CUPTI

DA-05679-001 _v12.3 | October 2023

User's Guide

TABLE OF CONTENTS

LIST OF TABLES

OVERVIEW

The *CUDA Profiling Tools Interface* (CUPTI) enables the creation of profiling and tracing tools that target CUDA applications. CUPTI provides the following APIs: the *Activity API*, the *Callback API*, the *Event API*, the *Metric API*, the *Profiling API*, the *PC Sampling API*, the *SASS Metric API* and the *Checkpoint API*. Using these APIs, you can develop profiling tools that give insight into the CPU and GPU behavior of CUDA applications. CUPTI is delivered as a dynamic library on all platforms supported by CUDA.

In this CUPTI document, **Tracing** refers to the collection of timestamps and additional information for CUDA activities such as CUDA APIs, kernel launches and memory copies during the execution of a CUDA application. Tracing helps in identifying performance issues for the CUDA code by telling you which parts of a program require the most time. Tracing information can be collected using the *Activity* and *Callback* APIs.

In this CUPTI document, **Profiling** refers to the collection of GPU performance metrics for a single kernel or a set of kernels in isolation. Profiling might involve multiple replays of the kernel/s or the entire application to collect GPU performance metrics. For Volta and earlier GPU architectures, these metrics can be collected using CUPTI *Event* and *Metric* APIs. For Volta and later GPU architectures, the low overhead CUPTI *Profiling* and *Perfworks Metric* APIs replace this functionality, and a new CUPTI *PC Sampling* API is supported.

Table 1 Description of CUPTI APIs

CUPTI API	Feature Description
Activity	Asynchronously record CUDA activities, e.g. CUDA API, Kernel, memory copy
Callback	CUDA event callback mechanism to notify subscriber that a specific CUDA event executed e.g. "Entering CUDA runtime memory copy"
Event	Collect kernel performance counters for a kernel execution
Metric	Collect kernel performance metrics for a kernel execution
Profiling	Collect performance metrics for a range of execution
PC Sampling	Collect continuous mode PC Sampling data without serializing kernel execution
SASS Metrics	Collect kernel performance metrics at the source level using SASS patching

CUPTI API	Feature Description
Checkpoint	Provides support for automatically saving and restoring the functional state of the CUDA device

CUPTI Profiling API vs. NVIDIA Nsight Perf SDK

CUPTI Profiling API supports profiling of CUDA kernels and it allows collection of GPU performance metrics for a particular kernel or range of kernels at the CUDA context level. NVIDIA Nsight Perf SDK supports graphics APIs (i.e. DirectX, Vulkan, OpenGL) allowing collection of GPU performance metrics at graphics device, context and queue levels. Both NVIDIA Nsight PerfSDK and CUPTI Profiling API share the host APIs (i.e. metrics enumeration, configuration and evaluation) but differ in which GPU APIs they target on the device.

Chapter 1. RELEASE NOTES

CUPTI Release Notes.

Release notes, including new features and important bug fixes. Supported platforms and GPUs.

1.1. Release Notes

1.1.1. Updates in CUDA 12.3

New Features

- New attributes **`CUPTI_ACTIVITY_OVERHEAD_RUNTIME_TRIGGERED_MODULE_LOADING`** and **`CUPTI_ACTIVITY_OVERHEAD_LAZY_FUNCTION_LOADING`** are added in the activity overhead enum **`CUpti_ActivityOverheadKind`** to provide the overhead information for CUDA runtime triggered module loading and lazy function loading respectively.
- New API **`cuptiGetGraphExecId`** provides the unique ID of the executable graph.
- Added support for collecting graph level trace for device launched graphs. A new API **`cuptiActivityEnableDeviceGraph`** is added to enable the collection of records for device launched graphs.
- CUDA Graphs can be executed on multiple devices i.e. the root node could be launched on one device and the leaf node could be launched on the another device. New fields **`endDeviceId`** and **`endContextId`** are added to identify the ids of device and context respectively which are used to execute the last node of the graph. To accomodate this change, activity record **`CUpti_ActivityGraphTrace`** is deprecated and it is replaced by a new activity record **`CUpti_ActivityGraphTrace2`**.

- Added WSL profiling support on Windows 10 WSL with OS build version 19044 and greater. WSL profiling is not supported on Windows 10 WSL for systems that exceed 1 TB of system memory.
- Several performance improvements are done in the tracing path. One of the key improvements is to allow clients to request CUPTI to maintain the activity buffers at the thread level instead of global buffers. This can be achieved by setting the option **`CUPTI_ACTIVITY_ATTR_PER_THREAD_ACTIVITY_BUFFER`** of the enum **`CUpti_ActivityAttribute`**. This can help in reducing the collection overhead for applications which launch CUDA activities from multiple host threads.
- Frame pointers are enabled for Linux x86_64 libraries.
- The deprecated Activity APIs and structures have been moved to a new header cupti_activity_deprecated.h, which is included in the header cupti_activity.h. Header cupti_activity.h contains only the latest version of APIs and structures.
- CUPTI no longer uses profiling semaphore pool to store the profiling data. Coresponding attributes **`CUPTI_ACTIVITY_ATTR_PROFILING_SEMAPHORE_POOL_SIZE`**, **`CUPTI_ACTIVITY_ATTR_PROFILING_SEMAPHORE_POOL_LIMIT`** and **`CUPTI_ACTIVITY_ATTR_PROFILING_SEMAPHORE_PRE_ALLOCATE_VALUE`** have been deprecated.

Resolved Issues

- Fixed SASS metric profiling for cuda graph.
- Fixed race condition in the API **`cuptiSetThreadIdType`** for late subscription.

1.1.2. Updates in CUDA 12.2 Update 2

New Features

- SASS Metric APIs introduced in the CUDA 12.2 GA release are transitioning from the beta to the production release.
  - Added support for collecting SASS metrics for CUDA Graphs which are launched from host.
  - Added a new field **`numOfDroppedRecords`** in the struct **`CUpti_SassMetricsDisable_Params`** to indicate the number of dropped records when SASS data is flushed prior to calling the disable API.
- Added a new field **`api`** in the struct **`CUpti_Profiler_DeviceSupported_Params`** which can be used to check the configuration support level for profiler APIs like Profiling, PC Sampling and SASS Metric APIs.

Resolved Issues

- Fixed the tracing and profiling support for the GA103 GPU.
- Fixed a hang which can occur when activity buffer gets full while collecting the sampling data using the PC Sampling Activity API.

- Fixed the issue of incorrect timestamps for graph level trace when a graph node is disabled using the APIs **`cuGraphNodeSetEnabled`** or **`cudaGraphNodeSetEnabled`**.

1.1.3. Updates in CUDA 12.2 Update 1

Support for Confidential Computing

CUPTI supports some APIs while running in CC-devtools mode:

- Callback API
- Activity API

The profiling APIs are not supported in CC-devtools mode with this release. Using these APIs should return an error indicating the configuration is not supported:

- Profiling API
- PC Sampling API
- Checkpoint API
- SASS Metrics API

Additionally, CUPTI is not supported at all in full CC mode. CC-devtools mode must be used for tools support.
Some CUDA APIs are not supported or behave differently when running in CC or CC-devtools mode; notably, host pinned memory requests will be traced as managed memory requests, and any CUDA memcopies on these converted pointers are traced as Device to Device copies irrespective of the locality of the source or destination pointers.
For details on how to configure CC or CC-devtools mode, system and software requirements, as well as documentation on CUDA API changes, please see the confidential compute release documentation at https://docs.nvidia.com/confidential-computing/

Resolved Issues

- Fixed timestamps for graph-level tracing for CUDA graphs running across multiple GPUs.
- Fixed a potential hang when CUPTI is unable to fetch attributes for an activity.

1.1.4. Updates in CUDA 12.2

New Features

- A new set of CUPTI APIs for collection of SASS metric data at the source level are provided in the header file `cupti_sass_metrics.h`. These support a larger set of metrics compared to the CUPTI Activity APIs for source-level analysis. SASS to source correlation can be done in the offline mode, similar to the PC sampling APIs. Hence the runtime overhead during data collection is lower. Refer to the section

CUPTI SASS Metrics API for more details. Please note that this is a Beta feature, interface and functionality are subject to change in a future release.

- CUPTI now reports fatal errors, non-fatal errors and warnings instantaneously through callbacks. A new callback domain **CUPTI_CB_DOMAIN_STATE** is added for subscribing to the instantaneous error reporting. Corresponding callback ids are provided in the struct **CUpti_CallbackIdState**.
- Added support for profiling of device graphs and host graphs that launch device graphs. There are some known limitations, please refer to the Known Issues section for details.
- Change in the stream attribute value is communicated by issuing the resource callback. Refer to the struct **CUpti_StreamAttrData** and callback id **CUPTI_CBID_RESOURCE_STREAM_ATTRIBUTE_CHANGED** added in the enum **CUpti_CallbackIdResource**.
- New API **cuptiGetErrorMessage** provides descriptive message for CUPTI error codes.
- Removed the deprecated API **cuptiDeviceGetTimestamp** from the header `cupti_events.h`.
- Added metrics for Tensor core operations to count different types of tensor instructions. These metrics are named as **sm[sp]__ops_path_tensor_src_{src}[_dst_{dst}[_sparsity_{on,off}]].** These are available for devices with compute capability 7.0 and higher, except for Turing TU11x GPUs.

Resolved Issues

- Fixed crash for the graph-level trace for device graphs which are launched from the host.

1.1.5. Updates in CUDA 12.1 Update 1

Resolved Issues

- Fixed CUPTI tracing failure when just-in-time compilation of embedded PTX code is disabled using the environment variable **CUDA_DISABLE_PTX_JIT**.
- Fixed a crash in the API **cuptiFinalize**.

1.1.6. Updates in CUDA 12.1

New Features

- Field **wsl** is added in the struct **CUpti_Profiler_DeviceSupported_Params** to indicate whether Profiling API is supported on Windows Subsystem for Linux (WSL) system or not.

1.1.7. Updates in CUDA 12.0 Update 1

Resolved Issues

- Reduced the host memory overhead by avoiding caching copies of cubin images at the time of loading CUDA modules. Copies of cubin images are now created only when profiling features that need it are enabled.
- By default CUPTI switches back to the device memory, instead of the pinned host memory, for allocation of the profiling buffer for concurrent kernel tracing. This might help in improving the performance of the tracing run. Memory location can be controlled using the attribute `CUPTI_ACTIVITY_ATTR_MEM_ALLOCATION_TYPE_HOST_PINNED` of the activity attribute enum `CUpti_ActivityAttribute`.
- CUPTI now captures the **`cudaGraphLaunch`** API and its kernels when CUPTI is attached after the graph is instantiated using the API **`cudaGraphInstantiate`** but it is attached before the graph is launched using the API **`cudaGraphLaunch`**. Some data in the kernel record would be missing i.e. `cacheConfig`, `sharedMemoryExecuted`, `partitionedGlobalCacheRequested`, `partitionedGlobalCacheExecuted`, `sharedMemoryCarveoutRequested` etc. This fix requires the matching CUDA driver which ships with the CUDA 12.0 Update 1 release.

1.1.8. Updates in CUDA 12.0

New Features

- Added new fields **`maxPotentialClusterSize`** and **`maxActiveClusters`** to help in calculating the cluster occupancy correctly. These fields are valid for devices with compute capability 9.0 and higher. To accomodate this change, activity record **`CUpti_ActivityKernel8`** is deprecated and replaced by a new activity record **`CUpti_ActivityKernel9`**.
- Enhancements for PC Sampling APIs:
  - CUPTI creates few worker threads to offload certain operations like decoding of the hardware data to the CUPTI PC sampling data and correlation of the PC data to the SASS instructions. CUPTI wakes up these threads periodically. To control the sleep time of the worker threads, a new attribute **`CUPTI_PC_SAMPLING_CONFIGURATION_ATTR_TYPE_WORKER_THREAD_PERIODIC_SLEEP_SPA`** is added in the enum **`CUpti_PCSamplingConfigurationAttributeType`**.
  - Improved error reporting for hardware buffer overflow. When hardware buffer overflows, CUPTI returns the out of memory error code. And a new field **`hardwareBufferFull`** added in the struct **`CUpti_PCSamplingData`** is set to differentiate it from other out of memory cases. User can either increase the hardware buffer size or flush the hardware buffer at a higher frequency to avoid overflow.
- Profiling APIs are supported on Windows Subsystem for Linux (WSL) with WSL version 2, NVIDIA display driver version 525 or higher and Windows 11.
- CUPTI support for Kepler GPUs is dropped in CUDA Toolkit 12.0.

Resolved Issues

- Removed minor CUDA version from the SONAME of the CUPTI shared library for compatibility reasons. For example, SONAME of CUPTI library is libcupti.so.12 instead of libcupti.so.12.0 in CUDA 12.0 release.
- Activity kinds **CUPTI_ACTIVITY_KIND_MARKER** and **CUPTI_ACTIVITY_KIND_MARKER_DATA** can be enabled together.

1.1.9. Updates in CUDA 11.8

New Features

- CUPTI adds tracing and profiling support for Hopper and Ada Lovelace GPU families.
- Added new fields **clusterX**, **clusterY**, **clusterZ** and **clusterSchedulingPolicy** to output the Thread Block Cluster dimensions and scheduling policy. These fields are valid for devices with compute capability 9.0 and higher. To accomodate this change, activity record **CUpti_ActivityKernel7** is deprecated and replaced by a new activity record **CUpti_ActivityKernel8**.
- A new activity kind **CUPTI_ACTIVITY_KIND_JIT** and corresponding activity record **CUpti_ActivityJit** are introduced to capture the overhead involved in the JIT (just-in-time) compilation and caching of the PTX or NVVM IR code to the binary code. New record also provides the information about the size and path of the compute cache where the binary code is stored.
- PC Sampling API is supported on Tegra platforms - QNX, Linux (aarch64) and Linux (x86_64) (Drive SDK).

Resolved Issues

- Resolved an issue that might cause crash when the size of the device buffer is changed, using the attribute **CUPTI_ACTIVITY_ATTR_DEVICE_BUFFER_SIZE**, after creation of the CUDA context.

1.1.10. Updates in CUDA 11.7 Update 1

Resolved Issues

- Resolved an issue for PC Sampling API **cuptiPCSamplingGetData** which might not always return all the samples when called after the PC sampling range defined by using the APIs **cuptiPCSamplingStart** and **cuptiPCSamplingStop**. Remaining samples were delivered in the successive call of the API **cuptiPCSamplingGetData** after the next range.
- Disabled tracing of nodes in the CUDA Graph when user enables tracing at the Graph level using the activity kind **CUPTI_ACTIVITY_KIND_GRAPH_TRACE**.
- Fixed missing **channelID** and **channelType** information for kernel records. Earlier these fields were populated for CUDA Graph launches only.

1.1.11. Updates in CUDA 11.7

New Features

- A new activity kind **CUPTI_ACTIVITY_KIND_GRAPH_TRACE** and activity record **CUpti_ActivityGraphTrace** are introduced to represent the execution for a graph without giving visibility about the execution of its nodes. This is intended to reduce overheads involved in tracing each node separately. This activity can only be enabled for drivers of version 515 and above.
- A new API **cuptiActivityEnableAndDump** is added to provide snapshot of certain activities like device, context, stream, NVLink and PCIe at any point during the profiling session.
- Added sample cupti_correlation to show correlation between CUDA APIs and corresponding GPU activities.
- Added sample cupti_trace_injection to show how to build an injection library using the activity and callback APIs which can be used to trace any CUDA application.

Resolved Issues

- Fixed corruption in the function name for PC Sampling API records.
- Fixed incorrect timestamps for GPU activities when user calls the API **cuptiActivityRegisterTimestampCallback** in the late CUPTI attach scenario.
- Fixed incomplete records for device to device memcopies in the late CUPTI attach scenario. This issue manifests mainly when application has a mix of CUDA graph and normal kernel launches.

1.1.12. Updates in CUDA 11.6 Update 1

Resolved Issues

- Fixed hang for the PC Sampling API **cuptiPCSamplingStop**. This issue is seen for the PC sampling start and stop resulting in generation of large number of sampling records.
- Fixed timing issue for specific device to device memcpy operations.

1.1.13. Updates in CUDA 11.6

New Features

- Two new fields **channelID** and **channelType** are added in the activity records for kernel, memcpy, peer-to-peer memcpy and memset to output the ID and type of the hardware channel on which these activities happen. Activity records `CUpti_ActivityKernel6`, `CUpti_ActivityMemcpy4`, `CUpti_ActivityMemcpyPtoP3` and `CUpti_ActivityMemset3` are deprecated and replaced by new activity records `CUpti_ActivityKernel7`,

`CUpti_ActivityMemcpy5`, `CUpti_ActivityMemcpyPtoP4` and `CUpti_ActivityMemset4`.

- New fields **`isMigEnabled`**, **`gpuInstanceId`**, **`computeInstanceId`** and **`migUuid`** are added in the device activity record to provide MIG information for the MIG enabled GPU. Activity record **`CUpti_ActivityDevice3`** is deprecated and replaced by a new activity record **`CUpti_ActivityDevice4`**.
- A new field **`utilizedSize`** is added in the memory pool and memory activity record to provide the utilized size of the memory pool. Activity record **`CUpti_ActivityMemoryPool`** and **`CUpti_ActivityMemory2`** are deprecated and replaced by a new activity record **`CUpti_ActivityMemoryPool2`** and **`CUpti_ActivityMemory3`** respectively.
- API **`cuptiActivityRegisterTimestampCallback`** and callback function **`CUpti_TimestampCallbackFunc`** are added to register a callback function to obtain timestamp of user's choice instead of using CUPTI provided timestamp in activity records.
- Profiling API supports profiling OptiX application.

Resolved Issues

- Fixed multi-pass metric collection using the Profiling API in the auto range and kernel replay mode for Cuda Graph.
- Fixed the performance issue for the PC sampling API **`cuptiPCSamplingStop`**.
- Fixed corruption in variable names for OpenACC activity records.
- Fixed corruption in the fields of the struct **`memoryPoolConfig`** in the activity record **`CUpti_ActivityMemory3`**.
- Filled the fields of the struct **`memoryPoolConfig`** in the activity record **`CUpti_ActivityMemory3`** when a memory pointer allocated via memory pool is released using **`cudaFree`** CUDA API.

1.1.14. Updates in CUDA 11.5 Update 1

Resolved Issues

- Resolved an issue that causes incorrect range name for NVTX event attributes. The issue was introduced in CUDA 11.4.
- Made NVTX initialization APIs **`InitializeInjectionNvtx`** and **`InitializeInjectionNvtx2`** thread-safe.

1.1.15. Updates in CUDA 11.5

New Features

- A new API **`cuptiProfilerDeviceSupported`** is introduced to expose overall Profiling API support and specific requirements for a given device. Profiling API

must be initialized by calling **cuptiProfilerInitialize** before testing device support.

- PC Sampling struct **CUpti_PCSamplingData** introduces a new field **nonUsrKernelsTotalSamples** to provide information about the number of samples collected for all non-user kernels.
- Activity record **CUpti_ActivityDevice2** for device information has been deprecated and replaced by a new activity record **CUpti_ActivityDevice3**. New record adds a flag **isCudaVisible** to indicate whether device is visible to CUDA.
- Activity record **CUpti_ActivityNvLink3** for NVLink information has been deprecated and replaced by a new activity record **CUpti_ActivityNvLink4**. New record can accommodate NVLink port information upto a maximum of 32 ports.
- A new CUPTI Checkpoint API is introduced, enabling automatic saving and restoring of device state, and facilitating development of kernel replay tools. This is helpful for User Replay mode of the CUPTI Profiling API, but is not limited to use with CUPTI.
- Tracing is supported on the Windows Subsystem for Linux version 2 (WSL 2).
- CUPTI is not supported on NVIDIA Crypto Mining Processors (CMP). A new error code **CUPTI_ERROR_CMP_DEVICE_NOT_SUPPORTED** is introduced to indicate it.

Resolved Issues

- Resolved an issue that causes crash for tracing of device to device memcopy operations.
- Resolved an issue that causes crash for OpenACC activity when it is enabled before other activities.

1.1.16. Updates in CUDA 11.4 Update 1

Resolved Issues

- Resolved serialization of CUDA Graph launches for applications which use multiple threads to launch work.
- Previously, for applications that use CUDA Dynamic Parallelism (CDP), CUPTI detects the presence of the CDP kernels in the CUDA module. Even if CDP kernels are not called, it fails to trace the application. There is a change in the behavior, CUPTI now traces all the host launched kernels until it encounters a host launched kernel which launches child kernels. Subsequent kernels are not traced.

1.1.17. Updates in CUDA 11.4

New Features

- Profiling APIs support profiling of the CUDA kernel nodes launched by a CUDA Graph. Auto range profiling with kernel replay mode and user range profiling with

user replay and application replay modes are supported. Other combinations of range profiling and replay modes are not supported.

- Added support for tracing and profiling on NVIDIA virtual GPUs (vGPUs) on an upcoming GRID/vGPU release.
- Added sample profiling_injection to show how to build injection library using the Profiling API.
- Added sample concurrent_profiling to show how to retain the kernel concurrency across streams and devices using the Profiling API.

Resolved Issues

- Resolved the issue of not tracing the device to device memcopy nodes in a CUDA Graph.
- Fixed the issue of reporting zero size for local memory pool for mempool creation record.
- Resolved the issue of non-collection of samples for the default CUDA context for PC Sampling API.
- Enabled tracking of all domains and registered strings in NVTX irrespective of whether the NVTX activity kind or callbacks are enabled. This state tracking is needed for proper working of the tool which creates these NVTX objects before enabling the NVTX activity kind or callback.

1.1.18. Updates in CUDA 11.3

New Features

- A new set of CUPTI APIs for PC sampling data collection are provided in the header file `cupti_pcsampling.h` which support continuous mode data collection without serializing kernel execution and have a lower runtime overhead. Along with these a utility library is provided in the header file `cupti_pcsampling_util.h` which has APIs for GPU assembly to CUDA-C source correlation and for reading and writing the PC sampling data from/to files. Refer to the section CUPTI PC Sampling API for more details.
- Enum `CUpti_PcieGen` is extended to include PCIe Gen 5.
- The following functions are deprecated and will be removed in a future release:
  - Struct `NVPA_MetricsContext` and related APIs `NVPW_MetricsContext_*` from the header `nvperf_host.h`. It is recommended to use the struct `NVPW_MetricsEvaluator` and related APIs `NVPW_MetricsEvaluator_*` instead. Profiling API samples have been updated to show how to use these APIs.
  - `cuptiDeviceGetTimestamp` from the header `cupti_events.h`.

Resolved Issues

- Overhead reduction for tracing of CUDA memcopies.

- To provide normalized timestamps for all activities, CUPTI uses linear interpolation for conversion from GPU timestamps to CPU timestamps. This method can cause spurious gaps or overlap on the timeline. CUPTI improves the conversion function to provide more precise timestamps.
- Generate overhead activity record for semaphore pool allocation.

1.1.19. Updates in CUDA 11.2

New Features

- A new activity kind `CUPTI_ACTIVITY_KIND_MEMORY_POOL` and activity record `CUpti_ActivityMemoryPool` are introduced to represent the creation, destruction and trimming of a memory pool. Enum `CUpti_ActivityMemoryPoolType` lists types of memory pool.
- A new activity kind `CUPTI_ACTIVITY_KIND_MEMORY2` and activity record `CUpti_ActivityMemory2` are introduced to provide separate records for memory allocation and release operations. This helps in correlation of records of these operations to the corresponding CUDA APIs, which otherwise is not possible using the existing activity record `CUpti_ActivityMemory` which provides a single record for both the memory operations.
- Added a new pointer field of type `CUaccessPolicyWindow` in the kernel activity record to provide the access policy window which specifies a contiguous region of global memory and a persistence property in the L2 cache for accesses within that region. To accomodate this change, activity record `CUpti_ActivityKernel5` is deprecated and replaced by a new activity record `CUpti_ActivityKernel6`. This attribute is not collected by default. To control the collection of launch attributes, a new API `cuptiActivityEnableLaunchAttributes` is introdcued.
- New attributes `CUPTI_ACTIVITY_ATTR_DEVICE_BUFFER_PRE_ALLOCATE_VALUE` and `CUPTI_ACTIVITY_ATTR_PROFILING_SEMAPHORE_PRE_ALLOCATE_VALUE` are added in the activity attribute enum `CUpti_ActivityAttribute` to set and get the number of device buffers and profiling semaphore pools which are preallocated for the context.
- CUPTI now allocates profiling buffer for concurrent kernel tracing in the pinned host memory in place of device memory. This might help in improving the performance of the tracing run. Memory location can be controlled using the attribute `CUPTI_ACTIVITY_ATTR_MEM_ALLOCATION_TYPE_HOST_PINNED` of the activity attribute enum `CUpti_ActivityAttribute`.
- The compiler generated line information for inlined functions is improved due to which CUPTI can associate inlined functions with the line information of the function call site that has been inlined.
- Removed support for NVLink performance metrics (**`nvlrx__*`** and **`nvltx__*`**) from the Profiling API due to a potential application hang during data collection. The metrics will be added back in a future CUDA release.

Resolved Issues

- Execution overheads introduced by CUPTI in the tracing path is reduced.
- For the concurrent kernel activity kind `CUPTI_ACTIVITY_KIND_CONCURRENT_KERNEL`, CUPTI instruments the kernel code to collect the timing information. Previously, every kernel in the CUDA module was instrumented, thus the overhead is proportional to the number of different kernels in the module. This is a static overhead which happens at the time of loading the CUDA module. To reduce this overhead, kernels are not instrumented at the module load time, instead a single instrumentation code is generated at the time of loading the CUDA module and it is applied to each kernel during the kernel execution, thus avoiding most of the static overhead at the CUDA module load time.

1.1.20. Updates in CUDA 11.1

New Features

- CUPTI adds tracing and profiling support for the NVIDIA Ampere GPUs with compute capability 8.6.
- Added a new field `graphId` in the activity records for kernel, memcpy, peer-to-peer memcpy and memset to output the unique ID of the CUDA graph that launches the activity through CUDA graph APIs. To accomodate this change, activity records `CUpti_ActivityMemcpy3`, `CUpti_ActivityMemcpyPtoP2` and `CUpti_ActivityMemset2` are deprecated and replaced by new activity records `CUpti_ActivityMemcpy4`, `CUpti_ActivityMemcpyPtoP3` and `CUpti_ActivityMemset3`. And kernel activity record `CUpti_ActivityKernel5` replaces the padding field with `graphId`. Added a new API `cuptiGetGraphId` to query the unique ID of the CUDA graph.
- Added a new API `cuptiActivityFlushPeriod` to set the flush period for the worker thread.
- Added support for profiling cooperative kernels using Profiling APIs.
- Added NVLink performance metrics (nvlrx__* and nvltx__*) using the Profiling APIs. These metrics are available on devices with compute capability 7.0, 7.5 and 8.0, and these can be collected at the context level. Refer to the table Metrics Mapping Table for mapping between earlier CUPTI metrics and the Perfworks NVLink metrics for devices with compute capability 7.0.

Resolved Issues

- Resolved an issue that causes CUPTI to not return full and completed activity buffers for a long time, CUPTI now attempts to return buffers early.
- To reduce the runtime overhead, CUPTI wakes up the worker thread based on certain heuristics instead of waking it up at a regular interval. New API

`cuptiActivityFlushPeriod` can be used to control the flush period of the worker thread. This setting overrides the CUPTI heurtistics.

1.1.21. Updates in CUDA 11.0

New Features

- CUPTI adds tracing and profiling support for devices with compute capability 8.0 i.e. NVIDIA A100 GPUs and systems that are based on A100.
- Enhancements for CUDA Graph:
  - Support to correlate the CUDA Graph node with the GPU activities: kernel, memcpy, memset.
    - Added a new field `graphNodeId` for Node Id in the activity records for kernel, memcpy, memset and P2P transfers. Activity records `CUpti_ActivityKernel4`, `CUpti_ActivityMemcpy2`, `CUpti_ActivityMemset` and `CUpti_ActivityMemcpyPtoP` are deprecated and replaced by new activity records `CUpti_ActivityKernel5`, `CUpti_ActivityMemcpy3`, `CUpti_ActivityMemset2` and `CUpti_ActivityMemcpyPtoP2`.
    - `graphNodeId` is the unique ID for the graph node.
    - `graphNodeId` can be queried using the new CUPTI API `cuptiGetGraphNodeId()`.
    - Callback `CUPTI_CBID_RESOURCE_GRAPHNODE_CREATED` is issued between a pair of the API enter and exit callbacks.
  - Introduced new callback `CUPTI_CBID_RESOURCE_GRAPHNODE_CLONED` to indicate the cloning of the CUDA Graph node.
  - Retain CUDA driver performance optimization in case memset node is sandwiched between kernel nodes. CUPTI no longer disables the conversion of memset nodes into kernel nodes for CUDA graphs.
  - Added support for cooperative kernels in CUDA graphs.
- Added support to trace Optix applications. Refer the Optix Profiling section.
- CUPTI overhead is associated with the thread rather than process. Object kind of the overhead record `CUpti_ActivityOverhead` is switched to `CUPTI_ACTIVITY_OBJECT_THREAD`.
- Added error code `CUPTI_ERROR_MULTIPLE_SUBSCRIBERS_NOT_SUPPORTED` to indicate the presense of another CUPTI subscriber. API `cuptiSubscribe()` returns the new error code than `CUPTI_ERROR_MAX_LIMIT_REACHED`.
- Added a new enum `CUpti_FuncShmemLimitConfig` to indicate whether user has opted in for maximun dynamic shared memory size on devices with compute capability 7.x by using function attributes `CU_FUNC_ATTRIBUTE_MAX_DYNAMIC_SHARED_SIZE_BYTES` or `cudaFuncAttributeMaxDynamicSharedMemorySize` with CUDA driver

and runtime respectively. Field `shmemLimitConfig` in the kernel activity record `CUpti_ActivityKernel5` shows the user choice. This helps in correct occupancy calulation. Value `FUNC_SHMEM_LIMIT_OPTIN` in the enum `cudaOccFuncShmemConfig` is the corresponding option in the CUDA occupancy calculator.

Resolved Issues

- Resolved an issue that causes incorrect or stale timing for memcopy and serial kernel activities.
- Overhead for PC Sampling Activity APIs is reduced by avoiding the reconfiguration of the GPU when PC sampling period doesn't change between successive kernels. This is applicable for devices with compute capability 7.0 and higher.
- Fixed issues in the API `cuptiFinalize()` including the issue which may cause the application to crash. This API provides ability for safe and full detach of CUPTI during the execution of the application. More details in the section Dynamic Detach.

1.1.22. Updates in CUDA 10.2

New Features

- CUPTI allows tracing features for non-root and non-admin users on desktop platforms. Note that events and metrics profiling is still restricted for non-root and non-admin users. More details about the issue and the solutions can be found on this web page.
- CUPTI no longer turns off the performance characteristics of CUDA Graph when tracing the application.
- CUPTI now shows memset nodes in the CUDA graph.
- Fixed the incorrect timing issue for the asynchronous cuMemset/cudaMemset activity.
- Several performance improvements are done in the tracing path.

1.1.23. Updates in CUDA 10.1 Update 2

New Features

- This release is focused on bug fixes and stability of the CUPTI.
- A security vulnerability issue required profiling tools to disable all the features for non-root or non-admin users. As a result, CUPTI cannot profile the application when using a Windows 419.17 or Linux 418.43 or later driver. More details about the issue and the solutions can be found on this web page.

1.1.24. Updates in CUDA 10.1 Update 1

New Features

- Support for the IBM POWER platform is added for the
  - Profiling APIs in the header `cupti_profiler_target.h`
  - Perfworks metric APIs in the headers `nvperf_host.h` and `nvperf_target.h`

1.1.25. Updates in CUDA 10.1

New Features

- This release is focused on bug fixes and performance improvements.
- The new set of profiling APIs and Perfworks metric APIs which were introduced in the CUDA Toolkit 10.0 are now integrated into the CUPTI library distributed in the CUDA Toolkit. Refer to the sections CUPTI Profiling API and Perfworks Metric APIs for documentation of the new APIs.
- Event collection mode `CUPTI_EVENT_COLLECTION_MODE_CONTINUOUS` is now supported on all device classes including Geforce and Quadro.
- Support for the NVTX string registration API `nvtxDomainRegisterStringA()`.
- Added enum `CUpti_PcieGen` to list PCIe generations.

1.1.26. Updates in CUDA 10.0

New Features

- Added tracing support for devices with compute capability 7.5.
- A new set of metric APIs are added for devices with compute capability 7.0 and higher. These provide low and deterministic profiling overhead on the target system. These APIs are currently supported only on Linux x86 64-bit and Windows 64-bit platforms. Refer to the CUPTI web page for documentation and details to download the package with support for these new APIs. Note that both the old and new metric APIs are supported for compute capability 7.0. This is to enable transition of code to the new metric APIs. But one cannot mix the usage of the old and new metric APIs.
- CUPTI supports profiling of OpenMP applications. OpenMP profiling information is provided in the form of new activity records `CUpti_ActivityOpenMp`. New API `cuptiOpenMpInitialize` is used to initialize profiling for supported OpenMP runtimes.
- Activity record for kernel `CUpti_ActivityKernel4` provides shared memory size set by the CUDA driver.
- Tracing support for CUDA kernels, memcpy and memset nodes launched by a CUDA Graph.
- Added support for resource callbacks for resources associated with the CUDA Graph. Refer enum `CUpti_CallbackIdResource` for new callback IDs.

1.1.27. Updates in CUDA 9.2

New Features

- Added support to query PCI devices information which can be used to construct the PCIe topology. See activity kind `CUPTI_ACTIVITY_KIND_PCIE` and related activity record `CUpti_ActivityPcie`.
- To view and analyze bandwidth of memory transfers over PCIe topologies, new set of metrics to collect total data bytes transmitted and recieved through PCIe are added. Those give accumulated count for all devices in the system. These metrics are collected at the device level for the entire application. And those are made available for devices with compute capability 5.2 and higher.
- CUPTI added support for new metrics:
  - Instruction executed for different types of load and store
  - Total number of cached global/local load requests from SM to texture cache
  - Global atomic/non-atomic/reduction bytes written to L2 cache from texture cache
  - Surface atomic/non-atomic/reduction bytes written to L2 cache from texture cache
  - Hit rate at L2 cache for all requests from texture cache
  - Device memory (DRAM) read and write bytes
  - The utilization level of the multiprocessor function units that execute tensor core instructions for devices with compute capability 7.0
- A new attribute `CUPTI_EVENT_ATTR_PROFILING_SCOPE` is added under enum `CUpti_EventAttribute` to query the profiling scope of a event. Profiling scope indicates if the event can be collected at the context level or device level or both. See Enum `CUpti_EventProfilingScope` for avaiable profiling scopes.
- A new error code `CUPTI_ERROR_VIRTUALIZED_DEVICE_NOT_SUPPORTED` is added to indicate that tracing and profiling on virtualized GPU is not supported.

1.1.28. Updates in CUDA 9.1

New Features

- Added a field for correlation ID in the activity record `CUpti_ActivityStream`.

1.1.29. Updates in CUDA 9.0

New Features

- CUPTI extends tracing and profiling support for devices with compute capability 7.0.

- Usage of compute device memory can be tracked through CUPTI. A new activity record `CUpti_ActivityMemory` and activity kind `CUPTI_ACTIVITY_KIND_MEMORY` are added to track the allocation and freeing of memory. This activity record includes fields like virtual base address, size, PC (program counter), timestamps for memory allocation and free calls.
- Unified memory profiling adds new events for thrashing, throttling, remote map and device-to-device migration on 64 bit Linux platforms. New events are added under enum `CUpti_ActivityUnifiedMemoryCounterKind`. Enum `CUpti_ActivityUnifiedMemoryRemoteMapCause` lists possible causes for remote map events.
- PC sampling supports wide range of sampling periods ranging from 2^5 cycles to 2^31 cycles per sample. This can be controlled through new field `samplingPeriod2` in the PC sampling configuration struct `CUpti_ActivityPCSamplingConfig`.
- Added API `cuptiDeviceSupported()` to check support for a compute device.
- Activity record `CUpti_ActivityKernel3` for kernel execution has been deprecated and replaced by new activity record `CUpti_ActivityKernel4`. New record gives information about queued and submit timestamps which can help to determine software and hardware latencies associated with the kernel launch. These timestamps are not collected by default. Use API `cuptiActivityEnableLatencyTimestamps()` to enable collection. New field `launchType` of type `CUpti_ActivityLaunchType` can be used to determine if it is a cooperative CUDA kernel launch.
- Activity record `CUpti_ActivityPCSampling2` for PC sampling has been deprecated and replaced by new activity record `CUpti_ActivityPCSampling3`. New record accomodates 64-bit PC Offset supported on devices of compute capability 7.0 and higher.
- Activity record `CUpti_ActivityNvLink` for NVLink attributes has been deprecated and replaced by new activity record `CUpti_ActivityNvLink2`. New record accomodates increased port numbers between two compute devices.
- Activity record `CUpti_ActivityGlobalAccess2` for source level global accesses has been deprecated and replaced by new activity record `CUpti_ActivityGlobalAccess3`. New record accomodates 64-bit PC Offset supported on devices of compute capability 7.0 and higher.
- New attributes `CUPTI_ACTIVITY_ATTR_PROFILING_SEMAPHORE_POOL_SIZE` and `CUPTI_ACTIVITY_ATTR_PROFILING_SEMAPHORE_POOL_LIMIT` are added in the activity attribute enum `CUpti_ActivityAttribute` to set and get the profiling semaphore pool size and the pool limit.

1.1.30. Updates in CUDA 8.0

New Features

- Sampling of the program counter (PC) is enhanced to point out the true latency issues, it indicates if the stall reasons for warps are actually causing stalls in the issue pipeline. Field `latencySamples` of new activity record `CUpti_ActivityPCSampling2` provides true latency samples. This field is valid for devices with compute capability 6.0 and higher. See section PC Sampling for more details.
- Support for NVLink topology information such as the pair of devices connected via NVLink, peak bandwidth, memory access permissions etc is provided through new activity record `CUpti_ActivityNvLink`. NVLink performance metrics for data transmitted/received, transmit/receive throughput and respective header overhead for each physical link. See section NVLink for more details.
- CUPTI supports profiling of OpenACC applications. OpenACC profiling information is provided in the form of new activity records `CUpti_ActivityOpenAccData`, `CUpti_ActivityOpenAccLaunch` and `CUpti_ActivityOpenAccOther`. This aids in correlating OpenACC constructs on the CPU with the corresponding activity taking place on the GPU, and mapping it back to the source code. New API `cuptiOpenACCInitialize` is used to initialize profiling for supported OpenACC runtimes. See section OpenACC for more details.
- Unified memory profiling provides GPU page fault events on devices with compute capability 6.0 and 64 bit Linux platforms. Enum `CUpti_ActivityUnifiedMemoryAccessType` lists memory access types for GPU page fault events and enum `CUpti_ActivityUnifiedMemoryMigrationCause` lists migration causes for data transfer events.
- Unified Memory profiling support is extended to Mac platform.
- Support for 16-bit floating point (FP16) data format profiling. New metrics inst_fp_16, flop_count_hp_add, flop_count_hp_mul, flop_count_hp_fma, flop_count_hp, flop_hp_efficiency, half_precision_fu_utilization are supported. Peak FP16 flops per cycle for device can be queried using the enum `CUPTI_DEVICE_ATTR_FLOP_HP_PER_CYCLE` added to `CUpti_DeviceAttribute`.
- Added new activity kinds `CUPTI_ACTIVITY_KIND_SYNCHRONIZATION`, `CUPTI_ACTIVITY_KIND_STREAM` and `CUPTI_ACTIVITY_KIND_CUDA_EVENT`, to support the tracing of CUDA synchronization constructs such as context, stream and CUDA event synchronization. Synchronization details are provided in the form of new activity record `CUpti_ActivitySynchronization`. Enum `CUpti_ActivitySynchronizationType` lists different types of CUDA synchronization constructs.
- APIs `cuptiSetThreadIdType()`/`cuptiGetThreadIdType()` to set/get the mechanism used to fetch the thread-id used in CUPTI records. Enum `CUpti_ActivityThreadIdType` lists all supported mechanisms.
- Added API `cuptiComputeCapabilitySupported()` to check the support for a specific compute capability by the CUPTI.

- Added support to establish correlation between an external API (such as OpenACC, OpenMP) and CUPTI API activity records. APIs `cuptiActivityPushExternalCorrelationId()` and `cuptiActivityPopExternalCorrelationId()` should be used to push and pop external correlation ids for the calling thread. Generated records of type `CUpti_ActivityExternalCorrelation` contain both external and CUPTI assigned correlation ids.
- Added containers to store the information of events and metrics in the form of activity records `CUpti_ActivityInstantaneousEvent`, `CUpti_ActivityInstantaneousEventInstance`, `CUpti_ActivityInstantaneousMetric` and `CUpti_ActivityInstantaneousMetricInstance`. These activity records are not produced by the CUPTI, these are included for completeness and ease-of-use. Profilers built on top of CUPTI that sample events may choose to use these records to store the collected event data.
- Support for domains and annotation of synchronization objects added in NVTX v2. New activity record `CUpti_ActivityMarker2` and enums to indicate various stages of synchronization object i.e. `CUPTI_ACTIVITY_FLAG_MARKER_SYNC_ACQUIRE`, `CUPTI_ACTIVITY_FLAG_MARKER_SYNC_ACQUIRE_SUCCESS`, `CUPTI_ACTIVITY_FLAG_MARKER_SYNC_ACQUIRE_FAILED` and `CUPTI_ACTIVITY_FLAG_MARKER_SYNC_RELEASE` are added.
- Unused field `runtimeCorrelationId` of the activity record `CUpti_ActivityMemset` is broken into two fields `flags` and `memoryKind` to indicate the asynchronous behaviour and the kind of the memory used for the memset operation. It is supported by the new flag `CUPTI_ACTIVITY_FLAG_MEMSET_ASYNC` added in the enum `CUpti_ActivityFlag`.
- Added flag `CUPTI_ACTIVITY_MEMORY_KIND_MANAGED` in the enum `CUpti_ActivityMemoryKind` to indicate managed memory.
- API `cuptiGetStreamId` has been deprecated. A new API `cuptiGetStreamIdEx` is introduced to provide the stream id based on the legacy or per-thread default stream flag.

1.1.31. Updates in CUDA 7.5

New Features

- Device-wide sampling of the program counter (PC) is enabled by default. This was a preview feature in the CUDA Toolkit 7.0 release and it was not enabled by default.
- Ability to collect all events and metrics accurately in presence of multiple contexts on the GPU is extended for devices with compute capability 5.x.
- API `cuptiGetLastError` is introduced to return the last error that has been produced by any of the CUPTI API calls or the callbacks in the same host thread.

- Unified memory profiling is supported with MPS (Multi-Process Service)
- Callback is provided to collect replay information after every kernel run during kernel replay. See API `cuptiKernelReplaySubscribeUpdate` and callback type `CUpti_KernelReplayUpdateFunc`.
- Added new attributes in enum `CUpti_DeviceAttribute` to query maximum shared memory size for different cache preferences for a device function.

1.1.32. Updates in CUDA 7.0

New Features

- CUPTI supports device-wide sampling of the program counter (PC). Program counters along with the stall reasons from all active warps are sampled at a fixed frequency in the round robin order. Activity record `CUpti_ActivityPCSampling` enabled using activity kind `CUPTI_ACTIVITY_KIND_PC_SAMPLING` outputs stall reason along with PC and other related information. Enum `CUpti_ActivityPCSamplingStallReason` lists all the stall reasons. Sampling period is configurable and can be tuned using API `cuptiActivityConfigurePCSampling`. This feature is available on devices with compute capability 5.2.
- Added new activity record `CUpti_ActivityInstructionCorrelation` which can be used to dump source locator records for all the PCs of the function.
- All events and metrics for devices with compute capability 3.x and 5.0 can be collected accurately in presence of multiple contexts on the GPU. In previous releases only some events and metrics could be collected accurately when multiple contexts were executing on the GPU.
- Unified memory profiling is enhanced by providing fine grain data transfers to and from the GPU, coupled with more accurate timestamps with each transfer. This information is provided through new activity record `CUpti_ActivityUnifiedMemoryCounter2`, deprecating old record `CUpti_ActivityUnifiedMemoryCounter`.
- MPS tracing and profiling support is extended on multi-gpu setups.
- Activity record `CUpti_ActivityDevice` for device information has been deprecated and replaced by new activity record `CUpti_ActivityDevice2`. New record adds device UUID which can be used to uniquely identify the device across profiler runs.
- Activity record `CUpti_ActivityKernel2` for kernel execution has been deprecated and replaced by new activity record `CUpti_ActivityKernel3`. New record gives information about Global Partitioned Cache Configuration requested and executed. Partitioned global caching has an impact on occupancy calculation. If it is ON, then a CTA can only use a half SM, and thus a half of the registers available per SM. The new fields apply for devices with compute capability 5.2 and higher.

Note that this change was done in CUDA 6.5 release with support for compute capabilty 5.2.

1.1.33. Updates in CUDA 6.5

New Features

- Instruction classification is done for source-correlated Instruction Execution activity `CUpti_ActivityInstructionExecution`. See `CUpti_ActivityInstructionClass` for instruction classes.
- Two new device attributes are added to the activity `CUpti_DeviceAttribute`:
  - `CUPTI_DEVICE_ATTR_FLOP_SP_PER_CYCLE` gives peak single precision flop per cycle for the GPU.
  - `CUPTI_DEVICE_ATTR_FLOP_DP_PER_CYCLE` gives peak double precision flop per cycle for the GPU.
- Two new metric properties are added:
  - `CUPTI_METRIC_PROPERTY_FLOP_SP_PER_CYCLE` gives peak single precision flop per cycle for the GPU.
  - `CUPTI_METRIC_PROPERTY_FLOP_DP_PER_CYCLE` gives peak double precision flop per cycle for the GPU.
- Activity record `CUpti_ActivityGlobalAccess` for source level global access information has been deprecated and replaced by new activity record `CUpti_ActivityGlobalAccess2`. New record additionally gives information needed to map SASS assembly instructions to CUDA C source code. And it also provides ideal L2 transactions count based on the access pattern.
- Activity record `CUpti_ActivityBranch` for source level branch information has been deprecated and replaced by new activity record `CUpti_ActivityBranch2`. New record additionally gives information needed to map SASS assembly instructions to CUDA C source code.
- Sample `sass_source_map` is added to demonstrate the mapping of SASS assembly instructions to CUDA C source code.
- Default event collection mode is changed to Kernel (`CUPTI_EVENT_COLLECTION_MODE_KERNEL`) from Continuous (`CUPTI_EVENT_COLLECTION_MODE_CONTINUOUS`). Also Continuous mode is supported only on Tesla devices.
- Profiling results might be inconsistent when auto boost is enabled. Profiler tries to disable auto boost by default, it might fail to do so in some conditions, but profiling will continue. A new API `cuptiGetAutoBoostState` is added to query the auto boost state of the device. This API returns error `CUPTI_ERROR_NOT_SUPPORTED` on devices that don't support auto boost. Note that auto boost is supported only on certain Tesla devices from the Kepler+ family.

- Activity record `CUpti_ActivityKernel2` for kernel execution has been deprecated and replaced by new activity record `CUpti_ActivityKernel3`. New record additionally gives information about Global Partitioned Cache Configuration requested and executed. The new fields apply for devices with 5.2 Compute Capability.

1.1.34. Updates in CUDA 6.0

New Features

- Two new CUPTI activity kinds have been introduced to enable two new types of source-correlated data collection. The `Instruction Execution` kind collects SASS-level instruction execution counts, divergence data, and predication data. The `Shared Access` kind collects source correlated data indication inefficient shared memory accesses.
- CUPTI provides support for CUDA applications using Unified Memory. A new activity record reports Unified Memory activity such as transfers to and from a GPU and the number of Unified Memory related page faults.
- CUPTI recognized and reports the special MPS context that is used by CUDA applications running on a system with MPS enabled.
- The CUpti_ActivityContext activity record `CUpti_ActivityContext` has been updated to introduce a new field into the structure in a backwards compatible manner. The 32-bit `computeApiKind` field was replaced with two 16 bit fields, `computeApiKind` and `defaultStreamId`. Because all valid `computeApiKind` values fit within 16 bits, and because all supported CUDA platforms are little-endian, persisted context record data read with the new structure will have the correct value for `computeApiKind` and have a value of zero for `defaultStreamId`. The CUPTI client is responsible for versioning the persisted context data to recognize when the `defaultStreamId` field is valid.
- To ensure that metric values are calculated as accurately as possible, a new metric API is introduced. Function `cuptiMetricGetRequiredEventGroupSets` can be used to get the groups of events that should be collected at the same time.
- Execution overheads introduced by CUPTI have been dramatically decreased.
- The new activity buffer API introduced in CUDA Toolkit 5.5 is required. The legacy `cuptiActivityEnqueueBuffer` and `cuptiActivityDequeueBuffer` functions have been removed.

1.1.35. Updates in CUDA 5.5

New Features

- Applications that use CUDA Dynamic Parallelism can be profiled using CUPTI. Device-side kernel launches are reported using a new activity kind.

- Device attributes such as power usage, clocks, thermals, etc. are reported via a new activity kind.
- A new activity buffer API uses callbacks to request and return buffers of activity records. The existing `cuptiActivityEnqueueBuffer` and `cuptiActivityDequeueBuffer` functions are still supported but are deprecated and will be removed in a future release.
- The Event API supports kernel replay so that any number of events can be collected during a single run of the application.
- A new metric API `cuptiMetricGetValue2` allows metric values to be calculated for any device, even if that device is not available on the system.
- CUDA peer-to-peer memory copies are reported explicitly via the activity API. In previous releases these memory copies were only partially reported.

1.2. Known Issues

The following are known issues with the current release.

- A security vulnerability issue required profiling tools to disable features using GPU performance counters for non-root or non-admin users when using a Windows 419.17 or Linux 418.43 or later driver. By default, NVIDIA drivers require elevated permissions to access GPU performance counters. On Tegra platforms, profile as root or using sudo. On other platforms, you can either start profiling as root or using sudo, or by enabling non-admin profiling. More details about the issue and the solutions can be found on the ERR_NVGPUCTRPERM web page.

  

  CUPTI allows tracing features for non-root and non-admin users on desktop platforms only, Tegra platforms require root or sudo access.

- Profiling results might be inconsistent when auto boost is enabled. Profiler tries to disable auto boost by default. But it might fail to do so in some conditions and profiling will continue and results will be inconsistent. API **`cuptiGetAutoBoostState()`** can be used to query the auto boost state of the device. This API returns error `CUPTI_ERROR_NOT_SUPPORTED` on devices that don't support auto boost. Note that auto boost is supported only on certain Tesla devices with compute capability 3.0 and higher.
- CUPTI doesn't populate the activity structures which are deprecated, instead the newer version of the activity structure is filled with the information.
- Because of the low resolution of the timer on Windows, the start and end timestamps can be same for activities having short execution duration on Windows.
- The application which calls CUPTI APIs cannot be used with Nvidia tools like `nvprof`, `Nvidia Visual Profiler`, `Nsight Compute`, `Nsight Systems`, `Nvidia Nsight Visual Studio Edition`, `cuda-gdb` and `cuda-memcheck`.

- PCIe and NVLink records, when enabled using the API **`cuptiActivityEnable`**, are not captured when CUPTI is initialized lazily after the CUDA initialization. API **`cuptiActivityEnableAndDump`** can be used to dump the records for these activities at any point during the profiling session.
- CUPTI fails to profile the OpenACC application when the OpenACC library linked with the application has missing definition of the OpenACC API routine/s. This is indicated by the error code `CUPTI_ERROR_OPENACC_UNDEFINED_ROUTINE`.
- OpenACC profiling might fail when OpenACC library is linked statically in the user application. This happens due to the missing definition of the OpenACC API routines needed for the OpenACC profiling, as compiler might ignore definitions for the functions not used in the application. This issue can be mitigated by linking the OpenACC library dynamically.
- Unified memory profiling is not supported on the ARM architecture.
- Profiling a C++ application which overloads the new operator at the global scope and uses any CUDA APIs like cudaMalloc() or cudaMallocManaged() inside the overloaded new operator will result in a hang.
- Devices with compute capability 6.0 and higher introduce a new feature, compute preemption, to give fair chance for all compute contexts while running long tasks. With compute preemption feature-
  - If multiple contexts are running in parallel it is possible that long kernels will get preempted.
  - Some kernels may get preempted occasionally due to timeslice expiry for the context.

  If kernel has been preempted, the time the kernel spends preempted is still counted towards kernel duration.

  Compute preemption can affect events and metrics collection. The following are known issues with the current release:

  - Events and metrics collection for a MPS client can result in higher counts than expected on devices with compute capability 7.0 and higher, since MPS client may get preempted due to termination of another MPS client.
  - Events warps_launched and sm_cta_launched and metric inst_per_warp might provide higher counts than expected on devices with compute capability 6.0 and higher. Metric unique_warps_launched can be used in place of warps_launched to get correct count of actual warps launched as it is not affected by compute preemption.

  To avoid compute preemption affecting profiler results try to isolate the context being profiled:

  - Run the application on secondary GPU where display is not connected.
  - On Linux if the application is running on the primary GPU where the display driver is connected then unload the display driver.

    - Run only one process that uses GPU at one time.
- Devices with compute capability 6.0 and higher support demand paging. When the kernel is scheduled for the first time, all the pages allocated using cudaMallocManaged and that are required for execution of the kernel are fetched in the global memory when GPU faults are generated. Profiler requires multiple passes to collect all the metrics required for kernel analysis. The kernel state needs to be saved and restored for each kernel replay pass. For devices with compute capability 6.0 and higher and platforms supporting Unified memory, in the first kernel iteration the GPU faults will be generated and all pages will be fetched in the global memory. Second iteration onwards GPU page faults will not occur. This will significantly affect the memory related events and timing. The time taken from trace will include the time required to fetch the pages but most of the metrics profiled in multiple iterations will not include time/cycles required to fetch the pages. This causes inconsistency in the profiler results.
- When profiling an application that uses CUDA Dynamic Parallelism (CDP) there are several limitations to the profiling tools. CUDA 12.0 adds support for revamped CUDA Dynamic Parallelism APIs (referred to as CDP2), offering substantial performance improvements vs. the legacy CUDA Dynamic Parallelism APIs (referred to as CDP1).
    - For Legacy CUDA Dynamic Parallelism (CDP1), CUPTI supports tracing of all host and device kernels for devices with compute capability 5.x and 6.x. For devices with compute capability 7.0 and higher, CUPTI traces all the host launched kernels until it encounters a host launched kernel which launches child kernels; subsequent kernels are not traced.
    - For CUDA Dynamic Parallelism (CDP2), CUPTI supports tracing of host launched kernels only, it can't trace device launched kernels.
    - CUPTI doesn't report CUDA API calls for device launched kernels.
    - CUPTI doesn't support profiling of device launched kernels i.e. it doesn't report detailed event, metric, and source-level results for device launched kernels. Event, metric, and source-level results collected for CPU-launched kernels will include event, metric, and source-level results for the entire call-tree of kernels launched from within that kernel.
- When profiling an application that uses CUDA Device Graphs, there are some limitations to the profiling tools.
    - CUPTI traces the device graph when it is launched from the host. When the graph is launched from the device, graph level tracing is supported, but node level tracing is not.
- Compilation of samples autorange_profiling and userrange_profiling requires a host compiler which supports C++11 features. For some g++ compilers, it is required to use the flag -std=c++11 to turn on C++11 features.
- PC Sampling Activity API is not supported on Tegra platforms, while PC Sampling API is supported on Tegra platforms.

- As of CUDA 11.4 and R470 TRD1 driver release, CUPTI is supported in a vGPU environment which requires a vGPU license. If the license is not obtained after 20 minutes, the reported performance data including metrics from the GPU will be inaccurate. This is because of a feature in vGPU environment which reduces performance but retains functionality as specified here.
- CUPTI is not supported on NVIDIA Crypto Mining Processors (CMP). This is reported using the error code **`CUPTI_ERROR_CMP_DEVICE_NOT_SUPPORTED`**. For more information, please visit the web page.
- CUPTI versions shipped in the CUDA Toolkit 11.7 and CUDA Toolkit 11.8 don't support Kepler (sm_35 and sm_37) devices. Refer to the webpages CUPTI 11.7 and CUPTI 11.8 for location of the CUPTI packages having the support for these Kepler devices.
- Support for the GA103 GPU was added in the CUDA 11.6 release but it was broken for releases from CUDA 11.8 to CUDA 12.2 Update 1.
- Unified memory profiling is broken for Maxwell devices on Windows platform.

1.2.1. Profiling

The following are common known issues for both the event and metric APIs and the profiling APIs:

- Profiling may significantly change the overall performance characteristics of the application. Refer to the section CUPTI Overhead for more details.
- Profiling a kernel while other contexts are active on the same device (e.g. X server, or secondary CUDA or graphics application) can result in varying metric values for L2/FB (Device Memory) related metrics. Specifically, L2/FB traffic from non-profiled contexts cannot be excluded from the metric results. To completely avoid this issue, profile the application on a GPU without secondary contexts accessing the same device (e.g. no X server on Linux).
- Profiling is not supported for multidevice cooperative kernels, that is, kernels launched by using the API functions `cudaLaunchCooperativeKernelMultiDevice` or `cuLaunchCooperativeKernelMultiDevice`.
- Enabling certain events can cause GPU kernels to run longer than the driver's watchdog time-out limit. In these cases the driver will terminate the GPU kernel resulting in an application error and profiling data will not be available. Please disable the driver watchdog time out before profiling such long running CUDA kernels
  - On Linux, setting the X Config option Interactive to false is recommended.
  - For Windows, detailed information about TDR (Timeout Detection and Recovery) and how to disable it is available at https://docs.microsoft.com/en-us/windows-hardware/drivers/display/timeout-detection-and-recovery

1.2.1.1. Event and Metric API

The following are known issues related to Event and Metric API:

- The CUPTI event APIs from the header `cupti_events.h` and metric APIs from the header `cupti_metrics.h` are not supported for the devices with compute capability 7.5 and higher. These are replaced by Profiling API and Perfworks metric API. Refer to the section Migration to the Profiling API.
- While collecting events in continuous mode, event reporting may be delayed i.e. event values may be returned by a later call to readEvent(s) API and the event values for the last readEvent(s) API may get lost.
- When profiling events, it is possible that the domain instance that gets profiled gives event value 0 due to absence of workload on the domain instance since CUPTI profiles one instance of the domain by default. To profile all instances of the domain, user can set event group attribute `CUPTI_EVENT_GROUP_ATTR_PROFILE_ALL_DOMAIN_INSTANCES` through API **`cuptiEventGroupSetAttribute()`**.
- Profiling results might be incorrect for CUDA applications compiled with nvcc version older than 9.0 for devices with compute capability 6.0 and 6.1. Profiling session will continue and CUPTI will notify it using error code `CUPTI_ERROR_CUDA_COMPILER_NOT_COMPATIBLE`. It is advised to recompile the application code with nvcc version 9.0 or later. Ignore this warning if code is already compiled with the recommended nvcc version.
- For some metrics, the required events can only be collected for a single CUDA context. For an application that uses multiple CUDA contexts, these metrics will only be collected for one of the contexts. The metrics that can be collected only for a single CUDA context are indicated in the metric reference tables.
- Some metric values are calculated assuming a kernel is large enough to occupy all device multiprocessors with approximately the same amount of work. If a kernel launch does not have this characteristic, then those metric values may not be accurate.
- Some events and metrics are not available on all devices. For list of metrics, you can refer to the metric reference tables.
- CUPTI can give out of memory error for event and metrics profiling, it could be due to large number of instructions in the kernel.
- Profiling is not supported for CUDA kernel nodes launched by a CUDA Graph.
- These APIs are not supported on below system configurations:
  - 64-bit ARM Server CPU architecture (arm64 SBSA).
  - Virtual GPUs (vGPU).
  - Windows Subsystem for Linux (WSL).

1.2.1.2. Profiling and Perfworks Metric API

The following are known issues related to the Profiling and Perfworks Metric API:

- Profiling a kernel while any other GPU work is executing on the same MIG compute instance can result in varying metric values for all units. Care should be taken to serialize, or otherwise prevent concurrent CUDA launches within the target application to ensure those kernels do not influence each other. Be aware that GPU work issued through other APIs in the target process or workloads created by non-target processes running simultaneously in the same MIG compute instance will influence the collected metrics. Note that it is acceptable to run CUDA processes in other MIG compute instances as they will not influence the profiled MIG compute instance.
- For devices with compute capability 8.0, the NVLink topology information is available but NVLink performance metrics (**`nvlrx__*`** and **`nvltx__*`**) are not supported due to a potential application hang during data collection.
- Profiling is not supported under MPS (Multi-Process Service).
- For profiling the CUDA kernel nodes launched by a CUDA Graph, not all combinations of range profiling and replay modes are supported. Here are some limitations:
  - User replay and application replay modes with auto range are not supported.
  - In the user range mode, entire graph is profiled as one workload i.e. all the kernel nodes launched by the CUDA Graph will be profiled and single result will be provided, user can't do the profiling for a range of kernels.
  - For Device Graph profiling in the auto range and kernel replay mode, each kernel node will be profiled except for the nodes which launch device graphs.
- Profiling kernels executed on a device that is part of an SLI group is not supported.
- Refer to the section for differences from event and metric APIs.
- Profiling on Windows Subsystem for Linux (WSL) is only supported with WSL version 2, NVIDIA display driver version 525 or higher and Windows 11.

1.3. Support

Information on supported platforms and GPUs.

1.3.1. Platform Support

Table 2 Platforms supported by CUPTI

Platform	Support
Windows	Yes

Platform	Support
Windows Subsystem for Linux version 2 (WSL 2)	Yes*
Linux (x86_64)	Yes
Linux (ppc64le)	Yes
Linux (aarch64 sbsa)	Yes*
Linux (x86_64) (Drive SDK)	Yes*
Linux (aarch64)	Yes*
QNX	Yes*
Mac OSX	No
Android	No

Tracing and profiling of 32-bit processes is not supported.

Event and Metric APIs are not supported on Linux (aarch64 sbsa) and WSL 2 platforms.

1.3.2. GPU Support

Table 3 GPU architectures supported by different CUPTI APIs

CUPTI API	Supported GPU architectures	Notes
Activity	Maxwell and later GPU architectures, i.e. devices with compute capability 5.0 and higher	
Callback	Maxwell and later GPU architectures, i.e. devices with compute capability 5.0 and higher	
Event	Maxwell, Pascal, Volta	Not supported on Turing and later GPU architectures, i.e. devices with compute capability 7.5 and higher
Metric	Maxwell, Pascal, Volta	Not supported on Turing and later GPU architectures, i.e. devices with compute capability 7.5 and higher
Profiling	Volta and later GPU architectures, i.e. devices with compute capability 7.0 and higher	Not supported on Maxwell and Pascal GPUs
PC Sampling	Volta and later GPU architectures, i.e. devices with compute capability 7.0 and higher	Not supported on Maxwell and Pascal GPUs
SASS Metric	Volta and later GPU architectures, i.e. devices with compute capability 7.0 and higher	Not supported on Maxwell and Pascal GPUs
Checkpoint	Volta and later GPU architectures, i.e. devices with compute capability 7.0 and higher	Not supported on Maxwell and Pascal GPUs

Chapter 2. USAGE

2.1. CUPTI Compatibility and Requirements

New versions of the CUDA driver are backwards compatible with older versions of CUPTI. For example, a developer using a profiling tool based on CUPTI 10.0 can update to a more recently released CUDA driver. Refer to the table CUDA Toolkit and Compatible Driver Versions for minimum version of the CUDA driver required for each release of CUPTI from the corresponding CUDA Toolkit release. CUPTI calls will fail with error code `CUPTI_ERROR_NOT_INITIALIZED` if the CUDA driver version is not compatible with the CUPTI version.

2.2. CUPTI Initialization

CUPTI initialization occurs lazily the first time you invoke any CUPTI function. For the Activity, Event, Metric, and Callback APIs there are no requirements on when this initialization must occur (i.e. you can invoke the first CUPTI function at any point). See the CUPTI Activity API section for more information on CUPTI initialization requirements for the activity API.

It is recommended for CUPTI clients to call the API `cuptiSubscribe()` before starting the profiling session i.e. API `cuptiSubscribe()` should be called before calling any other CUPTI API. This API will return the error code `CUPTI_ERROR_MULTIPLE_SUBSCRIBERS_NOT_SUPPORTED` when another CUPTI client is already subscribed. CUPTI client should error out and not make further CUPTI calls if `cuptiSubscribe()` returns an error. This would prevent multiple CUPTI clients to be active at the same time otherwise those might interfere with the profiling state of each other.

2.3. CUPTI Activity API

The CUPTI Activity API allows you to asynchronously collect a trace of an application's CPU and GPU CUDA activity. The following terminology is used by the activity API.

Activity Record

CPU and GPU activity is reported in C data structures called activity records. There is a different C structure type for each activity kind (e.g. `CUpti_ActivityAPI`). Records are generically referred to using the `CUpti_Activity` type. This type contains only a field that indicates the kind of the activity record. Using this kind, the object can be cast from the generic `CUpti_Activity` type to the specific type representing the activity. See the `printActivity` function in the activity_trace_async sample for an example.

Activity Buffer

An activity buffer is used to transfer one or more activity records from CUPTI to the client. CUPTI fills activity buffers with activity records as the corresponding activities occur on the CPU and GPU. But CUPTI doesn't guarantee any ordering of the activities in the activity buffer as activity records for few activity kinds are added lazily. The CUPTI client is responsible for providing empty activity buffers as necessary to ensure that no records are dropped.

An *asynchronous* buffering API is implemented by `cuptiActivityRegisterCallbacks` and `cuptiActivityFlushAll`.

It is not required that the activity API be initalized before CUDA initialization. All related activities occuring after initializing the activity API are collected. You can force initialization of the activity API by enabling one or more activity kinds using `cuptiActivityEnable` or `cuptiActivityEnableContext`, as shown in the `initTrace` function of the activity_trace_async sample. Some activity kinds cannot be directly enabled, see the API documentation for `CUpti_ActivityKind` for details. The functions `cuptiActivityEnable` and `cuptiActivityEnableContext` will return `CUPTI_ERROR_NOT_COMPATIBLE` if the requested activity kind cannot be enabled.

The activity buffer API uses callbacks to request and return buffers of activity records. To use the asynchronous buffering API, you must first register two callbacks using `cuptiActivityRegisterCallbacks`. One of these callbacks will be invoked whenever CUPTI needs an empty activity buffer. The other callback is used to deliver a buffer containing one or more activity records to the client. To minimize profiling overhead the client should return as quickly as possible from these callbacks. Client can pre-allocate a pool of activity buffers and return an empty buffer from the pool when requested by CUPTI. Activity buffer size should be chosen carefully, smaller buffers can result in frequent requests by CUPTI and bigger buffers can delay the automatic delivery of completed activity buffers. For typical workloads, it's suggested to choose a size between 1 and 10 MB. The functions `cuptiActivityGetAttribute` and

`cuptiActivitySetAttribute` can be used to read and write attributes that control how the buffering API behaves. See the API documentation for more information.

Flushing of the activity buffers

CUPTI is expected to deliver the activity buffer automatically as soon as it gets full and all the activity records in it are completed. For performance reasons, CUPTI calls the underlying methods based on certain heuristics, thus it can cause delay in the delivery of the buffer. However client can make a request to deliver the activity buffer/s at any time, and this can be achieved using the APIs `cuptiActivityFlushAll` and `cuptiActivityFlushPeriod`. Behavior of these APIs is as follows:

- For on-demand flush using the API `cuptiActivityFlushAll` with the flag set as 0, CUPTI returns all the activity buffers which have all the activity records completed, buffers need not to be full though. It doesn't return buffers which have one or more incomplete records. This flush can be done at a regular interval in a separate thread.
- For on-demand forced flush using the API `cuptiActivityFlushAll` with the flag set as CUPTI_ACTIVITY_FLAG_FLUSH_FORCED, CUPTI returns all the activity buffers including the ones which have one or more incomplete activity records. It's suggested to do the forced flush before the termination of the profiling session to allow remaining buffers to be delivered.
- For periodic flush using the API `cuptiActivityFlushPeriod`, CUPTI returns only those activity buffers which are full and have all the activity records completed. It's allowed to use the API `cuptiActivityFlushAll` to flush the buffers on-demand, even when client sets the periodic flush.

Note that activity record is considered as completed if it has all the information filled up including the timestamps (if any).

The activity_trace_async sample shows how to use the activity buffer API to collect a trace of CPU and GPU activity for a simple application.

CUPTI Threads

CUPTI creates a worker thread to minimize the perturbance for the application created threads. CUPTI offloads certain operations from the application threads to the worker thread, this incldues synchronization of profiling resources between host and device, delivery of the activity buffers to the client using the buffer completed callback registered in the API `cuptiActivityRegisterCallbacks` etc. To minimize the overhead, CUPTI wakes up the worker thread based on certain heuristics. API `cuptiActivityFlushPeriod` introduced in CUDA 11.1 can be used to control the flush period of the worker thread. This setting overrides the CUPTI heurtistics. It's allowed to use the API `cuptiActivityFlushAll` to flush the data on-demand, even when client sets the periodic flush.

Further, CUPTI creates separate threads when certain activity kinds are enabled. For example, CUPTI creates one thread each for activity

kinds CUPTI_ACTIVITY_KIND_UNIFIED_MEMORY_COUNTER and CUPTI_ACTIVITY_KIND_ENVIRONMENT to collect the information from the backend.

2.3.1. SASS Source Correlation

While high-level languages for GPU programming like CUDA C offer a useful level of abstraction, convenience, and maintainability, they inherently hide some of the details of the execution on the hardware. It is sometimes helpful to analyze performance problems for a kernel at the assembly instruction level. Reading assembly language is tedious and challenging; CUPTI can help you to build the correlation between lines in your high-level source code and the executed assembly instructions.

Building SASS source correlation for a PC can be split into two parts:

- Correlation of the PC to SASS instruction - subscribe to any one of the CUPTI_CBID_RESOURCE_MODULE_LOADED, CUPTI_CBID_RESOURCE_MODULE_UNLOAD_STARTING, or CUPTI_CBID_RESOURCE_MODULE_PROFILED callbacks. This returns a CUpti_ModuleResourceData structure having the CUDA binary. The binary can be disassembled using the nvdisasm utility that comes with the CUDA toolkit. An application can have multiple functions and modules, to uniquely identify there is a functionId field in all source level activity records. This uniquely corresponds to a CUPTI_ACTIVITY_KIND_FUNCTION, which has the unique module ID and function ID in the module.
- Correlation of the SASS instruction to CUDA source line - every source level activity has a sourceLocatorId field which uniquely maps to a record of kind CUPTI_ACTIVITY_KIND_SOURCE_LOCATOR, containing the line and file name information. Please note that multiple PCs can correspond to a single source line.

When any source level activity (global access, branch, PC Sampling, etc.) is enabled, a source locator record is generated for the PCs that have the source level results. The record CUpti_ActivityInstructionCorrelation can be used, along with source level activities, to generate SASS assembly instructions to CUDA C source code mapping for all the PCs of the function, and not just the PCs that have the source level results. This can be enabled using the activity kind CUPTI_ACTIVITY_KIND_INSTRUCTION_CORRELATION.

The sass_source_map sample shows how to map SASS assembly instructions to CUDA C source.

2.3.2. PC Sampling

CUPTI supports device-wide sampling of the program counter (PC). The PC Sampling gives the number of samples for each source and assembly line with various stall reasons. Using this information, you can pinpoint portions of your kernel that are introducing latencies and the reason for the latency. Samples are taken in round robin

order for all active warps at a fixed number of cycles, regardless of whether the warp is issuing an instruction or not.

Devices with compute capability 6.0 and higher have a new feature that gives latency reasons. The latency samples indicate the reasons for holes in the issue pipeline. While collecting these samples, there is no instruction issued in the respective warp scheduler, hence these give the latency reasons. The latency reasons will be one of the stall reasons listed in the enum `CUpti_ActivityPCSamplingStallReason`, except stall reason `CUPTI_ACTIVITY_PC_SAMPLING_STALL_NOT_SELECTED`.

The activity record `CUpti_ActivityPCSampling3`, enabled using activity kind `CUPTI_ACTIVITY_KIND_PC_SAMPLING`, outputs the stall reason along with PC and other related information. The enum `CUpti_ActivityPCSamplingStallReason` lists all the stall reasons. Sampling period is configurable and can be tuned using API `cuptiActivityConfigurePCSampling`. A wide range of sampling periods, ranging from 2^5 cycles to 2^31 cycles per sample, is supported. This can be controlled through the field `samplingPeriod2` in the PC sampling configuration struct `CUpti_ActivityPCSamplingConfig`. The activity record `CUpti_ActivityPCSamplingRecordInfo` provides the total and dropped samples for each kernel profiled for PC sampling.

This feature is available on devices with compute capability 5.2 and higher, excluding mobile devices. For Pascal and older chips `cuptiActivityConfigurePCSampling` api must be called before enabling activity kind `CUPTI_ACTIVITY_KIND_PC_SAMPLING`, for Volta and newer chips order does not matter. For Volta and newer GPU architectures if `cuptiActivityConfigurePCSampling` API is called in the middle of execution, PC sampling configuration will be updated for subsequent kernel launches. PC sampling can significantly change the overall performance characteristics of the application because all kernel executions are serialized on the GPU.

The pc_sampling sample shows how to use these APIs to collect PC Sampling profiling information for a kernel.

A new set of PC Sampling APIs was introduced in the CUDA 11.3 release, which supports continuous mode data collection without serializing kernel execution and have a lower runtime overhead. Refer to the section CUPTI PC Sampling API for more details. PC Sampling APIs from the header `cupti_activity.h` would be referred as *PC Sampling Activity APIs* and APIs from the header `cupti_pcsampling.h` would be referred as *PC Sampling APIs*.

2.3.3. NVLink

NVIDIA NVLink is a high-bandwidth, energy-efficient interconnect that enables fast communication between the CPU and GPU, and between GPUs. CUPTI provides NVLink topology information and NVLink transmit/receive throughput metrics.

The activity record `CUpti_ActivityNVLink3`, enabled using activity kind `CUPTI_ACTIVITY_KIND_NVLink`, outputs NVLink topology information in terms of logical NVLinks. A logical NVLink is connected between 2 devices, the device can be of type NPU (NVLink Processing Unit), which can be CPU or GPU. Each device can support up to 12 NVLinks, hence one logical link can comprise of 1 to 12 physical NVLinks. The field `physicalNvLinkCount` gives the number of physical links in this logical link. The fields `portDev0` and `portDev1` give information about the slot in which physical NVLinks are connected for a logical link. This port is the same as the instance of NVLink metrics profiled from a device. Therefore, port and instance information should be used to correlate the per-instance metric values with the physical NVLinks, and in turn to the topology. The field `flag` gives the properties of a logical link, whether the link has access to system memory or peer device memory, and has capabilities to do system memory or peer memmory atomics. The field `bandwidth` gives the bandwidth of the logical link in kilobytes/sec.

CUPTI provides some metrics for each physical link. Metrics are provided for data transmitted/received, transmit/receive throughput, and header versus user data overhead for each physical NVLink. These metrics are also provided per packet type (read/write/ atomics/response) to get more detailed insight in the NVLink traffic.

This feature is available on devices with compute capability 6.0 and 7.0. For devices with compute capability 8.0, the NVLink topology information is available but NVLink performance metrics (**`nvlrx__*`** and **`nvltx__*`**) are not supported due to a potential application hang during data collection.

The nvlink_bandwidth sample shows how to use these APIs to collect NVLink metrics and topology, as well as how to correlate metrics with the topology.

2.3.4. OpenACC

CUPTI supports collecting information for OpenACC applications using the OpenACC tools interface implementation of the PGI runtime. OpenACC profiling is available only on Linux x86_64, IBM POWER and Arm server platform (arm64 SBSA) platforms. This feature also requires PGI runtime version 19.1 or higher.

The activity records `CUpti_ActivityOpenAccData`, `CUpti_ActivityOpenAccLaunch`, and `CUpti_ActivityOpenAccOther` are created, representing the three groups of callback events specified in the OpenACC tools interface. `CUPTI_ACTIVITY_KIND_OPENACC_DATA`, `CUPTI_ACTIVITY_KIND_OPENACC_LAUNCH`, and

CUPTI_ACTIVITY_KIND_OPENACC_OTHER can be enabled to collect the respective activity records.

Due to the restrictions of the OpenACC tools interface, CUPTI cannot record OpenACC records from within the client application. Instead, a shared library that exports the acc_register_library function defined in the OpenACC tools interface specification must be implemented. Parameters passed into this function from the OpenACC runtime can be used to initialize the CUPTI OpenACC measurement using cuptiOpenACCInitialize. Before starting the client application, the environment variable ACC_PROFLIB must be set to point to this shared library.

cuptiOpenACCInitialize is defined in cupti_openacc.h, which is included by cupti_activity.h. Since the CUPTI OpenACC header is only available on supported platforms, CUPTI clients must define CUPTI_OPENACC_SUPPORT when compiling.

The openacc_trace sample shows how to use CUPTI APIs for OpenACC data collection.

2.3.5. CUDA Graphs

CUPTI can collect trace of CUDA Graphs applications without breaking driver performance optimizations. CUPTI has added fields graphId and graphNodeId in the kernel, memcpy and memset activity records to denote the unique ID of the graph and the graph node respectively of the GPU activity. CUPTI issues callbacks for graph operations like graph and graph node creation/destruction/cloning and also for executable graph creation/destruction. The cuda_graphs_trace sample shows how to collect GPU trace and API trace for CUDA Graphs and how to correlate a graph node launch to the node creation API by using CUPTI callbacks for graph operations.

2.3.6. External Correlation

CUPTI supports correlation of CUDA API activity records with external APIs. Such APIs include OpenACC, OpenMP, and MPI. This associates CUPTI correlation IDs with IDs provided by the external API. Both IDs are stored in a new activity record of type CUpti_ActivityExternalCorrelation.

CUPTI maintains a stack of external correlation IDs per CPU thread and per CUpti_ExternalCorrelationKind. Clients must use cuptiActivityPushExternalCorrelationId to push an external ID of a specific kind to this stack and cuptiActivityPopExternalCorrelationId to remove the latest ID. If a CUDA API activity record is generated while any CUpti_ExternalCorrelationKind-stack on the same CPU thread is non-empty, one CUpti_ActivityExternalCorrelation record per CUpti_ExternalCorrelationKind-stack is inserted into the activity buffer before the respective CUDA API activity record. The CUPTI client is responsible for tracking passed external API correlation IDs, in order to eventually associate external API calls with CUDA API calls. Along with the activity kind

CUPTI_ACTIVITY_KIND_EXTERNAL_CORRELATION, it is necessary to enable the CUDA API activity kinds i.e. CUPTI_ACTIVITY_KIND_RUNTIME and CUPTI_ACTIVITY_KIND_DRIVER to generate external correlation activity records.

If both CUPTI_ACTIVITY_KIND_EXTERNAL_CORRELATION and any of CUPTI_ACTIVITY_KIND_OPENACC_* activity kinds are enabled, CUPTI will generate external correlation activity records for OpenACC with externalKind CUPTI_EXTERNAL_CORRELATION_KIND_OPENACC.

The cupti_external_correlation sample shows how to use CUPTI APIs for external correlation.

2.3.7. Dynamic Attach and Detach

CUPTI provides mechanisms for attaching to or detaching from a running process to support on-demand profiling. CUPTI can be attached by calling any CUPTI API as CUPTI supports lazy initialization. To detach CUPTI, call the API cuptiFinalize() which destroys and cleans up all the resources associated with CUPTI in the current process. After CUPTI detaches from the process, the process will keep on running with no CUPTI attached to it. Any subsequent CUPTI API call will reinitialize the CUPTI. You can attach and detach CUPTI any number of times. For safe operation of the API, it is recommended that API cuptiFinalize() is invoked from the exit call site of any of the CUDA Driver or Runtime API. Otherwise, CUPTI client needs to make sure that CUDA synchronization and CUPTI activity buffer flush is done before calling the API cuptiFinalize(). To understand the need for calling the API cuptiFinalize() from specific point/s in the code flow, consider multiple application threads performing various CUDA activities. While one thread is in the middle of the cuptiFinalize(), it is quite possible that other threads continue to call into the CUPTI and try to access the state of various objects (device, context, thread state etc) maintained by CUPTI, which might be rendered invalid as part of the cuptiFinalize(), thus resulting in the crash. We have to block the other threads until CUPTI teardown is completed via cuptiFinalize(). API exit call site is one such location where we can ensure that the work submitted by all the threads has been completed and we can safely teardown CUPTI. cuptiFinalize() is a heavy operation as it does context synchronization for all active CUDA contexts and blocks all the application threads until CUPTI teardown is

completed. Sample code showing the usage of the API `cuptiFinalize()` in the cupti callback handler code:

```
void CUPTIAPI
cuptiCallbackHandler(void *userdata, CUpti_CallbackDomain domain,
    CUpti_CallbackId cbid, void *cbdata)
{
    const CUpti_CallbackData *cbInfo = (CUpti_CallbackData *)cbdata;

    // Take this code path when CUPTI detach is requested
    if (detachCupti) {
        switch(domain)
        {
        case CUPTI_CB_DOMAIN_RUNTIME_API:
        case CUPTI_CB_DOMAIN_DRIVER_API:
            if (cbInfo->callbackSite == CUPTI_API_EXIT) {
                // call the CUPTI detach API
                cuptiFinalize();
            }
            break;
        default:
            break;
        }
    }
}
```

Full code can be found in the sample cupti_finalize.

2.4. CUPTI Callback API

The CUPTI Callback API allows you to register a callback into your own code. Your callback will be invoked when the application being profiled calls a CUDA runtime or driver function, or when certain events occur in the CUDA driver. The following terminology is used by the callback API.

Callback Domain
Callbacks are grouped into domains to make it easier to associate your callback functions with groups of related CUDA functions or events. There are currently four callback domains, as defined by `CUpti_CallbackDomain`: a domain for CUDA runtime functions, a domain for CUDA driver functions, a domain for CUDA resource tracking, and a domain for CUDA synchronization notification.

Callback ID
Each callback is given a unique ID within the corresponding callback domain so that you can identify it within your callback function. The CUDA driver API IDs are defined in `cupti_driver_cbid.h` and the CUDA runtime API IDs are defined in `cupti_runtime_cbid.h`. Both of these headers are included for you when you include `cupti.h`. The CUDA resource callback IDs are defined by `CUpti_CallbackIdResource`, and the CUDA synchronization callback IDs are defined by `CUpti_CallbackIdSync`.

Callback Function
Your callback function must be of type `CUpti_CallbackFunc`. This function type has two arguments that specify the callback domain and ID so that you know why

the callback is occurring. The type also has a `cbdata` argument that is used to pass data specific to the callback.

Subscriber

A subscriber is used to associate each of your callback functions with one or more CUDA API functions. There can be at most one subscriber initialized with `cuptiSubscribe()` at any time. Before initializing a new subscriber, the existing subscriber must be finalized with `cuptiUnsubscribe()`.

Each callback domain is described in detail below. Unless explicitly stated, it is not supported to call any CUDA runtime or driver API from within a callback function. Doing so may cause the application to hang.

2.4.1. Driver and Runtime API Callbacks

Using the callback API with the `CUPTI_CB_DOMAIN_DRIVER_API` or `CUPTI_CB_DOMAIN_RUNTIME_API` domains, you can associate a callback function with one or more CUDA API functions. When those CUDA functions are invoked in the application, your callback function is invoked as well. For these domains, the `cbdata` argument to your callback function will be of the type `CUpti_CallbackData`.

It is legal to call `cudaThreadSynchronize()`, `cudaDeviceSynchronize()`, `cudaStreamSynchronize()`, `cuCtxSynchronize()`, and `cuStreamSynchronize()` from within a driver or runtime API callback function.

The following code shows a typical sequence used to associate a callback function with one or more CUDA API functions. To simplify the presentation, error checking code has been removed.

```
CUpti_SubscriberHandle subscriber;
MyDataStruct *my_data = ...;
...
cuptiSubscribe(&subscriber,
               (CUpti_CallbackFunc)my_callback , my_data);
cuptiEnableDomain(1, subscriber,
                  CUPTI_CB_DOMAIN_RUNTIME_API);
```

First, `cuptiSubscribe` is used to initialize a subscriber with the `my_callback` callback function. Next, `cuptiEnableDomain` is used to associate that callback with all the CUDA runtime API functions. Using this code sequence will cause `my_callback` to be called twice each time any of the CUDA runtime API functions are invoked, once on entry to the CUDA function and once just before exit from the CUDA function. CUPTI callback API functions `cuptiEnableCallback` and `cuptiEnableAllDomains` can also be used to associate CUDA API functions with a callback (see reference below for more information).

The following code shows a typical callback function.

```
void CUPTIAPI
my_callback(void *userdata, CUpti_CallbackDomain domain,
            CUpti_CallbackId cbid, const void *cbdata)
{
  const CUpti_CallbackData *cbInfo = (CUpti_CallbackData *)cbdata;
  MyDataStruct *my_data = (MyDataStruct *)userdata;

  if ((domain == CUPTI_CB_DOMAIN_RUNTIME_API) &&
      (cbid == CUPTI_RUNTIME_TRACE_CBID_cudaMemcpy_v3020))  {
    if (cbInfo->callbackSite == CUPTI_API_ENTER) {
        cudaMemcpy_v3020_params *funcParams =
             (cudaMemcpy_v3020_params *)(cbInfo->
                 functionParams);

        size_t count = funcParams->count;
        enum cudaMemcpyKind kind = funcParams->kind;
        ...
      }
  ...
```

In your callback function, you use the `CUpti_CallbackDomain` and `CUpti_CallbackID` parameters to determine which CUDA API function invocation is causing this callback. In the example above, we are checking for the CUDA runtime `cudaMemcpy` function. The `cbdata` parameter holds a structure of useful information that can be used within the callback. In this case, we use the `callbackSite` member of the structure to detect that the callback is occurring on entry to `cudaMemcpy`, and we use the `functionParams` member to access the parameters that were passed to `cudaMemcpy`. To access the parameters, we first cast `functionParams` to a structure type corresponding to the `cudaMemcpy` function. These parameter structures are contained in `generated_cuda_runtime_api_meta.h`, `generated_cuda_meta.h`, and a number of other files. When possible, these files are included for you by `cupti.h`.

The **callback_event** and **callback_timestamp** samples described on the samples page both show how to use the callback API for the driver and runtime API domains.

2.4.2. Resource Callbacks

Using the callback API with the `CUPTI_CB_DOMAIN_RESOURCE` domain, you can associate a callback function with some CUDA resource creation and destruction events. For example, when a CUDA context is created, your callback function will be invoked with a callback ID equal to `CUPTI_CBID_RESOURCE_CONTEXT_CREATED`. For this domain, the `cbdata` argument to your callback function will be of the type `CUpti_ResourceData`.

Note that APIs `cuptiActivityFlush` and `cuptiActivityFlushAll` will result in deadlock when called from stream destroy starting callback identified using callback ID `CUPTI_CBID_RESOURCE_STREAM_DESTROY_STARTING`.

2.4.3. Synchronization Callbacks

Using the callback API with the `CUPTI_CB_DOMAIN_SYNCHRONIZE` domain, you can associate a callback function with CUDA context and stream synchronizations. For example, when a CUDA context is synchronized, your callback function will be invoked with a callback ID equal to `CUPTI_CBID_SYNCHRONIZE_CONTEXT_SYNCHRONIZED`. For this domain, the `cbdata` argument to your callback function will be of the type `CUpti_SynchronizeData`.

2.4.4. NVIDIA Tools Extension Callbacks

Using the callback API with the `CUPTI_CB_DOMAIN_NVTX` domain, you can associate a callback function with NVIDIA Tools Extension (NVTX) API functions. When an NVTX function is invoked in the application, your callback function is invoked as well. For these domains, the `cbdata` argument to your callback function will be of the type `CUpti_NvtxData`.

The NVTX library has its own convention for discovering the profiling library that will provide the implementation of the NVTX callbacks. To receive callbacks, you must set the NVTX environment variables appropriately so that when the application calls an NVTX function, your profiling library receives the callbacks. The following code sequence shows a typical initialization sequence to enable NVTX callbacks and activity records.

```
/* Set env so CUPTI-based profiling library loads on first nvtx call. */
char *inj32_path = "/path/to/32-bit/version/of/cupti/based/profiling/library";
char *inj64_path = "/path/to/64-bit/version/of/cupti/based/profiling/library";
setenv("NVTX_INJECTION32_PATH", inj32_path, 1);
setenv("NVTX_INJECTION64_PATH", inj64_path, 1);
```

The following code shows a typical sequence used to associate a callback function with one or more NVTX functions. To simplify the presentation, error checking code has been removed.

```
CUpti_SubscriberHandle subscriber;
MyDataStruct *my_data = ...;
...
cuptiSubscribe(&subscriber,
               (CUpti_CallbackFunc)my_callback , my_data);
cuptiEnableDomain(1, subscriber,
                  CUPTI_CB_DOMAIN_NVTX);
```

First, `cuptiSubscribe` is used to initialize a subscriber with the `my_callback` callback function. Next, `cuptiEnableDomain` is used to associate that callback with all the NVTX functions. Using this code sequence will cause `my_callback` to be called once each time any of the NVTX functions are invoked. CUPTI callback API functions `cuptiEnableCallback` and `cuptiEnableAllDomains` can also be used to associate NVTX API functions with a callback (see reference below for more information).

The following code shows a typical callback function.

```
void CUPTIAPI
my_callback(void *userdata, CUpti_CallbackDomain domain,
            CUpti_CallbackId cbid, const void *cbdata)
{
  const CUpti_NvtxData *nvtxInfo = (CUpti_NvtxData *)cbdata;
  MyDataStruct *my_data = (MyDataStruct *)userdata;

  if ((domain == CUPTI_CB_DOMAIN_NVTX) &&
      (cbid == CUPTI_CBID_NVTX_nvtxRangeStartEx))  {
    nvtxRangeStartEx_params *params = (nvtxRangeStartEx_params *)nvtxInfo->
             functionParams;
    nvtxRangeId_t *id = (nvtxRangeId_t *)nvtxInfo->functionReturnValue;
    ...
  }
  ...
```

In your callback function, you use the `CUpti_CallbackDomain` and `CUpti_CallbackID` parameters to determine which NVTX API function invocation is causing this callback. In the example above, we are checking for the `nvtxRangeStartEx` function. The `cbdata` parameter holds a structure of useful information that can be used within the callback. In this case, we use the `functionParams` member to access the parameters that were passed to `nvtxRangeStartEx`. To access the parameters, we first cast `functionParams` to a structure type corresponding to the `nvtxRangeStartEx` function. These parameter structures are contained in `generated_nvtx_meta.h`. We also use `functionReturnValue` member to access the value returned by `nvtxRangeStartEx`. To access the return value, we first cast `functionReturnValue` to the return type corresponding to the `nvtxRangeStartEx` function. If there is no return value for the NVTX function, `functionReturnValue` is NULL.

The sample cupti_nvtx shows the initialization sequence to enable NVTX callbacks and activity records.

If your CUPTI-based profiling library links static CUPTI library, you can define and export your own NvtxInitializeInjection and NvtxInitializeInjection2 functions, which would be called by setting the NVTX environment variables.

If you want CUPTI to handle NVTX calls, these functions should call CUPTI's corresponding initialization functions, as shown in the example below so that when the application calls a NVTX function, your profiling library receives the callbacks. The

following code sequence shows how this can be done to receive callbacks and activity records when linking static CUPTI library.

```
/* Set env so CUPTI-based profiling library loads on first nvtx call. */
char *inj32_path = "/path/to/32-bit/version/of/cupti/based/profiling/library";
char *inj64_path = "/path/to/64-bit/version/of/cupti/based/profiling/library";
setenv("NVTX_INJECTION32_PATH", inj32_path, 1);
setenv("NVTX_INJECTION64_PATH", inj64_path, 1);

/* Extern the CUPTI NVTX initialization APIs. The APIs are thread-safe */
extern "C" CUptiResult CUPTIAPI cuptiNvtxInitialize(void* pfnGetExportTable);
extern "C" CUptiResult CUPTIAPI cuptiNvtxInitialize2(void* pfnGetExportTable);

extern "C" int InitializeInjectionNvtx(void* p)
{
  CUptiResult res = cuptiNvtxInitialize(p);
  return (res == CUPTI_SUCCESS) ? 1 : 0;
}

extern "C" int InitializeInjectionNvtx2(void* p)
{
  CUptiResult res = cuptiNvtxInitialize2(p);
  return (res == CUPTI_SUCCESS) ? 1 : 0;
}
```

Alternatively, if you want to handle NVTX calls directly in your profiling library, you can attach your own callbacks to the NVTX client in these functions.

NVTX v1 and v2 both have the initialization code in a single injection library shared by all users of NVTX in the whole process, so the initialization will happen only once per process. NVTX v3 embeds the initialization code into your own binaries, so if NVTX v3 is in multiple dynamic libraries, each one of those sites will initialize the first time a NVTX call is made from that dynamic library. These first calls could be on different threads. So if you are wiring up your own NVTX handlers, you should ensure that code is thread-safe when called from multiple threads at once.

2.4.5. State Callbacks

Any fatal error encountered by an explicit CUPTI API call is returned by the API itself, whereas errors encountered by CUPTI in the background is returned to the user only during the next explicit CUPTI API call. Using the callback API with the `CUPTI_CB_DOMAIN_STATE` domain, you can associate a callback function with errors in CUPTI, and receive the reported error instantaneously. For example, when a CUPTI runs into a fatal error, your callback function will be invoked with a callback ID equal to `CUPTI_CBID_STATE_FATAL_ERROR`. For this domain, the `cbdata` argument to your callback function will be of the type `CUpti_StateData`.

As part of CUpti_StateData, you can receive the error code of the failure, along with an appropriate error message with possible causes or appropriate links to documentation. The example usage of these callbacks can be found in the CUPTI trace samples.

2.5. CUPTI Event API

The CUPTI Event API allows you to query, configure, start, stop, and read the event counters on a CUDA-enabled device. The following terminology is used by the event API.

Event
An event is a countable activity, action, or occurrence on a device.

Event ID
Each event is assigned a unique identifier. A named event will represent the same activity, action, or occurrence on all device types. But the named event may have different IDs on different device families. Use `cuptiEventGetIdFromName` to get the ID for a named event on a particular device.

Event Category
Each event is placed in one of the categories defined by `CUpti_EventCategory`. The category indicates the general type of activity, action, or occurrence measured by the event.

Event Domain
A device exposes one or more event domains. Each event domain represents a group of related events available on that device. A device may have multiple instances of a domain, indicating that the device can simultaneously record multiple instances of each event within that domain.

Event Group
An event group is a collection of events that are managed together. The number and type of events that can be added to an event group are subject to device-specific limits. At any given time, a device may be configured to count events from a limited number of event groups. All events in an event group must belong to the same event domain.

Event Group Set
An event group set is a collection of event groups that can be enabled at the same time. Event group sets are created by `cuptiEventGroupSetsCreate` and `cuptiMetricCreateEventGroupSets`.

You can determine the events available on a device using the `cuptiDeviceEnumEventDomains` and `cuptiEventDomainEnumEvents` functions. The **cupti_query** sample described on the samples page shows how to use these functions. You can also enumerate all the CUPTI events available on any device using the `cuptiEnumEventDomains` function.

Configuring and reading event counts requires the following steps. First, select your event collection mode. If you want to count events that occur during the execution of a kernel, use `cuptiSetEventCollectionMode` to set mode `CUPTI_EVENT_COLLECTION_MODE_KERNEL`. If you want to continuously sample the event counts, use mode `CUPTI_EVENT_COLLECTION_MODE_CONTINUOUS`.

Next, determine the names of the events that you want to count, and then use the `cuptiEventGroupCreate`, `cuptiEventGetIdFromName`, and `cuptiEventGroupAddEvent` functions to create and initialize an event group with those events. If you are unable to add all the events to a single event group, then you will need to create multiple event groups. Alternatively, you can use the `cuptiEventGroupSetsCreate` function to automatically create the event group(s) required for a set of events.

It's possible that all the requested events can't be collected in the single pass due to hardware or software limitations, one needs to replay the exact same set of GPU workloads multiple times. Number of passes can be queried using the API `cuptiEventGroupSetsCreate`. Profiling one event always takes single pass. Multiple passes might be required when we want to profile multiple events together. Code snippet showing how to query number of passes:

```
CUpti_EventGroupSets *eventGroupSets = NULL;
size_t eventIdArraySize = sizeof(CUpti_EventID) * numEvents;
CUpti_EventID *eventIdArray = (CUpti_EventID *)malloc(sizeof(CUpti_EventID) *
 numEvents);
// fill in event Ids
cuptiEventGroupSetsCreate(context, eventIdArraySize, eventIdArray,
 &eventGroupSets);
// number of passes required to collect all the events
passes = eventGroupSets->numSets;
```

To begin counting a set of events, enable the event group or groups that contain those events by using the `cuptiEventGroupEnable` function. If your events are contained in multiple event groups, you may be unable to enable all of the event groups at the same time i.e. in the same pass. In this case, you can gather the events across multiple executions of the application or you can enable kernel replay. If you enable kernel replay using `cuptiEnableKernelReplayMode`, you will be able to enable any number of event groups and all the contained events will be collected.

Use the `cuptiEventGroupReadEvent` and/or `cuptiEventGroupReadAllEvents` functions to read the event values. When you are done collecting events, use the `cuptiEventGroupDisable` function to stop counting the events contained in an event group. The **callback_event** sample described on the samples page shows how to use these functions to create, enable, and disable event groups, and how to read event counts.

For event collection mode `CUPTI_EVENT_COLLECTION_MODE_KERNEL`, event or metric collection may significantly change the overall performance characteristics of the application because all kernel executions that occur between the `cuptiEventGroupEnable` and `cuptiEventGroupDisable` calls are serialized on the GPU. This can be avoided by using mode

`CUPTI_EVENT_COLLECTION_MODE_CONTINUOUS`, and restricting profiling to events and metrics that can be collected in a single pass.

All the events and metrics except NVLink metrics are collected at the context level, irrespective of the event collection mode. That is, events or metrics can be attributed to the context being profiled and values can be accurately collected, when multiple contexts are executing on the GPU. NVLink metrics are collected at device level for all event collection modes.

In a system with multiple GPUs, events can be collected simultaneously on all the GPUs; in other words, event profiling doesn't enforce any serialization of work across GPUs. The event_multi_gpu sample shows how to use the CUPTI event and CUDA APIs on such setups.

Event APIs from the header `cupti_events.h` are not supported for devices with compute capability 7.5 and higher. It is advised to use the CUPTI Profiling API instead. Refer to the section Migration to the Profiling API.

2.5.1. Collecting Kernel Execution Events

A common use of the event API is to count a set of events during the execution of a kernel (as demonstrated by the **callback_event** sample). The following code shows a typical callback used for this purpose. Assume that the callback was enabled only for a kernel launch using the CUDA runtime (i.e., by `cuptiEnableCallback(1, subscriber, CUPTI_CB_DOMAIN_RUNTIME_API,`

CUPTI_RUNTIME_TRACE_CBID_cudaLaunch_v3020). To simplify the presentation, error checking code has been removed.

```
static void CUPTIAPI
getEventValueCallback(void *userdata,
                      CUpti_CallbackDomain domain,
                      CUpti_CallbackId cbid,
                      const void *cbdata)
{
  const CUpti_CallbackData *cbData =
                (CUpti_CallbackData *)cbdata;

  if (cbData->callbackSite == CUPTI_API_ENTER) {
    cudaDeviceSynchronize();
    cuptiSetEventCollectionMode(cbInfo->context,
                                CUPTI_EVENT_COLLECTION_MODE_KERNEL);
    cuptiEventGroupEnable(eventGroup);
  }

  if (cbData->callbackSite == CUPTI_API_EXIT) {
    cudaDeviceSynchronize();
    cuptiEventGroupReadEvent(eventGroup,
                             CUPTI_EVENT_READ_FLAG_NONE,
                             eventId,
                             &bytesRead, &eventVal);

    cuptiEventGroupDisable(eventGroup);
  }
}
```

Two synchronization points are used to ensure that events are counted only for the execution of the kernel. If the application contains other threads that launch kernels, then additional thread-level synchronization must also be introduced to ensure that those threads do not launch kernels while the callback is collecting events. When the cudaLaunch API is entered (that is, before the kernel is actually launched on the device), cudaDeviceSynchronize is used to wait until the GPU is idle. The event collection mode is set to CUPTI_EVENT_COLLECTION_MODE_KERNEL so that the event counters are automatically started and stopped just before and after the kernel executes. Then event collection is enabled with cuptiEventGroupEnable.

When the cudaLaunch API is exited (that is, after the kernel is queued for execution on the GPU) another cudaDeviceSynchronize is used to cause the CPU thread to wait for the kernel to finish execution. Finally, the event counts are read with cuptiEventGroupReadEvent.

2.5.2. Sampling Events

The event API can also be used to sample event values while a kernel or kernels are executing (as demonstrated by the **event_sampling** sample). The sample shows one possible way to perform the sampling. The event collection mode is set to CUPTI_EVENT_COLLECTION_MODE_CONTINUOUS so that the event counters run continuously. Two threads are used in **event_sampling**: one thread schedules the kernels and memcpys that perform the computation, while another thread wakes up periodically to sample an event counter. In this sample, there is no correlation of the event samples with what is happening on the GPU.

2.6. CUPTI Metric API

The CUPTI Metric API allows you to collect application metrics calculated from one or more event values. The following terminology is used by the metric API.

Metric
A characteristic of an application that is calculated from one or more event values.

Metric ID
Each metric is assigned a unique identifier. A named metric will represent the same characteristic on all device types. But the named metric may have different IDs on different device families. Use `cuptiMetricGetIdFromName` to get the ID for a named metric on a particular device.

Metric Category
Each metric is placed in one of the categories defined by `CUpti_MetricCategory`. The category indicates the general type of the characteristic measured by the metric.

Metric Property
Each metric is calculated from input values. These input values can be events or properties of the device or system. The available properties are defined by `CUpti_MetricPropertyID`.

Metric Value
Each metric has a value that represents one of the kinds defined by `CUpti_MetricValueKind`. For each value kind, there is a corresponding member of the `CUpti_MetricValue` union that is used to hold the metric's value.

The tables included in this section list the metrics available for each device, as determined by the device's compute capability. You can also determine the metrics available on a device using the `cuptiDeviceEnumMetrics` function. The **cupti_query** sample described on the samples page shows how to use this function. You can also enumerate all the CUPTI metrics available on any device using the `cuptiEnumMetrics` function.

CUPTI provides two functions for calculating a metric value. `cuptiMetricGetValue2` can be used to calculate a metric value when the device is not available. All required event values and metric properties must be provided by the caller. `cuptiMetricGetValue` can be used to calculate a metric value when the device is available (as a CUdevice object). All required event values must be provided by the caller, but CUPTI will determine the appropriate property values from the CUdevice object.

Configuring and calculating metric values requires the following steps. First, determine the name of the metric that you want to collect, and then use the `cuptiMetricGetIdFromName` to get the metric ID. Use `cuptiMetricEnumEvents` to get the events required to calculate the metric, and follow instructions in the CUPTI Event API section to create the event groups for those events.

When creating event groups in this manner, it is important to use the result of `cuptiMetricGetRequiredEventGroupSets` to properly group together events that must be collected in the same pass to ensure proper metric calculation.

Alternatively, you can use the `cuptiMetricCreateEventGroupSets` function to automatically create the event group(s) required for metrics' events. When using this function, events will be grouped as required to most accurately calculate the metric; as a result, it is not necessary to use `cuptiMetricGetRequiredEventGroupSets`.

If you are using `cuptiMetricGetValue2`, then you must also collect the required metric property values using `cuptiMetricEnumProperties`.

Collect event counts as described in the CUPTI Event API section, and then use either `cuptiMetricGetValue` or `cuptiMetricGetValue2` to calculate the metric value from the collected event and property values. The **callback_metric** sample described on the samples page shows how to use the functions to calculate event values and calculate a metric using `cuptiMetricGetValue`. Note that as shown in the example, you should collect event counts from all domain instances, and normalize the counts to get the most accurate metric values. It is necessary to normalize the event counts because the number of event counter instances varies by device and by the event being counted.

For example, a device might have 8 multiprocessors but only have event counters for 4 of the multiprocessors, and might have 3 memory units and only have events counters for one memory unit. When calculating a metric that requires a multiprocessor event and a memory unit event, the 4 multiprocessor counters should be summed and multiplied by 2 to normalize the event count across the entire device. Similarly, the one memory unit counter should be multiplied by 3 to normalize the event count across the entire device. The normalized values can then be passed to `cuptiMetricGetValue` or `cuptiMetricGetValue2` to calculate the metric value.

As described, the normalization assumes the kernel executes a sufficient number of blocks to completely load the device. If the kernel has only a small number of blocks, normalizing across the entire device may skew the result.

It's possible that all the requested metrics can't be collected in the single pass due to hardware or software limitations, one needs to replay the exact same set of GPU workloads multiple times. Number of passes can be queried using the API `cuptiMetricCreateEventGroupSets`. Profiling a single metric can also take

multiple passes depending on the number and type of events it is calculated from. Code snippet showing how to query number of passes:

```
CUpti_EventGroupSets *eventGroupSets = NULL;
size_t metricIdArraySize = sizeof(CUpti_MetricID) * numMetrics;
CUpti_MetricID metricIdArray = (CUpti_MetricID *)malloc(sizeof(CUpti_MetricID) *
 numMetrics);
// fill in metric Ids
cuptiMetricCreateEventGroupSets(context, metricIdArraySize, metricIdArray,
 &eventGroupSets);
// number of passes required to collect all the metrics
passes = eventGroupSets->numSets;
```

Metric APIs from the header `cupti_metrics.h` are not supported for devices with compute capability 7.5 and higher. It is advised to use the CUPTI Profiling API instead. Refer to the section Migration to the Profiling API.

2.6.1. Metrics Reference

This section contains detailed descriptions of the metrics that can be collected by the CUPTI. A scope value of "Single-context" indicates that the metric can only be accurately collected when a single context (CUDA or graphics) is executing on the GPU. A scope value of "Multi-context" indicates that the metric can be accurately collected when multiple contexts are executing on the GPU. A scope value of "Device" indicates that the metric will be collected at device level, that is, it will include values for all the contexts executing on the GPU.

2.6.1.1. Metrics for Capability 5.x

Devices with compute capability 5.x implement the metrics shown in the following table. Note that for some metrics, the "Multi-context" scope is supported only for specific devices. Such metrics are marked with "Multi-context*" under the "Scope" column. Refer to the note at the bottom of the table.

Table 4 Capability 5.x Metrics

Metric Name	Description	Scope
achieved_occupancy	Ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor	Multi-context
atomic_transactions	Global memory atomic and reduction transactions	Multi-context
atomic_transactions_per_request	Average number of global memory atomic and reduction transactions performed for each atomic and reduction instruction	Multi-context
branch_efficiency	Ratio of non-divergent branches to total branches expressed as percentage	Multi-context
cf_executed	Number of executed control-flow instructions	Multi-context

Metric Name	Description	Scope
cf_fu_utilization	The utilization level of the multiprocessor function units that execute control-flow instructions on a scale of 0 to 10	Multi-context
cf_issued	Number of issued control-flow instructions	Multi-context
double_precision_fu_utilization	The utilization level of the multiprocessor function units that execute double-precision floating-point instructions on a scale of 0 to 10	Multi-context
dram_read_bytes	Total bytes read from DRAM to L2 cache. This is available for compute capability 5.0 and 5.2.	Multi-context*
dram_read_throughput	Device memory read throughput. This is available for compute capability 5.0 and 5.2.	Multi-context*
dram_read_transactions	Device memory read transactions. This is available for compute capability 5.0 and 5.2.	Multi-context*
dram_utilization	The utilization level of the device memory relative to the peak utilization on a scale of 0 to 10	Multi-context*
dram_write_bytes	Total bytes written from L2 cache to DRAM. This is available for compute capability 5.0 and 5.2.	Multi-context*
dram_write_throughput	Device memory write throughput. This is available for compute capability 5.0 and 5.2.	Multi-context*
dram_write_transactions	Device memory write transactions. This is available for compute capability 5.0 and 5.2.	Multi-context*
ecc_throughput	ECC throughput from L2 to DRAM. This is available for compute capability 5.0 and 5.2.	Multi-context*
ecc_transactions	Number of ECC transactions between L2 and DRAM. This is available for compute capability 5.0 and 5.2.	Multi-context*
eligible_warps_per_cycle	Average number of warps that are eligible to issue per active cycle	Multi-context
flop_count_dp	Number of double-precision floating-point operations executed by non-predicated threads (add, multiply, and multiply-accumulate). Each multiply-accumulate operation contributes 2 to the count.	Multi-context
flop_count_dp_add	Number of double-precision floating-point add operations executed by non-predicated threads.	Multi-context
flop_count_dp_fma	Number of double-precision floating-point multiply-accumulate operations executed by non-predicated threads. Each multiply-accumulate operation contributes 1 to the count.	Multi-context
flop_count_dp_mul	Number of double-precision floating-point multiply operations executed by non-predicated threads.	Multi-context
flop_count_hp	Number of half-precision floating-point operations executed by non-predicated threads	Multi-context*

Metric Name	Description	Scope
	(add, multiply and multiply-accumulate). Each multiply-accumulate operation contributes 2 to the count. This is available for compute capability 5.3.	
flop_count_hp_add	Number of half-precision floating-point add operations executed by non-predicated threads. This is available for compute capability 5.3.	Multi-context*
flop_count_hp_fma	Number of half-precision floating-point multiply-accumulate operations executed by non-predicated threads. Each multiply-accumulate operation contributes 1 to the count. This is available for compute capability 5.3.	Multi-context*
flop_count_hp_mul	Number of half-precision floating-point multiply operations executed by non-predicated threads. This is available for compute capability 5.3.	Multi-context*
flop_count_sp	Number of single-precision floating-point operations executed by non-predicated threads (add, multiply, and multiply-accumulate). Each multiply-accumulate operation contributes 2 to the count. The count does not include special operations.	Multi-context
flop_count_sp_add	Number of single-precision floating-point add operations executed by non-predicated threads.	Multi-context
flop_count_sp_fma	Number of single-precision floating-point multiply-accumulate operations executed by non-predicated threads. Each multiply-accumulate operation contributes 1 to the count.	Multi-context
flop_count_sp_mul	Number of single-precision floating-point multiply operations executed by non-predicated threads.	Multi-context
flop_count_sp_special	Number of single-precision floating-point special operations executed by non-predicated threads.	Multi-context
flop_dp_efficiency	Ratio of achieved to peak double-precision floating-point operations	Multi-context
flop_hp_efficiency	Ratio of achieved to peak half-precision floating-point operations. This is available for compute capability 5.3.	Multi-context*
flop_sp_efficiency	Ratio of achieved to peak single-precision floating-point operations	Multi-context
gld_efficiency	Ratio of requested global memory load throughput to required global memory load throughput expressed as percentage.	Multi-context*
gld_requested_throughput	Requested global memory load throughput	Multi-context
gld_throughput	Global memory load throughput	Multi-context*

Metric Name	Description	Scope
gld_transactions	Number of global memory load transactions	Multi-context*
gld_transactions_per_request	Average number of global memory load transactions performed for each global memory load.	Multi-context*
global_atomic_requests	Total number of global atomic(Atom and Atom CAS) requests from Multiprocessor	Multi-context
global_hit_rate	Hit rate for global loads in unified l1/tex cache. Metric value maybe wrong if malloc is used in kernel.	Multi-context*
global_load_requests	Total number of global load requests from Multiprocessor	Multi-context
global_reduction_requests	Total number of global reduction requests from Multiprocessor	Multi-context
global_store_requests	Total number of global store requests from Multiprocessor. This does not include atomic requests.	Multi-context
gst_efficiency	Ratio of requested global memory store throughput to required global memory store throughput expressed as percentage.	Multi-context*
gst_requested_throughput	Requested global memory store throughput	Multi-context
gst_throughput	Global memory store throughput	Multi-context*
gst_transactions	Number of global memory store transactions	Multi-context*
gst_transactions_per_request	Average number of global memory store transactions performed for each global memory store	Multi-context*
half_precision_fu_utilization	The utilization level of the multiprocessor function units that execute 16 bit floating-point instructions and integer instructions on a scale of 0 to 10. This is available for compute capability 5.3.	Multi-context*
inst_bit_convert	Number of bit-conversion instructions executed by non-predicated threads	Multi-context
inst_compute_ld_st	Number of compute load/store instructions executed by non-predicated threads	Multi-context
inst_control	Number of control-flow instructions executed by non-predicated threads (jump, branch, etc.)	Multi-context
inst_executed	The number of instructions executed	Multi-context
inst_executed_global_atomics	Warp level instructions for global atom and atom cas	Multi-context
inst_executed_global_loads	Warp level instructions for global loads	Multi-context
inst_executed_global_reductions	Warp level instructions for global reductions	Multi-context

Metric Name	Description	Scope
inst_executed_global_stores	Warp level instructions for global stores	Multi-context
inst_executed_local_loads	Warp level instructions for local loads	Multi-context
inst_executed_local_stores	Warp level instructions for local stores	Multi-context
inst_executed_shared_atomics	Warp level shared instructions for atom and atom CAS	Multi-context
inst_executed_shared_loads	Warp level instructions for shared loads	Multi-context
inst_executed_shared_stores	Warp level instructions for shared stores	Multi-context
inst_executed_surface_atomics	Warp level instructions for surface atom and atom cas	Multi-context
inst_executed_surface_loads	Warp level instructions for surface loads	Multi-context
inst_executed_surface_reductions	Warp level instructions for surface reductions	Multi-context
inst_executed_surface_stores	Warp level instructions for surface stores	Multi-context
inst_executed_tex_ops	Warp level instructions for texture	Multi-context
inst_fp_16	Number of half-precision floating-point instructions executed by non-predicated threads (arithmetic, compare, etc.) This is available for compute capability 5.3.	Multi-context*
inst_fp_32	Number of single-precision floating-point instructions executed by non-predicated threads (arithmetic, compare, etc.)	Multi-context
inst_fp_64	Number of double-precision floating-point instructions executed by non-predicated threads (arithmetic, compare, etc.)	Multi-context
inst_integer	Number of integer instructions executed by non-predicated threads	Multi-context
inst_inter_thread_communication	Number of inter-thread communication instructions executed by non-predicated threads	Multi-context
inst_issued	The number of instructions issued	Multi-context
inst_misc	Number of miscellaneous instructions executed by non-predicated threads	Multi-context
inst_per_warp	Average number of instructions executed by each warp	Multi-context
inst_replay_overhead	Average number of replays for each instruction executed	Multi-context
ipc	Instructions executed per cycle	Multi-context
issue_slot_utilization	Percentage of issue slots that issued at least one instruction, averaged across all cycles	Multi-context
issue_slots	The number of issue slots used	Multi-context
issued_ipc	Instructions issued per cycle	Multi-context
l2_atomic_throughput	Memory read throughput seen at L2 cache for atomic and reduction requests	Multi-context

Metric Name	Description	Scope
l2_atomic_transactions	Memory read transactions seen at L2 cache for atomic and reduction requests	Multi-context*
l2_global_atomic_store_bytes	Bytes written to L2 from Unified cache for global atomics (ATOM and ATOM CAS)	Multi-context*
l2_global_load_bytes	Bytes read from L2 for misses in Unified Cache for global loads	Multi-context*
l2_global_reduction_bytes	Bytes written to L2 from Unified cache for global reductions	Multi-context*
l2_local_global_store_bytes	Bytes written to L2 from Unified Cache for local and global stores. This does not include global atomics.	Multi-context*
l2_local_load_bytes	Bytes read from L2 for misses in Unified Cache for local loads	Multi-context*
l2_read_throughput	Memory read throughput seen at L2 cache for all read requests	Multi-context*
l2_read_transactions	Memory read transactions seen at L2 cache for all read requests	Multi-context*
l2_surface_atomic_store_bytes	Bytes transferred between Unified Cache and L2 for surface atomics (ATOM and ATOM CAS)	Multi-context*
l2_surface_load_bytes	Bytes read from L2 for misses in Unified Cache for surface loads	Multi-context*
l2_surface_reduction_bytes	Bytes written to L2 from Unified Cache for surface reductions	Multi-context*
l2_surface_store_bytes	Bytes written to L2 from Unified Cache for surface stores. This does not include surface atomics.	Multi-context*
l2_tex_hit_rate	Hit rate at L2 cache for all requests from texture cache	Multi-context*
l2_tex_read_hit_rate	Hit rate at L2 cache for all read requests from texture cache. This is available for compute capability 5.0 and 5.2.	Multi-context*
l2_tex_read_throughput	Memory read throughput seen at L2 cache for read requests from the texture cache	Multi-context*
l2_tex_read_transactions	Memory read transactions seen at L2 cache for read requests from the texture cache	Multi-context*
l2_tex_write_hit_rate	Hit Rate at L2 cache for all write requests from texture cache. This is available for compute capability 5.0 and 5.2.	Multi-context*
l2_tex_write_throughput	Memory write throughput seen at L2 cache for write requests from the texture cache	Multi-context*
l2_tex_write_transactions	Memory write transactions seen at L2 cache for write requests from the texture cache	Multi-context*
l2_utilization	The utilization level of the L2 cache relative to the peak utilization on a scale of 0 to 10	Multi-context*

Metric Name	Description	Scope
l2_write_throughput	Memory write throughput seen at L2 cache for all write requests	Multi-context*
l2_write_transactions	Memory write transactions seen at L2 cache for all write requests	Multi-context*
ldst_executed	Number of executed local, global, shared and texture memory load and store instructions	Multi-context
ldst_fu_utilization	The utilization level of the multiprocessor function units that execute shared load, shared store and constant load instructions on a scale of 0 to 10	Multi-context
ldst_issued	Number of issued local, global, shared and texture memory load and store instructions	Multi-context
local_hit_rate	Hit rate for local loads and stores	Multi-context*
local_load_requests	Total number of local load requests from Multiprocessor	Multi-context*
local_load_throughput	Local memory load throughput	Multi-context*
local_load_transactions	Number of local memory load transactions	Multi-context*
local_load_transactions_per_request	Average number of local memory load transactions performed for each local memory load	Multi-context*
local_memory_overhead	Ratio of local memory traffic to total memory traffic between the L1 and L2 caches expressed as percentage	Multi-context*
local_store_requests	Total number of local store requests from Multiprocessor	Multi-context*
local_store_throughput	Local memory store throughput	Multi-context*
local_store_transactions	Number of local memory store transactions	Multi-context*
local_store_transactions_per_request	Average number of local memory store transactions performed for each local memory store	Multi-context*
pcie_total_data_received	Total data bytes received through PCIe	Device
pcie_total_data_transmitted	Total data bytes transmitted through PCIe	Device
shared_efficiency	Ratio of requested shared memory throughput to required shared memory throughput expressed as percentage	Multi-context*
shared_load_throughput	Shared memory load throughput	Multi-context*
shared_load_transactions	Number of shared memory load transactions	Multi-context*

Metric Name	Description	Scope
shared_load_transactions_per_request	Average number of shared memory load transactions performed for each shared memory load	Multi-context*
shared_store_throughput	Shared memory store throughput	Multi-context*
shared_store_transactions	Number of shared memory store transactions	Multi-context*
shared_store_transactions_per_request	Average number of shared memory store transactions performed for each shared memory store	Multi-context*
shared_utilization	The utilization level of the shared memory relative to peak utilization on a scale of 0 to 10	Multi-context*
single_precision_fu_utilization	The utilization level of the multiprocessor function units that execute single-precision floating-point instructions and integer instructions on a scale of 0 to 10	Multi-context
sm_efficiency	The percentage of time at least one warp is active on a specific multiprocessor	Multi-context*
special_fu_utilization	The utilization level of the multiprocessor function units that execute sin, cos, ex2, popc, flo, and similar instructions on a scale of 0 to 10	Multi-context
stall_constant_memory_dependency	Percentage of stalls occurring because of immediate constant cache miss	Multi-context
stall_exec_dependency	Percentage of stalls occurring because an input required by the instruction is not yet available	Multi-context
stall_inst_fetch	Percentage of stalls occurring because the next assembly instruction has not yet been fetched	Multi-context
stall_memory_dependency	Percentage of stalls occurring because a memory operation cannot be performed due to the required resources not being available or fully utilized, or because too many requests of a given type are outstanding	Multi-context
stall_memory_throttle	Percentage of stalls occurring because of memory throttle	Multi-context
stall_not_selected	Percentage of stalls occurring because warp was not selected	Multi-context
stall_other	Percentage of stalls occurring due to miscellaneous reasons	Multi-context
stall_pipe_busy	Percentage of stalls occurring because a compute operation cannot be performed because the compute pipeline is busy	Multi-context
stall_sync	Percentage of stalls occurring because the warp is blocked at a __syncthreads() call	Multi-context
stall_texture	Percentage of stalls occurring because the texture sub-system is fully utilized or has too many outstanding requests	Multi-context

Metric Name	Description	Scope
surface_atomic_requests	Total number of surface atomic(Atom and Atom CAS) requests from Multiprocessor	Multi-context
surface_load_requests	Total number of surface load requests from Multiprocessor	Multi-context
surface_reduction_requests	Total number of surface reduction requests from Multiprocessor	Multi-context
surface_store_requests	Total number of surface store requests from Multiprocessor	Multi-context
sysmem_read_bytes	Number of bytes read from system memory	Multi-context*
sysmem_read_throughput	System memory read throughput	Multi-context*
sysmem_read_transactions	Number of system memory read transactions	Multi-context*
sysmem_read_utilization	The read utilization level of the system memory relative to the peak utilization on a scale of 0 to 10. This is available for compute capability 5.0 and 5.2.	Multi-context
sysmem_utilization	The utilization level of the system memory relative to the peak utilization on a scale of 0 to 10. This is available for compute capability 5.0 and 5.2.	Multi-context*
sysmem_write_bytes	Number of bytes written to system memory	Multi-context*
sysmem_write_throughput	System memory write throughput	Multi-context*
sysmem_write_transactions	Number of system memory write transactions	Multi-context*
sysmem_write_utilization	The write utilization level of the system memory relative to the peak utilization on a scale of 0 to 10. This is available for compute capability 5.0 and 5.2.	Multi-context*
tex_cache_hit_rate	Unified cache hit rate	Multi-context*
tex_cache_throughput	Unified cache throughput	Multi-context*
tex_cache_transactions	Unified cache read transactions	Multi-context*
tex_fu_utilization	The utilization level of the multiprocessor function units that execute global, local and texture memory instructions on a scale of 0 to 10	Multi-context
tex_utilization	The utilization level of the unified cache relative to the peak utilization on a scale of 0 to 10	Multi-context*

Metric Name	Description	Scope
texture_load_requests	Total number of texture Load requests from Multiprocessor	Multi-context
warp_execution_efficiency	Ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor	Multi-context
warp_nonpred_execution_efficiency	Ratio of the average active threads per warp executing non-predicated instructions to the maximum number of threads per warp supported on a multiprocessor	Multi-context

*** The "Multi-context" scope for this metric is supported only for devices with compute capability 5.0 and 5.2.**

2.6.1.2. Metrics for Capability 6.x

Devices with compute capability 6.x implement the metrics shown in the following table.

Table 5 Capability 6.x Metrics

Metric Name	Description	Scope
achieved_occupancy	Ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor	Multi-context
atomic_transactions	Global memory atomic and reduction transactions	Multi-context
atomic_transactions_per_request	Average number of global memory atomic and reduction transactions performed for each atomic and reduction instruction	Multi-context
branch_efficiency	Ratio of non-divergent branches to total branches expressed as percentage	Multi-context
cf_executed	Number of executed control-flow instructions	Multi-context
cf_fu_utilization	The utilization level of the multiprocessor function units that execute control-flow instructions on a scale of 0 to 10	Multi-context
cf_issued	Number of issued control-flow instructions	Multi-context
double_precision_fu_utilization	The utilization level of the multiprocessor function units that execute double-precision floating-point instructions on a scale of 0 to 10	Multi-context
dram_read_bytes	Total bytes read from DRAM to L2 cache	Multi-context
dram_read_throughput	Device memory read throughput. This is available for compute capability 6.0 and 6.1.	Multi-context
dram_read_transactions	Device memory read transactions. This is available for compute capability 6.0 and 6.1.	Multi-context

Metric Name	Description	Scope
dram_utilization	The utilization level of the device memory relative to the peak utilization on a scale of 0 to 10	Multi-context
dram_write_bytes	Total bytes written from L2 cache to DRAM	Multi-context
dram_write_throughput	Device memory write throughput. This is available for compute capability 6.0 and 6.1.	Multi-context
dram_write_transactions	Device memory write transactions. This is available for compute capability 6.0 and 6.1.	Multi-context
ecc_throughput	ECC throughput from L2 to DRAM. This is available for compute capability 6.1.	Multi-context
ecc_transactions	Number of ECC transactions between L2 and DRAM. This is available for compute capability 6.1.	Multi-context
eligible_warps_per_cycle	Average number of warps that are eligible to issue per active cycle	Multi-context
flop_count_dp	Number of double-precision floating-point operations executed by non-predicated threads (add, multiply, and multiply-accumulate). Each multiply-accumulate operation contributes 2 to the count.	Multi-context
flop_count_dp_add	Number of double-precision floating-point add operations executed by non-predicated threads.	Multi-context
flop_count_dp_fma	Number of double-precision floating-point multiply-accumulate operations executed by non-predicated threads. Each multiply-accumulate operation contributes 1 to the count.	Multi-context
flop_count_dp_mul	Number of double-precision floating-point multiply operations executed by non-predicated threads.	Multi-context
flop_count_hp	Number of half-precision floating-point operations executed by non-predicated threads (add, multiply, and multiply-accumulate). Each multiply-accumulate operation contributes 2 to the count.	Multi-context
flop_count_hp_add	Number of half-precision floating-point add operations executed by non-predicated threads.	Multi-context
flop_count_hp_fma	Number of half-precision floating-point multiply-accumulate operations executed by non-predicated threads. Each multiply-accumulate operation contributes 1 to the count.	Multi-context
flop_count_hp_mul	Number of half-precision floating-point multiply operations executed by non-predicated threads.	Multi-context
flop_count_sp	Number of single-precision floating-point operations executed by non-predicated threads (add, multiply, and multiply-accumulate). Each multiply-accumulate operation contributes 2 to	Multi-context

Metric Name	Description	Scope
	the count. The count does not include special operations.	
flop_count_sp_add	Number of single-precision floating-point add operations executed by non-predicated threads.	Multi-context
flop_count_sp_fma	Number of single-precision floating-point multiply-accumulate operations executed by non-predicated threads. Each multiply-accumulate operation contributes 1 to the count.	Multi-context
flop_count_sp_mul	Number of single-precision floating-point multiply operations executed by non-predicated threads.	Multi-context
flop_count_sp_special	Number of single-precision floating-point special operations executed by non-predicated threads.	Multi-context
flop_dp_efficiency	Ratio of achieved to peak double-precision floating-point operations	Multi-context
flop_hp_efficiency	Ratio of achieved to peak half-precision floating-point operations	Multi-context
flop_sp_efficiency	Ratio of achieved to peak single-precision floating-point operations	Multi-context
gld_efficiency	Ratio of requested global memory load throughput to required global memory load throughput expressed as percentage.	Multi-context
gld_requested_throughput	Requested global memory load throughput	Multi-context
gld_throughput	Global memory load throughput	Multi-context
gld_transactions	Number of global memory load transactions	Multi-context
gld_transactions_per_request	Average number of global memory load transactions performed for each global memory load.	Multi-context
global_atomic_requests	Total number of global atomic(Atom and Atom CAS) requests from Multiprocessor	Multi-context
global_hit_rate	Hit rate for global loads in unified l1/tex cache. Metric value maybe wrong if malloc is used in kernel.	Multi-context
global_load_requests	Total number of global load requests from Multiprocessor	Multi-context
global_reduction_requests	Total number of global reduction requests from Multiprocessor	Multi-context
global_store_requests	Total number of global store requests from Multiprocessor. This does not include atomic requests.	Multi-context
gst_efficiency	Ratio of requested global memory store throughput to required global memory store throughput expressed as percentage.	Multi-context
gst_requested_throughput	Requested global memory store throughput	Multi-context

Metric Name	Description	Scope
gst_throughput	Global memory store throughput	Multi-context
gst_transactions	Number of global memory store transactions	Multi-context
gst_transactions_per_request	Average number of global memory store transactions performed for each global memory store	Multi-context
half_precision_fu_utilization	The utilization level of the multiprocessor function units that execute 16 bit floating-point instructions on a scale of 0 to 10	Multi-context
inst_bit_convert	Number of bit-conversion instructions executed by non-predicated threads	Multi-context
inst_compute_ld_st	Number of compute load/store instructions executed by non-predicated threads	Multi-context
inst_control	Number of control-flow instructions executed by non-predicated threads (jump, branch, etc.)	Multi-context
inst_executed	The number of instructions executed	Multi-context
inst_executed_global_atomics	Warp level instructions for global atom and atom cas	Multi-context
inst_executed_global_loads	Warp level instructions for global loads	Multi-context
inst_executed_global_reductions	Warp level instructions for global reductions	Multi-context
inst_executed_global_stores	Warp level instructions for global stores	Multi-context
inst_executed_local_loads	Warp level instructions for local loads	Multi-context
inst_executed_local_stores	Warp level instructions for local stores	Multi-context
inst_executed_shared_atomics	Warp level shared instructions for atom and atom CAS	Multi-context
inst_executed_shared_loads	Warp level instructions for shared loads	Multi-context
inst_executed_shared_stores	Warp level instructions for shared stores	Multi-context
inst_executed_surface_atomics	Warp level instructions for surface atom and atom cas	Multi-context
inst_executed_surface_loads	Warp level instructions for surface loads	Multi-context
inst_executed_surface_reductions	Warp level instructions for surface reductions	Multi-context
inst_executed_surface_stores	Warp level instructions for surface stores	Multi-context
inst_executed_tex_ops	Warp level instructions for texture	Multi-context
inst_fp_16	Number of half-precision floating-point instructions executed by non-predicated threads (arithmetic, compare, etc.)	Multi-context
inst_fp_32	Number of single-precision floating-point instructions executed by non-predicated threads (arithmetic, compare, etc.)	Multi-context
inst_fp_64	Number of double-precision floating-point instructions executed by non-predicated threads (arithmetic, compare, etc.)	Multi-context

Metric Name	Description	Scope
inst_integer	Number of integer instructions executed by non-predicated threads	Multi-context
inst_inter_thread_communication	Number of inter-thread communication instructions executed by non-predicated threads	Multi-context
inst_issued	The number of instructions issued	Multi-context
inst_misc	Number of miscellaneous instructions executed by non-predicated threads	Multi-context
inst_per_warp	Average number of instructions executed by each warp	Multi-context
inst_replay_overhead	Average number of replays for each instruction executed	Multi-context
ipc	Instructions executed per cycle	Multi-context
issue_slot_utilization	Percentage of issue slots that issued at least one instruction, averaged across all cycles	Multi-context
issue_slots	The number of issue slots used	Multi-context
issued_ipc	Instructions issued per cycle	Multi-context
l2_atomic_throughput	Memory read throughput seen at L2 cache for atomic and reduction requests	Multi-context
l2_atomic_transactions	Memory read transactions seen at L2 cache for atomic and reduction requests	Multi-context
l2_global_atomic_store_bytes	Bytes written to L2 from Unified cache for global atomics (ATOM and ATOM CAS)	Multi-context
l2_global_load_bytes	Bytes read from L2 for misses in Unified Cache for global loads	Multi-context
l2_global_reduction_bytes	Bytes written to L2 from Unified cache for global reductions	Multi-context
l2_local_global_store_bytes	Bytes written to L2 from Unified Cache for local and global stores. This does not include global atomics.	Multi-context
l2_local_load_bytes	Bytes read from L2 for misses in Unified Cache for local loads	Multi-context
l2_read_throughput	Memory read throughput seen at L2 cache for all read requests	Multi-context
l2_read_transactions	Memory read transactions seen at L2 cache for all read requests	Multi-context
l2_surface_atomic_store_bytes	Bytes transferred between Unified Cache and L2 for surface atomics (ATOM and ATOM CAS)	Multi-context
l2_surface_load_bytes	Bytes read from L2 for misses in Unified Cache for surface loads	Multi-context
l2_surface_reduction_bytes	Bytes written to L2 from Unified Cache for surface reductions	Multi-context

Metric Name	Description	Scope
l2_surface_store_bytes	Bytes written to L2 from Unified Cache for surface stores. This does not include surface atomics.	Multi-context
l2_tex_hit_rate	Hit rate at L2 cache for all requests from texture cache	Multi-context
l2_tex_read_hit_rate	Hit rate at L2 cache for all read requests from texture cache. This is available for compute capability 6.0 and 6.1.	Multi-context
l2_tex_read_throughput	Memory read throughput seen at L2 cache for read requests from the texture cache	Multi-context
l2_tex_read_transactions	Memory read transactions seen at L2 cache for read requests from the texture cache	Multi-context
l2_tex_write_hit_rate	Hit Rate at L2 cache for all write requests from texture cache. This is available for compute capability 6.0 and 6.1.	Multi-context
l2_tex_write_throughput	Memory write throughput seen at L2 cache for write requests from the texture cache	Multi-context
l2_tex_write_transactions	Memory write transactions seen at L2 cache for write requests from the texture cache	Multi-context
l2_utilization	The utilization level of the L2 cache relative to the peak utilization on a scale of 0 to 10	Multi-context
l2_write_throughput	Memory write throughput seen at L2 cache for all write requests	Multi-context
l2_write_transactions	Memory write transactions seen at L2 cache for all write requests	Multi-context
ldst_executed	Number of executed local, global, shared and texture memory load and store instructions	Multi-context
ldst_fu_utilization	The utilization level of the multiprocessor function units that execute shared load, shared store and constant load instructions on a scale of 0 to 10	Multi-context
ldst_issued	Number of issued local, global, shared and texture memory load and store instructions	Multi-context
local_hit_rate	Hit rate for local loads and stores	Multi-context
local_load_requests	Total number of local load requests from Multiprocessor	Multi-context
local_load_throughput	Local memory load throughput	Multi-context
local_load_transactions	Number of local memory load transactions	Multi-context
local_load_transactions_per_request	Average number of local memory load transactions performed for each local memory load	Multi-context
local_memory_overhead	Ratio of local memory traffic to total memory traffic between the L1 and L2 caches expressed as percentage	Multi-context

Metric Name	Description	Scope
local_store_requests	Total number of local store requests from Multiprocessor	Multi-context
local_store_throughput	Local memory store throughput	Multi-context
local_store_transactions	Number of local memory store transactions	Multi-context
local_store_transactions_per_request	Average number of local memory store transactions performed for each local memory store	Multi-context
nvlink_overhead_data_received	Ratio of overhead data to the total data, received through NVLink. This is available for compute capability 6.0.	Device
nvlink_overhead_data_transmitted	Ratio of overhead data to the total data, transmitted through NVLink. This is available for compute capability 6.0.	Device
nvlink_receive_throughput	Number of bytes received per second through NVLinks. This is available for compute capability 6.0.	Device
nvlink_total_data_received	Total data bytes received through NVLinks including headers. This is available for compute capability 6.0.	Device
nvlink_total_data_transmitted	Total data bytes transmitted through NVLinks including headers. This is available for compute capability 6.0.	Device
nvlink_total_nratom_data_transmitted	Total non-reduction atomic data bytes transmitted through NVLinks. This is available for compute capability 6.0.	Device
nvlink_total_ratom_data_transmitted	Total reduction atomic data bytes transmitted through NVLinks This is available for compute capability 6.0.	Device
nvlink_total_response_data_received	Total response data bytes received through NVLink, response data includes data for read requests and result of non-reduction atomic requests. This is available for compute capability 6.0.	Device
nvlink_total_write_data_transmitted	Total write data bytes transmitted through NVLinks. This is available for compute capability 6.0.	Device
nvlink_transmit_throughput	Number of Bytes Transmitted per second through NVLinks. This is available for compute capability 6.0.	Device
nvlink_user_data_received	User data bytes received through NVLinks, doesn't include headers. This is available for compute capability 6.0.	Device
nvlink_user_data_transmitted	User data bytes transmitted through NVLinks, doesn't include headers. This is available for compute capability 6.0.	Device

Metric Name	Description	Scope
nvlink_user_nratom_data_transmitted	Total non-reduction atomic user data bytes transmitted through NVLinks. This is available for compute capability 6.0.	Device
nvlink_user_ratom_data_transmitted	Total reduction atomic user data bytes transmitted through NVLinks. This is available for compute capability 6.0.	Device
nvlink_user_response_data_received	Total user response data bytes received through NVLink, response data includes data for read requests and result of non-reduction atomic requests. This is available for compute capability 6.0.	Device
nvlink_user_write_data_transmitted	User write data bytes transmitted through NVLinks. This is available for compute capability 6.0.	Device
pcie_total_data_received	Total data bytes received through PCIe	Device
pcie_total_data_transmitted	Total data bytes transmitted through PCIe	Device
shared_efficiency	Ratio of requested shared memory throughput to required shared memory throughput expressed as percentage	Multi-context
shared_load_throughput	Shared memory load throughput	Multi-context
shared_load_transactions	Number of shared memory load transactions	Multi-context
shared_load_transactions_per_request	Average number of shared memory load transactions performed for each shared memory load	Multi-context
shared_store_throughput	Shared memory store throughput	Multi-context
shared_store_transactions	Number of shared memory store transactions	Multi-context
shared_store_transactions_per_request	Average number of shared memory store transactions performed for each shared memory store	Multi-context
shared_utilization	The utilization level of the shared memory relative to peak utilization on a scale of 0 to 10	Multi-context
single_precision_fu_utilization	The utilization level of the multiprocessor function units that execute single-precision floating-point instructions and integer instructions on a scale of 0 to 10	Multi-context
sm_efficiency	The percentage of time at least one warp is active on a specific multiprocessor	Multi-context
special_fu_utilization	The utilization level of the multiprocessor function units that execute sin, cos, ex2, popc, flo, and similar instructions on a scale of 0 to 10	Multi-context
stall_constant_memory_dependency	Percentage of stalls occurring because of immediate constant cache miss	Multi-context
stall_exec_dependency	Percentage of stalls occurring because an input required by the instruction is not yet available	Multi-context

Metric Name	Description	Scope
stall_inst_fetch	Percentage of stalls occurring because the next assembly instruction has not yet been fetched	Multi-context
stall_memory_dependency	Percentage of stalls occurring because a memory operation cannot be performed due to the required resources not being available or fully utilized, or because too many requests of a given type are outstanding	Multi-context
stall_memory_throttle	Percentage of stalls occurring because of memory throttle	Multi-context
stall_not_selected	Percentage of stalls occurring because warp was not selected	Multi-context
stall_other	Percentage of stalls occurring due to miscellaneous reasons	Multi-context
stall_pipe_busy	Percentage of stalls occurring because a compute operation cannot be performed because the compute pipeline is busy	Multi-context
stall_sync	Percentage of stalls occurring because the warp is blocked at a __syncthreads() call	Multi-context
stall_texture	Percentage of stalls occurring because the texture sub-system is fully utilized or has too many outstanding requests	Multi-context
surface_atomic_requests	Total number of surface atomic(Atom and Atom CAS) requests from Multiprocessor	Multi-context
surface_load_requests	Total number of surface load requests from Multiprocessor	Multi-context
surface_reduction_requests	Total number of surface reduction requests from Multiprocessor	Multi-context
surface_store_requests	Total number of surface store requests from Multiprocessor	Multi-context
sysmem_read_bytes	Number of bytes read from system memory	Multi-context
sysmem_read_throughput	System memory read throughput	Multi-context
sysmem_read_transactions	Number of system memory read transactions	Multi-context
sysmem_read_utilization	The read utilization level of the system memory relative to the peak utilization on a scale of 0 to 10. This is available for compute capability 6.0 and 6.1.	Multi-context
sysmem_utilization	The utilization level of the system memory relative to the peak utilization on a scale of 0 to 10. This is available for compute capability 6.0 and 6.1.	Multi-context
sysmem_write_bytes	Number of bytes written to system memory	Multi-context
sysmem_write_throughput	System memory write throughput	Multi-context
sysmem_write_transactions	Number of system memory write transactions	Multi-context

Metric Name	Description	Scope
sysmem_write_utilization	The write utilization level of the system memory relative to the peak utilization on a scale of 0 to 10. This is available for compute capability 6.0 and 6.1.	Multi-context
tex_cache_hit_rate	Unified cache hit rate	Multi-context
tex_cache_throughput	Unified cache throughput	Multi-context
tex_cache_transactions	Unified cache read transactions	Multi-context
tex_fu_utilization	The utilization level of the multiprocessor function units that execute global, local and texture memory instructions on a scale of 0 to 10	Multi-context
tex_utilization	The utilization level of the unified cache relative to the peak utilization on a scale of 0 to 10	Multi-context
texture_load_requests	Total number of texture Load requests from Multiprocessor	Multi-context
unique_warps_launched	Number of warps launched. Value is unaffected by compute preemption.	Multi-context
warp_execution_efficiency	Ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor	Multi-context
warp_nonpred_execution_efficiency	Ratio of the average active threads per warp executing non-predicated instructions to the maximum number of threads per warp supported on a multiprocessor	Multi-context

2.6.1.3. Metrics for Capability 7.0

Devices with compute capability 7.0 implement the metrics shown in the following table.

Table 6 Capability 7.x (7.0 and 7.2) Metrics

Metric Name	Description	Scope
achieved_occupancy	Ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor	Multi-context
atomic_transactions	Global memory atomic and reduction transactions	Multi-context
atomic_transactions_per_request	Average number of global memory atomic and reduction transactions performed for each atomic and reduction instruction	Multi-context
branch_efficiency	Ratio of branch instruction to sum of branch and divergent branch instruction	Multi-context
cf_executed	Number of executed control-flow instructions	Multi-context

Metric Name	Description	Scope
cf_fu_utilization	The utilization level of the multiprocessor function units that execute control-flow instructions on a scale of 0 to 10	Multi-context
cf_issued	Number of issued control-flow instructions	Multi-context
double_precision_fu_utilization	The utilization level of the multiprocessor function units that execute double-precision floating-point instructions on a scale of 0 to 10	Multi-context
dram_read_bytes	Total bytes read from DRAM to L2 cache	Multi-context
dram_read_throughput	Device memory read throughput	Multi-context
dram_read_transactions	Device memory read transactions	Multi-context
dram_utilization	The utilization level of the device memory relative to the peak utilization on a scale of 0 to 10	Multi-context
dram_write_bytes	Total bytes written from L2 cache to DRAM	Multi-context
dram_write_throughput	Device memory write throughput	Multi-context
dram_write_transactions	Device memory write transactions	Multi-context
eligible_warps_per_cycle	Average number of warps that are eligible to issue per active cycle	Multi-context
flop_count_dp	Number of double-precision floating-point operations executed by non-predicated threads (add, multiply, and multiply-accumulate). Each multiply-accumulate operation contributes 2 to the count.	Multi-context
flop_count_dp_add	Number of double-precision floating-point add operations executed by non-predicated threads.	Multi-context
flop_count_dp_fma	Number of double-precision floating-point multiply-accumulate operations executed by non-predicated threads. Each multiply-accumulate operation contributes 1 to the count.	Multi-context
flop_count_dp_mul	Number of double-precision floating-point multiply operations executed by non-predicated threads.	Multi-context
flop_count_hp	Number of half-precision floating-point operations executed by non-predicated threads (add, multiply, and multiply-accumulate). Each multiply-accumulate contributes 2 or 4 to the count based on the number of inputs.	Multi-context
flop_count_hp_add	Number of half-precision floating-point add operations executed by non-predicated threads.	Multi-context
flop_count_hp_fma	Number of half-precision floating-point multiply-accumulate operations executed by non-predicated threads. Each multiply-accumulate contributes 2 or 4 to the count based on the number of inputs.	Multi-context

Metric Name	Description	Scope
flop_count_hp_mul	Number of half-precision floating-point multiply operations executed by non-predicated threads.	Multi-context
flop_count_sp	Number of single-precision floating-point operations executed by non-predicated threads (add, multiply, and multiply-accumulate). Each multiply-accumulate operation contributes 2 to the count. The count does not include special operations.	Multi-context
flop_count_sp_add	Number of single-precision floating-point add operations executed by non-predicated threads.	Multi-context
flop_count_sp_fma	Number of single-precision floating-point multiply-accumulate operations executed by non-predicated threads. Each multiply-accumulate operation contributes 1 to the count.	Multi-context
flop_count_sp_mul	Number of single-precision floating-point multiply operations executed by non-predicated threads.	Multi-context
flop_count_sp_special	Number of single-precision floating-point special operations executed by non-predicated threads.	Multi-context
flop_dp_efficiency	Ratio of achieved to peak double-precision floating-point operations	Multi-context
flop_hp_efficiency	Ratio of achieved to peak half-precision floating-point operations	Multi-context
flop_sp_efficiency	Ratio of achieved to peak single-precision floating-point operations	Multi-context
gld_efficiency	Ratio of requested global memory load throughput to required global memory load throughput expressed as percentage.	Multi-context
gld_requested_throughput	Requested global memory load throughput	Multi-context
gld_throughput	Global memory load throughput	Multi-context
gld_transactions	Number of global memory load transactions	Multi-context
gld_transactions_per_request	Average number of global memory load transactions performed for each global memory load.	Multi-context
global_atomic_requests	Total number of global atomic(Atom and Atom CAS) requests from Multiprocessor	Multi-context
global_hit_rate	Hit rate for global load and store in unified l1/ tex cache	Multi-context
global_load_requests	Total number of global load requests from Multiprocessor	Multi-context
global_reduction_requests	Total number of global reduction requests from Multiprocessor	Multi-context

Metric Name	Description	Scope
global_store_requests	Total number of global store requests from Multiprocessor. This does not include atomic requests.	Multi-context
gst_efficiency	Ratio of requested global memory store throughput to required global memory store throughput expressed as percentage.	Multi-context
gst_requested_throughput	Requested global memory store throughput	Multi-context
gst_throughput	Global memory store throughput	Multi-context
gst_transactions	Number of global memory store transactions	Multi-context
gst_transactions_per_request	Average number of global memory store transactions performed for each global memory store	Multi-context
half_precision_fu_utilization	The utilization level of the multiprocessor function units that execute 16 bit floating-point instructions on a scale of 0 to 10. Note that this doesn't specify the utilization level of tensor core unit	Multi-context
inst_bit_convert	Number of bit-conversion instructions executed by non-predicated threads	Multi-context
inst_compute_ld_st	Number of compute load/store instructions executed by non-predicated threads	Multi-context
inst_control	Number of control-flow instructions executed by non-predicated threads (jump, branch, etc.)	Multi-context
inst_executed	The number of instructions executed	Multi-context
inst_executed_global_atomics	Warp level instructions for global atom and atom cas	Multi-context
inst_executed_global_loads	Warp level instructions for global loads	Multi-context
inst_executed_global_reductions	Warp level instructions for global reductions	Multi-context
inst_executed_global_stores	Warp level instructions for global stores	Multi-context
inst_executed_local_loads	Warp level instructions for local loads	Multi-context
inst_executed_local_stores	Warp level instructions for local stores	Multi-context
inst_executed_shared_atomics	Warp level shared instructions for atom and atom CAS	Multi-context
inst_executed_shared_loads	Warp level instructions for shared loads	Multi-context
inst_executed_shared_stores	Warp level instructions for shared stores	Multi-context
inst_executed_surface_atomics	Warp level instructions for surface atom and atom cas	Multi-context
inst_executed_surface_loads	Warp level instructions for surface loads	Multi-context
inst_executed_surface_reductions	Warp level instructions for surface reductions	Multi-context
inst_executed_surface_stores	Warp level instructions for surface stores	Multi-context
inst_executed_tex_ops	Warp level instructions for texture	Multi-context

Metric Name	Description	Scope
inst_fp_16	Number of half-precision floating-point instructions executed by non-predicated threads (arithmetic, compare, etc.)	Multi-context
inst_fp_32	Number of single-precision floating-point instructions executed by non-predicated threads (arithmetic, compare, etc.)	Multi-context
inst_fp_64	Number of double-precision floating-point instructions executed by non-predicated threads (arithmetic, compare, etc.)	Multi-context
inst_integer	Number of integer instructions executed by non-predicated threads	Multi-context
inst_inter_thread_communication	Number of inter-thread communication instructions executed by non-predicated threads	Multi-context
inst_issued	The number of instructions issued	Multi-context
inst_misc	Number of miscellaneous instructions executed by non-predicated threads	Multi-context
inst_per_warp	Average number of instructions executed by each warp	Multi-context
inst_replay_overhead	Average number of replays for each instruction executed	Multi-context
ipc	Instructions executed per cycle	Multi-context
issue_slot_utilization	Percentage of issue slots that issued at least one instruction, averaged across all cycles	Multi-context
issue_slots	The number of issue slots used	Multi-context
issued_ipc	Instructions issued per cycle	Multi-context
l2_atomic_throughput	Memory read throughput seen at L2 cache for atomic and reduction requests	Multi-context
l2_atomic_transactions	Memory read transactions seen at L2 cache for atomic and reduction requests	Multi-context
l2_global_atomic_store_bytes	Bytes written to L2 from L1 for global atomics (ATOM and ATOM CAS)	Multi-context
l2_global_load_bytes	Bytes read from L2 for misses in L1 for global loads	Multi-context
l2_local_global_store_bytes	Bytes written to L2 from L1 for local and global stores. This does not include global atomics.	Multi-context
l2_local_load_bytes	Bytes read from L2 for misses in L1 for local loads	Multi-context
l2_read_throughput	Memory read throughput seen at L2 cache for all read requests	Multi-context
l2_read_transactions	Memory read transactions seen at L2 cache for all read requests	Multi-context
l2_surface_load_bytes	Bytes read from L2 for misses in L1 for surface loads	Multi-context

Metric Name	Description	Scope
l2_surface_store_bytes	Bytes read from L2 for misses in L1 for surface stores	Multi-context
l2_tex_hit_rate	Hit rate at L2 cache for all requests from texture cache	Multi-context
l2_tex_read_hit_rate	Hit rate at L2 cache for all read requests from texture cache	Multi-context
l2_tex_read_throughput	Memory read throughput seen at L2 cache for read requests from the texture cache	Multi-context
l2_tex_read_transactions	Memory read transactions seen at L2 cache for read requests from the texture cache	Multi-context
l2_tex_write_hit_rate	Hit Rate at L2 cache for all write requests from texture cache	Multi-context
l2_tex_write_throughput	Memory write throughput seen at L2 cache for write requests from the texture cache	Multi-context
l2_tex_write_transactions	Memory write transactions seen at L2 cache for write requests from the texture cache	Multi-context
l2_utilization	The utilization level of the L2 cache relative to the peak utilization on a scale of 0 to 10	Multi-context
l2_write_throughput	Memory write throughput seen at L2 cache for all write requests	Multi-context
l2_write_transactions	Memory write transactions seen at L2 cache for all write requests	Multi-context
ldst_executed	Number of executed local, global, shared and texture memory load and store instructions	Multi-context
ldst_fu_utilization	The utilization level of the multiprocessor function units that execute shared load, shared store and constant load instructions on a scale of 0 to 10	Multi-context
ldst_issued	Number of issued local, global, shared and texture memory load and store instructions	Multi-context
local_hit_rate	Hit rate for local loads and stores	Multi-context
local_load_requests	Total number of local load requests from Multiprocessor	Multi-context
local_load_throughput	Local memory load throughput	Multi-context
local_load_transactions	Number of local memory load transactions	Multi-context
local_load_transactions_per_request	Average number of local memory load transactions performed for each local memory load	Multi-context
local_memory_overhead	Ratio of local memory traffic to total memory traffic between the L1 and L2 caches expressed as percentage	Multi-context
local_store_requests	Total number of local store requests from Multiprocessor	Multi-context

Metric Name	Description	Scope
local_store_throughput	Local memory store throughput	Multi-context
local_store_transactions	Number of local memory store transactions	Multi-context
local_store_transactions_per_request	Average number of local memory store transactions performed for each local memory store	Multi-context
nvlink_overhead_data_received	Ratio of overhead data to the total data, received through NVLink.	Device
nvlink_overhead_data_transmitted	Ratio of overhead data to the total data, transmitted through NVLink.	Device
nvlink_receive_throughput	Number of bytes received per second through NVLinks.	Device
nvlink_total_data_received	Total data bytes received through NVLinks including headers.	Device
nvlink_total_data_transmitted	Total data bytes transmitted through NVLinks including headers.	Device
nvlink_total_nratom_data_transmitted	Total non-reduction atomic data bytes transmitted through NVLinks.	Device
nvlink_total_ratom_data_transmitted	Total reduction atomic data bytes transmitted through NVLinks.	Device
nvlink_total_response_data_received	Total response data bytes received through NVLink, response data includes data for read requests and result of non-reduction atomic requests.	Device
nvlink_total_write_data_transmitted	Total write data bytes transmitted through NVLinks.	Device
nvlink_transmit_throughput	Number of Bytes Transmitted per second through NVLinks.	Device
nvlink_user_data_received	User data bytes received through NVLinks, doesn't include headers.	Device
nvlink_user_data_transmitted	User data bytes transmitted through NVLinks, doesn't include headers.	Device
nvlink_user_nratom_data_transmitted	Total non-reduction atomic user data bytes transmitted through NVLinks.	Device
nvlink_user_ratom_data_transmitted	Total reduction atomic user data bytes transmitted through NVLinks.	Device
nvlink_user_response_data_received	Total user response data bytes received through NVLink, response data includes data for read requests and result of non-reduction atomic requests.	Device
nvlink_user_write_data_transmitted	User write data bytes transmitted through NVLinks.	Device
pcie_total_data_received	Total data bytes received through PCIe	Device
pcie_total_data_transmitted	Total data bytes transmitted through PCIe	Device

Metric Name	Description	Scope
shared_efficiency	Ratio of requested shared memory throughput to required shared memory throughput expressed as percentage	Multi-context
shared_load_throughput	Shared memory load throughput	Multi-context
shared_load_transactions	Number of shared memory load transactions	Multi-context
shared_load_transactions_per_request	Average number of shared memory load transactions performed for each shared memory load	Multi-context
shared_store_throughput	Shared memory store throughput	Multi-context
shared_store_transactions	Number of shared memory store transactions	Multi-context
shared_store_transactions_per_request	Average number of shared memory store transactions performed for each shared memory store	Multi-context
shared_utilization	The utilization level of the shared memory relative to peak utilization on a scale of 0 to 10	Multi-context
single_precision_fu_utilization	The utilization level of the multiprocessor function units that execute single-precision floating-point instructions on a scale of 0 to 10	Multi-context
sm_efficiency	The percentage of time at least one warp is active on a specific multiprocessor	Multi-context
special_fu_utilization	The utilization level of the multiprocessor function units that execute sin, cos, ex2, popc, flo, and similar instructions on a scale of 0 to 10	Multi-context
stall_constant_memory_dependency	Percentage of stalls occurring because of immediate constant cache miss	Multi-context
stall_exec_dependency	Percentage of stalls occurring because an input required by the instruction is not yet available	Multi-context
stall_inst_fetch	Percentage of stalls occurring because the next assembly instruction has not yet been fetched	Multi-context
stall_memory_dependency	Percentage of stalls occurring because a memory operation cannot be performed due to the required resources not being available or fully utilized, or because too many requests of a given type are outstanding	Multi-context
stall_memory_throttle	Percentage of stalls occurring because of memory throttle	Multi-context
stall_not_selected	Percentage of stalls occurring because warp was not selected	Multi-context
stall_other	Percentage of stalls occurring due to miscellaneous reasons	Multi-context
stall_pipe_busy	Percentage of stalls occurring because a compute operation cannot be performed because the compute pipeline is busy	Multi-context
stall_sleeping	Percentage of stalls occurring because warp was sleeping	Multi-context

Metric Name	Description	Scope
stall_sync	Percentage of stalls occurring because the warp is blocked at a __syncthreads() call	Multi-context
stall_texture	Percentage of stalls occurring because the texture sub-system is fully utilized or has too many outstanding requests	Multi-context
surface_atomic_requests	Total number of surface atomic(Atom and Atom CAS) requests from Multiprocessor	Multi-context
surface_load_requests	Total number of surface load requests from Multiprocessor	Multi-context
surface_reduction_requests	Total number of surface reduction requests from Multiprocessor	Multi-context
surface_store_requests	Total number of surface store requests from Multiprocessor	Multi-context
sysmem_read_bytes	Number of bytes read from system memory	Multi-context
sysmem_read_throughput	System memory read throughput	Multi-context
sysmem_read_transactions	Number of system memory read transactions	Multi-context
sysmem_read_utilization	The read utilization level of the system memory relative to the peak utilization on a scale of 0 to 10	Multi-context
sysmem_utilization	The utilization level of the system memory relative to the peak utilization on a scale of 0 to 10	Multi-context
sysmem_write_bytes	Number of bytes written to system memory	Multi-context
sysmem_write_throughput	System memory write throughput	Multi-context
sysmem_write_transactions	Number of system memory write transactions	Multi-context
sysmem_write_utilization	The write utilization level of the system memory relative to the peak utilization on a scale of 0 to 10	Multi-context
tensor_precision_fu_utilization	The utilization level of the multiprocessor function units that execute tensor core instructions on a scale of 0 to 10	Multi-context
tensor_int_fu_utilization	The utilization level of the multiprocessor function units that execute tensor core int8 instructions on a scale of 0 to 10. This metric is only available for device with compute capability 7.2.	Multi-context
tex_cache_hit_rate	Unified cache hit rate	Multi-context
tex_cache_throughput	Unified cache to Multiprocessor read throughput	Multi-context
tex_cache_transactions	Unified cache to Multiprocessor read transactions	Multi-context
tex_fu_utilization	The utilization level of the multiprocessor function units that execute global, local and texture memory instructions on a scale of 0 to 10	Multi-context

Metric Name	Description	Scope
tex_utilization	The utilization level of the unified cache relative to the peak utilization on a scale of 0 to 10	Multi-context
texture_load_requests	Total number of texture Load requests from Multiprocessor	Multi-context
warp_execution_efficiency	Ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor	Multi-context
warp_nonpred_execution_efficiency	Ratio of the average active threads per warp executing non-predicated instructions to the maximum number of threads per warp supported on a multiprocessor	Multi-context

2.7. CUPTI Profiling API

Starting with CUDA 10.0, a new set of metric APIs are added for devices with compute capability 7.0 and higher. These APIs provide low and deterministic profiling overhead on the target system. These are supported on all CUDA supported platforms except Android, and are not supported under MPS (Multi-Process Service), Confidential Compute, or SLI configured systems. In order to determine whether a device is compatible with this API, a new function **`cuptiProfilerDeviceSupported`** is introduced in CUDA 11.5 which exposes overall Profiling API support and specific requirements for a given device. Profiling API must be initialized by calling **`cuptiProfilerInitialize`** before testing device support.

This section covers performance profiling Host and Target APIs for CUDA. Broadly profiling APIs are divided into following four sections:

- Enumeration (Host)
- Configuration (Host)
- Collection (Target)
- Evaluation (Host)

Host APIs provide a metric interface for enumeration, configuration and evaluation that doesn't require a compute(GPU) device, and can also run in an offline mode. In the samples section under extensions, profiler host utility covers the usage of host APIs. Target APIs are used for data collection of the metrics and requires a compute (GPU) device. Refer to samples auto_rangeProfiling and userrange_profiling for usage of profiling APIs.

The list of metrics has been overhauled from earlier generation metrics and event APIs, to support a standard naming convention based upon **`unit__(subunit?)_(pipestage?)_quantity_qualifiers`**

2.7.1. Multi Pass Collection

NVIDIA GPU hardware has a limited number of counter registers and cannot collect all possible counters concurrently. There are also limitations on which counters can be collected together in a single pass. This is resolved by replaying the exact same set of GPU workloads multiple times, where each replay is termed a pass. On each pass, a different subset of requested counters are collected. Once all passes are collected, the data is available for evaluation. Certain metrics have many counters as inputs; adding a single metric may require many passes to collect. CUPTI APIs support multi pass collection through different collection attributes.

Sample cupti_metric_properties shows how to query number of passes required to collect a set of counters.

2.7.2. Range Profiling

Each profiling session runs a series of replay passes, where each pass contains a sequence of ranges. Every metric enabled in the session's configuration is collected separately per unique range-stack in the pass. CUPTI supports auto and user defined ranges.

2.7.2.1. Auto Range

In a session with auto range mode, ranges are defined around each kernel automatically with a unique name assigned to each range, while profiling is enabled. This mode is useful for tight metric collection around each kernel. A user can choose one of the supported replay modes, pseudo code for each is described below:

Kernel Replay

The replay logic (multiple pass, if needed) is done by CUPTI implicitly (opaque to the user), and usage of CUPTI replay API's **`cuptiProfilerBeginPass`** and **`cuptiProfilerEndPass`** will be a no-op in this mode. This mode is useful for collecting metrics around a kernel in tight control. Each kernel launch is synchronized to segregate its metrics into a separate range, and a CPU-GPU sync is made to ensure the profiled data is collected from GPU. Counter Collection can be enabled and disabled with

cuptiProfilerEnableProfiling and **cuptiProfilerDisableProfiling**. Refer to the sample autorange_profiling

```
/* Assume Inputs(counterDataImagePrefix and configImage) from configuration
 phase at host */
void Collection(std::vector<uint8_t>& counterDataImagePrefix,
 std::vector<uint8_t>& configImage)
{
 CUpti_Profiler_Initialize_Params profilerInitializeParams =
 { CUpti_Profiler_Initialize_Params_STRUCT_SIZE };
 cuptiProfilerInitialize(&profilerInitializeParams);

 std::vector<uint8_t> counterDataImages;
 std::vector<uint8_t> counterDataScratchBuffer;
 CreateCounterDataImage(counterDataImages, counterDataScratchBuffer,
 counterDataImagePrefix);

 CUpti_Profiler_BeginSession_Params beginSessionParams =
 { CUpti_Profiler_BeginSession_Params_STRUCT_SIZE };
 CUpti_ProfilerRange profilerRange = CUPTI_AutoRange;
 CUpti_ProfilerReplayMode profilerReplayMode = CUPTI_KernelReplay;

 beginSessionParams.ctx = NULL;
 beginSessionParams.counterDataImageSize = counterDataImage.size();
 beginSessionParams.pCounterDataImage = &counterDataImage[0];
 beginSessionParams.counterDataScratchBufferSize =
 counterDataScratchBuffer.size();
 beginSessionParams.pCounterDataScratchBuffer = &counterDataScratchBuffer[0];
 beginSessionParams.range = profilerRange;
 beginSessionParams.replayMode = profilerReplayMode;
 beginSessionParams.maxRangesPerPass = num_ranges;
 beginSessionParams.maxLaunchesPerPass = num_ranges;

 cuptiProfilerBeginSession(&beginSessionParams));

 CUpti_Profiler_SetConfig_Params setConfigParams =
 { CUpti_Profiler_SetConfig_Params_STRUCT_SIZE };
 setConfigParams.pConfig = &configImage[0];
 setConfigParams.configSize = configImage.size();

 cuptiProfilerSetConfig(&setConfigParams));

 kernelA <<<grid, tids >>>(...);                  // KernelA not profiled

 CUpti_Profiler_EnableProfiling_Params enableProfilingParams =
 { CUpti_Profiler_EnableProfiling_Params_STRUCT_SIZE };
 cuptiProfilerEnableProfiling(&enableProfilingParams);
 {

  kernelB <<<grid, tids >>>(...);              // KernelB profiled and captured
 in an unique range.
  kernelC <<<grid, tids >>>(...);              // KernelC profiled and captured
 in an unique range.
  kernelD <<<grid, tids >>>(...);              // KernelD profiled and captured
 in an unique range.
 }

 CUpti_Profiler_DisableProfiling_Params disableProfilingParams =
 { CUpti_Profiler_DisableProfiling_Params_STRUCT_SIZE };
 cuptiProfilerDisableProfiling(&disableProfilingParams);

 kernelE <<<grid, tids >>>(...);                  // KernelE not profiled

 CUpti_Profiler_UnsetConfig_Params unsetConfigParams =
 { CUpti_Profiler_UnsetConfig_Params_STRUCT_SIZE };
 cuptiProfilerUnsetConfig(&unsetConfigParams);

 CUpti_Profiler_EndSession_Params endSessionParams =
 { CUpti_Profiler_EndSession_Params_STRUCT_SIZE };
 cuptiProfilerEndSession(&endSessionParams);
}
```

User Replay

The replay (multiple passes, if needed) is done by the user using the replay API's **`cuptiProfilerBeginPass`** and **`cuptiProfilerEndPass`**. It is user responsibility to flush the counter data **`cuptiProfilerFlushCounterData`** before ending the session to ensure collection of metric data in CPU. Counter collection can be enabled and disabled

with **cuptiProfilerEnableProfiling**/ **cuptiProfilerDisableProfiling**. Refer to the sample autorange_profiling

```
   /* Assume Inputs(counterDataImagePrefix and configImage) from configuration
phase at host */

   void Collection(std::vector<uint8_t>& counterDataImagePrefix,
std::vector<uint8_t>& configImage)
   {
       CUpti_Profiler_Initialize_Params profilerInitializeParams =
{CUpti_Profiler_Initialize_Params_STRUCT_SIZE};
       cuptiProfilerInitialize(&profilerInitializeParams);

       std::vector<uint8_t> counterDataImages;
       std::vector<uint8_t> counterDataScratchBuffer;
       CreateCounterDataImage(counterDataImages, counterDataScratchBuffer,
counterDataImagePrefix);

       CUpti_Profiler_BeginSession_Params beginSessionParams =
{CUpti_Profiler_BeginSession_Params_STRUCT_SIZE};
       CUpti_ProfilerRange profilerRange = CUPTI_AutoRange;
       CUpti_ProfilerReplayMode profilerReplayMode = CUPTI_UserReplay;

       beginSessionParams.ctx = NULL;
       beginSessionParams.counterDataImageSize = counterDataImage.size();
       beginSessionParams.pCounterDataImage = &counterDataImage[0];
       beginSessionParams.counterDataScratchBufferSize =
counterDataScratchBuffer.size();
       beginSessionParams.pCounterDataScratchBuffer =
&counterDataScratchBuffer[0];
       beginSessionParams.range = profilerRange;
       beginSessionParams.replayMode = profilerReplayMode;
       beginSessionParams.maxRangesPerPass = num_ranges;
       beginSessionParams.maxLaunchesPerPass = num_ranges;

       cuptiProfilerBeginSession(&beginSessionParams));

       CUpti_Profiler_SetConfig_Params setConfigParams =
{CUpti_Profiler_SetConfig_Params_STRUCT_SIZE};
       setConfigParams.pConfig = &configImage[0];
       setConfigParams.configSize = configImage.size();

       cuptiProfilerSetConfig(&setConfigParams));

       CUpti_Profiler_FlushCounterData_Params cuptiFlushCounterDataParams =
   {CUpti_Profiler_FlushCounterData_Params_STRUCT_SIZE};

       CUpti_Profiler_EnableProfiling_Params enableProfilingParams =
{CUpti_Profiler_EnableProfiling_Params_STRUCT_SIZE};

       CUpti_Profiler_DisableProfiling_Params disableProfilingParams =
{CUpti_Profiler_DisableProfiling_Params_STRUCT_SIZE};


       kernelA<<<grid, tids>>>(...);                  // KernelA neither
profiled, nor replayed

       CUpti_Profiler_BeginPass_Params beginPassParams =
{CUpti_Profiler_BeginPass_Params_STRUCT_SIZE};
       CUpti_Profiler_EndPass_Params endPassParams =
{CUpti_Profiler_EndPass_Params_STRUCT_SIZE};

       cuptiProfilerBeginPass(&beginPassParams);
       {
           kernelB<<<grid, tids>>>(...);              // KernelB replayed but
not profiled

           cuptiProfilerEnableProfiling(&enableProfilingParams);

           kernelC<<<grid, tids>>>(...);              // KernelC profiled and
captured in an unique range.
           kernelD<<<grid, tids>>>(...);              // KernelD profiled and
captured in an unique range.
```

Application Replay

This replay mode is same as user replay, instead of in process replay, you can replay the whole process again. You will need to update the pass index while setting the config **cuptiProfilerSetConfig** and reload the intermediate counterDataImage on each pass.

2.7.2.2. User Range

In a session with user range mode, ranges are defined by you, **cuptiProfilerPushRange** and **cuptiProfilerPopRange**. Kernel launches are concurrent within a range. This mode is useful for metric data collection around a specific section of code, instead of per-kernel metric collection. Kernel replay is not supported in user range mode. You own the responsibility of replay using **cuptiProfilerBeginPass** and **cuptiProfilerEndPass**.

User Replay

The replay (multiple passes, if needed) is done by the user using the replay API's **cuptiProfilerBeginPass** and **cuptiProfilerEndPass**. It is your responsibility to flush the counter data using **cuptiProfilerFlushCounterData** before ending the session. Counter collection can be enabled/disabled with

cuptiProfilerEnableProfiling and **cuptiProfilerDisableProfiling**. Refer to the sample userrange_profiling

```
>
    /* Assume Inputs(counterDataImagePrefix and configImage) from configuration
phase at host */

    void Collection(std::vector<uint8_t>& counterDataImagePrefix,
std::vector<uint8_t>& configImage)
    {
        CUpti_Profiler_Initialize_Params profilerInitializeParams =
{CUpti_Profiler_Initialize_Params_STRUCT_SIZE};
        cuptiProfilerInitialize(&profilerInitializeParams);

        std::vector<uint8_t> counterDataImages;
        std::vector<uint8_t> counterDataScratchBuffer;
        CreateCounterDataImage(counterDataImages, counterDataScratchBuffer,
counterDataImagePrefix);

        CUpti_Profiler_BeginSession_Params beginSessionParams =
{CUpti_Profiler_BeginSession_Params_STRUCT_SIZE};
        CUpti_ProfilerRange profilerRange = CUPTI_UserRange;
        CUpti_ProfilerReplayMode profilerReplayMode = CUPTI_UserReplay;

        beginSessionParams.ctx = NULL;
        beginSessionParams.counterDataImageSize = counterDataImage.size();
        beginSessionParams.pCounterDataImage = &counterDataImage[0];
        beginSessionParams.counterDataScratchBufferSize =
counterDataScratchBuffer.size();
        beginSessionParams.pCounterDataScratchBuffer =
&counterDataScratchBuffer[0];
        beginSessionParams.range = profilerRange;
        beginSessionParams.replayMode = profilerReplayMode;
        beginSessionParams.maxRangesPerPass = num_ranges;
        beginSessionParams.maxLaunchesPerPass = num_ranges;

        cuptiProfilerBeginSession(&beginSessionParams));

        CUpti_Profiler_SetConfig_Params setConfigParams =
{CUpti_Profiler_SetConfig_Params_STRUCT_SIZE};
        setConfigParams.pConfig = &configImage[0];
        setConfigParams.configSize = configImage.size();

        cuptiProfilerSetConfig(&setConfigParams));

        CUpti_Profiler_FlushCounterData_Params cuptiFlushCounterDataParams =
{CUpti_Profiler_FlushCounterData_Params_STRUCT_SIZE};

        kernelA<<<grid, tids>>>(...);                  // KernelA neither
profiled, nor replayed

        CUpti_Profiler_BeginPass_Params beginPassParams =
{CUpti_Profiler_BeginPass_Params_STRUCT_SIZE};
        CUpti_Profiler_EndPass_Params endPassParams =
{CUpti_Profiler_EndPass_Params_STRUCT_SIZE};

        cuptiProfilerBeginPass(&beginPassParams);
        {
            kernelB<<<grid, tids>>>(...);              // KernelB replayed but
not profiled

            CUpti_Profiler_PushRange_Params enableProfilingParams =
{CUpti_Profiler_PushRange_Params_STRUCT_SIZE};
            pushRangeParams.pRangeName = "RangeA";
            cuptiProfilerPushRange(&pushRangeParams);

            kernelC<<<grid, tids>>>(...);
            kernelD<<<grid, tids>>>(...);

            cuptiProfilerPopRange(&popRangeParams);     // Kernel C and Kernel D
are captured in rangeA without any serialization introduced by profiler
        }
        cuptiProfilerEndPass(&endPassParams);
        cuptiProfilerFlushCounterData(&cuptiFlushCounterDataParams);
```

Application Replay

This replay mode is same as user replay, instead of in process replay, you can replay the whole process again. You will need to update the pass index while setting the config using the **`cuptiProfilerSetConfig`** API, and reload the intermediate counterDataImage on each pass.

2.7.3. CUPTI Profiler Definitions

Definitions of glossary used in this section.

Counter:
The number of occurrences of a specific event on the device.

Configuration Image:
A Blob to configure the session for counters to be collected.

CounterData Image:
A Blob which contains the values of collected counters

CounterData Prefix:
A metadata header for CounterData Image

Device:
A physical NVIDIA GPU.

Event:
An event is a countable activity, action, or occurrence on device.

Metric:
A high-level value derived from counter values.

Pass:
A repeatable set of operations, with consistently labeled ranges.

Range:
A labeled region of execution

Replay:
Performing the repeatable set of operation.

Session:
A profiling session where GPU resources needed for profiling are allocated. The profiler is in armed state at session boundaries, and power management may be disabled at session boundaries. Outside of a session, the GPU will return to its normal operating state.

2.7.4. Differences from event and metric APIs

Here is the list of features which are supported by the event and metric APIs but these are not available with the Profiling API:

- Continuous mode or sampling of the metrics.
- Profiling API provides closest equivalent metrics for most of the events and metrics supported by the event and metric APIs. However, there are some events and metrics, for example NVLink performance metrics, for which there is no

equivalent metrics in the Profiling API. Tables Metrics Mapping Table and Events Mapping Table can be referred to find the equivalent Perfworks metrics for compute capability 7.0.

- Per-instance metrics i.e. users can't collect metrics for each instance of the hardware units like SM, FB etc separately. However Profiling API provides sub-metrics which can be used to get the avg/sum/min/max across all instances of a hardware unit.

2.8. Perfworks Metric API

Introduction:

The Perfworks Metric API supports the enumeration, configuration and evaluation of metrics. The binary outputs of the configuration phase are inputs to the CUPTI Range Profiling API. The output of Range Profiling is the **CounterData**, which is passed to the Derived Metrics Evaluation APIs.

GPU Metrics are generally presented as counts, ratios and percentages. The underlying values collected from hardware are raw counters (analogous to CUPTI events), but those details are hidden behind derived metric formulas.

The Metric APIs are split into two layers: Derived Metrics and Raw Metrics. Derived Metrics contains the list of named metrics and performs evaluation to numeric results, serving a similar purpose as the previous CUPTI Metric API. Most user interaction will be with derived metrics. Raw Metrics contains the list of raw counters and generates configuration file images analogous to the previous CUPTI Event API.

Metric Enumeration

Metric Enumeration is the process of listing available counters and metrics.

Refer to file `List.cpp` used by the cupti_metric_properties sample.

Metrics are grouped into three types i.e. counters, ratios and throughput. Except ratios metric type each metrics have four type of sub-metrics also known as rollup metrics i.e. sum, avg, min, max.

For enumerating supported metrics for a chip, we need to calculate the scratch buffer needed for host operation and to initialize the Metric Evaluator.

- Call **`NVPW_CUDA_MetricsEvaluator_CalculateScratchBufferSize`** for calculating scratch buffer size required for allocating memory for host operations.
- Call **`NVPW_CUDA_MetricsEvaluator_Initialize`** for initializing the Metrics Evaluator which creates a NVPW_MetricsEvaluator object.

The outline for enumerating supported counter metrics for a chip:

- Call **NVPW_MetricsEvaluator_GetMetricNames** for **NVPW_METRIC_TYPE_COUNTER** metric type for listing all the counter metrics supported.
- Call **NVPW_MetricsEvaluator_GetSupportedSubmetrics** to list all the sub-metric supported for **NVPW_METRIC_TYPE_COUNTER** metric type.
- Call **NVPW_MetricsEvaluator_GetCounterProperties** to give description of the counter and the collection hardware unit.

Similarly, for enumerating ratio and throughput metrics we need to pass **NVPW_METRIC_TYPE_RATIO** and **NVPW_METRIC_TYPE_THROUGHPUT** as metric types to **NVPW_MetricsEvaluator_GetMetricNames** and **NVPW_MetricsEvaluator_GetSupportedSubmetrics**.

For more details about the metric properties call **NVPW_MetricsEvaluator_GetRatioMetricProperties** and **NVPW_MetricsEvaluator_GetThroughputMetricProperties** respectively.

Configuration Workflow

Configuration is the process of specifying the metrics that will be collected and how those metrics should be collected. The inputs for this phase are the metric names and metric collection properties. The output for this phase is a **ConfigImage** and a **CounterDataPrefix** Image.

Refer to file `Metric.cpp` used by the userrange_profiling sample.

The outline for configuring metrics:

- As input, take a list of metric names.
- Before creating ConfigImage or CounterDataPrefixImage, we need a list of NVPA_RawMetricRequest for the metrics listed for collection.
  - We need to calculate the scratch buffer size required for the host operation and to initialize the Metric Evaluator like in the Enumeration phase.
  - For each metric, Call **NVPW_MetricsEvaluator_ConvertMetricNameToMetricEvalRequest** for creating a **NVPW_MetricEvalRequest**.
  - Call **NVPW_MetricsEvaluator_GetMetricRawDependencies** which takes the **NVPW_MetricsEvaluator** and **NVPW_MetricEvalRequest** as input, for getting raw dependencies for given metrics.
- Create an **NVPA_RawMetricRequest** with `keepInstances=true` and `isolated=true`
- Pass the **NVPA_RawMetricRequest** to **NVPW_RawMetricsConfig_AddMetrics** for the **ConfigImage**.
- Pass the **NVPA_RawMetricRequest** to **NVPW_CounterDataBuilder_AddMetrics** for the **CounterDataPrefix**.

- Generate binary configuration "images" (file format in memory):
  - **ConfigImage** from **NVPW_RawMetricsConfig_GetConfigImage**
  - **CounterDataPrefix** from **NVPW_CounterDataBuilder_GetCounterDataPrefix**

Metric Evaluation

Metric Evaluation is the process of forming metrics from the counters stored in the **CounterData** image.

Refer to file `Eval.cpp` used by the userrange_profiling sample.

The outline for configuring metrics:

- As input, take the same list of metric names as used during configuration.
- As input, take a **CounterDataImage** collected on a target device.
- We need to calculate the scratch buffer size required for the host operation and to initialize the Metric Evaluator like in the Enumeration phase.
- Query the number of ranges collected via **NVPW_CounterData_GetNumRanges**.
- For each metric:
  - Call **NVPW_MetricsEvaluator_ConvertMetricNameToMetricEvalRequest** for creating **NVPW_MetricEvalRequest**
  - For each range:
    - Call **NVPW_Profiler_CounterData_GetRangeDescriptions** to retrieve the range's description, originally set by **cuptiProfilerPushRange**.
    - Call **NVPW_MetricsEvaluator_SetDeviceAttributes** to set the current range for evaluation on the **NVPW_MetricEvalRequest**.
    - Call **NVPW_MetricsEvaluator_EvaluateToGpuValues** to query an array of numeric values corresponding to each input metric.

2.8.1. Derived metrics

Metrics Overview

The PerfWorks API comes with an advanced metrics calculation system, designed to help you determine what happened (counters and metrics), and how close the program reached to peak GPU performance (throughputs as a percentage). Every counter has associated peak rates in the database, to allow computing its throughput as a percentage.

Throughput metrics return the maximum percentage value of their constituent counters. Constituents can be programmatically queried via **NVPW_MetricsEvaluator_GetMetricNames** with **NVPW_METRIC_TYPE_THROUGHPUT** as metric types. These constituents have

been carefully selected to represent the sections of the GPU pipeline that govern peak performance. While all counters can be converted to a %-of-peak, not all counters are suitable for peak-performance analysis; examples of unsuitable counters include qualified subsets of activity, and workload residency counters. Using throughput metrics ensures meaningful and actionable analysis.

Two types of peak rates are available for every counter: burst and sustained. Burst rate is the maximum rate reportable in a single clock cycle. Sustained rate is the maximum rate achievable over an infinitely long measurement period, for "typical" operations. For many counters, burst == sustained. Since the burst rate cannot be exceeded, percentages of burst rate will always be less than 100%. Percentages of sustained rate can occasionally exceed 100% in edge cases. Burst metrics are only supported with MetricsContext APIs and these will be deprecated in a future CUDA release. These metrics are not supported with NVPW_MetricsEvaluator APIs.

Metrics Entities

The Metrics layer has 3 major types of entities:

- Metrics : these are calculated quantities, with the following static properties:
  - Description string.
  - Dimensional Units : a list of ('name', exponent) in the style of dimensional analysis. Example string representation: `pixels / gpc_clk`.
  - Raw Metric dependencies : the list of raw metrics that must be collected, in order to evaluate the metric.
  - Every metric has the following sub-metrics built in.

`.peak_sustained`	the peak sustained rate
`.peak_sustained_active`	the peak sustained rate during unit active cycles
`.peak_sustained_active.per_second`	the peak sustained rate during unit active cycles, per second *
`.peak_sustained_elapsed`	the peak sustained rate during unit elapsed cycles
`.peak_sustained_elapsed.per_second`	the peak sustained rate during unit elapsed cycles, per second *
`.peak_sustained_region`	the peak sustained rate over a user-specified "range"

.peak_sustained_region.per_second	the peak sustained rate over a user-specified "range", per second *
.peak_sustained_frame	the peak sustained rate over a user-specified "frame"
.peak_sustained_frame.per_second	the peak sustained rate over a user-specified "frame", per second *
.per_cycle_active	the number of operations per unit active cycle
.per_cycle_elapsed	the number of operations per unit elapsed cycle
.per_cycle_in_region	the number of operations per user-specified "range" cycle
.per_cycle_in_frame	the number of operations per user-specified "frame" cycle
.per_second	the number of operations per second
.pct_of_peak_sustained_active	% of peak sustained rate achieved during unit active cycles
.pct_of_peak_sustained_elapsed	% of peak sustained rate achieved during unit elapsed cycles
.pct_of_peak_sustained_region	% of peak sustained rate achieved over a user-specified "range" time
.pct_of_peak_sustained_frame	% of peak sustained rate achieved over a user-specified "frame" time

* sub-metrics added in CUPTI 11.3.

- **Counters** may be either a raw counter from the GPU, or a calculated counter value. Every counter has four sub-metrics under it, which are also called *roll-ups*:

.sum	The sum of counter values across all unit instances.

.avg	The average counter value across all unit instances.
.min	The minimum counter value across all unit instances.
.max	The maximum counter value across all unit instances.

- **Ratios** have three sub-metrics under it:

.pct	The value expressed as a percentage.
.ratio	The value expressed as a ratio.
.max_rate	The ratio's maximum value.

- **Throughputs** indicate how close a portion of the GPU reached to peak rate. Every throughput has the following sub-metrics:

.pct_of_peak_sustained_active	% of peak sustained rate achieved during unit active cycles
.pct_of_peak_sustained_elapsed	% of peak sustained rate achieved during unit elapsed cycles
.pct_of_peak_sustained_region	% of peak sustained rate achieved over a user-specified "range" time
.pct_of_peak_sustained_frame	% of peak sustained rate achieved over a user-specified "frame" time

At the configuration step, you must specify metric names. Counters, ratios, and throughputs are not directly schedulable.

Note: Burst metrics are only supported with MetricsContext APIs.

From CUPTI 11.3 onwards, due to not being useful for performance optimization following **counter** sub-metrics are not present in MetricEvaluator APIs and are only supported with MetricsContext APIs:

.peak_burst	the peak burst rate
.pct_of_peak_burst_active	% of peak burst rate achieved during unit active cycles
.pct_of_peak_burst_elapsed	% of peak burst rate achieved during unit elapsed cycles
.pct_of_peak_burst_region	% of peak burst rate achieved over a user-specified "range"

.pct_of_peak_burst_frame	% of peak burst rate achieved over a user-specified "frame"

From CUPTI 11.3 onwards, due to not being useful for performance optimization following **throughput** sub-metrics are not present in MetricEvaluator APIs and are only supported with MetricsContext APIs:

.pct_of_peak_burst_active	% of peak burst rate achieved during unit active cycles
.pct_of_peak_burst_elapsed	% of peak burst rate achieved during unit elapsed cycles
.pct_of_peak_burst_region	% of peak burst rate achieved over a user-specified "range" time
.pct_of_peak_burst_frame	% of peak burst rate achieved over a user-specified "frame" time

Metrics Examples

```
## non-metric names -- *not* directly evaluable
sm__inst_executed                   # counter
smsp__average_warp_latency          # ratio
sm__throughput                      # throughput

## a counter's four roll-ups as sub-metrics -- all evaluable
sm__inst_executed.sum               # metric
sm__inst_executed.avg               # metric
sm__inst_executed.min               # metric
sm__inst_executed.max               # metric

## all names below are metrics -- all evaluable
l1tex__data_bank_conflicts_pipe_lsu.sum
l1tex__data_bank_conflicts_pipe_lsu.sum.peak_burst
l1tex__data_bank_conflicts_pipe_lsu.sum.peak_sustained
l1tex__data_bank_conflicts_pipe_lsu.sum.per_cycle_active
l1tex__data_bank_conflicts_pipe_lsu.sum.per_cycle_elapsed
l1tex__data_bank_conflicts_pipe_lsu.sum.per_cycle_in_region
l1tex__data_bank_conflicts_pipe_lsu.sum.per_cycle_in_frame
l1tex__data_bank_conflicts_pipe_lsu.sum.per_second
l1tex__data_bank_conflicts_pipe_lsu.sum.pct_of_peak_sustained_active
l1tex__data_bank_conflicts_pipe_lsu.sum.pct_of_peak_sustained_elapsed
l1tex__data_bank_conflicts_pipe_lsu.sum.pct_of_peak_sustained_region
l1tex__data_bank_conflicts_pipe_lsu.sum.pct_of_peak_sustained_frame
```

Metrics Naming Conventions

Counters and metrics _generally_ obey the naming scheme:

- Unit-Level Counter :
  `unit__(subunit?)_(pipestage?)_quantity_(qualifiers?)`

- Interface Counter : `unit__(subunit?)_(pipestage?)_(interface)_quantity_(qualifiers?)`
- Unit Metric : `(counter_name).(rollup_metric)`
- Sub-Metric : `(counter_name).(rollup_metric).(submetric)`

where

- unit: A logical of physical unit of the GPU
- subunit: The subunit within the unit where the counter was measured. Sometimes this is a pipeline mode instead.
- pipestage: The pipeline stage within the subunit where the counter was measured.
- quantity: What is being measured. Generally matches the "dimensional units".
- qualifiers: Any additional predicates or filters applied to the counter. Often, an unqualified counter can be broken down into several qualified sub-components.
- interface: Of the form `sender2receiver`, where `sender` is the source-unit and `receiver` is the destination-unit.
- rollup_metric: One of sum,avg,min,max.
- submetric: refer to section Metric Entities

Components are not always present. Most top-level counters have no qualifiers. Subunit and pipestage may be absent where irrelevant, or there may be many subunit specifiers for detailed counters.

Cycle Metrics

Counters using the term `cycles` in the name report the number of cycles in the unit's clock domain. Unit-level cycle metrics include:

- `unit__cycles_elapsed` : The number of cycles within a range. The cycles' DimUnits are specific to the unit's clock domain.
- `unit__cycles_active` : The number of cycles where the unit was processing data.
- `unit__cycles_stalled` : The number of cycles where the unit was unable to process new data because its output interface was blocked.
- `unit__cycles_idle` : The number of cycles where the unit was idle.

Interface-level cycle counters are often (not always) available in the following variations:

- `unit__(interface)_active` : Cycles where data was transferred from source-unit to destination-unit.
- `unit__(interface)_stalled` : Cycles where the source-unit had data, but the destination-unit was unable to accept data.

2.8.2. Raw Metrics

The raw metrics layer contains a list of low-level GPU counters, and the "scheduling" logic needed to program the hardware. The binary output files (**ConfigImage** and **CounterDataPrefix**) can be generated offline, stored on disk, and used on any compatible GPU. They do not need to be generated on a machine where a GPU is available.

Refer to Metrics Configuration to see where Raw Metrics fit into the overall data flow of the profiler.

2.8.3. Metrics Mapping Table

The table below lists the CUPTI metrics for devices with compute capability 7.0. For each CUPTI metric the closest equivalent Perfworks metric or formula is given. If no equivalent Perfworks metric is available the column is left blank. Note that there can be some difference in the metric values between the CUPTI metric and the Perfworks metrics.

Table 7 Metrics Mapping Table from CUPTI to Perfworks for Compute Capability 7.0

CUPTI Metric	Perfworks Metric or Formula
achieved_occupancy	sm__warps_active.avg.pct_of_peak_sustained_active
atomic_transactions	l1tex__t_set_accesses_pipe_lsu_mem_global_op_atom.sum + l1tex__t_set_accesses_pipe_lsu_mem_global_op_red.sum
atomic_transactions_per_request	(l1tex__t_sectors_pipe_lsu_mem_global_op_atom.sum + l1tex__t_sectors_pipe_lsu_mem_global_op_red.sum) / (l1tex__t_requests_pipe_lsu_mem_global_op_atom.sum + l1tex__t_requests_pipe_lsu_mem_global_op_red.sum)
branch_efficiency	smsp__sass_average_branch_targets_threads_uniform.pct
cf_executed	smsp__inst_executed_pipe_cbu.sum + smsp__inst_executed_pipe_adu.sum
cf_fu_utilization	
cf_issued	
double_precision_fu_utilization	smsp__inst_executed_pipe_fp64.avg.pct_of_peak_sustained_active
dram_read_bytes	dram__bytes_read.sum
dram_read_throughput	dram__bytes_read.sum.per_second
dram_read_transactions	dram__sectors_read.sum
dram_utilization	dram__throughput.avg.pct_of_peak_sustained_elapsed
dram_write_bytes	dram__bytes_write.sum
dram_write_throughput	dram__bytes_write.sum.per_second
dram_write_transactions	dram__sectors_write.sum

CUPTI Metric	Perfworks Metric or Formula
eligible_warps_per_cycle	smsp__warps_eligible.sum.per_cycle_active
flop_count_dp	smsp__sass_thread_inst_executed_op_dadd_pred_on.sum + smsp__sass_thread_inst_executed_op_dmul_pred_on.sum + smsp__sass_thread_inst_executed_op_dfma_pred_on.sum * 2
flop_count_dp_add	smsp__sass_thread_inst_executed_op_dadd_pred_on.sum
flop_count_dp_fma	smsp__sass_thread_inst_executed_op_dfma_pred_on.sum
flop_count_dp_mul	smsp__sass_thread_inst_executed_op_dmul_pred_on.sum
flop_count_hp	smsp__sass_thread_inst_executed_op_hadd_pred_on.sum + smsp__sass_thread_inst_executed_op_hmul_pred_on.sum + smsp__sass_thread_inst_executed_op_hfma_pred_on.sum * 2
flop_count_hp_add	smsp__sass_thread_inst_executed_op_hadd_pred_on.sum
flop_count_hp_fma	smsp__sass_thread_inst_executed_op_hfma_pred_on.sum
flop_count_hp_mul	smsp__sass_thread_inst_executed_op_hmul_pred_on.sum
flop_count_sp	smsp__sass_thread_inst_executed_op_fadd_pred_on.sum + smsp__sass_thread_inst_executed_op_fmul_pred_on.sum + smsp__sass_thread_inst_executed_op_ffma_pred_on.sum * 2
flop_count_sp_add	smsp__sass_thread_inst_executed_op_fadd_pred_on.sum
flop_count_sp_fma	smsp__sass_thread_inst_executed_op_ffma_pred_on.sum
flop_count_sp_mul	smsp__sass_thread_inst_executed_op_fmul_pred_on.sum
flop_count_sp_special	
flop_dp_efficiency	smsp__sass_thread_inst_executed_ops_dadd_dmul_dfma_pred_on.avg.pct_of_peak_sus
flop_hp_efficiency	smsp__sass_thread_inst_executed_ops_hadd_hmul_hfma_pred_on.avg.pct_of_peak_sus
flop_sp_efficiency	smsp__sass_thread_inst_executed_ops_fadd_fmul_ffma_pred_on.avg.pct_of_peak_sust
gld_efficiency	smsp__sass_average_data_bytes_per_sector_mem_global_op_ld.pct
gld_requested_throughput	
gld_throughput	l1tex__t_bytes_pipe_lsu_mem_global_op_ld.sum.per_second
gld_transactions	l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum
gld_transactions_per_request	l1tex__average_t_sectors_per_request_pipe_lsu_mem_global_op_ld.ratio
global_atomic_requests	l1tex__t_requests_pipe_lsu_mem_global_op_atom.sum
global_hit_rate	l1tex__t_sectors_pipe_lsu_mem_global_op_{op}_lookup_hit.sum / l1tex__t_sectors_pipe_lsu_mem_global_op_{op}.sum
global_load_requests	l1tex__t_requests_pipe_lsu_mem_global_op_ld.sum
global_reduction_requests	l1tex__t_requests_pipe_lsu_mem_global_op_red.sum
global_store_requests	l1tex__t_requests_pipe_lsu_mem_global_op_st.sum
gst_efficiency	smsp__sass_average_data_bytes_per_sector_mem_global_op_st.pct
gst_requested_throughput	
gst_throughput	l1tex__t_bytes_pipe_lsu_mem_global_op_st.sum.per_second
gst_transactions	l1tex__t_bytes_pipe_lsu_mem_global_op_st.sum

CUPTI Metric	Perfworks Metric or Formula
gst_transactions_per_request	l1tex__average_t_sectors_per_request_pipe_lsu_mem_global_op_st.ratio
half_precision_fu_utilization	smsp__inst_executed_pipe_fp16.avg.pct_of_peak_sustained_active
inst_bit_convert	smsp__sass_thread_inst_executed_op_conversion_pred_on.sum
inst_compute_ld_st	smsp__sass_thread_inst_executed_op_memory_pred_on.sum
inst_control	smsp__sass_thread_inst_executed_op_control_pred_on.sum
inst_executed	smsp__inst_executed.sum
inst_executed_global_atomics	smsp__sass_inst_executed_op_global_atom.sum
inst_executed_global_loads	smsp__inst_executed_op_global_ld.sum
inst_executed_global_reductions	smsp__inst_executed_op_global_red.sum
inst_executed_global_stores	smsp__inst_executed_op_global_st.sum
inst_executed_local_loads	smsp__inst_executed_op_local_ld.sum
inst_executed_local_stores	smsp__inst_executed_op_local_st.sum
inst_executed_shared_atomics	smsp__inst_executed_op_shared_atom.sum + smsp__inst_executed_op_shared_atom_dot_alu.sum + smsp__inst_executed_op_shared_atom_dot_cas.sum
inst_executed_shared_loads	smsp__inst_executed_op_shared_ld.sum
inst_executed_shared_stores	smsp__inst_executed_op_shared_st.sum
inst_executed_surface_atomics	smsp__inst_executed_op_surface_atom.sum
inst_executed_surface_loads	smsp__inst_executed_op_surface_ld.sum + smsp__inst_executed_op_shared_atom_dot_alu.sum + smsp__inst_executed_op_shared_atom_dot_cas.sum
inst_executed_surface_reductions	smsp__inst_executed_op_surface_red.sum
inst_executed_surface_stores	smsp__inst_executed_op_surface_st.sum
inst_executed_tex_ops	smsp__inst_executed_op_texture.sum
inst_fp_16	smsp__sass_thread_inst_executed_op_fp16_pred_on.sum
inst_fp_32	smsp__sass_thread_inst_executed_op_fp32_pred_on.sum
inst_fp_64	smsp__sass_thread_inst_executed_op_fp64_pred_on.sum
inst_integer	smsp__sass_thread_inst_executed_op_integer_pred_on.sum
inst_inter_thread_communication	smsp__sass_thread_inst_executed_op_inter_thread_communication_pred_on.sum
inst_issued	smsp__inst_issued.sum
inst_misc	smsp__sass_thread_inst_executed_op_misc_pred_on.sum
inst_per_warp	smsp__average_inst_executed_per_warp.ratio
inst_replay_overhead	
ipc	smsp__inst_executed.avg.per_cycle_active
issue_slot_utilization	smsp__issue_active.avg.pct_of_peak_sustained_active
issue_slots	smsp__inst_issued.sum

CUPTI Metric	Perfworks Metric or Formula
issued_ipc	smsp__inst_issued.avg.per_cycle_active
l2_atomic_throughput	lts__t_sectors_srcunit_l1_op_atom.sum.per_second
l2_atomic_transactions	lts__t_sectors_srcunit_l1_op_atom.sum
l2_global_atomic_store_bytes	lts__t_bytes_equiv_l1sectormiss_pipe_lsu_mem_global_op_atom.sum
l2_global_load_bytes	lts__t_bytes_equiv_l1sectormiss_pipe_lsu_mem_global_op_ld.sum
l2_local_global_store_bytes	lts__t_bytes_equiv_l1sectormiss_pipe_lsu_mem_local_op_st.sum + lts__t_bytes_equiv_l1sectormiss_pipe_lsu_mem_global_op_st.sum
l2_local_load_bytes	lts__t_bytes_equiv_l1sectormiss_pipe_lsu_mem_local_op_ld.sum
l2_read_throughput	lts__t_sectors_op_read.sum.per_second
l2_read_transactions	lts__t_sectors_op_read.sum
l2_surface_load_bytes	lts__t_bytes_equiv_l1sectormiss_pipe_tex_mem_surface_op_ld.sum
l2_surface_store_bytes	lts__t_bytes_equiv_l1sectormiss_pipe_tex_mem_surface_op_st.sum
l2_tex_hit_rate	lts__t_sector_hit_rate.pct
l2_tex_read_hit_rate	lts__t_sector_op_read_hit_rate.pct
l2_tex_read_throughput	lts__t_sectors_srcunit_tex_op_read.sum.per_second
l2_tex_read_transactions	lts__t_sectors_srcunit_tex_op_read.sum
l2_tex_write_hit_rate	lts__t_sector_op_write_hit_rate.pct
l2_tex_write_throughput	lts__t_sectors_srcunit_tex_op_read.sum.per_second
l2_tex_write_transactions	lts__t_sectors_srcunit_tex_op_read.sum
l2_utilization	lts__t_sectors.avg.pct_of_peak_sustained_elapsed
l2_write_throughput	lts__t_sectors_op_write.sum.per_second
l2_write_transactions	lts__t_sectors_op_write.sum
ldst_executed	
ldst_fu_utilization	smsp__inst_executed_pipe_lsu.avg.pct_of_peak_sustained_active
ldst_issued	
local_hit_rate	
local_load_requests	l1tex__t_requests_pipe_lsu_mem_local_op_ld.sum
local_load_throughput	l1tex__t_bytes_pipe_lsu_mem_local_op_ld.sum.per_second
local_load_transactions	l1tex__t_sectors_pipe_lsu_mem_local_op_ld.sum
local_load_transactions_per_request	l1tex__average_t_sectors_per_request_pipe_lsu_mem_local_op_ld.ratio
local_memory_overhead	
local_store_requests	l1tex__t_requests_pipe_lsu_mem_local_op_st.sum
local_store_throughput	l1tex__t_sectors_pipe_lsu_mem_local_op_st.sum.per_second
local_store_transactions	l1tex__t_sectors_pipe_lsu_mem_local_op_st.sum
local_store_transactions_per_request	l1tex__average_t_sectors_per_request_pipe_lsu_mem_local_op_st.ratio

CUPTI Metric	Perfworks Metric or Formula
nvlink_data_receive_efficiency	
nvlink_data_transmission_efficiency	
nvlink_overhead_data_received	
nvlink_overhead_data_transmitted	
nvlink_receive_throughput	
nvlink_total_data_received	
nvlink_total_data_transmitted	
nvlink_total_nratom_data_transmitted	
nvlink_total_ratom_data_transmitted	
nvlink_total_response_data_received	
nvlink_total_write_data_transmitted	
nvlink_transmit_throughput	
nvlink_user_data_received	
nvlink_user_data_transmitted	
nvlink_user_nratom_data_transmitted	
nvlink_user_ratom_data_transmitted	
nvlink_user_response_data_received	
nvlink_user_write_data_transmitted	
pcie_total_data_received	pcie__read_bytes.sum
pcie_total_data_transmitted	pcie__write_bytes.sum
shared_efficiency	smsp__sass_average_data_bytes_per_wavefront_mem_shared.pct
shared_load_throughput	l1tex__data_pipe_lsu_wavefronts_mem_shared_op_ld.sum.per_second
shared_load_transactions	l1tex__data_pipe_lsu_wavefronts_mem_shared_op_ld.sum
shared_load_transactions_per_request	
shared_store_throughput	l1tex__data_pipe_lsu_wavefronts_mem_shared_op_st.sum.per_second
shared_store_transactions	l1tex__data_pipe_lsu_wavefronts_mem_shared_op_st.sum
shared_store_transactions_per_request	
shared_utilization	l1tex__data_pipe_lsu_wavefronts_mem_shared.avg.pct_of_peak_sustained_elapsed
single_precision_fu_utilization	smsp__pipe_fma_cycles_active.avg.pct_of_peak_sustained_active
sm_efficiency	smsp__cycles_active.avg.pct_of_peak_sustained_elapsed
special_fu_utilization	smsp__inst_executed_pipe_xu.avg.pct_of_peak_sustained_active
stall_constant_memory_dependency	smsp__warp_issue_stalled_imc_miss_per_warp_active.pct
stall_exec_dependency	smsp__warp_issue_stalled_short_scoreboard_per_warp_active.pct + smsp__warp_issue_stalled_wait_per_warp_active.pct
stall_inst_fetch	smsp__warp_issue_stalled_no_instruction_per_warp_active.pct

CUPTI Metric	Perfworks Metric or Formula
stall_memory_dependency	smsp__warp_issue_stalled_long_scoreboard_per_warp_active.pct
stall_memory_throttle	smsp__warp_issue_stalled_drain_per_warp_active.pct + smsp__warp_issue_stalled_lg_throttle_per_warp_active.pct
stall_not_selected	smsp__warp_issue_stalled_not_selected_per_warp_active.pct
stall_other	smsp__warp_issue_stalled_misc_per_warp_active.pct + smsp__warp_issue_stalled_dispatch_stall_per_warp_active.pct
stall_pipe_busy	smsp__warp_issue_stalled_mio_throttle_per_warp_active.pct + smsp__warp_issue_stalled_math_pipe_throttle_per_warp_active.pct
stall_sleeping	smsp__warp_issue_stalled_sleeping_per_warp_active.pct
stall_sync	smsp__warp_issue_stalled_membar_per_warp_active.pct + smsp__warp_issue_stalled_barrier_per_warp_active.pct
stall_texture	smsp__warp_issue_stalled_tex_throttle_per_warp_active.pct
surface_atomic_requests	l1tex__t_requests_pipe_tex_mem_surface_op_atom.sum
surface_load_requests	l1tex__t_requests_pipe_tex_mem_surface_op_ld.sum
surface_reduction_requests	l1tex__t_requests_pipe_tex_mem_surface_op_red.sum
surface_store_requests	l1tex__t_requests_pipe_tex_mem_surface_op_st.sum
sysmem_read_bytes	lts__t_sectors_aperture_sysmem_op_read* 32
sysmem_read_throughput	lts__t_sectors_aperture_sysmem_op_read.sum.per_second
sysmem_read_transactions	lts__t_sectors_aperture_sysmem_op_read.sum
sysmem_read_utilization	
sysmem_utilization	
sysmem_write_bytes	lts__t_sectors_aperture_sysmem_op_write * 32
sysmem_write_throughput	lts__t_sectors_aperture_sysmem_op_write.sum.per_second
sysmem_write_transactions	lts__t_sectors_aperture_sysmem_op_write.sum
sysmem_write_utilization	
tensor_precision_fu_utilization	sm__pipe_tensor_cycles_active.avg.pct_of_peak_sustained_active
tex_cache_hit_rate	l1tex__t_sector_hit_rate.pct
tex_cache_throughput	
tex_cache_transactions	l1tex__lsu_writeback_active.avg.pct_of_peak_sustained_active + l1tex__tex_writeback_active.avg.pct_of_peak_sustained_active
tex_fu_utilization	smsp__inst_executed_pipe_tex.avg.pct_of_peak_sustained_active
tex_utilization	
texture_load_requests	l1tex__t_requests_pipe_tex_mem_texture.sum
warp_execution_efficiency	smsp__thread_inst_executed_per_inst_executed.ratio
warp_nonpred_execution_efficiency	smsp__thread_inst_executed_per_inst_executed.pct

2.8.4. Events Mapping Table

The table below lists the CUPTI events for devices with compute capability 7.0. For each CUPTI event the closest equivalent Perfworks metric or formula is given. If no equivalent Perfworks metric is available the column is left blank. Note that there can be some difference in the values between the CUPTI event and the Perfworks metrics.

Table 8 Events Mapping Table from CUPTI events to Perfworks metrics for Compute Capability 7.0

CUPTI Event	Perfworks Metric or Formula
active_cycles	sm__cycles_active.sum
active_cycles_pm	sm__cycles_active.sum
active_cycles_sys	sys__cycles_active.sum
active_warps	sm__warps_active.sum
active_warps_pm	sm__warps_active.sum
atom_count	smsp__inst_executed_op_generic_atom_dot_alu.sum
elapsed_cycles_pm	sm__cycles_elapsed.sum
elapsed_cycles_sm	sm__cycles_elapsed.sum
elapsed_cycles_sys	sys__cycles_elapsed.sum
fb_subp0_read_sectors	dram__sectors_read.sum
fb_subp1_read_sectors	dram__sectors_read.sum
fb_subp0_write_sectors	dram__sectors_write.sum
fb_subp1_write_sectors	dram__sectors_write.sum
global_atom_cas	smsp__inst_executed_op_generic_atom_dot_cas.sum
gred_count	smsp__inst_executed_op_global_red.sum
inst_executed	sm__inst_executed.sum
inst_executed_fma_pipe_s0	smsp__inst_executed_pipe_fma.sum
inst_executed_fma_pipe_s1	smsp__inst_executed_pipe_fma.sum
inst_executed_fma_pipe_s2	smsp__inst_executed_pipe_fma.sum
inst_executed_fma_pipe_s3	smsp__inst_executed_pipe_fma.sum
inst_executed_fp16_pipe_s0	smsp__inst_executed_pipe_fp16.sum
inst_executed_fp16_pipe_s1	smsp__inst_executed_pipe_fp16.sum
inst_executed_fp16_pipe_s2	smsp__inst_executed_pipe_fp16.sum
inst_executed_fp16_pipe_s3	smsp__inst_executed_pipe_fp16.sum
inst_executed_fp64_pipe_s0	smsp__inst_executed_pipe_fp64.sum
inst_executed_fp64_pipe_s1	smsp__inst_executed_pipe_fp64.sum
inst_executed_fp64_pipe_s2	smsp__inst_executed_pipe_fp64.sum

CUPTI Event	Perfworks Metric or Formula
inst_executed_fp64_pipe_s3	smsp__inst_executed_pipe_fp64.sum
inst_issued1	sm__inst_issued.sum
l2_subp0_read_sector_misses	lts__t_sectors_op_read_lookup_miss.sum
l2_subp1_read_sector_misses	lts__t_sectors_op_read_lookup_miss.sum
l2_subp0_read_sysmem_sector_queries	lts__t_sectors_aperture_sysmem_op_read.sum
l2_subp1_read_sysmem_sector_queries	lts__t_sectors_aperture_sysmem_op_read.sum
l2_subp0_read_tex_hit_sectors	lts__t_sectors_srcunit_tex_op_read_lookup_hit.sum
l2_subp1_read_tex_hit_sectors	lts__t_sectors_srcunit_tex_op_read_lookup_hit.sum
l2_subp0_read_tex_sector_queries	lts__t_sectors_srcunit_tex_op_read.sum
l2_subp1_read_tex_sector_queries	lts__t_sectors_srcunit_tex_op_read.sum
l2_subp0_total_read_sector_queries	lts__t_sectors_op_read.sum + lts__t_sectors_op_atom.sum + lts__t_sectors_op_red.sum
l2_subp1_total_read_sector_queries	lts__t_sectors_op_read.sum + lts__t_sectors_op_atom.sum + lts__t_sectors_op_red.sum
l2_subp0_total_write_sector_queries	lts__t_sectors_op_write.sum + lts__t_sectors_op_atom.sum + lts__t_sectors_op_red.sum
l2_subp1_total_write_sector_queries	lts__t_sectors_op_write.sum + lts__t_sectors_op_atom.sum + lts__t_sectors_op_red.sum
l2_subp0_write_sector_misses	lts__t_sectors_op_write_lookup_miss.sum
l2_subp1_write_sector_misses	lts__t_sectors_op_write_lookup_miss.sum
l2_subp0_write_sysmem_sector_queries	lts__t_sectors_aperture_sysmem_op_write.sum
l2_subp1_write_sysmem_sector_queries	lts__t_sectors_aperture_sysmem_op_write.sum
l2_subp0_write_tex_hit_sectors	lts__t_sectors_srcunit_tex_op_write_lookup_hit.sum
l2_subp1_write_tex_hit_sectors	lts__t_sectors_srcunit_tex_op_write_lookup_hit.sum
l2_subp0_write_tex_sector_queries	lts__t_sectors_srcunit_tex_op_write.sum
l2_subp1_write_tex_sector_queries	lts__t_sectors_srcunit_tex_op_write.sum
not_predicated_off_thread_inst_executed	smsp__thread_inst_executed_pred_on.sum
pcie_rx_active_pulse	
pcie_tx_active_pulse	
prof_trigger_00	
prof_trigger_01	
prof_trigger_02	
prof_trigger_03	
prof_trigger_04	
prof_trigger_05	
prof_trigger_06	

CUPTI Event	Perfworks Metric or Formula
prof_trigger_07	
inst_issued0	smsp__issue_inst0.sum
sm_cta_launched	sm__ctas_launched.sum
shared_load	smsp__inst_executed_op_shared_ld.sum
shared_store	smsp__inst_executed_op_shared_st.sum
generic_load	smsp__inst_executed_op_generic_ld.sum
generic_store	smsp__inst_executed_op_generic_st.sum
global_load	smsp__inst_executed_op_global_ld.sum
global_store	smsp__inst_executed_op_global_st.sum
local_load	smsp__inst_executed_op_local_ld.sum
local_store	smsp__inst_executed_op_local_st.sum
shared_atom	smsp__inst_executed_op_shared_atom.sum
shared_atom_cas	smsp__inst_executed_op_shared_atom_dot_cas.sum
shared_ld_bank_conflict	l1tex__data_bank_conflicts_pipe_lsu_mem_shared_op_ld.sum
shared_st_bank_conflict	l1tex__data_bank_conflicts_pipe_lsu_mem_shared_op_st.sum
shared_ld_transactions	l1tex__data_pipe_lsu_wavefronts_mem_shared_op_ld.sum
shared_st_transactions	l1tex__data_pipe_lsu_wavefronts_mem_shared_op_st.sum
tensor_pipe_active_cycles_s0	smsp__pipe_tensor_cycles_active.sum
tensor_pipe_active_cycles_s1	smsp__pipe_tensor_cycles_active.sum
tensor_pipe_active_cycles_s2	smsp__pipe_tensor_cycles_active.sum
tensor_pipe_active_cycles_s3	smsp__pipe_tensor_cycles_active.sum
thread_inst_executed	smsp__thread_inst_executed.sum
warps_launched	smsp__warps_launched.sum

2.9. Migration to the Profiling API

The CUPTI event APIs from the header `cupti_events.h` and metric APIs from the header `cupti_metrics.h` will be deprecated in a future CUDA release. The NVIDIA Volta platform is the last architecture on which these APIs are supported. These are being replaced by the Profiling API in the header `cupti_profiler_target.h` and Perfworks Metric API in the headers `nvperf_host.h` and `nvperf_target.h`. These provide low and deterministic profiling overhead on the target system. These APIs also have other significant enhancements such as:

- Range Profiling
- Improved metrics
- Lower overhead for PC Sampling

GPU architectures supported by different CUPTI APIs are listed at the table. Both the event and metric APIs and the profiling APIs are supported for Volta. This is to enable transition of code to the profiling APIs. But one cannot mix the usage of the event and metric APIs and the profiling APIs.

The Profiling APIs are supported on all CUDA supported platforms except Android.

It is important to note that for support of future GPU architectures and feature improvements (such as performance overhead reduction and additional performance metrics), users should use the Profiling APIs. There are few features which are not supported by Profiling APIs, refer to the section for differences from event and metric APIs.

However note that there are no changes to the CUPTI Activity and Callback APIs and these will continue to be supported for the current and future GPU architectures.

2.10. CUPTI PC Sampling API

A new set of CUPTI APIs for PC sampling data collection are provided in the header file `cupti_pcsampling.h` which support continuous mode data collection without serializing kernel execution and have a lower runtime overhead. Along with these a utility library is provided in the header file `cupti_pcsampling_util.h` which has APIs for GPU assembly to CUDA-C source correlation and for reading and writing the PC sampling data from/to files.

The PC Sampling APIs are supported on all CUDA supported platforms. These are supported on Volta and later GPU architectures, i.e. devices with compute capability 7.0 and higher.

Overview of Features:

- Two sampling modes – Continuous (concurrent kernels) or Serialized (one kernel at a time).
- Option to select stall reasons to collect.
- Ability to collect GPU PC sampling data for entire application duration or for specific CPU code ranges (defined by start and stop APIs).
- API to flush GPU PC sampling data.
- APIs to support Offline and Runtime correlation of GPU PC samples to CUDA C source lines and GPU assembly instructions.

Samples are provided to demonstrate how to write the injection library to collect the PC sampling information, and how to parse the generated files using the utility APIs to print the stall reasons counter values and associate those with the GPU assembly

instructions and CUDA-C source code. Refer to the samples pc_sampling_continuous, pc_sampling_utility and pc_sampling_start_stop.

PC Sampling APIs from the header `cupti_activity.h` would be referred as *PC Sampling Activity APIs* and APIs from the header `cupti_pcsampling.h` would be referred as *PC Sampling APIs*.

2.10.1. Configuration Attributes

The following table lists the PC sampling configuration attributes which can be set using the `cuptiPCSamplingSetConfigurationAttribute()` API.

Table 9 PC Sampling Configuration Attributes

Configuration Attribute	Description	Default Value	Comparison of PC Sampling APIs with CUPTI PC Sampling Activity APIs	Guideline to Tune Configuration Option
Collection mode	PC Sampling collection mode - Continuous or Kernel Serialized	Continuous	Continuous mode is new. Kernel Serialized mode is equivalent to the kernel level functionality provided by the CUPTI PC sampling Activity APIs.	
Sampling period	Sampling period for PC Sampling. Valid values for the sampling periods are between 5 to 31 both inclusive. This will set the sampling period to (2^samplingPeriod) cycles. e.g. for sampling period = 5 to 31, cycles = 32, 64, 128,..., 2^31	CUPTI defined value is based on number of SMs	Dropped current support for 5 levels(MIN, LOW, MID, HIGH, MAX) for sampling period. The new "sampling period" is equivalent to the "samplingPeriod2" field in CUpti_ActivityPCSamplingConfig.	Low sampling period means a high sampling frequency which can result in dropping of samples. Very high sampling period can cause low sampling frequency and no sample generation.
Stall reason	Stall reasons to collect Input is a pointer to an array of the stall reason indexes to collect.	All stall reasons will be collected	With the CUPTI PC sampling Activity APIs there is no option to select which stall reasons to collect. Also the	

Configuration Attribute	Description	Default Value	Comparison of PC Sampling APIs with CUPTI PC Sampling Activity APIs	Guideline to Tune Configuration Option
			list of supported stall reasons has changed.	
Scratch buffer size	Size of SW buffer for raw PC counter data downloaded from HW buffer. Approximately it takes 16 Bytes (and some fixed size memory) to accommodate one PC with one stall reason e.g. 1 PC with 1 stall reason = 32 Bytes 1 PC with 2 stall reason = 48 Bytes 1 PC with 4 stall reason = 96 Bytes	1 MB (which can accommodate approximately 5500 PCs with all stall reasons)	New	Clients can choose scratch buffer size as per memory budget. Very small scratch buffer size can cause runtime overhead as more iterations would be required to accommodate and process more PC samples
Hardware buffer size	Size of HW buffer in bytes. If sampling period is too less, HW buffer can overflow and drop PC data	512 MB	New	Device accessible buffer for samples. Less hardware buffer size with low sampling periods, can cause overflow and dropping of PC data. High hardware buffer size can impact application execution due to lower amount of device memory being available
Enable start/ stop control	Control over PC Sampling data collection range. 1 - Allows user to start and stop PC Sampling using APIs	0 (disabled)	New	

2.10.2. Stall Reasons Mapping Table

The table below lists the stall reasons mapping from PC Sampling Activity APIs to PC Sampling APIs. Note: Stall reasons with suffix _not_issued represents latency samples. These samples indicate that no instruction was issued in that cycle from the warp scheduler from where the warp was sampled.

Table 10 Stall Reasons Mapping Table from PC Sampling Activity APIs to PC Sampling APIs

PC Sampling Activity API Stall Reasons (common prefix: CUPTI_ACTIVITY_PC_SAMPLIN	PC Sampling API Stall Reasons (common prefix: smsp__pcsamp_warps_issue_stalled_)
NONE	selected selected_not_issued
INST_FETCH	branch_resolving branch_resolving_not_issued no_instructions no_instructions_not_issued
EXEC_DEPENDENCY	short_scoreboard short_scoreboard_not_issued wait wait_not_issued
MEMORY_DEPENDENCY	long_scoreboard long_scoreboard_not_issued
TEXTURE	tex_throttle tex_throttle_not_issued
SYNC	barrier barrier_not_issued membar membar_not_issued
CONSTANT_MEMORY_DEPENDENCY	imc_miss imc_miss_not_issued
PIPE_BUSY	mio_throttle mio_throttle_not_issued math_pipe_throttle math_pipe_throttle_not_issued
MEMORY_THROTTLE	drain drain_not_issued lg_throttle lg_throttle_not_issued
NOT_SELECTED	not_selected not_selected_not_issued

PC Sampling Activity API Stall Reasons (common prefix: CUPTI_ACTIVITY_PC_SAMPLIN	PC Sampling API Stall Reasons (common prefix: smsp__pcsamp_warps_issue_stalled_)
OTHER	misc misc_not_issued dispatch_stall dispatch_stall_not_issued
SLEEPING	sleeping sleeping_not_issued

2.10.3. Data Structure Mapping Table

The table below lists the data structure mapping from PC Sampling Activity APIs to PC Sampling APIs.

Table 11 Data structure Mapping Table from PC Sampling Activity APIs to PC Sampling APIs

PC Sampling Activity API structures	PC Sampling API structures
CUpti_ActivityPCSamplingConfig	CUpti_PCSamplingConfigurationInfo
CUpti_ActivityPCSamplingStallReason	CUpti_PCSamplingStallReason Refer Stall Reasons Mapping Table
CUpti_ActivityPCSampling3	CUpti_PCSamplingPCData
CUpti_ActivityPCSamplingRecordInfo	CUpti_PCSamplingData

2.10.4. Data flushing

CUPTI clients can periodically flush GPU PC sampling data using the API `cuptiPCSamplingGetData()`. Besides periodic flushing of GPU PC sampling data, CUPTI clients need to also flush the GPU PC sampling data at the following points to maintain the uniqueness of PCs:

- For continuous collection mode CUPTI_PC_SAMPLING_COLLECTION_MODE_CONTINUOUS - after each module load-unload-load sequence.
- For serialized collection mode CUPTI_PC_SAMPLING_COLLECTION_MODE_KERNEL_SERIALIZED - after completion of each kernel.

- For range profiling using the configuration option CUPTI_PC_SAMPLING_CONFIGURATION_ATTR_TYPE_ENABLE_START_STOP_CONTROL - at the end of the range i.e. after `cuptiPCSamplingStop()` API.

If application is profiled in the continuous collection mode with range profiling disabled, and there is no module unload, CUPTI clients can collect data in two ways:

- By using `cuptiPCSamplingGetData()` API periodically.
- By using `cuptiPCSamplingDisable()` on application exit and reading GPU PC sampling data from sampling data buffer passed during configuration.

In case, `cuptiPCSamplingGetData()` API is not called periodically, the sampling data buffer passed during configuration should be big enough to hold the data for all the PCs.

Field `remainingNumPcs` of the struct `CUpti_PCSamplingData` helps in identifying the number of PC records available with CUPTI. User can adjust the periodic flush interval based on it. Further user need to ensure that all remaining records can be accommodated in the sampling data buffer passed during configuration before disabling the PC sampling.

2.10.5. SASS Source Correlation

Building SASS source correlation for a PC can be split into two parts:

- **Correlation of a PC to a SASS instruction** - PC to SASS correlation is done during PC sampling at run time and the SASS data is available in the PC record. Fields `cubinCrc`, `pcOffset` and `functionName` in the PC record help in correlatation of a PC with a SASS instruction. You can extract cubins from the application executable or library using the `cuobjdump` utility by executing the command **`cuobjdump -xelf all exe/lib`**. The cuobjump utility version should match with the CUDA Toolkit version used to build the CUDA application executable or library files. You can find the cubinCrc for extracted cubins using the `cuptiGetCubinCrc()` API. With the help of cubinCrc you can find out the cubin to which a PC belongs. The cubin can be disassembled using the `nvdisasm` utility that comes with the CUDA toolkit.
- **Correlation of a SASS instruction to a CUDA source line** - Correlation of GPU PC samples to CUDA C source lines can be done offline as well as at runtime with the help of the `cuptiGetSassToSourceCorrelation()` API.

JIT compiled cubins - In case of JIT compiled cubins, it is not possible to extract the cubin from the executable or library. For this case one can subscribe to one of the `CUPTI_CBID_RESOURCE_MODULE_LOADED` or `CUPTI_CBID_RESOURCE_MODULE_UNLOAD_STARTING` or

`CUPTI_CBID_RESOURCE_MODULE_PROFILED` callbacks. It returns a `CUpti_ModuleResourceData` structure having the CUDA binary. This binary can be stored in a file and can be used for offline CUDA C source correlation.

2.10.6. API Usage

Here is a pseudo code which shows how to collect the PC sampling data for specific CPU code ranges:

```
void Collection()
{
    // Select collection mode
    CUpti_PCSamplingConfigurationInfoParams pcSamplingConfigurationInfoParams =
 {};

    CUpti_PCSamplingConfigurationInfo collectionMode = {};
    collectionMode.attributeData.collectionModeData.collectionMode =
CUPTI_PC_SAMPLING_COLLECTION_MODE_CONTINUOUS;

    pcSamplingConfigurationInfoParams.numAttributes = 1;
    pcSamplingConfigurationInfoParams.pPCSamplingConfigurationInfo =
&collectionMode;

cuptiPCSamplingSetConfigurationAttribute(&pcSamplingConfigurationInfoParams);

    // Select stall reasons to collect
    {
        // Get number of supported stall reasons
        cuptiPCSamplingGetNumStallReasons();
        // Get number of supported stall reason names and corresponding indexes
        cuptiPCSamplingGetStallReasons();
        // Set selected stall reasons
        cuptiPCSamplingSetConfigurationAttribute();
    }

    // Select code range using start/stop APIs
    // Opt-in for start and stop PC Sampling using APIs cuptiPCSamplingStart and
cuptiPCSamplingStop
    CUpti_PCSamplingConfigurationInfo enableStartStop = {};
    enableStartStop.attributeType =
CUPTI_PC_SAMPLING_CONFIGURATION_ATTR_TYPE_ENABLE_START_STOP_CONTROL;

enableStartStop.attributeData.enableStartStopControlData.enableStartStopControl
= true;

    pcSamplingConfigurationInfoParams.numAttributes = 1;
    pcSamplingConfigurationInfoParams.pPCSamplingConfigurationInfo =
&enableStartStop;

cuptiPCSamplingSetConfigurationAttribute(&pcSamplingConfigurationInfoParams);

    // Enable PC Sampling
    cuptiPCSamplingEnable();

    kernelA <<<blocks, threads, 0, s0>>>(...);                  // KernelA is
not sampled

    // Start PC sampling collection
    cuptiPCSamplingStart();
    {
        // KernelB and KernelC might run concurrently since 'continuous'
sampling collection mode is selected
        kernelB <<<blocks, threads, 0, s0>>>(...);              // KernelB is
sampled
        kernelC <<<blocks, threads, 0, s1>>>(...);              // KernelC is
sampled
    }
    // Stop PC sampling collection
    cuptiPCSamplingStop();
    // Flush PC sampling data
    cuptiPCSamplingGetData();

    kernelD <<<blocks, threads, 0, s0>>>(...);                  // KernelD is
not sampled
```

2.10.7. Limitations

Known limitations and issues:

- In the serial mode, PC Sampling APIs do not provide information for correlation of PC sampling data for a kernel to the CUDA kernel launch API. This is supported by the PC Sampling activity APIs. For continuous mode, this cannot be supported due to hardware limitations.
- PC Sampling APIs don't support simultaneous sampling of multiple CUDA contexts on a GPU. However, simultaneous sampling of single CUDA context per GPU is supported. Before enabling and configuring the PC sampling on a different CUDA context on the same GPU, PC sampling needs to be disabled on the other context.

2.11. CUPTI SASS Metric API

The SASS metric APIs support collecting metric data at SASS assembly instruction level. These support a larger set of SASS instruction level metrics compared to the CUPTI Activity APIs. The set of sass metrics supported for each GPU architecture can be queried. These APIs are supported on Volta and later GPU architectures, i.e. devices with compute capability 7.0 and higher.

These APIs support SASS instruction to CUDA C source line correlation in offline mode. Hence the runtime overhead during data collection is lower.

2.11.1. API usage

- **Enumerate metrics:** Use the API **`cuptiSassMetricsGetNumOfMetrics()`** for the number of metrics supported by the chip. Then allocate the buffer of type **`CUpti_SassMetrics_MetricDetails`** and pass it to the API **`cuptiSassMetricsGetMetrics()`** where CUPTI will list out all the SASS metrics and put it in the user-allocated buffer.
- **Create config image:** For all the selected SASS metrics, create a list of **`CUpti_SassMetrics_Config`** structures. For creating the config buffer for a metric we need the metric id and the output granularity for the metric. The metric id can be queried by using the API **`cuptiSassMetricsGetProperties()`**. The output granularity tells at what level data will be collected. CUPTI supports collection at three levels -
  - CUPTI_SASS_METRICS_OUTPUT_GRANULARITY_GPU (at GPU level),
  - CUPTI_SASS_METRICS_OUTPUT_GRANULARITY_SM (at Streaming Multiprocessor level, the metric instance count will be the number of SMs present in the chip),

  - CUPTI_SASS_METRICS_OUTPUT_GRANULARITY_SMSP (SM sub-partition level, the number of instances will be the sum of all the SMSP present in the chip i.e num of SMs * num of sub-partitions in each SM)
- **Set config for the CUDA device:** API **`cuptiSassMetricsSetConfig()`** should be used for setting the config on the device for SASS metrics collection. This API takes the device index and list of **`CUpti_SassMetrics_Config`** stucts as input parameters. Then set the config for the device on which the kernel is running else CUPTI will report a **`CUPTI_ERROR_INVALID_OPERATION`** error.
- **Enable SASS metric profiling:** After setting the config for the CUDA device one needs to enable SASS patching for the context on which the kernel will be launched using the API **`cuptSassMetricsEnable()`**.

  CUPTI provides control over when the kernel will be patched. For Lazy patching mode, CUPTI will only patch the kernel at the first launch instance and then unpatch the kernel when the API **`cuptiSassMetricsDisable`** is called. Otherwise, CUPTI will patch all the kernels in the module for the context, regardless of whether kernels would be launched in the enable/disable range. Set the **`enableLazyPatching`** flag to enable the lazy patching mode for profiling. Lazy patching is suitable for applications that have a large number of kernels in the module and a small set of kernels are launched.
- **Flush SASS metric profiling data:** Once kernel execution is completed, metric data is stored in an internal format. One needs to query the size of the buffer to store the metrics data. API **`cuptiSassMetricsGetDataProperties()`** can be used to query the number of patched instructions and the number of hardware instances. Then allocate the buffer based on retrieved data, where CUPTI will flush the profiled metric data. For flushing the data, call the API **`cuptiSassMetricsFlushData()`**.
- **Disable SASS metric profiling:** Once the profiling of the kernel is done, call the API **`cuptiSassMetricsDisable()`** for resetting the patched kernel and remove all the profiled metric data which has been collected for the kernels.

  One thing to note is that CUPTI will remove all the metric data which has been collected for kernels launched since the API **`cuptiSassMetricsFlushData()`** call. So it is the user's responsibility to call flush data API for retrieving all the metric data. Calling API **`cuptiSassMetricsFlushData()`** after **`cuptiSassMetricsDisable()`** will report the error **`CUPTI_ERROR_INVALID_OPERATION`**.
- **Unset configuration for the CUDA device:** CUPTI maintains internal state for each CUDA device for which SASS metric collection is enabled. API **`cuptiSassMetricsUnsetConfig()`** should be called to clean-up the state. This API should be called for each device for which SASS metric collection has been configured.

2.11.2. Sample code

CUPTI sample **sass_metric** has two core functions – function *ListSupportedMetrics()* shows how to enumerate all metrics supported by the chip and function *CollectSassMetrics()* show how to collect SASS metrics.

Code snippet for enumerating SASS metrics (refer the *ListSupportedMetrics()* function in the CUPTI sass_metric sample):

```
CUpti_Device_GetChipName_Params
 getChipParams{ CUpti_Device_GetChipName_Params_STRUCT_SIZE };
cuptiDeviceGetChipName(&getChipParams);

CUpti_SassMetrics_GetNumOfMetrics_Params getNumOfMetricParams;
getNumOfMetricParams.pChipName = getChipParams.pChipName;
cuptiSassMetricsGetNumOfMetrics(&getNumOfMetricParams);

std::vector<CUpti_SassMetrics_MetricDetails>
 supportedMetrics(getNumOfMetricParams.numOfMetrics);
CUpti_SassMetrics_GetMetrics_Params getMetricsParams
 {CUpti_SassMetrics_GetMetrics_Params_STRUCT_SIZE};
getMetricsParams.pChipName = getChipParams.pChipName;
getMetricsParams.pMetricsList = supportedMetrics.data();
getMetricsParams.numOfMetrics = supportedMetrics.size();
cuptiSassMetricsGetMetrics(&getMetricsParams);
for (size_t i = 0; i < supportedMetrics.size(); ++i)
{
    std::cout << "Metric Name: " << supportedMetrics[i].pMetricName
            << ", MetricID: " << supportedMetrics[i].metricId
            << ", Metric Description: " <<
 supportedMetrics[i].pMetricDescription << "\n";
}
```

Code snippet for collecting SASS metrics (refer the *CollectSassMetrics()* function in the CUPTI sass_metric sample):

```
cuptiSassMetricsSetConfig();

// Enable SASS Patching
sassMetricsEnableParams.enableLazyPatching = 1;
cuptiSassMetricsEnable();

// As lazy patching has been enabled, VectorAdd will be patched here at the
 first launch instance
VectorAdd<<<gridSize, blockSize>>>();

cuptiSassMetricsGetDataProperties();

if (getDataPropParams.numOfInstances != 0 &&
 getDataPropParams.numOfPatchedInstructionRecords != 0)
{
    // allocate memory for getting patched data.

    flushDataParams.numOfInstances = getDataPropParams.numOfInstances;
    flushDataParams.numOfPatchedInstructionRecords =
 getDataPropParams.numOfPatchedInstructionRecords;
    flushDataParams.pMetricsData =

 (CUpti_SassMetrics_Data*)malloc(getDataPropParams.numOfPatchedInstructionRecords
 * sizeof(CUpti_SassMetrics_Data));

    for (size_t recordIndex = 0;
         recordIndex < getDataPropParams.numOfPatchedInstructionRecords;
         ++recordIndex)
    {
        flushDataParams.pMetricsData[recordIndex].pInstanceValues =
            (CUpti_SassMetrics_InstanceValue*)
 malloc(getDataPropParams.numOfInstances *
 sizeof(CUpti_SassMetrics_InstanceValue));
    }

    cuptiSassMetricsFlushData();
    // Store the data for post-processing the data (e.g. SASS to source
 correlation)
    // Cleanup memory
}

// As this is the first VectorSub launch, the patching will be done here.
VectorSub<<<gridSize, blockSize>>>();

// As cuptiSassMetricsFlushData() API is not called, VectorSub SASS metric data
 will be discarded.
// All the kernels which were patched earlier will be reset to its original
 state.
cuptiSassMetricsDisable();

// VectorMultiply function will not get patched as it is called outside the
 enable/disable range.
VectorMultiply<<<gridSize, blockSize>>>();

cuptiSassMetricsUnsetConfig();
```

2.12. CUPTI Checkpoint API

Starting with CUDA 11.5, CUPTI ships with a new library to assist tool developers who wish to replay kernels under direct control, such as tools using the Profiling API User Replay mode. This new `Checkpoint` library provides support for automatically saving and restoring device state for many common uses.

A device checkpoint is a managed copy of device functional state - including values in memory, along with some (but not all) other user visible state of the device. When a checkpoint is saved, this state is saved to internal buffers, preferentially using free device, then host, and finally filesystem space to save the data. The user tool maintains a handle to a checkpoint, and is able to restore the checkpoint with a single call, restoring the state so a kernel may be re-executed and expect to have the same device state as when the checkpoint was saved.

Once saved, a checkpoint may be restored any time including after multiple kernels have been launched, though currently there are limitations on which user calls (CUDA or driver API calls) have been validated to work between a `Save` and `Restore`. It currently is known safe to launch multiple kernels on a context and to do memcpy calls before restoring a checkpoint. Future versions of CUPTI will extend this to support additional API calls between a `Save` and `Restore`.

Checkpoints may be saved during injected kernel launch callbacks or directly coded into a target application.

Certain APIs are known to not work with the version of the `Checkpoint` API shipped with CUPTI 11.5, including Stream Capture mode.

2.12.1. Usage

There is one header for the library, `cupti_checkpoint.h`, which needs to be included, and libcheckpoint needs to be linked in to the application or injection library. Though the checkpoint library doesn't depend on cupti, the error codes returned by the API are shared with cupti, so linking libcupti in is needed in order to translate the return codes to string representations.

The `Checkpoint` API follows a similar design to other `CUPTI` APIs. API behavior is controlled through a structure, **`CUpti_Checkpoint`**, which is initialized by a tool or application, then passed to **`cuptiCheckpointSave`**. If the call is successful, the structure saves a handle to a checkpoint. At this point, the application may make a series of calls which modify device state (kernels which update memory, memcopies, etc), and when the device state should be restored, the tool can use the same structure in calls to **`cuptiCheckpointRestore`**, and finally a call to **`cuptiCheckpointFree`** to release the resources used by the checkpoint object.

Multiple checkpoints may be saved at the same time. If multiple checkpoints exist, they operate entirely independently - each checkpoint consumes the full resources needed to restore the device state at the point it was saved. Order of operations between multiple checkpoints is not enforced by the API - while a common use for multiple checkpoints may be a nested pattern, it is also possible to interleave checkpoint operations.

Between a **cuptiCheckpointSave** and **cuptiCheckpointRestore**, any number of standard kernel launches (or equivalent API calls such as **cuLaunchKernel**) or memcpy calls may be made. Additionally, any host (cpu) side calls may be made that do not affect device state. It is possible that other CUDA or driver API calls may be made, but have not been validated with the 11.5 release.

Several options exist in the **CUpti_Checkpoint** structure. They must be set prior to the initial **cuptiCheckpointSave** using that structure. Any further changes to the structure are ignored until after a call to **cuptiCheckpointFree**, at which point the structure can be re-configured and re-used.

Important per-checkpoint options:

- **structSize** - must be set to the value of **CUpti_Checkpoint_STRUCT_SIZE**
- **ctx** - if NULL, the checkpoint will be of the default CUDA context, otherwise, specifies which context
- **reserveDeviceMB** - Restrict a checkpoint save from using at least this much device memory
- **reserveHostMB** - Restrict a checkpoint save from using at least this much host memory
- **allowOverwrite** - It is normally an error to call Save using an existing checkpoint handle (one which has not been Freed). When set, this option allows the Save operation to be called multiple times on a handle. Note that when using this option, the **CUpti_Checkpoint** options are not re-read on any subsequent Save. To read new options, the handle must be passed to **cuptiCheckpointFree** prior to the **cuptiCheckpointSave** call.
- **optimizations** - Bitmask of options for checkpoint behavior
  - **CUPTI_CHECKPOINT_OPT_TRANSFER** - Normally when restoring a checkpoint, all existing device memory at the time of the save is restored. This optimization adds a test to see whether a block of memory has changed before restoring it and caches the results for subsequent calls to Restore. Use of this option requires that all Restore calls be done at the same point in an application for a given checkpoint. As the optimization may be computationally expensive, it is most useful when there is a significant amount of data that can be skipped and there will be several calls to Restore the checkpoint.

2.12.2. Restrictions

Checkpoints API calls may not be made during a stream capture. They also may not be inserted into a graph. Beyond kernel launches (cuLaunchKernel, standard kernel<<<>>> launches, etc) and memcpy calls, the remaining CUDA and driver API calls have not been validated within a `Checkpoint` **`Save`** and **`Restore`** region. Any other CUDA or driver API calls (example - device malloc or free) may work, or may cause undetermined behavior. Additional APIs will be validated to work with the `Checkpoint` API in future releases.

The Checkpoint API does not have visibility into which API calls have been made between **`cuptiCheckpointSave`** and **`cuptiCheckpointRestore`** calls, and may not be able to correctly detect error cases if unsupported calls have been made. In this case it is possible that device state may only be partially restored by **`cuptiCheckpointRestore`**, which may casue functionally incorrect behavior in subsequent device calls.

The Checkpoint API only restores functionally visible device state, not performance critical state. Some performance characteristics, such as state of the caches, will not be saved by a checkpoint, and saving or restoring a checkpoint may change the occupancy and alter performance for subsequent device calls.

The Checkpoint API makes no attempt to restore host (non-device) state, beyond freeing the resources it internally uses during a call to **`cuptiCheckpointFree`**.

The Checkpoint API by default uses device memory, host memory, and finally the filesystem to back up the device state. It is possible that addition of a **`cuptiCheckpointSave`** causes a later device allocation to fail due to the increased device memory usage. (Similarly, host memory is also used, and may be affected by a checkpoint). To allow the user to guarantee a certain amount of device or host memory remains available for later use, **`reserveDeviceMB`** and **`reserveHostMB`** fields in the **`CUpti_Checkpoint`** struct may be set. Use of these fields will guarantee that the device or host memory will leave that much memory free during a **`cuptiCheckpointSave`** call, but may cause the Checkpoint API call performance to degrade due to increased use of slower storage spaces.

2.12.3. Examples

The `Checkpoint` API does not require any other `CUPTI` calls. A simple use case could be to compare the output of three different implementations of a kernel. Pseudocode for this could look like:

```
CUpti_Checkpoint cp = { CUpti_Checkpoint_STRUCT_SIZE };

int kernel = 0;
do
{
  if (kernel == 0)
    cuptiCheckpointSave(&cp);
  else
    cuptiCheckpointRestore(&cp);

  if (kernel == 0)
    kernel_1<<<>>>(...);
  else if (kernel == 1)
    kernel_2<<<>>>(...);
  else if (kernel == 2)
    kernel_3<<<>>>(...);
} while (kernel++ < 3);

cuptiCheckpointFree(&cp);
```

In this example, even if any of the kernels modify their own input data, the subsequent passes through the loop will still run correctly - the modified input data would be restored by each call to **`cuptiCheckpointRestore`** before the next kernel runs. This is particularly useful when a programmer does not know the exact state of the device prior to a kernel call - the Checkpoint API ensures that all needed data is saved and restored, which would not otherwise be practical or perhaps even possible in some complex cases.

Another possible use case could be for fuzzing - randomly modifying input to a kernel, and ensuring it performs as expected. Instead of manually restoring device state to a known good point, the Checkpoint API and initialize a good state, and the fuzzer can modify only what is needed.

```
CUpti_Checkpoint cp = { CUpti_Checkpoint_STRUCT_SIZE };

int i = 0;
do
{
  if (i == 0)
    cuptiCheckpointSave(&cp);
  else
    cuptiCheckpointRestore(&cp);

  setup_test<<<>>>(i, ...);

  kernel<<<>>>(...);

  validate_result<<<>>>(i, ...);
} while (i++ < num_tests);

cuptiCheckpointFree(&cp);
```

Finally, the Checkpoint API is very useful for the User Replay mode of the CUPTI Profiling API. The User Replay mode can be very desireable as it allows kernels to run concurrently, which Kernel Replay mode does not, and only replays parts of the application which are within a performance region, unlike Applicatin Replay mode. However, in this mode, a kernel potentially needs to be launched multiple times in order to gather all requested metrics. This is complicated when the kernel may modify some of its own input data, and without the Checkpoint API, would require the tool developer to handle restoring any modified input data manually. It is difficult for a tool to automatically know whether any data needs to be restored before each iteration, or even what the existing state of the device is. Using the Checkpoint API, the tool can guarantee that input data will be restored each pass.

```
CUpti_Checkpoint cp = { CUpti_Checkpoint_STRUCT_SIZE };

// Pseudocode - assume all Profiling API structures are already initialized
 correctly
cuptiProfilerBeginSession(&beginSessionParams);
cuptiProfilerSetConfig(&setConfigParams);
int numPasses = 0;
bool lastPass = false;
do
{
  if (numPasses == 0)
    cuptiCheckpointSave(&cp);
  else
    cuptiCheckpointRestore(&cp);

  cuptiProfilerBeginPass(&beginPassParams);
  cuptiProfilerEnableProfiling(&enableProfilingParams);
  cuptiProfilerPushRange(&pushRangeParams);

  // Kernel launch on N separate streams - will be profiled while running
 concurrently
  kernel<<<..., stream0>>>(...);
  kernel<<<..., stream1>>>(...);
  ...
  kernel<<<..., streamN>>>(...);

  cudaStreamSynchronize(stream0);
  cudaStreamSynchronize(stream1);
  ...
  cudaStreamSynchronize(streamN);

  cuptiProfilerPopRange(&popRangeParams);
  cuptiProfilerDisableProfiling(&disableProfilingParams);
  lastPass = cuptiProfilerEndPass(&endPassParams);
} while (lastPass == false);
cuptiProfilerFlushCounterData(&flushCounterDataParams);
cuptiProfilerUnsetConfig(&unsetConfigParams);
cuptiProfilerEndSession(&endSessionParams);
```

In this example, the Profiler range will span all concurrently running kernels, which may modify their own input data - each pass through the loop will restore the initial values.

2.13. CUPTI overhead

CUPTI incurs overhead when used for tracing or profiling of the CUDA application. Overhead can vary significantly from one application to another. It largely depends on the density of the CUDA activities in the application; lesser the CUDA activities, less the CUPTI overhead. In general overhead of tracing i.e. activity APIs is much lesser than the profiling i.e. event and metric APIs.

2.13.1. Tracing Overhead

One of the goal of the tracing APIs is to provide a non-invasive collection of the timing information of the CUDA activities. Tracing is a low-overhead mechanism for collecting fine-grained runtime information.

2.13.1.1. Execution overhead

Factors affecting the execution overhead under tracing are:

- Serial kernel trace enabled using the activity kind `CUPTI_ACTIVITY_KIND_KERNEL` can significantly change the overall performance characteristics of the application because all kernel executions are serialized on the GPU. For applications which use only a single CUDA stream and therefore cannot have concurrent kernel execution, this mode can be useful as it usually (not always) incurs less profiling overhead compared to the concurrent kernel mode.
- Concurrent kernel trace enabled using the activity kind `CUPTI_ACTIVITY_KIND_CONCURRENT_KERNEL` doesn't affect the concurrency of the kernels in the application. In this mode, CUPTI instruments the kernel code to collect the timing information. A single instrumentation code is generated at the time of loading the CUDA module and applied to each kernel during the kernel execution. Instrumentation code generation overhead is attributed as `CUPTI_ACTIVITY_OVERHEAD_CUPTI_INSTRUMENTATION` in the activity record `CUpti_ActivityOverhead2`.
- Due to the code instrumentation, concurrent kernel mode can add significant runtime overhead if used on kernels that execute a large number of blocks and that have short execution durations.

2.13.1.2. Memory overhead

CUPTI allocates device and pinned system memory for storing the tracing information:

- **Static memory allocation:** CUPTI allocates 3 buffers of 3 MB each in the pinned system memory for each CUDA context by default during the context creation phase. This is used for storing the concurrent kernel, serial kernel, memcopy and memset tracing information and these buffers are sufficient for storing information

for about 300K such activities. The number of buffers is controlled using the attribute `CUPTI_ACTIVITY_ATTR_DEVICE_BUFFER_PRE_ALLOCATE_VALUE` and the size of the buffer is determined by the attribute `CUPTI_ACTIVITY_ATTR_DEVICE_BUFFER_SIZE`. User can change the buffer size at any time during the profiling session, but this setting takes effect only for new buffer allocations. It is recommended to adjust the buffer size before the creation of any CUDA context to make sure that all the pre-allocated buffers are of the adjusted size.

- **Dynamic memory allocation:** Once profiling buffers to store the tracing information are exhausted, CUPTI allocates another buffer of the same size. Note that memory footprint will not always scale with the kernel, memcopy, memset count because CUPTI reuses the buffer after processing all the records in the buffer. For applications with a high density of these activities CUPTI may allocate more buffers.

All of the CUPTI allocated memory associated with a context is freed when the context is destroyed. Memory allocation overhead is attributed as `CUPTI_ACTIVITY_OVERHEAD_CUPTI_RESOURCE` in the activity record `CUpti_ActivityOverhead2`. If there are no CUDA contexts created then CUPTI will not allocate corresponding buffers.

CUPTI allocates memory to store unique kernel names, NVTX ranges, CUDA module cubin:

- **Kernel trace:** For kernel tracing enabled using the activity kind `CUPTI_ACTIVITY_KIND_KERNEL` or `CUPTI_ACTIVITY_KIND_CONCURRENT_KERNEL` CUPTI allocates memory to store the kernel name in the records. It is recommended to not free the memory allocated for the kernel name in the kernel activity record as the kernel name memory space might be common across all kernel records having the same kernel name.
- **NVTX ranges:** For NVTX enabled using the activity kind `CUPTI_ACTIVITY_KIND_MARKER` CUPTI allocates memory to store the range name in the records. It is recommended to not free the memory allocated for the NVTX range name in the marker activity record as the NVTX range name memory space will be common across all NVTX range records having the same name.
- **CUDA module cubin:** CUPTI caches copies of cubin images at the time of loading CUDA modules. This is done only for the profiling features that need it are enabled. All of the CUPTI allocated memory associated with the cubin image of the module is freed when the module is unloaded.

2.13.2. Profiling Overhead

Events and metrics collection using CUPTI incurs runtime overhead. This overhead depends on the number and type of events and metrics selected. Since each metric is computed from one or more events, metric overhead depends on the number and type

of underlying events. The overhead includes time spent in configuration of hardware events and reading of hardware event values.

Factors affecting the execution overhead under profiling are:

- Overhead is less for hardware provided events and metrics.
  - For event and metric APIs, events which are collected using the collection method `CUPTI_EVENT_COLLECTION_METHOD_PM` or `CUPTI_EVENT_COLLECTION_METHOD_SM` fall in this category.
  - For Profiling APIs, metrics which don't have string "sass" in the name fall in this category.
- Software instrumented events and metrics are expensive as CUPTI needs to instrument the kernel to collect these. Further these events and metrics cannot be combined with any other event or metric in the same pass as otherwise instrumented code will also contribute to the event value.
  - For event and metric APIs, the collection method `CUPTI_EVENT_COLLECTION_METHOD_INSTRUMENTED` fall in this cateogry.
  - For Profiling APIs, metrics which have string "sass" in the name fall in this category.
- In the serial mode, profiling may significantly change the overall performance characteristics of the application because all kernel executions are serialized on the GPU. This is done to enable tight event or metric collection around each kernel.
  - For event and metric APIs, the collection mode `CUPTI_EVENT_COLLECTION_MODE_KERNEL`, serializes all kernel executions on the GPU that occur between the APIs `cuptiEventGroupEnable` and `cuptiEventGroupDisable`. On the other hand, kernel concurrency can be maintained by using the collection mode `CUPTI_EVENT_COLLECTION_MODE_CONTINUOUS` and restricting profiling to events and metrics that can be collected in a single pass.
  - For Profiling APIs, auto range mode serializes all kernel executions on the GPU. On the other hand, kernel concurrency can be maintained by using the user range mode.
- When all the requested events or metrics cannot be collected in the single pass due to hardware or software limitations, one needs to replay the exact same set of GPU workloads multiple times. This can be achieved at the kernel granularity by replaying kernel multiple times or by launching the entire application multiple times. CUPTI provides support for kernel replay only. Application replay can be done by the CUPTI client.
- When kernel replay is used the overhead to save and restore kernel state for each replay pass depends on the amount of device memory used by the kernel. Application replay is expected to perform better than kernel replay for the case when the size of device memory used by the kernel is high.

2.14. Reproducibility

Some CUPTI APIs are not guaranteed to return perfectly reproducible results between runs. Numerous factors introduce measurable run-to-run variation in software and hardware performance. There are several suggestions for users who want more reproducible results.

2.14.1. Fixed Clock Rate

Many metrics are directly affected by GPU SM and memory clock frequencies. By default, the GPU keeps clock rates low until work is launched, but clock rates do not boost to full speed immediately, so initial work launched after an idle period may run at low clock speed. Additionally, the target clock rates may vary based on power, thermal, and other factors. Complex interactions between different part of the system mean that these dynamic clock rates may not be reproducible between runs.

To reduce the effect of dynamic clock rates, it is possible to set a fixed clock rate. The GPU will no longer opportunistically boost clock rates above this rate, but it will eliminate the variability after GPU idle and effects of power and thermal variation. Several different methods exist to fix the SM or memory clock rates. The simplest may be nvidia-smi, but see this NVIDIA blog entry for more suggestions.

2.14.2. Serialization

Work may be submitted to the GPU which can run asynchronously and concurrently. This improves performance by using more of the GPU resources at once, but complicates profiling in two ways - first, kernels running concurrrently can impact each other through contention for shared resources. Measurements of these shared resources will include the impact of any concurrently kernels, and it may not be possible to determine the particular impact of any given kernel. Second, by contending for resources with other kernels that are running without precisely guaranteed timing, the timing for a given kernel may be impacted in irreproducible ways.

When `CUPTI_ACTIVITY_KIND_CONCURRENT_KERNEL` is used to measure kernel timing, kernels are allowed to run concurrently on device. `CUPTI_ACTIVITY_KIND_KERNEL` may be used instead to measure serialized kernel timing. This will eliminate GPU concurrency within this process, and should provide better run-to-run reproducibility, but the timing may not be as realistic in this mode - kernels will not have to contend for shared resources, which can impact their performance.

2.14.3. Other Issues

Beyond variable clock rates and conconcurrent kernel execution, several other factors can affect application and kernel performance.

The driver normally does not stay loaded when not in use. It takes some time to load and initialize the driver, which may affect performance in noticeable and somewhat irreproducibile ways. It is possible to keep the driver persistently loaded which will eliminate this initialization overhead. `nvidia-persistenced` is one tool to configure this; it can also be configured through `nvidia-smi`.

2.15. Samples

The CUPTI installation includes several samples that demonstrate the use of the CUPTI APIs. These samples can be referred to for the usage of different APIs supported by CUPTI. The samples are:

Activity API

activity_trace_async
This sample shows how to collect a trace of CPU and GPU activity using the new asynchronous activity buffer APIs.

callback_timestamp
This sample shows how to use the callback API to record a trace of API start and stop times.

cuda_graphs_trace
This sample shows how to collect the trace of CUDA graphs and correlate the graph node launch to the node creation API using CUPTI callbacks.

cuda_memory_trace
This sample shows how to collect the trace of CUDA memory operations. The sample also traces CUDA memory operations done via default memory pool.

cupti_correlation
This sample shows how to do the correlation between CUDA APIs and corresponding GPU activities.

cupti_external_correlation
This sample shows how to do the correlation of CUDA API activity records with external APIs.

cupti_finalize
This sample shows how to use API `cuptiFinalize()` to dynamically detach and attach CUPTI.

cupti_nvtx
This sample shows how to receive NVTX callbacks and collect NVTX records in CUPTI.

cupti_trace_injection
This sample shows how to build an injection library using the CUPTI activity and callback APIs. It can be used to trace CUDA APIs and GPU activities for any CUDA application. It does not require the CUDA application to be modified.

nvlink_bandwidth
This sample shows how to collect NVLink topology and NVLink throughput metrics in continuous mode.

openacc_trace
This sample shows how to use CUPTI APIs for OpenACC data collection.

pc_sampling
This sample shows how to collect PC Sampling profiling information for a kernel using the PC Sampling Activity APIs.

sass_source_map
This sample shows how to generate CUpti_ActivityInstructionExecution records and how to map SASS assembly instructions to CUDA C source.

unified_memory
This sample shows how to collect information about page transfers for unified memory.

Event and Metric APIs

callback_event
This sample shows how to use both the callback and event APIs to record the events that occur during the execution of a simple kernel. The sample shows the required ordering for synchronization, and for event group enabling, disabling, and reading.

callback_metric
This sample shows how to use both the callback and metric APIs to record the metric's events during the execution of a simple kernel, and then use those events to calculate the metric value.

cupti_query
This sample shows how to query CUDA-enabled devices for their event domains, events, and metrics.

event_multi_gpu
This sample shows how to use the CUPTI event and CUDA APIs to sample events on a setup with multiple GPUs. The sample shows the required ordering for synchronization, and for event group enabling, disabling, and reading.

event_sampling
This sample shows how to use the event APIs to sample events using a separate host thread.

Profiling API

extensions
This includes utilities used in some of the samples.

autorange_profiling
This sample shows how to use profiling APIs to collect metrics in autorange mode.

callback_profiling
This sample shows how to use callback and profiling APIs to collect the metrics during the execution of a kernel. It shows how to use different phases of profiling i.e. enumeration, configuration, collection and evaluation in the appropriate callbacks.

concurrent_profiling
This sample shows how to use the profiling API to record metrics from concurrent kernels launched in two different ways - using multiple streams on a single device, and using multiple threads with multiple devices.

cupti_metric_properties
This sample shows how to query various properties of metrics using the Profiling APIs. The sample shows collection method (hardware or software) and number of passes required to collect a list of metrics.

nested_range_profiling
This sample shows how to profile nested ranges using the Profiling APIs.

profiling_injection
This sample for Linux systems shows how to build an injection library which can automatically enable CUPTI's Profiling API using Auto Ranges with Kernel Replay mode. It can attach to an application which was not instrumented using CUPTI and profile any kernel launches.

userrange_profiling
This sample shows how to use profiling APIs to collect metrics in user specified range mode.

PC Sampling API

pc_sampling_continuous
This injection sample shows how to collect PC Sampling profiling information using the PC Sampling APIs. A perl script libpc_sampling_continuous.pl is provided to run the CUDA application with different PC sampling options. Use the command './libpc_sampling_continuous.pl --help' to list all the options. The CUDA application code does not need to be modified. Refer the README.txt file shipped with the sample for instructions to build and use the injection library.

pc_sampling_start_stop
This sample shows how to collect PC Sampling profiling information for kernels within a range using the PC Sampling start/stop APIs.

pc_sampling_utility
This utility takes the pc sampling data file generated by the pc_sampling_continuous injection library as input. It prints the stall reason counter values at the GPU assembly instruction level. It also does GPU assembly to CUDA-C source correlation and shows the CUDA-C source file name and line number. Refer the README.txt file shipped with the sample for instructions to build and run the utility.

SASS Metric API

sass_metric

This sample shows how to use the SASS metric API to enumerate metrics supported by a device and how to collect metrics at the source level using SASS patching.

Checkpoint API

checkpoint_kernels

This sample shows how to use the Checkpoint API to restore device memory, allowing a kernel to be replayed, even if it modifies its input data.

Chapter 3. LIBRARY SUPPORT

CUPTI can be used to profile CUDA applications, as well as applications that use CUDA via NVIDIA or third-party libraries. For most such libraries, the behavior is expected to be identical to applications using CUDA directly. However, for certain libraries, CUPTI has certain restrictions, or alternate behavior.

3.1. OptiX

CUPTI supports profiling of OptiX applications, but with certain restrictions.

Tracing

- **Internal Kernels**

  Kernels launched by OptiX that contain no user-defined code are given the generic name *NVIDIA internal*. CUPTI provides the tracing information for these kernels.
- **User Kernels**

  Kernels launched by OptiX can contain user-defined code. OptiX identifies these kernels with a custom name. This name starts with *raygen__* (for "ray generation"). These kernels can be traced.

Profiling

CUPTI can profile both internal and user kernels using the Profiling APIs. In the auto range mode, range names will be numeric values starting from 0 to total number of kernels including internal and user defined kernels or maximum number of range set while calling set config API, whichever is minimum.

It is suggested to create the profiling session and enable the profiling at resource allocation time (e.g. context creation) and disable the profiling at the context destruction time.

Limitations

- CUPTI doesn't issue any driver or runtime API callback for user kernels.

- Event, Metric and PC sampling APIs are not supported for OptiX applications.

Chapter 4. SPECIAL CONFIGURATIONS

4.1. Multi-Instance GPU (MIG)

Multi-Instance GPU (MIG) is a feature that allows a GPU to be partitioned into multiple CUDA devices. The partitioning is carried out on two levels: First, a GPU can be split into one or multiple GPU Instances. Each GPU Instance claims ownership of one or more streaming multiprocessors (SM), a subset of the overall GPU memory, and possibly other GPU resources, such as the video encoders/decoders. Second, each GPU Instance can be further partitioned into one or more Compute Instances. Each Compute Instance has exclusive ownership of its assigned SMs of the GPU Instance. However, all Compute Instances within a GPU Instance share the GPU Instance's memory and memory bandwidth. Every Compute Instance acts and operates as a CUDA device with a unique device ID. See the driver release notes as well as the documentation for the **`nvidia-smi`** CLI tool for more information on how to configure MIG instances.

From the profiling perspective, a Compute Instance can be of one of two types: *isolated* or *shared*.

An *isolated* Compute Instance owns all of it's assigned resources and does not share any GPU unit with another Compute Instance. In other words, the Compute Instance is of the same size as its parent GPU Instance and consequently does not have any other sibling Compute Instances. Tracing and Profiling works for isolated Compute Instances.

A *shared* Compute Instance uses GPU resources that can potentially also be accessed by other Compute Instances in the same GPU Instance. Due to this resource sharing, collecting profiling data from shared units is not permitted. Attempts to collect metrics from a shared unit will result in NaN values. Better error reporting will be done in a future release. Collecting metrics from GPU units that are exclusively owned by a shared Compute Instance is still possible. Tracing works for shared Compute Instances.

To allow users to determine which metrics are available on a target device, new APIs have been added which can be used to query counter availability before starting the

profiling session. See APIs `NVPW_RawMetricsConfig_SetCounterAvailability` and `cuptiProfilerGetCounterAvailability`.

All Compute Instances on a GPU share the same clock frequencies. To get consistent metric values with multi-pass collection, it is recommended to lock the GPU clocks during the profiling session. CLI tool **`nvidia-smi`** can be used to configure a fixed frequency for the whole GPU by calling **`nvidia-smi --lock-gpu-clocks=tdp,tdp`**. This sets the GPU clocks to the base TDP frequency until you reset the clocks by calling **`nvidia-smi --reset-gpu-clocks`**.

4.2. NVIDIA Virtual GPU (vGPU)

CUPTI supports tracing and profiling features on NVIDIA virtual GPUs (vGPUs). Activity, Callback and Profiling APIs are supported but Event and Metric APIs are not supported on NVIDIA vGPUs. If you want to use profiling features that NVIDIA vGPU supports, you must enable them for each vGPU VM that requires them. These can be enabled by setting a vGPU plugin parameter **`enable_profiling`**. How to set the parameter for a vGPU VM depends on the hypervisor that you are using. Tracing is enabled by default, it doesn't require any specific setting. However tracing results might not be accurate after virtual machine (VM) migration. Therefore it is recommended to set the vGPU plugin parameter **`enable_profiling`** for accurate results. Refer to the NVIDIA Virtual GPU Software documentation for the list of supported GPUs, how to enable profiling features using the vGPU plugin parameter and for limitations on use of CUPTI with NVIDIA vGPU.

4.3. Windows Subsystem for Linux (WSL)

WSL or Windows Subsystem for Linux is a Windows feature that enables users to run native Linux applications, containers and command-line tools directly on Windows 10 and later OS builds. CUPTI supports tracing APIs *Activity* and *Callback* on the second generation of WSL (WSL 2) on Volta and later GPU architectures. Profiler APIs *Event* and *Metric* are not supported on WSL while *Profiling* and *PC Sampling* APIs are only supported on WSL 2 and Windows 11 systems.

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication of otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2007-2023 NVIDIA Corporation and affiliates. All rights reserved.

This product includes software developed by the Syncro Soft SRL (http://www.sync.ro/).