TorchInductor CUDA Backend

# User Code:

import torch

@torch.compile

def matmul\_model(A, B):

return A @ B

The code defines a simple matrix multiplication model using PyTorch’s new torch.compile feature. It is a part of PyTorch’s compiler stack introduced in version 2.0.

In the code above @torch.compile is a decorator that tells PyTorch to compile the function using TorchDynamo.

This improves the performance of PyTorch code by transforming it into a graph and optimizing it under the hood through backend compilers like TorchInductor, TVM, XLA, etc.

result = matmul\_model(A, B)

The above call triggers a compilation and optimization pipeline.

# TorchDynamo:

TorchDynamo intercepts this python function call. Meaning it replaces the orignal matmul\_model function with a wrapper created by TorchDynamo. This wrapper captures the python bytecode being executed inside the function. It tracks all PyTorch operations (A@ B becomes torch.matmul(A,B) internally). And it also builds a computation graph (IR) of the operations.

Eg: compiled\_model = torch\_dynamo\_wrapper(matmul\_model)

**Note: TorchDynamo works at Python level.**

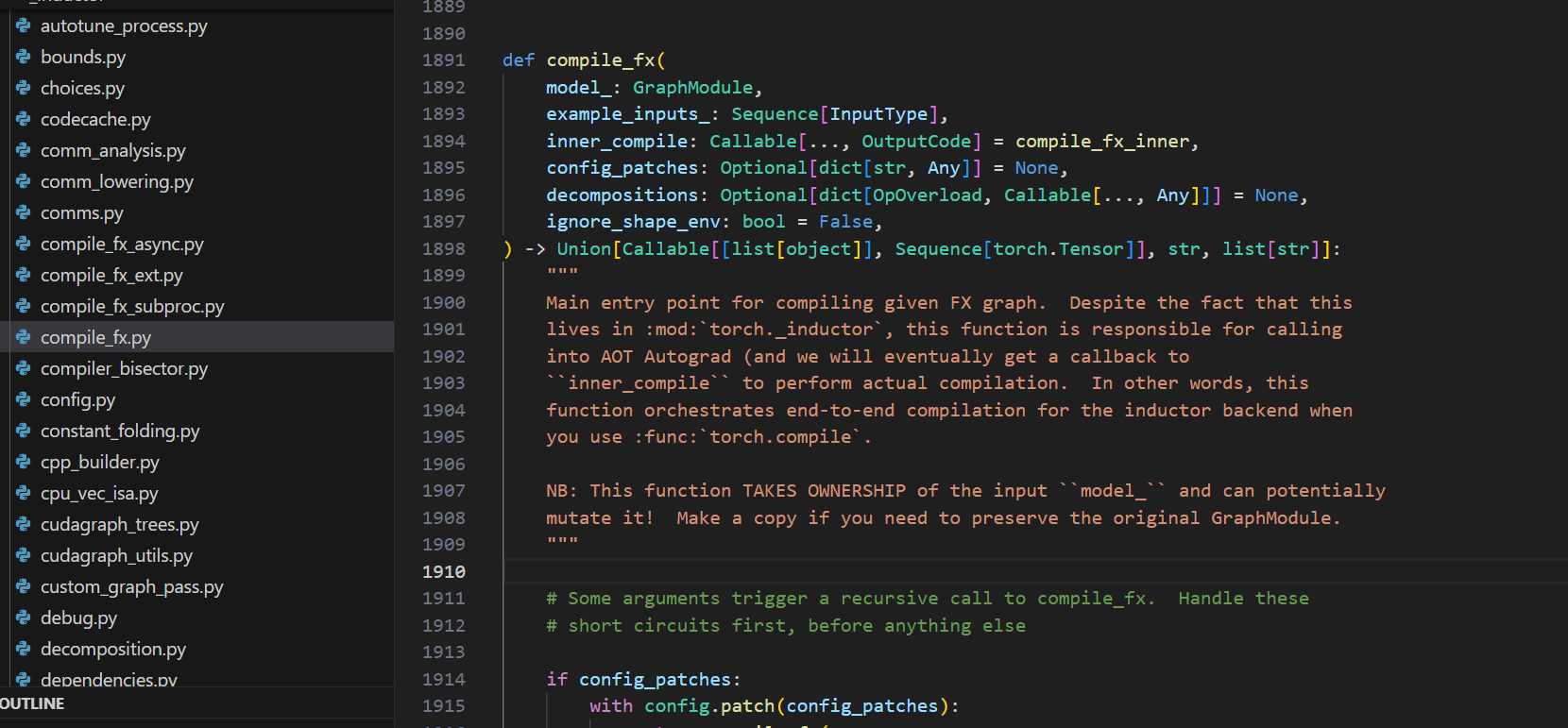
The graph (FX Graph) built by the TorchDynamo is sent to the TorchInductor or another backend for optimization and code generation.

Example of FX graph:

Tensor = aten.mm(%x, %y)

# TorchInductor:

The graph produced by the TorchDynamo is given to the TorchInductor.

**compile\_fx** is the bridge between the TorchDynamo and TorchInductor. TorchDynamo hands the FX graph to the compile\_fx which applies the graph\_level optimizations. 

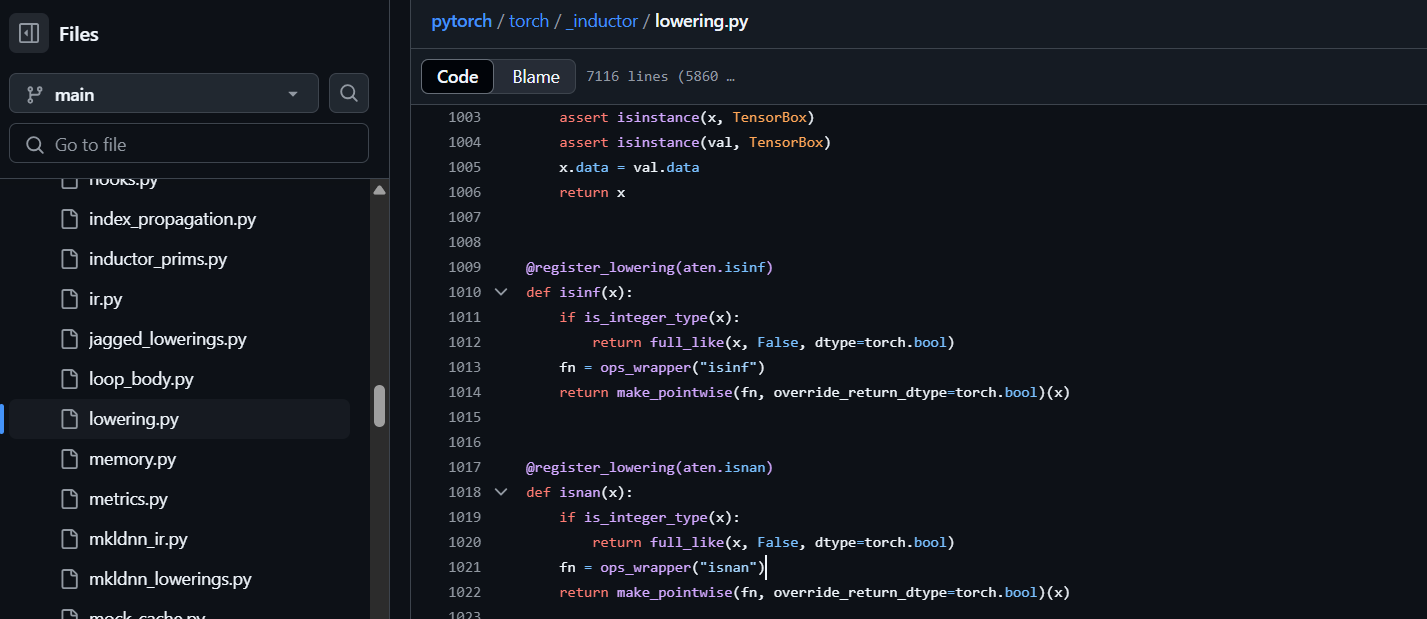
After receiving the FX Graph from the TorchDynamo this function calls either the pattern-matcher or it calls lowering.py to convert the graph into the IR nodes.

There are two pathways to lowering the high level ATen operators like aten.mm/aten.matmul, aten.add, aten.relu, etc. into lower level IR that will eventually turn into CUDA kernels: 1. Dispatcher based lowering and 2. Pattern-matching based lowering

1. Dispatcher based lowering:

This is the default one to one lowering of a single aten operator into an IR node. The operator is decorated with @register\_lowering(aten.op)

The logic is defined in: torch/\_inductor/lowering.py and torch/\_inductor/lowering\_common.py (for CPU ops)



**How lowering works?**

Consider the following function code from the lowering.py file. **@register\_lowering(aten.avg\_pool2d, type\_promotion\_kind=None)**

**def avg\_pool2d( x,**

**kernel\_size,**

**stride=(),**

**padding=0,**

**ceil\_mode=False,**

**count\_include\_pad=True,**

**divisor\_override=None, ):**

**return \_avg\_poolnd( x, kernel\_size, stride, padding, ceil\_mode,**  **count\_include\_pad, divisor\_override, dim=2, )**

The function is decorated with **@register\_lowering(aten.avg\_pool2d, type\_promotion\_kind=None)** decorator. The decorator registers the function **avg\_pool2d()** to lower the **aten.avg\_pool2d** PyTorch operation.

The **avg\_pool2d()** function deligates the lowering to **\_avg\_poolnd()** function. This function is defined in the same lowering.py file. The function is responsible to emit the IR representation for the respective aten operator. In this way, IR representations for different aten operators are emitted/generated.

1. Pattern-matching based lowering:

In Pattern matching based lowering in TorchInductor multiple aten operators are fused into a single efficient CUDA kernel.

Many ML workloads involve sequenses like: output = relu(matmul(x, w) + bias)

Instead of lowering these operations separately TorchInductor matches this pattern and replaces it with fused IR node.

**torch/\_inductor/pattern\_matcher.py** contains all the rules for identifying subgraphs.

The main entry point to the pattern matcher is **register\_replacement().** This function is used to register a new pattern match and its optimized replacement. To this function a search function and replacement function is provided. A search function describes the pattern to look for and the replacement function provides the replacement for that pattern. These functions are traced to create patterns. Along with the search function and register function, **trace\_fn** function is also provided to trace these search and replacement functions.

@register\_replacement(...)

def pattern(a, b, bias):

return torch.relu(torch.add(torch.mm(a, b), bias))

Creating patterns manually is cumbersome and error prone process. Thus patterns are created by providing the search function and replacement function. These functions are traced and converted into a graph.

The pattern would be internally represented for example as:

CallFunction(relu)

↓

CallFunction(add)

↓

CallFunction(mm)

However the pattern could be defined using two ways, either using the register\_replacement() function or using the gen\_register\_replacement() function. When pattern is described using the register\_replacement() function, it puts compile-time overhead during the execution at runtime. Hence to avoid the compile-time overhead gen\_register\_replacement() function is used which pre-compiles the pattern ahead of time. This prevents the compile-time overhead during the runtime. For both the functions the arguments are same, however the gen\_register\_replacement() function takes one additional argument for unique name which is a lookup key.

Internally the pattern matcher represents patterns as a graph (DAG (Directed Acyclic Graph)).

Each operation or node in the DAG (pattern) is a **PatternExpr** (base class for types of patterns) object.

For example, to match this chain: relu(add(mm(A, B), bias))

It will get converted into a DAG made up of three PatternExpr objects as shown in the above graph.

The nodes which are made of PatternExpr object contains a method called **\_match()** that checks weather this part of the pattern match a corresponding node in the real FX graph.

To show how \_match() works:

* CallFunction(relu):

\_match(node) of this node checks if the node is call\_function and node.target==torch.relu.

If True, it moves on to match the input i.e. CallFunction(add)

* CallFunction(add):

\_match(node.args[0]) checks if that input node is torch.add. If True, it continues to check the input of add.

* CallFunction(mm):

\_match(node.args[0].args[0]) checks for mm.

This continues recursively down the entire pattern DAG.

If all the nodes match it returns a **Match** object to the replacement function else it returns a **FailedMatch** object.

**Match:** It is an object that represents a successful match between a pattern and a corresponding sequence of nodes in an FX graph. **Match object stores pattern, FX nodes from the actual graph that matched the pattern and any arguments that are passed to those matched functions like inputs to add, relu, etc.**

After the Match object gets returned the replacement function is invoked. This function takes the matched FX nodes and replaces it with the subgraph of fused FX nodes.

Apart from **register\_replacement()** function, pattern\_matcher.py contains a function named **register\_lowering\_pattern()** which is responsible to register a pattern-matching rule which can be converted directly into the lower IR representations via a provided handler. The function that is decorated using the **register\_lowering\_pattern()** is saved and then called at the lowering time.

Also it contains a **register\_graph\_pattern()** which registers a pattern-matching rule and applies custom transformation on the specific parts of the graph matched using the handler function passed to the function.

**PatternMatcherPass:** PatternMatcherPass class is a key component in the pattern\_matcher.py that encapsulates a set of pattern-matching rules and provides a mechanism to apply these rules on the FX graph. It iterates over each node of the FX graph and applies transformation defined by the associated handlers.

**GraphLowering :**

The class GraphLowering traverse a transformed FX graph and emits a sequence of lower lervel IR nodes. These IR nodes can later be codegen’ed into C++/CUDA code. GraphLowering inherits from the torch.fx.Interpreter, which provides a framework for waling a FX graph.

* GraphLowering Initialization: The GraphLowering is created with the FX graph, backend target (e.g. CUDA) and the list of inputs and outputs. After initialization GtaphLowering.run() is called.
* GraphLowering.run(): GraphLowering.run() begins iterating over the FX nodes. Each FX node gets executed through the GraphLowering.run\_node(node). The run\_node() internally dispatches the nodes target to the lowering function in the lowering.py.

Eg:

@register\_lowering(fused\_mm\_add\_relu)

def fused\_mm\_add\_relu\_lowering(a, b, c):

# Emit a composite IR node, or a sequence of IR ops

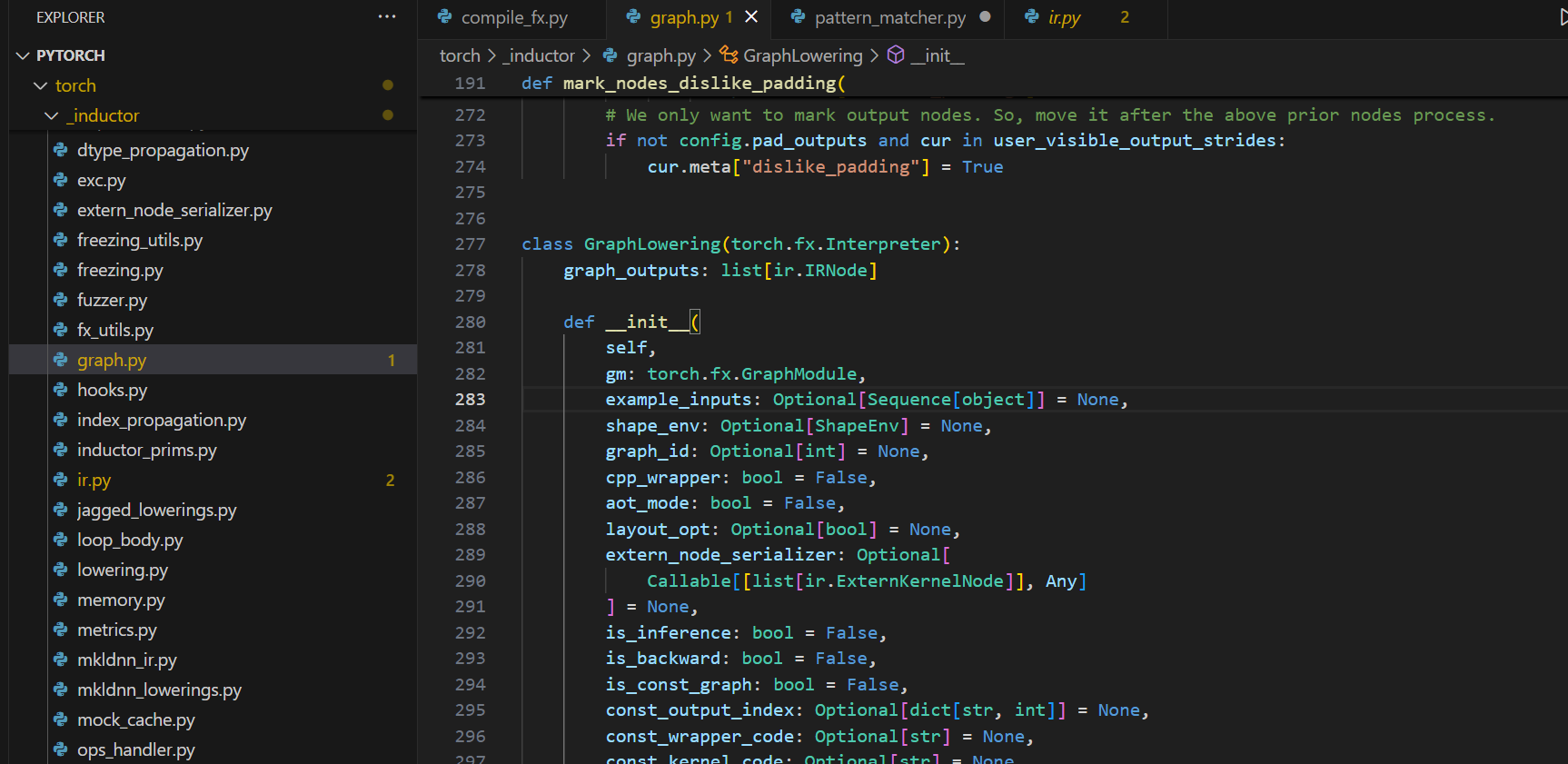
tmp = ir.MatMul(a, b)

tmp2 = ir.Add(tmp, c)

return ir.ReLU(tmp2)

The above function defines how the **high-level operation** fused\_mm\_add\_relu (like torch.nn.functional.relu(a @ b + c)) is lowered into a series of **intermediate representation (IR) nodes**.

This would return the IRNode instances from the **torch/\_inductor/ir.py** like MatMul, Add, ReLU,etc. The returned IRNodes are added to the GraphLowering.graph\_outputs. GraphLowering.graph\_outputs is a list of IR nodes which represents the entire computation.



These IR Nodes are converted to target code (CUDA, etc.) by the codegen modules.

# TorchInductor CUDA Codegen Pipeline

1. cuda\_cpp\_scheduling.py

Once the GraphLowering lowers the FX graph into IR representations, the combined scheduler (CUDACombinedScheduling) drives codegen for CUDA. The combined scheduler dispatches IR nodes either to the Triton-based scheduler or to a CUDA-C++ scheduler. The template IR nodes instances like CUDATemplateBuffer or complex operations (e.g. fused operations, matrix multiplications, etc.) trigger the CUDA-C++ path and the ordinary elementwise or reduction loops go to the Triton (Triton is an open-source language and compiler for writing custom GPU kernels developed by OpenAI) backend.

CUDACombinedScheduling.choose\_node\_backend function checks if a node is a CUDATemplateBuffer (A special type of IR node that holds a CUDA template). And if it is wrapped with the CUDATemplateBuffer it uses cuda\_cpp\_scheduling.py

cuda\_cpp\_scheduling.py decides the organization and fusion of operations into CUDA C++ kernels and helps generate the C++ code for those kernels. It supports generating CUDA C++ kernels using templates. It can also combine multiple operations into a single GPU kernel to run faster.

When the CUDA C++ kernel is ready to be emitted, it renders the source code from the template. It gives it a unique name based on the hash of the code.

It is also responsible to manage kernel execution. Once the kernel is generated, it marks the kernel as ready to run. It calls kernel at runtime with proper arguments.

Triton: Triton is an open source language and a compiler for writing optimized GPU kernels. It is used to generate CUDA kernels from lower IR.

**CUDACPPScheduling class:**

This class handles fusion decisions and CUDA C++ specific template code generation. It inherits from the **BaseScheduling** class and implements the methods from it.

This class consists of two methods- **define\_kernel** and **codegen\_template**.

**define\_kernel** accepts two parameters- **src\_code** and **node\_schedule**. **node\_schedule** is a collection of nodes representing the operations fused into the kernel. The **define\_kernel** defines a function (the kernel) that runs on the GPU to perform computations, such as matrix operations, element-wise calculations, or other parallel tasks. The **src\_code** is typically a template with a placeholder name (e.g., KERNEL\_NAME) for the kernel function. It includes CUDA-specific syntax, like thread indexing, memory access, and parallel execution logic. It is basically the raw, uncompiled CUDA C++ code that needs to be named, registered, and prepared for compilation. **define\_kernel** gives the kernel a unique name, prepares it for compilation, adds metadata for tracking, registers it in the compiler, and returns the name. It ensures the kernel is ready to be compiled and used efficiently in the GPU program.

The **codegen\_template** method confirms the template node is a CUDA C++ template node. It creates the kernel object and render function for **CUDATemplateBuffer**. And generates (or triggers the generation of) the CUDA C++ code using the **CUDATemplateBuffer** class. It names the kernel and prepares the code for compilation and GPU execution using the **define\_kernel** method of the same class.

1. cuda\_template.py

A CUDA kernel template is a C++ pattern (structure) with placeholders. It is used to create custom CUDA kernels. These templates are defined **torch/\_inductor/codegen/cuda/cuda\_template.py.** And these templates are just python classes that generate CUDA C++ code.

* CUDATemplate (Base class):

**CUDATemplate class is the base class for all CUDA template kernels.**

It is responsible for setting up inputs and outputs the CUDA kernel will use. It is also responsible for generating the actual CUDA-C++ code and packaging everything into a callable object that can be compiled later.

For eg. The following subclass of CUDATemplate will be created for the fused operation (matmul, add and relu). And this class is responsible to generate the C++/CUDA kernel that performs the **relu(a@b+c)** PyTorch operation.

**class FusedMMAddReLU(CUDATemplate): def render(self, kernel, \*\*kwargs): inputs = [a, b, c] # These are input Buffers (IR nodes) output = self.output\_node # Result tensor**

**kernel.def\_kernel(inputs, [output], "fused\_mm\_add\_relu")**  
  
 **# Write the CUDA kernel logic**  
 **kernel.writeline("int row = blockIdx.y \* blockDim.y + threadIdx.y;")**  
 **kernel.writeline("int col = blockIdx.x \* blockDim.x + threadIdx.x;")**  
 **kernel.writeline("if (row < M && col < N) {")**  
 **kernel.writeline(" float sum = 0.0f;")**  
 **kernel.writeline(" for (int k = 0; k < K; ++k)")**  
 **kernel.writeline(" sum += A[row \* K + k] \* B[k \* N + col];")**  
 **kernel.writeline(" sum += C[row \* N + col];")**  
 **kernel.writeline(" out[row \* N + col] = max(sum, 0.0f);") # ReLU**  
 **kernel.writeline("}")**

The above code is just an example, it is not the real code that gets generate

In file: common.py

KernelTemplate:

Indent\_except\_first: handles kernel code indentation

\_template\_from\_string(source: str): \_template\_from\_string(source: str): Parses a string into a Jinja2 Template object, applying a custom indent filter. Returns the Template object if parsing succeeds, None if the Jinja2 environment is unavailable, or raises a DetailedTemplateSyntaxError with detailed context if the source has Jinja2 syntax errors.

\_fake\_get\_dtype: Creates a dictionary (lookup) mapping buffer names to their data types (torch.dtype), based on fake\_outs. Returns a new function (get\_dtype) that looks up the data type of a buffer by its name, using the lookup dictionary first and falling back to the original V.graph.get\_dtype method if the name isn’t found.

1. cuda\_kernel.py:

The responsibility of the cuda\_kernel.py is to build a CUDA kernel code structure. It creates an object that holds the kernel name, argument definations, prologue, body and epilogue code snippets, etc. It defines kernel arguments.

**It is also responsible to accept the kernel logic from the templates. Kernel templates call methods on the object created above to write/collect code lines (writeline(....)), define loops and thread logic, etc. To put it simply the cuda\_template.py defines the logic and the cuda\_kernel.py helps build the kernel line by line. cuda\_kernel.py only provides the utility functions that are used by the templates to build the final kernel structure.**

The scheduler call the render() on the template. Then it fills the object created above with the actual source code lines.

Lets suppose the generated CUDA kernel is as shown below-

**extern "C" global void fused\_mm\_add\_relu(const float\* A, const float\* B, const float\* C, float\* Out, int M, int N, int K) {**

**int row = blockIdx.y \* blockDim.y + threadIdx.y;**

**int col = blockIdx.x \* blockDim.x + threadIdx.x;**

**if (row < M && col < N) {  
 float sum = 0.0f;  
 for (int k = 0; k < K; ++k) {  
 sum += A[row \* K + k] \* B[k \* N + col];  
 }  
 sum += C[row \* N + col];  
 Out[row \* N + col] = max(sum, 0.0f); // ReLU  
 }**

**}**

TorchInductor uses the NVRTC (NVIDIA Runtime Compilation)/NVCC to compile the code string above into GPU-executable PTX (Parallel Thread Execution) binary.

Further the PTX is loaded via the CUDA Driver API and the kernel is launched.

1. cuda\_evt.py

cuda\_evt.py consists of the following three functions: get\_cuda\_arch(), get\_cuda\_version() and nvcc\_exist().

The get\_cuda\_arch() basically retrieves the cuda architecture of the target GPU. It helps to determine which GPU features and instructions are available, guiding PyTorch Inductor’s code generation to produce optimized CUDA kernels compatible with the target hardware.

The get\_cuda\_version() retrives the cuda version used by the system. The CUDA version is essential for PyTorch Inductor to ensure that generated CUDA kernels are compatible with the installed CUDA toolkit and drivers. It influences compilation flags (e.g., for nvcc, checked by nvcc\_exist) and kernel features (e.g., support for specific CUDA APIs or GPU instructions).

nvcc\_exist() function is a utility function used to check if the nvcc compiler is accessible on the system to compile the generated cuda kernel.