An Hardware Efficient Normalized Cross Correlation FPGA Coprocessor

18-545 Fall 2013
An Astroteam Production

Wennie Tabib — Gun Charnmanee — Dylan Koenig
wtabib@andrew.cmu.edu — gcharnma@andrew.cmu.edu — djkoenig@andrew.cmu.edu

December 10, 2013

Contents

1 Introduction

The goal of this project was to develop a Field Programmable Gate Array (FPGA) co-processor to accelerate the computationally expensive task of feature matching in a visual odometry algorithm. Visual odometry is a method of computing a robot or vehicle's pose (position and orientation) in 3D space relative to a fixed starting position. Visual odometry is not a method absolute positioning. Thus, it cannot tell a robot's position on Earth. Instead it outputs the robot's position relative to its position when the code began executing.

Visual odometry has been used on many robots with varying degrees of success. For example, visual odometry was deployed on the winning DARPA Grand Challenge robot *Stanley*, developed by Sebastian Thrun's Stanford team [5].

A visual odometry system was also developed for the Mars Exploration Rovers and Mars Science Laboratory Rover to determine the position and orientation of the robots on the Martian surface. The software was deployed to IBM RAD6000s and RAD750 radiation-hardened single board computers based on the IBM RISC Single Chip CPU. While these chips are tolerant to radiation, they suffer from lack of computational power and impede the progress of space exploration. Indeed, the state-of-practice optical navigation technologies in space process a pair of low resolution images in 2-3 minutes and Martian rover traverses top out at around 10 meters/hour [1].

Space-rated computing systems lag the performance of their terrestrial counterparts, and computation-intensive operations in space are performed slowly or altogether avoided. One way to address this problem is by offloading computationally expensive vision algorithms to radiation-hardened FPGAs. Xilinx has developed a radiation tolerant Virtex-5QV chip that could be used for hyper-intensive computing in space. The aim of this project was to develop normalized cross correlation on FPGA for feature matching in visual odometry algorithms. This feature matching could also be used for other computer vision applications such as map registration, hazard detection, and object recognition.

Greater computational power enables the deployment of rovers and spacecraft with increased autonomy and facility, leading to the development of smarter landers capable of navigating to safe touchdown in hazard strewn terrain, rovers capable of high velocity navigation limited only by physical laws, and satellites capable of expanding their critical roles in communication, navigation and observation.

2 Overview

The visual odometry algorithm proceeds in the following fashion. First, a left image is loaded from either a camera or memory. Features in the image are detected using a feature detection algorithm (Figure 2b). While many feature detection algorithms exist for this purpose, this project implemented a harris corner detector in C++ due to its speed and robustness. Once a list of features has been detected for the left image, the points are suppressed based on corner strength and proximity

to other corners. 80×80 pixel patches are then extracted around the (x, y) feature points (Figure 2c). These patches are downsamples by a factor of 5 to 16×16. Next, the right image is loaded and 400×400 pixel patches are extracted around the corresponding (x, y) location in the right image. These 400×400 patches are downsampled to 80×80. A normalized cross correlation is performed to match the 16×16 window with its corresponding location in the 80×80 window in the right image (Figure 2d). Once these features are matched, the same algorithm is performed with the next left image and the resulting feature matches can be used to compute an estimate of the motion from current image frame to the image frame.

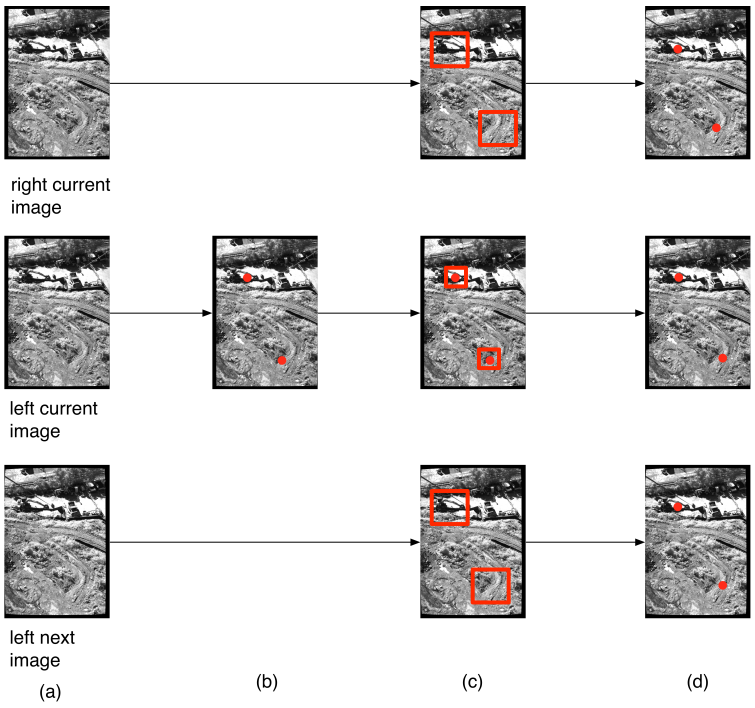


Figure 1: Overview of the System

3 Algorithm

The following sections detail the mathematical foundations of the harris corner detector, normalized cross correlation, and motion estimation algorithms.

Feature Detection
A harris corner detector was implemented as the feature detection algorithm. The harris corner detector considers a an image patch over the area (u,v) and shifts it over by (x,y). The weighted *sum of squared differences (SSD)* between these two patches, denoted S, is given by:

$$S(x, y) = \sum_u \sum_v w(u, v)(I(u + x, v + y) - I(u, v))^2$$

I(u+x, v+y) is approximated with a Taylor expansion. I_x and I_y are the partial derivatives of I such that

$$I(u+x, v+y) \approx I(u,v) + I_x(u,v)x + I_y(u,v)y$$

which produces the approximation

$$S(x,y) \approx \sum_u \sum_v w(u,v)(I_x(u,v)x + I_y(u,v)y)^2$$

which can be written in matrix form:

$$S(x,y) \approx \begin{pmatrix} x & y \end{pmatrix} A \begin{pmatrix} x \\ y \end{pmatrix}$$

where A is the structure tensor:

$$A = \sum_u \sum_v w(u,v) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} = \begin{bmatrix} \langle I_x^2 \rangle & \langle I_x I_y \rangle \\ \langle I_x I_y \rangle & \langle I_y^2 \rangle \end{bmatrix}$$

This matrix is a Harris matrix, and angle brackets denote averaging, or summation over (u, v). A corner is characterized by a large variation of S in all directions of the vector $\begin{pmatrix} x & y \end{pmatrix}$. The *determinant* and *trace* of A are computed to find corners. The following function computes a score of the strength of a point as a potential corner:

$$M_c = det(A) - \kappa trace^2(A)$$

κ must be set empirically, but the literature reports values in the range of 0.04 - 0.15 as suggested values.

Normalized Cross Correlation
This project referenced [6] and [7] for implementing the normalized cross correlation.

A normalized cross correlation is a standard approach to feature matching. It is motivated by the distance measure (squared Euclidean distance):

$$e_{w,d}^2(u,v) = \sum_{x,y} [w(x,y) - d(x-u, y-v)]^2$$

where w is the image and the sum is over x, y under the patch containing the descriptor d positioned at u, v. In the expansion of e^2

$$e_{w,d}^2(u,v) = \sum_{x,y} [w^2(x,y) - 2w(x,y)d(x-u, y-v) + d^2(x-u, y-v)]$$

the term $\sum d^2(x-u, y-v)$ is constant. If the term $\sum w^2(x,y)$ is approximately constant then the remaining cross-correlation term

$$c(u,v) = \sum_{x,y} w(x,y)d(x-u, y-v) \quad (1)$$

is a measure of the similarity between the image and the descriptor. There are several disadvantages to using (1) for template matching:

1. If the image energy $\sum w^2(x,y)$ varies with position, matching can fail. For example, the correlation between the descriptor and an exactly matching region in the image may be less than the correlation between the descriptor and a bright spot.
2. The range of $c(u,v)$ is dependent on the size of the descriptor
3. Eq. (1) is not invariant to changes in image amplitude such as those caused by changing lighting conditions across the image sequence.

The *correlation coefficient* overcomes these difficulties by normalizing the image and descriptor vectors to unit length yielding a cosine-like correlation coefficient

$$\frac{\sum_{x,y}[w(x,y) - \bar{w}_{u,v}][d(x-u, y-v) - \bar{d}]}{\left\{\sum_{x,y}[w(x,y) - \bar{w}_{u,v}]^2 \sum_{x,y}[d(x-u, y-v) - \bar{d}]^2\right\}^{0.5}} \quad (2)$$

where $\bar{d}$ is the mean of the descriptor and $\bar{w}_{u,v}$ is the mean of $w(x,y)$ in the region under the descriptor. Equation 2 is referred to as the *normalized cross correlation*.

Motion Estimation

The motion estimation algorithm is a flavor of the Perspective N Points algorithm for computing motion. Specifically, the algorithm is called Perspective-Three-Point (P3P) [4].

The Perspective-n-Point (PnP) problem originated from camera calibration. It retrieves the position and orientation of the camera with respect to a scene object from n corresponding 3D points. P3P is the particular case of PnP for $n = 3$. The P3P is the smallest subset of control points that yields a finite number of solutions. When the intrinsic camera parameters are known and we have $n \geq 4$ points the solution is generally unique.

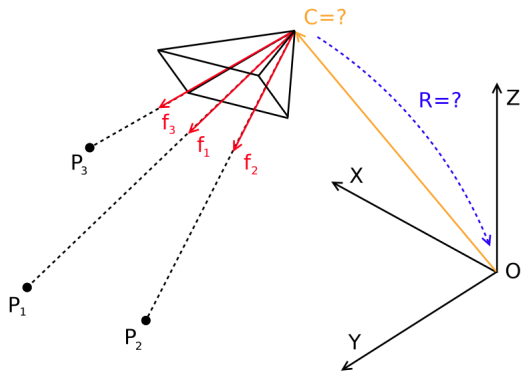


Figure 2: Synopsis of the problem

The goal of the P3P algorithm is to find the exact position of the camera center C and orientation matrix R of a camera with respect to the world frame (O,X,Y,Z), under the condition that the absolute spatial coordinates of three observed feature points P_1, P_2, and P_3 are given [4]. In order to compute this succesfully, the intrinsic camera parameters must be known. With this knowledge, the unitary vectors $\vec{f_1}, \vec{f_2}$, and $\vec{f_3}$–pointing towards the three considered feature points from the camera–can be derived [4].

Let the original camera frame be denoted by v. First, a new, intermediate camera frame τ from the feature vectors $\vec{f_1}$ and $\vec{f_2}$ inside v [4]. As shown in Fig. 3 the new camera frame is defined as $\tau = (C, \vec{t}_x, \vec{t}_y, \vec{t}_z)$, where $\tau = (C, \vec{t}_x, \vec{t}_y, \vec{t}_z)$ and the intermediate world frame $\eta = (P_1, \vec{n}_x, \vec{n}_y, \vec{n}_z)$ [4].

Using the transformation matrix $T = [\vec{t}_x, \vec{t}_y, \vec{t}_z]^T$, feature vectors can be transformed into τ using

$$\vec{f_i^\tau} \quad = \quad T \cdot \vec{f_i}$$

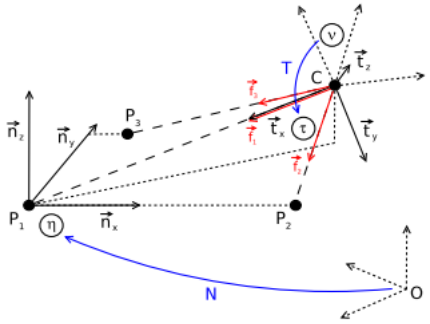


Figure 3: Illustration of the intermediate camera frame τ = (C, $\vec{t}_x\vec{t}_y\vec{t}_z$) and the intermediate world frame $\eta = P_1, \vec{n}_x\vec{n}_y\vec{n}_z$

Next the transformation matrix N and the world point P_3^η must be computed using

$$P_i^\eta \quad = \quad N \cdot (P_i - P_1)$$

After P_3^η is computed, the points $p1$ and $p2$ are extracted from P_3^η. Next d_{12} and b are computed using

$$\begin{aligned} b &= cot\beta \\ &= \pm\sqrt{\frac{1}{1 - cos^2\beta} - 1} \\ &= \pm\sqrt{\frac{1}{1 - (\vec{f_1} \cdot \vec{f_2})^2} - 1} \end{aligned}$$

ϕ_1 and ϕ_2 are computed using

$$\phi_1 \quad = \quad \frac{f_{3,x}^\tau}{f_{3,z}^\tau} and \phi_2 = \frac{f_{3,y}^\tau}{f_{3,z}^\tau} \tag{3}$$

The factors $a_4, a_3, a_2, a_1, and a_0$ are computed from polynomial

$$a_4 \cdot cos^4\theta + a_3 \cdot cos^3\theta + a_2 \cdot cos^2\theta + a_1 \cdot cos\theta + a_0 \quad = \quad 0$$

The real roots of the polynomial, or values for $cos\theta$ must be found and for each solution, the values for $cot\alpha$ must be found using

$$\left\{ \begin{array}{ll} \phi_1 = & P^{\tau}_{3,x} \\ \phi_2 = & P^{\tau}_{3,y} \end{array} \right\}$$

$$\leftrightarrow \left\{ \begin{array}{ll} \phi_1 = & \frac{-cos\alpha \cdot p_1 - sin\alpha cos\theta \cdot p_2 + d_{12}(sin\alpha \cdot b + cos\alpha)}{-sin\alpha \cdot p_2} \\ \phi_2 = & \frac{sin\alpha \cdot p_1 - cosalphacos\theta \cdot p_2}{-sin\theta \cdot p_2} \end{array} \right\}$$

$$\Longrightarrow cot\alpha = \frac{\frac{\phi_1}{\phi_2} p_1 + cos\theta \cdot p_2 - d_{12} \cdot b}{\frac{\phi_1}{\phi_2} cos\theta \cdot p_2 - p_1 + d_{12}}$$

All the necessary trigonometric forms of α and θ must be found using trigonometric relationships and the rstricted parameter domains. For each solution, C^{η} and Q must be computed using the following formulas:

$$\begin{array}{rcl} C^{\eta}(\alpha, \theta) & = & R_{\theta} \cdot C^{\Pi} \\ & = & \begin{pmatrix} d_{12} cos\alpha (sin\alpha \cdot b + cos\alpha) \\ d_{12} sin\alpha cos\theta (sin\alpha \cdot b + cos\alpha) \\ d_{12} sin\alpha sin\theta (sin\alpha \cdot b + cos\alpha) \end{pmatrix} \\ Q(\alpha, \theta) & = & [R_{\theta} \cdot (\vec{t}^{\Pi}_x \vec{t}^{\Pi}_y \vec{t}^{\Pi}_z)]^T \\ & = & \begin{pmatrix} -cos\alpha & -sin\alpha cos\theta & -sin\alpha sin\theta \\ sin\alpha & -cos\alpha cos\theta & -cos\alpha sin\theta \\ 0 & -sin\theta & cos\theta \end{pmatrix} \end{array}$$

For each solution the absolute camera center C and orientation R must be computed using the following two equations

$$\begin{array}{rcl} C & = & P_1 + N^T \cdot C^{\eta} \\ C & = & N^T \cdot Q^T \cdot T \end{array}$$

Finally, a fourth point is backprojected for disambiguation.

4 Hardware

4.1 Structural Description

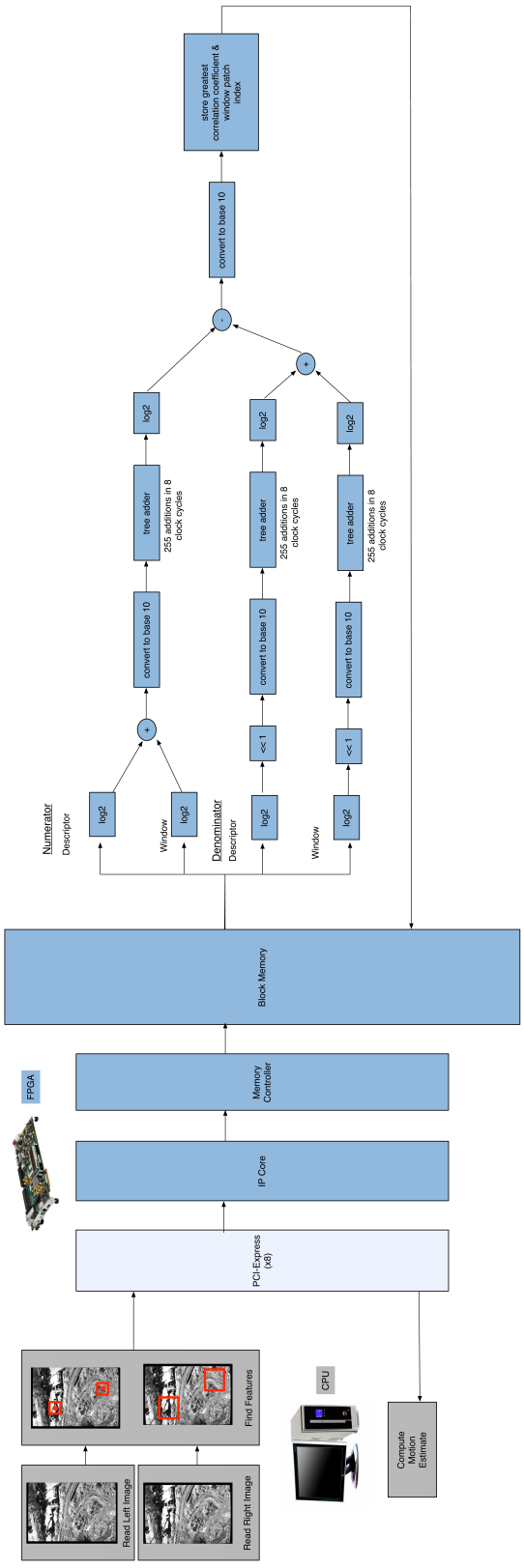


Figure 4: Structural Diagram

This diagram gives an overview of the general architecture of the system, from images being capture, to image preprocessing on the CPU, to data transfer using PCI-E, to NCC on the FPGA, and back to the CPU with the resulting correlation coefficient and match location.

4.2 Normalized Cross Correlation

One of the core pieces of this project was to implement a hardware efficient normalized cross correlation (NCC) based on previously published work [7]. NCC is the math through which two image patches can be ranked based on how similar, or correlated, they are. The normalized part means the ranking will be largely invariant to differences in lighting in the two scenes. The following equation details the math necessary to compute the correlation coefficient for two image patches, which is the rank mentioned earlier. The coefficient will be a number between 0 and 1, 1 being the highest correlation score. A traditional CPU would not be phased by the division and multiplications present in the equation, but performing these operations on an FPGA would be both slow and resource-inefficient. To make this math possible, we take advantage of logarithm properties, specifically log base 2 (log2).

4.2.1 Log2

When the pixels of the window and descriptor come into the NCC computation block of the FPGA, they are immediately converted to log2 so they can be easily multiplied. To make the log2 operation efficient, we use an approximation detailed in [7]. The result of the operation is a fixed point log2 number with an integer and fractional part. The process consists of finding the most significant "1" bit in the pixel value, the index of which will become the integer portion of the log2 result. Then, all the bits to the right of that bit become the fractional portion of the result. This method is an approximation, but the performance benefits outweigh the small loss of precision here. We use 64 bits split into 10 bits for the integer portion and 54 for the fractional portion in order to maintain as much precision as possible.

In order to calculate the summations in the NCC equation, we need to operate on base 10 numbers. This means we need an inverse log2 (ilog2) module to go between the two representations. The ilog2 module is simply the reverse of the log2 procedure. First, the integer portion of the log2 number is used to shift a "1" bit into the correct index of the ilog2 result. Then, the fractional portion's bits are placed immediately after that "1" bit, and the result is complete. We need to use fixed point numbers to represent negative log2 numbers in base 10, so we use 32 bits for the integer portion and 32 bits for the fractional portion.

4.2.2 Processing Element

The computation of the numerator and denominator begins in the 16x16 grid of processing elements (PEs). The 16x16 descriptor is loaded 4 pixels at a time into the PEs using the following datapath illustrated in 5. Upon being loaded, they are converted to log2 and stored in registers. Then, the 256 window pixels are loaded simultaneously into the PE grid:

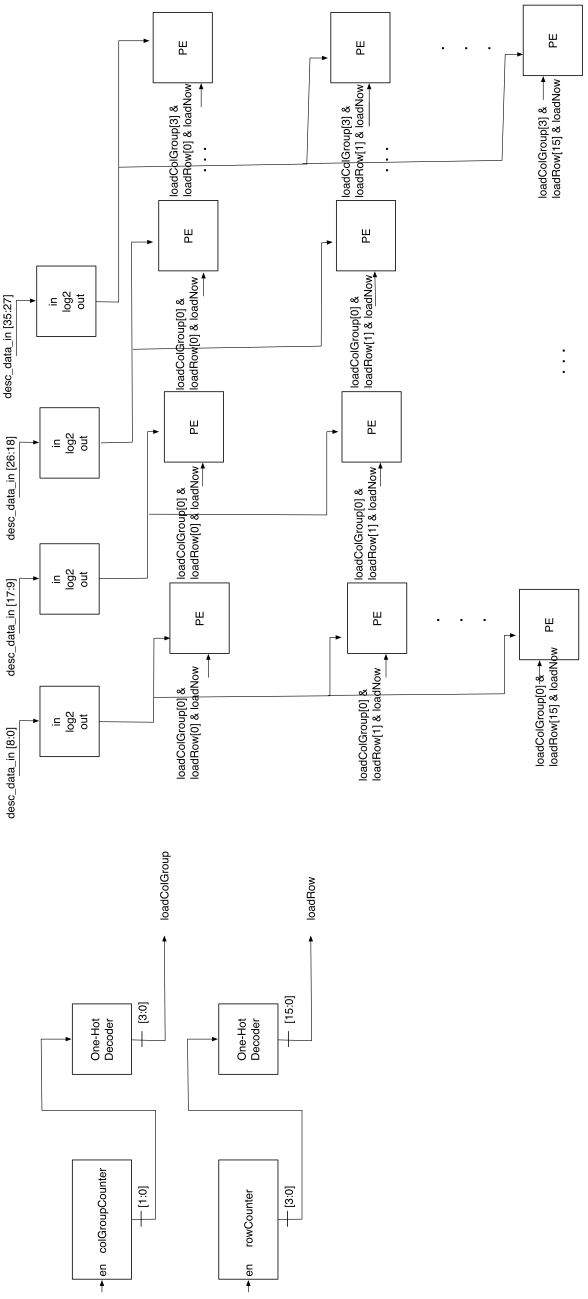


Figure 5: Descriptor Loading Datapath

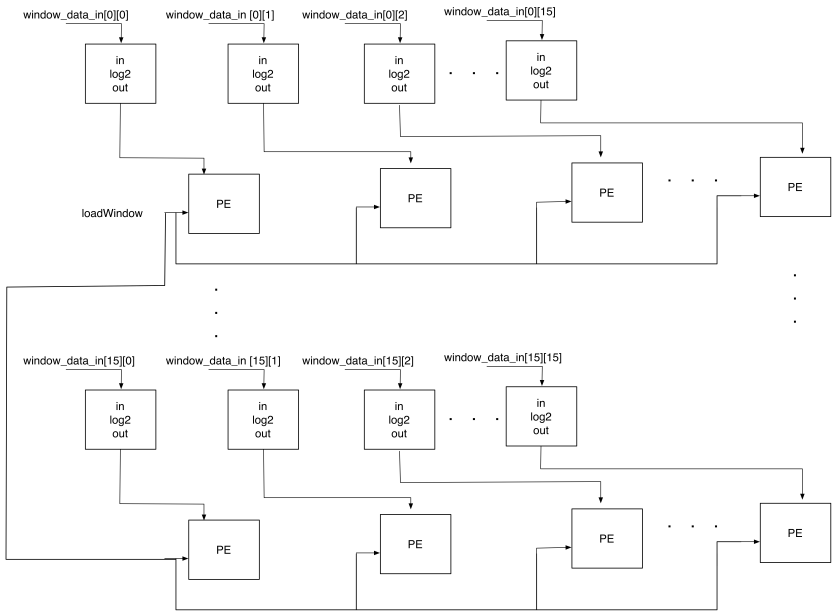


Figure 6: Window Loading Datapath

Numerator

The PEs are responsible for multiplying (adding in log2) each overlapping descriptor and window pixel, then summing the ilog2 of the products, which completes the numerator. Each PE outputs the square (left bit shift in log2) of its descriptor and window pixel in preparation of the denominator computation.

Denominator

For the denominator, the sum of the squares (SOS) of each pixel in the descriptor and window is calculated. Then, the two SOS are multiplied (added in log2) and the square root (right bit shift in log2) of that product completes the denominator.

Correlation Coefficient

Once both the numerator and denominator are complete, we subtract the two in log2 to perform the division. The difference is converted back to a fixed point base 10 number, which is the final correlation coefficient for that descriptor/window pair. A "priority register" is used to keep track of the greatest correlation cofficient as well as the index of the descriptor/window pair which yielded that result. After all window patches have been compared to their descriptor, the priority register's output is sent back to the PC as the final result.

4.3 PCI-Express

Xilinx's 7 Series Integrated Block for PCI Express(2.1) IP core is used in this project. The IP core handles formation of data packets and signals between FPGA and PC allowing easy usage and extension of the existing structure. For this project, a Master-Slave setup is used. The PC is in control of the program flow and determines which set of data to send to FPGA, when to start computation, and when to return the result. The PCI Express technology maps an array of memory on the PC side to the FPGA side, allowing any data from the write operation on the PC side to appear on the FPGA side and vice versa. For the sake of simplicity, memory-mapped control signals are used instead of interrupt signals. This has the downside of being slower but reduces the task of designing hardware structures and PC drivers that handle interrupt signals and control flows.

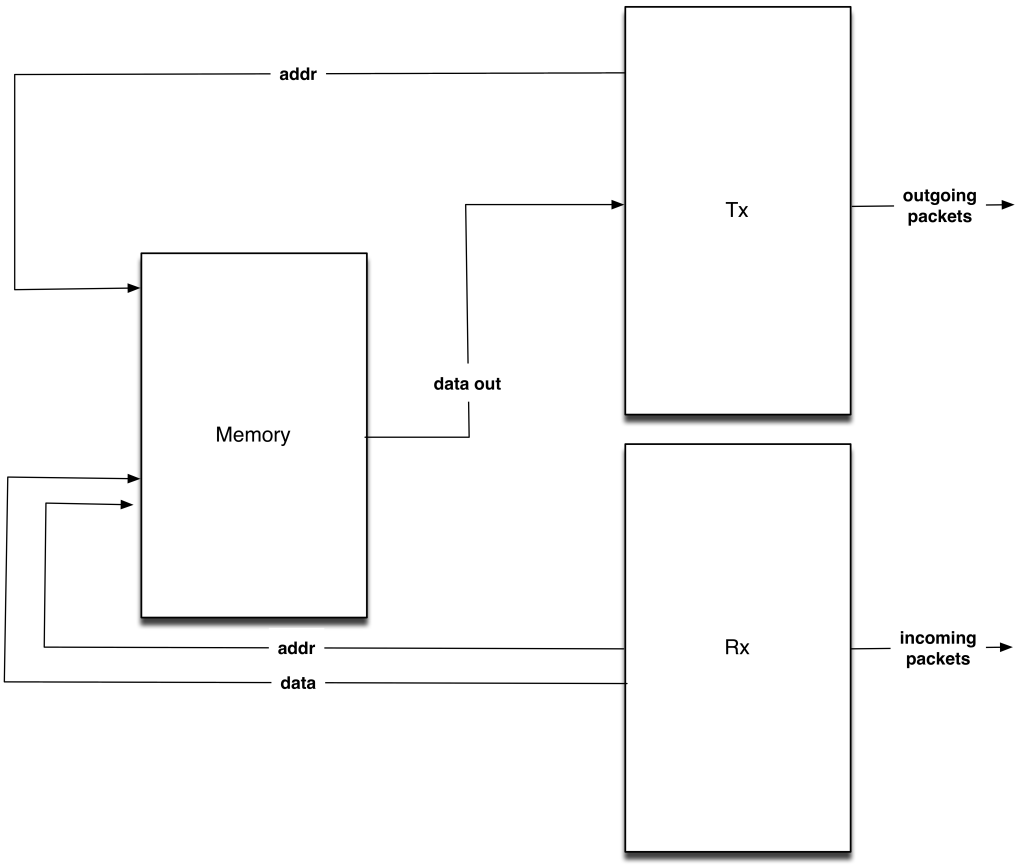


Figure 7: Transceiver module for the block memory

On the FPGA side, the IP core is configured to run at 250MHz, 8x lanes, and one Based Address Register(BAR) with size of 2MB. Only the first MB is addressable in the 4 byte chunk due to memory structure limitations. Any write or read operation to a higher address than 0x3FFFFF would yield an invalid result except at address 0x7FFFFE which is the address of the control registers. Each memory access is 4 byte aligned. The IP core uses AXI-stream interface which is connected to the transceiver module. The transceiver module is comprised of receiver and transmitter FSMs. Depending on the received command from the IP core, the data packets are either stripped into 32 bit chunks and stored in memory, or 32 bit chunks of data are pulled out from memory and formed into packets.

4.4 Memory Structure

True dual-port 256KB block RAM is the chosen component from Xilinx's IP catalog for the memory structure. Each block is configured with a 32 bit input and output interface and 64 pieces of primitive 36K BRAMs. 4 components are instantiated in total providing a combined storage of 1MB with 18 bits of addressable space due to each space being 32 bits wide. A small controlling logic is used to handle timing between each memory request due to memory accesses requiring more than one clock cycle to receive the data properly. It also handles routing incoming memory requests to the correct memory banks. The BRAMs are configured with one stage of registers on the output to shorten the critical path.

4.5 Pipelining

As a backup plan to meet timing requirements, the team also developed a pipelined version of the normalized cross correlation. The pipelined version is shown in Figure 8. In this implementation,

a pipeline register was instantiated between computationally expensive operations to increase the clock speed of the entire system. While this modification increases the latency, it has the advantage of enabling greater data processing and throughput of the normalized cross correlation modules by performing more operations per second.

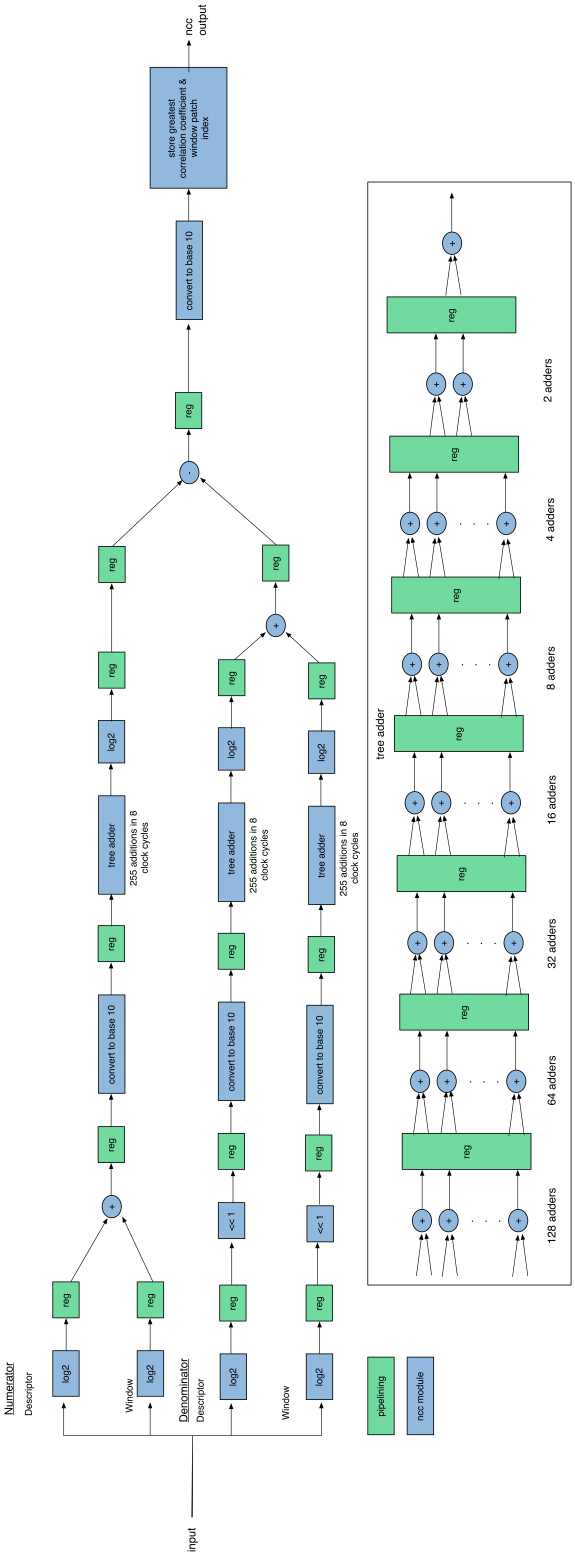


Figure 8: Pipelined Normalized Cross Correlation

The pipelining was extremely conservative. That is, registers were instantiated to maximize the clock speed so that timing would be as close to 250MHz as possible. In the original implementation, the tree adder existed as 255 additions per clock cycle. The pipelined implementation reworks this logic so that 255 additions still occur at every clock cycle, but with 8 clock cycles of latency until a

correct result is obtained.

4.6 Timing

The PCI-Express IP core and data transfer logic are set up to run at 250MHz for a maximum link speed of of 5.0GT/s, while the NCC algorithm runs on a 25MHz clock due to the longer critical path. The interactions between two clock domains are the passing of data to be processed and control signals of when to start processing. True dual-port BRAM is quite useful in this situation as it allows one port to be clocked at a different frequency than the other. On one port, data can be transferring in and out of BRAM at 250MHz to the PCI-Express IP core. On the other, NCC algorithm FSMs can be requesting data and writing back results into BRAM at 25 MHz. Control signal registers exist in the 250 MHz domain. To avoid metastability, any control signal crossing from or to the 25MHz domain is buffered with double registers using (*ASYNC_REG = "TRUE" *) Pragma. This asks the synthesizer to place both registers on the same logic slice, significantly reducing the physical path between the two.

The two different clocking frequencies are realized through the use of the Mixed-Mode Clock Manager(MMCM) module on the FPGA. The module takes in a reference clock, VC707 board's system clock is 100MHz, and produces a high quality low jitter output clock at a different frequency. It is also possible to specify the duty cycle and phase of the output signal.

5 Software

There are several tasks that are run PC-side in order to prepare data to be sent to the FPGA as well as receive the results that the FPGA computes.

5.1 Feature Detection

First, we must find suitable location candidates for matching across images. To do this, we use the Harris corner detector [3]. Then, the corners are filtered using non-maximal suppression, followed by adaptive non-maximal suppression (ANMS) to limit the number of corners to 150 [2]. The result is a set of 150 feature points in an image that have strong corner strengths and are well distributed throughout the image.

5.2 Feature Extraction

Once we have the set of feature points, feature descriptors are created for each location. First, 80x80 "descriptor" patches of pixels centered around each point location are extracted from the left camera's image. Then, the patches are downsampled 5x to a size of 16x16, blurred, and normalized. This process ensures our feature descriptors cover a large area of the image and do not contain so much detail that they become hard to match due to high frequency variances across different camera images. In the right camera's image, a larger "window" patch of size 400x400 is extracted around the same point locations. This window then goes through the same downsampling and normalized process as the descriptor. We assume that the feature point will be in a similar location in both the left, right, and next left image. Therefore, we just need to search the large window for the point that most strongly resembles the smaller descriptor.

5.3 Plotting Results

After the NCC computation is finished, the FPGA sends the strongest correlation coefficient as well as the index of the descriptor/window pair which yielded the best match back to the PC in order to accumulate and view the results. Using a MATLAB script, the PC can plot the detected feature points in the left image as well as the matching point locations in the right and next left

image found using the FPGA.

5.4 PCI-Express Driver

A kernel driver was written to transfer data from the CPU to the FPGA. The kernel driver took several iterations before a stable version could be produced that did not cause kernel panics. The kernel contains a set of functions that can be called from user space.

6 Design Decisions

6.1 PC-FPGA Data Transfer Protocol

During the course of this project, several methods of transfering raw data to FPGA and receiving computed result from FPGA were considered. The first option explored is Ethernet. However, PCI-Express was quickly chosen to replace Ethernet.

6.1.1 Ethernet

VC707 has a Gigabit Ethernet port and a controller chip the FPGA has to route data through. After exploring Xilinx's documents, it was determined that Ethernet support for the new system is limited. This includes outdated example designs(compatible with Virtex-5 only) and lack of up-to-date documentation for the VC707. Two IP cores for 7 series have to be used in conjunction to fully implement the functionality of basic data transfer suitable for the project due to each individual part only capable of certain functionality on VC707 board. There is also no document on how to format and send data on the PC side. The two IP cores needed for VC707 are Tri Mode Ethernet MAC core and Ethernet 1000BASE-X PCS/PMA or SGMII core.

6.1.2 PCI-Express

After determining Ethernet to be unsuitable for the project, PCI-Express is chosen. The first step taken was to explore reports and data previous teams have done before. F11 SideKick report is quite helpful in learning how PCI-Express works. Linux Device Driver writting guide mentioned in the report is almost very helpful when writting this project's driver. However, F11 SideKick's design is based on reference design for Virtex-5. While the Bus Mastering Design offers higher efficiency, It would be difficult to migrate from Virtex-5 to Virtex-7 without integration problems. The chosen Master-slave design is simpler to implement a working design with a tolerable margin of performance degradation.

Writing a driver for PCI-Express is an interesting journey. First we started off looking at F11 SideKick project which points us to Linux Device Driver tutorial which is a greate book for this project. After several trials and countless Kernel panics, some simple read and write commands were implemented. The commands are capable of writting an arbitrary size of data, as long as it is equal to or smaller than 1MB, to any portion of the memory space on FPGA. This is accomplished by copying a buffer on PC with a specified length and offset. This is not the most efficient way to utilize PCI-Express capabilities but without interrupts or Bus Mastering implemented, this was deemed sufficient for our project. To use the device driver, a few information needs to be set such as PCI-Express IP core's ID and manufacturer information. Other than that, it is quite simple to call pread and pwrite functions.

6.2 Memory Structure

Early on in the project design stage, it was determined that block RAMs will be used as the basis of the memory hierarchy. This is because BRAM is relatively simple to use and there are abundance amount on VC707. Distributed memory which comprise of LUTs is not ideal due to inefficient usage of resources available on the board. The SODIMM-DRAM which is part of the evaluation board is also very complex and was deemed too much trouble to try to get it working

than it was worth. The first BRAM design planned for 4MB but this was quickly revised down to 2MB of storage space. This worked well with the dummy test unit which increments every value in the memory by one every time the PC sends a start signal. However, once the memory hierachy is integrated with other components of the project, the synthesis tools reported timing issues. The Xilinx's synthesis and implementation tool chain was having a hard time doing place and route due to increased amount of hardware logics needed. This causes instability within the memory hierarchy and corrupts some amount of data transfered depending on the distance the paths were routed. This is a difficult problem to fix as it would require re-implementation of the memory hierarchy. This problem was discovered late into the project which did not leave much room for the ideal solution. Instead, the size is reduced from 2MB down to 1MB which offers better stability but will fail some time on different synthesis runs.

6.3 BRAM-NCC Data Transfer Protocol

This section is the controlling FSM-Ds which utilize NCC algorithm hardware to compute the desired result. This part includes querying memory requests for data to be processed, handling write back of results to memory, and keeps track of how many sets are processed.

6.3.1 FSMs

a total of four FSMs are used to control the computation flow. The first FSM is in charge of arbitrating access to the BRAM and pass on incoming control signals. All read and write request has to go through this FSM to share the memory port on FPGA side. The second FSM controls the overall computation flow by interacting with the first FSM to receive instructions and data. It also handles when to request a write back to memory. This FSM controls the activation of Descriptor Handler FSM and WIndow Handler FSM. Descriptor Handler FSM controls the formation of data packets being fed into NCC algorithm. This is achieved by computing the row, column, and frame index of the data to be requested. Since the width of memory is 32 bits, only four 8 bits pixels can be requested at a time. The more detailed description of how these FSMs-D operates is below.

6.3.2 Descriptor Handler

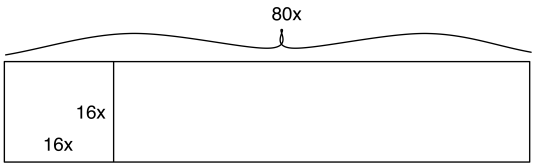
The descriptor Handler handles the request of image pixels of the 16x16 pixels descriptor. The FSM handles which address in memory to request the data. Since there is a cycle delay before data returns from the memory bank, the FSM handles this delay and pass on the correct pixel indices to the NCC algorithm.

6.3.3 Window Handler

The window Handler handles sending NCC algorithm 16x16 patches out of the 80x80 window which exist in the memory bank. Imagine having a 5 by 5 square. The goal is to form every possible 4 by 4 square. There are four possible squares with the top left corner locating at (0,0) , (0,1) , (1,0) , (1,1). For this specific NCC implementation, there would be a total of 4225 squares from 65x65 locations. This is a challenging problem to deal with efficiently in term of trade offs between hardware size vs speed due to only being able to request 4 pixels from memory at a time and the restriction that most of the pixels need to be used more than once.

Figure 9: example graph

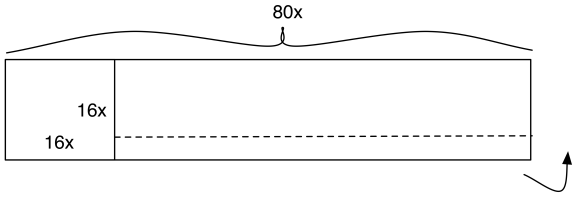
The first design explored is a 16x80 pixels holder with a sliding window logic. This design was believed to be a good compromise between area required and memory requests. A set of pixels have to be requested only once which allows maximum utilization of the limited bandwidth.



first, 16x16 patch for the top left corner is loaded into the window holder. Once this is done, the first window will be sent out to NCC algorithm and the next four column in the top row will be loaded into the window holder. Then, the other 15 rows of those 4 columns will be loaded. Once that is done, the window will slide to the right by one pixel, effectively sending another 16x16 patch that is one pixel to the right of the previous patch. The process is repeated until the last patch on the far right is reached. The next step is moving down by one row. This is done by doing a row-wise shift up operation. Row 0 holds row 1 data and row 14 holds row 15 data leaving row 15 free to be filled. pixels row 16 in the 80x80 window is then loaded into row 15 of the window holder. The window slide logic resets back to the far left and the process of loading in and shifting right is repeated. The operation continues until the bottom right corner of the 80x80 window is reached.

However, this design has a fatal flaw which was overlooked during the design stage. The window sliding logic can not be efficiently implemented in hardware due to the number of logic elements needed to handle the routing of data out from the registers. This becomes evident when the problem was assumed to be faulty FSM logic and a similar but larger and simpler 80x80 window holder is experimented with. The 80x80 window holder could not be synthesized. When it does synthesize, it causes other components integrated to it to fail due to timing issues stemmed from difficulty to place and route efficiently. Once The source of the problem is located, a shift register 16x20 window holder is implemented instead.

The operating principle of shift register 16x20 window holder is instead of sliding the window over register outputs, the registers' values are slided into the window. First, the first four columns are loaded into column 17-20 of the shift registers. Once the entire column is filled, the registers will shift all the values to the left by four resulting in the first four columns at location 13-16. Then the next four columns are loaded into the far right registers. This process is repeated until column registers 0-15 are filled in. At this point, the 16x16 patch is valid and gets sent to NCC algorithm. For each time the 16x4 pixels are loaded into the far right registers, the shift registers can move four times. Once the column 80 of the window is loaded in and shifted, it is time to move down to the next row. This is done by clearing all registers and started the process again with an incremented row offset.



The new design requires less logic components and is synthesizable. However, it would require more clock cycles due to the fact that certain pixels have to be requested more than once. This is an acceptable compromise considering how the first design causes more problems than the potential performance gain.

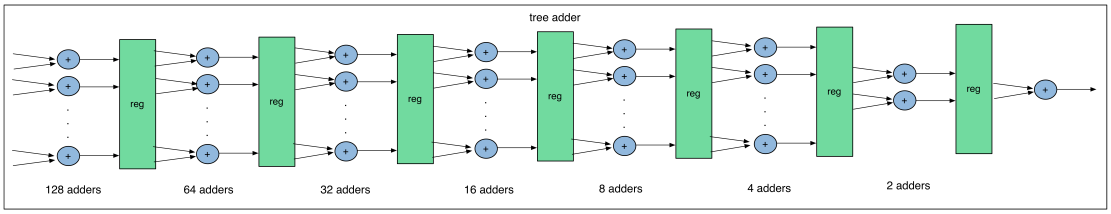
6.4 Tree Adders

As part of the NCC algorithm, there are instances where computed operands have to be summed. At first, the adders are implemented in SystemVerilog as follows

```
always_comb begin
               descSumOfSquares = 0;
               winSumOfSquares = 0;
```

```
            for (int rowI = 0; rowI < 16; rowI++) begin
                    for (int colI = 0; colI < 16; colI++) begin
                            descSumOfSquares = descSumOfSquares + descPixelOut[rowI*16 + colI];
                            winSumOfSquares = winSumOfSquares + winPixelOut[rowI*16 + colI];
                    end
            end
    end
```

However, this is synthesized into a chain of serial adders which creates a very long critical path, lowering the performance of the system. Thus a binary tree adder module is created to lower addition stages from (o) down to log2(o). Instead of having to add 256 operands sequencially, this module sums all the operands in eight stages, greatly reducing the critical path. As part of the effort to increase possible clock speed, this module is pipelined into 8 stages further reducing the delay if implemented in the pipeline version of the NCC algorithm.



7 Testing Methodology

7.1 NCC

The hardware modules used to compute NCC include log2 conversion, ilog2 conversion, descriptor loading, window loading, result calculation, and keeping track of the greatest NCC result. Testbenches were written to verify the correctness of each of these sub modules and that was considered the base line for unit testing, even though some of these modules instantiate other simple modules such as decoders and multiplexers. By verifying these modules' individual correctness, we can more easily transition to integration testing without having to exhaustively test all possible inputs.

The next step is to combine these sub modules to simulate the behavior of the entire NCC computation from the time pixels are loaded into the PEs to the time the result is put on the output. A testbench is used to act as the memory and control FSM, and monitor the output of the NCC block to ensure correctness of both the result and the timing of various operations.

7.2 Data Transfer

During the development stage of the PCI-Express, memory hierachy, and controlling FSMs, small components are individually unit tested in simulation. This is followed by an integration test of various module in simulation.

7.2.1 Unit testing control FSMs

FSM-Ds with particular tasks are unit tested for correct operations. It is crucial to break down components into small modules and submodules which handle specific tasks. These small modules offer robustness if a certain functionality needed to be replaced with a different implementation. A very good example is window_handler.sv which has three different implementations due to physical limitations. Unit tests also eliminate most of the bugs except ones which stem from integration process.

7.2.2 Integration Testbench

tbUserinterface.sv is the main integration testbench used in this project. It effectively simulates incoming control signals and the Block RAMs which hold the data to be processed. It also looks for out going control signals and result from NCC calculation. A good amount of integration bugs are caught with the use of this testbench. Integration bugs are almost inevitable when tasks are split between team members. While the interface and specifications of each modules have been defined before hand, it is likely that there will be something somewhere which slips through. A good amount of time was spent using tbUserinterface.sv to ensure the overall system on FPGA side works as intended.

7.3 PCI-Express

testing PCI-Exress and data transfer to the memory hiearchy is quite tricky. Perfect simulation result does not guarantee a perfectly working synthesized version. It was later discovered in the project that due to critical path being too long, certain memory space is not stable. This leads to trial and error approach to confirm the stability of the system. The development of the Linux PCI-Express driver is also by trial and error approach as there is no real easy way to simulate the operation of the driver.

7.4 Simulation in Software

7.5 Synthesis

Once the simulation results looked good, synthesis of the entire system began. We immediately ran into timing issues, among other things, which caused us to re-evaluate our architecture. By using two clock domains, one for PCI-E and one for the rest of the system, we were able to mostly avoid the timing issues but our modules still were not performing correctly on the board. Testing the synthesized system was a bit tricky; we monitored our state using the board's LEDs and tried to use chipscope when necessary, but we could not observe all of the status and control points we needed, possibly due to them being optimized away by the synthesis tools. As a result, testing and debugging on the board became a long and puzzling process made worse by long (¿ 2 hours) synthesis times and the necessity to restart the PC every time the board was programmed in order to set up the PCI-E connection. We would make 3 or 4 different changes on 3 or 4 different Vivado projects, then synthesize them simultaneously on different machines in order to expedite the procedure.

8 Results

This project successfully demonstrated PCI-express data transfer at a rate of 60 frames per second (fps). Additionally, a block memory was instantiated to hold the descriptors and windows for an entire set of images. A special register was integrated into this block memory so that the FPGA could be instructed to perform particular tasks based upon the instruction loaded from the CPU. For example, the demo the team had put together used this instruction area to signal to the FPGA whether the data contained in the block RAM should be incremented before being transferred back to the CPU. Managing the transfer of data to and from the block memory required developing systemVerilog code to signal between the normalized cross correlation modules and the rest of the system to ensure that data was not dropped or lost.

CPU code was written for a visual odometry algorithm in C++. Normalized cross correlation was stripped out of the C++ code and replacement code was inserted to interface with the FPGA so that data transfer between the CPU and the FPGA could occur. MATLAB scripts were written as a sanity-check for the hardware implementation. The normalized cross correlation was broken up into sensible sub-modules written in systemVerilog, simulated with VCS, and synthesized at 5MHz. Furthermore, a kernel driver was written to transfer data packets to and from the FPGA for processing on the CPU. A pipelined implementation of the normalized cross correlation code was also

written.

The team instantiated multiple clock domains into the design and successfully synthesized the entire system. Data transfer between the block memory and the normalized cross correlation code was demonstration. Unfortunately, this data was corrupted due to the large difference in timing between the PCI-express and the normalized cross correlation modules.

9 Individual Comments

9.1 Dylan Koenig

My primary objective for this project was to implement the computer vision algorithms in both hardware and software. This consisted of adapting and implementing [7] to suit our needs, coming up with protocols to interface between the FPGA's memory and the NCC computational blocks, researching appropriate data transfer protocols, writing the C++ code to detect, extract, and match features, as well as the C++ code for the rest of the visual odometry computer vision processes, writing the systemVerilog description to perform NCC, as well as testing these modules. Of course, the entire project was a group effort and so we all helped out each other when necessary, especially when debugging.

When we first began this project, we had a fairly ambitious set of goals. As the reality of the semester closed in on us, we took Professor Nace's advice and realized that parts that would get done "if there was enough time" were simply not going to get done. We settled on a more plausible final goal, and removed some of the excess features. By using systemVerilog, we limited our programming environment to Vivado and our FPGA to the Virtex 7. This decision created a large obstacle for us early on in our project since the Virtex 7 was lacking in documentation (especially regarding Ethernet IP and PCI-E app-notes) and of course none of us were familiar with Vivado. Once we switched from Ethernet-based data transfers to PCI-E, and found a suitable PCI-E design to try to get running on the Virtex 7, our project began to pick up in pace as we surpassed the typical early hurdles. It was at this point that I wish we had spent some more time fleshing out the rest of our project, as it would have saved us many head aches in the future. Towards the end of the semester, we discovered an architectural problem that would require major reworking of how data was being transferred to the NCC modules and how the math was being computed within those modules. I had to adapt many of my previously-complete modules to operate on signed and fixed-point numbers, which then necessitated re-evaluating even basic modules for correctness that had been in a working state for a long time.

For any group about to embark upon a similar 18-545 project, I recommend implementing any complicated algorithms in a software programming language such as MATLAB first so that you can become intimately familar with it, and get a better idea of how the hardware description will behave before you actually begin drawing it out or coding it up. I also recommend drawing EVERYTHING out in excruciating detail on paper first before you type a single line of code. All datapaths and FSMs should exist on paper, not just for the sake of easing the implementation, but also because it will help keep your team members, who may be working on completely different tasks, on the same page.

This project definitely increased my systemVerilog proficiency, as well as my less "technical" skills such as how to start an ambitious project from scratch and see it through to (near) completion. Planning things out and making schedules helped, but our timeline was a bit hectic due to the need to make unforeseen changes to the design. One thing I wish we had done earlier was attempted to synthesize our code, even if it wasn't in a final form. We began synthesizing the whole system too late, and as a result we got hung up on timing issues, size issues, weird Vivado quirks, and just a lack of familiarity with the tools. Our end result was not what we set out to achieve, but we did succeed in synthesizing fast PCI-E data transfer for the Virtex 7 board which will hopefully be of use for future projects. The iterative nature of this class means that anyone can take our work and build off of it, so if we are able to help a group get past that initial speed bump, awesome! We were also able to verify correctness of an efficient hardware NCC in simulation, so I have no doubt that we were right on the brink of a fully working implementation. Nearly every weekend was spent in the lab during the semester with few exceptions, but I think that is par for the course. The last week or so of the nearly every day was spent in the lab. During Thanksgiving break, I worked remotely with Gun and Wennie back at base.

I loved this course for giving me the opportunity to work with extremely talented people on

whatever we wanted. If I could make one change, I would add some kind of supplemental lecture where tools like Vivado could be presented in detail to expedite the process of becoming acquainted with them. Since projects and tools vary from team to team, this may not be possible, but I think it'd be a big benefit to the teams as it will get them past some initial technical difficulties and on to implementing something awesome.

9.2 Gun Charnmanee

At the start of the project I didn't really know what NCC algorithm or visual odometry is so we had a meeting where Wennie and Dylan explained briefly what we need to do in this project. This leads to me being tasked with handling data transfer between PC and FPGA. Since we only had a general idea of what we wanted to do and what the final result will look like, we had to explore several options and choosing the best one for this project. At first Wennie and I experimented with Ethernet as the choice of data transfer. This turned out to be difficult due to several reasons. The documentations on Xilinx's IP cores and software available for VC707 are quite limited and often a nightmare to go through. I feel like a good portion of this class early in the semester was spent reading through various documents which often times are out-dated due to multiple versions being available on Xilinx's website. When working with documents from Xilinx's website, make sure the documents are the latest version.

We ended up choosing PCI-Express as our choice of data transfer due to two reasons. One, a team from previous semester had implemented PCI-Express before which is a good sign indicating it is doable. Two, PCI-Express theoretically has better performance than Ethernet. I went through F11 SideKick writeups and code base and learned quite a bit about how PCI-Express works. Hours of internet surfing on various sites also reinforce my understanding of the protocol. Xilinx has an easy to implement PCI-Express IP core so I decided to use that as the base for our project. First I set up the IP core's example design and try to write a driver to interact with the design since I could not locate a suitable Linux Driver which will work 100% with our setup. Writting the driver includes reading Linux Device Driver tutorial per suggestion from F11 SideKick which is very useful in getting a glimpse at how Linux Kernel works. However, Wennie and I still had to experiment with various configurations and function calls before we have a system which meets our specification. The data transfer at first is really slow but after spending a week or two we were able to locate the right configuration which boosts the speed significantly. I think we might have spent a little too much time on this. The time which could have been spent on other parts of the project.

After we were sure that we can transfer data to FPGA and read back at a reasonable speed, I started working on the memory hierarchy which is a bunch of true dual-port Block RAMs stitched together to provide enough space to hold data to be computed. This involves reading more documents about how Block RAMs operates, its usefulness and limitations. I also updated transceiver FSMs timing to support the new changes. After this is done, I started working on the controlling FSMs which will feed data to Dylan's NCC algorithm code. This step involves some planning on what the interface and operation should look like. The first thing I implement is a dummy module which interface with Block RAM and control signals and do some operation on the data. In this case I made the module increment each integer by 1 and store it back into the same location. The end result is a system where user can place data in a memory buffer, send a signal by calling a C code function call on PC side, wait a little bit, and read back data which has each integer incremented by 1.

Towards the end of the project I worked on designing and implementing controlling FSMs on FPGA side by adapting the dummy module developed earlier. This involves devising a plan to deal with limited data width accessable per clock cycle. This leads to creation of more than a few FSMs which handle different tasks. I created the outline and prototypes of those FSMs but towards the end of the project, Dylan and Wennie debugged and modified them to working condition. A big problem we ran into near the end of the project is the fact that when HDL is synthesized, things will not go according to plan. A module I developed to be efficient turned out to not be efficent space-wise on the FPGA and causes stability issues. Two major issues I worked on towards the end of the project are the unstable memory hierachy and the signaling across clock domains. When multiple clock domains is used, special care needs to be taken to avoid metatestability. This is something we didn't think about until we had to use a very slow clock for NCC alogirthm due to a long critical path since the first version implemented does not use pipeline.

Vivado as the choice of tools: At the start of the project we opt in to use the newest Virtex-7 board, the VC707. The board is the newest technology available for us to use and it is definitely powerful in terms of logic components and capabilities. The Vivado chain tools also support SystemVerilog files which is a great plus. However these advantages do not come without a cost. I find the lack of support for the newest and latest device to be bothersome. There are many great

example designs and tips available for previous boards but not for VC707. Migrating these designs also needs great effort as it requires knowledge of the design and board specific configurations. If a team would like to use the newest technology, be prepared to spend a significant amount of time familiarizing with the tool chains and do not expect help to be easily discoverable on the internet. It might have been better for us to stick with a board which have been used and tested in this class before.

Synthesis and implementation: I feel like one of the major reason we did not accomplish what we set out to do is the fact that we planned integration and synthesis of the design too late into the semester. I thought a few weeks would be enough time to integrate and synthesize a working design. I was proven very wrong when things start breaking down and it takes more than twenty minutes to run an iteration of tests.(our design takes about twenty minutes with the dummy module) The synthesis and Implementation time increased to over forty minutes with the NCC algorithm and FSMs due to the large amount of logics Vivado has to place and route. To avoid problems, I believe teams should aim for two-third of the way through the semester as the latest point to start integration and synthesis.

Overall I enjoyed working and learning in this class. On an average I probably spent somewhere between 15-17 hours per week on this class, many of the hours are during the last two weeks. The experience in this class is very different from normal college classes as it is mostly self-studying and exploring anything we want to use to accomplish the project. A very important lesson I learned is the importance and difficulty in synthesizing and implementing our design on FPGA. I believe Vivado tool is very powerful and is capable of high efficiency but it also has a very high skill ceiling to utilize the tools effectively. It would be great if I could learn more about how to design a project which is synthesis friendly.

9.3 Wennie Tabib

At the beginning of the semester I researched how to synthesize Ethernet on the FPGA board so we could transfer images from a host PC to the VC707 board. Gun and I discovered that we needed a Tri-Mode Ethernet Media Access Controller (TEMAC) and transceiver. We used the TEMAC and LogiCORE IP Ethernet 1000BASE-X PCS/PMA SGMII documentation to learn about the Xilinx IP. We tried using the IP integratrator to connect the inputs and outputs of the blocks we needed by copying a design from page 238 of version 13.0 of the pg047 manual. We couldn't get everything to line up properly and we ended up trying to synthesize a .rar file we found in the Xilinx Community forum that claimed to have a usable ethernet core. However, we were never able to get this to work and decided to get PCI-express instantiated on our board instead.

Next, I researched how to get PCI-express integrated into our design, and tried to use Coregen directly. Neither Gun nor I could get this approach to work. Finally, I found xapp1022.zip on the Xilinx website which was built for a Virtex-5. When I read xapp1022.pdf I realized that I could probably jsut synthesize the example design and use the driver from xapp1022.zip to read and write data to the Programmed Input/Output (PIO). I synthesized the design and Gun and I began working on compiling and installing the kernel driver on the lab machines to est the write/read from the CPU. We learned that we had to reboot the machine after downloading the bitfile to the FPGA or the CPU would not recognize that the VC707 board was connected to it and was unable to read or write to the FPGA. Once we compiled and installed the driver in the kernel directories using *insmod* we ran the program and it worked!

At this point, the team had working PCI-express but it was very slow. It took about 15s to transfer a single image of data to the FPGA because the kernel driver included in xapp1022.zip only wrote and read 4 bytes of data to the FPGA and back. This meant that a context switch had to occur every time an integer was written to or read from the FPGA chip. I increased the amount of data that could be stored in the kernel's buffer space. In particular, there was one place in the kernel driver that mapped a write command used in User Space to a write command used in kernel space. The user space functions are called *read* and *write* but the kernel space functions that these were mapped to are called *XPCIe_Write* and *xPCIe_Read*. There were also 4 functions in the kernel driver that were instantiated but not used. These functions were *XPCIe_WriteMem* and *XPCIe_ReadMem*. We tried to replace the mapping of read and write to these functions, but the result was a kernel panic in both cases. Because we couldn't get that to work, Gun and I tried to replace the call to *memcpy* in the *XPCIe_Write* and *XPCIe_Read* functions in the kerel driver. We replaced memcpy with *memcpy_toio* in *XPCIE_Write* and *memcpy_fromio* in *XPCIe_Read*. This worked and we were able to speed up the data transfer by a factor of 3x.

Next, I realized that the example design for PIO on the VC707 didn't utilize the full block RAM for data transfer. Gun and I made the modification and we were able to transfer almost twice as much data in the same amount of time. The next step was to instantiate a DMA controller but we decided not to do this because this task in itself could take a semester or more to accomplish.

The team decided that we needed a large memory that Dylan could use to store data. Gun and I worked together to instantiate a 2MB memory using a block memory generator. The block memory used up roughly 50% of the logic fabric on the board. After we accomplished this task, I worked on getting offsets to work in our linux driver so that we could write and read to different places in the memory. I got it to work and was able to transfer 4 pixels (8 bits) packed into a *uint32_t* to the FPGA.

I wrote the FSM that interfaces between Dylan's NCC code and the block memory that Gun and I put together. Gun and I also started synthesizing the modules we needed to interface to Dylan's code.

A week before Thanksgiving break we found a problem in our design. We realized that we needed signed 9-bit pixels instead of unsigned 8-bit pixels because we forgot to subtract off the mean from

each pixel before performing the normalized cross correlation. Gun and I spent the better part of Thanksgiving break re-architecting our code so that we didn't have to overhaul the whole design. I prototyped the paper we based our design off of in matlab using the fixed-point library to ensure what was happening on the FPGA was correct. I heavily edited the C++ code we had for visual odometry to get test data for the FPGA NCC. I learned to use the boost library in order to do this. I edited the kernel driver and code to interface with the FPGA so that we could repackage the data transfer descriptors and features to the FPGA. I also integrated Gun's and Dylan's code and wrote testbenches to verify correct simulation results. I also helped Gun track down a few errors in his FSMs.

Finally, I implemented a pipelined version of Dylan's code in the hopes of getting a higher clock speed for our design.

On average, I spent about 16 hours per week on this class. The last two weeks of class I spent all day, every day on this class. This was one of the hardest classes I've taken at CMU along with 15-251, 16-861, and 16-865. If I had to redo this class over again, I would organize my semester so that I could spend most of my time on this class.

I wish we had started synthesizing at around mid-semester. We waited way too late to synthesize the whole system. I wish we had put more muscle behind the normalized cross correlation development so we completed at mid-semester and then spent the rest of the semester working out the bugs in the system and getting the whole system working for the final demo.

I think we made an excellent decision switching over from Ethernet to PCI-express. We had really fast PCI-express during our demo and working simulations in System Verilog. However, I think we made a huge mistake in choosing a project that either completely worked or did not work at all. I would advise future students to choose a project that has incremental deliverables. It would have been better if we could have demonstrated sound or a working display by the end of the semester, but we didn't have that.

What I think would be really cool is if Professor Nace could invite back students to talk about their experiences in the class. I think it would be more powerful if students from previous semesters came back to the class to give their final presentation to the students from the current semester. It would also give students from the current semester the opportunity to ask questions. Also, I got my whiteboard way too late. Whiteboards would be awesome...and more cots.

Overall, this class was pretty awesome. I went into this class wanting to learn how to transfer data quickly from a CPU to the FPGA because that's been something that I lacked in my research. After this class, I can say that I figured it out and I can use it for my research project.

References

[1] Jeffrey Biesiadecki, Chris P. Leger, and Mark w. Maimone. Tradeoffs Between Directed and Autonomous Driving on the Mars Exploration Rovers. *The International Journal of Robotics Research*, 26:91–104, 2008.

[2] Matthew Brown, Richard Szeliski, and Simon Winder. Multi-image matching using multi-scale oriented patches. pages 1 – 8. Microsoft Research.

[3] Chris Harris and Mike Stephens. A combined corner and edge detector. pages 147 – 152. Alvey Vision Conference.

[4] Laurent Kneip, Davide Scaramuzza, and Roland Siegwart. A Novel Parametrization of the Perspective-Three-Point Problem for a Direct Computation of Absolute Camera Position and Orientation. *Computer Vision and Pattern Recognition*, pages 2969 – 2976, 2011.

[5] Kurt Konolige, Motilal Agrawal, Morten R. Blas, Robert C. Bolles, Brian Gerkey, Joan Sola, and Aravind Sundaresan. Mapping, Navigation, and Learning for Off-Road Traversal. *Journal of Field Robotics*, 26:88–113, 2008.

[6] J. P. Lewis. Fast normalized cross correlation. *Vision Interface*, pages 1 – 7, 1995.

[7] Ming Z. Zhang and Vijayan Asari. A Hardware Efficient High Performance Digital Architecture for Run-Time Computation of Normalized Cross Correlation. pages 1116–1123. WSEAS Transactions on Circuits and Systems, 2006.