4-2015

High Level Synthesis, a Use Case Comparison with Hardware Description Language

Michael D. Zwagerman
Grand Valley State University, zwagermm@mail.gvsu.edu

High Level Synthesis, a Use Case Comparison with Hardware Description Language

Michael Zwagerman

A Thesis Submitted to the Graduate Faculty of

GRAND VALLEY STATE UNIVERSITY

In

Partial Fulfillment of the Requirements

For the Degree of

Master of Science in Engineering

School of Engineering

April 2015

Abstract

This paper compares Vivado High-Level Synthesis (HLS), a new mainstream technology offered by Xilinx Inc., against the typical Hardware Description Language (HDL) design approach. An example video filter application was implemented via both methods and compared for differences in performance and Non-Reoccurring Engineering (NRE). Lessons learned using HLS are also provided. The objective of this paper is to provide actual comparison data on the current state of mainstream HLS to enable informed decision making for designs considering HLS.

The Xilinx Zync System on a Chip (SoC) offering is used as a platform for both the traditional HDL methods and HLS. This platform includes Field Programmable Gate Array (FPGA) fabric combined with a high speed application microprocessor. These single silicon SoC solutions appear to be a platform capable of effectively utilizing HLS. The example video application selected for implementation is a 9 by 9 kernel convolution filter performed on 24 bit 1080p video at 60 frames per second. The 2013 Xilinx Vivado tool suite was used for both HLS and HDL methods.

HLS proved to be very easy to use to create a functional RTL design. With naïve implementations in both, HLS did not perform well in resource utilization. HLS also provided a design with a slower maximum clock frequency.

Contents

Figures

Tables

Acronyms and Abbreviations

ARM	ADVANCED RISC MACHINE
AXI	ADVANCED EXTENSIBLE INTERFACE
BRAM	BLOCK RAM
CDFG	CONTROL AND DATA FLOW GRAPH
DSP	DIGITAL SIGNAL PROCESSING
DSP48	DSP RESOURCE WITHIN SEVERAL FAMILYS OF XILINX FPGAS
EDA	ELECTRONIC DESIGN AUTOMATION
FF	FLIP FLOP
FPGA	FIELD PROGRAMMABLE GATE ARRAY
HDL	HARDWARE DESCRIPTION LANGUAGE
HDMI	HIGH-DEFINITION MULTIMEDIA INTERFACE
HLS	HIGH LEVEL SYNTHESIS
IP	INTELLECTUAL PROPERTY
LUT	LOOK UP TABLE
NRE	NON-RECURRING ENGINEERING
OS	OPERATING SYSTEM
RAM	RANDOM ACCESS MEMORY
RISC	REDUCED INSTRUCTION SET COMPUTING
SD	SECURE DIGITAL
SOC	SYSTEM ON A CHIP
VHDL	VHSIC HARDWARE DESCRIPTION LANGUAGE
VHSIC	VERY HIGH SPEED INTEGRATED CIRCUIT

Introduction

Hardware description languages (HDLs) are a category of programming languages used to design for Field Programmable Gate Arrays (FPGAs). These languages provide special features to design sequential logic (time varying output via a clock) or combinational logic (the system output is a direct function of its input). While these languages have been used over the past decades to design hardware, they have many drawbacks. The languages are very verbose, the syntax is ridged and error prone, and they often lack advanced editor support (found for high-level programming languages). More-over, these languages can generate sub-optimal, faulty hardware which can be very difficult to debug.

A new technique has been under research to replace hand coding of HDLs (VHDL/Verilog). This new technique is called HLS (High Level Synthesis). HLS tools parse an existing high level programming language as an input and generate the corresponding HDL. Some of these tools are quite popular in the Electronic Design Automation (EDA) industry such as CatapultC from Mentor Graphics and Mathworks Matlab HDL coder, but are very expensive. Xilinx recently integrated the support of HLS into their Vivado tool chain at a more mainstream price point.

HLS has been a hot topic for custom logic engineers and vendors. It has the potential to be the next technological advancement for custom logic designers. HLS raises the abstraction layer from RTL to the algorithmic/behavioral level, allowing designers to focus on what needs to be done instead of the specifics of how to implement on a particular target. The stepping to HLS appears to be analogous to software designers moving to use high-level programming languages (C/C++) instead of assembly language. Today writing in assembly is niche and only done where absolutely necessary for performance or compactness[1]. If the analogy holds and the current tools perform as well as hand modified HDL, HLS could provide a significant increase in productivity. The purpose of this paper is to determine if the potential is attainable. This paper provides a use case example with lessons learned from implementing a typical image processing algorithm in both traditional methods (Xilinx Vivado VHDL) and with HLS (Xilinx Vivado HLS). The implementations in this paper used Xilinx's SoC Zynq as the target device. The paper includes a comparison of the results of this effort in terms of device resource utilization, design performance, and incurred non-reoccurring engineering effort.

Research Background

Reactively compensating for growth in device capabilities and complexity, HLS offers a design paradigm shift to a higher abstraction level which could become a more productive solution for implementing algorithms in an FPGA[1]. HLS tools have been around for over 30 years, but have not yet been adopted widely in industry[6]. In early 2011 Xilinx purchased AutoESL Design Technologies which produced the AutoPilot High-Level Synthesis Tool[7] and later rebranded the tool as Vivado HLS[8]. Xilinx Vivado HLS is a mainstream offering that may finally subvert traditional HDL methods. It is worth noting that Vivado HLS is not free, (nor inexpensive), a node locked license of Vivado HLS from Xilinx Inc is sold for $1995 (node-locked) or $2395 (Floating)[2].

Xilinx suggests that the design time it takes to implement a software application in HLS is much less than implementing in HDL (RTL). They also indicate that the performance of the HLS modules is worse, but close to their HDL counterparts. They provide graphs in UG998[9] to visualize this. These graphs are included in Figure 1 and Figure 2.

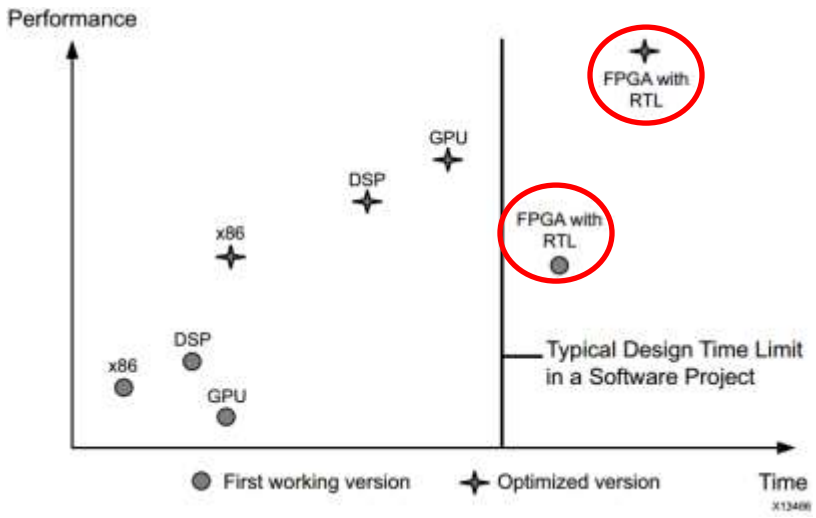

Figure 1: Design Time vs Application Performance with RTL Design Entry

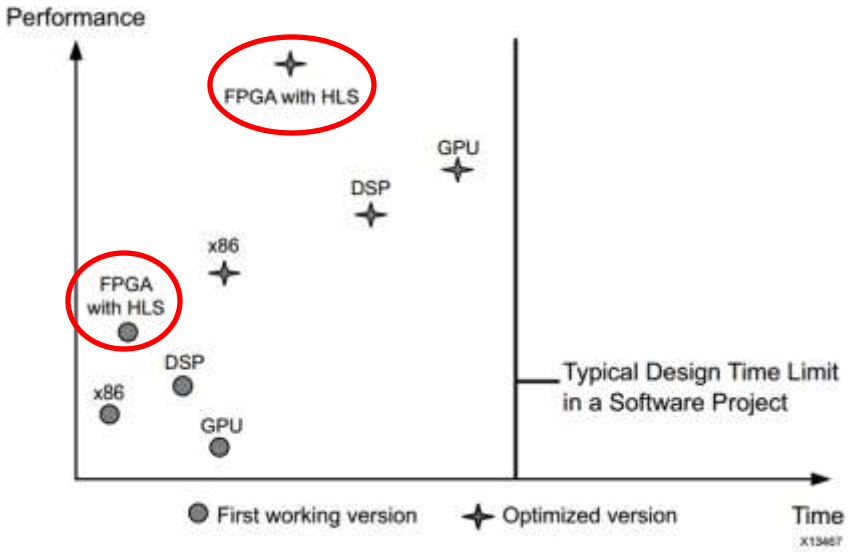

Figure 2: Design Time vs. Application Performance with Vivado HLS Compiler

HLS tools parse the high level language source code and compile it to an internal representation called a Control and Data Flow Graph (CDFG). CDFG is optimized based on automatic or manual algorithms for allocation (allocation of computing and storage resources), scheduling (clocking and timing), and binding (mapping operations to allocated computational or storage resources). After optimization is complete, then RTL in the form of HDL is generated [6]. A basic HLS tool flow is shown in Figure 3.

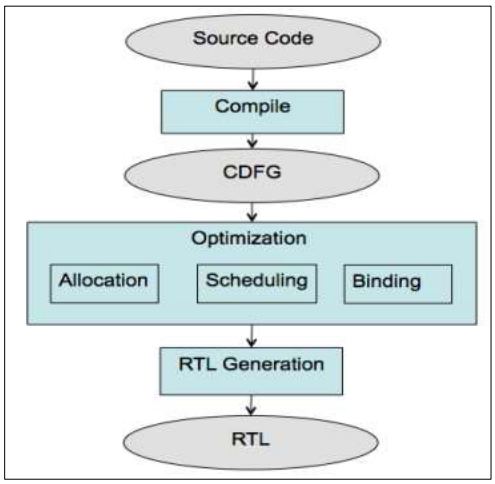

Figure 3: HLS Basic Flow [6]

Problem Research

An example, or typical, FPGA solution was needed for this comparative research. A video application was selected because video applications commonly utilize an FPGA to assist in image processing due to their computationally intensive pixel based operations which can bog down an embedded microprocessor. FPGAs are well suited for pixel operations that can be accomplished in a streaming (or pixel pipelined) manner.

Developing algorithms for an embedded platform commonly go through an embedded platform algorithmic life cycle. Figure 4 depicts the embedded platform algorithmic life cycle.

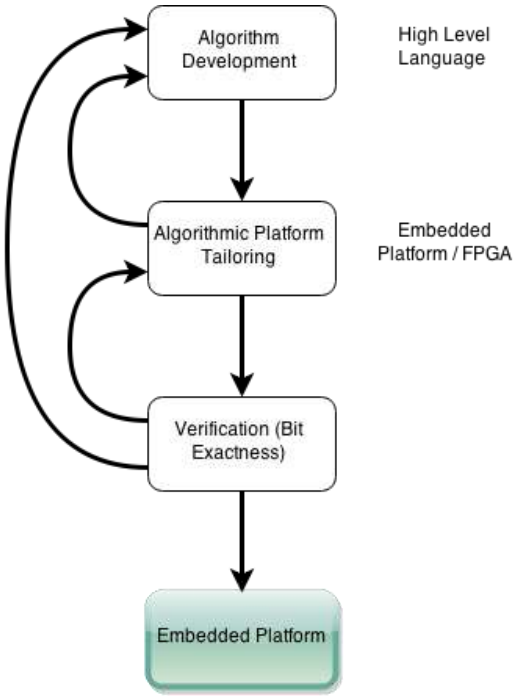

Figure 4: Embedded Platform Algorithmic Life Cycle

Video applications are typically designed on software computation platforms utilizing a high level language such as Matlab or C++. These computation platforms are typically higher end computers that provide near real time visual feedback. After an application algorithm has been designed, it often must be modified or tailored for the FPGA embedded production platform on which it must be executed in the field. Porting software algorithms to an FPGA is not always straight forward. Software algorithms are ultimately sequential instructions and are typically single threaded operations. FPGAs do not efficiently lend themselves to sequential operation.

FPGAs consist of hardware resources which are ‘connected’ via a configuration. Utilizing an FPGA for software like sequential operation can require a significant quantity of resources to emulate the sequential ordering. To minimize the required quantity of FPGA resources the originally specified algorithm can be refactored with parallelization or hardware design in mind. (Note: Typically an FPGA consisting of fewer resources is less expensive.) This type of refactoring requires skilled FPGA designers and additional effort. Due to the ‘human factor’ in the refactoring process, it is also possible that the results from the FPGA implementation diverge from the original design intent. Additional testing and simulation must be performed to provide confidence in bit-exactness between the implementation and algorithm.

Hardware Tool Kit

The target hardware platform for the FPGA implementations is the Xilinx 7c702 evaluation board[4] as pictured in Figure 5.

Figure 5: ZYNQ-7000 EPP ZC702 Evaluation Kit

The ZC702 development board is designed to evaluate the Zynq XC7Z020 SoC component. This platform was selected because Xilinx provides a base Targeted Reference Design (TRD)[5] to help get started quickly evaluating the Zynq XC7Z020 SoC on this board. This TRD includes example source and binaries for running Xilinx Petalinux with an example 1080p video demo application. This video demo included an example filter project for generating a video processing IP block.

Implementation

An image filter technique of convolving each input image pixel with a kernel was selected as the example algorithm for implementation. This technique is used for causing a range of image effects [3]. Kernel convolution creates a new ‘output pixel’ using coefficient weights applied to an ‘input pixel’ and its neighbor pixels. An output image is created by performing kernel convolution on every pixel in the input image. The 9x9 kernel used in this paper performs a blurring filter. The 9x9 kernel size is larger than a more typical 3x3 kernel size used in other video filters, but most video application utilize multiple kernel operations, and a larger kernel should amplify the implementation efficiency. With 1080p video, a 9x9 kernel performs 167,961,600 immediate multiplies per color channel per frame. The coefficients used to form the 9x9 kernel used are listed in Figure 6.

$$\begin{bmatrix} 2 & 3 & 4 & 4 & 5 & 4 & 4 & 3 & 2 \\ 3 & 4 & 6 & 7 & 7 & 7 & 6 & 4 & 3 \\ 4 & 6 & 8 & 9 & 10 & 9 & 8 & 6 & 4 \\ 4 & 7 & 9 & 11 & 12 & 11 & 9 & 7 & 4 \\ 5 & 7 & 10 & 12 & 12 & 12 & 10 & 7 & 5 \\ 4 & 7 & 9 & 11 & 12 & 11 & 9 & 7 & 4 \\ 4 & 6 & 8 & 9 & 10 & 9 & 8 & 6 & 4 \\ 3 & 4 & 6 & 7 & 7 & 7 & 6 & 4 & 3 \\ 2 & 3 & 4 & 4 & 5 & 4 & 4 & 3 & 2 \end{bmatrix}$$

Figure 6: 9x9 Kernel Coefficients

The coefficients of the kernel are approximately Gaussian, but were increased to have more averaging effect. Some pixels were also increased slightly to achieve a matrix coefficient sum of 512. A power of 2 sum is desired because the final intensity normalization is then simply a shift. The 9x9 kernel convolution process is outlined in Figure 7. The first step is a Hardamard product (An entry wise product). The values in the resultant matrix are summed and then normalized to produce a single output pixel.

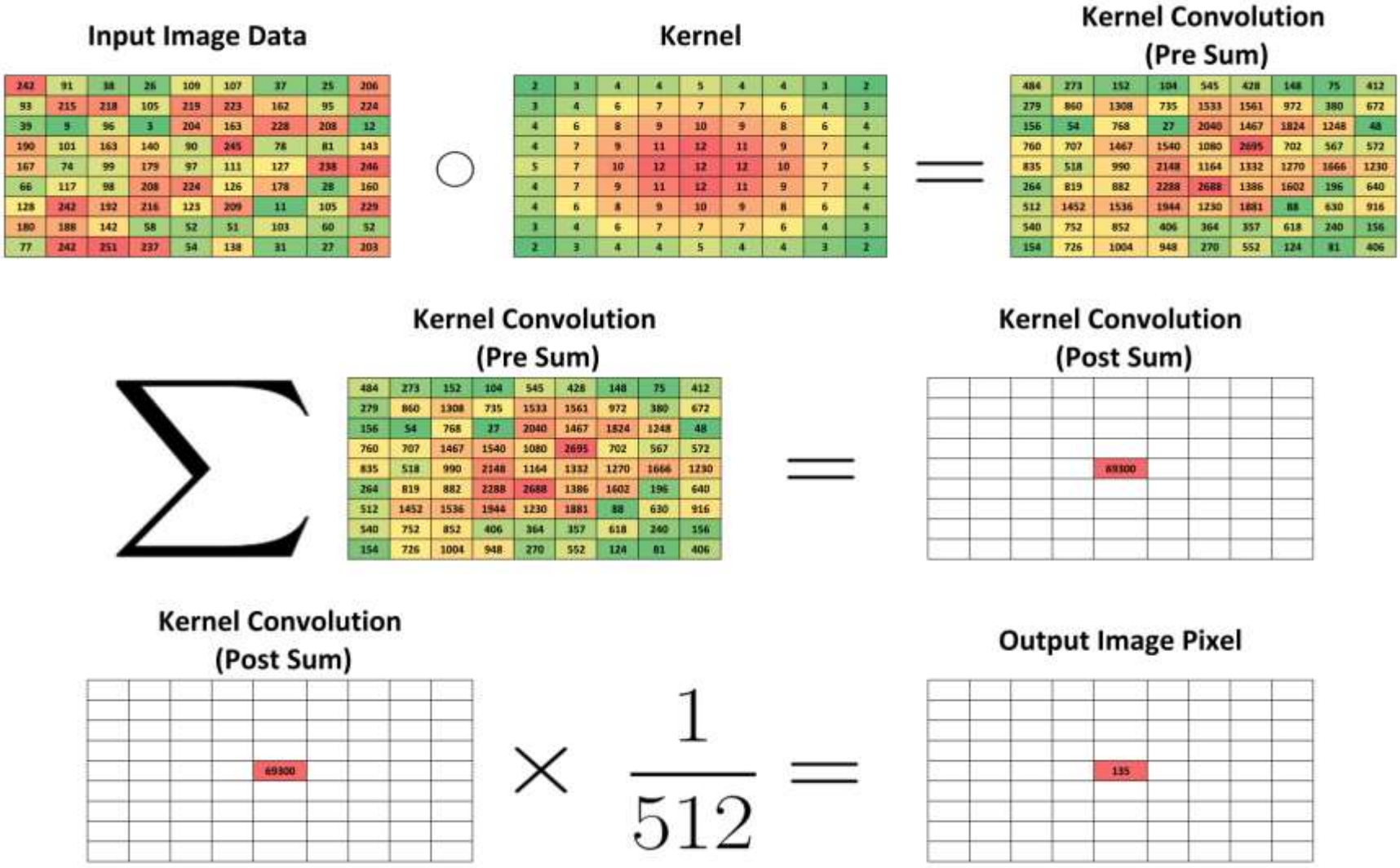

Figure 7: Kernel Convolution

Visual Filter Example

Our selected filter implements a blurring effect that is visually evident when comparing the source and resultant images. Figure 8 through Figure 11 display the effect of the filter on an image with high detail. Figure 8 and Figure 9 are scaled down 1080p images. The change from Figure 8 to Figure 9 is subtle because of the relative size of the 9x9 kernel pixels to a 1080p image Figure 10 and Figure 11 are 500 x 300 pixel cropped subsections. When comparing Figure 10 to Figure 11 the blurring is pronounced.

Figure 8: Example Raw (Pre Filter) Full Image

Figure 9: Example Image Post Kernel Filter Full Image

Figure 10: Example Raw (Pre Filter) Image Cropped to Show Detail

Figure 11: Example Image Post Kernel Filter Cropped to Show Detail

Note: the original full version of Figure 8 was included in the Xilinx Zynq TRD compiled for the ZC702 evaluation platform.

Xilinx TRD

The Xilinx TRD was used as the infrastructure backbone for efficiently utilizing the custom kernel implementations. According to Xilinx the "TRD is an embedded video processing application designed to showcase various features and capabilities of the Zynq Z-7020 AP SoC device for the embedded domain"[5]. As a demo application, the TRD includes multiplicity of interworking components, most notably: a running OS, a complete ARM configuration, and a custom logic video subsystem that utilizes a test pattern generator and a 1080p HDMI output.
The TRD provided a functioning system into which the custom logic evaluated in this paper was inserted.

For Xilinx's demo purposes the TRD included a custom logic Sobel filter that performed two 3x3 kernel convolutions. This Sobel filter was replaced with the custom 9x9 blurring filter.

The TRD video subsystem is not rudimentary. The video filter is only a single component of the video subsystem. Figure 12 is the block diagram of the Vivado TRD Processing System (PS) subsystem with the video processing block highlighted. This block diagram is only an abstracted view of the internal modules. Figure 13 is the expanded block diagram of the video processing block highlighted in Figure 12. The highlighted Sobel filter block is the component that was replaced.

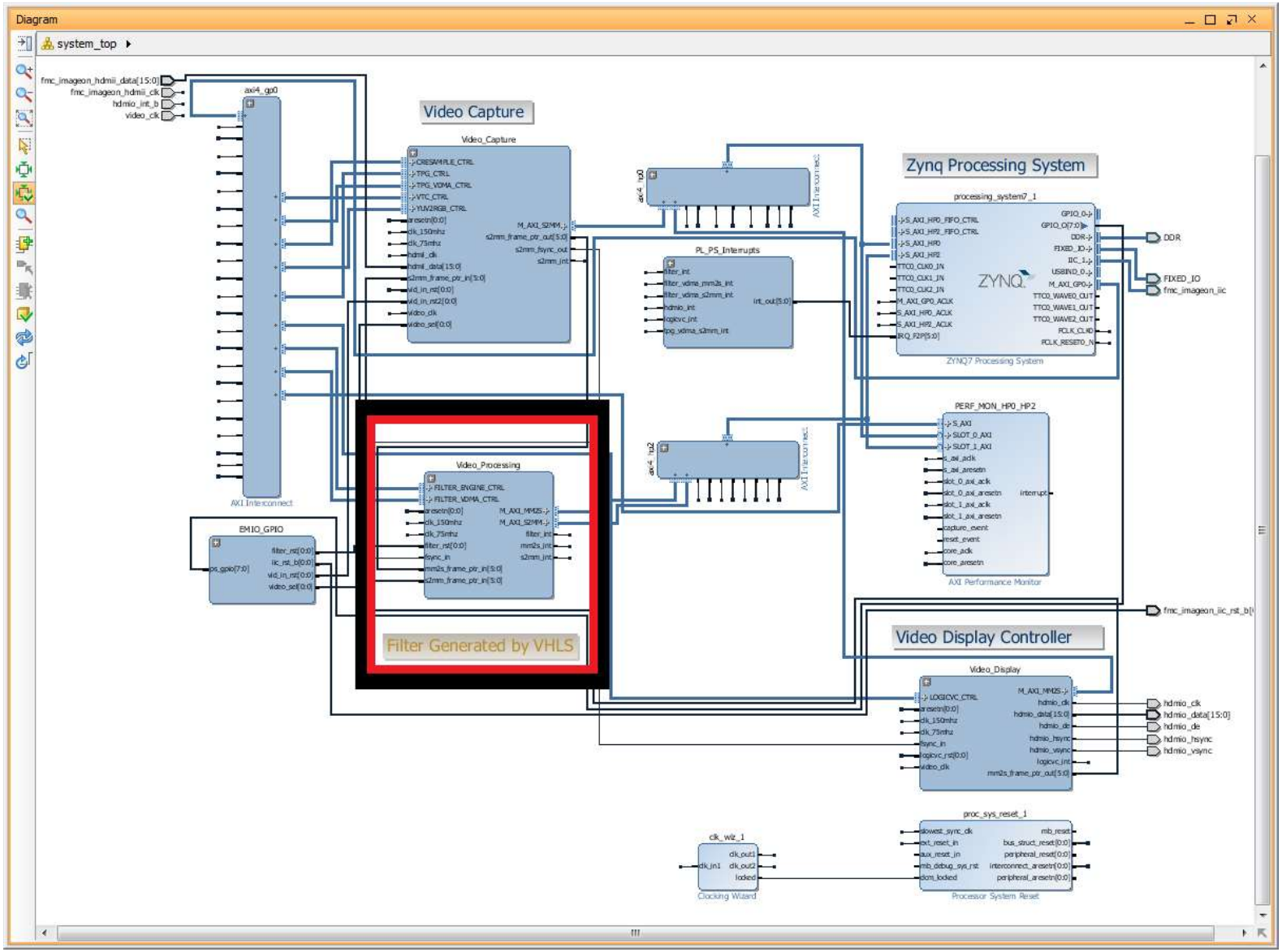

Figure 12: Vivado TRD PS Subsystem with Video Processing Highlighted

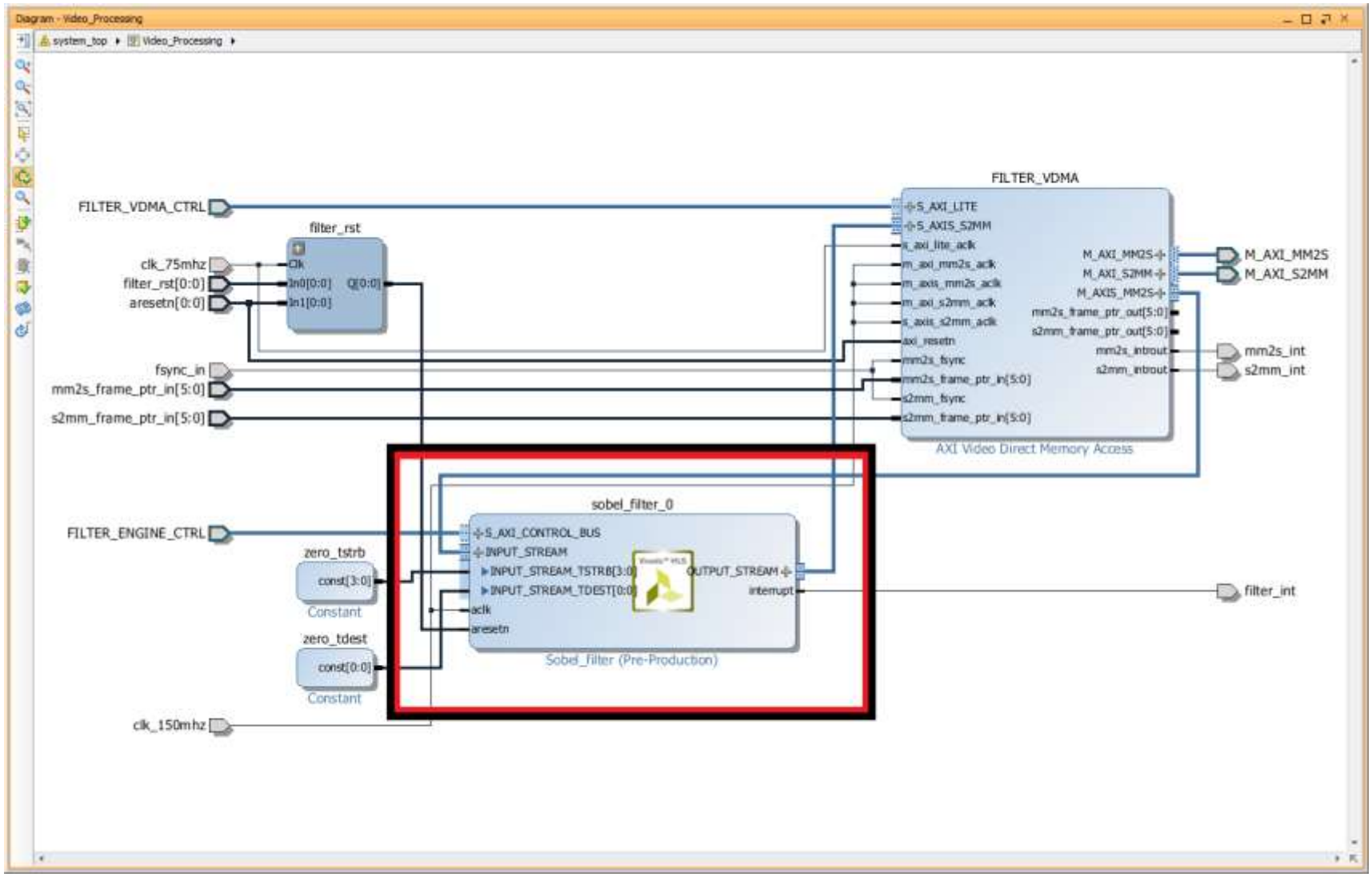

Figure 13: Vivado TRD Video Processing with Sobel Filter Highlighted

Using the TRD did not come without its share of compromises which are listed here:

- Because the TRD is a full-fledged demo, synthesizing and implementing the TRD project took approximately 1 hour on a 3.7Ghz hexa-core system.
- Due to a Microsoft Windows 7 path limitation, building is only possible if directories are kept very short—shorter than approximately 7 characters from drive root.
- Building the boot.bin file uses Petalinux which must be installed on a computer running Linux.
- The TRD Video filter assumes to have an Advanced eXtensible Interface (AXI) stream interface.
- Only some SD cards were found to work (Experimentally).
- The 2013.4 TRD was found unable to boot. (The 2013.3 TRD works fine.)
- Modifying the filter requires repackaging the HLS output into an Intellectual Property (IP) block (increasing the revision number), copying the IP block over to the Vivado project, upgrading the IP in the Vivado project. (Additional note: The project needs to be closed and reopened, otherwise Vivado IP status report won't detect the new IP).
- If the design doesn't meet timing requirements, the reported Vivado error confusingly indicates that you don't have a license for a free piece of IP.

Implementing in HLS

The HLS solution was generated with the Xilinx HLS tool suite version documented in Figure 14.

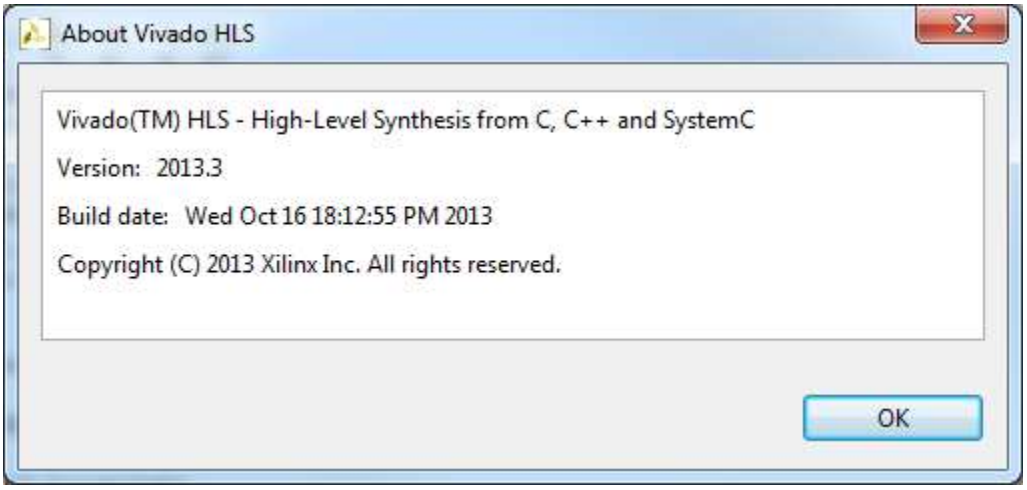

Figure 14: HLS Version Info

Implementing the kernel in HLS involved modifying the existing HLS Sobel project bundled with the Xilinx TRD. This example project provided the AXI streaming interface needed to integrate into the TRD. A code snippet containing the kernel convolution is included in Figure 15.

```
RGB operation(WINDOW *window)
{

    ap_uint<17> k_sum_R = 0;
    ap_uint<17> k_sum_B = 0;
    ap_uint<17> k_sum_G = 0;

    RGB pixel = {0,0,0};

    char i;
    char j;

    // Blurring Filter Kernel (Coefficients)
    const ap_uint<4> kernel[9][9] =  //< Note size limited to 4 bits!
    {
        // NOTE UNSIGNED!
        {2, 3, 4,  4,  5,  4,  4,  3, 2},
        {3, 4, 6,  7,  7,  7,  6,  4, 3},
        {4, 6, 8,  9,  10, 9,  8,  6, 4},
        {4, 7, 9,  11, 12, 11, 9,  7, 4},
        {5, 7, 10, 12, 12, 12, 10, 7, 5},
        {4, 7, 9,  11, 12, 11, 9,  7, 4},
        {4, 6, 8,  9,  10, 9,  8,  6, 4},
        {3, 4, 6,  7,  7,  7,  6,  4, 3},
        {2, 3, 4,  4,  5,  4,  4,  3, 2},
    };

    // Compute Hadamard product of pixel window and our kernel.
    // Sum all of the resultant product matrix elements
    for( i=0; i < 9; i++ ) //< Row iterator
    {
        for( j = 0; j < 9; j++ ) //< Column iterator
        {
            // Multiply each input window element with the corresponding kernel matrix element
            // Accumulate each each element wise product for each color channel
            k_sum_R = k_sum_R + (window->getval(i,j).R * kernel[i][j]); //< Red Channel
            k_sum_G = k_sum_G + (window->getval(i,j).G * kernel[i][j]); //< Green Channel
            k_sum_B = k_sum_B + (window->getval(i,j).B * kernel[i][j]); //< Blue Channel
        }
    }

    // Normalize the pixel sum via a shift
    pixel.R = k_sum_R.range(16,9); //< Red Channel
    pixel.G = k_sum_G.range(16,9); //< Green Channel
    pixel.B = k_sum_B.range(16,9); //< Blue Channel

    return pixel;
}
```

Figure 15: HLS Kernel Convolution Code Snippet

The operation code is straight-forward, given a window of 9x9 pixels:

1. Multiply each pixel in the window by the kernel coefficient.
2. Sum all the pixel and coefficient products.
3. Shift the final sum to reduce the change in intensity.

Special size limited types (ap_unit) were used to reduce the quantity of bits required in FPGA fabric. The ‘ap_’ data types were provided by Xilinx. Testing for correct operation was simplified by the TRD included test bench which executed the C++ and generated an image with the filter applied. Xilinx HLS can easily export the HLS solution to an RTL IP core with the click of a button. This IP Core can be utilized by a Vivado project.

A couple of lessons gleaned while implementing this core in HLS.

1. The HLS tool uses gigabytes of RAM memory: Figure 16 includes a sample usage while running C synthesis on the kernel implementation. Note: the java.exe executable is also called from Vivado HLS
2. Chasing timing closure failures in Vivado requires:
    a. Examining an obfuscated failure net
    b. Modifying C++ in an area that seems related
    c. Exporting to an RTL Core (Synthesizing C++ to HDL)
    d. Upgrading the Vivado project with the new core
    e. Re-implementing
    f. Re-running timing analysis
3. Certain C++ code used by HLS is designed specifically for C synthesis and not C simulation. This code needs to be programmatically removed for the type of action taken. (Commented out, Preprocessor define)
4. Creating AXI interfaces is very easy and requires very little work.
5. Selecting different types (unsigned int, unsigned char, etc) will impact the required FPGA resources; the C synthesis will use the bit length for types specified even if most significant bits remain unused.

6. The HLS generated HDL code is very obfuscated and difficult to understand and modify. (See Figure 17.)
7. The HLS generated HDL code commonly uses inverted logic. (See Figure 17.)
8. While HLS will export VHDL source, the IP Core generated by HLS is Verilog only.

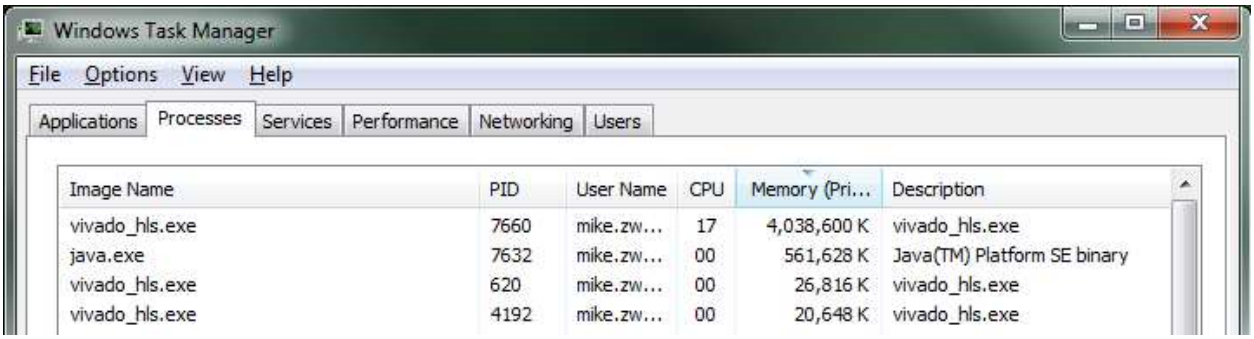

Figure 16: HLS C Synthesis Memory Usage

```
/// grp_operation_fu_1821_ap_ce assign process. ///
always @ (ap_CS_fsm or ap_sig_bdd_147 or ap_reg_ppiten_pp0_it1 or ap_reg_ppstg_tmp_7_reg_6253_pp0_it10
 or ap_sig_bdd_191 or ap_reg_ppiten_pp0_it11 or ap_reg_ppstg_tmp_7_reg_6253_pp0_it1 or
ap_reg_ppstg_tmp_7_reg_6253_pp0_it2 or ap_reg_ppstg_tmp_7_reg_6253_pp0_it3 or
ap_reg_ppstg_tmp_7_reg_6253_pp0_it4 or ap_reg_ppstg_tmp_7_reg_6253_pp0_it5 or
ap_reg_ppstg_tmp_7_reg_6253_pp0_it6 or ap_reg_ppstg_tmp_7_reg_6253_pp0_it7 or
ap_reg_ppstg_tmp_7_reg_6253_pp0_it8 or ap_reg_ppstg_tmp_7_reg_6253_pp0_it9 or or_cond2_reg_6569 or
ap_reg_ppstg_or_cond2_reg_6569_pp0_it2 or ap_reg_ppstg_or_cond2_reg_6569_pp0_it3 or
ap_reg_ppstg_or_cond2_reg_6569_pp0_it4 or ap_reg_ppstg_or_cond2_reg_6569_pp0_it5 or
ap_reg_ppstg_or_cond2_reg_6569_pp0_it6 or ap_reg_ppstg_or_cond2_reg_6569_pp0_it7 or
ap_reg_ppstg_or_cond2_reg_6569_pp0_it8 or ap_reg_ppstg_or_cond2_reg_6569_pp0_it9 or
ap_reg_ppstg_or_cond2_reg_6569_pp0_it10)
begin
    if (((ap_ST_pp0_stg0_fsm_2 == ap_CS_fsm) & ~((ap_sig_bdd_147 & (ap_const_logic_1 ==
    ap_reg_ppiten_pp0_it1)) | (ap_sig_bdd_191 & (ap_const_logic_1 == ap_reg_ppiten_pp0_it11))) & ((~(
    ap_const_lv1_0 == ap_reg_ppstg_tmp_7_reg_6253_pp0_it1) & (ap_const_lv1_0 == or_cond2_reg_6569)) |
    (~(ap_const_lv1_0 == ap_reg_ppstg_tmp_7_reg_6253_pp0_it2) & (ap_const_lv1_0 ==
    ap_reg_ppstg_or_cond2_reg_6569_pp0_it2)) | (~(ap_const_lv1_0 ==
    ap_reg_ppstg_tmp_7_reg_6253_pp0_it3) & (ap_const_lv1_0 == ap_reg_ppstg_or_cond2_reg_6569_pp0_it3))
     | (~(ap_const_lv1_0 == ap_reg_ppstg_tmp_7_reg_6253_pp0_it4) & (ap_const_lv1_0 ==
    ap_reg_ppstg_or_cond2_reg_6569_pp0_it4)) | (~(ap_const_lv1_0 ==
    ap_reg_ppstg_tmp_7_reg_6253_pp0_it5) & (ap_const_lv1_0 == ap_reg_ppstg_or_cond2_reg_6569_pp0_it5))
     | (~(ap_const_lv1_0 == ap_reg_ppstg_tmp_7_reg_6253_pp0_it6) & (ap_const_lv1_0 ==
    ap_reg_ppstg_or_cond2_reg_6569_pp0_it6)) | (~(ap_const_lv1_0 ==
    ap_reg_ppstg_tmp_7_reg_6253_pp0_it7) & (ap_const_lv1_0 == ap_reg_ppstg_or_cond2_reg_6569_pp0_it7))
     | (~(ap_const_lv1_0 == ap_reg_ppstg_tmp_7_reg_6253_pp0_it8) & (ap_const_lv1_0 ==
    ap_reg_ppstg_or_cond2_reg_6569_pp0_it8)) | (~(ap_const_lv1_0 ==
    ap_reg_ppstg_tmp_7_reg_6253_pp0_it9) & (ap_const_lv1_0 == ap_reg_ppstg_or_cond2_reg_6569_pp0_it9))
     | (~(ap_const_lv1_0 == ap_reg_ppstg_tmp_7_reg_6253_pp0_it10) & (ap_const_lv1_0 ==
    ap_reg_ppstg_or_cond2_reg_6569_pp0_it10))))) begin
        grp_operation_fu_1821_ap_ce = ap_const_logic_1;
    end else begin
        grp_operation_fu_1821_ap_ce = ap_const_logic_0;
    end
end
```

Figure 17: Example of HLS generated Verilog Code.

Implementing in HDL

The HDL implementation was coded in VHDL. The kernel convolution was inserted into a HLS generated pass through (i.e. no filter) project. The pass through filter project was used to generate the input and output AXI stream interfaces used to connect into the Xilinx TRD. Using HLS generated AXI Stream interfaces removed any resource or performance difference caused by differing AXI implementations.

To operate within the TRD framework timing, the entire kernel convolution needed to happen in a single 'fast' 150 MHz cycle. The first step in the kernel design was to buffer up 9 full rows of pixels. FPGA internal block RAM resources were used to form pixel buffers. These pixel buffers cannot be used directly for kernel convolution because block RAMs do not provide single cycle random access to multiple addressable locations. Data in the pixel window used for convolution would need to be cached locally from the pixel buffers for simultaneous access. This caching was accomplished with pixel shift registers. The shift registers are fed via the line buffers and are shift data through on every clock cycle. The convolution is performed with each pixel in the shift register on every cycle. Each pixel in the kernel window cache is multiplied by a kernel coefficient. The VHDL kernel coefficients are shown in Figure 18. The kernel convolution is shown in Figure 19.

```
signal kernel_coef : kernel_coef_matrix := (0 => (2, 3, 4,  4,  5,  4,  4,  3, 2),
                                            1 => (3, 4, 6,  7,  7,  7,  6,  4, 3),
                                            2 => (4, 6, 8,  9,  10, 9,  8,  6, 4),
                                            3 => (4, 7, 9,  11, 12, 11, 9,  7, 4),
                                            4 => (5, 7, 10, 12, 12, 12, 10, 7, 5),
                                            5 => (4, 7, 9,  11, 12, 11, 9,  7, 4),
                                            6 => (4, 6, 8,  9,  10, 9,  8,  6, 4),
                                            7 => (3, 4, 6,  7,  7,  7,  6,  4, 3),
                                            8 => (2, 3, 4,  4,  5,  4,  4,  3, 2));
```

Figure 18: HDL Kernel Coefficient Code Snippet

```
281         -- Compute Hadamard product of pixel window and our kernel
282         ------------------------------------------------------------
283         for i in 8 downto 0 loop
284            for j in 8 downto 0 loop
285              -- Multiply through by constant (lut... not DSP multiply)
286              -- Note: [R,G,B]_kernel_window_result is an array of 81 values
287              R_kernel_window_result((i*9)+(j)) <= std_logic_vector(unsigned(
                 R_kernel_window_cache(i)(j)) * to_unsigned(kernel_coef(i)(j),4));
288              G_kernel_window_result((i*9)+(j)) <= std_logic_vector(unsigned(
                 G_kernel_window_cache(i)(j)) * to_unsigned(kernel_coef(i)(j),4));
289              B_kernel_window_result((i*9)+(j)) <= std_logic_vector(unsigned(
                 B_kernel_window_cache(i)(j)) * to_unsigned(kernel_coef(i)(j),4));
290            end loop;
291         end loop;
292         ------------------------------------------------------------
```

Figure 19: HDL Kernel Convolution Code Snippet

Summing the elements of the resultant matrix in a single cycle would require a significant number of resources. To reduce the number of required resources by this implementation, the sum operation was pipelined. The pipeline consists of four stages. In the pipeline three intermediates are summed at each stage. The number of intermediates at each state is 81, 27, 9, 3, then 1. The snippet of VHDL that performs the sum is included in Figure 20.

```
294         -- Stage 1 Sum (Reduce 81 entries to 27 partial sums)
295         ----------------------------------------------------
296         for i in 26 downto 0 loop
297            R_kernel_sum_stage_1(i) <= std_logic_vector(resize(unsigned(
               R_kernel_window_result(i)),17) + resize(unsigned(R_kernel_window_result(i+27)),
               17) + resize(unsigned(R_kernel_window_result(i+54)),17));
298            G_kernel_sum_stage_1(i) <= std_logic_vector(resize(unsigned(
               G_kernel_window_result(i)),17) + resize(unsigned(G_kernel_window_result(i+27)),
               17) + resize(unsigned(G_kernel_window_result(i+54)),17));
299            B_kernel_sum_stage_1(i) <= std_logic_vector(resize(unsigned(
               B_kernel_window_result(i)),17) + resize(unsigned(B_kernel_window_result(i+27)),
               17) + resize(unsigned(B_kernel_window_result(i+54)),17));
300         end loop;
301         ----------------------------------------------------
302
303         -- Stage 2 Sum (Reduce 27 partial sums to 9 parital sums)
304         ----------------------------------------------------------
305         for i in 8 downto 0 loop
306            R_kernel_sum_stage_2(i) <= std_logic_vector(unsigned(R_kernel_sum_stage_1(i)) +
                unsigned(R_kernel_sum_stage_1(i+9)) + unsigned(R_kernel_sum_stage_1(i+18)));
307            G_kernel_sum_stage_2(i) <= std_logic_vector(unsigned(G_kernel_sum_stage_1(i)) +
                unsigned(G_kernel_sum_stage_1(i+9)) + unsigned(G_kernel_sum_stage_1(i+18)));
308            B_kernel_sum_stage_2(i) <= std_logic_vector(unsigned(B_kernel_sum_stage_1(i)) +
                unsigned(B_kernel_sum_stage_1(i+9)) + unsigned(B_kernel_sum_stage_1(i+18)));
309         end loop;
310         ----------------------------------------------------------
311
312         -- Stage 3 Sum (Reduce 9 partial sums to 3 parital sums)
313         ----------------------------------------------------------
314         for i in 2 downto 0 loop
315            R_kernel_sum_stage_3(i) <= std_logic_vector(unsigned(R_kernel_sum_stage_2(i)) +
                unsigned(R_kernel_sum_stage_2(i+3)) + unsigned(R_kernel_sum_stage_2(i+6)));
316            G_kernel_sum_stage_3(i) <= std_logic_vector(unsigned(G_kernel_sum_stage_2(i)) +
                unsigned(G_kernel_sum_stage_2(i+3)) + unsigned(G_kernel_sum_stage_2(i+6)));
317            B_kernel_sum_stage_3(i) <= std_logic_vector(unsigned(B_kernel_sum_stage_2(i)) +
                unsigned(B_kernel_sum_stage_2(i+3)) + unsigned(B_kernel_sum_stage_2(i+6)));
318         end loop;
319         ----------------------------------------------------------
320
321         -- Stage 4 Sum (Reduce 3 partial sums to 1 sum)
322         -----------------------------------------------
323         R_kernel_sum_stage_4 <= std_logic_vector(unsigned(R_kernel_sum_stage_3(0)) +
            unsigned(R_kernel_sum_stage_3(1)) + unsigned(R_kernel_sum_stage_3(2)));
324         G_kernel_sum_stage_4 <= std_logic_vector(unsigned(G_kernel_sum_stage_3(0)) +
            unsigned(G_kernel_sum_stage_3(1)) + unsigned(G_kernel_sum_stage_3(2)));
325         B_kernel_sum_stage_4 <= std_logic_vector(unsigned(B_kernel_sum_stage_3(0)) +
            unsigned(B_kernel_sum_stage_3(1)) + unsigned(B_kernel_sum_stage_3(2)));
326         -----------------------------------------------
```

Figure 20: HDL Kernel Product Sum Code Snippet

Working Implementation

Both implementations were brought up successfully on the ZC702 with the TRD and work on 1080p video without dropping frames. For each implementation, the TRD Sobel filter was removed and replaced with the custom kernel filter. Figure 21 is a photograph of the TRD generated test pattern without any filtering. Figure 22 is a photograph of the test pattern after enabling the filtering. In Figure 22 the fine lines that were present in Figure 21 are filtered into a solid color.

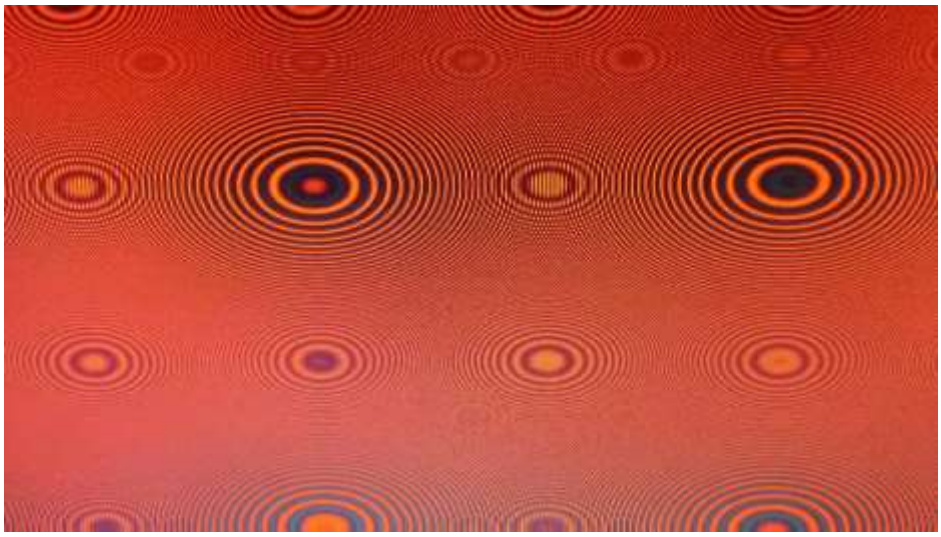

Figure 21: TRD Test Pattern Video (No Filter)

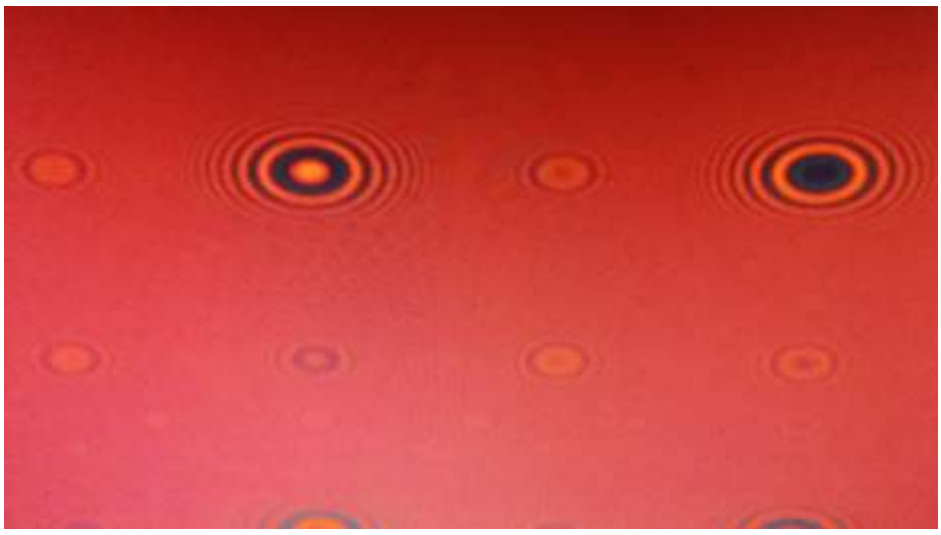

Figure 22: Test Pattern Video (Filtered)

Analysis

The HLS solution and HDL solution were compared for differences in performance (resource utilization, maximum theoretical frequency) and Non-Recurring Engineering (NRE) cost.

1. Resources

One of the more important metrics applied to a custom logic design is resource utilization. FPGA's contain finite resources, and typically FPGA's with fewer resources cost less. Designs with fewer resources use less power. This paper includes utilization for Look Up Tables (LUTs), Flip Flops (FFs), Block RAMs (BRAMs) and Digital Signal Processing (DSP) FPGA primitives (DSP48s).

Optimal Routing Utilization

Each RTL implementation was synthesized, routed, and placed in an empty project to get an idea of the ideal resource utilization. The version of Vivado used is documented in Figure 23.

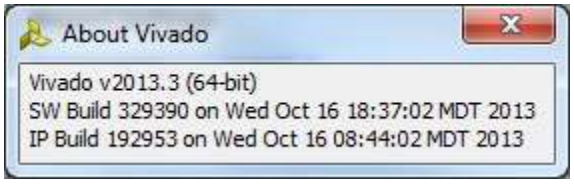

Figure 23: Vivado 2013.3 Version Info

The device targeted was xc7z020clg484-1. The 'Vivado Synthesis Defaults (Vivado Synthesis 2013)' synthesis strategy was used. The 'Performance_Explore (Vivado Implementation 2013)' implementation strategy was used. The device clock was constrained to 150 MHz.

Table 1: Empty Project Resource Utilization

	HDL	HLS	Total Available
LUT	2989	4827	53200
FF	6139	5970	106400
BRAM	12	12	140
DSP48	0	0	220

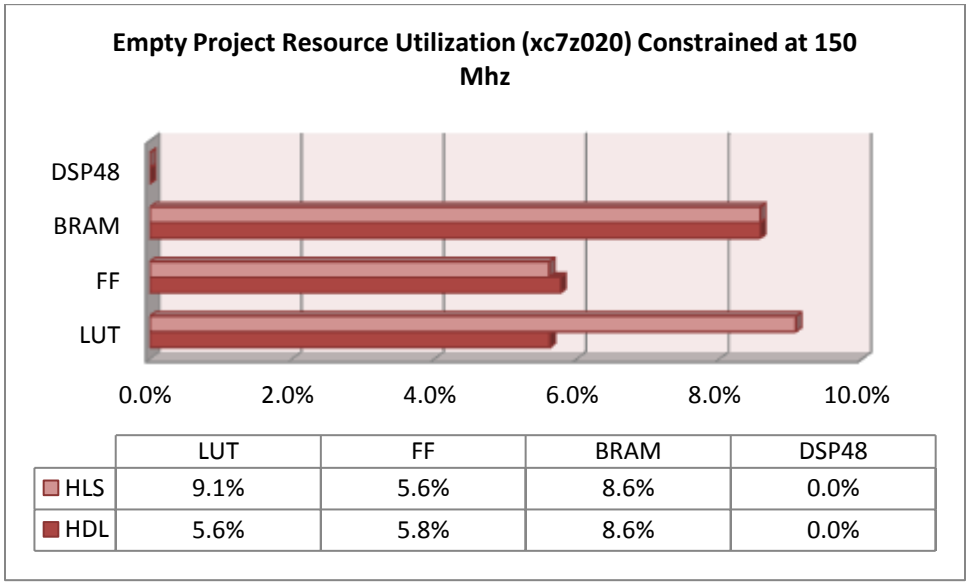

Figure 24: Empty Project Resource Utilization

The HLS implementation used an additional 61% (1838) more LUTs than the HDL implementation. All other resources were negligible.

Congested Routing Utilization

The resource utilization of the full TRD was also recorded. The TRD includes many other custom logic modules; resource utilization results from the TRD better indicate real world (non-ideal) results.

Synthesis and implementation of the Xilinx TRD was performed with Xilinx Vivado 2013.4. Full version information is documented in Figure 25.

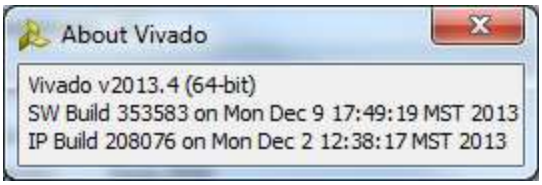

Figure 25: Vivado 2013.4 Version Info

The device targeted was the xc7z020clg484-1. The 'Vivado Synthesis Defaults (Vivado Synthesis 2013)' synthesis strategy was used. The 'Performance_Explore (Vivado Implementation 2013)' implementation strategy was used. The device clock was constrained to 150 MHz.

Table 2: TRD Resource Utilization

	HDL	HLS	Total Available
LUT	23767	25598	53200
FF	34160	34038	106400
BRAM	59.5	59.5	140
DSP48	23	23	220

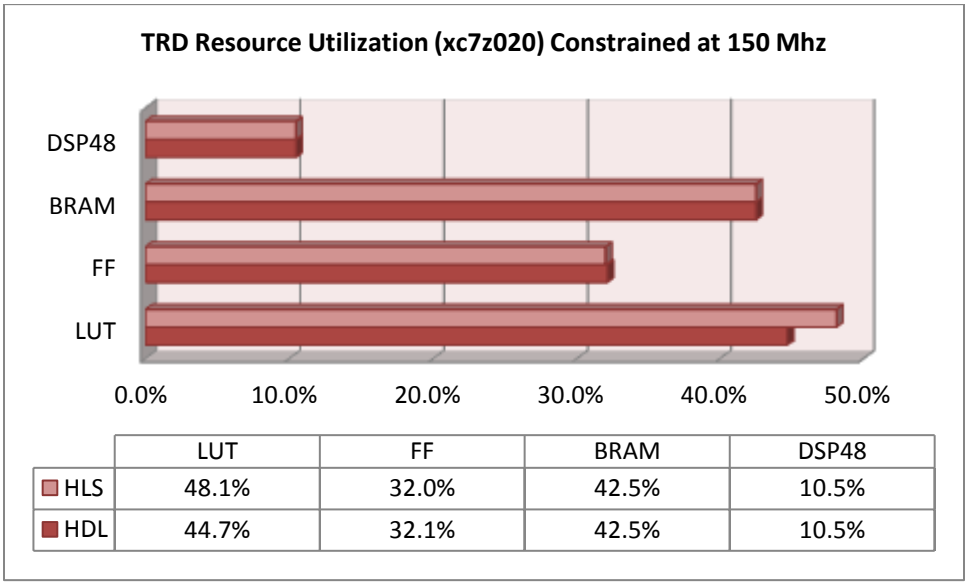

Figure 26: TRD Resource Utilization

Summary

The HLS implementation used (1831) more LUTs than the HDL implementation. All other resources were negligible.

2. Speed

Maximum operational speed is a measure of how fast the implementation can run before the design no longer functions. Specific designs have additional constraints including routing congestion and keep out areas that cause project specific clock speed reductions, but maximum operational speed provides an indication to whether a particular module design will have problems operating at a specific speed.

Maximum operational speed was determined by decreasing the clock period constraint in the 'Empty Project' and checking implementation timing closure. Summary results from this experiment are documented in Table 3.

Table 3: Empty Project Timing Closure

	181 MHz	200 MHz	222 MHz
HLS	Pass	Fail	Fail
HDL	Pass	Pass	Fail

Summary

The HDL implementation maximum frequency was marginally faster than the HLS implementation's. Both implementations closed timing at the required 150 MHz.

3. Non-recurring Engineering

Reducing the amount of engineering effort required to complete a custom logic design can be a very important performance objective. Total Non-recurring Engineering (NRE) was recorded for each of the implementations. NRE can be dependent on the skills and abilities of the engineer along with proper training. To help understand the NRE listed in this section, some background on the engineer performing the work is warranted. The primary author of this paper (Mike Zwagerman) has 8 years of industry experience: 4.5 years of embedded software, and 3.5 years

of custom logic design with HDLs. Various tutorials on HLS were performed before attempting the kernel design and were not included in the design NRE sum. NRE totals are available in Table 4.

Table 4: NRE Totals

	NRE
HLS	15 hrs
HDL	33 hrs

Note: 16 hours of additional effort (not included in the current total) were required to update the HLS pass through project to accept the HDL design. See an example of HLS generated code in Figure 17.

Summary

Implementing the HDL design took more than double the effort of the HLS design.

Analysis Summary

The HLS design was implemented in half of the time, but required 61% more LUTs and did not perform as fast in operational maximum frequency tests.

Conclusion

High Level Synthesis is a design process that is anticipated to replace the antiquated and time consuming approach of designing digital logic in HDLs. Instead of coding RTL with cumbersome HDLs, the HLS process parses high level software languages and generates synthesizable RTL. Designing in high level software languages provide a mechanism for rapid development and easy modification. Xilinx Vivado recently offered a new HLS tool (Vivado HLS) to mainstream audiences. HLS claims to provide similar implementation performance to traditional methods. It can be difficult to determine the suitability of HLS vs HDLs without benchmarking. This paper provides a use case analysis of an example digital logic algorithm. Resultant performance and NRE data from this use case analysis can be used as benchmark data for deciding between HLS and HDLs design flows

This paper documents the implementation of an example digital logic algorithm in both HLS and HDL process flows. The HLS implementation took significantly less time to produce a functional module, but the HLS implementation was slower and uses significantly more hardware resources. This paper also includes a number of lessons learned regarding using the HLS design process flow.

Future Work

This paper describes an implementation of an example algorithm; future work could implement a different or many different algorithms to determine of the results in this paper were statistically significant. The FPGA implementation used for this paper was naïve un-optimized HSL and HDL; future work could optimize the implementations to determine if the results remain consistent across optimization level.

Works Cited

1. Coussy, P., D. D. Gajski, M. Meredith, and A. Takach. "An Introduction to High-Level Synthesis." *Design & Test of Computers, IEEE* 26.4 (2009): 8-17. Web. 29 Jan. 2015.

2. "Vivado High-Level Synthesis." . Xilinx Inc., n.d. Web. 2 Dec. 2014. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

3. "Kernel (image processing)." . Wikipedia, 9 Aug. 2014. Web. 2 Dec. 2014. http://en.wikipedia.org/wiki/Kernel_%28image_processing%29

4. "ZYNQ-7000 EPP ZC702 EVALUATION KIT ." . Xilinx Inc., 2012. Web. 2 Dec. 2014. http://www.xilinx.com/publications/prod_mktg/zynq-7000-kit-product-brief.pdf

5. "Zynq Base TRD 2013.3." . Xilinx Inc., n.d. Web. 2 Dec. 2014. http://www.wiki.xilinx.com/Zynq+Base+TRD+2013.3

6. Haoxing, Ren. "A Brief Introduction on Contemporary High-Level Synthesis." *IC Design & Technology (ICICDT), 2014 IEEE International Conference on 28-30 May 2014* : 1-4. Print.

7. "Xilinx buys high-level synthesis EDA vendor." . EE Times, 31 Jan. 2011. Web. 2 Dec. 2014. http://www.eetimes.com/document.asp?doc_id=1258504

8. "XCN12014 - Product Change Notice for AutoESL." . Xilinx Inc., 6 Aug. 2012. Web. 2 Dec. 2014. http://www.xilinx.com/support/documentation/customer_notices/xcn12014.pdf

9. "UG998 - Introduction to FPGA Design with Vivado High-Level Synthesis." . Xilinx Inc., 2 July 2013. Web. 2 Dec. 2014.