Lucas-Kanade Optical Flow Accelerator

Jud Porter Mike Thomson Adam Wahab

May 11, 2011

1 Project Objective

Optical flow algorithms are used to detect the relative direction and magnitude of environmental motion observed in reference to an "observer." The "observer" is usually a camera, and motion-quantifying processing is done on the differences between two subsequent captured images. Optical flow has a wide range of applications, especially in robotics. For example, optical flow can be used for pose estimation, obstacle avoidance (by moving away from regions of high optical flow), and image segmentation (by dividing regions into areas of similar optical flow, comparable to stereo vision).

The goal of this MIT 6.375 [3] project is to develop an optical flow design that can be incorporated into the Harvard RoboBee project [2]. This project aims to build micro-mechanical, autonomous, biologically inspired flapping wing robots. Much research has been done showing the importance of optical flow to the behavior of real honeybees [7]. Therefore, it is a natural choice of algorithm to implement on a robotic bee. The main design constraints of this project are weight and power, given the small size of the platform [5]. The estimated power consumption of all sensing and processing in the RoboBee is on the order 10 mW, so efficient computation is of the utmost importance.

Using a traditional general purpose CPU for this application would consume too much power to be a viable option. Typical low-power mobile CPUs still consume on the order of several watts of power, order of magnitude greater than the power budget for this project. A low-power microcontroller might fit within the power budget, but would not have enough processing power to meet the performance goals required for stable flight. The only way to achieve the computational efficiency required is through custom hardware design.

Traditional hardware design however is difficult, time consuming, and expensive. Using Bluespec^{TM} [1] to implement our project allows us to implement, verify, and refine our design much faster than if we had used a language such as Verilog or VHDL. Bluespec^{TM} also allows us to target both ASICs and FPGA systems. We can leverage FPGAs to verify hardware functionality and to integrate with sensors and components being developed by other group while the project is still in the development phase. Once the design been fully tested and integrated on the FPGA, it could be fabricated into an ASIC for maximum performance and energy efficiency with minimal additional effort.

1.1 Design Requirements

In order for this architecture to be useful for the RoboBee application, it must meet several performance requirements. The requirements are based on the frequency of actuation and control needed to maintain stable flight. The RoboBee's wings flap at around 100 Hz, and can only be controlled at the start of a new stroke. Therefore, a new optical flow reading must be made every 10 ms, setting the minimum throughput desired to be 100 frames-per-second (FPS). Initial experiments have shown that a minimum of 64×64 pixel camera resolution with 8 bit pixels is required to provide enough accuracy for adequate control. Assuming that only one new frame needs to be read in per optical flow computation (a frame from the previous operation can be reused), this equates to a minimum throughput of 100 FPS $\times$ (64×64) Bytes $\approx$ 400 kB/s.

2 Background

The general aim of optical flow is to quantify the amount of "flow" or visual movement between images. While many different optical flow algorithms exist, we have implemented the Lucas-Kanade (LK) optical flow algorithm [6] for this project. LK is an older algorithm, but is well-established and widely used. Many other algorithms build upon LK or use similar operations, making this a suitable reference point for later exploring the space of different algorithms.

Optical flow generally starts with the "brightness constraint" assumption. This assumption states that the brightness of a pixel does not change between frames. There are situations that can cause the brightness constraint to not hold, such as long times between frames, or occlusions and boundaries, but in general, it holds for fast frame-rates and low intra-frame motion. The system of equations generated from the brightness constraint alone though is underdetermind (there are fewer equations than unknowns), so additional assumptions are required in order to solve for the optical flow field. In LK, the additional assumption is that the optical flow is constant in a small local neighborhood of pixels. LK calculates a dense optical flow field, meaning that a vector is calculated for every pixel.

Calculating optical flow using LK generally starts with a discrete estimation of the spatial and temporal derivatives. The spatial derivatives can be calculated using convolution with a Sobel filter. The difference of the two input images can be used as an estimate of the temporal derivative. Once the derivatives are calculated, the system of equations shown in Equation 1 is solved for the optical flow field. To do this, the various per-pixel products of the derivatives are calculated, and the resulting products summed over a small neighborhood of pixels (3×3 for example). This neighborhood summation is equivalent to convolution with a boxcar-type filter. These resulting summations make up the elements of the matrices in Equation 1. This equation is finally solved to generate the resulting optical flow field.

$$\begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = - \begin{bmatrix} \sum I_x I_t \\ \sum I_y I_t \end{bmatrix} \tag{1}$$

This algorithm can be computationally expensive, especially as image sizes grow, making it ill suited for general purpose CPUs. It does have a lot of exploitable data parallelism, however. This makes it amenable to acceleration. Additionally, the algorithm can be composed of a number of discrete operations, making it suitable to pipelining.

3 High-Level Design

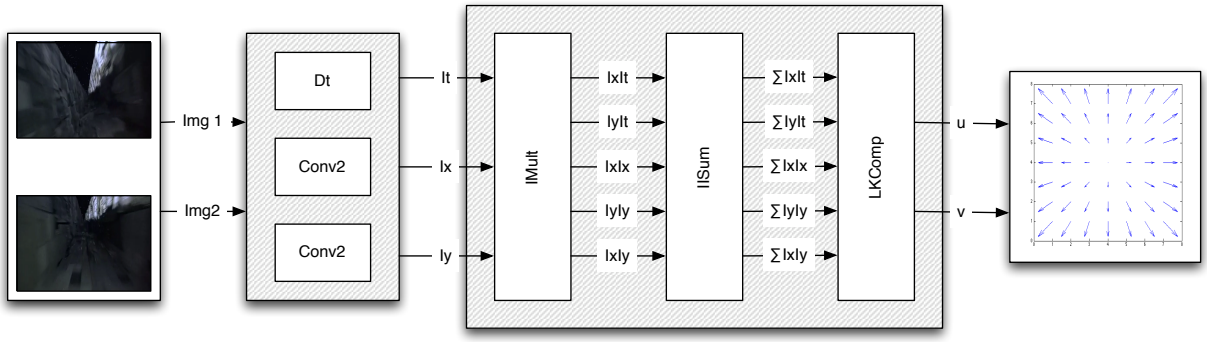

Figure 1: High level architecture.

Figure 1 shows a high-level depiction of the computational stages of the Lucas-Kanade accelerator. Initially, both images are convolved with a Sobel filter in the `Conv2` blocks to calculate the x and y derivatives. The output of the `Conv2` modules is averaged element-by-element to generate I_x and I_y. The I_t block similarly performs an element-by-element subtraction as an estimate of the time derivative. The resulting values are passed to the `IMult` module, which is responsible for calculating the various per-pixel products of I_x, I_y, and I_t. These products are sent to the `IISum` module, which buffers and sums the products over the appropriately sized neighborhood (3 is used here). Finally, these summations go to the `LKComp` module, which solves (1) for the velocity components $[u_1, u_2]$.

4 Test Plan

The test setup consists of a C++ software testbench, the optical flow implementation, and a SceMi interface layer using a PCI Express bridge (see Figure 2). This arrangement is similar to the MIPS testbench used in earlier 6.375 assignments. The testbench runs on the host processor, and the optical flow implementation can be tested using either simulation or the FPGA. The SceMi PCIe bridge uses ports that expose a chip reset, a kernel (Sobel image filter), image data, and resulting optical flow data between the testbench and the DUT.

Figure 3 shows the general execution of the software testbench. When the test starts, a reset and a high-pass filter kernel must be sent to the optical flow design (via the SceMi interface). Then the testbench sends image data to the FPGA and receives the resulting optical flow data from the FPGA.

The design is verified by comparing its output data with known-good reference data. Reference data is generated by running the same image data through a MATLAB^{TM}reference design.

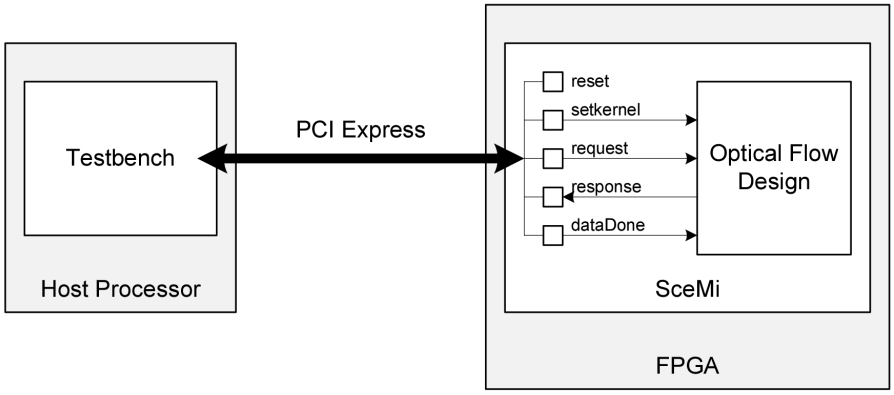

Figure 2: Test setup for the optical flow implementation.

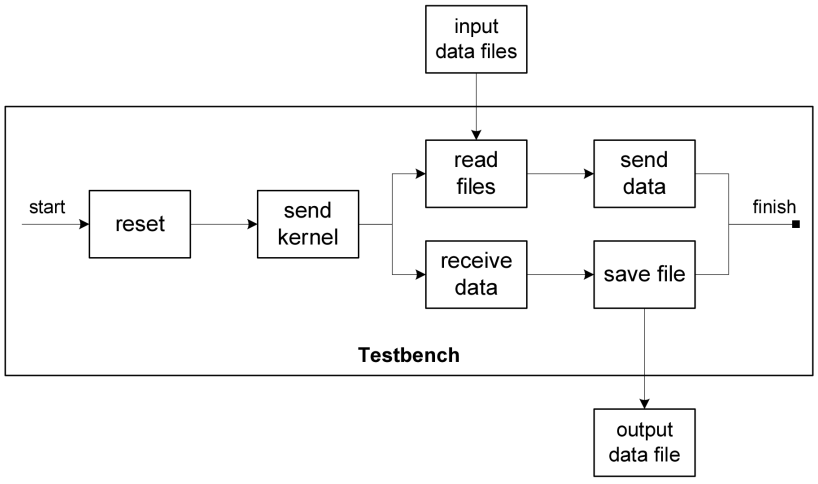

Figure 3: The execution flow of the software testbench.

5 Microarchitectural Description

5.1 Optical Flow Pipeline

The main pipeline module `OFlowPipeline` strings the various sub-modules that are required to compute optical flow. Data is passed between sub-modules via a series of stages connected with FIFOs for which there are rules to propagate the data. Each stage in `OFlowPipeline` corresponds to a stage of the LK optical flow pipeline as presented in Figure 1. Table 1 describes the individual stages.

Stage	Description
`dt`	Compute temporal derivative
`conv2_to_imult`	Pass results from `Conv2` and `dt` to `IMult`
`imult_to_iisum`	Pass results from `IMult` to `IISum`
`iisum_to_lkcomp`	Pass results from `IISum` to `LKComp`

Table 1: Stages comprising `OFlowPipeline`.

5.2 Temporal Derivative

The approximate temporal derivative of pixel intensity I_t is evaluated by the `DT` module which is instantiated in the `OFlowPipeline` module. A pair of pixel intensities from the two input frames are passed into this module via a FIFO. The input values are typecasted to match the output format of the `Conv2` module. The difference of the bit-extended values is then computed and loaded into an output FIFO. The temporal derivative approximation may be expressed as

$$I_t(u, v) = B(u, v) - A(u, v)$$

where frame B was acquired immediately after frame A.

Extra logic is also included to avoid generating values at the borders of the image. This is required because the `Conv2` module only outputs valid data, it does not produce data at the image boundaries. The same logic is used in `DT` to ensure that the output of `DT` stays synchronized with the output of the `Conv2` modules.

The temporal derivative result is read from the output FIFO and passed to the `IMult` module along with the output of the `Conv2` modules in the `conv_to_imult` rule.

5.3 Convolution Module

The convolution module `Conv2` convolves a $p \times p$ kernel with an $n \times n$ image. Mathematically, the 2-dimensional convolution operation may be expressed as

$$A_x(u, v) = \sum_{i=0}^{p-1} \sum_{j=0}^{p-1} A(u - i, v - j) K_x(i, j)$$

where K and A represent the kernel and image matrices, respectively. Parameters u and v represent horizontal and vertical offsets of the kernel's position with respect to the upper-left corner of A. As the kernel is incrementally shifted across the image matrix, u and v both vary from 0 to $(n - p)$.

The main pipeline involves four instantiations of `Conv2` in order to concurrently compute the following four spatial derivatives

$$A_x = A * K_x$$
$$B_x = B * K_x$$
$$A_y = A * K_y$$
$$B_y = B * K_y$$

where A and B are two input frames and K_x and K_y are the horizontal and vertical derivative kernels, respectively. A diagram illustrating how input data is routed amongst the four instantiations is provided in Figure 4.

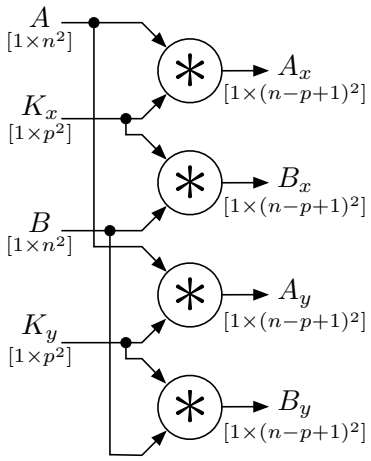

Figure 4: Four instantiations of the convolution module concurrently compute the spatial derivatives of input frames A and B.

5.3.1 Module Description

Rather than buffering an entire frame prior to performing the convolution, the module computes individual elements of the convolution result array as the input is streamed in on a pixel-by-pixel basis.

The `Conv2` module contains two rules: `buffering` and `processing`. As its name suggests, the `buffering` rule stores input pixel values in a minimum-length shift register of size $1 \times [n \times (p-1) + p]$.

The `processing` rule computes the sum of products for one element of the convolution result array and updates the convolution shift parameters u and v. Upon reaching the end of a row, the horizontal shift parameter u is reset and the vertical shift parameter v is incremented. When the final column of the final row of the input image has been processed, both u and v are reset in preparation for a new frame. For each (u, v) pair, the rule computes p^2 products comprised of the elements $(n \times i)...[(n \times i) + (p-1)]$ of the input buffer, where $i = 0...(p-1)$, and elements $0...(p^2-1)$ of the kernel storage array. Figure 5 illustrates how the contents of the input buffer relate to the elements of an input image for two instances.

The `processing` rule implements a linear inelastic pipeline for convolving subsets of input pixel values with the filter kernels. A diagram illustrating the pipeline structure and signal routing is included in Figure 6.

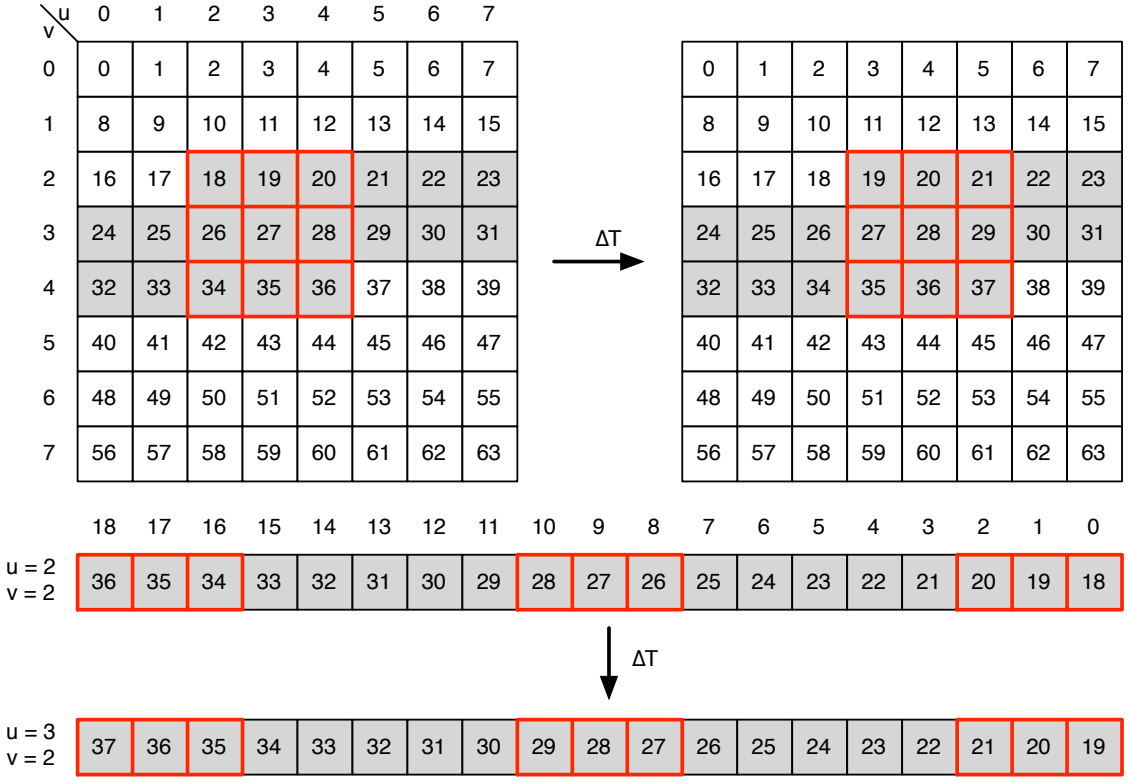

Figure 5: A minimum-length input buffer (gray) stores pixel values from an input frame (white). For each (u, v) pair, the `processing` stage computes the products of select input pixel values and the kernel (red). The products are summed and (u, v) is incremented, indicated here by ΔT. In this example, $n = 8$ and $p = 3$.

The two rules are mutually exclusive in that the `processing` rule will not execute unless the minimum-length input buffer is full and the `buffering` rule will not shift in a new value until the processing rule has decremented the counter `bufCount`, which keeps track of the number of values stored in the input buffer.

5.3.2 Module Interface

Prior to processing any input data, the $p \times p$ values comprising kernel K are streamed into the module and stored as a $1 \times p^2$ vector of registers with the following mapping:

$$K = \begin{bmatrix} k_{0,0} & \cdots & k_{0,p-1} \\ \vdots & \ddots & \vdots \\ k_{p-1,0} & \cdots & k_{p-1,p-1} \end{bmatrix} \rightarrow K = \begin{bmatrix} k_{0,0} & \cdots & k_{0,p-1} & \cdots & k_{p-1,0} & \cdots & k_{p-1,p-1} \end{bmatrix}$$

After initializing, input pixel values are passed into the module via an LFIFO. The relationship between the

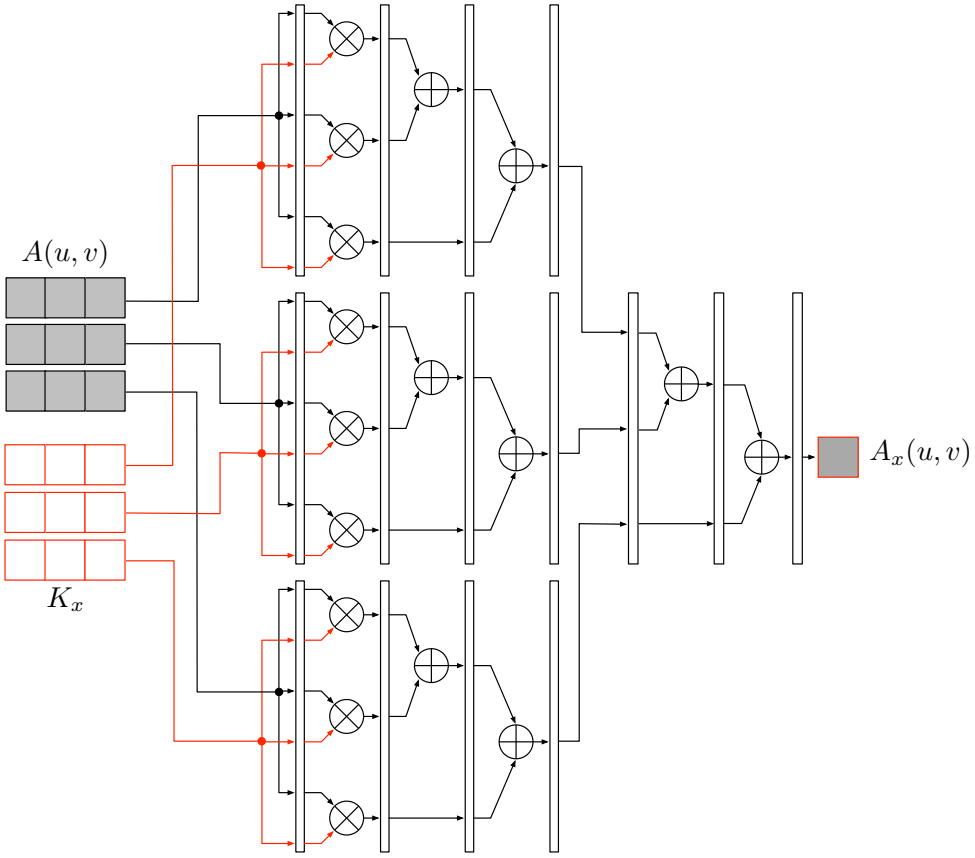

Figure 6: The convolution `processing` stage is accomplished using a linear inelastic pipeline. For a given set of (u, v), a $p \times p$ sub-set of input pixels values $A(u, v)$ is multiplied by the filter kernel K_x. The resulting products are then summed via an adder tree to obtain one element of the convolution output matrix.

2-D array representing input frame A and the pixel stream passed into the module is as follows:

$$A = \begin{bmatrix} a_{0,0} & \cdots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{n-1,0} & \cdots & a_{n-1,n-1} \end{bmatrix} \rightarrow A = \begin{bmatrix} a_{0,0} & \cdots & a_{0,n-1} & \cdots & a_{n-1,0} & \cdots & a_{n-1,n-1} \end{bmatrix}$$

As they are computed, elements of the convolution result array are passed out of the module via an LFIFO. Details of the interface implementation are included near the end of this section.

5.4 Lucas-Kanade Computation

The results from the previously described modules, which calculated I_x, I_y, and I_t for each input image, are fed into a submodule which performs the rest of the calculations to find the optical flow. This submodule performs the computation shown in Equation 1. The LK computation module is divided into three separate submodules (`IMult`, `IISum`, and `LKComp`) to simplify operation, and to allow for easier pipelining and submodule reuse.

5.4.1 Derivative Product Generation

The `Conv2` and `DT` modules produce the spatial (I_x and I_y) and temporal (I_t) derivatives respectively. However, as can be in seen in Equation 1 the products of these values (I_x^2, I_y^2, I_xI_y, I_xI_t, and I_yI_t) are required to solve for the optical flow field. The `IMult` module is responsible for generating these products. The current implementation simply reads the three input parameters from a FIFO and uses five multiplier FIFOs in parallel to generate the output result, which is placed in an output FIFO.

5.4.2 Neighborhood Summation

As previously described, the key assumption in the Lucas-Kanade optical flow algorithm is that the optical flow in a small local neighborhood of pixels is constant. This is assumption manifests itself as the summation of each element in Equation 1, and allows an otherwise underdetermined system of equations to be solved. The `IISum` module is responsible for performing this summation. In this implementation, a 3×3 local neighborhood is used.

The operation of the `IISum` module is very similar to the `Conv2` module, but is less flexible since it only needs to sum the values in a neighborhood, rather than convolve them with a filter. To perform this summation, an appropriate number of input values for each input product must be buffered. The size of the input buffer is a factor of the image and window sizes ($n = 64$ and $p = 3$, respectively) and is expressed as follows

$$\begin{aligned} \text{size(FIFO)} &= n \times (p-1) + p \\ &= 64 \times (3-1) + 3 \\ &= 131 \end{aligned}$$

The products generated by `IMult` are fed into a separate FIFO for each product. Once the FIFOs are fully buffered the appropriate elements from each are read and summed together to generate the required elements

of Equation 1. Figure 7 shows the relationship between the input images, the corresponding FIFO elements, and the buffered elements used for summation. A tree of adders, generated with the `fold` command, is used to perform the reduction of all the elements.

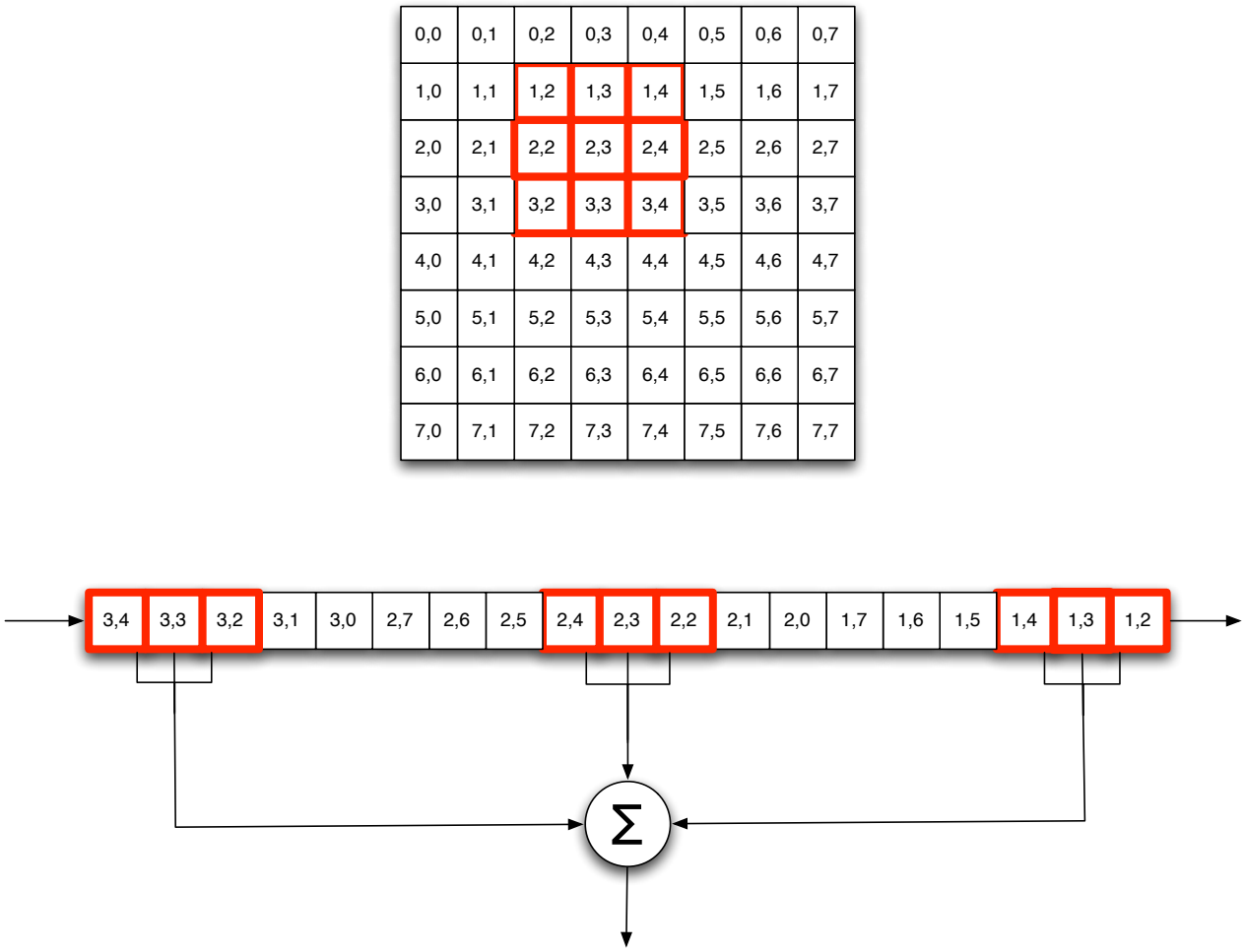

Figure 7: The matrix at the top represents the values generated from the multipliers calculating I_x^2, I_y^2, etc. The summation of a subset of these elements is desired. To calculate this, the outputs from the multipliers are fed into FIFOs, and the appropriate subelements of the FIFOs are selected and summed together. To enhance this structure, a tree of adders is used to perform the summation.

5.4.3 Matrix Computation

After the neighborhood summation, each element in Equation 1 has been calculated. The `LKComp` module takes these values and solves for the final optical flow vector result ($\vec{u}$). One key note about this module is that it requires fixed-point output due to the division introduced by matrix inversion. Until this point, only addition, subtraction, and multiplication are used, so all intermediate results are kept as integers. The output however is a fixed-point result.

Equation 1 is rewritten below eq:lkmatrixsimp with variables substituted for clarity.

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = - \begin{bmatrix} E \\ F \end{bmatrix} \tag{2}$$

Solving for $\vec{u}$ therefore becomes $u_1 = -\frac{DE-BF}{AD-BC}$ and $u_2 = -\frac{-CE+AF}{AD-BC}$. `LKComp` is responsible for solving for these equations. It does so by calculating both numerators and the denominator in parallel and places the results on the input FIFOs of the divider modules.

5.4.4 Pipelined Divider

Initial implementations of the optical flow design used a standard non-restoring division algorithm. However, this approach only allowed parametrization of the number of iterations per cycle, resulting in a divider that blocked all future division operations until the current operation was completed. This introduced many stalls in the pipeline during the division computation, which was undesirable. A more flexible divider was designed to replace the original one.

The new divider is still a non-restoring implementation, but in addition to specifying how many iterations per cycle to perform it also can be configured to use multiple pipeline stages so that multiple division operations can be running in the pipeline simultaneously. This is done by specifying the desired number of blocking clock cycles per stage. It is essentially a standard pipeline, with small folded pipelines contained at each stage (see Figure 8). The total number of stages used in the pipelined divider is determined by calculating

$$s = \frac{bw}{ipc \times cps}$$

where s is the number of stages, bw is the operands' bit width, ipc is the number of bit iterations per cycle and cps is the number of cycles per stage. The number of bits in the operands do not need to be multiples of the number of divider pipeline stages.

5.5 Host and DUT Interaction

Figures 9 and 10 show the details of the software and hardware interaction. Refer to these figures for the hierarchy, data types, and interfaces used in the test architecture. This portion of the design is adapted from the audio pipeline testbench used in the 6.375 course lab assignments.

5.5.1 Software

The software portion of the test architecture is shown in Figure 9. The input data files contain the input grayscale image pixel data. Two image frames are read at the same time so that two bytes are sent simultaneously (one from each frame) to the optical flow pipeline. Each byte represents the grayscale value for one pixel. The software testbench keeps the pixel data as an 8-bit wide stream as it gets passed to the FPGA via an `InportProxyT` object in the SceMi interface.

The `OutportProxyT` object that receives data via SceMi from the FPGA calls a method every time data is received so that it can be passed to an output file. Each output data value includes two 16-bit values.

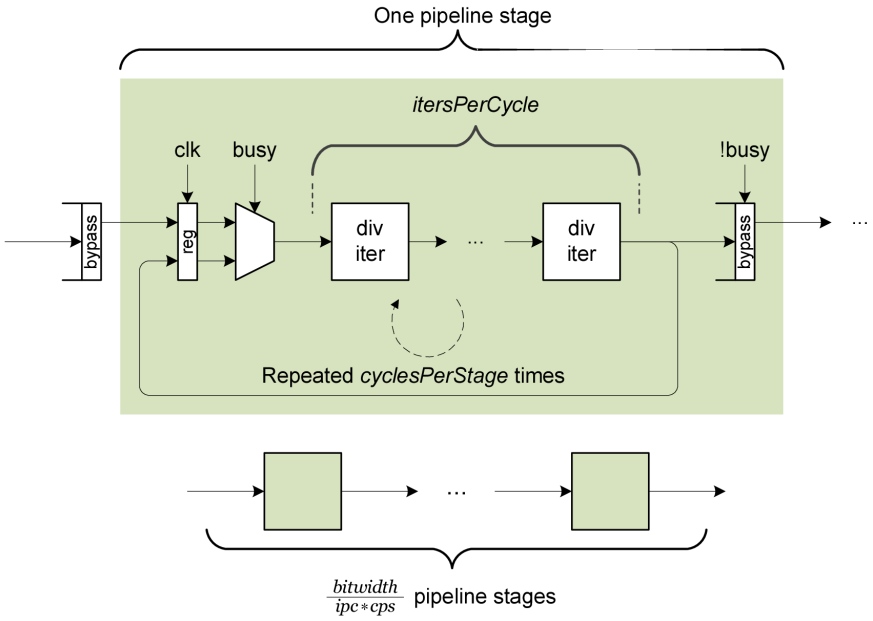

Figure 8: Pipelined divider architecture.

The two 16-bit values define the u_1 and u_2 components of the pixel motion vectors. They are written to the output file as signed fixed-point values (8 bits for the integer portion and 8 bits for the fraction).

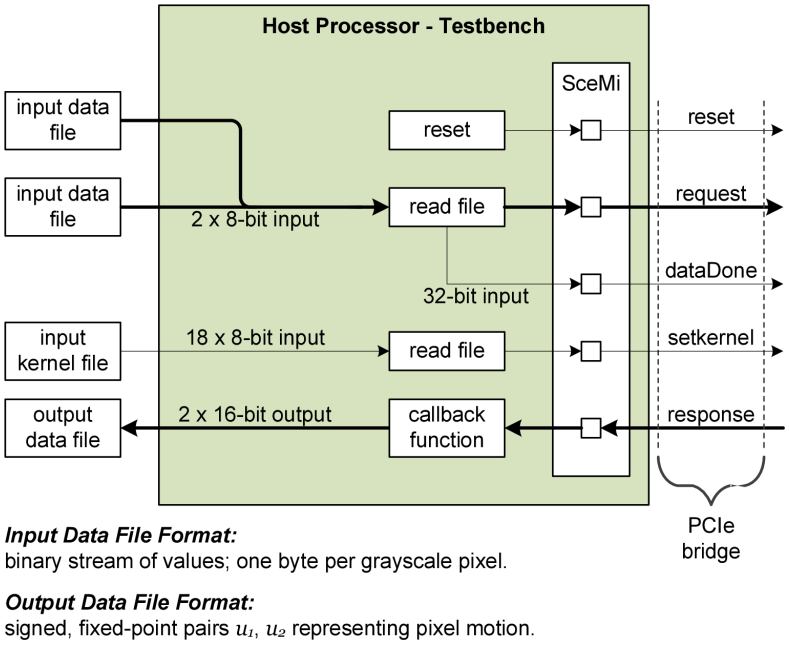

Figure 9: Software testbench structure.

5.5.2 Hardware

The hardware portion of the testbench (Figure 10) is connected to the software via SceMi. The design implements five SceMi ports: `reset`, `request`, `dataDone`, `setkernel`, and `response`. SceMi provides the PCIe bridge to connect these ports between the software and hardware components of the testbench.

The SceMiLayer module instantiates a wrapper which subsequently instantiates the optical flow DUT and its interface. The `Put` and `Get` interfaces are pre-defined in Bluespec^{TM}, and the `Action` and `ActionValue` will put and get the values to the DUT pipeline, respectively.

DDR2 memory and some FIFOs are used to buffer the input data, as shown in Figure 11. All of the input data is read and stored to the `DDR2` module before sending any data to the DUT. The `DDR2` module has a data width of 256 bits. The SceMi sends image data two bytes at a time, which are stored into a FIFO that is 16 elements deep. The data is removed from the FIFO and stored into a 256-bit shift register (again, two bytes at a time). Once all 256 bits have been shifted into the register, the register is then written to the `DDR2` module. Read responses from the `DDR2` module are saved into a FIFO, which eventually sends the data to the optical flow design. Data received back from the DUT is buffered in a FIFO before being sent back to the host (via SceMi).

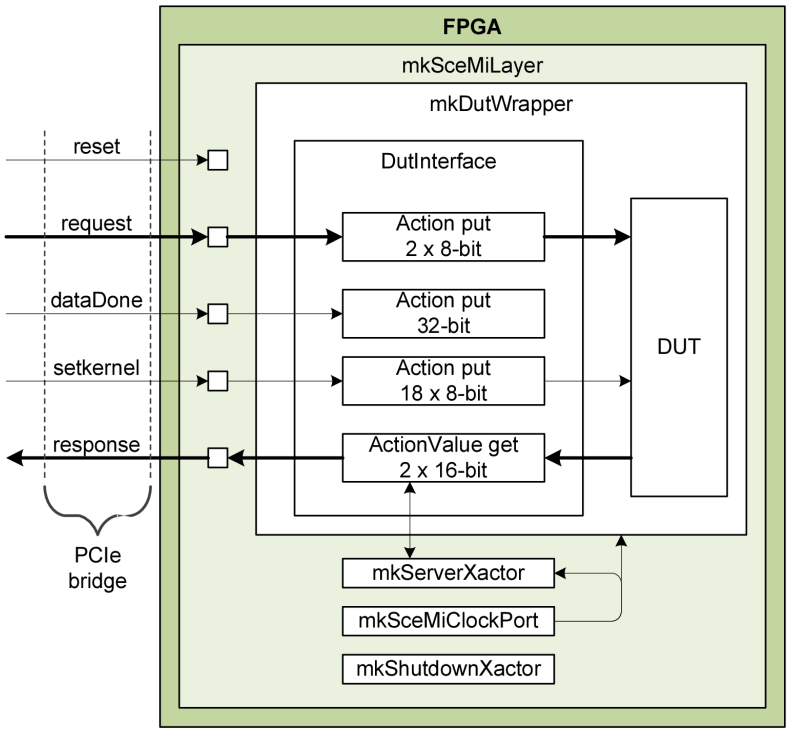

Figure 10: Hardware testbench structure.

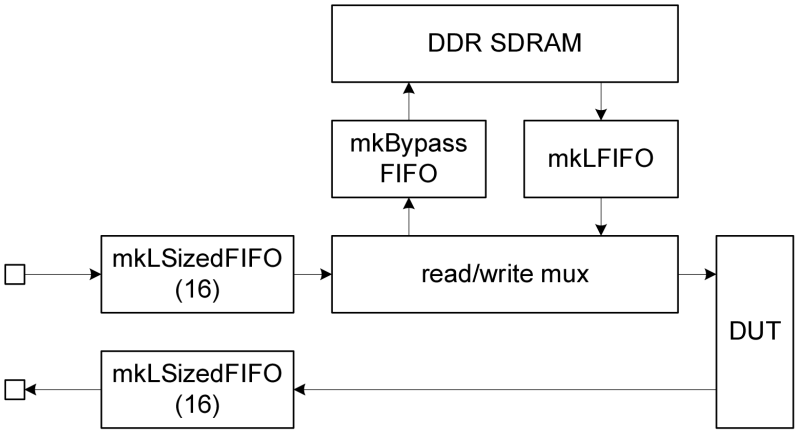

Figure 11: Input data buffering using DDR2 memory.

5.6 Interface Description

5.6.1 Convolution Module

The `Conv2` module implements the 2-D convolution operation. It also includes a testbench (`Conv2Test`) for verifying functionality. The `Conv2` module interface is as follows:

```
interface Conv2#(numeric type n, numeric type p, numeric type q);
        interface Conv2Engine#(n,p,q) convolve; //Convolution Engine sub-interface
        interface Put#(Vector#(TMul#(p,p),Int#(q))) loadKernel; //Load kernel
endinterface
```

where n is the dimension of the input frame, p is the kernel dimension, and q is the image pixel bit depth. Details of the `convolve` convolution engine sub-interface are described below. The `loadKernel` sub-interface is used to load the convolution kernel's elements during initialization. The 2-D kernel's contents are read in as a 1-D array ($1 \times p^2$) of signed q-bit integer values.

The convolution engine has the following sub-interface:

```
typedef Server#(
        Maybe#(UInt#(q)),       // Input pixel values one at a time
        Int#(TMul#(q,2))        // Output is signed integer with double the input data bit-width
) Conv2Engine#(numeric type n, numeric type p, numeric type q);
```

The engine streams in the input frame's pixel values (unsigned q-bit integer) one-at-a-time. The input values are tagged as valid by the data source. The engine returns the convolution result as a stream of ($2 \times q$)-bit signed integer values.

The `Conv2Test` testbench generates an 8×8 dummy image array containing values numbered 0 to 63. The kernel is a 3×3 array containing zeros with the exception of the first element, which is unity. During initialization, the testbench transfers the kernel's values to the `Conv2` module as a 1-D array. Following initialization, the testbench reads individual elements from the image array, tags them as valid, shifts them into the convolution engine input FIFO. Currently, the testbench obtains the convolution results as they are made available by the convolution engine.

5.6.2 Lucas-Kanade Computation

The `IMult`, `IISum`, and `LKComp` module interfaces are all implemented using the `Server` class, which provides `Put` and `Get` interfaces. The inputs and outputs of all three modules are connected to FIFOs, which makes connecting and passing data in the pipeline straightforward. The challenge is ensuring that the data types passed between modules are appropriately sized for the data they represent.

IMult The `IMult` module reads in the input data from the `Conv2` and `DT` modules and generates the appropriate products of the input. The input is represented by the `Is` struct, which contains integers representing the three input derivatives. The output struct is named `IProds` and contains ints representing the appropriate products. To ensure that no data is lost, the output data size is doubled to hold all possible values of the multiplication result.

```
typedef Server#(
    Is#(size),
    IProds#(TMul#(size, 2))
) IMult#(numeric type size);
```

IISum In the case of `IISum`, the input and output are both represented with the `IProds` struct. This struct contains the five products generated. In this case, `BITWIDTH_IISUM_OUT` is twice the size as `BITWIDTH_IISUM_IN`, again, to ensure that no precision is lost due to overflow.

```
typedef Server#(
    IProds#(BITWIDTH_IISUM_IN),
    IProds#(BITWIDTH_IISUM_OUT)
) IISum#(numeric type n); // n is input image size
```

LKComp The last module, `LKComp` takes in the `IProds` output from `IISum`, and produces two fixed point values. The results are contained in the `OF` struct, which is instantiated with `BITWIDTH_LKCOMP_OUT` as both the integer and fixed point sizes.

```
typedef Server#(
    IProds#(BITWIDTH_LKCOMP_IN),
    OF#(BITWIDTH_LKCOMP_OUT, BITWIDTH_LKCOMP_OUT)
) LKComp;
```

5.6.3 Pipelined Divider

The pipelined divider interface has one `Put` interface (for the division operands) and one `Get` interface (for the division result). The module also has two parameters: `itersPerCycle` and `cyclesPerStage`, as previously described in Figure 8. The interface is polymorphic, including support for fixed point types. The divider's interface is defined as follows:

```
typedef Server#(
    Tuple2#(word, word),
    Tuple2#(word, word)
) DividerPipelined#(type word, numeric type itersPerCycle, numeric type cyclesPerStage);

interface Put request = toPut(stgFIFOs[0]);
interface Get response = toGet(stgFIFOs[numStages]);
```

5.6.4 SceMi Testbench

The SceMi testbench has been set up to pass data from two input data files to the DUT, by first buffering the data to the `DDR2` module. The interface for the `DDR2` module is a 256-bit `Put/Get` interface that provides and retrieves data to/from the memory. The module that connects the SceMi with the DUT is called `PassThru`. The interfaces for the `PassThru` module are shown below:

```
typedef Server#(
    Bit#(px_data_size),
    VelOutput
) PassThru#(numeric type px_data_size, numeric type px_vec_size);

interface SettablePassThru#(
        numeric type px_data_size,
        numeric type px_vec_size
    );

    interface PassThru#(px_data_size, px_vec_size)  passthru;
    interface Put#(KernelPacket)                    setkernel;
    interface Put#(Bit#(32))                        dataDone;
    interface DDR2Client                            ddr2;
endinterface
```

5.6.5 Functionality

Verification of the optical flow module has been performed by comparing the results with a reference MATLAB^{TM}implementation of the algorithm. Functionality has been verified for a sequence of 64×64 input images generated from the Yosemite optical flow test sequence [4] using the SceMi simulator. The Yosemite sequence is a commonly used sequence of images with available ground truth that can be used to

test accuracy. Since the input images are larger than 64×64, a subimage of the full sequence is used to generate test inputs of the desired size.

A number of unexpected issues were encountered when testing.

1. The original input image consisted of a moving vertical gradient pattern. When testing with this pattern, the outputs appeared to be overflowing, as the result was all ones. However, it was determined that this was due to a divide by 0 issue. This occurred because the input images generated singular matrices, which the module would then try to invert. Since the determinant of these matrices was 0, a divide by 0 resulted. The solution was to use a different input pattern which would not result in this error occurring. A better way to handle this is to have the divider output 0 when a divide by 0 is encountered, since from the perspective of performance of the robot, it's safer to assume 0 optical flow instead of max optical flow. Another solution is to add a flag indicating that a divide by 0 has occurred, and to ignore the output. It's also possible to add a small amount of random noise to the input, to help prevent this case.
2. The initial stage of the pipeline calculates I_x and I_y using the convolution module, and I_t using a different method. The convolution module includes logic to avoid outputting invalid values around the edges. However, the I_t calculation did not include such logic. This resulted in the indices of the pixels used for the I_t calculation to not match those used by I_x and I_y.
3. At the end of the calculation, there is a large multiplication that occurs. The results from this multiplication were being stored in an Int of a larger bit size (`extend(A*B)`), however, the multiplication occurred at the input size of the multiplicands. This resulted in the product overflowing, even though it was sized correctly. The solution was to extend A and B before multiplying to ensure this overflow did not occur.
4. During initial testing, the values generated by the module were correct, but execution was ending too early and not generating the expected number of outputs. The solution was to pad the input with additional 0s to flush out the pipeline. This occurred because the `IISum` module did not originally perform valid index checking. It therefore generated results even at edges or before enough data was fully buffered. Later, the `IISum` module was updated to add additional boundary checks, to ensure that only valid data was produced. Therefore, no extra values were required to be pushed into the pipeline to get the appropriate number of results.

Functionality has been verified, but there are a few areas where verification could be improved. Currently, the test module does not include automatic value checking. Instead the output results are verified by hand. While this works acceptably for small image size, it does not scale well as image size and count increases. Adding output verification is a priority for future work to ensure functionality as further modifications to the design are made. Another issue is that the MATLAB^{TM} implementation is not currently bit accurate. Some work has been done to resolve this using the Fixed-Point Toolbox, but has not yet been completed. This would also allow us to do experiments with tradeoffs between bit-width and accuracy at a higher level.

6 Implementation Evaluation

We compiled the Bluespec design using version 2011.01.beta1 of the Bluespec toolchain. FPGA synthesis was performed using Xilinx ISE 11.5 and targeted the XUPV-LV110T board. Results from synthesis are presented below.

Module	Slice Registers	LUTs	DSP48E
Total (SceMi + Pipeline + DDR)	42911 (62%)	56435 (82%)	29 (45%)
`OFlowPipeline`	29907	45521	29
`Conv2`	112	366	0
`IISum`	1986	2595	0
`IMult`	89	1711	5
`LKComp`	161	516	24
`divPipelined`	13466	20370	0

Table 2: Total pipeline device utilization.

6.1 Convolution Module

In linear pipelined form, the four concurrent implementations of the convolution module utilized the majority of DSP48E slices, where utilization scaled as $4 \times p^2$. Recognizing that the kernel values for a Sobel filter ($p = 3$) consist of $\pm 2, \pm 1$, and 0, the multiplication operation was replaced by simple logic. Specifically, multiplication by 2 was replaced with a single bitshift, and negation is accomplished by evaluating the two's complement of a given value. This implementation requires no DSP48E slices at the cost of generality.

When computing the sum of products, the Bluespec^{TM} `fold()` function was used to automatically generate an appropriately sized binary adder tree during compilation. Initially, it was expected that an adder tree module would have to be designed, thus the existence of this function greatly reduced the time required for design and debugging of the convolution module.

6.2 Full Pipeline

The current state of the design successfully completes synthesis and place-and-route. The current design has bits flowing end-to-end both when run in simulation and when run on the FPGA.

6.2.1 Area

Table 2 shows the slice, LUT, and DSP48E (multiplier) usage for the synthesized design with 64×64 input images. The total utilization, along with utilization of several of the submodules is presented. The design is able to complete synthesis and fits on the FPGA. The DSP48Es are currently close to fully utilized with the `Conv2` module using a large percentage of the total.

6.2.2 Timing

The reported maximum frequency from synthesis is 71.018 MHz. The current critical path is in the divider. This could be adjusted by changing the number of cycles that the divider uses. However, this frequency already gives significant margin for meeting the desired throughput, so no further adjustments to the critical path are necessary.

6.2.3 Cycle Count

In simulation, the current optical flow pipeline implementation is able to process one optical flow computation every 96 cycles. For a 64×64 pixel resolution, this equates to 205 frames-per-second when running at 71.018 MHz, which is well above our target of 100 FPS. Note that only 3600 computations are performed for a frame comprised of 4096 pixels. The latency through the pipeline is 607 cycles, or 8.5 µs. When synthesized on the FPGA, performance characteristics initially matched those observed during simulation. Over time, the SceMi responses began to stall the pipeline, resulting in degraded performance with only 26 FPS. Future explorations might fix this discrepancy by multiplexing the `DDR2` module so that it also buffers the output data in addition to the input. Alternatively, one might implement a SceMi output interface that sends a larger data type so larger data transactions occur during each SceMi response.

6.3 ASIC Synthesis

As a point of reference, we synthesized the generated Verilog for the design using Synopsys Design Compiler, an RTL to gate synthesis tool. Version D-2010.06-SP1 of the tool was used. This tool can also generate preliminary area and power estimates. The design was synthesized for UMC 130 nm process, with a supply voltage of 1.2 V, at a target frequency of 50 MHz. The reported area for the design was estimated to be 55 350 μm^2. Static power was estimated as 1.8296 mW and dynamic power was estimated as 3.6967 mW, for a total reported power consumption of 5.5263 mW. This compares favorably with the original design target of 10 mW, and using a more modern process would lower power consumption further.

One issue encountered is that synthesis did not appear to complete correctly, with warnings about unresolved references to FIFOL blocks. Unfortunately, the current standard cell library we have available for synthesis does not include an implementation for LFIFO modules, so it is unlikely that the design could be implemented without some modification to the original implementation. Replacing LFIFOs with FIFOs would likely solve the issue, but would add additional cycles into the pipeline.

7 Design Exploration

Key performance metrics of this algorithm are frame rate (maximum throughput), power consumption, area, and memory usage. The design should aim to meet the desired performance requirement (∼100 FPS) while minimizing the other metrics. There are several places where the amount of parallelism and reuse can be varied. There is great opportunity for design tradeoff exploration. The straightforward approach would be to implement separate, parallel modules for each of the computations previously described. However, this would result in a design with potentially large area, and might not fit on the FPGA.

To improve upon this, we explored the amount of pipelining possible in the algorithm. For example, since the x and y derivative values are reused only for the pixels in the surrounding neighborhood, the amount of temporary storage can be reduced by a significant amount. Our aim was to find the amount of module reuse to maximally reduce area that still allows the design to meet the minimum performance requirements.

7.1 Convolution Module

Two different architectures were explored for the processing stage of the convolution module. A set of two adder trees provides a high-throughput design for performing the convolution sums.

A semi-circular pipelined variant of the original linear inelastic pipelined convolution module was added. The original structure required p^2 concurrent multiplications when computing one element of the convolution output matrix. The so-called semi-circular structure requires only p concurrent multiplications but requires p cycles to compute one element of the convolution result. A block diagram illustrating the semi-circular architecture is provided in Figure 12.

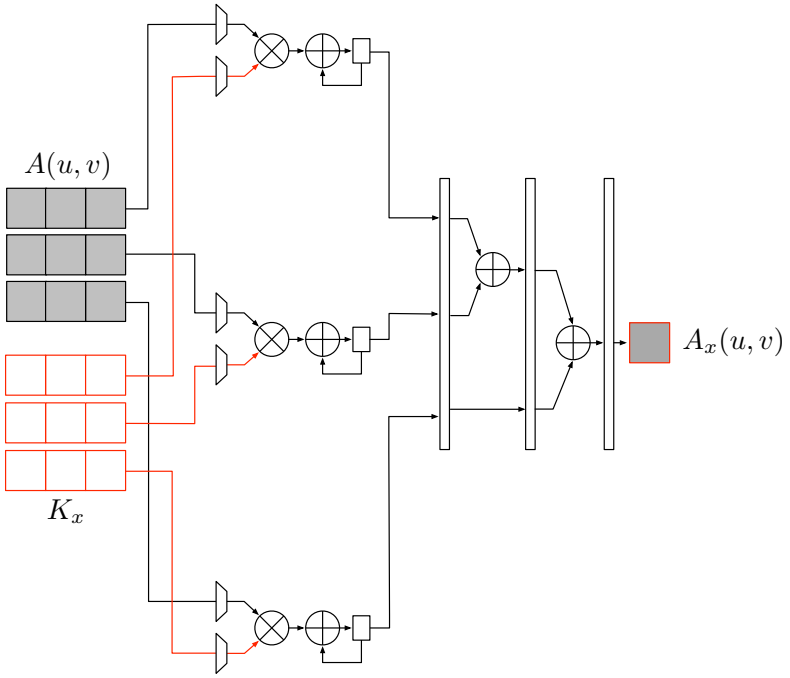

Figure 12: The semi-circular convolution pipeline architecture reuses multipliers, at the expense of increased LUT and slice register utilization.

Table 3 compares the overall device utilization when the pipeline is synthesized using the linear inelastic and semi-circular convolution modules. DSP slices that are no longer required for the semi-circular convolution stage may instead be used elsewhere in the pipeline.

Totals	Linear	Semi-circular	Linear-bitwise
DSP48E	36/64 (56.2%)	12/41 (29.3%)	0/29 (0%)
LUTs	848/56547 (1.5%)	8472/64464 (13.1%)	1464/57005 (2.6%)
Slice Registers	348/42728 (0.8%)	4800/46831 (10.3%)	448/42911 (1%)

Table 3: Percent of design utilization for various convolution pipeline structures ($n = 64$, $p = 3$). Each value represents four instantiations of `Conv2`.

The architecture selected for the final design implements the linear pipeline architecture, however the multiplications have been replaced by bitwise operations by exploiting the type of kernel used in this application. This design provides the benefits of high-throughput without requiring any DSP48E slices.

7.2 Lucas-Kanade Module

The bulk of the Lucas-Kanade computation module is in the neighborhood summation (`IISum`) submodule. Therefore, this submodule was a primary focus for potential design exploration. Currently, each of the 5 products from the previous submodule are buffered in individual line buffers. These line buffers are currently implemented as arrays of registers. Since using slices for storage can be very expensive on the FPGA, this may not be the ideal design. A possible solution could be to use the FPGA's BRAMs to hold the line buffers. The use of BRAMs should be more space efficient on the FPGA, but will complicate the interface and perhaps add some latency as not all elements will likely be able to accessed in a single cycle. However, the design was not sufficiently constrained by slice utilization to warrant this modification, and so remains an item for future work.

7.3 Divider Module

The divider takes up a large portion of the overall FPGA registers and LUTs. As such, there was interest to explore how the different divider parameters would affect the area usage. The `itersPerCycle` parameter was kept at 1 because increasing its value had too large of a negative impact on the maximum clock speed. This exploration therefore consisted of only changing the `cyclesPerStage` parameter. There is a roughly linear relationship between the `cyclesPerStage` divider module parameter and the area used by the dividers. The `cyclesPerStage` parameter was tested with values of 16, 20, 24, 32, and 48. Table 4 shows the resulting resource utilization. Note that the input operands are 48 bits wide, so using a `cyclesPerStage` value of 48 generates a single folded pipeline stage that fully blocks newer operations until it is completely finished with current operation. `cyclesPerStage` was chosen to be 16 since it allows for the more pipeline stages than any of the other options in the table, thus increasing the throughput the most. Using values less than 16 would tend to have timing errors reported during synthesis, so lower values were avoided.

`cyclesPerStage`:	16	20	24	32	48
DSP48E	0	0	0	0	0
LUTs	40693 (59%)	33694 (49%)	26669 (39%)	19635 (28%)	12626 (18%)
Slice Registers	26940 (39%)	22286 (32%)	17628 (26%)	12972 (19%)	8316 (12%)

Table 4: Total device utilization for two divider pipeline structures ($n = 64$, $p = 3$).

7.4 SceMi Testbench

There were two main areas of planned exploration for the SceMi testbench. First, evaluating how the performance is affected by writing more or less data to the RAM before transitioning to read mode. Second, how large to make the FIFOs that sit between the RAM and the DUT.

To address the write-to-read transition question, it was concluded that it would be best to try and mimic the streaming of data from a camera, making it necessary to try to separate the DUT performance from the

SceMi-to-DDR performance. Therefore, the entire image data files are buffered into the DDR before reading any of it back to the DUT. The exploration regarding sizes of the FIFOs after the DDR was not needed because the large output FIFO that is already part of the `DDR2` module design provided sufficient read buffer space.

8 Conclusion

Overall this project was a success. We were able to fit our design on the FPGA whilst exceeding the target performance specifications. Initial synthesis results also suggest that the design would satisfy the stringent power constraints posed by the RoboBee project. Our hope is to eventually include this design as an ASIC that will serve as the RoboBee brain. The ASIC would be composed of this and other similar accelerators, with each accelerator targeted at different task or workload. Key questions in that design will be how best to connect multiple accelerators, and balancing programmability with computational efficiency.

Using Bluespec^{TM} in our design flow proved to be invaluable in getting a working system ready in a relatively short amount of time. It allowed design efforts to focus more on the optical flow algorithm and the associated design explorations. Additionally, the wide range of reference code and library modules helped provide a seamless transition from software simulations to FPGA synthesis. These same benefits will be important as the design is taken further with the RoboBee project and its ASIC implementation.

Acknowledgments

We would like to express our sincere gratitude to Arvind, Richard Uhler, and Abhinav Agarwal for providing invaluable feedback and guidance throughout the course of this project.

References

[1] *Bluespec, Inc.*, `http://bluespec.com`, Accessed May, 2011.

[2] *Harvard RoboBee Project*, `http://robobees.seas.harvard.edu`, Accessed May, 2011.

[3] *MIT 6.375*, `http://csg.csail.mit.edu/6.375/6_375_2011_www/index.html`, Accessed May, 2011.

[4] *Yosemite Optical Flow Sequence*, `http://www.cs.brown.edu/~black/images.html`, Accessed May, 2011.

[5] M. Karpelson, J.P. Whitney, G.-Y. Wei, and R.J. Wood, *Design and fabrication of ultralight high-voltage power circuits for flapping-wing robotic insects*, to appear: Applied Power Electronics Conf., 2011.

[6] B.D. Lucas and T. Kanade, *An iterative image registration technique with an application to stereo vision*, International joint conference on artificial intelligence, vol. 3, 1981, pp. 674–679.

[7] M Srinivasan, S Zhang, J Chahl, G Stange, and M Garratt, *An overview of insect-inspired guidance for application in ground and airborne platforms*, Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering **218** (2004), no. 6, 375–388.