HLS_Profiler: Non-Intrusive Profiling tool for HLS based Applications

Nupur Sumeet
nupur.sumeet@tcs.com
Tata Consultancy Services Research
Mumbai, Maharashtra, India

Deeksha
deeksha.14@tcs.com
Tata Consultancy Services Research
Mumbai, Maharashtra, India

Manoj Nambiar
m.nambiar@tcs.com
Tata Consultancy Services Research
Mumbai, Maharashtra, India

ABSTRACT

The High-Level Synthesis (HLS) tools aid in simplified and faster design development without familiarity with Hardware Description Language (HDL) and Register Transfer Logic (RTL) design flow that can be implemented on an FPGA (Field Programmable Gate Array). However, it is not straight forward to trace and link source code to synthesized hardware design. On the other hand, the traditional RTL-based design development flow provides the fine-grained performance profile through waveforms. With the same level of visibility in HLS designs, the designers can identify the performance-bottlenecks and obtain the target performance by iteratively fine-tuning the source code. Although, the HLS development tools provide the low-level waveforms, interpreting them in terms of source code variables is a challenging and tedious task. Addressing this gap, we propose to demonstrate an automated profiler tool, HLS_Profiler, that provides a performance profile of source code in a cycle-accurate manner.

CCS CONCEPTS

• **Hardware → Software tools for EDA**; **Board- and system-level test**; • **General and reference → Evaluation**.

KEYWORDS

Hardware profiling, HLS Designs, Performance Profile, Performance Analysis

ACM Reference Format:
Nupur Sumeet, Deeksha, and Manoj Nambiar. 2022. HLS_Profiler: Non-Intrusive Profiling tool for HLS based Applications. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering (ICPE '22 Companion), April 9–13, 2022, Bejing, China.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3491204.3527496

1 PROBLEM MOTIVATION AND SOLUTION

A recent approach for hardware design development for FPGAs is through High Level Synthesis (HLS) tools [1, 4]. With HLS, the design development productivity improves as it supports high-level languages (C/C++) and harwdare-specific details such as HDL description, RTL datapath, operation scheduling *etc.* are abstracted away from the developer. The HLS development flow includes HLS compiler for generating HDL description of source code and include co-simulation for cycle-accurate functional analysis. The special directives (e.g. #pragma in Xilinx HLS tools[4]) available in HLS tools help in design space exploration to improve the design micro-architecture and FPGA hardware matching, but their efficient use depends on the programming abilities and experience of the developer.

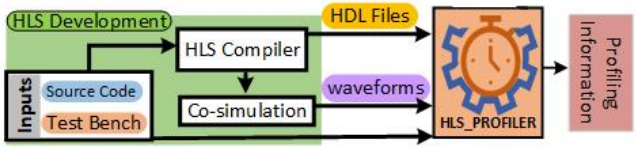

Figure 1: Representative Profiling flow for HLS-based FPGA design.

As is the case in software design, the performance profile of hardware design can help identify the performance bottlenecks and aid the developer in fine-tuning the design performance. Vivado HLS [4] and Intel HLS [1] are popular HLS tools used in industry. These HLS tools provide the overall latency of the source code along with the cycle-count at the loop or sub-function level but the cycle accounting for every line of source code is not available.

Addressing this gap, we developed HLS_Profiler framework [3], an automated and non-intrusive performance profiling tool that provides a cycle-by-cycle association to every line of source code for the entire application execution time. For this, the HLS_Profiler uses the static analysis and dynamic trace available from the HLS compilers along with associative rules to maintain correctness in the profiling. The HLS_Profiler generated profiling data is indicative of fraction of time spent on a certain section or line of code. The representative diagram of HLS_Profiler is shown in Fig. 1.

1.1 Performance Tuning using HLS_Profiler: User View

Developer can tune the performance of the design using HLS_Profiler following simple steps as listed below:

1. Inputs: Source code, Test Bench, Synthesis Frequency and Top Function Name.
2. Run the HLS_Profiler script.
3. Performance profile of the source code is available in a text file.
4. Identify, optimize bottleneck and update source code with appropriate pragma directive.
5. Repeat Steps 2-4 till performance target is met.

ICPE '22 Companion, April 9–13,2022, Bejing, China

ACM ISBN 978-1-4503-9159-7/22/04.
https://doi.org/10.1145/3491204.3527496

Table 1: HLS_Profiler Output1: Summary reports for w/o and with pragma versions of GEMM kernel.

	TYPE = double, row_size = col_size = 64, N = row_size*col_size	**w/o Pragma**			**with Pragma**		
		L_{Ln}	%time spent	Overlapping Ln (% overlap)	L_{Ln}	%time spent	Overlapping Ln (% overlap)
1:	**void** GEMM (**TYPE** m1[N], **TYPE** m2[N], **TYPE** prod[N]) {						
2:	*outer:* **for** (**int** i=0; i<row_size; i++) {	1	0.41	**3**(88.9), **4**(88.9)	1	19.2	**3**(58.6), **8**(100), **9**(82.8), **11**(89.6)
3:	*middle:* **for** (**int** j=0; j<col_size; j++) {	1	1.83	**2**(20), **4**(17.5), **6**(80) , **7**(40) , **8**(40)	1	10.1	**2**(100), **8**(100), **9**(70.6), **11**(100)
4:	**int** i_col = i*col_size;	2	0.35	**2**(100), **3**(87.5)	-	-	-
5:	**TYPE** sum = 0;	-	-	-	-	-	-
6:	*inner:* **for** (**int** k=0; k<row_size; k++) {	1	7.58	**3**(20) , **7**(40) , **8**(80), **9**(40) , **11**(10)	-	-	-
7:	**int** k_col = k * col_size;	1	1.98	**3**(25) , **6**(100) , **8**(100), **9**(75)	-	-	-
8:	**TYPE** mult = m1[i_col + k] * m2[k_col + j];	8	49.44	**3**(3.1) , **6**(25) , **7**(12.5) , **9**(21.9)	6	22.9	**2**(87.9), **3**(51.5), **9**(84.9), **11**(90.9)
9:	sum += mult;	6	37.74	**6**(16.6) , **7**(12.5) , **8**(29.2), **11**(4.2)	5	23.6	**2**(77.4), **3**(38.7), **8**(90.3), **11**(90.3)
10:	}						
11:	prod[i_col + j] = sum;	1	0.66	**6**(100) , **9**(100)	2	24.3	**2**(78.8), **3**(51.5), **8**(90.9), **9**(84.8)
12:	}	L_{Ln}- Line-wise latency. *inner* loop is unrolled using pragma. Ln #5 is missed since it has initialization statement. With Pragma, Ln # 4, 6, 7 are optimized out by HLS compiler. Initialization and optimized section of the source code are not recoverable by HLS_profiler and are limitations of the framework. [3]					
13:	}						
14:	}						

The HLS_Profiler framework takes the C source code, C test bench, synthesis frequency and top function name as user inputs. These inputs are processed by the HLS_Profiler script to generate the Source code Performance Profile. The framework collates the HLS compiler generated static and dynamic information to provide the performance profile for all the design execution clock cycles.

2 HLS_PROFILER DEMONSTRATION

The HLS tools used for this demo are Vitis HLS 2020.2.

The first step of this demo comprises of analysing the performance profile of design and identifying the performance bottleneck. For this purpose, we will use the Generic Matrix to Matrix Multiplication (GEMM) kernel from Machsuite benchmarks[2]. The GEMM source code, shown in Table 1, contains three nested **for** loops that iterate over # of rows and columns of input matrices (*m1* and *m2*) to generate *prod* matrix. The HLS_profiler tool generates a summary report that contains line-wise latency (L_{Ln}), % of time spent on every line of source code and % overlap with other lines. The summary report indicates the performance bottleneck in the source code. For instance, Ln #8 takes 49.44% of the total time spent on executing GEMM kernel and is the performance bottleneck. The line-wise latency denotes the # of clock cycles to execute a particular source code line. Furthermore, the overlapping lines denote lines executing in parallel and the extent of parallelism is captured as overlapping %. For instance, if operation on Ln *b* takes *100* cycles and operation on Ln *a* shares *p* of those 100 cycles, the overlapping % of Ln *a* with Ln *b* is p%. This information aids developer understand the operation scheduling of target design.

As the next step, we will demonstrate performance bottleneck elimination by means of HLS pragma. To achieve the same in GEMM kernel, the suitable pragmas are pipelining, unrolling and array partitioning [5]. Multiple loop iterations can run simultaneously because of pipelining+unrolling. Array partitioning fragments the memory block so that all elements of the array are accessible simultaneously. We will again run the HLS_profiler on pragma-enabled GEMM kernel and generate the summary report. The summary report indicates that the % time spent on Ln #8 has reduced and its overlapping % for all lines has increased. Additionally, Ln # 3, 6 and 7 are optimized out by HLS compiler and line-wise latency for Ln # 8 and 9 has reduced. These pragma-triggered design changes help eliminate the performance bottleneck and reduce design latency.

Following this, we will showcase the detailed cycle-wise profiling report for GEMM. The cycle-wise profile links the source code with the hardware specific temporal information. This profile can be used to trace the program execution, active variables and their value on a per cycle basis. Table 2 shows the performance profile for few cycles of GEMM kernel for i=0, j=0 and k=0, 1 iterations. It is worth noting that the increment for variables *i, j,* and *k* are compute in a pre-emptive manner. However, the incremented value is used only at the start of next iteration. The performance profile also indicates DSP processing as bottleneck operation.

Table 2: HLS_Profiler Output2: Performance Profile for w/o pragma version of GEMM kernel.

Cycle#	Ln#	Source Variable with value
15	2	i=0
16	2, 3, 4	i++ (compute), i_col=0, j=0
17	3, 6, 7, 8	j++ (compute), sum=0, k=0, k_col=0
18	6, 8	k++ (compute), m1=0.85 , m2=0.67
19-23	8	5-stage DSP Multiplier
24	8,9	mult=0.57
25-28	9	5-stage DSP Adder
29	6, 7, 8, 9	k=1 (k++ assign), sum=0.57
30	6, 8	k++ (compute), k_col=4, m1=0.93 , m2=0.05
31-35	8	5-stage DSP Multiplier
36	8,9	mult=0.046
37-40	9	5-stage DSP Adder
Profile continues till 809 cycles.		

REFERENCES

[1] INTEL. 2019. *INTEL® High Level Synthesis Compile.* https://www.intel.in/content/www/in/en/software/programmable/quartus-prime/hls-compiler.html

[2] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. MachSuite: Benchmarks for Accelerator Design and Customized Architectures. In *Intl. Sym.on Workload Characterization.* 110–119.

[3] Nupur Sumeet, Deeksha Deeksha, and Manoj Nambiar. 2022. HLS_Profiler: Non-Intrusive Profiling Tool for HLS Based Applications. In *ACM/SPEC Intl. Conf. on Performance Engineering* (Beijing, China) *(ICPE '22).* Association for Computing Machinery, New York, NY, USA, 187–198. https://doi.org/10.1145/3489525.3511684

[4] Xilinx. 2019. *Vivado Design Suite - HLx Editions.* https://www.xilinx.com/products/design-tools/vivado.html

[5] Xilinx. 2021. *Vitis High-Level Synthesis: User Guide.* https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1399-vitis-hls.pdf