# TABLE OF CONTENTS

Chapter 1. Introduction
Chapter 2. Sections
  2.1. Section Files
  2.2. Section Definition
  2.3. Metric Options
  2.4. Missing Sections
  2.5. Derived Metrics
Chapter 3. Rule System
  3.1. Writing Rules
  3.2. Integration
  3.3. Rule System Architecture
  3.4. NvRules API
  3.5. Rule File API
  3.6. Rule Examples
Chapter 4. Python Report Interface
  4.1. Basic Usage
  4.2. NVTX Support
  4.3. Sample Script
Chapter 5. Source Counters
Chapter 6. Report File Format
  6.1. Version 7 Format
LIST OF TABLES

Table 1  Top-level report file format ........................................................................ 18
Table 2  Per-Block report file format ........................................................................ 18
Table 3  Block payload report file format ................................................................... 19
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.
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"
      }
    }
  }
}
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 \(--\text{query-metrics}\) 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 \(\text{sections}\) directory within the installation directory. In NVIDIA Nsight Compute CLI, the search paths can be overwritten using the \(--\text{section-folder}\) and \(--\text{section-folder-recursive}\)
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:

```plaintext
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.
   \( a_1 + b_1 \\
   a_2 + b_2 \\
   a_3 \\
   a_4 \)

3. Else if both metrics have correlation ids, the sets of correlation ids from both metrics are joined and then operator-combined as applicable.
   \( a_1 + b_1 \\
   a_2 \\
   b_3 \\
   a_4 + b_4 \\
   b_5 \)

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.
   \( a_1 + b \\
   a_2 + b \\
   a_3 + b \)

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.
   \( a + b_1 + b_2 + b_3 \)

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.

```plaintext
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_x2.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.
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 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.

```python
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 version of Python newer than Python 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 for this module is a usage guide. For a more formal description of the exposed API, please refer to the the NvRules API documentation.

4.1. Basic Usage

The module is called ncu_report and can be imported like most Python modules:

```python
>>> 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.

```python
>>> 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 inspected through the Python module, kernel profiling results are grouped in *ranges* represented by an *IRange* object. 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)`.

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

**Querying actions**

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

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

In the same way *ranges* can be obtained using their indices, individual *actions* can be obtained using the `action_by_idx(index)` method of the *IRange* object and are represented by the *IAction* class.

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

As explained 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.

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

**Querying metrics**

To get a tuple of all the metric names available within that *action* use the `metric_names()` method. This is meant to be combined with the `metric_by_name()` method which returns an *IMetric* object. The metric names are the same as the ones you can use when using the *--metrics* flag with Nsight Compute. Once you have extracted a metric from an *action*, you can obtain its value by using one of 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 the kernel was profiled on you can query the `device__attribute_display_name` metric.

```python
>>> 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.

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

### 4.2. 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.

### 4.3. 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.

```plaintext
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.
<table>
<thead>
<tr>
<th>Address</th>
<th>Source</th>
<th>Sampling Data (IR)</th>
<th>Sampling Data (No Issue)</th>
<th>Instructions Executed</th>
<th>Predicated-On Thread Instructions Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000002</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000003</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000004</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000005</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000006</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000007</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000008</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000009</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000012</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000013</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000014</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000015</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000016</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000017</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000018</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000019</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000001A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000001B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000001C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000001D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000001E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000001F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

www.nvidia.com
Nsight Compute
v2022.2.1 | 17
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

<table>
<thead>
<tr>
<th>Offset [bytes]</th>
<th>Entry</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Magic Number</td>
<td>Binary</td>
<td>NVP:0</td>
</tr>
<tr>
<td>4</td>
<td>Integer</td>
<td>Binary</td>
<td>sizeof(File Header)</td>
</tr>
<tr>
<td>8</td>
<td>File Header</td>
<td>Proto</td>
<td>Report version</td>
</tr>
<tr>
<td>8 + sizeof(File Header)</td>
<td>Block 0</td>
<td>Mixed</td>
<td>CUDA CUBIN source, profile results, session information</td>
</tr>
<tr>
<td>8 + sizeof(File Header) + sizeof(Block 0)</td>
<td>Block 1</td>
<td>Mixed</td>
<td>CUDA CUBIN source, profile results, session information</td>
</tr>
</tbody>
</table>

Table 2 Per-Block report file format

<table>
<thead>
<tr>
<th>Offset [bytes]</th>
<th>Entry</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Integer</td>
<td>Binary</td>
<td>sizeof(Block Header)</td>
</tr>
<tr>
<td>Offset [bytes]</td>
<td>Entry</td>
<td>Type</td>
<td>Value</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------</td>
<td>------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>4</td>
<td>Block Header</td>
<td>Proto</td>
<td>Number of entries per payload type, payload size</td>
</tr>
<tr>
<td>4 + sizeof(Block Header)</td>
<td>Block Payload</td>
<td>Mixed</td>
<td>Payload (CUDA CUBIN sources, profile results, session information, string table)</td>
</tr>
</tbody>
</table>

Table 3  Block payload report file format

<table>
<thead>
<tr>
<th>Offset [bytes]</th>
<th>Entry</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Integer</td>
<td>Binary</td>
<td>sizeof(Payload type 1, entry 1)</td>
</tr>
<tr>
<td>4</td>
<td>Payload type 1, entry 1</td>
<td>Proto</td>
<td></td>
</tr>
<tr>
<td>4 + sizeof(Payload type 1, entry 1)</td>
<td>Integer</td>
<td>Binary</td>
<td>sizeof(Payload type 1, entry 2)</td>
</tr>
<tr>
<td>8 + sizeof(Payload type 1, entry 1)</td>
<td>Payload type 1, entry 2</td>
<td>Proto</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Integer</td>
<td>Binary</td>
<td>sizeof(Payload type 2, entry 1)</td>
</tr>
<tr>
<td>...</td>
<td>Payload type 2, entry 1</td>
<td>Proto</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
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

© 2018-2022 NVIDIA Corporation and affiliates. All rights reserved.

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