WHITE PAPER

Intel® HLS Compiler

Intel® HLS Compiler: Fast Design, Coding, and Hardware

The Modern FPGA Workflow

Authors

Melissa Sussmann
HLS Product Manager
Intel® Corporation

Tom Hill
OpenCL™ Product Manager
Intel Corporation

Abstract

This paper presents the design flow enabled by the Intel® HLS Compiler while displaying an image processing design example. The Intel HLS Compiler tool flow performs important optimization tasks such as datapath pipelining and features the ability to target hardened floating-point blocks on Intel Arria® 10 FPGAs. The design flow begins with an algorithm, followed by a software C++ implementation, compilation, verification, and optimization of the FPGA design. The Intel HLS Compiler features and results are highlighted as we move through the design example.

Introduction

The Intel HLS Compiler is a high-level synthesis (HLS) tool that takes in untimed C++ as input and generates production-quality RTL that is optimized for Intel FPGAs. This tool accelerates verification time over RTL by raising the abstraction level for FPGA hardware design. Models developed in C++ are typically verified orders of magnitude faster than RTL and require 80% fewer lines of code†. The Intel HLS Compiler generates reusable, high-quality code that meets performance and is within 10%-15% of the area of hand-coded RTL.[1†]

This example targets an Intel Arria 10 FPGA device family, which includes up to 1,688 independent IEEE 754 single precision floating-point multiply-add blocks that deliver up to 1.5 tera floating point operations per second (TFLOPs) of digital signal processing (DSP) performance. These DSP blocks can also be configured for fixed-point arithmetic and support up to 3,376 independent 18x19 multipliers. The Intel HLS Compiler will generate hardware targeting these arithmetic blocks based on the user-defined data types.

Design Example Overview

We use a simple small convolution model to realize the composition of collections of linear image filters. These image filters have a range of applications such as photo filtering (http://setosa.io/ev/image-kernels/) and deep-learning layers used in deep-learning stacks involving object recognition (https://hackernoon.com/visualizing-parts-of-convolutional-neural-networks-using-keras-and-cats-5cc01b214e59).

Table of Contents

The following design process steps will be explored:

Step #1 - Use the default workflow to generate an initial implementation of a single convolution in software that will target single precision data types. We simplify the implementation by targeting floating point and thus allow the input image to remain in floating point, eliminating the need to worry about clipping and other artifacts.

Step #2 - Use the throughput and resource analysis tools included with the Intel HLS Compiler to implement micro-architectural optimizations for timing and memory.

Step #3 - Demonstrate how simple it is to support composed convolutions for achieving multiple-filter effects.

HLS Design Example

A common filtering operation to implement over a multidimensional array is a convolution filter. This is a ubiquitous operation across a variety of disciplines, and best envisioned as a dot product of one array (the Kernel (K) array) over the other (the Target (T) array). The result is defined as multiplying each cell in the kernel by that in the target, and summing them up. The value of the resultant summation is assigned to the cell in the target array for which convolved array is seen. Where* is the convolution operator, we denote C=K*T as the convolution of T by K. In this case, the dimension of the K is small relative to the T. When K and T are large enough, it is faster to use the fast Fourier transform and the convolution theorem to compute this filter. In our case, the T is an image of size ~512x512, and the K is a relatively small, of size (2N+1) x (2N+1), where N represents an integer number.

We can achieve different results with the same design by simply changing the values of the K coefficient table. We verify the functional correctness of our algorithm with the following three examples:

Blur Example

The following example shows the result of calculating the dot product for the operator

K =⅛(½,1,½; 1,2,1; ½,1,½) against the image. The K refers to a Blur kernel. The example is referenced from http://setosa.io/ev/image-kernels/. Let's walk through the example on how to apply the following 3x3 blur kernel to the image of a face using the following K array:

.0625	.125	.0625
.125	.25	.125
.0625	.125	.0625

For each 3x3 block of pixels shown on the left image of Figure 1, we multiply each pixel by the corresponding entry of the kernel and then take the sum. That sum becomes the new pixel shown on the right image of Figure 1.

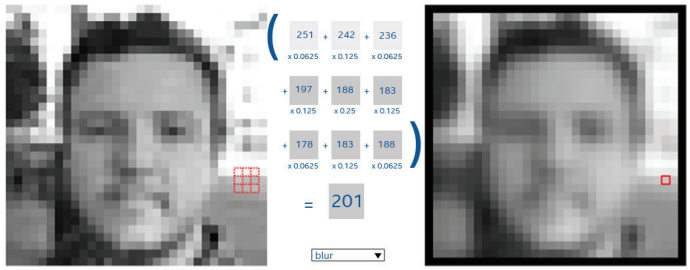

Figure 1. Blur Example

Sharpen Example

The second example performs image sharpening using the same algorithm but with the K array set to the values shown below. The image shown in Figure 2 is converted to greyscale.

0.0	-1.0	0.0
-1.0	5.0	-1.0
0.0	-1.0	0.0

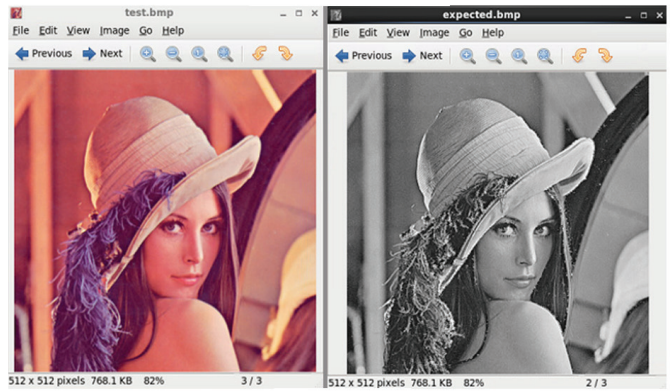

Figure 2. Sharpen Example

Outline Example

The third example detects image outlines by setting the K array to the values shown below. The image shown in Figure 3 is also converted to greyscale.

-1.0	-1.0	-1.0
-1.0	8.0	-1.0
-1.0	-1.0	1.0

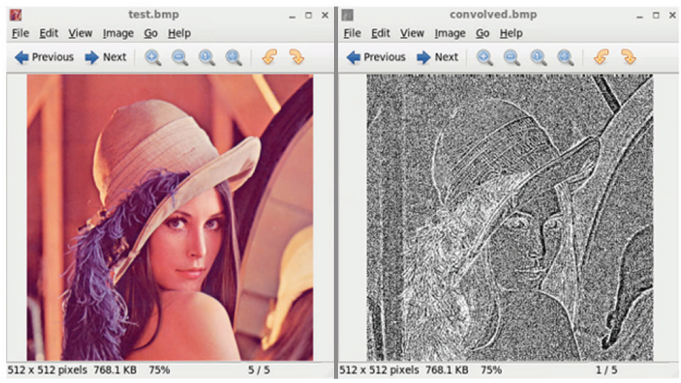

Figure 3. Outline Example

C++ Software Model

In the following code example, we create the C++ software model and testbench using the gcc compiler and perform a 3x3 convolution on an existing 512x512 image.

```
void matrix_conv(mat& x,  ker& k, mat& r)
{
r.m = x.m; //rows
r.n = x.n; //cols
// Non-separable kernel
int dx=(k.km-1)/2;  //kernel middle x
int dy=(k.kn-1)/2;  //kernel middle y
//Constrain convolve window to be fully contained inside image.
// Boundary pixels are not processed, yet are initialized at 0.0f.
    for (int yix = dy ; yix < x.n-dy; yix++) {
    for (int xix = dx; xix < x.m-dx; xix++) {
      float sum = 0.0f;
      for (int kxix = 0; kxix < k.km; kxix++) {
            for (int kyix = 0; kyix < k.km; kyix++) {
                  sum += x(xix-dx+kxix,yix-dy+kyix)*k(kxix,kyix);
                   }
              }
            r(xix,yix) = sum;
          }
}
}
```

Note that the large image is stored in matrix x, and the kernel k stores the coefficient matrix. Note also that the type of values stored in x and k are not defined so this code can easily be converted from floating-point to fixed-point data types.

The software model is compiled and executed with the Intel HLS Compiler using the following command line:

```
i++ bmp_tools.cpp  mat.cpp conv.cpp main.cpp
-march=x86-64 -o test-x86-64
```

The Intel HLS Compiler supports a software-centric use model and will execute the function on the host machine using the command line syntax that is similar with gcc.

Generating Initial Hardware Results

The Intel HLS Compiler generates hardware from the C++ source file using the following command:

```
i++ bmp_tools.cpp mat.cpp conv.cpp main.cpp -v
-march=Arria10 --component conv -o test-fpga
```

The command for generating hardware is similar with the command for software execution with two exceptions. The target architecture (-march) and the top-level component (--component) must be set to target the FPGA. This invokes the Intel HLS Compiler to perform high-level synthesis on the C++ model and generate RTL along with interactive reports. Table 1 shows a summary of the estimated area in Pass 1. Note that you need to run the Intel Quartus® Prime software to get the actual results.

PASS	DESCRIPTION	ALUTS	FFS	RAMS	DSPS	f_{MAX}	LATENCY
1	Initial run from i++	5,333	6,927	512	2	N/A	N/A

Table 1. Initial Hardware Results

In this design, we use a single hard floating-point multiply-add that was created from two DSP blocks to implement the 3x3 convolution. This is the most area-efficient way to implement hardware but comes at the expense of increased latency. The Intel HLS Compiler measures the latency during the RTL verification step.

RTL Verification and Latency Analysis

The FPGA test also automatically sets up and configures a co-simulation of the software and hardware models using the ModelSim*-Intel FPGA software or the RTL simulator. The ModelSim-Intel FPGA software verification of the top-level component is achieved by running the executable (test-fpga) produced during hardware generation. This kicks off a co-simulation run with the ModelSim-Intel FPGA simulator and the C++ software executable. Interfaces are automatically generated, and data produced by the C++ testbench is streamed through the top-level component during the co-simulation of the software model and the synthesized RTL model.

The results of the co-simulation verification include any mismatches in output data, and a measurement of both the input cycles (time from data-in to data-out), and the throughput rate, which if the design is fully pipelined will be close to 1. The latency for this design in number of clock cycles is listed in Pass 2 of Table 2.

PASS	DESCRIPTION	ALUTS	FFS	RAMS	DSPS	f_{MAX}	LATENCY
1	Initial run from i++	5,333	6,927	512	2	N/A	N/A
2	RTL verification latency analysis	5,333	6,927	512	2	N/A	22,891,447

Table 2. RTL Verification and Latency Analysis

Measurement of the data rates starts on the first valid word seen on the interface and ends when the last valid word is sent or received on the interface. The data rate is then calculated as (Number of words sent on the interface) / (Number of cycles required to send the data). A data rate of 1 word / cycle is optimal.

Generating Accurate Reports Using the Intel Quartus Prime Software

The final step in the Intel HLS Compiler design flow is completed by running the hardware using the Intel Quartus Prime software compile from the <design>.prj/quartus directory using the command:

```
quartus_sh --flow compile quartus_compile
```

With these steps done, you can view the generated report to see the final performance and area results listed in Pass 3.

PASS	DESCRIPTION	ALUTS	FFS	RAMS	DSPS	f_{MAX}	LATENCY
1	Initial run from i++	5,333	6,927	512	2	N/A	N/A
2	RTL verification latency analysis	5,333	6,927	512	2	N/A	22,891,447
3	Initial run – Intel Quartus Prime software results	4,592 (ALMs)	7,024	515	2	266 MHz	22,891,447

Table 3. Intel® Quartus Prime Software Results

The report tells us that our design runs at 266 MHz, uses two hardened floating-point units, consumes 515 of the internal RAM blocks, and occupies 4,592 adaptive logic modules (ALMs) of the logic in the target device. After checking that co-simulation has completed properly, we check the image before and after running the design. Assuming the image loads correctly, we can move on to the following steps.

Deployment to FPGA

The Intel HLS Compiler generates an intellectual property (IP) block that can be integrated into a system design in the Intel Quartus Prime software environment to create the complete FPGA design. The user must run the Intel Quartus Prime software to integrate, place and route, and generate a bitstream file. These steps are beyond the scope of this paper.

Optimizing Hardware Results

After initial results are achieved the next step in the Intel HLS Compiler design process is to optimize the results. This is typically achieved either through code modifications or by using pragmas that instruct the tool to generate a different hardware implementation. A summary of the commonly used pragmas supported by the Intel HLS Compiler is shown in the following section.

Parallelism

Remember that you only use two floating-point units in your design. By using the analysis report from the Intel HLS Compiler to look back at your design, you see that they are used to compute all the sums over all the windows that the convolution uses. To parallelize this, you can envision having one floating-point unit for each cell in the kernel of the convolution. If the data is loaded properly into the (2N+1) x (2N+1) kernel array, you can compute the sum needed by doing all the multiplies in one floating-point multiply cycle, followed by summing those values together.

UNROLL PRAGMA	DESCRIPTION
#pragma unroll	This will allow you to unroll the inside kernel loops to get one floating-point unit per kernel cell.

Table 4. Pragma Unroll

Streaming Data

The Streaming pragma is for feeding the kernel window with data as it accesses the image data while reducing the total number of pixels in the image that you need to store at one time. You can envision your image window being accessed in chunks of 2N+1 rows at a time, as that is precisely what you'll need to flow the kernel across that number of rows and compute your filter one row at a time. As you finish computing one buffered row, you can roll in the next buffered row.

STREAMING PRAGMA	DESCRIPTION
ihc::stream_in<unsigned int>	This construct allows you to both stream the image in, one pixel at a time, to control your own buffering of 2N+1 rows of image data that can flow the kernel across, and it allows you to direct the compiler to create a standard streaming interface unit to get data in and out of our hardware component in a standard, efficient way.

Table 5. Streaming Pragma

Iteration Interval

The iteration interval pragma will instruct the Intel HLS Compiler on how many system clock cycles are required to produce a valid output. If II is close to 1, a valid output will be generated on each clock. Loops will be pipelined and parallelism will be inserted into the design until this constraint is met.

ITERATION INTERVAL PRAGMA	DESCRIPTION
#pragma ii N	Forces the loop that this is applied on to have an II of N, where N is greater than 0.

Table 6. Iteration Interval Pragma

The Intel HLS Compiler also inserts pipeline registers and performs algorithmic retiming on the design to meet performance based on the clock f_{MAX} and device settings.

Improving Results Using Pragmas

In this section the iteration interval was set to II = 1 using a pragma and the results were regenerated by the Intel HLS Compiler and summarized in Pass 4 of Table 7:

PASS	DESCRIPTION	ALUTS	FFS	RAMS		f_{MAX}	LATENCY
1	Initial run from i++	5,333		512	2	N/A	N/A
2	RTL verification latency analysis	5,333	6,927	512	2	N/A	22,891,447
3	Initial run – Intel Quartus Prime software results	4,592 (ALMs)	7,024	515	2	266 MHz	22,891,447
4	Optimized (II = 1) – Intel Quartus Prime software results	3,662 (ALMs)	5,773	4	10	302 MHz	263,757

Table 7. Improved Results with Pragmas

The preliminary report shows that we are using 10 DSPs (hardened floating-point units). We have reduced the RAMs from 515 to 4, and the logic by a small amount. By setting II=1 for the full kernel, we see that the loop has been fully unrolled and has significantly increased the overall performance of the design.

The other key metric is the f_{MAX} from the Intel Quartus Prime software run, which give us the maximum frequency the component can operate in the FPGA. The f_{MAX} shown for Pass 4 in Table 7 is 302 MHz.

Conclusion

We have discussed how to use the Intel HLS Compiler toolchain to convert a software application to a FPGA accelerator application. The following steps summarizes the workflow for using the Intel HLS Compiler tools:

1. Start with an existing C++ design. Get it running using the native data types (.e.g. Floating-point arrays), and verify the results with a working C++ testbench.
2. Generate or synthesize the first FPGA target, and cross-verify by running co-simulation of generated hardware model against the software model created above.
3. Identify micro-architectural optimizations by thinking about how to parallelize and pipeline the algorithm, and then use the Intel HLS Compiler data analysis GUI to guide the placement of loop-unrolling, data dependency, and pipelining pragmas. Get the pipelining to where II=1 if possible for maximal throughput.
4. Start converting your design by identifying the part to be accelerated in the FPGA (e.g. top component) by adding in streaming I/O primitives to manage data flow to the accelerated component.
5. Implement the newly accelerated and optimized component into new applications, such as real-time multi filtering of video streams.

Where to get more information

For more information, visit www.altera.com/intel-hls-compiler.

[1] Benchmark is performed using the following hardware and software—Intel HLS Compiler v0.9, ModelSim*-Intel FPGA-SE-64 10.4d, and 2x8-core Intel Xeon® processor ES-2680 at 2.7 GHz, 256 GB RAM.

[†] Tests measure performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit www.intel.com/benchmarks.