Google Summer of Code

# Google Summer of Code 2024 Proposal:

## Enhancing CuPy coverage of NumPy/SciPy functions

- Shyam Sathvik

# Google Summer Of Code 2024

## ❖Table of Contents

# Google Summer Of Code 2024

---



---

## ❖ Personal Details

| | |
|---|---|
| **Name** | K K N SHYAM SATHVIK |
| **Email ID.** | 1. b22ee036@iitj.ac.in<br>2. zaregototsukai@gmail.com |
| **University** | Indian Institute of Technology, Jodhpur |
| **Year** | Sophomore (2nd Year) |
| **Major** | Electrical Engineering |
| **Github** | https://github.com/boku13 |
| **Linkedin** | https://www.linkedin.com/in/shyamsathvik/ |
| **Resume** | 🗀 Shyam's Resume |

---

Google Summer Of Code 2024

## ❖Abstract


The following write-up aims to propose a plan of action to extend upon the functionality in the **CuPy** library, an open source library for GPU-accelerated computing, while leveraging its existing framework and structure. This write-up aims at proposing a project and intends to cover a sufficiently described method of work, the plan of action, the implementation details, the testing methodology, other software engineering aspects of the APIs added, and finally, my qualifications for the task at hand and the proposed timeline for the project.

## ❖Introduction

I am Shyam Sathvik, a sophomore majoring in Electrical Engineering at Indian Institute of Technology Jodhpur, India. I am currently deeply engaged in exploring the intersection of Computer Science and Artificial Intelligence. Academically, I'm an **Undergraduate Student**

# Google Summer Of Code 2024

---

**Researcher** in the area of **Control Systems** and **Robotics** under the expertise of Dr. Anoop Jain.

I'm also skilled in the domains of **web development, deep learning and competitive programming.**

---

## ❖Programming Experience

Although I did not have prior exposure to programming or computer science in my formal education before my freshman year, I embarked on a **self-learning** journey driven by curiosity and a desire to understand various domains such as machine learning, deep learning, systems design, DevOps and cybersecurity. Courses like 'Introduction to Computer Science' and 'Data Structures and Algorithms' along with participation in club activities have been instrumental in broadening my knowledge base.

I was taught the C programming language and the basics of computer programming in my freshman year at college, and I passed the course at the top of my class with an A grade. After my initial exposure to programming, it has become my biggest hobby.
I'm a part of a handful of technical clubs like **DevlUp Labs** (the Software Development Club), **RAID** (the Artificial Intelligence Club), **Google DSC** (Google tech culture and software promotion club), and the **Robotics Club** of my university.
I have also been a part of various **group projects as well as self projects,** some of which are linked here:

- "*FCFC: File Converter and File Compiler*" with Next.js and node.
- "*FRAIL: Building a Face Recognition System on a Raspberry Pi*" with *Python, PyTorch*.
- "*Self Driving Car with Carla and Convolutional Neural Networks*" with *Python, SQLite and Carla Simulation Software*.

# Google Summer Of Code 2024

---

These projects served as my initial learning, and I've since gone on to build multiple projects in Python as a part of my Data Structures and Algorithms course, including a "Travel Organization Software", "Metro Rail Booking Software", "Freight Car Transport System", and a "PDF Parser". I also built a file conversion and compression website with the Next.js framework as a part of my Engineering Design project. It was selected to be showcased at IIT Jodhpur's Institute Foundation Day out of about 50 other such projects. I also do competitive programming as a hobby and have worked as a front-end developer on my college's Literature Society's official website.

I'm a huge AI nerd and have been two years in a row the institute-wide Kaggle Data Science competition winner in both my fresher and sophomore years.

Furthermore, I have also indulged in Embedded Programming in the first semester of my sophomore year and presented the following work at the Indian Control Conference ‘23 through a project at the Motion Capture Laboratory: https://arxiv.org/abs/2401.17591.

***When I first interviewed for the student researcher position in the lab mentioned above, I only had a vague, basic idea of the technologies involved, such as ROS and Embedded Programming. I barely worked with drones or robots back then. Yet, as soon as I got onboard, I learned the necessary skills in a short month, grew familiar with the completely new environment, and finally implemented our objective of swarm control in a polar curve. My above experience shows excellent evidence of my drive to learn, and through this example, I’d like to show that I can fit into any new space, learn things, and excel in them, especially if I'm interested in them!***

I like to think of myself as someone who does programming for fun; so far I’ve pursued it as a hobby. I love exploring, learning fresh skills (especially related to programming) and gaining expertise over them.

Google Summer Of Code 2024

## ❖Open Source Experience and Motivation behind participating in GSoC

I’m almost a complete beginner when it comes to actually contributing code in open source development, but I’ve always been interested in open source and free software. Hence, I have been looking for programs to participate in to improve my skill set and make the leap into the industry and real-life projects which millions of people use. I'm grateful to have come across a program such as GSoC which lets new contributors to open source do so. I've always admired open-source principles like transparency, collaboration, and teamwork and have advocated using open-source software, as evidenced by my year-long journey as an Arch Linux user.

My enthusiasm for this program stems from my passion for working with technology and my thirst for learning. GSoC also holds a certain amount of value for me, as I’ve always seen it as a significant milestone in a student’s programming career. Hence, I’ve always wanted to be a part of GSoC from a couple of years before I’ve started learned programming.

## ❖ Project Description[1]

### Project Title: Enhancing CuPy coverage of NumPy/SciPy functions

1. **Description :**

[1] Mostly transcribed from https://github.com/cupy/cupy/wiki/GSoC-2024-Project-Ideas



CuPy covers many of the functions that are provided by NumPy/SciPy, but not all of them. I have reviewed the functions that CuPy doesn't cover yet, and propose to implement them in CuPy to use the full speed of the GPU.

I've taken a look at the NumPy/SciPy functions that are not yet implemented in CuPy, and prioritized that list by functions which I believed would help the most and also which interested me the most.

- Comparison Table
- Issues with contribution-welcome label
- Tracker issues for APIs to be implemented
    - NumPy APIs: https://github.com/cupy/cupy/issues/6078
    - SciPy APIs: https://github.com/cupy/cupy/issues/6324
    - Distributions in cupy.random.Generator: https://github.com/cupy/cupy/issues/4557

## 2. Expected Outcomes

- **GPU implementation of NumPy/SciPy APIs.**
    - Routines should be implemented in such a way that they take advantage of the GPU compute model rather than simply copy-pasting NumPy source code.

- **Documentation and tests.**

    - Routines have to be properly documented and with the required tests that cover all the code paths and range of inputs (dtype/shape of the input data, options, etc.)

- **Performance measurement results when necessary.**

- Performance comparison with CPU (NumPy/SciPy) or other GPU implementations might be requested during the review.

**3. Skills Required**

- Python 3
- Experience in NumPy/SciPy API
- Understanding of the CUDA semantics (e.g., use of CUDA Toolkit APIs, parallel programming on GPU, host/device synchronization)

**4. Mentors :**

- @takagi

**5. Expected Size of Project**

- Medium (175 hours)

**6. Difficulty**

- Easy to Medium

---

# ❖ Project

This section aims to discuss the specifics of the project proposal, expanding upon the areas that I intend to work upon, the structure in which I plan to organize the tasks I propose henceforth and the plan of action with which I'd like to approach each task. This proposal is intended by the author to discuss a plan for a medium-sized project, spread over a time period of 175 hours.



---

Appropriate care has been taken to avoid any overpromises in terms of the tasks that I aim to finish during the GSoC period. *Algorithmic thinking takes some time to develop, and is a necessary skill to write optimized code. Reasonable time has been allocated for the same, and deliverables have been laid out while keeping in mind the current skill level of the author (myself).*

I would also like to preface here that reasonable amounts of my ideas and programmatic implementations are derived from the GitHub discussions and blogs of the previous GSoC contributors at CuPy, namely, ***Khushi Agarwal (2022)*** and ***Praveen Sahu (2021).***

---

## ❖Overview:

In this project I propose to work on two parts of CuPy. First, I would like to extend the functionality of CuPy's `cupy.random.Generator` API. The generator provides the functionality for generating random numbers drawn from a variety of probability distributions. I plan to work on adding more distribution functions to the Generator in the same lines of CuPy's 2021 GSOC contributor Praveen Sahu.

Towards the second phase of the GSoC period, I would like to work towards adding more functions to CuPy's `cupyx.scipy.sparse.csgraph` module.

Prior to the GSoC period, I've started contributing to CuPy's codebase to make myself familiar with the codebase. I've made 4 PRs, with 2 of them involving the cuda backend, while the other 2 made changes to the pythonic front end. I've had the most fun working with the CUDA layer, since I could get to learn more about CUDA specific concepts which I'm a beginner at. Furthermore, I understand the benefits and the underlying basic principles of Parallel and GPU Programming due to my

background in AI/ML. But this is the first time I've gotten to work on a real project using the CUDA technology, and I've had a blast doing so. Hence, I've gone and chosen two modules of the codebase, both of which involve working with the CUDA layer directly and optimizing the algorithms at the GPU hardware level.

The first phase of my proposed plan involves working with the Generator module while extending the supported generator distributions, while the second phase involves working with the `csgraph` module. I've studied **Graph Theory** and related algorithms in a rigorous classroom environment at IIT Jodhpur, and I'm pretty confident of my ability to be able to combine that knowledge with my project specific learnings of Parallel Programming and apply it towards optimizing the algorithms for CuPy.

*Note: Even though appropriate care has been taken while picking tasks, on the account of finishing the proposed work too early, I would like to discuss with the mentor to assign myself a set of other tasks which I can work on until the 12 weeks duration.*

---

## ❖Part 1: Extend the functionality of `cupy.random.Generator`: (Week 1-9)

In the first phase of my GSoC tenure, I would like towards closing the issue : https://github.com/cupy/cupy/issues/4557
by adding more probability distributions to the Generator's existing set of distributions.

Praveen has contributed the following distributions to CuPy's Random Generator during his GSoC period:

- ☑ ~~binomial: Add binomial distribution to new Generator #5429~~

# Google Summer Of Code 2024

---

- ☑ ~~chisquare: Add Chi-square distribution to Generator #5645~~
- ☑ ~~dirichlet: Add Dirichlet distribution to Generator #5648~~
- ☑ ~~f: Add F distribution to Generator #5655~~
- ☑ ~~geometric: Add geometric distribution to new Generator #5443~~
- ☑ ~~hypergeometric: Add hypergeometric distribution to new Generator #5560~~
- ☑ ~~logseries: Add Log Series distribution to Generator #5618~~

As a part of my project I would like to implement the following distributions into CuPy:

- ☐ **negative_binomial :** `cupy.random.Generator.negative_binomial()`
- ☐ **noncentral_chisquare :** `cupy.random.Generator.noncentral_chisquare()`
- ☐ **noncentral_f :** `cupy.random.Generator.noncentral_f()`
- ☐ **standard_t :** `cupy.random.Generator.standard_t()`
- ☐ **normal :** `cupy.random.Generator.normal()`
- ☐ **logistic :** `cupy.random.Generator.logistic()`
- ☐ **lognormal :** `cupy.random.Generator.lognormal()`
- ☐ **paretto :** `cupy.random.Generator.paretto()`
- ☐ **weibull :** `cupy.random.Generator.weibull()`

---

## ❖Implementation Details:

The CUDA implementations for the probability distributions are straightforward; they can be modeled after the functions available at numpy/random/src/distributions/distributions.c. For example, model conversion of the `.c` code to `.cu` to add the `standard_t` distribution to the Random Generator can be seen here:



---

Numpy's standard_t at
numpy/numpy/random/src/distributions/distributions.c

```
 double random_standard_t(bitgen_t *bitgen_state, double df) {
  double num, denom;

  num = random_standard_normal(bitgen_state);
  denom = random_standard_gamma(bitgen_state, df / 2);
  return sqrt(df / 2) * num / sqrt(denom);
}
```

Equivalent CUDA code to be added into CuPy:

```
template<typename T>
__device__ double rk_standard_t(T& state, double df) {
     double num, denom;
     num = rk_standard_normal(state);
     denom = rk_standard_gamma(state, df / 2);
     return sqrt(df / 2) * num / sqrt(denom);
}
```

To explain my workflow better, I'll be picking the standard_t distribution function and explain my entire workflow through the complete integration of the API into the new CuPy random generator in the following sections.

---

## Function Profile:

- **API:** `numpy.random.Generator.standard_t()`
- **Distribution:** Standard Student's t distribution.
- **Returns: out** (ndarray or scalar)
    - Drawn samples from the parameterized standard Student's t distribution.
- **Parameters:**
    - **df** (float or array_like of floats) : Degrees of freedom, must be > 0.
    - **size** (int or tuple of ints, optional) : Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `df` is a scalar. Otherwise, `np.array(df).size` samples are drawn.

## ❖ Workflow:

The implementation of `numpy.random.Generator.standard_t()` would roughly follow the following path. This section discusses all the different modifications and insertions that need to be made during the course of the project. To explain my workflow better, I'll explain the changes that I plan to make to CuPy's codebase while adding the distribution:

### Step 1:

- The file **cupy/random/cupy_distributions.cu** needs to be modified to introduce the above discussed implementation, and the corresponding kernel dispatcher.

# Google Summer Of Code 2024

- The first addition would be the Template function, `rk_standard_t()`, designed to generate random variables from a Student's t-distribution given a degree of freedom (`df`).

```
template<typename T>
__device__ double rk_standard_t(T& state, double df) {
    double num, denom;
    num = rk_standard_normal(state);
    denom = rk_standard_gamma(state, df / 2);
    return sqrt(df / 2) * num / sqrt(denom);
}
```

- This code calls `rk_standard_normal()` and `rk_standard_gamma` both of which are currently implemented in **`cupy/random/cupy_distributions.cu`.**
- Next comes the **`standard_t_functor`,** a wrapper struct around the previously defined function enabling the function call to be treated as an object, used further in the kernel call.

```
struct standard_t_functor {
    template<typename... Args>
    __device__ double operator () (Args&&... args) {
        return rk_standard_t(args...);
    }
};
```

- **`void standard_t()`,** kernel launcher function, essentially dispatches the work to the GPU by launching CUDA kernels.

```
void standard_t(int generator, intptr_t state, ssize_t state_size, intptr_t out,
ssize_t size, intptr_t stream, intptr_t df) {
    kernel_launcher<standard_t_functor, double> launcher(state_size,
reinterpret_cast<cudaStream_t>(stream));
    generator_dispatcher(generator, launcher, state, state_size, out, size,
reinterpret_cast<array_data<double>*>(df));
}
```

## Step 2:

- The file **cupy/random/cupy_distributions.cuh** needs to be edited with the following line(s) to expose the CUDA function to the Cython layer.

```
// if block...
void standard_t(int generator, intptr_t state, ssize_t state_size,
intptr_t out, ssize_t size, intptr_t stream, intptr_t df) ;

//else block...
void standard_t(int generator, intptr_t state, ssize_t state_size,
intptr_t out, ssize_t size, intptr_t stream, intptr_t df) {}
```

## Step 3:

- The file **cupy/random/_generator_api.pyx** needs to be edited to introduce the distribution into the Cython layer of CuPy. This function checks the types of the input arguments and calls the kernel dispatcher:

```
def standard_t(self, df, size=None, dtype=numpy.float64):
```

```
    # documentation

    cdef ndarray y
    cdef ndarray df_arr

    if not isinstance(df, ndarray):
        if type(df) in (float, int):
            df_a = ndarray((), numpy.float64)
            df_a.fill(df)
            df = df_a
        else:
            if df is None:
                raise TypeError('df must be real number, not NoneType')
            raise ValueError('df is required to be a cupy.ndarray or a
scalar')
    else:
        df = df.astype('d', copy=False)

    if size is not None and not isinstance(size, tuple):
        size = (size,)
    elif size is None:
        size = df.shape if out is None else out.shape

    y = ndarray(size if size is not None else (), numpy.float64)

    df = cupy.broadcast_to(df, y.shape)
    df_arr = _array_data(df)
    df_ptr = df_arr.data.ptr

    _launch_dist(self.bit_generator, standard_t, y, (df_ptr,))

    return y
```

## Step 4:



- The following files:
  - `tests/cupy_tests/random_tests/common_distributions.py`
  - `tests/cupy_tests/random_tests/test_generator.py`
  - `tests/cupy_tests/random_tests/test_generator_api.py`

  need to be edited to add tests to ensure the correct working of the API by matching the outputs of CuPy to see whether they're consistent with Numpy's outputs:

```
@testing.with_requires('numpy>=1.17.0')
@testing.gpu
@testing.parameterize(
    {'size': None},
    {'size': (1, 2, 3)},
    {'size': 3},
    {'size': (3, 3)},
    {'size': ()},
)
@testing.fix_random()
class TestStandardT(GeneratorTestCase):
    target_method = 'standard_t'

    @testing.for_dtypes('fd')
    @testing.repeat_with_success_at_least(10, 3)
    def test_standard_t_ks(self, dtype):
        self.check_ks(0.05)(df=10, size=self.size, dtype=dtype)

@testing.with_requires('numpy>=1.17.0')
@testing.gpu
@testing.fix_random()
class TestStandardTInvalid(InvalidOutsMixin, GeneratorTestCase):
    target_method = 'standard_t'

    def test_invalid_dtypes(self):
        for dtype in 'bhiqleFD':
            with pytest.raises(TypeError):
                self.generate(df=10, size=(3, 2), dtype=dtype)
```

```
# Specific testing taken from Numpy
class TestStandardTSpecific(GeneratorTestCase):
    def test_standard_t(self):
        random = Generator(MT19937(self.seed))
        actual = random.standard_t(df=10, size=(3, 2))
        desired = np.array([[-1.484666193042647,  0.30597891831161 ],
                            [ 1.056684299648085, -0.407312602088507],
                            [ 0.130704414281157, -2.038053410490321]])
        np.testing.assert_array_almost_equal(actual, desired, decimal=15)
```

The tests are modeled after the tests written for the [cupy.random.Generator.standard_normal](cupy.random.Generator.standard_normal) function and will be adjusted as necessary during the actual implementation.

---

## ❖ Documentation:

Appropriate care will be put into documenting the added APIs appropriately, following the existing Numpy style documentation as per the specification in the sphinxcontrib.napoleon package. Carrying forward the above example, the documentation for CuPy's numpy.random.Generator.standard_t() API would look something like the following.

```
def standard_t(self, df, size=None, dtype=numpy.float64):
      """Student's t-distribution.

      Returns an array of samples drawn from the Student's t-distribution.
      It's probability density function is defined as:

      .. math::
      f(t) = \\\\frac{\\\\Gamma(\\\\frac{\\\\nu +
1}{2})}{\\\\sqrt{\\\\nu\\\\pi}\\\\Gamma(\\\\frac{\\\\nu}{2})} \\\\left(1 +
```

```
\\\\frac{t^2}{\\\\nu} \\\\right)^{-\\\\frac{\\\\nu + 1}{2}}

      where :math:`\\\\nu` is the degrees of freedom.

      Args:
      df (float or array_like of float): The degrees of freedom :math:`\\\\nu`
of the distribution.
      size (int or tuple of ints): The shape of the array. If ``None``, a
zero-dimensional array is generated.
      dtype: Data type specifier. Only :class:`numpy.float64` types are allowed
due to the implementation specifics.

      Returns:
      cupy.ndarray: Samples drawn from the Student's t-distribution.
      """
```

## ❖ Testing:

Appropriate tests will be added leveraging the extensive testing framework available at cupy.testing. This framework provides functionality to make testing easier by providing decorators which run the tests for all dtypes, decorators which compare CuPy's output with its corresponding Numpy output, etc. Here are some of the relevant examples:

- numpy cupy array equal: The decorator to check if their results are equal
- numpy cupy allclose: Decorator to check if their results are close within the specified relative and absolute tolerance
- numpy cupy raises: To check if they throw the same errors.

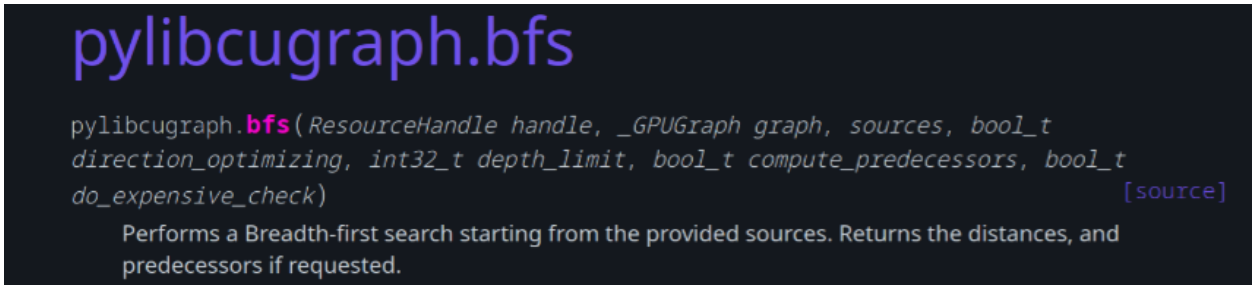
## ❖Part 2: Extend the functionality of (cupyx.scipy.sparse.csgraph): (Week 10-12)

The scipy.sparse.csgraph module in SciPy provides a collection of routines for analyzing the properties of sparse graphs. It can work with graphs represented as sparse matrices, making it suitable for efficiently handling large, complex networks where the adjacency matrix would be mostly zeros. This module leverages the sparse matrix representations from scipy.sparse to perform graph algorithms without the need to convert to a dense matrix form, significantly reducing memory usage and computation time for large graphs.

Towards the end of my GSoC period, I propose to work on the following two APIs,

- cupyx.scipy.sparse.csgraph.breadth_first_order
- cupyx.scipy.sparse.csgraph.shortest_path

implement them into the codebase by leveraging pylibcugraph's existing APIs in the same way that cupyx.scipy.sparse.csgraph.connected_components does:




The workflow should pretty much be the same as above for the main implementation, exposing the API to the topmost layer, the documentation, and the testing, along with the addition of using **Elementwise Kernels.** Some research might be required into the usage of other types of kernels to see if the implementations can further be optimized. I'm confident that I'll be able to finish both the parts of my project's proposal in the speculated time of 12 weeks.

---

## ❖Pre-GSoC Contributions:

This section aims to expand more upon why I believe I'll be able to deliver upon the project ideas that I propose in this particular report. It discusses briefly the **Issues** and **Pull Requests in CuPy** that I've worked upon so far. I've tried to keep the work self sufficient, with only asking the maintainers questions on the design choices and other higher level info about the codebase. Throughout the pre-GSoC period, while exploring the organization, I've had some of the best work related eureka moments while working on CuPy and it was *immensely satisfactory* to see my code merged into the codebase for people to use!

# Google Summer Of Code 2024

## ❖ Pull Requests:

| ID | Pull Request | Status |
|---|---|---|
| #8224 | Add the nearest method for percentile/quantile estimation | Merged |
| #8219 | add cudaStreamCreateWithPriority | Merged |
| #8199 | add cupy.put_along_axis API | Merged |
| #8234 | add scipy.special.stdtr() API | Open[2] |

### 1. #8199:

- This was my first Pull Request at CuPy. The PR added the `cupy.put_along_axis()` API to CuPy. The implementation was taken from numpy's `put_along_axis()` API. The key challenge that I'd overcome during this PR was to get myself familiar with the codebase and understand how to write tests.
- Another small issue was to recognise that the implementation couldn't be directly taken from Numpy since CuPy doesn't support `setitem` for tuple indices yet. Hence, I'd looked through the codebase and found a suitable replacement in the form of `cupy.put()` which exhibits the necessary behavior.

[2] Click here to see my latest list of contributions to CuPy

---

**2. #8219:**

- My second **PR** involved working directly with the CUDA layer, and I'd learnt a lot of things while working on this PR. One of those was the concept of CUDA streams. I'd spent a considerable amount of time reading the CUDA API documentation to understand the things I'd worked on here.
- I'd also learnt about the **HIP** and Stub backends and their roles in the codebase.

**3. #8224:**

- This **PR** was relatively simpler involving finishing a To-Do left behind in the codebase. I added a method for percentile/quantile estimation by using the `cupy.around()` API.

**4. #8234:**

- This **PR** is my first attempt at writing code to support scipy functions. I'd ported over the `scipy.special.stdtr()` by using the `_core.create_ufunc` to support elementwise operations.

---

## ❖Timeline:

The proposed timeline reflects my current understanding of how long each task might take, this might easily change as the coding period begins, and new challenges arise. I would like to keep the timeline flexible by finishing easier tasks as quickly as possible so that I may spend more time on the harder areas, which in this case would be

the considerable efforts that would be required from my side to optimize the implementations as much as possible.

- **Community Bonding Period:**

  During this period I plan to-

  - Get to know more about the mentors, the community and the user base of CuPy.
  - Set up a simple and elegant blog and add the notes I've made while working on CuPy during the Pre-GSoC period.
  - Write down my thoughts and work flow while working on CuPy. I plan to put down all the new things I learn. I hope this might help out future GSoC mentees under CuPy and take some work off the mentors.
  - Set up and learn how to use NVIDIA Nsight Systems as having a visualization tool might help a lot while optimizing algorithms. It seems to have helped out the previous GSoC mentees a lot. It should come handy during Phase 2.
  - Start programming!

- **Week 1-2:**

  - Work on finishing up adding the API **cupy.random.Generator.standard_t**
  - Add tests, add documentation.
  - Make a performance comparison against Numpy.
  - Make a PR.

- **Week 3:**

  - I've condensed the time that should be taken for each API into shorter time periods since I would be used to the workflow after the first two weeks.

- Work on moving the `cupy.random.Generator.negative_binomial` to the new Generator.
- Add tests, add documentation.
- Open a Pull Request with the Performance Comparison against Numpy.

- **Week 4:**
  - Work on moving the `cupy.random.Generator.noncentral_chisquare` to the new Generator.
  - Add tests, add documentation.
  - Open a Pull Request with the Performance Comparison against Numpy.

- **Week 5:**
  - Work on moving the `cupy.random.Generator.noncentral_f` to the new Generator.
  - Add tests, add documentation.
  - Open a Pull Request with the Performance Comparison against Numpy.

- **Week 6:**
  - Work on moving the `cupy.random.Generator.normal` to the new Generator.
  - Add tests, add documentation.
  - Open a Pull Request with the Performance Comparison against Numpy.

- **Phase 1 Evaluation**

# Google Summer Of Code 2024

- **Week 7:**
    - Work on moving the `cupy.random.Generator.logistic` to the new Generator.
    - Add tests, add documentation.
    - Open a Pull Request with the Performance Comparison against Numpy.
- **Week 8:**
    - Work on moving the `cupy.random.Generator.lognormal` to the new Generator.
    - Add tests, add documentation.
    - Open a Pull Request with the Performance Comparison against Numpy.
- **Week 9:**
    - Work on moving the `cupy.random.Generator.paretto` and `cupy.random.Generator.weibull` to the new Generator.
    - Add tests, add documentation.
    - Open a Pull Request with the Performance Comparison against Numpy.
- **Week 10:**
    - Start the work on implementing the functions
        - `cupyx.scipy.sparse.csgraph.breadth_first_order`
        - `cupyx.scipy.sparse.csgraph.shortest_path`
    - Become familiar with the pylibcugraph API.
- **Week 11:**

- Finish the first draft of the implementation.
- Test the implementation with NVIDIA Nsight Systems I and identify areas of improvement and optimization.

- **Week 12:**

  - Finish implementing the functionality with sufficient tests, documentation.
  - Use the last few days to finish the blog and report all the work to the mentor.

- **Final Evaluation**

---

## ❖Time Commitments during the GSoC period:

As someone who's wanted to be a part of GSoC for a couple of years since before I've learnt programming, it holds a special value for me. I can positively dedicate a time of at least 20-30 hours every week towards the GSoC project. Currently I have no other commitments apart from GSoC to dedicate time towards during the Summer apart from upskilling and improving my programming expertise among the other skills to prepare myself to enter real world Software Development. Apart from dedicating adequate amounts of time, I would also like to state that I will attempt to finish my work well in advance so that I may have enough time to **document my work thoroughly** and **tackle any particularly irritating bugs all by myself and emerge as a better developer through the project** :D

---



## ❖Post GSoC period Goals:

- In case of unforeseen circumstances (such as health issues), I will wrap up the work by the final deadline in November.
- I will document my entire workflow and my thoughts, the issues and obstacles I face, and the solutions that I come up with. I promise a deliverable in the form of an extensive blog documenting my journey with CuPy.
- Parallel Programming is a fascinating subject to me, so I can see myself sticking with CuPy helping out the maintainers with contributions regularly even after the GSoC period.
- Open Source is extremely exciting and there are many other projects apart from computational math which are interesting to me. I will take the learnings from my GSoC journey with CuPy to contribute more to other such open source projects too.

## ❖Acknowledgements:


- I would like to thank all the members of CuPy at Github for being really welcoming and supportive during the duration of the Pre-GSoC period.
- I would really like to thank Masayuki Takagi san for corresponding with me over email during the application period and giving me an initial overview of the work that needs to be done.
- I would also really like to thank the other maintainers who've reviewed my code and helped me make it better so that it could be integrated into the codebase.
- Namely, I'd like to thank @leofang, @asi1024, @kmaeshi and @takagi. I'd like to express my gratitude here for helping me improve my coding skills through their direct and indirect mentorship!

## ❖References:


- [CuPy’s Documentation]
- [SciPy’s Documentation]
- [Numpy's Documentation]
- **Okuta, R., Unno, Y., Nishino, D., Hido, S., & Loomis, C. (2017) : [CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations.]**


**THE END**