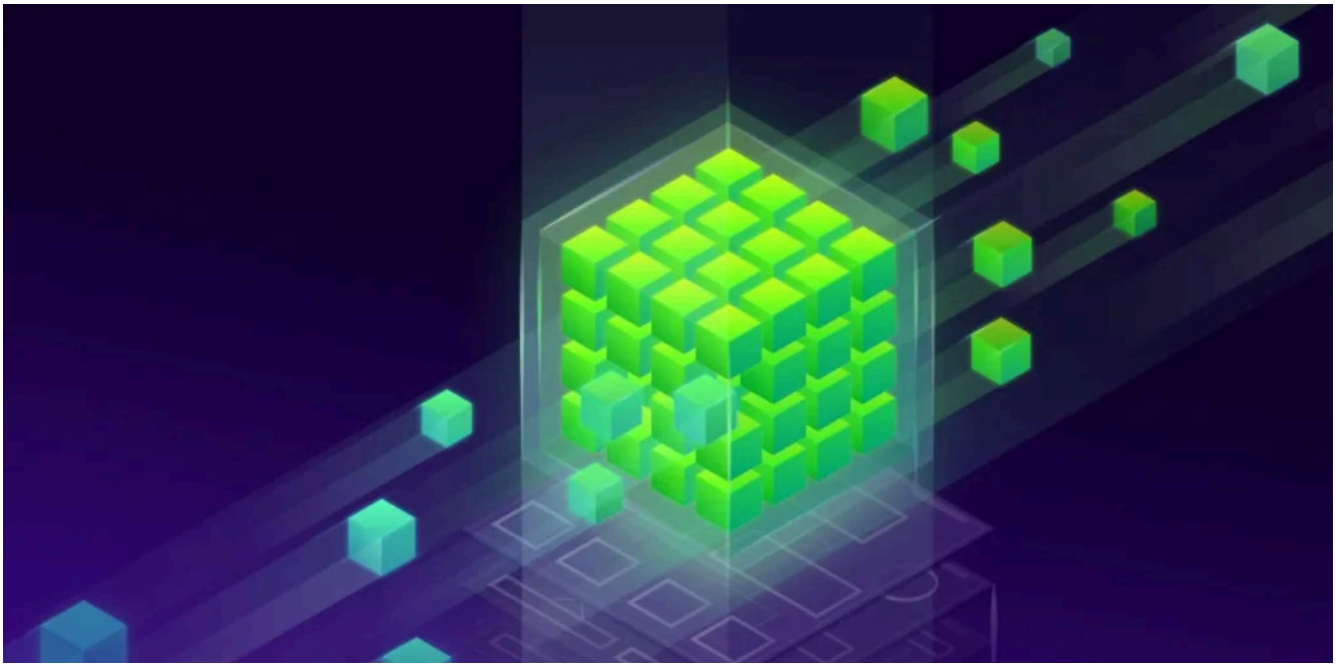
Delivering the Missing Building Blocks for NVIDIA CUDA Kernel Fusion in Python

Jul 09, 2025

By Ashwin Srinath and Andy Terrel



C++ libraries like CUB and Thrust provide high-level building blocks that enable NVIDIA CUDA application and library developers to write speed-of-light code that is portable across architectures. Many widely used projects, such as PyTorch, TensorFlow, XGBoost, and RAPIDS, use these abstractions to implement core functionality.

The same abstractions are missing in Python. There are high-level array and tensor libraries such as CuPy and PyTorch, and low-level kernel authoring tools like numba.cuda. However, the lack of

Introducing `cuda.cccl`

`cuda.cccl` provides Pythonic interfaces to the CUDA Core Compute Libraries CUB and Thrust. Instead of using C++ or writing complex CUDA kernels from scratch, you can now compose algorithms that deliver the best performance across different GPU architectures.

`cuda.cccl` is composed of two libraries:

- `parallel` provides composable algorithms that act on entire arrays, tensors, or data ranges (iterators).
- `cooperative` enables you to write fast, flexible `numba.cuda` kernels by providing algorithms that act on blocks or warps.

This post introduces the `parallel` library.

A simple example: custom reduction

To show what cuda.cccl can do, here's a toy example that combines pieces of functionality from `parallel` to compute the sum 1 – 2 + 3 – 4 + … N.

See the full code example.

```
# define some simple Python functions that we'll use later
def add(x, y): return x + y


def transform(x):
    return -x if x % 2 == 0 else x

# create a counting iterator to represent the sequence 1, 2, 3, ... N
counts = parallel.CountingIterator(np.int32(1))

# create a transform iterator to represent the sequence 1, -2, 3, ... N
seq = parallel.TransformIterator(counts, transform)

# create a reducer object for computing the sum of the sequence
out = cp.empty(1, cp.int32)  # holds the result
reducer = parallel.reduce_into(seq, out, add, initial_value)

# compute the amount of temporary storage needed for the
# reduction, and allocate a tensor of that size
tmp_storage_size = reducer(None, seq, out, size, initial_value)
tmp_storage = cp.empty(temp_storage_size, cp.uint8)


# compute the sum, passing in the required temporary storage
```

Is it fast?

Let's time the algorithm we just built using `parallel` alongside a naive implementation that uses CuPy's array operations. These timings were done on an NVIDIA RTX 6000 Ada Generation. See the comprehensive benchmarking script.

Here are the timings using array operations:

```
seq = cp.arange(1, 10_000_000)

%timeit cp.cuda.runtime.deviceSynchronize(); (seq * (-1) ** (seq + 1)).sum(); cp.cuda.runti
690 µs ± 266 ns per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Here are the timings using the algorithm we built using `parallel`:

```
seq = TransformIterator(CountingIterator(np.int32(1)), transform_op)

def parallel_reduction(size):
     temp_storage_size = reducer(None, seq, out_tensor, size, initial_value)
     temp_storage = cp.empty(1, dtype=cp.uint8)
     reducer(temp_storage, seq, out_tensor, size, initial_value)
     return out_tensor

%timeit cp.cuda.runtime.deviceSynchronize(); parallel_reduction(); cp.cuda.runtime.deviceSy
203 µs ± 703 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Related posts

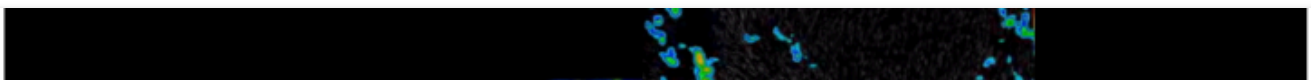
We see that our approach, combining `parallel` iterators and algorithms, is faster than a naive approach.

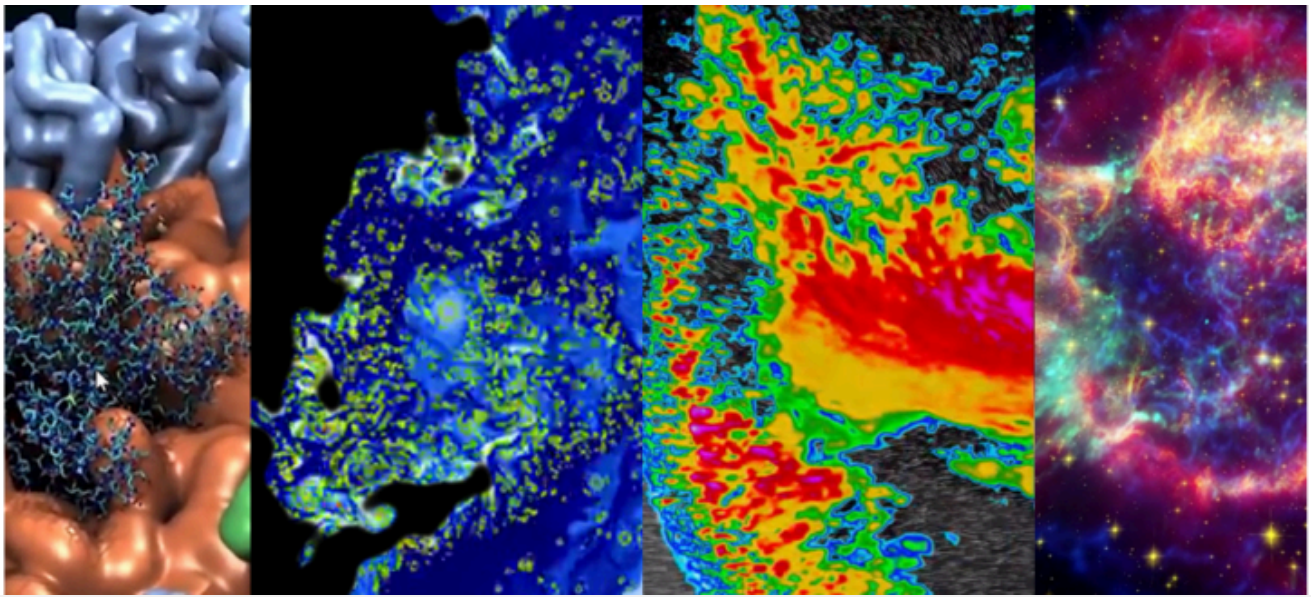
Where does the speedup come from?

Many core CUDA operations in CuPy use CUB and Thrust—the same C++ libraries that `parallel` exposes to Python. Why do we see better performance using `parallel`?

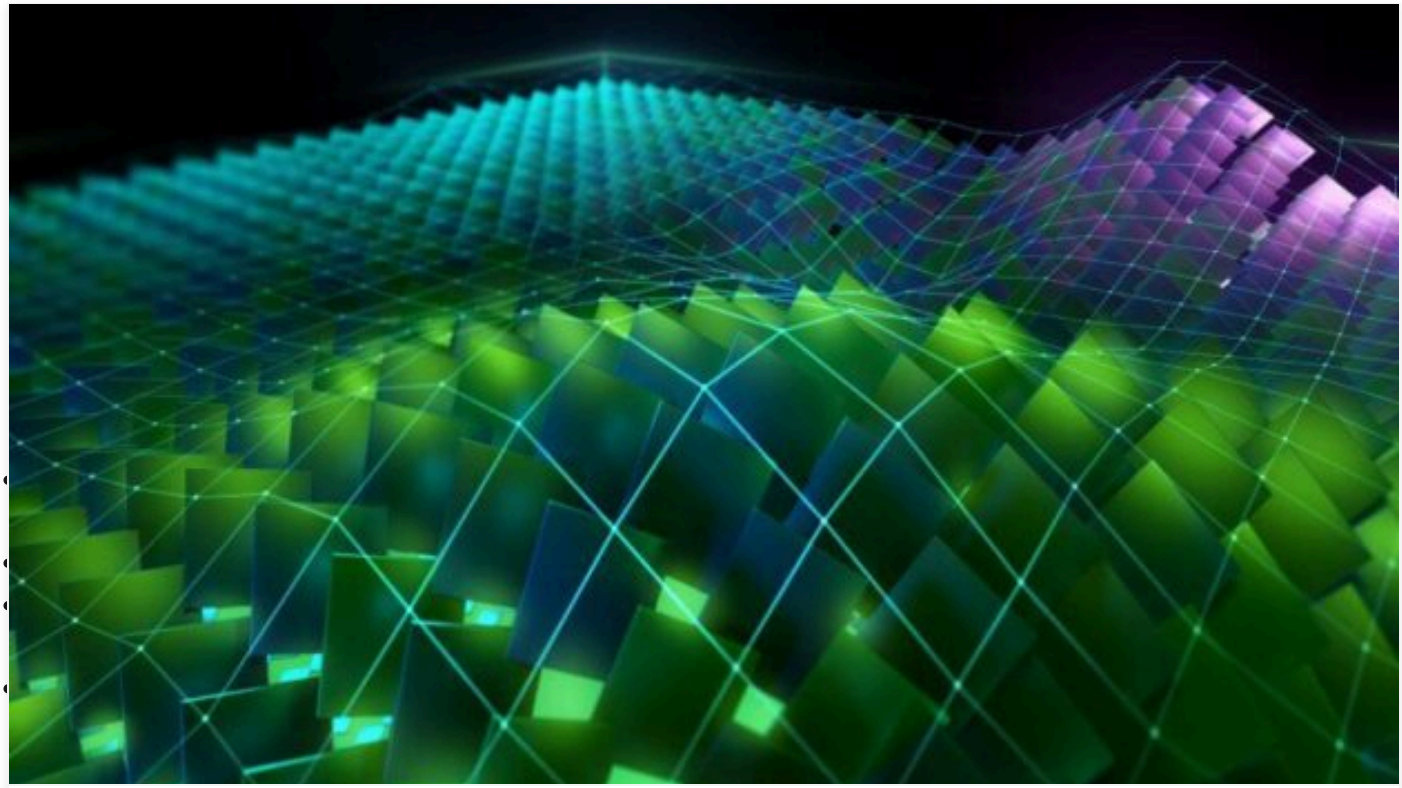
There's no magic here. `parallel` enables more control and flexibility for you to write general-purpose algorithms. In particular:

- **Less memory allocation:** A major advantage of `parallel` is the ability to use iterators like `CountingIterator` and `TransformIterator` as inputs to algorithms like `reduce_into`. Iterators can represent sequences without allocating memory for them.
- **Explicit kernel fusion:** Using iterators in this way "fuses" all the work into a single kernel—the naive





Developing Accelerated Code with Standard Language Parallelism



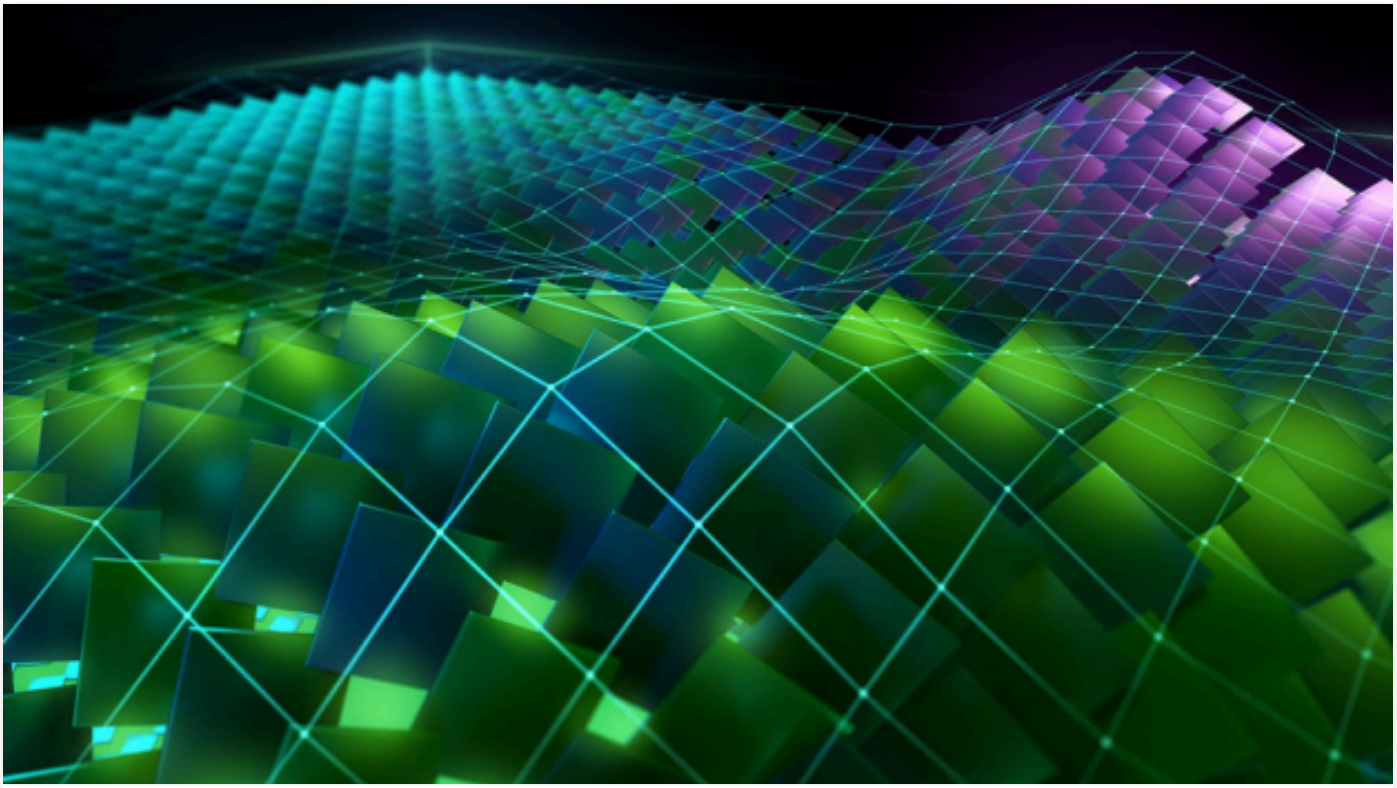
Revealing New Features in the CUDA 11.5 Toolkit

building blocks used internally by many libraries like CuPy and PyTorch.

Next steps

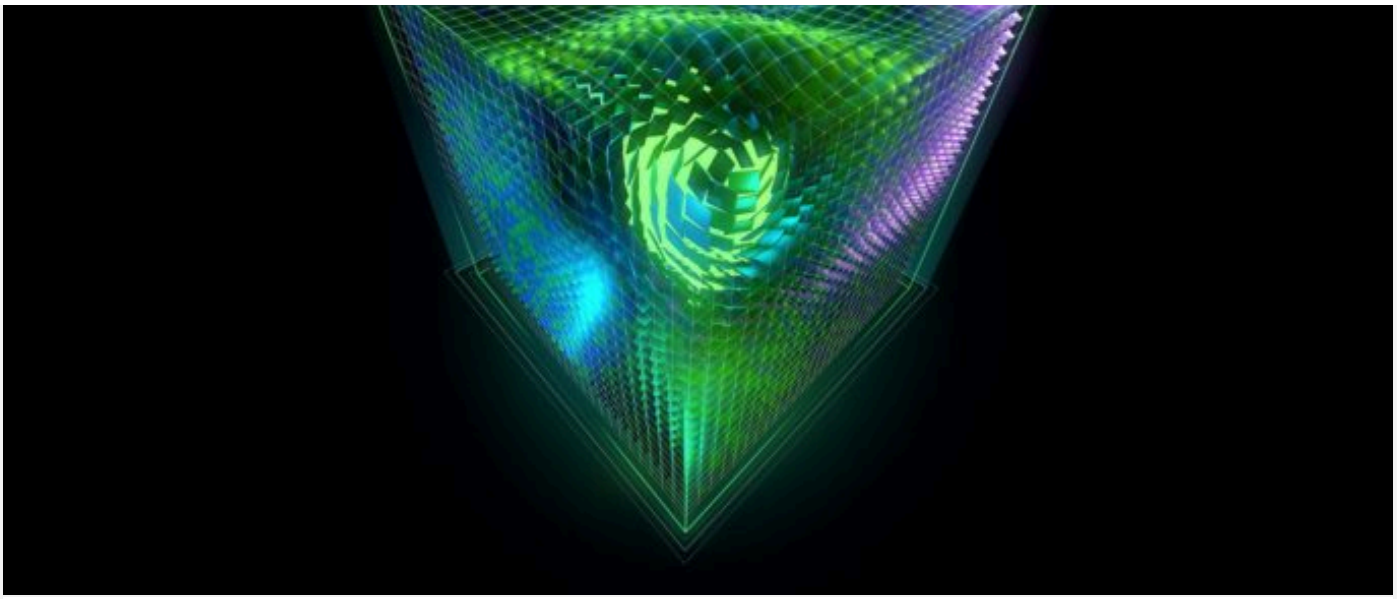
Unifying the CUDA Python Ecosystem

Discuss (6) +10 Like



Announcing NVIDIA CUDA 11.3 Toolkit Availability and Preview Release of CUDA Python

team. His research focused on domain-specific languages to generate high-performance code for physics simulations with the PETSc and FEniCS projects. Andy is a leader in the Python open-source software community. He's most notably a co-creator of the Dask distributed



Running Python UDFs in Native NVIDIA CUDA Kernels with the RAPIDS cuDF

```
  File "cupy/_core/core.pyx", line 1, in init cupy._core.core
  File "C:\Users\...\AppData\Local\Programs\Python\Python313\Lib\site-packages\cupy\cuda\__init__.py", lin
    from cupy.cuda import compiler  # NOQA
  File "C:\Users\...\AppData\Local\Programs\Python\Python313\Lib\site-packages\cupy\cuda\compiler.py", li
    from cupy.cuda import device
  File "cupy/cuda/device.pyx", line 105, in init cupy.cuda.device
  File "cupy/_util.pyx", line 52, in cupy._util.memoize.decorator
  File "C:\Users\...\AppData\Local\Programs\Python\Python313\Lib\functools.py", line 57, in update_wrapper
    setattr(wrapper, attr, value)
AttributeError: attribute '__name__' of 'builtin_function_or_method' objects is not writable
```

How can I get around it?

July 11, 2025

ashwint

Thanks! It looks like you might be running into **Support Cython 3.1 · Issue #9128 · cupy/cupy · GitHub**. What version of CuPy do you have installed?

```
pip list | grep cupy
```

July 11, 2025

LaszloHars

cupy 13.4.1, installed automatically
Windows-11

Maybe CUDA doesn't support Python 3.13 yet.
For what it's worth, I'm currently using **Python 3.12** with **CUDA 12.9.1**, and **cuda.cccl** is working fine.

July 11, 2025

LaszloHars

After upgrading cupy to 13.5.1, I got another error:

```
 Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v.1943 64 bit (AMD64)] on win32
C:\Users\...\Documents\Python\transform-iterator.py
Traceback (most recent call last):
  File "C:\Users\...\Documents\Python\transform-iterator.py", line 14, in <module>
    import cuda.cccl.parallel.experimental as parallel
ModuleNotFoundError: No module named 'cuda'
```

The module is named as cuda-cccl, not cuda.cccl, but no hyphen is allowed in import names. Should I change the module name or use importlib.import_module()?

Continue the discussion at forums.developer.nvidia.com

1 more reply

Participants