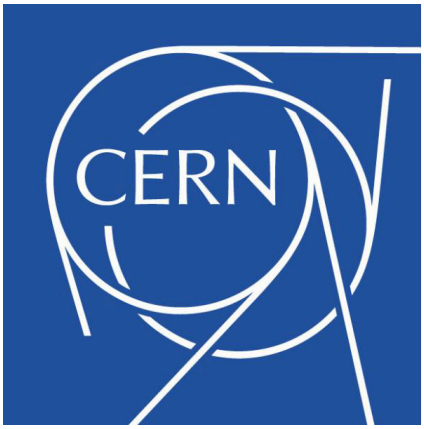
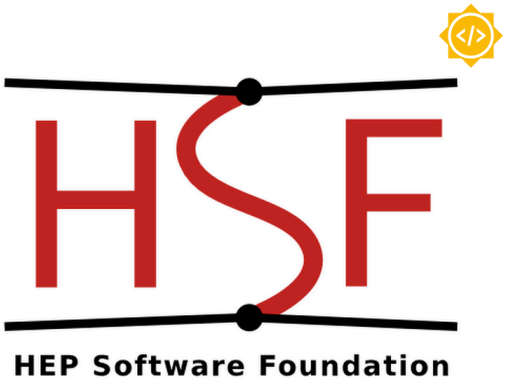
## GSoC 2024 Project Proposal

CERN HSF
High Energy Physics Software Foundation




# STL/Eigen: Automatic conversion and plugins for Python-based ML-backends

## Mentors

- Aaron Jomy
- Wim Lavrijsen
- Vassil Vassilev

## Personal Details

Name: Khushiyant Chauhan
GitHub link: https://github.com/Khushiyant
Email: khushiyant2002@gmail.com

## About Me

I am a professional working as a data scientist for a US-based company called Turing as well as a research assistant under Dr. Swati Aggarwal (Marie Curie Fellow) at NTNU in the field of high-performance clusters and the use of deep learning in brain signal processing. I am an

active contributor to some open-source projects and have made some decent contributions as a result, which are as follows (Top 3):

- Docker Python SDK (Core Contributor)
    - Implemented healthcheck `start_interval` as implemented in Go Engine #3226
    - Fixed none output type in `exec_run` #3201
    - Fixed keyerror when creating new configuration #3200

- Unify (ML Framework Transpilation)
    - Added support for Raw Ops: Log1p #9898
    - Added support for Raw Ops: Rsqrt #9894
    - Added support for make_ndarray : function + test #9731

I have confidence in and experience with ML learning frameworks and their internal functionalities, despite my lack of experience with C++ bindings for Python. My research on various energy-efficient techniques (both published in peer-reviewed journals; see here) also supports my strong grasp of mathematical concepts related to implementation.

In terms of my background in education, I am an undergraduate student in my final year at GGSIPU (State University), majoring in computer science and engineering. I am confident that my technical skills and experience, combined with my passion for problem solving, will make me an asset to this project. I am excited to collaborate with and learn from the mentors and other project contributors.

# Project Details

## Synopsis

Presently, Cpppy's `stl::vector` is accessed by `cpppy.gbl.std.vector` doesn't support arbitrary dimensions, and there is no support for conversion mechanisms between Python built-in types, numpy.ndarray, and STL/Eigen data structures. Cpppy is an automatic, run-time Python-C++ bindings generator for calling C++ from Python and Python from C++. Cppyy uses pythonized wrappers of useful classes from libraries like STL and Eigen that allow the user to utilize them on the Python side.

We can divide the project milestones into two buckets, namely, extension & improvement of `STL/Eigen` types and development & integration of some experimental plugins for JAX using inter-conversion of types

Firstly, we plan to extend support for arbitrary dimensions for `stl::vector` as well as improve the initialization approach for `Eigen` classes as base milestones to achieve. Simultaneously, work on conversion utilities for various types to facilitate the support for experimental plugins,

which will include convolution, multiplication, concatenation, subtraction, and addition (subjected to change after discussion) while using *CUTLASS* for binding to Python API

For example, arbitary dimension support will allow processing types like *<class cppyy.gbl.std.vector<cppyy.gbl.std.vector<double>> at 0x121053940>*

## Implementation Details

### Proposed Approach

***Extend STL support for std::vectors of arbitrary dimensions***

***Note**: All attached code samples are examples of theories to implement and require immense changes*

```
bool Cpppy: IsTemplate(const std::string& template_name)
{
// Existing Code

// Logic to handle std::vetor with arbitrary dimensions
Size_t pos = template_name.find("std::vector<");
If (pos != std::string::npos) {
      Std::string remaining = template_name.substr(pos + 12); //
Extract the remaining string after "std::vector<"
while((pos = remaining.find("std::vector<"))  != std::string::npos) {
 remaining = remaining.substr(pos+12);
    }
  }
return False;
}
```

- Firstly, it would be required to modify template handling to check for nested vector templates with different types, which would require modification in *cppyy-backend/clingwrapper/src/clingwrapper.cxx*
- below code sample checks if a given template name represents a *std::vector* of any dimension with a valid base type while extracting "*std::vector<*" substrings.

- Secondly, we have handle object construct return, which would also require modification in *cppyy-backend/clingwrapper/src/clingwrapper.cxx*
- Below is a code sample that handles returning the required object construct for multi-dimensional vector (code limited to 3 dimensions)

```
Cppyy:: TCppObject_t Cppy:: Construct(TCppType_t type, void* arena) {
// Existing code...
// Can't use recussive approach due to static nature types

if (IsVectorType(type)) {

int dimensions = GetDimensions (type); // GetDimensions has to be
defined
TCpType_t base_type = GetBaseType(type); // GetBaseType has to be
defined
if (dimensions == 1) {
    return new (arena) std:: vector<base_type>;
} else if (dimensions == 2) {
    return new (arena) std:: vector<std:: vector<base_type»>;
} else if (dimensions == 3) {
    return new (arena) std: :vector<std:: vector<std::
vector<base_type>>>;
} else {
    // Handle more dimensions...
  }
}
    // Existing code...
}
```

***Improve the initialization approach for Eigen classes***

- Improving initialization is matter of discussion with mentors to help guide the flow, since may be achieved by creating a usable overloaded wrapper around eigen matrix initialization

***Develop a streamlined interconversion mechanism between Python built-in-types, numpy.ndarray, and STL/Eigen data structures***

- It is required to create a utility suite for conversion between the mentioned types, which will consist of *numpy_to_stl*, which converts a numpy array to an STL vector; *stl_to_numpy*, which converts an STL vector to a numpy array;

*numpy_to_eigen*, which converts a numpy array to an Eigen matrix; and *eigen_to_numpy*, which converts an Eigen matrix to a numpy array.

- utility suite will be accompanied by independent test suite as well as custom *Exception* classes for error handling
- This suite has to rely on PyObject handling and can even be defined as pythonized functions, e.g. __numpy__ functions or decorators

```
#include <vector>
#include <Eigen/Dense>
#include <numpy/ndarrayobject.h>

// Convert a numpy array to an STL vector
std::vector<double> numpy_to_stl(PyArrayObject* arr) {
    int len = PyArray_SIZE(arr);
    std::vector<double> vec(len);
    for (int i = 0; i < len; i++) {
        vec[i] = *(double*)PyArray_GETPTR1(arr, i);
    }
    return vec;
}

// Convert an STL vector to a numpy array
PyArrayObject* stl_to_numpy(const std::vector<double>& vec) {
    npy_intp size = vec.size();
    double* data = size ? const_cast<double*>(&vec[0]) : nullptr;
    return (PyArrayObject*)PyArray_SimpleNewFromData(1, &size,
NPY_DOUBLE, data);
}

// Convert a numpy array to an Eigen matrix
Eigen::MatrixXd numpy_to_eigen(PyArrayObject* arr) {
    int rows = PyArray_DIM(arr, 0);
    int cols = PyArray_DIM(arr, 1);
    Eigen::MatrixXd mat(rows, cols);
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            mat(i, j) = *(double*)PyArray_GETPTR2(arr, i, j);
        }
    }
    return mat;
}

// Convert an Eigen matrix to a numpy array
PyArrayObject* eigen_to_numpy(const Eigen::MatrixXd& mat) {
    int rows = mat.rows();
```

```
    int cols = mat.cols();
    npy_intp dims[2] = {rows, cols};
    PyArrayObject* arr = (PyArrayObject*)PyArray_SimpleNew(2, dims,
NPY_DOUBLE);
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            *(double*)PyArray_GETPTR2(arr, i, j) = mat(i, j);
        }
    }
    return arr;
}
```

***Implement experimental plugins that perform basic computational operations in frameworks like JAX***

- We have to create a *jax_plugins.h* to provide support for JAX framework-based computation while using CUTLASS C++ Helpers defined in *cutlass_binder.h* and *cutlass_binder.cxx* for GEMM operations with help of previously defined *conversion_utils.cxx*
- Below is a code sample illustrate the implementation of *convolve_signals* functionality, which is using CutlassMatMul helper, which is implemented using *CUTLASS GEMM*

**C++ Code:**

```
#include <Eigen/Dense>
#include <vector>

std::vector<double> convolve_signals(const std::vector<double>& a,
const std::vector<double>& b) {
    Eigen::VectorXd a_eigen = Eigen::Map<const
Eigen::VectorXd>(a.data(), a.size());
    Eigen::VectorXd b_eigen = Eigen::Map<const
Eigen::VectorXd>(b.data(), b.size());

    // Perform the convolution using Eigen vectors
    int result_size = a.size() + b.size() - 1;
    Eigen::VectorXd result_eigen(result_size);
    result_eigen = CutlassMatMul(a_eigen, b_eigen);  // Assuming
CutlassMatMul is defined elsewhere

    // Convert the result back to a std::vector
    std::vector<double> result(result_eigen.data(),
```

```
 result_eigen.data() + result_eigen.size());
     return result;
 }
```

**Python Calling Code:**

```
import jax.numpy as jnp
import cppyy

cppyy.include('path/to/cpp/file')
cppyy.load_library('path/to/compiled/library')

def convolve_signals(a, b):
    # Convert the inputs to std::vectors
    a_std = cppyy.gbl.std.vector['double'](a)
    b_std = cppyy.gbl.std.vector['double'](b)

    # Perform the convolution using the C++ function
    result_std = cppyy.gbl.convolve_signals(a_std, b_std)

    # Convert the result back to a numpy array
    result = jnp.array([result_std[i] for i in range(len(result_std))])
    return result
```

***Integrating plugins with toolkits like CUTLASS that utilise the bindings to provide a Python API***

- Since *CUTLASS* provide high-performance GEMM, which could be used by JAX plugins to perform certain operation with help of *conversion_utils.cxx,* this will require a well-thought-out process
- example of basic matrix multiplication can be seen in below code sample

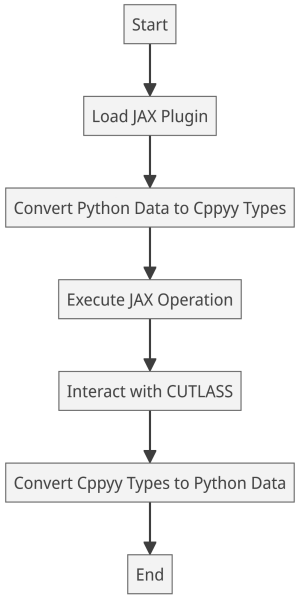
```
// my_cutlass_code.h
#include <cutlass/numeric_types.h>
#include <cutlass/core_io.h>
#include <cutlass/cutlass.h>

std:: vector<std:: vector<int>CutlassMatMul(const
std::vector<std::vector<int>>& a, const
std:: vector<std: :vector<int>>& b) {
// Use CUTLASS to perform matrix multiplication
// This is a simplified example; actual usage of CUTLASS would be more
complex
std:: vector<std:: vector<int>> result = ...; // Compute result using
```

```
CUTLASS
return result;
}
```

- Later, it can be used in developing JAX plugins by including and using its GEMM capabilities:

```
import cpppy
cpppy.include('cutlass_code.h')
```

- This would kindly follow the below depicted code execution



### *Expected Usages after completion*

expected usage

```python
from cppyy.gbl.std import vector

# Current Utilisation
vec1d = vector[int](range(10)) # Generator
vec1d = vector([x for x in range(10)])  # type inferred

# Proposed Utilisation

# Create a 2D vector
vec2d = vector([vector([1, 2, 3]), vector([4, 5, 6]), vector([7, 8, 9])])

# Access elements
print(vec2d[0][0])  # prints 1
print(vec2d[1][1])  # prints 5
print(vec2d[2][2])  # prints 9

# Modify elements
vec2d[0][0] = 10
print(vec2d[0][0])  # prints 10
```

expected usage(jax_plugins)

```python
import cppyy
from cppyy.gbl.std import vector
from cppyy.jax_plugins import CutlassMatMul  # Assuming plugin is named CutlassMatMul

# Create some data
x = vector[vector[int]](2, vector[int](2))  # 2x2 matrix
y = vector[vector[int]](2, vector[int](2))  # 2x2 matrix

# Initialize the matrices
x[0][0], x[0][1], x[1][0], x[1][1] = 1, 2, 3, 4
y[0][0], y[0][1], y[1][0], y[1][1] = 5, 6, 7, 8

# Using JAX plugin
result = CutlassMatMul(x, y)

# The result is a cppyy vector that you can use like any other cppyy vector
print(result)
```

Benefits and deliverables

The community will benefit from the following factors after project deliverables:

- *`stl:vector`* will have robust support for arbitrary dimensions
- Decent improvement in *`Eigen`* class initialisation
- Conversion utilities to handle inter-package types conversion, which is even more useful and beneficial in future plugin development and interaction

- Support of some basic experimental plugins for frameworks that could be useful in future enhancements
- interaction of these plugins with `CUTLASS` will help in Python API binding

### Changes Required

The following changes to the code will be required (assumed):

- Modification in `cppyy-backend/clingwrapper/src/clingwrapper.cxx` to handle templating and object construction
- Addition of `conversion_utils.cxx` to `CPyCppyy` to handle interconversion mechanism between Python buil-tin-types, `numpy.ndarray`, and `STL/Eigen` data structures
- Addition of `jax_plugins.h` to provide basic experimental computation tasks for JAX framework
- addition of `cutlass_binder.h` and `cutlass_binder.cxx` as C++ function helper to provide support for GEMM in JAX plugins

# Timeline of the project

| *Community Bonding Period* | |
|---|---|
| Month 0:<br>01.04.2024-26.04.2024 | Engage with the community. Establish regular meetings with the mentors. Set up the development environment and add Cutlass and Eigen libraries to codebase |
| *coding period begins.* | |
| Week 1<br>27.05.2024-02.06.2024 | Modifications to template handling and object construction to facilitate arbitrary dimensions for vectors<br><br>**Deliverable**: Added logic handling to `cppyy-backend/clingwrapper/src/clingwrapper.cxx` in `bool Cppyy::IsTemplate(const std::string& template_name)` and `Cppyy::TCppObject_t Cppyy::Construct(TCppType_t type, void* arena)` |
| Week 2<br>03.06.2024−10.06.2024 | Brainstorm Eigen Class Initialization Improvement Techniques and final format, e.g. an overload wrapper around Eigen initialization |

| | |
|---|---|
| Week 3 - Week 4<br>11.06.2024-25.06.2024 | Implementation of decided improvement technique for eigen initialization<br><br>**Deliverable**: Overload Wrapper around for Eigen Initialization in `cpppy/python` |
| Week 5 - Week 6<br>26.06.2024-08.07.2024 | Implementation of conversion utilities suite along with unit tests, respectively<br><br>**Deliverable**: Added support for conversion utils between Python builtin-types, `numpy.ndarray`, and STL/Eigen data structures via added conversion_utils.py in cpppy/python |
| Week 7<br>9.07.2024-12.07.2024 | Add unit tests for conversion utils into current stl and eigen test suites<br><br>**Deliverable**: `tests_conversion_utils.py` to cover added utility functions |
| *Midterm Evaluations* | |
| Week 8:<br>12.07.2024−19.07.2024 | Brainstorm the implementation of JAX plugins and CUTLASS GEMM operations |
| Week 9:<br>20.07.2024-27.07.2024 | Implementation of basic GEMM operations using CUTLASS to act as base for JAX plugin helpers<br><br>**Deliverable**: Support for GEMM operations helpers via added `cutlass_code.h` and its source file |
| Week 10 - Week 11:<br>28.07.2024-10.08.2024 | Implementation of JAX plugins, which will include operations like convolution, addition, subtraction, and multiplication, with support for multiple types<br><br>**Deliverable**: Added file `jax_plugins.h` to facilitate support for some basic experimental computations with help of `conversion_utils.cxx` and `cutlass_code.h` |
| Week 12:<br>11.08.2024-18.08.2024 | Buffer Week |
| Week 13:<br>19.08.2024−26.09.2024 | Extended testing, developing documentation, and presenting the work.<br><br>**Deliverables**: test cases, demonstrated reduction of the binary sizes, blog post about the achieved results, presentation at the compiler-research.org team meeting. |

## Why CERN-HSF?

Actually, I have been in research since the very start of college, and CERN, being the pinnacle of research, has always been a place that I want to experience. Consequently, I applied for Summer Student Program but didn't get any response so for me, GSoC is way to atleast work with best of mentors in CERN and learn something from them

## Commitments During Summer

I have been working as data scientist at company called Turing, which has very flexible arrangement so I would be able to commit at least 30-35 hours per week towards this project

## Preferred medium of communication

I'm perfectly fine with any means of communication, including direct mailing lists, and am open to weekly or biweekly meetups to streamline communication. I'm perfectly comfortable using English as medium of communication