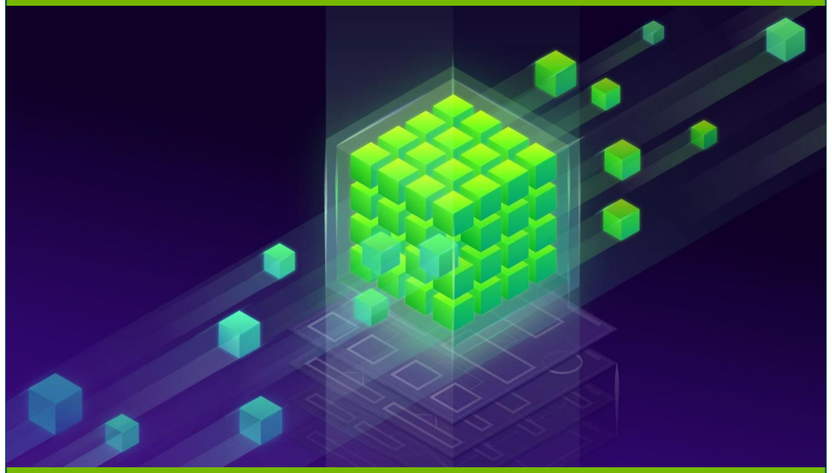
High-Performance CUDA Ops in Python: JIT-Compiling CCCL with cuda.compute



https://github.com/NVIDIA/cccl/

CUDA Core Compute Libraries for Python

What is it?

`cuda.compute` is a Python package that brings composable, high-performance CUDA algorithms to arrays, tensors, and data ranges (iterators). It fills the missing middle layer: optimized GPU kernels that act as reusable building blocks, making it easy to write efficient algorithms—without needing deep CUDA expertise.

Who is it for?

Python developers who want to write custom algorithms portable across GPU architectures without dropping down to CUDA C++.

CCCL Tops GPU MODE Leaderboard

Simple Python implementations based on `cuda.compute` top the GPU Mode leaderboard. `cuda.compute` achieves performance on par with CUDA C++, i.e., state-of-the-art performance across GPU architectures.

GPU MODE

	H100	A100	L4	T4
grayscale	10th	N/A	N/A	N/A
histogram	1st	1st	1st	4th
prefixsum	1st	1st	1st	1st
sort	1st	1st	1st	1st
vectoradd	1st	1st	1st	1st
vectorsum	2nd	2nd	4th	2nd

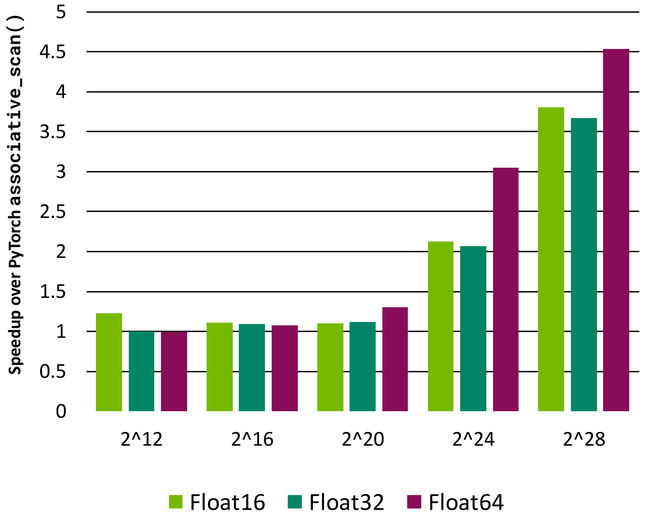
Build PyTorch ops in pure Python

Faster than `torch.compile()`

PyTorch recently introduced the higher order operator `associative_scan` that computes a parallel prefix scan over an input tensor with a user-provided associative operator.

We implemented `associative_scan` entirely in Python with no C++ code using cuda.compute's `scan`. We compared our implementation's performance to the existing PyTorch implementation compiled with `torch.compile()`. Benchmarks showed that our implementation achieves up to a 5x speedup.

`cuda.compute` offers speed-of-light performance and flexibility through Python defined operators and custom datatypes, making it ideal for developing PyTorch operators.



Speed of Light Algorithms, no manually written CUDA kernels

```python
import cuda.compute
import torch

# Scan algorithm showcasing iterators, custom operators, and kernel fusion

# Create a counting iterator to represent the sequence 1, 2, ... N
counts = cuda.compute.CountingIterator(1)

def square(x):
    return x * x

# Create a transform iterator to represent the sequence 1, 4, 9, ... N^2
input = cuda.compute.TransformIterator(counts, square)

# Create an output array to store the result
output = torch.empty(2**20, dtype=torch.int64).cuda()

def add(a, b):
    return a + b

# Perform the scan
cuda.compute.inclusive_scan(input, output, add, init=0, num_items=output.size())

# Merge sort algorithm showcasing a custom datatype to be used as input to sort
@cuda.compute.gpu_struct
class Pair:
    a: torch.int16
    b: torch.float64

# Prepare sample input
a_keys = torch.tensor([3, -1, 3, 0, -1, 2, 0], dtype=np.int16).cuda()
b_keys = torch.tensor([1.5, 2.3, 0.5, 9.5, -7.0, 4, -1.0], dtype=torch.float64).cuda()
keys = cuda.compute.ZipIterator(a_keys, b_keys)

# Struct-aware comparator
def compare_op(lhs: Pair, rhs: Pair):
    return lhs.b < rhs.b if lhs.a == rhs.a else lhs.a < rhs.a

# Perform the merge sort in-place
cuda.compute.merge_sort(keys, None, keys, None, compare_op, num_items=keys.size())
```

Explicit Fusion: Since iterators are "lazy", all computations are fused into a single kernel call here.

Nader Al Awar (NVIDIA), Ashwin Srinath (NVIDIA)

NVIDIA