Maximising Parallel Memory Access for Low Latency FPGA Designs

Stewart Denholm and Wayne Luk
Department of Computing, Imperial College London, UK
{swd10, w.luk}@imperial.ac.uk

***Abstract*—Memory-based computing stores pre-computed function results in memory to be read at runtime. FPGAs group together multiple block memories (BRAMs) to form this memory, all accessed as a single monolithic device. We introduce a novel ring-based architecture to leverage parallel accesses to these constituent BRAMs, benefiting low latency applications that rely on: highly-complex functions; numerical precision via iterative computation; or many parallel data-paths accessing a shared memory resource. The implemented function's performance is independent of its complexity, enabling significant latency reductions for compute-bound operations. We assess common functions (sqrt, power, trigonometric, hyperbolic functions) on the Xilinx Alveo U280 FPGA. Our function-agnostic memory-compute core can serve 1024 parallel function calls at 300MHz and reduce latency 4.4-29x versus traditional FPGA implementations.**

***Index Terms*—memory, parallelism, low latency, memory-based computing**

I. Introduction

Latency is often traded-off for data precision in iterative operations—such as CORDIC-based implementations [1]—or traded for throughput by increasing the number of pipeline stages to allow for higher operating frequencies [2]. Applications that depend on low latencies are then at a disadvantage if they employ operations that are typically improved by these methods, or require higher degrees of numerical precision that can only come from latency-costly computation.

Such latency-dependent designs play a key role in real-time applications, like tele-operations and automation, as well as all manner of consumer and business applications, from robotics to high frequency trading. These functions may be: formed of a chain of multiple, elementary operations; involve floating point, multi-cycle, or other computationally expensive operations; or be a bespoke, user-defined function to which no standard approximation will apply. For brevity, we term these simply as "complex functions", and quantify them further during our implementation.

Traditional function evaluation and approximation, such as distributed arithmetic [3], polynomial evaluation [4], or table-based methods [5] can offer latency improvements, but pre-computing function values and storing them in low latency FPGA block memories (BRAMs) is the least latency approach. While costly in terms of resources, it can be applied to any deterministic function, and requires just 1 cycle to return the result of any given function input.

This can suffer when: **1)** scaled to larger numbers of BRAMs; **2)** we require higher degrees of data-path parallelism; or **3)** there are limited options for parallel memory access. The unbalanced nature of available memory ports not matching the required number of parallel accesses also becomes an issue.

Within an FPGA, a large memory is formed by grouping together N_{SM} smaller sub-memory units, which is then accessed as a single, monolithic structure. Independent access to the constituent BRAMs is no longer possible. Storing pre-computed values in look-up tables is not new, but little work has been done to leverage parallel accesses to the underlying BRAMs to benefit low latency designs.

This paper introduces a novel approach for highly parallel function evaluation by exploiting the parallelism of FPGA BRAMs. Our memory-compute core serves multiple, parallel calls to its implemented function by independently accessing the BRAMs that are traditionally accessed as one, single unit.

Our architecture's resources and access times scale linearly to both higher degrees of parallelism, i.e., more calls to the implemented function, and the number of BRAMs accessed in parallel. The interconnection scheme requires little logic and routing, automatically arbitrating parallel accesses between the multiple function calls and BRAMS. Our approach decouples latency from the complexity of the implemented function, allowing for higher clock speeds and improved throughput.

In summary, the main contributions of our work are:

- a novel, minimally routed and interconnected architecture that independently accesses the multiple BRAMs that collectively serve as a function look-up table, and scales to larger numbers of BRAMs without the resource, routing and latency costs seen by traditional approaches;
- a ring-based topology facilitating multiple, parallel calls to the implemented function, decoupled from the number of underlying BRAMs and available memory ports;
- the implementation and evaluation of square root, Gaussian, trigonometric and hyperbolic functions on a Xilinx Alveo U280, showing enhanced performance, scalability and resource utilisation over existing FPGA designs.

II. Targeting Low Latency Applications

Latency reduction via multi-bank memory accesses are more commonly explored for High Bandwidth Memory (HBM) and DDR memory. Both experience prohibitively long latencies, so we instead must focus on parallel BRAM accesses. Shuhai [6] finds HBM access to take 48+ cycles (106.7ns) and DDR4 22+ cycles (73.3ns) on the Alveo U280, both of which are non-deterministic due to possible bank conflicts and additional cycles needed for data refreshing, column/row switching, etc.

Assuming a 1 cycle read latency for BRAMs, D data-paths accessing a single memory unit would normally require D cycles to serve all of the data-paths. However, by independently interrogating each of the constituent N_{SM} BRAMs

TABLE I: Comparison of Memory Interconnects (N = N_{SM})

Interconnect	Links	Latency		Arbitration Logic
		Best	Worst	
Point-to-Point	$O(N^2)$	1	$O(N)$	Yes
Crossbar	$O(N^2)$	1	$O(N)$	Yes
Multi-Stage	$O(NlogN)$	$O(logN)$	$O(logN)$	No
Carousel (this work)	$O(N)$	1	$O(N)$	No

in parallel—each holding $\frac{1}{N_{\mathrm{SM}}}$th of the possible function results—it now takes D cycles in the worst case, and $\lceil \frac{D}{N_{\mathrm{SM}}} \rceil$ in the best. When the number of BRAMs is larger than the incoming number of data paths, it is even possible—with ideal data alignment—to serve all D data-paths in a single cycle.

Implementing arbitrary operations in memory reduces their latency to a single cycle. Combining the parallel needs of multiple incoming data-paths with the parallel accesses of many sub-memories presents opportunities to improve the performance of low latency designs.

To benefit from this, designs require one or more of the following: **1)** complex, multi-cycle functions; **2)** high levels of data precision, but lacking the latency budget for iterative solutions; **3)** high levels of parallelism, either for processing multiple, independent operations—like multi sensor robotics or monitoring systems [7] [8]—or a single operation comprising many parallel computations, like neural network inference's [9] non-linear activation functions.

III. Memory Configuration and Routing

A. Converting Functions to Memory Look-ups

A function is implemented in memory by pre-computing all possible outputs then storing them in FPGA memory. The concatenated function inputs serve as the address to read the result out of memory at runtime. Using 8-bit numbers of any format, a single input function such as $\sqrt{x}$ requires $2^8 = 256$ results to be stored in memory, while a two-input function, like addition, would require $2^{(8+8)} = 65536$ results to be stored.

Adapting a function with input width W_{ip}, output width W_{op}, and implemented using FPGA sub-memories of size S_{SM} bits, the number of sub-memories N_{SM} required to store this function is given by:

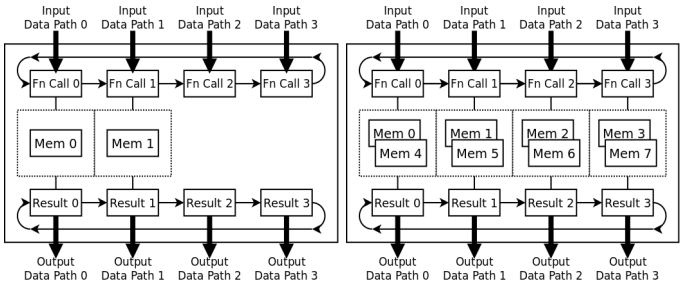
$$N_{\mathrm{SM}} = \left\lceil \frac{2^{W_{ip}} \times W_{op}}{S_{\mathrm{SM}}} \right\rceil \tag{1}$$

For example, a Xilinx BRAM holds 18k-bits—simplified as 2^{14} for this example—so a function with two 8-bit inputs and a 16-bit output needs: $N_{\mathrm{SM}} = \left\lceil \frac{2^{8+8} \times 16}{2^{14}} \right\rceil = 64$ sub-memories.

Additionally, as computation is now simply a memory look-up, we can support arbitrary data formats and precision. Floating point operations can return faithfully rounded results [10], responses to invalid inputs—such as NaNs and infinities—can be hard-coded, and discontinuous functions represented without additional control or comparison logic.

B. Sub Memory Interconnect

To minimise latency when accessing N_{SM} sub-memories, we build upon a uni-directional ring topology that we call a carousel architecture. Table I compares the most commonly used interconnects. Best case latencies occur when all D incoming reads are served immediately and independently by the $N = N_{\mathrm{SM}}$ sub-memories, while the worst occur when all D memory reads target the same sub-memory.



(a) 2 sub-memories processing 4 fn. calls (b) 8 sub-memories processing 4 fn. calls

Fig. 1: Examples of carousel architecture configurations

Our ring-based, carousel topology matches the best and worst case latencies of point-to-point and crossbar networks, but without the prohibitive scaling and routing requirements, or additional area needs. Multi-stage interconnects—e.g., Omega, Benes, butterfly networks, etc—can have lower latencies [11], but these never fall below $O(logN)$, even in the ideal case where incoming reads and sub-memories are perfectly aligned.

These interconnect latencies occur both when addressing memory and returning the result. As latency scales with the number of sub-memories, *not* the number of parallel data-paths, this added, minimum level of latency can be significant in practice. This is the primary motivation for not basing our architecture on traditional multi-stage interconnects.

IV. Carousel Architecture

Our proposed carousel architecture connects every parallel data-path, i.e., each call to the implemented function, to a set of sub-memories in one cycle then a different set of sub-memories in the next cycle. Data are rotated from one data-path to the next using two circular shift registers, one at the input and one at the output (Fig. 1).

We term this a *carousel* architecture, as memory and logic connections are fixed to specific input and output registers, and the data themselves rotate around them. This enables concurrent lookup in all sub-memories, maximizing parallelism while reducing latency. There is no fan-in, fan-out, or arbitration logic, and only a small amount of control logic for the two circular shift registers. This significantly reduces the logic and routing overhead traditionally seen when controlling parallel accesses to a shared resource.

With this uni-directional ring topology we provide additional "capacity" for parallelism. Additional sub-memories can be added and more data-paths can be served using fewer resources compared to the traditional approach of simply duplicating cores or compute modules to increase parallelism.

Fig. 1 gives example carousel architecture configurations, with $D = 4$ parallel data-paths sharing a memory-compute unit. Function calls not initially served by sub-memories, such as *Fn Call 2* and *Fn Call 3* in Fig. 1a, are rotated past these memories in future cycles. Where sub-memories outnumber calls, like Fig. 1b, multiple sub-memories serve each call. The maximum latency is then bounded by the number of data-paths D, rather than the number of sub-memories. In the extreme case, all sub-memories can be connected to a single data-path.

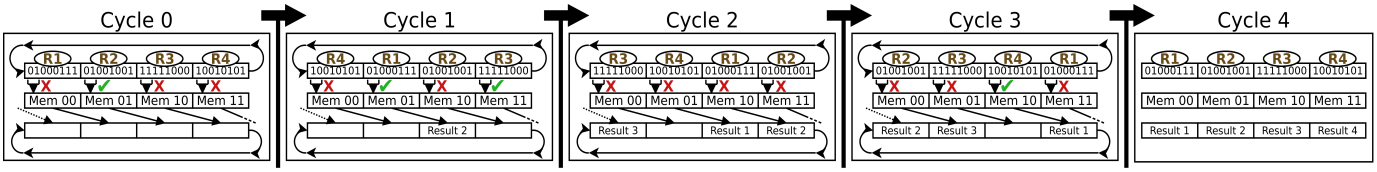


Fig. 2: An example of a carousel architecture processing four 8-bit inputs

An example memory-compute unit is given in Fig. 2, with $N_{\text{SM}} = 4$ sub-memories, and processing $D = 4$, 8-bit function calls. Each sub-memory holds the full result for one quarter of the possible inputs. Each cycle the sub-memories interrogate the data in their connected register. In cycle 0, the data in register *R2* targets that sub-memory (shown with a green tick) so the corresponding result is read out on the next cycle. Each cycle the function call data are shifted one place to the right, so all data can ultimately be presented to each sub-memory.

Cycles 1 and 2 respectively show multiple function calls, or none, can be served each cycle. The memory-compute unit outputs the final result to the requesting data-path as soon as it is available. When all function calls have been fulfilled, requiring a maximum of $D = 4$ cycles in this example, a new batch of D inputs are loaded into the carousel. This simplifies input routing, allows results to be output as soon as possible, and ensures the input pipelines remain synchronised. With careful pre-alignment of function call data with sub-memories, it is possible to serve $N_{\text{SM}} = 4$ function calls in a single cycle.

A. Latency

Having addressed the issues of increased signal routing and control logic brought about by increasing parallelism from both the number of data-paths and the BRAMs, we are left with determining the latency cost of such an architecture. For traditional, pipelined architectures accessing a shared resource, the latency to process D data-paths is:

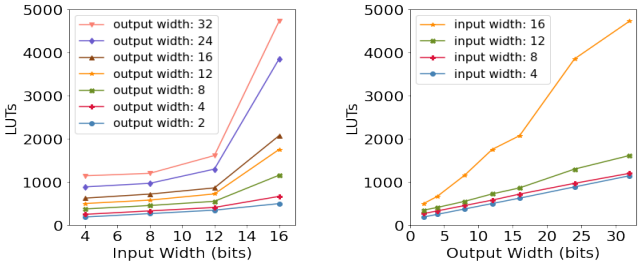
$$L_{\text{PIPE}} = D + N_{\text{ST}} \tag{2}$$

where N_{ST} is the number of pipeline stages. Long pipelines allow higher frequency designs and improve throughput, but can be detrimental to designs that: require low latencies; access a shared resource; or deal with non-streaming, irregular data.

With our carousel architecture, the worst-case latency is D cycles for D function calls. As we can serve multiple calls at once, the lower-bound latency for *any* degree of parallelism approaches the 1 cycle sub-memory access.

Ordering parallel function calls to align, or be more likely to align, with their targeted sub-memories means their results are present in the first sub-memories we check. To reduce latency, future work will explore reordering incoming function calls by analysing their distribution and access patterns.

Finally, the memory architecture itself plays a key role in improving latency and resource utilisation. Additional read ports for each sub-memory effectively increase the number of sub-memories. That the contents of this memory are already present in our architecture does not result in redundant accesses. When interrogating each sub-memory in turn, our step-size when moving data from one sub-memory to the next is then just equal to the number of available read ports, P. Thus,



(a) Varying Input Data Widths (b) Varying Output Data Widths

Fig. 3: Function input and output widths vs. LUT utilisation

the worst-case latency for our architecture when processing D parallel calls to a function is found to be:

$$L_{\text{MEM}} = \left\lceil \frac{D}{P} \right\rceil + 2 \tag{3}$$

The additional 2 cycles of latency are a result of architectural overhead (not present in the Fig. 2 example), and become an increasingly negligible factor at higher levels of parallelism.

V. Implementation & Results

We use the required function features identified in Section II—complexity, data precision, and/or parallelism—to assess our memory-based computation architecture by implementing a range of design configurations on the Xilinx Alveo U280 accelerator card. All designs operate at 300MHz, and are built using the Xilinx Vitis and Vivado 2019.2 design tools.

While we validate our design using Xilinx BRAMs, it is also applicable to Intel M10K and M20K [12] block memories, and extendable to any uniform-latency memory.

A. Resource Scaling, Parallelism and Data Precision

Fig. 3 plots resource use versus data width, and shows an inflection point at 12-bit inputs. Below this we use only a single BRAM to store function values, so cannot leverage our carousel architecture's parallel accesses. Functions with data widths less than this will see little or no resource reduction with our approach. 16-bit input and output functions do not require significantly more resources than 12-bit, and as a commonly used data width is the more practical choice.

Given the data width budget, the use of univariate functions is recommended to maximise data precision per function argument. Multivariate functions are supported as long as total data width is below our target limit, e.g., one 16-bit input is equivalent to two 8-bit inputs. In addition, as computation is simply a memory look-up, we are able to support arbitrary

TABLE II: Carousel Arch. Resources for any 16-bit Function

	Data-paths									
	2	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
LUTs	889	1152	1450	2072	3412	7231	11000	20506	38398	75080
Registers	2419	2749	3020	3639	4170	7296	11898	20279	37246	70487
18K BRAMs	64	64	64	64	64	64	64	64	64	64

$LUTs = 72.3264D + 1321.0198 \quad P < 0.001$

$REGs = 66.6371D + 2686.3473 \quad P < 0.001$

TABLE III: Our Approach vs. Common FPGA Functions in Past Works

		Comp. Intensity	Width (bits)	Per Function Call					Platform	Process	Latency Speed up
				LUTs	DSPs	Memory (kbits)	Cycles	Latency			
sqrt	CORDIC [13]	4.0	16	247	0	0	16@330MHz	48.48ns	Virtex-US+	16nm	**29.0x**
sin, cos	CORDIC [14]	3.5	16	697	0	0	14@421MHz	33.32ns	Cyclone II 2C70	90nm	**20.0x**
	CORDIC [13]	5.0	16	995	0	0	20@720MHz	27.78ns	Virtex-US+	16nm	**16.6x**
	FloPoCo [15]	1.0	16	297	5	0	4@298MHz	13.42ns	Virtex-5	65nm	**8.0x**
	CORDIC [15]	1.0	16	713	2	0	4@238MHz	16.81ns	Virtex-5	65nm	**10.1x**
	CORDIC [15]	3.3	16	719	2	0	13@368MHz	35.33ns	Virtex-5	65nm	**21.2x**
sinh, cosh	CORDIC [13]	5.0	16	1143	0	0	20@720MHz	27.78ns	Virtex-US+	16nm	**16.6x**
tanh	Fn. Approximation [16]	1.5	13	137	0	-	5@436MHz	11.47ns	Virtex-4	90nm	**6.9x**
	DCT Interpolation [17]	0.5	16	129	0	1250	2@264MHz	7.60ns	Virtex-7	28nm	**4.6x**
	CORDIC [13]	5.3	16	550	0	0	21@720MHz	29.17ns	Virtex-US+	16nm	**17.5x**
atan2	Taylor [18]	1.3	16	329	6	72	5@220MHz	22.73ns	Virtex-6	40nm	**13.6x**
	Taylor [18]	1.3	16	331	6	72	1@135MHz	7.41ns	Virtex-6	40nm	**4.4x**
	CORDIC [18]	2.0	16	796	0	0	8@389MHz	20.57ns	Virtex-6	40nm	**12.3x**
	CORDIC [18]	2.0	16	816	0	0	2@191MHz	10.47ns	Virtex-6	40nm	**6.3x**
Gaussian	FloPoCo [19]	2.8	16	770	0	32	11@300MHz	36.67ns	Virtex-US+	16nm	**22.0x**
any function	**This Work**	**0.25**	**16**	**81**	**0**	**1152**	**0.5@300MHz**	**1.67ns**	**Virtex-US+**	**16nm**	

data formats and precision. INT8—for 2-input functions—and 16-bit are common formats in GPU processing and machine learning [20] [21], and Gupta [9] finds 16-bits sufficient for training some deep neural networks.

As the most practical choice, we now focus on 16-bit designs. When moving from supporting 2 to 1024 parallel function calls, Table II shows the required LUTs increase from 889 to 75080. A 512x increase in parallelism uses only 85x the look-up tables, while the number of required 18K BRAMs remains constant. The higher the degree of required parallelism, the more resource efficient our implementation becomes while still maintaining a 300MHz operating frequency.

B. Latency Reduction of Complex Functions

Following a generalised roofline model for FPGAs [22], we determine complexity using computational intensity, i.e., the number of operations per byte: the more operations a function performs on each datum, the more cycles it is likely to require. Our architecture offers a uniform computational intensity of 0.25 ops/byte for any 16-bit function, doubling to 0.5 ops/byte for 8-bit functions. This sets the minimum boundary for performance improvement.

Our approach is function agnostic, but simple functions like integer arithmetic (0.17 ops/byte) or more "fundamental" operations supported by DSPs—like multiply-accumulate (0.33 ops/byte)—see little benefit from memory-based computing in general. We instead target functions with higher computational intensity, such as sqrt, sinh, cosh, tanh, or any user-defined, multi-cycle function. These will see their computational intensity reduced, and latency lowered to just 1 cycle.

Table III compares past works to a parallel, 16-bit implementation of our memory-compute architecture on the Xilinx Alveo U280. Assessed against interpolation [17], approximation [16], CORDIC [14] [15] [18] and FloPoCo [15] [18] [19], we see that it is difficult for other function implementations to match the 1-cycle latency of memory look-ups.

The improvements with our approach do not come just from the inherent benefit of memory. A lowered computational intensity makes functions more sensitive to I/O bandwidth limitations, so our choice of a simplified carousel interconnect, described in Section III-B, ensures we maintain a high 300MHz operating frequency up to 1024 parallel data-paths. This is the key benefit to our approach: a single cycle latency and a high, sustained 300MHz operating frequency at higher degrees of parallelism for *any* 16-bit function. Additionally, by using 2 read ports per BRAM we are able to lower our average latency further, to just 0.5 cycles per function call (Eqn. 3).

We achieve up to 29x latency reduction, as well as a 37-92% reduction in LUTs per function call using our architecture. The linear regression results from Table II show this continues to scale favourably to higher degrees of parallelism. Memory use can also be improved. Abdelsalam et al. [17] uses 1250-kbits of memory to store approximations for a single tanh function, while we use 1152-kbits to support the entire function. Their memory use can be reduced at the cost of data precision.

VI. Conclusion and Future Work

We present a novel approach for highly parallel function evaluation of computationally-intensive, low latency designs. Using pre-computed values and memory-based look-ups, we use a new, ring-based topology to provide independent access to the individual BRAMs that are traditionally grouped together and accessed as a single, monolithic memory unit.

Our interconnect requires little logic and routing, scaling linearly with both higher degrees of parallelism, i.e., more calls to the implemented function, and the number of BRAMs accessed in parallel. When assessed against square root, Gaussian, trigonometric and hyperbolic functions, we see a 37-92% reduction in LUTs per parallel function call on the Xilinx Alveo U280, and a 4.4x to 29x latency speed up.

For *any* 16-bit function, a single memory-compute core can serve 1024 parallel calls to its implemented function at 300MHz, averaging 0.5 cycles per function call.

Future work will focus on further lowering latency based on the data distribution and memory access patterns of function calls. Power consumption will be examined to better quantify our benefits versus dedicated cores and co-processors. We will extend our FPGA architecture to more favourably support 18-bit functions, aligning us with a DSP block's lower data width operations. This will make it easier to transition existing designs to our architecture, and lower the level of computational complexity required to benefit from our approach.

Acknowledgements
We gratefully acknowledge the support of Intel, Xilinx, XACC at ETH Zürich, and the U.K. EPSRC (EP/N031768/1, EP/P010040/1, EP/S030069/1).

REFERENCES

[1] Xilinx, CORDIC v6.0 LogiCORE IP Product Guide, 2021. https://www.xilinx.com/support/documentation/ip_documentation/cordic/v6_0/pg105-cordic.pdf

[2] Xilinx, Floating-Point Operator v7.1 LogiCORE IP Product Guide, 2020. https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_1/pg060-floating-point.pdf (accessed Mar. 01, 2021).

[3] M. Rawski, Modified Distributed Arithmetic Concept for Implementations Targeted at Heterogeneous FPGAs, International Journal of Electronics and Telecommunications, vol. 56, pp. 345-350, 2010, doi: 10.2478/v10177-010-0045-9.

[4] S. Xu, S. A. Fahmy, and I. V. McLoughlin, Square-rich fixed point polynomial evaluation on FPGAs, in Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays, New York, NY, USA, Feb. 2014, pp. 99-108, doi: 10.1145/2554688.2554779.

[5] J. Detrey and F. de Dinechin, Table-based polynomials for fast hardware function evaluation, in 2005 IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP05), Jul. 2005, pp. 328-333. doi: 10.1109/ASAP.2005.61.

[6] Z. Wang, H. Huang, J. Zhang, and G. Alonso, Shuhai: Benchmarking High Bandwidth Memory On FPGAS, in 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), May 2020, pp. 111-119. doi: 10.1109/FCCM48280.2020.00024.

[7] F. Cabello, J. Len, Y. Iano, and R. Arthur, Implementation of a fixed-point 2D Gaussian Filter for Image Processing based on FPGA, in 2015 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA), Sep. 2015, pp. 28-33. doi: 10.1109/SPA.2015.7365108.

[8] D. G. Lowe, Object recognition from local scale-invariant features, in Proceedings of the Seventh IEEE International Conference on Computer Vision, Sep. 1999, vol. 2, pp. 1150-1157 vol.2. doi: 10.1109/ICCV.1999.790410.

[9] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, Deep Learning with Limited Numerical Precision, in Proceedings of the 32nd International Conference on Machine Learning, Lille, France, Jul. 2015, vol. 37, pp. 1737-1746. [Online]. Available: http://proceedings.mlr.press/v37/gupta15.html

[10] M. Lange and S. M. Rump, Faithfully Rounded Floating-point Computations, ACM Trans. Math. Softw., vol. 46, no. 3, p. 21:1-21:20, Jul. 2020, doi: 10.1145/3290955.

[11] W. J. Dally and B. P. Towles, Principles and Practices of Interconnection Networks. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[12] Intel Stratix 10 Embedded Memory User Guide. Accessed: Mar. 01, 2021. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/ug-s10-memory.pdf

[13] Xilinx, Performance and Resource Utilization for CORDIC v6.0. https://www.xilinx.com/html_docs/ip_docs/pru_files/cordic.html (accessed May. 28, 2021).

[14] A. P. Renardy, N. Ahmadi, A. A. Fadila, N. Shidqi, and T. Adiono, FPGA implementation of CORDIC algorithms for sine and cosine generator, in 2015 International Conference on Electrical Engineering and Informatics (ICEEI), Aug. 2015, pp. 1-6. doi: 10.1109/ICEEI.2015.7352460.

[15] F. de Dinechin, M. Istoan, and G. Sergent, Fixed-point trigonometric functions on FPGAs, SIGARCH Comput. Archit. News, vol. 41, no. 5, pp. 83-88, Jun. 2014, doi: 10.1145/2641361.2641375.

[16] S. Gomar, M. Mirhassani, and M. Ahmadi, Precise digital implementations of hyperbolic tanh and sigmoid function, in 2016 50th Asilomar Conference on Signals, Systems and Computers, 2016, pp. 1586-1589, doi: 10.1109/ACSSC.2016.7869646.

[17] A. M. Abdelsalam, J. M. P. Langlois, and F. Cheriet, A Configurable FPGA Implementation of the Tanh Function Using DCT Interpolation, in 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Apr. 2017, pp. 168-171, doi: 10.1109/FCCM.2017.12.

[18] F. D. Dinechin and M. Istoan, Hardware Implementations of Fixed-Point Atan2, in 2015 IEEE 22nd Symposium on Computer Arithmetic, Jun. 2015, pp. 34-41, doi: 10.1109/ARITH.2015.23.

[19] F. de Dinechin, Reflections on 10 Years of FloPoCo, in 2019 IEEE 26th Symposium on Computer Arithmetic (ARITH), Jun. 2019, pp. 187-189. doi: 10.1109/ARITH.2019.00042.

[20] M. Langhammer, B. Pasca, G. Baeckler, and S. Gribok, Extracting INT8 Multipliers from INT18 Multipliers, in 2019 29th International Conference on Field Programmable Logic and Applications (FPL), Sep. 2019, pp. 114-120. doi: 10.1109/FPL.2019.00027.

[21] F. Zhu et al., Towards Unified INT8 Training for Convolutional Neural Network, in 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Jun. 2020, pp. 1966-1976. doi: 10.1109/CVPR42600.2020.00204.

[22] B. da Silva, A. Braeken, E. H. D'Hollander, and A. Touhafi, Performance Modeling for FPGAs: Extending the Roofline Model with High-Level Synthesis Tools, International Journal of Reconfigurable Computing, vol. 2013, pp. 1-10, Dec. 2013, doi: 10.1155/2013/428078.