Hardware-Accelerated Algorithm for Complex Function Roots Density Graph Plotting

Ruibai Tang
Tsinghua University
Haidian Qu, Beijing Shi, China
trb21@tsinghua.org.cn

Chengbin Quan
Tsinghua University
Haidian Qu, Beijing Shi, China
quancb@tsinghua.edu.cn

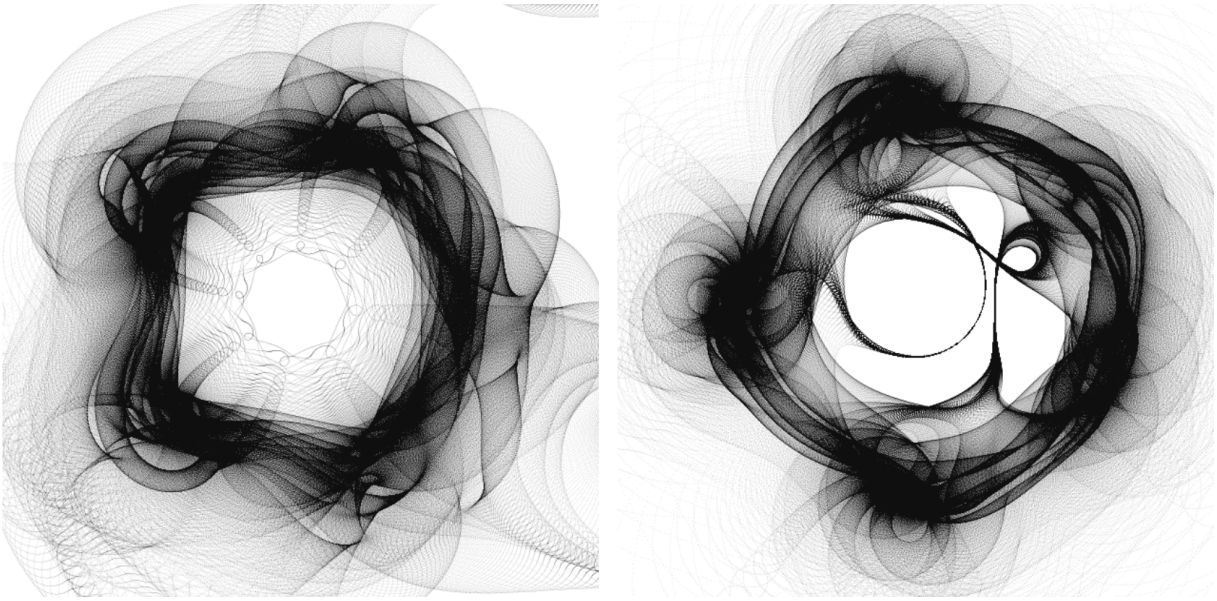

Figure 1: Roots Density Graph of Two 5-degree Polynominals with Variable Parameters as Coefficients

Abstract

Solving and visualizing the potential roots of complex functions is essential in both theoretical and applied domains, yet often computationally intensive. We present a hardware-accelerated algorithm for complex function roots density graph plotting by approximating functions with polynomials and solving their roots using single-shift QR iteration. By leveraging the Hessenberg structure of companion matrices and optimizing QR decomposition with Givens rotations, we design a pipelined FPGA architecture capable of processing a large amount of polynomials with high throughput. Our implementation achieves up to 65× higher energy efficiency than CPU-based approaches, and while it trails modern GPUs in performance due to differences in fabrication technique.

1 Introduction

In mathematical analysis, investigating the properties of solutions to equations—particularly the existence of such solutions—is a fundamental and central problem. Whether in the context of pure mathematical theory [8, 10, 19, 23, 30, 42] or in the construction of models in physics, engineering, and applied sciences [14, 17, 33], the validity of many theoretical inferences critically depends on the existence of solutions to underlying equations. While for certain specific equations, one may hope to derive an explicit analytic solution using algebraic manipulation or elementary integration, in practice, the available mathematical tools are often insufficient to express such solutions in terms of elementary functions. This challenge is especially prominent for nonlinear equations, high-dimensional partial differential equations, and problems involving complex boundary conditions.

In the observation stage of equation solutions, researchers frequently resort to numerical methods by discretizing the problem and applying computational algorithms to approximate the distribution of solutions over the domain. However, numerical approaches typically face two key challenges. **First**, as the discretization becomes finer, the number of sampling points increases dramatically, leading to large-scale scientific computations that are often time-consuming. **Second**, the numerical process may generate a vast amount of raw output data, which requires additional processing and analysis to extract meaningful insights or to inspire further innovative research. Some of these tasks, such as data visualization, are particularly common and labor-intensive.

2 Problem Description

Consider the following general problem: given a complex function $f(z)$ defined over a subset $D \subset \mathbb{C}$, visualize the distribution of its zeros, i.e., the solutions to $f(z) = 0$. A straightforward approach is to partition the domain D into a large number of small subregions

D_i, and within each region, numerically approximate f using a sequence of polynomial basis to obtain an approximate polynomial g, such that $||f - g|| < \varepsilon$. If g has a root within D_i, then f is likely to have a root in the same region, and the distribution of zeros of g serves as a good approximation to that of f. Thus, the problem reduces to solving the roots of a large number of polynomials g.

2.1 Solving the Roots of a Polynomial

For a complex polynomial $P(z) = z^n + a_{n-1}z^{n-1} + ... + a_0$, according to linear algebra knowledge, the eigenvalues of its Frobenius companion matrix, as shown in Equation (1), correspond precisely to the roots of the polynomial. Therefore, instead of solving the polynomial roots directly, one can equivalently compute the eigenvalues of its associated companion matrix.

$$\begin{pmatrix} -a_{n-1} & -a_{n-2} & \dots & -a_1 & -a_0 \\ 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ & & \ddots & & \\ 0 & 0 & & 0 & 0 \\ 0 & 0 & \dots & 1 & 0 \end{pmatrix} \tag{1}$$

In numerical computation, the problem of computing the eigenvalues of a matrix is a well-established and extensively studied topic, with numerous mature solution techniques available. [6, 11, 29, 37] Among them, the **QR iteration with single shift**, its pseudocode is shown in Algorithm (1), is known for its rapid convergence properties to solve all eigenvalues of a small matrix. This method introduces a shift (typically near an estimated eigenvalue) to accelerate the convergence of the QR iteration process. The overall computational complexity of the algorithm depends on the matrix size n, the number of iterations `iter`, and the cost of performing QR decomposition at each step.

Algorithm 1 QR Iteration with Single Shift

Input: $A \in M_n(\mathbb{C})$ $(a_{i,j}, 0 \le i, j \le n-1)$
Output: $eig[0...n-1]$

$m \leftarrow n$
while $m \ge 2$ **do**
  for $i = 0, 1, ..., \texttt{iter}$ **do**
    $s \leftarrow a_{m-1,m-1}$
    $A \leftarrow A - sI_m$
    $(Q, R) \leftarrow \text{QR\_DECOMP}(A)$
    $A \leftarrow RQ + sI_m$
  end for
  $eig[m-1] \leftarrow a_{m-1,m-1}$
  $m \leftarrow m - 1$
  $A \leftarrow A[0...m-1][0...m-1]$
end while
$eig[0] \leftarrow a_{0,0}$
return $eig[0...n-1]$

2.2 QR Decomposition using Givens Rotation

The **QR decomposition** of a complex matrix A refers to its factorization in the form $A = QR$, where Q is a unitary matrix satisfying $QQ^H = I$, and R is an upper triangular matrix. One effective method for performing QR decomposition is the **Givens rotation** [11], which applies a sequence of plane rotations to zero out selected elements. Each Givens rotation has the following form:

$$\begin{pmatrix} c & s \\ -\bar{s} & \bar{c} \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \sqrt{|a|^2 + |b|^2} \\ 0 \end{pmatrix}$$

where

$$c = \frac{\bar{a}}{\sqrt{|a|^2 + |b|^2}}, \quad s = \frac{\bar{b}}{\sqrt{|a|^2 + |b|^2}}$$

In our problem, the companion matrix A, constructed from the coefficients of the polynomial, has a highly structured form. Specifically, A is a **Hessenberg matrix**, as shown in Equation (2). It can be proved that throughout the execution of Algorithm (1), the matrix A preserves its Hessenberg structure at every iteration step. This structural invariance allows for a more compact storage representation and the use of simpler algorithms for matrix operations.

$$\begin{pmatrix} \times & \times & \dots & \times & \times \\ \times & \times & \dots & \times & \times \\ & \times & \dots & \times & \times \\ & & \ddots & \times & \times \\ & & & \times & \times \end{pmatrix} \tag{2}$$

By applying a sequence of Givens rotations, we can sequentially eliminate the subdiagonal entries of A below the main diagonal. Let Q_i denote the unitary matrix representing the Givens rotation applied at step i, where $i = 1, 2, ..., n-1$.

$$Q_i = \begin{pmatrix} I_{i-1} & & & \\ & c_i & s_i & \\ & -\bar{s_i} & \bar{c_i} & \\ & & & I_{n-1-i} \end{pmatrix}$$

Then the operations of Givens rotation will affect matrix A as follows:

$$Q_1 \begin{pmatrix} \times & \times & \dots & \times & \times \\ \times & \times & \dots & \times & \times \\ & \times & \dots & \times & \times \\ & & \ddots & \times & \times \\ & & & \times & \times \end{pmatrix} = \begin{pmatrix} \times & \times & \dots & \times & \times \\ 0 & \times & \dots & \times & \times \\ & \times & \dots & \times & \times \\ & & \ddots & \times & \times \\ & & & \times & \times \end{pmatrix} \tag{3}$$

$$Q_2 \begin{pmatrix} \times & \times & \dots & \times & \times \\ & \times & \dots & \times & \times \\ & \times & \dots & \times & \times \\ & & \ddots & \times & \times \\ & & & \times & \times \end{pmatrix} = \begin{pmatrix} \times & \times & \dots & \times & \times \\ & \times & \dots & \times & \times \\ & 0 & \dots & \times & \times \\ & & \ddots & \times & \times \\ & & & \times & \times \end{pmatrix} \tag{4}$$

$$Q_{n-1} \begin{pmatrix} \times & \times & \dots & \times & \times \\ & \times & \dots & \times & \times \\ & & \ddots & \times & \times \\ & & & \times & \times \\ & & & \times & \times \end{pmatrix} = \begin{pmatrix} \times & \times & \dots & \times & \times \\ & \times & \dots & \times & \times \\ & & \ddots & \times & \times \\ & & & \times & \times \\ & & & 0 & \times \end{pmatrix} \tag{5}$$

In the Equation (3)(4)(5), the entries marked in blue indicate the value at that position may be changed as a result of the Givens rotation, while the remaining elements remain numerically unaffected.

Now we have:

$$(Q_{n-1} \cdots Q_2 Q_1)A = R$$

Moving the product $Q^H = Q_{n-1} \cdots Q_2 Q_1$ to the right-hand side yields the QR decomposition $A = QR$.

However, in practice, each iteration in the Algorithm (1) includes the following step:

$$A \leftarrow RQ = (Q_{n-1}...Q_2Q_1)A(Q_1^H Q_2^H ...Q_{n-1}^H) \tag{6}$$

As a result, the QR decomposition does not need to be explicitly performed. The matrix A can be updated directly by accumulating the Givens rotation in-place, like Equation (3)(4)(5). That is, Operation (6) can be dived into two procedures $A \leftarrow (Q_{n-1}...Q_2Q_1)A$ and $A \leftarrow A(Q_1^H Q_2^H ...Q_{n-1}^H)$. During the first procedure, every time performing $A \leftarrow Q_i A$, we only need to update **two rows** of matrix A. During the second procedure, every time performing $A \leftarrow AQ_i^H$, similarly, we only need to update **two columns** of matrix A. Note that A is an upper triangular matrix at the end of the first procedure, it is not difficult to figure out that A is still in the form of Hessenberg matrix during the second procedure.

3 Architecture of Our Design

To accelerate both the root-finding process and the visualization process, we propose an open-source hardware-accelerated algorithm architecture that processes a batch of N polynomials—typically $N \sim 10^9$ or more—with high throughput. This architecture efficiently computes and visualizes the aggregated distribution of all polynomial roots, offering a scalable and computationally efficient solution to the zero-distribution visualization problem.

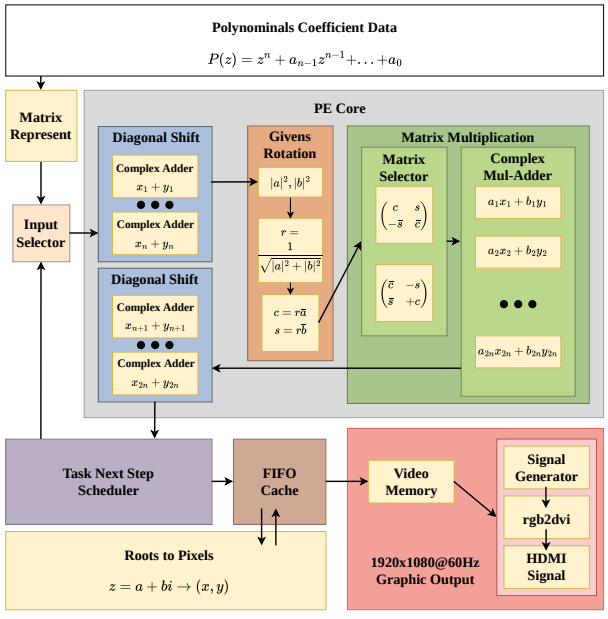

Figure 2: Architecture of Our Design

Figure 2 illustrates the architecture of our design. We employ a Task Next Step Scheduler to manage and update the algorithm

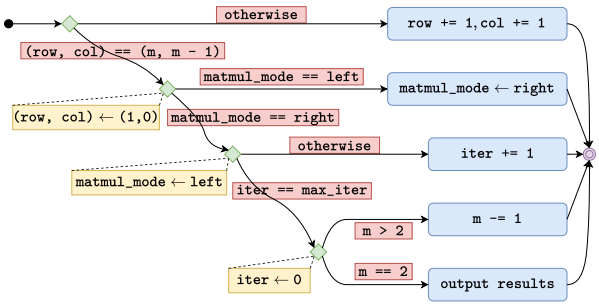

Figure 3: Status Transition Logic in Task Next Step Scheduler

progress of each matrix. The PE (Processing Element) Core is a non-blocking pipeline. During each pass, one or more internal modules are activated to operate on the matrix, while the inactive modules effectively serve as data buffers. This design eliminates the need for an additional memory pool to store intermediate states.

At the input port of the PE Core, whenever a pipeline gap is detected, the Input Selector prioritizes accepting a task issued from the Task Next Step Scheduler. Otherwise, it fetches a new input from the external source and initiates a new task of Algorithm (1). If a task satisfies the termination condition after a pass to the PE Core, the eigenvalue results will be written into the FIFO cache within only one cycle to avoid pipeline blocking. Finally, the results stored in FIFO cache will be converted into pixel coordinates on the screen and be written into the video memory for visualizing.

The PE core is composed of four major modules in the pipeline order: 1. Diagonal Shift (Subtract), 2. Givens Rotation, 3. Matrix Multiplication, 4. Diagonal Shift (Add).

The Diagonal Shift modules are triggered only when the input matrix is at the beginning or end of a QR iteration; they add or subtract the shift value $s = a_{m-1,m-1}$ to the diagonal values of the matrix. The Givens Rotation module calculates a pair of rotation parameters (c, s) based on matrix entries located at positions (i, j) and $(i-1, j)$. The Matrix Multiplication module is specialized to perform only $ax + by$ operations, reflecting the fact that we only apply the 2×2 submatrices of Q_i or Q_i^H, which affects only two rows or two columns.

Naturally, a single pass through the PE Core is not sufficient to complete all operations of the algorithm. We illustrate this with a simulation of a single QR iteration in the Algorithm (1).

During the first pass of matrix A through the PE Core, the following modules are activated: Diagonal Shift (Subtract), Givens Rotation, and Matrix Multiplication:

$$A \leftarrow A - sI_m$$
$$A \leftarrow Q_1 A$$

For the subsequent passes $i = 2, 3, \ldots, m-1$, only the Givens Rotation and Matrix Multiplication modules are activated:

$$A \leftarrow Q_i A$$

After the $(m-1)$th pass, matrix A is transformed into an upper triangular matrix R. To conserve resources, we overwrite matrix A directly without explicitly storing matrices R or Q. However, We

have to retain the sequence of computed Givens coefficients $(c_i,\ s_i)$ for later passes.

For passes $m-1+i$ with $i = 1, 2, \ldots, m-1$, we reuse the previously computed coefficients $(c_i,\ s_i)$ to multiply Q_i^H on the right:

$$A \leftarrow AQ_i^H$$

After the last matrix multiplication in the $(2m-2)$th pass, the Diagonal Shift (Add) module is also triggered:

$$A \leftarrow A + sI_m$$

Then, the Task Next Step Scheduler may choose to begin a new QR iteration, reduce the problem size m, or output the results to FIFO cache. Figure 3 shows the detailed function of the Task Next Step Scheduler.

Finally, there for sure can be multiple PE Cores for parallel computing as long as the FIFO cache has enough capability to handle the stream of the results output. In our experiments, we only employ one PE core and one bus for the FIFO cache to write the video memory.

4 Evaluation

4.1 Pipeline Efficiency Analysis

Assuming the PE Core pipeline consists of N_p stages and advancing one stage takes one clock cycle. Therefore, at most N_p inputs can be fetched consecutively from the external source before the pipeline must wait for internal processing to complete. If a single matrix must pass through the pipeline K times to complete its computation, then the total number of cycles from the first input of a batch to the last output is given by:

$$C_{\text{batch}} = (K+1)N_p$$

The average number of cycles per input is therefore:

$$C = \frac{1}{N_p} C_{\text{batch}} = K + 1$$

However, due to the design of the input selector, a new batch of inputs can immediately follow the previous batch's output, which hides the startup latency of N_p cycles. As a result, the true average cycle count per input is:

$$C = K$$

We now analyze the composition of K. Let the original size of the input matrix be n, and the current size be m, the same as in the Algorithm (1). Each matrix size level m undergoes QR iterations of T times. Each iteration requires $2m-2$ passes through the pipeline. When $m = 2$, after all iterations are complete, the process finishes, we have:

$$K = T \sum_{m=2}^{n} (2m-2) = n(n-1)T$$

In our experiments, we set $n = 6$ and $T = 10$, which is sufficient to achieve six digits of precision in the computed roots, satisfying the accuracy requirements. This results in:

$$C = K = 300$$

This means that computing all complex roots of a 6-degree complex polynomial requires at most **300 clock cycles** in average in our hardware design using only one PE Core.

4.2 Implementation

We Implemented the hardware design using `SystemVerilog` and carried out simulation, synthesis, and implementation using `Xilinx Vivado 2019.2` [3] . The target FPGA device is `XC7A200T-2FBG484I` [4] . The project is **open-sourced** on GitHub to facilitate inspection and reuse by readers; the repository link is at [34]. Detailed resource utilization for each module is shown in Table 1. Area estimations without routing fabric are based on the parameters listed in Table 2, the parameters are calculated by counting the pixels as an example shown in Figure 4, and then scaling to the real FPGA size $23 \times 23mm^2$ provided by Xilinx Documentation [4]. An overview of the overall design is presented in Table 3.

It's clear that the Matrix Multiplication and the Video Memory module are the most resource-consuming and area-intensive components in the design. Specifically, the Matrix Multiplication module consumes 61% of the LUTs, 65% of the registers/flip-flops, and 75% of the DSPs, compared to the whole resource used by our design, occupying an estimated area of $18.53mm^2$. The Video Memory module utilizes 256 BRAM tiles, corresponding to an area of $15.50mm^2$. Note that the routing fabric is not included in the area estimation.

4.3 Experiment and Comparison

We implemented two versions of Algorithm 1 using C/C++ and CUDA, targeting execution on the CPU and GPU, respectively. Both implementations employ Givens rotation for QR decomposition, and the matrix multiplication follows the same optimization strategy as the hardware design: only the necessary two rows and two columns are involved in operations of the form $ax + by$, and neither the Q nor R matrices are explicitly stored. All computations are performed directly on the input matrix A. The algorithm and parameter settings are identical to those used in the hardware implementation.

Table 4 summarizes the environment configuration used for experiments, including the operating system, CPU and GPU models, compiler versions, and compilation flags. For the CPU implementation, OpenMP (`#pragma omp parallel for`) [9] is used to process different input polynomials in parallel. The code is compiled with `g++` using the `-Ofast` optimization flag. For the GPU implementation, each input polynomial is processed by a single thread within a thread block of 256 threads. Assuming there are M input polynomials, then the CUDA kernel will launch $\lceil M/256 \rceil$ thread blocks which will be scheduled by the GPU later. The code is compiled using `nvcc` with the optimization flags `-O3 -arch=sm_80 -use_fast_math`.

All experiments are conducted on an Arch Linux x86_64 operating system (kernel version 6.12.8-arch1-1). Power consumption during runtime is monitored by using commands

```
cat /sys/class/powercap/intel-rapl:0/energy_uj
nvidia-smi -query-gpu=power.draw -format=csv
```

for the CPU and GPU, respectively. The measured runtime power is 34.6W/70W for CPU/GPU. Based on the throughput of the number of input polynomials, we compute both the performance and energy efficiency of the CPU and GPU implementations, and compare them with our hardware design, as shown in Table 5.

It can be found that our design outperforms the CPU in both throughput and energy efficiency, achieving 65× higher energy

Table 1: Resource Usage of Modules (Routing Fabric not Included)

Module	Slice LUTs	Slice Registers	Block RAM Tile	DSPs	Area (mm^2)
Total Usage	48859	74283	268	352	44.47
PE Core	40886 (84%)	62796 (85%)	-	316 (90%)	24.25
- Diagonal Shift (Subtract)	3514 (7%)	5482 (7%)	-	12 (3%)	1.79
- Givens Rotation	3996 (8%)	4102 (6%)	-	28 (8%)	2.12
- Matrix Multiplication	**29712 (61%)**	**47992 (65%)**	-	**264 (75%)**	**18.53**
- Diagonal Shift (Add)	3664 (7%)	5220 (7%)	-	12 (3%)	1.81
Roots to Pixels	6655 (14%)	10080 (14%)	-	36 (10%)	3.62
Video Memory	468 (1%)	14 (0%)	**256 (96%)**	-	**15.50**
FIFO Cache	287 (1%)	214 (0%)	12 (4%)	-	0.82
Others	563 (1%)	1179 (2%)	-	-	0.28

Table 2: Parameters Used for Area Estimation

LUT	Reg/FF	BRAM	DSP
$291\mu m^2$	$96\mu m^2$	$0.06mm^2$	$0.02mm^2$

Table 3: Overall Information of Implementation

Check List	Information
Platform	Xilinx Vivado 2019.2
FPGA	XC7A200T-2FBG484I
Technique	28 *nm*
Frequency	100 MHz
Frequency (Video)	148.5 MHz
Precision	FP32
Polynomial Degree	n = 6
QR Iterations	T = 10
Power (PE Core)	1.43 W
Power (Total)	2.22 W
Average Performance	13.93 GFLOP/s
Ceiling Performance	20.40 GFLOP/s
Energy Effificiency	9.74 GFLOP/(s·W)

efficiency. However, our implementation falls significantly short compared to the GPU. In particular, our design delivers only about one-third the energy efficiency of the GPU. This discrepancy is likely attributed to differences in fabrication technique: our FPGA is manufactured using a 28nm process, whereas the GPU adopts a more advanced 8nm [26] node. As a result, the interconnect delay between logic elements on the FPGA is higher, which constrains our maximum operating frequency, which makes us unable to operate at higher frequencies, such as 1GHz.

Furthermore, we experimentally evaluated an alternative Matrix Multiplication module design and identified critical inefficiencies. In our current architecture, we utilize $2n$ complex mul-add units to simultaneously process two rows or two columns of the matrix. However, as illustrated in Equation (5), during operations such as left-multiplying by Q_{n-1}, only three positions require computation, makes the other $2n-3$ units unused. This may lead to additional power overhead.

Table 4: Experiment Settings

Environment	Settings
OS	Arch Linux x86_64 (6.12.8-arch1-1)
CPU	11th Gen Intel i7-11800H (16) @ 4.600GHz
GPU	NVIDIA GeForce RTX 3060 Mobile / Max-Q
`g++` (14.2.1)	`-Ofast -fopenmp`
`nvcc` (12.6.85)	`-O3 -arch=sm_80 -use_fast_math`

Table 5: Experiment Results

Device	Throughput/s	GFLOP/s	GFLOP/(s·W)
CPU	1.22×10^5	5.11	0.15
GPU	5.01×10^7	2089.50	29.85
FPGA	3.33×10^5	13.93	9.74

As an alternative, we considered a low-resource design employing only 2 complex mul-add units. To support this, we would need to increase the status granularity of the Task Next Step Scheduler such that only two positions from two rows or two columns are processed at a time, continuing until the entire row or column is complete. This approach, however, increases the average number of cycles per input polynomial to:

$$C' = K' = T \sum_{m=2}^{n} \left(2 \sum_{k=2}^{m} k \right) = \frac{T}{3} n(n^2 + 3n - 4) = 1000.$$

The resulting throughput drops to 30% of the original, and although the PE Core power consumption is reduced to approximately 0.68W (i.e., 48% of the original), the overall energy efficiency also declines, reaching only $0.3/0.48 = 62.5\%$ of the energy efficiency of our original design. This result justifies our choice of using 2n complex mul-add units in the final implementation, despite the apparent underutilization in some computational phases.

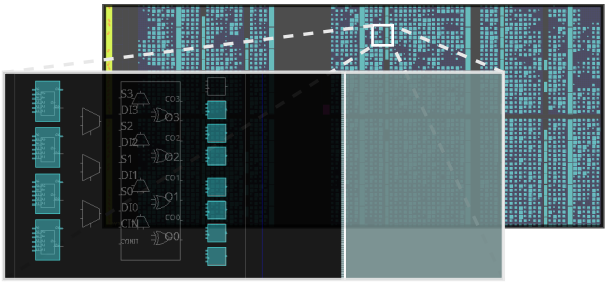

Figure 4: Implementation on `XC7A200T-2FBG484I` with Used FFs (Small Square), LUTs (Medium Rectangle), DSPs (Big Bar), BRAMs (Huge Bar) Highlighted in Color Blue

5 Related Works

QR Decomposition. There are plenty of hardware designs for solving general QR decomposition problem, such as [2, 5, 18, 24, 27, 31, 32, 36, 43]. For specialized matrices, [1] is a FPGA-based QR decomposition design for solving stiff ODE numerical methods, [7, 25] are soft-hardware designs and embedded processing architectures for QR decomposition recursive least square algorithm.

Eigenvalues of Matrices. There are also a large amount of hardware approches to calculate eigenvalues of general matrices, like [13, 15, 20, 39]. For GPU approches, [22] is an implementation of the QR iterations for finding eigenvalues of matrices with CUDA. For specialized problems, [44, 45] implemented eigenvalue decomposition on FPGA for the Direction of Arrival (DOA) estimation.

Givens Rotation. [12] is a square root and division free Givens rotation hardware design for solving least squares problems. [28] evaluates some new CORDIC algorithms implemented on FPGA for the Givens rotator.

Roots of Polynomials. [40, 41] both compared their hardware and software implementations for solving real polynomial roots using Newton's method.

Numerical Calculation Software. Matlab [21], Mathematica [16] and WolframAlpha [38] are most famous for mathematical modeling, symbolic calculation, numerical analysis, function graph plotting and other purposes. SageMath [35] is a large platform integrating multiple math libraries.

6 Conclusion

In this paper, we presented a hardware-accelerated algorithm for visualizing the root density distribution of complex functions by approximating them with polynomials and solving their roots via single-shift QR iteration. By leveraging the Hessenberg form of companion matrices and optimizing QR decomposition with Givens rotations, we designed a fully pipelined, resource-efficient hardware architecture implemented on FPGA.

Our system is capable of processing a large amount of polynomials with high throughput. We demonstrated that our FPGA implementation achieves significant performance and energy efficiency gains over CPU-based approaches, reaching a 65× improvement in energy efficiency. Although our implementation lags behind modern GPUs in performance, the observed gap is primarily due to differences in fabrication technique.

Additionally, we analyzed the trade-offs between resource usage and computational throughput in Matrix Multiplication module designs. Our final implementation, which uses $2n$ complex mul-add units, achieves a favorable balance between energy efficiency and throughput.

The proposed design provides a scalable and open-source solution for complex root visualization, and may serve as a reference for future research in accelerating numerical algorithms via hardware, especially in domains where large-scale root-finding and visualization are crucial.

Acknowledgments

I would like to express my sincere gratitude to the instructors and teaching assistants of the Digital Logic Design course at Tsinghua University for their excellent teaching and the provision of abundant experimental resources and facilities. In particular, I am deeply grateful to Professor Chengbin Quan and Teaching Assistant Jiajie Chen for their patient and meticulous support when I encountered debugging difficulties during the laboratory sessions.

References

[1] Carlos Alberto Oliveira de Souza Junior, João Bispo, João M. P. Cardoso, Pedro C. Diniz, and Eduardo Marques. 2020. Exploration of FPGA-Based Hardware Designs for QR Decomposition for Solving Stiff ODE Numerical Methods Using the HARP Hybrid Architecture. *Electronics* 9, 5 (2020). doi:10.3390/electronics9050843

[2] Abdulrahman Alhamed and Saleh Alshebeili. 2016. FPGA implementation of complex-valued QR decomposition. In *2016 5th International Conference on Electronic Devices, Systems and Applications (ICEDSA)*. 1–4. doi:10.1109/ICEDSA.2016.7818557

[3] AMD Xilinx. 2019. *Vivado Logic Simulation*. https://docs.xilinx.com/r/en-US/ug900-vivado-logic-simulation

[4] AMD Xilinx. 2020. *7 Series FPGAs Data Sheet: Overview*. https://docs.amd.com/v/u/en-US/ds180_7Series_Overview

[5] Semih Aslan, Sufeng Niu, and Jafar Saniie. 2012. FPGA implementation of fast QR decomposition based on givens rotation. In *2012 IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS)*. 470–473. doi:10.1109/MWSCAS.2012.6292059

[6] Zhaojun Bai, James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst. 2000. *Templates for the Solution of Algebraic Eigenvalue Problems*. Society for Industrial and Applied Mathematics. doi:10.1137/1.9780898719581

[7] D. Boppana, K. Dhanoa, and J. Kempa. 2004. FPGA based embedded processing architecture for the QRD-RLS algorithm. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. 330–331. doi:10.1109/FCCM.2004.34

[8] Luis A. Caffarelli. 1990. Interior W2, p Estimates for Solutions of the Monge-Ampere Equation. *Annals of Mathematics* 131, 1 (1990), 135–150. http://www.jstor.org/stable/1971510

[9] L. Dagum and R. Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55. doi:10.1109/99.660313

[10] Jean-Marc Delort. 1991. Existence de Nappes de Tourbillon en Dimension Deux. *Journal of the American Mathematical Society* 4, 3 (1991), 553–586. http://www.jstor.org/stable/2939269

[11] Gene H Golub and Charles F Van Loan. 2013. *Matrix computations*. JHU press.

[12] J. Götze and U. Schwiegelshohn. 1991. A Square Root and Division Free Givens Rotation for Solving Least Squares Problems on Systolic Arrays. *SIAM J. Sci. Statist. Comput.* 12, 4 (1991), 800–807. doi:10.1137/0912042

[13] Krishna Deep Gupta, Mohd Wajid, Rehan Muzammil, and Syed Javed Arif. 2019. Hardware Architecture for Eigenvalues Computation using the Modified Jacobi Algorithm on FPGA. In *2019 5th International Conference on Signal Processing, Computing and Control (ISPCC)*. 243–246. doi:10.1109/ISPCC48220.2019.8988376

[14] S. W. Hawking and G. F. R. Ellis. 1973. *The Large Scale Structure of Space-Time*. Cambridge University Press.

[15] Ahmed Imtiaz, Muhammad Sajid, Muhammad Ahmed, and Muhammad Sagheer. 2011. FPGA-based Householder implementation for efficient eigenvalues calculation. *International journal of innovative computing, information & control: IJICIC* 7 (10 2011).

[16] Wolfram Research, Inc. 2024. Mathematica, Version 14.2. https://www.wolfram.com/mathematica

[17] Yu. N. Kosovtsov. 2022. Existence and smoothness of the Navier-Stokes equations and semigroups of linear operators. arXiv:2209.06587

[18] Martin Langhammer and Bogdan Pasca. 2018. High-Performance QR Decomposition for FPGAs. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, CALIFORNIA, USA) *(FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 183–188. doi:10.1145/3174243.3174273

[19] Hongxia Lin, Qing Sun, Sen Liu, and Heng Zhang. 2023. The Stability and Decay for the 2D Incompressible Euler-Like Equations. *Journal of Mathematical Fluid Mechanics* 25 (10 2023). doi:10.1007/s00021-023-00824-5

[20] Yang Liu, C.-S. Bouganis, P.Y.K. Cheung, P.H.W. Leong, and S.J. Motley. 2006. Hardware Efficient Architectures for Eigenvalue Computation. In *Proceedings of the Design Automation & Test in Europe Conference*, Vol. 1. 1–6. doi:10.1109/DATE.2006.243838

[21] MathWorks. 2025. *MATLAB*. https://www.mathworks.com/products/matlab.html

[22] Meirui, Meihui, Weiping Zhang, Tao Wang, Ning Tian, Jinbao Li, and Longjiang Guo. 2013. An implementation of the QR iterations for finding eigenvalues of matrices with CUDA on GPU. In *Proceedings 2013 International Conference on Mechatronic Sciences, Electric Engineering and Computer (MEC)*. 1572–1577. doi:10.1109/MEC.2013.6885312

[23] Luciano Modica. 1987. The gradient theory of phase transitions and the minimal interface criterion. *Archive for Rational Mechanics and Analysis* 98 (1987), 123–142.

[24] Sergio D. Muñoz and Javier Hormigo. 2015. High-Throughput FPGA Implementation of QR Decomposition. *IEEE Transactions on Circuits and Systems II: Express Briefs* 62, 9 (2015), 861–865. doi:10.1109/TCSII.2015.2435753

[25] Sufeng Niu, Sizhou Wang, Semih Aslan, and Jafar Saniie. 2013. Hardware and software design for QR Decomposition Recursive Least Square algorithm. In *2013 IEEE 56th International Midwest Symposium on Circuits and Systems (MWSCAS)*. 117–120. doi:10.1109/MWSCAS.2013.6674599

[26] NVIDIA. 2020. *NVIDIA AMPERE GA102 GPU ARCHITECTURE*. https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.1.pdf

[27] Michael Parker, Volker Mauer, and Dan Pritsker. 2016. QR decomposition using FPGAs. In *2016 IEEE National Aerospace and Electronics Conference (NAECON) and Ohio Innovation Summit (OIS)*. 416–421. doi:10.1109/NAECON.2016.7856841

[28] Pawel Poczekajlo, Leonid Moroz, Ewa Deelman, and Pawel Gepner. 2025. Evaluation of new CORDIC algorithms implemented on FPGA for the Givens Rotator. *Journal of Computational Science* 87 (2025), 102567. doi:10.1016/j.jocs.2025.102567

[29] Yousef Saad. 2011. *Numerical Methods for Large Eigenvalue Problems*. Society for Industrial and Applied Mathematics. doi:10.1137/1.9781611970739

[30] Richard Schoen. 1984. Conformal deformation of a Riemannian metric to constant scalar curvature. *Journal of Differential Geometry* 20, 2 (1984), 479 – 495. doi:10.4310/jdg/1214439291

[31] Anatoli Sergyienko and Oleg Maslennikov. 2001. Implementation of Givens QR-Decomposition in FPGA. In *Proceedings of the Th International Conference on Parallel Processing and Applied Mathematics-Revised Papers (PPAM '01)*. Springer-Verlag, Berlin, Heidelberg, 458–465.

[32] A. V. Sokolovskiy, V. N. Tyapkin, E. A. Veisov, and Yu.L. Fateev. 2019. The Pipelined QR Decomposition Hardware Architecture Based On Givens Rotation CORDIC Algorithm. In *2019 International Siberian Conference on Control and Communications (SIBCON)*. 1–4. doi:10.1109/SIBCON.2019.8729615

[33] Frank H. Stillinger and Thomas A. Weber. 1984. Packing Structures and Transitions in Liquids and Solids. *Science* 225, 4666 (1984), 983–989. doi:10.1126/science.225.4666.983

[34] Ruibai Tang and Chengbin Quan. 2025. *Hardware-Accelerated Algorithm for Complex Function Roots Density Graph Plotting*. repo: https://github.com/trrbivial/HPC-Auxiliary-Plotter.

[35] The Sage Developers. 2025. *SageMath, the Sage Mathematics Software System*. https://www.sagemath.org.

[36] Xiaojun Wang and Miriam Leeser. 2009. A truly two-dimensional systolic array FPGA implementation of QR decomposition. *ACM Trans. Embed. Comput. Syst.* 9, 1, Article 3 (Oct. 2009), 17 pages. doi:10.1145/1596532.1596535

[37] David S. Watkins. 1982. Understanding the QR Algorithm. *SIAM Rev.* 24, 4 (1982), 427–440. http://www.jstor.org/stable/2029539

[38] Wolfram Research, Inc. 2025. WolframAlpha Online. https://www.wolframalpha.com/

[39] Di Yan, Wei-Xing Wang, and Xiao-Wei Zhang. 2022. High-Performance Matrix Eigenvalue Decomposition Using the Parallel Jacobi Algorithm on FPGA. *Circuits Syst. Signal Process.* 42, 3 (Sept. 2022), 1573–1592. doi:10.1007/s00034-022-02180-7

[40] Kuo-Pao Yang and Theresa Beaubouef. 2010. Equivalence of Hardware and Software: A case study for solving polynomial functions. In *2010 42nd Southeastern Symposium on System Theory (SSST)*. 318–322. doi:10.1109/SSST.2010.5442817

[41] Kuo-pao Yang and Theresa Beaubouef. 2010. Hardware vs. software implementations for calculating roots of polynomials. *J. Comput. Sci. Coll.* 25, 4 (April 2010), 15–21.

[42] Shing-Tung Yau. 1978. On the ricci curvature of a compact kähler manifold and the complex monge-ampére equation, I. *Communications on Pure and Applied Mathematics* 31, 3 (1978), 339–411. doi:10.1002/cpa.3160310304

[43] Jianfeng Zhang, Paul Chow, and Hengzhu Liu. 2015. CORDIC-Based Enhanced Systolic Array Architecture for QR Decomposition. *ACM Trans. Reconfigurable Technol. Syst.* 9, 2, Article 9 (Dec. 2015), 22 pages. doi:10.1145/2827700

[44] Xiao-Wei Zhang, Di Yan, Lei Zuo, Ming Li, and Jian-Xin Guo. 2023. High-Performance of Eigenvalue Decomposition on FPGA for the DOA Estimation. *IEEE Transactions on Vehicular Technology* 72, 5 (2023), 5782–5797. doi:10.1109/TVT.2022.3221915

[45] Shuang Zhou and Li Zhou. 2024. Field Programmable Gate Array (FPGA) Implementation of Parallel Jacobi for Eigen-Decomposition in Direction of Arrival (DOA) Estimation Algorithm. *Remote Sensing* 16, 20 (2024). doi:10.3390/rs16203892