Accelerating Image Processing Pipelines On FPGA Architectures

Undergraduate Thesis

Submitted in partial fulfillment of the requirements of
BITS F421T Thesis

By

Pasupuleti Suryachandra prasad
ID No. 2014B2A30949G

Under the supervision of:

Dr. Suresh Purini
&
Dr. Ashish Chittora

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, GOA CAMPUS
December 2018

Declaration of Authorship

I, Suryachandra prasad PASUPULETI, declare that this thesis titled, "Accelerating Image Processing Pipelines on FPGA Architecture" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: P. Surya

Date: 10-12-2018

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI,GOA CAMPUS

Abstract

Dr. Suresh Purini
Computer Systems Group(IIIT Hyderabad)

Electrical and Electronics Engineering

Accelerating Image Processing Pipelines on FPGA Architecture

by Suryachandra prasad PASUPULETI

Domain Specific Languages for Image Processing have been gaining popularity in recent years as they concentrate on improving the performance of the applications and provides an higher abstraction to write the image processing applications.State-of-the-art DSLs like Halide,Darkroom and Polymage are developed specifically for multi-cores and GPUs as the target platforms.In recent years, FPGAs have been employed in the clouds by amazon etc due to their acceleration capability and low power usage.So, there is an active research going on in developing the DSLs for various applications targeting FPGAs but most of them try to extend the DSL meant for CPU/GPU to FPGAs like Halide or focused on developing the tools to target the already present high level synthesis abstraction tools, it creates bottlenecks as they cannot exploit the available resources to maximum extent.Hence, the thesis discusses about a DSL which is focused only on FPGAs called FlowPix and gives a demonstration of FlowPix by explaining two applications Bilateral filter and Bilateral Grid.

Contents

List of Figures

List of Tables

Chapter 1

Background

1.1 Image Processing Filters

Filtering is a technique for modifying or enhancing an image.Spatial domain filtering is a neighbourhood operation, where the output of the processed pixel is determined by applying an algorithm to the value of pixels in the neighbour of the corresponding input pixel. Filters like Gaussian filter and Bilateral filter are spatial dependent filters.They are generally used to reduce noise.

1.1.1 Gaussian and Bilateral Filters

Gaussian filter in addition to decreasing the noise also smooths the image across edges, which makes edges less sharper or even disappear. This problem can be solved by using a advanced version of Gaussian filter called Bilateral filter . Unlike the Gaussian filter, where the weights of the neighbourhood pixels only depend on Euclidean distance ,weights in Bilateral filter also depends on the range difference like color intensity preserving edges.They are many applications and techniques which requires to perform an operation in a smooth surface but not across edges like image segmentation, HDR Tone Mapping etc.

add the threshold figures here and also an image if possible and also the equations

$$GF(x) = \frac{\sum_{y \in N(x)} G_s(x-y) * I(y)}{\sum_{y \in N(x)} G_s(x-y)} \tag{1.1}$$

$$BF(x) = \frac{\sum_{y \in N(x)} G_s(x-y) * G_r(I(x)-I(y)) * I(y)}{\sum_{y \in N(x)} G_s(x-y) * G_r(I(x)-I(y))} \tag{1.2}$$

The above equation (1.1) and (1.2) represents the output of Gaussian and Bilateral filter for an input pixel x respectively and y belongs to the neighbourhood of x.The only difference between them is the term $G_r(I(x) - I(y))$ which represents the Gaussian weights due to the Intensity difference. $G_s(x - y)$ represents the Gaussian weights due to the Euclidean distance between the pixels x and y.

1.2 Simplification and Acceleration Techniques of Bilateral Filter

The direct implementation standard bilateral filter expressed in equation (1.2) requires O(σ_s^2) operations per pixel , where σ_s is the radius of the spatial kernel.Consequently

a lot of study on simplifying and accelerating it can be found in the literature.

1.2.1 Durand bilateral filter and 3D bilateral Grid

Durand and Dorsey proposed a piecewise linear bilateral filter in [1]. They quantize the intensity into several segments and perform FFT-based linear filtering for each segment.The final results are pooled using a linear interpolation on these linearly filtered images.The complexity of Durand's piecewise linear bilateral filter is $O(\log \sigma_s)$.

Later Paris et al. [2] generalize the idea of piecewise linear bilateral filter by Durand to form a 3-D grid.Then the bilateral filter can be interpreted as a linear convolution of a vector-valued image.They made 3D grid manipulations to Durand bilateral filter so that they can take advantage of the parallelism available due to GPUs.

The research on accelerating bilateral filter was focused on CPUs and GPUs.FPGAs can provide a good amount of parallelism and the next chapters will be based on implementation of bilateral filter on FPGAs using Bluespec Systemverilog and also an IR to generate the bluespec code which acts as DSL.

1.3 Bilateral Grid

Bilateral Grid is presented as a new data structure that enables fast edge-aware image processing in the paper [3] . They claim that working in the bilateral grid,algorithms such as bilateral filtering,local histogram equalization become simple manipulations that are both local and independent.

They also state that fast numerical schemes for approximating the bilateral filter are able to process large images in less than a second, like the seperable approximation described by Pham et al. [4] which works well for small kernels but suffers from artifacts when larger kernels are used in other applications and Weiss uses local image histograms which limits the approach to box spatial kernels instead of smooth Guassian Kernel.

They extend the fast bilateral filter introduced by Durand described in previous section and recast the computation as a higher-dimensional linear convolution followed by trilinear interpolation and a division.

1.3.1 Construction

The first step is the constrcution of the grid from the image.It is simply a histogram construction, a mapping of x pixels into y values and each y value corresponds to a plane.The image is divided into nxn tiles and each tile is mapped to the y planes ,where each plane corresponds to a range of intensity(the intensity range of 0-255 is divided into y ranges).The number of pixels in each plane is also stored which is used for division in the final step during slicing to get the final output pixel value.This step completes the creation of a grid and storing the values in the grid.A 3-dimensional grid is created from 2-dimensional image.

1.3.2 Processing

The second step is the processing step where the grid is convolved with a 5x5 Gaussian kernel or a 3x3 Gaussian kernel,by elevating to 3-dimensional space,a complex non-linear filter is replaced by a linear gaussian filter.The output is also a 3-dimensional grid.

1.3.3 Slicing

The final step is to get back to 2-dimensional space from the grid. It is done by using trilinear interpolation which uses the input image and the output grid values to reconstruct the output image.

1.4 Introduction

We need some interface to communicate between our software code running on CPU and hardware design on FPGA, this is generally done via pcie and writing device drivers for your hardware design.It's a tedious work. So researchers from University of Cambridge have designed a framework called Connectal,which generates hardware/software interface implementations from abstract Interface Design Language(IDL) specifications.They chose Bluespec interfaces as the IDL for connectal's interface compiler.

1.5 Connectal

Bluespec System verilog was used for writing the hardware components of COnnectal libraries as it enables higher level of abstraction and supports parameterized types.The software components are written in c/c++.As our DSL for image processing is also written in Bluespec , it provides us an opportunity for smooth integration of the DSL with Connectal .It also offers portabilty across FPGAs and how they are connected to CPUs and the interfaces used whether they are PCIe or AXI as its hardware and software libraries are largely platform independent . More information on connectal can be found in the paper[5].

Chapter 2

FlowPix,Bilateral Filter,Bilateral Grid

2.1 Introduction

This Chapter is related to the **FlowPix** Domain Specific Language and how one can use it for image processing on FPGAs.An implementation of Bilateral Filter and Bilateral Grid is discussed in detail ..Section 2.2 is dedicated to the explanation of the **FlowPix** Domain Specific Language,Section 2.3 and Section 2.4 is about the implementation of Bilater Filter and Bilateral Grid using **FlowPix** and the results are compared and conclusions are drawn in Section 2.5.

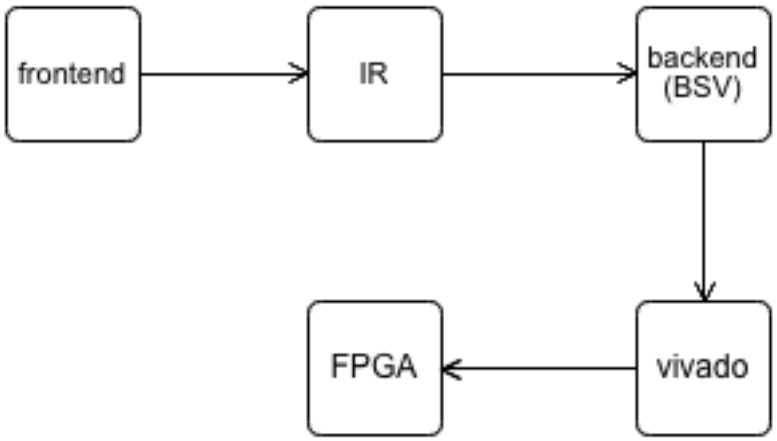

FIGURE 2.1: FlowPix flow

2.2 FlowPix

As of now an IR(intermediate representation) is used as an interface for writing the code which is embedded in Python.A frontend language can be added on top of the IR.The IR code generates the FlowPix backend code written in BSV which generates verilog files which are imported in vivado.Most of the tools and compilers for accelerating image processing on FPGAs target the vivado HLS which takes care of the

verilog generation and generated files are used in vivado.In FlowPix, we have more control over the verilog code generated.Frontend is scala language for FlowPix.

2.2.1 Backend

The backend for the compiler is written in bluespec systemverilog which generates verilog code when compiled .The above IR generates the bluespec systemverilog code and subsequently the verilog code when compiled.The verilog code is then exported to the vivado design flow ,where the synthesis,implementation,routing and timming analysis takes place.At the end a bitstream file is generated, which is required to program the FPGA by dumping the code via PCIE for FPGAs.

2.3 Bilateral Filter using FlowPix

Bilateral Filter has been explained in the previous chapter.This section is about it's implemented architecture in FlowPix.

2.3.1 Architecture

The figure 4.1 shows the architecture of bilateral filter .TILE in the figure represents a two dimensional buffer which stores the pixels of input image.It is an efficient way of storing as it increases the throughput and the processed pixel values are replaced by new required pixels which are needed for further processing.

A sliding window is passed through the TILE which takes nxn values(where n =3,5 etc) as required.This strategy gives us an output pixel for every cycle after the first n rows are filled.The obtained window is then passed to 3 parallel blocks.

The first block doesnt do any operation on the nxn pixels , it just passes the window on next cycle.The second block produces the gaussian kernel values for kernel size n.The third block generates the kernel due to range differences.The next two steps are one to one multiplication between the three windows and their summation respectively which gives us two values and then passing them through divide block gives us the output value of the pixel.I have not added them all at once to reduce the clock period, I added them in batches.

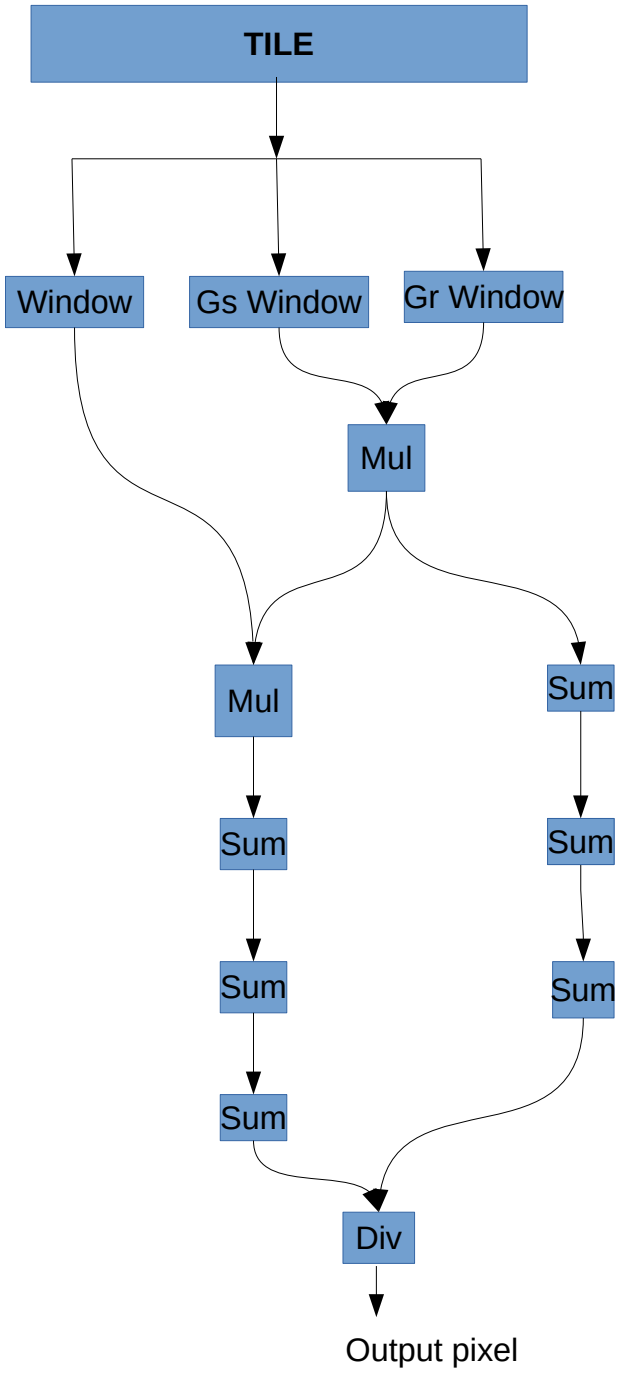

FIGURE 2.2: Bilateral Filter (Architecture).

2.3.2 IR code

The following is the IR code for bilateral filter

```
image  <= [256,256,1]<Image>;
```

```
source <= [image,1]<Source>;
tile   <= [source,5,5,1,1,1]<Tile>;

tileGauss <= {
              Vector#(VectorLength,DataType) resultOut = replicate(0);
              for(UInt#(10) r=0;r <25;r =r +1) begin
                resultOut[r] = x[r];
              end
              return resultOut;
             }[tile,25,25]<Permute>;



spatialGauss  <= {
    Vector#(VectorLength,DataType) resultOut = replicate(0);
            for(UInt#(10) r =0;r < 25;r=r+1) begin
               resultOut[r] = 0.04;
            end
           return resultOut;
          }[tile,25,25]<Permute>;


rangeGauss  <= {
     Vector#(VectorLength,DataType) resultOut = replicate(0);
DataType t1=1;
DataType t2=2;
                DataType range1 = expr(x[12] -t2);
                DataType range2 = expr(x[12] -t1);
                DataType range3 = x[12];
                DataType range4 = expr(x[12] + t1);
                DataType range5 = expr(x[12] + t2);
               for(UInt#(10) r=0;r <25;r =r +1) begin
                 if(x[r] == range1 || x[r] == range5)
                     resultOut[r] = 0.33;
                 if(x[r] == range2 || x[r] == range4)
                     resultOut[r] = 0.66;
                 if(x[r] == range3)
                     resultOut[r] = 1;
                end

                return resultOut;
          }[tile,25,25]<Permute>;


srmul <= [spatialGauss,rangeGauss,0,25]<Multiply>;

num_mul <= [srmul,tileGauss,0,25]<Multiply>;

den_perm1 <= {
                DataType a = expr(x[0] + x[1] + x[2]);
                DataType b = expr(x[3] + x[4] + x[5]);
```

```
                DataType c = expr(x[6] + x[7] + x[8]);
                DataType d = expr(x[9] + x[10] + x[11]);
                DataType e = expr(x[12] + x[13] + x[14]);
                DataType f = expr(x[15] + x[16] + x[17]);
                DataType g = expr(x[18] + x[19] + x[20]);
                DataType h = expr(x[21] + x[22]);
                DataType i = expr(x[23] + x[24]);
return Vector(a,b,c,d,e,f,g,h,i);
              }[srmul,25,9]<Permute>;

den_perm2 <= {
                DataType a = expr(x[0] + x[1] + x[2]);
                DataType b = expr(x[3] + x[4] + x[5]);
                DataType c = expr(x[6] + x[7] + x[8]);
                return Vector(a,b,c);
             }[den_perm1,9,3]<Permute>;

den_perm3 <= {
                DataType a = expr(x[0] + x[1] + x[2]);
                return Vector(a);
            }[den_perm2,3,1]<Permute>;

num_perm1 <= {
                DataType a = expr(x[0] + x[1] + x[2]);
                DataType b = expr(x[3] + x[4] + x[5]);
                DataType c = expr(x[6] + x[7] + x[8]);
                DataType d = expr(x[9] + x[10] + x[11]);
                DataType e = expr(x[12] + x[13] + x[14]);
                DataType f = expr(x[15] + x[16] + x[17]);
                DataType g = expr(x[18] + x[19] + x[20]);
                DataType h = expr(x[21] + x[22]);
                DataType i = expr(x[23] + x[24]);
return Vector(a,b,c,d,e,f,g,h,i);
              }[num_mul,25,9]<Permute>;

num_perm2 <= {
                DataType a = expr(x[0] + x[1] + x[2]);
                DataType b = expr(x[3] + x[4] + x[5]);
                DataType c = expr(x[6] + x[7] + x[8]);
                return Vector(a,b,c);
             }[num_perm1,9,3]<Permute>;

num_perm3 <= {
                DataType a = expr(x[0] + x[1] + x[2]);
                return Vector(a);
            }[num_perm2,3,1]<Permute>;
```

In the above code the the text in between "< >" are the names of the modules written in BSV and the text before <= is the instance of the module.The synthesis results will be discussed in section 4.5.

2.4 Bilateral Grid

As described in previous chapter Bilateral grid is divided into three stages,grid construction,processing,slicing respectively.It is easier to explain the technique using an example, so consider a image size of 256x256 and intensity levels are 4 and the dimensions of the kernel size are 4x4 .

2.4.1 Grid Construction

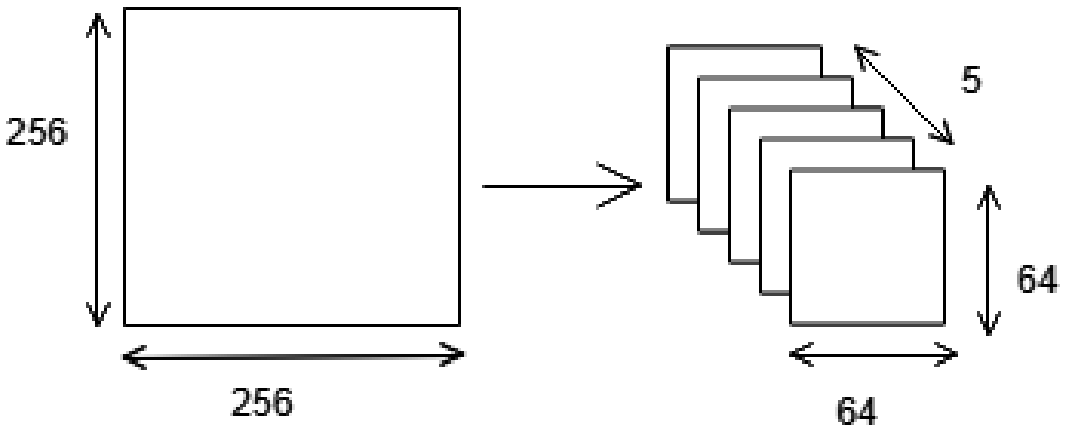

FIGURE 2.3: Grid Construction

The figure 4.3 shows the construction of the grid , in simpler terms it gets down sampled from 256x256 to 64x64x5.The mapping of the pixels to the plane follows the equation 4.1, where [] symbol indicates nearest integer.Each point in the grid stores the cummulative pixel values along with the number of pixels.

$$G([x/4],[y/4],[I/4])+=(I,1) \tag{2.1}$$

A pixel is stored only in one plane which is decided by the [I/4] .For this specific example plane 1 corresponds to intensity values between 0-32,plane 2 between 32-96 and 96-160,160-224,224-255 respectively for the remaining planes.

Next step is the convolution in each layer.

2.4.2 Processing

Each plane in the grid is convolved with gausian kernel and resultant is also of same size. In the next step all the planes are added point wise with the plane above and below it in the ratio 1:2:1 so we get a three dimensional convolution . The final result of the processing step is a 64x64x5 grid. Padding must be done wherever necessary. The final step is the conversion of the grid to output image, this is achieved by using

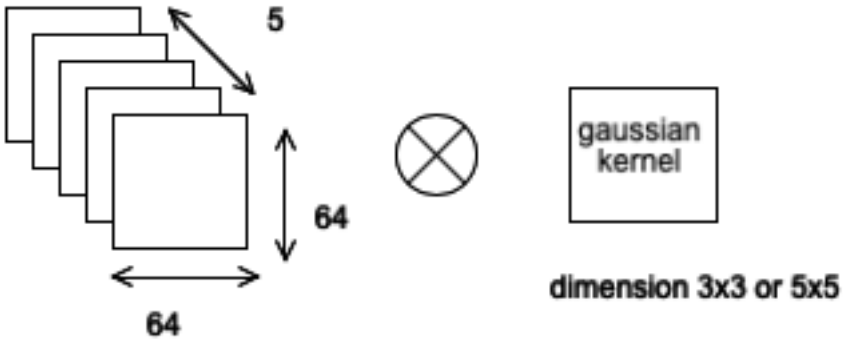

FIGURE 2.4: Grid Convolution layer by layer

a technique called trilinear interpolation.

2.4.3 Reconstruction

The input image is needed along with the grid , no parallelism can be exploited in this step so implementing it on cpu side is better than on FPGA.I have used the matlab code for this step.

2.4.4 Comparison with Matlab

The output from FlowPix is almost similar to the matlab output under same settings.THe matlab code uses floating point operation and Flowpix data type is set to 24 bit(16 bits for integer part and 8 bits for fractional part).

FIGURE 2.5: Input Image

FIGURE 2.6: Output Image(FlowPix)

FIGURE 2.7: Output Image(Matlab)

2.5 results and comparison

In this section the synthesis and performance results for bilateral filter and grid are presented.Table 2.1 shows the usage resources for vertex-vc709 board.

TABLE 2.1: Comparison of bilateral filter(5x5) on FlowPix and xfopencv

Resource	**FlowPix**	**xfopencv**
clock period	4.6ns	not available
clock cycles	65576	not available
cycles/pixel	1.0003	not available
LUT	26611	4943
LUTRAM	5500	not available
FF	24830	5481
BRAM	2.5	12
DSP	50	30
IO	36	not available
power	0.624w	not available

Xfopencv is a library in Xilinx SDSOC platform, it provides many ready to use image processing filters.

It can be seen from table 2.1 that there is an increase of flip flops and LUTs in flowpix but huge decrease in BRAM usage(1 BRAM size is 18k).It was mentioned in the xfopencv library manual that for image of sixe 1080x1920,latency is 7.20ms at 168MHz.Bilateral filter implemented using FlowPix has a latency of 9.5 ms for the same image.

TABLE 2.2: Comparison of bilateral filter(5x5) and bilateral grid using FlowPix

Resource	**Bilateral Filter**	**Bilateral Grid**
clock period	4.6ns	4.9ns
clock cycles	65576	77283
cycles/pixel	1.0003	1.17
LUT	26611	117169
LUTRAM	5500	26632
FF	24830	98313
BRAM	2.5	46
DSP	50	126
IO	36	176
power	0.624w	1.4w

The table 2.2 shows the comparison between the bilateral filter and bilateral grid in FlowPix framework.Bilateral filter having an advantage of having shared pixels between the tiles which is not present in bilateral grid.they were also implementation in which the input image is stored in on-chip memory and having a dedicated computational unit for each tile stored in memory, which gives us parallelization.This was the approach taken in the paper [6]. Bilateral filter performs well than bilateral grid.

There is a need to study further on the applications of bilateral filter , to develop bilateral filter strategies which can take advantange of resources available and can perform well.

There is abundance of literature on the bilateral filter applications and is also integrated in Convolution neural networks pipelines for certain applications.

Appendix A

IR code for bilateral grid

```
image  <= [268,268,1]<Image>;
vc     <= [1,2,1,2,4,2,1,2,1]<Constant,Vector>;
source <= [image,1]<Source>;
tile   <= [source,4,4,1,4,4]<Tile>;
v1   <= [tile,16]<Vectorize>;
v2   <= [tile,16]<Vectorize>;
v3   <= [tile,16]<Vectorize>;
v4   <= [tile,16]<Vectorize>;
v5   <= [tile,16]<Vectorize>;
v6   <= [tile,16]<Vectorize>;
v7   <= [tile,16]<Vectorize>;
v11   <= [tile,16]<Vectorize>;
v21  <= [tile,16]<Vectorize>;
v31   <= [tile,16]<Vectorize>;
v41   <= [tile,16]<Vectorize>;
v51   <= [tile,16]<Vectorize>;
v61   <= [tile,16]<Vectorize>;
v71   <= [tile,16]<Vectorize>;
//############ plane 1 ####################
simd1  <=  {
if(x[0] >= 0 && x[0] < 0.125)
return x[0];
else
return 0;
   }[v1,16]<SIMD>;
out1  <=   [simd1,16,1]<DeVectorize>;
acc1  <=   {
               return expr(in[1] + in[0]);
           }[out1,4,4,True,1]<Accumulate>;
tile1 <=   [acc1,3,3,1,1,1]<Tile>;
conv2 <=   [tile1,3,vec,1]<Convolution>;
//########################################
simd11  <=  {
               if(x[0] >= 0 && x[0] < 0.125)
                       return 1;
               else
                       return 0;
           }[v11,16]<SIMD>;
out11  <=   [simd11,16,1]<DeVectorize>;
acc11  <=   {
```

```
                return expr(in[1] + in[0]);
            }[out11,4,4,True,1]<Accumulate>;
tile11 <=   [acc11,3,3,1,1,1]<Tile>;
conv21 <=   [tile11,3,vec,1]<Convolution>;
//############## plane 2 ##################
simd2 <=   {
                if(x[0] >= 0.125 && x[0] < 0.375)
                        return x[0];
                else
                        return 0;
            }[v2,16]<SIMD>;
out2  <=   [simd2,16,1]<DeVectorize>;
acc2  <=   {
                return expr(in[1] + in[0]);
            }[out2,4,4,True,1]<Accumulate>;
tile2 <=   [acc2,3,3,1,1,1]<Tile>;
conv3 <=   [tile2,3,vec,1]<Convolution>;
//########################################
simd21 <=   {
                if(x[0] >= 0.125 && x[0] < 0.375)
                        return 1;
                else
                        return 0;
            }[v21,16]<SIMD>;
out21  <=   [simd21,16,1]<DeVectorize>;
acc21  <=   {
                return expr(in[1] + in[0]);
            }[out21,4,4,True,1]<Accumulate>;
tile21 <=   [acc21,3,3,1,1,1]<Tile>;
conv31 <=   [tile21,3,vec,1]<Convolution>;

//############### plane 3 ################
simd3 <=   {
                if(x[0] >= 0.375 && x[0] < 0.625)
                        return x[0];
                else
                        return 0;
            }[v3,16]<SIMD>;
out3  <=   [simd3,16,1]<DeVectorize>;
acc3  <=   {
                return expr(in[1] + in[0]);
            }[out3,4,4,True,1]<Accumulate>;
tile3 <=   [acc3,3,3,1,1,1]<Tile>;
conv4 <=   [tile3,3,vec,1]<Convolution>;
//########################################
simd31 <=   {
                if(x[0] >= 0.375 && x[0] < 0.625)
                        return 1;
                else
                        return 0;
            }[v31,16]<SIMD>;
```

```
out31  <=   [simd31,16,1]<DeVectorize>;
acc31  <=   {
                return expr(in[1] + in[0]);
            }[out31,4,4,True,1]<Accumulate>;
tile31 <=   [acc31,3,3,1,1,1]<Tile>;
conv41 <=   [tile31,3,vec,1]<Convolution>;
//################ plane 4 ##################
simd4 <=   {
                if(x[0] >= 0.625 && x[0] < 0.875)
                        return x[0];
                else
                        return 0;
           }[v4,16]<SIMD>;
out4  <=   [simd4,16,1]<DeVectorize>;
acc4  <=   {
                return expr(in[1] + in[0]);
           }[out4,4,4,True,1]<Accumulate>;

tile4 <=   [acc4,3,3,1,1,1]<Tile>;
conv5 <=   [tile4,3,vec,1]<Convolution>;
//#########################################
simd41 <=   {
                if(x[0] >= 0.625 && x[0] < 0.875)
                        return 1;
                else
                        return 0;
           }[v41,16]<SIMD>;
out41  <=   [simd41,16,1]<DeVectorize>;
acc41  <=   {
                return expr(in[1] + in[0]);
           }[out41,4,4,True,1]<Accumulate>;

tile41 <=   [acc41,3,3,1,1,1]<Tile>;
conv51 <=   [tile41,3,vec,1]<Convolution>;
//################## plane 5 ################
simd5 <=   {
                        return 0;
           }[v5,16]<SIMD>;
out5  <=   [simd5,16,1]<DeVectorize>;
acc5  <=   {
                return expr(in[1] + in[0]);
           }[out5,4,4,True,1]<Accumulate>;
tile5 <=   [acc5,3,3,1,1,1]<Tile>;
conv1 <=   [tile5,3,vec,1]<Convolution>;
//#########################################
simd51 <=   {
                        return 0;
           }[v51,16]<SIMD>;
out51  <=   [simd51,16,1]<DeVectorize>;
acc51  <=   {
                return expr(in[1] + in[0]);
```

```
          }[out51,4,4,True,1]<Accumulate>;
tile51 <=   [acc51,3,3,1,1,1]<Tile>;
conv11 <=   [tile51,3,vec,1]<Convolution>;
//################## plane 6 ################
simd6 <=   {
                        return 0;
          }[v6,16]<SIMD>;
out6  <=   [simd6,16,1]<DeVectorize>;
acc6  <=   {
               return expr(in[1] + in[0]);
          }[out6,4,4,True,1]<Accumulate>;
tile6 <=   [acc6,3,3,1,1,1]<Tile>;
conv6 <=   [tile6,3,vec,1]<Convolution>;
//########################################
simd61 <=   {
                        return 0;
          }[v61,16]<SIMD>;
out61  <=   [simd61,16,1]<DeVectorize>;
acc61  <=   {
               return expr(in[1] + in[0]);
          }[out61,4,4,True,1]<Accumulate>;
tile61 <=   [acc61,3,3,1,1,1]<Tile>;
conv61 <=   [tile61,3,vec,1]<Convolution>;
//################## plane 7 ################
simd7 <=   {
               if(x[0] >= 0.875 && x[0] < 1)
                        return x[0];
               else
                        return 0;
          }[v7,16]<SIMD>;
out7  <=   [simd7,16,1]<DeVectorize>;
acc7  <=   {
               return expr(in[1] + in[0]);
          }[out7,4,4,True,1]<Accumulate>;
tile7 <=   [acc7,3,3,1,1,1]<Tile>;
conv7 <=   [tile7,3,vec,1]<Convolution>;
//########################################
simd71 <=   {
               if(x[0] >= 0.875 && x[0] < 1)
                        return 1;
               else
                        return 0;
          }[v71,16]<SIMD>;
out71  <=   [simd71,16,1]<DeVectorize>;
acc71  <=   {
               return expr(in[1] + in[0]);
          }[out71,4,4,True,1]<Accumulate>;
tile71 <=   [acc71,3,3,1,1,1]<Tile>;
conv71 <=   [tile71,3,vec,1]<Convolution>;
```

```
3dConv1 <= {
DataType t = 2;
return expr(x[0] + t* x[1] + x[2]);
   }[conv1,conv2,conv3,1]<Point>;

3dConv11 <= {
DataType t = 2;
return expr(x[0] + t* x[1] + x[2]);
   }[conv11,conv21,conv31,1]<Point>;
3dConv2 <= {
DataType t = 2;
                return expr(x[0] + t*x[1] + x[2]);
           }[conv2,conv3,conv4,1]<Point>;
3dConv21 <= {
DataType t = 2;
                return expr(x[0] + t*x[1] + x[2]);
           }[conv21,conv31,conv41,1]<Point>;
3dConv3 <= {
DataType t = 2;
                return expr(x[0] + t*x[1] + x[2]);
           }[conv3,conv4,conv5,1]<Point>;
3dConv31 <= {
DataType t = 2;
                return expr(x[0] + t*x[1] + x[2]);
           }[conv31,conv41,conv51,1]<Point>;

3dConv4 <= {
DataType t = 2;
                return expr(x[0] + t*x[1] + x[2]);
           }[conv4,conv5,conv7,1]<Point>;
3dConv41 <= {
DataType t = 2;
                return expr(x[0] + t*x[1] + x[2]);
           }[conv41,conv51,conv71,1]<Point>;
3dConv5 <= {
DataType t = 2;
                return expr(x[0] + t*x[1] + x[2]);
           }[conv5,conv7,conv6,1]<Point>;
3dConv51 <= {
DataType t = 2;
                return expr(x[0] + t*x[1] + x[2]);
           }[conv51,conv71,conv61,1]<Point>;
```

Bibliography:

1) Durand, F., Dorsey, J.: Fast bilateral filtering for the display of high-dynamic-range images. In: SIGGRAPH (2002)

2) Paris, S., Durand, F.: A fast approximation of the bilateral filter using a signal processing approach. In: Leonardis, A., Bischof, H., Pinz, A. (eds.) ECCV 2006. LNCS, vol. 3954, pp. 568–580. Springer, Heidelberg (2006)

3) Jiawen Chen , Sylvain Paris , Frédo Durand, Real-time edge-aware image processing with the bilateral grid, ACM Transactions on Graphics (TOG), v.26 n.3, July 2007

4) Pham, T., and van Vliet, L. 2005. Separable bilateral filtering for fast video preprocessing. IEEE Intl. Conf. on Multimedia and Expo

5) M. King, I. Hicks, I. Ankcorn, "Software-driven hardware development", Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 13-22, 2015.

6) A. Mazumdar, A. Alaghi, J. T. Barron, D. Gallup, L. Ceze, M. Oskin, and S. M. Seitz. A hardware-friendly bilateral solver for real-time virtual reality video. In HPG, pages 13:1--13:10, 2017