SSRG International Journal of Electronics and Communication Engineering
ISSN: 2348-8549/ https://doi.org/10.14445/23488549/IJECE-V13I8P102

Volume 13 Issue 8, 26-39, August 2026


*Original Article*

# GPU Implementation of Parallel Computing Algorithms for Parabolic, Wright, and Riemann Zeta Functions

Chaitanya S. Jage[1], Vishwesh A. Vyawahare[2], Mukesh D. Patil[3]

[1,2,3]*Department of Electronics Engineering, Ramrao Adik Institute of Technology, DY Patil Deemed to be University, Nerul, Navi Mumbai, India.*

[1]*Corresponding Author : chaitanya.jage@rait.ac.in*

Received: 25 March 2026 Revised: 19 May 2026 Accepted: 15 July 2026 Published: 31 August 2026

***Abstract*** *- Fast and accurate computation of special functions such as Parabolic function, Wright function and Riemann Zeta function is essential in problems involving fractional calculus, quantum models and large-scale numerical simulations. These functions require extensive series expansions and recurrence-based formulations, which lead to significant computational overhead when computed sequentially. To solve this problem, this work reports the design of optimised parallel computing algorithms for the Parabolic function, Wright function and Riemann Zeta function, and implementation on a Graphics Processing Unit (GPU) platform using Compute Unified Device Architecture (CUDA). The decomposition of these functions substantially speeds up their computation on concurrent CUDA threads, enabling efficient large-scale computation. Techniques to reduce shared memory and strategies to preserve precision are integrated, guaranteeing numerical stability and accuracy in the calculation. The GPU implementation achieves 100x to 160x speed-up over the sequential C implementation, thereby proving the effectiveness of the proposed method for fast and reliable computations of these functions.*

***Keywords*** *- Parabolic function, Wright function, Riemann Zeta function, Numerical Computation, Parallel Algorithms, CUDA, GPU Computing.*

## 1. Introduction

Special functions are the analytical basis for many mathematical models and numerical simulations in computational engineering, physics and applied mathematics. Parabolic functions have naturally arisen in the study of quantum harmonic oscillators, electromagnetic wave propagation, and variable coefficient second order differential equations [1, 2]. The Wright function, in its original definition by means of generalized Bessel-type expansions, has later become one of the essential elements for the modeling of fractional diffusion processes, viscoelastic relaxation and anomalous transport phenomena [3-5]. In a broader theoretical context, the Riemann zeta function is still at the forefront of number theory, with applications in spectral analysis, quantum gas thermodynamics and statistical description of energy-level distributions [6, 7].

These applications include evaluation of special functions, on a dense grid over large ranges of parameters, or in the course of iterative steps in Monte Carlo simulations, stability analyses, computations of option prices, and solutions of fractional differential equations [8-10]. Additionally, these computations are frequently computationally demanding. Asymptotic expansions, recurrence relations, and slowly convergent series are examples of series-based representations that are computationally stable and numerically challenging. If done in a loop, they are also rather expensive. Particularly for models described by fractional order dynamics involving Wright and Mittag-Leffler functions, increasing simulation dimensions will make CPU-based computation a critical bottleneck [11].

Parallel hardware has been shown to be an effective means of addressing this requirement. Graphics Processing Unit (GPU) provide enormous parallelism across threads, high memory bandwidth, and efficient execution of independent arithmetic operations. These features make them attractive for evaluating special functions, which can be expressed as combinations of term-wise or element-wise operations [12]. Over the last decade, tremendous progress has been achieved in employing GPUs to expedite the computation of numerical kernels such as Bessel functions [13], hypergeometric functions [14], Mittag-Leffler functions [15], and generic mathematical transforms in scientific computation [16, 17]. Even though there are a few numerical kernels that have been implemented on GPUs, there is relatively little information in the existing literature about algorithms optimized for GPUs for the Parabolic function, the Wright function, or the Riemann Zeta function in fractional calculus problems. The research gaps identified and the motivation discussed in this

work are as follows:

1. Existing studies do not provide a unified GPU-oriented framework for the efficient computation of Parabolic Cylinder, Wright, and Riemann Zeta functions.
2. Despite the increasing use of GPUs in scientific computing, CUDA-based algorithms for the efficient evaluation of special functions such as the Parabolic, Wright, and Riemann Zeta functions are not available in existing literature.
3. Comparison of sequential CPU implementations with parallel GPU algorithms in CUDA is limited in the literature.

Special functions play an important role in fractional calculus as they are often used to provide analytical and numerical solutions to fractional differential equations in physics, engineering, control systems and signal processing. In Fractional Order (FO) models, functions such as the Gamma function, the Mittag-Leffler function, the Confluent Hypergeometric function and other transcendental functions are commonly used to describe the memory and heredity properties of dynamical systems. However, these functions are computationally expensive to compute due to the need for infinite series expansions, recurrence relations, integral transforms and complex arithmetic operations. Researchers have increasingly investigated GPU-based parallel computing techniques to overcome those limitations and speed up the evaluation of special functions used in fractional calculus. Independent arithmetic operations and summation terms can be parallelized, which has resulted in significant improvements in execution speed and scalability of CUDA-oriented implementations [18]. Furthermore, hybrid GPU CPU computational frameworks and parallel discretization strategies have been presented to efficiently solve FO systems while reducing computational overhead and memory complexity [19]. Also, recent works showed that fast convolution techniques and hybrid numerical approaches can be employed to further improve the efficiency of large-scale Fractional Differential Equation (FDE) simulations [20, 21]. These developments underscore the growing importance of GPU-accelerated computing for efficient numerical analysis in fractional calculus applications. However, the GPU computing of Parabolic Cylinder, Wright, and Riemann Zeta functions has not been addressed in the literature. These functions are employed in quantum mechanics, fractional differential equations, anomalous diffusion models, signal processing, spectral analysis, wave propagation, and large-scale scientific computing applications.

This work addresses the research gap by designing parallel computing algorithms using the Compute Unified Device Architecture (CUDA) framework that enables efficient computation of these functions. The new approach transforms the traditional numerical forms of these special functions into a structure well suited for parallel computation, with thousands of computing threads on the GPU working concurrently to compute individual terms. We have included reductions, log transforms, and truncation strategies with precision taken into account, and a C code version for a more realistic speed comparison. The novel contributions of this work are presented as follows:

1. A systematic framework of parallel computing methods for efficient numerical evaluation of specific functions, designed for high-performance implementation.
2. The GPGPU platform is used to implement a GPU-based framework for parallel algorithms to speed-ups the computation of the selected special mathematical functions.
3. The proposed parallel method implementation's performance is evaluated against the best sequential version to assess improvements in accuracy, runtime, and scalability.

The paper is structured as follows. Section 2 deals with the Fractional Calculus and its relation to the considered functions. Section 3 introduces the Special Functions that are considered in this work and the mathematical details needed to compute them. Parallel computing and GPU acceleration are discussed in Section 4. The principles of the CUDA-based execution are explained. Section 5 discusses parallel computing algorithms for computing the selected special functions. Section 6 gives the results and performance comparisons of the CPU and the GPU. The work is concluded in Section 7, with the main findings and future directions.

## 2. Fractional Calculus

Fractional calculus is a generalization of classical differentiation and integration to arbitrary order, i.e. real or complex. This mathematical technique has gained considerable interest in recent years due to its ability to deal with more complex phenomena in nature and physical systems. There are several publications on the properties of FDE and their solutions [8]. Currently, fractional derivatives FD and integrals are used in physics, engineering, biology, and control systems for their ability to capture memory effects and nonlocality. Several mathematical approaches have been proposed to define derivatives of non-integer order. The three most popular definitions of FD are as follows,

### *2.1. Grünwald–Letnikov Derivative*

This approach defines the fractional derivative through a limiting process involving backward finite differences:

$$ {}_aD_x^{\mu} f(\mathrm{x}) = \lim_{h\to 0} h^{-\mu} \sum_{k=0}^{\left[\frac{x-\mu}{h}\right]} (-1)^k \, {}^{\mu}C_k, f(x-kh), \qquad (1) $$

where, $[x]$ is the integer part of x, and the binomial coefficient is given ${}^{\mu}C_k$ .

### *2.2. Riemann–Liouville (RL) Derivative*

The RL definition involves FO integration of the function

followed by its classical derivative. It is given as:

$$ {}_aD_x^{\mu} f(x) = \frac{1}{\Gamma(m-\mu)} \frac{d^m}{dt^m} \int_{\mu}^{x} \frac{f(\mathrm{p})}{(y-\mathrm{p})^{\mu-m+1}}, d\mathrm{p}, \quad (2) $$

for $(m-1) < \mu < m, m \in Z^+$ and $\Gamma(\cdot)$ is the Gamma function.

### *2.3. Caputo's definition of order $\alpha \in R^+$*

The Caputo derivative is a basic tool of fractional calculus. And it is expressed as,

$$ {}_aD_x^{\mu} f(x) = \frac{1}{\Gamma(m-\mu)} \int_{\mu}^{x} \frac{f^{(m)}(\mathrm{p})}{(y-\mathrm{p})^{\mu-m+1}}, d\mathrm{p}, (3) $$

for $(m-1) < \mu < m, m \in Z^+$where $f^{(m)}(\mathrm{p})$ is the $m^{th}$ -order derivative of the function $f(x)$.

The Caputo derivative is especially evident in engineering applications and physics because it enables the direct implementation of initial conditions, which are meaningful in physics, into an FO model.

Differentiation and integration are extended to functions with non-integer orders in fractional calculus, offering an effective method for studying processes with memory and genetic characteristics. Fractional derivative processes are highly effective in simulating long-range dependencies in physical and biological processes because they rely on both local and historical data of a function, as opposed to classical derivative processes that only rely on local data. FDEs have become widely used in physics and biology as a result [22]. There are several fractional derivative formulations suited to specific problems. While the Riemann-Liouville fractional derivative formulations are preferred for analytical and theoretical modeling, others are preferred in engineering for initial-value problems where it is advantageous to have initial conditions given as equations with integer orders [23]. Another fractional derivative formulation that provides a discretized representation of fractional derivative processes is given by Grünwald and Letnikov and is preferred for numerical computations, especially for finite difference approximations of fractional differential equations.

Fractional calculus typically produces mathematical formulations with solutions described in terms of nonclassical special functions. Wright functions are often used to solve fractional diffusion and relaxation phenomena, whereas general space fractional derivatives may result in a solution in the form of generalized Gaussian-type kernels and Parabolic Functions. These special functions encode the nonlocal properties of the corresponding fractional processes and have been applied to the study of viscoelasticity, anomalous diffusion, bioengineering, and ultra-slow relaxation phenomena [24, 25]. Computationally, the fractional differential equation is challenging due to the history-dependent kernels and the repeated evaluation of the aforementioned special function at many points in time and space. Generally, standard numerical approaches for simulating fractional differential equations can be highly computationally intensive, especially for complex simulations and finer grids [26]. With the rising complexity of simulating fractional models, highly efficient numerical approaches, preferably with parallel computation, are needed.

## 3. Special Functions in Fractional Calculus

Special functions have significance in fractional calculus because they are developed while solving fractional differential and integral equations. The Parabolic function, the Wright function, and the Riemann Zeta function are among the special functions commonly utilized to tackle anomalous diffusion problems and nonlocal relaxation processes [27]. These functions are generally presented in forms of infinite series, asymptotic series, or recurrence relations. The disadvantage of these forms is the potential instability and convergence problems when solving these functions numerically. The definitions of these functions are presented in the following sections.

### *3.1. Parabolic Functions*

Parabolic functions appear as solutions of the second-order differential equation.

$$ \frac{d^2y}{dt^2} + \left( \mathrm{A} + \frac{1}{2} - \frac{t^2}{4} \right) = 0 \quad (4) $$

Where A is a real or complex parameter [1], they arise in models of quantum harmonic oscillators, the propagation of electromagnetic waves, and the stability of differential systems [2]. Also, the parabolic functions arise in some problems of fractional diffusion, in which Gaussian-type distributions are affected by fractional dynamics [28, 29]. The functions can be written in terms of two independent solutions, generally denoted by U(A, t) and V(A, t). The series representations of these functions permit separation into even and odd parts, which is helpful in numerical calculations.

### *3.2. Even Parabolic Function*

The even solution contains only even powers of t:

$$ D_{\mathrm{A}}^{(e)}(t) = e^{-t^2/4} \sum_{k=0}^{\infty} \frac{\left(\frac{1}{2}\mathrm{A}\right)_k}{(2k)!} \left(\frac{t^2}{2}\right)^k \quad (5) $$

Where $(A)_k$ is the Pochhammer symbol [30]. Its representation is stable for small and moderate values of t.

### *3.3. Odd Parabolic Function*

The odd solution contains only odd powers of t:

$$D_{\mathrm{A}}^{(o)}(t) = te^{-t^2/4} \sum_{k=0}^{\infty} \frac{\left(\frac{1}{2}(\mathrm{A}+1)\right)_k}{(2k+1)!} \left(\frac{t^2}{2}\right)^k \qquad (6)$$

This allows the construction of general solutions through linear combinations [31]. Such functions appear in fractional stochastic processes and models involving fractional Hermite operators [32].

### *3.4. Wright Function*

The Wright function can be defined as follows:

$$W_{\alpha,\beta}(t) = \sum_{k=0}^{\infty} \frac{t^k}{k!\ \Gamma(\alpha k + \beta)} \qquad (7)$$

It is a fundamental building component of fractional calculus [3]. It is a key component of solutions to time fractional diffusion equations, representing probability densities for subdiffusive processes [33].

$$u(x,t) = \frac{1}{2t^{\alpha/2}} W_{-\alpha/2,\,1-\alpha/2}\left(-\frac{|x|}{t^{\alpha/2}}\right) \qquad (8)$$

Here, the probability density is expressed as $u(x,t), x \in R$is the spatial variable, while $t > 0$ denotes time. The parameter α controls the order of the fractional derivative, indicating the sub-diffusive nature of the process [34, 35]. The Wright function is the primary solution to the time-fractional diffusion equation, and it defines the heavy-tailed distribution found in sub-diffusive fractional transport processes [35, 36].

### *3.5. Riemann Zeta Function*

The Riemann Zeta function is conventionally defined for Re(s) > 1 as:

$$\zeta(s) = \sum_{k=1}^{\infty} k^{-s} \qquad (9)$$

The parameter $s$ denotes the argument of the zeta function and may take real or complex values, with the convergence condition $R(s) > 1$. In the present study, $s$ is a positive real number, which is sufficiently general for most numerical applications. This is then summed over the index , which is a summation variable and runs from $k = 1\ to\ \infty$, counting the successive terms of the series. The infinite sum is approximated computationally by a truncated sum with a finite upper limit $N$, for which both efficient sequential and parallel implementations are possible. And it also extends analytically in other parts of the complex plane [6]. Apart from number theory, the Zeta function arises in spectral sums, fractional quantum mechanics, and eigenvalue regularization of fractional Laplacians [37]. Efficient schemes become necessary when a significant summation limit or high accuracy is required. Thus, there is a motivation to employ parallel computing approaches [38].

## 4. Parallel Computing and GPU Acceleration

Numerical integration to compute special functions like the Parabolic, Wright, or Riemann Zeta functions requires numerous calculations involving expansions, gamma, or power functions, which are computationally intensive, especially as the grid point or cut-off parameter increases. Conventional processors rely mainly on pipelined execution and lack robust parallel processing capabilities, making them unsuitable for the above kinds of calculations. GPUs are designed to provide massive parallel processing, substantial memory bandwidth, and a hierarchical architecture well suited to applications requiring numerous simultaneous arithmetic operations, as in the processing of special functions used in fractional calculus, anomalous diffusion, spectral methods, and so on.

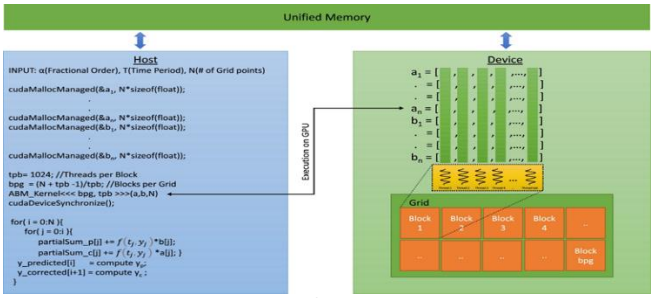


**Fig. 1 A graphic representation of the proposed algorithm employed in this study**

Modern GPUs are now made up of Streaming Multiprocessors (SMs), which include hundreds of lightweight CUDA cores, warp schedulers, registers, shared memory banks, and specialized functional units. Threads are processed in batches of 32, known as warps, in a Single-Instruction, Multiple Thread parallel processing architecture. This makes processing thousands of threads in a single operation extremely effective for performing term-by-term series calculations and algorithms that involve the summation of values, which is optimally suited to GPUs.A graphic representation of the proposed algorithm is illustrated in Figure 1 [12]. Another significantly important factor is the present hierarchical architecture in GPUs, which has global memory meant for extensive volume data, constant/texture memory designed for cached accesses, registers for storage meant for each processing thread, and shared memory that has persistently low latency, enabling threads in a single block to distribute intermediate values in a shared manner, hence significantly speeding up computation in cases where summations or reductions in special functions are involved [39].

CUDA is a parallel computing platform created and used by NVIDIA that allows programmers to translate mathematical operations into kernels to execute on this platform. Each execution of a kernel launches a two-dimensional array of thread blocks and allows the programmer to assign operations to thread blocks to compute a part of the

mathematical operation. For special functions, a term can be directly transferred to a thread due to the independence of the series terms in the Parabolic series, the Wright function's series expansion, and the Riemann Zeta-function's summation form. The processing phase of the program contains a reduction step, which uses the shared memory model across the blocks of threads to compute the total and store it back on the CPU [40].

This model follows the already used algorithms for the parallelization of mathematical functions involving fractional calculus on a GPU and applies this approach to the developed framework for the mathematical functions investigated in this paper. The applicability of GPUs to special function computation is further evident in the context of fractional functions, which are often used in numerical solutions to fractional diffusion or oscillation problems.

For instance, in fractional diffusion problems, numerical solvers often require computing thousands of Wright functions at every time step, whereas in fractional oscillations, Parabolic function computation is often involved. In such cases, besides speeding up computation, GPUs can also facilitate computation in problems where a CPU setup would incur impractical computational costs, a direct result of the parallel architecture being independent of problem scale [41].

## 5. Parallel Computing Framework for Special Functions

This section presents the design of GPU-accelerated algorithms for computing the Even and Odd Parabolic functions, the Wright function, and the Riemann Zeta function. The method used here is similar to the parallelization framework described above, where each series expansion is decomposed into independent terms, which are then mapped to CUDA threads [12]. The original sequential formulation of each function is first reformulated to a suitable form for execution on the GPU. Each term is then computed independently and accumulated to get the final numerical result.

The CUDA algorithms are developed for Even Parabolic function, and Odd Parabolic function are presented in Algorithms 1 and 2, respectively. Algorithms 3 and 4 are designed for the Wright and Riemann Zeta functions, respectively.

The validation with their CPU counterparts, as depicted in Figures 4-7, to ensure the computational consistency of the results. The Mean Squared Error (MSE) between the two outputs can be calculated as [42]:

$$\text{MSE} = \frac{1}{j}\sum_{q=1}^{j}\left(y_q - \widehat{y_q}\right)^2$$

Where $q$ signifies the total number of data samples, $y_q$ represents the serial execution on the CPU at the $y^{\text{th}}$ sample, $\widehat{y_q}$ represents the parallel execution on the GPU at the $q^{\text{th}}$ Sample. The GPU implementation maintains the same level of precision as the verified CPU model, and the computed values are below a predetermined threshold, as measured by the MSE. This validation shows that the parallel algorithm is robust to its parameter while achieving a notable speed-up [42].

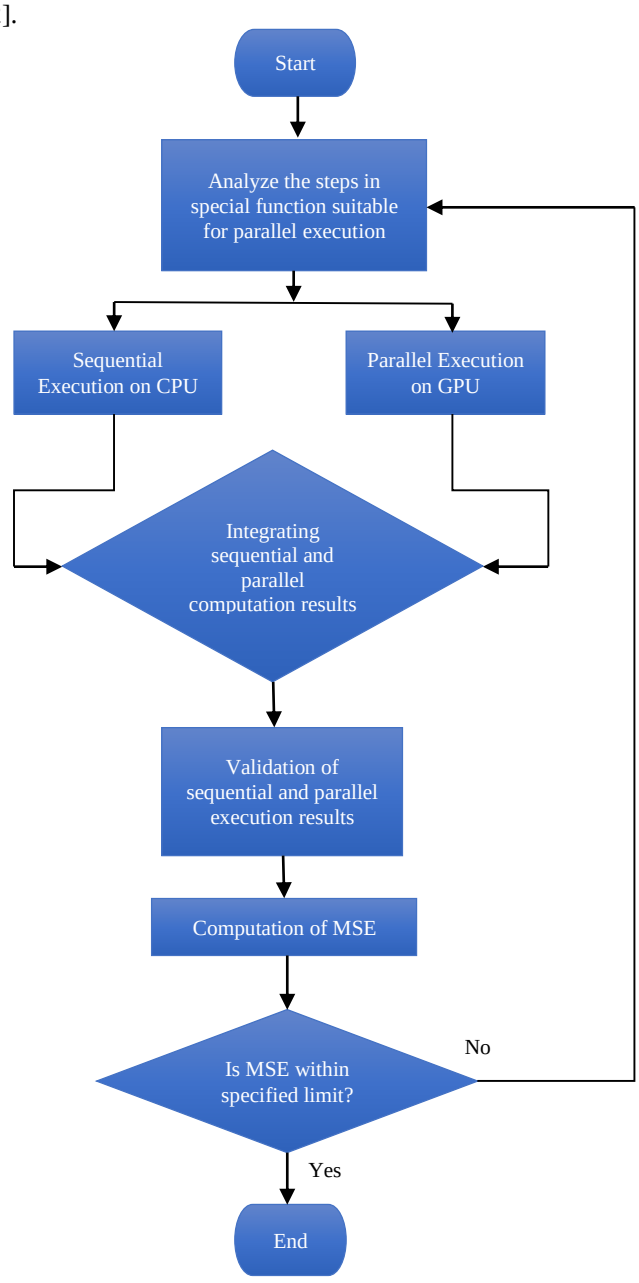


**Fig. 2 Flow chart for the proposed approach employed in this study**

The performance gain achieved through GPU acceleration is quantified using the speed-up, and it is defined as:

$$\text{Speed-up} = \frac{\text{Execution time of serial CPU implementation}}{\text{Execution time of parallel GPU implementation.}}$$

A substantial speed-up ensures proper utilization of GPU resources. The entire process of the proposed methodology is presented in Figure 2. This flowchart presents how one can move from function component analyses for parallelization to testing and validation of parallel computing for both CPU and GPU devices. This ensures efficient calculation of special functions using parallel computing methods.

### *5.1. Even Parabolic Function*

The Even Parabolic Function is defined by:

$$D_{\mathrm{A}}^{(e)}(t) = \sum_{k=0}^{N} \frac{\left(\frac{\mathrm{A}}{2}\right)_k}{(2k)!}\left(\frac{t^2}{2}\right)^k e^{-t^2/4} \qquad (10)$$

Each summation term is independent and thus suitable for GPU-based parallel evaluation.

$$E_k = \frac{\left(\frac{\mathrm{A}}{2}\right)_k}{(2k)!}\left(\frac{t^2}{2}\right)^k \qquad (11)$$

The prefactor $e^{-t^2/4}$, is computed once per input value, while the terms are computed in parallel on the GPU and then reduced to obtain the final sum.

**Algorithm 1: Even Parabolic Function $D_{\mathrm{A}}^{(e)}(t)$ on GPU**

1. Set $(t, A)$ and truncation limit $N$
2. Compute $prefactor \leftarrow \exp(-t^2/4)$
3. Allocate memory for $d\_even\_host[\,]and\ d\_even\_device[\,]$
4. Start timer
5. Launch CUDA kernel $compute\_D\_even$
6. $tid \leftarrow threadIdx.x$
7. $id \leftarrow blockIdx.x \times blockDim.x + tid$
8. If $id \geq num\_values$ then Return
9. return
10. $sum \leftarrow 0.0$
11. For $k = 0\ to\ N$ in parallel do
12. Compute Pochhammer: $poc \leftarrow pochhammer(A/2, k)$
13. Compute:
$$E_k = \frac{poc \cdot (xt^2/2)^k}{(2k)!}$$
14. $sum \leftarrow sum + E_k$
15. end for
16. Output: $D_A^{(e)}(t) = prefactor \times sum$
17. Copy device results to the host
18. Stop the timer and free memory.

### *5.2. Odd Parabolic Function*

The Odd Parabolic Function is given by:

$$D_{\mathrm{A}}^{(o)}(t) = t\, e^{-t^2/4} \sum_{k=0}^{N} \frac{\left(\frac{\mathrm{A}+1}{2}\right)_k}{(2k+1)!}\left(\frac{t^2}{2}\right)^k \qquad (12)$$

Each term in the series is computed independently within a GPU thread:

$$O_k = \frac{\left(\frac{\mathrm{A}+1}{2}\right)_k}{(2k+1)!}\left(\frac{t^2}{2}\right)^k \qquad (13)$$

**Algorithm 2: Odd Parabolic Function $D_A^{(o)}(t)$ on GPU**

1. Set $(t, \mathrm{A})$ and truncation limit $N$
2. Compute $prefactor \leftarrow t \cdot \exp(-t^2/4)$
3. Allocate memory for $d\_odd\_host[\,]\ and\ d\_odd\_device[\,]$
4. Start timer
5. Launch CUDA kernel $compute\_D\_odd$
6. $tid \leftarrow threadIdx.x$
7. $id \leftarrow blockIdx.x \times blockDim.x + tid$
8. $id \geq num\_values$
9. return
10. $sum \leftarrow 0.0$
11. for $k = 0\ to\ N$ in parallel do
12. Compute Pochhammer: $poc \leftarrow pochhammer\big((\mathrm{A}+1)/2, k\big)$
13. Compute:
$$O_k = \frac{poc \cdot (t^2/2)^k}{(2k+1)!}$$
14. $sum \leftarrow sum + t_k$
15. end for
16. Output: $D_{\mathrm{A}}^{(o)}(t) = prefactor \times sum$
17. Copy device results to the host
18. Stop the timer and free memory.

### *5.3. Wright Function*

The Wright function is defined by:

$$W_{\alpha,\beta}(t) = \sum_{k=0}^{N} \frac{t^k}{k!\ \Gamma(\alpha k + \beta)} \qquad (14)$$

Each summation term is independent and computed via:

$$W_k = \frac{t^k}{k!\ \Gamma(\alpha k + \beta)} \qquad (15)$$

Computations are performed in logarithmic form to avoid overflow,

$$L_k = k\log(t) - \log(k!) - \log\Gamma(\alpha k + \beta) \qquad (16)$$

**Algorithm 3: Wright Function $W_{\alpha,\beta}(t)$ on GPU**

1. Set parameters $(\alpha, \beta, t)$and truncation limit $N$
2. Allocate memory for $wright\_host[\,]\ and\ wright\_device[\,]$
3. Start timer
4. Launch CUDA kernel $compute\_wright$
5. $id \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$
6. $id \geq num\_values$
7. return
8. $sum \leftarrow 0.0$
9. for $k = 0\ to\ N$ in parallel do
10. Compute: $L_k = k\log(t) - \log(k!) - \log\Gamma(\alpha k + \beta)$
11. Compute $t_k = \exp(L_k)$
12. $sum \leftarrow sum + t_k$
13. end for
14. Output: $W_{\alpha,\beta}(t) = sum$
15. Copy results to host
16. Stop the timer and free memory.

### *5.4. Riemann Zeta Function*

For $R(s) > 1$, the Riemann Zeta function is given as:

$$\zeta(s) = \sum_{k=1}^{N} k^{-s} \tag{17}$$

Each term is independently evaluated by:

$$t_k = k^{-s} \tag{18}$$

**Algorithm 4: Riemann Zeta Function $\zeta(s)$ on GPU**

1. Set input $s$ and truncation limit $N$
2. Allocate memory for $zeta\_host[\,]\ and\ zeta\_device[\,]$
3. Start timer
4. Launch CUDA kernel $compute\_zeta$
5. $id \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$
6. $id \geq num\_values$
7. return
8. $sum \leftarrow 0.0$
9. for $n = 1\ to\ N$ in parallel do
10. Compute: $t_k = k^{-s}$
11. $sum \leftarrow sum + t_k$
12. end for
13. Output: $\zeta(s) = sum.$
14. Copy results to host
15. Stop the timer and free memory.

## 6. Results and Discussion

For the GPU version, the mathematical expressions of the three functions were reformulated into summation components suitable for thread-level decomposition. Each CUDA kernel was designed so that every thread computed one term of the series, while shared memory handled partial summations within a block. This reduced dependence on slower global memory and enabled efficient accumulation of results. The approach mirrors the procedure described in earlier parallel computing frameworks, where special function evaluation is mapped onto the many-core GPU architecture to reduce computational latency.

**Table 1. Hardware specifications of the computing environment**

| Parameter | Details |
|---|---|
| GPU | RTX A5000 NVIDIA |
| CUDA Cores | 8192 |
| Base Clock | 1170 MHz |
| GPU Memory | 24 GB GDDR6 |
| CPU | Intel® Xeon® |
| Cache | 22.5 MB |
| Cores | 16 |
| Base Clock | 2.6 GHz |
| RAM | 12 GB |

**Fig. 3 Computational platform with NVIDIA GPU**

To maintain high throughput, grid and block topologies were chosen depending on the number of evaluation points and the truncation limitations of each function. Host-device memory transfers were minimized by keeping term-wise computation entirely on the GPU. Exponential, power, and Gamma-related expressions utilized CUDA intrinsic functions to provide faster execution compared to CPU arithmetic functions. Once the kernel processing finished, values were retrieved from the device for comparison with CPU values. For every input, results from both processors were identical within floating precision, ensuring that parallel expressions were accurate. Performance studies have shown persistent speed-ups for all three functions.

The hardware configuration used for the experimental evaluation is dictated by the technical specifications of the computing platform presented in Table 1 and the computational environment shown in Figure 3. The environment is a heterogeneous CPU-GPU environment where the GPU is used mainly for computationally intensive

numerical calculations, and the CPU is responsible for task scheduling, control flow and data coordination. This arrangement permits efficient use of system resources and scalability to larger problem sizes. The results achieved for the parallel computing algorithm by using a GPU always indicate better computational performance. This confirms the effectiveness of GPU-accelerated parallel processing for complex mathematical computations.

The Parabolic functions were easily amenable to parallelization of polynomial expressions for the Gamma function, whereas for the Wright function, there was proportionally faster acceleration owing to its exponential as well as Gamma function dominants. The Riemann Zeta function also recorded notable performance speed-ups, primarily for truncated limits of substantial size, as they necessitate deeper summations. In essence, all studies have validated the efficiency of using identical parallelization techniques for other special functions as well.

### *6.1. Even Parabolic Function*

Computational performance of the Even Parabolic function for varying values of the parameter $A$ is highlighted in Table 2, with the truncation order fixed at $N = 200$ and a fine time discretization $t = -5{:}10^{-5}{:}5$. The sequential execution time remains nearly constant across all cases, indicating minimal sensitivity to changes in $A$. The parallel processing implementation completes the process up to 130 to 150 times faster. As a result, speed-up of over two orders of magnitude is achieved, validating the efficiency of the novel parallel method. Figure 4 validates the parallel implementation by showing an almost exact match between the parallel and serial results across the entire time domain. The strong agreement, including near the extrema, confirms that substantial performance gains are achieved without loss of numerical accuracy. This is quantitatively verified by the low Mean Squared Error (MSE) values reported in Table 3.

### *6.2. Odd Parabolic Function*

The execution time and speed-up obtained for the Odd Parabolic function for different readings of the parameter $A$ reported in Table 4, with the truncation order fixed at $N = 200$ and a fine time discretization $t = -5{:}10^{-5}{:}5$. The sequential execution time remains nearly constant across all test cases, indicating that the CPU workload is mainly independent of $A$. In contrast, the parallel implementation completes the computation in a small fraction of a second. As a result, speed-ups exceeding 100× are consistently achieved, demonstrating the efficacy of the parallel strategy for evaluating the odd parabolic formulation. Figure 5 shows the validation of the Odd Parabolic function by comparing the outputs from the serial and the parallel executions. The two results are in good agreement over the entire time domain, also in regions close to extrema where the numerical sensitivity is higher. Such a close overlap confirms that the substantial computational speed-up achieved by parallel execution for the algorithms is with numerical accuracy, as confirmed by the MSE values presented in Table 5.

**Table 2. Speed-up for Even Parabolic function**

| Arguments | | | $T_{seq}(s)$ | $T_{par}(s)$ | Speed-up$(T_{seq}/T_{par})$ |
|---|---|---|---|---|---|
| $A$ | $N$ | $t$ | | | |
| 1 | 200 | -5:1e-5:5 | 35.108406 | 0.234194 | 149.9116374 |
| 2 | 200 | -5:1e-5:5 | 33.783613 | 0.23514 | 143.674462 |
| 3 | 200 | -5:1e-5:5 | 33.38298 | 0.240366 | 138.883952 |
| 4 | 200 | -5:1e-5:5 | 34.305408 | 0.246413 | 139.2191483 |
| 5 | 200 | -5:1e-5:5 | 34.285809 | 0.240158 | 142.7635515 |
| 6 | 200 | -5:1e-5:5 | 35.295322 | 0.237028 | 148.9078168 |
| 7 | 200 | -5:1e-5:5 | 35.297272 | 0.24796 | 142.3506695 |
| 8 | 200 | -5:1e-5:5 | 35.518056 | 0.247988 | 143.224898 |
| 9 | 200 | -5:1e-5:5 | 35.370777 | 0.245564 | 144.0389349 |

**Table 3. Mean-squared error for Even Parabolic function**

| Arguments | | | MSE |
|---|---|---|---|
| $A$ | $N$ | $t$ | |
| 1 | 200 | 5:1e-5:5 | 3.03E-31 |
| 2 | 200 | 5:1e-5:5 | 1.52E-30 |
| 3 | 200 | 5:1e-5:5 | 5.96E-30 |
| 4 | 200 | 5:1e-5:5 | 1.98E-29 |
| 5 | 200 | 5:1e-5:5 | 5.95E-29 |
| 6 | 200 | 5:1e-5:5 | 1.61E-28 |
| 7 | 200 | 5:1e-5:5 | 4.27E-28 |
| 8 | 200 | 5:1e-5:5 | 1.07E-27 |
| 9 | 200 | 5:1e-5:5 | 2.53E-27 |

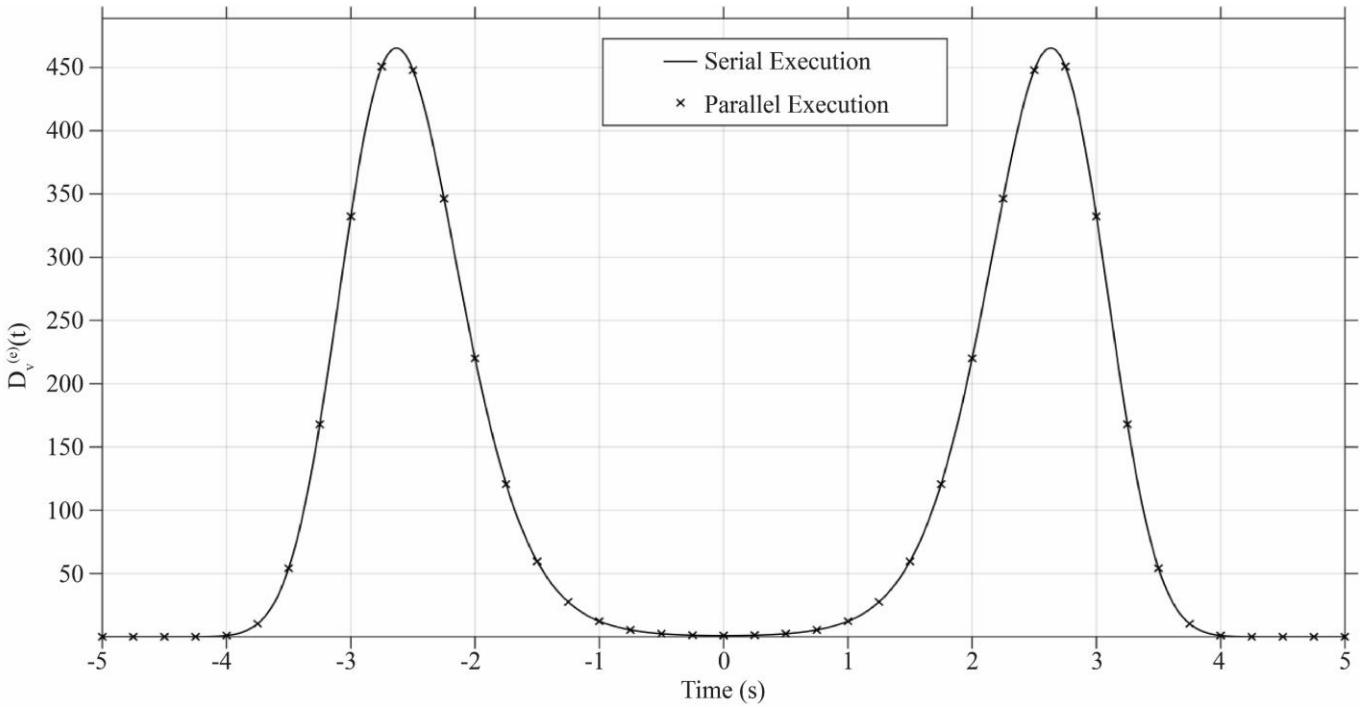


**Fig. 4 Validation of Even Parabolic function**

**Table 4. Speed-up for Odd Parabolic function**

| Arguments | | | $T_{seq}(s)$ | $T_{par}(s)$ | Speed-up$(T_{seq}/T_{par})$ |
|---|---|---|---|---|---|
| $A$ | $N$ | $t$ | | | |
| 0.1 | 200 | 5:1e-5:5 | 32.421434 | 0.23716 | 136.707 |
| 0.2 | 200 | 5:1e-5:5 | 31.981821 | 0.248174 | 128.869 |
| 0.3 | 200 | 5:1e-5:5 | 32.887151 | 0.248169 | 132.519 |
| 0.4 | 200 | 5:1e-5:5 | 32.946062 | 0.240758 | 136.843 |
| 0.5 | 200 | 5:1e-5:5 | 33.977335 | 0.238302 | 142.581 |
| 0.6 | 200 | 5:1e-5:5 | 33.944482 | 0.248187 | 136.77 |
| 0.7 | 200 | 5:1e-5:5 | 33.986502 | 0.237713 | 142.973 |
| 0.8 | 200 | 5:1e-5:5 | 33.984737 | 0.2484 | 136.815 |
| 0.9 | 200 | 5:1e-5:5 | 34.428411 | 0.23445 | 146.848 |

**Table 5. Mean-squared error for Odd Parabolic function**

| Arguments | | | MSE |
|---|---|---|---|
| $A$ | $N$ | $t$ | |
| 0.1 | 200 | 5:1e-5:5 | 2.57E-31 |
| 0.2 | 200 | 5:1e-5:5 | 7.14E-31 |
| 0.3 | 200 | 5:1e-5:5 | 1.91E-30 |
| 0.4 | 200 | 5:1e-5:5 | 4.89E-30 |
| 0.5 | 200 | 5:1e-5:5 | 1.15E-29 |
| 0.6 | 200 | 5:1e-5:5 | 2.72E-29 |
| 0.7 | 200 | 5:1e-5:5 | 6.16E-29 |
| 0.8 | 200 | 5:1e-5:5 | 1.32E-28 |
| 0.9 | 200 | 5:1e-5:5 | 2.86E-28 |

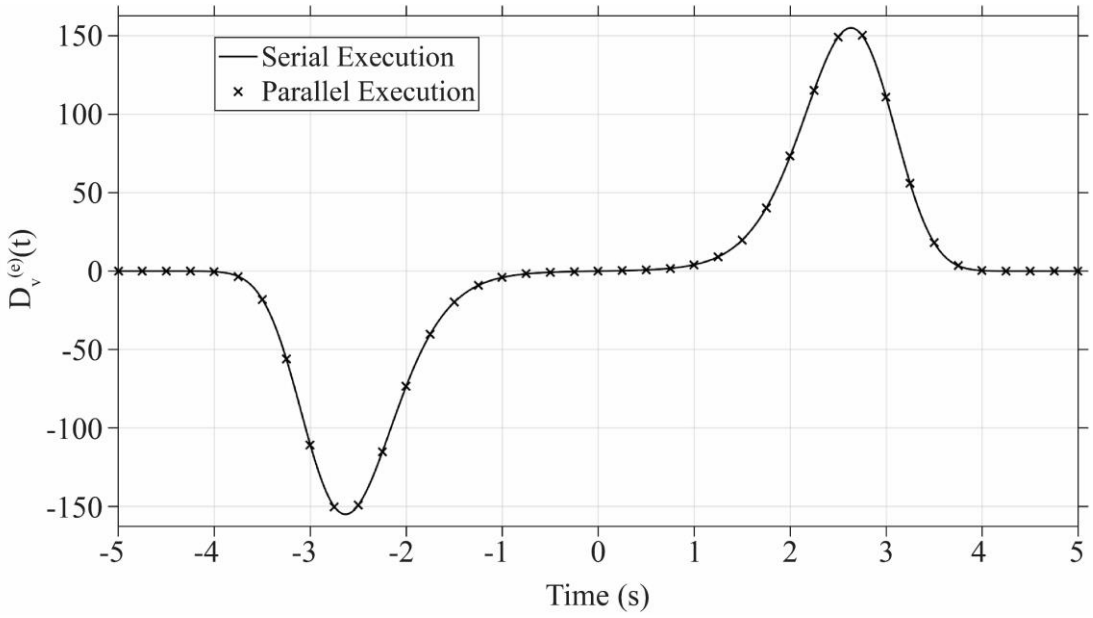


**Fig. 5 Validation of Odd Parabolic function**

### *6.3. Wright Function*

The time for the execution and resulting speed-up for the calculation of the Wright function based on different parameter choices of the α and β, are presented in Table 6, with the truncation order fixed at $N = 200$ and a fine time discretization $t = 0:10^{-5}:10$. The time for sequential execution varies very little across all parameter settings, and the time for parallel execution varies by only a few milliseconds. This results in a speed-up of 160 times, clearly indicating the efficiency of the GPU-based parallel implementation of the Wright function. Figure 6 shows a comparison of the values of the Wright function calculated during serial and parallel runs during the time interval considered. The parallel results show excellent numerical agreement, closely matching the serial reference solution over the entire domain. This close correspondence confirms that the substantial performance gains reported in Table 6 are achieved without compromising numerical accuracy, with the MSE results in Table 7 confirming the numerical equivalence of the sequential and parallel implementations.

### *6.4. Riemann Zeta Function*

The execution duration with obtained speed-up for the computation of the Riemann zeta function is presented in Table 8, for different readings of the parameter $s$, with a large truncation limit fixed at $N = 900000$ and a fine time grid $t = 0:10^{-5}:5$. The sequential execution time is essentially constant for all s values, demonstrating that the computational cost on the CPU is determined by the truncation size rather than the parameter. On the other hand, the parallel implementation reduces the execution time drastically to the order of microseconds. As a result, speed-up factors on the order of 82 to 84 times are consistently obtained. The results show that the parallel technique is effective in speeding up the evaluation of the Riemann zeta function while maintaining stable performance for diverse parameter values.

Validation of the Riemann zeta function by comparing the results obtained from serial and parallel executions over the considered range of $s$ given in Figure 7. The parallel results closely follow the serial reference curve throughout the domain, including the region near $s \approx 1$, where the function exhibits rapid variation. The near-complete overlap confirms that the parallel implementation preserves numerical accuracy while achieving the substantial performance improvements reported in Table 8. The low MSE values reported in Table 9 further provide quantitative evidence of the accuracy and robustness of the proposed parallel implementation.

**Table 6. Speed-up for the Wright function**

| Arguments | | | | $T_{seq}(s)$ | $T_{par}(s)$ | Speed-up $(T_{seq}/T_{par})$ |
|---|---|---|---|---|---|---|
| $\alpha$ | $\beta$ | $N$ | $t$ | | | |
| 0.1 | 0.1 | 200 | 0:1e-5:10 | 1.141606 | 0.00707 | 161.472 |
| 0.2 | 0.7 | 200 | 0:1e-5:10 | 1.141066 | 0.007082 | 161.122 |
| 0.3 | 0.5 | 200 | 0:1e-5:10 | 1.141347 | 0.007072 | 161.39 |
| 0.4 | 0.4 | 200 | 0:1e-5:10 | 1.14106 | 0.007073 | 161.326 |
| 0.5 | 0.2 | 200 | 0:1e-5:10 | 1.141054 | 0.007073 | 161.325 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 0.6 | 0.4 | 200 | 0:1e-5:10 | 1.141205 | 0.007068 | 161.461 |
| 0.7 | 0.7 | 200 | 0:1e-5:10 | 1.141076 | 0.007072 | 161.351 |
| 0.8 | 0.3 | 200 | 0:1e-5:10 | 1.141401 | 0.007067 | 161.511 |
| 0.9 | 0.9 | 200 | 0:1e-5:10 | 1.14104 | 0.00704 | 162.08 |

**Table 7. Mean-squared error for Wright function**

| Arguments | | | | MSE |
|---|---|---|---|---|
| $\alpha$ | $\beta$ | $N$ | $t$ | |
| 0.1 | 0.1 | 200 | 0:1e-5:10 | 3.17E-33 |
| 0.2 | 0.7 | 200 | 0:1e-5:10 | 1.19E-31 |
| 0.3 | 0.5 | 200 | 0:1e-5:10 | 3.13E-31 |
| 0.4 | 0.4 | 200 | 0:1e-5:10 | 1.41E-30 |
| 0.5 | 0.2 | 200 | 0:1e-5:10 | 5.51E-30 |
| 0.6 | 0.4 | 200 | 0:1e-5:10 | 8.94E-30 |
| 0.7 | 0.7 | 200 | 0:1e-5:10 | 2.73E-29 |
| 0.8 | 0.3 | 200 | 0:1e-5:10 | 9.40E-29 |
| 0.9 | 0.9 | 200 | 0:1e-5:10 | 3.53E-29 |

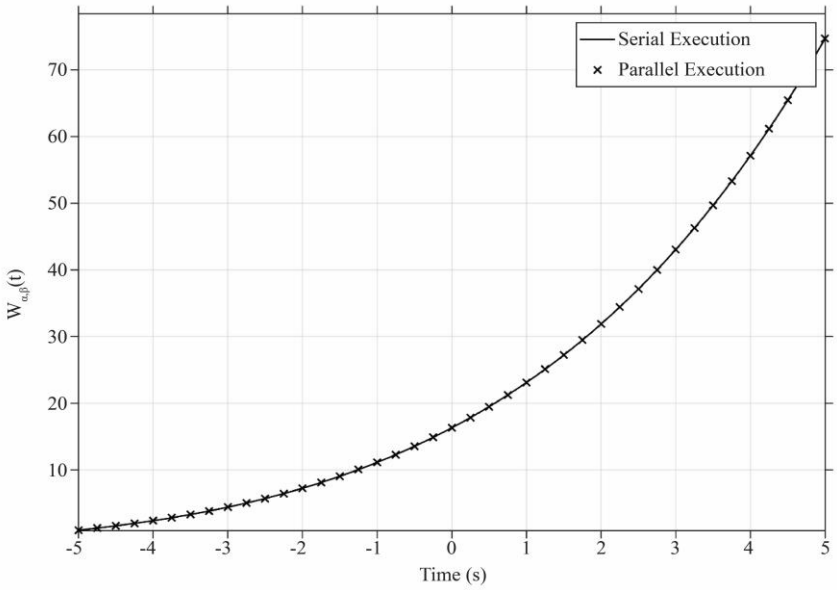


**Fig. 6 Validation of the Wright function**

**Table 8. Speed-up for Riemann Zeta function**

| Arguments | | | $T_{seq}(s)$ | $T_{par}(s)$ | Speed-up $(T_{seq}/T_{par})$ |
|---|---|---|---|---|---|
| $s$ | $N$ | $t$ | | | |
| 1 | 900000 | 0:1e-5:5 | 0.040729 | 0.000486 | 83.8045 |
| 2 | 900000 | 0:1e-5:5 | 0.040726 | 0.000484 | 84.1446 |
| 3 | 900000 | 0:1e-5:5 | 0.040701 | 0.000492 | 82.7256 |
| 4 | 900000 | 0:1e-5:5 | 0.040732 | 0.000493 | 82.6207 |
| 5 | 900000 | 0:1e-5:5 | 0.040715 | 0.000496 | 82.0867 |
| 6 | 900000 | 0:1e-5:5 | 0.040727 | 0.000492 | 82.7785 |
| 7 | 900000 | 0:1e-5:5 | 0.04072 | 0.000489 | 83.272 |
| 8 | 900000 | 0:1e-5:5 | 0.04072 | 0.00049 | 83.102 |
| 9 | 900000 | 0:1e-5:5 | 0.040715 | 0.000487 | 83.6037 |

**Table 9. Mean-squared error for Riemann Zeta function**

| Arguments | | | MSE |
|---|---|---|---|
| ***s*** | ***N*** | ***t*** | |
| 1 | 900000 | 0:1e-5:5 | 3.63E-32 |
| 2 | 900000 | 0:1e-5:5 | 3.29E-32 |
| 3 | 900000 | 0:1e-5:5 | 3.19E-32 |
| 4 | 900000 | 0:1e-5:5 | 3.23E-32 |
| 5 | 900000 | 0:1e-5:5 | 3.19E-32 |
| 6 | 900000 | 0:1e-5:5 | 2.53E-32 |
| 7 | 900000 | 0:1e-5:5 | 2.39E-32 |
| 8 | 900000 | 0:1e-5:5 | 3.26E-32 |
| 9 | 900000 | 0:1e-5:5 | 3.59E-32 |

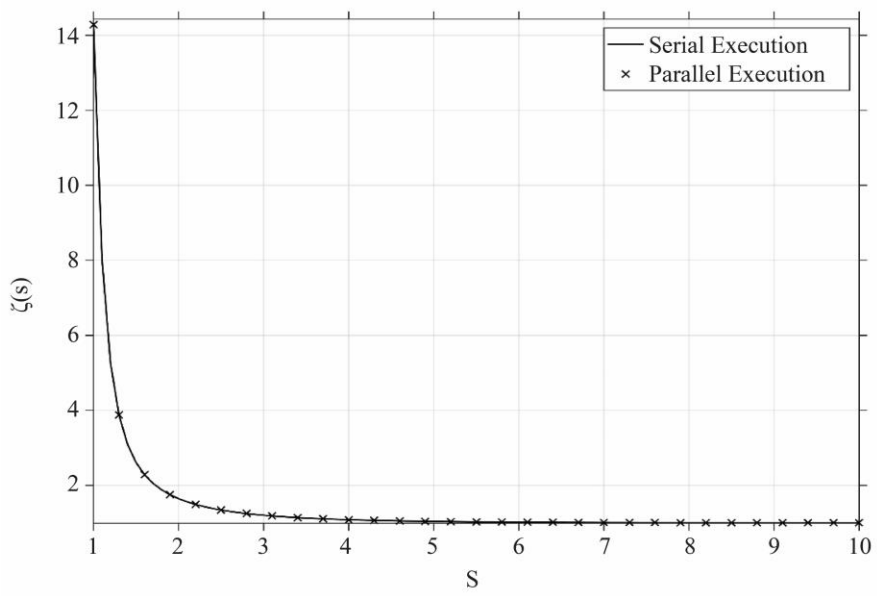


**Fig. 7 Validation of Riemann Zeta function**

### *6.5. Observation*

- Implementation of CUDA-based parallel algorithm for Parabolic Cylinder, Wright and Riemann Zeta functions achieved significant speed-up compared to sequential CPU implementation, indicating the effectiveness of GPU parallelization.
- Parallel execution of independent arithmetic operations and recurrence computations in the case of the Parabolic Cylinder function has led to a noticeable reduction in execution time.
- The GPU implementation was efficient in the case of the Wright function with respect to repeated series expansions and gamma function evaluations, leading to stable computational performance and better throughput.
- The Riemann Zeta function enjoyed a major speed-up under CUDA execution, due to parallel summation and optimized kernel execution on GPU architectures.
- The results obtained show that the variations of the parameters of the functions and the computational ranges do not significantly influence the robustness and accuracy of the parameters of the proposed GPU-based algorithms.
- Improved computational efficiency and reduced execution latency for all functions through efficient utilization of GPU resources, e.g., thread-level parallelism and memory bandwidth.
- The proposed CUDA-based parallel algorithms showed better scalability, higher execution speed, and better numerical performance than conventional CPU-oriented sequential methods, with accuracy of results.

## 7. Conclusion

The mathematical functions useful in non-integer calculus, namely, the Even Parabolic, Odd Parabolic, Wright, and Riemann Zeta functions, are computed in parallel framework using CUDA on GPU platform using the CUDA framework. The acceleration of computing these special mathematical functions is achieved through parallel processing algorithms that exploit multiple processing cores of the graphics processor via CUDA kernels. The series representations of these functions were modified to facilitate their parallel computation. This led to a significant reduction in the computation time, while maintaining a high numerical

accuracy. The results demonstrate that these functions are well suited for thread-level parallelization and can achieve speed-ups of 90x to 160x over conventional CPU-based sequential implementations. The reliability of the approach has been confirmed by comparing the results of the proposed parallel algorithms with those of their serial execution, as well as by demonstrating the algorithms' convergence and accuracy. Moreover, the developed parallel computing framework can handle large-scale simulations of fractional calculus problems. Possible future directions may include the use of adaptive precision techniques, implementation in general-purpose solvers for more complex fractional-order differential equations and real-time implementation on embedded GPU platforms.

## References


[1] Jacques E. Romain, *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, American Institute of Physics, 1966. [CrossRef] [Google Scholar] [Publisher Link]

[2] Martin Crowder, *NIST Handbook of Mathematical Functions edited by Frank W. J. Olver, Daniel W. Lozier, Ronald F. Boisvert, Charles W. Clark*, International Statistical Review, vol. 79, no. 1, pp. 131-132, 2011. [CrossRef] [Google Scholar] [Publisher Link]

[3] E. M. Wright, "The Generalized Bessel Function of Order Greater than One," *The Quarterly Journal of Mathematics*, vol. 11, no. 1, pp. 36-48, 1940. [CrossRef] [Google Scholar] [Publisher Link]

[4] R. Hilfer, *Applications of Fractional Calculus in Physics*, World Scientific Publishing, pp. 1-472, 2000. [CrossRef] [Google Scholar] [Publisher Link]

[5] Francesco Mainardi, "A Tutorial on the Basic Special Functions of Fractional Calculus," *WSEAS Transactions on Mathematics*, vol. 19, pp. 74-98, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[6] H. Heilbronn, *The Theory of the Riemann Zeta-Functions, By E. C. Titchmarsh Pp. 346. 40s. 1951. (Oxford University Press)*, The Mathematical Gazette, vol. 36, no. 317, pp. 230-231, 1952. [CrossRef] [Google Scholar] [Publisher Link]

[7] John P. Boyd, *Chebyshev and Fourier Spectral Method*, Dover Publications, 2001. [Google Scholar] [Publisher Link]

[8] I. Podlubny, *Numerical Solution of Fractional Differential Equations*, Mathematics in Science and Engineering, vol. 198, pp. 223-242, 1999. [CrossRef] [Google Scholar] [Publisher Link]

[9] Ralf Metzler, and Joseph Klafter, "The Random Walk's Guide to Anomalous Diffusion: A Fractional Dynamics Approach," *Physics Reports*, vol. 339, no. 1, pp. 1-77, 2000. [CrossRef] [Google Scholar] [Publisher Link]

[10] Yury Luchko, "Maximum Principle and its Application for the Time-Fractional Diffusion Equations," *Fractional Calculus and Applied Analysis*, vol. 14, no. 1, pp. 110-124, 2011. [CrossRef] [Google Scholar] [Publisher Link]

[11] Andrzej Dzieliński, and Dominik Sierociuk, "Stability of Discrete Fractional Order State-Space Systems," *Journal of Vibration and Control*, vol. 14, no. 9-10, pp. 1543-1556, 2008. [CrossRef] [Google Scholar] [Publisher Link]

[12] Tolga Soyata, *Introduction to GPU Parallelism and CUDA*, GPU Parallel Program Development Using CUDA, CRC Press, Boca Raton, FL, USA, pp. 137-183, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[13] D. N. Tumakov, "The Faster Methods for Computing Bessel Functions of the First Kind of an Integer Order with Application to Graphic Processors," *Lobachevskii Journal of Mathematics*, vol. 40, no. 10, pp. 1725-1738, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[14] Fredrik Johansson, "Computing Hypergeometric Functions Rigorously," *ACM Transactions on Mathematical Software*, vol. 45, no. 3, pp. 1-26, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[15] Roberto Garrappa, and Marina Popolizio, "Evaluation of Generalized Mittag–Leffler Functions on the Real Line," *Advances in Computational Mathematics*, vol. 39, no. 1, pp. 205-225, 2012. [CrossRef] [Google Scholar] [Publisher Link]

[16] David B. Kirk, and Wen-mei W. Hwu, *Parallel Programming and Computational Thinking*, Programming Massively Parallel Processors, pp. 281-295, 2013. [CrossRef] [Google Scholar] [Publisher Link]

[17] Volodymyr V. Kindratenko et al., "GPU Clusters for High-Performance Computing," *2009 IEEE International Conference on Cluster Computing and Workshops*, New Orleans, LA, USA, pp. 1-8, 2009. [CrossRef] [Google Scholar] [Publisher Link]

[18] Rudolf Gorenflo et al., *Mittag-Leffler Functions, Related Topics and Applications*, Springer Berlin Heidelberg, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[19] Andrada Baban et al., "Parallel Simulations for Fractional-Order Systems," *2016 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, Timisoara, Romania, pp. 141-144, 2016. [CrossRef] [Google Scholar] [Publisher Link]

[20] Shidong Jiang et al., "Fast Evaluation of the Caputo Fractional Derivative and Its Applications to Fractional Diffusion Equations," *Communications in Computational Physics*, vol. 21, no. 3, pp. 650-678, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[21] Luis X. Vivas-Cruz et al., "Hybrid Finite Element and Laplace Transform Method for Efficient Numerical Solutions of Fractional PDEs on Graphics Processing Units," *Physica Scripta*, vol. 99, no. 10, 2024. [CrossRef] [Google Scholar] [Publisher Link]

[22] Anatoly A. Kilbas, Hari M. Srivastava, and Juan J. Trujillo, "Preface," *Theory and Applications of Fractional Differential Equations, North-Holland Mathematics Studies*, vol. 204, pp. 7-10, 2006. [CrossRef] [Publisher Link]

[23] Richard Herrmann, *Fractional Calculus: An Introduction for Physicists*, World Scientific, 2011. [CrossRef] [Google Scholar] [Publisher Link]

[24] Vasily E. Tarasov, *Fractional Dynamics*, Berlin, Germany: Springer Berlin Heidelberg, 2010. [CrossRef] [Google Scholar] [Publisher Link]

[25] Richard L. Magin, "Fractional Calculus in Bioengineering: A Tool to Model Complex Dynamics," *Proceedings of the 13th International Carpathian Control Conference (ICCC)*, High Tatras, Slovakia, pp. 464-469, 2012. [CrossRef] [Google Scholar] [Publisher Link]

[26] Kai Diethelm, *The Analysis of Fractional Differential Equations: An Application-Oriented Exposition using Differential Operators of Caputo Type*, Springer Berlin Heidelberg, 2010. [CrossRef] [Google Scholar] [Publisher Link]

[27] Back Matter, *Fractional Calculus and Waves in Linear Viscoelasticity*, pp. 155-347, 2010. [CrossRef] [Publisher Link]

[28] Wei-Ching Chen, "Nonlinear Dynamics and Chaos in a Fractional-Order Financial System," *Chaos, Solitons & Fractals*, vol. 36, no. 5, pp. 1305-1314, 2008. [CrossRef] [Google Scholar] [Publisher Link]

[29] G. Dattoli, "Generalized Polynomials, Operational Identities and Their Applications," *Journal of Computational and Applied Mathematics*, vol. 118, no. 1-2, pp. 111-123, 2000. [CrossRef] [Google Scholar] [Publisher Link]

[30] H. van Haeringen, and L. P. Kok, "Higher Transcendental Functions," *Mathematics of Computation*, vol. 41, 1983. [CrossRef] [Google Scholar] [Publisher Link]

[31] J.L. Blanchard, and E.H. Newman, "Numerical Evaluation of Parabolic Cylinder Functions," *IEEE Transactions on Antennas and Propagation*, vol. 37, no. 4, pp. 519-522, 1989. [CrossRef] [Google Scholar] [Publisher Link]

[32] V. V. Anh, and N. N. Leonenko, "Spectral Analysis of Fractional Kinetic Equations with Random Data," *Journal of Statistical Physics*, vol. 104, pp. 1349-1387, 2001. [CrossRef] [Google Scholar] [Publisher Link]

[33] Humberto Rafeiro, and Stefan Samko, "Fractional Integrals and Derivatives: Mapping Properties," *Fractional Calculus and Applied Analysis*, vol. 19, no. 3, pp. 580-607, 2016. [CrossRef] [Google Scholar] [Publisher Link]

[34] R.B. Paris, "Exponentially Small Expansions in the Asymptotics of the Wright Function," *Journal of Computational and Applied Mathematics*, vol. 234, no. 2, pp. 488-504, 2010. [CrossRef] [Google Scholar] [Publisher Link]

[35] Ralf Metzler, and Joseph Klafter, "The Restaurant at the End of the Random Walk: Recent Developments in the Description of Anomalous Transport by Fractional Dynamics," *Journal of Physics A: Mathematical and General*, vol. 37, no. 31, 2004. [CrossRef] [Google Scholar] [Publisher Link]

[36] Yuri Luchko, "Fractional Derivatives and the Fundamental Theorem of Fractional Calculus," *Fractional Calculus and Applied Analysis*, vol. 23, no. 4, pp. 939-966, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[37] E. Elizalde, *Physical Application: The Casimir Effect*, Ten Physical Applications of Spectral Zeta Functions, pp. 97-127, 1995. [CrossRef] [Google Scholar] [Publisher Link]

[38] Fredrik Johansson, "Arb: Efficient Arbitrary-Precision Midpoint-Radius Interval Arithmetic," *IEEE Transactions on Computers*, vol. 66, no. 8, pp. 1281-1292, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[39] John Nickolls et al., "Scalable Parallel Programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40-53, 2008. [CrossRef] [Google Scholar] [Publisher Link]

[40] Peter S. Pacheco, and Matthew Malensek, *GPU Programming with CUDA*, An Introduction to Parallel Programming, 2nd ed., Philadelphia: Morgan Kaufmann, pp. 291-360, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[41] Lin Shi et al., "vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines," *IEEE Transactions on Computers*, vol. 61, no. 6, pp. 804-816, 2012. [CrossRef] [Google Scholar] [Publisher Link]

[42] John D. Owens et al., "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879-899, 2008. [CrossRef] [Google Scholar] [Publisher Link]