Kaunas University of Technology

Faculty of Electrical and Electronics Engineering

Research on Machine Learning Algorithm Acceleration Using FPGAs

Master's Final Degree Project

Electronics Engineering (6211EX012)

Lukas Stasytis

Project author

Prof. Žilvinas Nakutis

Supervisor

Kaunas, 2022

Kaunas University of Technology

Faculty of Electrical and Electronics Engineering

Research on Machine Learning Algorithm Acceleration Using FPGAs

Master's Final Degree Project

Electronics Engineering (6211EX012)

Lukas Stasytis

Project author

Prof. Žilvinas Nakutis

Supervisor

Prof. Darius Gailius

Reviewer

Kaunas, 2022

Kaunas University of Technology

Faculty of Electrical and Electronics Engineering

Lukas Stasytis

Research on Machine Learning Algorithm Acceleration Using FPGAs

Declaration of Academic Integrity

I confirm the following:

1. I have prepared the final degree project independently and honestly without any violations of the copyrights or other rights of others, following the provisions of the Law on Copyrights and Related Rights of the Republic of Lithuania, the Regulations on the Management and Transfer of Intellectual Property of Kaunas University of Technology (hereinafter – University) and the ethical requirements stipulated by the Code of Academic Ethics of the University;

2. All the data and research results provided in the final degree project are correct and obtained legally; none of the parts of this project are plagiarised from any printed or electronic sources; all the quotations and references provided in the text of the final degree project are indicated in the list of references;

3. I have not paid anyone any monetary funds for the final degree project or the parts thereof unless required by the law;

4. I understand that in the case of any discovery of the fact of dishonesty or violation of any rights of others, the academic penalties will be imposed on me under the procedure applied at the University; I will be expelled from the University and my final degree project can be submitted to the Office of the Ombudsperson for Academic Ethics and Procedures in the examination of a possible violation of academic ethics.

Lukas Stasytis

Confirmed electronically

Lukas Stasytis. Research on Machine Learning Algorithm Acceleration Using FPGAs. Master's Final Degree Project / supervisor Prof. Žilvinas Nakutis; Faculty of Electrical and Electronics Engineering, Kaunas University of Technology.

Study field and area (study field group): Electronics Engineering, Engineering Science.

Keywords: machine learning, FPGA, CORDIC, accelerator, SVD

Kaunas, 2022. 31 p.

Summary

With recent advances in machine learning (ML) and a rapidly accelerating new frontier of digital logic design tooling for Field-Programmable Gate Array (FPGA) development, new opportunities exist for exploring the use of FPGAs in the ML field. In this work, the current state of the art digital logic design tool-chains and their use in the field of ML is explored and a design methodology is proposed for efficiently utilizing FPGAs as accelerators for compute-intense ML tasks. An FPGA prototype design for accelerating the Singular Value Decomposition (SVD) algorithm as well as the QR decomposition using the Coordinate Rotational Computer (CORDIC) as an abstraction layer arithmetic core is presented and analyzed with emphasis on DSP utilization.

Lukas Stasytis. Mašininio mokymosi algoritmų spartinimo su programuojama logika tyrimas. Magistro baigiamasis projektas / vadovas Prof. Žilvinas Nakutis; Kauno technologijos universitetas, Elektros ir elektronikos fakultetas.

Studijų kryptis ir sritis (studijų krypčių grupė): Elektronikos inžinerija, inžinerijos mokslai.

Reikšminiai žodžiai: mašininis mokymasis, programuojama logika, koordinačių posūkio kompiuteris (CORDIC), spartintuvas, singuliarioji matricų dekompozicija (SVD)

Kaunas, 2022. 31 p.

Santrauka

Atsižvelgiant į naujausius pasiekimus mašininio mokymosi (ML) srityje bei sparčiai besiplečiančią modernių skaitmeninės logikos projektavimo įrankių aibę programuojamos logikos (FPGA) projektavimui, atsiveria naujos galimybės FPGA panaudojimui ML srityje. Šiame darbe išnagrinėjami moderniausi skaitmeninės logikos projektavimo įrankiai bei jų panaudojimas ML srityje ir pristatoma projektavimo metodika, leidžianti efektyviai išnaudoti FPGA kaip didelius skaičiavimo kiekius turinčių ML užduočių spartintuvą. Šiame darbe taip pat pateikiamas ir išanalizuojamas FPGA spartintuvo prototipas, skirtas pagreitinti Singuliarioji matricų dekompozicija (SVD) bei QR dekomponavimo algoritmus, pritaikant Koordinačių Posūkio Skaitmeninį Kompiuterį (CORDIC) kaip abstrakcijos sluoksnį lokalių signalų apdorojimo procesorių panaudojimui.

Table of contents

List of figures

List of abbreviations and terms

Abbreviations:

FPGA. – Field Programmable Gate Array

ML. – Machine Learning

CORDIC – Coordinate Rotation Digital Computer

SVD – Singular Value Decomposition

DL – Deep Learning

DSP – Digital Signal Processor/Processing

GPU – Graphics Processing Unit

IoT – Internet of Things

CPU – Central Processing Unit

ASIC – Application Specific Integrated Circuit

RTL – Register Transfer Level

MUSIC – Multiple Signal Classification

DOA – Direction of Arrival

HLS – High Level Synthesis

HDL – Hardware Description Language

DMA – Direct Memory Access

Introduction

Over the last decade, there has been a massive increase in the use of ML & Deep Learning (DL) algorithms [1], both at the datacenter level as well as the internet of things (IoT). Accelerated by the growing computational capacity of Graphics Processing Units (GPUs), ML is now used in a wide variety of fields from signal processing to natural language generation. In the field of digital signal processing, adaptable filters are used for noise cancellation [2], audio detection as well as classification [3]. In the field of image processing, linear regression-based algorithms are used for classification tasks, such as face recognition [4], autonomous driving [5] and defect detection in factories [6]. Natural Language Processing is used for translation, sentiment analysis [7] and even generating new text [8]. The field is incredibly wide and many of the tasks have unique requirements such as achieving low power, low latency or high bandwidth. FPGAs are especially suited for problems having unique requirements such as these. By adapting hardware to unique problem sets, overheads can be eliminated which exist in conventional processing units such as CPUs and GPUs. Additionally, while an optimized processing unit could be turned into an application-specific integrated circuit (ASIC), FPGAs fill the niche of design, allowing for more rapid prototyping and exploration of efficient hardware architectures. Additionally, FPGAs feature reconfigurability – the ability to change the design at runtime. Many studies have been conducted on the use of FPGAs in these fields showing great result such as in [9], [10], and [11]. In this work, lucrative use cases for FPGA acceleration are outlined in detail, with focus on matrix algorithms and how they may be transformed to better fit FPGA acceleration.

In recent years, there has also been an explosion of RTL design tools. Given the slow nature of RTL development using conventional languages like Verilog, VHDL and SystemVerilog and the growing complexity of designs, a necessity for raising productivity in hardware development has arisen. It can take months to years to design accelerators using Verilog, while a software implementation of an identical solution from an algorithm perspective can often be achieved magnitudes of time faster. Software has advanced significantly in improving work-flow, with rising popularity of functional and object-oriented programming. The second most popular programming language in the world as of this writing is Python [12], a highly abstract multi-paradigm language. Given the vast amount of redundancy in designing low-level RTL, the gap could be bridged, adapting software-based design methodologies for hardware development. In this work, high-level language libraries and tools are explored which allow rapid development of RTL with special focus put on the Migen Python library, which allows Verilog design modeling using Python object-oriented constructs.

Lastly, an algorithm that has not seen much attention in recent years, yet shows vast potential as a standardized arithmetic unit in FPGAs is the Coordinate Rotational Computer (CORDIC) [13]. This algorithm can be used for computing many of the standard arithmetic operations necessary in ML algorithms, while having great malleability in regards to design trade-offs and can be used as a powerful abstraction layer for FPGA DSP cores. This work outlines a prototype implementation, utilizing CORDIC for solving the underlying arithmetic operations of the Singular Value Decomposition (SVD)[14] as well as QR factorization using Givens Rotations.

1. Overview

The following overview sections present current trends in the use of ML in the context of FPGAs, focusing on data center and edge computing use cases as two separate domains (1.1). Numerous software libraries in high level programming languages are overviewed as potential acceleration targets (1.2). In section (1.3), a set of design tools for hardware development with FPGAs are outlined. Numerous algorithms which are of great interest for acceleration are covered in (1.4). Lastly, the CORDIC algorithm is explored in (1.5) as a potential architecture for designing FPGA accelerators for the aforementioned algorithms of previous sections with a specialized variant also reviewed. Section (1.7) summarizes the literature review, states the hypothesis and the primary tasks of this work.

1.1. Fields of application

1.1.1. Datacenters

Most computational tasks can be split into two domains in which they are executed: server tasks and client tasks.

Datacenters have made heavy use of FPGA acceleration in recent years for a wide array of problems. [15] is a framework for hybrid CPU-FPGA databases. [16] explores Microsoft's use of FPGAs in their datacenters to serve deep neural networks at scale. Meanwhile, [17] explores data processing pipeline acceleration with FPGAs. Additionally, Intel has made large strides in the datacenter field with Intel FPGA SDK for OpenCL, which has already been reviewed in a number of published works, [18] examines relational query processing, [19] examines spatial-spectral classification and [20] explores single-precision floating point vector addition kernels. From the literature, a strong set of use cases can be gathered for FPGAs in datacenters for big data tasks.

One of the most computationally intensive task in the field of ML when looking at big data is image processing [21]. While this field is dominated by GPUs, there exists a portion of imagine processing that is often left for CPUs – preprocessing. Before an image can be loaded into a deep learning engine, it needs to pass a wide range of preprocessing steps, such as: reshaping, resizing, cropping, recoloring, normalization, whitening and so on. These steps, while majorly consisting of basic matrix multiplication, have enough uniqueness to their algorithms where a specialized solution could be used. In [22] the authors explore DL image preprocessing with FPGAs in a cloud setting. In the outlined research, *1.35x* to *2.4x* image preprocessing throughput was achieved while using only 1/10 of the CPU cores and reaching 1/3 the latency when performing inference. Furthermore, the authors noted the bottlenecks that develop when using CPUs and GPUs for preprocessing when tackling tasks such as image cropping, resizing and rotation which measure as high as a *30* percent performance degradation. In this work, the Singular Value Decomposition (SVD) and QR Decomposition algorithms are tackled as an exploration of image processing acceleration using FPGAs.

1.1.2. Edge Computing

From the field of DSP, the Multiple Signal Classification (MUSIC) algorithm continues to be widely used for Direction of Arrival (DOA) estimation, [23] while having a relatively simple implementation. There has been work done in optimizing different aspects of spectrogram-based DOA algorithms, which could be split into a.) steering matrix calculation, b.) spectrogram generation

and c.) peak finding. Notably, [24] explores the use of the Particle Swarm Optimization algorithm for peak finding, while [25] explores novel algorithms for steering matrix calculation with arbitrary 3D microphone array layouts. At the same time, [26] is a dataset for exploring DOA using a drone with 8 mounted microphones in a closed room. This dataset can be used to explore FPGA implementations of DOA algorithms, given that FPGAs match the requirements of mobile, small form factor devices such as drones. Some examples of such requirements being: latency, energy efficiency and hardware utilization. Additionally, spectrogram generation is often done using Eigenvalue Decomposition [27] [28] (EVD-MUSIC) To this end, the Coordinate Rotation Digital Computer (CORDIC) [29] [30] is an algorithm of significant note, allowing to implement trigonometric functions via the use of vector rotations, using only addition and bit shifting operations, bypassing the need for multiplication and division units. There is little research done on the use of CORDIC for DOA.

1.2. Software computational libraries

In an effort to explore which types of libraries are commonly used for data center workloads, popular cloud providers are looked at.

Google has multiple articles on deploying Scikit models at scale [31] and cite libraries for improving parallelism written in Python. [32] A strong push for the use of Tensorflow in regards to ML tasks is also noted. [33]. [34] outlines tradeoffs of using FPGAs for accelerating Tensorflow with promising results. Meanwhile, Amazon presents tools such as Sagemaker as Python libraries and urge the use of Scikit for data analytic tasks. [35] Additionally, the majority of Kaggle competition problems are indeed solved using Python. [36] Thus, Python is looked at as the language that is commonly used by consumers for their data analytics tasks and from there three libraries stand at the forefront: Tensorflow[37], Scikit [38] and Numpy [39].

From looking at varying workloads, it can be seen that while Tensorflow can be used for general computing tasks, it is primarily used for building machine learning models, while preprocessing steps are left to be done in Scikit or Numpy [40]. Additionally, looking further into Scikit, it can be noted that the primary functions wrap Numpy calls for the computations. Scikit simply acts as a higher abstraction layer for implementing specific algorithms. Numerous attempts have been made to accelerate Numpy with GPUs to great success, showing further promise in the pursuit. For example, [41] accelerated Numpy operations with the CUDA framework. [42] Explored efficient numerical computation in Python using GPUs.

Additionally, looking at digital signal processing libraries, a notable example is [43] in which Numpy is heavily utilized to solve beamforming and eigen decomposition related tasks for sound signal processing and direction of arrival estimation.

Thus, it is concluded that in order to accelerate the outlined image and digital signal processing algorithms, Numpy is a strong candidate library to accelerate. This would also allow the employment of this type of acceleration for a multitude of other computational tasks which use Numpy underneath.

In addition, further push to use Python as 'glue logic' to write performance-sensitive applications exists in regards to message passing. The Reactor model [77] is a newly proposed communication model for real time systems. The model works by the same principle as the actor model, but introduces

logical time, allowing to control determinism of the system. The approach is highly promising for encapsulating hardware accelerators and providing deterministic data movement while applying partial reconfiguration on the FPGA fabric. Notably, the Reactor model borrows many ideas from RTL design, with basic communicating elements (reactors) having the same input output module structure as Verilog/VHDL modules. This makes encapsulation of an FPGA accelerator much more straightforward and allows to greatly utilize the determinism which can be obtained from a hardware design and retain it in the software layer.

1.3. High-level RTL design tools

Commercially, the leading vendors of FPGAs are Xilinx, Intel, National Instruments and LatticeSemi, each hosting their own Electronic Design Automation (EDA) toolchains. Vivado, Vivado HLS [44] and Vitis all being juggernaut toolchains for each step of RTL design in the case of Xilinx. Meanwhile "Intel SDK for OpenCL" as well as Quartus Prime [45] are the toolchains used by Intel. The NI LabVIEW [46] and Lattice Diamond [47] are notable toolchain designed by National Instruments and LatticeSemi respectively. Synpsys, MentorGraphics and Cadence are also notable EDA suppliers, working on verification, Place and Route, simulation and HLS.[48] [49], [50].

Looking further at Xilinx, they provide two methods for increasing the abstraction level of RTL design – block diagram design and Vivado HLS. In the case of block diagrams, the user can utilize Xilinx's repository of IP cores (or add their own) and connect them using a user interface. This provides plenty of convenience for rapidly designing accelerators for standard FPGA problem sets. They are, however, limited to standard interfaces provided by Xilinx unless the user implements their own interface wrapper cores. In the case of HLS, users can write C/C++ code which is then converted into Verilog or VHDL. This is a large design system, focused on allowing the user to specify hardware functionality rather than design. The tooling then infers the necessary hardware constructs. Heavy use of pragmas is employed for specifying hardware generation specifics for different portions of C code. For example, users would write loops and then specify a pragma to unwrap said loops. This, in turn, converts the loops into a series of parallel processing units in hardware.

Additionally, Xilinx recently introduced the Versal Architecture[93], which features an ‚all in one' package of both deterministic real-time and high-performance processors, AI cores, DSP cores, expansive interconnect and dense FPGA cells. The solution is marketed for adaptability and necessitates rethinking hardware design further to fully utilize local processors of varying forms to solve different tasks. The architecture is designed for using a combination of C++ HLS, Matlab graphs and packaged RTL source files.

Other notable HLS-style solutions are Bluespec [51], Catapult C [52], Matlab HDL Coder [53] and Status HLS (by Cadence) [54] among many others. However, the vast majority of HLS tools input C code and work in a similar fashion to Vivado HLS. Focusing on problem statements and pragmas. The authors have not seen any notable higher level language (such as Python) HLS solutions. Multiple papers have been written on the prospect of HLS in current FPGA design. [55] is a notable open-source tool suite for binding Python to HLS-suitable C functions and generating wrappers for ease of accelerator development. [56] further surveys the HLS tool landscape in full.

Other solutions in the RTL design field are based around new hardware description languages (HDLs). All of these solutions are based around defining hardware systems in high-level languages such as Python or Scala which are then converted into Verilog, VHDL or SystemVerilog. (In the vast

majority of cases – Verilog). To further emphasize the difference in approach, HLS-style solutions are focused on modeling algorithms and giving compiler hints for how they should be implemented in hardware, while high-level HDL solutions are used to directly model RTL, but with additional high-level constructs to ease design.

Two of the most notable solutions are Chisel [57] and Migen[58]. Chisel is a Scala-based HDL, allowing the designer to write hardware designs using functional and object oriented programming patterns in the Scala programming language which are thus converted into Verilog for hardware implementation. Migen, on the other hand, is a Python programming language solution, focused entirely on object-oriented design patterns, allowing the designer to define Verilog modules as Python class objects.

Both of these solutions are under active development and have garnered significant attention from the hardware community. Migen in particular, has sparked numerous side projects, such as Litex [59],[60], which is a high level Python library for designing systems on chips (SoCs). There now exist standard Python DSP libraries which allow converting FIR, NIR and similar filters into Verilog by using Migen as an intermediary [61]. All of these solutions have already been used for design of shipped hardware products.

In this work, Migen is chosen as the key design solution for modeling a CORDIC accelerator, interleaving Verilog-based designs where necessary.

1.3.1. Vivado Block Diagram design

The Vivado Block Diagram is Xilinx's Graphical UI-based hardware development methodology. Individual hardware modules are packaged as IP cores, which can then be moved around in a graphical interface, with pin connections being possible to add by hand or using automation tools. This presents a clear and hierarchical picture of the design and can help increase engineer productivity. Figure 1 features an example block diagram. A standard ZYNQ design using the block diagram would feature a processing system IP core, which connects the Programmable Logic (PL) and Processing System (PS), with additional compute units being connected to the PS. A full application meeting generic IP core specifications can be made using just the block diagram, without having to look at any HDL code. In the example presented in Figure 1, a CORDIC core has been connected to an AXI stream, which, in turn, is connected to the PS. After a hardware design bitstream has been generated, it would be possible to directly call the CORDIC function from software running on the device CPU. The necessity to use standard interfaces and Xilinx's IP being black boxes, however, are the main downsides, which can severely limit the potential scope of applications the design methodology can be used on. While any application is possible to implement in block diagrams by using low level primitives for operations, it would severely lack design malleability, which is one of the focuses of this work.

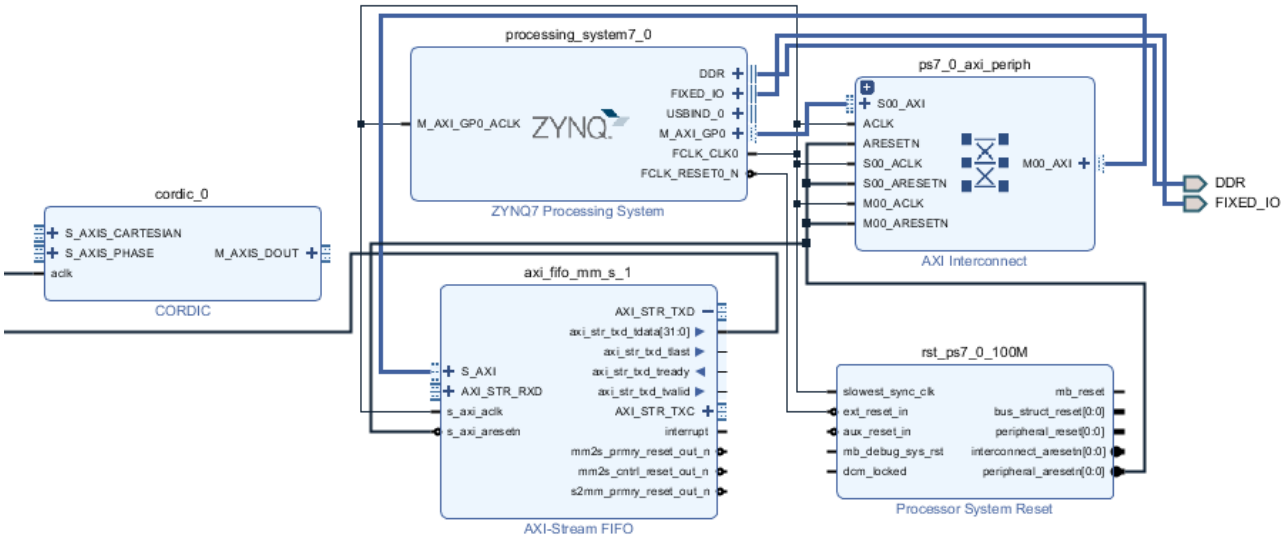


Fig. 1. Example Vivado block diagram design. Features such as the Processing System and acceleration cores can be added at will

1.3.2. ZYNQ

The ZYNQ architecture [81] is Xilinx's heterogenous computing solution for FPGAs. The general outline is to host a hard CPU on the same device as the FPGA fabric and present a large software stack allowing to interface FPGA programmable logic registers from software constructs, such as C variables, while at the same time, allowing Direct Access Memory (DMA) from either direction. The main two parts of the chip having this interface being the PL and PS. Developers create hardware designs and load them onto the PL via the PS using bitstreams and then interface with the PL design from PS. The main benefit of a ZYNQ architecture is the severely decreased latency between the PL and SDRAM, given that both are featured on the chip, while a conventional homogeneous solution would have to interface the PL with external memory in the host device via an interface such as Ethernet or PCIe, which are substantially slower than DMA.

The ZYNQ architecture, however, presents an overhead in the data management interface between the PL and PS. Exploration into how efficient this interface really is can be a target of research.

In this work, a PYNQ board-based prototype is developed for the SVD and QR algorithms. The Xilinx DMA core is used to this end in conjunction with the processing system core.

1.4. Often used and demanding algorithms in Machine Learning

1.4.1. EVD-MUSIC

Eigenvalue Decomposition-based MUSIC is an algorithm designed for spectrogram generation using a steering vector as well as a sorted and filtered out eigenvector matrix. The combination of the two can generate spectrograms which accurately pinpoint the direction of arrival of incoming signals. To this end, EVD-MUSIC is widely used for DOA tasks. [85], [86] including designs implemented via FPGAs [87]. Notable is the use of sine transforms for the beamforming portion of EVD-MUSIC, which could be prone to acceleration using CORDIC.

1.4.2. Particle Swarm Optimization

The Particle Swarm Optimization algorithm [82] is a nature-inspired optimizer based on modeling a flock of birds. Using a series of particles, each having their own velocity, position and search-space, as well as having a connected global bias depending on the positions of other particles, it is possible to find minimum points of a multi-dimensional plane. PSO is of great interest in the field of ML given how it can be used for peak finding with a minimal, yet extremely wide search field. The algorithm is highly parallelizable, suggesting that a unified FPGA solution could be of great interest.

1.4.3. Principle Component Analysis

The Principle Component Analysis algorithm [83] is used in tandem with the Singular Value Decomposition (SVD) to capture the primary features of a dataset. In addition, PCA can be used to further augment individual samples of a dataset by applying a filtering which is based on the dataset as a whole. This can be used for tasks such as dimensionality reduction. [84]. PCA is generally computed using the SVD and thus matrix decompositions are of key interest as the underlying algorithms.

1.4.4. Matrix Decompositions

Matrix Decompositions are of key interest in the field ML due to their abundant use cases. They may be utilized to achieve dimensionality reduction with methods such as the PCA, solve linear systems of equations, capture eigenvalues.[89] Two decomposition methods stand out as being of particular interest in the ML field: SVD and QR. Using the SVD, one can begin the Principle Component Analysis scheme [63] to obtain the primary components of a matrix. This has a wide variety of uses in image processing applications in regards to image augmentation, feature extraction as well as compression. The QR decomposition, on the other hand, can be used for linear problem solving, such as least-squares using Gaussian Elimination [88] as well as an intermediate step for SVD computation.
The SVD is expressed as the matrix product $A = UWV^T$, where *U* and *V* are orthogonal matrices and *W* is a diagonal matrix of singular values. The terms *U*, *V* and *W* can be used to reconstruct the original matrix with *W* value truncation leading to dimensionality reduction.
The QR factorization scheme, on the other hand, is expressed as the matrix product $A = QR$, where Q is an orthogonal matrix and *R* is an upper triangular matrix. Each of sizes *n*x*n* for an input matrix of size *n*x*n* for both decompositions.

The SVD and QR decompositions can both be computed using a wide variety of methods, in many cases the same ones, such as the one-sided or two-sided Jacobi matrix method [64] more widely known as Jacobi-SVD or Jacobi-QR, Householder reflections[91], Gram-Schmidt [90], Givens rotations [65] and many more. Of note is the method of utilizing QR for solving the SVD: the Kogbetliantz method[78].

The method explored in this work is the utilization of Givens Rotations [65] for implementing both, the SVD and the QR decompositions utilizing the Kogbetliantz method for the SVD and shall be explored in the following section.

1.4.5. Givens Rotations for Matrix Decomposition

The Givens Rotation [65] approach for computing matrix decompositions functions by sequentially eliminating matrix elements from the original matrix, while adjusting the row or column values in the original matrix of the respective eliminated element to achieve a vector rotation. Figure 2 illustrates the process. Each iteration, to eliminate element a_{ij} from the matrix, a 2x2 matrix, known as as Givens Rotation is computed using $a_{ij}, a_{i-1j}, a_{i-1j}, a_{i-1j-1}$ elements by solving a system of equations as per Equations 1-12.

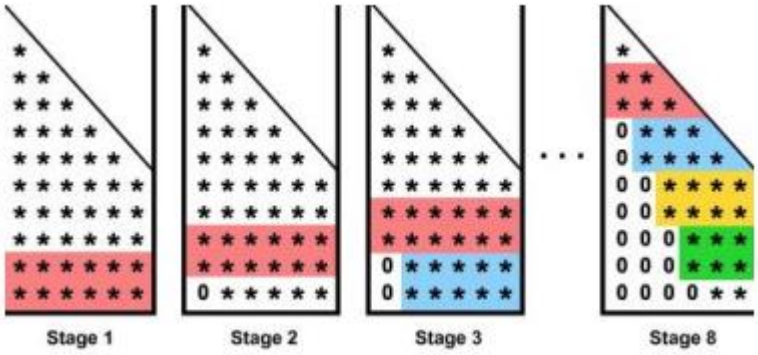


Fig. 2. A Givens Rotation process for QR decomposition. Matrix elements are eliminated 1 by 1, while adjusting the remaining elements of the row to retain matrix information. Image taken from [94]

Subsequently, the operands A,B,C,D of Equation 13 equate the [0,0],[0,1],[1,0] and [1,1] elements respectively of each 2x2 matrix. For the QR decomposition, only the first product matrix, composed of sines and cosines of the β value are necessary. The matrix is multiplied by the original input matrix to zero out the element a_{ij} while adjusting the remaining values of the row (or column, given that the algorithm is symmetrical) as can be seen in Figure 2. When every lower triangular element of the input matrix has been eliminated, the upper triangular portion equals the R matrix in a QR decomposition, while all Givens Rotations computed up to that point can then be multiplied by an eye matrix to achieve the Q portion of the decomposition.

$$E = \frac{A + D}{2} \tag{1}$$

$$F = \frac{A - D}{2} \tag{2}$$

$$G = \frac{B + C}{2} \tag{3}$$

$$H = \frac{B - C}{2} \tag{4}$$

$$Q = \sqrt{E^2 + H^2} \tag{5}$$

$$R = \sqrt{F^2 + G^2} \tag{6}$$

$$w_1 = Q + R \tag{7}$$

$$w_2 = Q - R \tag{8}$$

$$a_1 = \text{atan2}(G, F) \tag{9}$$

$$a_2 = \text{atan2}(H, E) \tag{10}$$

$$\gamma = \frac{a_2 - a_1}{2} \tag{11}$$

$$\beta = \frac{a_2 + a_1}{2} \tag{12}$$

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} cos\,\beta & sin\,\beta \\ -sin\,\beta & cos\,\beta \end{bmatrix} \begin{bmatrix} w_1 & 0 \\ 0 & w_2 \end{bmatrix} \begin{bmatrix} cos\,\gamma & sin\,\gamma \\ -sin\,\gamma & cos\,\gamma \end{bmatrix} \tag{13}$$

The Givens rotation-based approach for computing the SVD by incorporating a QR decomposition, known as the Kogbetliantz method[78], does so by first computing the QR matrices of the initial matrix, then breaking down the upper-triangular matrix R into a series of 2x2 matrices and repeating the Givens Rotation scheme as with the QR decomposition, but this time eliminating all the remaining non-diagonal elements of the matrix with two-sided rotations. Givens rotations are applied from both left and right to R for each Givens Rotation, nullifying the two off-diagonal elements each time and modifying both a row and a column. A series of sweeps are done on the entire matrix, such that data between elements of different intermediate matrices traverse the entire global matrix.

For the SVD incorporating Givens Rotations, equation 14 outlines how the computed values equate each factor of the decomposition:

$$A = U(\beta)WV(\gamma) \tag{14}$$

Once the set of sine and cosine of the final γ and β values is taken from all the Givens Rotations used, together with the *w* values, a final SVD decomposed matrix is achieved by additionally applying them on the original Q matrix.

This approach avoids the $O(\mathrm{mn}^2 + \mathrm{n}^3)$ time complexity necessary for a standard dot-product based SVD computation for *n*x*m* size matrices and has a complexity of $O(\frac{1}{2}\mathrm{mn}^2)$ with a critical added benefit of being highly parallelizable as shall be outlined in the implementation section. The downside of the algorithm is the necessity to compute sine, cosine values as well as a higher number of arithmetic operations needed in general for small matrices. However, as shall be explored in the following sections, a CORDIC accelerator can be used to highly efficiently compute these trigonometric operations.

1.5. CORDIC

1.5.1. General theory

The Coordinate Rotation Digital Computer (CORDIC) [29] is a hardware algorithm first proposed by Volder [13] and later generalized by Walter [66] which uses rotations of vectors to calculate trigonometric, linear and hyperbolic functions.

CORDIC's power lies in its simplicity, being capable of handling most arithmetic operations using only two bit-shifters, three adders and three multiplexers. As such, it belongs in the class of 'shift and add' algorithms. The algorithm is approximate, in the classical implementation taking up *n* iterations to compute *n* bits of precision. This also makes it an algorithm with substantially lower propagation delay, allowing for much higher clock speeds to be achieved, with the highest propagation delay coming from an *n* bit adder for the standard implementation using ripple-carry adders.

CORDIC's general principle is to implement a rotation of a two-dimensional vector $p_0 = [x_0, y_0]$ by an angle θ to obtain a rotated vector $p_n = [x_n, y_n]$ by the use of a matrix product $p_n = Rp_0$ where R is the rotation matrix:

$$R = \begin{bmatrix} cos\,\theta & -sin\,\theta \\ sin\,\theta & cos\,\theta \end{bmatrix} \tag{15}$$

If the cosine term in (15) is factored out, the rotation matrix *R* can be rewritten as (16) and can then be interpreted as a product of a scale-factor K as in (17) with a pseudo rotation matrix R_c as in (18)

$$R_c = [(1 + tan^2\theta)^{\frac{-1}{2}}] \begin{bmatrix} 1 & -tan\,\theta \\ tan\,\theta & 1 \end{bmatrix} \tag{16}$$

$$K = \left[(1 + \tan^2\theta)^{\frac{-1}{2}}\right] \tag{17}$$

$$R_c = \begin{bmatrix} 1 & -tan\,\theta \\ tan\,\theta & 1 \end{bmatrix} \tag{18}$$

The given pseudo rotation operation rotates p_0 by an angle θ to change its magnitude by a factor of $K = \cos(theta)$ and produce a pseudo-rotated vector $\grave{p_n} = R_c p_0$. The most basic CORDIC iterations are written as in (19,20 and 21). One of two modes of operation is selected: vectoring mode (VM) or rotation mode (RM), which influences how the direction of the subsequent micro rotation σ changes as iterations progress, with rotation mode causing the direction to be picked based on the *z* input value's sign after a micro rotation or *y* value's sign in the case of vectoring mode. Circular, Hyperbolic and Linear functional modes are additionally picked between and influence the *x* input value convergence by setting the update function to 0 in the case of linear or the inverse in the case of hyperbolic mode. The set of modes for CORDIC are architectural decisions, however, they can also be implemented in the same circuit at the cost of additional area.

This results in a wide variety of trigonometric, logarithm, exponent, square root and similar operations, all of which can be computed on vectors with the use of only adders and bit shifters. CORDIC variations capable of each set of operations can be obtained by only changing the update function condition statements, making it easy to design and adapt for certain operator needs. Notable is the parameter μ which was implemented by Walter [66] to generalize CORDIC for the use with hyperbolic functions, expanding the range of functions CORDIC can calculate tremendously.

$$x_{i+1} = x_i - \sigma_i \cdot 2^{-i} \cdot y_i \tag{19}$$

$$y_{i+1} = y_i + \sigma_i \cdot 2^{-i} \cdot x_i \tag{20}$$

$$\omega_{i+1} = \omega_i - \sigma_i \cdot \alpha_i \tag{21}$$

A notable feature of CORDIC is the bit precision of an operation run in CORDIC depending entirely on the number of iterations. This makes CORDIC capable of runtime precision to latency tradeoffs, which can be highly useful in a real-time environment.

The primary downside of the CORDIC algorithm is the introduction of the scaling factor [71]. When a microrotation is performed inside CORDIC, a K_i magnitude is added to the vector, which, while

not impacting the way future microrotations are performed, given that the angle remains the same, will accumulate over the course of the algorithm's runtime to a constant *K* and will need to be accounted for to yield a correct result. Additionally, CORDIC handles values in the range of -2.0 to +2.0, requiring normalized inputs.

The upside is that the generalized CORDIC's scaling factor is indeed constant and converges to 1.6467... This results in a $variable * constant$ multiplication which can be performed cheapy as a final step of a series of CORDIC operations. In algorithms dealing with large arrays, multiple values can also be calculated using CORDIC without accounting for the scaling factor initially and then multiplying the entire array by the scaling factor when the result is needed, which is still substantially more efficient to do than computing individual multiplications of two variables. [72]. Additionally, operations such as sine and cosine can be computed without scaling factor adjustment by assigning the value $\frac{1}{K}$) to one of the three inputs. Additionally, the multiplication and division operations obtained from a vector mode CORDIC do not require a scaling factor adjustment entirely. The normalization downside may also often be ignored if the data is used in the machine learning context, given that many such computations already normalize their values to -1.0 - +1.0 range. [92]

It must also be noted, that by itself, a circular CORDIC only allows input angles in two quadrants $z_i < \frac{\pi}{2}$ in rotation mode and $z > 0$ in vector mode. Thus, an initial angle correction stage must be present in each architecture if the full range of input angles is desired. In the implementation section, a separate pipeline stage for this correction shall be reviewed.

Since the original design in 1959, CORDIC has been researched in detail over the years, with numerous architectures developed, all with varying trade-offs in latency, throughput, area, power and precision. [29] and [30] provide field surveys on the varying architectures, however some must be mentioned.

The Higher Radix CORDIC algorithm [67] halves the number of micro-rotations needed to compute a full vector rotation, but requires more hardware to implement the standard rotation cell by using a higher radix form for the intermediate values.

Angle Recoding [68] is an implementation substantially speeding up CORDIC but only if the angle of rotation is known ahead of time via the use of additional lookup tables.

Hybrid CORDIC [69] is an idea of having two separate CORDICs for different stages of the computation, one focused on ROM accesses and the other on the shift-add operation.

Low-Latency CORDIC [76] utilizes multipliers and lookup tables to lower the amount of CORDIC microrotations done for an operation to 2/n and then approximate the remaining 2/n bits.

Lastly, the redundant-number-based CORDIC [70] is an approach of using redundant arithmetic to eliminate carries, further increasing performance by substantially lowering the adder propagation delays at the cost of a varying scaling factor.

While only small number of CORDIC architectures have been mentioned in this section, a key insight is that CORDIC can be adapted to meet different design constraints at the cost of specific downsides. The existence of area-throughput-latency tradeoffs by adapting the architecture imply the use of

CORDIC as a malleable general purpose arithmetic unit, utilizing underlying FGPA logic to make positive trade-offs.

However, the rate at which CORDIC is being researched has been steadily decreasing over the years which can largely be attributed to the fact that modern ASIC and FPGA boards already feature hardware multipliers while CORDIC's key selling point was always being a tremendously low area cost arithmetic unit.

Furthermore, not much research has been done on exploring how CORDIC could be used in conjunction with modern multipliers and increased memory capacity. A unifying approach is of great interest.

1.5.2. Approximation scheme

The decreasing popularity of CORDIC [73] as a research topic might be attributed to the fact that modern ASIC and FPGA circuits tend to have on-chip multiplication units, such as the DSP units found in modern Xilinx FPGAs. These units can perform many of the primary operations which a CORDIC block can, but are hardwired into the chip, giving substantially better performance per area.

However, making use of hardware DSPs is non-trivial, much as making use of FPGAs in general. Although methodologies such as High-Level Synthesis are gaining in popularity [74], most high-performance accelerators targeting FPGAs are still written in RTL, like Bluespec [75]. Utilizing DSPs in Xilinx FPGAs for complicated arithmetic operations such as square root and division, for instance, generally require using specific Xilinx IP cores and are notoriously difficult to implement by hand using the DSPs. The designs are then also non-vendor agnostic, with a Xilinx IP-utilizing accelerator no longer being usable for Intel FPGAs without large additional redesign time.

A key point to easing the approachability of reconfigurable hardware could be increasing the level of abstraction and hiding the underlying circuity. The move towards general IP cores as building blocks of modern designs are a common example of this with CORDIC itself being often plugged into designs as an IP core, hiding all underlying design details, often as a sine/cosine generator.

The proposal of this work is that CORDIC, being a highly generalized arithmetic unit, could be used as an intermediately layer in hardware to make use of other hardware resources, additionally acting as a proof of concept for the use of shift-add hardware algorithms in such a manner. Specifically, one approach is to implement the CORIDC 2/n bit approximation as in [76]. LUTs and FFs can be used to implement the CORDIC arithmetic unit itself for the first 2/n stages while a DSP unit as well as additional memory blocks for lookup tables may be used to implement the remaining 2/n bit approximation. A set of CORDIC units may then perform hardware arithmetic in a manner focused on minimizing their propagation delay or area, utilizing vastly fewer DSP blocks than when using conventional multiplication and division schemes, allowing to achieve a more balanced utilization of all available FPGA resources.

This approach can be generalized to using any hardware logic available to enable larger CORDIC designs. Instead of using DSPs and a few multipliers in an FPGA chip, an array of CORDICs can be used with the DSP units being used to enable said CORDIC units. This scheme can further be balanced by varying the number of CORDIC units to DSP units used as well as the type of hardware utilized for specific portions of the computation.

Figure 3 illustrates the proposed abstract design: Data is streamed in from the host to an array of pipelined CORDICs in FPGA fabric, abstracted from the designer, with a set of accelerators tied to the array which handle bit approximations as a separate stage in the overarching pipeline. Notably, for higher throughput, only two values need to be streamed in and out for most CORDIC operations and a 3rd input value can be loaded into a register as a constant. This approach may additionally extend into 3D CORDIC architectures.

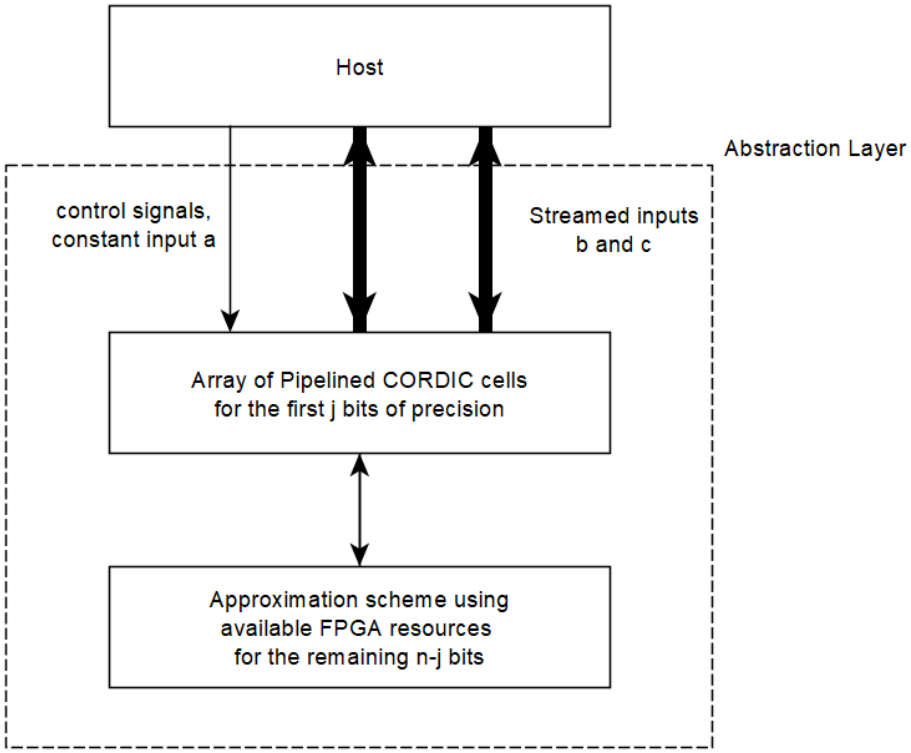


Fig. 3. Abstract CORDIC compute unit array with domain FPGA resources being used for approximating a portion of the operation precision

The choice of a CORDIC algorithm to accelerate in this manner is non-trivial. The generalized CORDIC is the most straightforward approach, using the DSPs to simply readjust the scaling factor by the factor stored in a register. A redundant-CORDIC implementation, however, would put higher burden on the accelerator, which could be positive if looking to balance the use of LUTs for CORDIC units and the number of DSPs needed for the scaling adjustment, however the propagation delay gains are lost if the minimal propagation delay of the DSP multiplications are higher than the redundant adders. The proposal in this work is to realize a CORDIC implementation as in [76] for Vectoring and Rotation modes as two separate designs, utilizing the DSP units found on the FPGA as a stand-in for constant multipliers to approximate the final outputs of the initial 2/n stages as well as BRAMs to implement a reciprocal lookup table for Vectoring Mode, Figure 3 illustrates this exact scheme. The details of such a design are covered in the Implementation section.

1.6. Summary

The primary takeaways from the overview section are that:

1. There is indeed a plethora of compute problems in the ML field where the utilization of FPGAs is of great interest and the technology has already been utilized in various forms to good success.

2. Many of the solutions are highly specialized and domain specific, while there exist plenty of software libraries which feature a wider scope of algorithmic problems and are thus more used and could be used as potential targets for underlying FPGA acceleration.

3. RTL design has also sharply improved in recent years with higher abstraction level methodologies such as object oriented programing presenting new opportunities for hardware design.

4. Vivado block diagrams and the ZYNQ architecture provides further means of incorporating FPGA designs into the software domain, but could make use of more reusable compute units to not restrain design use cases.

5. There are indeed algorithms that utilize overlapping underlying arithmetic, which at a library level could be offloaded to FPGAs, the most notable example being matrix decomposition using Givens Rotations.

6. The CORDIC algorithm, especially the low-latency variant, is an algorithm of great note in terms of more generalized arithmetic units, capable of handling a plethora of arithmetic operations using the same interface and a highly overlapping design. It can also make good use of all available FPGA resources to further facilitate effective utilization of the technology.

Hypothesis: the low-latency CORDIC algorithm, utilizing DSPs and BRAMs found on modern FPGAs, could be incorporated into modern ML algorithms, notably, the matrix decomposition methods such as the SVD and QR factorization using overlapping arithmetic operations in Givens Rotations at the software level, providing a straightforward interface to available hardware resources. Additionally, by utilizing modern hardware design methodologies and tools, highly reusable designs could be made which could further facilitate FPGA adoption in the field of ML without requiring hardware expertise.
The following set of tasks is outlined to verify the validity of the hypothesis:

1. Model and implement an FPGA CORDIC accelerator interfaceable from a Python environment for use in accelerating software algorithms designed with the Numpy compute library.
2. Model and implement a CORDIC variant utilizing on-chip DSP and BRAM units present in modern FPGAs as CORDIC back-ends to solve classical trigonometric operations versus implementing them in the DSPs themselves.
3. Implement the SVD and QR factorization for decomposing matrix datasets via the Numpy compute library, using the Givens Rotations method for the underlying algorithms.
4. Parallelize the SVD and QR algorithms for use with a CORDIC accelerator.
5. Benchmark the CORDIC accelerators for solving classic trigonometric operations such as sine, cosine, multiplication, and division, comparing resource utilization, precision, and throughput with a direct DSP approach when possible.

2. Research Methodology

2.1. Criteria for benchmarking

The primary evaluation metric proposed in this work is to focus on throughput per logic element & DSP and BRAM cell consumed by the design as well as operation precision. Special care should be put into comparing how different designs perform versus using standard DSP cells found in modern FPGA boards as well as hybrid approaches. Rather than focusing on the maximal throughput achievable on a device, the standard and logic cell count for the exact arithmetic units will be considered. The following list summaries the benchmarks:

- Logic Unit utilization / CORDIC processing unit, count
- Flip-Flop utilization / CORDIC processing unit, count
- FPGA-local Digital Signal Processor utilization / CORDIC processing unit, count
- Block RAM utilization / CORDIC processing unit, count in Kilobytes
- Mean square error of the final arithmetic operations, absolute value
- CORDIC throughput with data loading from RAM, Megabytes / second

2.2. Verification of implemented algorithms

To properly verify the correct workings of the CORDIC core, as well as the matrix decomposition computation steps in the software side, a test suite is to be designed for testing individual CORDIC operations (sine, cosine, multiplication, division, square root) on a large array to be run with an input sizes carried from 2 to the maximum allowed buffer space in the PYNQ-Z1 device, then compared to running Python's standard numerical library operations on the same array (Numpy.sin, Numpy.cos etc).

2.3. Conditions for the test bench

Standard IEEE-754 Floating Point format is to be used as inputs to the software algorithms and the CORDIC accelerator, with 1 sign bit, 8 exponent bits and 23 mantissa bits. Conversion to fixed point arithmetic is to be done in hardware, using 1 bit for storing the integer portion and 30 bits for storing the fractional portion as well as the sign bit. All input data needs to be normalized to a -1.0 to +1.0 range due to CORDIC's internal limitation of requiring 1 integer bit side fixed point inputs.

3. Implementation

The following sections feature a proposed DSP-enabled architecture for the CORDIC arithmetic cell and a design for abstracting the CORDIC accelerator for use in software designs. A methodology for computing Givens rotations using the CORDIC accelerator is also presented.

3.1. Basic CORDIC architecture

For high-throughput arithmetic, the conventional iterative CORDIC must be converted to a pipelined structure. The reason for doing so is the fact that an iterative CORDIC requires a lookup table for storing the potential arctangent angles of microrotations used in each iteration. A pipelined implementation can use wired shifters and wired register-stored microrotation angles for each iteration. In other words, each of the *n* stages of a pipelined structure for *n* bit inputs uses dedicated hardware for a given bit/stored angle. This increases throughput by a factor of *n*, while hardware complexity is increased by less than *n*.
The block diagram in Figure 4 illustrates the pipelined classic configuration for both modes.

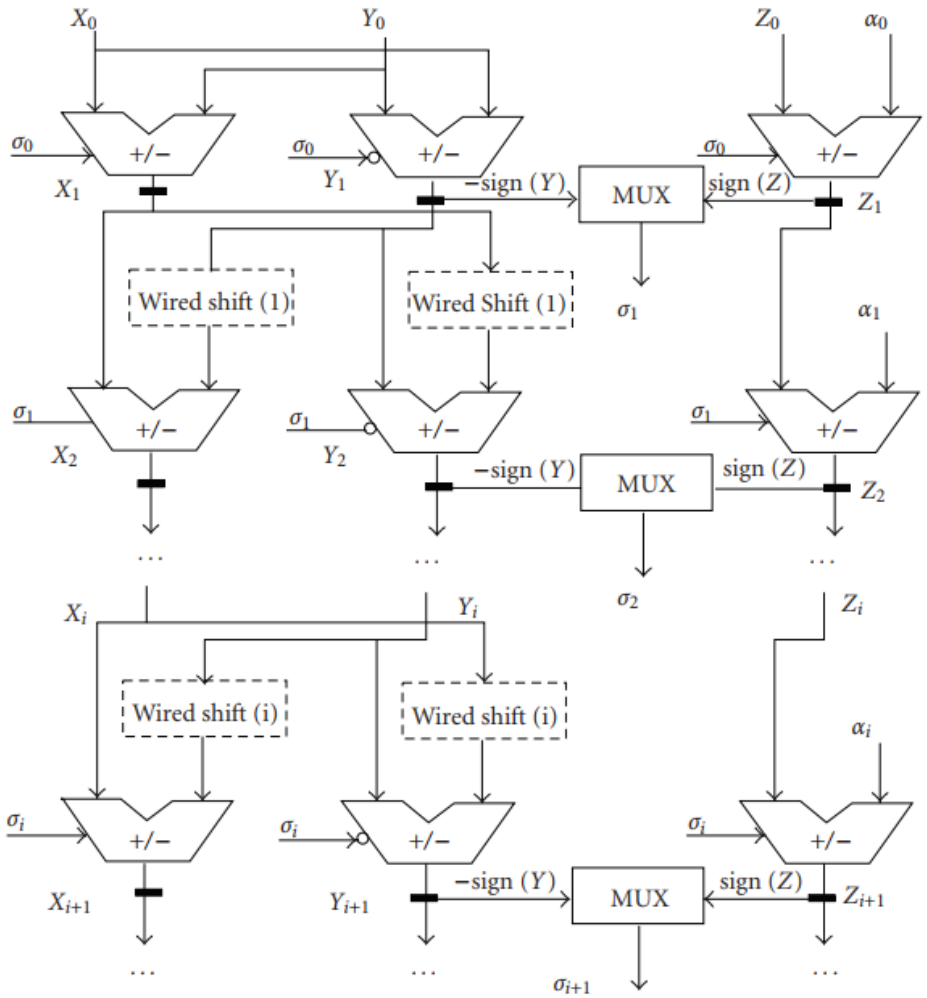


Fig. 4. pipelined generalized conventional CORDIC. 3 Adders and two bit shifters per stage are required. Image taken from: [95]

The implementation inputs the data values *x*,*y* and *z* as well as a value *alpha* for choosing between rotation and vectoring modes, to the left can be seen individual atan value inputs hard-wired instead of using a lookup table. The baseline design used in this work omits the scaling factor correction, for

which either dedicated hardware is used or a final correction is done in software. Conventionally, ripple adders are used for each stage of the pipeline, introducing a propagation delay equivalent to propagating an *n* bit carry in a single ripple adder. FPGAs are specifically optimized to have fast ripple carries inside individual CLBs (combinational logic blocks) thus redundant arithmetic is not considered in this design due to the DSP unit use which shall be outlined in the following section dwarfing the maximum propagation delay. Lastly, while the alpha value is mentioned , in this work, two separate CORDIC variants are explored and modified: Rotation Mode and Vector Mode, rather than using a unified system. The reason for this being that the modifications proposed in later sections diverge in resource requirements of the two variants, making a unified approach inefficient if only one type of operation is used with an individual CORDIC compute unit.

3.2. DSP-enabled CORDIC arithmetic cell modification for Vector and Rotation modes

The low latency pipelined 2D CORDIC as described in [76] utilizes the fact that only the first n/2 bits have to be calculated using the add-shift scheme of the conventional CORDIC, while the remaining 2/n bits can, in fact, be approximated using a truncated constant multiplier and a lookup table. This approach is based on the calculation that the final n/2 bits, if approximated with a single rotation, suffer from an error rate as low, or even lower, than the full mean error of a conventional CORDIC implementation with full *n* micro rotations.

The primary upside of this approach is the need to only compute the first 2/n iterations, which halves the depth of the pipeline, cutting the necessary hardware in half. The downside is the need for a multiplier for implementing the final approximation as well as a lookup table in the case of vectoring mode. However, these additional multipliers are truncated to a bit width of $2/n+m$ where m is $\log2(n)$ (guard bits and overflow bits), which are substantially cheaper to implement in hardware than full bit width multipliers. The lookup table also features a depth proportionate to the number of bits being approximated and can be controlled for varying FPGA chip memory budgets.

The cited work did not feature dedicated multipliers (DSPs) for implementing the CORDICs, for a constant multiplier is substantially cheaper to implement in FPGA logic than a regular one and thus, the use of DSPs would have lead to minimal gains. The authors additionally outlined that the implementation can straightforwardly be used with redundant arithmetic for the first $2/n$ pipeline stages.

In this work, the novelty presented is to use CORDIC rotations for the first $2/n$ stages, using the implementation presented in [76] for the remaining $2/n+m$ bits and to utilize DSP units for the multiplication instead of constant multipliers and BRAMs for reciprocal computation instead of standard lookup tables.

The prototype design implemented in this work takes in 32-bit floating-point inputs with 8 exponent and 23 mantissa bits, converts them to fixed point with 1 integer bit, runs 16 CORDIC micro-rotation stages on the first 16 bits of the inputs and then utilizes Xilinx series 7 DSP48E cells for the remaining 16 bits approximation

The architecture of the CORDIC cells is presented in Figure 5 and illustrates the proposed implementation block diagram with the following set of fully pipelined operations:

- Control signals and the constant a value are loaded to the CORDIC auxiliary registers using an AXI-lite interface.
- The b and c streamed inputs are converted from floating point to fixed point using standard 3-stage pipeline converters.
- Using the control signals in the auxiliary block, the a, b and c values are multiplexed for the desired functionality of a given CORDIC compute unit.
- An angle correction stage is passed during which the initial rotation quadrant is tested to be in the range 0 - $\frac{\pi}{2}$ and adjusted accordingly if needed.
- n/2+1 stages of general CORDIC micro rotations are passed, computing the first n/2 bits of the output vector.
- In both designs, after n/2+1 stages, all 3 values are stored in hold registers to delay by 1 clock cycle.
- In vector mode, this clock cycle is used to look up the 1/x[q] reciprocal in a lookup table stored in FPGA BRAM where x[q] equals the n/2 leading bits of x after n/2 iterations of CORDIC.
- In rotation mode, this cycle is used to compute n/2 x n/2 multiplications needed for a and b value approximation.

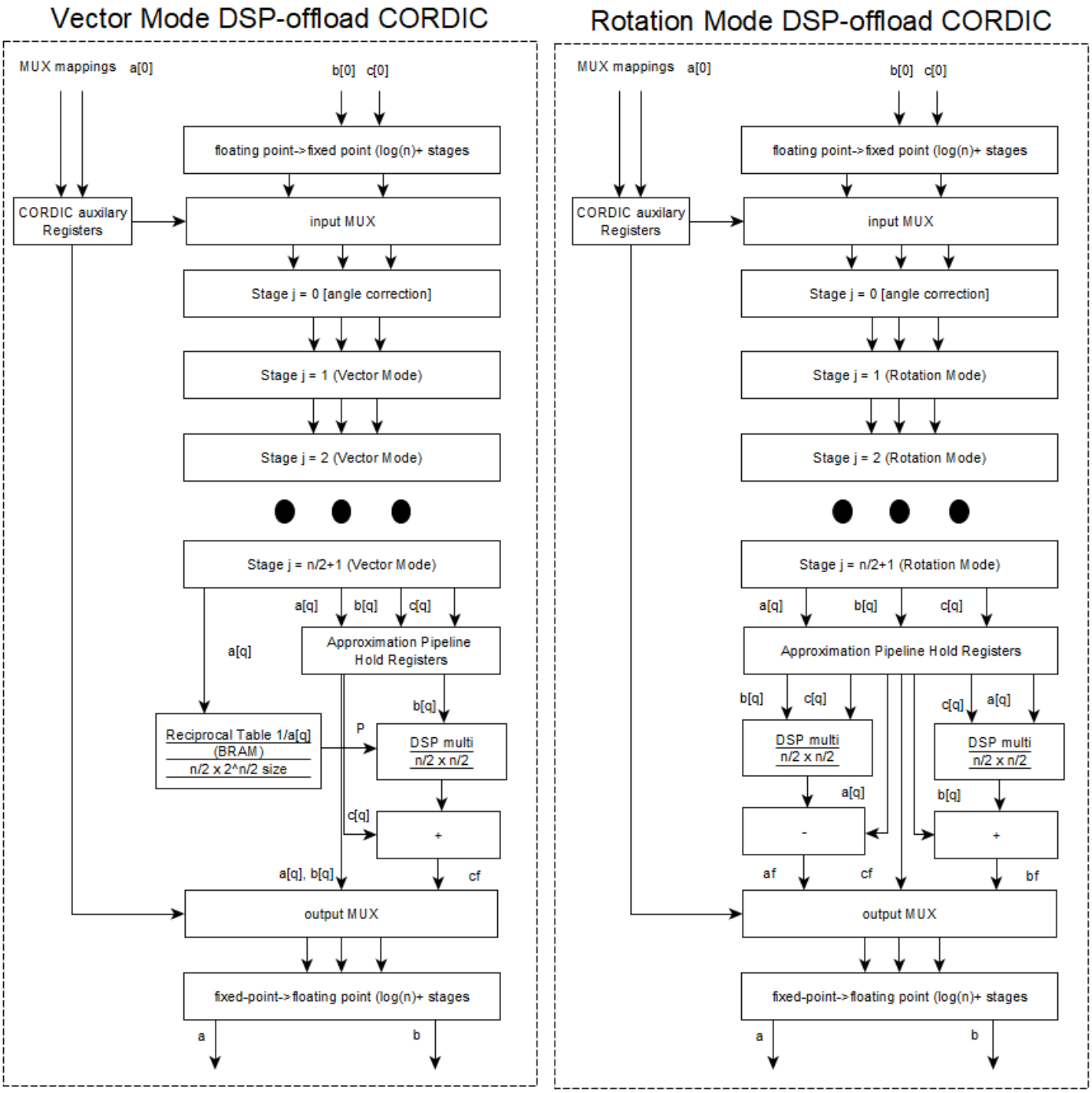


Fig. 5. The proposed CORDIC architectures with DSP and BRAM utilization. The Stage j blocks are optimized ripple carry adder CORDIC microrotation update blocks. The reciprocal computation is done using BRAM-based lookup tables

- In vector mode, the reciprocal table addressed data is multiplied by the b[q] value to obtain the final cf value after an addition.
- In both circuits, the final values are multiplexed again according to control signals.
- Lastly, the values are converted back to floating point format in yet another 3-stage pipeline.

3.3. Generalization of the CORDIC cell for use as a hardware accelerator

The generalization of the CORDIC cell for use as an accelerator is now presented. The architecture presented in the previous section realizes a pipelined 32-bit CORDIC core, surrounded by two sets of multiplexers for generalizing the core to take in 1 register-mapped constant value and two streamed values. The design interface can be incorporated into an Advanced eXtensible Interface Stream (AXIS) system.

A homogeneous Xilinx FPGA (ZYNQ), specifically the PYNQ device is used to implement the design, using Direct Access Memory (DMA) implementing the AXI4 and AXIS interfaces to feed data from the host device to the arithmetic core. A simplified outline can be seen in Figure 8 and a full one in Figure 9.

In the full block diagram, gp_cordic_0 directly corresponds to Figure 7 diagram, using a templated Xilinx AXI Stream slave and master interfaces of 64 bit widths and a set of 3 3:1 input and output MUXes.

The gp_cordic_aux_0 block is a simple AXI Lite slave wrapped for directing the initialization signals to the gp_cordic_0 block. The rest are generic Xilinx IP blocks, with the DMA set to maximum 512 value burst reads with a 32bit DDR width interface.

As concrete implementations to be experimented upon, Vector and Rotation Mode CORDIC implementations are realized using the classic CORDIC architecture design as described in [66] as well as the modified variants utilizing DSP slices and BRAM cells. Lastly, Xilinx Divider and Multiplier IP cores are incorporated using the exact same logic up to and including the input and output multiplexers, but switching out the CORDIC core for the IP blocks to compare resource utilization. The designs are synthesized with two guard stages for overflow protection.

Migen [58] is used for modeling the generalized CORDIC RTL design, converting to Verilog and synthesizing with the Vivado toolchain.

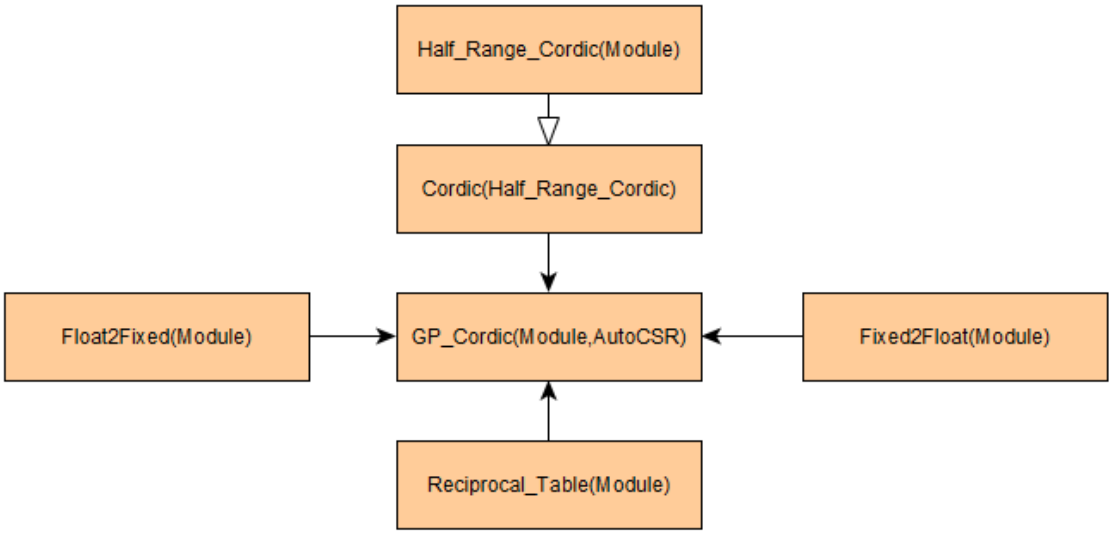


Fig. 6. Migen Class hierarchy for the Generalized CORDIC cell, white arrow – inheritance, black - encapsulation

As shown in Figure 6, the **Half_Range_Cordic** class (white arrow signifies inheritance, black-encapsulation), implements the main CORDIC stages with their respective micro rotation and lookup tables for atan values. This class is inherited by CORDIC, which has the additional stage for angle correction if it does not fall into the 0 - $\frac{\pi}{2}$ range. The class is instantiated alongside **Float2Fixed, Fixed2Float** and **Reciprocal_Table** classes inside the **GP_Cordic** class, which glues all the components together with additional pipelining stages. The reciprocal table incorporates the **Memory** class construct provided by Migen for initializing large memories and was initialized with reciprocal values using Numpy. The entire design utilizes branch conditions to reuse most of the code for both Vector and Rotation modes, while branching key portions of the design to implement either mode by itself. This makes the design highly reusable and parametrizable. Notable, the design can be initialized with the parameters: input width, stage count, guard bit count, evaluation mode (pipelined, combinational, iterated), CORDIC modes between vector and rotate and functional modes between linear and hyperbolic.

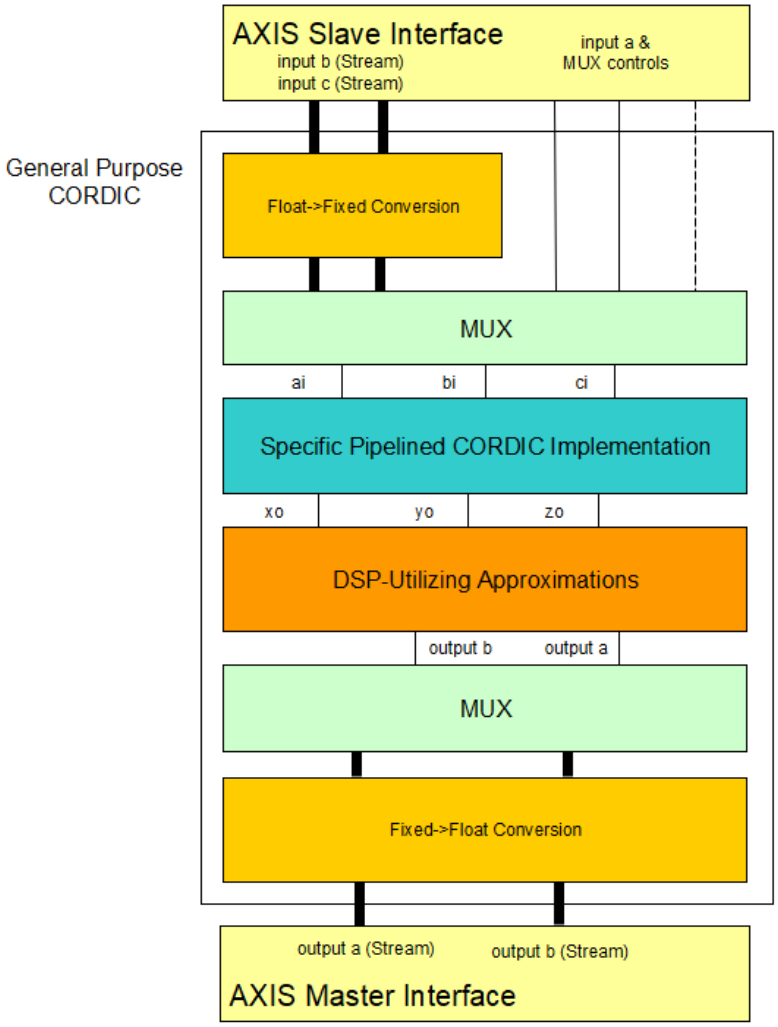


Fig. 7. Pipelined CORDIC with DSP acceleration, format conversion and control multiplexing in the context of an axis interface

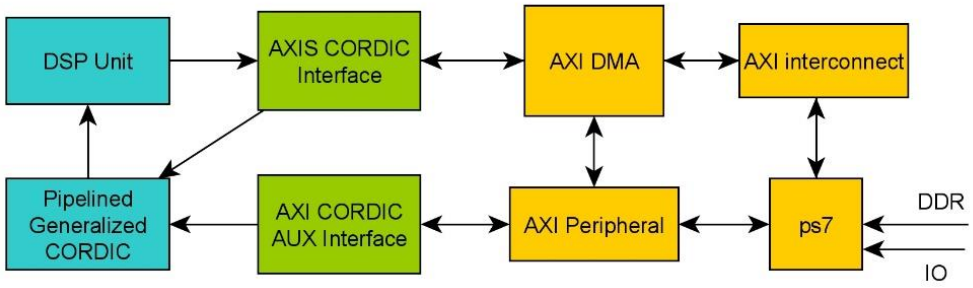


Fig. 8. Accelerated CORDIC on a PYNQ device simplified block diagram for AXI connections

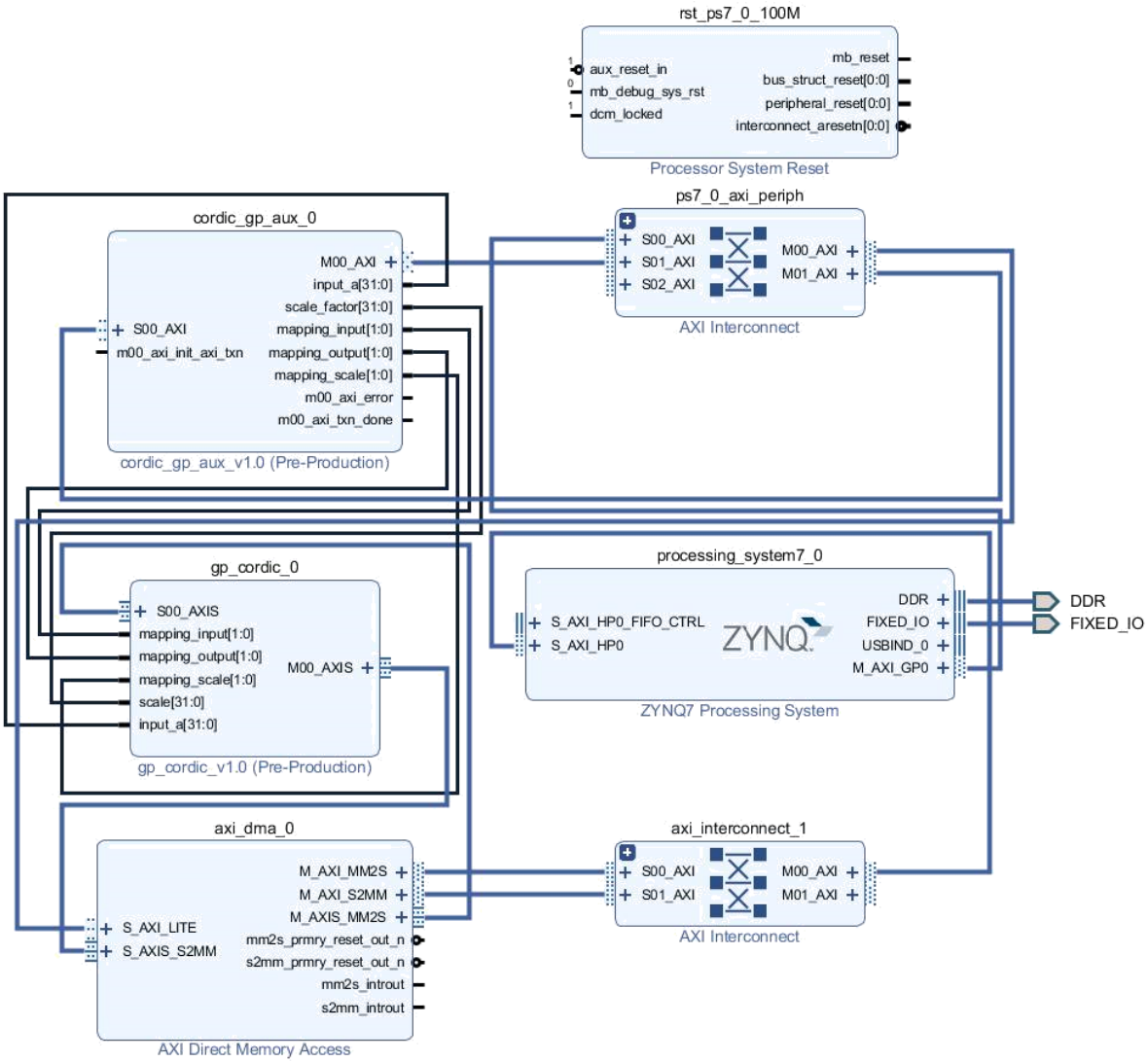


Fig. 9. The Vivado Block Diagram of the general purpose CORDIC unit communicating via DMA with the ZYNQ processing system. Clock signals routing from rst_ps7_0_100M omitted for clarity

A processing system IP for connecting the programmable logic with the ARM cores found in the PYNQ device is used, alongside AXI4 peripheral, Interconnect and DMA cores for routing the data streams and register values to the CORDIC core. Two interface modules are used for AXI Stream and AXI Lite protocol wrapping around the core.

3.4. AXI-Stream interface using Direct Memory Access

Three layers of communication are used between the ZYNQ processing system on the XILINX PYNQ FPGA board and the CORDIC compute units (CUs): AXI-Stream interfaces, a direct memory access interface, which moves data from the RAM memory and sends to the AXIS interface and, lastly, a round robin scheduler to distribute the compute work among a varying number of CORDIC compute units.
Standard XILINX IP cells are used for implementing the AXI and DMA interfaces but shall be briefly discussed regardless.

3.4.1. DMA

The Xilinx AXI DMA 7 core, as illustrated in Figure 10 is utilized to initiate streamed data transfers between the host main memory (RAM) and AXI-compliant peripherals on the FPGA fabric. S2MM and MMS2 ports are used for connecting to the Xilinx processing system which is the master controller / interface between the main memory and the FPGA chip. 64-bit words are used, for sending two inputs of 32 bits each (1 input is always set constant ahead of time as an AXI-lite signal to be kept in a register) and a maximal 256 burst size is utilized. This allows to maximally saturate the memory bus and stream data at maximum capacity that the CORDIC CUs can handle. AXI4 memory map read/write is used as the port for sending the control signals, as the 3rd, constant, input.

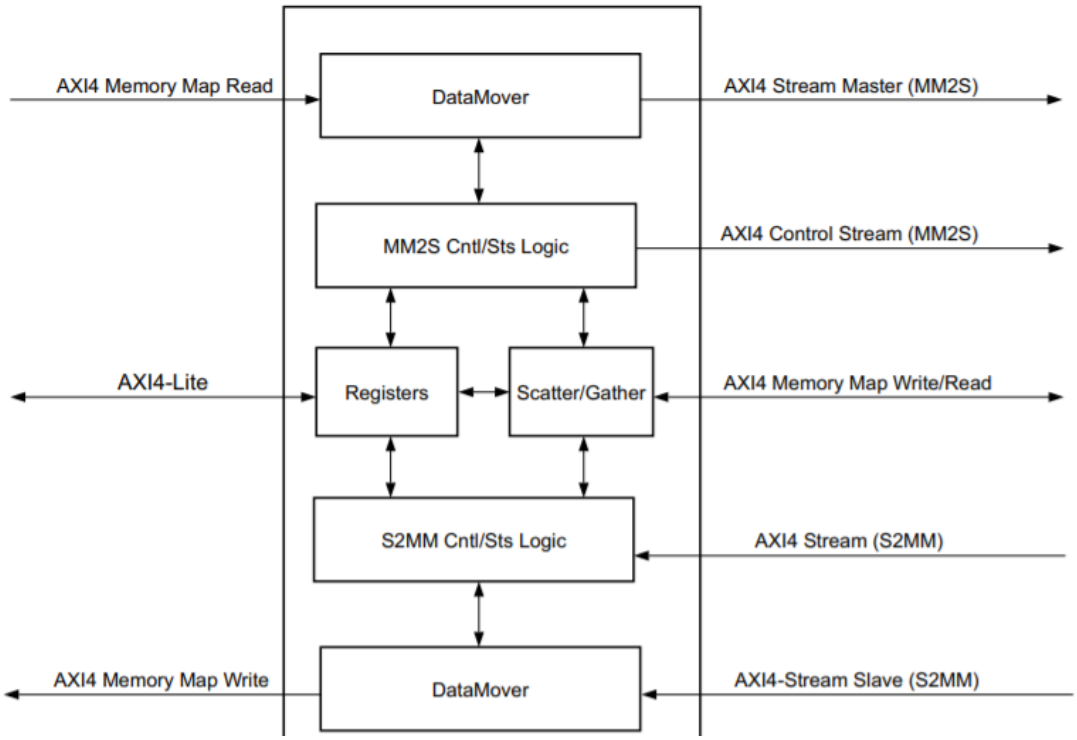


Fig. 10. Xilinx AXI DMA IP block diagram, image taken from [97]

The AXI4 Memory map read and write interfaces are then connected to the AXIS peripherals

3.4.2. AXIS protocol

AXIS is a handshake protocol with VALID, READY, LAST and DATA signals as illustrated in Figure 11.

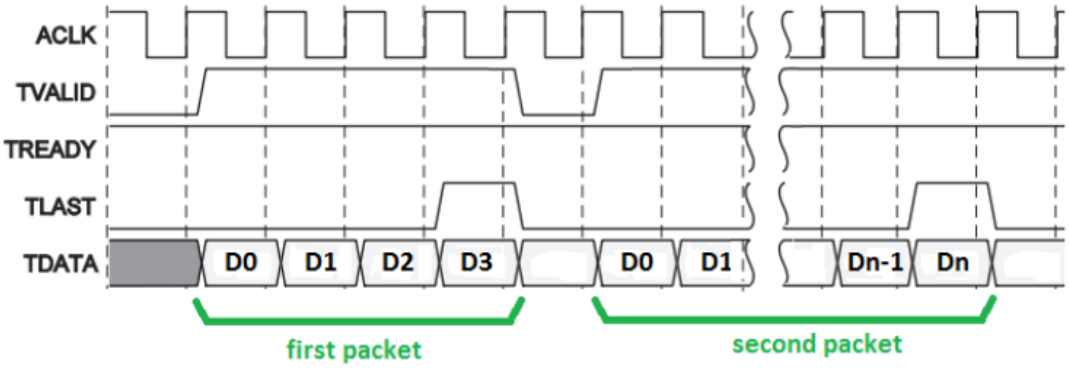


Fig. 11. AXIS protocol waveform. Image taken from [96]

The protocol is guaranteed to supply operands in a FIFO order and can be used for both read and write operations in parallel. In the proposed design, each packet consists of 64-bit words, which are then split into two 32-bit words and loaded into the CORDIC first pipeline stage registers each clock cycle.

3.5. Givens Rotation-based SVD acceleration using the Generalized CORDIC

The previous sections of this work outlined a method of computing the SVD using Givens rotations, as well as a hardware implementation of a generalized CORDIC accelerator. This section goes over how the accelerator may be used in the SVD and QR decomposition algorithms.

The core of the SVD and QR computations is the Givens rotation itself. Given a 2x2 matrix, to compute the final β, γ and *w* values as well as their sine and cosines. Equations 1-9 already outlined the necessary arithmetic for obtaining these values.

Given the set of operations for a Givens rotation, the tanh operations can be replaced with CORDIC operations with inputs (x=c+b,y=d-a,z=0 using vectoring hyperbolic mode CORDIC) for a_1 using the same setup with different x and y values according to the previous expression for a_2.

Next, the β and γ angle coefficients can be extracted using 2 series of Linear Vector mode CORDIC functions with value z set to 0. Each of the Givens rotation matrix's cos and sin values can be calculated using CORDIC's Circular Rotation mode. In total, 4 Vector mode and 8 Rotation mode CORDIC calculations are enough to compute the Givens Rotation, followed by 2/n Vector mode multiplications for each rotation (not required for the 2x2 matrices themselves).

The next step is generalizing into larger matrices. For the method to converge and appropriately zero out all off-diagonal elements in the initial matrix, only disjoint 2x2 sub-matrices may be used in a single step of a sweep in parallel. Figure 12 illustrates a modulus pivot strategy [79] for parallelizing the rotations in an n=7 sized square matrix. There are many other proposed schemes, such as [78] and further strategies outlined in [79], like the butterfly form.

It has to be noted that for small matrices, the scheme would be severely bottlenecked by memory transfers, given that all future Givens Rotation operations require previous sets to complete. An *nxn*

size matrix may require up to *n* subsequent Givens rotation batches which cannot be streamed due to race conditions. The two primary approaches to overcome this bottleneck are to either compute multiple matrices in parallel or to move the final matrix being computed to the hardware memory, avoiding any memory transfers between system RAM and CORDIC units. In this work, multiple matrixes are computed in parallel as the testcase to simplify design, given that the CORDIC accelerator primarily design goal is to work as a simple SIMD-type accelerator.

0	x	2	3	4	5	6	7
1		x	4	5	6	7	1
2			x	6	7	1	2
3				x	1	2	3
4					x	3	4
5						x	5
6							x
x/y	**0**	**1**	**2**	**3**	**4**	**5**	**6**

Fig. 12. A single sweep across an input matrix using Givens Rotations for the SVD. Values sharing the same index are computed in parallel. For example, during the 3rd iteration, Givens Rotations are computed for $S_1 = \{2,0\}, S_2 = \{5,4\}, S_3 = \{6,2\}$ indexed elements

The final software system implements the stage diagram as can be seen in Figure 13.

An input array/matrix is loaded in a Python program, the CORDIC registers are first set using the AXI Lite interface, assigning the constant value and multiplexing of the inputs and outputs.

Afterwards, the input array is decomposed into a series of 2x2 matrices which are parallelizable without creating a race condition. The basic addition operation is done in software, followed by a loading of the input buffers to the FPGA, proceeded by the sending of the data to the device. In the device, the CORDIC compute units are streamed the values, converting to fixed point representation and proceeding through the CORDIC cell. Once an output is obtained and converted back to floating point format, they are sent back to the host device and a test is done if further Givens rotations are necessary, repeating the cycle until the desired precision or maximum sweep count has been reached.

To handle the scaling factor, for the Rotation mode, input a *a* is set to $\frac{1}{\mathrm{K}}$ (0.606) to obtain sine results without the need for scaling factor adjustment, while the vector mode multiplication operation does not generate any gain. For the *tahn* operation, the final matrix is multiplied once by $\frac{1^s}{\mathrm{K}}$ where *s* equals the number of total sweeps.

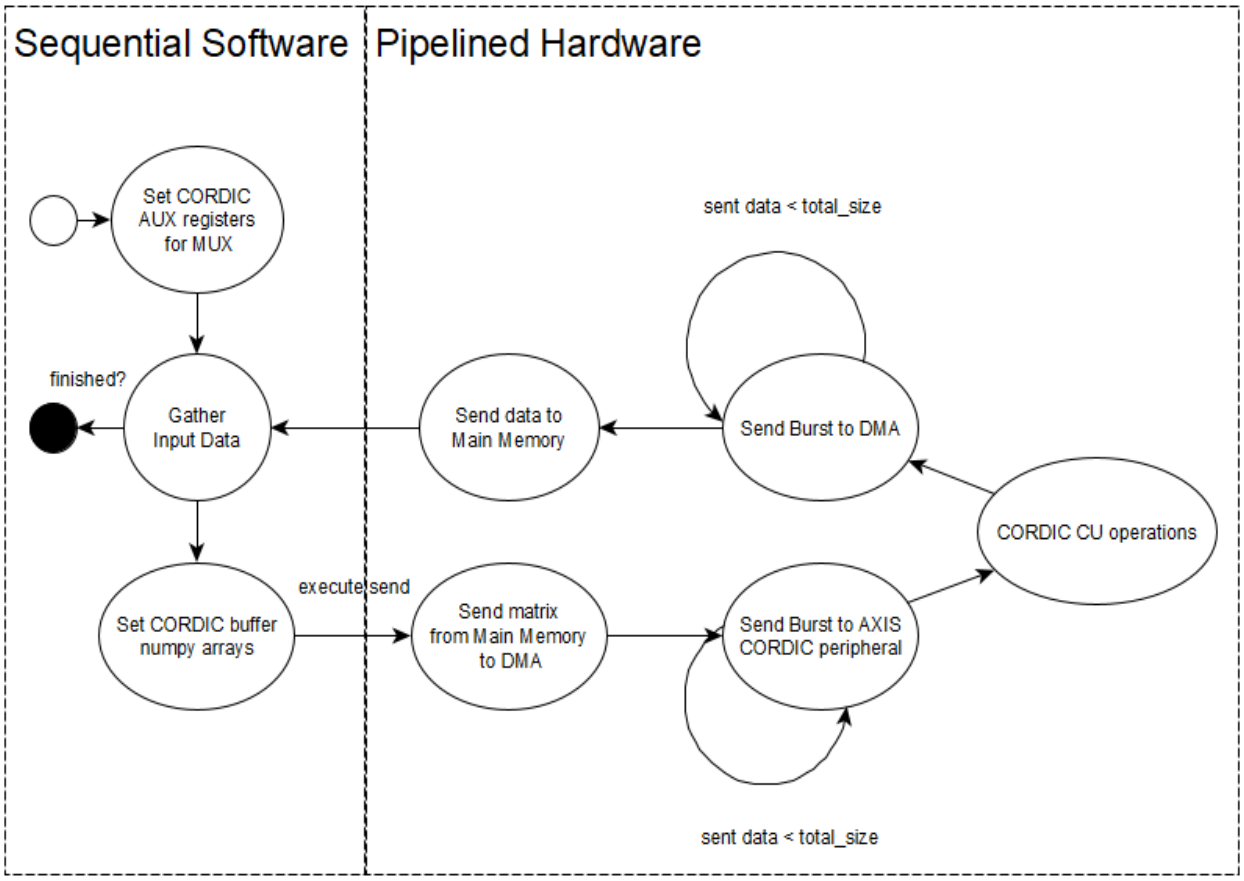


Fig. 13. State diagram for the GP CORDIC accelerator usage from software. Initially CORDIC auxilary registers are set for desired CORDIC functionality and constant (a) value. Once data for processing is ready, buffers are loaded to send to the device in a streamed manner

3.6. The test bench

The final design test bench comprises of CORDIC design simulations for testing appropriate compute unit responses to varying input data sets, testing sine, cosine, multiplication and division operations. The datasets comprise of randomly generated Numpy arrays of float values. The rotation and vector mode versions of both variants of the CORDIC algorithm (the standard and DSP-enabled) as well as the XILINX IP cores for division and multiplication are used for the tests. Resource consumption measurements are taken during this phase.

The CORDIC variants are then tested on the PYNQ-Z1 board, downloading generated bitstreams of the 4 possible sets of designs to the device and comparing:

1. The precision as well as throughput for sine, cosine, multiplication and division operations, with varying input sizes is measured and compared to a software Numpy realization on the same data arrays.
2. CORDIC designs with varying approximated bit counts are generated and compared in regard to resource utilization.

4. Results

4.1. Utilization of resources

Tables 1 and 2 feature the resource consumption of the standard CORDIC algorithm as well as the DSP-offloading prototype and DSP-only implementations synthesized for the PYNQ-Z1 board, the results in these tables are for the processing units themselves, without additional functionality blocks such as the DMA engine, which are featured in a later figure.

Each implementation's LUT, FF, DSP and BRAM costs are considered only of the exact instance of the arithmetic unit (gp_cordic_0), discounting the additional costs of the AXI DMA interface and interconnects but including the floating point - fixed point converters. The cost is presented as the raw number of resources as well as the percentage of the PYNQ-Z1 board's resources available. "4 Converters" equates to the floating point to fixed point and vice versa converters for inputs *a* and *b*. Width equates the input bit-widths for the accelerator, stages-number of CORDIC pipelined stages, while 'approx' indicates the number of bits being approximated by DSPs/BRAMs.

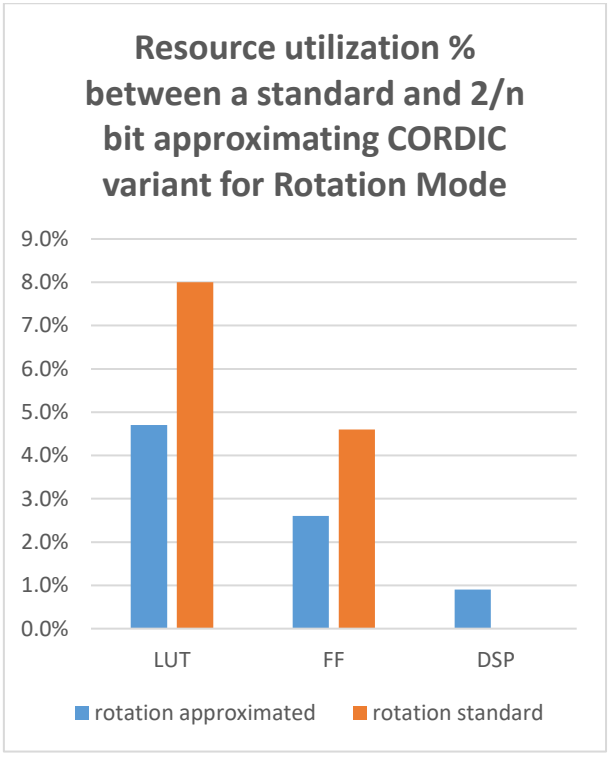
Table 1. Resource utilization of single arithmetic unit instantiations in a PYNQ-Z1 board with 4 converters for float-fixed and back in each case

CORDIC mode	ops	width	stages	approx	LUT	FF	BRAM	DSP
rotation	sin/cos	32	16	16	2608	2579	0	2
rotation	sin/cos	32	32	0	4480	4624	0	0
vector	multi/div	32	16	16	1737	2029	32	1
vector	multi/div	32	32	0	3176	3461	0	0
DSP multi	multi/sin	32	0	0	608	484	0	3
DSP div	div	32	0	0	917	1705	1	13
4 Converters	conversion	32	4	0	411	497	0	0

Table 2. Resource utilization of single arithmetic unit instantiations in a PYNQ-Z1 board with 4 converters for float-fixed and back in each case. Percentages in the PYNQ-Z1 board

CORDIC mode	ops	width	stages	approx	LUT	FF	BRAM	DSP
rotation	sin/cos	32	16	16	4.7%	2.6%	0.0%	0.9%
rotation	sin/cos	32	32	0	8.0%	4.6%	0.0%	0.0%
vector	multi/div	32	16	16	3.1%	2.0%	22.9%	0.5%
vector	multi/div	32	32	0	3.1%	2.0%	0.0%	0.0%
DSP multi	multi/sin	32	0	0	1.1%	0.5%	0.0%	1.4%
DSP div	div	32	0	0	1.6%	1.7%	0.7%	5.9%
4 Converters	conversion	32	4	0	0.7%	0.5%	0.0%	0.0%

The resource utilization between a standard CORDIC (rotation/vector standard) and one using the 16-bit approximation scheme (rotation/vector approximated) is compared in Figure 14 and showcases the significant reduction in LUT and FF resource consumption at the cost of BRAM and DSP use.



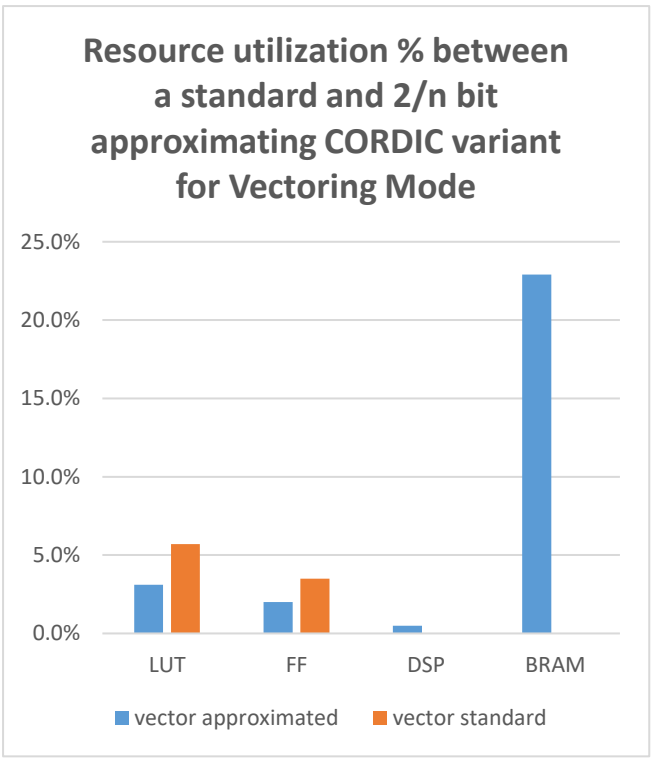
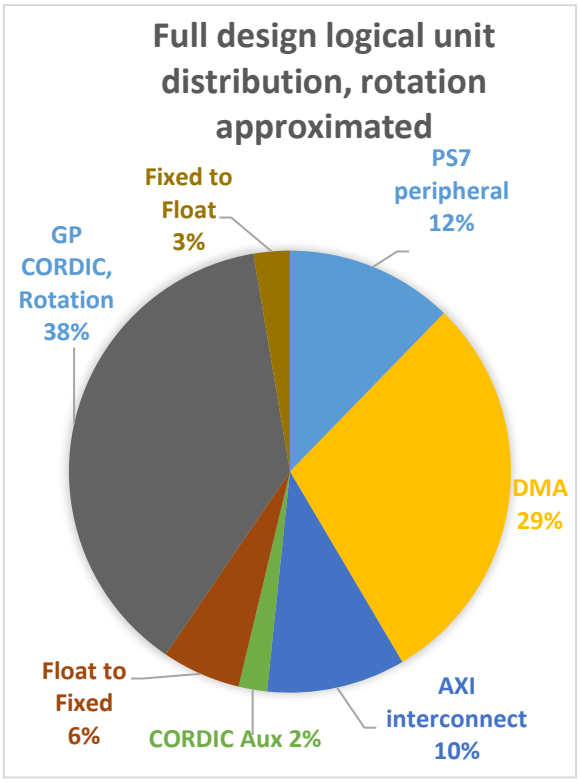


Fig.14. Resource utilization between the standard and approximated CORDIC cores

Next, the LUT and FF resource distribution among different parts of the design is considered. Figure 15 illustrates the PYNQ-Z1 device FPGA fabric‘s resource utilization for each primary component.



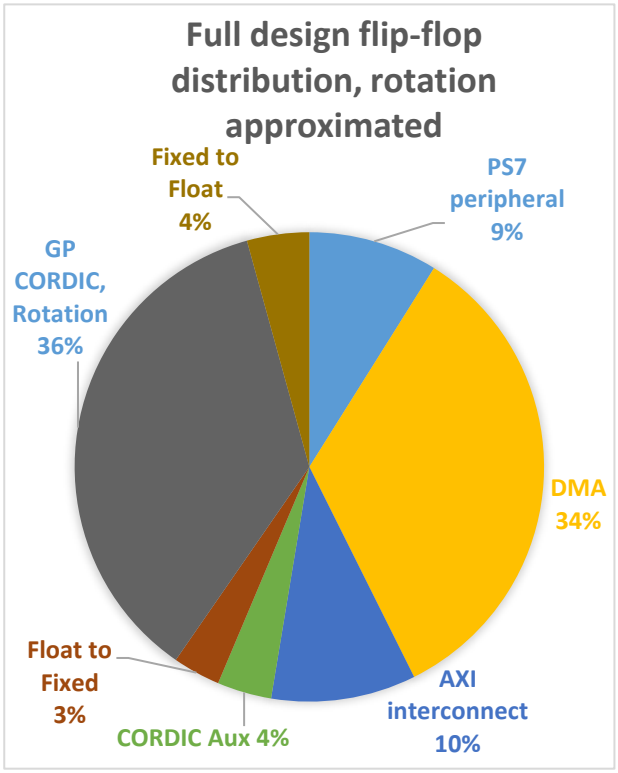


Fig. 15. LUT and FF distributions between different components of the design in Rotation mode with the approximation scheme

As can be seen, the General Purpose CORDIC cell, took up roughly half of both the total LUT and FF resources needed for the design, with the remaining half being the processing system (PS7) and its respective interconnection engines (AXI interconnect and DMA). This is for a single fully pipelined CORDIC core and in total takes up 13.07% of the entire PYNQ-Z1 board‘s LUT count, with the CORDIC core itself taking up 6.11%. Thus, a much larger array of CORDIC processors could be incorporated into the design for more complex operations.

Lastly, the % BRAM and LUT consumption for the CORDIC core itself is compared for the Vector CORDIC mode with the approximated bit count varied from 2 to 16. The results of this can be seen in Figure 16 and illustrates the exponential growth in BRAM utilization as larger lookup tables are required for the reciprocal computation. At 13 bits of precision, the percentage utilization is equal, suggesting 19 CORDIC stages with the remaining 13 bits approximated as the optimal ratio if optimizing for even resource utilization for 32 bits of precision.

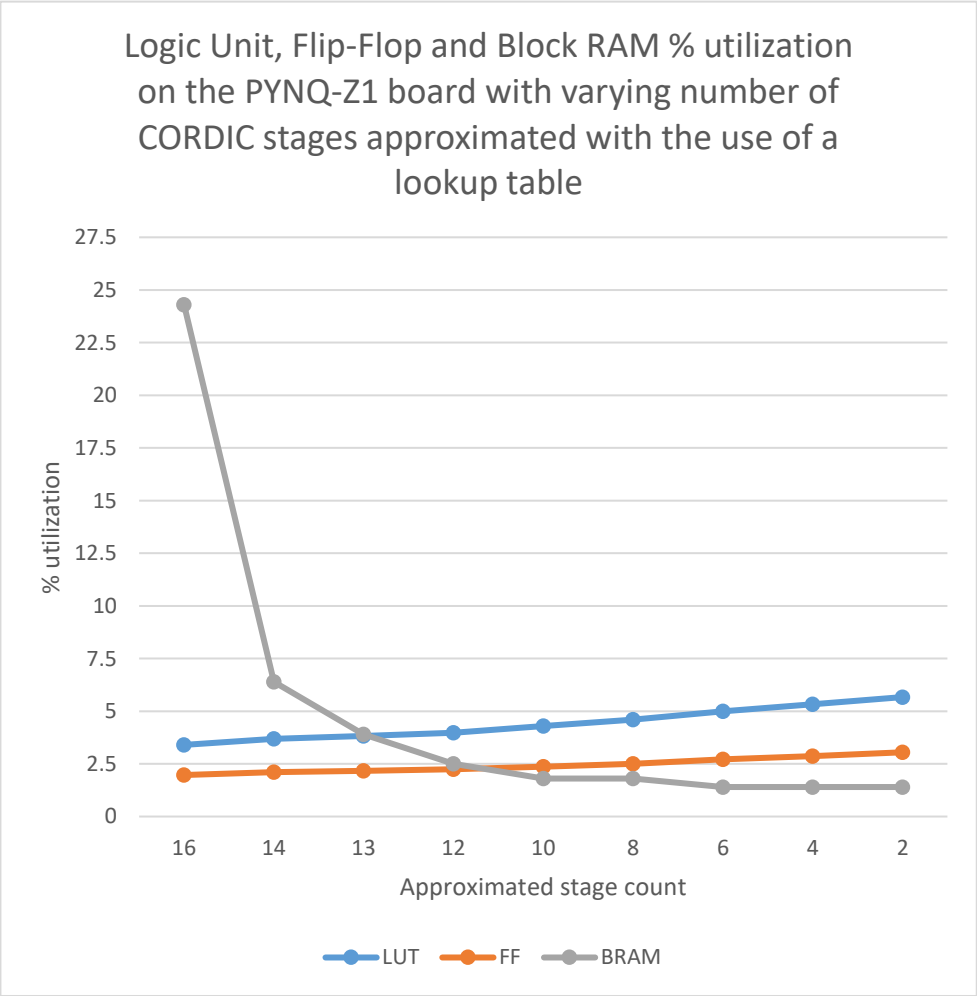


Fig. 16. LUT, FF and BRAM resource % utilization on the PYNQ-Z1 board for the bit approximating CORDIC vector mode, 13 bits being approximated leads to a roughly even utilization of LUT and BRAM

4.2. Throughput and precision

Each implementation was synthesized and routed for 100 MHz clock speeds with 64 bits total being handled each clock cycle.

After the final Vivado tool place and route operations, the design achieved a 1.032ns negative slack and a hold time of 0.029ns, meeting the 100 MHz clock restraint.

The PYNQ device's Jupyter Notebook environment test bench achieved 700 to 750 MB/s throughput using two DMA channels for reads and writes at 100 MHz clock speed. As seen in Figure 14, this requires around 2 MBs worth of data in the buffers. According to documentation, the maximum throughput under these conditions for the PYNQ-Z1 is 800 MB/s, thus all implementations managed to nearly fully saturate the DMA's maximum capacity. For this same reason, the testbench is limited to 100 MHz, due to additional clock rate increases not yielding a performance improvement. Running the same operations using Python's Numpy compute library on the PYNQ-Z1 board's Cortex-A9 ARM core yielded 20 MB/s throughput as a comparison.

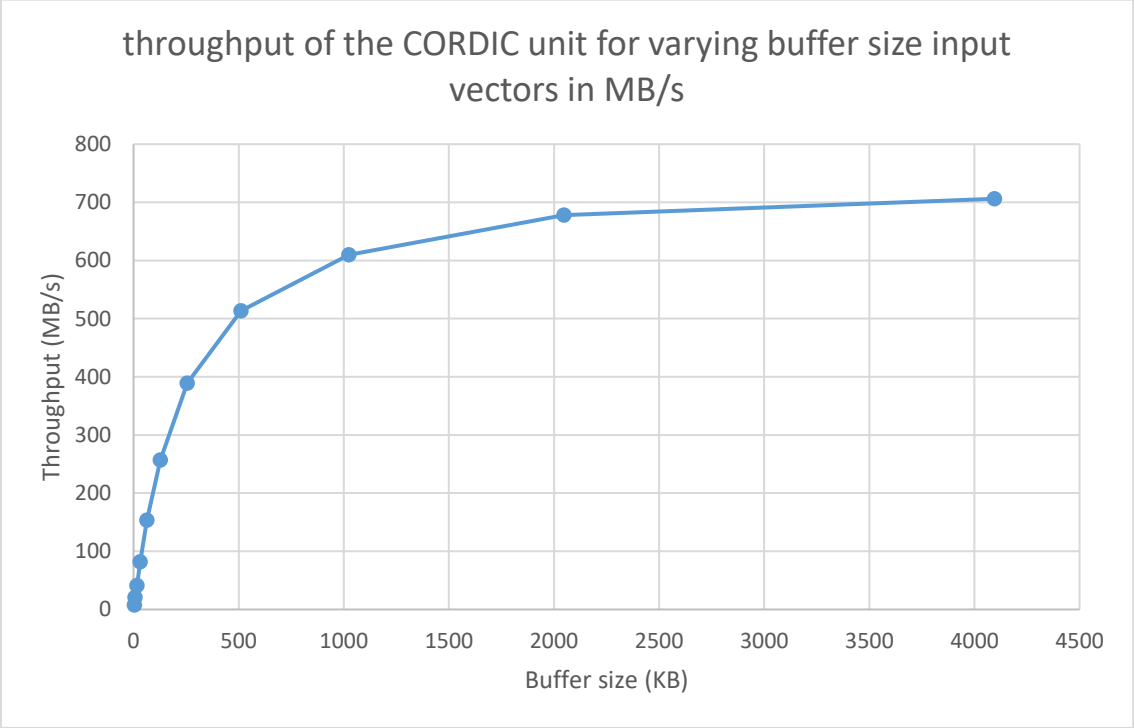


Fig. 17. CORDIC unit throughput to buffer size plot in MB/s. Around 1 MB of data is necessary for the DMA to start to get saturated

As an additional experiment, a software converter from floating point to normalized fixed-point format with 1 sign bit, 1 integer bit and 30 fractional bits was implemented and tested. This was done in software using the Numpy computing algorithm. Floating point - fixed point conversion operations were done by dividing the FP32 by a 32-bit max integer and shifting to the left and right by 1 for the input and output stages respectively. This leads to a 9-bit precision loss and is additionally a 20 MB/s throughput bottleneck using the Cortex-A9 processor ARM cores found in the PYNQ-Z1 board. The performance bottleneck motivated to introduce a hardware floating-fixed point conversion set of stages in the final design.

For the final SVD/QR algorithm acceleration tests, a mean squared error of $1e-23$ was achieved for each value upon return for each operation tested (sine, cosine, multiplication, division), which would make up a Givens Rotation computation as well as the matrix product.

The primary precision loss is assumed to be the floating to fixed point conversion, given that only 1 bit is used for storing the integer portion of the fixed point, given CORDIC‘s limitation. Thus, normalizing values to -2.0 - +2.0 in the standard 8 exponent, 23 mantissa bit format simply waste a great deal of precision due to not utilizing the full scope of the floating-point format.

Conclusions

1. Considering the results, it can be argued that when it comes to complex arithmetic operations, a generalized CORDIC unit allows the effective utilization of the entire programmable logic in tandem with the DSP units to substantially increase device throughput, which is a strong case for the use of CORDIC in a high-performance computing workload. Just two DSPs and 160 KB of BRAM are enough to decrease the required LUT and FF sizes by 40% in a standard CORDIC implementation.

2. CORDIC allows efficient utilization of BRAM as a lookup table for computing reciprocals by shrinking input data widths to a manageable size, with a 32-bit input having 16 bits handled with CORDIC allowing to allocate a lookup table approach for the remaining 16 bits at the cost of 25% of a PYNQ-Z1 device's available BRAM.

3. Significant size inputs are required for the effective utilization of CORDIC in a matrix decomposition context, with the PYNQ-Z1 board testbench requiring a full 2 MB of data for the input buffers to reach near-maximum capacity.

4. CORDIC shows great promise as an algorithm 'lost in time' with the potential to become a staple for FPGA accelerator design in the context of arithmetic-heavy compute workloads, especially when an algorithm can be implemented using trigonometric operations as the primary source of computational complexity.

List of references

[1] M. Ahmed and A. N. Islam, Deep learning: hope or hype, Annals of Data Science, pp. 1–6, 2020.

[2] H. J. Park, K. Chan, and R. Li, Systems, methods, apparatus, and computer-readable media for adaptive active noise cancellation, May 27 2014, uS Patent 8,737,636.

[3] A. Doering, H. Jäger, H. Witte, M. Galicki, C. Schelenz, M. Specht, K. Reinhart, and M. Eiselt, Adaptable preprocessing units and neural classification for the segmentation of eeg signals, Methods of information in medicine, vol. 38, no. 03, pp. 214–224, 1999.

[4] O. M. Parkhi, A. Vedaldi, and A. Zisserman, Deep face recognition, 2015.

[5] M. Al-Qizwini, I. Barjasteh, H. Al-Qassab, and H. Radha, Deep learning algorithm for autonomous driving using googlenet, in 2017 IEEE Intelligent Vehicles Symposium (IV). IEEE, 2017, pp. 89–96.

[6] Y. Li, W. Zhao, and J. Pan, Deformable patterned fabric defect detection with fisher criterion-based deep learning, IEEE Transactions on Automation Science and Engineering, vol. 14, no. 2, pp. 1256–1264, 2016.

[7] L. Stasytis, Using natural language processing in an instant messaging environment for user analysis, IVUS 2018, 2018.

[8] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, Bart: Denoising sequence-tosequence pre-training for natural language generation, translation, and comprehension, arXiv preprint arXiv:1910.13461, 2019.

[9] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray et al., A reconfigurable fabric for accelerating large-scale datacenter services, Communications of the ACM, vol. 59, no. 11, pp. 114–122, 2016.

[10] J. T. Dirk Koch. "fpgasort: A high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting conference: Proceedings of the acm/sigda 19th international symposium on field programmable gate arrays, fpga 2011, monterey, california, usa, february 27, march 1,2011".

[11] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung et al., Azure accelerated networking: Smartnics in the public cloud, in 15th fUSENIXg Symposium on Networked Systems Design and Implementation (fNSDIg 18), 2018, pp. 51–66.

[12] Tiobe Index programming language ranking https://www.tiobe.com/tiobe-index/ visited: 11/16/2020.

[13] J. Volder, The cordic computing technique, in Papers presented at the the March 3-5, 1959, western joint computer conference, 1959, pp. 257–261.

[14] I. K. Fodor, A survey of dimension reduction techniques, Lawrence Livermore National Lab., CA (US), Tech. Rep., 2002.

[15] M. Owaida, D. Sidler, K. Kara, and G. Alonso, Centaur: A framework for hybrid cpu-fpga databases, in 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE, 2017, pp. 211–218.

[16] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman et al., Serving dnns in real time at datacenter scale with project brainwave, IEEE Micro, vol. 38, no. 2, pp. 8–20, 2018.

[17] M. Owaida, G. Alonso, L. Fogliarini, A. Hock-Koon, and P.-E. Melet, Lowering the latency of data processing pipelines through fpga based hardware acceleration, Proceedings of the VLDB Endowment, vol. 13, no. 1, pp. 71–85, 2019.

[18] Z. Wang, J. Paul, H. Y. Cheah, B. He, and W. Zhang, Relational query processing on opencl-based fpgas, in 2016 26th International Conference on Field Programmable Logic and Applications (FPL). IEEE, 2016, pp. 1–10.
[19] R. Domingo, R. Salvador, H. Fabelo, D. Madroñal, S. Ortega, R. Lazcano, E. Juárez, G. Callicó, and C. Sanz, High-level design using intel fpga opencl: A hyperspectral imaging spatial-spectral classifier, in 2017 12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC). IEEE, 2017, pp. 1–8.
[20] Z. Jin, K. Yoshii, H. Finkel, and F. Cappello, Evaluation of the single-precision floating point vector add kernel using the intel fpga sdk for opencl, Argonne National Lab.(ANL), Argonne, IL (United States), Tech. Rep., 2017.
[21] O. Y. Al-Jarrah, P. D. Yoo, S. Muhaidat, G. K. Karagiannidis, and K. Taha, Efficient machine learning for big data: A review, Big Data Research, vol. 2, no. 3, pp. 87–93, 2015.
[22] Y. Cheng, D. Li, Z. Guo, B. Jiang, J. Lin, X. Fan, J. Geng, X. Yu, W. Bai, L. Qu et al., Dlbooster: Boosting end-to-end deep learning workflows with offloading data preprocessing pipelines, in Proceedings of the 48th International Conference on Parallel Processing, 2019, pp. 1–11.
[23] R. Schmidt, Multiple emitter location and signal parameter estimation, IEEE transactions on antennas and propagation, vol. 34, no. 3, pp. 276–280, 1986.
[24] J.-C. Chang and A.-C. Chang, Doa estimation in the time–space cdma system using modified particle swarm optimization, AEU-International Journal of Electronics and Communications, vol. 67, no. 4, pp. 340–347, 2013.
[25] M. Sánchez-Fernández, V. Jamali, J. Llorca, and A. Tulino, Gridless multidimensional angle of arrival estimation for arbitrary 3d antenna arrays, arXiv preprint arXiv:2008.12323, 2020.
[26] M. Strauss, P. Mordel, V. Miguet, and A. Deleforge, Dregon: Dataset and methods for uav-embedded sound source localization, in 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, 2018, pp. 1–8.
[27] J. Benesty, Adaptive eigenvalue decomposition algorithm for passive acoustic source localization, the Journal of the Acoustical Society of America, vol. 107, no. 1, pp. 384–391, 2000.
[28] I. Bravo, C. Vázquez, A. Gardel, J. L. Lázaro, and E. Palomar, High level synthesis fpga implementation of the jacobi algorithm to solve the eigen problem, Mathematical Problems in Engineering, vol. 2015.
[29] P. K. Meher, J. Valls, T.-B. Juang, K. Sridharan, and K. Maharatna, 50 years of cordic: Algorithms, architectures, and applications, IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 56, no. 9, pp. 1893–1907, 2009.
[30] 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, 1998, pp. 191–200.
[31] Google's Scikit use tutorial: https://towardsdatascience.com/deploying-Scikitlearn-models-at-scale-f632f86477b8.
[32] Joblib parallelism library for Python: https://joblib.readthedocs.io/en/latest/parallel.html.
[33] Google's Tensorflow use blog: https://developers.google.com/machinelearning/crash-course/first-steps-withTensorflow/toolkit.
[34] S. Hadjis and K. Olukotun, Tensorflow to cloud fpgas: Tradeoffs for accelerating deep neural networks, in 2019 29th International Conference on Field Programmable Logic and Applications (FPL). IEEE, 2019, pp. 360–366.
[35] Amazon Sagemaker: https://aws.amazon.com/blogs/machinelearning/preprocess-input-data-before-makingpredictions-using-amazon-Sagemaker-inferencepipelines-and-Scikit-learn/.
[36] Kaggle notebooks: https://www.kaggle.com/notebooks.

[37] Tensorflow – An end-to-end open-source machine learning platform, link: https://www.Tensorflow.org/ ,viewed 2022, May
[38] Scikit-learn machine learning in Python package, link: https://scikit-learn.org/stable/index.html ,viewed 2022, May
[39] Numpy numerical computing Python library, link: https://numpy.org ,viewed 2022, May
[40] Kaggle competition problem with image augmentation, link: https://www.kaggle.com/pedrolcn/imageaugmentation-with-keras-preprocessing-api ,viewed 2022, May
[41] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, Cupy: A Numpy-compatible library for nvidia gpu calculations, in Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS), 2017.
[42] Gpupy: Transparently and efficiently using a gpu for numerical computation in Python, benjamin eitzen and robert r. lewis school of eecs washington state university.
[43] The pyroomacoustics audio array processing python package. Link: https://github.com/LCAV/pyroomacoustics ,viewed 2022, May
[44] Vitis Model Composer toolbox by Xilinx, link: https://www.xilinx.com/products/design-tools/vitis/vitis-model-composer.html ,viewed 2022, May
[45] Intel Quartus Prime development toolchain by Intel, link: https://www.intel.com/content/www/us/en/software/programmable/overview.html ,viewed 2022, May
[46] The NI LabVIEW high-performance FPGA developer's guide, link: https://www.ni.com/en-us/support/documentation/supplemental/13/the-ni-labview-high-performance-fpga-developer-s-guide.html ,viewed 2022, May
[47] Lattice Diamond FPGA development toolchain, link: https://www.latticesemi.com/latticediamond ,viewed 2022, May
[48] Synopsys FPGA-Based Design synthesis solution, link: https://www.synopsys.com/implementation-and-signoff/fpga-based-design.html ,viewed 2022, May
[49] Siemens EDA tool by Siemens, link: https://eda.sw.siemens.com/en-US/ ,viewed 2022, May
[50] Cadence FPGA development flow tool, link: https://www.cadence.com/en_US/home/solutions/fpga-development.html ,viewed 2022, May
[51] T. Bourgeat, C. Pit-Claudel, and A. Chlipala, The essence of bluespec: a core language for rule-based hardware design, in Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, 2020, pp. 243–257.
[52] Catapult synthesis datasheet, 10th ed., mentor graphics, 2006.
[53] HDL Coder hardware generation tool by Mathworks, link: https://www.mathworks.com/products/hdl-coder.html ,viewed 2022, May
[54] D. Pursley and T.-H. Yeh, High-level low-power system design optimization, in 2017 International Symposium on VLSI Design, Automation and Test (VLSIDAT). IEEE, 2017, pp. 1–4.
[55] S. Skalicky, J. Monson, A. Schmidt, and M. French, Hot & spicy: Improving productivity with Python and hls for fpgas, in 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE, 2018, pp. 85–92.
[56] M. W. Numan, B. J. Phillips, G. S. Puddy, and K. Falkner, Towards automatic high-level code deployment on reconfigurable platforms: A survey of high-level synthesis tools and toolchains, IEEE Access, vol. 8, pp. 174 692–174 722, 2020.
[57] Chisel Hardware development language by the University of California, Berkeley, link: https://github.com/freechipsproject/chisel3 ,viewed 2022, May

[58] Hardware Description Language Migen, by the Migen contributors, link: https://github.com/m-labs/migen ,viewed 2022, May
[59] Hardware SoC design framework Litex by Enjoy-Digital, link: https://github.com/enjoy-digital/litex ,viewed 2022, May
[60] F. Kermarrec, S. Bourdeauducq, H. Badier, and J.-C. Le Lann, Litex: an open-source soc builder and library based on migen Python dsl, in OSDA 2019, colocated with DATE 2019 Design Automation and Test in Europe, 2019.
[61] Python Filter Design Analysis Tool by pyFDA Project Contributors. Link: https://github.com/chipmuenk/pyfda/ ,viewed 2022, May
[62] Z. Drmac, Algorithm 977: A qr–preconditioned qr-svd method for computing the svd with high accuracy, ACM Transactions on Mathematical Software(TOMS), vol. 44, no. 1, pp. 1–30, 2017.
[63] S. Wold, K. Esbensen, and P. Geladi, Principal component analysis, Chemometrics and intelligent laboratory systems, vol. 2, no. 1-3, pp. 37–52, 1987.
[64] Z. Drmac and K. Veseli ˇ c, New fast and accurate jacobi svd algorithm. i, SIAM Journal on matrix analysis and applications, vol. 29, no. 4, pp. 1322–1342, 2008.
[65] F. Ling, Givens rotation based least squares lattice and related algorithms, IEEE Transactions on Signal Processing, vol. 39, no. 7, pp. 1541–1551, 1991.
[66] J. S. Walther, A unified algorithm for elementary functions, in Proceedings of the May 18-20, 1971, spring joint computer conference, 1971, pp. 379–385.
[67] E. Antelo, J. Villalba, J. D. Bruguera, and E. L. Zapata, High performance rotation architectures based on the radix-4 cordic algorithm, IEEE Transactions on Computers, vol. 46, no. 8, pp. 855–870, 1997.
[68] Y. H. Hu and S. Naganathan, An angle recoding method for cordic algorithm implementation, IEEE Transactions on Computers, vol. 42, no. 1, pp. 99–102, 1993.
[69] S. Wang, V. Piuri, and E. Wartzlander, Hybrid cordic algorithms, IEEE Transactions on Computers, vol. 46, no. 11, pp. 1202–1207, 1997.
[70] N. Takagi, T. Asada, and S. Yajima, Redundant cordic methods with a constant scale factor for sine and cosine computation, IEEE Computer Architecture Letters, vol. 40, no. 09, pp. 989–995, 1991.
[71] B. Lakshmi and A. Dhar, Cordic architectures: A survey, VLSI design, vol. 2010, 2010.
[72] Z. Liu, K. Dickson, and J. McCanny, Applicationspecific instruction set processor for soc implementation of modern signal processing algorithms, IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 52, no. 4, pp. 755–765, 2005.
[73] Google trends worldwide for the keyword ‘cordic algorithm’ from 2004 to 2022 https://trends.google.com/trends/explore?date=all&q=cordic%20algorithm ,viewed 2022, May
[74] P. Coussy and A. Morawiec, High-level synthesis. Springer, 2010, vol. 1.
[75] R. Nikhil, Bluespec system verilog: efficient, correct rtl from high level specifications, in Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE’04. IEEE, 2004, pp. 69–70.
[76] E. Antelo, J. Villalba, and E. L. Zapata, A lowlatency pipelined 2d and 3d cordic processors, IEEE Transactions on Computers, vol. 57, no. 3, pp. 404–417, 2008.
[77] H. M. F. Lohstroh, Reactors: A deterministic model of concurrent computation for reactive systems, Ph.D. dissertation, UC Berkeley, 2020.
[78] V. Novakovic and S. Singer, A kogbetliantz-type ´ algorithm for the hyperbolic svd, Numerical Algorithms, pp. 1–39, 2021.

[79] V. Hari and V. Zadelj-Martic, Parallelizing the kogbetliantz method: a first attempt, J. Num. Anal. Industr. Appl. Math, vol. 2, pp. 49–66, 2007.
[80] C. Shi and R. W. Brodersen, Floating-point to fixedpoint conversion, 2004.
[81] The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc. Crockett, Louise H and Elliot, Ross A and Enderwitz, Martin A and Stewart, Robert W, 2014, Strathclyde Academic Media
[82] Kennedy, James, and Russell Eberhart. "Particle swarm optimization." Proceedings of ICNN'95-international conference on neural networks. Vol. 4. IEEE, 1995.
[83] Wold, Svante, Kim Esbensen, and Paul Geladi. "Principal component analysis." Chemometrics and intelligent laboratory systems 2.1-3 (1987): 37-52.
[84] Van Der Maaten, Laurens, Eric Postma, and Jaap Van den Herik. "Dimensionality reduction: a comparative." J Mach Learn Res 10.66-71 (2009): 13.
[85] Huang, Kai, et al. "An efficient FPGA implementation for 2-D MUSIC algorithm." Circuits, Systems, and Signal Processing 35.5 (2016): 1795-1805.
[86] Argentieri, Sylvain, and Patrick Danes. "Broadband variations of the MUSIC high-resolution method for sound source localization in robotics." 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE, 2007.
[87] Kim, Minseok, Koichi Ichige, and Hiroyuki Arai. "Implementation of FPGA based fast DOA estimator using unitary MUSIC algorithm [cellular wireless base station applications]." 2003 IEEE 58th Vehicular Technology Conference. VTC 2003-Fall (IEEE Cat. No. 03CH37484). Vol. 1. IEEE, 2003.
[88] Björck, Åke. Numerical methods for least squares problems. Society for Industrial and Applied Mathematics, 1996.
[89] Lay, David C. (2016). Linear algebra and its applications. Steven R. Lay, Judith McDonald (Fifth Global ed.). Harlow. p. 142. ISBN 1-292-09223-8. OCLC 920463015.
[90] Leon, Steven J., Åke Björck, and Walter Gander. "Gram-Schmidt orthogonalization: 100 years and more." Numerical Linear Algebra with Applications 20.3 (2013): 492-532.
[91] George, Alan, and Joseph WH Liu. "Householder reflections versus Givens rotations in sparse orthogonal decomposition." Linear Algebra and its Applications 88 (1987): 223-238.
[92] Cooijmans, Tim, et al. "Recurrent batch normalization." arXiv preprint arXiv:1603.09025 (2016).
[93] Versal Architecture White Paper, Link: https://docs.xilinx.com/v/u/en-US/wp518-ai-edge-intro, viewed 2022, May
[94] Andrew, Robert, and Nicholas Dingle. "Implementing QR factorization updating algorithms on GPUs." Parallel Computing 40.7 (2014): 161-172.
[95] Khurshid, Burhan, and Javeed Jeelani Khan. "An Efficient Fixed-Point Multiplier Based on CORDIC Algorithm." Journal of Circuits, Systems and Computers 30.05 (2021): 2150080.
[96] AXI Stream tutorial blogpost by Cluadio Avi Chami, link: http://fpgasite.blogspot.com/2017/07/xilinx-axi-stream-tutorial-part-1.html ,viewed 2022, May
[97] Xilinx AXI DMA LogiCORE IP Protduct Guide (PG021), Figure: AXI DMA Block Diagram. Link: https://docs.xilinx.com/r/en-US/pg021_axi_dma/Overview, viewed 2022, May