

1 NCCL Profiler Plugin API – A Feasibility Study

2

3 **Contents**

4	1 Abstract	2
5	2 Introduction to NCCL	2
6	2.1 Comparison to MPI	3
7	2.2 Relevant NCCL internals	3
8	3 Profiler Plugin	6
9	3.1 Profiler plugin loading mechanism	6
10	3.2 Profiler API	9
11	3.2.1 init	9
12	3.2.2 startEvent	10
13	3.2.3 stopEvent	11
14	3.2.4 recordEventState	15
15	3.2.5 finalize	15
16	3.2.6 name	16
17	4 Code examples and visualizations	16
18	4.1 One Device per Thread	18
19	4.1.1 Multiple Devices per Thread (ncclGroup)	20
20	4.1.2 Aggregated operations	22
21	5 Performance and scalability of the Profiler Plugin API	23

²²	6 Discussion	25
²³	6.1 Considerations for developers of a Profiler Plugin	25
²⁴	6.2 Known limitations	27
²⁵	6.3 Potential Integration with Score-P	27
²⁶	7 Conclusion	28

²⁷ 1 Abstract

- ²⁸ 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.
- ³⁸ First, this study provides background information on NCCL, followed by an explanation of the Profiler API is explained accompanied with code examples and visualizations. Next, considerations for developers of the Profiler API and its potential integration with Score-P is discussed. Finally, the study concludes with a summary of the findings.

⁴² 2 Introduction to NCCL

- ⁴³ NCCL was first introduced by NVIDIA in 2015 at the Supercomputing Conference¹ with code being made available on GitHub². The release of NCCL 2.0 in 2017 brought support for NVLink, however this 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.
- ⁴⁸ Before taking a closer look at the Profiler Plugin API, it is helpful to have some rudimentary understanding on certain designs in NCCL.

¹<https://images.nvidia.com/events/sc15/pdfs/NCCL-Woolley.pdf>

²<https://github.com/NVIDIA/nccl>

50 **2.1 Comparison to MPI**

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

- 53 • **MPI:** Communication is CPU-based. A rank corresponds to a single CPU process within a
54 communicator.
- 55 • **NCCL:** Communication is GPU-based, with CPU threads handling orchestration. A rank
56 corresponds to a GPU device within a communicator; the mapping from ranks to devices
57 is surjective. A single CPU thread can manage multiple ranks (i.e., multiple devices) in a
58 communicator using the functions `ncclGroupStart` and `ncclGroupEnd`. A CPU thread can
59 also manage multiple ranks from different communicators (i.e same device allotted by multiple
60 ranks from different communicators) through communicator creation with `ncclCommSplit` or
61 `ncclCommShrink`. This means the mapping from ranks to threads is also surjective.

62 **2.2 Relevant NCCL internals**

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

64 A typical NCCL application follows this basic structure:

- 65 • create nccl communicators
- 66 • allocate memory for computation and communication
- 67 • do computation and communication
- 68 • clean up nccl communicators

69 During NCCL communicator creation, NCCL internally spawns a thread called `ProxyService`. This
70 thread lazily starts another thread called `ProxyProgress`³, which handles network requests for GPU
71 communication during collective and P2P operations. See Fig. 1.

72 if-guards ensure that these threads are created once per `ncclSharedResources`⁴. By default every
73 NCCL communicator has its own shared resource. When the application calls `ncclCommSplit` or
74 `ncclCommShrink`, where the original communicator was initialized with a
75 `ncclConfig_t` with fields `splitShare` or `shrinkShare` set to 1, the newly created communicator
76 uses the same shared resource (and the proxy threads) as the parent communicator.

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

79 The ProxyProgress thread reads from this pool when calling `ncclProxyGetPostedOps` and pro-
80 gresses the ops. See Fig. 3.

³<https://github.com/NVIDIA/nccl/tree/master/src/proxy.cc>

⁴<https://github.com/NVIDIA/nccl/tree/master/src/include/comm.h>

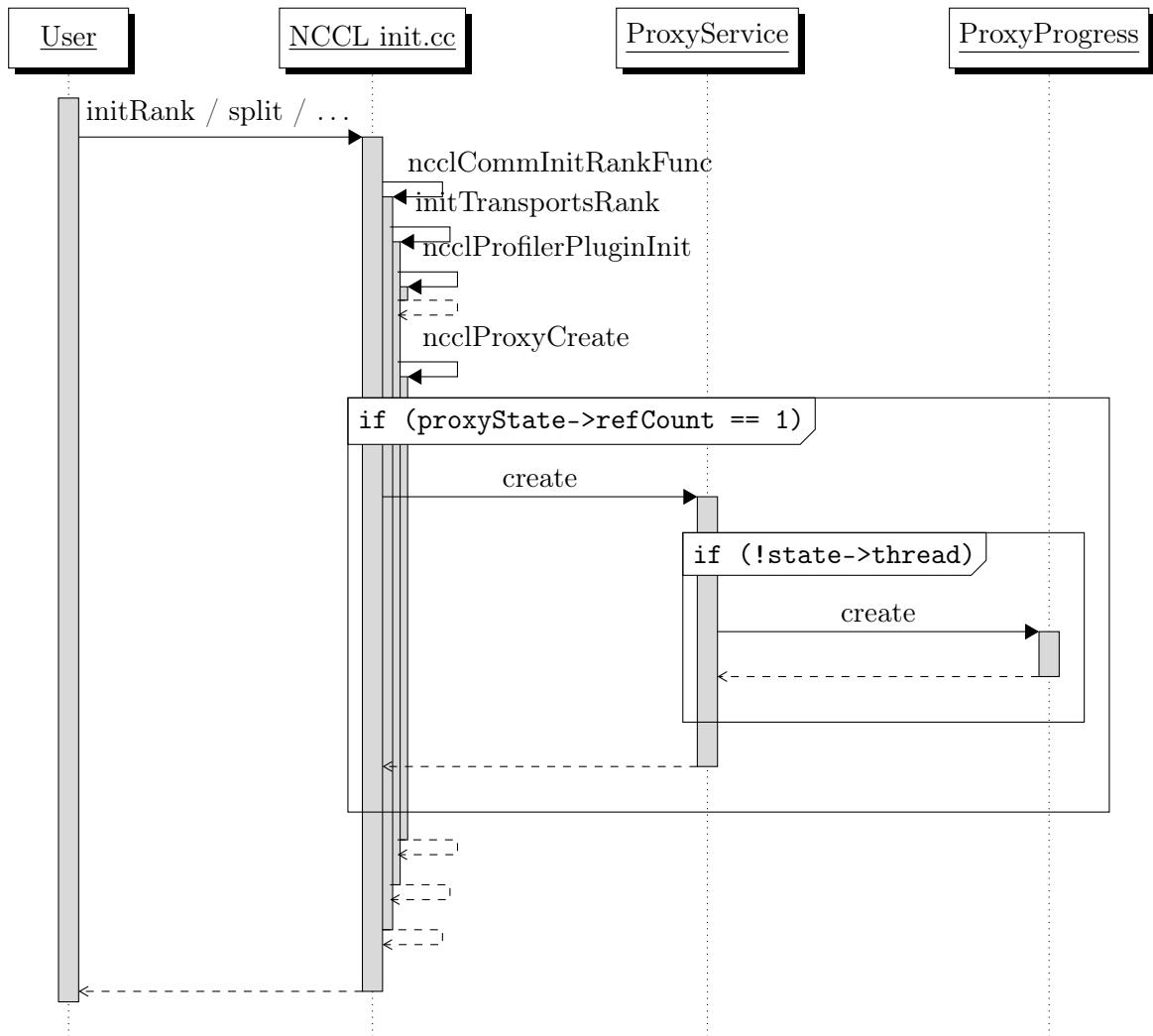


Figure 1: Thread creation: User API → NCCL internal init → create ProxyService → create ProxyProgress.

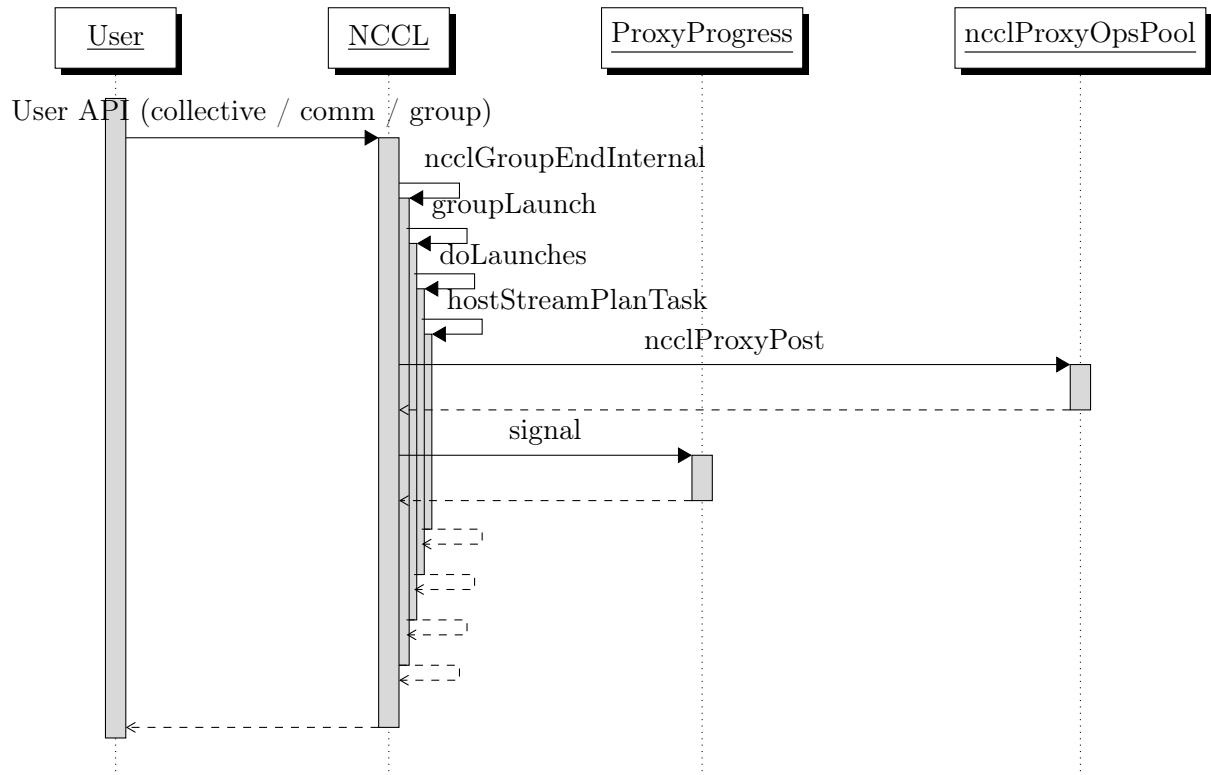


Figure 2: Flow from User API to `ncclProxyPost`

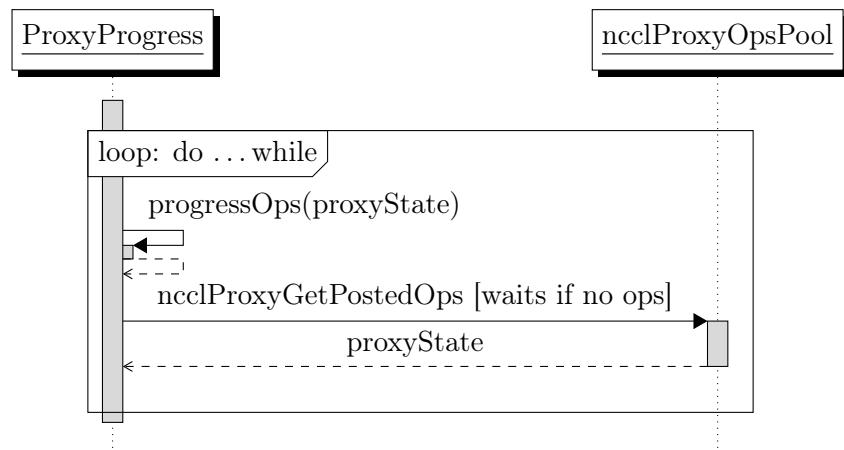


Figure 3: `/src/proxy.cc` `ncclProxyProgress` progressing loop: progress ops, then get posted ops (or wait).

81 Familiarity with this network activity pattern will aid in understanding the Profiler Plugin API's
82 behavior discussed in the following section.

83 3 Profiler Plugin

84 Whenever a communicator is created, NCCL looks for the existence of a profiler plugin and loads
85 it if it has not already been loaded on the process. NCCL then initializes the plugin with the
86 created communicator. Whenever the application makes calls to the Collectives or P2p API (e.g.
87 `ncclAllReduce`) with that communicator, NCCL calls the profiler API in different regions of the
88 internal code. When the communicator is destroyed, the profiler plugin is unloaded if this was the
89 only communicator on the process.

90 3.1 Profiler plugin loading mechanism

91 Each time a NCCL communicator is created, `ncclProfilerPluginLoad`⁵ is called, where NCCL
92 looks for a shared library that represents the profiler plugin by checking an environment variable.
93 It then calls `dlopen`⁶ and `dlsym` to load the library immediately with local symbol visibility:

```
94 profilerName = ncclGetEnv("NCCL_PROFILER_PLUGIN");
95 // ...
96 handle* = dlopen(name, RTLD_NOW | RTLD_LOCAL);
97 // ...
98 ncclProfiler_v5 = (ncclProfiler_v5_t*)dlsym(handle, "ncclProfiler_v5");
99
100
```

101 If the library has already been loaded on the process, this procedure is skipped.
102 A `profilerPluginRefCount` keeps track of the number of calls to this procedure to ensure correct
103 unloading during finalization. See Fig. 4. The NCCL documentation⁷ also describes some further
104 loading logic:

- 105 • 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`.
- 106 • If `NCCL_PROFILER_PLUGIN` is not set: attempt `libnccl-profiler.so`.
- 107 • If no plugin was found: profiling is disabled.
- 108 • If `NCCL_PROFILER_PLUGIN` is set to `STATIC_PLUGIN`, the plugin symbols are searched
in the program binary.

111 The plugin loading mechanism expects the struct variable name to follow the naming convention
112 `ncclProfiler_v{versionNum}`, which also indicates the API version.

113 The profiler API has changed multiple times with newer NCCL releases. NCCL features a fallback
114 mechanism to load older struct versions. However one instance is known, where a profiler plugin

⁵<https://github.com/NVIDIA/nccl/tree/master/src/plugin/profiler.cc>

⁶https://github.com/NVIDIA/nccl/tree/master/src/plugin/plugin_open.cc

⁷<https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/env.html#nccl-profiler-plugin>

115 being developed against the NCCL release 2.25.1 with Profiler API version 2, was unable to run with
116 the latest NCCL release⁸. Around this time, the NCCL repository has undergone a refactor related
117 to the profiler plugin. NCCL developers may be required to actively maintain older API versions,
118 to ensure they safely work when old behaviour is getting deprecated and do not unexpectedly get
119 handed new features from new API versions, of which the an older Profiler Plugin is not aware of
120 (when faithfully implementing the old API version).

⁸<https://github.com/variemai/ncclsee>



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

121 **3.2 Profiler API**

122 The plugin must implement a profiler API specified by NCCL by exposing a struct⁹. This struct
123 should contain pointers to all functions required by the API. A plugin may expose multiple versioned
124 structs for backwards compatibility with older NCCL versions.

```
125 ncclProfiler_v5_t ncclProfiler_v5 = {  
126     const char* name;  
127     ncclResult_t (*init)(...); // called when a communicator is created  
128     ncclResult_t (*startEvent)(...); // at start of operations/activities  
129     ncclResult_t (*stopEvent)(...); // at end of these operations/activities  
130     ncclResult_t (*recordEventState)(...); // to record state of certain operations  
131     ncclResult_t (*finalize)(...); // called when a communicator is destroyed  
132 };  
133 }
```

135 As of NCCL v2.29.2, version 6 is the latest, which was released on Dec 24, 2025. This release
136 happened well after the begin of this report, so the focus will be on version 5. Version 6 introduced
137 additional profiler API callbacks for Copy-Engine based collective operations, otherwise version 6
138 and version 5 remain the same.

139 Five functions must be implemented for the API. Internally NCCL wraps calls to the profiler API
140 in custom functions which are all declared in a single file¹⁰.

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

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

145 **3.2.1 init**

146 `init` initializes the profiler plugin with a communicator. `init` is called immediately after
147 `ncclProfilerPluginLoad`, which happens every time a communicator is created (see Fig. 4). This
148 may happen multiple times for the same profiler instance, if further communicators are created on
149 that process. NCCL passes following arguments:

```
150  
151 ncclResult_t init(  
152     void** context, // out param - opaque profiler context  
153     uint64_t commId, // communicator id  
154     int* eActivationMask, // out param - bitmask for which events are tracked  
155     const char* commName, // user assigned communicator name  
156     int nNodes, // number of nodes in communicator  
157     int nranks, // number of ranks in communicator  
158     int rank, // rank identifier in communicator  
159     ncclDebugLogger_t logfn // logger function  
160 );
```

⁹https://github.com/NVIDIA/nccl/tree/master/src/include/plugin/profiler/profiler_v5.h

¹⁰<https://github.com/NVIDIA/nccl/tree/master/src/include/profiler.h>

162 If the profiler plugin `init` function does not return `ncclSuccess`, NCCL disables the plugin.

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

166 The plugin developer should set `int* eActivationMask` to a bitmask¹¹, indicating which event
167 types the profiler wants to track:

```
168 enum {
169     ncclProfileGroup = (1 << 0), // group event type
170     ncclProfileColl = (1 << 1), // host collective call event type
171     ncclProfileP2p = (1 << 2), // host point-to-point call event type
172     ncclProfileProxyOp = (1 << 3), // proxy operation event type
173     ncclProfileProxyStep = (1 << 4), // proxy step event type
174     ncclProfileProxyCtrl = (1 << 5), // proxy control event type
175     ncclProfileKernelCh = (1 << 6), // kernel channel event type
176     ncclProfileNetPlugin = (1 << 7), // network plugin-defined, events
177     ncclProfileGroupApi = (1 << 8), // Group API events
178     ncclProfileCollApi = (1 << 9), // Collective API events
179     ncclProfileP2pApi = (1 << 10), // Point-to-Point API events
180     ncclProfileKernelLaunch = (1 << 11), // Kernel launch events
181 };
182 
```

184 The default value is to 0, which means no events are tracked by the profiler. Setting it to 4095 will
185 track all events.

186 `ncclDebugLogger_t logfn` is a function pointer to NCCL's internal debug logger (`ncclDebugLog`).
187 NCCL passes this so the plugin can emit log lines through the same channel and filtering as NCCL:
188 the plugin may store the callback and call it with `(level, flags, file, line, fmt, ...)` when
189 it wants to log. Messages then appear in NCCL's debug output (e.g. `stderr` or `NCCL_DEBUG_FILE`)
190 and respect the user's `NCCL_DEBUG` level and subsystem mask. Using `logfn` keeps profiler output
191 consistent with NCCL's own logs.

192 3.2.2 startEvent

193 `startEvent` is called when NCCL begins certain operations:

```
194
195 ncclResult_t startEvent(
196     void* context, // opaque profiler context object
197     void** eHandle, // out param - event handle
198     ncclProfilerEventDescr_v5_t* eDescr // pointer to event descriptor
199 );
200 
```

201 As of release v2.29.2 NCCL does not use the return value. `void** eHandle` may point to a custom
202 event object; this pointer is passed again in `stopEvent` and `recordEventState`. `eDescr`¹² describes
203 the started event.

¹¹https://github.com/NVIDIA/nccl/tree/master/src/include/plugin/nccl_profiler.h

¹²https://github.com/NVIDIA/nccl/tree/master/src/include/plugin/profiler_v5.h

- 204 The field `void* parentObj` in the event descriptor is the `eHandle` of a parent event (or null). The
 205 use of this field can be explained as following:
- 206 All User API calls to Collective or P2P operations will start a Group API event. When networking is
 207 required, ProxyCtrl Events may be emitted. Depending on the `eActivationMask` bitmask returned
 208 in the `init` function, further (child) events will be emitted in deeper regions of the nccl code base.
 209 It can be thought of as an event hierarchy¹³ with several depth levels:

```

210
211     Group API event
212     |
213     +- Collective API event
214     |   |
215     |   +- Collective event
216     |   |
217     |   +- ProxyOp event
218     |   |   |
219     |   |   +- ProxyStep event
220     |   |   |
221     |   |   +- NetPlugin event
222     |   |
223     |   +- KernelCh event
224
225     +- Point-to-point API event
226     |   |
227     |   +- Point-to-point event
228     |   |
229     |   +- ProxyOp event
230     |   |   |
231     |   |   +- ProxyStep event
232     |   |   |
233     |   |   +- NetPlugin event
234     |   |
235     |   +- KernelCh event
236     |
237     +- Kernel Launch event
238
239     ProxyCtrl event
  
```

- 241 The `parentObj` inside `eDescr` will be a reference to the `eHandle` of the respective parent event for
 242 the current event according to this hierarchy. Thus, if the `eActivationMask` set during `init` enables
 243 tracking for event types lower in the hierarchy, NCCL always also tracks their parent event types.

244 3.2.3 stopEvent

```

245
246     ncclResult_t stopEvent(void* eHandle); // handle to event object
  
```

- 248 `stopEvent` tells the plugin that the event has stopped. `stopEvent` for collectives simply indicates
 249 to the profiler that the collective has been enqueued and not that the collective has been completed.
- 250 As of NCCL v2.29.2 NCCL does not use the return value.

¹³<https://github.com/NVIDIA/nccl/tree/master/ext-profiler/README.md>

251 `stopEvent` is called in the same functions that call `startEvent`, except for the GroupApi event.
252 Fig. 5 shows when NCCL emits `startEvent` and `stopEvent` after a user API call. The Proxy-
253 Progress thread also emits `startEvent` and `stopEvent` while progressing ops (see Fig. 6).

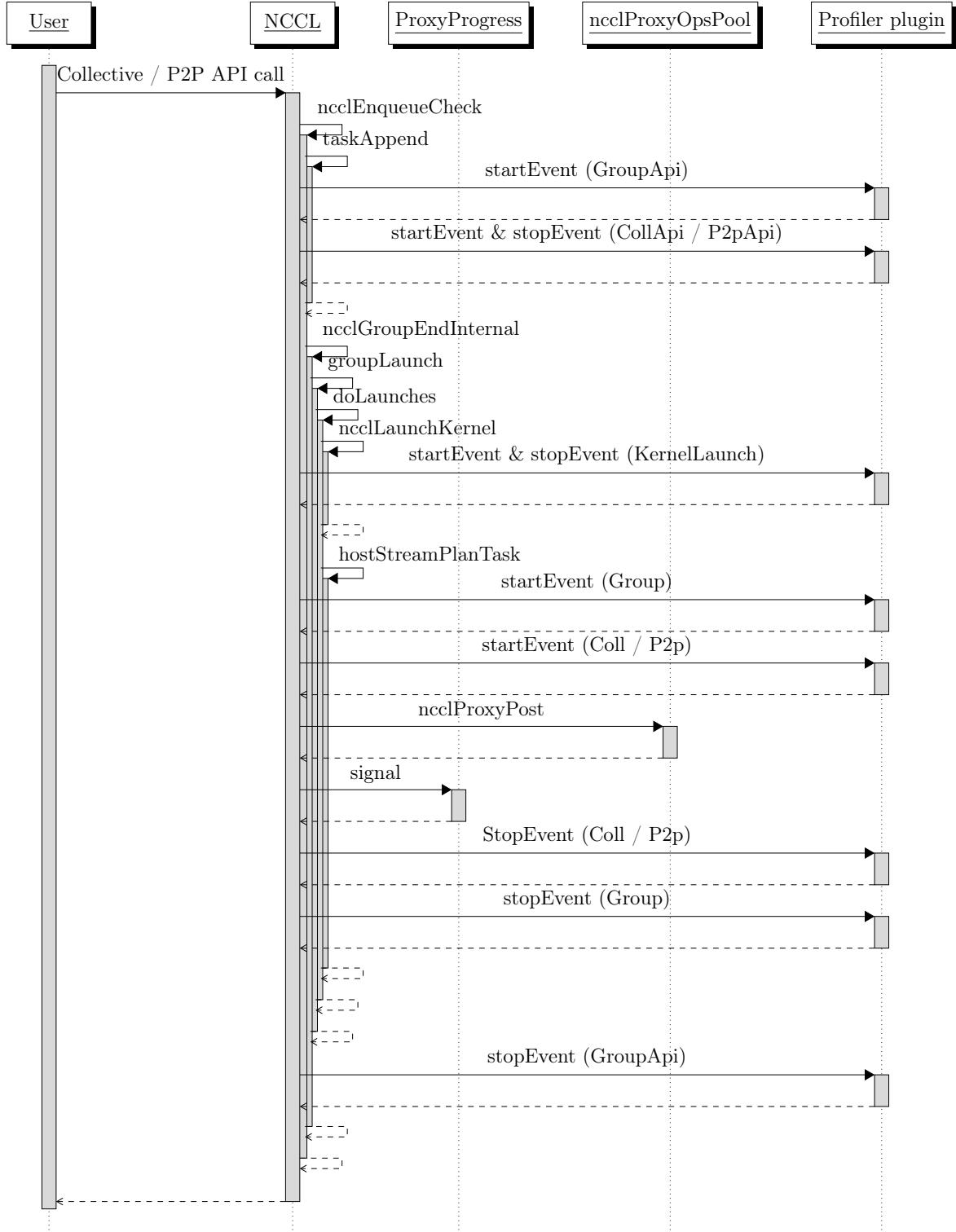


Figure 5: Flow from NCCL API calls to profiler events. In case of `ncclGroupStart` / `ncclGroupEnd`, multiple events of everything (except GroupApi) are called. Internally, some Collectives (e.g. `ncclAlltoAll`) are implemented as multiple p2p ops, triggering many P2pApi and P2p events.

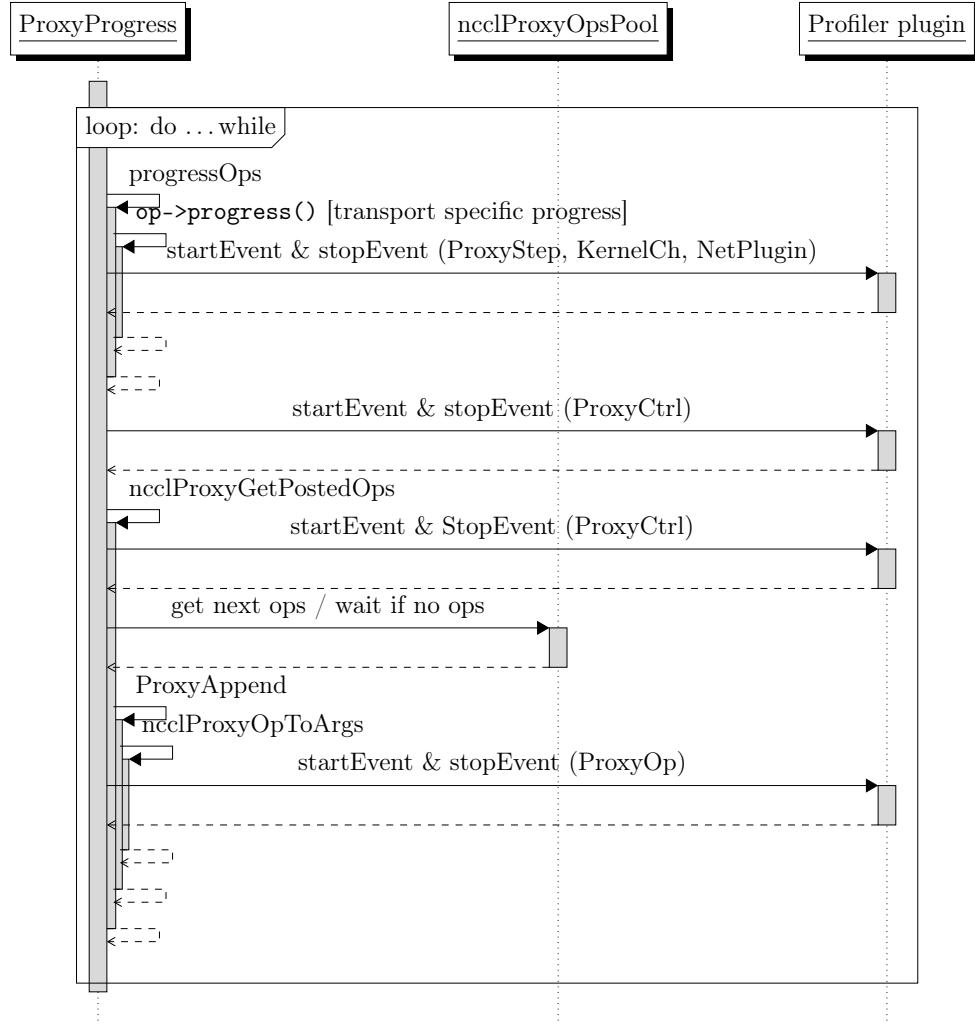


Figure 6: `ncclProxyProgress`: `progressOps` emits `ProxyStep`/`KernelCh`/`NetPlugin` events. `getPostedOps` emits `ProxyOp` events. Several events `ProxyCtrl` are also emitted

254 `op->progress()` progresses transport specific ops. This is implemented as a function pointer type¹⁴.
 255 Confusingly the variable is called ‘`op`’, although its type is `ncclProxyArgs` and *not* `ncclProxyOp`.

```

256
257 typedef ncclResult_t (*proxyProgressFunc_t)(struct ncclProxyState*, struct ncclProxyArgs
258   *);
259
260 struct ncclProxyArgs {
261     proxyProgressFunc_t progress;
262     struct ncclProxyArgs* next;
263     /* other fields */
264 }
265

```

266 This allows calls to different the implementations of the `progress` function for different transport

¹⁴<https://github.com/NVIDIA/nccl/tree/master/src/include/proxy.h>

267 methods¹⁵¹⁶¹⁷¹⁸. Each implementations calls the profiler API to inform about a different event type
268 (ProxyStep, KernelCh or Network plugin specific).

269 **3.2.4 recordEventState**

```
270 ncclResult_t recordEventState(  
271     void* eHandle,  
272     ncclProfilerEventState_v5_t eState,  
273     ncclProfilerEventStateArgs_v5_t* eStateArgs  
274 );  
275  
276
```

277 Some event types can be updated by NCCL through `recordEventState` (state and attributes)¹⁹.
278 `recordEventState` is called in the same functions that call `startEvent` and are happening after
279 `startEvent`.

280 **3.2.5 finalize**

```
281 ncclResult_t finalize(void* context);  
282  
283
```

284 After a user API call to free resources associated with a communicator, `finalize` is called. After-
285 wards, a reference counter tracks how many communicators are still being tracked by the profiler
286 plugin. If it reaches 0, the plugin will be closed via `dlclose(handle)`. Fig. 7 depicts the flow from
287 user API call to `finalize`.

¹⁵<https://github.com/NVIDIA/nccl/tree/master/src/transport/net.cc>

¹⁶https://github.com/NVIDIA/nccl/tree/master/src/transport/coll_net.cc

¹⁷<https://github.com/NVIDIA/nccl/tree/master/src/transport/p2p.cc>

¹⁸<https://github.com/NVIDIA/nccl/tree/master/src/transport/shm.cc>

¹⁹https://github.com/NVIDIA/nccl/tree/master/src/include/plugin/profiler_v5.h

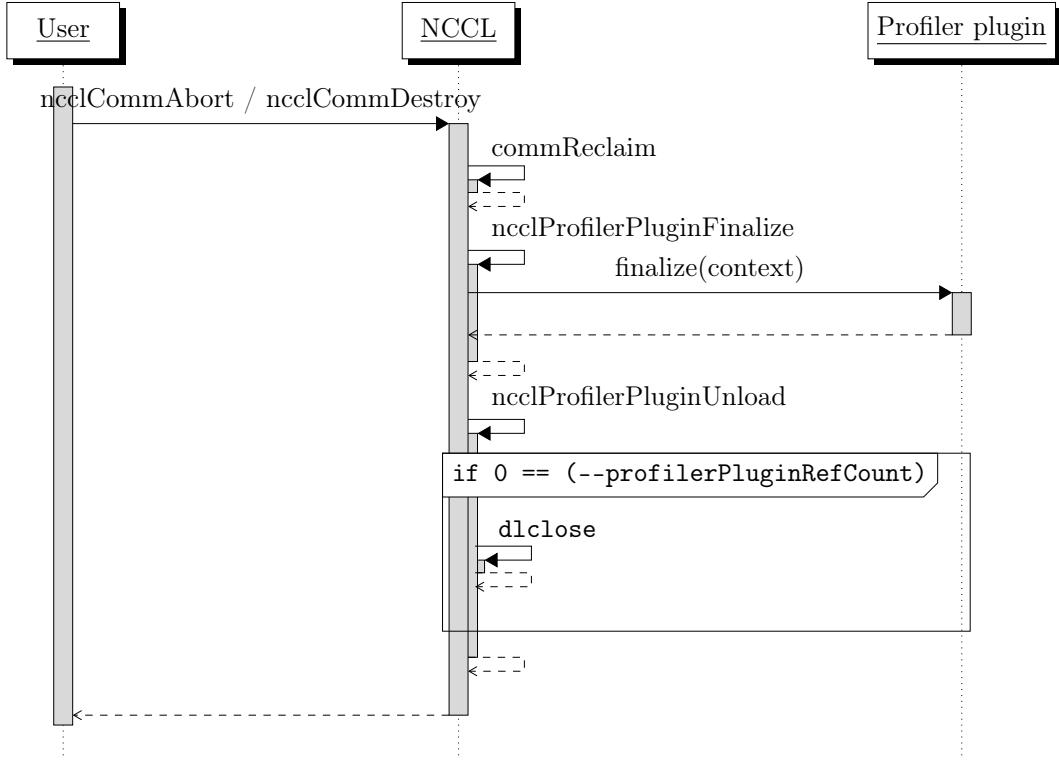


Figure 7: User API → `commReclaim` → `finalize` → plugin unload.

288 3.2.6 name

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

292 4 Code examples and visualizations

293 The following examples illustrate the profiling behavior for different user application settings:

- 294 • One Device per Thread
- 295 • Multiple Devices per Thread via `ncclGroupStart` and `ncclGroupEnd`
- 296 • One Device per Thread and aggregated operations via `ncclGroupStart` and `ncclGroupEnd`

297 A profiler plugin that logs all call information to a file has been developed and is used in all examples.
 298 An exemplary illustration is shown below:

```

299
300 struct MyContext { /* custom context struct */ };
301 struct MyEvent { /* custom event struct */ };

```

```

302
303 MyEvent* allocEvent(args) { /* handles event allocation */ }
304 uint64_t getTime() { /* gets time */ }
305 void writeJsonl() { /* writes call details to process specific log file as structured
306 jsonl */ }
307
308 ncclResult_t myInit( /* args - **context, *eActivationMask, ... */ ) {
309     *context = malloc(sizeof(struct MyContext));
310     *eActivationMask = 4095; /* enable ALL event types */
311
312     writeJsonl(getTime(), "Init", args);
313     return ncclSuccess;
314 }
315
316 ncclResult_t myStartEvent( /* args - **eHandle, ... */ ) {
317     *eHandle = allocEvent(args);
318
319     writeJsonl(getTime(), "StartEvent", args);
320     return ncclSuccess;
321 }
322
323 ncclResult_t myStopEvent(void* eHandle) {
324     writeJsonl(getTime(), "StopEvent", eHandle);
325
326     free(eHandle)
327     return ncclSuccess;
328 }
329
330 ncclResult_t myRecordEventState( /* args - ... */ ) {
331     writeJsonl(getTime(), "RecordEventState", args);
332     return ncclSuccess;
333 }
334
335 ncclResult_t myFinalize(void* context) {
336     writeJsonl(getTime(), "Finalize", args);
337
338     free(context);
339     return ncclSuccess;
340 }
341
342 ncclProfiler_v5_t ncclProfiler_v5 = {
343     "MyProfilerPlugin",
344     myInit,
345     myStartEvent,
346     myStopEvent,
347     myRecordEventState,
348     myFinalize,
349 };
350

```

351 Alongside the logging profiler plugin, a visualization tool has been built, that ingests the profiler logs
352 to inspect the exact behavior of internal calls from NCCL to the Profiler API. It displays the events
353 as colored bars on a timeline and separates them on different lanes. Each lane also displays some
354 information about the communicator, rank and thread corresponding to the event. Additionally,

355 blue dotted lines indicate the relationship between events according to the `parentObj` field and red
356 lines indicate which collective events belong to the same collective operation.

357 Further, a hover feature was added to inspect all details of an event, however this feature is not
358 used in the following illustrative examples.

359 **4.1 One Device per Thread**

360 This example visualizes an AllReduce collective across multiple GPUs (see Fig. 8 and Fig. 9). Each
361 NCCL thread manages a single GPU. This may be achieved by starting out with the same number
362 of MPI tasks with each task running single threaded; or by having less MPI tasks, but the tasks
363 create multiple thread workers. Custom initialization without MPI is also possible if desired.

```
364 // broadcast a commId
365
366 // ...
367
368 ncclCommInitRank(&rootComm, nRanks, commId, myRank);
369
370 // ...
371
372 ncclAllReduce(sendBuff, recvBuff, BUFFER_SIZE, ncclFloat, ncclSum, rootComm, streams);
373
374 // ...
375
376 ncclCommDestroy(rootComm);
```

379 The profiler API calls are visualized in Fig. 8 and Fig. 9. Below follows a full description of the
380 calls to the profiler API induced by the example program:

381 First, the profiler API `init` is called for each rank. This occurs during NCCL's internal com-
382 municator creation, when the application calls `ncclCommInitRank`. After the application calls
383 `ncclAllReduce`, many Profiler API calls to `stateEvent`, `stopEvent`, and `recordEventState` are
384 triggered: Intially, startEvent for the `groupApi` (green bar) is called. Below it, the startEvent
385 and soon the stopEvent for the AllReduce `collApi` event are called. The yellow bar shows when
386 NCCL enqueues the GPU kernel launch (`KernelLaunch` event). The two bars below represent the
387 `group` and `coll` events. NCCL also spawns a proxy progress thread per rank, which does addi-
388 tional profiler API calls. The first red `ProxyCtrl` event shows the proxy progress thread was asleep.
389 Next, a new `ProxyCtrl` event shows time for the proxy thread to append proxy ops. Then, ap-
390 pended ops start progressing (`ProxyOps` events), which in `op->progress()` starts `ProxyStep` and
391 `KernelCh` events that inform about low level network activity in updates via `recordEventState`
392 like `ProxyStepRecvGPUWait` (see Fig. 9). Network activity eventually completes and the AllReduce
393 collective finishes. The next `ProxyCtrl` event only shows the proxy thread sleeping again. Finally,
394 profiler `finalize` is called, which happens when the application cleans up NCCL communicators
395 and no further communicators are tracked in the profiler in each respective thread.

396 `ProxyStep` events are emitted in cross node communication environments. If this type of commu-
397 nication is not required, then `ProxyStep` events will not happen either.

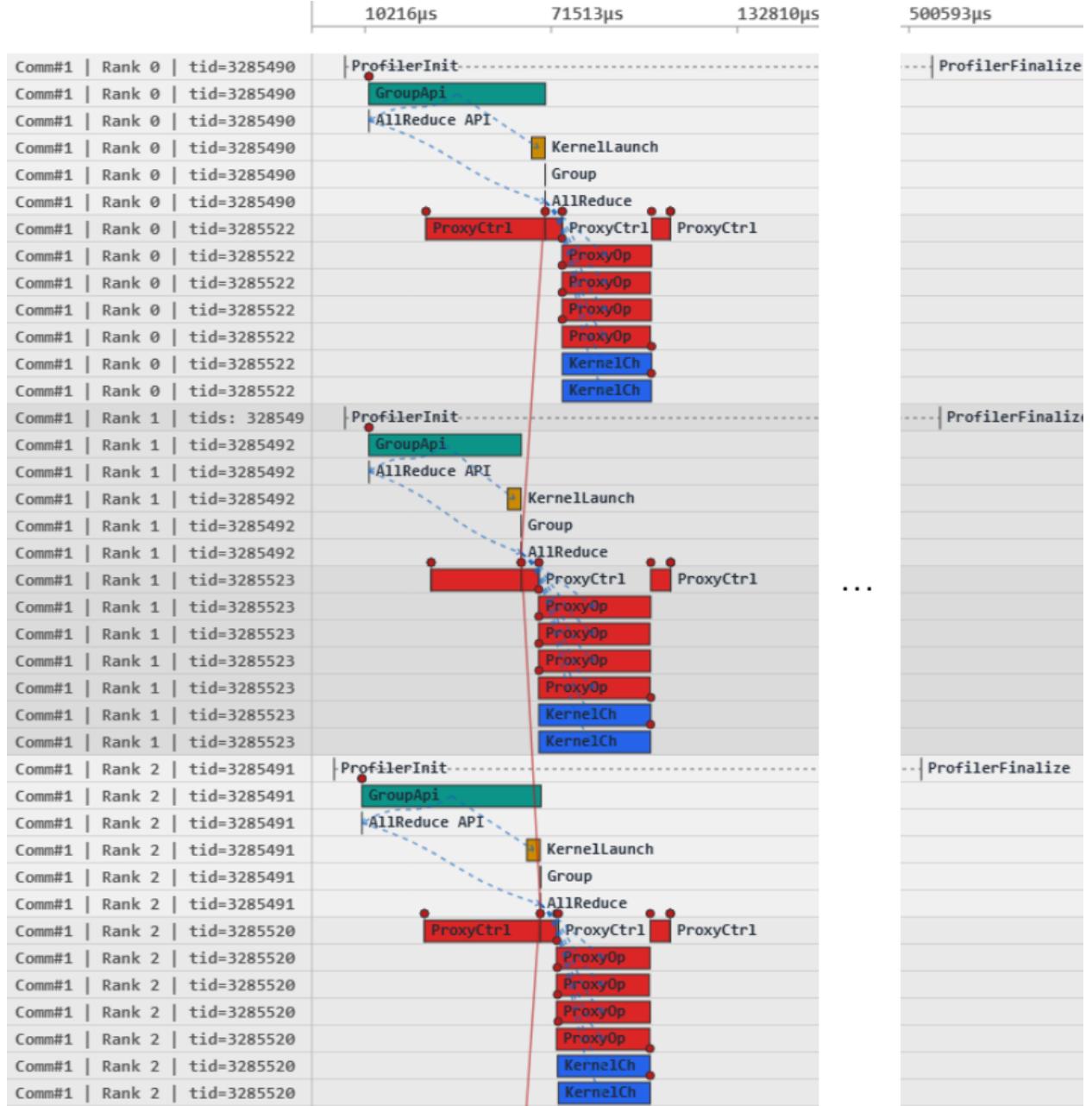


Figure 8: One device per thread: A visualization of the calls generated to the Profiler API, starting from communicator creation, followed by a collective operation and communicator destruction. ProxyStep events have been omitted for visual clarity, see Fig. 9 for a depiction.

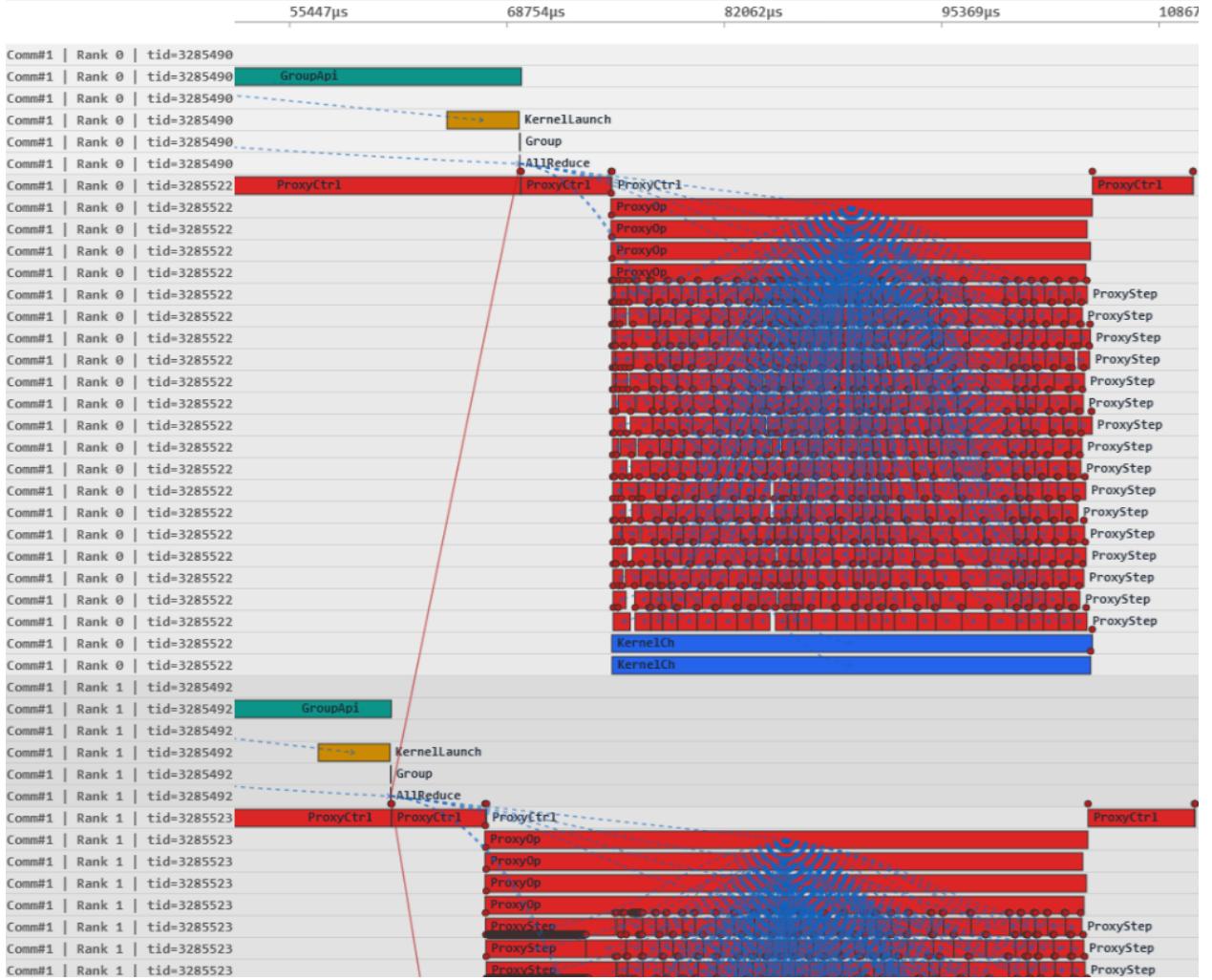


Figure 9: One device per thread: In Fig. 8 ProxyStep events have been omitted for visual clarity. However, in multinode settings, many additional profiler API calls for proxyStep events happen, informing about the low level network steps in their event details via `recordEventState` (indicated as red circles above each of the event bars). The blue dotted lines indicate the `parentObj` of each proxyStep event, which are the above proxyOp events.

398 4.1.1 Multiple Devices per Thread (ncclGroup)

399 In this example²⁰, one NCCL thread manages all GPUs on the same node. This is achieved by
400 wrapping communication initialization in `ncclGroupStart` and `ncclGroupEnd` for each managed
401 GPU. In this orchestration setting, NVIDIA’s documentation states that **collective API**
402 **calls should also be wrapped in ncclGroup**. Here, only one collective operation (per device)
403 is inside the `ncclGroup`:

```
404 // broadcast a commId
```

²⁰https://github.com/NVIDIA/nccl/tree/master/examples/03_collectives/01_allreduce/

```

406 // ...
407
408 ncclGroupStart();
409 for (int i=0; i<ngpus; i++) {
410     cudaSetDevice(dev);
411     ncclCommInitRank(comms+i, ngpus*nRanks, id, myRank*ngpus+i);
412 }
413 ncclGroupEnd();
414
415 // alternatively to above method, NCCL provides the convenience function
416 // ncclCommInitAll();
417
418 // ...
419
420 ncclGroupStart();
421 for (int i = 0; i < num_gpus; i++) {
422     ncclAllReduce( /* ... */ );
423 }
424 ncclGroupEnd();
425
426 // ...
427
428 for (int i = 0; i < num_gpus; i++) {
429     ncclCommDestroy(comms[i]);
430 }
431
432

```

433 In this example case, the profiler API behavior remains largely the same: The one difference is that
 434 NCCL internally calls the profiler API groupApi event only one time in total for aggregated opera-
 435 tions within a thread. Otherwise all other events are processed as usual and are called their usual
 436 amount of times irrespective of `ncclGroup`. This is visualized in Fig. 10. This behaviour also holds
 437 true within a process. It also holds when grouping (single) collectives for different communicators.

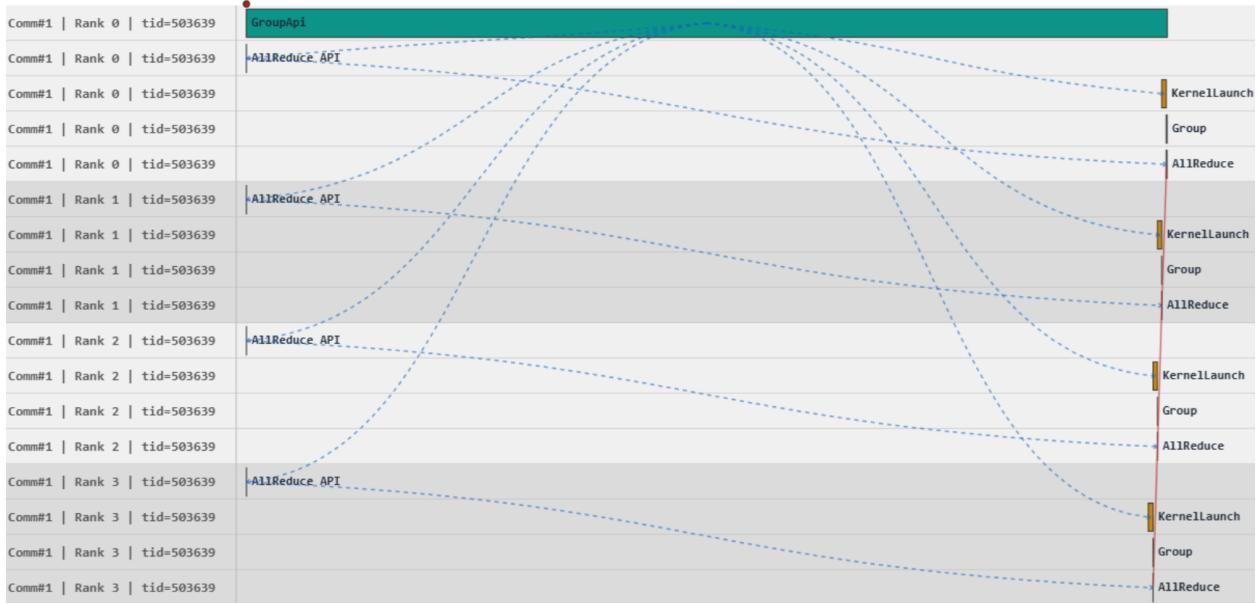


Figure 10: Multiple devices per thread: Events from the proxy thread as well as init and finalize calls are omitted. Collective API calls from multiple GPUs managed by a single thread only trigger a single `GroupApi` event.

438 4.1.2 Aggregated operations

439 In this example, the setting is such that only a single GPU is managed by a thread, but multiple
 440 collective operations are grouped (i.e. to optimize communication efficiency):

```

441 // broadcast a commId
442
443 // ...
444
445
446 ncclCommInitRank(&rootComm, nRanks, rootId, myRank);
447
448 // ...
449
450 ncclGroupStart();
451 ncclAllReduce( /* ... */ );
452 ncclBroadcast( /* ... */ );
453 ncclReduce( /* ... */ );
454 ncclAllGather( /* ... */ );
455 ncclReduceScatter( /* ... */ );
456 ncclGroupEnd();
457
458 // ...
  
```

460 The behavior changes can be described as follow:

- 461 • single `GroupApi` event per thread

- 462 • single KernelLaunch event per thread
 463 • single Group event per thread

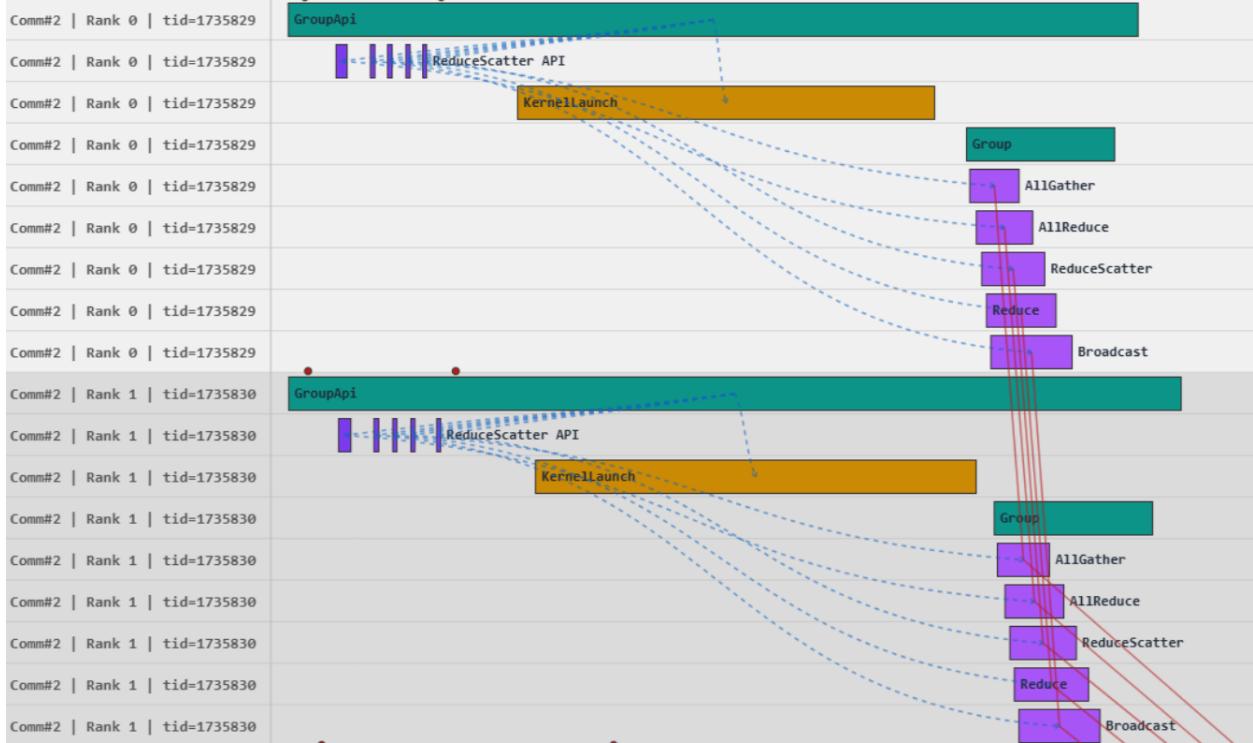


Figure 11: one GPU per thread with aggregated operations: multiple collective calls are grouped together and nccl does only a single kernel launch per thread.

464 5 Performance and scalability of the Profiler Plugin API

465 Experiments were run to assess the performance and scalability of profiler plugins. These experiments
 466 measure the overhead induced internally by NCCL to serve the profiler plugin, but do not
 467 intend to measure the performance of a profiler plugin itself as the plugin is fully customizable to
 468 the needs of the developer.

469 Thus, the profiler developed for the experiments only initializes a dummy context struct, returns
 470 NULL for event handles and tracks all events (`eActivationMask` set to 4095).

```
471 // an 'empty' NCCL Profiler Plugin
472
473
474 struct MyContext {
475     char dummy;
476 };
477
478 ncclResult_t myInit(void** context, uint64_t commId, int* eActivationMask, const char*
479     commName, int nNodes, int nranks, int rank, ncclDebugLogger_t logfn) {
```

```

480     *context = malloc(sizeof(struct MyContext));
481     *eActivationMask = 4095; /* enable ALL event types */
482     return ncclSuccess;
483 }
484
485 ncclResult_t myStartEvent(void* context, void** eHandle, ncclProfilerEventDescr_v5_t*
486     eDescr) {
487     *eHandle = NULL;
488     return ncclSuccess;
489 }
490
491 ncclResult_t myStopEvent(void* eHandle) {
492     return ncclSuccess;
493 }
494
495 ncclResult_t myRecordEventState(void* eHandle, ncclProfilerEventState_v5_t eState,
496     ncclProfilerEventStateArgs_v5_t* eStateArgs) {
497     return ncclSuccess;
498 }
499
500 ncclResult_t myFinalize(void* context) {
501     free(context);
502     return ncclSuccess;
503 }
504
505 ncclProfiler_v5_t ncclProfiler_v5 = {
506     "EmptyProfiler",
507     myInit,
508     myStartEvent,
509     myStopEvent,
510     myRecordEventState,
511     myFinalize,
512 };
513

```

- 514 For testing the performance overhead in collective and P2P operations, **nccl-tests** from NVIDIA
 515 was used²¹.
- 516 The applications `sendrecv_perf` and `all_reduce_perf` were launched with following test parameters:
 517 message size 64 B, 1 000 000 iterations per size, 100 warmup iterations. Single-node jobs used
 518 one node and 4 GPUs; multi-node jobs used 2 nodes, 4 GPUs per node, 8 MPI ranks in total. For
 519 each experiment, the application was run once without the profiler and once with the empty profiler
 520 plugin.
- 521 The Table 1 shows the average latency per operation (time in μ s) across iterations. The empty
 522 profiler adds roughly 8 to 9 μ s overhead per operation in single-node runs (4 GPUs), but introduces
 523 negligible overhead in multi-node runs (8 GPUs across 2 nodes).

²¹<https://github.com/NVIDIA/nccl-tests>

Table 1: Profiler overhead: nccl-tests `sendrecv_perf` (P2P) and `all_reduce_perf` (collectives). Latency averaged over 1M iterations.

Test	Environment	Without profiler (μ s)	With profiler (μ s)
P2P (<code>sendrecv_perf</code>)	Single-node (4 GPUs)	14.3	23.88
	Multi-node (2×4 GPUs)	13.05	12.95
Collectives (<code>all_reduce_perf</code>)	Single-node (4 GPUs)	14.96	23.29
	Multi-node (2×4 GPUs)	17.99	18.34

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

526 6 Discussion

527 6.1 Considerations for developers of a Profiler Plugin

528 **Profiler Visualization.** The visualization tool used in the code examples is helpful for under-
 529 standing the internal call behavior to the Profiler API by NCCL and will be made available along
 530 with this report. It may serve as a reference to compare against for other developers that build a
 531 profiler plugin or visualizer

532 **Correlating Collective Events with seqNumber.** When profiling is enabled, NCCL counts the
 533 number of calls for each type of collective function per communicator.

```
534 /src/include/comm.h
535
536 struct ncclComm {
537     uint64_t seqNumber[NCCL_NUM_FUNCTIONS];
538     /* other fields */
539 }
```

```
541 /src/plugin/profiler.cc
542
543 ncclResult_t ncclProfilerStartTaskEvents(struct ncclKernelPlan* plan) {
544     /* other code */
545     __atomic_fetch_add(&plan->comm->seqNumber[ct->func], 1, __ATOMIC_RELAXED);
546     /* other code */
547 }
```

549 This value is present in the `eDescr` for collective events and can be used to identify which collectives
 550 operations belong together across processes (see Fig. 12).

551 **Tracing low level activity back to NCCL API calls with parentObj.** If a plugin developer
 552 wants utilize this field, they should ensure that potential address reuse does not create ambiguity
 553 to what the parentObj was originally pointing to. *Custom memory management is advised.* This

554 field is useful when trying to understand which user API call triggered which events of lower level
 555 operations or activity such as network activity (see Fig. 12).

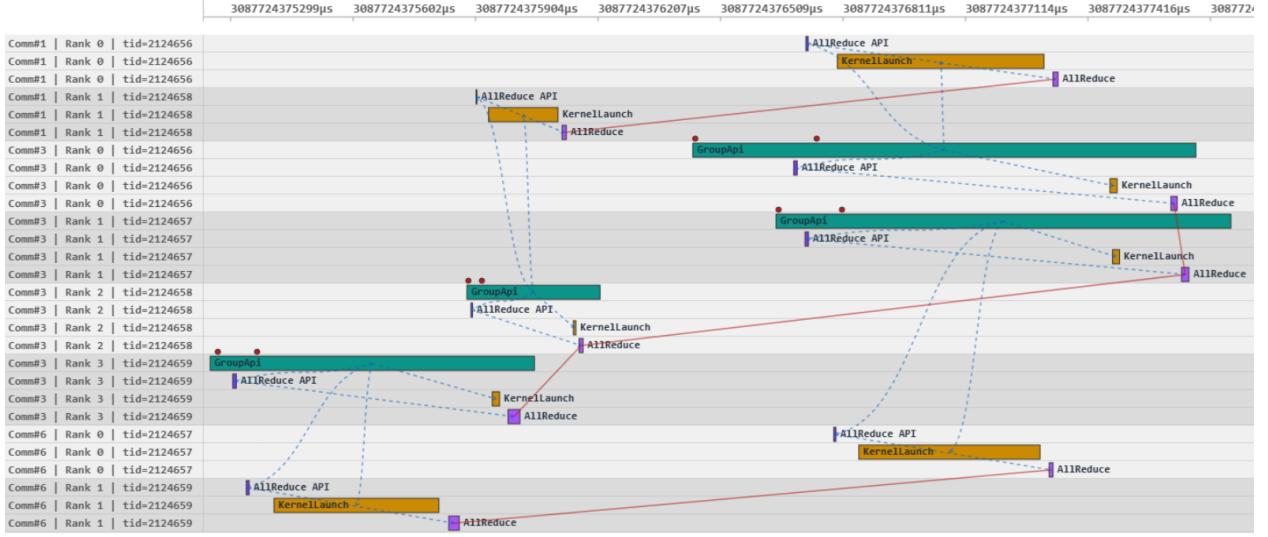


Figure 12: An example illustrating how `parentObj` and `seqNumber` can be used to better understand the timing of concurrent collective operations.

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

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

565 **Tracking communicator parent–child relationships.** With the current Profiler plugin API,
 566 it is not possible to detect whether a communicator originates from another one (e.g., via
 567 `ncclCommSplit` or `ncclCommShrink`). The plugin's `init` callback only receives a single communi-
 568 cator ID (`commId`, which corresponds to `comm->commHash`), as well as `commName`, `nNodes`, `nRanks`,
 569 and `rank`; there is no `parentCommId` or similar argument. In split/shrink, the `commHash` of the child
 570 node is calculated internally as a one-way digest of the `commHash` of the parent node and the split
 571 parameters (`splitCount`, `color`). Therefore, the relationship cannot be restored based on the ID
 572 alone.

573 **6.2 Known limitations**

574 Kernel event instrumentation uses counters exposed by the kernel to the host and the proxy progress
575 thread. Thus the proxy progress thread infrastructure is shared between network and profiler. If
576 the proxy is serving network requests, reading kernel profiling data can be delayed, causing loss of
577 accuracy. Similarly, under heavy CPU load and delayed scheduling of the proxy progress thread,
578 accuracy can be lost.

579 From profiler version 4, NCCL uses a per-channel ring buffer of 64 elements. Each counter is
580 complemented by two timestamps (ptimers) supplied by the NCCL kernel (start and stop of the
581 operation in the kernel). NCCL propagates these timestamps to the profiler plugin so it can convert
582 them to the CPU time domain.

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

584 **6.3 Potential Integration with Score-P**

585 The Score-P measurement infrastructure²² is a highly scalable and easy-to-use tool suite for profiling
586 and event tracing of HPC applications. It supports a number of analysis tools. Currently, it works
587 with Scalasca, Vampir, and Tau and is open for other tools and produces OTF2 traces and CUBE4
588 profiles.

589 For Score-P, it is important that communicator identities are unique. NCCL achieves this for
590 `ncclGetUniqueId`²³ without a central coordinator. Each call fills the bootstrap handle embedded
591 in `ncclUniqueId` with

- 592
 - a random 64-bit `magic` value from `/dev/urandom`, and
 - the socket `address` of a new listening socket (IP, port), whose port is chosen by the operating
593 system.

595 The pair (random magic, IP+port) is unique in practice: different MPI tasks or repeated calls in one
596 process each get distinct random magic and a distinct OS-assigned port so collisions are avoided.

597 The NCCL profiler plugin is callback-driven *by NCCL*. NCCL loads it via `dlopen` and invokes
598 `startEvent`, `stopEvent`, and `recordEventState` during collective and P2P operations.

599 In one potential integration strategy, a developer would implement the NCCL profiler API and
600 could use Score-P's user instrumentation API (e.g., `SCOREP_USER_REGION_BY_NAME_BEGIN/END`) to
601 inject NCCL Profiler Events as regions into Score-P. The region name could be derived from the
602 event descriptor (e.g., `collApi.func` for `ncclAllReduce`). In this design, NCCL drives the profiler
603 and the profiler forwards events into Score-P. NCCL collective operations then appear as regions in
604 Score-P profiles and traces.

²²<https://www.vi-hps.org/projects/score-p/overview>

²³<https://github.com/NVIDIA/nccl/tree/master/src/init.cc>

605 Alternatively, NVTX and CUPTI could be leveraged with CUDA adapters from Score-P. Similarly,
 606 the NCCL profiler plugin would use the NVTX API²⁴ to annotate regions. Additionally,
 607 kernel tracing of with CUPTI can be integrated directly into the NCCL profiler plugin. The
 608 CUPTI API provides functions to correlate application code regions with CUPTI activity via
 609 `cuptiActivityPushExternalCorrelationId` and
 610 `cuptiActivityPopExternalCorrelationId`. This can be handily integrated with the profiler API,
 611 whenever `KernelLaunch` events are started and stopped, while continuously incrementing the corre-
 612 lation id in a thread-safe manner. Fig. 13 provides an example visualization of this method. CUPTI
 613 can be initialized and cleaned up within the profiler plugin’s own `init` and `finalize` functions.

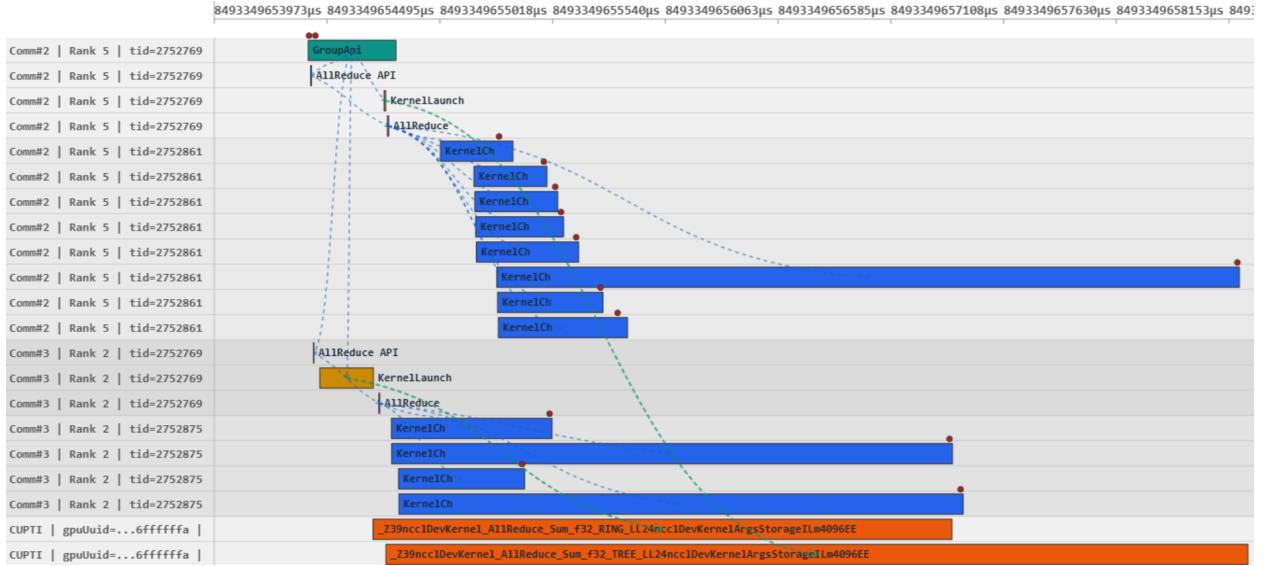


Figure 13: CUPTI activity is visualized as orange event bars. Through a correlation Id, it is possible to trace the activity back to `KernelLaunch` events

614 7 Conclusion

615 This feasibility study examined the NCCL Profiler Plugin API and its suitability for integration
 616 with Score-P. The report provided background on NCCL and its design, explained how the profiler
 617 plugin is detected and loaded, and described the API definition with its five core callbacks `init`,
 618 `startEvent`, `stopEvent`, `recordEventState`, `finalize`. Code examples and visualizations illustrate
 619 the event flow from API calls to NCCL’s internal profiler callbacks. Performance experiments showed
 620 that an empty profiler adds roughly 8–9 μ s overhead per operation in single-node runs but introduces
 621 negligible overhead in multi-node runs, and scaling to many GPUs across nodes required no changes
 622 to the profiler plugin. The discussion covered developer considerations, known limitations, and a
 623 potential integration strategy with Score-P.

624 The NCCL Profiler API allows for highly customized plugins tailored to the analysis needs, whether
 625 for simple timing, kernel tracing via CUPTI, or integration with external tools such as Score-P.

²⁴https://nvidia.github.io/NVTX/doxygen/group__m_a_r_k_e_r_s__a_n_d__r_a_n_g_e_s.html

626 A notable advantage is its low overhead: NVIDIA advertises their `inspector`²⁵ implementation
627 as efficient enough for “always-on” profiling in production. On the downside, profiler plugins may
628 require maintenance and active development, since NCCL is actively developed. API versions evolve
629 and new features are being introduced.

²⁵<https://github.com/NVIDIA/nccl/tree/master/ext-profiler/inspector>