SoftwareX 29 (2025) 102009

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

BetaGPU: Harnessing GPU power for parallelized beta distribution functions

Alejandro Fernández-Fraga, Jorge González-Domínguez *, María J. Martín

Computer Architecture Group, CITIC, Universidade da Coruña, A Coruña, Spain

ARTICLE INFO

Keywords:
Beta distribution
High performance computing
GPU
CUDA
OpenMP
R
Python

ABSTRACT

The efficient computation of Beta distribution functions, particularly the Probability Density Function (PDF) and Cumulative Distribution Function (CDF), is critical in various scientific fields, including bioinformatics and data analysis. This work presents BetaGPU, a high-performance software package written in C++ and CUDA that leverages the parallel processing capabilities of Graphics Processing Units (GPUs) to significantly accelerate these computations, with an OpenMP version for multiCPU systems, and integrated seamlessly with popular statistical programming languages R and Python. This open-source package provides an accessible, accurate, and scalable solution for researchers and practitioners. By offloading intensive calculations to the GPU, this software is significantly faster than traditional single-core CPU-based methods, facilitating faster data analysis and enabling real-time applications. The software's high performance and ease of use make it an invaluable tool for users in bioinformatics and other data-intensive domains.

Code metadata

Current code version	v1.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-24-00555
Permanent link to Reproducible Capsule	–
Legal Code License	GNU General Public License (GPL)
Code versioning system used	Git
Software code languages, tools, and services used	C++, Python, R, CUDA
Compilation requirements, operating environments & dependencies	CUDA Toolkit, GCC, CMake
If available Link to developer documentation/manual	https://github.com/UDC-GAC/BetaGPU
Support email for questions	a.fernandez3@udc.es

1. Motivation and significance

The Beta distribution is a continuous probability distribution defined on the interval [0, 1] in terms of two shape parameters: alpha (α) and beta (β). It is of particular importance in Bayesian statistics, where it serves as a conjugate prior for binomial proportions. The Beta distribution is crucial in various fields with an increasing demand in efficient statistical computation to perform Big Data analyses in reasonable time, such as finance, machine learning or biology. For example, the tools `CancerDetector` [1] and `CancerLocator` [2] use it to characterize methylation patterns and detect tumor-derived cell-free DNA in plasma for cancer diagnosis; the R software package `freqpcr` [3] uses the Beta distribution to model population allele frequencies; and `MDF-SA-DD` [4], a machine learning tool used to predict drug-drug interactions, employs it in the data augmentation process. Moreover, different studies have also used the Beta distribution to model heterogeneity in character change rates in morphological phylogenetic analyses [5], assess risk in financial networks [6] or even detect fake news in social media [7]. This variety of applications highlights the importance of the Beta distribution, showcasing its versatility and effectiveness in modeling a wide range of data.

The computational complexity of the Beta distribution functions, especially when handling large datasets or performing extensive simulations, which is common in the above-mentioned fields, poses significant challenges. Several software packages and algorithms exist for Beta distribution computations, but they typically rely on CPU-based calculations. Notable examples include `scipy.stats` [8] in Python,

* Corresponding author.
E-mail address: jorge.gonzalezd@udc.es (Jorge González-Domínguez).

https://doi.org/10.1016/j.softx.2024.102009
Received 17 October 2024; Received in revised form 5 December 2024; Accepted 6 December 2024
Available online 18 December 2024

`R-stats`[1] in R and the GNU Scientific Library (GSL)[2] in C/C++. These tools, while robust and widely used, do not leverage the parallel processing capabilities of GPUs, limiting their performance with large datasets.

While no GPU-accelerated approaches specifically addresses the beta distribution, there are other High Performance Computing (HPC) libraries designed to tackle a variety of computational challenges on large datasets. Probably, the best-known library is NVBIO[3] developed by NVIDIA as a modular library which includes data structures, algorithms, and utility routines useful for building complex computational genomics applications on their GPUs. Other examples include GASAL [9] and GASAL 2 [10] (both focused on sequence alingment) or bioScience [11], a novel library that integrates Python and parallel programming to accelerate bioinformatics pipelines on different HPC facilities. These solutions highlight the potential of HPC frameworks to significantly enhance the efficiency of large-scale data processing routines

Additionally, recent advancements in GPU computing, such as NVIDIA's cuRAND[4] and cuDNN[5] libraries, have shown the potential for significant performance improvements in statistical computations. BetaGPU builds on these advancements, offering a specialized solution for Beta distribution functions that surpasses the capabilities of existing software.

Our library is a high-performance software package developed using C++ and CUDA that leverages the parallel processing power of Graphics Processing Units (GPUs). GPUs offer the advantage of being more energy-efficient for the performance they deliver, which represents a significant benefit for the software. By applying array programming and offloading computations to the GPU, the package can compute the Beta distribution functions significantly faster than traditional CPU array-based implementations. In addition, a version for multiple CPUs using OpenMP is also included.

The software is designed to be user-friendly, with straightforward integration into existing R and Python workflows, following the same API as popular statistical modules such as the aforesaid `scipy.stats` [8] and `R-stats`[2]. Users can install the package via standard package managers (e.g., CRAN for R and PyPI for Python) and utilize it through familiar function calls, ensuring a seamless adoption process. In addition, the software is based on GSL[3] in order to guarantee precision in the computations, ensuring that the results maintain high accuracy and reliability.

By addressing the need for high-performance statistical computations, this software represents a significant advancement in the field, as it provides researchers and practitioners with a powerful tool to accelerate their work and facilitate new discoveries. The software presented in this article enables order-of-magnitude faster computations, while its results maintain the original high accuracy and precision of an implementation based on the functions available on the widely used and tested GSL library.

2. Software description

The software provides implementations for the Probability Density Function (PDF) and the Cumulative Distribution Function (CDF), which are the most computationally demanding functions used in statistical analysis of the Beta distribution. The PDF represents the probability of a continuous random variable taking a specific value, while the CDF represents the probability of the random variable being less than or equal to a given value.

[1] https://www.rdocumentation.org/packages/stats/versions/3.6.2
[2] https://www.gnu.org/software/gsl/
[3] https://developer.nvidia.com/nvbio
[4] https://docs.nvidia.com/cuda/curand/index.html
[5] https://developer.nvidia.com/cudnn

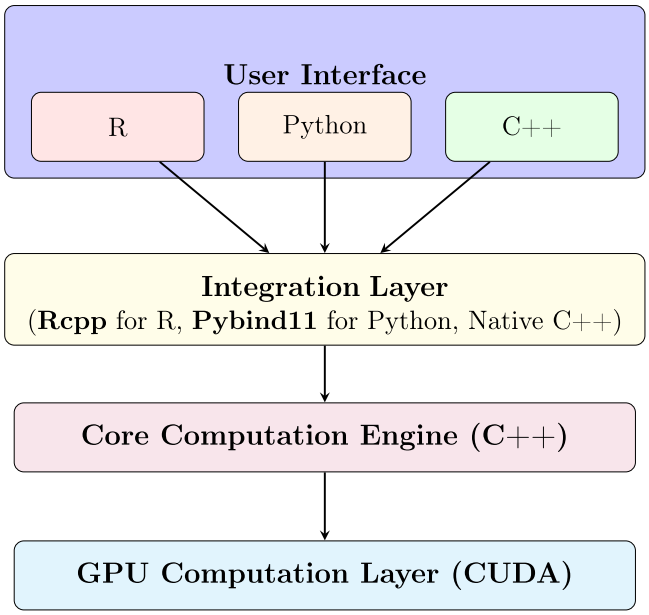

Fig. 1. BetaGPU high level architecture.

In particular for the Beta distribution, the PDF is defined as:

$$f(x;\alpha,\beta) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)} \tag{1}$$

where $B(\alpha,\beta)$ is the beta function, defined as:

$$B(\alpha,\beta) = \int_0^1 t^{\alpha-1}(1-t)^{\beta-1}dt \tag{2}$$

And for the Beta distribution, the CDF is defined as:

$$F(x;\alpha,\beta) = \frac{B(x;\alpha,\beta)}{B(\alpha,\beta)} \tag{3}$$

where $B(x;\alpha,\beta)$ is the incomplete beta function, defined as:

$$B(x;\alpha,\beta) = \int_0^x t^{\alpha-1}(1-t)^{\beta-1}dt \tag{4}$$

2.1. Software architecture

The software provides functions to compute the PDF and CDF of the Beta distribution using the GPU, as well as a mixed approach that combines CPU and GPU computations. The software is designed to be user-friendly, with a simple and intuitive interface, which is common in C++, R and Python.

BetaGPU's architecture can be depicted as shown in Fig. 1. It is structured into several layers, each one serving a specific purpose to ensure optimal performance and usability.

- **GPU Computation Layer (CUDA)**: Utilizes GPU parallelism to perform intensive calculations rapidly, significantly speeding up Beta distribution function evaluations.
- **Core Computation Engine (C++)**: Handles the primary logic for Beta distribution computations and memory transfers, while offloading the necessary computations to the GPU. It is specifically designed for high performance and compatibility with the GPU computation layer.
- **Integration Layer**: Uses Rcpp for R, Pybind11 for Python, and native C++ interfaces to create seamless integrations, allowing users to call the high-performance functions from their preferred statistical environments.
- **User Interface**: Provides the functions available to the end user to facilitate ease of use in R, Python, and C++.

Algorithm 1: Pseudocode for the Beta PDF function

Input: A data point x in the range [0, 1], shape parameters α and β
Output: A PDF value

$log\_B \leftarrow log_e(\alpha + \beta) - log_e(\alpha) - log_e(\beta)$
$B \leftarrow e^{log\_B}$
$pdf \leftarrow x^{\alpha-1} * (1-x)^{\beta-1} * B$
return pdf

Algorithm 2: Pseudocode for the Beta CDF function

Input: A data point x in the range [0, 1], shape parameters α and β
Output: A CDF value

`/* Compute the prefactor */`
$log\_B \leftarrow log_e(\alpha + \beta) - log_e(\alpha) - log_e(\beta)$
$log\_prefactor \leftarrow -log\_B + \alpha * log_e(x) + b * log_e(1-x)$
$prefactor \leftarrow e^{log\_prefactor}$
`/* Compute the continued fraction */`
$continued\_fraction \leftarrow compute\_continued\_fraction(x, \alpha, \beta)$
`/* Compute the CDF */`
$cdf \leftarrow prefactor * continued\_fraction/\alpha$
return cdf

2.2. Implementation details

As shown in Eq. (1), the PDF function only involves a simple mathematical expression. To increase the numerical stability of the computations, and avoid intermediate underflow or overflow, the software uses the logarithm of the gamma function as an intermediate value and then exponentiates the result. Algorithm 1 shows the pseudocode for the PDF function.

The CDF is more complex to compute, as it involves the incomplete beta function shown in Eq. (4), which is not as straightforward, as shown in Eq. (3). In addition to performing some mathematical operations (called *prefactor* in Algorithm 2), it is also necessary to compute the continued fraction expansion, an iterative method that needs a different number of iterations to converge depending on the input values. The higher the shape parameters α and β, the more iterations are needed for the continued fraction to converge and the larger the difference in iterations to converge among data points. For some extreme values of α and β, the continued fraction may have up to ten times more computational cost than the prefactor. Moreover, while the calculation of the prefactor is ideal for GPU execution, an iterative method makes it a rough candidate for GPU parallelization. That is why the software includes two GPU approaches to compute the CDF: one that uses the GPU only to compute the prefactor and the CPU to compute the continued fraction; and another that uses the GPU to compute both the prefactor and the continued fraction. Algorithm 2 shows the pseudocode for the CDF function.

The parallel approach to distribute the work among the processing elements is the same for all the functions included in the software: each processing element gets a data point form the input array. That is, for the GPU, one thread is created for each data point and for the CPU, data points are dynamically assigned to the OpenMP threads.

The iterative algorithm used for computing the CDF poses a challenge for the CPU and the GPU implementations, as it creates a workload imbalance among the data points. That is, one data point may take more iterations to converge than other. For the CPU this was solved by using the OpenMP dynamic scheduling, ensuring that all threads are kept busy and the workload is distributed evenly. However, the GPU presents a more complex challenge due to its SIMD architecture, which requires all threads in a warp to execute the same instruction. In other words, the iterative nature of the algorithm can lead to divergent execution paths, where most of the threads remain idle waiting for the threads that take the most iterations to converge in each warp. This could lead to an underutilization of the GPUs resources, and can degrade the performance of the algorithm. The challenge was addressed by assigning data points that are close to each other to the same thread block, since, for this particular algorithm, the divergence is minimized when the data points are close. As a result of applying this strategy, the method maximizes the utilization of the GPU resources.

In addition to the computational challenges, memory management and data transfer between the CPU and GPU are critical aspects of the software's performance. For that reason, it is recommended to use pinned memory, special CPU memory pages that cannot be paged out. This memory improves the performance of the software when transferring data because it only involves one copy between the CPU and the GPU, instead of the normal paged memory which requires two copies: one between CPU paged memory and CPU pinned memory, and another one from CPU to GPU. Regarding memory management during kernel execution, initially data is available in the GPU global memory, which is then read by the threads and stored in registers (the fastest memory available). Shared memory is not used, as the threads do not need to communicate with each other during the computation of the functions and there are no repetitive memory reads that can benefit from its low latency. Finally, the results are written back from registers to global memory only once, so the global memory is only used when strictly necessary. This means that throughout the computation, threads rely on registers to store all its information. This can be dangerous, as the number of registers is limited and an oversubscription can limit occupancy. To avoid this, the software has been optimized to ensure that the register pressure is kept low.

On top of that, GPU memory may not be enough to store the whole array all at once, and a chunk version of the functions was implemented. That is, instead of transferring all the data points at once, only a block of them is copied between devices at a time, allowing to alleviate the memory requirements of the software, so that it can adapt to any GPU. To maximize the throughput of the software, this chunk version was implemented using different CUDA streams to overlap communications with computations, ensuring efficient utilization of the GPU resources. During a preliminary evaluation, it was proven that the optimal configuration was to use two CUDA streams and a chunk size as big as possible, according to the memory available in the GPU. All of these optimizations are transparent to the user. This means that the library automatically detects the memory size of the GPU and takes into account the array length in the input so that the user can benefit from the software's high performance without needing to understand the underlying complexities.

Listing 1 illustrates how this chunked version is implemented. This code snippet is a simplified version of the actual code, as it does not take into account error checking, data elements not being a multiple of the chunk size and other details that are left out for the sake of clarity. First, the function checks the available memory in the GPU and, based on that, calculates the number of chunks that can be used to compute the functions. Then, it creates the necessary CUDA streams and allocates the memory in the GPU. After that, it iterates over the chunks, copying the data from the CPU to the GPU, executing the kernel and copying the results back to the CPU. Finally, it destroys the streams and frees the memory in the GPU.

With regards to the GPU kernel execution configuration, the software uses a one-dimensional grid of blocks, where each block has a fixed number of threads. The one-dimensional grid has been chosen to match the nature of the data. Block size is a critical parameter for the performance of the software, so values that achieve the maximum occupancy for different compute capabilities have been checked using the NVIDIA Nsight Compute occupancy calculator tool. Among the ones that achieve the maximum occupancy, 256 threads per block has been selected based on empirical testing, as it was the best for all different

problem sizes tested. Lastly, the software showed no performance benefit from asynchronous execution, so multiple CUDA streams were only used in the chunk version of the functions.

Listing 1 Example of the chunk-based function to compute the Beta distribution functions

```
template <typename data_t, typename kernel_t>
void beta_func_chunked(data_t* input_h, data_t* output_h,
                       data_t alpha, data_t beta,
                       size_t size, kernel_t kernel){
 const int N_STREAMS = 2;
 const int BLOCK_SIZE = 256;

 // Get GPU memory information
 size_t free_bytes, total_bytes;
 cudaMemGetInfo( &free_bytes, &total_bytes);
 size_t needed_bytes = (2 * size) + 2 * sizeof(data_t);

 // Get the chunks
 int n_chunks = getNumChunks(free_bytes, needed_bytes);
 size_t chunk_size = getChunkSize(size, n_chunks);
 int n_blocks = (chunk_size + BLOCK_SZ - 1) / BLOCK_SZ;

 // Create streams and allocate memory in the GPU
 cudaStream_t streams[N_STREAMS];
 data_t *input_d[N_STREAMS], *output_d[N_STREAMS];
 for (int i = 0; i < N_STREAMS; i++) {
  cudaStreamCreate(&streams[i]);
  cudaMalloc(&input_d[i], chunk_size * sizeof(data_t));
  cudaMalloc(&output_d[i], chunk_size * sizeof(data_t));
 }

 // Process chunks
 for (int i = 0; i < n_chunks; i++) {
  int s_id = i % N_STREAMS; // Stream_index
  cudaMemcpyAsync(input_d[s_id], input_h
    + (i * chunk_size),
    chunk_size * sizeof(data_t), cudaMemcpyHostToDevice,
    streams[s_id]);
  kernel<<<n_blocks, BLOCK_SIZE, 0,
  streams[s_id]>>>(input_d[i],
    output_d[i], alpha, beta, chunk_size);
  cudaMemcpyAsync(output_h + (i * chunk_size),
    output_d[i],
    chunk_size * sizeof(data_t), cudaMemcpyDeviceToHost,
    streams[s_id]);
 }

 // Destroy streams and Free the memory in the GPU
 for (int i = 0; i < NUM_STREAMS; i++) {
  cudaStreamDestroy(streams[i]);
  cudaFree(input_d[i]);
  cudaFree(output_d[i]);
 }
}
```

2.3. Software functionalities

BetaGPU exposes five different functions for the computation of the PDF and CDF, using either a CPU multithreaded implementation with OpenMP, a GPU implementation with CUDA, or a mixed implementation of the CDF using both CUDA and OpenMP for different parts of the computation. All of them are exposed through the different interfaces to R, Python and C++, but with slightly different naming conventions tailored to the specific language, in order to make it easier for users to adopt the software. For the sake of simplicity, in this section we will refer to the functions using their C++ names. More information about naming convention is available in the reference manual of the library, which can be downloaded from the git repository of the tool. Each function takes as input the shape parameters α and β of the Beta distribution, and an array of data points x, and returns the corresponding PDF or CDF values for each data point. The interface includes the following functions:

- `ompBetaPDF`: Computes the PDF of the Beta distribution using the CPU with multiple threads.
- `ompBetaCDF`: Computes the CDF of the Beta distribution using the CPU with multiple threads.
- `cudaBetaPDF`: Computes the PDF of the Beta distribution using the GPU.
- `cudaBetaCDF`: Computes the CDF of the Beta distribution using the GPU.
- `cudaOmpBetaCDF`: Computes the CDF of the Beta distribution using a mixed CPU–GPU approach.

2.4. Code snippets

Listings 2, 3 and 4 include code snippets illustrating the use of BetaGPU from the three available programming languages. The use of the software from all of them is very similar, with slight differences due to the syntactic conventions of each language.

Listing 2 Example of use of the library from C++

```
#include <beta_dist.h>
#include <vector>

double alpha = 2.0;
double beta = 5.0;
std::vector<double> in = {0.1, 0.5, 0.9};
std::vector<double> pdf_out(input.size());
std::vector<double> cdf_out(input.size());
std::vector<double> mix_out(input.size());

// Compute Beta PDF using GPU
pdf_out.data() = compute_beta_pdf(alpha, beta, in.data());

// Compute Beta CDF using GPU
cdf_out.data() = compute_beta_cdf(alpha, beta, in.data());

// Compute Beta CDF using CPU–GPU
mix_out.data() = compute_beta_cdf_mixed
(alpha, beta, in.data());
```

Listing 3 Example of use of the library from Python

```
import beta_distribution_package as bdp

# Define parameters
alpha = 2.0
beta = 5.0
data = [0.1, 0.5, 0.9]

# Compute Beta PDF using GPU
pdf_out = bdp.compute_beta_pdf(alpha, beta, data)

# Compute Beta CDF using GPU
cdf_out = bdp.compute_beta_cdf(alpha, beta, data)

# Compute Beta CDF using CPU–GPU
cdf_out_mixed = bdp.compute_beta_cdf_mixed
(alpha, beta, data)
```

Listing 4 Example of use of the library from R

```
library(betaDistributionPackage)

# Define parameters
```

```
alpha <- 2.0
beta <- 5.0
data <- c(0.1, 0.5, 0.9)

# Compute Beta PDF using GPU
pdf_out <- compute_beta_pdf(alpha, beta, data)

# Compute Beta CDF using GPU
cdf_out <- compute_beta_cdf(alpha, beta, data)

# Compute Beta CDF using CPU–GPU
cdf_out_mixed <- compute_beta_cdf_mixed
(alpha, beta, data)
```

3. Illustrative examples

In this section the performance benefits of BetaGPU are highlighted, showcasing the speedups achieved in different scenarios.

3.1. Experimental setup

The experiments were conducted on a heterogeneous system, with two Intel Xeon Ice Lake 8352Y CPUs (32 cores at 2.2 GHz), 256 GB of RAM and different GPU accelerators: a NVIDIA Tesla T4 and a newer NVIDIA A100. Table 1 shows the main specifications of the GPUs used in the experiments. To evaluate the software we used a large number of datasets to cover a wide range of scenarios (see Table 2). The range of the alpha and beta parameters was selected based on the values used in a real bioinformatic application, CancerLocator [2], when using different input data. Also different sizes of input arrays have been used to evaluate the software performance as the dataset size grows. To ensure statistical significance each experiment was repeated seven times and the median of execution times was recorded.

Some preliminary experiments showcased that data communication between CPU and GPU was the main bottleneck in our software, being more than an order of magnitude slower than the computation itself. Hence, all the experiments were conducted using pinned memory to highly mitigate this bottleneck and achieve the best performance. In fact, the use of pinned memory reduced the execution time of the functions to less than a half of the time needed when using paged memory.

The performance of the OpenMP and GPU versions of the Beta distribution functions was evaluated by comparing them to the GSL sequential counterparts.

Table 3 and 4 show the speedup obtained by the PDF and CDF functions, using the median values of alpha (9.34) and beta (11.34). For the CPU versions all the 32 cores of the CPU were used. Besides the pure GPU and CPU functions, Table 4 shows the speedup for the mixed CPU–GPU version described in Section 2.2.

Table 1
Specifications of GPU devices.

Feature	NVIDIA Tesla T4	NVIDIA A100
GPU Architecture	NVIDIA Turing	NVIDIA Ampere
GPU Memory	16 GB GDDR6	40 GB HBM2
Memory Bandwidth	320 GB/s	1555 GB/s
NVIDIA CUDA® Cores	2560	6912
Max Clock rate	1.59 GHz	1.41 GHz

Table 2
Experimental parameters used for benchmarking BetaGPU.

Parameter	Values
Beta distribution Alpha param	[0.1796 , 190.69]
Beta distribution Beta param	[1.2166 , 1695.87]
Input array size	$10^7, 10^8, 10^9$
Input array values	[0,1]

Table 3
Speedup of the parallel implementations available in BetaGPU for the PDF function, using the GSL sequential implementation as baseline for different input sizes.

Input size	Implementation	Speedup A100	Speedup T4
10^7	cudaBetaPDF	255.23x	36.43x
	CPU-based PDF	29.39x	29.39x
10^8	cudaBetaPDF	247.33x	42.59x
	CPU-based PDF	29.66x	29.66x
10^9	cudaBetaPDF	265.98x	42.71x
	CPU-based PDF	30.51x	30.51x

Table 4
Speedup of the parallel implementations available in BetaGPU for the CDF function, using the GSL sequential implementation as baseline for different input sizes.

Input size	Implementation	Speedup A100	Speedup T4
10^7	cudaBetaCDF	266.91x	38.34x
	cudaOmpBetaCDF	53.36x	43.03x
	CPU-based CDF	26.62x	29.62x
10^8	cudaBetaCDF	342.58x	39.19x
	cudaOmpBetaCDF	54.35x	44.79x
	CPU-based CDF	29.31x	29.31x
10^9	cudaBetaCDF	314.25x	44.67x
	cudaOmpBetaCDF	54.06x	50.70x
	CPU-based CDF	29.07x	29.07x

The multithreaded CPU functions show near-ideal speedups, with the performance improving as the dataset size increases. Specifically, a speedup of up to x30.51 was achieved using 32 cores with up to 95% efficiency. In comparison, the GPU results are significantly better, especially when utilizing the newer NVIDIA A100 GPUs. These GPUs demonstrated remarkable performance, achieving speedups of up to x342.58 for the CDF function. This highlights the substantial advantage of the A100 GPUs over traditional CPU-based implementations.

Comparing the results obtained with the T4 and A100 GPUs, the experimental evaluation shows that the software achieves substantially higher speedups with the A100. In fact, we see that for the CDF function on the T4 GPU the performance of the mixed approach surpasses the pure GPU one. This shows that the mixed approach can be attractive for users with state-of-the-art CPUs but older GPUs. In addition, it can also be noticed that the T4 GPU obtains higher speedups that the pure CPU-based approach, showing that not state-of-the-art GPUs still can outperform newer CPUs.

Note that, when using the biggest dataset in the T4, the GPU memory is not enough to fit the whole dataset all at once. In this case, the software uses the chunk-based approach explained in Section 2.2 where the dataset is divided into smaller chunks that fit in the GPU memory, and the computations are performed iteratively on each chunk. This approach allows the software to handle large datasets that exceed the GPU memory capacity, while still achieving significant speedups compared to the sequential implementation.

A variation in the values of alpha and beta does not affect the performance of the PDF function, as the same amount of computations are performed, just with different values. On the other hand, the CDF function is more sensitive to these variations, since the number of iterations needed to converge can vary depending on these values. To prove the significance of the results on different scenarios, the software was also evaluated using extreme alpha and beta values from real executions of CancerLocator. In particular, it was evaluated with the lowest and highest values of alpha (0.1796, 190.69) and beta (1.2166, 1695.87). The results for the largest dataset are shown in Table 5, and prove that speedups are either comparable or significantly higher than those obtained for the median values, reaching up to 764.27x for the A100 GPU and 124.38x for the T4 GPU.

Table 5
Speedup of the parallel implementations available in BetaGPU for the CDF function, using the GSL sequential implementation as baseline for an input size of 10^9 and different values of the shape parameters.

Parameters	Implementation	Speedup A100	Speedup T4
$\alpha = 0.1796$	cudaBetaCDF	764.27x	110.30x
$\beta = 1.2166$	cudaOmpBetaCDF	145.67x	124.38x
	CPU-based CDF	30.70x	30.70x
$\alpha = 190.69$	cudaBetaCDF	485.87x	82.00x
$\beta = 1.2166$	cudaOmpBetaCDF	103.82x	98.54x
	CPU-based CDF	30.94x	30.94x
$\alpha = 0.1796$	cudaBetaCDF	315.07x	94.92x
$\beta = 1695.87$	cudaOmpBetaCDF	107.40x	103.04x
	CPU-based CDF	29.11x	29.11x
$\alpha = 190.69$	cudaBetaCDF	363.72x	77.48x
$\beta = 1695.87$	cudaOmpBetaCDF	98.03x	85.25x
	CPU-based CDF	29.62x	29.62x

4. Impact

The primary contribution of this work is to propose the first CUDA implementation of the Beta distribution functions, taking advantage of the massively parallel capabilities of modern GPUs. From a practical perspective, our Beta distribution CUDA library is not only the most efficient implementation of this statistical method but also integrates seamlessly with popular data analysis environments like R and Python. This integration brings high-performance computing capabilities to these languages, making it accessible for a broader audience, including data scientists and researchers who rely on these platforms for their analytical workflows. The library's capability to efficiently analyze large amounts of data makes it particularly suitable for high-dimensional data analysis and Big Data applications.

This is especially relevant in fields such as genomics, where billions of base pairs have to be processed by applications to discover genetic markers or identify genetic variations among others; finance, where real-time analysis on streaming data is crucial for making rapid informed decisions; and machine learning, where the efficient processing of large datasets is key to obtain relevant information.

Some notable applications of the Beta distribution in bioinformatics include uncovering of detailed population history in human genetics [12], describing the distribution of allele frequencies in populations [13], and the modeling of methylation patterns in several areas, such as cancer detection [1,2], multifactor Whole-Genome Bisulfite Sequencing (WGBS) experiments [14], or estimation of Single Nucleotide Polymorphisms (SNP) genotypes and genetic relatedness [15].

The HPC capabilities of our library can significantly accelerate the analyses in these fields, which can take up to several days for large datasets, enabling researchers to process their data more efficiently and uncover new insights in reasonable time.

Moreover, the library is designed to maintain the accuracy in the computations that the original GSL implementation provides, hence ensuring high mathematical precision in the results.

5. Conclusions

The development of this high-performance software package for computing Beta distribution functions represents a significant advancement in statistical analysis. By leveraging GPU acceleration with CUDA and array programming, the software achieves substantial speedups, ranging from 36x for the PDF function on the smallest dataset on a T4 GPU, to 342x for the CDF function on the A100 GPU for a dataset with a input array size of 10^8. Additionally, the software includes an OpenMP version with an almost ideal performance to use in those computers where a NVIDIA GPU is not available. Furthermore, the software is designed to efficiently manage large datasets, ensuring scalability without compromising performance.

The seamless integration with R and Python ensures accessibility and ease of use, democratizing high performance computing for a broader audience. This accessibility, combined with the open-source nature of the project, fosters continuous improvement and collaboration within the scientific community.

In summary, this software package not only addresses the immediate need for faster statistical computations but also opens up new research possibilities and enhances the pursuit of existing questions. Its impact on various fields underscores its significance as a tool for advancing scientific research and improving data-driven decision-making in both academic and commercial settings.

CRediT authorship contribution statement

Alejandro Fernández-Fraga: Writing – original draft, Validation, Software, Conceptualization. **Jorge González-Domínguez:** Writing – review & editing, Supervision, Conceptualization, Funding acquisition. **María J. Martín:** Writing – review & editing, Supervision, Conceptualization, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by grants TED2021-130599A-I00 and PID2022-136435NB-I00, funded by MCIN/AEI/ (TED2021 also funded by "NextGenerationEU"/PRTR and PID2022 by "ERDF A way of making Europe", EU); grant TSI-100925-2023-1, funded by Ministry for Digital Transformation and Civil Service and Next-GenerationEU/RRF; and FPU predoctoral grant of Alejandro Fernández-Fraga ref. FPU21/03408, funded by the Ministry of Science, Innovation and Universities, Spain. We gratefully thank the Galician Supercomputing Center (CESGA) for the access granted to its supercomputing resources. Funding for open access charge: Universidade da Coruña/CISUG.

References

[1] Li W, Li Q, Kang S, Same M, Zhou Y, Sun C, et al. CancerDetector: ultrasensitive and non-invasive cancer detection at the resolution of individual reads using cell-free DNA methylation sequencing data. Nucleic Acids Res 2018;46(15):e89.
[2] Kang S, Li Q, Chen Q, Zhou Y, Park S, Lee G, et al. CancerLocator: non-invasive cancer diagnosis and tissue-of-origin prediction using methylation profiles of cell-free DNA. Genome Biol 2017;18:1–12.
[3] Sudo M, Osakabe M. freqpcr: Estimation of population allele frequency using qPCR $\Delta\Delta$Cq measures from bulk samples. Mol Ecol Resour 2022;22(4):1380–93.
[4] Lin S, Wang Y, Zhang L, Chu Y, Liu Y, Fang Y, et al. MDF-SA-DDI: predicting drug–drug interaction events based on multi-source drug fusion, multi-source feature fusion and transformer self-attention mechanism. Brief Bioinform 2022;23(1):bbab421.
[5] Wright AM, Lloyd GT, Hillis DM. Modeling character change heterogeneity in phylogenetic analyses of morphology through the use of priors. Syst Biol 2016;65(4):602–11.
[6] Gandy A, Veraart LA. A Bayesian methodology for systemic risk assessment in financial networks. Manage Sci 2017;63(12):4428–46.
[7] Yang S, Shu K, Wang S, Gu R, Wu F, Liu H. Unsupervised fake news detection on social media: A generative approach. In: Proceedings of the AAAI conference on artificial intelligence, vol. 33. 2019, p. 5644–51.
[8] Virtanen P, Gommers R, Oliphant TE, Haberland M, Reddy T, Cournapeau D, et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods 2020;17:261–72. http://dx.doi.org/10.1038/s41592-019-0686-2.
[9] Ahmed N, Mushtaq H, Bertels K, Al-Ars Z. GPU accelerated API for alignment of genomics sequencing data. In: 2017 IEEE international conference on bioinformatics and biomedicine. IEEE; 2017, p. 510–5.
[10] Ahmed N, Lévy J, Ren S, Mushtaq H, Bertels K, Al-Ars Z. GASAL2: a GPU accelerated sequence alignment library for high-throughput NGS data. BMC Bioinformatics 2019;20:1–20.

[11] López-Fernández A, Gómez-Vela FA, Gonzalez-Dominguez J, Bidare-Divakarachari P. bioScience: A new python science library for high-performance computing bioinformatics analytics. SoftwareX 2024;26:101666.
[12] Liu J, Ji X, Chen H. Beta-PSMC: uncovering more detailed population history using beta distribution. BMC Genomics 2022;23(1):785.
[13] Guerrero Montero J, Blythe RA. Self-contained Beta-with-Spikes approximation for inference under a Wright–Fisher model. Genetics 2023;225(2):iyad092. http://dx.doi.org/10.1093/genetics/iyad092.
[14] Dolzhenko E, Smith AD. Using beta-binomial regression for high-precision differential methylation analysis in multifactor whole-genome bisulfite sequencing experiments. BMC Bioinformatics 2014;15:1–8.
[15] Jiang Y, Qu M, Jiang M, Jiang X, Fernandez S, Porter T, et al. MethylGenotyper: Accurate Estimation of SNP Genotypes and Genetic Relatedness from DNA Methylation Data. Genom Proteom Bioinform 2024;22(3):qzae044. http://dx.doi.org/10.1093/gpbjnl/qzae044.