On the HLS Design of Bit-Level Operations and Custom Data Types

On the HLS Design of Bit-Level Operations and Custom Data Types

Jose Raul Garcia Ordaz[a] and Dirk Koch[a]
[a]School of Computer Science, The University of Manchester, United Kingdom

Abstract

Modern high-level synthesis (HLS) tools provide software programmers with a path to accelerate complex processing systems by automating the time-consuming task of generating RTL code. Typically, a HLS tool takes a compute-intensive portion of a software application and produces a functionally equivalent hardware unit. However, in particular for bit-level operations, the coding style used to develop the source code corresponding to a compute-intensive kernel hampers the synthesis of high quality results. In this paper, we explore design guidelines for bit-level operations and custom data types that can help software developers to write HDL-friendly C code in order to automatically produce high quality hardware.

1 Introduction

High-level synthesis (HLS) is a relatively recent paradigm that aims to take the design of complex digital systems to a higher-level of abstraction. High-level synthesis provides many advantages over standard RTL code design flows. For example, productivity is increased and design cycles are reduced as no RTL code needs to be manually developed. Additionally, software developers can adopt HLS design flows more easily than traditional hardware development tools. While the HLS paradigm has the aforementioned benefits, it also has some drawbacks.

To perform the same task, an algorithm implemented in software (targeting standard processors) is in most cases fundamentally different than its equivalent hardware implementation (targeting reconfigurable devices such as FPGAs). Translating an existing software algorithm into RTL commonly causes overhead. Consequently, a handcrafted RTL algorithm is normally faster and more efficient than the hardware produced by the HLS tool [10], [5].

To close the gap between handcrafted RTL hardware and HLS-generated hardware, HLS tools normally provide a number of user-defined optimizations. For example, the Legup HLS tool [16], which is based on the LLVM compiler, allows the use of built-in LLVM optimizations. Similarly, the Vivado HLS tool (VHLS) from the Vendor Xilinx also provides a number of optimization pragmas such as loop transformations [11].

While applying the aforementioned optimization pragmas can produce substantial speedups, HLS tools might still be unable to automatically produce hardware circuits that perform as well as hand made RTL circuits. The cause is that for the same application, a software algorithm targeting a conventional CPU is normally implemented as a sequence of machine operations, while an algorithm targeting an FPGA would exploit the parallelism inherent in those devices. Thus, a solution is to develop a software application using a more HLS-friendly style so that the HLS compiler can infer more easily the developer's intent.

With this approach, the HLS-generated code will resemble more closely the quality of handcrafted RTL code. In this paper, we will explore different C coding styles targeting HLS. We will compare the quality of their corresponding RTL code. And based on this experiment, we will derive design guidelines that will help HLS users to develop C code that is friendly to the HLS compiler. We will show how the features provided by an HLS tool, such as specialized hardware libraries, can be exploited to generate high-quality hardware, in particular for sub-word and bit-level applications. The major contributions of this work are:

- We present design guidelines for developing C bit-level applications that can be synthesized into high-quality RTL code.
- We demonstrate the effectiveness of our design guidelines by providing specific bit-level applications case studies.
- We propose a methodology to evaluate the quality of the RTL produced by an HLS tool for bit-level applications.

The rest of this paper is structured as follows: In Section 2, we discuss ways to influence the quality of results (QoR) produced by HLS tools. Next, in Section 3, we explain how C libraries provided by some HLS tools can be leveraged to model hardware. Then in Section 4 we discuss the necessity of accurately modelling bit-level operations in C code to obtain high-quality results. Furthermore, we introduce design guidelines based on our analysis. In Section 5 we present cases studies that illustrate the effectiveness of the here introduced guidelines for bit-level-applications. Finally, in Section 6 we present our conclusions.

2 Influencing HLS Outcome

While significant efforts in academia and industry has lead to the development of HLS tools that are able to produce results of acceptable quality [12] [11], in general, handcrafted RTL applications still perform better than those generated through HLS [1]. The quality of the results produced by HLS tools mainly depends on two factors: 1) the type of user-defined optimizations applied to the C source code by the HLS tool and 2) the coding style and the C structures used to implement the software algorithms.

Modern HLS tools (e.g. Catapult C, Legup and Vivado HLS) feature optimization pragmas which can, for example, accelerate a particularly time-consuming loop at the cost of hardware resources. So far, it is mostly up to the user to apply these optimizations to critical sections of the code until the required performance is obtained. The task of finding the best balance between performance and hardware cost is known as HLS design space exploration (DSE). Research aiming to facilitate or to automate the DSE of HLS design has been conducted [3] [2].

While DSE-guided optimizations can impact the performance of HLS-generated hardware, the coding style used to implement the software algorithm, can greatly influence the quality of the outcome. By using HLS tools to synthesise efficient RTL for well-known benchmarks (SHA, AES, ADPCM, etc.), the authors in [7] identify guidelines to help software programmers to develop HLS-friendly source code.

These guidelines include source code modifications to loops and arrays in order to make them hardware-friendly. The importance of developing source code that helps the HLS to understand the *programmer's intent* is discussed in [5]. In that work the authors state that for a clustering kernel (K-means), a standard software implementation can show a 30× difference in latency compared to HLS-friendly source code. Similar observations are highlighted in [6] and [9].

Control-oriented applications can also benefit from a HLS-friendly coding style as demonstrated in [4]. In that work, the authors use an I2C IP design as a research vehicle and find that a similar performance can be achieved between a HLS-generated IP and the same design generated with HDL. While a HLS-generated circuit consumes 84% more area than the solution generated using a standard RTL design flow, the authors argue that the HLS approach increases productivity by 100%.

The work presented in [8] describes how a HLS tool can be used to produce hardware structures that replicate the outcome of traditional RTL design flows with the help of an unorthodox use of C language constructs. The authors use the design of a hierarchical NoC structure as an illustrative example that compares the outcomes from HLS-generated RTL with a typical RTL design flow. In [10], the authors take a software implementation of the AES application and generate several versions of the same application by performing transformations at the source-level (e.g. removing unsupported C constructs) until a HLS-generated design achieves RTL code of similar quality as a handwritten RTL design. Furthermore, in [17], the authors use a HLS tool to generate hardware accelerators for convolutional neural network (CNN) applications. In this work HLS pseudocode for a convolution unit is presented.

The body of research described above points out the need of providing the HSL tool with hardware-friendly source code to synthesise high quality outcomes for applications from different domains. According to our survey, applications that heavily rely on bit-level operations have received relatively little attention. In this paper, we investigate how software applications that normally make use of complex and time-consuming bit-fiddling operations can be rewritten to generate hardware-friendly C source code that provides results of similar quality as handcrafted RTL code. We show that synthesising C source code that implements standard software bit-level optimization techniques (i.e. using bit-fiddling tricks) produces RTL code of inferior quality than HLS-friendly C source code.

3 HLS C Libraries for Hardware Design

Vivado HLS (VHLS) is a relatively new HLS tool from the FPGA vendor Xilinx [11]. VHLS has roots in AutoPilot [12], and has been developed in an effort to make FPGAs a mainstream technology for developers from the software domain. The Vivado HLS User Guide [11] describes the C language constructs supported by the tool, and importantly, the unsupported ones. Unsuported C constructs include system calls to the operating system (e.g. memory allocation system calls), function pointers, and recursive functions. Similar open-source HLS tools face similar limitations [16].

Furthermore, VHLS provides a mechanism to facilitate the modelling of hardware structures using software. VHLS introduces high-level synthesis C libraries including, amongst others, an Arbitrary Precision Data Types Library, and HLS libraries for math, video, and DSP operations. In this paper, we focus on the VHLS Arbitrary Precision Data Types Library because it *allows the implementation of custom data types of arbitrary bit widths* of up to 1024 bits (e.g. 5-bit, 42-bit, 800-bit variables can be defined).

By using bit-accurate custom data types, hardware waste is avoided and in consequence, high-quality results are produced (e.g. LegUp provides an automated bitwidth minimization compiler pass that allows the generation of custom datapaths [14]).

Moreover, the VHLS Arbitrary Precision Data Types Library also provides a set of associated functions for these custom *–arbitrary precision (AP)–* data types. For example, string conversion, print, binary arithmetic, boolean, shift, and bit-level operations (i.e. bit manipulation, concatenation, bit set/get operations, bit reduction). VHLS

built-in compiler (i.e. APCC) supports the synthesis of custom data types into bit-accurate hardware units. APCC is compatible with GCC which means that standard C data types are also supported.

The VHLS Arbitrary Precision Data Types Library can be used by adding the “ap_cint.h" header file in an application’s source code. While in this paper we use Vivado HLS to perform our experiments, there are other HLS tools that provide users with hardware libraries that allow explicitly declaring synthetizable arbitrary-width variables. Catapult-C and BlueSpec are examples of such HLS tools. Technically, the here presented HLS design guidelines could also be used in combination with those HLS tools.

3.1 VHLS Design Flow

As software developers are unlikely to be familiar with EDA tools for FPGA design, we elaborate on the design flow of VHLS. First, a software developer would take the specification of a certain application (Figure 1-a) and would develop a software implementation corresponding to that specification (Figure 1-b). The developer must avoid C constructs that are unsupported by the HLS tool. Note that a typical software application would normally be implemented as a sequence of operations where standard data types are used to store operands. Next, time-consuming software functions are selected by the user to be synthesized by the HLS tool (Figure 1-c). Then, a report is generated by the HLS tool. This report contains information about the generated RTL code (Figure 1-c), including latency and area consumed by the resulting hardware functions.

As *latency* and *area* are concepts that might be unfamiliar to software developers, it suffices to say that as a rule of thumb, less latency (i.e. the time that it takes a hardware unit to complete its task) is better and that less area (i.e. the amount of FPGA resources consumed by a hardware unit) is better. However, software developers should be aware that producing a functional unit with low latency, usually means that more resources are occupied and conversely, a small hardware unit will usually be slow. For example, by applying a loop optimization pragma (i.e. pipelining, unrolling) to a time-consuming section of the code, the application will become faster, however it will also consume more area.

A more detailed discussion of these design trade-offs can be found in [11]. In some cases, many iterations are needed to obtain RTL code of acceptable quality (Figure 1-d). Finally, the HLS tool can pack the RTL code into an IP block which can be stored into an IP library for posterior use (Figure 1-e).

4 Bit-Level Operations with HLS

Standard processors struggle to perform bit manipulation operations. This is caused because the smallest standard

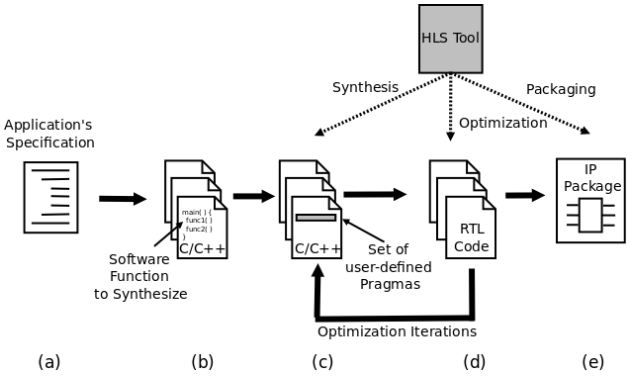

Figure 1 High-Level Synthesis Design Flow.

data type in the C language is the “char" data type which holds a single byte. As a consequence, in order to perform operations at the bit-level, as required by some applications, a software algorithm must complete an often time-consuming sequence of shift, arithmetic, and boolean operations encoded as individual CPU instructions.

Furthermore, memory may be left underutilized in order to save instructions. For example, many C compilers targeting software to be executed under a 64-bit OS use a full 64-bit word to store a boolean value. And arrays of booleans would translate into arrays of 64-bit words. This is obviously not what a hardware solution should do as it can directly work on arbitrary bit vectors.

In order to reduce the execution time of bit-level applications, software developers frequently recode the source code and implement hacks that produce programs that execute in less cycles than standard software implementations [13]. However, these optimizations are tailored for a CPU target and will commonly not result in good hardware implementations as we show in Section 5.

Contrary to processing bit-level operations on a general-purpose CPU, Field Programmable Gate Arrays (FPGAs) are inherently fast at performing bit-level operations. FPGAs can access (virtually for free) individual bits or a range of bits, in order to effortlessly manipulate them.

However, to develop the RTL code that is required to implement the FPGA configuration, some expertise in hardware development and some familiarity in the use of EDA tools is necessary. HLS provides software developers with an opportunity to circumvent this hardware design expertise.

From the experience of using VHLS to produce RTL code for bit-manipulation acceleration units, we have developed a set of design guidelines that help us to produce high-quality results: (1) Understand the characteristics of the algorithm to be implemented. This includes, the used bit-width of inputs and outputs, the current performance of its software implementation, and the most time-consuming sections of the algorithm. (2) Investigate if the algorithm in question has a standard hardware implementation. If such a hardware circuit exists, use the HLS C libraries to develop C source code that accurately models hardware structures.

(3) When the quality of the HLS results is lower than required, recode the C function leveraging the VHLS C libraries such as the Arbitrary Precision Data Types library. By using this library, bit-accurate data types can be defined and bit-level operations can be performed.

The next section presents use cases where bit-level applications are implemented in HLS-friendly coding style. Code snippets, and quality of results for different implementations are presented.

5 Case Studies

In this chapter, we present two cases where the HLS design guidelines presented in Section 4 can be applied in combination with a C hardware library in order to develop hardware-friendly C code which can be synthesized into high-quality RTL code for bit-level applications.

5.1 CRC

The CRC circuit is used extensively by storage and network devices to detect errors in digital data. We select this function to show how the design guidelines presented in Section 4 can impact the level of quality of the RTL code produced by an HLS tool.

For clarity, we start by illustrating the use of a HLS-friendly coding style for a CRC application with a small polynomial. Then, we provide quality numbers (i.e. area and latency) for a CRC-32 circuit using (a) a standard C coding style and (b) a HLS-friendly coding style.

5.1.1 Computing CRC with a Small Polynomial

A handcrafted RTL implementation (in Verilog language) of the CRC circuit is presented in Listing 1. While HLS tools makes it unnecessary for software developers to understand this code, we use it to illustrate the advantages of hardware implementations over software implementations for bit-level applications.

This RTL code consists of a hardware module which produces 4-bit CRC output values from 8-bit inputs. In order to compute the CRC values, we evaluate boolean equations (Listing 1, Lines 12-21) formed by individual input bits and individual polynomial bits ($G = x^3 + x + 1$). By performing bit-level operations, the RTL code presented in Listing 1 generates high-quality results.

In contrast, a software implementation of the same CRC function on a standard processor translates into executing time-consuming sequences of instructions (i.e. load, exclusive or, shift, add, store, etc.). The lack of CRC-specific instructions on general purpose CPUs causes a sub-optimal computation of CRC values.

This is normally mitigated by using different optimizing software techniques to force the compiler to compute CRC values faster [13]. However, even these optimized software implementations are typically slower than hardware implementations.

Listing 1 HDL code to compute CRC values ($G = x^3 + x + 1$).

```
1  module CRC(
2  input [7:0] d_in,
3  input crc_en,
4  output [3:0] crc_out,
5  input rst,
6  input clk);
7
8  reg [3:0] crc_q,crc_c;
9  assign crc_out = crc_q;
10
11 always @(*) begin // Evaluate boolean equations
12 crc_c[0] = crc_q[2] ^ crc_q[3] ^ d_in[0] ^ d_in[1]
13            ^ d_in[2] ^ d_in[6] ^ d_in[7];
14
15 crc_c[1] = crc_q[2] ^ d_in[0] ^ d_in[3] ^ d_in[6];
16
17 crc_c[2] = crc_q[0] ^ crc_q[3] ^ d_in[1] ^ da_in[4]
18            ^ d_in[7];
19
20 crc_c[3] = crc_q[1] ^ crc_q[2] ^ crc_q[3] ^ d_in[0]
21            ^ d_in[1] ^ d_in[5] ^ d_in[6] ^ d_in[7];
22 end
23
24 always @(posedge clk, posedge rst) begin
25 if(rst) begin
26     crc_q <= {4{1'b1}};
27     end
28     else begin
29     crc_q <= crc_en ? crc_c : crc_q;
30     end
31
32 end
33
34 endmodule
```

Listing 2 HLS-friendly C code to compute CRC values ($G = x^3 + x + 1$).

```
1  #include "ap_cint.h"
2
3  unsigned int crc_ap( char m  ) {
4
5  uint1 d_in_0, d_in_1, d_in_2, d_in_3;
6  uint1 d_in_4, d_in_5, d_in_6, d_in_7;
7
8  int i0 = 0, i1 = 0, i2 = 0, i3 = 0;
9  int i4 = 0, i5 = 0, i6 = 0, i7 = 0;
10
11 d_in_0  = apint_get_bit(m, i0);
12 d_in_1  = apint_get_bit(m, i1);
13 d_in_2  = apint_get_bit(m, i2);
14 d_in_3  = apint_get_bit(m, i3);
15 d_in_4  = apint_get_bit(m, i4);
16 d_in_5  = apint_get_bit(m, i5);
17 d_in_6  = apint_get_bit(m, i6);
18 d_in_7  = apint_get_bit(m, i7);
19
20 uint1 crc_q_0 = 1;
21 uint1 crc_q_1 = 1;
22 uint1 crc_q_2 = 1;
23 uint1 crc_q_3 = 1;
24
25 uint1 crc_c_0 = crc_q_2 ^ crc_q_3 ^ d_in_0 ^ d_in_1 \
26                 ^ d_in_2 ^ d_in_6 ^ d_in_7;
27
28 uint1 crc_c_1 = crc_q_2 ^ d_in_0 ^ d_in_3 ^ d_in_6;
29
30 uint1 crc_c_2 = crc_q_0 ^ crc_q_3 ^ d_in_1 ^ d_in_4 \
31                 ^ d_in_7;
32
33 uint1 crc_c_3 = crc_q_1 ^ crc_q_2 ^ crc_q_3 ^ d_in_0 \
34                 ^ d_in_1 ^ d_in_5 ^ d_in_6 ^ d_in_7;
35
36 uint2 crc1_out=apint_concatenate(crc_c_1,lfsr_c_0);
37 uint3 crc2_out=apint_concatenate(crc_c_2,crc1_out);
38 uint4 crc3_out=apint_concatenate(crc_c_3,crc2_out);
39
40 return crc3_out;
41
42 }
```

Alternatively, hardware C libraries provided by some HLS tools (e.g. VHLS, BlueSpec, Catapult-C) can be leveraged to produce high-quality results. Listing 2 shows C code to compute CRC values with the same polynomial as in Listing 1.

Note that, similarly as in the RTL design flow, the hardware circuit is closely modelled. However in this case, hardware-friendly C constructs are used instead of hardware description language (HDL). Note that in Listing 2 the "ap_cint.h" library is included to enable the use of the VHLS built-in bit-manipulation functions. Also note that a `crc_ap( )` function is defined (Listing 2, Line 3). This function takes a standard `char` data type as its input and returns a standard `int` data type as its output.

In this case, the built-in `apint_get_bit( )` function is used to split each of the 8 bits that compose the `m` input bits into individual 1-bit variables. Then those 1-bit variables are bitwise XOR-ed as specified by the polynomial.

Next, the `apint_concatenate( )` function is used to form the final CRC value which is returned as a standard data type. Note that with this approach, a C software function (which takes C standard data types as input), can clearly expresses the developer's intent. This C function can be translated into RTL code that perform bit-level operations on an FPGA.

5.1.2 Exploring Standard and HLS-Friendly C Coding Styles for HLS Compilation

In order to explore the impact that different coding styles have on the quality of hardware produced through HLS compilation, we use the following methodology.

First, we take a basic implementation of the CRC-32 function as shown in Listing 3 (taken from [13]). This implementation consists of a main loop (Listing 3, Line 11) that processes the message input 1 byte at the time. Inside that loop, another loop manipulates the input byte to compute the CRC checksum. Note that this manipulation, comprises mostly *shift and XOR operations* (Listing 3, Lines 20-24). Also note that these operations are performed *at the byte level*, which results on an sub-optimal computation of the CRC checksum.

In order to be able to compare the latency of this basic software implementation with its corresponding hardware implementation generated through HLS, we obtained latency numbers for this software function using the Gem5 simulator. We configured Gem5 to simulate the micro-architectural characteristics of an ARM Cortex-A9 CPU (operating frequency = 650 MHz, out-of-order execution, L1 = 32 kB, L2 = 512 kB). We selected this CPU because it is widely used in the mobile and embedded systems domains.

Then we used Vivado HLS to generate a CRC-32 hardware function from the same basic implementation (Listing 3) and we obtained quality numbers (i.e. area and latency) for this hardware design, as presented in Table 1. According to our experiments, the HLS-generated hardware function synthesized from the basic software implementation is 7.29× faster than its software equivalent.

An inspection of the RTL code synthesized from the basic CRC-32 implementation showed that the HLS tool generated a set of sequential (always) blocks containing *shift and logical XOR operations* which process the input bytes similarly as implemented by the basic software implementation (Listing 3).

Listing 3 C code for the CRC-32 algorithm [13].

```
1  unsigned reverse(unsigned x) {
2  x = ((x & 0x55555555) << 1) | ((x >> 1) & 0x55555555);
3  x = ((x & 0x33333333) << 2) | ((x >> 2) & 0x33333333);
4  x = ((x & 0x0F0F0F0F) << 4) | ((x >> 4) & 0x0F0F0F0F);
5  x = (x << 24) | ((x & 0xFF00) << 8) | \
6      ((x >> 8) & 0xFF00) | (x >> 24);
7
8  return x;
9  }
10
11 unsigned int crc32(unsigned char *message) {
12 int i, j;
13 unsigned int byte, crc;
14
15 i = 0;
16 crc = 0xFFFFFFFF;
17 while (message[i] != 0) {
18     byte = message[i];
19     byte = reverse(byte);
20         for (j = 0; j <= 7; j++) {
21             if ((int)(crc ^ byte) < 0)
22                 crc = (crc << 1) ^ 0x04C11DB7;
23             else crc = crc << 1;
24     byte = byte << 1;
25     }
26 i = i + 1;
27
28 }
29
30 return reverse(~crc);
31
32 }
```

In other words, the HLS compiler was *unable to generate RTL code that resembles a handcrafted RTL implementation* which normally would exploit the ability of FPGAs to perform efficient bit-level operations (e.g. Listing 1).

As the quality of the RTL code produced using a basic CRC-32 implementation was sub-optimal, we investigated whether other CRC-32 implementations can be synthesized into RTL code that is closer to handcrafted RTL quality. We selected a collection of CRC-32 software functions that use standard software techniques aimed at reducing execution cycles [13]. For example, a second version of the CRC-32 implementation uses a coding style that avoids the byte reversal sub-function by performing shifts in the opposite direction and by reversing the polynomial. Moreover, other versions are implemented using coding styles that make use of lookup tables or case statements [13].

Similarly as with the basic CRC-32 implementation, for each one of the selected CRC-32 versions we (1) obtained latency numbers (the binaries, compiled with optimization level O3, were executed on an ARM Cortex-A CPU using Gem5 simulator). (2) We used VHLS to synthesize the CRC-32 source code into RTL code. (3) We compared the latency of the software implementations with their corresponding HLS-generated hardware functions. Table 1 summarizes these latency numbers.

Additionally, the area, expressed in FPGA resources (LUT,

DSP, memory blocks) consumed by each hardware CRC-32 function is presented. After inspecting the RTL code produced by the HLS tool for each of the different CRC-32 versions, we concluded that in each case, the HLS tool had failed to produce RTL code of similar quality as handcrafted RTL code.

Finally, we developed a HLS-friendly C program for CRC-32 which leverages the VHLS hardware C library similarly as described in Listing 2. We obtained quality numbers for this implementation and we compared this result to the ones previously generated. Table 1 summarizes this comparison.

Table 1 Comparison of QoR produced by the VHLS tool for 5 software implementations of the CRC-32 algorithm using different coding styles in C language. Additionally, a hancrafted (h/c) CRC function developed in HDL is also presented.

CRC-32 Function	Latency SW (ns)	Latency HW (ns)	Speedup t_{sw}/t_{hw}	Area LUT, DSP, BRAM
crc_v1 (std)	35189	4825	7.29×	1018, 0, 1
crc_v2	22895	5005	4.57×	770, 0, 1
crc_v3	17708	4842	3.66×	828, 0, 2
crc_v4	22701	3190	7.12×	721, 0, 1
crc_v5 (hw-f)	– [1]	260	68.11×	318, 0, 0
crc_v6 (h/c)	–	165	107.65	121, 0, 0

From that table, it can be noted that the hardware-friendly C code (crc_v5 (hw-f)) is synthesized into a CRC-32 function that is *68× faster than the fastest software implementation* (crc_v3) and *9× faster than the fastest hardware CRC-32 function* generated from a software function using a standard coding style (crc_v1 (std)). Moreover, the hardware-friendly C code generates a CRC-32 function (crc_v5 (hw-f)) that is *3.2× smaller that the fastest CRC-32 function* generated with a standard coding style. Please note that we have not reported software execution results as the code is not designed to run on a conventional CPU.

Additionally, Figure 2 presents a graphical comparison of the quality of the different hardware functions generated using the previously described coding styles for the CRC-32 application. For reference, we also included the quality of a handcrafted CRC-32 unit developed in Verilog HDL (crc_v6 (h/c)).

5.2 Low-Bitwidth CNN

Convolutional neural networks (CNN) have became one of the most successful methodologies for domains such as computer vision and machine learning. CNNs are based on the convolution operation. On a convolution, a filter (i.e. a convolution kernel that is represented as a matrix) is applied to an input (e.g. an image) through a sliding window process. The value generated by the convolution operation is the sum of the product of a vector input value (e.g. image pixels) by the filter matrix.

[1]The hls-friendly CRC-32 application uses the VHLS hardware library and can't be compiled to run on a conventional CPU.

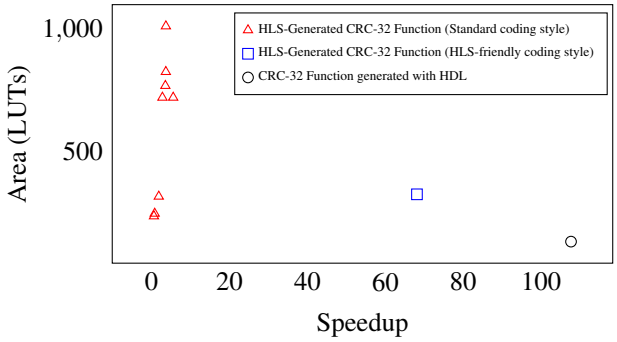

Figure 2 Area vs Speedup values for different hardware implementations of the CRC-32 function.

The high performance provided by CNNs has a substantial cost in terms of computing complexity and power consumption. In this section we will elaborate on the approaches that have been proposed to reduce the computation complexity of CNNs and we will demonstrate how a HLS-friendly C program can be developed to implement kernels that accelerate the convolution function by using bit-level operations.

5.2.1 Low-Bidwidth Convolution Operations

The convolution kernels used by CNNs to filter an input is normally comprised of floating-point values. For this reason, the operations performed on the convolution layer of a CNN consumes most of the execution time of the CNN algorithm.

In order to reduce the execution time (and its associated power consumption) of floating-point-based convolution operations, alternative approaches have been suggested. For instance, in [18] low-bitwidth convolution operations are evaluated. It has been demonstrated that filter weights with low-bitwidths can achieve convolution operations with similar precision as its floating-point counterparts. Weights and operands of arbitrary bitwidths such as 10-bit, 5-bit, and 3-bit have been proposed to perform the convolution operations [19]. The advantage of using low-bitwidth convolution operands is that less computation complexity is required and consequently, less power is demanded. Furthermore, the weight set is reduced which saves latency and power for moving weights.

Conventional CNNs have been implemented on GPUs to exploit the ability of those devices to process large amounts of floating-point operations. However, low-bitwidth CNNs can not be efficiently mapped into general-purpose CPUs or to GPUs because they are bounded to standard data types and bitwidths (i.e. 64-bit, 32-bit, 16-bit, 8-bit boundaries). CPUs and GPUs can't handle sub-byte operations efficiently. In contrast, FPGAs can perform arbitrary bit-level operations very fast.

5.2.2 HLS-Friendly C Code for Low-Bitwidth CNN

In order to show software programmers how to leverage HLS tools to develop low-bitwidth CNNs applications (tar-

geting FPGAs), we present a case study where a multiply-accumulate (MAC) function (heavily used in CNNs) is implemented in a hardware-friendly C language function. To demonstrate the quality of the produced RTL code, we compare it with the RTL code produced by VHLS for an equivalent software function which uses a standard coding style.

The methodology that we followed to perform this comparison is: (1) First we took the software program in Listing 4 and used VHSL to produce RTL code corresponding to that code. Note that this program makes use of standard C constructs only.

Listing 4 Standard C code to compute a dot product kernel.

```
1  unsigned createMask(unsigned high, unsigned low)
2  {
3
4  unsigned r = 0;
5
6  for (unsigned i = low; i <= high; i++)
7          r |= 1 << i;
8          return r;
9  }
10
11 int dot_p(V,W){
12 unsigned int mask_0, mask_1, mask2, mask_3;
13 unsigned int mul_0, mul_1, mul_2, mul_3;
14 unsigned char v_0, v_1, v_2, v_3;
15 unsigned char w_0, w_1, w_2, w_3;
16 unsigned int acc;
17
18 mask0 = createMask( 4,  0);
19 mask1 = createMask(12,  8);
20 mask2 = createMask(20, 16);
21 mask3 = createMask(28, 24);
22
23 v_0 = mask_0 & V;
24 v_1 = mask_1 & V;
25 v_2 = mask_2 & V;
26 v_3 = mask_3 & V;
27
28 w_0 = mask_0 & W;
29 w_1 = mask_1 & W;
30 w_2 = mask_2 & W;
31 w_3 = mask_3 & W;
32
33 mul_0 =  v0 * w0;
34 mul_1 =  v1 * w1;
35 mul_2 =  v2 * w2;
36 mul_3 =  v3 * w3;
37
38 acc = mul0 + mul1 + mul2 + mul3;
39
40 return acc;
41
42 }
```

Table 2 Comparison of the QoR obtained for a dot product kernel using HLS. A dot product function implemented in standard C code (dp_v1 (std) is compared to the same function implemented with a hardware friendly coding style (dp_v2 (hw-f)).

Dot Product Function	Latency SW (ns)	Latency HW (ns)	Speedup t_{sw}/t_{hw}	Area LUT, DSP, BRAM
dp_v1 (std)	895	34	26.37×	45, 8, 0
dp_v2 (hw-f)	–[2]	13	68.19×	20, 4, 0

[2]The hls-friendly dot product application uses the VHLS hardware library and can't be compiled to run on a conventional CPU.

The code defines a `"createMask( )"` subfunction (mostly comprised of shift and OR operations) which is responsible for producing a bitmask that allow us to select specific bit-ranges on an `int` (32-bit variable). Note that this subfunction is required because a direct access to individual bits or set of bits can't be performed on a conventional CPU.

The `"createMask( )"` subfunction is called by a `"dot_p( )"` function to select the 5 least significant bits of each byte in an `int` variable (Listing 4, Lines 18-21). Low-bitwidth input vector elements and filter weights elements are then multiplied and accumulated on a result variable. According to our experiments, the hardware produced by the VHDL tool for this program is 26.37× faster than its software counterpart. The next step of our experiment was to develop a hardware-friendly version of the dot product program.

Listing 5 HLS-friendly C code to compute a dot product kernel.

```
1  #include "ap_cint.h"
2
3  int dot_p_ap(V, W)
4  {
5
6  uint32 v_tmp = V;
7  uint32 w_tmp = W;
8
9  uint5 v_0, v_1, v_2, v_3;;
10 uint5 w_0, w_1, w_2, w_3;
11 uint10 mul_0, mul_1, mul_2, mul_3;
12 uint10 acc;
13
14 v_0 = apint_get_range(v_tmp, 4,  0);
15 v_1 = apint_get_range(v_tmp, 12,  8);
16 v_2 = apint_get_range(v_tmp, 20, 16);
17 v_3 = apint_get_range(v_tmp, 28, 24);
18
19 w_0 = apint_get_range(w_tmp,  4,  0);
20 w_1 = apint_get_range(w_tmp, 12,  8);
21 w_2 = apint_get_range(w_tmp, 20, 16);
22 w_3 = apint_get_range(w_tmp, 28, 24);
23
24 mul_0 = v_0 * w_0;
25 mul_1 = v_1 * w_1;
26 mul_2 = v_2 * w_2;
27 mul_3 = v_3 * w_3;
28
29 acc = mul_0 + mul_1 + mul_2 + mul_3;
30
31 return acc;
32
33 }
```

The C program in Listing 5 leverages the bit-level functions provided in the VHLS C hardware library. Note that instead of creating masks, the program uses the `"apint_get_range( )"` built-in function which allow us to directly access arbitrary bit ranges on a n-bit variable. In this case, 5-bit input operands and 5-bit weight values are multiplied and accumulated to obtain the result. According to our experiments, the hardware produced by the HLS tool for this hardware-friendly code, is 68.19× faster than its software equivalent as presented in Table 2.

Consequently, the RTL code produced by the HLS tool from a software application using a hardware-friendly coding style is 2.6× faster than the RTL code produced from a software application that uses standard C constructs only.

Furthermore, as the numbers in Table 2 show, the area

consumed by the HLS-generated hardware corresponding to the hardware-friendly C application is 2.25× and 2× smaller for LUTs and DSPs, respectively. Which makes the former design more area-efficient. Finally, please note that the CNN is just an example and that the same code could basically be used for computing dot product as used in traditional multilayer perceptrons.

6 Conclusion

In this paper we described how the limitations presented by conventional CPUs to perform bit-level operations can be circumvented by leveraging the hardware libraries provided by HLS tools such as Vivado HLS. We demonstrated the benefits of using a hardware-friendly coding style to develop bit-level operations. As case studies, we examined a CRC-32 function and a Dot Product function.

For the CRC-32 function, we evaluated 5 different coding styles, 4 of them using standard C constructs and 1 using a hardware-friendly coding style. We demonstrated that the former is 68× faster than the fastest software implementation and 9× faster than the fastest hardware generated by the HLS tool from a software application that uses standard coding style.

For the Dot Product function, we argued that applications can benefit from the ability of HLS tools to define and use custom data types. We showed that by using custom data types, smaller, faster hardware can be generated by the HLS tool.

In summary, we showed that a hardware-friendly version of the dot product program can generate hardware that is 2.25× smaller and 2.6× faster than the hardware produced using an equivalent application that uses standard C data types only.

7 Acknowledgment

This work is kindly supported by the Mexican National Council for Science and Technology (CONACyT) under grant 381920.

8 Literature

[1] J. Oppermann and A. Koch. “Detecting Kernels Suitable for C-Based High-Level Hardware Synthesis". *UIC / ATC / ScalCom / CBDCom / IoP / SmartWorld Conference*, 2016.

[2] Q. Zhu and M. Tatsuoka. “High Quality IP Design using High-Level Synthesis Design Flow". *IEEE Design Automation Conference*, 2016.

[3] D. Liu and B. Carrion Schafer. “Efficient and Reliable High-Level Synthesis Design Space Explorer for FPGAs". *IEEE FPL*, 2016.

[4] S. Lahti et al. “Designing a Clock Cycle Accurate Application with High-Level Synthesis". *IEEE Industrial Electronics Society*, 2016.

[5] F. Winterstein et al. “High-Level Synthesis of Dynamic data Structures: A Case Study Using Vivado HLS". *IEEE FPT*, 2013.

[6] F. Winterstein et al. “Separation Logic-Assisted Code Transformations for Efficient High-Level Synthesis". *IEEE FCCM* , 2014.

[7] Y. Liang et al. “High-level Synthesis: Productivity, Performance, and Software Constraints". *ACM Journal of Electrical and Computer Engineering, Special issue on ESL Design Methodology*, 2012.

[8] Z. Zao and James C. Hoe. “Using Vivado-HLS for Structural Design: a NoC Case Study". *ACM FiPGA*, 2017.

[9] J. S. Monson and Brad L. Hutchings. “Using Source-Level Transformations to Improve High-Level Synthesis Debug and Validation on FPGAs". *ACM Field-Programmable Gate Arrays*, 2015.

[10] E. Homsirikamol and Kris Gaj. “Can High-Level Synthesis Compete Against a Hand-Written Code in the Cryptographic Domain? A Case Study". *IEEE ReConFigurable Computing and FPGAs*, 2014.

[11] Xilinx. “Vivado Design Suite User Guide, (UG902)". Online: https://www.xilinx.com.

[12] J. Cong et al. “High-Level Synthesis for FPGAs: From Prototyping to Deployment". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2011.

[13] H. S. Warren. “Hacker’s Delight, Second Edition". *Addison-Wesley*, 2013.

[14] D. Koch et al. “FPGAs for Software Programmers". *Springer*, 2016.

[15] Y. Hilewitz and R. B. Lee. “Fast Bit Gather, Bit Scatter and Bit Permutation Instructions for Commodity Microprocessors". *Journal of Signal Processing Systems*, 2008.

[16] A. Canis et al. “LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems". *ACM FPGA*, 2011.

[17] R. Zhao et al. “Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs". *ACM FPGA*, 2017.

[18] M. Courbariaux et al. “Training Deep Neural Networks with Low Precision Multiplications". *arXiv preprint arXiv:1412.7024*, 2014.

[19] S. Han et al. “Deep compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding". *arXiv preprint arXiv:1510.00149*, 2015.

[20] ARM. “ARM Architecture Reference Manual, ARMv7-A and ARMv7-R". Online: www.arm.com.