



NSIGHT COMPUTE

v2023.3.0 | September 2023

Customization Guide



TABLE OF CONTENTS

Chapter 1. Introduction.....	1
Chapter 2. Metric Sections.....	2
2.1. Section Files.....	2
2.2. Section Definition.....	5
2.3. Metric Options and Filters.....	5
2.4. Missing Sections.....	7
2.5. Derived Metrics.....	7
2.5.1. Metric Types.....	7
2.5.1.1. Instanced Metrics.....	7
2.5.1.2. Single Value Metrics.....	8
2.5.2. Metric Value Types.....	8
2.5.3. Defining Derived Metrics.....	8
2.5.3.1. Addition.....	9
2.5.3.2. Subtraction.....	9
2.5.3.3. Multiplication.....	9
2.5.3.4. Divison.....	10
2.5.3.5. Functional Operators.....	10
Chapter 3. Rule System.....	11
3.1. Writing Rules.....	11
3.2. Integration.....	11
3.3. Rule System Architecture.....	12
3.4. NvRules API.....	13
3.5. Rule File API.....	13
3.6. Rule Examples.....	14
Chapter 4. Python Report Interface.....	15
4.1. Basic Usage.....	15
4.2. High-Level Interface.....	17
4.3. Metric attributes.....	17
4.4. NVTX Support.....	18
4.5. Sample Script.....	19
Chapter 5. Source Counters.....	20
Chapter 6. Report File Format.....	22
6.1. Version 7 Format.....	22

LIST OF TABLES

Table 1 Top-level report file format 22

Table 2 Per-Block report file format 22

Table 3 Block payload report file format23

Chapter 1.

INTRODUCTION

The goal of NVIDIA Nsight Compute is to design a profiling tool that can be easily extended and customized by expert users. While we provide useful defaults, this allows adapting the reports to a specific use case or to design new ways to investigate collected data. All the following is data driven and does not require the tools to be recompiled.

While working with section files or rules files it is recommended to open the *Metric Selection* tool window from the *Profile* menu item. This tool window lists all sections and rules that were loaded. Rules are grouped as children of their associated section or grouped in the *[Independent Rules]* entry. For files that failed to load, the table shows the error message. Use the *Reload* button to reload rule files from disk.

Chapter 2.

METRIC SECTIONS

The [Details](#) page consists of [sections](#) that focus on a specific part of the kernel analysis each. Every section is defined by a corresponding [section file](#) that specifies the data to be collected as well as the visualization used in the UI or CLI output for this data. Simply modify a deployed section file to add or modify what is collected.

2.1. Section Files

The section files delivered with the tool are stored in the **sections** sub-folder of the NVIDIA Nsight Compute install directory. Each section is defined in a separate file with the **.section** file extension. At runtime, the installed stock sections (and rules) are deployed to a user-writable directory. This can be disabled with an [environment variable](#). Section files from the deployment directory are loaded automatically at the time the UI connects to a target application or the command line profiler is launched. This way, any changes to section files become immediately available in the next profile run.

A section file is a text representation of a *Google Protocol Buffer* message. The full definition of all available fields of a section message is given in [Section Definition](#). In short, each section consists of a unique *Identifier* (no spaces allowed), a *Display Name*, an optional *Order* value (for sorting the sections in the [Details page](#)), an optional *Description* providing guidance to the user, an optional header table, an optional list of metrics to be collected but not displayed, optional bodies with additional UI elements, and other elements. See **ProfilerSection.proto** for the exact list of available elements. A small example of a very simple section is:

```

Identifier: "SampleSection"
DisplayName: "Sample Section"
Description: "This sample section shows information on active warps and cycles."
Header {
  Metrics {
    Label: "Active Warps"
    Name: "smsp__active_warps_avg"
  }
  Metrics {
    Label: "Active Cycles"
    Name: "smsp__active_cycles_avg"
  }
}

```

On data collection, this section will cause the two PerfWorks metrics **smsp__active_warps_avg** and **smsp__active_cycles_avg** to be collected.

▶ Sample Section			
Active Warps	15,590,870.75	Active Cycles	1,189,536.17

By default, when not available, metrics specified in section files will only generate a warning during data collection, and would then show up as "N/A" in the UI or CLI. This is in contrast to metrics requested via **--metrics** which would cause an error when not available. How to specify metrics as required for data collection is described in [Metric Options and Filters](#).

More advanced elements can be used in the body of a section. See the **ProfilerSection.proto** file for which elements are available. The following example shows how to use these in a slightly more complex example. The usage of regexes is allowed in tables and charts in the section *Body* only and follows the format **regex:** followed by the actual regex to match *PerfWorks* metric names.

The supported list of metrics that can be used in sections can be queried using the [command line interface](#) with the **--query-metrics** option. Each of these metrics can be used in any section and will be automatically collected if they appear in any enabled section. Note that even if a metric is used in multiple sections, it will only be collected once. Look at all the shipped sections to see how they are implemented.

```

Identifier: "SampleSection"
DisplayName: "Sample Section"
Description: "This sample section shows various metrics."
Header {
  Metrics {
    Label: "Active Warps"
    Name: "smsp__active_warps_avg"
  }
  Metrics {
    Label: "Active Cycles"
    Name: "smsp__active_cycles_avg"
  }
}
Body {
  Items {
    Table {
      Label: "Example Table"
      Rows: 2
      Columns: 1
      Metrics {
        Label: "Avg. Issued Instructions Per Scheduler"
        Name: "smsp__inst_issued_avg"
      }
      Metrics {
        Label: "Avg. Executed Instructions Per Scheduler"
        Name: "smsp__inst_executed_avg"
      }
    }
  }
  Items {
    Table {
      Label: "Metrics Table"
      Columns: 2
      Order: ColumnMajor
      Metrics {
        Name: "regex:.*__elapsed_cycles_sum"
      }
    }
  }
  Items {
    BarChart {
      Label: "Metrics Chart"
      CategoryAxis {
        Label: "Units"
      }
      ValueAxis {
        Label: "Cycles"
      }
      Metrics {
        Name: "regex:.*__elapsed_cycles_sum"
      }
    }
  }
}

```




2.2. Section Definition

Protocol buffer definitions are in the NVIDIA Nsight Compute installation directory under **extras/FileFormat**. To understand section files, start with the definitions and documentation in **ProfilerSection.proto**.

To see the list of available *PerfWorks* metrics for any device or chip, use the **--query-metrics** option of the [command line](#).

2.3. Metric Options and Filters

Sections allow the user to specify alternative *options* for metrics that have a different metric name on different GPU architectures. Metric options use a min-arch/max-arch range *filter*, replacing the base metric with the first metric option for which the current GPU architecture matches the filter. While not strictly enforced, options for a base metric are expected to share the same meaning and subsequently unit, etc., with the base metric.

In addition to its options, the base metric can be filtered by the same criteria. This is useful for metrics that are only available for certain architectures or in limited collection scopes. See **ProfilerMetricOptions.proto** for which filter options are available.

In the below example, the metric **dram__cycles_elapsed.avg.per_second** is collected on SM 7.0 and SM 7.5-8.6, but not on any in between. It uses the same metric name on these architectures.

```

Metrics {
  Label: "DRAM Frequency"
  Name: "dram__cycles_elapsed.avg.per_second"
  Filter {
    MaxArch: CC_70
  }
  Options {
    Name: "dram__cycles_elapsed.avg.per_second"
    Filter {
      MinArch: CC_75
      MaxArch: CC_86
    }
  }
}

```

In the next example, the metric in the section header is only collected for launch-based collection scopes (i.e. kernel- and application replay for CUDA kernels or CUDA Graph nodes), but not in range-based scopes.

```

Header {
  Metrics {
    Label: "Theoretical Occupancy"
    Name: "sm__maximum_warps_per_active_cycle_pct"
    Filter {
      CollectionFilter {
        CollectionScopes: CollectionScope_Launch
      }
    }
  }
}

```

Similarly, **CollectionFilters** can be used to set the **Importance** of a metric, which specifies an expectation on its availability during data collection. **Required** metrics, for instance, are expected to be collectable and would generate an error in case they are not available, whereas **Optional** metrics would only generate a warning. Here is a minimal example, illustrating the functionality:

```

Metrics {
  Label: "Compute (SM) Throughput"
  Name: "sm__throughput.avg.pct_of_peak_sustained_elapsed"
  Filter {
    CollectionFilter {
      Importance: Required
    }
  }
}

```

Filters can be applied to an entire section instead of or in addition to being set for individual metrics. If both types of filters are specified, they are combined, such that **Metrics**-scope filters take precedence over section-scope filters.

2.4. Missing Sections

If new or updated section files are not used by NVIDIA Nsight Compute, it is most commonly one of two reasons:

The file is not found: Section files must have the `.section` extension. They must also be on the section search path. The default search path is the `sections` directory within the installation directory. In NVIDIA Nsight Compute CLI, the search paths can be overwritten using the `--section-folder` and `--section-folder-recursive` options. In NVIDIA Nsight Compute, the search path can be configured in the *Profile* options.

Syntax errors: If the file is found but has syntax errors, it will not be available for metric collection. However, error messages are reported for easier debugging. In NVIDIA Nsight Compute CLI, use the `--list-sections` option to get a list of error messages, if any. In NVIDIA Nsight Compute, error messages are reported in the *Metric Selection* tool window.

2.5. Derived Metrics

Derived Metrics allow you to define new metrics composed of constants or existing metrics directly in a section file. The new metrics are computed at collection time and added permanently to the profile result in the report. They can then subsequently be used for any tables, charts, rules, etc.

2.5.1. Metric Types

Metrics collected in NVIDIA Nsight Compute are of two types: Instanced Metrics & Single Value Metrics.

2.5.1.1. Instanced Metrics

Instanced metrics are metrics which consist of a list of key-value pairs along with an associated aggregate value that conveys a summary of the instances. The aggregate value of an instanced metric could mean multiple things depending on the metric itself.

Typically, aggregate values are one of the following:

- ▶ Sum of all instances (default)
- ▶ Average of all instances
- ▶ Minimum value found across all instances
- ▶ Maximum value found across all instances

For example, a metric could have two instances with two keys, two values associated with those keys, and an aggregate value summarizing these instances. A metric with an **aggregate value of 9⁽¹⁾** could look something like this:

Correlation ID	Instance Value
0 ⁽²⁾	4 ⁽³⁾
1 ⁽²⁾	5 ⁽³⁾

(1) Aggregate value of 9 (sum of instance values 4 and 5)

(2) Keys (also known as Correlation IDs) 0 & 1

(3) Values 4 & 5

2.5.1.2. Single Value Metrics

On the other hand, a single value metric does not have any instances; it consists of just that, a single value. Hence, it does not have correlation IDs associated with it, nor does it have instance values.

2.5.2. Metric Value Types

For the most part, NVIDIA Nsight Compute metric value types follow a C/C++ typing paradigm. Metric value types in Nsight Compute are one of the following:

1. Unsigned Integers (32 & 64 bits)
2. Floating point
3. Double precision
4. String

2.5.3. Defining Derived Metrics

Derived metrics provide the functionality of using already existing metrics to construct a new meaningful metric to be included in reports. This is done by defining these metrics in the corresponding section file.

An example of such metric definition would be the following:

```
MetricDefinitions {
  MetricDefinitions {
    Name: "derived_avg_thread_executed"
    Expression: "thread_inst_executed_true / inst_executed"
  }
}
```

This derived metric provides a hint on the number of threads executed on average at each instruction.

Expressions support the following operators:

1. +, -, *, /
2. ()
3. Some functional operators described in a later [section](#)

Resultant metric types are defined based on C/C++ type upcasting. That means that whatever operand has the higher precision in an operation dictates the type of the resulting derived metric.

It is worth noting that when operating on metrics, in addition to Instanced Metrics and Single Value metrics, constants can also be used. For the purposes of these operations, constants are synonymous with Single Value metrics. Constants are always of type unsigned integer (64bits) or double.

2.5.3.1. Addition

Addition (+)	Single Value Metric / Constant	Instanced Metric
Single Value Metric / Constant	Result is a single value metric ⁽¹⁾	Result is an instanced metric ⁽²⁾
Instanced Metric	Result is an instanced metric ⁽²⁾	Result is an instanced metric ⁽³⁾

(1) Adding two Single Value metrics (or constants) results in a single value metric with the highest precision provided

(2) Adding a Single Value metric and a constant results in an Instanced Metric. The constant is added to the instances of the metric, then the aggregate value is revised based on the resultant's instance values

(3) Adding two Instanced Metrics results in an Instanced Metric. Instances with a matching Correlation ID (key) are added together, and any other unique metrics are copied over to the resulting metric. The aggregate value is then revised based on the resultant's instance values

2.5.3.2. Subtraction

Subtraction (-)	Single Value Metric / Constant	Instanced Metric
Single Value Metric / Constant	Result is a single value metric ⁽¹⁾	Result is an instanced metric ⁽²⁾
Instanced Metric	Result is an instanced metric ⁽²⁾	Result is an instanced metric ⁽³⁾

(1) Subtracting two Single Value metrics (or constants) results in a single value metric with the highest precision provided. In the case where both operands are unsigned, and the result is a negative number, the resulting derived metric type is then changed to a double

(2) Subtracting a Single Value metric and a constant results in an Instanced Metric. The constant is subtracted from the instances of the metric (or vice versa), then the aggregate value is revised based on the resultant's instance values

(3) Subtracting two Instanced Metrics results in an Instanced Metric. Instances with a matching Correlation ID (key) are subtracted from each other, and any other unique metrics are copied over to the resulting metric (i.e a union operation). The aggregate value is then revised based on the resultant's instance values

2.5.3.3. Multiplication

Multiplication (*)	Single Value Metric / Constant	Instanced Metric
Single Value Metric / Constant	Result is a single value metric ⁽¹⁾	Result is an instanced metric ⁽²⁾
Instanced Metric	Result is an instanced metric ⁽²⁾	Result is an instanced metric ⁽³⁾

(1) Multiplying two Single Value metrics (or constants) results in a single value metric with the highest precision provided

(2) Multiplying a Single Value metric and a constant results in an Instanced Metric. The constant is multiplied by the instances of the metric (or vice versa), then the aggregate value is revised based on the resultant's instance values

(3) Multiplying two Instanced Metrics results in an Instanced Metric. Instances with a matching Correlation ID (key) are multiplied, and any other unique metrics are discarded (i.e Intersection operation). The aggregate value is then revised based on the resultant's instance values

2.5.3.4. Divison

Divison (/)	Single Value Metric / Constant	Instanced Metric
Single Value Metric / Constant	Result is a single value metric ⁽¹⁾	Result is an instanced metric ⁽²⁾
Instanced Metric	Result is an instanced metric ⁽²⁾	Result is an instanced metric ⁽³⁾

(1) Dividing two Single Value metrics (or constants) results in a single value metric with the highest precision provided

(2) Dividing a Single Value metric and a constant results in an Instanced Metric. The constant is divided by the instances of the metric (or vice versa), then the aggregate value is revised based on the resultant's instance values

(3) Dividing two Instanced Metrics results in an Instanced Metric. Instances with a matching Correlation ID (key) are divided, and any other unique metrics are discarded (i.e Intersection operation). The aggregate value is then revised based on the resultant's instance values

2.5.3.5. Functional Operators

In addition to the usual mathematical operations described above, there are also a few functional operators that allow some more control over the resulting derived metric. These functional operators are used in a similar fashion to a C/C++ function with a single argument.

As described before, the aggregative value is typically the sum of the instance metrics. To override that functionality and change the aggregate value to another type of aggregation, the following is provided:

- ▶ **revise_min()** This functional operator revises the aggregate value to be the minimum of the instance values. For example:
`revise_min(example_instance_metric + 1.0)`
- ▶ **revise_max()** This functional operator revises the aggregate value to be the maximum of the instance values. For example:
`revise_max(example_instance_metric + 1.0)`
- ▶ **revise_avg()** This functional operator revises the aggregate value to be the average of the instance values. For example:
`revise_avg(example_instance_metric + 1.0)`
- ▶ **revise_sum()** This functional operator revises the aggregate value to be the sum of the instance values. For example: `revise_sum(example_instance_metric + 1.0)`

In addition to the above functional operators, there is another special operator that allows us to change an Instanced Metric to a Single Value Metric consisting only of the aggregate value: **simplify()** This operator discards all instance values and maintains the aggregate value only.

Chapter 3.

RULE SYSTEM

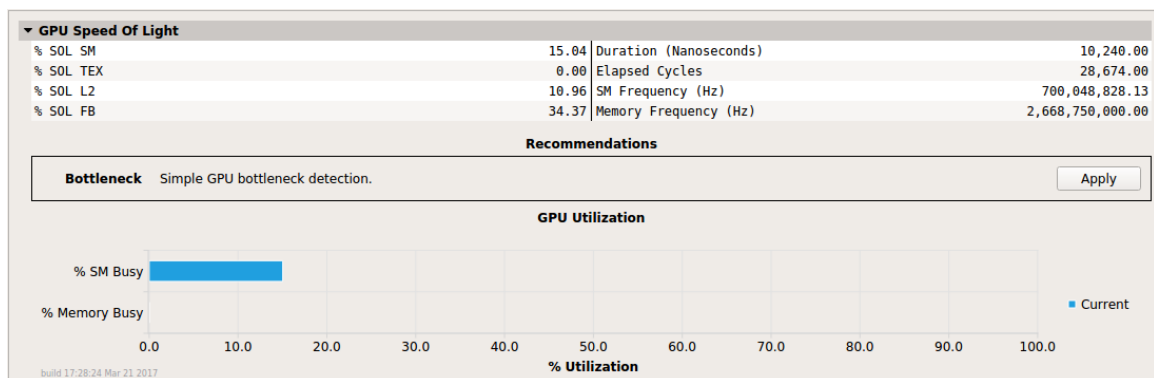
NVIDIA Nsight Compute features a new Python-based rule system. It is designed as the successor to the *Expert System* (un)guided analysis in NVIDIA Visual Profiler, but meant to be more flexible and more easily extensible to different use cases and APIs.

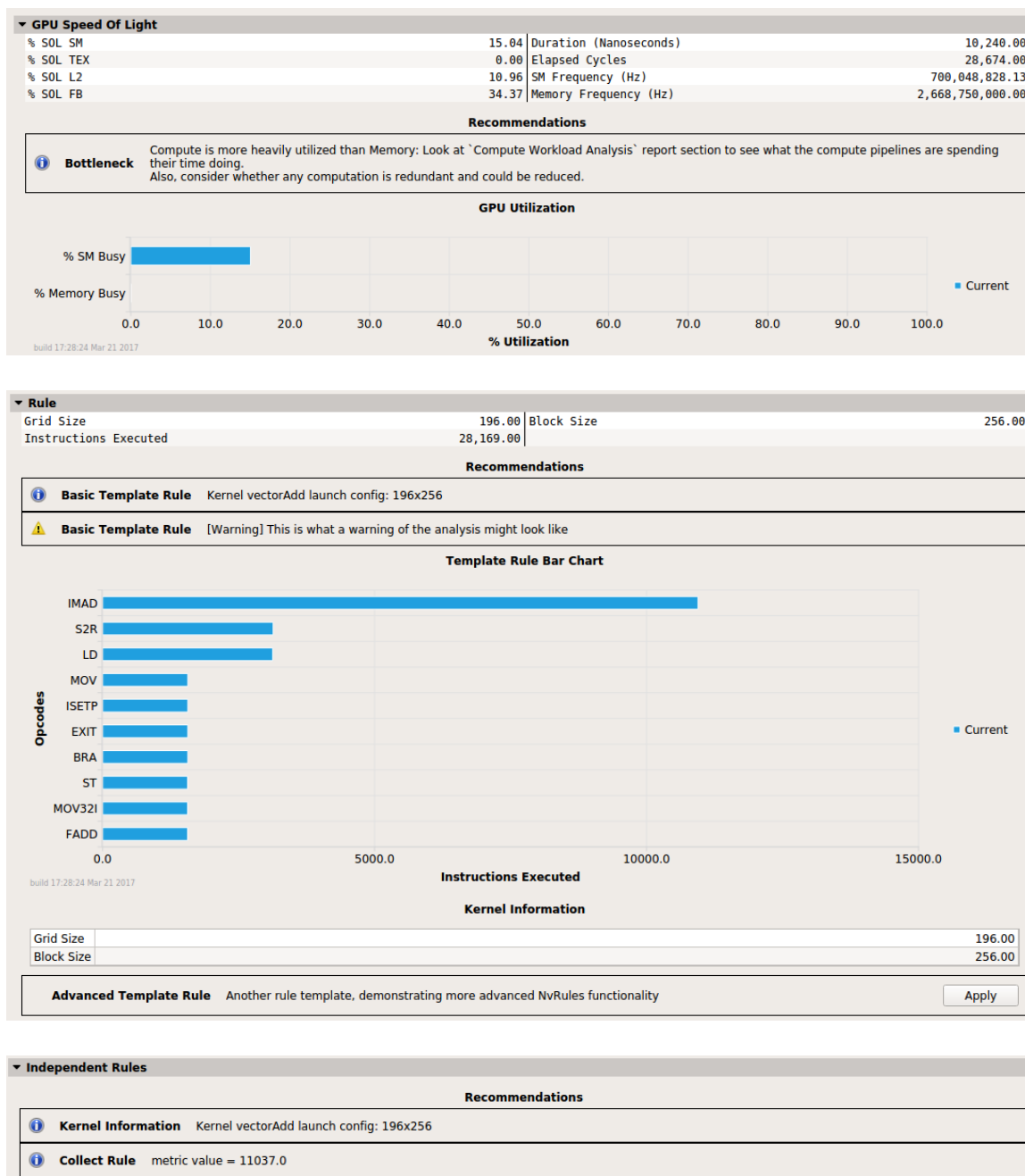
3.1. Writing Rules

To create a new rule, you need to create a new text file with the extension **.py** and place it at some location that is detectable by the tool (see Nsight Compute Integration on how to specify the search path for rules). At a minimum, the rule file must implement two functions, **get_identifier** and **apply**. See Rule File API for a description of all functions supported in rule files. See NvRules for details on the interface available in the rule's **apply** function.

3.2. Integration

The rule system is integrated into NVIDIA Nsight Compute as part of the profile report view. When you profile a kernel, available rules will be shown in the report's *Details* page. You can either select to apply all available rules at once by clicking *Apply Rules* at the top of the page, or apply rules individually. Once applied, the rule results will be added to the current report. By default, all rules are applied automatically.





3.3. Rule System Architecture

The rule system consists of the Python interpreter, the *NvRules C++ interface*, the *NvRules Python interface* (NvRules.py) and a set of rule files. Each rule file is valid Python code that imports the NvRules.py module, adheres to certain standards defined by the [Rule File API](#) and is called to from the tool.

When applying a rule, a handle to the rule *Context* is provided to its apply function. This context captures most of the functionality that is available to rules as part of the [NvRules](#)

API. In addition, some functionality is provided directly by the `NvRules` module, e.g. for global error reporting. Finally, since rules are valid Python code, they can use regular libraries and language functionality that ship with Python as well.

From the rule *Context*, multiple further objects can be accessed, e.g. the *Frontend*, *Ranges* and *Actions*. It should be noted that those are only interfaces, i.e. the actual implementation can vary from tool to tool that decides to implement this functionality.

Naming of these interfaces is chosen to be as API-independent as possible, i.e. not to imply CUDA-specific semantics. However, since many compute and graphics APIs map to similar concepts, it can easily be mapped to CUDA terminology, too. A *Range* refers to a CUDA stream, an *Action* refers to a single CUDA kernel instance. Each action references several *Metrics* that have been collected during profiling (e.g. **instructions executed**) or are statically available (e.g. the launch configuration). *Metrics* are accessed via their names from the *Action*.

Each CUDA stream can contain any number of kernel (or other device activity) instances and so each *Range* can reference one or more *Actions*. However, currently only a single *Action* per *Range* will be available, as only a single CUDA kernel can be profiled at once.

The *Frontend* provides an interface to manipulate the tool UI by adding messages, graphical elements such as line and bar charts or tables, as well as speedup estimations, focus metrics and source markers. The most common use case is for a rule to show at least one message, stating the result to the user, as illustrated in `extras/RuleTemplates/BasicRuleTemplate.py`. This could be as simple as "No issues have been detected," or contain direct hints as to how the user could improve the code, e.g. "Memory is more heavily utilized than Compute. Consider whether it is possible for the kernel to do more compute work." For more advanced use cases, such as adding speedup estimates, key performance indicators (a.k.a. focus metrics) or source markers to annotate individual lines of code to your rule, see the templates in `extras/RuleTemplates`.

3.4. NvRules API

The *NvRules API* is defined as a C/C++ style interface, which is converted to the `NvRules.py` Python module to be consumable by the rules. As such, C++ class interfaces are directly converted to Python classes and functions. See the [NvRules API](#) documentation for the classes and functions available in this interface.

3.5. Rule File API

The *Rule File API* is the implicit contract between the rule Python file and the tool. It defines which functions (syntactically and semantically) the Python file must provide to properly work as a rule.

Mandatory Functions

- `get_identifier()`: Return the unique rule identifier string.

- ▶ **apply(handle)**: Apply this rule to the rule context provided by handle. Use `NvRules.get_context(handle)` to obtain the *Context* interface from handle.
- ▶ **get_name()**: Return the user-consumable display name of this rule.
- ▶ **get_description()**: Return the user-consumable description of this rule.

Optional Functions

- ▶ **get_section_identifier()**: Return the unique section identifier that maps this rule to a section. Section-mapped rules will only be available if the corresponding section was collected. They implicitly assume that the metrics requested by the section are collected when the rule is applied.
- ▶ **evaluate(handle)**:

Declare required metrics and rules that are necessary for this rule to be applied. Use `NvRules.require_metrics(handle, [...])` to declare the list of metrics that must be collected prior to applying this rule.

Use e.g. `NvRules.require_rules(handle, [...])` to declare the list of other rules that must be available before applying this rule. Those are the only rules that can be safely proposed by the *Controller* interface.

3.6. Rule Examples

The following example rule determines on which major GPU architecture a kernel was running.

```
import NvRules

def get_identifier():
    return "GpuArch"

def apply(handle):
    ctx = NvRules.get_context(handle)
    action = ctx.range_by_idx(0).action_by_idx(0)
    ccMajor =
    action.metric_by_name("device__attribute_compute_capability_major").as_uint64()
    ctx.frontend().message("Running on major compute capability " + str(ccMajor))
```

Chapter 4.

PYTHON REPORT INTERFACE

NVIDIA Nsight Compute features a Python-based interface to interact with exported report files.

The module is called **ncu_report** and works on any Python version from 3.4¹. It can be found in the **extras/python** directory of your NVIDIA Nsight Compute package.

In order to use the Python module, you need a report file generated by NVIDIA Nsight Compute. You can obtain such a file by saving it from the graphical interface or by using the **--export** flag of the command line tool.

The types and functions in the **ncu_report** module are a subset of the ones available in the **NvRules** API. The documentation in this section serves as a tutorial. For a more formal description of the exposed API, please refer to the [NvRules API](#) documentation.

4.1. Basic Usage

In order to be able to import **ncu_report** you will either have to navigate to the **extras/python** directory, or add its absolute path to the **PYTHONPATH** environment variable. Then, the module can be imported like any Python module:

```
>>> import ncu_report
```

Importing a report

Once the module is imported, you can load a report file by calling the **load_report** function with the path to the file. This function returns an object of type **IContext** which holds all the information concerning that report.

```
>>> my_context = ncu_report.load_report("my_report.ncu-rep")
```

Querying ranges

¹ On Linux machines you will also need a GNU-compatible libc and **libgcc_s.so**.

When working with the Python module, kernel profiling results are grouped into *ranges* which are represented by **IRange** objects. You can inspect the number of *ranges* contained in the loaded report by calling the **num_ranges()** member function of an **IContext** object and retrieve a *range* by its index using **range_by_idx(index)**.

```
>>> my_context.num_ranges()
1
>>> my_range = my_context.range_by_idx(0)
```

Querying actions

Inside a *range*, kernel profiling results are called *actions*. You can query the number of *actions* contained in a given *range* by using the **num_actions** method of an **IRange** object.

```
>>> my_range.num_actions()
2
```

In the same way *ranges* can be obtained from an **IContext** object by using the **range_by_idx(index)** method, individual *actions* can be obtained from **IRange** objects by using the **action_by_idx(index)** method. The resulting *actions* are represented by the **IAction** class.

```
>>> my_action = my_range.action_by_idx(0)
```

As mentioned previously, an *action* represents a single kernel profiling result. To query the kernel's name you can use the **name()** member function of the **IAction** class.

```
>>> my_action.name()
MyKernel
```

Querying metrics

To get a tuple of all metric names contained within an *action* you can use the **metric_names()** method. It is meant to be combined with the **metric_by_name()** method which returns an **IMetric** object. However, for the same task you may also use the **[]** operator, as explained in the [High-Level Interface](#) section below.

The metric names displayed here are the same as the ones you can use with the **--metrics** flag of NVIDIA Nsight Compute. Once you have extracted a *metric* from an *action*, you can obtain its value by using one of the following three methods:

- ▶ **as_string()** to obtain its value as a Python **str**
- ▶ **as_uint64()** to obtain its value as a Python **int**
- ▶ **as_double()** to obtain its value as a Python **float**

For example, to print the display name of the GPU on which the kernel was profiled you can query the **device__attribute__display_name** metric.

```
>>> display_name_metric =
    my_action.metric_by_name('device__attribute_display_name')
>>> display_name_metric.as_string()
'NVIDIA GeForce RTX 3060 Ti'
```

Note that accessing a metric with the wrong type can lead to unexpected (conversion) results.

```
>>> display_name_metric.as_double()
0.0
```

Therefore, it is advisable to directly use the [High-Level](#) function `value()`, as explained below.

4.2. High-Level Interface

On top of the low-level [NvRules API](#) the Python Report Interface also implements part of the [Python object model](#). By implementing special methods, the Python Report Interface's exposed classes can be used with built-in Python mechanisms such as iteration, string formatting and length querying.

This allows you to access *metrics* objects via the `self[key]` instance method of the **IAction** class:

```
>>> display_name_metric = my_action["device__attribute_display_name"]
```

There is also a convenience method **IMetric.value()** which allows you to query the value of a *metric* object without knowledge of its type:

```
>>> display_name_metric.value()
'NVIDIA GeForce RTX 3060 Ti'
```

All the available methods of a class, as well as their associated Python docstrings, can be looked up interactively via

```
>>> help(ncu_report.IMetric)
```

or similarly for other classes and methods. In your code, you can access the docstrings via the `__doc__` attribute, i.e. `ncu_report.IMetric.value.__doc__`.

4.3. Metric attributes

Apart from the possibility to query the `name()` and `value()` of an **IMetric** object, you can also query the following additional metric attributes:

- ▶ `metric_type()`

- ▶ `metric_subtype()`
- ▶ `rollup_operation()`
- ▶ `unit()`
- ▶ `description()`

The first method `metric_type()` returns one out of three *enum* values (`IMetric.MetricType_COUNTER`, `IMetric.MetricType_RATIO`, `IMetric.MetricType_THROUGHPUT`) if the metric is a hardware metric, or `IMetric.MetricType_OTHER` otherwise (e.g. for launch or device attributes).

The method `metric_subtype()` returns an *enum* value representing the subtype of a metric (e.g. `IMetric.MetricSubtype_PEAK_SUSTAINED`, `IMetric.MetricSubtype_PER_CYCLE_ACTIVE`). In case a metric does not have a subtype, `None` is returned. All available values (without the necessary `IMetric.MetricSubtype_` prefix) may be found in the [NvRules API](#) documentation, or may be looked up interactively by executing `help(ncu_report.IMetric)`.

`IMetric.rollup_operation()` returns the operation which is used to accumulate different values of the same *metric* and can be one of `IMetric.RollupOperation_AVG`, `IMetric.RollupOperation_MAX`, `IMetric.RollupOperation_MIN` or `IMetric.RollupOperation_SUM` for averaging, maximum, minimum or summation, respectively. If the *metric* in question does not specify a rollup operation `None` will be returned.

Lastly, `unit()` and `description()` return a (possibly empty) string of the metric's *unit* and a short textual *description* for hardware metrics, respectively.

The above methods can be combined to filter through all *metrics* of a report, given certain criteria:

```
for metric in metrics:
    if metric.metric_type() == IMetric.MetricType_COUNTER and \
       metric.metric_subtype() == IMetric.MetricSubtype_PER_SECOND and \
       metric.rollup_operation() == IMetric.RollupOperation_AVG:
        print(f"{metric.name()}: {metric.value()} {metric.unit()}")
```

4.4. NVTX Support

The `ncu_report` has support for the NVIDIA Tools Extension (NVTX). This comes through the `INvtxState` object which represents the NVTX state of a profiled kernel.

An `INvtxState` object can be obtained from an action by using its `nvtx_state()` method. It exposes the `domains()` method which returns a tuple of integers representing the domains this kernel has state in. These integers can be used with the `domain_by_id(id)` method to get an `INvtxDomainInfo` object which represents the state of a domain.

The `INvtxDomainInfo` can be used to obtain a tuple of *Push-Pop*, or *Start-End* ranges using the `push_pop_ranges()` and `start_end_ranges()` methods.

There is also a **actions_by_nvtx** member function in the **IRange** class which allows you to get a tuple of actions matching the NVTX state described in its parameter.

The parameters for the **actions_by_nvtx** function are two lists of strings representing the state for which we want to query the actions. The first parameter describes the NVTX states to include while the second one describes the NVTX states to exclude. These strings are in the same format as the ones used with the **--nvtx-include** and **--nvtx-exclude** options.

4.5. Sample Script

NVTX *Push-Pop* range filtering

This is a sample script which loads a report and prints the names of all the profiled kernels which were wrapped inside **BottomRange** and **TopRange** *Push-Pop* ranges of the default NVTX domain.

```
#!/usr/bin/env python3

import sys

import ncu_report

if len(sys.argv) != 2:
    print("usage: {} report_file".format(sys.argv[0]), file=sys.stderr)
    sys.exit(1)

report = ncu_report.load_report(sys.argv[1])

for range_idx in range(report.num_ranges()):
    current_range = report.range_by_idx(range_idx)
    for action_idx in current_range.actions_by_nvtx(["BottomRange/*TopRange"],
    []):
        action = current_range.action_by_idx(action_idx)
        print(action.name())
```

Chapter 5.

SOURCE COUNTERS

The *Source* page provides correlation of various metrics with CUDA-C, PTX and SASS source of the application, depending on availability.

Which *Source Counter* metrics are collected and the order in which they are displayed in this page is controlled using section files, specifically using the *ProfilerSectionMetrics* message type. Each *ProfilerSectionMetrics* defines one ordered group of metrics, and can be assigned an optional *Order* value. This value defines the ordering among those groups in the *Source* page. This allows, for example, you to define a group of memory-related source counters in one and a group of instruction-related counters in another section file.

```
Identifier: "SourceMetrics"
DisplayName: "Custom Source Metrics"
Metrics {
  Order: 2
  Metrics {
    Label: "Instructions Executed"
    Name: "inst_executed"
  }
  Metrics {
    Label: ""
    Name: "collected_but_not_shown"
  }
}
```

If a *Source Counter* metric is given an empty label attribute in the section file, it will be collected but not shown on the page.

#	Address	Source	Sampling Data (All)	Sampling Data (No Issue)	Instructions Executed	Predicated-On Thread Instructions Executed
1	16ff2d80 @IPT SHFL.IDX PT, RZ, RZ, RZ, RZ		45		1 65,536	2,097,152
2	16ff2d90 IMAD.MOV.U32 R1, RZ, RZ, c[0x0][0x28]		18		0 65,536	2,097,152
3	16ff2da0 S2R R2, SR_CTAID.Y		5		4 65,536	2,097,152
4	16ff2db0 S2R R3, SR_CTAID.X		0		1 65,536	2,097,152
5	16ff2dc0 S2R R5, SR_TID.Y		0		1 65,536	2,097,152
6	16ff2dd0 S2R R8, SR_TID.X		1		2 65,536	2,097,152
7	16ff2de0 IMAD R2, R2, c[0x0][0xc], R3 {0}		9		10 65,536	2,097,152
8	16ff2df0 IMAD R2, R2, c[0x0][0x4], R5 {1}		3		2 65,536	2,097,152
9	16ff2e00 IMAD R2, R2, c[0x0][0x0], R0 {2}		3		5 65,536	2,097,152
10	16ff2e10 ISETP.GE.U32.AND P0, PT, R2, c[0x0][0x168], PT, !PT		8		8 65,536	2,097,152
11	16ff2e20 BSSY B0, 0xb16ff2f20		0		2 65,536	2,097,152
12	16ff2e30 PRMT R3, RZ, 0x7610, R3		0		1 65,536	2,097,152
13	16ff2e40 @P0 BRA 0xb16ff2f10		2		6 65,536	2,097,152
14	16ff2e50 LOP3.LUT R4, R2.reuse, 0x7f, RZ, 0xc0, !PT		0		3 65,536	2,097,152
15	16ff2e60 LOP3.LUT R3, R2, 0xffffffff80, RZ, 0xc0, !PT		0		0 65,536	2,097,152
16	16ff2e70 IMAD.SHL.U32 R4, R4, 0x4, RZ		0		3 65,536	2,097,152
17	16ff2e80 IMAD R3, R3, 0x330, R4		0		2 65,536	2,097,152
18	16ff2e90 IADD3 R6, P0, R3, c[0x3][0x430], RZ		37		7 65,536	2,097,152
19	16ff2ea0 IMAD.X R7, RZ, RZ, c[0x3][0x434], P0		2		0 65,536	2,097,152
20	16ff2eb0 LDG.E.U8.STRONG.CTA R6, [R6+0x10003]		6		0 65,536	2,097,152
21	16ff2ec0 IMAD.MOV.U32 R3, RZ, 0x1		0		0 65,536	2,097,152
22	16ff2ed0 SHF.L.U32 R3, R3, R6, RZ {0}		519	165	65,536	2,097,152
23	16ff2ee0 LOP3.LUT R3, R3, c[0x0][0x16c], RZ, 0xc0, !PT		2		6 65,536	2,097,152

Chapter 6.

REPORT FILE FORMAT

This section documents the internals of the profiler report files (reports in the following) as created by NVIDIA Nsight Compute. **The file format is subject to change in future releases without prior notice.**

6.1. Version 7 Format

Reports of version 7 are a combination of raw binary data and serialized Google Protocol Buffer version 2 messages (proto). All binary entries are stored as little endian. Protocol buffer definitions are in the NVIDIA Nsight Compute installation directory under **extras/FileFormat**.

Table 1 Top-level report file format

Offset [bytes]	Entry	Type	Value
0	Magic Number	Binary	NVR10
4	Integer	Binary	sizeof(File Header)
8	File Header	Proto	Report version
8 + sizeof(File Header)	Block 0	Mixed	CUDA CUBIN source, profile results, session information
8 + sizeof(File Header) + sizeof(Block 0)	Block 1	Mixed	CUDA CUBIN source, profile results, session information
...

Table 2 Per-Block report file format

Offset [bytes]	Entry	Type	Value
0	Integer	Binary	sizeof(Block Header)

Offset [bytes]	Entry	Type	Value
4	Block Header	Proto	Number of entries per payload type, payload size
4 + sizeof(Block Header)	Block Payload	Mixed	Payload (CUDA CUBIN sources, profile results, session information, string table)

Table 3 Block payload report file format

Offset [bytes]	Entry	Type	Value
0	Integer	Binary	sizeof(Payload type 1, entry 1)
4	Payload type 1, entry 1	Proto	
4 + sizeof(Payload type 1, entry 1)	Integer	Binary	sizeof(Payload type 1, entry 2)
8 + sizeof(Payload type 1, entry 1)	Payload type 1, entry 2	Proto	
...
...	Integer	Binary	sizeof(Payload type 2, entry 1)
...	Payload type 2, entry 1	Proto	
...

Notice

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

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

Trademarks

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

Copyright

© 2018-2023 NVIDIA Corporation and affiliates. All rights reserved.

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