CUPTI

DA-05679-001 _v10.0 | December 2018

User's Guide

TABLE OF CONTENTS

LIST OF TABLES

OVERVIEW

The *CUDA Profiling Tools Interface* (CUPTI) enables the creation of profiling and tracing tools that target CUDA applications. CUPTI provides four APIs: *the Activity API*, the *Callback API*, the *Event API*, and the *Metric 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 the 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 Profiling API section for documentation of the 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 you 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.
- The 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.

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 9.0 can update to a more recently released CUDA driver. However, new versions of CUPTI are not backwards compatible with older versions of the CUDA driver. For example, a developer using a profiling tool based on CUPTI 9.0 must have a version of the CUDA driver released with CUDA Toolkit 9.0 (or later) installed as well. CUPTI calls will fail with `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.

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_ActivityMemcpy`). 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. The CUPTI client is responsible for providing empty activity buffers as necessary to ensure that no records are dropped.

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

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

The activity buffer API uses callbacks to request and return buffers of activity records. To use the asynchronous buffering API, you must first register two callbacks using cuptiActivityRegisterCallbacks. One of these callbacks will be invoked whenever CUPTI needs an empty activity buffer. The other callback is used to deliver a buffer containing one or more activity records to the client. To minimize profiling overhead the client should return as quickly as possible from these callbacks. The function cuptiActivityFlushAll can be used to force CUPTI to deliver any activity buffers that contain completed activity records. 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.

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.

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.

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

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_ActivityNVLink2`, 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 6 NVLinks, hence one logical link can comprise of 1 to 6 physical NVLinks. The field `physicalNvLinkCount` gives the number of physical links in this logical link. The fields `portDev0` and `portDev1` give information about the slot in which physical NVLinks are connected for a logical link. This port is the same as the instance of NVLink metrics profiled from a device. Therefore, port and instance information should be used to correlate the per-instance metric values with the physical NVLinks, and in turn to the topology. The field `flag` gives the properties of a logical link, whether the link has access to system memory or peer device memory, and has capabilities to do system memory or peer memmory atomics. The field `bandwidth` gives the bandwidth of the logical link in kilobytes/sec.

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

This feature is available on devices with compute capability 6.0 and 7.0.

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

On Linux x86_64, CUPTI supports collecting information for OpenACC applications using the OpenACC tools interface implementation of the PGI runtime. In addition to being available only on 64bit Linux platforms, this feature also requires PGI runtime version 15.7 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. 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`.

1.4. 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 = ...;
...
cuptiSubscribe(&subscriber,
               (CUpti_CallbackFunc)my_callback , my_data);
cuptiEnableDomain(1, subscriber,
                  CUPTI_CB_DOMAIN_NVTX);
```

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

The following code shows a typical callback function.

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

  if ((domain == CUPTI_CB_DOMAIN_NVTX) &&
      (cbid == 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`.

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.

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, due to device limitations. 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.

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.

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. The events for these metrics can be collected at device level using `CUPTI_EVENT_COLLECTION_MODE_CONTINUOUS`. When these metrics are collected for a kernel using `CUPTI_EVENT_COLLECTION_MODE_KERNEL`, they exhibit the behavior of single-context. **Note that NVLink metrics collected for kernel mode exhibit the behavior of "Single-context."**

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

Metric Name	Description	Scope
cf_fu_utilization	The utilization level of the multiprocessor function units that execute control-flow instructions on a scale of 0 to 10	Multi-context
cf_issued	Number of issued control-flow instructions	Multi-context
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*
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

Metric Name	Description	Scope
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
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

Metric Name	Description	Scope
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
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*

Metric Name	Description	Scope
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*
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*

Metric Name	Description	Scope
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 global memory load throughput expressed as percentage	Multi-context*
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*

Metric Name	Description	Scope
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 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. 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	Multi-context

Metric Name	Description	Scope
	10. This is available for compute capability 3.0, 3.5 and 3.7.	
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*
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
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	Multi-context

Metric Name	Description	Scope
	multiply-accumulate operation contributes 2 to the count.	
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*
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

Metric Name	Description	Scope
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*
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	Multi-context*

Metric Name	Description	Scope
	scale of 0 to 10. This is available for compute capability 5.3.	
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.) 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

Metric Name	Description	Scope
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*
l2_tex_hit_rate	Hit rate at L2 cache for all requests from texture cache	Multi-context*

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

Metric Name	Description	Scope
local_store_throughput	Local memory store throughput	Multi-context*
local_store_transactions	Number of local memory store transactions	Multi-context*
local_store_transactions_per_request	Average number of local memory store transactions performed for each local memory store	Multi-context*
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 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*

Metric Name	Description	Scope
sysmem_write_utilization	The write utilization level of the system memory relative to the peak utilization on a scale of 0 to 10. This is available for compute capability 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
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

Metric Name	Description	Scope
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
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

Metric Name	Description	Scope
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
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

Metric Name	Description	Scope
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 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

Metric Name	Description	Scope
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
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

Metric Name	Description	Scope
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 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	Multi-context

Metric Name	Description	Scope
	store and constant load instructions on a scale of 0 to 10	
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
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	Device

Metric Name	Description	Scope
	read requests and result of non-reduction atomic requests. This is available for compute capability 6.0.	
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
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

Metric Name	Description	Scope
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
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

Metric Name	Description	Scope
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
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
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

Metric Name	Description	Scope
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 multiply-accumulate operation contributes 2 to the count. The count does not include special operations.	Multi-context
flop_count_sp_add	Number of single-precision floating-point add operations executed by non-predicated threads.	Multi-context
flop_count_sp_fma	Number of single-precision floating-point multiply-accumulate operations executed by non-predicated threads. Each multiply-accumulate operation contributes 1 to the count.	Multi-context
flop_count_sp_mul	Number of single-precision floating-point multiply operations executed by non-predicated threads.	Multi-context
flop_count_sp_special	Number of single-precision floating-point special operations executed by non-predicated threads.	Multi-context
flop_dp_efficiency	Ratio of achieved to peak double-precision floating-point operations	Multi-context
flop_hp_efficiency	Ratio of achieved to peak half-precision floating-point operations	Multi-context
flop_sp_efficiency	Ratio of achieved to peak single-precision floating-point operations	Multi-context
gld_efficiency	Ratio of requested global memory load throughput to required global memory load throughput expressed as percentage.	Multi-context
gld_requested_throughput	Requested global memory load throughput	Multi-context
gld_throughput	Global memory load throughput	Multi-context
gld_transactions	Number of global memory load transactions	Multi-context

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 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
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

Metric Name	Description	Scope
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
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

Metric Name	Description	Scope
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
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

Metric Name	Description	Scope
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
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

Metric Name	Description	Scope
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
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	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	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
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

Metric Name	Description	Scope
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
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

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 new 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.

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 asynchronized 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_ReplayKernel;

 beginSessionParams.ctx = NULL;
 beginSessionParams.counterDataImageSize = counterDataImage.size();
 beginSessionParams.pCounterDataImage = &counterDataImage[0];
 beginSessionParams.counterDataScratchBufferSize =
 counterDataScratchBuffer.size();
 beginSessionParams.pCounterDataScratchBuffer = &counterDataScratchBuffer[0];
 beginSessionParams.collectionMethod = profilerCollectionMethod;
 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.
  kernelB <<<grid, tids >>>(...);              // KernelB Profiled and captured
 in an unique range.
  kernelC <<<grid, tids >>>(...);              // KernelC Profiled and captured
 in a unique range.
 }

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

 kernelD <<<grid, tids >>>(...)                  // KernelA 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_ReplayKernel;

       beginSessionParams.ctx = NULL;
       beginSessionParams.counterDataImageSize = counterDataImage.size();
       beginSessionParams.pCounterDataImage = &counterDataImage[0];
       beginSessionParams.counterDataScratchBufferSize =
counterDataScratchBuffer.size();
       beginSessionParams.pCounterDataScratchBuffer =
&counterDataScratchBuffer[0];
       beginSessionParams.collectionMethod = profilerCollectionMethod;
       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
profiler, 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>>>(...);              // Replayed but not
profiled

           cuptiProfilerEnableProfiling(&enableProfilingParams);

           kernelB<<<grid, tids>>>(...);              // KernelB Profiled and
captured in an unique range.
           kernelC<<<grid, tids>>>(...);              // KernelC 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_AutoRange;
       CUpti_ProfilerReplayMode profilerReplayMode = CUPTI_ReplayKernel;

       beginSessionParams.ctx = NULL;
       beginSessionParams.counterDataImageSize = counterDataImage.size();
       beginSessionParams.pCounterDataImage = &counterDataImage[0];
       beginSessionParams.counterDataScratchBufferSize =
counterDataScratchBuffer.size();
       beginSessionParams.pCounterDataScratchBuffer =
&counterDataScratchBuffer[0];
       beginSessionParams.collectionMethod = profilerCollectionMethod;
       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
profiler, 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>>>(...);              // Replayed but not
profiled

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

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

           cuptiProfilerPopRange(&popRangeParams);     // Kernel B and Kernel C
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 userrange_profiling sample.

The outline for enumerating metrics expanded by Perfworks:

- Call **`NVPW_MetricsContext_GetMetricNames_Begin`** to allow Perfworks to expand the metric names.
- Copy the string names from the output buffer.
- Call **`NVPW_MetricsContext_GetMetricNames_End`** to free the string names allocated by Perfworks by `_Begin`.

The outline for enumerating counters:

- Call **`NVPW_MetricsContext_GetCounterNames_Begin`** to allow Perfworks to expand the metric names.
- Copy the string names from the output buffer.
- Call **`NVPW_MetricsContext_GetCounterNames_End`** to free the string names allocated by Perfworks by `_Begin`.
- Generate metric names from the counter names, using the formulaic expansions described in Metric Entities.

Ratios and throughputs follow a similar pattern, with **`NVPW_MetricsContext_GetRatioNames_Begin`** and **`NVPW_MetricsContext_GetThroughputNames_Begin`**.

To programmatically determine the constituents of a Throughput metric:

- Call **`NVPW_MetricsContext_GetThroughputBreakdown_Begin`** + `_End` to retrieve the list of counters and sub-throughputs
- For each sub-throughput, recursively repeat the procedure of querying counters and sub-throughputs, until none remain.

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.
- For each metric, call **`NVPW_MetricsContext_GetMetricProperties_Begin`** to query its raw metric dependencies.
- For each raw metric dependency in `NVPW_MetricsContext_GetMetricProperties_Begin_Params::ppRawMetricDependenc`
    - 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.
- Query the number of ranges collected via **`NVPW_CounterData_GetNumRanges`**.
- For each range:
    - Call **`NVPW_Profiler_CounterData_GetRangeDescriptions`** to retrieve the range's description, originally set by **`cuptiProfilerPushRange`**.
    - Call **`NVPW_MetricsContext_SetCounterData`** to set the current range for evaluation on the **`NVPA_MetricsContext`**.

- Call **`NVPW_MetricsContext_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_MetricsContext_GetThroughputNames_Begin`**. 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.

Metrics Entities

The Metrics layer has 3 major types of entities:

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

`.peak_burst`	the peak burst rate
`.peak_sustained`	the peak sustained rate

.per_active_cycle	the number of operations per unit active cycle
.per_elapsed_cycle	the number of operations per unit elapsed cycle
.per_region_cycle	the number of operations per user-specified "range" cycle
.per_frame_cycle	the number of operations per user-specified "frame" cycle
.per_second	the number of operations per user-specified "frame" cycle
.pct_of_peak_burst_active	% of peak burst rate achieved during unit active cycles
.pct_of_peak_burst_elapsed	% of peak burst rate achieved during unit elapsed cycles
.pct_of_peak_burst_region	% of peak burst rate achieved over a user-specified "range" time
.pct_of_peak_burst_frame	% of peak burst rate achieved over a user-specified "frame" time
.pct_of_peak_sustained_active	% of peak sustained rate achieved during unit active cycles
.pct_of_peak_sustained_elapsed	% of peak sustained rate achieved during unit elapsed cycles
.pct_of_peak_sustained_region	% of peak sustained rate achieved over a user-specified "range" time
.pct_of_peak_sustained_frame	% of peak sustained rate achieved over a user-specified "frame" time

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

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

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

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

`.pct`	The value expressed as a percentage.
`.ratio`	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:

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

At the configuration step, you must specify metric names. Counters, ratios, and throughputs are not directly schedulable. The sum,avg,min,max sub-metrics for counters are also called "rollups".

Metrics Examples

```
## non-metric names -- *not* directly evaluable
sm__inst_executed                    # counter
smsp__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 of physical unit of the GPU
- subunit: The subunit within the unit where the counter was measured. Sometimes this is a pipeline mode instead.
- pipestage: The pipeline stage within the subunit where the counter was measured.
- quantity: What is being measured. Generally matches the "dimensional units".
- qualifiers: Any additional predicates or filters applied to the counter. Often, an unqualified counter can be broken down into several qualified sub-components.
- interface: Of the form `sender2receiver`, where `sender` is the source-unit and `receiver` is the destination-unit.

- rollup_metric: One of sum,avg,min,max.
- submetric: refer to section Metric Entities

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

Cycle Metrics

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

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

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

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

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 + l1tex__t_set_accesses_pipe_tex_mem_surface_op_atom.sum + l1tex__t_set_accesses_pipe_tex_mem_surface_op_red.sum
atomic_transactions_per_request	
branch_efficiency	
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
flop_count_dp_add	smsp__sass_thread_inst_executed_op_dadd_pred_on.sum
flop_count_dp_fma	smsp__sass_thread_inst_executed_op_dfma_pred_on.sum
flop_count_dp_mul	smsp__sass_thread_inst_executed_op_dmul_pred_on.sum
flop_count_hp	smsp__sass_thread_inst_executed_op_hadd_pred_on.sum + smsp__sass_thread_inst_executed_op_hmul_pred_on.sum + smsp__sass_thread_inst_executed_op_hfma_pred_on.sum
flop_count_hp_add	smsp__sass_thread_inst_executed_op_hadd_pred_on.sum
flop_count_hp_fma	smsp__sass_thread_inst_executed_op_hfma_pred_on.sum
flop_count_hp_mul	smsp__sass_thread_inst_executed_op_hmul_pred_on.sum
flop_count_sp	smsp__sass_thread_inst_executed_op_fadd_pred_on.sum + smsp__sass_thread_inst_executed_op_fmul_pred_on.sum + smsp__sass_thread_inst_executed_op_ffma_pred_on.sum
flop_count_sp_add	smsp__sass_thread_inst_executed_op_fadd_pred_on.sum

CUPTI Metric	Perfworks Metric or Formula
flop_count_sp_fma	smsp__sass_thread_inst_executed_op_ffma_pred_on.sum
flop_count_sp_mul	smsp__sass_thread_inst_executed_op_fmul_pred_on.sum
flop_count_sp_special	smsp__sass_thread_inst_executed_op_mufu_pred_on.sum
flop_dp_efficiency	
flop_hp_efficiency	
flop_sp_efficiency	
gld_efficiency	
gld_requested_throughput	
gld_throughput	l1tex__t_bytes_pipe_lsu_mem_global_op_ld.sum.per_second
gld_transactions	l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum
gld_transactions_per_request	
global_atomic_requests	l1tex__t_requests_pipe_lsu_mem_global_op_atom.sum
global_hit_rate	l1tex__t_sectors_pipe_lsu_mem_global_op_{op}_lookup_hit.sum / l1tex__t_sectors_pipe_lsu_mem_global_op_{op}.sum
global_load_requests	l1tex__t_requests_pipe_lsu_mem_global_op_ld.sum
global_reduction_requests	l1tex__t_requests_pipe_lsu_mem_global_op_red.sum
global_store_requests	l1tex__t_requests_pipe_lsu_mem_global_op_st.sum
gst_efficiency	
gst_requested_throughput	
gst_throughput	l1tex__t_bytes_pipe_lsu_mem_global_op_st.sum.per_second
gst_transactions	l1tex__t_bytes_pipe_lsu_mem_global_op_st.sum
gst_transactions_per_request	
half_precision_fu_utilization	smsp__inst_executed_pipe_fp16.sum
inst_bit_convert	smsp__sass_thread_inst_executed_op_conversion_pred_on.sum
inst_compute_ld_st	smsp__sass_thread_inst_executed_op_memory_pred_on.sum
inst_control	smsp__sass_thread_inst_executed_op_control_pred_on.sum
inst_executed	smsp__inst_executed.sum
inst_executed_global_atomics	
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

CUPTI Metric	Perfworks Metric or Formula
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.sum
l2_atomic_throughput	lts__t_sectors_srcunit_l1_op_atom.sum.per_second
l2_atomic_transactions	lts__t_sectors_srcunit_l1_op_atom.sum
l2_global_atomic_store_bytes	lts__t_bytes_equiv_l1sectormiss_pipe_lsu_mem_global_op_atom.sum
l2_global_load_bytes	lts__t_bytes_equiv_l1sectormiss_pipe_lsu_mem_global_op_ld.sum
l2_local_global_store_bytes	lts__t_bytes_equiv_l1sectormiss_pipe_lsu_mem_global_op_st.sum
l2_local_load_bytes	lts__t_bytes_equiv_l1sectormiss_pipe_lsu_mem_global_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_lsu_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

CUPTI Metric	Perfworks Metric or Formula
l2_tex_read_transactions	lts__t_sectors_srcunit_tex_op_read.sum
l2_tex_write_hit_rate	lts__t_sector_op_write_hit_rate.pct
l2_tex_write_throughput	lts__t_sectors_srcunit_tex_op_read.sum.per_second
l2_tex_write_transactions	lts__t_sectors_srcunit_tex_op_read.sum
l2_utilization	lts__t_sectors.avg.pct_of_peak_sustained_elapsed
l2_write_throughput	lts__t_sectors_op_write.sum.per_second
l2_write_transactions	lts__t_sectors_op_write.sum
ldst_executed	
ldst_fu_utilization	smsp__inst_executed_pipe_lsu.avg.pct_of_peak_sustained_active
ldst_issued	
local_hit_rate	
local_load_requests	l1tex__t_requests_pipe_lsu_mem_local_op_ld.sum
local_load_throughput	l1tex__t_bytes_pipe_lsu_mem_local_op_ld.sum.per_second
local_load_transactions	l1tex__t_sectors_pipe_lsu_mem_local_op_ld.sum
local_load_transactions_per_request	
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	
nvlink_data_receive_efficiency	
nvlink_data_transmission_efficiency	
nvlink_overhead_data_received	
nvlink_overhead_data_transmitted	
nvlink_receive_throughput	
nvlink_total_data_received	
nvlink_total_data_transmitted	
nvlink_total_nratom_data_transmitted	
nvlink_total_ratom_data_transmitted	
nvlink_total_response_data_received	
nvlink_total_write_data_transmitted	
nvlink_transmit_throughput	
nvlink_user_data_received	
nvlink_user_data_transmitted	
nvlink_user_nratom_data_transmitted	

CUPTI Metric	Perfworks Metric or Formula
nvlink_user_ratom_data_transmitted	
nvlink_user_response_data_received	
nvlink_user_write_data_transmitted	
pcie_total_data_received	
pcie_total_data_transmitted	
shared_efficiency	
shared_load_throughput	smsp__inst_executed_op_shared_ld.sum.per_second
shared_load_transactions	smsp__inst_executed_op_shared_ld.sum
shared_load_transactions_per_request	
shared_store_throughput	smsp__inst_executed_op_shared_st.sum.per_second
shared_store_transactions	smsp__inst_executed_op_shared_st.sum
shared_store_transactions_per_request	
shared_utilization	smsp__inst_executed_op_shared_ld.avg.pct_of_peak_sustained_elapsed + smsp__inst_executed_op_shared_st.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_miss_per_warp_active.pct
stall_inst_fetch	smsp__warp_issue_stalled_no_instruction_miss_per_warp_active.pct
stall_memory_dependency	smsp__warp_issue_stalled_long_scoreboard_miss_per_warp_active.pct
stall_memory_throttle	smsp__warp_issue_stalled_drain_miss_per_warp_active.pct
stall_not_selected	smsp__warp_issue_stalled_not_selected_miss_per_warp_active.pct
stall_other	smsp__warp_issue_stalled_misc_miss_per_warp_active.pct
stall_pipe_busy	smsp__warp_issue_stalled_misc_mio_throttle_per_warp_active.pct
stall_sleeping	smsp__warp_issue_stalled_misc_sleeping_per_warp_active.pct
stall_sync	smsp__warp_issue_stalled_misc_membar_per_warp_active.pct
stall_texture	smsp__warp_issue_stalled_misc_tex_throttle_per_warp_active.pct
surface_atomic_requests	l1tex__t_requests_pipe_tex_mem_surface_op_atom.sum
surface_load_requests	l1tex__t_requests_pipe_tex_mem_surface_op_ld.sum
surface_reduction_requests	l1tex__t_requests_pipe_tex_mem_surface_op_red.sum
surface_store_requests	l1tex__t_requests_pipe_tex_mem_surface_op_st.sum
sysmem_read_bytes	
sysmem_read_throughput	lts__t_sectors_aperture_sysmem_op_read.sum.per_second

CUPTI Metric	Perfworks Metric or Formula
sysmem_read_transactions	lts__t_sectors_aperture_sysmem_op_read.sum
sysmem_read_utilization	
sysmem_utilization	
sysmem_write_bytes	
sysmem_write_throughput	lts__t_sectors_aperture_sysmem_op_write.sum.per_second
sysmem_write_transactions	lts__t_sectors_aperture_sysmem_op_write.sum
sysmem_write_utilization	
tensor_precision_fu_utilization	sm__pipe_tensor_cycles_active.avg.pct_of_peak_sustained_active
tex_cache_hit_rate	l1tex__t_sector_hit_rate.pct
tex_cache_throughput	
tex_cache_transactions	l1tex__lsu_writeback_active.avg.pct_of_peak_sustained_active + l1tex__tex_writeback_active.avg.pct_of_peak_sustained_active
tex_fu_utilization	smsp__inst_executed_pipe_tex.avg.pct_of_peak_sustained_active
tex_sm_tex_utilization	l1tex__f_tex2sm_cycles_active.avg.pct_of_peak_sustained_elapsed
tex_sm_utilization	sm__mio2rf_writeback_active.avg.pct_of_peak_sustained_elapsed
tex_utilization	
texture_load_requests	l1tex__t_requests_pipe_tex_mem_texture.sum
warp_execution_efficiency	smsp__thread_inst_executed_per_inst_executed.ratio
warp_nonpred_execution_efficiency	smsp__thread_inst_executed_per_inst_executed.pct

1.9. 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_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.

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 new CUPTI profiling APIs to collect metrics in autorange mode.

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

Chapter 2. LIMITATIONS

The following are known issues with the current release.

- NVLink throughput metrics are not supported for devices with compute capability 7.5 or higher.
- The Continuous event collection mode `CUPTI_EVENT_COLLECTION_MODE_CONTINUOUS` is supported only on Tesla devices.
- Profiling results might be inconsistent when auto boost is enabled. The profiler tries to disable auto boost by default, but it might fail to do so in some conditions. If that occurs, profiling will continue and results will be inconsistent. The API **`cuptiGetAutoBoostState()`** can be used to query the auto boost state of the device. This API returns the 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.
- 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, you can set the event group attribute `CUPTI_EVENT_GROUP_ATTR_PROFILE_ALL_DOMAIN_INSTANCES` through API **`cuptiEventGroupSetAttribute()`**.
- Starting with CUDA Toolkit 9.0, CUPTI doesn't support CUDA Dynamic Parallelism (CDP) kernel launch tracing and source level metrics for devices with compute capability 7.0 and later.
- CUPTI doesn't support tracing and profiling on virtualized GPUs.

- 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. The 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.
- Because of the low resolution of the timer on Windows, the start and end timestamps can be the same for activities having short execution duration on Windows.
- Tracing and profiling (event and metric collection) is not supported for multidevice cooperative kernels; that is, kernels launched by using the API functions `cudaLaunchCooperativeKernelMultiDevice` or `cuLaunchCooperativeKernelMultiDevice`.
- The application which calls CUPTI APIs cannot be used with NVIDIA tools like `nvprof`, `NVIDIA Visual Profiler`, `NVIDIA Nsight Visual Studio Edition`, `cuda-gdb` and `cuda-memcheck`.
- Profiling is not supported for CUDA kernel nodes launched by a CUDA Graph.
- CUDA runtime and driver API callbacks for kernel launch are not issued when the stream is in the capture mode.
- Tracing of a CUDA Graph may change its performance characteristics.

Chapter 3. CHANGELOG

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 sets of metrics to collect total data bytes transmitted and recieved through PCIe are added. Those give an 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 capabilty 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.