NCCL Profiler Plugin API – eine Machbarkeitsstudie

Inhaltsverzeichnis

1 TODO / Struktur

1.1 Table of Contents (Entwurf)

- 0. Abstract – gpu communication profiling/tracing motivieren
- 0. Introduction – vorausgesetztes Verständis soweit nötig für bestimmte Konzepte erwähnen oder erläutern (z.B. MPI, SLURM, NCCL)
- 1. Die Profiler API
  - 1.1 Einbindung des Plugins in NCCL.
    - first draft
    - TODO final draft
  - 1.2 "rohe" API definition, kurz und knapp ohne viel Erläuterung.
    - first draft
    - TODO final draft
  - 1.3 Der Codeflow: application NCCL API calls → profiler API calls
    - TODO first draft
    - TODO final draft
  - 1.4 Der Codeflow++ (detailliertere Betrachtung)
    - ncclGroup, multi gpu streams
    - multi threaded application
    - multi-node environment: mehrere Prozesse und profiler instances
- 2. Was einem die Profiler Plugin API (nicht) ermöglicht

    - beispielhaft logging, running metrics, cupti, ...

- 3. Warum (nicht) die Profiler Plugin API in Erwägung ziehen?
    - Experimente/Benchmarking auswerten
    - Genauigkeit & consistency der timings der api calls diskutieren?
    - Besondere Vor- & Nachteile hervorheben

- 4. Conclusion – Nützlichkeit für P-Score Messsystem

1.2 Main content chunks / concepts

- einfache code example walkthroughs
- swim lane diagrams
    - user API call $\rightarrow$ nccl profiler init/finalize
    - user API call $\rightarrow$ nccl profiler start/stop/recordEventState
- benchmarking, measurements
- conclusion

2 Abstract

- AI – big use case for HPC
- Expensive workloads! Desire to understand and optimize application performance
- big part of AI workloads is GPU communication
- as they often span across many GPUs
- NCCL – the library that implements communication routines for NVIDIA GPUs
- provides an interface to plugin a custom profiler into NCCL to extract performance data

3 Introduction

TODO Mention (as needed):

- MPI concepts
- NCCL concepts
- SLURM concepts

3.1 NCCL Concepts

For understanding the behaviour of profiler it is helpful to first take a look what happens under the hood of NCCL when the application calls the NCCL API.

A usual nccl application program flow follows this code structure:

```
// create nccl communicators
ncclCommCreate();

// allocate memory for computation and communication
prepareDeviceForWork();

// do computation and commmunication
callNcclCollectives();
// ...

// finalize and clean up nccl communicators
ncclCleanup();
```

During nccl communicator creation, nccl will internaly spawn a thread called ProxyProgress, which will handle network requests for GPU communication during collective and p2p operations.

Whenever then the application calls nccl collectives, nccl will decide on what network operations to do and add them to a pool. The ProxyProgress thread will read these operations from the pool and progress them by making operation specific functions calls.

This behaviour is useful for understanding when network specific activity is profiled, which will be explained in section 1.3 (TODO: link).

Through following sections the feasability of the NCCL profiler plugin API will be made clear:

1. How does it work?
2. What can you do with it?
3. Why would you use it? pros & cons

4 How it works

4.1 How nccl detects the profiler plugin

NCCL looks for a shared library which represents the profiler plugin, checking for the environment variable `NCCL_PROFILER_PLUGIN`: `profilerName = ncclGetEnv(NNCCL_PROFILER_PLUGIN")`. It then calls `dlopen(name, RTLD_NOW | RTLD_LOCAL)` and `dlsym(handle, nncclProfiler_v5")`.

> If `NCCL_PROFILER_PLUGIN` is set, attempt loading the library with name specified by `NCCL_PROFILER_PLUGIN`; if that fails, attempt loading `libnccl-profiler-<NCCL_PROFILER_PLUGIN>.so`.

If NCCL_PROFILER_PLUGIN is not set, attempt loading libnccl-profiler.so. If no plugin was found, do not enable profiling. If NCCL_PROFILER_PLUGIN is set to STATIC_PLUGIN, the plugin symbols are searched in the program binary.

(Quelle: NCCL docs, Abschnitt NCCL_PROFILER_PLUGIN: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/env.html#nccl-profiler-plugin)

The exact implementation details can be found at /src/plugin/plugin_open.cc and /src/plugin/profiler.cc.

4.2 The profiler API

The plugin must implement a profiler API specified by NCCL, in the form of exposing a struct. This struct should contain pointers to all the functions required by the API:

```
ncclProfiler_v5_t ncclProfiler_v5 = {
  const char* name;
  ncclResult_t (*init)(...);
  ncclResult_t (*startEvent)(...);
  ncclResult_t (*stopEvent)(...);
  ncclResult_t (*recordEventState)(...);
  ncclResult_t (*finalize)(...);
};
```

The plugin loading mechanism expects the struct variable name to follow this naming convention `ncclProfiler_v{versionNum}`.

The exact profiler API can be found under /src/include/plugin/profiler/. As of NCCL release v2.29.1, five functions must be implemented.

4.2.1 init

```
ncclResult_t init(
  void** context, // return value - opaque profiler context object
  uint64_t commId, // communicator id
  int* eActivationMask, // return value - bitmask that sets which events are tracked
  const char* commName, // user assigned communicator name
  int nNodes, // number of nodes in communicator
  int nranks, // number of ranks in communicator
  int rank, // rank identifier in communicator
  ncclDebugLogger_t logfn // logger function
);
```

Notably, `void** context` is an opaque handle that the plugin developer should point to any custom context object. This pointer is then again passed as argument in the other API calls `startEvent` and `finalize`. This context object is separate across communicators.

The plugin developer should point `int* eActivationMask` to a bitmask, which tells nccl which type of events that the profiler plugin wants to track. Internally this bitmask is initialized with `0` (no events tracked). Setting it to `4095` will track all events.

TODO: what to do with `ncclDebugLogger_t logfn`?

4.2.2 startEvent

```
ncclResult_t startEvent(
  void* context, // opaque profiler context object
  void** eHandle, // return value - event handle for event
  ncclProfilerEventDescr_v5_t* eDescr // pointer to event descriptor
);
```

The plugin developer should point `void** eHandle` to a custom event object. This pointer is then passed again into `stopEvent` and `recordEventState`.

4.2.3 stopEvent

```
ncclResult_t stopEvent(void* eHandle); // handle to event object
```

4.2.4 recordEventState

```
ncclResult_t recordEventState(
  void* eHandle,
  ncclProfilerEventState_v5_t eState,
  ncclProfilerEventStateArgs_v5_t* eStateArgs
);
```

4.2.5 finalize

```
ncclResult_t finalize(void* context);
```

Besides these functions, the profiler plugin struct also contains a `name` field.

> The name field should point to a character string with the name of the profiler plugin. This will be used for all logging, especially when `NCCL_DEBUG=INFO` is set.

(Quelle: `/ext-profiler/README.md`)

4.3 Where nccl triggers profiler API callbacks

Below, the callbacks to the profiler API will be explained. They can be categorized as follows:

- callbacks for profiler initialization and finalization
- callbacks for events directly related to the application calling the NCCL API (e.g. `ncclAllReduce()`)
- callbacks for network events triggered by the proxy progress thread processing network requests
- TODO: add copy engine based events section?

4.3.1 Callbacks for profiler initialization and finalization

The profiler API's `init()` function is called in `ncclProfilerPluginInit()` during the initialization of nccl.

As of NCCL release 2.29.1, the following depicts the flow from user API call to NCCL calling `ncclProfilerPluginInit()`:

The implementation can be found at `/src/init.cc` and `/src/plugin/profiler.cc`.

The profiler API's `finalize()` function is called in `ncclProfilerPluginFinalize()` after a user API call is made to free resources associated with the communicator object.

```
User API Internal Flow
-----------------------------------------------------------------
ncclCommAbort() -+
ncclCommDestroy() -+-----------> commReclaim() ----+
                                           |
                                           v
                                 ncclProfilerPluginFinalize()
                                           |
                                           v
                                 ncclProfiler->finalize()
                                 ncclProfilerPluginUnload()
```

See implementation details at `/src/init.cc`, `/src/plugin/profiler.cc` and `/src/plugin/plugin_open.cc`.

4.3.2 Callbacks for events related to NCCL API calls

> NCCL profile API events are generated when the API calls are made, right after NCCL checks for graph capture information. They parent collective, point-to-point and kernel launch events and persist across multiple operations in a group.

(Quelle: `/ext-profiler/README.md`)

TODO: `@cursor_nccl_profiling_event_flow.md`

4.3.3 Proxy progress thread – callbacks when processing network requests

TODO: `@cursor_nccl_profiling_event_flow.md`

During nccl initialization, after calling `ncclProfilerPluginInit()`, `ncclProxyCreate(comm)` is called. This creates a new thread `ncclProxyService`. If needed this Proxy Service will launch another thread `ncclProxyProgress`.

TODO correction to above paragraph: proxy threads are not 'once per process' in the strict sense; they are once per shared-resource owner (once per distinct `sharedRes` that has a proxy). That can mean one set per process in the 'one root, split with share' case, but usually (and by default) means several sets per process when you have multiple roots or split/shrink without sharing (not shared by default).

> proxyState is shared among parent comm and split comms. comm->proxyState->thread is pthread_join()'d by commFree() in init.cc when the refCount reduces down to 0.

(Quelle: `/src/proxy.cc`)

> Due to the asynchronous nature of NCCL operations, events associated to collective and point-to-point operations are not easy to delimit precisely. StopEvent for collectives simply indicates to the profiler that the collective has been enqueued. Without both proxy and/or kernel activity it is impossible for the profiler to figure out when a collective operation completes. The profiler can leverage proxy and/or kernel event information, if these are enabled, to estimate when the collective ends.

(Quelle: `/ext-profiler/README.md`, leicht umformuliert)

4.3.4 Logging

- logging function from `init` (TODO)
- code snippet custom logging infra, timestamping

4.3.5 Tracking & running metrics

- code snippet showing where to CRUD custom context object
- code snippet showing where to CRUD custom event object

4.3.6 Kernel tracing with CUPTI

- cupti ext id mechanism briefly explained
- code snippet where to init/cleanup cupti & use cupti mechanism
- changing profiling behaviour during runtime (TODO: check example_profiler and inspector?)

5 Why would you use it? pros & cons

- customizable
- might require maintenance / active development since NCCL is actively developed
- overhead: nvidia advertises their `inspector` implementation as efficient enough for "always-on" in production

5.1 NCCL_DEBUG

See NCCL docs: `https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/env.html#nccl-debug`

- NCCL already comes with debug logging which can be set to various levels of granularity
- INFO – Prints debug information
- TRACE – Prints replayable trace information on every call.

5.2 Known limitations

Known limitations: `https://github.com/NVIDIA/nccl/tree/master/ext-profiler/README.md`

Kernel events instrumentation leverages counters exposed by the kernel to the host and the proxy progress thread. Thus, the proxy progress thread infrastructure is shared between the network and the profiler. If the proxy is serving network requests the kernel profiling probing can be delayed, causing loss of accuracy. Similarly, if the CPU is under heavy load and the scheduling of the proxy progress thread is delayed, a similar loss of accuracy can be encountered.

To mitigate this effect, with version 4 of the profiler NCCL uses a per-channel ring buffer of 64 elements. Every counter is complemented by two timestamps (ptimers) supplied by the NCCL kernel (one for start and one for stop of the operation in the kernel). NCCL propagates these timestamps to the profiler plugin that it can convert them to CPU time domain.

6 Comparison to MPI (TODO)

6.1 MPI

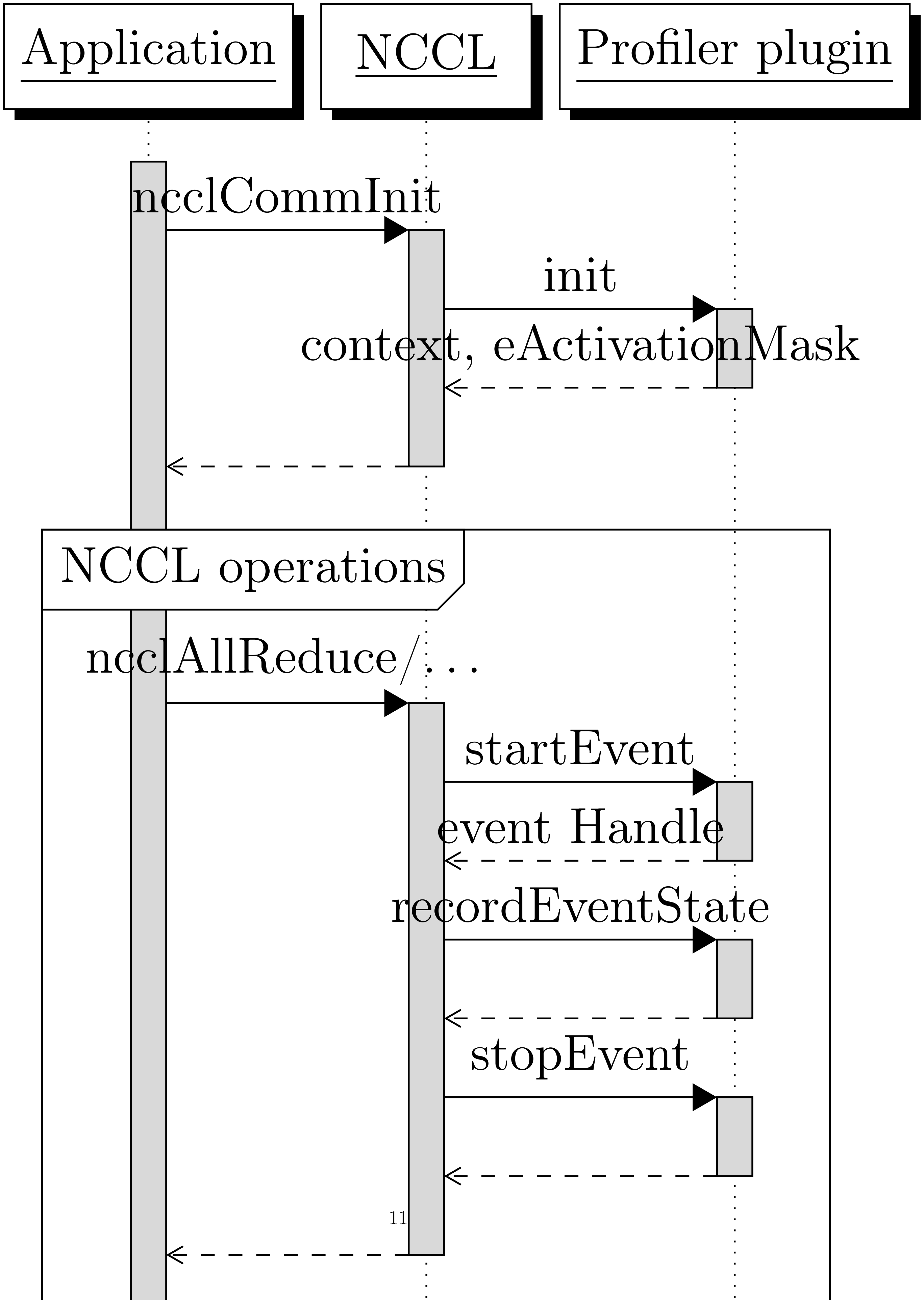
- centered around cpu processes communicating
- ranks/tasks per cpu process
- rank = CPU process

6.2 NCCL

- centered around GPUs communicating; cpu processes/threads exist for communication orchestration
- ranks/tasks possibly per cpu thread
- ranks/tasks possibly assigned to 1 GPU (or many GPUs? TODO)
- rank = GPU device

7 TODO

- link code snippets to source code files+lines
- do pytorch (+jax+tensorflow) use this profiler plugin api?


Application
NCCL
Profiler plugin
ncclCommInit
init
context, eActivationMask
NCCL operations
ncclAllReduce/. . .
startEvent
event Handle
recordEventState
stopEvent