Analysis of Library Functions for FPGA Compute Accelerators

Spenser Gilliland and Jafar Saniie
Illinois Institute of Technology
Chicago, Illinois 60616

Fernando Martinez Vallina
Xilinx, Inc
San Jose, California 95124

***Abstract*—As FPGAs have grown ever larger, there has been a shift in the manner in which they are programmed. Early on, it was typical for designers to design all FPGA firmware in house using VHDL and Verilog. This gradually shifted towards design reuse at the IP Core Level. However in modern times, even designs at the IP level are having trouble adapting quickly enough to customer demands. This has resulted in a change in focus towards higher level languages such as OpenCL. A key aspect of OpenCL is it's standard library and specifically the math builtins of the standard library. This paper performs an in-depth analysis of the math functions in the OpenCL standard library and develops a framework to perform further analysis of library functions being implemented on FPGAs.**

I. INTRODUCTION

In FPGA based OpenCL design, kernel developers have access to an array of library functions that make it possible to quickly implement complex elementary functions as part of an overall design [8]. It is important to have a clear understanding of the performance in terms of latency, area, and timing that these functions are able to achieve in order to determine their suitability in a given design.

OpenCL provides a unified programming enviornment where developers can use heterogeneous hardware. The kernel language for OpenCL is based on the C99 Programming Standard [1]. This standard defines a common library of elementary functions to application developers. The OpenCL standard extends the C99 programming standard by defining a specification for accuracy to these elementary functions [6].

In this paper, an in-depth anlysis of an implementation of the OpenCL math library functions is performed. To be conformant to the OpenCL standard, these functions must be accurate within the bounds specified in Table I.

Definition 1 (Unit in the Last Place) **-** If x lies between two finite consecutive floating-point numbers a and b without being equal to one of them, then $ulp(x) = |b - a|$, otherwise $ulp(x)$ is the distance between the two finite floating point numbers nearest x.

Unit in the Last Place (ULP) is used to measure the accuracy of the functions and is defined in Definition 1. To understand this definition in a more intuitive manner, it is possible to think of a ULP of 0.5 as a correctly rounded result; a ULP of 0 as an exact result, and a ULP of ∞ as no bounds on accuracy. For other integer values, a ULP of X means that the result is within X number of bits of the reference implementation.

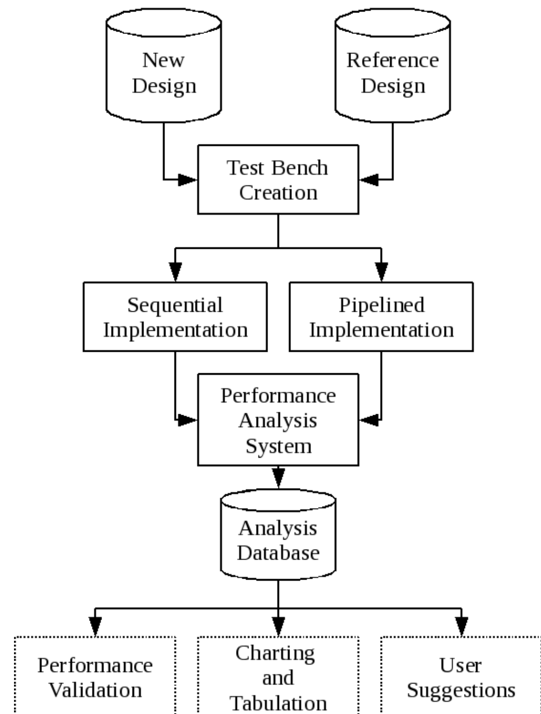

Fig. 1. SDAccel Based Library Performance Analysis Tool

The key value this paper provides is a tool that can be used to extract performance, timing, and latency information in an automated manner that provides in-depth analysis of the functional suitability of a library function for FPGA implementation. FPGA-based system design is unique in that their are tradeoffs in both space and time that must be accounted for when designing new systems [3]. A design can usually be made lower latency; however, it may result that the chip can only be programmed for this specific function [10]. Understanding the area vs latency tradeoff is the key aspect of developing advanced designs for FPGAs [7].

II. METHODOLOGY FOR LIBRARY FUNCTION IMPLEMENTATION

A. Introduction

In High Level Synthesis (HLS), there are two primary methodologies that can be used to implement a function. These methodologies are a trade off between area and performance. One methodology primarily focuses on creating a design

978-1-5090-4767-3/17/$31.00 ©2017 IEEE

Authorized licensed use limited to: Illinois Institute of Technology. Downloaded on October 02,2020 at 15:16:04 UTC from IEEE Xplore. Restrictions apply.

TABLE I. OPENCL MATH BUILTIN FUNCTION SPECIFICATION

Function	Description	Required Accuracy
acos(x)	arccosine	4 ULP
asin(x)	arcsine	4 ULP
atan(x)	arctangent($\frac{x}{y}$)	5 ULP
atan2(x, y)	arctangent	6 ULP
acosh(x)	area hyperbolic cosine	4 ULP
asinh(x)	area hyperbolic sine	4 ULP
cos(x)	cosine	4 ULP
cospi(x)	cosine($x\pi$)	4 ULP
sin(x)	sine	4 ULP
sinpi(x)	sine($x\pi$)	4 ULP
sqrt(x)	square root	4 ULP
tan(x)	tangent	5 ULP
tanpi(x)	tangent($x\pi$)	6 ULP
exp(x)	e^x	3 ULP
expm1(x)	$e^x - 1$	3 ULP
log(x)	natural logarithm	4 ULP
log2(x)	base two logarithm	4 ULP
log10(x)	base 10 logarithm	4 ULP
log1p(x)	natural logirthm of x plus 1	4 ULP
pow(x,y)	x^y	16 ULP
pown(x,y)	x^y where y is an integer	16 ULP
powr(x,y)	x^y for x greater than 0	16 ULP
cbrt(x)	cube root	4 ULP
rsqrt(x)	inverse square root	4 ULP
rootn(x, y)	$x^{\frac{1}{y}}$	16 ULP

which uses the least amount of area in the FPGA. For the purposes of this paper, this is defined as sequential implementation. In other instances, the primary focus is performance. In performance sensitive instances, users will generally try to use a pipeline to accomplish their task. Therefore, this methodology is defined as pipelined implementation. These two implementation styles lead to a distinct dichotomy that will have substantial impact in how the design is implemented in hardware.

For example, a user may want to use a sequential implementation of the sin function if they will be performing a translation of an image before processing via neural network. This allows the neural network to use more of the FPGA fabric and results in a lower latency implementation. However in options pricing applications, the normalization of the random numbers is in the critical path of the overall option pricing evaluation. Therefore, a designer would prefer to use a pipelined implementation for this use case.

While these two designs styles are incredibly different from a hardware perspective, they represent only a single pragma difference within an HLS function. Thus, a key focus of the library developer is to analyze the two different ways that function can be instantiated in order to create a design that is flexible enough to cover both use cases.

B. Sequential Implementation

Sequential implementation of a library function is defined when an application programmer uses the following symantics. The library function will be instantiated as a function call in either the top level function or a sub function of the top level function. However in neither the top level function nor the subfunctions will a pipeline pragma be used. In such an instance, the SDAccel tool will default to using the smallest possible area to create the function. An example of how to force this implementation style is shown in Listing 1.

Listing 1. Sequential Implementation Example

```
void hls_seq_cos(float *in, float *out) {
  #pragma HLS INTERFACE m_axi port=in \
    offset=slave bundle=gmem0
  #pragma HLS INTERFACE m_axi port=out \
    offset=slave bundle=gmem1
  #pragma HLS INTERFACE s_axilite \
    port=in bundle=control
  #pragma HLS INTERFACE s_axilite \
    port=out bundle=control
  #pragma HLS INTERFACE s_axilite \
    port=return bundle=control

  SEQ_LOOP: for(size_t i = 0; i < VALS; i++) {
    #pragma HLS PIPELINE off
    out[i] = cos(in[i]);
  }
}
```

C. Pipelined Implementation

Pipelined implementation of a library function is when the function is implemented such that either the top level function, a subfunction, a loop or a subloop of the top level function is pipelined. This informs the compiler that throughput is the most desired aspect for this code. As such, the compiler will transform the code into a pipelined block that attempts to accomplish one iteration of the loop each clock cycle with multiple iterations in flight at any given point in time. An example which forces pipelining of the cos function is shown in Listing 2.

Listing 2. Pipelined Implementation Example

```
void hls_pipe_cos(float *in, float *out) {
  #pragma HLS INTERFACE m_axi port=in \
    offset=slave bundle=gmem0
  #pragma HLS INTERFACE m_axi port=out \
    offset=slave bundle=gmem1
  #pragma HLS INTERFACE s_axilite \
    port=in bundle=control
  #pragma HLS INTERFACE s_axilite \
    port=out bundle=control
  #pragma HLS INTERFACE s_axilite \
    port=return bundle=control

  PIPE_LOOP: for(size_t i = 0; i < VALS; i++) {
    #pragma HLS PIPELINE II=1
    out[i] = cos(in[i]);
  }
}
```

III. PERFORMANCE ANALYSIS SYSTEM

A. Design of the Performance Analysis System

In this section, a performance analysis system is developed that can be used to analyze many library functions. The primary benefit of this system is that it makes it easier to analyze the performance, resource usage, and timing aspects of each function in a larger library.

A block diagram of the system is presented in Figure 1. This figure shows the primary building blocks of the overall system and how they fit together. Automated processes are shown with rectangles and databases are shown with cylinders. The last few automated processes (with dashed lines) are examples of use cases that can utilize the analysis database.

The analysis database is an easily parseable json document which describes the performance, timing, and resources usage aspects of the sequential and pipelined implementations. An example listing of the analysis database for a single function is shown in Listing 3.

Authorized licensed use limited to: Illinois Institute of Technology. Downloaded on October 02,2020 at 15:16:04 UTC from IEEE Xplore. Restrictions apply.

Listing 3. Example Performance Analysis Database

```
{
  ”cos”: {
    ”func”: ”cos”,
    ”pipe_hls”: {
      ”avg_case_latency”: 1232,
      ”best_case_latency”: 1232,
      ”bram”: 4,
      ”delay”: 4.95,
      ”dsp”: 90,
      ”ff”: 57826,
      ”loops”: {
        ”PIPE_LOOP”: {
          ”depth”: 221,
          ”ii”: 1,
          ”latency”: 1219
        }
      },
      ”lut”: 94665,
      ”period”: 5.0,
      ”slack”: 0.04999999999999982,
      ”worst_case_latency”: 1232
    },
    ”seq_hls”: {
      ”avg_case_latency”: 187012,
      ”best_case_latency”: 4012,
      ”bram”: 6,
      ”delay”: 4.95,
      ”dsp”: 90,
      ”ff”: 8931,
      ”loops”: {
        ”SEQ_LOOP”: {
          ”latency”: 421000
        }
      },
      ”lut”: 15116,
      ”period”: 5.0,
      ”slack”: 0.04999999999999982,
      ”worst_case_latency”: 421012
    }
  },
}
```

B. Implementation of the Performance Analysis System

This system is based on the SDAcccel 2016.3 release which includes Vivado HLS 2016.3. The system includes Google Test [5] for running functional level validation; a custom ULP analysis tool to analyze accuracy in floating point designs; and a custom test bench which utilizes SDAccel to provide runtime testing of the library functions in hardware.

For each analyzed function, a sequential and a pipelined implementation is created by the performance analysis system. Each implementation is then analyzed using a custom script that scrapes the xml versions of the HLS reports. From these reports, it is possible to get highly accurate estimates about the resource usage, latency, and timing performance of the implementations.

By controlling the top level function, the tool knows the label of the loop that implements the function. This allows it to grab detailed information about how long the loop should take to execute in best case, worst case, and average case. For pipelined implementations, it is known that the best case, worst case, and average case performance are equivalent. Thus, only a single value is presented to the user.

The tool uses function signatures to develop inputs, outputs and implement tests. There are many precreated function signatures available to users. In the case that precreated function signatures do not fit the users needs, the following functions will need to be created by the user. First, a type is created for the input and output to the function. Then, a user implements the test vector generation. Finally, a user provides a reference function for the validation.

IV. Example Analysis of OpenCL Math Functions

To provide an example of how this tool can improve the analysis of function design for FPGA based implementation, an example analysis of an implementation of the OpenCL math builtins is performed. In this example, we are using the math builtins described in [4]. The library was written utilizing CORDIC methods [2] and the Payne Hanek Reduction technique [9]. The result is a conformant but not yet completely optimized body of functions. In this section, the results of the performance analysis tool for this body of functions is analyzed to provide feedback and insight into the implementations.

A. Sequential Implementation Analysis

In Table II, some of the results from the sequential section of the analysis database are tabulated. The resource usage in terms of look up tables (LUT), block RAMs (BRAM), flip flops (FF), and digital signal processing (DSP) elements is shown as well as the total worst case execution latency for 1000 invocations of the function.

These results show that while there is room for improvement, the resulting designs are able to fit in most modern FPGAs. A major outlier is the DSP usage in the cos, sin, and tan implementations. This signals that some multipliers are being used aggressively by the HLS implementation and should be further investigated. Furthermore, the LUT usage in the acosh and pow instances seems to be high. These issues in resource usage will need to be further analyzed by the user.

The asinh and acosh functions take approximately $25ms$ to execute in the worst case. These functions will need to be investigated further to better understand why they are taking so long to execute. In addition, the logrithm and exponential functions are taking a substantial amount of time ($15ms$) to execute in the worst case Each of these functions will be made a priority in the next round of analysis.

B. Pipelined Implementation Analysis

In Table III, some of the results from the pipelined section of the performance analysis database are tabulated. Similar to Table II, the resource usage in terms of LUTs, BRAMs, FFs and DSPs is explored as well as the worst case latency for 1000 iterations. It should be noted, that in all cases the initiation interval (II) of the functions is one; therefore, it is not shown in the table.

The latency results match expectations as an II of one means that the total latency of the function is the number of executions plus the pipeline depth. This indicates that the pipeline is working properly. However, it can be observed that many functions are taking up to 300 more cycles than the number of iterations. This means that there are very deep pipelines being created to implemented these functions and that there is very high inter-dependence inside the functions. Further analysis of the inter-dependence inside the function will be a focus of the next round of analysis.

Authorized licensed use limited to: Illinois Institute of Technology. Downloaded on October 02,2020 at 15:16:04 UTC from IEEE Xplore. Restrictions apply.

TABLE II. SEQUENTIAL IMPLEMENTATION RESOURCE USAGE

Function	BRAM	DSP	FF	LUT	Latency
acos	6	3	5064	11126	556012
acosh	16	10	10177	24260	4849012
asin	6	3	4931	9880	559012
asinh	8	13	4800	9158	4792012
atan	6	3	4912	9483	374012
atan2	6	3	5394	10167	405012
cbrt	8	16	3479	7234	2485012
cos	6	90	8931	15116	421012
cospi	6	13	4858	12486	386012
exp	6	10	3508	8434	169012
exp10	6	10	3528	8401	169012
exp2	6	2	3329	7927	183012
expm1	6	12	4644	11219	180012
log	6	3	2985	6034	2332012
log10	6	3	2985	6034	2332012
log1p	8	5	4856	11637	2342012
log2	6	0	2797	5886	2327012
pow	8	55	22735	23169	2861012
pown	4	22	9465	6354	353012
powr	8	31	13285	14960	2858012
rootn	8	0	10810	14673	2551012
rsqrt	8	0	4263	8041	2486012
sin	6	90	8997	15253	422012
sinpi	6	13	4858	12486	386012
sqrt	8	0	3355	7076	2468012
tan	6	90	10035	16481	440012
tanpi	6	13	5864	13618	404012

TABLE III. PIPELINE IMPLEMENTATION RESOURCE USAGE

Function	BRAM	DSP	FF	LUT	Latency
acos	4	9	92226	171734	1371
acosh	388	24	121046	154521	1343
asin	4	9	104711	178906	1373
asinh	196	13	41748	74029	1289
atan	4	3	42734	85197	1190
atan2	4	6	45541	87661	1220
cbrt	100	16	30830	45731	1192
cos	4	90	57826	94665	1232
cospi	4	17	47039	88443	1196
exp	4	10	13132	20137	1084
exp10	4	10	13166	20104	1084
exp2	4	4	13240	19360	1097
expm1	4	22	25434	38859	1094
log	100	3	21766	31486	1133
log10	100	3	21766	31486	1133
log1p	196	8	42713	62366	1143
log2	100	0	21578	31341	1128
pow	100	1027	137566	94663	1365
pown	4	506	37660	21002	1338
powr	100	515	87303	69982	1354
rootn	100	0	40328	53552	1259
rsqrt	100	0	37315	46693	1194
sin	4	90	57797	94863	1232
sinpi	4	17	47039	88443	1196
sqrt	100	0	30148	45660	1175
tan	4	90	65715	96047	1250
tanpi	4	17	48045	89531	1214

With regards to the resources usage, results show that in many cases a pipelined implementation of the functions in this library would require a large FPGA device to implement. Specifically, the asin and acos implementations need further attention due to their high LUT resource usage. In addition, it appears many DSP resources are being used in the pow, pown, and powr functions. These resource usage issues will need to be evaluated in the next round of analysis.

V. CONCLUSION

In this paper, we have shown a tool to analyze the performance of libraries when implemented using high level synthesis. There is additional work to be completed for the presented libraries and the gaps become readily apparent when analyzed with the performance analysis tool.

A substantial amount of data is generated during each compilation of a target kernel in SDAccel. This data can be analyzed to provide true value that guides the developer towards a more optimal solution. Aggregating and cross coorelating this data makes it possible to drastically improve the productivity of library developers as well as the quality of results.

Future work will focus on extracting implementation results from Vivado as well as the profiling data from SDAccel. This will allow the user to examine the difference between the estimates from the HLS tool and the physical results as implemented on the chip. Another focus is to perform power analysis using the Vivado Power Estimation tool to generate estimates and the on-chip power regulator to measure actual usage.

REFERENCES

[1] "iso/iec 9899:1999 (e) contents."

[2] R. Andraka, "A survey of cordic algorithms for fpga based computers," in *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, ser. FPGA '98. New York, NY, USA: ACM, 1998, pp. 191–200. [Online]. Available: http://doi.acm.org/10.1145/275107.275139

[3] 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.

[4] S. Gilliland, J. Saniie, and F. M. Vallina, "Implementation of elementary functions for fpga compute accelerators," in *2016 IEEE International Conference on Electro Information Technology (EIT)*, May 2016, pp. 0179–0182.

[5] (2016, December) Google test. Google. [Online]. Available: https://github.com/google/googletest

[6] K. O. W. Group *et al.*, "The opencl specification," *version*, vol. 1, no. 29, p. 8, 2008.

[7] G. Guidi, E. Reggiani, L. D. Tucci, G. Durelli, M. Blott, and M. D. Santambrogio, "On how to improve fpga-based systems design productivity via sdaccel," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 247–252.

[8] F. Muslim, L. Ma, M. Roozmeh, and L. Lavagno, "Efficient fpga implementation of opencl high-performance computing applications via high-level synthesis," *IEEE Access*, vol. PP, no. 99, pp. 1–1, 2017.

[9] M. H. Payne and R. N. Hanek, "Radian reduction for trigonometric functions," *SIGNUM Newsl.*, vol. 18, no. 1, pp. 19–24, Jan. 1983. [Online]. Available: http://doi.acm.org/10.1145/1057600.1057602

[10] K. Wang and J. Nurmi, "Using opencl to rapidly prototype fpga designs," in *2016 IEEE Nordic Circuits and Systems Conference (NORCAS)*, Nov 2016, pp. 1–6.

Authorized licensed use limited to: Illinois Institute of Technology. Downloaded on October 02,2020 at 15:16:04 UTC from IEEE Xplore. Restrictions apply.