Optimization Techniques for a High Level Synthesis Implementation of the Sobel Filter

Josh Monson, Mike Wirthlin, Brad L Hutchings
NSF Center for High-Performance Reconfigurable Computing (CHREC)
Department of Electrical and Computer Engineering, Brigham Young University, Provo, Utah, USA
jsmonson@gmail.com, wirthlin@ee.byu.edu, hutch@ee.byu.edu

***Abstract*—For many application-specific computations, FPGA-based computing systems have been shown to provide superior performance per Watt than many general-purpose architectures. However, the benefits of FPGA-based computing are difficult to exploit since FPGAs are challenging to program and require advanced hardware design skills. Recent developments in High Level Synthesis (HLS) provide the ability to create FPGA compute accelerators entirely in 'C' code. Because the circuits are described in 'C', it may be possible for software programmers to "program" FPGA accelerator circuits. This paper explores the challenges faced by software programmers when using HLS to implement computing kernels within FPGAs and identifies the specific new knowledge and skills required by these programmers to succeed at the task. A high-performance Sobel edge-detection acceleration core is developed and used to demonstrate the use of the Vivado HLS tool. A variety of simple directives and code restructuring steps are applied to demonstrate a variety of Sobel edge-detection accelerators that vary in performance from 10.9 frames per second (fps) to 388 fps. The concepts outlined in this paper suggest that with proper training, software programmers are able to create a wide range of FPGA acceleration circuits.**

Keywords*—FPGA, accelerator, programmer, C-RTL, high-level synthesis

I. Introduction

FPGAs have been shown to provide high performance computing at relatively low power when compared to many other general-purpose programming architectures. FPGAs, however, are much more difficult to "program" than traditional general-purpose architectures. A number of high-level design tools have been introduced that allow the programmer to create FPGA computing engines with less-specialized knowledge. The Vivado HLS tool, for example, allows a person to create an FPGA design using the "C" programming language. The availability of "C" tools for hardware suggests that software programmers who are not familiar with designing FPGA circuits may be able to create FPGA computing circuits. This paper seeks to investigate if FPGA-based computing systems are still inaccessible to programmers in light of the latest developments in CAD.

The experimental process described in this paper begins with a conventional software implementation of the Sobel edge-detection algorithm obtained from a standard image-processing library[1]. The original code is then iteratively modified and refined until it approaches the theoretical performance of the system. Sobel edge-detection was chosen because it fits the scope of this paper and also because, excepting the constants used in the convolution kernel, it is identical to one of the modules used in a more complex image-processing system currently under development in our lab.

This paper focuses on the development of an FPGA-based accelerator in a constrained system environment by a software programmer. In particular, the Sobel application was designed to run on the ZYNQ-based ZED board but could easily target any FPGA system with a high-performance DDR interface. It assumes the existence of an FPGA board containing a fixed set of I/O interfaces that have been pre-determined by the manufacture or otherwise provided by a hardware designer. The programmer is provided with a standard interface for reading and writing data that can be accessed from their program. Maximum data bandwidth through the I/O interface is fixed and is documented for the programmers so that they can predict theoretical maximum performance and compare it with the performance of their accelerator as they refine and optimize their code. The development environment is the Eclipse-based Vivado High Level Synthesis (HLS) environment [2].

This paper purposely ignores two important questions: 1) will the generated FPGA design meet the clock frequency estimated by the synthesis tool?, and 2) will the generated design "fit" in the desired device? Although these questions will need to be considered, they are of secondary importance to this effort because this effort focuses on the programmer's ability to develop and functionally debug C code that describes an FPGA accelerator.

This effort is different from previously-reported work because it illustrates the code optimization process from the perspective of a programmer. The goal is to identify specific skills and knowledge that a software programmer must learn and apply in order to effectively use HLS to develop FPGA-based compute accelerators.

II. Previous Work

Berkley Design Technology, Inc (BDTi) published a report in 2010 [3] on the usability and quality of results of the HLS tool AutoPilot (purchased by Xilinx and renamed Vivado HLS). They found that creating a hardware accelerator with AutoPilot required about the same amount of effort as programming a DSP. They also concluded that "a typical DSP software engineer with an awareness of hardware architecture fundamentals (e.g., pipelining, latency) can learn to effectively use AutoPilot."

Other papers, however, have acknowledged issues that still make HLS challenging for programmers. Cong et. al. [4] and Neuendorffer and Vissers [5] explain that many HLS solutions

lack efficient memory hierarchy support and leave software programmers exposed to more low-level details than most are comfortable with. Cong et. al. also mentions in-system debugging and performance evaluation.

Handel-C was an early attempt at making FPGA programming accessible to software programmers [6]. Others have attempted to improve accessibility by building on-top of existing HLS tools. Such tools typically use specifications in dataflow programming languages [7], untimed C [8] or CUDA [9] and generate HLS amenable code and automatically apply tool-specific directives.

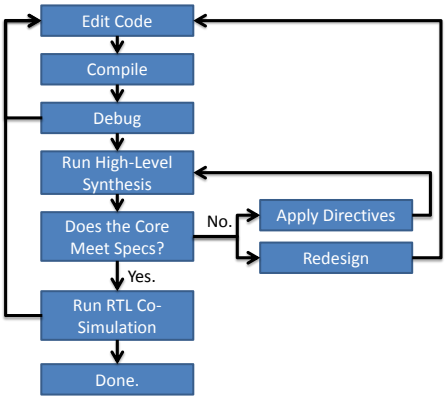

Fig. 1. Shows the general Flow of the High-Level Synthesis Process.

III. Vivado High Level Synthesis

The basic flow for HLS synthesis is shown in Figure 1. Many of the steps are similar to the edit-compile-debug loop familiar to programmers. Within the Vivado HLS environment, programmers can execute the compiled version of their code to determine if it is correct. Once the C code appears to be functionally correct, the programmer can invoke the synthesizer and view the synthesizer log-files to determine if the generated FPGA design achieves the desired performance.

There are two ways to influence how Vivado HLS translates C code into an FPGA design: *directives* and *code structure.* Directives are Vivado HLS commands that constrain how the tool synthesizes the C code into an FPGA design. Directives can affect how resources are used and allocated. The structure of the C code also heavily influences the organization of the FPGA accelerator and programmers typically must significantly restructure their code to meet performance goals.

We have identified four important concepts that a software programmer should understand in order to create effective circuits within VivaldoHLS. These concepts are include: (1) the VivadoHLS synthesizable subset of C, (2) the function and usage of FPGA resources, particularly memories, (3) clocking, scheduling, binding and concurrency, and (4) the HLS synthesis directives. Each of these will be described in more detail below.

a) Synthesizable Subset: Vivado HLS comes with a number of restrictions to facilitate hardware synthesis. These restrictions include: no system calls are allowed, memory allocation must be static, no support for recursion, and no pointers or references to member functions in user-defined classes.

b) FPGA Resource Guidelines: The following guidelines will help the programmer to make the best of use of FPGA resources. Operations requiring fewer bits will run faster and require fewer resources than standard C types. The use local memories will increase data bandwidth and facilitate data reuse and greater internal concurrency.

c) Clocking, Scheduling, Binding and Concurrency: Computations on an FPGA are sequenced relative to an external clock. During synthesis, HLS selects operations and schedules them relative to the external clock according to the semantics of the 'C' program. Once operations are scheduled, they are assigned to a specific resource on the FPGA. To improve performance, HLS will schedule operations concurrently where possible. The user can influence the schedule and resource usage by structuring loops, functions and conditional statements, as will be shown in the examples that follow.

d) Directives: It is vitally important that a software engineer be familiar with the Vivado HLS directives and know when they should be applied. These directives support user-defined HLS optimizations such as memory partitioning, pipelining, loop-unrolling, etc.

IV. Sobel Filter Overview and Performance Measures

The Sobel filter is an edge-detection operator commonly used in image processing. Listing 1 contains a source-code listing for Version 1 of the Sobel filter code that was obtained online [1]. It consists of two 3 X 3 convolution kernels (shown in Lines 1 and 2) that are applied in both the x and y directions. The loops starting at Lines 10 and 11 apply the kernels to every interior pixel in the image. The Sobel algorithm is highly parallel; within a single image all output pixels can be computed simultaneously.

A. System Organization and Maximum Performance

In streamed systems, the Sobel filter is typically an I/O bound computation where the architecture of the surrounding system puts an upper bound on performance. For this example, let us assume that the surrounding system streams 1 pixel/cycle at 150 MHz (Max Clock Rate of High Performance interface on Zynq [10]). Assuming a 640x480 image, the maximum throughput of such a system would be 488 frames per second (FPS).

When compiled to an 2.80 GHz Intel core i7, Listing 1 achieves a throughput of 38.75 FPS. Multi-core and SIMD implementations of similar filters have been reported achieving throughput of 100-1000 FPS [11]. Clearly, the code shown in Listing 1 is not optimized for CPU performance. However, the code is easy to read, contains very few non-synthesizable elements and provides a good starting point for HLS.

B. Performance Measures

In this paper, we optimize several versions of the Sobel filter. Each version is tested both in software and in simulation to ensure it is functionally correct. The estimated clock frequency and cycle latency of each accelerator is used to

```
1  int dx[3][3] = {{1,0,-1},{2,0,-2},{1,0,-1}};
2  int dy[3][3] = {{1,2,1},{0,0,0},{-1,-2,-1}};
3
4  void Sobel(IplImage* img, IplImage* dst) {
5    int step = img->widthStep/sizeof(uchar);
6    uchar* data = (uchar *)img->imageData;
7    uchar* data_dst = (uchar *)dst->imageData;
8
9    int s;
10   for (int i=1; i < img->height-1; i++)
11     for (int j=1; j < img->width-1; j++) {
12       // apply kernel in X ans Y direction
13       int sum_x=0; int sum_y=0;
14       for(int m=-1; m<=1; m++)
15         for(int n=-1; n<=1; n++) {
16           // get the (i,j) pixel value
17           s=data[(i+m)*step+j+n];
18           sum_x+=s*dx[m+1][n+1];
19           sum_y+=s*dy[m+1][n+1];
20         }
21       int sum=abs(sum_x)+abs(sum_y);
22       // set the (i,j) pixel value
23       data_dst[i*step+j]=(sum>255)?255:sum;
24     }
25 }
```

Listing 1. Original C++ Code

calculate the performance of the accelerator in frames (or images) per second (FPS). The performance of an accelerator is also measured in clock cycles per output pixel. In pipelined versions of the accelerator this is equivalent to the pipeline initiation interval.

V. Optimizing the Sobel Filter

To create a high performance accelerator with HLS, the programmer must use appropriate directives and code structure. Achieving high performance with current HLS tools usually requires a programmer to manually determine where to restructure code or apply synthesis directives. This is done by examining the log files and looking for performance bottlenecks. While this process is similar to conventional edit-compile-debug loop of code development, HLS requires additional knowledge and expertise beyond that required for conventional software development. This section will discuss several changes that were made to each version of the code to achieve greater throughput.

A. Optimization 1

The first optimization resulted in the first synthesizable hardware. Unsynthesizeable constructs were removed from the downloaded code and a FIFO-based I/O system was added to the code.

a) Non-Synthesizeable constructs: The process of identifying and replacing non-synthesizeable constructs is largely guided by Vivado HLS. Attempting to compile the code in Listing 1 will result in Vivado HLS issuing a `non-synthesizable type` error and an `unsupported memory access` error. Both of these synthesis errors can be remedied by replacing the `IplImage` pointers (Listing 1, Line 4) with fixed-sized arrays. Once the `IplImage` pointers have been removed, the references to the height and width (Listing 1, lines 10,11) need to be replaced with constants.

The decision to replace the `IplImage` pointers with fixed size arrays (see Listing 2 lines 1,2) is a result of a `array size not known at compile-time error`. Since we are using two-dimensional arrays we no longer need to calculate the offsets and can refer directly to the `src` and `dst` arrays (removing Lines 5-7 from Listing 1 and modifying Lines 17, 23). In this example, the loop bounds were replaced with constants since the height and width of the image are fixed (Listing 2, Lines 10,13).

b) FIFO-based I/O: Directives were added to the second version that specified that the randomly-accessible arrays were to be replaced with FIFOs. These FIFOs increase performance by providing streaming input/output data to and from the accelerator. The FIFOs were supported by line buffers that made it possible to provide random access to incoming streaming data and are implemented by Lines 5-8, 11-12, 15-16, 19, 28-29 in Listing 2.

The use of the 2-D arrays to represent streaming I/O can result in differences between the C-Simulation and RTL co-simulation if the streaming ordering is violated. RTL co-simulation is the process of comparing the output from the compiled C program against a cycle-by-cycle simulation of the RTL code generated by the HLS tool. If the miss-ordering is subtle, the difference may be difficult to detect without actually reading the simulation waveform trace (something a programmer would be uncomfortable with). Vivado HLS issues a warning when reads or writes may be out of order; however, this warning is issued very conservatively, as the warning arose in working versions of our code (as indicated by the results of the RTL cosimulation).

We encountered this issue during the first step in optimization (Optimization 1). Originally, the size of the source input array and the size of the destination array were the same. However, this version of the Sobel filter deals with the edge of the images by only computing the filter for the interior pixels of the image. Thus, results were only written out for the interior pixels of the image. Since the RTL co-simulation was run immediately after modifying the code, we were able to identify the problem without resorting to reading the waveform. However, had we waited to run co-simulation until the accelerator was further along we may have had to dig into the simulation waveform to diagnose the problem. To fix this issue, the size of the destination array was reduced and the array indices were adjusted (Listing 2, Lines 2, 25). The Vivado HLS user guide documents several points in which differences between C-Simulation and RTL co-simulation may differ [2]; however, extremely subtle differences might be difficult (for anyone) to detect without reading the simulation waveform.

Vivado HLS synthesized the code shown in (Listing 2) using its default synthesis behaviors and resulted in an accelerator with sub-optimal performance (42 cycles/pixel) and acheived a clock rate of 150 MHz.

B. Optimization 2

In this section, we use directives to override these default behaviors and improve the performance of the accelerator to 2 cycles/pixel. To further optimize the Sobel filter, one needs to understand how to use directives to optimize loops. By

```
1  void Sobel(uchar src[ROWS][COLS],
2             uchar dst[ROWS-2][COLS-2]) {
3  #pragma HLS INTERFACE ap_fifo port=src
4  #pragma HLS INTERFACE ap_fifo port=dst
5    uchar line_buffer[4][COLS];
6    for(int i=0; i<3;i++){
7      for(int j=0; j<COLS;j++){
8        line_buffer[i][j] = src[i][j]; }}
9    uchar s;
10   for (int i=1; i < ROWS-1; i++){
11     if(i<ROWS-2)
12       line_buffer[(i+2)%4][0] = src[i+2][0];
13     for (int j=1; j < COLS-1; j++) {
14       int sum_x=0; int sum_y=0;
15       if(i<ROWS-2)
16         line_buffer[(i+2)%4][j] = src[i+2][j];
17       for(int m=-1; m<=1; m++)
18         for(int n=-1; n<=1; n++) {
19           s=line_buffer[((i+m)%4)][j+n];
20           sum_x+=(int)s*dx[m+1][n+1];
21           sum_y+=(int)s*dy[m+1][n+1];
22         }
23       int sum=ABS(sum_x)+ABS(sum_y);
24       s=(sum>255)?255:sum;
25       dst[i-1][j-1]=s;
26     }
27     if(i<ROWS-2)
28       line_buffer[(i+2)%4][COLS-1]=
29         src[i+2][COLS-1]; }}
```

Listing 2. First Synthesizable Code Version

default, Vivado HLS leaves for-loops unoptimized (rolled and unpipelined). This behavior allows the programmer to decide whether to optimize the design for area or performance.

Vivado HLS describes the performance of for-loops using the following terms:

- *Loop Latency:* The number of total clock cycles required to execute all iterations of a loop.
- *Trip-Count:* The total number of iterations executed.
- *Iteration Latency:* The number of total clock cycles required to execute a single iteration of a loop. This can be calculated by using the following formula: Iteration Latency = Loop Latency / Trip Count.
- *Loop Pipelining:* Overlapping the execution of loop iterations.
- *Pipeline Initiation Interval (II):* The cycle interval at which a new loop iteration can start.
- *Pipelining Depth:* The latency of a pipelined loop iteration.

There is nothing in the above definitions that would require hardware expertise to understand. In fact, most hardware engineers would probably have to learn the HLS definition of these terms. The *trip count* and *loop latency* for each loop is found in the synthesis report. When a loop has been pipelined the synthesis report will also contain the *pipeline II* and *pipeline depth*.

The pipeline directive should be applied if the programmer is aware of parallelism between loop iterations. In this case, the programmer knows that each output pixel can be computed in parallel with all of the others. Thus, the iterations of Loop 2.1 (which iterates over pixels) should be able to overlap in execution. Therefore, the pipeline directive should be applied to Loop 2.1 (Listing 2, Line 13) .

The synthesis report shows that the latency of Loop 2.1 has decreased significantly (28,710 to 3,190 cycles), mostly due to unrolling the inner-loops for pipelining. However, we see that Vivado HLS was only able to achieve a pipeline II of 5 and a pipeline depth of 6. This indicates that only one cycle of successive iterations is overlapping (this can also be seen using the Vivado HLS schedule viewer). Someone familiar with hardware image filters would know that HLS should be able to reach an II of 1 (or 1 cycle/pixel). But how is a software programmer to know that an II of five is not the best he or she can do?

The Vivado HLS tool provides guidance on this issue. By default, when the pipeline directive is applied, Vivado HLS seeks the minimum II of 1. When it cannot achieve the II of 1, Vivado HLS provides a synthesis warning. Vivado HLS issued the following warning regarding its inability to meet an II of 1:

> *W@ [SCHED-69] Unable to schedule 'load' operation ('s', sobel_ind_fast.cpp:50) on array 'line_buffer' due to limited memory ports.*

To understand this warning, one must recognize that arrays are implemented as local memory resources with limited bandwidth – memory resources are limited to two memory accesses (load or stores) per clock cycle. It is important to note that the real understanding required here deals with memory resource limitations and scheduling rather than the number of ports on a block RAM. The idea of memory "ported-ness" should not be completely unfamiliar to software engineers as the concept of dual-ported register files are taught in undergraduate computer architecture classes. Essentially, the above warning is saying that Vivado HLS might be able to achieve a lower II if `line_buffer` had more ports.

By default, Vivado HLS maps large arrays (i.e., the `line_buffer` in the code examples) to a single memory resource with 2 ports. The warning listed earlier refers to (Listing 2, Line 19) where 9 pixels are read to compute the convolution kernel. Because the line buffer only has 2 ports, the schedule requires 5 cycles to perform 9 reads (and thus an II of 5).

The ARRAY_PARTITION directive allows a single array to be mapped to multiple memory resources. Multi-dimensional arrays can be partitioned along any of their dimensions. When an array is partitioned across a particular dimension, each element in that dimension is mapped to its own memory resource. For example, a 3 x 3 array partitioned across its rows (1st dimension) results in each row having its own resource. In other words, elements [0][0], [0][1], and [0][2] would be mapped into a memory resource with two ports. The second and third rows would be mapped similarly.

Applying the ARRAY_PARTITION directive to the wrong dimension of a large array can have a detrimental effect on the quality of accelerator. For example, partitioning the `line_buffer` across its columns results in each column being mapped to its own memory resource. Since `line_buffer` has 640 columns, this partitioning would require 640 memory resources. A hardware engineer would understand the hardware requirements of this partitioning (lots of BRAM and wide multiplexers), while a software engineer may not. Fortunately, Vivado HLS issues a warning if a partitioned

array will result in a poorly performing circuit; thus, directing a software programmer to try partitioning across the first dimension. Partitioning `line_buffer` across its rows (1st dimension) results in a much more efficient implementation and provides more ports for `line_buffer`.

c) Performance: The result of applying the PIPELINE directive to Loop 2.1 and partitioning `line_buffer` array along its rows resulted in a final II of 2. The reason for the II of 2 and not 1 will be discussed in the next section. The synthesis report estimated that the accelerator could operate at clock frequency of 125.8 MHz (7.95 ns clock period) with a latency of 615,197 clock cycles. The translates into a frame rate of 204 FPS.

C. Optimization 3

Another `limited memory ports on variable line_buffer` warning, stemming from the PIPELINE directive that was applied in the previous subsection, is the starting point for this optimization. Vivado HLS issues this warning because it did not meet the minimum II of one. The line buffer cannot be partitioned further so another solution must be found.

The "limited memory ports" warning leads naturally into an investigation into the number of available ports versus the number of attempted line buffer reads per cycle. The warning again points us to Listing 2, Line 19 where the pixel values are read out of the line buffer. By reviewing the code, we find that three lines of the line buffer are actively read (Line 19). The array was partitioned in the previous optimization so that each line in the line buffer was mapped to its own memory resource (Line 5). Memory resources provide two ports each, thus two ports are available per line. However, each line must be read three times prior to computing a single output pixel (this would require three ports per line). This makes it impossible to compute a new output pixel during each clock. This explains the II of 2 because reading 9 pixels require 2 clock cycles when there are only 6 ports available.

The solution to this problem is to reuse previously-read pixels. Each new output pixel is dependent upon a *neighborhood* of nine input pixels. Fortunately, adjacent neighborhoods overlap and share six of the nine input pixels. If we initially store the nine neighborhood pixels in a local memory resource, only three new pixels need to be read for each new output pixel (one pixel is read per line buffer). This new implementation is referred to as the *windowed* implementation and is shown in Listing 3. The changes between this code and the previous version include declaring the window array (Line 6), filling the window at the beginning of each line (Lines 14-19), reading a new column of pixels (Lines 24-26), and shifting the window (Lines 37-42).

To follow the reasoning used above, a programmer would need to understand multi-ported memories, concurrent reads and be able to perform dependency analysis on small code segments. Multiported memories and concurrent reads should be within the grasp of programmers who are reasonably familiar with microarchitecture, something that is commonly covered in many computer science programs. Dependency analysis is also commonly taught in these programs. Similar concepts also arise when programming DSPs or GPGPUs and these are

```
1  void Sobel(uchar src[ROWS][COLS],
2             uchar dst[ROWS-2][COLS-2]){
3    #pragma HLS INTERFACE ap_fifo port=src
4    #pragma HLS INTERFACE ap_fifo port=dst
5    uchar line_buffer[4][COLS];
6    uchar window[3][3];
7    for(int i=0; i<3;i++){
8      for(int j=0; j<COLS;j++){
9        line_buffer[i][j] = src[i][j]; }}
10   uchar s;
11   for (int i=1; i < ROWS-1; i++){
12     if(i<ROWS-2)
13       line_buffer[(i+2)%4][0] = src[i+2][0];
14     window[0][0] = line_buffer[(i-1)%4][0];
15     window[1][0] = line_buffer[(i)%4][0];
16     window[2][0] = line_buffer[(i+1)%4][0];
17     window[0][1] = line_buffer[(i-1)%4][1];
18     window[1][1] = line_buffer[(i)%4][1];
19     window[2][1] = line_buffer[(i+1)%4][1];
20     for (int j=1; j < COLS-1; j++){
21       int sum_x=0; int sum_y=0;
22       if(i<ROWS-2)
23         line_buffer[(i+2)%4][j] = src[i+2][j];
24       window[0][2] = line_buffer[(i-1)%4][j+1];
25       window[1][2] = line_buffer[(i)%4][j+1];
26       window[2][2] = line_buffer[(i+1)%4][j+1];
27       for(int m=-1; m<=1; m++)
28         for(int n=-1; n<=1; n++){
29           // get the (i,j) pixel value
30           s = window[m+1][n+1];
31           sum_x+=(int)s*dx[m+1][n+1];
32           sum_y+=(int)s*dy[m+1][n+1]; }
33       int sum=ABS(sum_x)+ABS(sum_y);
34       s=(sum>255)?255:sum;
35       // set the (i,j) pixel value
36       dst[i-1][j-1]=s;
37       window[0][0] = window[0][1];
38       window[1][0] = window[1][1];
39       window[2][0] = window[2][1];
40       window[0][1] = window[0][2];
41       window[1][1] = window[1][2];
42       window[2][1] = window[2][2]; }
43     if(i<ROWS-2)
44       line_buffer[(i+2)%4][COLS-1] =
45         src[i+2][COLS-1]; }}
```

Listing 3. Windowed Sobel Implementation

usually feasible programming targets for programmers. This redesign achieves an II of 1 (1 cycle/pixel), a clock rate of 120 MHz, and an throughput of 388 FPS.

D. Performance Summary

At this point, 79.5% (388 FPS out of 488 FPS) of the maximum theoretical performance has been acheived. Table I shows the performance of each version of the accelerator. We see that the cycle latency of OPT 3 is only 1% above the theoretical minimum cycle latency (maximum performance). This indicates that remaining performance gap is primarily due to the achieved clock rate of the accelerator. The programmer can respond to this issue by asking Vivado HLS to attempt to achieve a higher clock rate, however, this may or may not help. If Vivado HLS is unable to achieve a higher clock rate, an advanced timing analysis may reveal areas of the C source that could be change to improve timing. However, this kind of analysis is beyond the skill-set of a software programmer.

TABLE I. SUMMARY OF OPTIMIZATION RESULTS

Version	Cycles/Pixel	Cycle Latency	Frequency	FPS
Software	N/A	N/A	N/A	38.75
OPT. 1	42	13,726,264	150 MHz	10.9
OPT. 2	2	615,197	125.7 MHz	204.4
OPT. 3	1	310,711	120.6 MHz	388
Theoretical Max.	1	307,200	150 MHz	488

VI. CONCLUSIONS AND FUTURE WORK

As our Sobel filter example has shown, a new skill-set beyond traditional C programming is needed to effectively use HLS tools. The programmer needs to be able to evaluate an HLS design, identify bottle-necks, and apply directives and code restructing optimizations. A good understanding of the following can help software programmers using HLS to succeed.

- *clock-based scheduling, binding, and concurrency* ,
- *basic HLS terms (pipeline II, latency, etc.)*
- *usage and limitations of FPGA resources, particularly memories*,
- *the effect of HLS synthesis directives*

It is likely, however, that a software engineer will encounter problems that will require the assistance of a hardware engineer. For example, much of the functional verification can be performed by executing a normally-compiled C program using a standard debugger. However, verification is where things can become difficult for the programmer. Unfortunately, there is no guarantee that the generated FPGA design will behave the same as the executed C program. For example, in Optimization 1, FIFO buffers were inserted by using pragmas (Listing 2, Lines 3-4). The source and destination arguments (`src` and `dst` in the listing) are still declared within the C code to be normal randomly-accessible arrays in the code and they remain randomly-accessible to body of the code. However, in the generated FPGA hardware, the arrays will be replaced with FIFOs that can only produce and consume data in a specific order. This may lead to a mismatch between the behavior of the executed C code and the generated FPGA design if the C code violates read or write order; the C code may appear to function correctly but the generated hardware will not. Finding these kinds of mismatches can be difficult, especially for a programmer.

Vivado HLS does provide a C++ class (hls::stream) that will enforce the correct read and write ordering. Using the hls::stream class does require the programmer to make minor modifications to the test bench. During C-simulation the hls::stream class is modeled using an infinite queue; however, this is not the case during RTL co-simulation or in the FPGA implementation. At this point, the programmer must be careful to use the read and write model (blocking or non-blocking) that best matches the surrounding system or the use of this C++ model may cause differences between RTL co-simulation and the FPGA implementation.

Current in-system debug techniques fall short due to the lack of visibility and solutions (such as chipscope) require the engineer to read waveform diagrams (as well as determine which nets to monitor). Cong et al. proposed including functionality to monitor critical buffers for performance and dead lock conditions. They also suggested including the ability to set break-points at the C code and read-back internal hardware state. It is likely that formal techniques can prove or disprove the functional equivalence of the FPGA RTL and the original C code. However, these tools will only prove to be effective for programmers if they are fully integrated into the HLS suite and their use is automated. Note that none of these features exist in the current release of Vivado HLS (2012.4).

Even so, it is clear that FPGA accelerators are becoming much more accessible to software programmers. Programmers can currently write descriptions of FPGA accelerators and participate in their development and help debug them at the C level. When necessary, they can request assistance from a hardware designer if verification becomes an issue. As such programmers can play a significant role in teams that focus on the development of FPGA accelerators.

ACKNOWLEDGMENT

This work was supported by the I/UCRC Program of the National Science Foundation under Grant No. 0801876.

REFERENCES

[1] G. A. Ramirez. (2009, April) sobel.cpp. [Online]. Available: http://www.cs.utep.edu/ofuentes/AI/sobel.cpp

[2] Xilinx, *Vivado Design Suite User Guide: High-Level Synthesis (UG902 (v2012.4))*, PDF File, Xilinx, San Jose, California, December 2012.

[3] B. D. Technology. (2010) An independent evaluation of: The autoesl autopilot high-level synthesis tool. [Online]. Available: http://www.bdti.com/MyBDTI/pubs/AutoPilot.pdf

[4] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: from prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473 –491, Apr. 2011.

[5] S. Neuendorffer and K. Vissers, "Streaming systems in FPGAs," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, ser. Lecture Notes in Computer Science, M. Berekovi, N. Dimopoulos, and S. Wong, Eds. Springer Berlin Heidelberg, Jan. 2008, no. 5114, pp. 147–156. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-540-70550-5_17

[6] S. Loo, B. Wells, N. Freije, and J. Kulick, "Handel-c for rapid prototyping of VLSI coprocessors for real time systems," in *Proceedings of the Thirty-Fourth Southeastern Symposium on System Theory, 2002*, 2002, pp. 6 – 10.

[7] N. Siret, M. Wipliez, J. F. Nezan, and F. Palumbo, "Generation of efficient high-level hardware code from dataflow programs," in *Proceedings of Design, Automation and test in Europe (DATE)*, Dresden, Allemagne, Mar. 2012, p. NC. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00763804

[8] B. Schafer, T. Takenaka, and K. Wakabayashi, "Adaptive simulated annealer for high level synthesis design space exploration," in *International Symposium on VLSI Design, Automation and Test, 2009. VLSI-DAT '09*, Apr. 2009, pp. 106 –109.

[9] I. Mavroidis, I. Mavroidis, I. Papaefstathiou, L. Lavagno, M. Lazarescu, E. de la Torre, and F. Schafer, "FASTCUDA: open source FPGA accelerator & hardware-software codesign toolset for CUDA kernels," in *2012 15th Euromicro Conference on Digital System Design (DSD)*, Sep. 2012, pp. 343 –348.

[10] Xilinx, *Zynq-7000 All Programmable SoC Technical Reference Manual (UG585 (v1.4))*, PDF File, Xilinx, San Jose, California, November 2012.

[11] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: ACM, 2012, p. 4756. [Online]. Available: http://doi.acm.org/10.1145/2145694.2145704