Implementation of Elementary Functions for FPGA Compute Accelerators

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

Abstract—Field programmable gate arrays (FPGA) are growing from the role of glue logic into the area of application acceleration and compute. This is fostered by advances in silicon technologies as well as standards based methodologies for interacting with heterogeneous compute resources. As these standards generally require the implementation of elementary functions, this work outlines the implementation and evaluation of the elementary functions required by the heterogeneous programming standard OpenCL. It outlines the implementation of the math “builtin” functions using CORDIC methods and details the processes that will be taken to benchmark the resource usage, maximum frequency, and latency of each function on Xilinx 7 Series FPGAs. Because of the applicability and standardization of the OpenCL math functions, this benchmarking effort provides a basis for understanding and analysing future implementations.

I. INTRODUCTION

Mobile, data center, and high performance computing (HPC) are rapidly approaching a thermal and power wall due to the inefficiencies of homogeneous compute architectures. This has caused attention to shift from performance per dollar towards performance per watt as the primary metric which defines overall system performance. The change in focus has been quantified as part of the Green 500 [1] benchmarking effort. Due to these changes, modern computing systems are increasingly defined by a heterogeneous architecture, which includes one or more general purpose processors augmented by accelerators. Traditionally, these accelerators have been graphics processing units (GPU) due to their inclusion in most commercially available computing systems. However, FPGAs are an interesting competitor in this market due to their ability to achieve higher performance per watt for many applications.

With the advent of OpenCL, there now exists a defacto common language to express hardware acceleration on heterogeneous compute architectures. As the OpenCL kernel language is based on the C99 standard [2], FPGAs utilize high level synthesis (HLS) tools to create FPGA configurations from the kernel source code. HLS tools have made major strides in recent years and represent a possible alternative to register transfer level (RTL) schematic capture in many applications [3]. However, a major remaining step is the implementation of optimized standard math libraries compliant with the OpenCL standard.

The primary contribution of this paper is the implementation and benchmarking of the single precision math functions required by the embedded profile of the OpenCL 2.0 specification. These implementations were then benchmarked for latency, maximum frequency, and resource usage.

A. Hardware Accelerators and Heterogeneous Systems

The introduction of general purpose compute accelerators has increased the overall efficiency of compute intensive tasks. Accelerators generally improve throughput by trading single thread performance for a higher density of cores. When applications can take advantage of multiple cores, the performance is greatly improved by offloading the parallel tasks to an accelerator.

Fig. 1. Median Data Center Cost Breakdown

In Data Center environments, the primary objective is to achieve the lowest total cost of ownership for a workload. As most Data Center systems have a service life of between three to five years, the power and operational budget of such a system will generally make up approximately 15 percent of the total costs per year [4]. A total budget breakdown per year is shown in figure 1. As hardware accelerators offer increases in efficiency, they can allow systems to use less power, which allows operators to either increase density or decrease the total power usage.

Current HPC methodologies focus on using a large number of general purpose processors. General purpose processors are designed to maximize single core performance. As a result, designers utilize high transistor counts per processor, high voltages, and high frequencies to maximize performance. By using methods such as branch prediction, multi-level cache structures, scoreboard, etc, designers have increased single-core performance at the expense of idle transistors. These techniques limit the number of transistors that are being used for computation at any one time in exchange for lower...
latency in single-core applications. Additionally by increasing frequencies and voltages in order to extract the maximum performance out of the few transistors that are active in any given applications, static and dynamic power as well as heat dissipation are exponentially increased.

GPU-based heterogeneous computing is starting to establish a foothold in the HPC market. GPUs are simpler processors traditionally defined in the Flynn Taxonomy as single instruction multiple data (SIMD). The processors in these accelerators may have lower clock speeds and higher latencies than CPUs but make up for this deficiency by including an order of magnitude more cores per device.

FPGAs are an emerging competitor to GPUs for accelerated processing. FPGAs utilize a heterogeneous configurable fabric that interconnects DSP, Slice, BRAMs, and IOBs elements to create virtually any synchronous circuit. These devices are able to express extremely fine-grained parallelism and an optimized pipeline based on the complexity and requirements of the application.

FPGAs are currently being evaluated for their possible use as a platform for compute in several applications. Notably, they were recently selected by Microsoft for the implementation of search algorithms [5]. This has led to a larger interest in FPGA-based computing for high performance and data center.

B. OpenCL

OpenCL is a standard that allows interoperability between accelerators and algorithm developers. Its sole purpose is to enable the use of heterogeneous compute elements within a general purpose computing environment. This enables a standard methodology for utilizing computation accelerators such as GPU, multiprocessors (MP), FPGAs, and application specific integrated circuits (ASIC).

OpenCL divides the problem of heterogeneous compute platforms into two major components; the host application programming interface (API) and computation kernels. Compute kernels are written in a subset of the C99 programming standard, which must be fully supported by all OpenCL compliant accelerators. This enables kernels and host code to be functionally portable between accelerators.

The host API primarily provides functions to queue computation kernels, copy data to devices, and retrieve data from device accessible memory. As shown in figure 2, a host will accelerate applications using the following pattern: copy data to device accessible memory; initiate compute kernel; finally, copy results from device to host accessible memory. Other routines are available on the host including device discovery and kernel compilation. Host code runs on general purpose computing system such as an x86 or ARM.

C. OpenCL Math Builtins

Kernels have access to a library of standard functions or “builtins.” These include elementary math functions, image manipulation functions, conversion routines, etc. To be compliant with the OpenCL specification, each “builtin” must either exactly or within a specified error produce the same results as a reference function. The main benefit of OpenCL requiring full compliance is that kernels written for one device will be functionally portable between different accelerators.

Fig. 2. OpenCL context with multiple queues

II. ALGORITHMS FOR ELEMENTARY FUNCTION APPROXIMATIONS

A. Introduction

In FPGAs all functions must be implemented using the combination of DSP, LUT, and BRAM elements. As each of these elements are limited within a given FPGA, a general purpose implementation of the elementary functions should attempt to reduce resource usage to a minimum. This allows optimal flexibility for the HLS tools to pipeline and replicate the implementation. For this reason a CORDIC based implementation was chosen. CORDIC based algorithms are a class of shift and add algorithms which utilize the rotation of a vector to achieve convergence.

1) Unit in the Last Place: The primary goal of this proposal is the implementation of all functions required for the OpenCL 2.0 Embedded Profile Specification. The main metric used to describe accuracy of the implementation is unit in the last place (ULP). By using definition 1, it is clear that 0 ULP means a function has an exact answer and 0.5 means a function must be properly rounded.

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 \( x \).

B. CORDIC Algorithms

The CORDIC algorithm was originally developed by Volder [6] and adapted to the common case by Walther [7]. The main reference used throughout this implementation was [8]. However, it is also well described in other books such as [9], there is an excellent overview of the hardware implementation in [10], and error bounding is well described in [11]. In addition, some computations require double rotations which are described in [12], and [13]. The rotation of a vector using CORDIC is shown in figure 3.

The CORDIC iteration is defined as

\[
\begin{align*}
(x_{n+1}, y_{n+1}) &= \left( \frac{1}{d_n}, \frac{-d_n}{2^n} \right) (x_n, y_n) \\
t_{n+1} &= t_n - d_n \arctan(2^{-n})
\end{align*}
\]

1) Rotation Mode: Rotation mode is a method for solving sin and cos of \( z_0 \). In this mode, \( d_n \) is chosen such that \( d_n = \text{sign}(z_n) \).
on double rotations may be used. This can be used to compute $\sin$ and $\cos$ simultaneously using the following iteration scheme.

$$\begin{align*}
x_0 &= \frac{1}{K} \\
y_0 &= 0 \\
t_0 &= \theta
\end{align*}$$

(3)

2) Vectoring Mode: In vectoring mode the direction, $d_n$, is chosen such that $d_n = -\text{sign}(y_n)$.

$$\begin{align*}
x_n &= K \sqrt{x_0^2 + y_0^2} \\
y_n &= 0 \\
t_n &= t_0 + \text{arctan}\left(\frac{y_0}{x_0}\right)
\end{align*}$$

(4)

This can be used to compute $\text{atan}$, $\text{atan2}$ and $\text{hypot}$ functions simultaneously using the following iteration scheme.

$$\begin{align*}
x_0 &= \frac{x_0}{K} \\
y_0 &= \frac{y_0}{K} \\
t_0 &= 0
\end{align*}$$

(5)

$$\begin{align*}
x_n &= \sqrt{x_0^2 + y_0^2} \\
y_n &= 0 \\
t_n &= \text{arctan}\left(\frac{y_0}{x_0}\right)
\end{align*}$$

(6)

3) $\text{sin}^{-1}$ and $\text{cos}^{-1}$ Mode: A number of papers have been written on the topic of $\text{sin}^{-1}$ and $\text{cos}^{-1}$ computation using CORDIC [11], [14], [15]. The method used for implementation is described in [8].

To determine the $\text{sin}^{-1}(\theta)$, the following algorithm based on double rotations may be used.

$$\begin{align*}
x_{n+1} &= \left(1 - d_n 2^{-n}\right) x_n \\
y_{n+1} &= \left(1 - d_n 2^{-n}\right) y_n \\
t_{n+1} &= t_n + 2d_n \text{arctan}(2^{-n}) \\
v_{n+1} &= v_n + v_n 2^{-2n}
\end{align*}$$

In this algorithm, $d_n$ is chosen such that $d_n = \text{sign}(x)$ when $y_n <= v_n$ else $d_n = -\text{sign}(x)$.

$$\begin{align*}
v_0 &= \theta \\
v_n &= K^2 v_0 \\
x_0 &= \frac{1}{K^2} \\
x_n &= \sqrt{(K^2 x_0)^2 - c^2} \\
y_0 &= v_n \\
y_n &= v_n \\
t_0 &= 0 \\
t_n &= \text{sin}^{-1}(v_0)
\end{align*}$$

(7)

Thus by using the following initial arguments, it is possible to compute $\text{sin}^{-1}$.

A similar method can be used to determine $\text{cos}^{-1}$.

4) Hyperbolic Functions: The hyperbolic functions may be computed using CORDIC with minor changes. To allow a CORDIC core to implement both Hyperbolic and Trigonometric functions we introduce the parameter $m$.

$$\begin{align*}
x_{n+1} &= \left(1 - m d_n 2^{-n}\right) x_n \\
y_{n+1} &= \left(1 - m d_n 2^{-n}\right) y_n \\
t_{n+1} &= t_n - d_n \text{tanh}^{-1}(2^{-n})
\end{align*}$$

(8)

When utilizing a lookup table for the $\text{tanh}^{-1}$ and $\text{tan}^{-1}$ both the hyperbolic and trigonometric functions may be implemented in the same CORDIC.

III. PERFORMANCE COMPARISON

A. Microbenchmark Analysis

Microbenchmarks which evaluate each elementary function individually have been developed as part of this proposal. One tests the total size for a fully sequential implementation of the function, another tests a pipelined version with 256 different inputs. Another, utilizes a test bench to test accuracy using emulation on a computing cluster.
TABLE I. RESULTS OF MICROBENCHMARK ANALYSIS

<table>
<thead>
<tr>
<th>Function</th>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUTS</th>
<th>Delay (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pown</td>
<td>0</td>
<td>22</td>
<td>4562</td>
<td>5757</td>
<td>8.62</td>
</tr>
<tr>
<td>acos</td>
<td>4</td>
<td>3</td>
<td>2945</td>
<td>9196</td>
<td>8.34</td>
</tr>
<tr>
<td>asin</td>
<td>4</td>
<td>6</td>
<td>6285</td>
<td>19124</td>
<td>8.44</td>
</tr>
<tr>
<td>atan</td>
<td>4</td>
<td>3</td>
<td>3201</td>
<td>8681</td>
<td>8.34</td>
</tr>
<tr>
<td>atan2</td>
<td>4</td>
<td>3</td>
<td>3525</td>
<td>9847</td>
<td>8.74</td>
</tr>
<tr>
<td>cospi</td>
<td>4</td>
<td>13</td>
<td>3116</td>
<td>9394</td>
<td>8.63</td>
</tr>
<tr>
<td>sinh</td>
<td>4</td>
<td>13</td>
<td>3116</td>
<td>9394</td>
<td>8.63</td>
</tr>
<tr>
<td>tanpi</td>
<td>4</td>
<td>13</td>
<td>4089</td>
<td>10901</td>
<td>8.63</td>
</tr>
<tr>
<td>atanh</td>
<td>6</td>
<td>0</td>
<td>2503</td>
<td>8051</td>
<td>8.54</td>
</tr>
<tr>
<td>cosh</td>
<td>4</td>
<td>26</td>
<td>4738</td>
<td>12719</td>
<td>8.47</td>
</tr>
<tr>
<td>sinh</td>
<td>4</td>
<td>26</td>
<td>4799</td>
<td>12817</td>
<td>8.47</td>
</tr>
<tr>
<td>tanh</td>
<td>4</td>
<td>26</td>
<td>5837</td>
<td>14532</td>
<td>8.47</td>
</tr>
<tr>
<td>cos</td>
<td>4</td>
<td>90</td>
<td>5605</td>
<td>9599</td>
<td>8.34</td>
</tr>
<tr>
<td>sin</td>
<td>4</td>
<td>90</td>
<td>5637</td>
<td>9767</td>
<td>8.42</td>
</tr>
<tr>
<td>tan</td>
<td>4</td>
<td>90</td>
<td>6579</td>
<td>11370</td>
<td>8.42</td>
</tr>
<tr>
<td>log2</td>
<td>2</td>
<td>0</td>
<td>1050</td>
<td>2762</td>
<td>8.34</td>
</tr>
<tr>
<td>exp2</td>
<td>2</td>
<td>2</td>
<td>1483</td>
<td>5343</td>
<td>8.63</td>
</tr>
<tr>
<td>log</td>
<td>2</td>
<td>3</td>
<td>1196</td>
<td>3085</td>
<td>8.34</td>
</tr>
<tr>
<td>log10</td>
<td>2</td>
<td>3</td>
<td>1196</td>
<td>3085</td>
<td>8.34</td>
</tr>
<tr>
<td>exp</td>
<td>2</td>
<td>10</td>
<td>1545</td>
<td>4871</td>
<td>8.34</td>
</tr>
<tr>
<td>exp10</td>
<td>2</td>
<td>10</td>
<td>1579</td>
<td>4939</td>
<td>8.34</td>
</tr>
<tr>
<td>log1p</td>
<td>2</td>
<td>5</td>
<td>1899</td>
<td>4832</td>
<td>8.74</td>
</tr>
<tr>
<td>expml</td>
<td>2</td>
<td>12</td>
<td>2248</td>
<td>7143</td>
<td>8.34</td>
</tr>
<tr>
<td>pow</td>
<td>4</td>
<td>31</td>
<td>8105</td>
<td>14363</td>
<td>8.63</td>
</tr>
<tr>
<td>pow</td>
<td>4</td>
<td>55</td>
<td>14210</td>
<td>25922</td>
<td>8.73</td>
</tr>
<tr>
<td>rootn</td>
<td>4</td>
<td>0</td>
<td>2667</td>
<td>5722</td>
<td>8.48</td>
</tr>
<tr>
<td>cbrt</td>
<td>4</td>
<td>16</td>
<td>1646</td>
<td>4101</td>
<td>8.34</td>
</tr>
<tr>
<td>sqrt</td>
<td>4</td>
<td>0</td>
<td>1506</td>
<td>3923</td>
<td>8.52</td>
</tr>
<tr>
<td>rsqrt</td>
<td>4</td>
<td>0</td>
<td>2448</td>
<td>5276</td>
<td>8.52</td>
</tr>
<tr>
<td>acosh</td>
<td>10</td>
<td>10</td>
<td>6138</td>
<td>15264</td>
<td>8.74</td>
</tr>
<tr>
<td>asinh</td>
<td>4</td>
<td>13</td>
<td>2522</td>
<td>5966</td>
<td>8.75</td>
</tr>
</tbody>
</table>

1) Latency, Delay, Area, and Power: Latency combined with a delay specification is the metric which determines the speed with which an implementation can produce a result. Area is defined as the number of FPGA elements used in the implementation of the elementary function. This is measured in number of BRAM, DSP, and LUT elements. The results for pown, acos, asin, atan, atan2, cospi, sinh, tanpi, atanh, cosh, sinh, tanh, cos, sin, tan, log2, exp2, log, log10, exp, exp10, log1p, expml, powr, pow, rootn, cbrt, sqrt, rsqrt, acosh, and asinh are listed in Table I.

IV. CONCLUSION

As computing systems have evolved, there has been a long trend of using more and more transistors to accomplish a given task. FPGAs turn this idea on its head, they suggest that building highly efficient small blocks and connecting them together will yield a more efficient and more performant solution. Thus, FPGA based parallel processing is a solution to one of the largest problems on the ITRS 2013 road map: Power Management [16].

“Power management is now the primary issue across most application segments due to the 2x increase in transistor count per generation while cost effective heat removal from packaged chips remains almost flat.”

REFERENCES
