nvtx

Release 0.2.4

NVIDIA Corporation

May 24, 2022

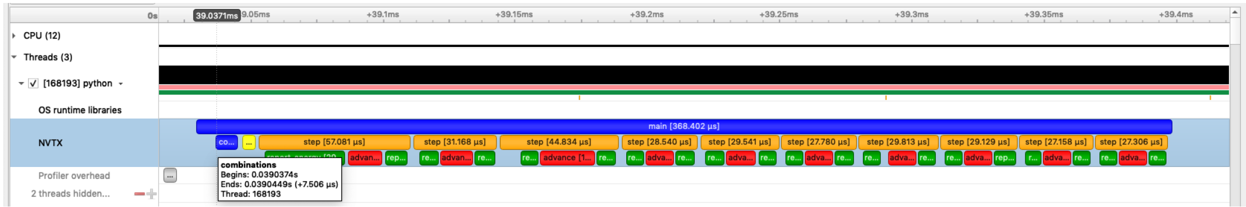
CONTENTS

1 Quick Demo **3**

2 Contents **5**
2.1 Installation 5
2.2 Tools for annotating code 5
2.3 Automatic function annotation 8
2.4 Reference 8

3 Indices and tables **9**

`nvtx` gives your tools to annotate your Python code (or automatically annotates it for you). Annotated code can be analyzed and visualized by third-party applications such as NVIDIA Nsight Systems. For example, you can produce detailed timelines of execution of Python programs annotated with `nvtx`:



CHAPTER

ONE

QUICK DEMO

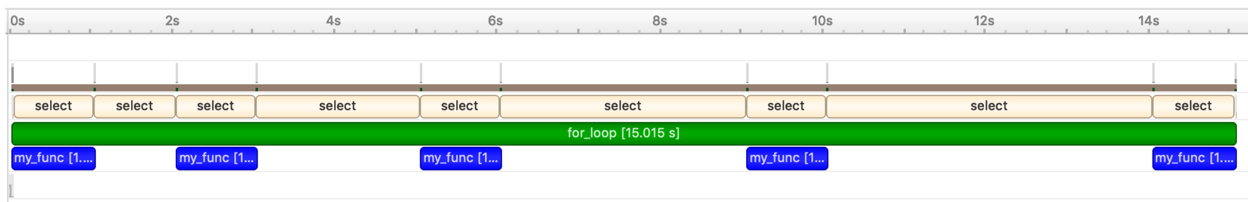
Here is an example of using the annotation tools provided by `nvtx`:

```
# example_lib.py

import time
import nvtx


def sleep_for(i):
    time.sleep(i)

@nvtx.annotate()
def my_func():
    time.sleep(1)

with nvtx.annotate("for_loop", color="green"):
    for i in range(5):
        sleep_for(i)
        my_func()
```

Adding annotations to your code doesn't achieve anything by itself. To derive something useful from annotated code, you'll need to use a third-party application that supports NVTX annotations. The command below uses the Nsight Systems command-line interface to collect information from the annotated code:

```
nsys profile python demo.py
```

This produces a `.qdrep` file containing information about the annotated code. Opening that file in the Nsight Systems GUI, you can see a timeline of execution of your program:



CHAPTER TWO

CONTENTS

2.1 Installation

`nvtx` requires Python >=3.6,<3.10, and is tested on Linux only.

Install using *conda* (preferred):

```
conda install -c conda-forge nvtx
```

Install using *pip*:

```
python -m pip install nvtx
```

Or *conda*:

```
conda install -c conda-forge nvtx
```

2.2 Tools for annotating code

2.2.1 annotate

The `annotate()` function annotates a code range, i.e., one or more statements. Each code range may have a message and a color associated with it. This makes it easy to distinguish ranges when visualizing them. `annotate` can be used in two ways:

As a decorator:

```
@nvtx.annotate(message="my_message", color="blue")
def my_func():
    pass
```

As a context manager:

```
with nvtx.annotate(message="my_message", color="green"):
    pass
```

When used as a decorator, the `message` argument defaults to the name of the function being decorated:

```
@nvtx.annotate()  # message defaults to "my_func"
def my_func():
    pass
```

2.2.2 start_range and end_range

In certain situations, it is impossible to use `annotate()`, e.g., when a code range spans multiple functions or in asynchronous code. In such cases, the `start_range()` and `end_range()` functions can be used instead.

The `start_range()` function is called at the beginning of a code range, and returns a handle. The handle is passed to the `end_range()` function, which is called at the end of the code range.

```
rng = nvtx.start_range(message="my_message", color="blue")
# ... do something ... #
nvtx.end_range(rng)
```

2.2.3 mark

The `mark()` function marks an instantaneous event in the execution of a program. For example, you may want to mark when an exceptional event occurs:

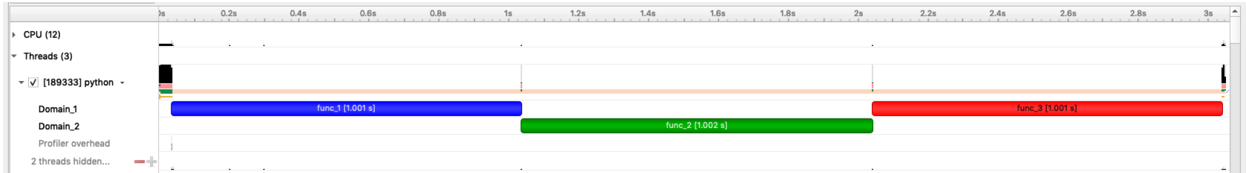
```
try:
    something()
except SomeError():
    nvtx.mark(message="some error occurred", color="red")
    # ... do something else ...
```

2.2.4 Domains

In addition to a message and a color, annotations can also have a *domain* associated with them. This allows grouping annotations.

```
import time
import nvtx


@nvtx.annotate(color="blue", domain="Domain_1")
def func_1():
    time.sleep(1)


@nvtx.annotate(color="green", domain="Domain_2")
def func_2():
    time.sleep(1)


@nvtx.annotate(color="red", domain="Domain_1")
def func_3():
    time.sleep(1)


func_1()
func_2()
func_3()
```

The timeline generated from the above:



Domains should be used sparingly as they are expensive to create. It is typically recommended to use a single domain per library. For grouping of annotations within a library, e.g., distinguishing annotations relating to compute, memory and I/O, use *Categories* instead.

2.2.5 Categories

Categories allow grouping of annotations within a domain.

```
import time
import nvtx


@nvtx.annotate(color="blue", domain="Domain_1", category="Cat_1")
def func_1():
    time.sleep(1)


@nvtx.annotate(color="green", domain="Domain_1", category="Cat_2")
def func_2():
    time.sleep(1)


@nvtx.annotate(color="red", domain="Domain_2", category="Cat_1")
def func_3():
    time.sleep(1)


@nvtx.annotate(color="red", domain="Domain_2", category=2)
def func_4():
    time.sleep(1)

func_1()
func_2()
func_3()
func_4()
```

In the example above, *func_1* and *func_2* are grouped under the domain *Domain1*, but under different categories within that domain.

Although *func_1* and *func_3* are both grouped under a category named *Cat_1*, they are unrelated as each domain maintains its own categories.

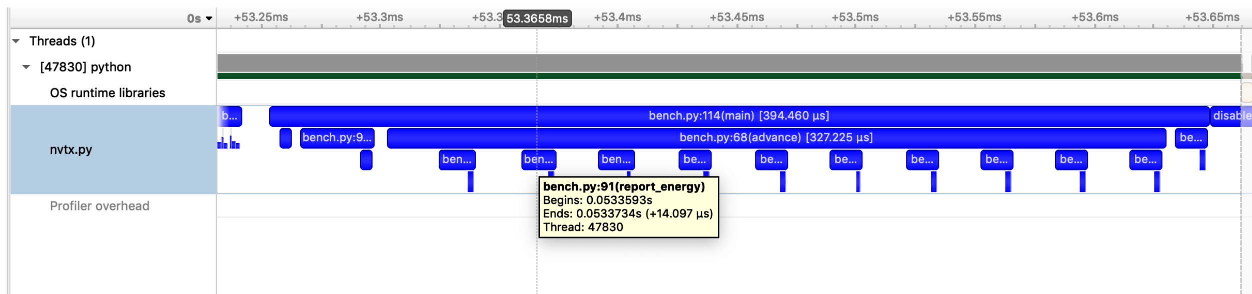
Unlike domains, categories are not expensive to create and manage. Thus, you should prefer categories for maintaining several groups of annotations.

2.3 Automatic function annotation

Annotating code manually is not always desirable, for example, when you have lots of functions to annotate, or when you want to capture information from third-party libraries.

`nvtx` can automatically annotate each function call in your program. Note that doing this adds a tiny amount of overhead to each and every function invocation, which can significantly impact the overall runtime (by more than 10x).

This can give you lots of useful information that manual annotation cannot



2.3.1 Command-line interface

You can invoke `nvtx` as a command-line script, which annotates every function call, with no changes to the source code:

```
python -m nvtx script.py
```

2.3.2 The Profile class

You can also use `Profile` to enable and disable automatic function annotation in different parts of your program:

```
pr = nvtx.Profile()
pr.enable()  # begin annotating function calls
# -- do something -- #
pr.disable()  # stop annotating function calls
```

2.4 Reference

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search