Hardware Benchmarking of Cryptographic Algorithms Using High-Level Synthesis Tools: The SHA-3 Contest Case Study

Ekawat Homsirikamol and Kris Gaj

Volgenau School of Engineering
George Mason University
Fairfax, Virginia 22030
{ehomsiri, kgaj}@gmu.edu

Abstract. The growing number of candidates competing in the cryptographic contests, such as SHA-3, makes the hardware performance evaluation extremely time consuming, tedious, and imprecise, especially in the early stages of the competitions. The main difficulties include the long time necessary to develop and verify HDL (hardware description language) codes of all candidates, and the need of developing (or at least tweaking) codes for multiple variants and architectures of each algorithm. High-level synthesis (HLS), based on the newly developed Xilinx Vivado HLS tool, offers a potential solution to the aforementioned problems. In order to verify a potential validity of this approach, we have applied our proposed methodology to the comparison of five Round 3 SHA-3 candidates. Our study has demonstrated that despite a noticeable performance penalty, caused by the use of high-level synthesis tools vs. manual design, the ranking of the evaluated candidates, in terms of four major performance metrics, frequency, throughput, area, and throughput to area ratio, has remained unchanged for Altera Stratix IV FPGAs.

Keywords: FPGA, High-level synthesis, Secure Hash Algorithm, SHA-3, cryptography, benchmarking, evaluation, hardware, case study

1 Introduction & Motivation

The first open competition for a new cryptographic standard was announced by the National Institute of Standards and Technology (NIST) in 1997 [1]. The Advanced Encryption Standard (AES) [2], currently one of the most well-known and widely-used symmetric-key block ciphers, is the result of this competition. Since then, there has been a proliferation of open competitions with different cryptographic transformations targeted in each of them. One of the most important of them was the contest for the new cryptographic hash function standard, SHA-3. This contest attracted over 50 candidates, and required about four years for their evaluation.

The cryptographic competitions have become the norm for the development of new cryptographic standards for several reasons. First, the open nature of a competition and its submitted algorithms provides assurance for future users that any potential backdoor (i.e., an ability to decrypt a message without the knowledge of the key used for its

encryption) is unlikely to exist. Secondly, the winner of such competition is guaranteed to have a strong security with good performance in software and hardware. Additionally, as cryptography is a broad field, studied by a limited number of scientists, a public competition enables focusing the attention of the entire cryptographic community on a specific area of cryptography, such as design of hash functions. As a result of an active involvement of the entire community, including leading experts in the specific area, the winning algorithm is likely to get endorsed and accepted as a new standard much faster than any algorithm developed by national intelligence organizations, such as the U.S. National Security Agency (NSA). While security has been commonly recognized as the

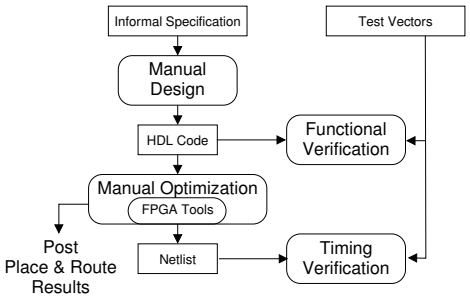


Fig. 1: Traditional Hardware Development and Benchmarking Flow

most important evaluation criterion, it is also a measure that is most difficult to evaluate and quantify, especially during a relatively short period of time reserved for the majority of contests. The typical outcome is that, after eliminating a fraction of candidates based on security flaws, a significant number of remaining candidates must be ranked based on their performance in hardware and software. In the past competitions, the differences among the competing algorithms in terms of their performance in hardware seemed to be particularly large, and often served as a tiebreaker when other criteria failed to identify a clear winner.

A traditional hardware benchmarking flow, shown in Fig. 1, typically starts with the translation of an informal specification to a hardware description language (HDL) code, e.g., the code written in VHDL or Verilog. Then, functional verification is performed based on test vectors generated using a reference software implementation. The post-place & route results are generated either using default options of FPGA tools, or tool options are optimized manually. Finally, the obtained netlist undergoes a timing verification for the last check up. This process has been further improved with the development of the Automated Tool for Hardware EvaluatioN (ATHENa) [3], allowing an automated optimization of FPGA tool options. This tool is application agnostic and works across supported FPGA vendors of Xilinx and Altera.

Nevertheless, hardware development for the purpose of benchmarking remains difficult due to various factors. These factors include a large number of candidates, large amount of time required to develop and verify HDL sources, multiple variants of algorithms, multiple hardware architectures targeting different needs, and dependence on skills of the designers.

A potential solution to the majority of aforementioned problems is High-Level Synthesis (HLS). HLS allows designers to use a high-level programming language, such as

C, C++, or Java, to generate synthesizable HDL. Until recently, this approach has been impractical due to the inadequate quality and high cost of HLS tools [4]. This situation has changed when Xilinx acquired AutoESL Design Technologies Inc. in 2011, and incorporated its HLS tool, AutoPilot, into its latest toolchain, Vivado, in 2012 [5]. The availability of the industrial-quality, low-cost tool has allowed the HLS-based design approach to become more realistic.

Previous studies have demonstrated that HLS can reduce the development time, while maintaining good performance as compared to software [6–8]. It has also been shown that a design using HLS-based approach can compete against a hand-written Register Transfer Level (RTL) code [9–13], at least in selected domains.

Apart from the use of HLS to benchmark candidate algorithms during cryptographic contests, the use of HLS could also provide an earlier feedback for designers of cryptographic algorithms. Traditionally, a design of a cryptographic algorithm involves only security analysis and software benchmarking. This situation has created several unpleasant surprises when the resulting algorithms performed poorly in hardware, which was the case for Mars in the AES contest, as well as BMW, ECHO and SIMD in the SHA-3 Contest.

As a result, this study aims to analyze and test the following hypothesis:

Ranking of candidate algorithms in cryptographic contests in terms of their performance in modern FPGAs will remain the same independently whether the HDL implementations are developed manually or generated automatically using High-Level Synthesis tools.

2 Methodology

The development and benchmarking process used for the HLS-based design approach is shown in Figure 2. In this process, a designer modifies a reference software implementation, which is typically provided by the algorithm's designers in C. These modifications typically involve the addition of HLS tool directives, provided as pragmas, and small tweaks in the reference implementations that make them more suitable for HLS. Once these modifications are completed, the code is verified in software for the correct functionality. Afterward, the HLS-ready C code is processed by the HLS tool to generate an RTL HDL code. This code is further processed using the design methodology described in Section 1. In case, the obtained results are worse than expected, e.g., in terms of the number of clock cycles required to process a single input block, the HLS-ready C code must be further tweaked, and possibly extended with additional pragmas.

In order to verify our hypotheses, the five finalists of the SHA-3 competition were selected as our test case. The most efficient (in terms of the throughput to area ratio) high-speed architectures were selected for each algorithm [14–16]. These architectures included: two times horizontally folded architecture of BLAKE (\2h), four times unrolled architecture of Skein (x4), and the basic iterative architectures for Groestl, JH and Keccak (x1).

To provide a fair and reliable reference for comparison, the designs developed manually, using the RTL-based approach, based on the studies published in [14, 17], have

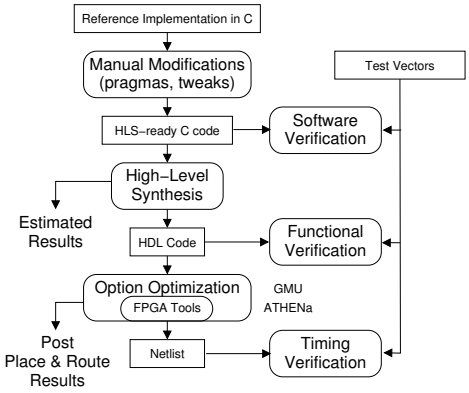


Fig. 2: HLS-Based Development and Benchmarking Flow

been benchmarked using the same FPGA families. These designs were selected for the following reasons:

- **Source codes** of the reference designs in RTL VHDL are easily accessible to the public [18]. This availability allows everybody to easily replicate the obtained results.
- **Detailed diagrams** of each design are also available in public domain [18]. The availability of these diagrams allows other designers to easily understand each specific hardware architecture, without analyzing the source codes, and thus saving valuable time.
- **Uniform and realistic interface**. The aforementioned studies utilized a standard FIFO-based interface, which can be easily adapted to support any bus-based interface for a system-on-chip, e.g., the AXI-4 Stream interface.
- **Standalone**. The reference designs are completely self-sufficient. They require the minimum number of external control signals.

The reference C codes for the HLS-based approach are based on the submission packages available from the SHA-3 contest website [20]. Each reference C implementation is then manually modified to create an optimum HLS-ready C code, and verified in software. This C code is then passed as an input to Vivado HLS, and the corresponding VHDL code is automatically generated. This VHDL code is then simulated for functional correctness, and the number of clock cycles required to process a block of data is determined. If the results are incorrect, or the number of clock cycles too high, the HLS-ready code needs to be modified, and the entire process repeated. If the HDL code performs as expected, this code is benchmarked using ATHENa, and the final netlist verified using timing simulation.

In order to minimize any discrepancies between the rankings obtained using HLS and RTL-based approaches, we apply the same tools and optimization techniques to both manually written and automatically generated HDL codes. The synthesis of HDL codes is accomplished using Vivado HLS v2014.1, with VHDL as a target language. Logic synthesis and implementation (mapping, placing, and routing) are performed

by ISE Design Suite v14.7 and Quartus II v13.0sp1, for Xilinx and Altera FPGAs, respectively. The gathering and uniform optimization of results were facilitated by ATHENa [3].

For the ease of comparison between the rankings, no dedicated FPGA resources, such as Block RAMs or DSP units are used. Our results are generated using two modern FPGA families: Stratix IV from Altera and Virtex 6 from Xilinx.

Finally, it must be noted that HDL code generated from Vivado HLS is generally vendor independent. So far, we have found that as long as we do avoid the use of dual-port memory, the tool is able to generate a design that is compatible to Altera tools as well.

3 Design

3.1 Top-Level Interconnect

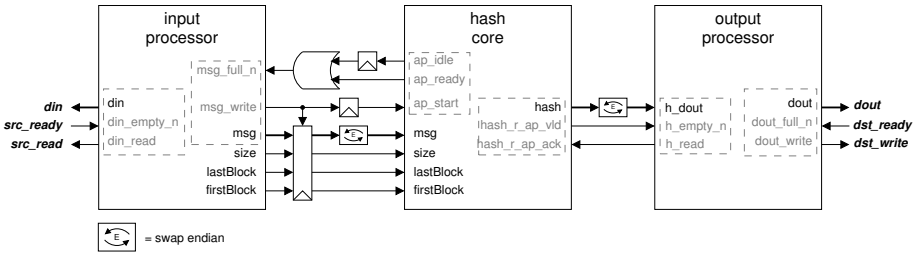


Fig. 3: Top-level diagram.

The interface and the communication protocol used in this study are based on the designs proposed in [17]. As we are operating at full speed, the input and output units must operate at the same time when a cryptographic core is hashing. The current HLS tool is not yet capable of generating a design that can perform these tasks automatically. As a result, these modules need to be connected manually at the top-level. The top-level diagram of all crypto cores is shown in Fig. 3. Each crypto core is comprised of three primary modules, *input processor*, *hash core* and *output processor*. Each module is generated using HLS tools independently, using a separate HLS-ready C code, in order to allow them to operate concurrently. The rectangular regions within each module represent groups of signals that are generated by the HLS tool from the same input. For instance, *din* port of *input processor* generates *din_empty_n* and *din_read* signals. These generated signals are the product of the INTERFACE pragma.

Communications between modules in critical area are registered to improve timing. Endianness of incoming and outgoing data for the hash core are swapped for correct operation. This is because when ARRAY_RESHAPE pragma is used to create a single dimensional array from a multi-dimensional array, a data block is formed based on the word size of the original array in the little endian format. For instance, forming a 128-bit

data block from an integer array of size four (int A[4]) would have the following order: A[3] A[2] A[1] A[0]. As a result, when input data arrives in the big-endian format, endianness must be adjusted accordingly.

3.2 Design Techniques

Design techniques play an important role in inferring our desired hardware architecture. These techniques are summarized below:

– **Match the code to the architecture**. In order to ensure that the synthesis tool can infer the desired hardware architecture from the reference C code, it is important to organize and modify the reference C code to match the target architecture as much as possible. An example is shown in Listing 1.1 and 1.2. In these example, the diagonal step is replaced by a second run through the column step, after the necessary permutation of the internal state words, as our target architecture (folded horizontally by a factor of two) contains only a half-round of BLAKE.

Listing 1.1: Original code before modification.

```
// (a) Before modification
  /*  do 14 rounds */
  for(round=0; round<NB_ROUNDS32; ++
       round)
  {
    /* column step */
    G32( 0, 4, 8,12, 0);
    G32( 1, 5, 9,13, 1);
    G32( 2, 6,10,14, 2);
    G32( 3, 7,11,15, 3);

    /* diagonal step */
    G32( 0, 5,10,15, 4);
    G32( 1, 6,11,12, 5);
    G32( 2, 7, 8,13, 6);
    G32( 3, 4, 9,14, 7);
  }
```

Listing 1.2: HLS-compatible code after modification.

```
// (b) After modification
  /*  do 14 rounds */
  for(round=0; round<NB_ROUNDS32*2;
       ++round)
  {
    /* column step */
    G32( 0, 4, 8,12, 0);
    G32( 1, 5, 9,13, 1);
    G32( 2, 6,10,14, 2);
    G32( 3, 7,11,15, 3);

    // Permutation
    if (round % 2 == 0)
      for(i=0; i<4; i++)
        for(j=0; j<4; j++)
          tmp[i*4+j] = v[((j+i)%4)+i
               *4];
    else
      for(i=0; i<4; i++)
        for(j=0; j<4; j++)
          tmp[i*4+j] = v[((j+4-i)%4)+
               i*4];
```

– **No timing constraints**. The HLS tool can optimize the HDL code to meet any specific timing constraints. Unfortunately, the desired hardware architecture is not guaranteed. In fact, if a design is overly constrained, it is more likely to generate a sub-optimal hardware architecture. For the purpose of benchmarking, this can be very counterproductive. As a result, it is often better to remove any timing constraints, in order to avoid an automatic optimization of the architecture by the HLS tool.

Listing 1.3: Original code before modification.

```
// (a) Before modification
  for(round=0; round<NB_ROUNDS; ++
       round)
  {
    if (round == NB_ROUNDS-1)
      single_round(state, 1);
    else
      single_round(state, 0);
  }
```

Listing 1.4: Code after modification for reduction of resource usage.

```
// (b) After modification
  for(round=0; round<NB_ROUNDS; ++
       round)
  {
    if (round == NB_ROUNDS-1)
      x = 1;
    else
      x = 0;
    single_round(state, x);
  }
```

– **Keep design small by reusing functions**. Each function call in the HLS-ready code generally translates to a new block of hardware. This feature can increase the design size significantly. While this effect is straightforward to understand, it can be counterintuitive to utilize in software. An example of such situation is shown in Listing 1.3 and 1.4. In this example, a *single_round()* function accepts either *1* or *0* as its second input. In software, the two approaches would have an almost identical performance and memory usage. However, in the hardware generated by HLS, the design inferred by (a) requires twice as many resources as the design inferred by (b). This is because the *single_round()* function is called twice in (a), while it is being called only once in (b).

PRAGMAs used by the aforementioned techniques are described in [13]. All our HLS-ready source codes are available at [18].

4 Results and Discussion

4.1 Ranking of Results

The results of all our implementations for the RTL and HLS-based design flows are summarized in Fig. 4 and Fig. 5. Each algorithm is represented using a different shape marker, with lines across the markers helping to compare the throughput to area ratios. The higher the gradient of the line, the more efficient the corresponding algorithm is. All diagrams demonstrate a very good correlation between the HLS and RTL results in case of Altera Stratix IV FPGAs, and a moderate correlation in case of Xilinx Virtex 6 FPGAs. The details of all results and the corresponding ratios are listed in Table 1.

In Table 2, all Round 3 SHA-3 candidates are ranked according to four major performance measures: frequency, throughput, area, and throughput to area ratio. A reference ranking based on the RTL approach is followed by the new ranking based on the HLS approach. Underneath each of the first four candidates is the distance in percents to the next candidate, denoted as Δ**(i+1)**, i.e., a relative difference between the candidate at position i in ranking and the algorithm at position i+1. The algorithms that have their positions changed in the HLS-based approach are marked in bold, with upward and downward arrows signifying the direction of change.

This table clearly demonstrates that for Stratix IV, all rankings remain perfectly the same. For Virtex 6, the rankings remain the same in case of frequency and throughput. In case of area, BLAKE moves from the position number 4 to the position number 2. In

Table 1: Results for the HLS and RTL-based approaches. Notation: Freq.: clock frequency, A: area in ALUTs for Altera Stratix IV and CLB slices for Xilinx Virtex 6, TP: Throughput in Mbits/s, TP/A: throughput over area ratio, RTL/HLS: ratio of the RTL result to the corresponding HLS result.

Altera Stratix IV												
	HLS				**RTL**				**RTL/HLS**			
	Freq.	**TP**	**A**	**TP/A**	**Freq.**	**TP**	**A**	**TP/A**	**Freq.**	**TP**	**A**	**TP/A**
BLAKE	119.6	2040	4557	0.45	132.3	2337	3543	0.66	1.11	1.15	0.78	1.47
Groestl	218.8	4871	7290	0.67	236.9	5776	7404	0.78	1.08	1.19	1.02	1.17
JH	336.8	3919	3256	1.20	399.7	4759	3210	1.48	1.19	1.21	0.99	1.23
Keccak	271.4	11356	4156	2.73	317.7	14401	3541	4.07	1.17	1.27	0.85	1.49
Skein	97.9	2387	5752	0.41	96.2	2592	3936	0.66	0.98	1.09	0.68	1.59
Xilinx Virtex 6												
	HLS				**RTL**				**RTL/HLS**			
	Freq.	**TP**	**A**	**TP/A**	**Freq.**	**TP**	**A**	**TP/A**	**Freq.**	**TP**	**A**	**TP/A**
BLAKE	150.6	2570	1289	1.99	126.1	2226	1257	1.77	0.84	0.87	0.98	0.89
Groestl	242.7	5403	2016	2.68	296.1	7220	1870	3.86	1.22	1.34	0.93	1.44
JH	291.8	3395	1141	2.98	454.6	5412	849	6.37	1.56	1.59	0.74	2.14
Keccak	211.2	8838	1494	5.92	261.2	11839	1086	10.90	1.24	1.34	0.73	1.84
Skein	107.3	2616	1426	1.83	125.2	3373	1005	3.36	1.17	1.29	0.70	1.83

Table 2: Algorithm rankings based on four major performance metrics. $\Delta(i+1)$ represents the relative difference in the measured metric between the algorithm at the current ranking position i and the algorithm at the subsequent position $i+1$.

Approach	**Frequency rankings**					**Area rankings**				
	1	2	3	4	5	1	2	3	4	5
Altera Stratix IV										
RTL	JH	Keccak	Groestl	BLAKE	Skein	JH	Keccak	BLAKE	Skein	Groestl
Δ(i+1)	26%	34%	79%	38%		-9%	-0%	-10%	-47%	
HLS	JH	Keccak	Groestl	BLAKE	Skein	JH	Keccak	BLAKE	Skein	Groestl
Δ(i+1)	24%	24%	83%	22%		-22%	-9%	-21%	-21%	
Xilinx Virtex 6										
RTL	JH	Groestl	Keccak	BLAKE	Skein	JH	Skein	Keccak	BLAKE	Groestl
Δ(i+1)	54%	13%	107%	1%		-16%	**-7%**	**-14%**	-33%	
HLS	JH	Groestl	Keccak	BLAKE	Skein	JH	**BLAKE**$\uparrow$	**Skein**$\downarrow$	**Keccak**$\downarrow$	Groestl
Δ(i+1)	20%	15%	40%	40%		-11%	-10%	-5%	-26%	

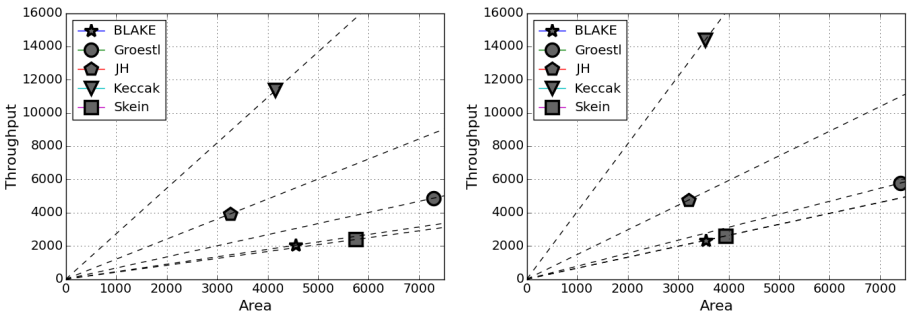
Approach	**Throughput rankings**					**Throughput/Area rankings**				
Altera Stratix IV										
RTL	Keccak	Groestl	JH	Skein	BLAKE	Keccak	JH	Groestl	BLAKE	Skein
Δ(i+1)	149%	21%	84%	11%		175%	90%	18%	0%	
HLS	Keccak	Groestl	JH	Skein	BLAKE	Keccak	JH	Groestl	BLAKE	Skein
Δ(i+1)	133%	24%	64%	17%		127%	80%	49%	8%	
Xilinx Virtex 6										
RTL	Keccak	Groestl	JH	Skein	BLAKE	Keccak	JH	Groestl	Skein	BLAKE
Δ(i+1)	64%	33%	60%	52%		71%	65%	15%	**90%**	
HLS	Keccak	Groestl	JH	Skein	BLAKE	Keccak	JH	Groestl	**BLAKE**$\uparrow$	**Skein**$\downarrow$
Δ(i+1)	64%	59%	30%	2%		99%	11%	34%	9%	

case of the throughput to area ratio, Skein and BLAKE swap places at positions 4 and 5.

The first of these changes can be explained by looking at the RTL/HLS ratios for area in Table 1. For Keccak and Skein, the HLS implementation trails the RTL implementation by about 30%. At the same time, for BLAKE, the area is almost identical in case of both design flows. Interestingly, in case of Altera, the ratios of RTL to HLS results are much better balanced for the aforementioned three candidates (Skein: 0.68, BLAKE: 0.78, Keccak: 0.85).

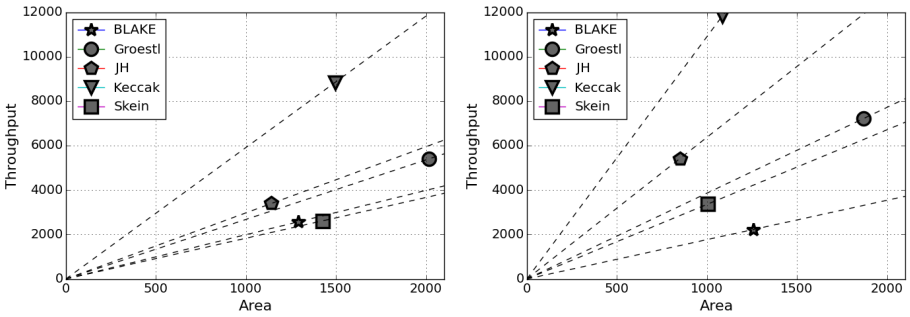
In case of ranking in terms of throughput to area ratio, the swap of places between Skein and BLAKE can be explained by the fact that unexpectedly, the HLS implementation of BLAKE outperforms the RTL implementation in terms of throughput, and is almost identical in terms of area. At the same time, the HLS implementation of Skein trails the corresponding RTL implementation by about 30% for both throughput and area. Interestingly, nothing similar happens for the same VHDL source codes in case of Altera Stratix IV. Thus, the only reasonable explanation seems to be a different operation of Xilinx and Altera tools for logic synthesis, mapping, placing, and routing, which favors some combinations of algorithms and VHDL coding styles (RTL vs. HLS-generated), and is disadvantageous for the others.

In Table 3 we summarize the spread of ratios, RTL/HLS, across investigated algorithms, for all performance metrics and the two investigated FPGA families. In case of frequency, for Altera FPGAs, the ratio varies between 0.98 to 1.19, the spread of only about 20%, which is not likely to affect the ranking of the candidates. The same spread of ratios for Xilinx Virtex 6 exceeds 70%, which is much more significant. For area, both spreads are similar, in the range of 30% for both Stratix IV and Virtex 6. Still, a particular combination of algorithms is causing the rankings of algorithms to remain the same in case of Stratix IV, and change substantially in terms of Virtex 6. In case of throughput to area ratio, the difference in the spread of ratios is much more significant (about 40% for Startix IV and more than 100% in case of Virtex 6).



(a) HLS (b) RTL

Fig. 4: Manual RTL vs. HLS-based Results for Altera Stratix IV



(a) HLS (b) RTL

Fig. 5: Manual RTL vs. HLS-based Results for Xilinx Virtex 6

Table 3: RTL/HLS performance ratios across investigated algorithms

FPGA Family	Freq.	Throughput	Area	TP/A
Altera Stratix IV	0.98-1.19	1.09-1.27	0.68-1.02	1.17-1.59
Xilinx Virtex 6	0.84-1.56	0.87-1.59	0.70-0.98	0.89-2.14

4.2 Lessons Learned

The correct inference of architecture requires careful consideration for almost each line of the HLS-ready C code. A simple mistake can lead to vastly different outcomes. As a result, the programmer needs to repeatedly check the estimated results, using the design cycle provided by the HLS tool, to ascertain whether the output circuit has no additional logic. Additionally, clock frequency can be also affected as a result of this issue as well.

Latency, on the other hand, is largely dependent on the control unit generated by the HLS tool. Developing this unit in the HLS-based approach is relatively straightforward compared to the manual RTL approach, as the control unit is generated automatically, based on the code in C. In the manual hardware design, the creation of the control logic is often the primary bottleneck in terms of the design time. The ability of HLS tools to automatically generate and verify the schedule of the circuits can save a huge amount of time.

Nonetheless, the generated control unit tends to be sub-optimal. This is because the generated unit is not capable of supporting an overlap between the completion of the last round of encryption/decryption and reading the next input block. Moreover, one additional clock cycle is required to initiate processing of the subsequent data block. As a result, the throughput of the HLS-based design tends to be lower than the throughput of the manual one, even if they both operate at the same frequency. The throughput formulas for the RTL and HLS-based approach are given by equations 1 and 2, respectively.

$$Throughput_{RTL} = Block\_size/(\#Rounds * T_{clk}) \quad (1)$$

$$Throughput_{HLS} = Block\_size/((\#Rounds + 2) * T_{clk}) \quad (2)$$

5 Conclusions

High-level synthesis offers a potential to allow hardware benchmarking in early stages of cryptographic contests. It can also be an effective method for gauging the hardware performance of early variants of a cryptographic algorithm during the design process by groups of cryptographers with limited experience in hardware design.

Our case study based on the five final SHA-3 candidates has demonstrated the correct ranking for Altera Stratix IV FPGAs in terms of all major performance measures: frequency, throughput, area, and the throughput to area ratio. For Xilinx Virtex 6 FPGAs, the rankings matched perfectly only in case of two out of four performance measures.

Thus, at this point, we recommend only the use of Altera FPGAs for HLS-based candidate benchmarking. Interestingly, the recommended development flow combines the use of HLS tools from Xilinx (Vivado HLS), lower-level FPGA tools from Altera (Quartus II), and open-source script-based tools for option optimization (ATHENa).

In order for the HLS-based design approach to be an effective replacement for the manual design approach, there are a few obstacles that need to be still overcome. These obstacles include for example the limited correlation between the manual RTL designs and the HLS-based designs for Xilinx FPGAs, and the generation of sub-optimal control units by Vivado HLS tools. Additionally, the preparation of the HLS-ready C code needs to be simplified, for example by an automatic preprocessing of the generic C codes developed without high-level synthesis in mind.

Acknowledgement. This work is supported by the US National Science Foundation (NSF) under grant number #1314540.

References

1. National Institute of Standards and Technology. (2000, Oct.) Report on the Development of the Advanced Encryption Standard (AES). [Online]. Available: http://csrc.nist.gov/archive/aes/round2/r2report.pdf
2. National Institute of Standards and Technology, *FIPS PUB 197: Advanced Encryption Standard (AES)*, Nov 2001. [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
3. K. Gaj, J. Kaps, V. Amirineni, M. Rogawski, E. Homsirikamol, and B. Brewster, "ATHENa - Automated Tool for Hardware EvaluatioN: Toward Fair and Comprehensive Benchmarking of Cryptographic Hardware Using FPGAs," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, Aug 2010, pp. 414–421.
4. G. Martin and G. Smith, "High-Level Synthesis: Past, Present, and Future," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 18–25, Jul. 2009.
5. C. Maxfield. (2012, Jul) First public access release to Xilinx Vivado design suite. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1317376
6. K. Rupnow, Y. Liang, Y. Li, and D. Chen, "A study of high-level synthesis: Promises and challenges," in *ASIC (ASICON), 2011 IEEE 9th International Conference on*, Oct 2011, pp. 1102–1105.
7. Y. Liang, K. Rupnow, Y. Li, D. Min, M. N. Do, and D. Chen, "High-level Synthesis: Productivity, Performance, and Software Constraints," *Journal of Electrical and Computer Engineering*, vol. 2012, Article ID 649057, 14 pages, Jan. 2012.

8. J. Davis, D. Buell, S. Devarkal, and G. Quan, "High-level synthesis for large bit-width multipliers on FPGAs: a case study," in *Hardware/Software Codesign and System Synthesis, 2005. CODES+ISSS '05. Third IEEE/ACM/IFIP International Conference on*, Sept 2005, pp. 213–218.
9. E. El-Araby, M. Taher, M. Abouellail, T. El-Ghazawi, and G. Newby, "Comparative Analysis of High Level Programming for Reconfigurable Computers: Methodology and Empirical Study," in *Programmable Logic, 2007. SPL '07. 2007 3rd Southern Conference on*, feb. 2007, pp. 99 –106.
10. F. Gruian and M. Westmijze, "VHDL vs. Bluespec System Verilog: A Case Study on a Java Embedded Architecture," in *Proceedings of the 2008 ACM Symposium on Applied Computing*, ser. SAC '08, 2008, pp. 1492–1497.
11. Berkeley Design Technology, Inc. (2010) High-Level Synthesis Tools for Xilinx FPGAs. [Online]. Available: http://www.bdti.com/MyBDTI/pubs/Xilinx_hlstcp.pdf
12. J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, pp. 473–491, April 2011.
13. E. Homsirikamol and K. Gaj, "Can High-Level Synthesis Compete Against a Hand-Written Code in the Cryptographic Domain? A Case Study," in *Reconfigurable Computing and FPGAs (ReConFig), 2014 International Conference on*, Dec 2014.
14. K. Gaj, E. Homsirikamol, M. Rogawski, R. Shahid, and M. U. Sharif, "Comprehensive Evaluation of High-Speed and Medium-Speed Implementations of Five SHA-3 Finalists Using Xilinx and Altera FPGAs," Cryptology ePrint Archive, Report 2012/368, 2012.
15. X. Guo, S. Huang, L. Nazhandali, and P. Schaumont, "Fair and Comprehensive Performance Evaluation of 14 Second Round SHA-3 ASIC Implementations," in *The Second SHA-3 Candidate Conference*, August 2010.
16. F. Gürkaynak, K. Gaj, B. Muheim, E. Homsirikamol, C. Keller, M. Rogawski, H. Kaeslin, and J.-P. Kap, "Lessons Learned from Designing a 65nm ASIC for Evaluating Third Round SHA-3 Candidates," in *The Third SHA-3 Candidate Conference*, Mar. 22-23 2012.
17. K. Gaj, E. Homsirikamol, and M. Rogawski, "Fair and Comprehensive Methodology for Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs," in *Cryptographic Hardware and Embedded Systems, CHES 2010*, ser. Lecture Notes in Computer Science, S. Mangard and F.-X. Standaert, Eds., vol. 6225. Springer Berlin Heidelberg, 2010, pp. 264–278. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-15031-9_18
18. CERG. GMU Source Codes. [Online]. Available: https://cryptography.gmu.edu/athena/index.php?id=source_codes
19. National Institute of Standards and Technology. (2014, Mar.) Third (Final) Round Candidates. [Online]. Available: http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/submissions_rnd3.html