NCCL Profiler Plugin API – A Feasibility Study

Contents

1 TODO / Structure (from Markdown)

1.1 Table of Contents

- Abstract – motivate GPU communication profiling/tracing
- Introduction – assumed background as needed (e.g. MPI, NCCL, SLURM, Score-P)
- The Profiler API
  - How NCCL detects the profiler plugin (first draft; TODO final draft)
  - The Profiler API definition (with codeflow/swimlane diagrams showing where NCCL calls the profiler api)
  - Code examples illustrating the codeflow/behaviour (simple example, multi-node, ncclGroup)
  - What is possible with the Profiler Plugin API? Considerations and Pitfalls for (logging, running metrics, CUPTI, ...) - section title WIP
  - Performance and scalability of the Profiler Plugin API (experiments, accuracy)
- Potential integration with Score-P
- Conclusion

1.2 Main content chunks / concepts

- Simple code example walkthroughs
- Swim-lane diagrams (User API → init/finalize; start/stop/recordEventState)
- Benchmarking, measurements
- Conclusion

2 Abstract

Artificial intelligence (AI) has established itself as a primary use case in high-performance computing (HPC) environments due to its compute-intensive and resource-intensive workloads. Analyzing and optimizing application performance is therefore essential to maximize efficiency and reduce costs. Many AI workloads involve communication between GPUs, often distributed across numerous GPUs in multi-node systems. The NVIDIA Collective Communication Library (NCCL) serves as the core library for implementing optimized communication primitives on NVIDIA GPUs. To provide detailed performance insights, NCCL offers a flexible profiler plugin API. This allows developers to directly integrate custom profiling tools into the library to extract detailed performance data on communication operations. This feasibility study explores the capabilities and integration mechanisms of the API.

TODO finally, mention the structure of this document?

3 About NCCL

NCCL (pronounced "Nickel") was first introduced by NVIDIA in 2015 at the Supercomputing Conference (SC15), with an accompanying presentation highlighting its optimized collectives for multi-GPU systems. Although code was made available on GitHub, the release of NCCL 2.0 in 2017, which brought support for NVLink, was initially only available as pre-built binaries. With the release of NCCL 2.3 in 2018, it returned to being fully open source. The NCCL Profiler Plugin API was even later introduced with NCCL 2.23 in early 2025.

Before taking a closer look at the Profiler Plugin API, it is helpful to have some rudimentary understanding on certain designs in NCCL.

3.1 Comparison to MPI

Although NCCL is inspired by the Message Passing Interface (MPI) in terms of API design and usage patterns, there are notable differences due to their respective focuses:

- **MPI**: Communication is CPU-based. A rank corresponds to a single CPU process, and within a communicator, each process is assigned exactly one unique rank.

- **NCCL**: Communication is GPU-based, with CPU threads handling orchestration. A rank corresponds to a GPU device, and a single CPU thread can manage multiple ranks (i.e., multiple devices) using functions such as `ncclGroupStart` and `ncclGroupEnd`.

3.2 Relevant NCCL internals

It helps to understand what NCCL does internally when an application calls the NCCL User API.

A typical NCCL application follows this basic structure:

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

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

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

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

During NCCL communicator creation, NCCL internally spawns a thread called `ProxyService`. This thread lazily starts another thread called `ProxyProgress`, which handles network requests for GPU communication during collective and P2P operations. See Figure 1.

The guards `if (proxyState->refCount == 1)` and `if (!state->thread)` ensure that these threads are created once per shared resource (struct `ncclSharedResources`). The SharedResource has a ProxyState field. The fields in ProxyState are used to ensure only one instance of each thread exists:

/src/include/comm.h

```
struct ncclSharedResources {
  struct ncclComm* owner; /* communicator which creates this shared res. */
  struct ncclProxyState* proxyState;
  // other fields
}
```

/src/include/proxy.h

```
struct ncclProxyState {
  int refCount;
  pthread_t thread;
  ncclProxyProgressState progressState;
  // other fields
}

struct ncclProxyProgressState {
  struct ncclProxyOpsPool* opsPool;
  // other fields
```

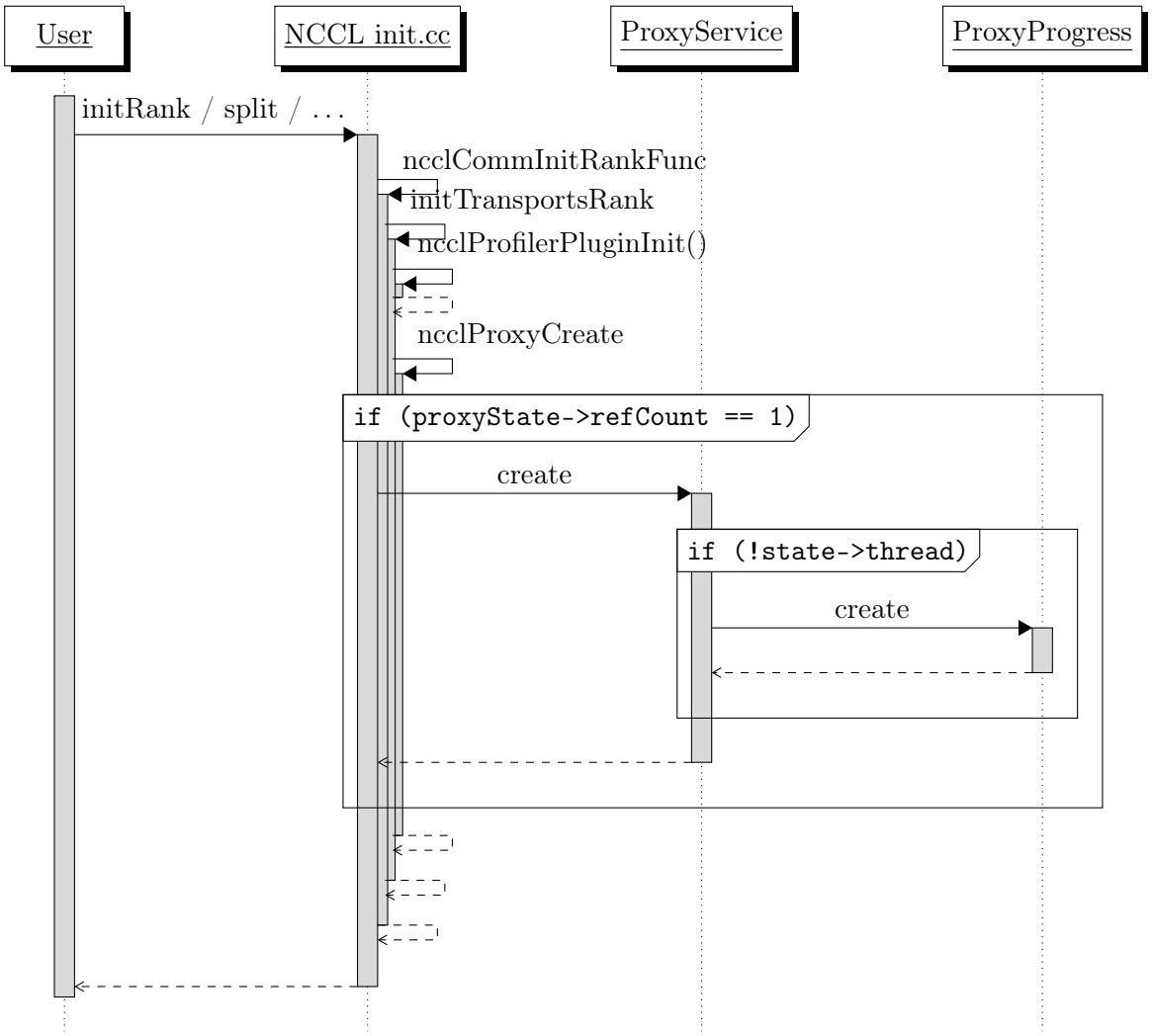

Figure 1: Thread creation: User API → NCCL internal init → create ProxyService → create ProxyProgress.

```
}

struct ncclProxyOpsPool {
  struct ncclProxyOp ops[MAX_OPS_PER_PEER*NCCL_MAX_LOCAL_RANKS];
  // other fields
}

struct ncclProxyOps {
  // other fields
}
```

By default every NCCL communicator has its own shared resource. When the application calls `ncclCommSplit()` or `ncclCommShrink()` where the original communicator was initialized with a `ncclConfig_t` with fields `splitShare` or `shrinkShare` set to `1`, the newly created communicator shares the shared resource (and the proxy threads) with the parent communicator.

```
/* 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.  */
```

(Quoted from /**src**/**proxy.cc**)

Later, whenever the application calls the NCCL User API, NCCL internally decides what network operations to perform and calls `ncclProxyPost()` to post them to a proxyOpsPool (See Figure 2).

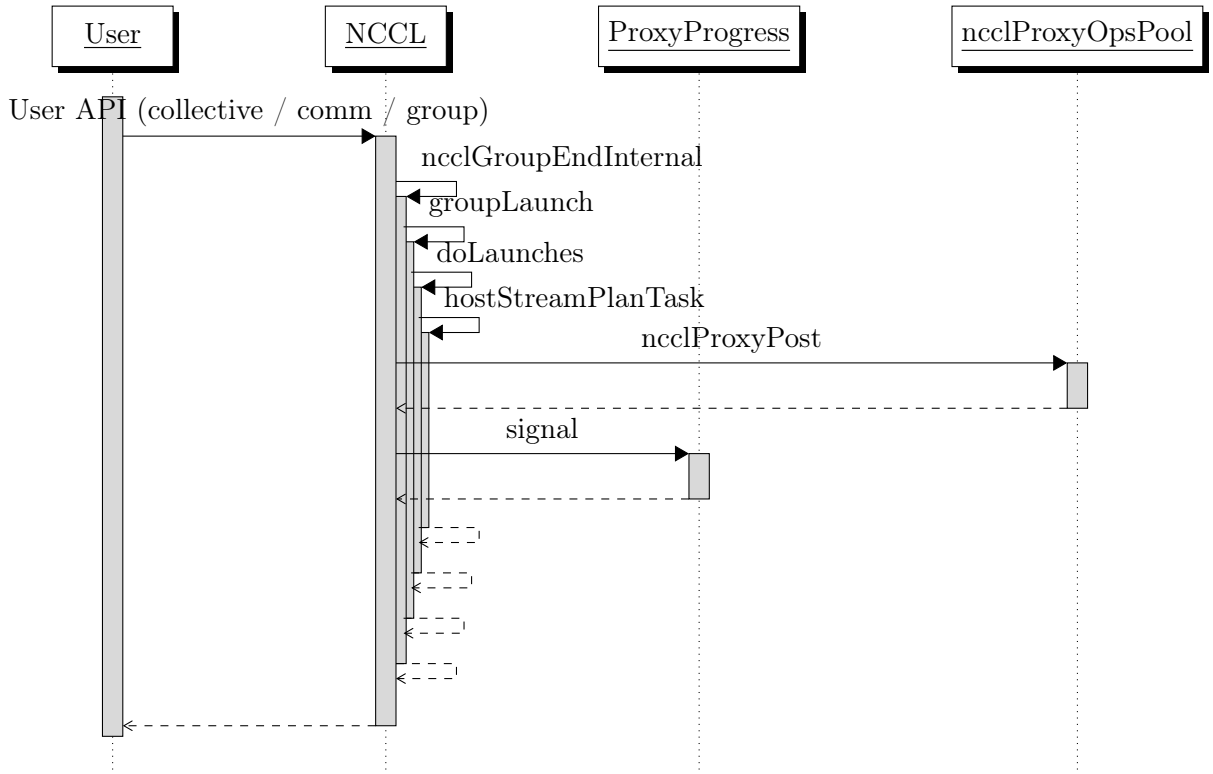

Figure 2: Flow from User API to `ncclProxyPost()`

The ProxyProgress thread reads from this pool when calling `ncclProxyGetPostedOps()` and progresses the ops. See Figure 3.

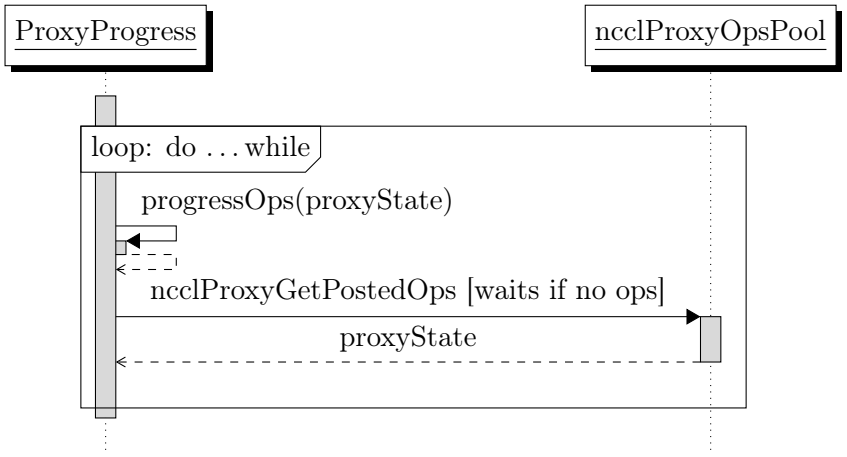

Figure 3: **/src/proxy.cc** `ncclProxyProgress()` progressing loop: progress ops, then get posted ops (or wait).

In the next section, understanding the behaviour of NCCL for network-related activity is helpful in relation to the behavior of Profiler Plugin API.

4 The Profiler API

4.1 How NCCL detects the profiler plugin

When a NCCL communicator is created, NCCL looks for a shared library that represents the profiler plugin by checking an environment variable:
`profilerName = ncclGetEnv("NCCL_PROFILER_PLUGIN");`
It then calls
`handle* = dlopen(name, RTLD_NOW | RTLD_LOCAL);`
and
`ncclProfiler_v5 = (ncclProfiler_v5_t*)dlsym(handle, "ncclProfiler_v5");`
to load the library immediately with local symbol visibility. See Figure 4.

- If `NCCL_PROFILER_PLUGIN` is set: attempt to load the library with the specified name; if that fails, attempt `libnccl-profiler-<NCCL_PROFILER_PLUGIN>.so`.
- If `NCCL_PROFILER_PLUGIN` is not set: attempt `libnccl-profiler.so`.
- If no plugin was found: profiling is disabled.
- If `NCCL_PROFILER_PLUGIN` is set to `STATIC_PLUGIN`, the plugin symbols are searched in the program binary.

(Source:
`https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/env.html#nccl-profiler-plugin`)

The profiler plugin is loaded when creating a communicator (before proxy thread creation). The plugin loading mechanism expects the struct variable name to follow the naming convention `ncclProfiler_v{versionNum}`, which also indicates the API version.

The profiler API has changed multiple times with newer NCCL releases; NCCL features a fallback mechanism to load older struct versions. However from personal experience, backwards compatibility with plugins from API version 2 may be limited. One instance is known, where a profiler plugin being developed against the NCCL release 2.25.1 with Profiler API version 2, was unable to run with the latest NCCL release. NCCL developers may be required to actively maintain older API versions, to ensure they safely work when old behaviour is getting deprecated and do not unexpectedly get handed new features from new API versions, of which the Profiler Plugin developer is not aware of (when faithfully implementing old API versions).

The exact implementation is in **/src/plugin/plugin_open.cc** and **/src/plugin/profiler.cc**.

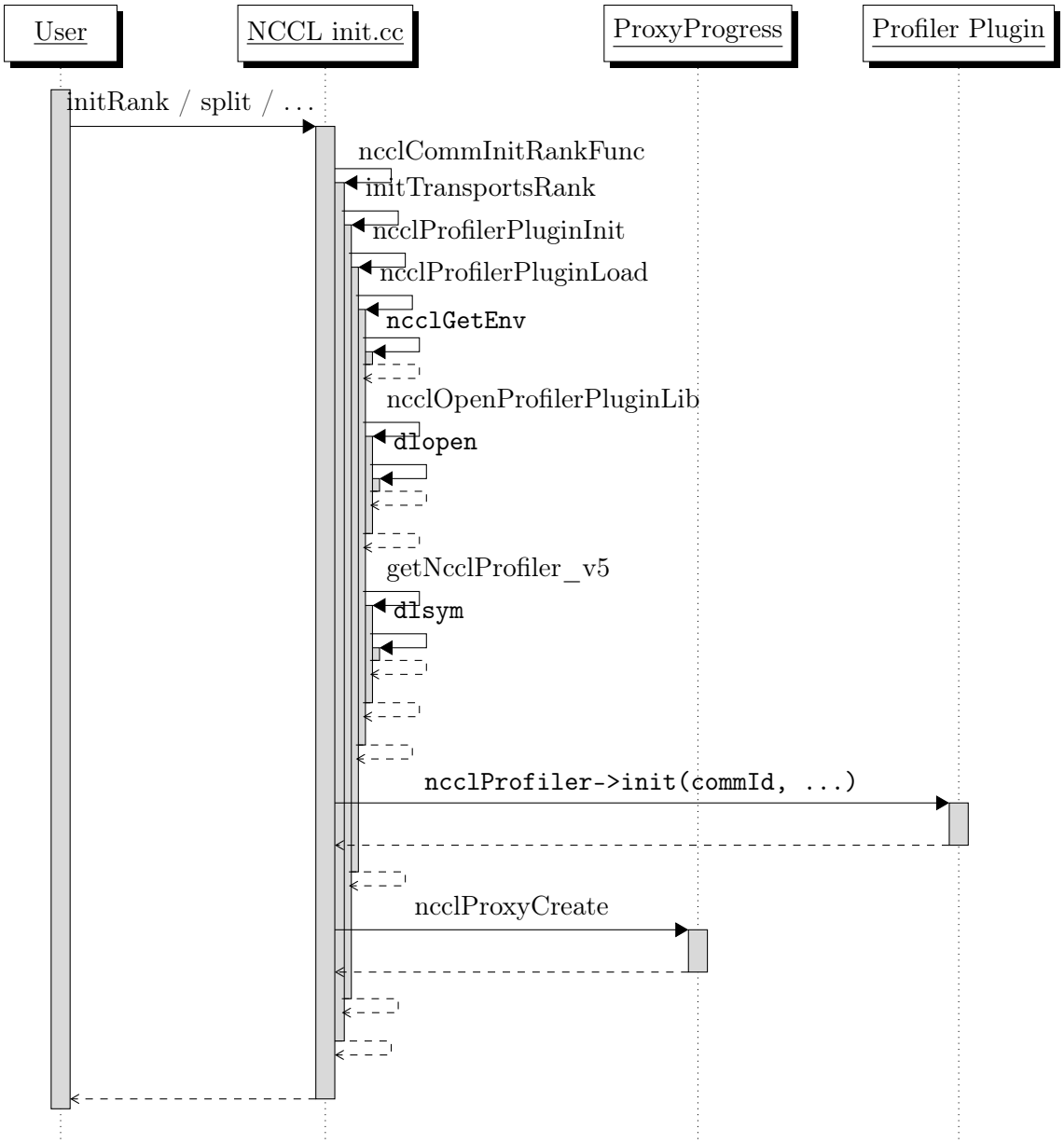

Figure 4: User API → NCCL init → load profiler plugin and call `profiler->init()`.

4.2 The profiler API definition

The plugin must implement a profiler API specified by NCCL by exposing a struct. This struct should contain pointers to all functions required by the API. A plugin may expose multiple versioned structs for backwards compatibility with older NCCL versions.

```
ncclProfiler_v5_t ncclProfiler_v5 = {
  const char* name;
  ncclResult_t (*init)(...); // NCCL calls this right after loading
  ncclResult_t (*startEvent)(...); // at start of operations/activities
  ncclResult_t (*stopEvent)(...); // at end of these operations/activities
```

```
  ncclResult_t (*recordEventState)(...); // to record state of certain operations
  ncclResult_t (*finalize)(...); // before unloading the plugin
};
```

The full profiler API is under **/src/include/plugin/profiler/profiler_v{versionNum}.cc**. As of NCCL v2.29.1, version 6 is the latest; five functions must be implemented. Internally NCCL wraps calls to the profiler API in custom functions (found in **/src/include/profiler.h**).

NCCL invokes the profiler API at different levels to capture start/stop of NCCL groups, collectives, P2P, proxy, kernel and network activity. As the API function names suggest, this will allow the profiler to track these operations and activities as events.

The API functions and where NCCL invokes them are explained in the following sections.

4.2.1 init

`init` initializes the profiler plugin. NCCL passes follwing arguments:

```
ncclResult_t init(
  void** context, // out param - opaque profiler context
  uint64_t commId, // communicator id
  int* eActivationMask, // out param - bitmask for 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
);
```

Every time a communicator is created, `init()` is called immediately upon successful plugin load in `ncclProfilerPluginLoad()` (see Figure 4). If the profiler plugin `init` function does not return `ncclSuccess`, NCCL disables the plugin.

> As soon as NCCL finds the plugin and the correct ncclProfiler symbol, it calls its init function. This allows the plugin to initialize its internal context used during profiling of NCCL events.

(Source: **/ext-profiler/README.md**)

`void** context` is an opaque handle that the plugin developer may point to any custom context object; this pointer is passed again in `startEvent` and `finalize`. This context object is separate per communicator.

The plugin developer should set `int* eActivationMask` to a bitmask indicating which event types the profiler wants to track. The mapping is in **/src/include/plugin/nccl_profiler.h**; internally the mask defaults to 0 (no events). Setting it to 4095 will track all events.

`ncclDebugLogger_t logfn` is a function pointer to NCCL's internal debug logger (`ncclDebugLog`). NCCL passes this so the plugin can emit log lines through the same channel and filtering as NCCL:

the plugin may store the callback and call it with `(level, flags, file, line, fmt, ...)` when it wants to log. Messages then appear in NCCL's debug output (e.g. stderr or `NCCL_DEBUG_FILE`) and respect the user's `NCCL_DEBUG` level and subsystem mask. Using `logfn` keeps profiler output consistent with NCCL's own logs.

4.2.2 startEvent

`startEvent` is called when NCCL begins certain operations:

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

As of release v2.29.1 NCCL does not care about the return value. `void** eHandle` may point to a custom event object; this pointer is passed again in `stopEvent` and `recordEventState`. `eDescr` describes the started event. Its exact details can be found in /**src/include/plugin/profiler**/.

The field `void* parentObj` in the event descriptor is the `eHandle` of a parent event (or null). The use of this field can be explained as following:

All User API calls to Collective or P2P operations will start a Group API event. When networking is required, ProxyCtrl Events may be emitted. Depending on the 'eActivationMask' bitmask returned in the 'init()' function, further (child) events will be emitted in deeper sections of the nccl code base. It can be thought of as an event hierarchy with several depth levels:

```
Group API event
|
+- Collective API event
|  |
|  +- Collective event
|     |
|     +- ProxyOp event
|     |  |
|     |  +- ProxyStep event
|     |     |
|     |     +- NetPlugin event
|     |
|     +- KernelCh event
|
+- Point-to-point API event
|  |
|  +- Point-to-point event
|     |
|     +- ProxyOp event
|     |  |
|     |  +- ProxyStep event
|     |     |
|     |     +- NetPlugin event
|     |
|     +- KernelCh event
```

```
  |
  +- Kernel Launch event

ProxyCtrl event
```

(Source: **/ext-profiler/README.md**)

If the profiler enables tracking for event types lower in the hierarchy, NCCL also tracks their parent event types. The `parentObj` inside `eDescr` will be a reference to the `eHandle` of the respective parent event for the current event according to this hierarchy.

4.2.3 stopEvent

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

`stopEvent` tells the plugin that the event has stopped. `stopEvent` for collectives simply indicates to the profiler that the collective has been enqueued and not that the collective has been completed.

NCCL ignores the return value. `stopEvent` is called in the same functions that call `startEvent`, except for the GroupApi event (see diagram).

Figure 5 shows when NCCL emits `startEvent` and `stopEvent` after a User API call. The Proxy-Progress thread also emits `startEvent` and `stopEvent` while progressing ops (see Figure 6).

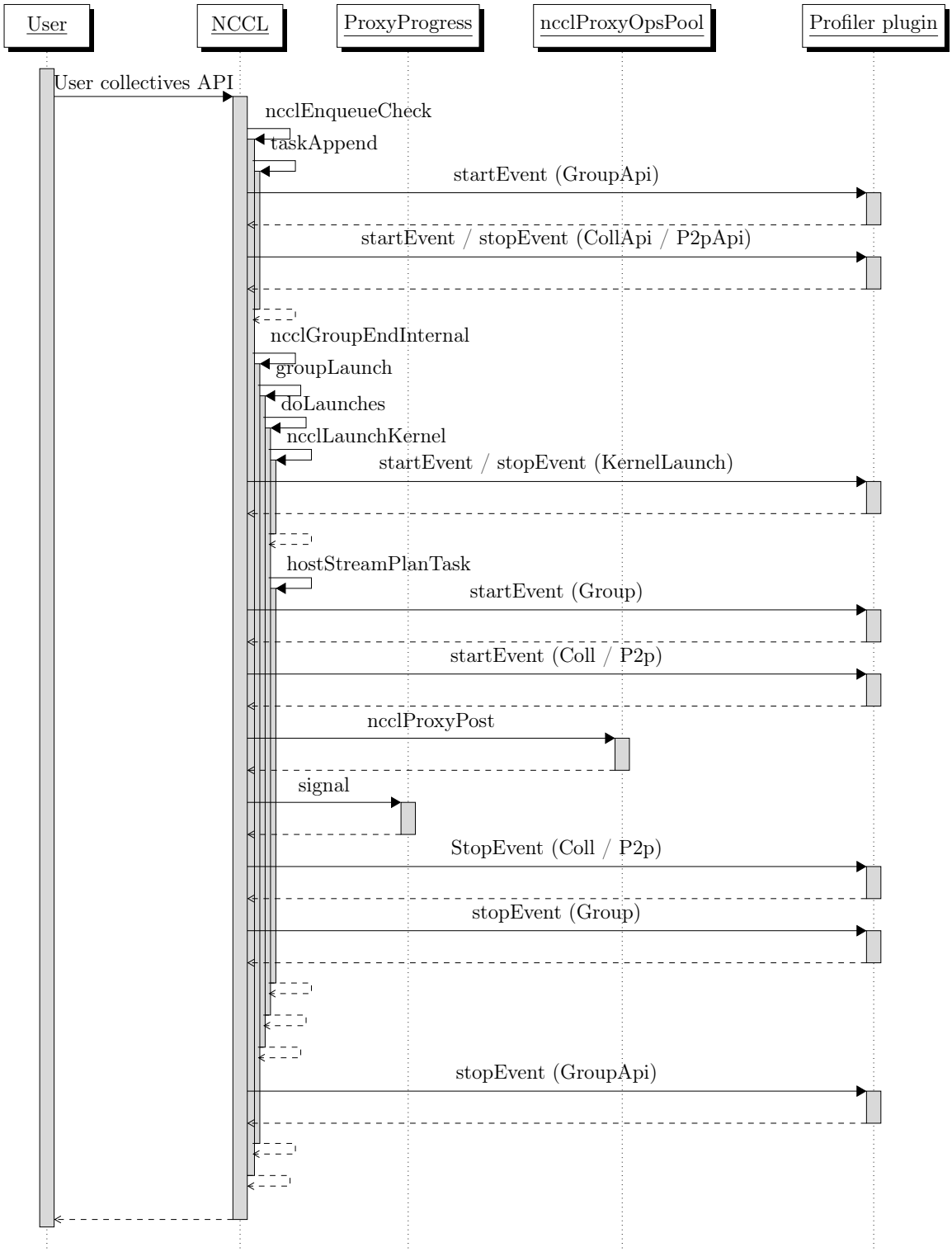

Figure 5: Flow from NCCL API calls to profiler events. In case of `ncclGroupStart / ncclGroupEnd`. multiple events of everything (except GroupApi) are called. internally, some Collectives (e.g. ncclAlltoAll) are implemented as many p2p ops, triggering many P2pApi and P2p events. Implementation: **/src/init.cc**, **/src/plugin/profiler.cc**.

Figure 6: `ncclProxyProgress`: progressOps emits ProxyStep/KernelCh/NetPlugin events. getPostedOps emits ProxyOp events. Several events ProxyCtrl are also emitted

`op->progress()` progresses transport specific ops. this is implemented as a function pointer type (defined in /**src/include/proxy.h**). Confusingly the variable is called 'op', although its type is `ncclProxyArgs` and NOT `ncclProxyOp`.

```
typedef ncclResult_t (*proxyProgressFunc_t)(struct ncclProxyState*, struct ncclProxyArgs
    *);

struct ncclProxyArgs {
  proxyProgressFunc_t progress;
  struct ncclProxyArgs* next;
  /* other fields */
}
```

Which specific function this calls depends on the Op. This also decides which profiler events (ProxyStep, KernelCh or Network plugin specific) are started and stopped. The transport-specific progress functions are in /**src/transport/net.cc**, **coll_net.cc**, **p2p.cc**, **shm.cc**.

4.2.4 recordEventState

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

Some event types can be updated by NCCL through `recordEventState` (state and attributes). Supported states can be found under **/src/include/plugin/profiler/profiler_v{versionNum}.h**.

Called at the same sites as `startEvent`.

4.2.5 finalize

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

After a user API call to free resources associated with a communicator, `finalize()` is called. Afterwards, a reference counter tracks how many communicators are still being tracked by the profiler plugin. If it reaches 0, the plugin will be closed via `dlclose(handle)`. Figure 7 depicts the flow from user API call to `finalize()`. See implementation at **/src/init.cc**, **/src/plugin/profiler.cc**, **/src/plugin/plugin_open.cc**.

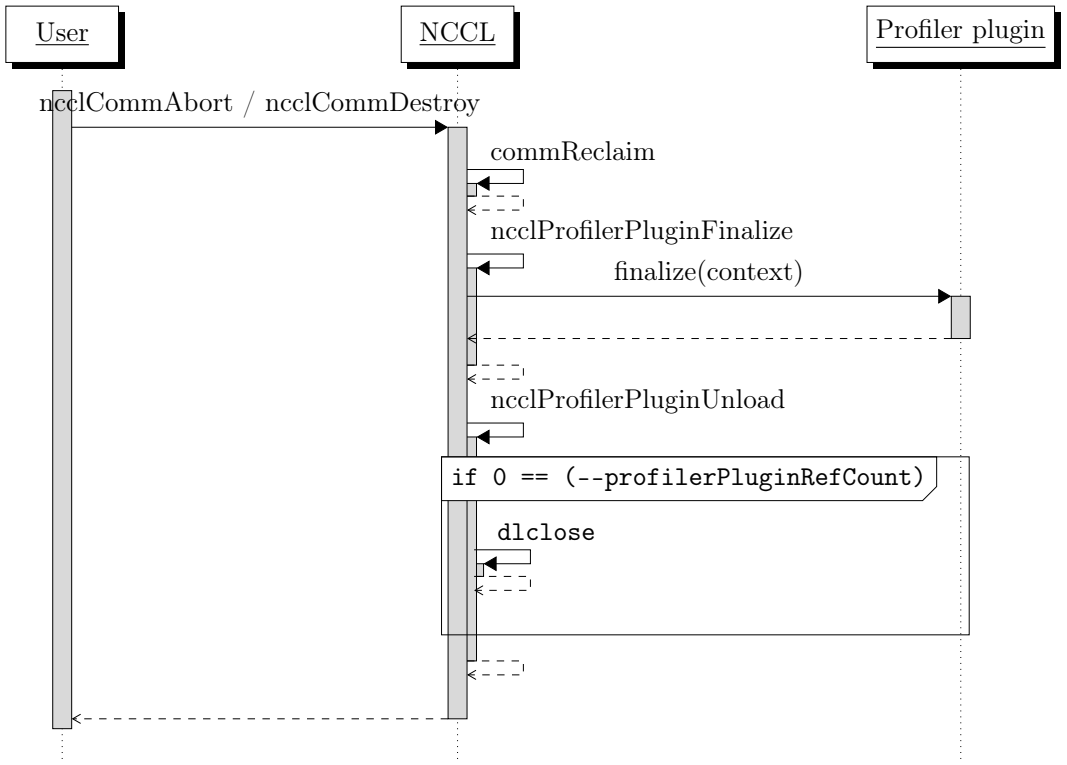

Figure 7: User API → `commReclaim()` → `finalize()` → plugin unload.

4.2.6 name

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

4.3 Code Examples

The following examples illustrate the profiling behavior under different environment settings and user application.

- single process multiple devices
- mpi multiple processes with multiple device per process (multiple threads per process)
- mpi multiple processes with multiple device per process (single thread per process, utilizing ncclGroup)
- behavior when utilizing ncclGroup for collective operations

TODO add exampless and pictures (profiler that just does logging + application examples)

An example implementation of a profiler plugin may just log all call information to a file. A simplified view of the plugin code used throughout the examples is as follows:

```
struct MyContext { /* custom context struct */ };
struct MyEvent { /* custom event struct */ };

MyEvent* allocEvent(args) { /* handles event allocation */ }
uint64_t getTime() { /* gets time */ }
void writeJsonl() { /* writes call details to log file as structured jsonl */ }

ncclResult_t myInit( /* args - *context, *eActivationMask, ... */ ) {
  *context = malloc(sizeof(struct MyContext));
  *eActivationMask = 4095; /* enable ALL event types */

  writeJsonl(getTime(), args);
  return ncclSuccess;
}

ncclResult_t myStartEvent( /* args - *eHandle, ... */ ) {
  *event = allocEvent(args);
  *eHandle = event;

  writeJsonl(getTime(), args);
  return ncclSuccess;
}

ncclResult_t myStopEvent(void* eHandle) {
  writeJsonl(getTime(), args);
```

```
  return ncclSuccess;
}

ncclResult_t myRecordEventState( /* args - ... */ ) {
  writeJsonl(getTime(), args);
  return ncclSuccess;
}

ncclResult_t myFinalize(void* context) {
  writeJsonl(getTime(), args);

  free(context);
  return ncclSuccess;
}

ncclProfiler_v5_t ncclProfiler_v5 = {
  "MyProfilerPlugin",
  myInit,
  myStartEvent,
  myStopEvent,
  myRecordEventState,
  myFinalize,
};
```

4.3.1 Single Process Multiple Devices

In this example, there is a single task running on one node with multiple GPUs. The example application for this can be found under /**examples/03_collectives/01_allreduce**/. In essence, it does the following:

- Detect Available GPUs
- Allocate Memory for Communicators, Streams, and Data Buffers
- Initialize NCCL Communicators for All Devices
- Create CUDA Streams and Allocate Device Memory
- Perform AllReduce Sum Operation
- Cleanup Resources and Report Results

The logs written by the profiler plugin can be visualized, creating following output:

TODO run this example (or a modified version with one additional warmup allreduce) on HPC 2761641

TODO organize and save outputs locally

TODO visualize output

TODO add screenshot here

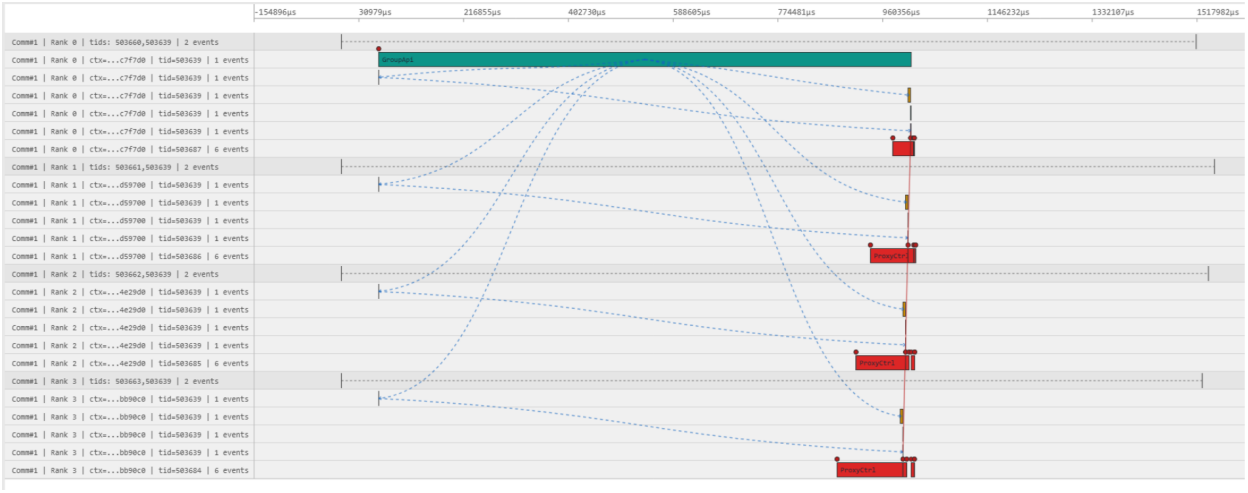

Figure 8: Single-GPU single-process-multiple-devices with ncclGroup trace: This visualization shows a view on an AllReduce collective, where all ranks are managed by a single thread on the same process.

TODO caption

Figure 9: Single-GPU single-process-multiple-devices with ncclGroup trace: Zoomed in TODO

TODO caption

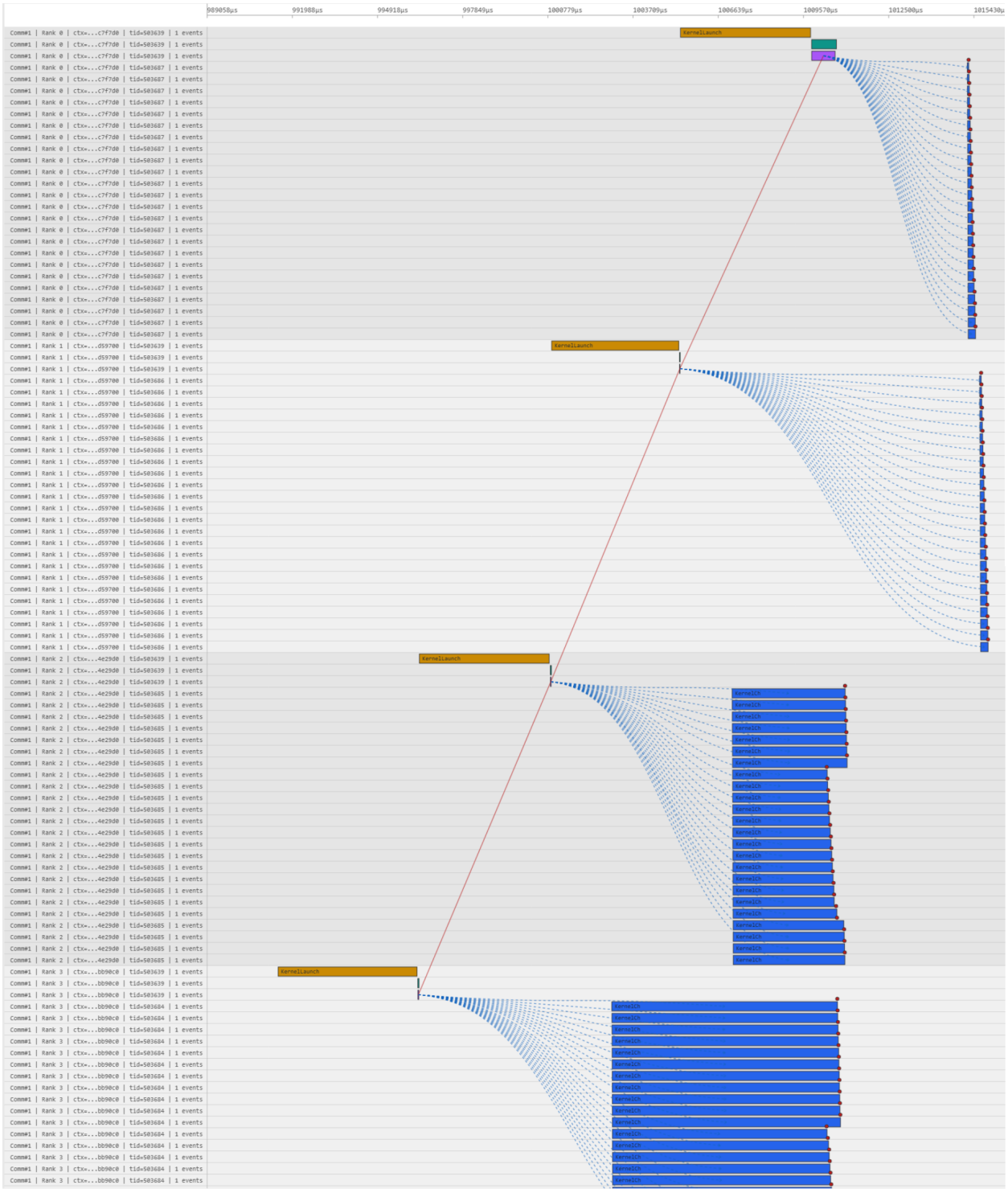

Figure 10: Single-GPU single-process-multiple-devices with ncclGroup trace: Zoomed in TODO

TODO explain what we see

4.3.2 Multiple Processes, Multiple Devices per Process, Multiple Threads per Process

TODO explain "@multi gpu single thread" code

TODO run this code (or a modified version with one additional warmup allreduce) on HPC

TODO organize and save outputs locally

TODO visualize output

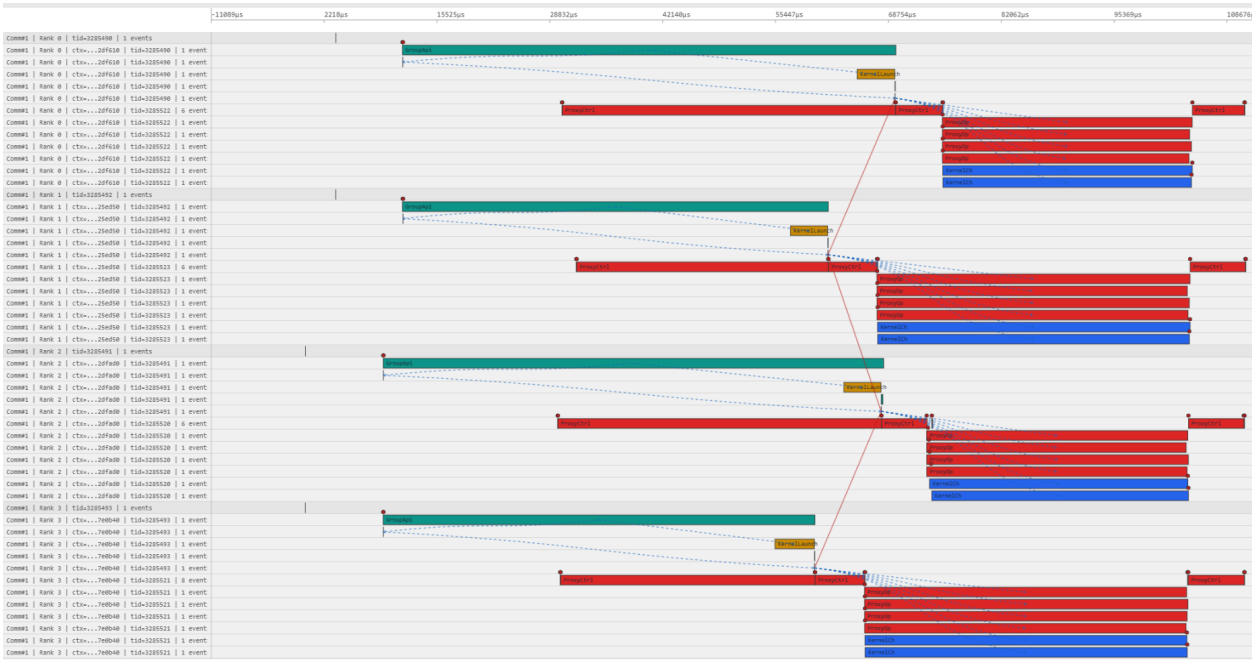

Figure 11: Multi-GPU single-thread-per-device trace: This visualization shows the AllReduce collective being called on 4 different ranks. a single thread corresponds to each rank. First a small bar (row above the green bar), corresponds with the timestamp when the profiler `init()` function was called for each rank. since init is called during NCCL's internal communicator creation, this corresponds to roughly when the application called `ncclCommInitRank`. Afterwards the application called an `ncclAllReduce`. This is visible in the profiler output as when the green bar (`groupApi` event) starts. Just below is another tiny bar, which represents the start and stop of the `collApi` event. The yellow bar represent the timing where NCCL enqueued the kernel for launch on the GPU (`KernelLaunch` event). Two small bars below represnt the `group` and `coll` events. Besides these 4 threads, NCCL spawned a proxy progress thread for each rank as well. The red `ProxyCtrl` event in the row below until that point was indicating that the proxy progress thread was asleep. the new ProxyCtrl event right after is the time it took for the Proxy Progress thread to append proxy ops. Next, multiple proxy ops are starting to be progressed (`ProxyOps` events), which in `op->progress()` leads to starting `KernelCh` network activity. At some point the KernelCh acitvity is completed, the AllReduce collective is finished. The ProxyCtrl event thereafter indicates the proxy progress thread went back to sleep. It is possible to visualize which `coll` events across ranks belong to the same collective operation. This is indicated by the red line connecting the events, enabled by the equal valued `seqNum` field (provided in `eDescr` arg in profiler API `init()` function call) across ranks.

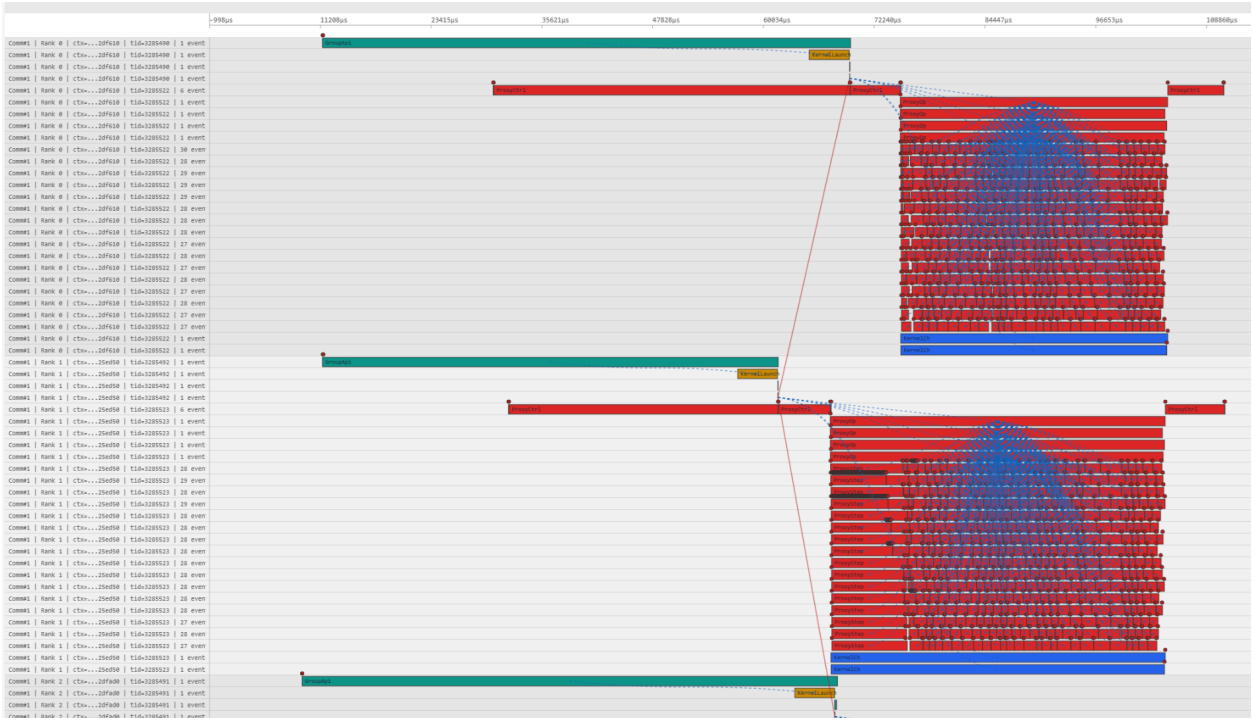

Figure 12: Multi-GPU single-thread-per-device trace: additionally showing proxy step events.

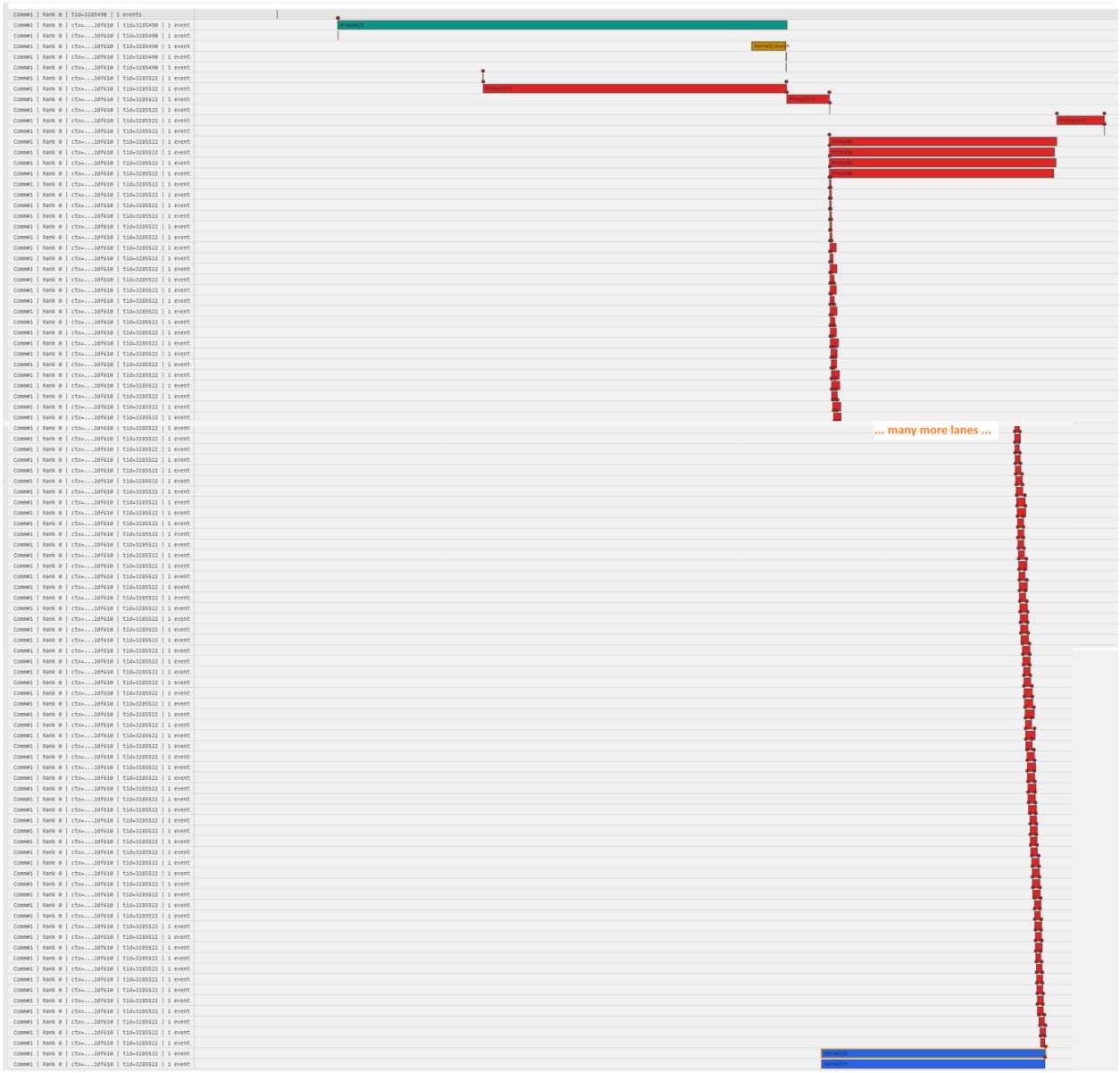

Figure 13: Multi-GPU single-thread-per-device trace: proxy step events, but always on new lane chronologically, no arrows.

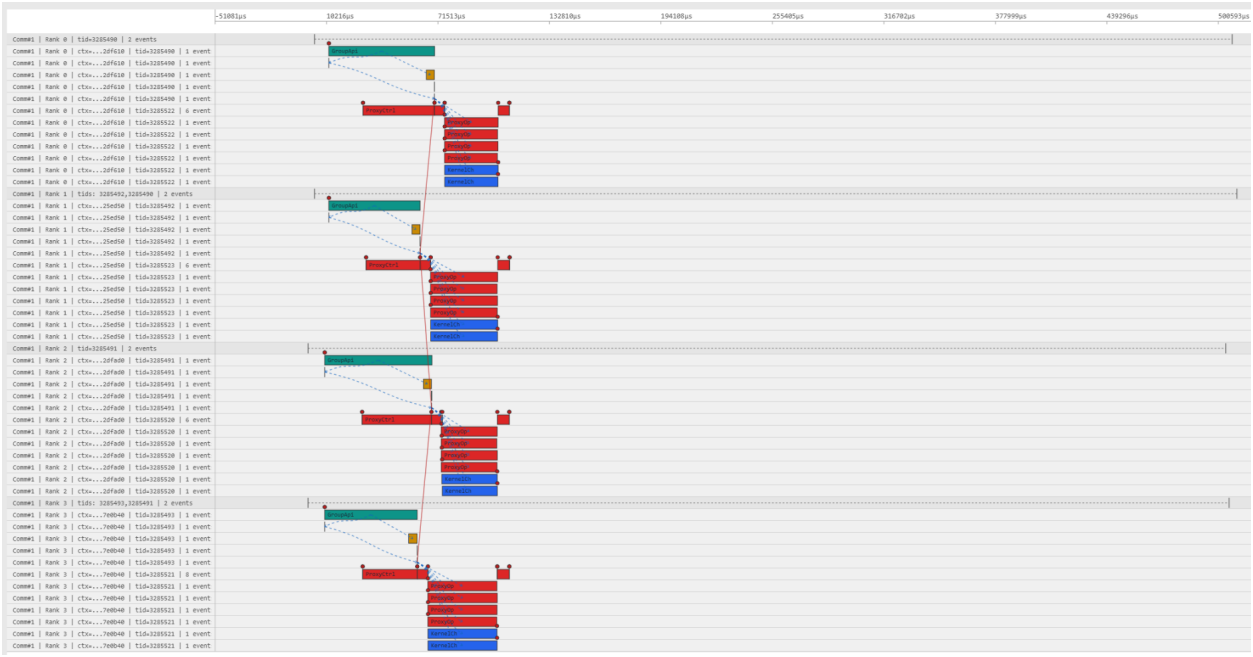

Figure 14: Multi-GPU single-thread-per-device trace: from `init` call until `finalize` call (a small bar to the very right in the slightly darker top row).

TODO explain what we see

4.3.3 Multiple Processes, Multiple Devices per Process, Single Threads per Process & using ncclGroup

skip this

TODO probably just need one of the multi process multi gpu per process examples. which one is better depends on whether 03_collectives/01_allreduce actually is single thread or multi threaded commInitAll is single threaded using ncclGroup

4.3.4 behavior when utilizing ncclGroup for collective operations

TODO explain node environment setup

TODO add selective screenshots of interesting things,

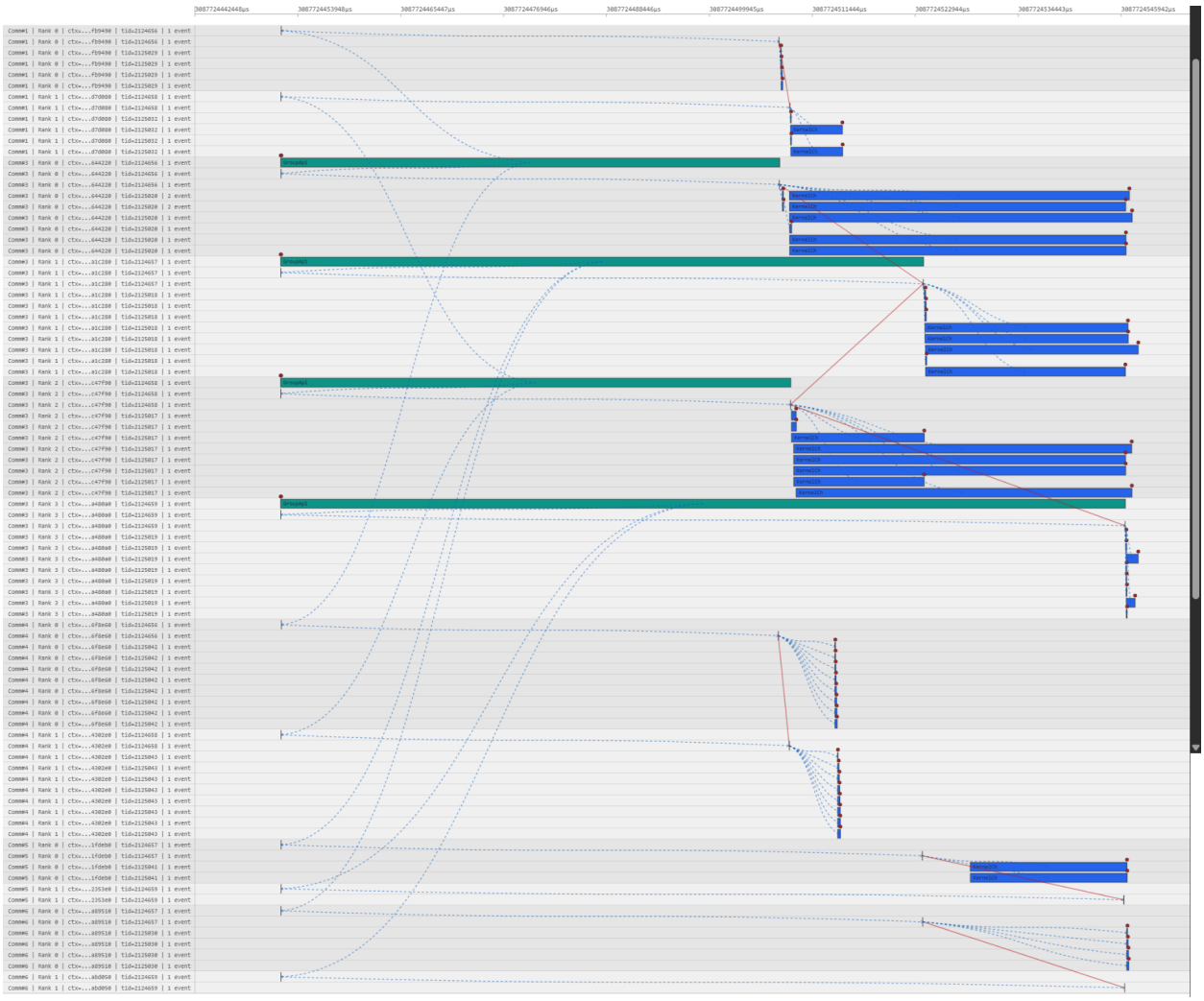

Figure 15: Multi-GPU multiple communicators per process with grouped collectives: here 6 different communicator cliques across 4 processes are shown. their collective operations are grouped together with `ncclGroupStart` and `ncclGroupEnd`. Only a single `GroupApi` event happens per process.

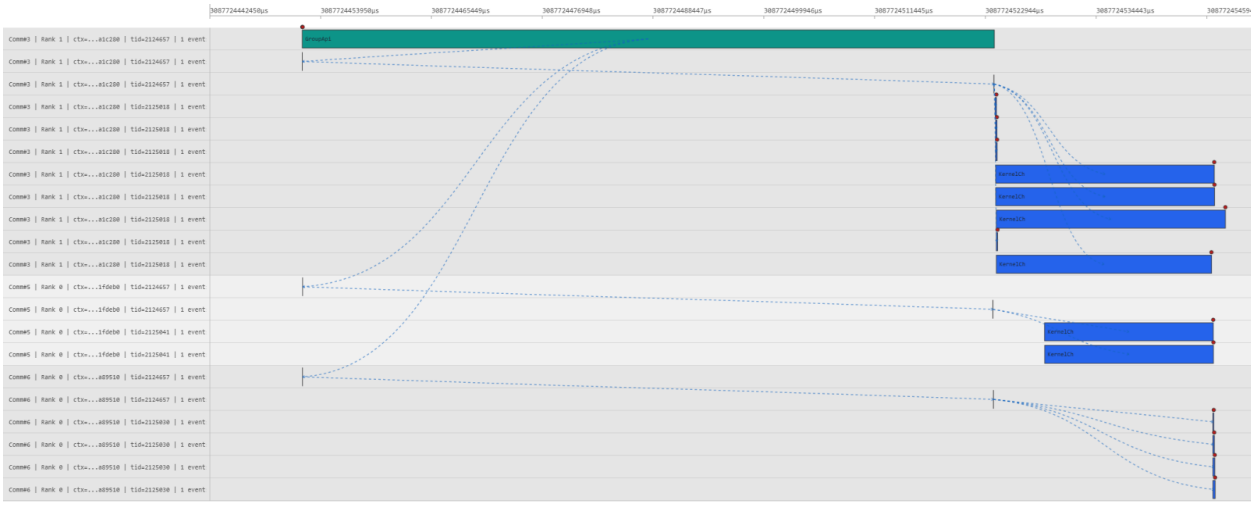

Figure 16: Multi-GPU multiple communicators per process with grouped collectives: View of just a single process. The `CollApi` events at the start show that they have the same GroupApi event as `parentObj`. However each rank still emits their own `KernelLaunch` event.

like some ncclGroup stuff, multi level split intereweaved stuff

TODO explain what we see

TODO go back up and explain the important "@stress test" code. and show snippets and/or summarize (like for 1st simple example)

TODO if not run on HPC in useful format yet, run this code on HPC

TODO organize and save outputs locally

4.4 What is possible with the Profiler Plugin API? Considerations and Pitfalls for (logging, running metrics, CUPTI, ...) when writing the plugin - section title WIP

TODO Logging. is this even worth talking about?

- Logging function from `init` (TODO)
- Code snippet: custom logging infrastructure, timestamping

TODO Tracking & running metrics

> Due to the asynchronous nature of NCCL operations, events associated with collectives and point-to-point are not easy to delimit precisely. `stopEvent` for collectives simply indicates to the profiler that the collective has been enqueued. Without proxy and/or

kernel activity the plugin cannot determine when a collective ends. With proxy/kernel events enabled, the plugin can estimate when it ends.

(slightly rephrased from **/ext-profiler/README.md**)

void* parentObj

if the plugin developer wants utilize this field, they should ensure that potential address reuse does not create ambiguity to what the parentObj was originally pointing to. Custom memory management is advised. This field is useful when trying to understand which user API call triggered which events of lower level operations or activity such as network activity.

TODO add picture

seqNumber

When profiling is enabled, NCCL counts the number of calls for each type of collective function per communicator.

/src/include/comm.h

```
struct ncclComm {
  uint64_t seqNumber[NCCL_NUM_FUNCTIONS];
  /* other fields */
}
```

/src/plugin/profiler.cc

```
ncclResult_t ncclProfilerStartTaskEvents(struct ncclKernelPlan* plan) {
  /* other code */
  __atomic_fetch_add(&plan->comm->seqNumber[ct->func], 1, __ATOMIC_RELAXED);
  /* other code */
}
```

This value is present in the eDescr for collective events and can be used to identify which collectives operations belong together across processes.

TODO add picture of collectives seqNum

PXN

Unless Setting the environment variable `NCCL_PXN_DISABLE`=0 (default 1), due to PXN (PCIe x NVLink) some proxy ops may be progressed in a proxy thread from another process, different to the one that originally generated the operation Then `parentObj` in `eDescr` is not safe to dereference; the `eDescr` for `ProxyOp` events includes the originator's PID, which the profiler can match against the local PID. The `eDescr` for `ProxyStep` does not provide this field. However a workaround is possible:

The passed `context` object in `startEvent()` is also unsafe to dereference due to PXN. the profiler plugin developer may internally track initialized contexts and whether the passed `context` belongs to the local process. This is also indicative of PXN.

TODO: fact check if context is actually unsafe to deref? from code it looks like it is (context is read from pool - which iirc is posted to by potentially the originating process, not the progressing proxy thread process?) looking at inspector plugin implementation, it casts the passed context in startEvent(), but doesnt dereference it. so it doesnt crash. looking at example plugin implementation, it dereferences it in startEvent(), which is why it crashes.

TODO verify this behavior one more time: slurm 2732796

TODO

- Code snippet: example CRUD of custom context object
- Code snippet: example CRUD of custom event object

TODO Kernel tracing with CUPTI

- CUPTI extension ID mechanism briefly explained
- Code snippet: where to CUPTI init/cleanup and usage

TODO Changing profiling behaviour at runtime (TODO: check example_profiler, inspector)

TODO commId hierarchical behavior. not so important. for split and shrink with ncclCommInitRankConfig

4.5 Performance and scalability of the Profiler Plugin API

Following Experiments were run to assess the performance and scalability of profiler plugins.

TODO

- synthetic performance measuring applications
- real application

TODO check out potential use of

```
@profiler (line 120 - 182)
#define ENABLE_TIMER 0
#include "timer.h"

#if ENABLE_TIMER
// These counters are used to measure profiler overheads for different part of the code
// These counters are only useful/meaningful in controlled test environments where there
// is only one thread updating each set of counters, i.e., every communicator has its
// own proxy thread and the network uses only one thread to make progress (this is true
// for net_ib plugin but might not be true for net_socket plugin).
...
```

setup

empty profiler: initializes dummy context struct, returns NULL for event handles, tracks all events 4095 (including those for network activity)

```
// an 'empty' NCCL Profiler Plugin

struct MyContext {
  char dummy;
};

ncclResult_t myInit(void** context, uint64_t commId, int* eActivationMask, const char*
    commName, int nNodes, int nranks, int rank, ncclDebugLogger_t logfn) {
  *context = malloc(sizeof(struct MyContext));
  *eActivationMask = 4095; /* enable ALL event types */
  return ncclSuccess;
}

ncclResult_t myStartEvent(void* context, void** eHandle, ncclProfilerEventDescr_v5_t*
    eDescr) {
  *eHandle = NULL;
  return ncclSuccess;
}

ncclResult_t myStopEvent(void* eHandle) {
  return ncclSuccess;
}

ncclResult_t myRecordEventState(void* eHandle, ncclProfilerEventState_v5_t eState,
    ncclProfilerEventStateArgs_v5_t* eStateArgs) {
  return ncclSuccess;
}

ncclResult_t myFinalize(void* context) {
  free(context);
  return ncclSuccess;
}

ncclProfiler_v5_t ncclProfiler_v5 = {
  "EmptyProfiler",
  myInit,
  myStartEvent,
  myStopEvent,
  myRecordEventState,
  myFinalize,
};
```

application:

TODO run mainly nccl-tests for synthetic. easily accessible so easier to reproduce

synthetic - TODO add application code in an appendix section or something, if there are some custom apps that are worthy but missing in nccl-tests

realistic - TODO maxtext zum laufen bringen

singlenode

- coll singlenode (1mil iters, 100 warmup iters):
  - sbatch run 1: 7.64 µs overhead. averaged 26.40 µs per iter w/o. 34.04 µs w/ profiler
- p2p singlenode (1mil iters 100 warmup iters):
  - sbatch run 1: 7.26 µs overhead. averaged 26.66 µs per iter w/o. 33.92 µs w/ profiler
- commops singlenode (100 iters, 10 warmup iters):
  - sbatch run 1: check slurm ...

multinode

- coll multinode (1mil iters, 100 warmup iters):
  - sbatch run 1: 0.33 µs overhead. averaged 22.12 µs per iter w/o. 22.45 µs w/ profiler
- p2p multinode (1mil iters 100 warmup iters):
  - sbatch run 1: not measurable. averaged 21.17 µs per iter w/o. 20.82 µs w/ profiler
- commops multinode (100 iters, 10 warmup iters):
  - sbatch run 1: "init, destroy" experiment
    - 1790.05 µs overhead. averaged 577455.37 µs per iter w/o. 579245.42 µs w/ profiler
  - sbatch run 1: "init, split, destroy" experiment
    - 1546.16 µs overhead. averaged 1079679.93 µs per iter w/o. 1078133.77 µs w/ profiler

TODO run experiments multiple times?

TODO accuracy and reliability/consistency of the the timings of profiler api calls from nccl

Using the profiler plugin when scaled to many gpus across multiple nodes is effortless and did not require any changes for the ran examples and experiments.

5 Potential Integration with Score-P

TODO Substrate Plugin Route

TODO Metric Plugin Route?

6 Conclusion - What i have shown. Why would you use it? pros & cons

- Customizable
- May require maintenance / active development since NCCL is actively developed
- Low overhead: NVIDIA advertises their `inspector` implementation as efficient enough for "always-on" in production

6.1 NCCL_DEBUG

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

NCCL already comes with debug logging at various levels of granularity:

- INFO – debug information
- TRACE – replayable trace information on every call
- Further options (v2.2.12 `NCCL_DEBUG_FILE`, v2.3.4 `NCCL_DEBUG_SUBSYS`, v2.26 timestamp format/levels)
- other profiling and tracing tools exists that are maintained by NVIDIA: nsight systems, nsight compute

6.2 Known limitations

> Kernel event instrumentation uses counters exposed by the kernel to the host and the proxy progress thread. Thus the proxy progress thread infrastructure is shared between network and profiler. If the proxy is serving network requests, reading kernel profiling data can be delayed, causing loss of accuracy. Similarly, under heavy CPU load and delayed scheduling of the proxy progress thread, accuracy can be lost.
>
> From profiler version 4, NCCL uses a per-channel ring buffer of 64 elements. Each counter is complemented by two timestamps (ptimers) supplied by the NCCL kernel (start and stop of the operation in the kernel). NCCL propagates these timestamps to the profiler plugin so it can convert them to the CPU time domain.

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

6.3 Summarize What i have shown TODO

TODO

7 TODO

- custom profiler code cleaning