

1 NCCL Profiler Plugin API – A Feasibility Study

2

3 **Contents**

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

22	<b>6 Discussion</b>	<b>23</b>
23	6.1 Considerations for developers of a Profiler Plugin . . . . .	23
24	6.2 Known limitations . . . . .	25
25	6.3 Potential Integration with Score-P . . . . .	25
26	<b>7 Conclusion</b>	<b>26</b>

## 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  
 29 and optimizing application performance is therefore essential to maximize efficiency and reduce  
 30 costs. Many AI workloads involve communication between GPUs, often distributed across numer-  
 31 ous GPUs in multi-node systems. The NVIDIA Collective Communication Library (NCCL) serves  
 32 as the core library for implementing optimized communication primitives on NVIDIA GPUs. To  
 33 provide detailed performance insights, NCCL offers a flexible profiler plugin API. This allows de-  
 34 velopers to directly integrate custom profiling tools into the library to extract detailed performance  
 35 data on communication operations. This feasibility study explores the capabilities and integration  
 36 mechanisms of the API.

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

## 42 **2 Introduction to NCCL**

43 NCCL was first introduced by NVIDIA in 2015 at the Supercomputing Conference<sup>1</sup> with code being  
 44 made available on GitHub<sup>2</sup>. The release of NCCL 2.0 in 2017 brought support for NVLink, however  
 45 this was initially only available as pre-built binaries. With the release of NCCL 2.3 in 2018, it  
 46 returned to being fully open source. The NCCL Profiler Plugin API was even later introduced with  
 47 NCCL 2.23 in early 2025.

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

---

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

<sup>2</sup><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`<sup>3</sup>, 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`<sup>4</sup>. 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.

---

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

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

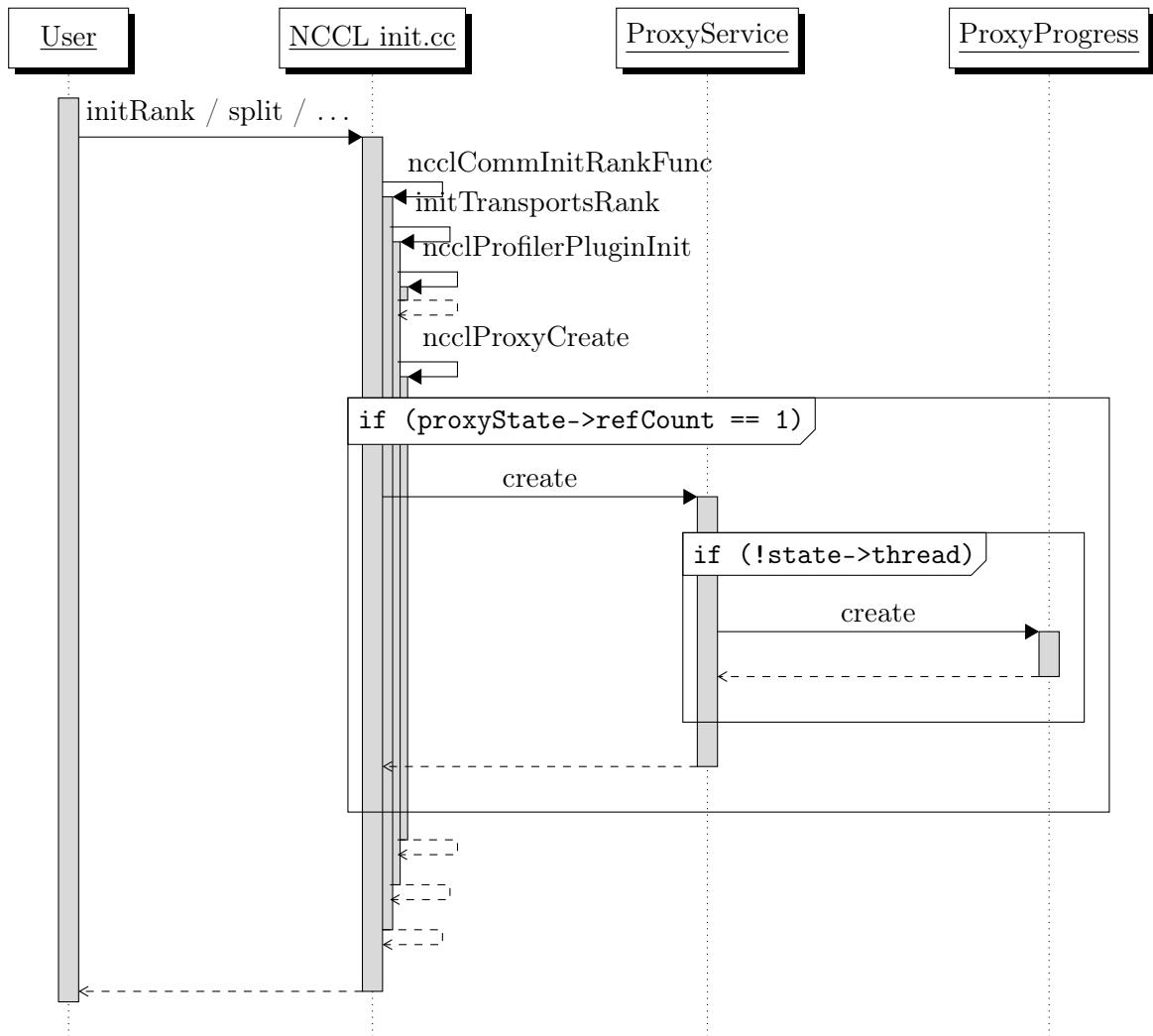


Figure 1: Thread creation: User API → NCCL internal init → create ProxyService → create ProxyProgress.

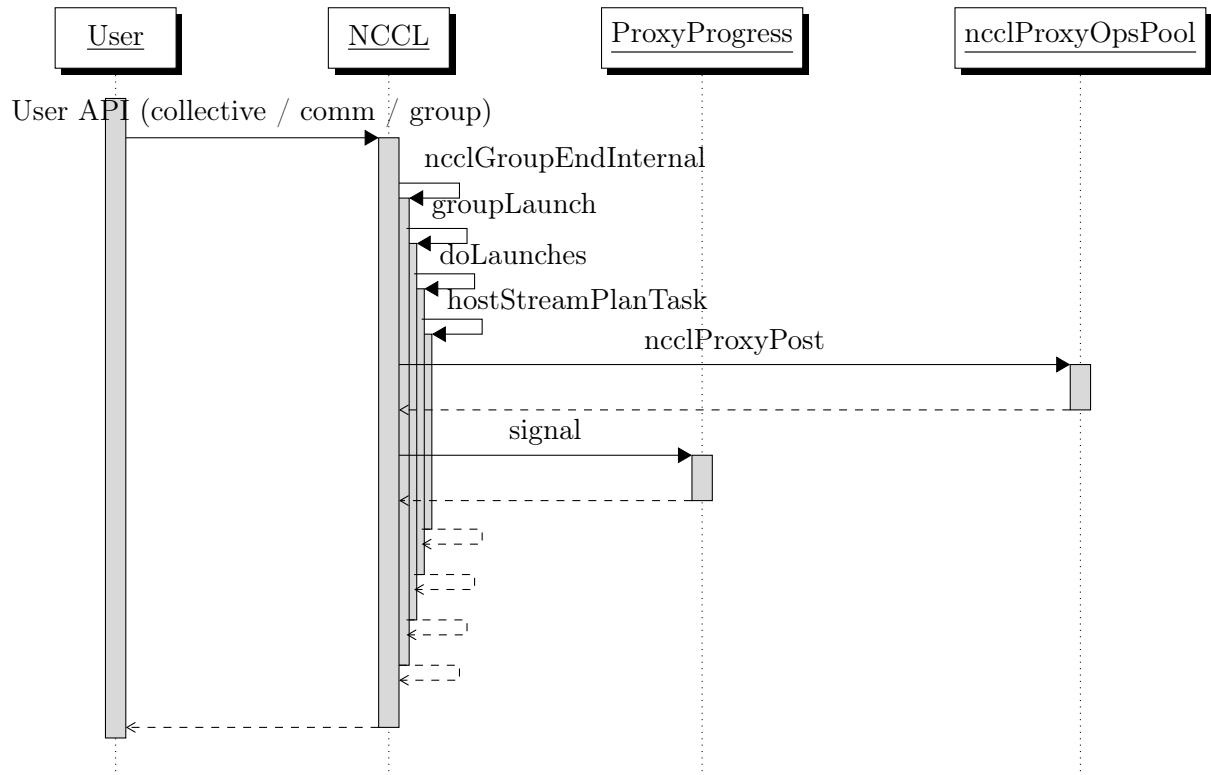


Figure 2: Flow from User API to `ncclProxyPost`

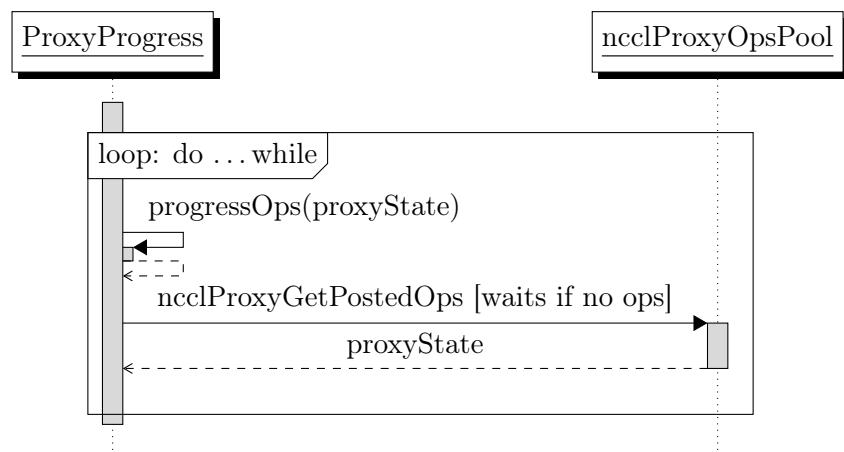


Figure 3: `/src/proxy.cc` `ncclProxyProgress` progressing loop: progress ops, then get posted ops (or wait).

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`<sup>5</sup> 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`<sup>6</sup> 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 documentation<sup>7</sup> 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

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

<sup>6</sup>[https://github.com/NVIDIA/nccl/tree/master/src/plugin/plugin\\_open.cc](https://github.com/NVIDIA/nccl/tree/master/src/plugin/plugin_open.cc)

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

<sup>115</sup> being developed against the NCCL release 2.25.1 with Profiler API version 2, was unable to run  
<sup>116</sup> with the latest NCCL release<sup>8</sup>. Around this time, the NCCL repository has undergone a refactor  
<sup>117</sup> related to the profiler plugin.



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

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

118 **3.2 Profiler API**

119 The plugin must implement a profiler API specified by NCCL by exposing a struct<sup>9</sup>. This struct  
120 should contain pointers to all functions required by the API. A plugin may expose multiple versioned  
121 structs for backwards compatibility with older NCCL versions.

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

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

136 Five functions must be implemented for the API. Internally NCCL wraps calls to the profiler API  
137 in custom functions which are all declared in a single file<sup>10</sup>.

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

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

142 **3.2.1 init**

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

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

<sup>9</sup>[https://github.com/NVIDIA/nccl/tree/master/src/include/plugin/profiler/profiler\\_v5.h](https://github.com/NVIDIA/nccl/tree/master/src/include/plugin/profiler/profiler_v5.h)

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

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

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

163 The plugin developer should set `int* eActivationMask` to a bitmask<sup>11</sup>, indicating which event  
164 types the profiler wants to track:

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

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

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

### 189 3.2.2 startEvent

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

```
191
192 ncclResult_t startEvent(
193     void* context, // opaque profiler context object
194     void** eHandle, // out param - event handle
195     ncclProfilerEventDescr_v5_t* eDescr // pointer to event descriptor
196 );
197 
```

198 As of release v2.29.2 NCCL does not use the return value. `void** eHandle` may point to a custom  
199 event object; this pointer is passed again in `stopEvent` and `recordEventState`. `eDescr`<sup>12</sup> describes  
200 the started event.

<sup>11</sup>[https://github.com/NVIDIA/nccl/tree/master/src/include/plugin/nccl\\_profiler.h](https://github.com/NVIDIA/nccl/tree/master/src/include/plugin/nccl_profiler.h)

<sup>12</sup>[https://github.com/NVIDIA/nccl/tree/master/src/include/plugin/profiler\\_v5.h](https://github.com/NVIDIA/nccl/tree/master/src/include/plugin/profiler_v5.h)

- 201 The field `void* parentObj` in the event descriptor is the `eHandle` of a parent event (or null). The  
 202 use of this field can be explained as following:
- 203 All User API calls to Collective or P2P operations will start a Group API event. When networking is  
 204 required, ProxyCtrl Events may be emitted. Depending on the `eActivationMask` bitmask returned  
 205 in the `init` function, further (child) events will be emitted in deeper regions of the nccl code base.  
 206 It can be thought of as an event hierarchy<sup>13</sup> with several depth levels:

```

207
208 Group API event
209 |
210 +- Collective API event
211 | |
212 | +- Collective event
213 | |
214 | +- ProxyOp event
215 | | |
216 | | +- ProxyStep event
217 | | |
218 | | +- NetPlugin event
219 | |
220 | +- KernelCh event
221 |
222 +- Point-to-point API event
223 | |
224 | +- Point-to-point event
225 | |
226 | +- ProxyOp event
227 | | |
228 | | +- ProxyStep event
229 | | |
230 | | +- NetPlugin event
231 | |
232 | +- KernelCh event
233 |
234 +- Kernel Launch event
235
236
237 ProxyCtrl event
  
```

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

### 241 3.2.3 stopEvent

```

242 ncclResult_t stopEvent(void* eHandle); // handle to event object
243
  
```

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

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

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

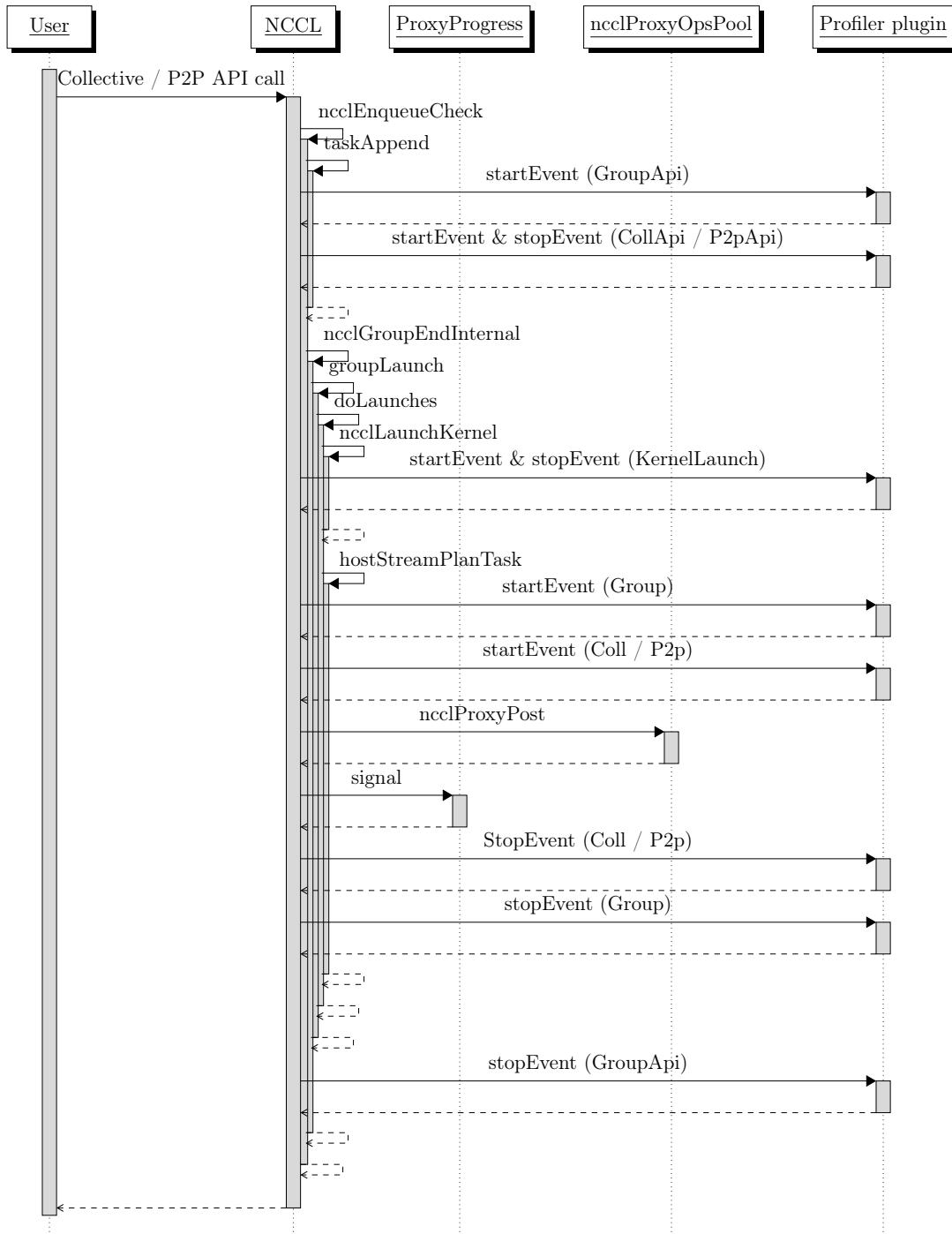


Figure 5: Flow from NCCL API calls to profiler events. In case of `ncclGroupStart` / `ncclGroupEnd`. multiple events of everything (except GroupApi) are called. Internally, some Collectives (e.g. `ncclAlltoAll`) are implemented as multiple p2p ops, triggering many P2pApi and P2p events.

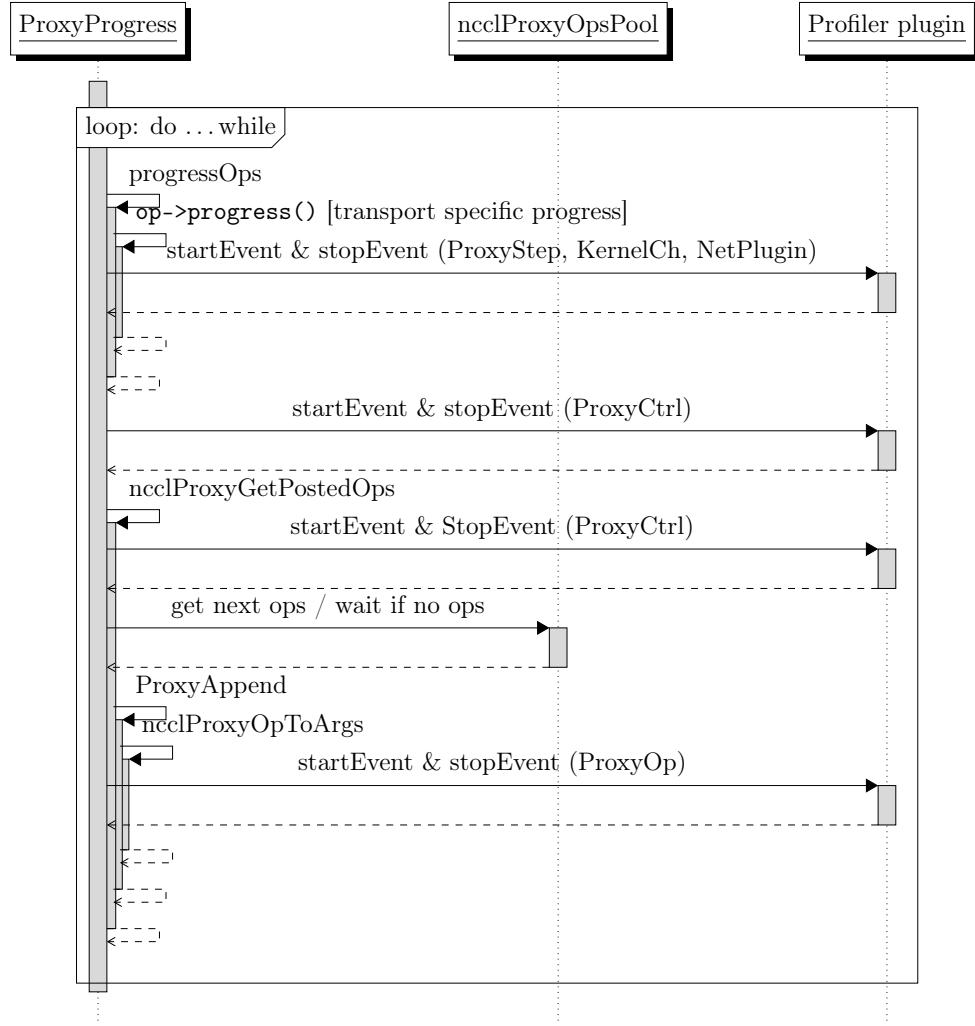


Figure 6: `ncclProxyProgress`: `progressOps` emits `ProxyStep`/`KernelCh`/`NetPlugin` events. `getPostedOps` emits `ProxyOp` events. Several events `ProxyCtrl` are also emitted

251   `op->progress()` progresses transport specific ops. This is implemented as a function pointer type<sup>14</sup>.  
 252   Confusingly the variable is called ‘`op`’, although its type is `ncclProxyArgs` and *not* `ncclProxyOp`.

```

253
254 typedef ncclResult_t (*proxyProgressFunc_t)(struct ncclProxyState*, struct ncclProxyArgs
255   *);
256
257 struct ncclProxyArgs {
258     proxyProgressFunc_t progress;
259     struct ncclProxyArgs* next;
260     /* other fields */
261 }

```

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

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

264 methods<sup>15161718</sup>. Each implementations calls the profiler API to inform about a different event type  
265 (ProxyStep, KernelCh or Network plugin specific).

266 **3.2.4 recordEventState**

```
267 ncclResult_t recordEventState(  
268     void* eHandle,  
269     ncclProfilerEventState_v5_t eState,  
270     ncclProfilerEventStateArgs_v5_t* eStateArgs  
271 );  
272
```

274 Some event types can be updated by NCCL through `recordEventState` (state and attributes)<sup>19</sup>.  
275 `recordEventState` is called in the same functions that call `startEvent` and are happening after  
276 `startEvent`.

277 **3.2.5 finalize**

```
278 ncclResult_t finalize(void* context);  
279
```

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

---

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

<sup>16</sup>[https://github.com/NVIDIA/nccl/tree/master/src/transport/coll\\_net.cc](https://github.com/NVIDIA/nccl/tree/master/src/transport/coll_net.cc)

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

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

<sup>19</sup>[https://github.com/NVIDIA/nccl/tree/master/src/include/plugin/profiler\\_v5.h](https://github.com/NVIDIA/nccl/tree/master/src/include/plugin/profiler_v5.h)

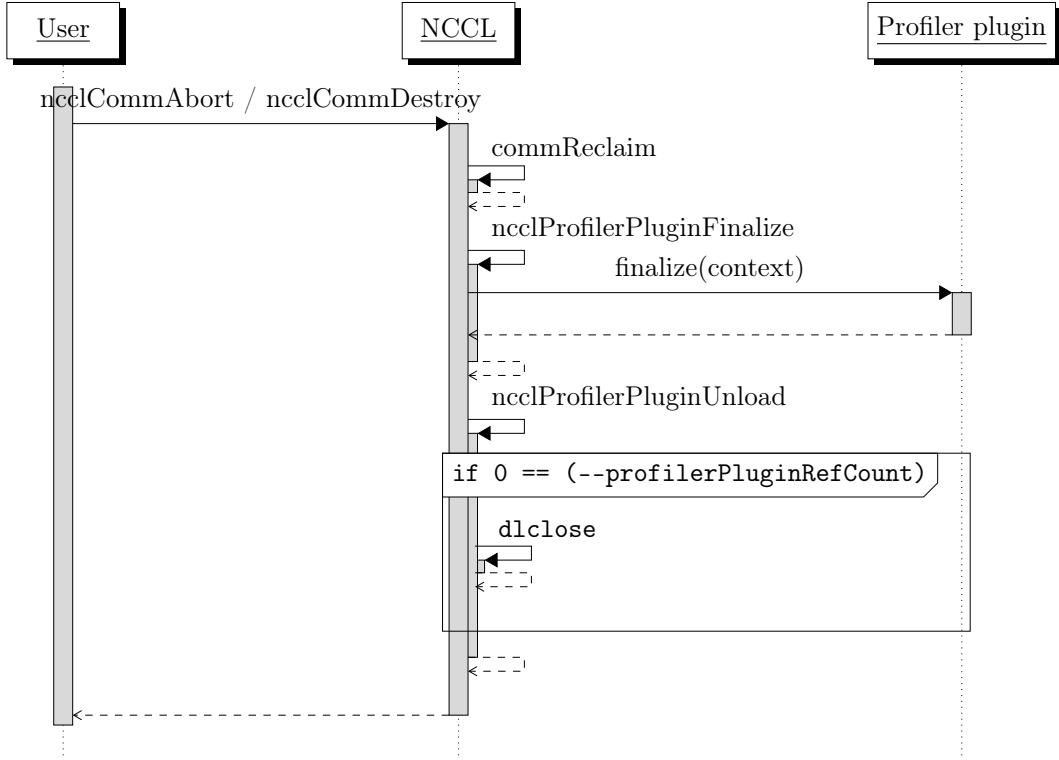


Figure 7: User API → `commReclaim` → `finalize` → plugin unload.

### 285 3.2.6 name

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

## 289 4 Code examples and visualizations

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

- 291 • One Device per Thread
- 292 • Multiple Devices per Thread via `ncclGroupStart` and `ncclGroupEnd`
- 293 • One Device per Thread and aggregated operations via `ncclGroupStart` and `ncclGroupEnd`

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

```

296
297 struct MyContext { /* custom context struct */ };
298 struct MyEvent { /* custom event struct */ };

```

```

299
300 MyEvent* allocEvent(args) { /* handles event allocation */ }
301 uint64_t getTime() { /* gets time */ }
302 void writeJsonl() { /* writes call details to process specific log file as structured
303   jsonl */ }
304
305 ncclResult_t myInit( /* args - **context, *eActivationMask, ... */ ) {
306   *context = malloc(sizeof(struct MyContext));
307   *eActivationMask = 4095; /* enable ALL event types */
308
309   writeJsonl(getTime(), "Init", args);
310   return ncclSuccess;
311 }
312
313 ncclResult_t myStartEvent( /* args - **eHandle, ... */ ) {
314   *eHandle = allocEvent(args);
315
316   writeJsonl(getTime(), "StartEvent", args);
317   return ncclSuccess;
318 }
319
320 ncclResult_t myStopEvent(void* eHandle) {
321   writeJsonl(getTime(), "StopEvent", eHandle);
322
323   free(eHandle)
324   return ncclSuccess;
325 }
326
327 ncclResult_t myRecordEventState( /* args - ... */ ) {
328   writeJsonl(getTime(), "RecordEventState", args);
329   return ncclSuccess;
330 }
331
332 ncclResult_t myFinalize(void* context) {
333   writeJsonl(getTime(), "Finalize", args);
334
335   free(context);
336   return ncclSuccess;
337 }
338
339 ncclProfiler_v5_t ncclProfiler_v5 = {
340   "MyProfilerPlugin",
341   myInit,
342   myStartEvent,
343   myStopEvent,
344   myRecordEventState,
345   myFinalize,
346 };

```

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

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

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

## 356 4.1 One Device per Thread

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

```
361 // broadcast a commId
362
363 // ...
364
365 ncclCommInitRank(&rootComm, nRanks, commId, myRank);
366
367 // ...
368
369 ncclAllReduce(sendBuff, recvBuff, BUFFER_SIZE, ncclFloat, ncclSum, rootComm, streams);
370
371 // ...
372
373 ncclCommDestroy(rootComm);
```

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

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

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

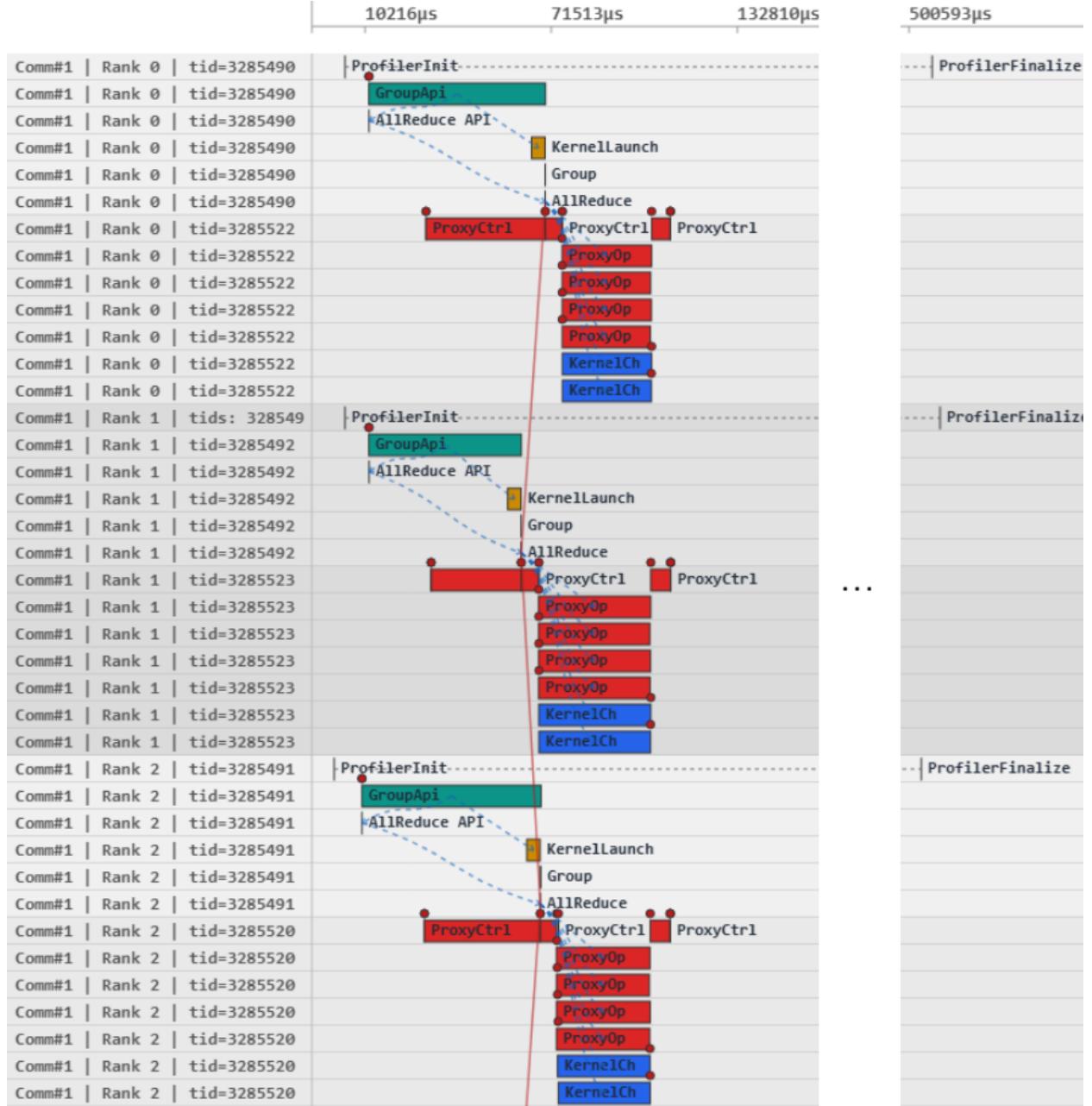


Figure 8: One device per thread: A visualization of the calls generated to the Profiler API, starting from communicator creation, followed by a collective operation and communicator destruction. ProxyStep events have been omitted for visual clarity, see Fig. 9 for a depiction.

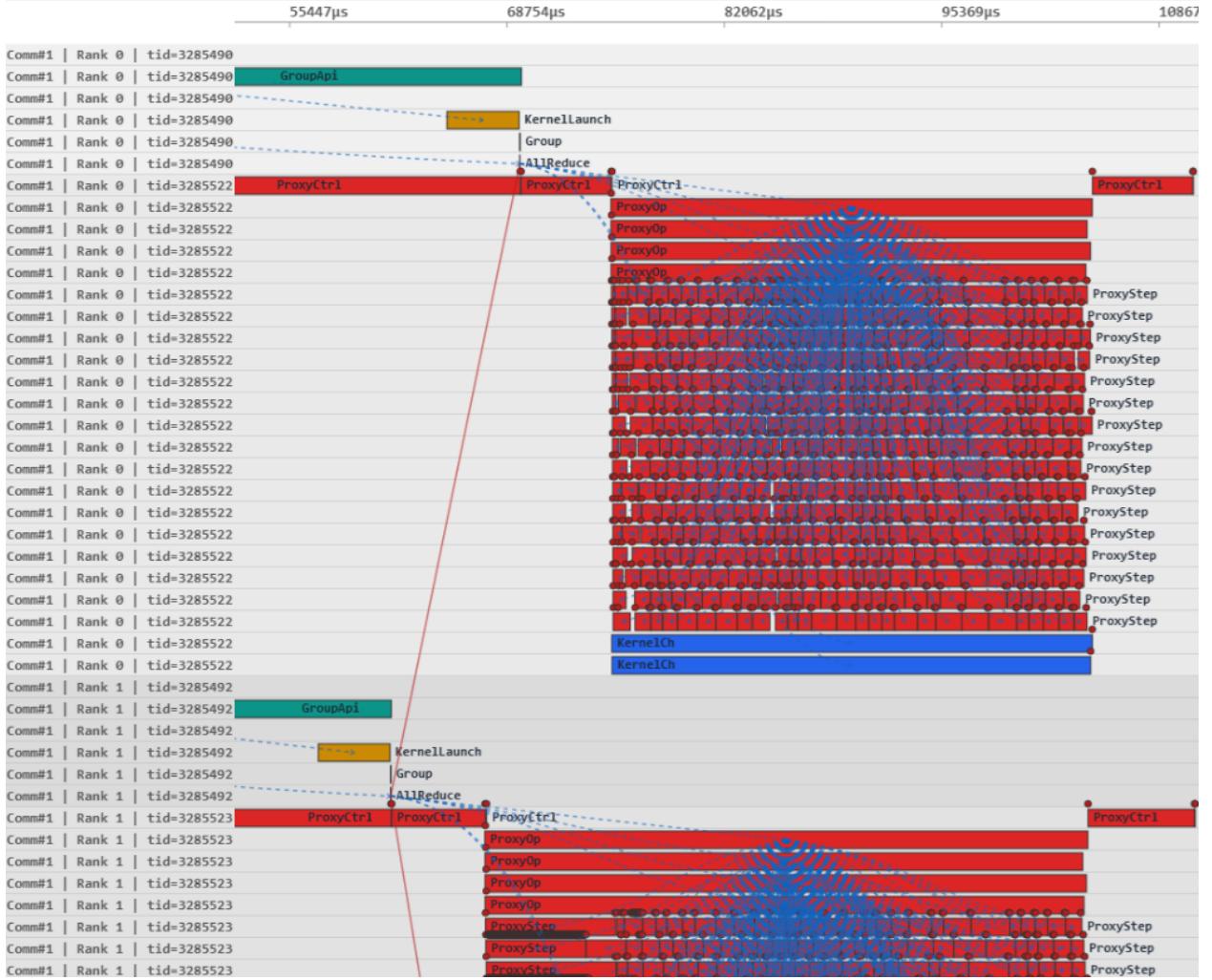


Figure 9: One device per thread: In Fig. 8 ProxyStep events have been omitted for visual clarity. However, in multinode settings, many additional profiler API calls for proxyStep events happen, informing about the low level network steps in their event details via `recordEventState` (indicated as red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

#### 395 4.1.1 Multiple Devices per Thread (ncclGroup)

396 In this example<sup>20</sup>, one NCCL thread manages all GPUs on the same node. This is achieved by  
 397 wrapping communication initialization in `ncclGroupStart` and `ncclGroupEnd` for each managed  
 398 GPU. In this orchestration setting, NVIDIA's documentation states that **collective API**  
 399 **calls should also be wrapped in ncclGroup**. Here, only one collective operation (per device)  
 400 is inside the `ncclGroup`:

```
401 // broadcast a commId
```

<sup>20</sup>[https://github.com/NVIDIA/nccl/tree/master/examples/03\\_collectives/01\\_allreduce/](https://github.com/NVIDIA/nccl/tree/master/examples/03_collectives/01_allreduce/)

```

403 // ...
404
405
406 ncclGroupStart();
407 for (int i=0; i<ngpus; i++) {
408     cudaSetDevice(dev);
409     ncclCommInitRank(comms+i, ngpus*nRanks, id, myRank*ngpus+i);
410 }
411 ncclGroupEnd();
412
413 // alternatively to above method, NCCL provides the convenience function
414 // ncclCommInitAll();
415
416 // ...
417
418 ncclGroupStart();
419 for (int i = 0; i < num_gpus; i++) {
420     ncclAllReduce( /* ... */ );
421 }
422 ncclGroupEnd();
423
424 // ...
425
426 for (int i = 0; i < num_gpus; i++) {
427     ncclCommDestroy(comms[i]);
428 }
```

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

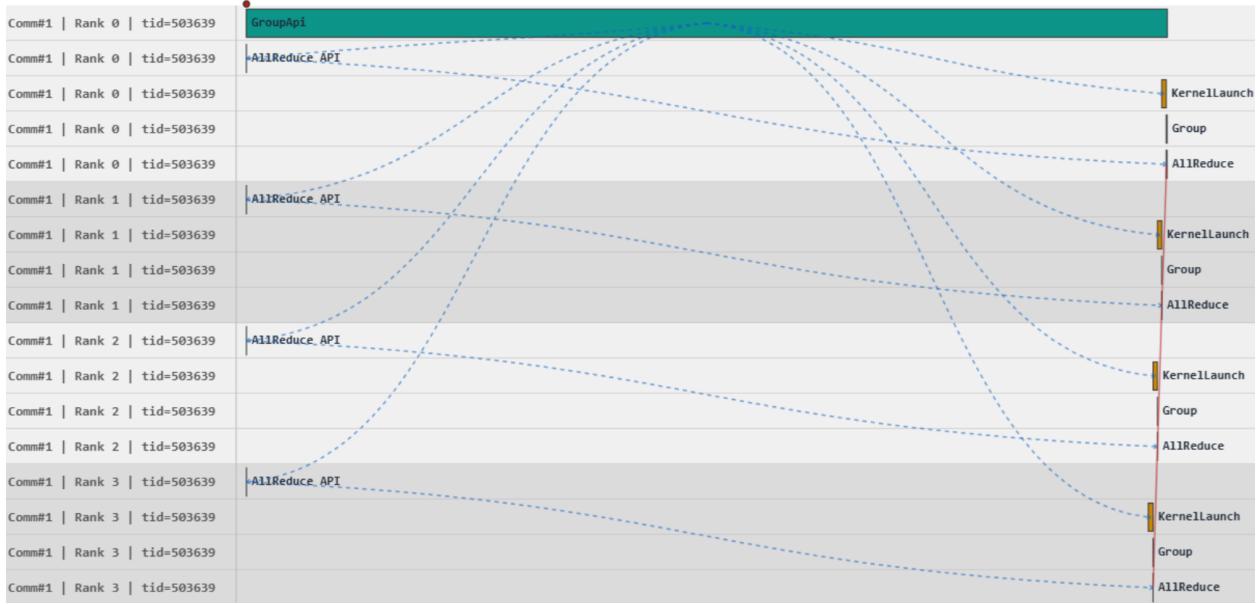


Figure 10: Multiple devices per thread: Events from the proxy thread as well as init and finalize calls are omitted. Collective API calls from multiple GPUs managed by a single thread only trigger a single `GroupApi` event.

#### 435 4.1.2 Aggregated operations

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

```

438
439 // broadcast a commId
440
441 // ...
442
443 ncclCommInitRank(&rootComm, nRanks, rootId, myRank);
444
445 // ...
446
447 ncclGroupStart();
448 ncclAllReduce( /* ... */ );
449 ncclBroadcast( /* ... */ );
450 ncclReduce( /* ... */ );
451 ncclAllGather( /* ... */ );
452 ncclReduceScatter( /* ... */ );
453 ncclGroupEnd();
454
455 // ...
  
```

457 The behavior changes can be described as follow:

- 458 • single `GroupApi` event per thread

- 459     • single KernelLaunch event per thread  
 460     • single Group event per thread

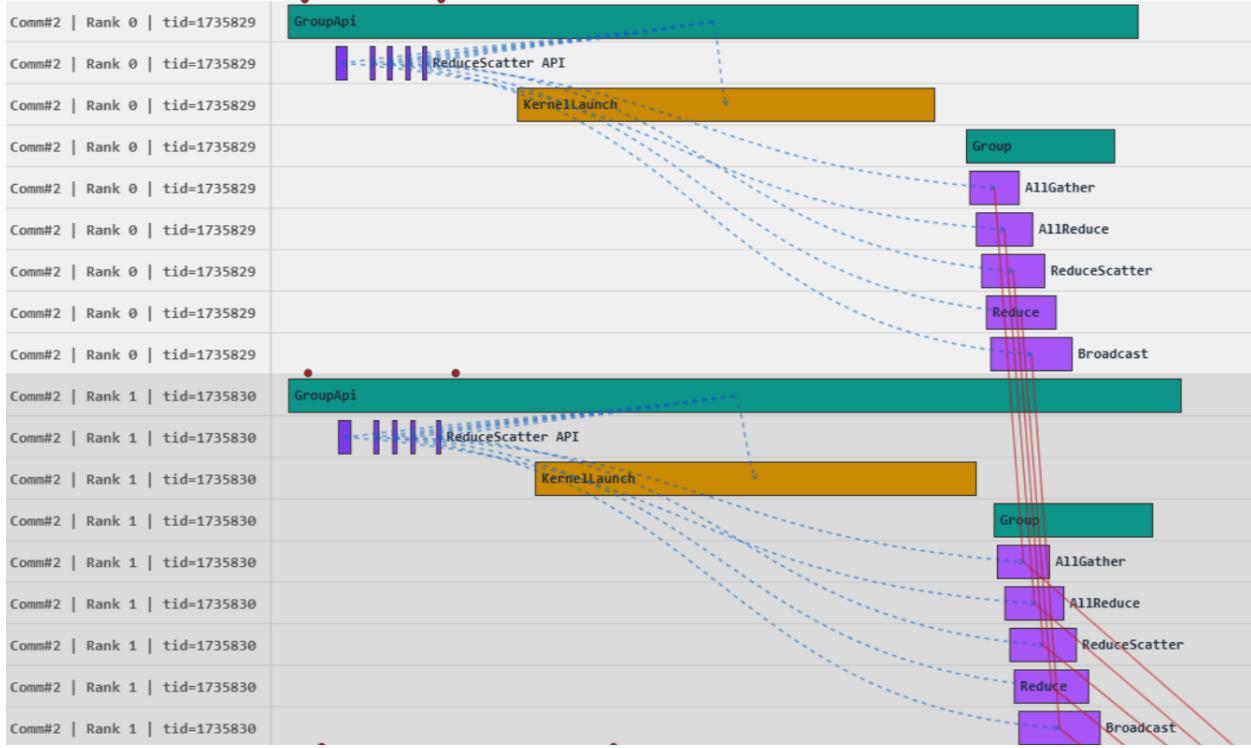


Figure 11: one GPU per thread with aggregated operations: multiple collective calls are grouped together and nccl does only a single kernel launch per thread.

## 461 5 Performance and scalability of the Profiler Plugin API

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

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

```
469 // an 'empty' NCCL Profiler Plugin
470
471 struct MyContext {
472     char dummy;
473 };
474
475 ncclResult_t myInit(void** context, uint64_t commId, int* eActivationMask, const char*
  476   commName, int nNodes, int nranks, int rank, ncclDebugLogger_t logfn) {
```

```

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

```

- 511 For testing the performance overhead in collective and P2P operations, **nccl-tests** from NVIDIA  
 512 was used<sup>21</sup>.
- 513 The applications `sendrecv_perf` and `all_reduce_perf` were launched with following test parameters:  
 514 message size 64 B, 1 000 000 iterations per size, 100 warmup iterations. Single-node jobs used  
 515 one node and 4 GPUs; multi-node jobs used 2 nodes, 4 GPUs per node, 8 MPI ranks in total. For  
 516 each experiment, the application was run once without the profiler and once with the empty profiler  
 517 plugin.
- 518 The Table 1 shows the average latency per operation (time in  $\mu$ s) across iterations. The empty  
 519 profiler adds roughly 8 to 9  $\mu$ s overhead per operation in single-node runs (4 GPUs), but introduces  
 520 negligible overhead in multi-node runs (8 GPUs across 2 nodes).

---

<sup>21</sup><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 ( $\mu$ s)	With profiler ( $\mu$ 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

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

## 523 6 Discussion

524 This section first discusses practical considerations for developers who implement or extend an  
 525 NCCL profiler plugin, as well as known limitations of the current profiling infrastructure, and then  
 526 shows how the plugin could be integrated with the Score-P measurement infrastructure for HPC-  
 527 wide tracing and analysis.

### 528 6.1 Considerations for developers of a Profiler Plugin

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

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

```
535 /src/include/comm.h
536
537 struct ncclComm {
538     uint64_t seqNumber[NCCL_NUM_FUNCTIONS];
539     /* other fields */
540 }
```

```
542 /src/plugin/profiler.cc
543
544 ncclResult_t ncclProfilerStartTaskEvents(struct ncclKernelPlan* plan) {
545     /* other code */
546     __atomic_fetch_add(&plan->comm->seqNumber[ct->func], 1, __ATOMIC_RELAXED);
547     /* other code */
548 }
```

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

552 **Tracing low level activity back to NCCL API calls with `parentObj`.** If a plugin developer  
 553 wants utilize this field, they should ensure that potential address reuse does not create ambiguity  
 554 to what the `parentObj` was originally pointing to. *Custom memory management is advised.* This  
 555 field is useful when trying to understand which user API call triggered which events of lower level  
 556 operations or activity such as network activity (see Fig. 12).

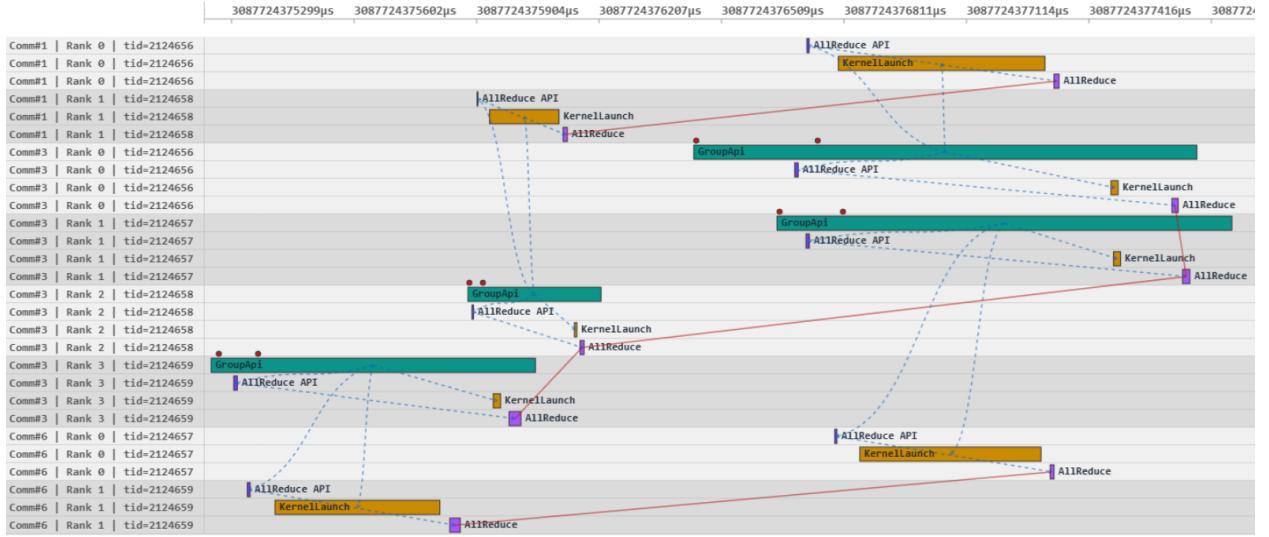


Figure 12: An example illustrating how `parentObj` and `seqNumber` can be used to better understand the timing of concurrent collective operations.

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

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

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

572 parameters (`splitCount`, `color`). Therefore, the relationship cannot be restored based on the ID  
573 alone.

574 **6.2 Known limitations**

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

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

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

585 **6.3 Potential Integration with Score-P**

586 The Score-P measurement infrastructure<sup>22</sup> is a highly scalable and easy-to-use tool suite for profiling  
587 and event tracing of HPC applications. It supports a number of analysis tools. Currently, it works  
588 with Scalasca, Vampir, and Tau and is open for other tools and produces OTF2 traces and CUBE4  
589 profiles. Integrating NCCL into this Score-P allows developers to see communication collectives  
590 alongside the application logic.

591 A prerequisite for distributed tracing is the unique identification of process groups. NCCL achieves  
592 this via `ncclGetUniqueId`<sup>23</sup> without a central coordinator. It generates a handle containing a  
593 random 64-bit `magic` value from `/dev/urandom` and the socket `address` of a new listening socket  
594 (IP, port), whose port is chosen by the operating system. Different MPI tasks or repeated calls in  
595 one process each get distinct random magic and distinct OS-assigned ports are assigned across a  
596 cluster, thus collisions are avoided in practice. Because of this, a Score-P integration can use these  
597 to define Process Groups.

598 The integration could be achieved in two ways, either using a direct Profiler API mapping or via  
599 an indirect NVTX/CUPTI annotation:

600 A direct integration would potentially involve implementing a NCCL profiler plugin that translates  
601 the `startEvent` and `stopEvent` callbacks into Score-P regions: The plugin maps NCCL event de-  
602 scriptors (e.g., `ncclAllReduce`) to Score-P regions using the instrumentation macros (e.g.,  
603 `SCOREP_USER_REGION_BY_NAME_BEGIN/END`).

604 Alternatively, the NCCL profiler plugin can act as a bridge to NVIDIA's Tools Extension (NVTX).  
605 If Score-P has been built with CUDA support it can intercept NVTX ranges. The NCCL profiler

---

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

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

606 plugin would emit `nvtxRangePush`<sup>24</sup> and `nvtxRangePop` around NCCL operations. Score-P records  
 607 these as labeled regions without requiring the plugin to link directly against Score-P libraries. This  
 608 approach decouples the NCCL plugin from the Score-P build environment and instead relies on  
 609 Score-P’s internal NVTX-to-OTF2 mapping logic.

610 The plugin can utilize `cuptiActivityPushExternalCorrelationId` to capture GPU activity during  
 611 the `startEvent` and `stopEvent` of `KernelLaunch` events, while incrementing a thread-safe correla-  
 612 tion ID (see Fig. 13) CUPTI can be initialized and cleaned up within the profiler plugin’s `init` and  
 613 `finalize` functions.

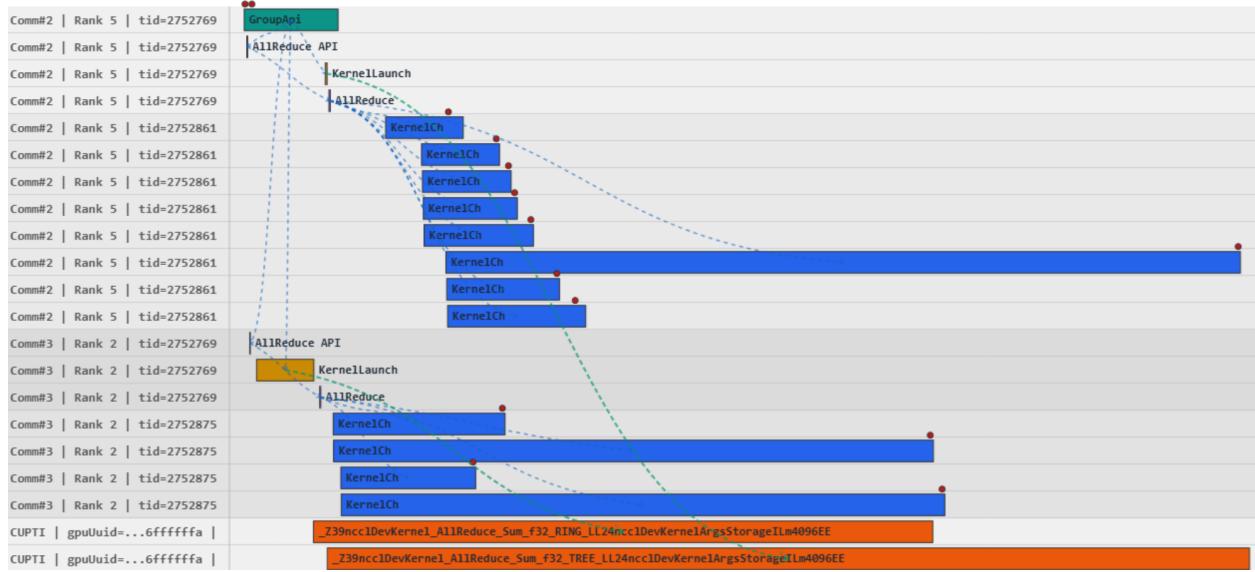


Figure 13: CUPTI activity is visualized as orange event bars. With a unique correlation Id, it is possible to trace the activity back to `KernelLaunch` events

## 614 7 Conclusion

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

<sup>24</sup>[https://nvidia.github.io/NVTX/doxygen/group\\_\\_m\\_a\\_r\\_k\\_e\\_r\\_s\\_\\_a\\_n\\_d\\_\\_r\\_a\\_n\\_g\\_e\\_s.html](https://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`<sup>25</sup> 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.

---

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