Enable CUDA compilation on Cppyy-Numba generated IR

CERN GSoC 2024 Proposal

Mentors :	**Vassil Vassilev**	**<Vassil.Vassilev@cern.ch>**
	Aaron Jomy	**<aaron.jomy@cern.ch>**
	Wim Lavrijsen	**<wlavrijsen@lbl.gov>**
	Jonas Rembser	**<jonas.rembser@cern.ch>**
Applicant :	**Riya Bisht**	<manasi.riya2003@gmail.com>

Contact Information

GitHub	:	chococandy32
Website	:	riyabisht.com
Twitter	:	chococandy63
Address	:	Graphic Era University, Dehradun, India (GMT +5:30)

About Me

My name is Riya Bisht. I am a third-year Computer Science & Engineering student at Graphic Era University, Dehradun, India.
I selected this project because I am working on a research project, Tiles Research[https://github.com/tilesresearch], which is trying to create a smarter and a faster way of working with Compute at the local/client side. We are incorporating emerging technologies like web assembly, webGPU and webTransport. Currently, the project is in its early stages where we are trying to do as many experiments as we can using the existing tools and technologies to build a toy project/demo. In order to work on this dream, I want to gain a deep understanding of the GPGPUs and compiler internals including the topics like automatic parallelization and single source kernels. Contributing to CUDA/LLVM IR will help me to get some hands-on experience with Compiler Tools & Technologies.

Why Me?

I am a fast learner and I have a strong motivation to complete this project. The experience I will gain from the process and outcomes of this program by contributing to compiler internals and open source collaboration will help me in my personal research project. My love for Low-Level systems started when I became a member of the Handmade community that aims to make low-level tech fun for everyone where we share our research [https://handmade.network/p/515/research-compilers/].
Some of my initiatives in Compiler space:
https://github.com/chococandy63/Interpreters-Compilers101
https://riyabisht.com/blog/toycompiler01/
https://nixpienotes.notion.site/Everything-About-WASM-3122aadd248a4d06afd429719c6861a8
https://github.com/chococandy63/hacks-core

Past Open-Source Experience

My open source journey started when I shifted to Linux Operating System full-time. I enjoyed building my linux-kernel from scratch, and tried to do testing for linux distributions like EndeavourOS, and Fedora. I was active in the KDE community where I solved some user-interfacing bugs, contributed to the Unikraft project, got a scholarship by The Linux Foundation in Women in Open Source track and another scholarship to attend KubeCon Chicago 2023 in-person.

Time Commitment

During the GSOC period, I will be working on the project full-time. I am not having any University classes or any part-time jobs/commitments from May-August so I will be able to commit a minimum 7-8 hours/day (minimum 49-52 hours/week), which can be extended if required. I have no major commitment other than GSoC for the following duration so I would be able to focus more on contributing to the project. I have also submitted the proposal for two other projects, one under the Compilers Research Group and another under the Unikraft organization.

Project Information

Title	: Enable CUDA compilation on Cppyy-Numba generated IR
Duration	: 350 hours
Time	: 12 Weeks
Mentor availability	: June-October
Difficulty level	: Medium
Technologies	: Python, C/C++, CUDA, LLVM
Topics	: Machine Learning, Big Data, Algorithmics, Particle Physics, Performance Optimisation

Abstract

Cppyy is an automatic, run-time, Python-C++ binding generator, for calling C++ from Python and Python from C++. Initial support has been added that allows Cppyy to hook into the high-performance Python compiler, Numba which compiles looped code containing C++ objects/methods/functions defined via Cppyy into fast machine code. The project aims to enhance Cppyy, by enabling CUDA compilation on Numba-generated intermediate representation (IR). This integration will allow seamless utilization of CUDA paradigms in Python without compromising performance.

Objectives

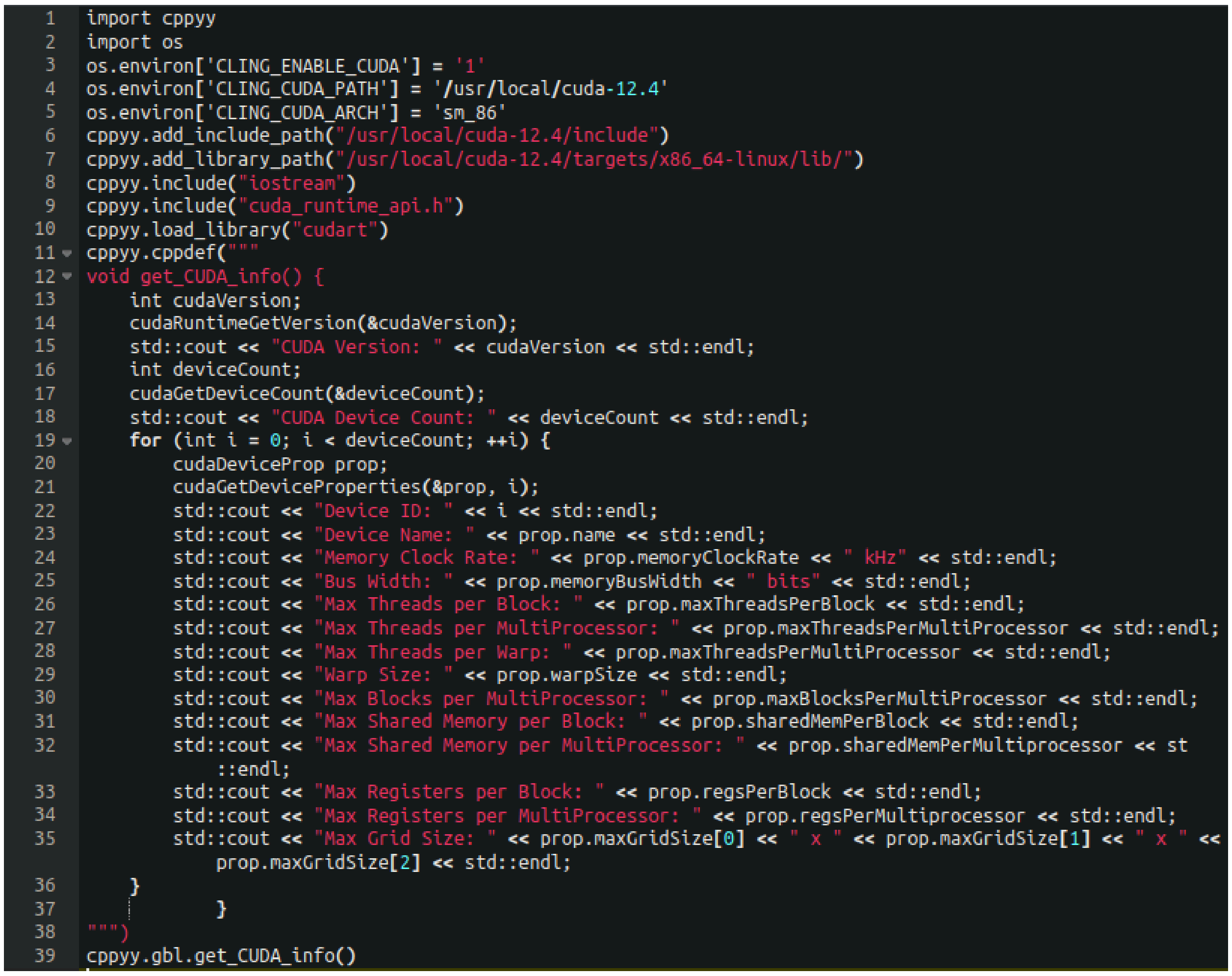
1. **Support for Cppyy-Defined CUDA Code**: Implement support for declaration and parsing of CUDA code defined in Cppyy within the Numba extension.
2. **CUDA Compilation Mechanism**: Design and develop a mechanism for CUDA compilation and execution within the Cppyy-Numba environment.
3. **Testing and Documentation**: Prepare comprehensive tests to ensure functionality and robustness. Create detailed documentation for users and developers.

Pre-GSoC: Evaluation Tasks

My primary OS is Ubuntu 22.04.2 LTS with GNOME and Bash. We created a function get_CUDA_info() that uses the following CUDA APIs:

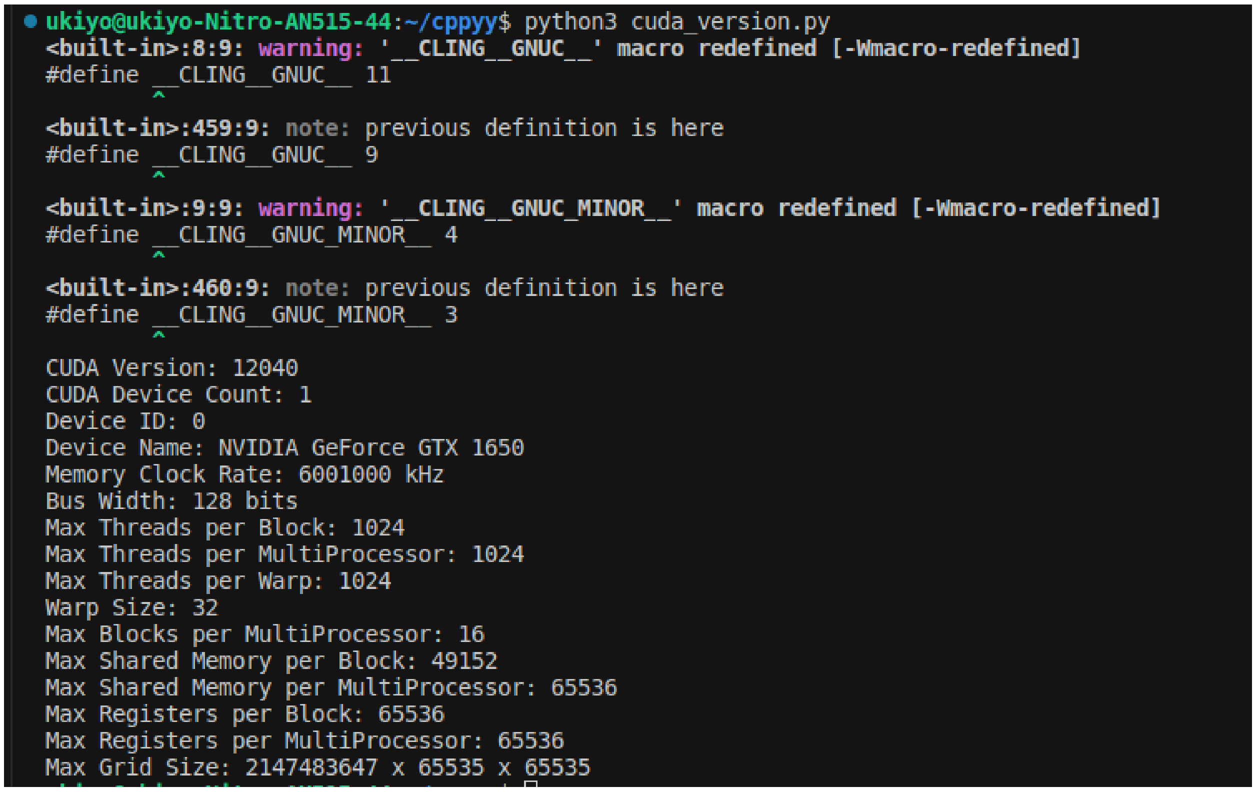
- “cudaRuntimeGetVersion(int* version)” to obtain the version of CUDA
- “cudaDeviceCount(int* count) ” this function returns the number of CUDA-capable devices available.
- “cudaGetDeviceProperties(cudaDeviceProp* prop, int device)” this function returns the properties of a CUDA device to structure cudaDeviceProp pointed by prop for the specified device.
- “cudaDeviceProp” this structure consists of various properties of CUDA devices like device name, memory clock rate, etc.

Code snippets:



```python
import cppyy
import os
os.environ['CLING_ENABLE_CUDA'] = '1'
os.environ['CLING_CUDA_PATH'] = '/usr/local/cuda-12.4'
os.environ['CLING_CUDA_ARCH'] = 'sm_86'
cppyy.add_include_path("/usr/local/cuda-12.4/include")
cppyy.add_library_path("/usr/local/cuda-12.4/targets/x86_64-linux/lib/")
cppyy.include("iostream")
cppyy.include("cuda_runtime_api.h")
cppyy.load_library("cudart")
cppyy.cppdef("""
void get_CUDA_info() {
    int cudaVersion;
    cudaRuntimeGetVersion(&cudaVersion);
    std::cout << "CUDA Version: " << cudaVersion << std::endl;
    int deviceCount;
    cudaGetDeviceCount(&deviceCount);
    std::cout << "CUDA Device Count: " << deviceCount << std::endl;
    for (int i = 0; i < deviceCount; ++i) {
        cudaDeviceProp prop;
        cudaGetDeviceProperties(&prop, i);
        std::cout << "Device ID: " << i << std::endl;
        std::cout << "Device Name: " << prop.name << std::endl;
        std::cout << "Memory Clock Rate: " << prop.memoryClockRate << " kHz" << std::endl;
        std::cout << "Bus Width: " << prop.memoryBusWidth << " bits" << std::endl;
        std::cout << "Max Threads per Block: " << prop.maxThreadsPerBlock << std::endl;
        std::cout << "Max Threads per MultiProcessor: " << prop.maxThreadsPerMultiProcessor << std::endl;
        std::cout << "Max Threads per Warp: " << prop.maxThreadsPerMultiProcessor << std::endl;
        std::cout << "Warp Size: " << prop.warpSize << std::endl;
        std::cout << "Max Blocks per MultiProcessor: " << prop.maxBlocksPerMultiProcessor << std::endl;
        std::cout << "Max Shared Memory per Block: " << prop.sharedMemPerBlock << std::endl;
        std::cout << "Max Shared Memory per MultiProcessor: " << prop.sharedMemPerMultiprocessor << std::endl;
        std::cout << "Max Registers per Block: " << prop.regsPerBlock << std::endl;
        std::cout << "Max Registers per MultiProcessor: " << prop.regsPerMultiprocessor << std::endl;
        std::cout << "Max Grid Size: " << prop.maxGridSize[0] << " x " << prop.maxGridSize[1] << " x " << prop.maxGridSize[2] << std::endl;
    }
        }
""")
cppyy.gbl.get_CUDA_info()
```

Output:



```
● ukiyo@ukiyo-Nitro-AN515-44:~/cppyy$ python3 cuda_version.py
  <built-in>:8:9: warning: '__CLING__GNUC__' macro redefined [-Wmacro-redefined]
  #define __CLING__GNUC__ 11
          ^

  <built-in>:459:9: note: previous definition is here
  #define __CLING__GNUC__ 9
          ^

  <built-in>:9:9: warning: '__CLING__GNUC_MINOR__' macro redefined [-Wmacro-redefined]
  #define __CLING__GNUC_MINOR__ 4
          ^

  <built-in>:460:9: note: previous definition is here
  #define __CLING__GNUC_MINOR__ 3
          ^

  CUDA Version: 12040
  CUDA Device Count: 1
  Device ID: 0
  Device Name: NVIDIA GeForce GTX 1650
  Memory Clock Rate: 6001000 kHz
  Bus Width: 128 bits
  Max Threads per Block: 1024
  Max Threads per MultiProcessor: 1024
  Max Threads per Warp: 1024
  Warp Size: 32
  Max Blocks per MultiProcessor: 16
  Max Shared Memory per Block: 49152
  Max Shared Memory per MultiProcessor: 65536
  Max Registers per Block: 65536
  Max Registers per MultiProcessor: 65536
  Max Grid Size: 2147483647 x 65535 x 65535
```

Reported a build error issue:
https://github.com/wlav/cppyy/issues/223

Bonus Task: Implementation Approach

My understanding of the tasks:
There are three environment variables to control Cling's handling of CUDA:

- `CLING_ENABLE_CUDA` (required): set to `1` to enable the CUDA backend.
- `CLING_CUDA_PATH` (optional): set to the local CUDA installation if not in a standard location.
- `CLING_CUDA_ARCH` (optional): set the architecture to target; default is `sm_35` (Clang9 is limited to `sm_75`)

After enabling CUDA with `CLING_ENABLE_CUDA=1` CUDA code can be used and kernels can be launched from JITed code by in `cppyy.cppdef()`

- Add numba support to cppyy runtime by adding `*import cppyy.numba_ext*` in your code explicitly.
- The `*@numba.njit*` decorator is used to compile the CUDA kernel function for GPU execution, leveraging Numba's GPU capabilities.

Example: To implement both a host and GPU function that adds two vectors.

```
import cppyy
import cppyy.numba_ext
import os
import numpy as np
from numba import cuda
import numpy as np
os.environ['CLING_CUDA_ARCH'] = 'sm_86'
cppyy.add_include_path("/usr/local/cuda-12.4/targets/x86_64-linux/include")
cppyy.add_library_path("/usr/local/cuda-12.4/targets/x86_64-linux/lib/")
cppyy.include("iostream")
cppyy.include("cuda_runtime_api.h")
cppyy.include("device_launch_parameters.h")
cppyy.include("vector")
cppyy.include("cuda_runtime.h")
cppyy.load_library("cuda")
cppyy.load_library("cudart")
cppyy.cppdef("""
// Guest function for vector addition
      namespace NumbaSupportExample {
     template <typename T>
     _global_ void guestVectorAdd(T* A, T* B, T* out, int size)
{
      int i = threadIdx.x;
      if (i < size) {
      out[i] = A[i] + B[i];
}
}
            }
// Host function for vector addition
void hostVectorAdd(int* a, int* b, int* result, int size) {
      for (int i = 0; i < size; ++i) {
      result[i] = a[i] + b[i];
      }
}
""")

@cuda.jit
```

```
def guestVectorAdd(A, B, out, size):
      i = cuda.grid(1)
      if i < size:
      out[i] = A[i] + B[i]

#To copy host->device a numpy array
@numba.njit
def runGuestVectorAdd( ):
ary = np.arange(10)
d_A= cuda.to_device(ary)
d_B = cuda.to_device(ary)
d_out = cuda.device_array_like(d_A)
nthreads = 256
size = len(d_A)
# Define grid and block dimensions
# blockspergrid = (A.size + (threadsperblock - 1)) // threadsperblock
nblocks = (len(d_A) // nthreads) + 1
cppy.gbl.guestVectorAdd[nblocks, nthreads](d_A, d_B, d_out,size)

#To copy device->host
result = d_out.copy_to_host()
print("Result:", result)
```

Project Implementation Approach

Step 1: Identify CUDA-Specific Requirements

Understand the CUDA-specific requirements for compilation to identify the CUDA specific constructs that are not supported by Cppyy-Numba toolchain, such as CUDA kernel definitions, memory management for GPU arrays, and launching CUDA kernels.

Steps to implement the cuda kernel definitions:

1- Define CUDA Kernel Functions:

- Define CUDA kernel functions using the __global__ qualifier in C/C++ to indicate that the function will run on the GPU.
- Implement the CUDA kernel logic for parallel execution on the GPU.

2- Integrate CUDA Kernel Definitions in numba_ext.py:

- Load the necessary CUDA libraries and headers.

- Define the CUDA kernel functions within the numba_ext.py file using Cppyy.

3- Utilize Numba for GPU Execution:

- Import the necessary modules from Numba to leverage GPU execution capabilities.
- Use the *@numba.njit* decorator to compile the CUDA kernel function for GPU execution.

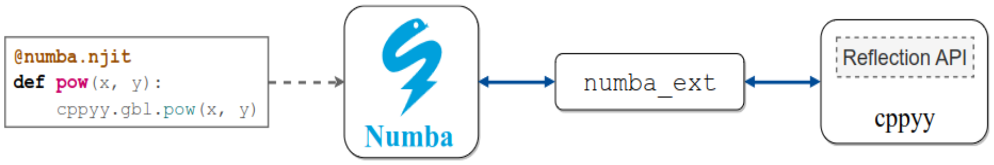


Figure 1. The interaction diagram for Numba, numba_ext and cppyy

Sample Code:

```
import cppyy
from numba import cuda

# Define CUDA kernel function within numba_ext.py using Cppyy
cppyy.cppdef("""
namespace NumbaSupportExample {
    template <typename T>
    _global_ void cudaVectorAdd(T* A, T* B, T* out, int size) {
        int i = threadIdx.x + blockIdx.x * blockDim.x;
        if (i < size) {
            out[i] = A[i] + B[i];
        }
    }
}
""")

# Use Numba for GPU execution and compile the CUDA kernel function
@numba.njit
def cudaVectorAdd(A, B, out, size):
    cudaVectorAdd_cuda[1, 1](A, B, out, size)

# Define CUDA kernel configuration and launch the kernel
@cuda.jit
def cudaVectorAdd_cuda(A, B, out, size):
    i = cuda.grid(1)
    if i < size:
        out[i] = A[i] + B[i]
```

Step 2: Modify the Cppyy-Numba Extension (*numba_ext.py*)
Modify the Cppyy-Numba extension to add CUDA specific data types, function calls, memory management operations.
Sample code: To add CUDA specific data types

```
# Define CUDA-specific data types
cppyy.cppdef("""
namespace CUDATypes {
    struct float3 {
        float x, y, z;
    };

    struct int3 {
        int x, y, z;
    };
}
""")
```

1- Extending the Type Mapping:
The `*_cpp2ir*` dictionary maps C++ types to their equivalent types in LLVM IR. CUDA-specific types that are not currently supported can be added to this dictionary. For example, CUDA has vector types like `*float3*` that are not supported in standard C++ types.

```
# Extend _cpp2ir dictionary to map CUDA data types to LLVM IR equivalents
_cpp2ir.update({
    'CUDATypes::float3': 'llvm::StructType::create({llvm::Type::getFloatTy(context), llvm::Type::getFloatTy(context), llvm::Type::getFloatTy(context)})',
    'CUDATypes::int3': 'llvm::StructType::create({llvm::Type::getInt32Ty(context), llvm::Type::getInt32Ty(context), llvm::Type::getInt32Ty(context)})'
})
```

2- Handling CUDA function calls:
The `*cpp2ir*` function can be extended to add the support for specific CUDA function calls.

```
# Update cpp2ir function to handle CUDA data typesdef cpp2ir(cpp_type):
    if cpp_type.startswith('CUDATypes::'):
        return _cpp2ir[cpp_type]
    else:
# Handle other data typespass
```

Numba uses the proxies to obtain function pointers and generate the necessary LLVM IR code to interact with C++ functions and classes. They handle the conversion of types, accessing data members and methods, and lowering the code to LLVM IR for interoperability. The key proxies used by Numba to obtain function pointers are `CppFunctionNumbaType` and `CppClassNumbaType`. They provide the `get_pointer()` method that retrieves the address of the corresponding C++ function or method. By using these proxies, Numba can seamlessly integrate with C++ code and compile it using LLVM, without relying on Cling for compilation.

Step 3: Modify the `CppFunctionNumbaType` class to handle CUDA kernel functions defined in Cppyy. This may involve:

- Detecting if a function is a CUDA kernel based on attributes or naming conventions. In this example, the `CppFunctionNumbaType` constructor checks if the function has a special attribute `__cuda_kernel__` or if its name starts with `__cuda_kernel` to determine if it is a CUDA kernel.

```
class CppFunctionNumbaType(nb_types.Callable):
    # ...
    def __init__(self, func, is_method=False, is_cuda_kernel=False):
        # ...
        self._is_cuda_kernel = is_cuda_kernel

        # Check if the function has CUDA kernel attributes or naming    conventions
        if hasattr(func, '__cuda_kernel__') or func.__name__.startswith('__cuda_kernel'):
            self._is_cuda_kernel = True
```

- Generating appropriate Numba CUDA kernel signatures and lowering code. In this example, if the function is a CUDA kernel, a special CUDA kernel signature is generated with a `void` return type. The `lower_cuda_kernel` function is registered to handle the lowering of the CUDA kernel. It declares the kernel function and generates the necessary CUDA kernel launch code using `context.launch_kernel`.

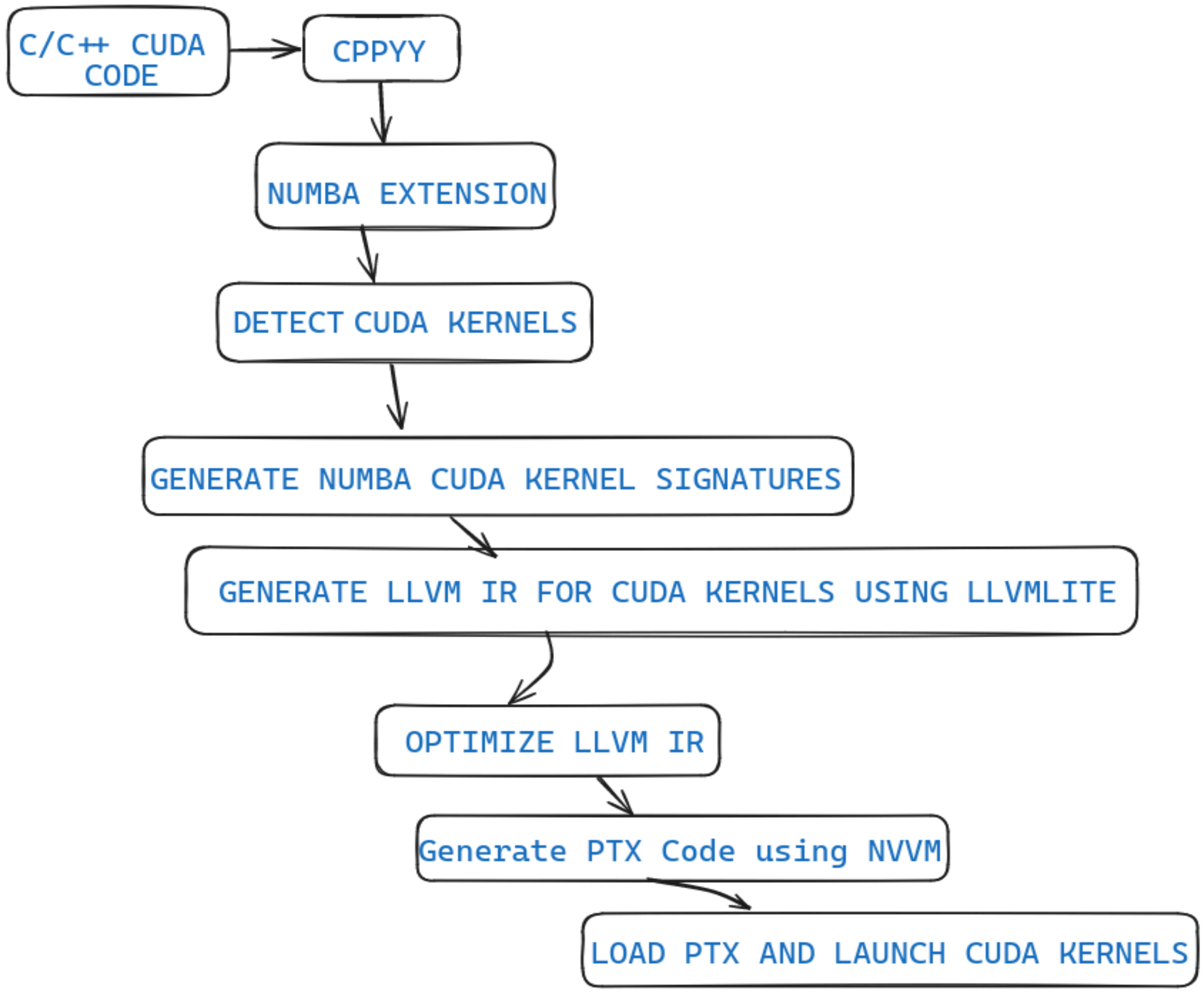
```
class CppFunctionNumbaType(nb_types.Callable):
    # ...

    def get_call_type(self, context, args, kwds):
        # ...

        if self._is_cuda_kernel:
            # Generate CUDA kernel signature
            cuda_sig = nb_typing.Signature(
                return_type=nb_types.void,
                args=args,
                recvr=None)

            # Register CUDA kernel lowering
            @nb_iutils.lower_builtin(ol, *args)
            def lower_cuda_kernel(context, builder, sig, args):
                # Generate CUDA kernel launch code
                kernel_fn = context.declare_function(ol.get_pointer, cuda_sig)
                context.launch_kernel(builder, kernel_fn, args)

            return cuda_sig
```

- Handling CUDA-specific function attributes and launch configurations.In this example, CUDA-specific attributes and launch configurations are extracted from the function, such as `__cuda_grid_dim__`, `__cuda_block_dim__`, and `__cuda_shared_mem__`. These attributes are then used in the `lower_cuda_kernel` function to configure the CUDA kernel launch using `context.launch_kernel`

```
class CppFunctionNumbaType(nb_types.Callable):
    # ...

    def get_call_type(self, context, args, kwds):
        # ...
    if self._is_cuda_kernel:
        # ...

        # Extract CUDA-specific attributes and launch configurations
        grid_dim = getattr(self._func, '__cuda_grid_dim__', (1, 1, 1))
        block_dim = getattr(self._func, '__cuda_block_dim__', (1, 1, 1))
        shared_mem = getattr(self._func, '__cuda_shared_mem__', 0)

        # Register CUDA kernel lowering with launch configurations
        @nb_iutils.lower_builtin(ol, *args)
        def lower_cuda_kernel(context, builder, sig, args):
            # Generate CUDA kernel launch code with launch configurations
            kernel_fn = context.declare_function(ol.get_pointer, cuda_sig)
            context.launch_kernel(builder, kernel_fn, args,
                    grid_dim=grid_dim, block_dim=block_dim,
                                  shared_mem=shared_mem)

        # ...
```

Step 4: Develop comprehensive tests:
Design test cases that cover different use cases and edge cases related to the CppFunctionNumbaType class modifications.

Compilation Pipeline


C/C++ CUDA CODE
CPPYY
NUMBA EXTENSION
DETECT CUDA KERNELS
GENERATE NUMBA CUDA KERNEL SIGNATURES
GENERATE LLVM IR FOR CUDA KERNELS USING LLVMLITE
OPTIMIZE LLVM IR
Generate PTX Code using NVVM
LOAD PTX AND LAUNCH CUDA KERNELS

Numba Extension:

- The Numba extension for Cppyy is responsible for handling the CUDA compilation and execution.
- It extends the existing Cppyy-Numba integration to support CUDA.

Detect CUDA Kernels:

- The Numba extension analyzes the exposed C++ CUDA code to identify CUDA kernel functions.
- It looks for specific attributes or naming conventions to distinguish CUDA kernels from regular functions.

- The `CppFunctionNumbaType` class can be modified to detect CUDA kernels based on these attributes or conventions.

Generate Numba CUDA Kernel Signatures:

- For each detected CUDA kernel, the Numba extension generates a corresponding Numba CUDA kernel signature.
- The signature includes the kernel function name, argument types, and return type.
- The `get_call_type` method of `CppFunctionNumbaType` can be extended to generate CUDA kernel signatures.

Generate LLVM IR for CUDA Kernels using llvmlite:

- The Numba extension generates LLVM IR code for the CUDA kernels using the llvmlite library.
- It translates the C++ CUDA code into equivalent LLVM IR, including device code and kernel launches.
- The `CppFunctionModel` and related classes can be modified to generate appropriate LLVM IR for CUDA kernels using llvmlite.

Output: Enables the support for the following functions:

- Memory Management Functions like `*CudaMalloc*`, `*CudaFree*`, `*CudaMemcpy*`.
- Device Management Functions like `*cudaSetDevice`, `cudaGetDevice*`.
- Kernel Launch Functions like `*cudaConfigureCall`, `cudaSetupArgument`, `cudaLaunch`.*
- Error Handling Functions like `*cudaGetLastError*

Timeline

Community Bonding Period	
May 1 - May 26	- Engage with the community. - Establish regular meetings with the mentors. - Make documentation audits and submit improvements to it where necessary. - Write a blog post announcing my about the community on the compiler-research.org webpage. - Understanding the project requirements, GPU architecture
Coding period begins	
Week 1 27.05.2024-2.06.2024	- Begin extending support for CUDA-specific data types in Cppyy-Numba (numba_ext.py) - Define CUDA data types like float3, int3 etc. in numba_ext.py **Deliverables**: - CUDA-specific data types defined in numba_ext.py
Week 2 3.06.2024- 10.06.2024	- Map CUDA-specific data types to equivalent LLVM IR types - Extend _cpp2ir dictionary for mapping - Modify cpp2ir function to handle translation of CUDA types to LLVM IR **Deliverables:** - CUDA data types mapped to LLVM IR in _cpp2ir - cpp2ir function updated to translate CUDA types
Week 3 11.06.2024-18.06.2024	- Modify CppFunctionNumbaType class to handle CUDA kernel functions - Detect CUDA kernels based on attributes/naming - Generate Numba CUDA kernel signatures for detected kernels - Register lowering to generate LLVM IR for kernels

	Deliverables: - CppFunctionNumbaType updated to support CUDA kernels - CUDA kernel detection and signature generation implemented - Kernel lowering to LLVM IR registered
Week 4 19.06.2024-25.06.2024	- Handle CUDA-specific function attributes and launch configurations in CppFunctionNumbaType - Extract grid dimensions, block dimensions, shared memory from kernel attributes - Use launch configurations in lower_cuda_kernel when generating LLVM IR **Deliverables:** - CUDA function attributes and launch configs handled in CppFunctionNumbaType - lower_cuda_kernel updated to use launch configurations
Week 5 26.06.2024 - 2.07.2024	- Implement memory management functions - cudaMalloc, cudaFree, cudaMemcpy - Add support in Cppyy-Numba extension to call these CUDA memory functions **Deliverables:** CUDA memory management functions supported
Week 6 3.07.2024 - 9.07.2024	- Implement device management functions - cudaSetDevice, cudaGetDevice - Enable switching between GPU devices in Cppyy-Numba **Deliverables:** - CUDA device management functions implemented - Ability to set/get GPU devices added
Week 7 10.07.2024 -16.07.2024	- Implement kernel launch functions - cudaConfigureCall, cudaSetupArgument, cudaLaunch - Generate LLVM IR to launch kernels using these functions **Deliverables:** - LLVM IR generation for launching kernels

	- CUDA kernel launch functions supported
Midterm Evaluations	
Week 8 17.07.2024 -23.07.2024	- Implement error handling functions like cudaGetLastError - Propagate CUDA errors appropriately in Cppyy-Numba **Deliverables:** - CUDA error handling functions added - Proper error propagation mechanism implemented
Week 9 24.07.2024 -30.07.2024	- Develop comprehensive test suite covering different use cases and edge cases - Test CUDA data types, memory management, device management, kernel launches, error handling **Deliverables:** - Comprehensive test suite implemented - All major CUDA functionality covered in tests
Week 10 31.07.2024 - 6.08.2024	- Test end-to-end CUDA compilation and execution by importing numba_ext in a demo program - Validate that CUDA kernels are compiled and run as expected on the GPU **Deliverables**: - End-to-end testing of CUDA compilation and execution - Demo program showcasing Cppyy-Numba CUDA integration
Week 11 07.08.2024 -13.08.2024	**Buffer Week** - Buffer period to handle any overflow tasks - Work on any pending items from previous weeks **Deliverables:** - Completion of any pending tasks - Code cleanup and refactoring

Week 12 14.08.2024 -20.08.2024	- Extended testing - Developing documentation - Presenting the work **Deliverable**: - Test cases - Demonstrated reduction of the binary sizes - Blog post about the achieved results - Presentation at the compiler-research.org team meeting

Post GSoC / Future Work

- Add advanced support for CUDA streams (synchronizing, process management)
- Maintain and update documentation
- Provide support and bug fixes for the CUDA integration

References

CUDA support in Cppyy:
https://cppyy.readthedocs.io/en/latest/cuda.html
Numba support in Cppyy:
https://cppyy.readthedocs.io/en/latest/numba.html
Numba CUDA kernels:
https://numba.readthedocs.io/en/stable/cuda/kernels.html
Extend the Cppyy support in Numba:
https://compiler-research.org/assets/docs/Aaron_Jomy_Proposal_2023.pdf
Numba-Cppyy test file:
https://github.com/wlav/cppyy/blob/master/test/test_numba.py
Numba-CUDA examples:
https://numba.readthedocs.io/en/stable/cuda/examples.html
Efficient and Accurate Automatic Python Bindings with Cppyy Cling:
https://indico.cern.ch/event/1106990/papers/4991292/files/11773-ACAT___Efficient_and_
Accurate_Automatic_Python_Bindings_with_Cppyy___Cling.pdf
High-performance Python-C++ bindings with PyPy and Cling:
https://wlav.web.cern.ch/wlav/Cppyy_LavrijsenDutta_PyHPC16.pdf
Cuda compilation support to Cling: JIT compile to GPUs
https://www.youtube.com/watch?v=XjjZRhiFDVs
https://www.youtube.com/watch?v=HEGDII5lAfo&t=67s