Difference of Gaussian Scale-Space Pyramids for SIFT[1] Feature Detection

Ballard Blair and Chris Murphy

Group 5

6.375: Complex Digital Systems Design

Final Project Report, Spring 2007

Introduction

The Scale-Invariant Feature Transform (SIFT[1)] algorithm has rapidly been adopted in the machine vision community as the "best-in-class" standard for feature detection and matching. Image Feature detection has a variety of applications across many domains, from object recognition and tracking in robotics to creating photo mosaics in consumer photography applications.

Developed by David Lowe in 2003 at the University of British Colombia, the SIFT algorithm takes a single image as input and returns a set of image features. Each point in the set has a location, scale, orientation, and description vector. Compared to other feature detecting algorithms, the Sift algorithm's description vectors and identified points are relatively invariant to illumination, scale, and viewpoint changes. This makes the algorithm extremely useful for tasks like object recognition where the object or camera may be moving, or image registration, the process of determining a shared coordinate frame between images.

A hardware implementing of SIFT would provide the timing and power requirements necessary for a real-time, mobile system, something current software implementations don't provide. Current SIFT software implementations typically involve the use of a high-power, general-purpose processor to achieve less-than-real-time performance (one the order of seconds per image). For many embedded applications, something that achieves sub-second performance without high power requirements is essential. With a hardware implementation, the full potential of the SIFT algorithm could be available to mobile sensing platforms without the overhead of a dedicated high-power processor.

We have implemented a subset of the SIFT algorithm, a Difference of Gaussian Scale-Space image pyramid generator. This operation forms the computational "core" of SIFT.

Our implementation takes an input image, repeatedly blurs it a fixed number of times, and computes the difference between successive blurring iterations. One of the resulting images is then down-sampled and the blurring/difference operation repeated. The result is a collection of "Difference of Gaussian" images, several for each scale.

The power of the "Difference of Gaussian" operation provides an efficient discrete approximation to the Laplacian. The Laplacian is formally defined as the divergence of the gradient, which in image processing translates to areas of high variation, characteristic of interesting features. These Laplacian or Difference of Gaussian pyramids were first identified in 1984[2], and since then have been adopted for a variety of applications in image processing well beyond their role in the SIFT algorithm.

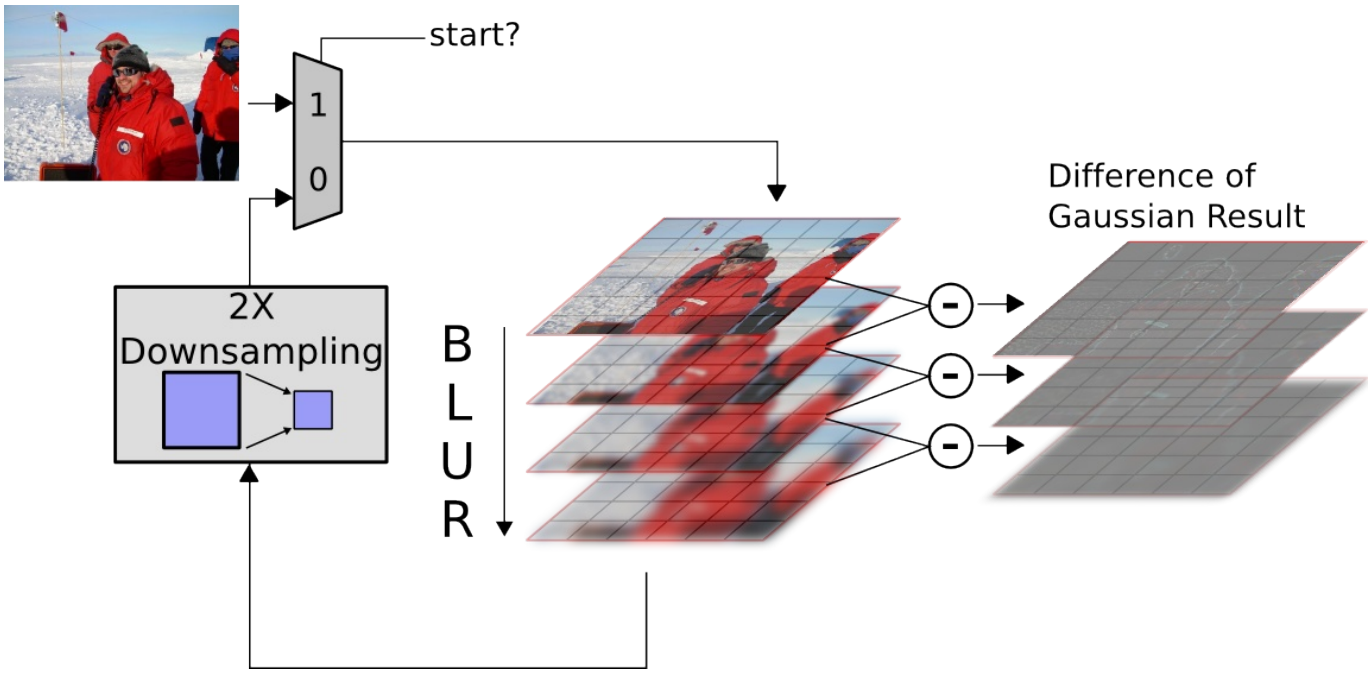

Figure 1: Architectural Overview

Algorithmic Overview

The Difference of Gaussian or Laplacian pyramid is generated from a single input image. The output is a 'pyramid' of several images, each being a unique difference of Gaussian. To generate the pyramid, the input image is repeatedly blurred; the difference

between consecutive blur amounts is then output as one “Octave” of the pyramid. One of the blurred images is down-sampled by a factor of two in each direction, and the process occurs again with output in a different size. Figure 1 illustrates this process in greater detail.

Architecture Overview

Our goal in implementing the difference of Gaussian module was to create an architecture that was flexible enough to allow for some architectural exploration but rigid enough to be implementable in hardware. The architecture we decided upon was five parallel convolution units, identical except for the coefficients of the Gaussian coefficients.

Rather than attempting to do a two-dimensional block convolution, which would require a large number of multiply units, we exploit the fact that a two-dimensional Gaussian convolution has the property of separability; we can perform first a vertical convolution followed by a horizontal convolution, and the result is identical to performing a full 2D convolution. In the current implementation, each convolution unit requires a frame buffer large enough to hold the entire image, two address generators, one for read and one for write, a one dimensional convolution unit hard wired with coefficients, and some FIFOs and control logic to tie everything together.

In addition to the convolution units we needed additional control units to handle image reading, down-sampling, and feedback, an additional memory and address generation units to store the original image for processing, and a difference unit to calculate the difference between neighboring images. Figure 2 shows a top level block description of how our blocks fit together. Later sections describe the functionality of the memory and some of the more important rules and functionality. For a detailed description of each file, see Appendix A.

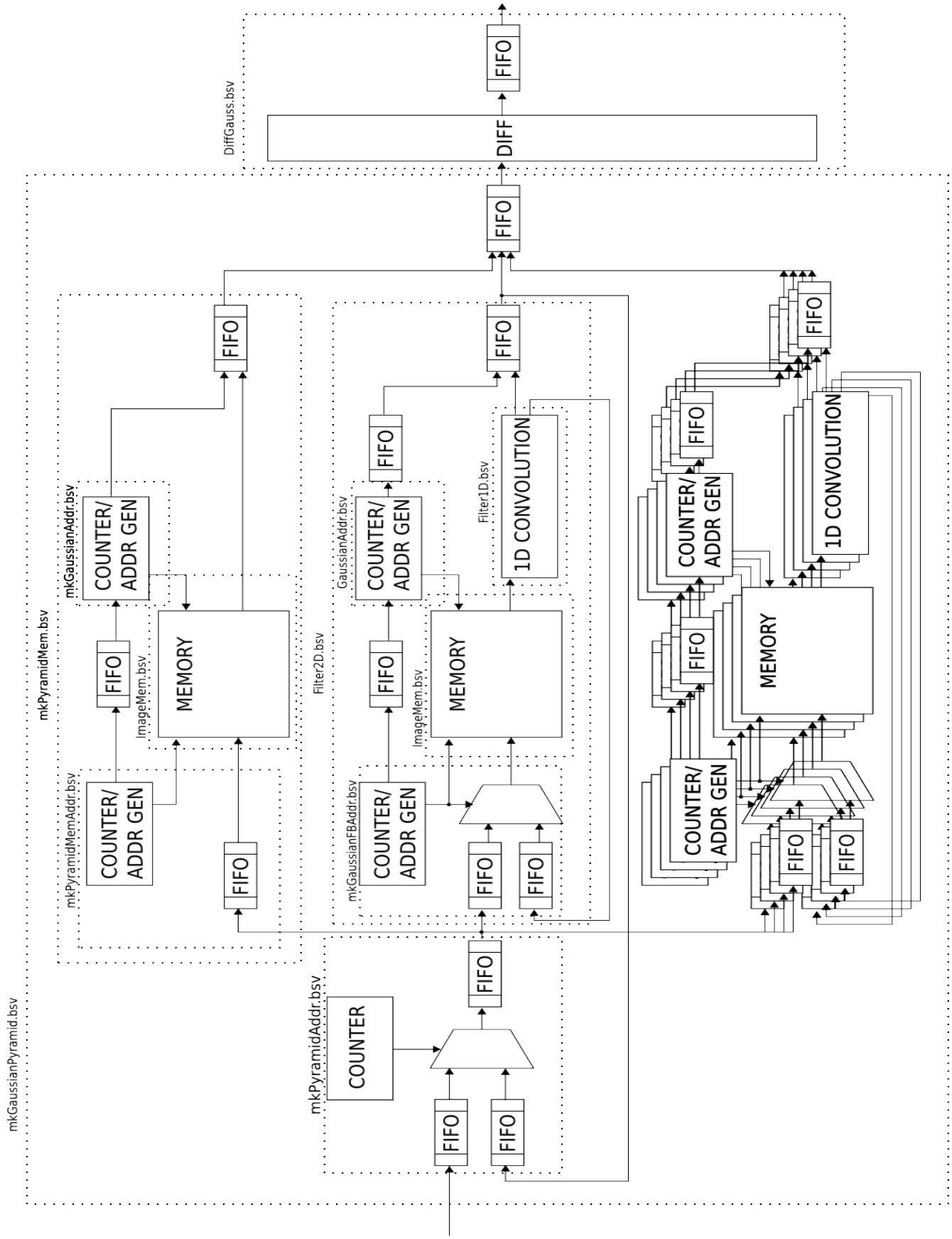

Figure 2: Difference of Gaussian image pyramid schematic and file hierarchy

Memory Management and Addressing

Our target design included an image size of 480 pixels by 360 pixels (pixel = 8 bits), separable convolution, and enough memory in each blur unit to store an entire image. Therefore, we needed to implement a memory that was large enough to hold 172,800 bytes. The Synopsis tool chain includes a memory generation tool, but this tool is limited to blocks of memories of 4096 words or less. In order to meet our target memory size while still leaving us the flexibility for architectural exploration through pixel precision changes we used 43 blocks of 4096 8-bit words.

The Bluespec memory block instantiates 43 copies of the RAM modules, which the memory generate creates in Verilog. Wrapping each of these in a Bluespec interface, we can simply pass the bottom 12 address bits along to the memory with clever chip select and enable lines to each memory to allow for simple, clear, interfaces to the memories. We included a total of 18 bits in our ImageMemAddr data-type to allow for some flexibility in the actual size of our memories.

Since the block RAM modules created using the memory generator are synchronous, there is a one cycle delay for either a read or a write. We did not need to worry about read after write data hazards in our memory since our design did not allow for such collisions. However, we did need to be careful with the timing for our reads.

We combine a simple rule, moveDataMemConv and a data type in Filter2D, which determines if the previous pixel was in the image (i.e. not just for convolution buffering, see convolution section). If the previous pixel read request corresponds to a pixel that was in the image, it enqueues the pixel value along with the pixel information into a convolution FIFO. If the pixel is only for convolution buffering, only the value is pushed onto the queue. To handle the latency in the memory reads, the rule determines if the current pixel request is the first, last, or some middle read request. If it is the first, it only issues a request with no

read, and if it is the last, it does not issue a request while it is reading it from memory. Otherwise it concurrently reads and requests.

Convolution Description and Details

The convolution block is at the heart of our difference of Gaussian block. For our design we decided to take advantage of the property of separation for the two-dimensional Gaussian blur by performing two one dimensional convolutions. The number of coefficients, or “taps” we used for each convolution block is the same (our final implementation had 33 taps), and the coefficients are read in from a function stored in a separate file. This allows us to quickly change the coefficients if need be without having to change any of our Bluespec directly.

The simplest description of a convolution that aptly describes what our block accomplishes is a weighted average. The coefficients describe the weightings which are multiplied by the current and surrounding pixels. The results are all summed together and output as a new value for the current pixel location.

In order to reduce edge effects (numerical error affecting pixels on the edges and corners of the image) we have implemented a “reflected” convolution, whereby the convolution unit is filled with a mirror image of the pixels that come after or before the edge pixel. For example, if we number the pixels consecutively we would first read in pixels 5,4,3,2 before we read in pixel 1. Next pixels 1,2,3,4 and 5 would be read in so the unit contains 5,4,3,2,1,2,3,4,5. The pixel value that is then output for pixel one is a weighted mirrored average which helps to reduce the number of false features detected on the edges in the SIFT algorithm, and matches typical software implementations.

The convolution is all preformed in the Filter1D block which contains a shift register, 33 multiply units, and a accumulator. If the pixel is not one of the buffer pixels described

above, there is a FIFO which holds other information about the pixel, such as position in the original image, whether it is the last pixel in the image, and current scale of the pyramid (i.e. how many times the image has been down-sampled). All pixels are read from memory and in order to perform the two dimensional convolution, first the image is read vertically from the memory and then it is read out horizontally and output to the difference block.

Image Down-sampling

Performing a two dimensional convolution with a variance of 3.2 pixels we lose at least half of the information per pixel. Therefore, after we have performed a two dimensional Gaussian, we can feed back a down-sampled version of the image that has been blurred by a Gaussian of variance 3.20. This will accomplish the same effect as if we had continued to blur images at the same scale, but will use less computational effort and allows for our hardware to be pipelined. This is the process used in software implementations of an "Image Pyramid" as well.

The image down-sampling is performed by throwing away every other row and every other column. Thus we are throwing away ¾ of the pixels, but only removing half of the information from a row or a column. The unit that performs the down-sampling uses counters to keep track of the row and column position so that it can accurately throw away the correct pixels for each scale. The scale in this case corresponds to the number of times an image has been down-sampled, so we start with scale 0, then scale 1, etc.

The size maximum number of down-sampling we can perform is a function of our original image size and the number of taps for the convolution. We need the number of taps to be less than the length of a row or column. This is why for an image of 360x480, we can only down sample 3 times (for a total of 4 scales). If we down-sampled any further, the length of the rows would be less than 33 (after 3 down-samples, each row is 45 pixels long).

Implementation Notes

When we originally started our design, we thought it would be possible to have one control for both the reads and the writes of the memory units. This turned out to over-complicate the logic and having separable logic quickly became the obvious answer. The problem with having coupled logic for both the reads and the writes is that while computing the horizontal blur, the memory is being simultaneously read and written to.

Our first thought was that we needed the coupling to protect the integrity of the data. However, the data was protected due to the nature of the feedback, and the only protection was that a new image could not be read in until the current image buffer had been flushed. This is a conservative requirement, but allows for a simple scheme to protect the data integrity.

Another issue involves the current way we are addressing our memories. Since we decided to have separable convolution, we needed a somewhat complex scheme for addressing the vertical image, since the data is being read in contiguously. This precluded us playing obvious games with the word size of the memory, such as switching to 32-bit blocks in the memory to reduce the number of memories we required by four. In order to make this change, we would have had to either devised an entirely new scheme for memory storage that accommodated both horizontal and vertical reading, or decreased the performance by a factor of two or more due to the latency of the memory and the byte masking complications. These trade-offs might still make interesting future explorations but time constraints prevented us from exploring them too much further.

Testing Procedure and Results

During development, we performed a variety of tests at each step of development. These tests often gave us direction in future development, leading us to, for instance, use 16

bit fixed point values during the convolution's multiplication step. We were able to easily perform these tests due to Verilog modules we wrote to read and write PGM gray scale images, vImageStreamReader and vImageStreamWriter. We tested the results of our Gaussian blur against results from a MATLAB implementation, and found that the results were excellent. If the same Gaussian blur was performed using full double-precision arithmetic in MATLAB and then converted back to a standard 8 bit image, we found that there was no error for many pyramid levels, as shown below.

Figure 4: The image used during testing

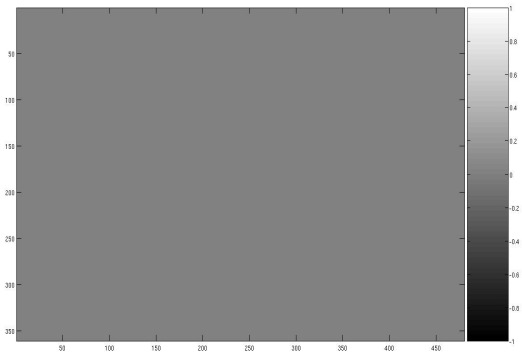

Figure 3: Gray represents zero error. White or black would represent an error of 1 gray level.

The results from the full operation of our Difference of Gaussian Pyramid Generator were also positive, although there remains some error compared to the DoG pyramid generated by a MATLAB implementation of SIFT. We believe this error stems from improper calculation of the amount of Gaussian blur to apply at each level of the pyramid, and numerical error. In software, the pyramid is generated by iteratively blurring a single image; this saves some computation time, but does not allow for any parallelization. We perform all of the blurs for a given image size simultaneously, which means that we must blur the image with different sized Gaussian. Although it is possible that we calculated these blur amounts incorrectly, (we didn't have time to consult with somebody who could confirm our math) the architecture we designed is flexible enough that changing the coefficients is trivial. The difference between two of our generated Gaussian images,

followed by the results of the full DoG hardware pyramid generation are shown below; contrast has been increased to improve visibility.

Figure 5: Difference between two Gaussians generated by hardware.

Figure 6: All DoG levels generated by hardware; contrast has been increased.

Synthesis Results

Using Synopsis synthesis tools, combined with some automatic scripts we obtain the following results for the top-level design:

Module	Size (λ^2)
mkSift	9111679.0
mkDiffGauss	1267.75
mkGaussianPyramid	9110410.0

We can see that the majority of the area for the synthesized design is in the mkGaussianPyramid module. This is to be expected since this module contains the memories. In order to synthesize the memories, we used the memory generator, so these memories are of an optimal size. For completeness, we include the numbers from the mkDiffGauss module, since it is nearly a top level module. We can see from the results that the majority of the mkDiffGauss unit is consumed by one FIFO.

Module	Size (λ^2)	Multiplicity
mkDiffGauss	9111679.0	1
FIFO	1035.25	1
sub	46.5	4

Finally, examining the synthesis results from the mkGaussianPyramid block, we note that obviously our size is dominated by memories. The remainder of the area is roughly divided between the multiply units, the FIFOs and all other logic. This shows us that unless we somehow removed the memories, it is not worth exploring anything other than something which will reduce our memory requirements. However, if we did switch to off chip memory, we would want to first explore reducing the size of our convolution and reducing the number of FIFOs that we use in our design to get the maximum impact.

Module	Size (λ^2)
mkGaussianPyramid	9111679.00
mkImageMem_0	1506599.75
mkImageMem_1	1506599.75
mkImageMem_2	1506599.75
mkImageMem_3	1506599.75
mkImageMem_4	1506599.75
mkImageMem_5	1506599.75
multiplication units	20892.25
FIFOs	22191.25
all other logic	28997.00

According to the synopsis tool chain, all of our modules were able to meet our timing constraint of 5ns, which corresponds to a clock speed of 200MHz.

Clock Period Constraint	5ns
Max Clock Speed (synthesis)	200MHz

Our longest path is from a register in our shift register through a multiply unit, an add unit and finally to a FIFO. This is not surprising since the memory was set to "don't touch" so it was not included in this analysis, and it makes sense that the multiply units would be part of the longest path.

Place and Route Results

Unfortunately, due to the way the current RAM generation tools are set up, we could either hand place all 258 rams manually or we could write a TCL script to help place them automatically. We did not have time to place all 258 rams manually. We also could not find any more than the size of each memory block in the files produced by the memory generator, so we were not able to automate this process or get final place and route numbers using the RAMs. However, since we know that each ram is 35016.3, we know that the total RAM size

will be approximately 9,000,000 um^2. This is at least 10 times bigger than the result we get for all of the other logic from P+R. If we had the time to determine how to appropriately place the RAMs, given the time constraints, we felt the time was better spent refining our design. Furthermore, if we were to go forward with this design, we would probably switch to off-chip memory instead of trying to have so much memory on-chip.

Total Area (w/o mem)	1058572.5 um^2
Total Gates (w/o mem)	112518
Estimated mem area from RAM-gen)	9,034,205.4 um^2
Estimated Tot Area (w/mem)	10,092,777.9 um^2
mkDiffGauss Area	15905.8 um^2
mkGaussianPyramid Area	1029978.4 um^2
Min Clock Period	6.415ns
Max Clock Speed (P+R)	155MHz
Approx. Clock cycles per image	~75000
Approx. Images per Second	~205

This table shows us that we can easily use our hardware for feature detection in images from a video stream (30 frames per second). Furthermore, the area, even including all of the memories (to the first order) is still reasonable. We could fit all of this onto a chip that is 10 mm^2 and get better performance for the DoG than is available using any current general purpose CPU.

Architectural Exploration

As this system was built from the base up, Blusepec allowed us to easily switch our architecture around as our system evolved. Significant architectural exploration was enabled during the development process, ranging from different memory configurations to different bit precisions. While many of these changes were trivial in Bluespec (like

redefining a single type) they could have been incredibly complicated to execute in a standard HDL.

Key explorations were to examine changing the precision of the pixel values for more better precision mathematics. Even though we were still reading in an 8-bit image, we thought we might get a lower error if we increased the precision to 16 bits due to the convolution operation. We discovered that the conversion back to an 8-bit image caused vastly more error than was possible to gain through intermediate precision.

As the Bluespec fixed point library supports truncation but not rounding, we created a rounding function which allowed us to have such low error in the Gaussian blur area of the code. This change was made after developing a significant amount of the system.

Future Improvements

There are several next steps now that we have shown proof of concept for our difference of Gaussian unit. First, we could use this unit to complete the SIFT tool chain. This would involve implementing a feature detector, and all other parts of the SIFT algorithm. This would not be terribly difficult, but would require some thought about how to further manage the memory so that we could retain all of the information we needed for the rest of the chain.

Another possible avenue is to explore different ways to manage the memory within the module we have already built. A streaming or blocking driven architecture that only works on part of the image could be an easy improvement that would have a large impact on memory and space requirements. Due to time constraints we unfortunately did not pursue these avenues further. If we continue this work it will be a logical next step.

Conclusions

We have shown that it is possible to implement the difference of Gaussian unit in hardware. This piece of hardware is the core of the SIFT algorithm and so with some more work we think it would be possible to implement the whole algorithm, opening up new avenues for research and possibilities in feature detection and robotic navigation.

We explored the design space by looking at ways to improve our numerical accuracy through both improving the precision of our computation and through the use of better rounding techniques. The results using only 8 bits of precision were still within one pixel value (1/256) of being correct, so the added area trade-off did not really seem worthwhile. We used simple rounding instead of truncation to improve our results slightly since this addition was at no great hardware cost.

Through our work we have created a complete tool chain that allows us to perform a difference of Gaussian operation in hardware along with test-benches such that we can read in and read out an arbitrary image from memory. We have also left our architecture sufficiently flexible such that it would be possible to continue to build upon and expand the hardware we have created to implement more complicated and higher functioning systems.

Thanks

The authors would like to thank the TAs and Professors for this class, in addition to the very helpful individuals on the Bluespec-support mailing list (Thanks Nirav, Mike!).

References

1. Lowe, David G. “Distinctive Image Features from Scale-Invariant Keypoints”. *International Journal of Computer Vision*, 60, 2 (2004), pp. 91-110.
2. Burt, Peter J. and Adelson, E. “The Laplacian Pyramid as a compact image code”. *IEEE Transactions on Communications,* 31, 4 (1983), pp. 532-540.

Appendix A. Bluespec File Descriptions

mkSift.bsv: This file is the top level of our architecture. It instantiates a difference of Gaussian pyramid block which includes 5 convolution blocks and a memory and outputs data packed as a ConvPacket (see the SiftTypes.bsv section for a description of data types). This data is streamed into a difference unit and finally gathered and sent out through an output interface.

mkGaussianPyramid.bsv: Top level file of the Gaussian image convolution pyramid. This file serves as a wrapper combining together control logic which handles either feeding in an outside image or feeding back an appropriate image from the convolution units. There is a rule to pass the data from the control logic into the convolution unit. There is a separate rule to feed back the data and to enqueue the data into an output queue.

DiffGauss.bsv: Difference unit which takes in a ConvPacket, a vector of 5 pixels plus some additional image information such as location in the original image, if it is the last pixel in an image stream, and the scale of the current convolution (see Image down-sampling section).

mkPyramid.Addr.bsv: Bluespec file describing two counters for reading in an image from the input, or some additional logic to down-sample an image that is fed back from the convolution block.

mkPyramidMem.bsv: Memory block for storing image while convolution is happening in convolution blocks. This file is very similar to Filter2D.bsv, except that there is no convolution unit, and the control block does not contain logic for reading the image out vertically or feeding the image back.

mkPyramidMemAddr.bsv: Control logic and address generation for the mkPyramidMem

block. The control is two counters combined with a look up table to determine the correct memory position based on the pixels placement in the image stream and the current scale.

Filter2D.bsv: Two dimensional convolution is implemented in this top level file. The data is passed to a control block that generates a memory address based on scale and position in the stream and then stores the pixel value in memory. The pixel is then read from memory by a separate block and fed into a convolution unit, first in vertical order (down the columns) and then in horizontal order. While the horizontal convolution is being computed, the image is fed to the output. While the vertical convolution is being computer, the image is fed back into its local image buffer.

mkGaussianFBAddr.bsv: Control logic for reading in an image from the input or feeding back an image from the 1D convolution unit. The control is two counters, one for row and one for column along with a lookup table for the appropriate step size based on the scale of the image. The scale is fed in from the outside, but the scale register is only updated at the beginning of an image. Thus it is not possible to accidentally change the scale in the middle of an image, so that the data is not accidentally corrupted.

ImageMem.bsv: A small amount of logic and a wrapper for housing the generated RAM modules from the Synopsis RAM generator. The bottom 12 bits of the input address are fed directly into the RAM blocks and the top 6 bits are used for chip enable lines to each of the blocks. There is as little logic as possible in this block, so all of the control for the memories, including read/write timing must be handled externally.

mkDataRam.bsv: Bluespec wrapper for the Verilog RAM modules generated form the Synopsis tool chain. This file is set to have one read port, one write port, and to have no conflict between simultaneous reads and writes (although if the same data is read and written in the same clock cycle, the returned data will be stale). Also, it is currently set up

for 8-bit words.

mkDataRam_h.v: Verilog wrapper for RAM module generated from Synopsis tool chain.

mkGaussianAddr.bsv: control logic for reading out the image stored in memory for the convolution unit. It is hard-coded to read the image first vertically and then horizontally. Also, this the output address type is a tagged union indicating when the last pixel is being read from memory to handle the memory read latency. The module includes methods to read the last address, whether the last pixel was in the image, the previous pixel position, whether the last pixel was the end of the image, and the scale of the last pixel. Note that the address generation is not simple linear in either the horizontal or vertical case since this file also handles pre-filling the convolution unit with reflected pixels to remove edge effects from the convolution.

Filter1d.bsv: One dimensional convolution block. This file include invokes a shift register, reads in coefficients from mkGaussianCoeffs1D.bsv (a generated file), and performs a one dimensional convolution with a fixed 33 taps. The one dimensional convolution can be used in concert with clever memory management to synthesize a two dimensional convolution.

ShiftReg.bsv: Simple parameterizable shift register, used in the convolution block.

mkGaussianCoeffs1D.py: Python file which creates mkGaussianCoeffs1D.bsv, a lookup table for the Gaussian coefficients. This file is necessary since Bluespec does not contain support for fixed point or floating point static elaboration.

SiftTypes.bsv: Description of data types used throughout files. See appendix B.

** Note that all files included in the source directory which end in "TH.bsv" or "TH_wrapper.bsv" are tests harnesses for the corresponding files. While they were used in development, some may have since fallen out of date as interfaces changed.*

Appendix B: SiftTypes.bsv

This file describes all of the data types used by our Bluespec design.

```
//*********************************************************************
 // Datatypes for the SIFT Algorithm
 //---------------------------------------------------------------------
package SiftTypes;
 import FixedPoint::*;
 import Vector::*;

// Data types for the Pixel values and 1-d convolutions
 typedef FixedPoint#(1,8) Pixel;
 typedef FixedPoint#(1,16) FilterCoeff;
 typedef FixedPoint#(2,8) DOGPixel;

// Some pixels (to deal with edge effects) should be used in calculations
// but should not have filtered values calculated for them.
 typedef struct { Pixel data; Bool compute; } FilterPixel deriving (Bits);

typedef Bit#(8) PxByte; // determines how bit of a chunk to use for pixels
typedef Bit#(9) FullPxByte; // determines how big pixel is when FP
typedef Bit#(18) ImageMemAddr;

// Tagged Union Type for Address
 typedef union tagged {
   ImageMemAddr ImagePixelAddr;
   void         LastPixel;
   void         NonPixel; // indicates no address was sent
 } ConvAddr deriving (Bits, Eq);

// State Types for the Convolution Block
 typedef enum { ReadIn, Vertical, Horizontal } GaussState deriving(Eq,Bits);

// size for counters in image
 typedef Bit#(9) CountType;

// for keeping track of scale
 typedef Bit#(2) ScaleType;
 // position of pixel in original image
 typedef struct { CountType row; CountType col; } PixelPos deriving(Bits);

// output of Difference of two dimensional convolution
 typedef struct { Pixel data; ScaleType scale; PixelPos pos; Bool eof; } SiftPixel
deriving(Bits);
 // Output of the Gaussian Pyramid block
 typedef struct { Vector#(6,Pixel) data; ScaleType scale; PixelPos pos; Bool eof; } ConvPacket
deriving(Bits);

// two identical data types for the output of the the difference of Gaussians block (different
implementations)
 typedef struct { Vector#(5,Pixel) data; ScaleType scale; PixelPos pos; Bool eof; } DiffPacket
deriving(Bits);
 typedef struct { Vector#(5,Pixel) data; ScaleType scale; PixelPos pos; Bool eof; } DOGPacket
deriving(Bits);

// Data type for memory address
 typedef struct { ImageMemAddr addr; PxByte data; } MemPktType deriving(Bits);

// Data type to pass into GaussianAddr for enable and state setup
 typedef struct { ScaleType scale; GaussState state; } ImRdyType deriving(Bits);
endpackage
```