. As each new address arrives, it either fits inside the current box or else causes the current box to be re-sized (grown).

Figure 4.6 shows a simulated window in memory with the input pixels which will be needed by the painting module highlighted. The minimum and maximum $i$ and $j$ values are called out to show the grid of input pixels which will be read into the cache. When the selection module is finished with a DEM grid, the final minimum and maximum $i$ and $j$ mark out the window of data which will need to be loaded into the DEM grid’s corresponding cache. The min/max module outputs a data complete signal when it is done reading a DEM grid to prepare the cache filler to load another cache.

Because the min/max module contains several divide functions, it was created with a configurable number of pipeline stages to allow it to meet various performance goals. The min/max module is able to achieve a throughput of one pixel per clock.
4.4.2 Cache Filler Sub-Module

The cache filler sub-module handles the sequential access interface to the off-chip DDR memory. When the filler is notified that it has new valid image window corners (from the min/max module), it tells a new cache to prepare for data and initializes the DDR interface. Then it resets the interface to the upper left corner address \((\text{min } i \text{ and min } j)\) and begins reading across the image until it hits the upper right corner, at which point it advances to the next row, resets its column index, and continues reading until it has read down to the bottom right corner \((\text{max } i \text{ and max } j)\). When finished loading a cache, the filler tells the cache that it is done and waits for the next valid image window corners.

To optimize DDR access, the cache filler reads in 64 bits of data at a time (eight 1-byte pixels). Because DDR access is more efficient when reading from 8-byte aligned addresses, the starting address is always chosen to be evenly divisible by 8 such that the windows of data retrieved from memory always contains all of the data needed to project a DEM grid, and possibly extra unused input data. Figure 4.7 shows an example of this windowed read. As can be seen in this

![Figure 4.7: 64-Bit Aligned Cache Filler](image-url)
figure, the addresses reported by the min/max module included [0x2,· · · ,0x6], [0x12,· · · ,0x16], and [0x22,· · · ,0x26]. However, the cache filler will include the addresses [0x0,· · · ,0x7], [0x10,· · · ,0x17], [0x20,· · · ,0x27], and [0x30,· · · ,0x37]. The inclusion of these extra addresses will result in little extra resource utilization, but will greatly increase the speed and efficiency of data retrieval from off-chip memory.

4.4.3 Quad Cache Sub-Module

Since every selected address \((i, j)\) needs a group of four pixels returned from the cache, it was determined that a good optimization was to create four caches, each of which holds one of the four pixels required for the painting module. When the cache filler feeds data into the cache to be loaded, the cache automatically sorts it into the correct cache based on the parity of the address the data came from. Data coming from an even row, even column is put into the first cache and data from an even row, odd column is put into the second cache. The third and fourth are filled in a similar way, using the odd row, even column and the odd row, odd column respectively. Figure 4.8a depicts the input image data as it is stored in DDR memory, while Figure 4.8b depicts the data as it is stored in the four sub-caches (A, B, C, and D). When the cache is read, it outputs one pixel from each sub-cache in order to get a grouping of four pixels. Using four separate caches allows for more efficient writes and reads from the cache module and further increases the system’s speed.

4.4.4 Cache Translator Sub-Module

The cache translator is a relatively simple module which translates the address from the painting module’s input image space into the quad cache’s memory space. Equation 4.9 is used to map pixel addresses from the input image space to the cache memory space.

\[
f(A_{in}) = A_{cache} = K_1 + K_2 \quad (4.9)
\]
where

\[ K_1 = \left\lfloor \frac{(A_{in} \% W_{image}) - (A_{start} \% W_{image})}{2} \right\rfloor \]  \hspace{1cm} (4.10)

and

\[ K_2 = \frac{W_{cache}}{2} \cdot \left\lfloor \frac{A_{in} \% W_{image}}{2} - \frac{A_{start} \% W_{image}}{2} \right\rfloor \]  \hspace{1cm} (4.11)

It reads in an address from the painting module and then uses the starting address of the cache as well as the width of the cache to calculate the translated address in the cache schema. Figure 4.9 shows an example of an input address being translated into the quad cache’s data space.
(a) Input Pixels $p(i, j)$ as Stored in DDR  

(b) Input Pixels $p(i, j)$ as Stored in Four BRAMs

Figure 4.9: Cache Translator Example
CHAPTER V

RESULTS

All of the tests performed for this work were done on a Dell 2950 server with dual quad core Intel Xeon X5460 processors running at 3.16Ghz and with 32GB of DDR2 RAM. The VHDL based algorithm was tested on a Stratix-IV 530 FPGA with 4.5GB of on-board DDR2 RAM which resides in the Dell server and communicates with software over PCIe. All of the images used in the following tests are part of a data collect taken over Wright Patterson Air Force Base by a camera which produces frames of a resolution of 9000x9000 pixels.

The original back-projection floating-point and fixed-point algorithms referenced by [7] were initially tested with a simple C++ software testbench which projects a single hard-coded frame per run. This code base was run against two images referenced throughout this section as Frame A (13,122x10,287 pixels) and Frame B (13,847x10,614 pixels), and was used to test the software floating-point algorithm, the software fixed-point algorithm, and the hardware single core fixed-point algorithm. A modified version of this testbench targeted a three core version of the hardware fixed-point algorithm by sending three copies of the same input image to the FPGA in parallel, initiating a test when they were all transferred, and then verifying the results of the test when all three cores completed processing. The three core test was performed to verify functionality, but was not used for timing results once it was shown that the transfer times for hardware were less than the processing time, and thus could be hidden by overlapping transfers of the images to the board. The three hardware cores are truly independent, allowing for staggered offset times. A final processing
time for the three core version was estimated by dividing the single core run times by three since the three cores are independent.

Next the software and hardware fixed-point back-projection processes were tested with a more optimized version of the testbench still written in C++ and compiled with g++. The floating-point code was not integrated into this testbench due to compatibility issues and the initial goals of the funding source for this work. The processing time for the three core version of this code was again estimated by dividing the single core run times by three. This testbench allows for a series of frames to be processed per run with a sample set of 271 input images which rotate $360^\circ$ around an area of ground through a series of angles. The average projected output resolution of these images was 15,053x11,686 pixels, or approximately 177 million pixels per image.

As can be seen in Table 5.1, for the optimized testbench the fixed-point integer software projected these images with an average time of 1.43 seconds per image while the single core hardware projected these images with an average time of 1.63 seconds per image. The floating-point code from [7] was tested with two of these input frames, but was not fully integrated into the larger testing system. The average time for the floating-point software with these two images was 3.69 seconds. With three instances of this algorithm on the same FPGA card, the hardware can process three images concurrently for a per frame processing time of 0.54 seconds.

The power draw for the software code was pulled from the power specifications for the CPU in the test server. Since offloading processing from the software to the hardware frees up the CPU to perform other tasks while the hardware is processing, the single and triple core hardware power was calculated by measuring the power draw of the FPGA card in the host computer during processing. This process is described in more detail later in this chapter.
Table 5.1: SWaP Results

<table>
<thead>
<tr>
<th></th>
<th>SW (FP)</th>
<th>SW (Int)</th>
<th>HW (1 Core Int)</th>
<th>HW (3 Core Int)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run Time Frame A (Sec)</td>
<td>3.63</td>
<td>2.99</td>
<td>2.82</td>
<td>0.94</td>
</tr>
<tr>
<td>Run Time Frame B (Sec)</td>
<td>3.76</td>
<td>3.21</td>
<td>3.28</td>
<td>1.09</td>
</tr>
<tr>
<td>Run Time (271 Frames)(Sec)</td>
<td>n/a**</td>
<td>1.4339</td>
<td>1.6285</td>
<td>0.5428</td>
</tr>
<tr>
<td>Power (W)</td>
<td>120</td>
<td>120</td>
<td>&lt;15</td>
<td>&lt;19</td>
</tr>
<tr>
<td>Average Energy per Frame (J)</td>
<td>443.4</td>
<td>372.0</td>
<td>45.8</td>
<td>19.3</td>
</tr>
</tbody>
</table>

* The hardware code for the 3 core system was proved functional but not integrated into the main processing chain. Thus the speeds were estimated based on initial performance.
** The floating-point software was not integrated into the main processing chain.
*** Average energy calculation computed using Frame A and Frame B for consistency.

The images produced by the hardware and software integer code match each other byte-for-byte, but are not a perfect match to the floating-point software. Table 5.2 shows the accuracy of the integer based algorithm in comparison to the floating-point code for two representative images. As can be seen in this table, the algorithms produce extremely similar results. Figures 5.1 and 5.2 show the output images for Frame B, and no visual difference between the figures are apparent. Figure 5.3 shows a contrast enhanced difference image for the software floating-point code and the hardware fixed-point integer code. Even contrast enhanced, this difference image is almost completely black, indicating that the images are extremely similar.

Table 5.2: Integer Results Accuracy

<table>
<thead>
<tr>
<th></th>
<th>MAE</th>
<th>MSE</th>
<th>PSNR</th>
</tr>
</thead>
<tbody>
<tr>
<td>HW Int to SW Int</td>
<td>0</td>
<td>0</td>
<td>∞</td>
</tr>
<tr>
<td>HW Int to SW FP - Frame A (13122x10287)</td>
<td>0.03503</td>
<td>0.14602</td>
<td>130.06544</td>
</tr>
<tr>
<td>HW Int to SW FP - Frame B (13847x10614)</td>
<td>0.02422</td>
<td>0.10667</td>
<td>133.20570</td>
</tr>
</tbody>
</table>
Figure 5.1: Floating-Point Software Projected Image

Figure 5.2: Software and Hardware Integer Projected Images

(a) Fixed-Point Integer Software Projected Image
(b) Fixed-Point Integer Hardware Projected Image
Figure 5.4 shows the time to project the series of 271 input images using integer based hardware and software with a single back-projection core in the hardware. For low pixel counts, the hardware processes images at an equivalent time to the software. As the pixel count increases, however, the software outperforms the hardware. Figure 5.5 shows an estimate for the time to project the
271 input images using three cores of back-projection in the hardware. Because the three core version was only able to be partially integrated into the testbench, and since the three cores are fully independent of each other, the three core processing times are estimated by dividing the measured times of the single core hardware version by three. As can be seen in this figure, using three cores allows the hardware to easily outperform the software. It can be seen from these figures that the hardware times are much more consistent than the software times.

Figure 5.6a shows a graph of the output frame diagonal size in pixels verse a normalized time to project the image. The time for this graph is normalized by dividing by the length of the diagonal of the image. Figure 5.6b shows a graph of the output frame pixel count verse a normalized time to project the image. The time for this graph is normalized by dividing by the number of pixels in the output image. These figures show that the projection time for the integer based software is independent of the diagonal length of the output image size (Figure 5.6a), while the projection time for the integer hardware is independent of the pixel count in the output image (Figure 5.6b).
As can be seen in Figure 5.7, when using three hardware cores the processing time per frame is decreased from the original software floating-point algorithm by a factor of 3.6x. Figure 5.8 shows a decrease in energy usage of 23x achieved by the three core integer based hardware over
the original floating-point code. The three core hardware uses less overall energy than the hardware with a single core because while the average power draw per core is not linear, the processing time per core is. Thus increasing the core count per FPGA provides an overall decrease in energy usage per core as well as a consistent processing capability per core.

![Average Energy Required to Process Single Frame](image)

**Figure 5.8: Per Frame Energy Comparison**

As mentioned earlier in this section, the integer based algorithm was tested by using input images which rotate around a fixed-point on the ground. Figure 5.9 shows the cyclic nature of this testing. As the input image rotates to cover an area less ‘parallel’ to the lines of latitude and longitude, the diagonal size of the output image increases. It can be seen in Figure 5.9 that the processing time for integer hardware and software is correspondingly cyclic, but that the hardware has larger and smoother swings than software, indicating a tighter correlation for the hardware and the diagonal size of the output image. As the diagonal size of the output frame decreases, a single core of hardware is able to achieve a faster projection time than software fixed-point. However, as the diagonal size increases the single core hardware becomes slower than software.
Figure 5.9: Image Diagonal Size and Processing Time for Hardware and Software Plotted by Sample Number

Figure 5.10 shows the projection times for the integer based software and hardware (single and triple core) with a best fit line shown for each. The hardware cores fit much tighter to their best fit line, again emphasizing the consistency of the hardware.

Figure 5.10: Projection Time with Best Fit Line
Table 5.3 shows the resource usage reported by the Altera compiler of the single core hardware design, which used a small enough percentage of the hardware that more cores could be fit into the current Stratix IV FPGA. The report for the three core version shows that the ALM resources would not support any additional cores. After growing the system, it was determined that three cores was the optimal layout for this chip as it matched the resources in the chip as well as the three separate memory banks located on the GiDEL [10] FPGA board used as a test platform. A larger chip could handle an increased amount of cores but would require a modification of the memory interface. While a three core version was shown to be functional and consistent with the processing performance of the single core, increasing the core count beyond three was not tested and would require further analysis to prove it does not decrease the processing speed of the algorithm or significantly change the power profile of the system.

To calculate the power draw of the PCIe based FPGA card, a PCIe bus isolation device is used. This allows the power for the card to be supplied by a separate power supply unit, and verifies that no power is supplied by the host system. Using this method, an accurate power consumption for the entire FPGA card can be computed. Figure 5.11 shows a graph of the power draw of the single core

Table 5.3: Hardware Resource Usage on Stratix IV FPGA as Reported by Altera Compiler

<table>
<thead>
<tr>
<th></th>
<th>1 Core</th>
<th>3 Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic utilization</td>
<td>23%</td>
<td>76%</td>
</tr>
<tr>
<td>ALMs utilization</td>
<td>29%</td>
<td>88%</td>
</tr>
<tr>
<td>M9K blocks</td>
<td>10%</td>
<td>24%</td>
</tr>
<tr>
<td>M144K blocks</td>
<td>23%</td>
<td>70%</td>
</tr>
<tr>
<td>DSP blocks (18-bit)</td>
<td>29%</td>
<td>52%</td>
</tr>
</tbody>
</table>
hardware version during the projection of two images. As seen in this figure, the FPGA has a power
draw of approximately 9W before a binary file is loaded onto the board. After programming, the
power draw increases to 13W. When the image files are transferred to the board, the draw jumps to
almost 14.6W, followed by a drop to approximately 14.2W during processing. Figure 5.12 shows
the power draw for the initial three core version of the hardware processing a total of three frames
(the three core test used the same frame sent to the board with three concurrent instances). It can
be seen from this graph that the board draws a higher power than the single core during processing
(peak of 19W), but that the processing times can be overlapped for frames since the data transfer
times are smaller than the processing times.

Figure 5.13 shows an annotated version of the single and triple core hardware systems with the
power draw indicated on the $Y$ axis. The first steady-state value for both single and triple cores is
approximately 9W while the board is in reset. The next steady-state is approximately 13W for the
single core and 16W for the triple core. At this point in time, the boards have been programmed with
a binary file but are in an idle state and are not performing any processing, thus this is considered a baseline static power loss for each system. Next, the boards begin loading input data and their power draw jumps as they are performing more DDR and PCIe transfers. Once this is finished, the software sends a signal for them to begin processing. As seen in Figure 5.13, the processing power for these systems is approximately 14W for the single core, and 19W for the triple core. This is what I consider the baseline dynamic processing state. If we subtract the static baseline from the dynamic baseline for both systems, we end up with a power loss due to processing of approximately 1W for the single core, and 3W for the triple core. Thus to perform three times the processing requires three times the dynamic power draw, but less than three times the power overall leading to an overall efficiency increase of the triple core system.

Figure 5.12: Power Draw of Triple Core Back-Projection for Three Concurrent Frames
Figure 5.13: Annotated Power Graph for Single and Triple Cores
CHAPTER VI
CONCLUSION

As images increase in size, it becomes more important to be able to orthorectify and otherwise manipulate them rapidly and with lower power. In this thesis I showed that by using the integer based back-projection proposed by [7] in an FPGA, a lower SWaP could be achieved over earlier fixed-point algorithms. Also, the figures produced by this fixed-point algorithm have a PSNR of approximately 130, showing they are extremely similar to the floating-point code.

By splitting this fixed-point design into various components in hardware, optimizations were able to be made which significantly improved performance over the software based algorithms. Creating the cache module proved to provide a significant speedup and lowered the bottleneck of the random accesses into memory required by this algorithm. Overall, the three core fixed-point hardware system reached a peak speed of approximately 1.02 seconds per frame verse the earlier 3.70 seconds per frame for software floating-point. When combined with the lower power of the hardware system, the FPGA achieved a performance increase of almost 23x the floating-point software design when comparing total per frame energy usage.

There are several future areas of research with this project which would be interesting to pursue. Intel is releasing a new Stratix 10 line which includes a variant with high speed memory modules attached directly to the transceivers of the FPGA. Porting the project to this new chip should provide incredible memory access time improvements in addition to the significantly larger capacity of these chips and the higher potential clock rates of the new fabric. Furthermore, an interesting test would be to vary the precision of the fixed-point variables in hardware and see how this affects the accuracy.
of the resulting image as compared to the resource utilization change in the FPGA. Earlier tests by [7] were performed in the software version of this algorithm, and the FPGA based tests would be an interesting follow-up to this research.
BIBLIOGRAPHY


