



CONFERENCE

—NORTH AMERICA 2026—

Observability tooling for CUDA Graphs workloads

Natalia Gimelshein Driss Guessous

Why CUDA Graphs

- On modern GPUs CUDA Graphs are needed to reduce CPU overheads
- To reduce jitter between processes on multi-GPU jobs
- Many cool new features (green contexts, locality domains) require CUDA Graphs to reduce associated overheads
- CUDA Graphs are a form of tracing, so some optimizations can be directly performed on CUDA Graph to speed up execution even more
- But there's a problem - it's very hard to do performance analysis for cudagraphed workloads

CUDA Graph profile



It's a wall of kernels

- Cpu-side information (which layer, which op) is unavailable during graph replay, so profiler can't capture it
- The execution stream doesn't necessarily match what stream was used in eager

Solution

- Record information during graph capture using cupti callbacks
- Correlate recorded information with the profile using graph node ids reported by profiler
- Register autograd hook to automatically project forward annotations to backward
- The same autograd hook would also make backward nodes display on the same stream as forward, if forward was marked

Graph capture

- `mark_kernels` context manager associates specified information with all annotatable nodes (kernels, memcopies, memsets, batch mem ops)
- `mark_stream` does the above, changes the stream to the specified stream, and adds stream entry to the annotations dictionary
- Requires cuda user mode driver ≥ 13.1

```
def forward(self, x: torch.Tensor) -> torch.Tensor:
    with mark_kernels({"layer": self.layer_idx, "stream": 0, "lane_name": "Compute"}):
        batch, seq, dim = x.shape
        with mark_kernels("attention_norm"):
            normalized = self.attn_norm(x)
        with mark_kernels("qkv"):
            qkv = self.qkv(normalized)
            qkv = qkv.view(batch, seq, 3, self.heads, self.head_dim)
            q, k, v = qkv.unbind(dim=2)
        with mark_kernels("attention"):
            attn = F.scaled_dot_product_attention(
                q.transpose(1, 2), k.transpose(1, 2), v.transpose(1, 2), is_causal=True
            )
```

Profiling

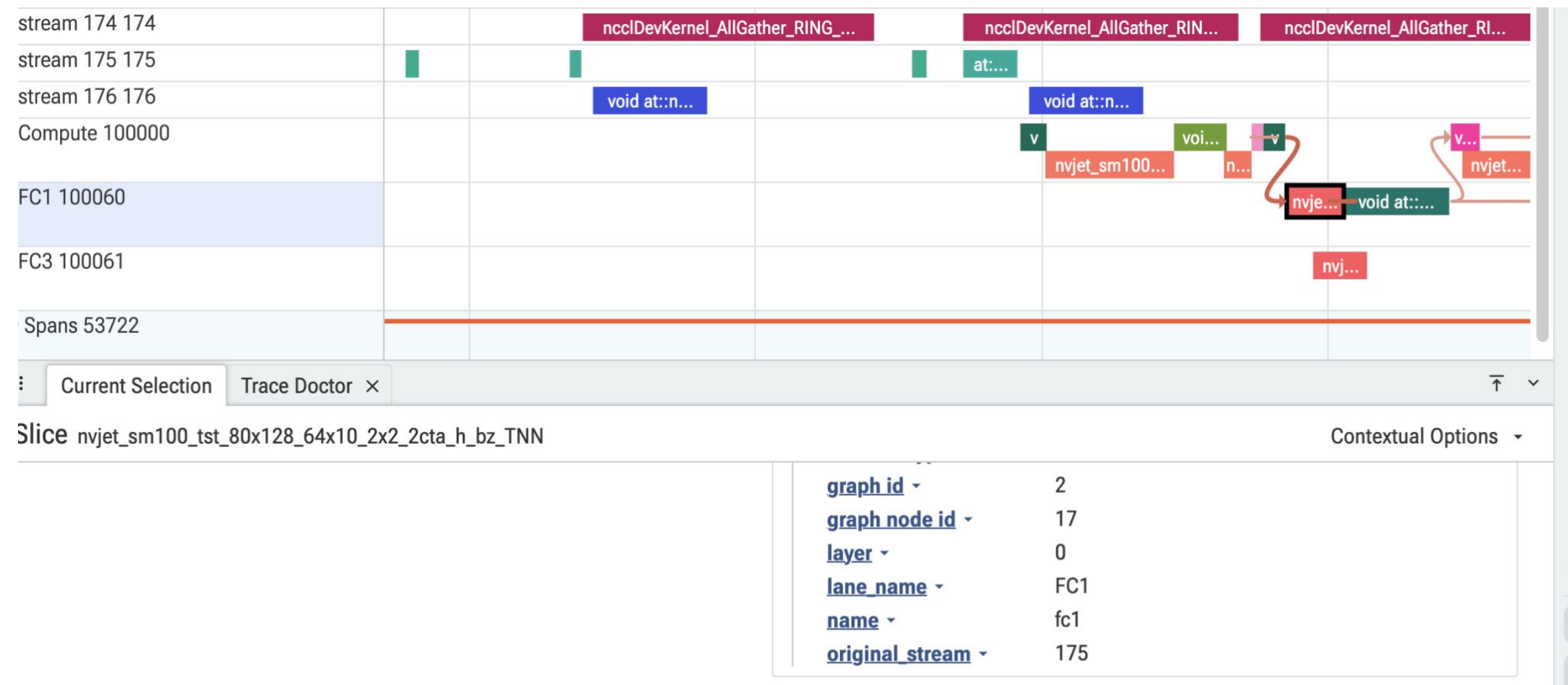
- Normally cupti allows only a single subscriber, so if you use cupti callback to record capture information, profiling would be unavailable
- We developed a new profiling backend. cuspy, that multiplexes its subscriber

```
activities=[  
    {  
        ProfilerActivity.CUDA: ProfilerActivityConfig(  
            profiler_configs=[CuspyConfig(enable_graph_dependencies=True)]  
        )  
    }  
]
```

- Annotations directly put into the output trace, no post-processing needed

Result

- Named lanes, kernel annotations, graph dependencies
- Can use all perfetto tools for even richer profile (visualize any args, macros)



Additional niceties

- Can use similar cupti callbacks to record stack traces of each created node to easily correlate profile to model source code
- Profiles can display pm sampling counters
- Can output rich information about graph topology (`get_graph_data`) - node topology, kernel names, external events
- Register hooks to run at the different stages of graph lifecycle - pre-capture, post-capture, post-instantiation, replay