



CUPTI

DA-05679-001 _v11.3 | May 2021

User's Guide



TABLE OF CONTENTS

| | |
|--|----|
| Overview..... | v |
| What's New..... | v |
| Chapter 1. Usage..... | 1 |
| 1.1. CUPTI Compatibility and Requirements..... | 1 |
| 1.2. CUPTI Initialization..... | 1 |
| 1.3. CUPTI Activity API..... | 2 |
| 1.3.1. SASS Source Correlation..... | 4 |
| 1.3.2. PC Sampling..... | 4 |
| 1.3.3. NVLink..... | 6 |
| 1.3.4. OpenACC..... | 6 |
| 1.3.5. CUDA Graphs..... | 7 |
| 1.3.6. External Correlation..... | 7 |
| 1.3.7. Dynamic Attach and Detach..... | 8 |
| 1.4. CUPTI Callback API..... | 8 |
| 1.4.1. Driver and Runtime API Callbacks..... | 9 |
| 1.4.2. Resource Callbacks..... | 11 |
| 1.4.3. Synchronization Callbacks..... | 11 |
| 1.4.4. NVIDIA Tools Extension Callbacks..... | 11 |
| 1.5. CUPTI Event API..... | 12 |
| 1.5.1. Collecting Kernel Execution Events..... | 15 |
| 1.5.2. Sampling Events..... | 16 |
| 1.6. CUPTI Metric API..... | 16 |
| 1.6.1. Metrics Reference..... | 18 |
| 1.6.1.1. Metrics for Capability 3.x..... | 18 |
| 1.6.1.2. Metrics for Capability 5.x..... | 26 |
| 1.6.1.3. Metrics for Capability 6.x..... | 35 |
| 1.6.1.4. Metrics for Capability 7.0..... | 44 |
| 1.7. CUPTI Profiling API..... | 53 |
| 1.7.1. Multi Pass Collection..... | 53 |
| 1.7.2. Range Profiling..... | 54 |
| 1.7.2.1. Auto Range..... | 54 |
| 1.7.2.2. User Range..... | 58 |
| 1.7.3. CUPTI Profiler Definitions..... | 60 |
| 1.8. Perfworks Metrics API..... | 60 |
| 1.8.1. Derived metrics..... | 63 |
| 1.8.2. Raw Metrics..... | 67 |
| 1.8.3. Metrics Mapping Table..... | 68 |
| 1.8.4. Events Mapping Table..... | 73 |
| 1.9. Migration to the Profiling API..... | 76 |
| 1.10. CUPTI PC Sampling API..... | 77 |

| | |
|---|-----------|
| 1.10.1. PC Sampling Configuration Attributes..... | 78 |
| 1.10.2. Stall Reasons Mapping Table..... | 80 |
| 1.10.3. Data Structure Mapping Table..... | 81 |
| 1.10.4. PC Sampling API Usage..... | 82 |
| 1.10.5. Limitations..... | 83 |
| 1.11. CUPTI overhead..... | 83 |
| 1.11.1. Tracing Overhead..... | 83 |
| 1.11.1.1. Execution overhead..... | 83 |
| 1.11.1.2. Memory overhead..... | 84 |
| 1.11.2. Profiling Overhead..... | 84 |
| 1.12. Multi Instance GPU..... | 85 |
| 1.13. Samples..... | 86 |
| Chapter 2. Library Support..... | 89 |
| 2.1. OptiX..... | 89 |
| Chapter 3. Limitations..... | 90 |
| Chapter 4. Changelog..... | 95 |

LIST OF TABLES

| | | |
|---------|--|----|
| Table 1 | Capability 3.x Metrics | 19 |
| Table 2 | Capability 5.x Metrics | 26 |
| Table 3 | Capability 6.x Metrics | 35 |
| Table 4 | Capability 7.x (7.0 and 7.2) Metrics | 44 |
| Table 5 | Metrics Mapping Table from CUPTI to Perfworks for Compute Capability 7.0 | 68 |
| Table 6 | Events Mapping Table from CUPTI events to Perfworks metrics for Compute Capability 7.0..... | 74 |
| Table 7 | PC Sampling Configuration Attributes | 78 |
| Table 8 | Stall Reasons Mapping Table from PC Sampling Activity APIs to PC Sampling APIs | 80 |
| Table 9 | Data structure Mapping Table from PC Sampling Activity APIs to PC Sampling APIs | 81 |

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* and the *Profiler 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.

What's New

CUPTI contains below change as part of the CUDA Toolkit 11.3 release.

- ▶ A new set of CUPTI APIs for PC sampling data collection are provided in the header file `cuprti_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 `cuprti_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.
- ▶ Overhead reduction for tracing of CUDA memcpyes.
- ▶ Enum `CUpri_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.
 - ▶ `cuprtiDeviceGetTimestamp` from the header `cuprti_events.h`.

Chapter 1.

USAGE

1.1. 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.

1.2. 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.

1.3. 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 initialized before CUDA initialization. All related activities occurring 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 perturbation for the application created threads. CUPTI offloads certain operations from the application threads to the worker thread, this includes 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 heuristics. 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.

1.3.1. 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.

1.3.2. 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 are introduced in CUDA 11.3 release, which support 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 new set of APIs from the header `cupti_pcsampling.h` would be referred as *PC Sampling APIs*.

1.3.3. 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 memory 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 (`nvlink__*` and `nvlink__*`) 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.

1.3.4. 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.

1.3.5. 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.

1.3.6. External Correlation

Starting with CUDA 8.0, 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.

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`.

1.3.7. 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 callsite of any of the CUDA Driver or Runtime API. Otherwise CUPTI client needs to make sure that required CUDA synchronization and CUPTI activity buffer flush is done before calling the API `cuptiFinalize()`. 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](#).

1.4. 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.

1.4.1. 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.

1.4.2. 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`.

1.4.3. 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`.

1.4.4. 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 = ...;
...
cuprtiSubscribe(&subscriber,
                (CUpti_CallbackFunc)my_callback , my_data);
cuprtiEnableDomain(1, subscriber,
                  CUPTI_CB_DOMAIN_NVTX);
```

First, `cuprtiSubscribe` is used to initialize a subscriber with the `my_callback` callback function. Next, `cuprtiEnableDomain` 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 `cuprtiEnableCallback` and `cuprtiEnableAllDomains` 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 CUPTI_API
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 == NVTX_CBID_CORE_NameOsThreadA)) {
        nvtxNameOsThreadA_params *params = (nvtxNameOsThreadA_params *)nvtxInfo->
            functionParams;
        ...
    }
    ...
}
```

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 `nvtxNameOsThreadA` 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 `nvtxNameOsThreadA`. To access the parameters, we first cast `functionParams` to a structure type corresponding to the `nvtxNameOsThreadA` function. These parameter structures are contained in `generated_nvtx_meta.h`.

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

1.5. 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.



Events 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](#).

1.5.1. 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`.

1.5.2. 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. To get some coarse correlation, you can use `cuptiDeviceGetTimestamp` to collect the GPU timestamp at the time of the sample and also at other interesting points in your application.

1.6. 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;
```



Metrics 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](#).

1.6.1. 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.

1.6.1.1. Metrics for Capability 3.x

Devices with compute capability 3.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 1 Capability 3.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 |
| alu_fu_utilization | The utilization level of the multiprocessor function units that execute integer and floating-point arithmetic instructions on a scale of 0 to 10 | Multi-context |
| atomic_replay_overhead | Average number of replays due to atomic and reduction bank conflicts for each instruction executed | Multi-context |
| atomic_throughput | Global memory atomic and reduction throughput | 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. This is available for compute capability 3.0. | 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 |
| dram_read_throughput | Device memory read throughput. This is available for compute capability 3.0, 3.5 and 3.7. | Multi-context* |
| dram_read_transactions | Device memory read transactions. This is available for compute capability 3.0, 3.5 and 3.7. | 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_throughput | Device memory write throughput. This is available for compute capability 3.0, 3.5 and 3.7. | Multi-context* |
| dram_write_transactions | Device memory write transactions. This is available for compute capability 3.0, 3.5 and 3.7. | Multi-context* |
| ecc_throughput | ECC throughput from L2 to DRAM. This is available for compute capability 3.5 and 3.7. | Multi-context* |

| Metric Name | Description | Scope |
|--------------------------|--|----------------|
| ecc_transactions | Number of ECC transactions between L2 and DRAM. This is available for compute capability 3.5 and 3.7. | 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_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_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 |

| Metric Name | Description | Scope |
|---------------------------------|---|----------------------------|
| 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_cache_replay_overhead | Average number of replays due to global memory cache misses for each instruction executed | Multi-context |
| global_replay_overhead | Average number of replays due to global memory cache misses | 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 [*] |
| 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_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 |

| Metric Name | Description | Scope |
|--------------------------|--|----------------------------|
| 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 |
| ipc_instance | Instructions executed per cycle for a single multiprocessor | 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 |
| l1_cache_global_hit_rate | Hit rate in L1 cache for global loads | Multi-context [*] |
| l1_cache_local_hit_rate | Hit rate in L1 cache for local loads and stores | Multi-context [*] |
| l1_shared_utilization | The utilization level of the L1/shared memory relative to peak utilization on a scale of 0 to 10. This is available for compute capability 3.0, 3.5 and 3.7. | 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_l1_read_hit_rate | Hit rate at L2 cache for all read requests from L1 cache. This is available for compute capability 3.0, 3.5 and 3.7. | Multi-context [*] |
| l2_l1_read_throughput | Memory read throughput seen at L2 cache for read requests from L1 cache. This is available for compute capability 3.0, 3.5 and 3.7. | Multi-context [*] |
| l2_l1_read_transactions | Memory read transactions seen at L2 cache for all read requests from L1 cache. This is available for compute capability 3.0, 3.5 and 3.7. | Multi-context [*] |
| l2_l1_write_throughput | Memory write throughput seen at L2 cache for write requests from L1 cache. This is available for compute capability 3.0, 3.5 and 3.7. | Multi-context [*] |
| l2_l1_write_transactions | Memory write transactions seen at L2 cache for all write requests from L1 cache. This is available for compute capability 3.0, 3.5 and 3.7. | 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 [*] |

| Metric Name | Description | Scope |
|--------------------------------------|--|----------------------------|
| l2_tex_read_transactions | Memory read transactions seen at L2 cache for read requests from the 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 3.0, 3.5 and 3.7. | Multi-context [*] |
| l2_tex_read_throughput | Memory read throughput seen at L2 cache for read 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 global, local and shared memory 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_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. This is available for compute capability 3.0, 3.5 and 3.7. | Multi-context [*] |
| local_replay_overhead | Average number of replays due to local memory accesses for each instruction executed | 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 [*] |
| nc_cache_global_hit_rate | Hit rate in non coherent cache for global loads | Multi-context [*] |
| nc_gld_efficiency | Ratio of requested non coherent global memory load throughput to required non coherent | Multi-context [*] |

| Metric Name | Description | Scope |
|---------------------------------------|---|----------------------------|
| | global memory load throughput expressed as percentage | |
| nc_gld_requested_throughput | Requested throughput for global memory loaded via non-coherent cache | Multi-context |
| nc_gld_throughput | Non coherent global memory load throughput | Multi-context [*] |
| nc_l2_read_throughput | Memory read throughput for non coherent global read requests seen at L2 cache | Multi-context [*] |
| nc_l2_read_transactions | Memory read transactions seen at L2 cache for non coherent global read requests | Multi-context [*] |
| 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_replay_overhead | Average number of replays due to shared memory conflicts for each instruction executed | 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 [*] |
| sm_efficiency | The percentage of time at least one warp is active on a multiprocessor averaged over all multiprocessors on the GPU | Multi-context [*] |
| sm_efficiency_instance | The percentage of time at least one warp is active on a specific multiprocessor | Multi-context [*] |
| stall_constant_memory_dependency | Percentage of stalls occurring because of immediate constant cache miss. This is available for compute capability 3.2, 3.5 and 3.7. | 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 | Multi-context |

| Metric Name | Description | Scope |
|---------------------------|---|----------------------------|
| | fully utilized, or because too many requests of a given type are outstanding. | |
| 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. This is available for compute capability 3.2, 3.5 and 3.7. | 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 |
| sysmem_read_throughput | System memory read throughput. This is available for compute capability 3.0, 3.5 and 3.7. | Multi-context [*] |
| sysmem_read_transactions | System memory read transactions. This is available for compute capability 3.0, 3.5 and 3.7. | 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 3.0, 3.5 and 3.7. | 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 3.0, 3.5 and 3.7. | Multi-context [*] |
| sysmem_write_throughput | System memory write throughput. This is available for compute capability 3.0, 3.5 and 3.7. | Multi-context [*] |
| sysmem_write_transactions | System memory write transactions. This is available for compute capability 3.0, 3.5 and 3.7. | 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 3.0, 3.5 and 3.7. | Multi-context |
| tex_cache_hit_rate | Texture cache hit rate | Multi-context [*] |
| tex_cache_throughput | Texture cache throughput | Multi-context [*] |

| Metric Name | Description | Scope |
|-----------------------------------|--|----------------|
| tex_cache_transactions | Texture cache read transactions | Multi-context* |
| tex_fu_utilization | The utilization level of the multiprocessor function units that execute texture instructions on a scale of 0 to 10 | Multi-context |
| tex_utilization | The utilization level of the texture cache relative to the peak utilization on a scale of 0 to 10 | 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 expressed as percentage | 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 expressed as percentage | Multi-context |

* The "Multi-context" scope for this metric is supported only for devices with compute capability 3.0, 3.5, and 3.7.

1.6.1.2. 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 2 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 |
| 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 |

| Metric Name | Description | Scope |
|---------------------------------|---|----------------|
| 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 (add, multiply and multiply-accumulate). Each multiply-accumulate operation contributes 2 to the count. This is available for compute capability 5.3. | Multi-context* |

| Metric Name | Description | Scope |
|--------------------------|---|----------------------------|
| 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 [*] |
| gld_transactions | Number of global memory load transactions | Multi-context [*] |

| Metric Name | Description | Scope |
|---------------------------------|--|----------------------------|
| 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 |
| inst_executed_global_stores | Warp level instructions for global stores | Multi-context |
| inst_executed_local_loads | Warp level instructions for local loads | Multi-context |

| Metric Name | Description | Scope |
|----------------------------------|---|----------------|
| 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 |
| l2_atomic_transactions | Memory read transactions seen at L2 cache for atomic and reduction requests | Multi-context* |

| Metric Name | Description | Scope |
|-------------------------------|---|----------------|
| 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* |
| l2_write_throughput | Memory write throughput seen at L2 cache for all write requests | Multi-context* |

| Metric Name | Description | Scope |
|--------------------------------------|--|----------------------------|
| 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 [*] |
| shared_load_transactions_per_request | Average number of shared memory load transactions performed for each shared memory load | Multi-context [*] |

| Metric Name | Description | Scope |
|---------------------------------------|---|----------------|
| 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 |
| surface_atomic_requests | Total number of surface atomic(Atom and Atom CAS) requests from Multiprocessor | Multi-context |

| Metric Name | Description | Scope |
|----------------------------|--|----------------|
| 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* |
| texture_load_requests | Total number of texture Load requests from Multiprocessor | Multi-context |

| Metric Name | Description | Scope |
|-----------------------------------|--|---------------|
| 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.**

1.6.1.3. Metrics for Capability 6.x

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

Table 3 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 |
| 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 |

| Metric Name | Description | Scope |
|--------------------------|---|---------------|
| 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 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 |

| Metric Name | Description | Scope |
|------------------------------|---|---------------|
| 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 |
| gst_throughput | Global memory store throughput | Multi-context |
| gst_transactions | Number of global memory store transactions | Multi-context |

| Metric Name | Description | Scope |
|----------------------------------|--|---------------|
| 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 |
| inst_integer | Number of integer instructions executed by non-predicated threads | Multi-context |

| Metric Name | Description | Scope |
|---------------------------------|---|---------------|
| 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 |
| l2_surface_store_bytes | Bytes written to L2 from Unified Cache for surface stores. This does not include surface atomics. | Multi-context |

| Metric Name | Description | Scope |
|-------------------------------------|--|---------------|
| 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 |
| local_store_requests | Total number of local store requests from Multiprocessor | Multi-context |
| local_store_throughput | Local memory store throughput | Multi-context |

| Metric Name | Description | Scope |
|--------------------------------------|---|---------------|
| 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 |
| nvlink_user_nratom_data_transmitted | Total non-reduction atomic user data bytes transmitted through NVLinks. This is available for compute capability 6.0. | Device |

| Metric Name | Description | Scope |
|---------------------------------------|--|---------------|
| 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 |
| 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 | Multi-context |

| Metric Name | Description | Scope |
|----------------------------|--|---------------|
| | the required resources not being available or fully utilized, or because too many requests of a given type are outstanding | |
| 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 |
| 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 |

| Metric Name | Description | Scope |
|-----------------------------------|--|---------------|
| 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 |

1.6.1.4. Metrics for Capability 7.0

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

Table 4 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 |
| 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 |

| Metric Name | Description | Scope |
|---------------------------------|---|---------------|
| 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 |
| 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 | Multi-context |

| Metric Name | Description | Scope |
|------------------------------|---|---------------|
| | multiply-accumulate operation contributes 2 to 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 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 |
| 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. 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 |
| 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 |

| Metric Name | Description | Scope |
|---------------------------------|---|---------------|
| 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 |
| 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 |

| Metric Name | Description | Scope |
|--------------------------------------|--|---------------|
| 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 |
| 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 |

| Metric Name | Description | Scope |
|--------------------------------------|--|---------------|
| 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 |
| 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 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 |
| 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 | 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 |
| 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 |

| Metric Name | Description | Scope |
|-----------------------------------|--|---------------|
| 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 |

1.7. CUPTI Profiling API

Starting with CUDA 10.0, a new set of metric APIs are added for the devices with compute capability 7.0 and higher. These APIs provide low and deterministic profiling overhead on the target system. These APIs are supported on all CUDA supported platforms except Mac and Android. These APIs are not supported under MPS (Multi-Process Service).

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`

1.7.1. 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.

1.7.2. 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.

1.7.2.1. 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.

1.7.2.2. 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.

1.7.3. 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.

1.8. Perfworks Metrics API

Introduction:

The Perfworks Metrics 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 Metrics 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.

1.8.1. 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.

| | |
|---|---|
| <code>.peak_burst</code> | the peak burst rate [*] |
| <code>.peak_sustained</code> | the peak sustained rate |
| <code>.per_cycle_active</code> | the number of operations per unit active cycle |
| <code>.per_cycle_elapsed</code> | the number of operations per unit elapsed cycle |
| <code>.per_cycle_region</code> | the number of operations per user-specified "range" cycle |
| <code>.per_cycle_frame</code> | the number of operations per user-specified "frame" cycle |
| <code>.per_second</code> | the number of operations per second |
| <code>.pct_of_peak_burst_active</code> | % of peak burst rate achieved during unit active cycles [*] |
| <code>.pct_of_peak_burst_elapsed</code> | % of peak burst rate achieved during unit elapsed cycles [*] |
| <code>.pct_of_peak_burst_region</code> | % of peak burst rate achieved over a user-specified "range" time [*] |
| <code>.pct_of_peak_burst_frame</code> | % of peak burst rate achieved over a user-specified "frame" time [*] |
| <code>.pct_of_peak_sustained_active</code> | % of peak sustained rate achieved during unit active cycles |
| <code>.pct_of_peak_sustained_elapsed</code> | % of peak sustained rate achieved during unit elapsed cycles |

| | |
|--|--|
| <code>.pct_of_peak_sustained_region</code> | % of peak sustained rate achieved over a user-specified "range" time |
| <code>.pct_of_peak_sustained_frame</code> | % of peak sustained rate achieved over a user-specified "frame" time |

- Counters : may be either a raw counter from the GPU, or a calculated counter value. Every counter has 4 sub-metrics under it:

| | |
|-------------------|--|
| <code>.sum</code> | The sum of counter values across all unit instances. |
| <code>.avg</code> | The average counter value across all unit instances. |
| <code>.min</code> | The minimum counter value across all unit instances. |
| <code>.max</code> | The maximum counter value across all unit instances. |

- Ratios : . Every counter has 2 sub-metrics under it:

| | |
|---------------------|--------------------------------------|
| <code>.pct</code> | The value expressed as a percentage. |
| <code>.ratio</code> | The value expressed as a ratio. |

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

| | |
|---|---|
| <code>.pct_of_peak_burst_active</code> | % of peak burst rate achieved during unit active cycles [*] |
| <code>.pct_of_peak_burst_elapsed</code> | % of peak burst rate achieved during unit elapsed cycles [*] |
| <code>.pct_of_peak_burst_region</code> | % of peak burst rate achieved over a user-specified "range" time [*] |
| <code>.pct_of_peak_burst_frame</code> | % of peak burst rate achieved over a user-specified "frame" time [*] |
| <code>.pct_of_peak_sustained_active</code> | % of peak sustained rate achieved during unit active cycles |
| <code>.pct_of_peak_sustained_elapsed</code> | % 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. The sum,avg,min,max sub-metrics for counters are also called "rollups".

*Burst metrics are supported with MetricsContext APIs only.

Metrics Examples

```
## non-metric names -- *not* directly evaluable
sm_inst_executed          # counter
smisp_average_warp_latency # ratio
sm_throughput             # throughput

## a counter's four 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_region
l1tex_data_bank_conflicts_pipe_lsu.sum.per_cycle_frame
l1tex_data_bank_conflicts_pipe_lsu.sum.per_second
l1tex_data_bank_conflicts_pipe_lsu.sum.pct_of_peak_burst_active
l1tex_data_bank_conflicts_pipe_lsu.sum.pct_of_peak_burst_elapsed
l1tex_data_bank_conflicts_pipe_lsu.sum.pct_of_peak_burst_region
l1tex_data_bank_conflicts_pipe_lsu.sum.pct_of_peak_burst_frame
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 or 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.

1.8.2. 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.

1.8.3. 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 5 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 | $\frac{(l1tex__t_sectors_pipe_lsu_mem_global_op_atom.sum + l1tex__t_sectors_pipe_lsu_mem_global_op_red.sum)}{(l1tex__t_requests_pipe_lsu_mem_global_op_atom.sum + l1tex__t_requests_pipe_lsu_mem_global_op_red.sum)}$ |
| 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 |
| 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 |

| CUPTI Metric | Perfworks Metric or Formula |
|-------------------------------|---|
| flop_count_hp | $\text{smsp_sass_thread_inst_executed_op_hadd_pred_on.sum} + \text{smsp_sass_thread_inst_executed_op_hmul_pred_on.sum} + \text{smsp_sass_thread_inst_executed_op_hfma_pred_on.sum} * 2$ |
| flop_count_hp_add | $\text{smsp_sass_thread_inst_executed_op_hadd_pred_on.sum}$ |
| flop_count_hp_fma | $\text{smsp_sass_thread_inst_executed_op_hfma_pred_on.sum}$ |
| flop_count_hp_mul | $\text{smsp_sass_thread_inst_executed_op_hmul_pred_on.sum}$ |
| flop_count_sp | $\text{smsp_sass_thread_inst_executed_op_fadd_pred_on.sum} + \text{smsp_sass_thread_inst_executed_op_fmul_pred_on.sum} + \text{smsp_sass_thread_inst_executed_op_ffma_pred_on.sum} * 2$ |
| flop_count_sp_add | $\text{smsp_sass_thread_inst_executed_op_fadd_pred_on.sum}$ |
| flop_count_sp_fma | $\text{smsp_sass_thread_inst_executed_op_ffma_pred_on.sum}$ |
| flop_count_sp_mul | $\text{smsp_sass_thread_inst_executed_op_fmul_pred_on.sum}$ |
| flop_count_sp_special | |
| flop_dp_efficiency | $\text{smsp_sass_thread_inst_executed_ops_dadd_dmul_dfma_pred_on.avg.pct_of_peak_sustained_active}$ |
| flop_hp_efficiency | $\text{smsp_sass_thread_inst_executed_ops_hadd_hmul_hfma_pred_on.avg.pct_of_peak_sustained_active}$ |
| flop_sp_efficiency | $\text{smsp_sass_thread_inst_executed_ops_fadd_fmul_ffma_pred_on.avg.pct_of_peak_sustained_active}$ |
| gld_efficiency | $\text{smsp_sass_average_data_bytes_per_sector_mem_global_op_ld.pct}$ |
| gld_requested_throughput | |
| gld_throughput | $\text{l1tex_t_bytes_pipe_lsu_mem_global_op_ld.sum.per_second}$ |
| gld_transactions | $\text{l1tex_t_sectors_pipe_lsu_mem_global_op_ld.sum}$ |
| gld_transactions_per_request | $\text{l1tex_average_t_sectors_per_request_pipe_lsu_mem_global_op_ld.ratio}$ |
| global_atomic_requests | $\text{l1tex_t_requests_pipe_lsu_mem_global_op_atom.sum}$ |
| global_hit_rate | $\text{l1tex_t_sectors_pipe_lsu_mem_global_op_}\{op\}\text{.lookup_hit.sum} / \text{l1tex_t_sectors_pipe_lsu_mem_global_op_}\{op\}\text{.sum}$ |
| global_load_requests | $\text{l1tex_t_requests_pipe_lsu_mem_global_op_ld.sum}$ |
| global_reduction_requests | $\text{l1tex_t_requests_pipe_lsu_mem_global_op_red.sum}$ |
| global_store_requests | $\text{l1tex_t_requests_pipe_lsu_mem_global_op_st.sum}$ |
| gst_efficiency | $\text{smsp_sass_average_data_bytes_per_sector_mem_global_op_st.pct}$ |
| gst_requested_throughput | |
| gst_throughput | $\text{l1tex_t_bytes_pipe_lsu_mem_global_op_st.sum.per_second}$ |
| gst_transactions | $\text{l1tex_t_bytes_pipe_lsu_mem_global_op_st.sum}$ |
| gst_transactions_per_request | $\text{l1tex_average_t_sectors_per_request_pipe_lsu_mem_global_op_st.ratio}$ |
| half_precision_fu_utilization | $\text{smsp_inst_executed_pipe_fp16.avg.pct_of_peak_sustained_active}$ |
| inst_bit_convert | $\text{smsp_sass_thread_inst_executed_op_conversion_pred_on.sum}$ |
| inst_compute_ld_st | $\text{smsp_sass_thread_inst_executed_op_memory_pred_on.sum}$ |
| inst_control | $\text{smsp_sass_thread_inst_executed_op_control_pred_on.sum}$ |
| inst_executed | $\text{smsp_inst_executed.sum}$ |

| CUPTI Metric | Perfworks Metric or Formula |
|----------------------------------|--|
| 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 |
| issued_ipc | smsp__inst_issued.avg.per_cycle_active |
| l1_sm_lg_utilization | l1tex__lsu_writeback_active.avg.pct_of_peak_sustained_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 |

| CUPTI Metric | Perfworks Metric or Formula |
|--------------------------------------|--|
| 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 | $smisp_inst_executed_pipe_lsu.avg.pct_of_peak_sustained_active$ |
| 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 | $l1tex_average_t_sectors_per_request_pipe_lsu_mem_local_op_st.ratio$ |
| nvlink_data_receive_efficiency | |
| nvlink_data_transmission_efficiency | |
| nvlink_overhead_data_received | |
| nvlink_overhead_data_transmitted | |
| nvlink_receive_throughput | |

| CUPTI Metric | Perfworks Metric or Formula |
|---------------------------------------|--|
| 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 |
| sm_tex_utilization | l1tex__texin_sm2tex_req_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 |
| 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 |

| CUPTI Metric | Perfworks Metric or Formula |
|-----------------------------------|--|
| stall_other | $\text{smsp_warp_issue_stalled_misc_per_warp_active.pct} + \text{smsp_warp_issue_stalled_dispatch_stall_per_warp_active.pct}$ |
| stall_pipe_busy | $\text{smsp_warp_issue_stalled_mio_throttle_per_warp_active.pct} + \text{smsp_warp_issue_stalled_math_pipe_throttle_per_warp_active.pct}$ |
| stall_sleeping | $\text{smsp_warp_issue_stalled_sleeping_per_warp_active.pct}$ |
| stall_sync | $\text{smsp_warp_issue_stalled_membar_per_warp_active.pct} + \text{smsp_warp_issue_stalled_barrier_per_warp_active.pct}$ |
| stall_texture | $\text{smsp_warp_issue_stalled_tex_throttle_per_warp_active.pct}$ |
| surface_atomic_requests | $\text{l1tex_t_requests_pipe_tex_mem_surface_op_atom.sum}$ |
| surface_load_requests | $\text{l1tex_t_requests_pipe_tex_mem_surface_op_ld.sum}$ |
| surface_reduction_requests | $\text{l1tex_t_requests_pipe_tex_mem_surface_op_red.sum}$ |
| surface_store_requests | $\text{l1tex_t_requests_pipe_tex_mem_surface_op_st.sum}$ |
| sysmem_read_bytes | $\text{lts_t_sectors_aperture_sysmem_op_read} * 32$ |
| sysmem_read_throughput | $\text{lts_t_sectors_aperture_sysmem_op_read.sum.per_second}$ |
| sysmem_read_transactions | $\text{lts_t_sectors_aperture_sysmem_op_read.sum}$ |
| sysmem_read_utilization | |
| sysmem_utilization | |
| sysmem_write_bytes | $\text{lts_t_sectors_aperture_sysmem_op_write} * 32$ |
| sysmem_write_throughput | $\text{lts_t_sectors_aperture_sysmem_op_write.sum.per_second}$ |
| sysmem_write_transactions | $\text{lts_t_sectors_aperture_sysmem_op_write.sum}$ |
| sysmem_write_utilization | |
| tensor_precision_fu_utilization | $\text{sm_pipe_tensor_cycles_active.avg.pct_of_peak_sustained_active}$ |
| tex_cache_hit_rate | $\text{l1tex_t_sector_hit_rate.pct}$ |
| tex_cache_throughput | |
| tex_cache_transactions | $\text{l1tex_lsu_writeback_active.avg.pct_of_peak_sustained_active} + \text{l1tex_tex_writeback_active.avg.pct_of_peak_sustained_active}$ |
| tex_fu_utilization | $\text{smsp_inst_executed_pipe_tex.avg.pct_of_peak_sustained_active}$ |
| tex_sm_tex_utilization | $\text{l1tex_f_tex2sm_cycles_active.avg.pct_of_peak_sustained_elapsed}$ |
| tex_sm_utilization | $\text{sm_mio2rf_writeback_active.avg.pct_of_peak_sustained_elapsed}$ |
| tex_utilization | |
| texture_load_requests | $\text{l1tex_t_requests_pipe_tex_mem_texture.sum}$ |
| warp_execution_efficiency | $\text{smsp_thread_inst_executed_per_inst_executed.ratio}$ |
| warp_nonpred_execution_efficiency | $\text{smsp_thread_inst_executed_per_inst_executed.pct}$ |

1.8.4. 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 6 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 |
| 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 |

| CUPTI Event | Perfworks Metric or Formula |
|---|--|
| 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 | dsmsp__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 | |
| prof_trigger_07 | |
| inst_issued0 | smmsp__issue_inst0.sum |
| sm_cta_launched | sm__ctas_launched.sum |

| CUPTI Event | Perfworks Metric or Formula |
|------------------------------|--|
| 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 |

1.9. 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 a new set of [Profiling API](#) in the header `cupti_profiler_target.h` and [Perfworks metrics 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

| CUPTI APIs | Feature Description | Supported GPU architectures | Notes |
|------------|--|--|--|
| Event | Collect kernel performance counters for a kernel execution | Kepler, Maxwell, Pascal, Volta | Not supported on Turing and later GPU architectures, i.e. devices with compute capability 7.5 and higher |
| Metric | Collect kernel performance metrics for a kernel execution | Kepler, Maxwell, Pascal, Volta | Not supported on Turing and later GPU architectures, i.e. devices with compute capability 7.5 and higher |
| Profiling | Collect performance metrics for a range of execution | Volta and later GPU architectures, i.e. devices with compute capability 7.0 and higher | Not supported on Kepler, Maxwell and Pascal GPUs |

Note that both the event and metrics 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 Mac and 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.

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.

1.10. 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 except Mac and Tegra platforms (QNX, Linux aarch64, Android). 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](#).

1.10.1. PC Sampling Configuration Attributes

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

Table 7 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 | 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 |

| Configuration Attribute | Description | Default Value | Comparison of PC Sampling APIs with CUPTI PC Sampling Activity APIs | Guideline to Tune Configuration Option |
|---------------------------|--|--|---|--|
| | | | "samplingPeriod2" field in CUpti_ActivityPCSamplingConfig. | 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 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 | |

1.10.2. 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 8 Stall Reasons Mapping Table from PC Sampling Activity APIs to PC Sampling APIs

| PC Sampling Activity API Stall Reasons (common prefix: CUPTI_ACTIVITY_PC_SAMPLING) | PC Sampling API Stall Reasons (common prefix: smp__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 |

| PC Sampling Activity API Stall Reasons (common prefix: CUPTI_ACTIVITY_PC_SAMPLING) | PC Sampling API Stall Reasons (common prefix: smp__pcsamp_warps_issue_stalled_) |
|---|--|
| | drain_not_issued lg_throttle lg_throttle_not_issued |
| NOT_SELECTED | not_selected not_selected_not_issued |
| OTHER | misc misc_not_issued dispatch_stall dispatch_stall_not_issued |
| SLEEPING | sleeping sleeping_not_issued |

1.10.3. Data Structure Mapping Table

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

Table 9 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 mapping |
| CUpti_ActivityPCSampling3 | CUpti_PCSamplingPCData |
| CUpti_ActivityPCSamplingRecordInfo | CUpti_PCSamplingData |

1.10.4. PC Sampling 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
}
```


1.10.5. 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 configuration overhead may not be compensated entirely for a small application workload and profiling overhead can be more than expected for such a case.

1.11. 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. events and metrics APIs.

1.11.1. 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.

1.11.1.1. Execution overhead

Factors affecting the execution overhead under tracing are:

- ▶ Enabling serial kernel 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 incurs less profiling overhead compared to the concurrent kernel mode.
- ▶ Enabling concurrent kernel 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. Since every kernel in the CUDA module is instrumented, the overhead is proportional to the number of kernels in the module. This is a static overhead which happens at the time of loading the CUDA module. The overhead is attributed as `CUPTI_ACTIVITY_OVERHEAD_CUPTI_INSTRUMENTATION` in the activity record

CUpti_ActivityOverhead. Starting with the CUDA 11.2, kernels are not instrumented at the module load time, instead a single instrumentation code is used for each kernel thus avoiding the static overhead at the CUDA module load time.

- ▶ 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.
- ▶ Collection of the kernel latency timestamps i.e. queued and submitted timestamps is a high overhead activity. These are not collected by default. One can enable the collection of these timestamps using the API `cuptiActivityEnableLatencyTimestamps()`.

1.11.1.2. Memory overhead

CUPTI allocates device memory for storing the tracing information:

- ▶ **Static device memory allocation:** CUPTI allocates 11 MB of GPU memory for each CUDA context by default. Out of which 9 MB is used for storing the concurrent kernel tracing information and this buffer is sufficient for tracing about 300K kernels. And 2 MB is used for storing the CUDA memcopy and serial kernel tracing information and this buffer is sufficient for tracing about 75K memcopies and kernels. Activity attributes `CUPTI_ACTIVITY_ATTR_DEVICE_BUFFER_SIZE` and `CUPTI_ACTIVITY_ATTR_PROFILING_SEMAPHORE_POOL_SIZE` can be used to configure the size of the buffers for concurrent kernel and memcopy/serial kernel tracing respectively.
- ▶ **Dynamic device memory allocation:** Once device buffer to store the tracing information is exhausted, CUPTI allocates another device buffer of the same size. Note that memory footprint will not scale with the kernel or memcopy count because CUPTI reuses the buffer after processing all the records in the buffer. For applications with a high density of kernels or memcopies CUPTI may allocate more device buffers.

All of the CUPTI allocated GPU memory associated with a context is freed when the context is destroyed.

1.11.2. 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 category.
 - ▶ 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.

1.12. Multi Instance GPU

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 its 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**.

1.13. Samples

The CUPTI installation includes several samples that demonstrate the use of the CUPTI APIs. The samples are:

activity_trace_async

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

callback_timestamp

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

cupti_nvtx

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

cupti_finalize

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

cupti_query

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

event_sampling

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

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.

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.

pc_sampling

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

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.

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.

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.

userrange_profiling

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

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.

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.

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.

Chapter 2.

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.

2.1. OptiX

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

- ▶ **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 but these cannot be profiled i.e we cannot collect events, metrics and PC sampling data.

- ▶ **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 and profiled by CUPTI.

Chapter 3.

LIMITATIONS

The following are known issues with the current release.

- ▶ 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](#).



Starting with CUDA 10.2, CUPTI allows tracing features for non-root and non-admin users on desktop platforms. But events and metrics profiling is still restricted for non-root and non-admin users.

- ▶ 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`.
- ▶ CUDA runtime and driver API callbacks for kernel launch are not issued when the stream is in the capture mode.
- ▶ PCIE and NVLINK records are not captured when CUPTI is initialized lazily after the CUDA initialization.

- ▶ 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.
 - ▶ Starting with CUDA Toolkit 9.0, CUPTI doesn't support CUDA Dynamic Parallelism (CDP) kernel launch tracing for devices with compute capability 7.0 and higher. 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.
 - ▶ CUPTI doesn't report CUDA API calls for device-launched kernels.
 - ▶ CUPTI 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.
- ▶ 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.
- ▶ Tracing and profiling features are not supported on virtual GPUs (vGPU).
- ▶ PC Sampling is not supported on Tegra platforms.

Profiling

The following are common known issues for both the event and metrics 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 is not supported for CUDA kernel nodes launched by a CUDA Graph.
- ▶ Profiling is not supported on virtual GPUs (vGPU).
- ▶ 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>

Events and Metrics APIs

The following are known issues related to Events and Metrics APIs:

- ▶ 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 metrics 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.
- ▶ These APIs are not supported on ARM server platform (ARM64 SBSA).

Profiling API

The following are known issues related to the Profiling API:

- ▶ In the current release, 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 (`nv1rx__*` and `nv1tx__*`) are not supported due to a potential application hang during data collection.
- ▶ Profiling is not supported under MPS (Multi-Process Service).
- ▶ `NVPW_MetricsEvaluator` APIs are not available in the static build of host library.

Chapter 4.

CHANGELOG

CUPTI changes in CUDA 11.3

CUPTI contains below change as part of the CUDA Toolkit 11.3 release.

- ▶ 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.
- ▶ Overhead reduction for tracing of CUDA memcpyes.
- ▶ 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`.

CUPTI changes in CUDA 11.2

CUPTI contains below change as part of the CUDA Toolkit 11.2 release.

- ▶ 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 introduced.
- ▶ 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`.
- ▶ Execution overheads introduced by CUPTI in the tracing path is reduced.
- ▶ 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 (`nvlink_*` 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.

CUPTI changes in CUDA 11.1

CUPTI contains below change as part of the CUDA Toolkit 11.1 release.

- ▶ 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.

CUPTI changes in CUDA 11.0

CUPTI contains below change as part of the CUDA Toolkit 11.0 release.

- ▶ 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.
- ▶ 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](#).
- ▶ Added support to trace Optix applications. Refer the [Optix Profiling](#) section.

- ▶ PC sampling overhead 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.
- ▶ 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 maximum 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 calculation. Value `FUNC_SHMEM_LIMIT_OPTIN` in the enum `cudaOccFuncShmemConfig` is the corresponding option in the CUDA occupancy calculator.

CUPTI changes in CUDA 10.2

CUPTI contains below changes as part of the CUDA Toolkit 10.2 release.

- ▶ 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.

CUPTI changes in CUDA 10.1 Update 2

CUPTI contains below changes as part of the CUDA Toolkit 10.1 Update 2 release.

- ▶ 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](#).

CUPTI changes in CUDA 10.1 Update 1

CUPTI contains below change as part of the CUDA Toolkit 10.1 Update 1 release.

- ▶ 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`

CUPTI changes in CUDA 10.1

CUPTI contains below changes as part of the CUDA Toolkit 10.1 release.

- ▶ 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.

CUPTI changes in CUDA 10.0

CUPTI contains below changes as part of the CUDA Toolkit 10.0 release.

- ▶ 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.

CUPTI changes in CUDA 9.2

CUPTI contains below changes as part of the CUDA Toolkit 9.2 release.

- ▶ 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.

CUPTI changes in CUDA 9.1

List of changes done as part of the CUDA Toolkit 9.1 release.

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

CUPTI changes in CUDA 9.0

List of changes done as part of the CUDA Toolkit 9.0 release.

- ▶ 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.

CUPTI changes in CUDA 8.0

List of changes done as part of the CUDA Toolkit 8.0 release.

- ▶ 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.

CUPTI changes in CUDA 7.5

List of changes done as part of the CUDA Toolkit 7.5 release.

- ▶ 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.

CUPTI changes in CUDA 7.0

List of changes done as part of the CUDA Toolkit 7.0 release.

- ▶ 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 capability 5.2.

CUPTI changes in CUDA 6.5

List of changes done as part of the CUDA Toolkit 6.5 release.

- ▶ 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.

CUPTI changes in CUDA 6.0

List of changes done as part of the CUDA Toolkit 6.0 release.

- ▶ 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.

CUPTI changes in CUDA 5.5

List of changes done as part of CUDA Toolkit 5.5 release.

- ▶ 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.

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

© 2007-2021 NVIDIA Corporation. All rights reserved.

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