NSIGHT COMPUTE

v2022.4.1 | January 2023

Customization Guide

TABLE OF CONTENTS

Chapter 1. Introduction....1
Chapter 2. Sections....2
2.1. Section Files....2
2.2. Section Definition....5
2.3. Metric Options....5
2.4. Missing Sections....5
2.5. Derived Metrics....6
Chapter 3. Rule System....8
3.1. Writing Rules....8
3.2. Integration....8
3.3. Rule System Architecture....9
3.4. NvRules API....10
3.5. Rule File API....10
3.6. Rule Examples....11
Chapter 4. Python Report Interface....12
4.1. Basic Usage....12
4.2. High-Level Interface....14
4.3. Metric attributes....14
4.4. NVTX Support....15
4.5. Sample Script....16
Chapter 5. Source Counters....17
Chapter 6. Report File Format....19
6.1. Version 7 Format....19

LIST OF TABLES

Table 1 Top-level report file format 19

Table 2 Per-Block report file format 19

Table 3 Block payload report file format 20

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 *Sections/ Rules* 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. 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 to present this data. Simply modify a section file to add or modify what is collected.

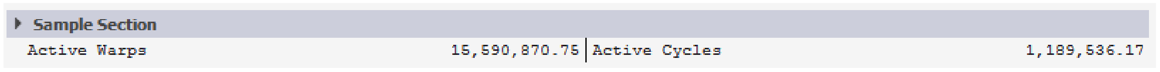
2.1. Section Files

By default, the section files 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. Section files are loaded automatically at the time the UI connects to a target application or the command line profiler is launched. That 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, and an optional body with additional UI 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.



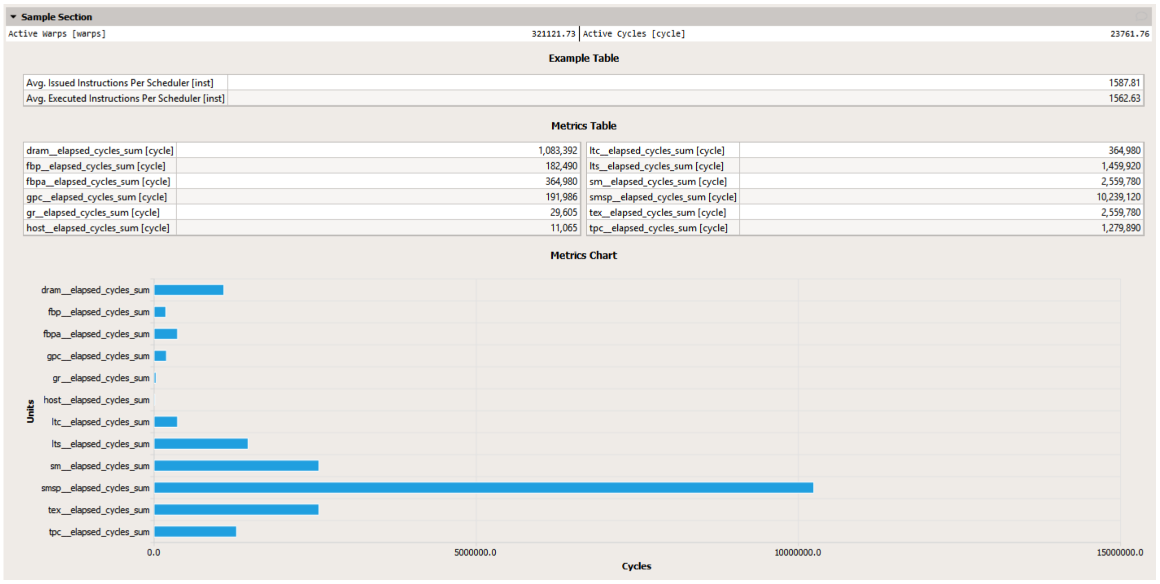
Sample Section

```
Active Warps                15,590,870.75 | Active Cycles                1,189,536.17
```

More advanced elements can be used in the body of a section. Currently, NVIDIA Nsight Compute supports tables and various bar charts. 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 NVIDIA Nsight Compute CLI with option **`--query-metrics`**. Each of these metrics can be used in any section and will be automatically be collected if they appear in any enabled section. Look at all the shipping 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"
      }
    }
  }
}
```



Sample Section

Active Warps [warps]	321121.73	Active Cycles [cycle]	23761.76

Example Table

Avg. Issued Instructions Per Scheduler [inst]	1587.81
Avg. Executed Instructions Per Scheduler [inst]	1562.63

Metrics Table

dram__elapsed_cycles_sum [cycle]	1,083,392	ltc__elapsed_cycles_sum [cycle]	364,980
fbp__elapsed_cycles_sum [cycle]	182,490	lts__elapsed_cycles_sum [cycle]	1,459,920
fbpa__elapsed_cycles_sum [cycle]	364,980	sm__elapsed_cycles_sum [cycle]	2,559,780
gpc__elapsed_cycles_sum [cycle]	191,986	smsp__elapsed_cycles_sum [cycle]	10,239,120
gr__elapsed_cycles_sum [cycle]	29,605	tex__elapsed_cycles_sum [cycle]	2,559,780
host__elapsed_cycles_sum [cycle]	11,065	tpc__elapsed_cycles_sum [cycle]	1,279,890

Metrics Chart

2.2. Section Definition

Protocol buffer definitions are in the NVIDIA Nsight Compute installation directory under **`extras/FileFormat`**.

To see the list of available *PerfWorks* metrics for any device or chip, use the **`--query-metrics`** option of the NVIDIA Nsight Compute CLI.

2.3. Metric Options

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 alternatives, the base metric can be filtered by the same criteria (currently min/max architecture). This is useful for metrics that are only available for certain architectures.

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 *Sections/Rules Info* tool window.

2.5. Derived Metrics

Derived Metrics allows 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.

NVIDIA Nsight Compute currently supports the following syntax for defining derived metrics in section file:

```
MetricDefinitions {
  MetricDefinitions {
    Name: "derived_metric_name"
    Expression: "derived_metric_expr"
  }
  MetricDefinitions {
    ...
  }
  ...
}
```

The actual metric expression is defined as follows:

```
derived_metric_expr ::= operand operator operand
operator            ::= + | - | * | /
operand             ::= metric | constant
metric              ::= (an existing metric name)
constant            ::= double | uint64
double              ::= (double-precision number of the form "N.(M)?", e.g. "5."
 or "0.3109")
uint64              ::= (64-bit unsigned integer number of the form "N", e.g.
 "2029")
```

Operators are defined as follows:

```
For op in (+ | - | *): For each element in a metric it is applied to, the
 expression left-hand side op-combined with expression right-hand side.
For op in (/): For each element in a metric it is applied to, the expression
 left-hand side op-combined with expression right-hand side. If the right-hand
 side operand is of integer-type, and 0, the result is the left-hand side value.
```

Since metrics can contain regular values and/or instanced values, elements are combined as below. Constants are treated as metrics with only a regular value.

```
1. Regular values are operator-combined.
a + b

2. If both metrics have no correlation ids, the first N values are operator-
combined, where N is the minimum of the number of elements in both metrics.
a1 + b1
a2 + b2
a3
a4

3. Else if both metrics have correlation ids, the sets of correlation ids from
 both metrics are joined and then operator-combined as applicable.
a1 + b1
a2
b3
a4 + b4
b5

4. Else if only the left-hand side metric has correlation ids, the right-hand
 side regular metric value is operator-combined with every element of the left-
hand side metric.
a1 + b
a2 + b
a3 + b

5. Else if only the right-hand side metric has correlation ids, the right-hand
 side element values are operator-combined with the regular metric value of the
 left-hand side metric.
a + b1 + b2 + b3
```

In all operations, the value kind of the left-hand side operand is used. If the right-hand side operand has a different value kind, it is converted. If the left-hand side operand is a string-kind, it is returned unchanged.

Examples for derived metrics are **`derived__avg_thread_executed`**, which provides a hint on the number of threads executed on average at each instruction, and **`derived__uncoalesced_l2_transactions_global`**, which indicates the ratio of actual L2 transactions vs. ideal L2 transactions at each applicable instruction.

```
MetricDefinitions {
  MetricDefinitions {
    Name: "derived__avg_thread_executed"
    Expression: "thread_inst_executed_true / inst_executed"
  }
  MetricDefinitions {
    Name: "derived__uncoalesced_l2_transactions_global"
    Expression: "memory_l2_transactions_global /
 memory_ideal_l2_transactions_global"
  }
  MetricDefinitions {
    Name: "sm__sass_thread_inst_executed_op_ffma_pred_on_x2"
    Expression:
 "sm__sass_thread_inst_executed_op_ffma_pred_on.sum.peak_sustained * 2"
  }
}
```

Chapter 3. RULE SYSTEM

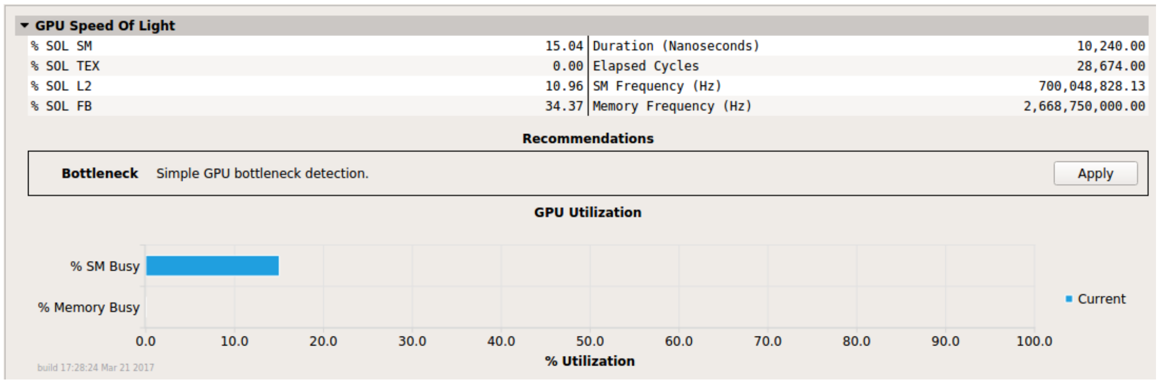
NVIDIA Nsight Compute features a new Python-based rule system. It is designed as the successor to the *Expert System* (un)guided analysis in NVIDIA Visual Profiler, but meant to be more flexible and more easily extensible to different use cases and APIs.

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

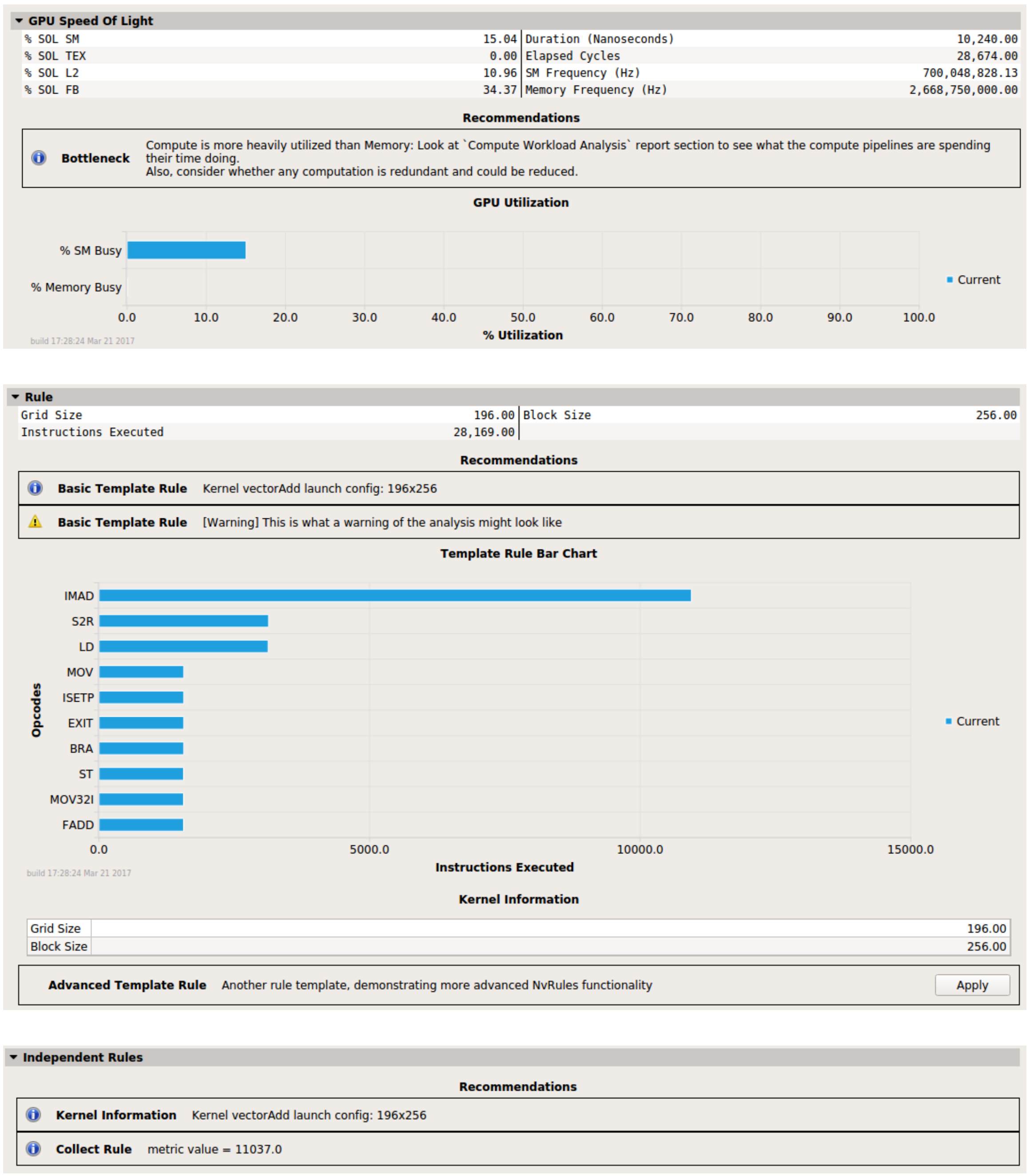
The rule system is integrated into NVIDIA Nsight Compute as part of the profile report view. When you profile a kernel, available rules will be shown in the report's *Details* page. You can either select to apply all available rules at once by clicking *Apply Rules* at the top of the page, or apply rules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


GPU Speed Of Light

% SOL SM	15.04	Duration (Nanoseconds)	10,240.00
% SOL TEX	0.00	Elapsed Cycles	28,674.00
% SOL L2	10.96	SM Frequency (Hz)	700,048,828.13
% SOL FB	34.37	Memory Frequency (Hz)	2,668,750,000.00

Recommendations

Bottleneck Simple GPU bottleneck detection.



GPU Speed Of Light

% SOL SM	15.04	Duration (Nanoseconds)	10,240.00
% SOL TEX	0.00	Elapsed Cycles	28,674.00
% SOL L2	10.96	SM Frequency (Hz)	700,048,828.13
% SOL FB	34.37	Memory Frequency (Hz)	2,668,750,000.00

Recommendations

Bottleneck Compute is more heavily utilized than Memory: Look at `Compute Workload Analysis` report section to see what the compute pipelines are spending their time doing.
Also, consider whether any computation is redundant and could be reduced.

Rule

Grid Size	196.00	Block Size	256.00
Instructions Executed	28,169.00		

Recommendations

Basic Template Rule Kernel vectorAdd launch config: 196x256

Basic Template Rule [Warning] This is what a warning of the analysis might look like

Kernel Information

Grid Size	196.00
Block Size	256.00

Advanced Template Rule Another rule template, demonstrating more advanced NvRules functionality

Independent Rules

Recommendations

Kernel Information Kernel vectorAdd launch config: 196x256

Collect Rule metric value = 11037.0

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 or graphical elements such as line and bar charts or tables. The most common use case is for a rule to show at least one message, stating the result to the user. 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."

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 und 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 3.4 [1]. 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 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())
```

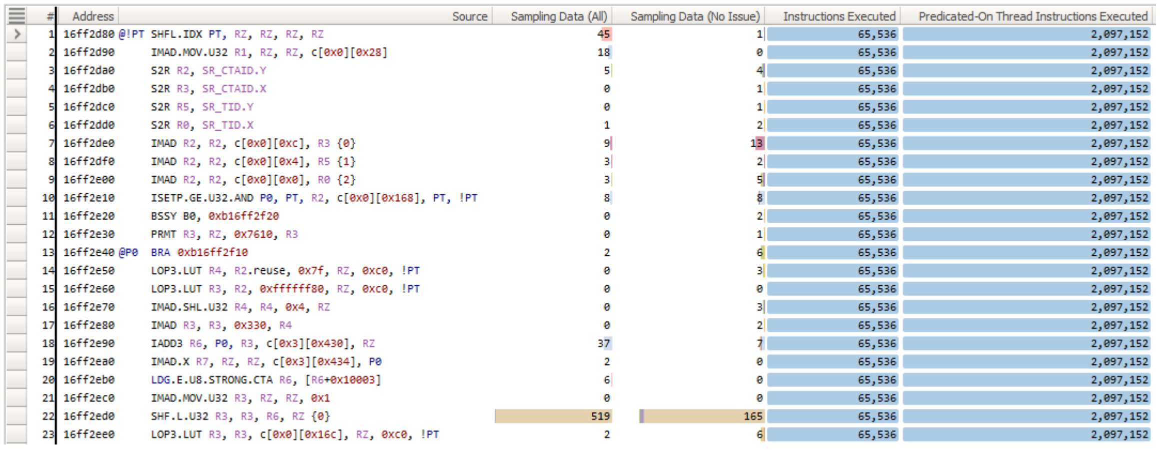
Chapter 5. SOURCE COUNTERS

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	@!PT 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 R0, SR_TID.X	1	2	65,536	2,097,152
7	16ff2de0	IMAD R2, R2, c[0x0][0xc], R3 {0}	9	13	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, 0xffffff80, 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, 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	NVP\0
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	
…	…	…	…