PROFILER

DU-05982-001_v5.0 | October 2012

User's Guide

TABLE OF CONTENTS

LIST OF TABLES

PROFILING OVERVIEW

This document describes NVIDIA profiling tools and APIs that enable you to understand and optimize the performance of your CUDA application. The Visual Profiler is a graphical profiling tool that displays a timeline of your application's CPU and GPU activity, and that includes an automated analysis engine to identify optimization opportunities. The Visual Profiler is available as both a standalone application and as part of Nsight Eclipse Edition. The new nvprof profiling tool enables you to collect and view profiling data from the command-line. The existing Command Line Profiler continues to be supported.

What's New

The profiling tools contain a number of changes and new features as part of the CUDA Toolkit 5.0 release.

- A new profiling tool, `nvprof`, enables you to collect and view profiling data from the command-line. See nvprof for more information.
- The Visual Profiler now supports *kernel instance analysis* in addition to the existing application analysis. Kernel instance analysis pinpoints optimization opportunities at specific source lines within a kernel. See Analysis View for more information.
- The Visual Profiler and `nvprof` now support concurrent kernel execution. If the application uses concurrent kernel execution, the Visual Profiler timeline and `nvprof` output will show multiple kernel instances executing at the same time on the GPU.
- You can now use `cudaProfilerStart()` and `cudaProfilerStop()` to control the region(s) of your application that should be profiled. The Visual Profiler and `nvprof` will collect and display profiling results only for those regions. See Focused Profiling for more information.
- The Visual Profiler now supports NVIDIA Tools Extension. If an application uses NVTX to name resource or mark ranges, then those names and ranges will be reflected in the Visual Profiler timeline.
- In most instances, the Visual Profiler, `nvprof`, and the command-line profiler can now collect events and metrics for all CUDA contexts in a multi-context application. In previous releases, the Visual Profiler and the command-line profiler could collect events and metrics for only a single context per application.
- The Visual Profiler now shows CUDA peer-to-peer memory copies on the timeline. A peer-to-peer memory copy is reported as a single DtoD memcpy if the memcpy

is performed using the copy engine of one of the devices. A peer-to-peer memory copy is reported as a DtoH memcpy followed by an HtoD memcpy if the memcpy is performed using a staging buffer on the host.

Chapter 1. PREPARING AN APPLICATION FOR PROFILING

The CUDA profiling tools do not require any application changes to enable profiling; however, by making some simple modifications and additions, you can greatly increase the usability and effectiveness of the profilers. This section describes these modifications and how they can improve your profiling results.

1.1 Focused Profiling

By default, the profiling tools collect profile data over the entire run of your application. But, as explained below, you typically only want to profile the region(s) of your application containing some or all of the performance-critical code. Limiting profiling to performance-critical regions reduces the amount of profile data that both you and the tools must process, and focuses attention on the code where optimization will result in the greatest performance gains.

There are several common situations where profiling a region of the application is helpful.

1. The application is a test harness that contains a CUDA implementation of all or part of your algorithm. The test harness initializes the data, invokes the CUDA functions to perform the algorithm, and then checks the results for correctness. Using a test harness is a common and productive way to quickly iterate and test algorithm changes. When profiling, you want to collect profile data for the CUDA functions implementing the algorithm, but not for the test harness code that initializes the data or checks the results.
2. The application operates in phases, where a different set of algorithms is active in each phase. When the performance of each phase of the application can be optimized independently of the others, you want to profile each phase separately to focus your optimization efforts.
3. The application contains algorithms that operate over a large number of iterations, but the performance of the algorithm does not vary significantly across those iterations. In this case you can collect profile data from a subset of the iterations.

To limit profiling to a region of your application, CUDA provides functions to start and stop profile data collection. `cudaProfilerStart()` is used to start profiling and `cudaProfilerStop()` is used to stop profiling (using the CUDA driver API, you get the same functionality with `cuProfilerStart()` and `cuProfilerStop()`). To use these functions you must include `cuda_profiler_api.h` (or `cudaProfiler.h` for the driver API).

When using the start and stop functions, you also need to instruct the profiling tool to disable profiling at the start of the application. For `nvprof` you do this with the `--profile-from-start-off` flag. For the Visual Profiler you use the "Start execution with profiling enabled" checkbox in the Settings View.

1.2 Marking Regions of CPU Activity

The Visual Profiler can collect a trace of the CUDA function calls made by your application. The Visual Profiler shows these calls in the Timeline View, allowing you to see where each CPU thread in the application is invoking CUDA functions. To understand what the application's CPU threads are doing outside of CUDA function calls, you can use the NVIDIA Tools Extension (NVTX). When you add NVTX markers and ranges to your application, the Timeline View shows when your CPU threads are executing within those regions.

1.3 Naming CPU and CUDA Resources

The Visual Profiler Timeline View shows default naming for CPU thread and GPU devices, context and streams. Using custom names for these resources can improve understanding of the application behavior, especially for CUDA applications that have many host threads, devices, contexts, or streams. You can use the NVIDIA Tools Extension to assign custom names for your CPU and GPU resources. Your custom names will then be displayed in the Timeline View.

1.4 Flush Profile Data

To reduce profiling overhead, the profiling tools collect and record profile information into internal buffers. These buffers are then flushed asynchronously to disk with low priority to avoid perturbing application behavior. To avoid losing profile information that has not yet been flushed, the application being profiled should call `cudaDeviceReset()` before exiting. Doing so forces all buffered profile information to be flushed.

1.5 Dynamic Parallelism

When profiling an application that uses Dynamic Parallelism there are several limitations to the profiling tools.

- The Visual Profiler timeline does not display device-launched kernels (that is, kernels launched from within other kernels). Only kernels launched from the CPU are displayed.
- The Visual Profiler timeline does not display CUDA API calls invoked from within device-launched kernels.
- The Visual Profiler does not display detailed event, metric, and source-level results for device-launched kernels. Event, metric, and source-level results collected for CPU-launched kernels will include event, metric, and source-level results for the entire call-tree of kernels launched from within that kernel.
- The `nvprof` and command-line profiler output does not include device-launched kernels. Only kernels launched from the CPU are included in the output.
- The `nvprof` and command-line profiler event output does not include results for device-launched kernels. Events collected for CPU-launched kernels will include events for the entire call-tree of kernels launched from within that kernel.
- Concurrent kernel execution is disabled when using any of the profiling tools.

Chapter 2. VISUAL PROFILER

The NVIDIA Visual Profiler is a tool that allows you to visualize and optimize the performance of your CUDA application. The Visual Profiler displays a timeline of your application's activity on both the CPU and GPU so that you can identify opportunities for performance improvement. In addition, the Visual Profiler will analyze your application to detect potential performance bottlenecks and direct you on how to take action to eliminate or reduce those bottlenecks.

The Visual Profiler is available as both a standalone application and as part of Nsight Eclipse Edition. The standalone version of the Visual Profiler, `nvvp`, is included in the CUDA Toolkit for all supported OSes. Within Nsight Eclipse Edition, the Visual Profiler is located in the Profile Perspective and is activated when an application is run in profile mode. Nsight Ecipse Edition, `nsight`, is included in the CUDA Toolkit for Linux and Mac OSX.

2.1 Getting Started

This section describes the steps you need to take to get started with the Visual Profiler.

2.1.1 Modify Your Application For Profiling

The Visual Profiler does not require any application changes; however, by making some simple modifications and additions, you can greatly increase its usability and effectiveness. Section Preparing An Application For Profiling describes how you can focus your profiling efforts and add extra annotations to your application that will greatly improve your profiling experience.

2.1.2 Creating a Session

The first step in using the Visual Profiler to profile your application is to create a new profiling *session*. A session contains the settings, data, and results associated with your application. Sessions gives more information on working with sessions.

You can create a new session by selecting the **Profile An Application** link on the Welcome page, or by selecting **New Session** from the **File** menu. In the **Create New**

Session dialog enter the executable for your application. Optionally, you can also specify the working directory, arguments, and environment. Then press **Next**. Notice that the **Run analysis** checkbox is selected. Press **Finish**.

2.1.3 Analyzing Your Application

Because the **Run analysis** checkbox was selected when you created your session, the Visual Profiler will immediately run you application to collect the data needed for the first stage of analysis. As described in Analysis View, you can visit each analysis stage to get recommendations on performance limiting behavior in your application. For each analysis result there is a link to more detailed documentation describing what actions you can take to understand and address each potential performance problem.

2.1.4 Exploring the Timeline

After the first analysis stage completes, you will see a timeline for your application showing the CPU and GPU activity that occurred as your application executed. Read Timeline View and Properties View to learn how to explore the profiling information that is available in the timeline. Navigating the Timeline describes how you can zoom and scroll the timeline to focus on specific areas of your application.

2.1.5 Looking at the Details

In addition to the results provided in the Analysis View, you can also look at the specific metric and event values collected as part of the analysis. Metric and event values are displayed in the Details View and the Detail Graphs View. You can collect specific metric and event values that reveal how the compute kernels in your application are behaving. You collect metrics and events as described in the Details View section.

2.2 Sessions

A session contains the settings, data, and profiling results associated with your application. Each session is saved in a separate file; so you can delete, move, copy, or share a session by simply deleting, moving, copying, or sharing the session file. By convention, the file extension `.nvvp` is used for Visual Profiler session files.

There are two types of sessions: an executable session that is associated with an application that is executed and profiled from within the Visual Profiler, and an import session that is created by importing data generated by nvprof or the Command Line Profiler.

2.2.1 Executable Session

You can create a new executable session for your application by selecting the **Profile An Application** link on the **Welcome** page, or by selecting **New Session** from the **File** menu. Once a session is created, you can edit the session's settings as described in the Settings View.

You can open and save existing sessions using the open and save options in the **File** menu.

To analyze your application and to collect metric and event values, the Visual Profiler will execute your application multiple times. To get accurate profiling results, it is important that your application conform to the requirements detailed in the Application Requirements section.

2.2.2 Import Session

You create an import session from the output of `nvprof` by using the **Import Nvprof Profile...** option in the **File** menu.

You create an import session from the CSV formatted output of the command-line profiler. You import a single CSV file using the **Open** option in the **File** menu. You import one or more CSV files into a single session with the **Import CSV Profile...** option in the **File** menu. When you import multiple CSV files, their contents are combined and displayed in a single timeline.

Because an executable application is not associated with an import session, the Visual Profiler cannot execute the application to collect additional profile data. As a result, analysis can only be performed with the data that is imported. Also, the Details View will show any imported event and metrics values but new metrics and events cannot be selected and collected for the import session.

When using the Command Line Profiler to create a CSV file for import into the Visual Profiler, the following requirement must be met:

1. `COMPUTE_PROFILE_CSV` must be 1 to generate CSV formatted output.
2. `COMPUTE_PROFILE_CONFIG` must point to a file that contains gpustarttimestamp and streamid configuration parameters. The configuration file may also contain other configuration parameters, including events.

2.3 Application Requirements

To collect performance data about your application, the Visual Profiler must be able to execute your application repeatedly in a deterministic manner. Due to software and hardware limitations, it is not possible to collect all the necessary profile data in a single execution of your application. Each time your application is run, it must operate on the same data and perform the same kernel and memory copy invocations in the same order. Specifically,

- For a device, the order of context creation must be the same each time the application executes. For a multi-threaded application where each thread creates its own context(s), care must be taken to ensure that the order of those context creations is consistent across multiple runs. For example, it may be necessary to create the contexts on a single thread and then pass the contexts to the other threads. Alternatively, the NVIDIA Tools Extension can be used to provide a custom name for each context. As long as the same custom name is applied to the same context on

each execution of the application, the Visual Profiler will be able to correctly associate those contexts across multiple runs.

- For a context, the order of stream creation must be the same each time the application executes. Alternatively, the NVIDIA Tools Extension can be used to provide a custom name for each stream. As long as the same custom name is applied to the same stream on each execution of the application, the Visual Profiler will be able to correctly associate those streams across multiple runs.
- Within a stream, the order of kernel and memcpy invocations must be the same each time the application executes.

If your application behaves differently on different executions, then the analyses performed by the Visual Profiler will likely be inaccurate, and the data shown in the Details View and Detail Graphs View will be difficult to compare and interpret. The Visual Profiler can detect many instances where your application behaves differently on different executions, and it will warn you in these instances.

2.4 Profiling Limitations

Due to software and hardware restrictions, there are a couple of limitations to the profiling and analysis performed by the Visual Profiler.

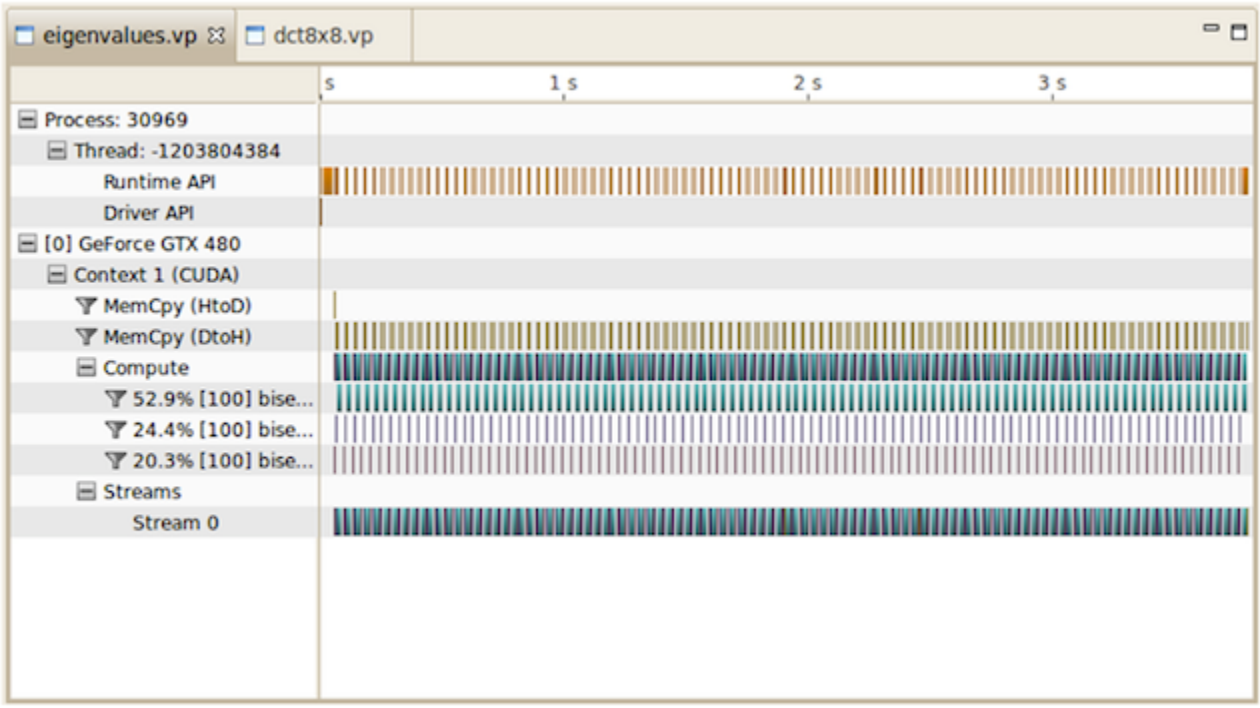
- The **Multiprocessor**, **Kernel Memory**, and **Kernel Instruction** analysis stages require metrics that are only available on devices with compute capability 2.0 or higher. When these analyses are attempted on a device with a compute capability of 1.x, the analysis results will show that the required data is "not available".
- The **Kernel Instruction** analysis stage that determines branch divergence requires a metric (Warp Execution Efficiency) that is not available on 3.0 devices. When this analysis is attempted on a device with a compute capability of 3.0 the analysis results will show that the required data is "not available".
- Some metric values are calculated assuming a kernel is large enough to occupy all device multiprocessors with approximately the same amount of work. If a kernel launch does not have this characteristic, then those metric values may not be accurate.

2.5 Visual Profiler Views

The Visual Profiler is organized into views. Together, the views allow you to analyze and visualize the performance of your application. This section describes each view and how you use it while profiling your application.

2.5.1 Timeline View

The Timeline View shows CPU and GPU activity that occurred while your application was being profiled. Multiple timelines can be opened in the Visual Profiler at the same time. Each opened timeline is represented by a different instance of the view. The name of the session file containing the timeline and related data is shown in the tab. The following figure shows a Timeline View with two open sessions, one for session **eigenvalues.vp** and the other for session **dct8x8.vp**.



Along the top of the view is a horizontal ruler that shows elapsed time from the start of application profiling. Along the left of the view is a vertical ruler that describes what is being shown for each horizontal row of the timeline, and that contains various controls for the timeline. These controls are described in Timeline Controls

The types of timeline rows that are displayed in the Timeline View are:

Process
A timeline will contain a **Process** row for each application profiled. The process identifier represents the pid of the process. The timeline row for a process does not contain any intervals of activity. Threads within the process are shown as children of the process.

Thread
A timeline will contain a **Thread** row for each thread in the profiled application that performed either a CUDA driver or runtime API call. The thread identifier is a unique id for that thread. The timeline row for a thread is does not contain any intervals of activity.

Runtime API
A timeline will contain a **Runtime API** row for each thread that performs a CUDA Runtime API call. Each interval in the row represents the duration of the call on the CPU.

Driver API
A timeline will contain a **Driver API** row for each thread that performs a CUDA Driver API call. Each interval in the row represents the duration of the call on the CPU.

Markers and Ranges
A timeline will contain a single **Markers and Ranges** row for each thread that uses the NVIDIA Tools Extension to annotate a time range or marker. Each interval in the row represents the duration of a time range, or the instantaneous point of a marker.

Profiling Overhead
A timeline will contain a single **Profiling Overhead** row for each process. Each interval in the row represents the duration of execution of some activity required for profiling. These intervals represent activity that does not occur when the application is not being profiled.

Device
A timeline will contain a **Device** row for each GPU device utilized by the application being profiled. The name of the timeline row indicates the device ID in square brackets followed by the name of the device. The timeline row for a device does not contain any intervals of activity.

Context
A timeline will contains a **Context** row for each CUDA context on a GPU device. The name of the timeline row indicates the context ID or the custom context name if the NVIDIA Tools Extension was used to name the context. The timeline row for a context does not contain any intervals of activity.

Memcpy
A timeline will contain memory copy row(s) for each context that performs memcpys. A context may contain up to three memcpy rows for device-to-host, host-to-device, and device-to-device memory copies. Each interval in a row represents the duration of a memcpy executing on the GPU.

Compute
A timeline will contain a **Compute** row for each context that performs computation on the GPU. Each interval in a row represents the duration of a kernel on the GPU device. The **Compute** row indicates all the compute activity for the context on a GPU device. The contained **Kernel** rows show activity of each individual application kernel.

Kernel
A timeline will contain a **Kernel** row for each type of kernel executed by the application. Each interval in a row represents the duration of execution of an instance of that kernel on the GPU device. Each row is labeled with a percentage that indicates the total execution time of all instances of that kernel compared to the total execution time of all kernels. Next, each row is labeled with the number of times the kernel was invoked (in square brackets), followed by the kernel name. For each context, the kernels are ordered top to bottom by execution time.

Stream
A timeline will contain a **Stream** row for each stream used by the application (including both the default stream and any application created streams). Each interval in a **Stream** row represents the duration of a memcpy or kernel execution performed on that stream.

2.5.1.1 Timeline Controls

The Timeline View has several controls that you use to control how the timeline is displayed. Some of these controls also influence the presentation of data in the Details View and the Analysis View.

Resizing the Vertical Timeline Ruler

The width of the vertical ruler can be adjusted by placing the mouse pointer over the right edge of the ruler. When the double arrow pointer appears, click and hold the left mouse button while dragging. The vertical ruler width is saved with your session.

Reordering Timelines

The **Kernel** and **Stream** timeline rows can be reordered. You may want to reorder these rows to aid in visualizing related kernels and streams, or to move unimportant kernels and streams to the bottom of the timeline. To reorder a row, left-click on the row label. When the double arrow pointer appears, drag up or down to position the row. The timeline ordering is saved with your session.

Filtering Timelines

Memcpy and **Kernel** rows can be filtered to exclude their activities from presentation in the Details View, the Detail Graphs View, and the Analysis View. To filter out a row, left-click on the filter icon just to the left of the row label. To filter all Kernel or Memcpy rows, **Shift**-left-click one of the rows. When a row is filtered, any intervals on that row are dimmed to indicate their filtered status.

Expanding and Collapsing Timelines

Groups of timeline rows can be expanded and collapsed using the **[+]** and **[-]** controls just to the left of the row labels. There are three expand/collapse states:

Collapsed
No timeline rows contained in the collapsed row are shown.

Expanded
All non-filtered timeline rows are shown.

All-Expanded
All timeline rows, filtered and non-filtered, are shown.

Intervals associated with collapsed rows are not shown in the Details View, the Detail Graphs View, and the Analysis View. For example, if you collapse a device row, then all memcpys, memsets, and kernels associated with that device are excluded from the results shown in those views.

Coloring Timelines

There are two modes for timeline coloring. The coloring mode can be selected in the **View** menu, in the timeline context menu (accessed by right-clicking in the timeline view), and on the Visual Profiler toolbar. In **kernel** coloring mode, each type of kernel is assigned a unique color (that is, all activity intervals in a kernel row have the same color). In **stream** coloring mode, each stream is assigned a unique color (that is, all memcpy and kernel activity occurring on a stream are assigned the same color).

2.5.1.2 Navigating the Timeline

The timeline can be scrolled, zoomed, and focused in several ways to help you better understand and visualize your application's performance.

Zooming

The zoom controls are available in the **View** menu, in the timeline context menu (accessed by right-clicking in the timeline view), and on the Visual Profiler toolbar. Zoom-in reduces the timespan displayed in the view, zoom-out increases the timespan displayed in the view, and zoom-to-fit scales the view so that the entire timeline is visible.

You can also zoom-in and zoom-out with the mouse wheel while holding the **Ctrl** key (for MacOSX use the **Command** key).

Another useful zoom mode is zoom-to-region. Select a region of the timeline by holding **Ctrl** (for MacOSX use the **Command** key) while left-clicking and dragging the mouse. The highlighted region will be expanded to occupy the entire view when the mouse button is released.

Scrolling

The timeline can be scrolled vertically with the scrollbar of the mouse wheel. The timeline can be scrolled horizontally with the scrollbar or by using the mouse wheel while holding the **Shift** key.

Highlighting/Correlation

When you move the mouse pointer over an activity interval on the timeline, that interval is highlighted in all places where the corresponding activity is shown. For example, if you move the mouse pointer over an interval representing a kernel execution, that kernel execution is also highlighted in the **Stream** and in the **Compute** timeline row. When a kernel or memcpy interval is highlighted, the corresponding driver or runtime API interval will also highlight. This allows you to see the correlation between the invocation of a driver or runtime API on the CPU and the corresponding activity on the GPU. Information about the highlighted interval is shown in the Properties View and the Detail Graphs View.

Selecting

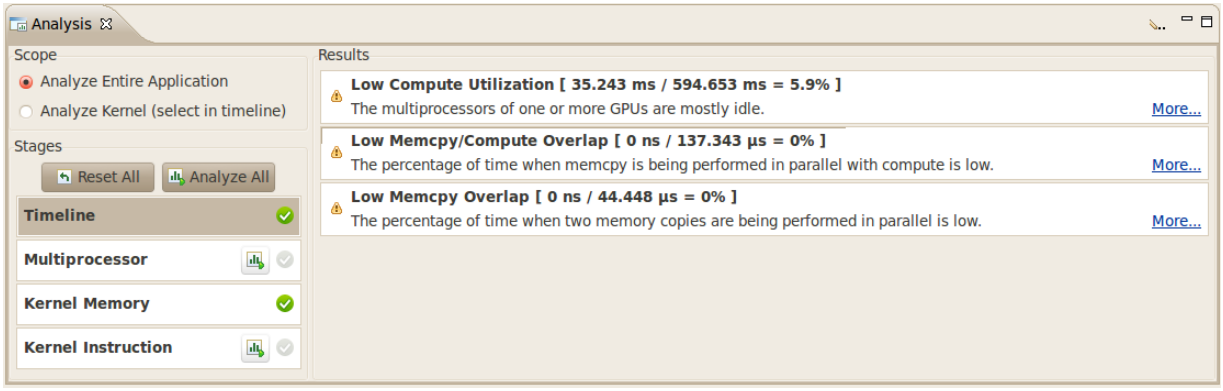
You can left-click on a timeline interval or row to select it. To unselect an interval or row simple left-click on it again. When an interval or row is selected, the information about that interval or row is pinned in the Properties View and the Detail Graphs View. In the Details View, the detailed information for the selected interval is shown in the table.

Measuring Time Deltas

Measurement rulers can be created by left-click dragging in the horizontal ruler at the top of the timeline. Once a ruler is created it can be activated and deactivated by left-clicking. Multiple rulers can be activated by **Ctrl**-left-click. Any number of rulers can be created. Active rulers are deleted with the **Delete** or **Backspace** keys. After a ruler is created, it can be resized by dragging the vertical guide lines that appear over the timeline. If the mouse is dragger over a timeline interval, the guideline will snap to the nearest edge of that interval.

2.5.2 Analysis View

The Analysis View is used to control application analysis and to display the analysis results. The left of the view shows the available analysis stages, and the right of the view shows the analysis results for that stage. The following figure shows the analysis view with the **Timeline** stage selected and the analysis results for that stage.



Analysis can be performed across the entire application, or for a single kernel instance. The type of analysis to perform is selected in the scope area of the Analysis view. When analyzing a single kernel instance, the kernel must be selected in the timeline.

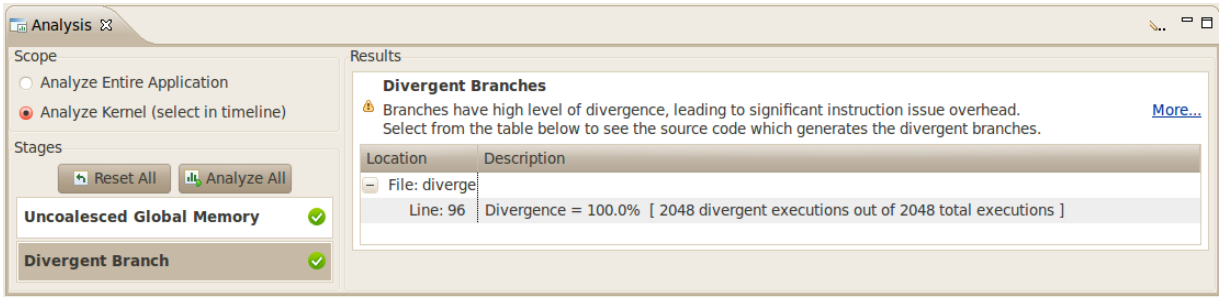
Application Analysis

Each application analysis stage has a **Run analysis** button that can be used to generate the analysis results for that stage. When the **Run analysis** button is selected, the Visual Profiler will execute the application one or more times to collect the profiling data needed to perform the analysis. The green checkmark next to an analysis stage indicates that the analysis results for that stage are available. Each analysis result contains a brief description of the analysis and a **More…** link to detailed documentation on the analysis. When you select an analysis result, the timeline rows or intervals associated with that result are highlighted in the Timeline View.

Kernel Instance Analysis

Each kernel analysis stage has a **Run analysis** button that operates in the same manner as for the application analysis stages. The following figure shows the analysis results for

the **Divergent Branch** analysis. The kernel instance analysis results are associated with specific source-lines within the kernel. To see the source associated with each result, select a Line entry from the location table. The source-file associated with that entry will open.



Other Analysis Controls

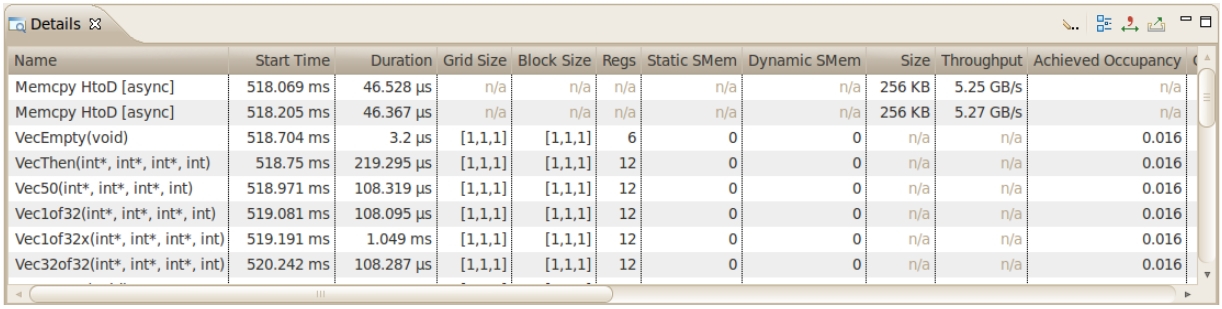
Use the **Analyze All** button to discard any existing analysis and perform all stages of analysis.

Use the **Reset All** button to discard any existing analysis results. You should reset analysis after making any changes to your application, as any existing analysis results may no longer be valid.

The analysis results shown in the Analysis View are filtered to include results that apply to visible, non-filtered timeline rows.

2.5.3 Details View

The Details View displays a table of information for each memory copy and kernel execution in the profiled application. The following figure shows the table containing several memcpy and kernel executions. Each row of the table contains general information for a kernel execution or memory copy. For kernels, the table will also contain a column for each metric or event value collected for that kernel. In the figure, the **Achieved Occupancy** column shows the value of that metric for each of the kernel executions.



Name	Start Time	Duration	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem	Size	Throughput	Achieved Occupancy
Memcpy HtoD [async]	518.069 ms	46.528 µs	n/a	n/a	n/a	n/a	n/a	256 KB	5.25 GB/s	n/a
Memcpy HtoD [async]	518.205 ms	46.367 µs	n/a	n/a	n/a	n/a	n/a	256 KB	5.27 GB/s	n/a
VecEmpty(void)	518.704 ms	3.2 µs	[1,1,1]	[1,1,1]	6	0	0	n/a	n/a	0.016
VecThen(int*, int*, int*, int)	518.75 ms	219.295 µs	[1,1,1]	[1,1,1]	12	0	0	n/a	n/a	0.016
Vec50(int*, int*, int*, int)	518.971 ms	108.319 µs	[1,1,1]	[1,1,1]	12	0	0	n/a	n/a	0.016
Vec1of32(int*, int*, int*, int)	519.081 ms	108.095 µs	[1,1,1]	[1,1,1]	12	0	0	n/a	n/a	0.016
Vec1of32x(int*, int*, int*, int)	519.191 ms	1.049 ms	[1,1,1]	[1,1,1]	12	0	0	n/a	n/a	0.016
Vec32of32(int*, int*, int*, int)	520.242 ms	108.287 µs	[1,1,1]	[1,1,1]	12	0	0	n/a	n/a	0.016

You can sort the data by a column by left clicking on the column header, and you can rearrange the columns by left clicking on a column header and dragging it to its new location. If you select a row in the table, the corresponding interval will be selected in the

Timeline View. Similarly, if you select a kernel or memcpy interval in the Timeline View the table will be scrolled to show the corresponding data.

If you hover the mouse over a column header, a tooltip will display describing the data shown in that column. For a column containing event or metric data, the tooltip will describe the corresponding event or metric. Section Metrics Reference contains more detailed information about each metric.

The detailed information shown in the Details View is filtered to include only those kernels and memory copies that are visible and non-filtered in the Timeline View. Thus, you can limit the table to show results only for the kernels and memory copies you are interested in by Timeline Controls rows in the Timeline View.

Collecting Events and Metrics

Specific event and metric values can be collected for each kernel and displayed in the details table. Use the toolbar icon in the upper right corner of the view to configure the events and metrics to collect for each device, and to run the application to collect those events and metrics.

Show Summary Data

By default the table shows one row for each memcpy and kernel invocation. Alternatively, the table can show summary results for each kernel function. Use the toolbar icon in the upper right corner of the view to select or deselect summary format.

Formatting Table Contents

The numbers in the table can be displayed either with or without grouping separators. Use the toolbar icon in the upper right corner of the view to select or deselect grouping separators.

Exporting Details

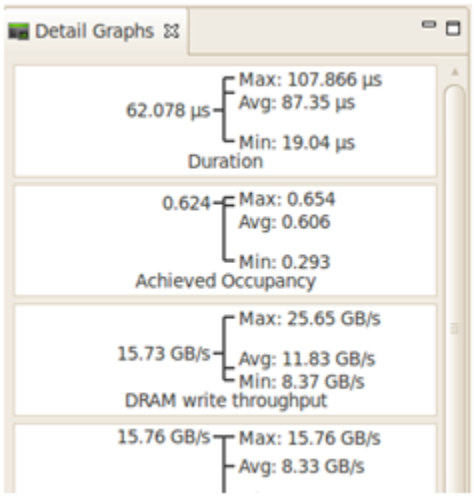
The contents of the table can be exported in CSV format using the toolbar icon in the upper right corner of the view.

2.5.4 Detail Graphs View

The Detail Graphs View shows the minimum, maximum, and average value for most of the data shown in the Details View. The graphs show information about the interval highlighted or selected in the Timeline View. If an interval is not selected, the displayed information tracks the motion of the mouse pointer. If an interval is selected, the displayed information is pinned to that interval.

If the highlighted or selected interval is a memory copy, the minimum, maximum, and average values are calculated across all memory copies currently being displayed in the Details View. If the highlighted or selected interval is a kernel, the minimum, maximum, and average values are calculated across all instances of that kernel in the timeline row containing the highlighted or selected kernel.

When a timeline interval is highlighted or selected, each graph shows the value for that specific interval. For example, in the following figure the minimum, maximum, and average durations of all kernels and memory copies shown in the Details View are 19.04µs, 107.866µs, and 87.35µs respectively. The duration of the currently highlighted or selected kernel or memory copy is 62.078µs.



2.5.5 Properties View

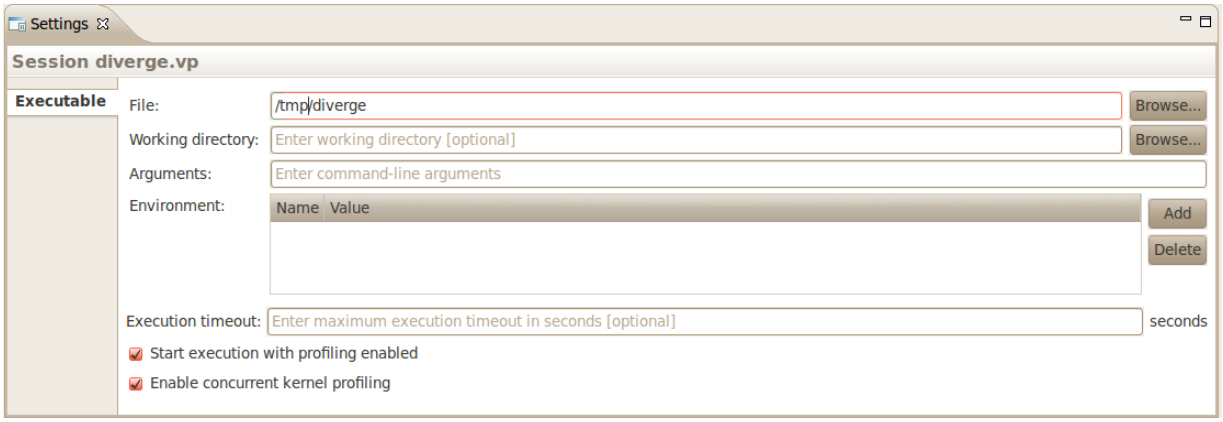
The Properties View shows information about the row or interval highlighted or selected in the Timeline View. If a row or interval is not selected, the displayed information tracks the motion of the mouse pointer. If a row or interval is selected, the displayed information is pinned to that row or interval.

2.5.6 Console View

The Console View shows the stdout and stderr output of the application each time it executes. If you need to provide stdin input to you application, you do so by typing into the console view.

2.5.7 Settings View

The Settings View allows you to specify execution settings for the application being profiled. As shown in the following figure, the **Executable** settings tab allows you to specify the executable file for the application, the working directory for the application, the command-line arguments for the application, and the environment for the application. Only the executable file is required, all other fields are optional.



Timeout

The **Executable** settings tab also allows you to specify and optional execution timeout. If the execution timeout is specified, the application execution will be terminated after that number of seconds. If the execution timeout is not specified, the application will be allowed to continue execution until it terminates normally.

Profile From Start

The **Start execution with profiling enabled** checkbox is set by default to indicate that application profiling begins at the start of application execution. If you are using `cudaProfilerStart()` and `cudaProfilerStop()` to control profiling within your application as described in Focused Profiling, then you should uncheck this box.

Concurrent Kernels

The **Enable concurrent kernel profiling** checkbox is set by default to enable profiling of applications that exploit concurrent kernel execution. If this checkbox is unset, the profiler will disable concurrent kernel execution. Disabling concurrent kernel execution can reduce profiling overhead in some cases and so may be appropriate for applications that do not exploit concurrent kernels.

2.6 Customizing the Visual Profiler

When you first start the Visual Profiler, and after closing the **Welcome** page, you will be presented with a default placement of the views. By moving and resizing the views, you can customize the Visual Profiler to meet you development needs. Any changes you make to the Visual Profiler are restored the next time you start the profiler.

2.6.1 Resizing a View

To resize a view, simply left click and drag on the dividing area between the views. All views stacked together in one area are resized at the same time. For example, in the

default view placement, the Properties View and the Detail Graphs View are resized together.

2.6.2 Reordering a View

To reorder a view in a stacked set of views, left click and drag the view tab to the new location within the view stack.

2.6.3 Moving a View

To move a view, left click the view tab and drag it to its new location. As you drag the view, an outline will show the target location for the view. You can place the view in a new location, or stack it in the same location as other views.

2.6.4 Undocking a View

You can undock a view from the Visual Profiler window so that the view occupies its own stand-alone window. You may want to do this to take advantage of multiple monitors or to maximum the size of an individual view. To undock a view, left click the view tab and drag it outside of the Visual Profiler window. To dock a view, left click the view tab (not the window decoration) and drag it into the Visual Profiler window.

2.6.5 Opening and Closing a View

Use the **X** icon on a view tab to close a view. To open a view, use the **View** menu.

Chapter 3. NVPROF

The nvprof profiling tool enables you to collect and view profiling data from the command-line. nvprof enables the collection of a timeline of your application's CPU and GPU activity: including kernel execution, memory transfers and CUDA API calls. nvprof also enables you to collect GPU hardware counter values. Profiling options are provided to nvprof through command-line options. Profiling results are displayed in the console after the application terminates, and may also be saved for later viewing by either nvprof or the Visual Profiler.

nvprof is included in the CUDA Toolkit for all supported OSes. Here's how to use nvprof to profile a CUDA application:

```
nvprof [options] [CUDA-application] [application-arguments]
```

nvprof and the Command Line Profiler are mutually exclusive profiling tools. If nvprof is invoked when the command-line profiler is enabled, nvprof will report an error and exit. To view the full help page, type nvprof --help.

3.1 Profiling Modes

nvprof operates in one of the modes listed below.

3.1.1 Summary Mode

Summary mode is the default operating mode for nvprof. In this mode, nvprof outputs a single result line for each kernel function and each type of CUDA memory copy performed by the application. For each kernel, nvprof outputs the total time of all instances of the kernel or type of memory copy as well as the average, minimum, and maximum time. Here's a simple example:

```
$ nvprof acos
======== NVPROF is profiling acos...
======== Command: acos
^^^^ elapsed = 0.00410604 sec  Gfuncs/sec=1.21772e-06
#### args: x= 2.510341934e-11 (2ddccfe0)
@@@@ total errors: 0
#### args: x= 2.646197497e-12 (2c3a35a8)
@@@@ total errors: 0
#### args: x=-2.586516325e-15 (a73a60ce)
```

```
@@@@ total errors: 0
#### args: x=-9.511874012e-23 (9ae5fba6)
@@@@ total errors: 0
#### args: x= 1.469321091e-07 (341dc464)
@@@@ total errors: 0
^^^^ ulperr: [0]=4  [1]=1  [2]=0  [3]=0
======== Profiling result:
    Time(%)      Time   Calls       Avg       Min       Max  Name
      99.89    3.03ms       1    3.03ms    3.03ms    3.03ms
 acos_main(acosParams)
       0.07    2.05us       1    2.05us    2.05us    2.05us  [CUDA memcpy DtoH]
       0.04    1.25us       1    1.25us    1.25us    1.25us  [CUDA memcpy HtoD]
```

3.1.2 GPU-Trace and API-Trace Modes

GPU-Trace and API-Trace modes can be enabled individually or at the same time. GPU-trace mode provides a timeline of all activities taking place on the GPU in chronological order. Each kernel execution and memory copy instance is shown in the output. For each kernel or memory copy detailed information such as kernel parameters, shared memory usage and memory transfer throughput are shown. Here's an example:

```
$ nvprof --print-gpu-trace acos
======== NVPROF is profiling acos...
======== Command: acos
^^^^ elapsed = 0.00208092 sec  Gfuncs/sec=2.40279e-06
#### args: x= 2.510341934e-11 (2ddccfe0)
@@@@ total errors: 0
#### args: x= 2.646197497e-12 (2c3a35a8)
@@@@ total errors: 0
#### args: x=-2.586516325e-15 (a73a60ce)
@@@@ total errors: 0
#### args: x=-9.511874012e-23 (9ae5fba6)
@@@@ total errors: 0
#### args: x= 1.469321091e-07 (341dc464)
@@@@ total errors: 0
^^^^ ulperr: [0]=4  [1]=1  [2]=0  [3]=0
======== Profiling result:
    Start  Duration           Grid Size     Block Size     Regs*    SSMem*
 DSMem*      Size  Throughput  Name
 159.92ms    2.28ms         (65520 1 1)      (256 1 1)        12        0B
  0B         -           -  acos_main(acosParams)
 160.43ms    1.25us                    -              -         -         -
   -       20B   16.03MB/s  [CUDA memcpy HtoD]
 162.61ms    2.02us                    -              -         -         -
   -       20B    9.92MB/s  [CUDA memcpy DtoH]

Regs: Number of registers used per CUDA thread.
SSMem: Static shared memory allocated per CUDA block.
DSMem: Dynamic shared memory allocated per CUDA block.
```

API-trace mode shows the timeline of all CUDA runtime and driver API calls invoked on the host in chronologinal order. Here's an example:

```
$nvprof --print-api-trace acos
======== NVPROF is profiling acos...
======== Command: acos
^^^^ elapsed = 0.00198507 sec  Gfuncs/sec=2.5188e-06
#### args: x= 2.510341934e-11 (2ddccfe0)
@@@@ total errors: 0
#### args: x= 2.646197497e-12 (2c3a35a8)
@@@@ total errors: 0
#### args: x=-2.586516325e-15 (a73a60ce)
@@@@ total errors: 0
#### args: x=-9.511874012e-23 (9ae5fba6)
@@@@ total errors: 0
```

```
#### args: x= 1.469321091e-07 (341dc464)
@@@@ total errors: 0
^^^^ ulperr: [0]=4  [1]=1  [2]=0  [3]=0
&&&& acos test PASSED
======== Profiling result:
     Start  Duration  Name
   29.62ms    3.00us  cuDeviceGetCount
   29.65ms    3.00us  cuDeviceGet
   29.66ms   37.00us  cuDeviceGetName
   29.70ms   37.00us  cuDeviceTotalMem
   29.74ms    2.00us  cuDeviceGetAttribute
   <...more output...>
  131.78ms   25.00us  cudaMalloc
  131.81ms   37.00us  cudaMemcpy
  131.85ms    2.00us  cudaGetLastError
  131.85ms    2.00us  cudaConfigureCall
  131.86ms    2.00us  cudaSetupArgument
  131.86ms   34.00us  cudaLaunch
  131.90ms    1.94ms  cudaThreadSynchronize
  133.84ms    2.00us  cudaGetLastError
  133.86ms   38.00us  cudaMemcpy
  166.83ms   62.00us  cudaFree
  166.89ms  213.00us  cudaFree
  167.11ms   37.32ms  cudaThreadExit
```

3.1.3 Event Summary Mode

An "event" corresponds to a counter value which is collected during kernel execution. To see a list of all available events on a particular NVIDIA GPU, type `nvprof --query-events`. `nvprof` is able to collect multiple events at the same time. Here's an example:

```
$ nvprof --events warps_launched,branch acos
======== NVPROF is profiling acos...
======== Command: acos
^^^^ elapsed = 0.00593686 sec  Gfuncs/sec=8.42196e-07
#### args: x= 2.510341934e-11 (2ddccfe0)
@@@@ total errors: 0
#### args: x= 2.646197497e-12 (2c3a35a8)
@@@@ total errors: 0
#### args: x=-2.586516325e-15 (a73a60ce)
@@@@ total errors: 0
#### args: x=-9.511874012e-23 (9ae5fba6)
@@@@ total errors: 0
#### args: x= 1.469321091e-07 (341dc464)
@@@@ total errors: 0
^^^^ ulperr: [0]=4  [1]=1  [2]=0  [3]=0
&&&& acos test PASSED
======== Profiling result:
    Invocations       Avg       Min       Max  Event Name
Device 0
 Kernel: acos_main(acosParams)
             1    524160    524160    524160  warps_launched
             1    524162    524162    524162  branch
Device 1
```

3.1.4 Event Trace Mode

In event trace mode, The event values are shown for each kernel execution. By default, event values are aggregated across all units in the GPU. For example, by default multiprocessor specific events are aggregated across all multiprocessors on the GPU. If `--aggregate-mode-off` is specified, values of each unit are shown. For example, in

the following example, the "branch" event value is shown for each multiprocessor on the GPU.

```
$ nvprof --events branch --print-gpu-trace --aggregate-mode-off acos
======== NVPROF is profiling acos...
======== Command: acos
^^^^ elapsed = 0.00585794 sec  Gfuncs/sec=8.53542e-07
#### args: x= 2.510341934e-11 (2ddccfe0)
@@@@ total errors: 0
#### args: x= 2.646197497e-12 (2c3a35a8)
@@@@ total errors: 0
#### args: x=-2.586516325e-15 (a73a60ce)
@@@@ total errors: 0
#### args: x=-9.511874012e-23 (9ae5fba6)
@@@@ total errors: 0
#### args: x= 1.469321091e-07 (341dc464)
@@@@ total errors: 0
^^^^ ulperr: [0]=4  [1]=1  [2]=0  [3]=0
&&&& acos test PASSED
======== Profiling result:
        Event Name, Kernel, Values
Device 0
        branch, acos_main(acosParams), 38472 37784 35944 38224 37752 36256 38328
 37344 36064 38770 37088 38944 37248 35944
```

3.2 Output

3.2.1 Adjust Units

By default, `nvprof` adjusts the time units automatically to get the most precise time values. The `--normalized-time-unit` options can be used to get fixed time units throughout the results.

3.2.2 CSV

For each profiling mode, option `--csv` can be used to generate output in comma-separated values (CSV) format. The result can be directly imported to spreadsheet software such as Excel.

3.2.3 Export/Import

For each profiling mode, option `--output-profile` can be used to generate a result file. This file is not human-readable, but can be imported to `nvprof` using the option `--import-profile`, or into the Visual Profiler.

3.2.4 Demangling

By default, `nvprof` demangles C++ function names. Use option `--no-demangling` to turn this feature off.

3.3 Profiling Controls

3.3.1 Timeout

A timeout (in seconds) can be provided to `nvprof`. The CUDA application being profiled will be killed by `nvprof` after the timeout. Profiling result collected before the timeout will be shown.

3.3.2 Concurrent Kernels

Concurrent-kernel profiling is supported. To turn the feature off, use the option `--concurrent-kernels-off`. This forces multiple kernel executions to be serialized when a CUDA application is run with `nvprof`.

3.4 Limitations

This section documents some `nvprof` limitations.

- Unlike the Visual Profiler, `nvprof` doesn't provde an option to collect metric values.
- `nvprof` only profiles the directly launched application. Any child processes spawned by that application are not profiled.

Chapter 4. COMMAND LINE PROFILER

The Command Line Profiler is a profiling tool that can be used to measure performance and find potential opportunities for optimization for CUDA applications executing on NVIDIA GPUs. The command line profiler allows users to gather timing information about kernel execution and memory transfer operations. Profiling options are controlled through environment variables and a profiler configuration file. Profiler output is generated in text files either in Key-Value-Pair (KVP) or Comma Separated (CSV) format.

4.1 Command Line Profiler Control

The command line profiler is controlled using the following environment variables:

COMPUTE_PROFILE: is set to either 1 or 0 (or unset) to enable or disable profiling.

COMPUTE_PROFILE_LOG: is set to the desired file path for profiling output. In case of multiple contexts you must add '%d' in the COMPUTE_PROFILE_LOG name. This will generate separate profiler output files for each context - with '%d' substituted by the context number. Contexts are numbered starting with zero. In case of multiple processes you must add '%p' in the COMPUTE_PROFILE_LOG name. This will generate separate profiler output files for each process - with '%p' substituted by the process id. If there is no log path specified, the profiler will log data to "cuda_profile_%d.log" in case of a CUDA context ('%d' is substituted by the context number).

COMPUTE_PROFILE_CSV: is set to either 1 (set) or 0 (unset) to enable or disable a comma separated version of the log output.

COMPUTE_PROFILE_CONFIG: is used to specify a config file for selecting profiling options and performance counters.

Configuration details are covered in a subsequent section.

The following old environment variables used for the above functionalities are still supported:

CUDA_PROFILE

CUDA_PROFILE_LOG

CUDA_PROFILE_CSV

CUDA_PROFILE_CONFIG

4.2 Command Line Profiler Default Output

Table 1 Command Line Profiler Default Columns describes the columns that are output in the profiler log by default.

Table 1 Command Line Profiler Default Columns

Column	Description
method	This is character string which gives the name of the GPU kernel or memory copy method. In case of kernels the method name is the mangled name generated by the compiler.
gputime	This column gives the execution time for the GPU kernel or memory copy method. This value is calculated as (gpuendtimestamp - gpustarttimestamp)/1000.0. The column value is a single precision floating point value in microseconds.
cputime	For non-blocking methods the cputime is only the CPU or host side overhead to launch the method. In this case: walltime = cputime + gputime For blocking methods cputime is the sum of gputime and CPU overhead. In this case: walltime = cputime Note all kernel launches by default are non-blocking. But if any of the profiler counters are enabled kernel launches are blocking. Also asynchronous memory copy requests in different streams are non-blocking. The column value is a single precision floating point value in microseconds.
occupancy	This column gives the multiprocessor occupancy which is the ratio of number of active warps to the maximum number of warps supported on a multiprocessor of the GPU. This is helpful in determining how effectively the GPU is kept busy. This column is output only for GPU kernels and the column value is a single precision floating point value in the range 0.0 to 1.0.

4.3 Command Line Profiler Configuration

The profiler configuration file is used to select the profiler options and counters which are to be collected during application execution. The configuration file is a simple format text file with one option on each line. Options can be commented out using the # character at the start of a line. Refer the command line profiler options table for the column names in the profiler output for each profiler configuration option.

4.3.1 Command Line Profiler Options

Table 2 Command Line Profiler Options contains the options supported by the command line profiler. Note the following regarding the profiler log that is produced from the different options:

- Typically, each profiler option corresponds to a single column is output. There are a few exceptions in which case multiple columns are output; these are noted where applicable in Table 2 Command Line Profiler Options.
- In most cases the column name is the same as the option name; the exceptions are listed in Table 2 Command Line Profiler Options.
- In most cases the column values are 32-bit integers in decimal format; the exceptions are listed in Table 2 Command Line Profiler Options.

Table 2 Command Line Profiler Options

Option	Description
gpustarttimestamp	Time stamp when a kernel or memory transfer starts. The column values are 64-bit unsigned value in nanoseconds in hexadecimal format.
gpuendtimestamp	Time stamp when a kernel or memory transfer completes. The column values are 64-bit unsigned value in nanoseconds in hexadecimal format.
timestamp	Time stamp when a kernel or memory transfer starts. The column values are single precision floating point value in microseconds. Use of the gpustarttimestamp column is recommended as this provides a more accurate time stamp.
gridsize	Number of blocks in a grid along the X and Y dimensions for a kernel launch. This option outputs the following two columns: ▸ gridsizeX ▸ gridsizeY
gridsize3d	Number of blocks in a grid along the X, Y and Z dimensions for a kernel launch. This option outputs the following three columns: ▸ gridsizeX ▸ gridsizeY ▸ gridsizeZ
threadblocksize	Number of threads in a block along the X, Y and Z dimensions for a kernel launch. This option outputs the following three columns: ▸ threadblocksizeX ▸ threadblocksizeY

	▸ threadblocksizeZ
dynsmemperblock	Size of dynamically allocated shared memory per block in bytes for a kernel launch. (Only CUDA)
stasmemperblock	Size of statically allocated shared memory per block in bytes for a kernel launch.
regperthread	Number of registers used per thread for a kernel launch.
memtransferdir	Memory transfer direction, a direction value of 0 is used for host to device memory copies and a value of 1 is used for device to host memory copies.
memtransfersize	Memory transfer size in bytes. This option shows the amount of memory transferred between source (host/device) to destination (host/device).
memtransferhostmemtype	Host memory type (pageable or page-locked). This option implies whether during a memory transfer, the host memory type is pageable or page-locked.
streamid	Stream Id for a kernel launch or a memory transfer.
localblocksize	This option is no longer supported and if it is selected all values in the column will be -1. This option outputs the following column: ▸ localworkgroupsize
cacheconfigrequested	Requested cache configuration option for a kernel launch: ▸ 0 CU_FUNC_CACHE_PREFER_NONE - no preference for shared memory or L1 (default) ▸ 1 CU_FUNC_CACHE_PREFER_SHARED - prefer larger shared memory and smaller L1 cache ▸ 2 CU_FUNC_CACHE_PREFER_L1 - prefer larger L1 cache and smaller shared memory ▸ 3 CU_FUNC_CACHE_PREFER_EQUAL - prefer equal sized L1 cache and shared memory
cacheconfigexecuted	Cache configuration which was used for the kernel launch. The values are same as those listed under cacheconfigrequested.

cudadevice <device_index>	This can be used to select different counters for different CUDA devices. All counters after this option are selected only for a CUDA device with index <device_index>.
	<device_index> is an integer value specifying the CUDA device index.
	Example: To select counterA for all devices, counterB for CUDA device 0 and counterC for CUDA device 1:
	`counterA`
	`cudadevice 0`
	`counterB`
	`cudadevice 1`
	`counterC`

profilelogformat [CSV|KVP]	Choose format for profiler log. ▸ CSV: Comma separated format ▸ KVP: Key Value Pair format

	The default format is KVP. This option will override the format selected using the environment variable COMPUTE_PROFILE_CSV.
countermodeaggregate	If this option is selected then aggregate counter values will be output. For a SM counter the counter value is the sum of the counter values from all SMs. For l1*, tex*, sm_cta_launched, uncached_global_load_transaction and global_store_transaction counters the counter value is collected for 1 SM from each GPC and it is extrapolated for all SMs. This option is supported only for CUDA devices with compute capability 2.0 or higher.
conckerneltrace	This option should be used to get gpu start and end timestamp values in case of concurrent kernels. Without this option execution of concurrent kernels is serialized and the timestamps are not correct. Only CUDA devices with compute capability 2.0 or higher support execution of multiple kernels concurrently. When this option is enabled additional code is inserted for each kernel and this will result in some additional execution overhead. This option cannot be used along with profiler counters. In case some counter is given in the configuration file along with "conckerneltrace" then a warning is printed in the profiler output file and the counter will not be enabled.
enableonstart 0|1	Use `enableonstart 1` option to enable or `enableonstart 0` to disable profiling from the start of application execution. If this option is not used then by default profiling is enabled from the start. To limit profiling to a region of your application, CUDA provides functions to start and stop profile data collection. `cudaProfilerStart()` is used to start profiling and `cudaProfilerStop()` is used to stop profiling (using the CUDA driver API, you get the same functionality with `cuProfilerStart()` and `cuProfilerStop()`). When using the start and stop functions, you also need to instruct the profiling tool to disable profiling at the start of the application. For command line profiler you do this by adding `enableonstart 0` in the profiler configuration file.

4.3.2 Command Line Profiler Counters

The command line profiler supports logging of event counters during kernel execution. The list of available events can be found using `nvprof --query-events` as described in Event Summary Mode. The event name can be used in the command line profiler configuration file. In every application run only a few counter values can be collected. The number of counters depends on the specific counters selected.

4.4 Command Line Profiler Output

If the `COMPUTE_PROFILE` environment variable is set to enable profiling, the profiler log records timing information for every kernel launch and memory operation performed by the driver.

Example 1: CUDA Default Profiler Log- No Options or Counters Enabled (File name: cuda_profile_0.log) shows the profiler log for a CUDA application with no profiler configuration file specified.

Example 1: CUDA Default Profiler Log- No Options or Counters Enabled (File name: `cuda_profile_0.log`)

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla C2075
# CUDA_CONTEXT 1
# TIMESTAMPFACTOR fffff6de60e24570
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 80.640 ] cputime=[ 278.000 ]
method=[ memcpyHtoD ] gputime=[ 79.552 ] cputime=[ 237.000 ]
method=[ _Z6VecAddPKfS0_Pfi ] gputime=[ 5.760 ] cputime=[ 18.000 ]
 occupancy=[ 1.000 ]
method=[ memcpyDtoH ] gputime=[ 97.472 ] cputime=[ 647.000 ]
```

The log above in Example 1: CUDA Default Profiler Log- No Options or Counters Enabled (File name: cuda_profile_0.log) shows data for memory copies and a kernel launch. The `method` label specifies the name of the memory copy method or kernel executed. The `gputime` and `cputime` labels specify the actual chip execution time and the driver execution time, respectively. Note that `gputime` and `cputime` are in microseconds. The 'occupancy' label gives the ratio of the number of active warps per multiprocessor to the maximum number of active warps for a particular kernel launch. This is the theoretical occupancy and is calculated using kernel block size, register usage and shared memory usage.

Example 2: CUDA Profiler Log- Options and Counters Enabled shows the profiler log of a CUDA application. There are a few options and counters enabled in this example using the profiler configuration file:

```
gpustarttimestamp
gridsize3d
threadblocksize
dynsmemperblock
stasmemperblock
regperthread
memtransfersize
memtransferdir
streamid
countermodeaggregate
active_warps
active_cycles
```

Example 2: CUDA Profiler Log- Options and Counters Enabled

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla C2075
# CUDA_CONTEXT 1
# TIMESTAMPFACTOR fffff6de5e08e990
gpustarttimestamp,method,gputime,cputime,gridsizeX,gridsizeY,gridsizeZ,threadblocksizeX,threa
gpustarttimestamp=[ 124b9e484b6f3f40 ] method=[ memcpyHtoD ] gputime=[ 80.800 ]
 cputime=[ 280.000 ] streamid=[ 1 ] memtransfersize=[ 200000 ]
 memtransferdir=[ 1 ]
gpustarttimestamp=[ 124b9e484b7517a0 ] method=[ memcpyHtoD ] gputime=[ 79.744 ]
 cputime=[ 232.000 ] streamid=[ 1 ] memtransfersize=[ 200000 ]
 memtransferdir=[ 1 ]
gpustarttimestamp=[ 124b9e484b8fd8e0 ] method=[ _Z6VecAddPKfS0_Pfi ]
 gputime=[ 10.016 ] cputime=[ 57.000 ] gridsize=[ 196, 1, 1 ]
 threadblocksize=[ 256, 1, 1 ] dynsmemperblock=[ 0 ] stasmemperblock=[ 0 ]
 regperthread=[ 4 ] occupancy=[ 1.000 ] streamid=[ 1 ]active_warps=[ 1545830 ]
 active_cycles=[ 40774 ]
```

```
gpustarttimestamp=[ 124b9e484bb5a2c0 ] method=[ memcpyDtoH ] gputime=[ 98.528 ]
 cputime=[ 672.000 ] streamid=[ 1 ] memtransfersize=[ 200000 ]
 memtransferdir=[ 2 ]
```

The default log syntax is easy to parse with a script, but for spreadsheet analysis it might be easier to use the comma separated format.

When `COMPUTE_PROFILE_CSV` is set to 1, this same test produces the output log shown in Example 3: CUDA Profiler Log- Options and Counters Enabled in CSV Format.

Example 3: CUDA Profiler Log- Options and Counters Enabled in CSV Format

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla C2075
# CUDA_CONTEXT 1
# CUDA_PROFILE_CSV 1
# TIMESTAMPFACTOR fffff6de5d77a1c0
gpustarttimestamp,method,gputime,cputime,gridsizeX,gridsizeY,gridsizeZ,threadblocksizeX,threa
124b9e85038d1800,memcpyHtoD,80.352,286.000,,,,,,,,,,1,,,200000,1
124b9e850392ee00,memcpyHtoD,79.776,232.000,,,,,,,,,,1,,,200000,1
124b9e8503af7460,_Z6VecAddPKfS0_Pfi,10.048,59.000,196,1,1,256,1,1,0,0,4,1.000,1,1532814,42030
```

Chapter 5. NVIDIA TOOLS EXTENSION

NVIDIA Tools Extension (NVTX) is a C-based Application Programming Interface (API) for annotating events, code ranges, and resources in your applications. Applications which integrate NVTX can use the Visual Profiler to capture and visualize these events and ranges. The NVTX API provides two core services:

1. Tracing of CPU events and time ranges.
2. Naming of OS and CUDA resources.

NVTX can be quickly integrated into an application. The sample program below shows the use of marker events, range events, and resource naming.

```
void Wait(int waitMilliseconds) {
  nvtxNameOsThread(“MAIN”);
  nvtxRangePush(__FUNCTION__);
  nvtxMark("Waiting...");
  Sleep(waitMilliseconds);
  nvtxRangePop();
}

int main(void) {
  nvtxNameOsThread("MAIN");
  nvtxRangePush(__FUNCTION__);
  Wait();
  nvtxRangePop();
}
```

5.1 NVTX API Overview

Files

The core NVTX API is defined in file nvToolsExt.h, whereas CUDA-specific extensions to the NVTX interface are defined in nvToolsExtCuda.h and nvToolsExtCudaRt.h. On Linux the NVTX shared library is called `libnvToolsExt.so` and on Mac OSX the shared library is called `libnvToolsExt.dylib`. On Windows the library (.lib) and runtime components (.dll) are named `nvToolsExt[bitness=32|64]_[version].{dll|lib}`.

Function Calls

All NVTX API functions start with an nvtx name prefix and may end with one out of the three suffixes: A, W, or Ex. NVTX functions with these suffixes exist in multiple variants, performing the same core functionality with different parameter encodings. Depending on the version of the NVTX library, available encodings may include ASCII (A), Unicode (W), or event structure (Ex).

The CUDA implementation of NVTX only implements the ASCII (A) and event structure (Ex) variants of the API, the Unicode (W) versions are not supported and have no effect when called.

Return Values

Some of the NVTX functions are defined to have return values. For example, the `nvtxRangeStart()` function returns a unique range identifier and `nvtxRangePush()` function outputs the current stack level. It is recommended not to use the returned values as part of conditional code in the instrumented application. The returned values can differ between various implementations of the NVTX library and, consequently, having added dependencies on the return values might work with one tool, but may fail with another.

5.2 NVTX API Events

Markers are used to describe events that occur at a specific time during the execution of an application, while ranges detail the time span in which they occur. This information is presented alongside all of the other captured data, which makes it easier to understand the collected information. All markers and ranges are identified by a message string. The Ex version of the marker and range APIs also allows category, color, and payload attributes to be associated with the event using the event attributes structure.

5.2.1 NVTX Markers

A marker is used to describe an instantaneous event. A marker can contain a text message or specify additional information using the Event Attributes Structure. Use `nvtxMarkA` to create a marker containing an ASCII message. Use `nvtxMarkEx()` to create a marker containing additional attributes specified by the event attribute structure. The `nvtxMarkW()` function is not supported in the CUDA implementation of NVTX and has no effect if called.

Code Example

```
nvtxMarkA("My mark");

nvtxEventAttributes_t eventAttrib = {0};
eventAttrib.version = NVTX_VERSION;
eventAttrib.size = NVTX_EVENT_ATTRIB_STRUCT_SIZE;
eventAttrib.colorType = NVTX_COLOR_ARGB;
eventAttrib.color = COLOR_RED;
```

```
    eventAttrib.messageType = NVTX_MESSAGE_TYPE_ASCII;
    eventAttrib.message.ascii = "my mark with attributes";
    nvtxMarkEx(&eventAttrib);
```

5.2.2 NVTX Range Start/Stop

A start/end range is used to denote an arbitrary, potentially non-nested, time span. The start of a range can occur on a different thread than the end of the range. A range can contain a text message or specify additional information using the Event Attributes Structure. Use `nvtxRangeStartA()` to create a marker containing an ASCII message. Use `nvtxRangeStartEx()` to create a range containing additional attributes specified by the event attribute structure. The `nvtxRangeStartW()` function is not supported in the CUDA implementation of NVTX and has no effect if called. For the correlation of a start/end pair, a unique correlation ID is created that is returned from `nvtxRangeStartA()` or `nvtxRangeStartEx()`, and is then passed into `nvtxRangeEnd()`.

Code Example

```
    // non-overlapping range
    nvtxRangeId_t id1 = nvtxRangeStartA("My range");
    nvtxRangeEnd(id1);

    nvtxEventAttributes_t eventAttrib = {0};
    eventAttrib.version = NVTX_VERSION;
    eventAttrib.size = NVTX_EVENT_ATTRIB_STRUCT_SIZE;
    eventAttrib.colorType = NVTX_COLOR_ARGB;
    eventAttrib.color = COLOR_BLUE;
    eventAttrib.messageType = NVTX_MESSAGE_TYPE_ASCII;
    eventAttrib.message.ascii = "my start/stop range";
    nvtxRangeId_t id2 = nvtxRangeStartEx(&eventAttrib);
    nvtxRangeEnd(id2);

    // overlapping ranges
    nvtxRangeId_t r1 = nvtxRangeStartA("My range 0");
    nvtxRangeId_t r2 = nvtxRangeStartA("My range 1");
    nvtxRangeEnd(r1);
    nvtxRangeEnd(r2);
```

5.2.3 NVTX Range Push/Pop

A push/pop range is used to denote nested time span. The start of a range must occur on the same thread thread as the end of the range. A range can contain a text message or specify additional information using the Event Attributes Structure. Use `nvtxRangePushA()` to create a marker containing an ASCII message. Use `nvtxRangePushEx()` to create a range containing additional attributes specified by the event attribute structure. The `nvtxRangePushW()` function is not supported in the CUDA implementation of NVTX and has no effect if called. Each push function returns the zero-based depth of the range being started. The `nvtxRangePop()` function is used to end the most recently pushed range for the thread. `nvtxRangePop()` returns the zero-based depth of the range being ended. If the pop does not have a matching push, a negative value is returned to indicate an error.

Code Example

```
nvtxRangePushA("outer");
nvtxRangePushA("inner");
nvtxRangePop();  // end "inner" range
nvtxRangePop();  // end "outer" range

nvtxEventAttributes_t eventAttrib = {0};
eventAttrib.version = NVTX_VERSION;
eventAttrib.size = NVTX_EVENT_ATTRIB_STRUCT_SIZE;
eventAttrib.colorType = NVTX_COLOR_ARGB;
eventAttrib.color = COLOR_GREEN;
eventAttrib.messageType = NVTX_MESSAGE_TYPE_ASCII;
eventAttrib.message.ascii = "my push/pop range";
nvtxRangePushEx(&eventAttrib);
nvtxRangePop();
```

5.2.4 Event Attributes Structure

The events attributes structure, `nvtxEventAttributes_t`, is used to describe the attributes of an event. The layout of the structure is defined by a specific version of NVTX and can change between different versions of the Tools Extension library.

Attributes

Markers and ranges can use attributes to provide additional information for an event or to guide the tool's visualization of the data. Each of the attributes is optional and if left unspecified, the attributes fall back to a default value.

Message
The message field can be used to specify an optional string. The caller must set both the `messageType` and `message` fields. The default value is `NVTX_MESSAGE_UNKNOWN`. The CUDA implementation of NVTX only supports ASCII type messages.

Category
The category attribute is a user-controlled ID that can be used to group events. The tool may use category IDs to improve filtering, or for grouping events. The default value is 0.

Color
The color attribute is used to help visually identify events in the tool. The caller must set both the `colorType` and `color` fields.

Payload
The payload attribute can be used to provide additional data for markers and ranges. Range events can only specify values at the beginning of a range. The caller must specify valid values for both the `payloadType` and `payload` fields.

Initialization

The caller should always perform the following three tasks when using attributes:

- Zero the structure

- Set the version field
- Set the size field

Zeroing the structure sets all the event attributes types and values to the default value. The version and size field are used by NVTX to handle multiple versions of the attributes structure.

It is recommended that the caller use the following method to initialize the event attributes structure.

```
nvtxEventAttributes_t eventAttrib = {0};
eventAttrib.version = NVTX_VERSION;
eventAttrib.size = NVTX_EVENT_ATTRIB_STRUCT_SIZE;
eventAttrib.colorType = NVTX_COLOR_ARGB;
eventAttrib.color = ::COLOR_YELLOW;
eventAttrib.messageType = NVTX_MESSAGE_TYPE_ASCII;
eventAttrib.message.ascii = "My event";
nvtxMarkEx(&eventAttrib);
```

5.3 NVTX Resource Naming

NVTX resource naming allows custom names to be associated with host OS threads and CUDA resources such as devices, contexts, and streams. The names assigned using NVTX are displayed by the Visual Profiler.

OS Thread

The `nvtxNameOsThreadA()` function is used to name a host OS thread. The `nvtxNameOsThreadW()` function is not supported in the CUDA implementation of NVTX and has no effect if called. The following example shows how the current host OS thread can be named.

```
// Windows
nvtxNameOsThread(GetCurrentThreadId(), "MAIN_THREAD");

// Linux/Mac
nvtxNameOsThread(pthread_self(), "MAIN_THREAD");
```

CUDA Runtime Resources

The `nvtxNameCudaDeviceA()` and `nvtxNameCudaStreamA()` functions are used to name CUDA device and stream objects, respectively. The `nvtxNameCudaDeviceW()` and `nvtxNameCudaStreamW()` functions are not supported in the CUDA implementation of NVTX and have no effect if called. The `nvtxNameCudaEventA()` and `nvtxNameCudaEventW()` functions are also not supported. The following example shows how a CUDA device and stream can be named.

```
nvtxNameCudaDeviceA(0, "my cuda device 0");

cudaStream_t cudastream;
cudaStreamCreate(&cudastream);
```

```
nvtxNameCudaStreamA(cudastream, "my cuda stream");
```

CUDA Driver Resources

The `nvtxNameCuDeviceA()`, `nvtxNameCuContextA()` and `nvtxNameCuStreamA()` functions are used to name CUDA driver device, context and stream objects, respectively. The `nvtxNameCuDeviceW()`, `nvtxNameCuContextW()` and `nvtxNameCuStreamW()` functions are not supported in the CUDA implementation of NVTX and have no effect if called. The `nvtxNameCuEventA()` and `nvtxNameCuEventW()` functions are also not supported. The following example shows how a CUDA device, context and stream can be named.

```
CUdevice device;
cuDeviceGet(&device, 0);
nvtxNameCuDeviceA(device, "my device 0");

CUcontext context;
cuCtxCreate(&context, 0, device);
nvtxNameCuContextA(context, "my context");

cuStream stream;
cuStreamCreate(&stream, 0);
nvtxNameCuStreamA(stream, "my stream");
```

Chapter 6. MPI PROFILING

The nvprof profiler and the Command Line Profiler can be used to profile individual MPI processes. The resulting output can be used directly, or can be imported into the Visual Profiler.

6.1 MPI Profiling With nvprof

To use nvprof to collect the profiles of the individual MPI processes, you must tell `nvprof` to send its output to specific files based on the rank of the MPI job. To do this, modify `mpirun` to launch a script which in turn launches `nvprof` and the MPI process. Below is an example script for OpenMPI and MVAPICH2.

```
#!/bin/sh
#
# Script to launch nvprof on an MPI process.  This script will
# create unique output file names based on the rank of the
# process.  Examples:
#   mpirun -np 4 nvprof-script a.out
#   mpirun -np 4 nvprof-script -o outfile a.out
#   mpirun -np 4 nvprof-script test/a.out -g -j
# In the case you want to pass a -o or -h flag to the a.out, you
# can do this.
#   mpirun -np 4 nvprof-script -c a.out -h -o
# You can also pass in arguments to nvprof
#   mpirun -np 4 nvprof-script --print-api-trace a.out
#

usage () {
 echo "nvprof-script [nvprof options] [-h] [-o outfile] a.out [a.out options]";
 echo "or"
 echo "nvprof-script [nvprof options] [-h] [-o outfile] -c a.out [a.out
 options]";
}

nvprof_args=""
while [ $# -gt 0 ];
do
    case "$1" in
        (-o) shift; outfile="$1";;
        (-c) shift; break;;
        (-h) usage; exit 1;;
        (*) nvprof_args="$nvprof_args $1";;
    esac
```

```
    shift
done

# If user did not provide output filename then create one
if [ -z $outfile ] ; then
    outfile=`basename $1`.nvprof-out
fi

# Find the rank of the process from the MPI rank environment variable
# to ensure unique output filenames.  The script handles Open MPI
# and MVAPICH.  If your implementation is different, you will need to
# make a change here.

# Open MPI
if [ ! -z ${OMPI_COMM_WORLD_RANK} ] ; then
    rank=${OMPI_COMM_WORLD_RANK}
fi
# MVAPICH
if [ ! -z ${MV2_COMM_WORLD_RANK} ] ; then
    rank=${MV2_COMM_WORLD_RANK}
fi

# Set the nvprof command and arguments.
NVPROF="nvprof --output-profile $outfile.$rank $nvprof_args"
exec $NVPROF $*

# If you want to limit which ranks get profiled, do something like
# this. You have to use the -c switch to get the right behavior.
# mpirun -np 2 nvprof-script --print-api-trace -c a.out -q
# if [ $rank -le 0 ]; then
#     exec $NVPROF $*
# else
#     exec $*
# fi
```

6.2 MPI Profiling With The Command-Line Profiler

The Command Line Profiler is enabled and controlled by environment variables and a configuration file. To correctly profile MPI jobs, the profile output produced by the command-line profiler must be directed to unique output files for each MPI process. The command-line profiler uses the COMPUTE_PROFILE_LOG environment variable for this purpose. You can use special substitute characters in the log name to ensure that different devices and processes record their profile information to different files. The '%d' is replaced by the device ID, and the '%p' is replaced by the process ID.

```
setenv COMPUTE_PROFILE_LOG cuda_profile.%d.%p
```

If you are running on multiple nodes, you will need to store the profile logs locally, so that processes with the same ID running on different nodes don't clobber each others log file.

```
setenv COMPUTE_PROFILE_LOG /tmp/cuda_profile.%d.%p
```

COMPUTE_PROFILE_LOG and the other command-line profiler environment variables must get passed to the remote processes of the job. Most `mpiruns` have a way to do

this. Examples for Open MPI and MVAPICH2 are shown below using the simpleMPI program from the CUDA Software Development Toolkit.

Open MPI

```
> setenv COMPUTE_PROFILE_LOG /tmp/cuda_profile.%d.%p
> setenv COMPUTE_PROFILE_CSV 1
> setenv COMPUTE_PROFILE_CONFIG /tmp/compute_profile.config
> setenv COMPUTE_PROFILE 1
> mpirun -x COMPUTE_PROFILE_CSV -x COMPUTE_PROFILE -x COMPUTE_PROFILE_CONFIG -x
 COMPUTE_PROFILE_LOG -np 6 -host c0-5,c0-6,c0-7 simpleMPI
Running on 6 nodes
Average of square roots is: 0.667282
PASSED
```

MVAPICH2

```
> mpirun_rsh -np 6 c0-5 c0-5 c0-6 c0-6 c0-7 c0-7 COMPUTE_PROFILE_CSV=1
 COMPUTE_PROFILE=1 COMPUTE_PROFILE_CONFIG=/tmp/compute_profile.config
 COMPUTE_PROFILE_LOG=cuda_profile.%d.%p simpleMPI
Running on 6 nodes
Average of square roots is: 0.667282
PASSED
```

Chapter 7. METRICS REFERENCE

This section contains detailed descriptions of the metrics that can be collected by the Visual Profiler. These metrics can be collected only from within the Visual Profiler. The command-line profiler and `nvprof` can collect low-level events but are not capable of collecting metrics.

Devices with compute capability less than 2.0 implement the metrics shown in the following table.

Table 3 Capability 1.x Metrics

Metric Name	Description	Formula
branch_efficiency	Ratio of non-divergent branches to total branches	100 * (branch - divergent_branch) / branch
gld_efficiency	Ratio of requested global memory load transactions to actual global memory load transactions	For CC 1.2 & 1.3: (gld_request / ((gld_32 + gld_64 + gld_128) / (2 * #SM))) For CC 1.0 & 1.1: gld_coherent / (gld_coherent + gld_incoherent)
gst_efficiency	Ratio of requested global memory store transactions to actual global memory store transactions	For CC 1.2 & 1.3: (gst_request / ((gst_32 + gst_64 + gst_128) / (2 * #SM))) For CC 1.0 & 1.1: gst_coherent / (gst_coherent + gst_incoherent)
gld_requested_throughput	Requested global memory load throughput	(gld_32 * 32 + gld_64 * 64 + gld_128 * 128) / gputime
gst_requested_throughput	Requested global memory store throughput	(gst_32 * 32 + gst_64 * 64 + gst_128 * 128) / gputime

Devices with compute capability between 2.0, inclusive, and 3.0 implement the metrics shown in the following table.

Table 4 Capability 2.x Metrics

Metric Name	Description	Formula
sm_efficiency	The ratio of the time at least one warp is active on a multiprocessor to the total time	100 * (active_cycles / #SM) / elapsed_clocks
achieved_occupancy	Ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor	100 * (active_warps / active_cycles) / max_warps_per_sm
ipc	Instructions executed per cycle	(inst_executed / #SM) / elapsed_clocks
branch_efficiency	Ratio of non-divergent branches to total branches	100 * (branch - divergent_branch) / branch
warp_execution_efficiency	Ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor	thread_inst_executed / (inst_executed * warp_size)
inst_replay_overhead	Percentage of instruction issues due to memory replays	100 * (inst_issued - inst_executed) / inst_issued
shared_replay_overhead	Percentage of instruction issues due to replays for shared memory conflicts	100 * l1_shared_bank_conflict / inst_issue
global_cache_replay_overhead	Percentage of instruction issues due to replays for global memory cache misses	100 * global_load_miss / inst_issued
local_replay_overhead	Percentage of instruction issues due to replays for local memory cache misses	100 * (local_load_miss + local_store_miss) / inst_issued
gld_efficiency	Ratio of requested global memory load throughput to actual global memory load throughput	100 * gld_requested_throughput/ gld_throughput

Metric Name	Description	Formula
gst_efficiency	Ratio of requested global memory store throughput to actual global memory store throughput	100 * gst_requested_throughput / gst_throughput
gld_throughput	Global memory load throughput	((128 * global_load_hit) + (l2_subp0_read_requests + l2_subp1_read_requests) * 32 - (l1_local_ld_miss * 128)) / gputime
gst_throughput	Global memory store throughput	(l2_subp0_write_requests + l2_subp1_write_requests) * 32 - (l1_local_ld_miss * 128)) / gputime
gld_requested_throughput	Requested global memory load throughput	(gld_inst_8bit + 2 * gld_inst_16bit + 4 * gld_inst_32bit + 8 * gld_inst_64bit + 16 * gld_inst_128bit) / gputime
gst_requested_throughput	Requested global memory store throughput	(gst_inst_8bit + 2 * gst_inst_16bit + 4 * gst_inst_32bit + 8 * gst_inst_64bit + 16 * gst_inst_128bit) / gputime
dram_read_throughput	DRAM read throughput	(fb_subp0_read + fb_subp1_read) * 32 / gputime
dram_write_throughput	DRAM write throughput	(fb_subp0_write + fb_subp1_write) * 32 / gputime
l1_cache_global_hit_rate	Hit rate in L1 cache for global loads	100 * l1_global_ld_hit / (l1_global_ld_hit + l1_global_ld_miss)
l1_cache_local_hit_rate	Hit rate in L1 cache for local loads and stores	100 * (l1_local_ld_hit + l1_local_st_hit)/(l1_local_ld_hit + l1_local_ld_miss + l1_local_st_hit + l1_local_st_miss)
tex_cache_hit_rate	Texture cache hit rate	100 * (tex0_cache_sector_queries - tex0_cache_misses) / tex0_cache_sector_queries

Metric Name	Description	Formula
tex_cache_throughput	Texture cache throughput	tex_cache_sector_queries * 32 / gputime
sm_efficiency_instance	The ratio of the time at least one warp is active on a multiprocessor to the total time	100 * active_cycles / elapsed_clocks
ipc_instance	Instructions executed per cycle	inst_executed / elapsed_clocks
l2_l1_read_hit_rate	Hitrate at L2 cache for read requests from L1 cache	100 * (l2_subp0_read_hit_sectors + l2_subp1_read_hit_sectors) / (l2_subp0_read_sector_queries + l2_subp1_read_sector_queries)
l2_tex_read_hit_rate	Hitrate at L2 cache for read requests from texture cache	100 * (l2_subp0_read_tex_hit_sectors + l2_subp1_read_tex_hit_sectors) / (l2_subp0_read_tex_sector_queries + l2_subp1_read_tex_sector_queries)
l2_l1_read_throughput	Memory read throughput at L2 cache for read requests from L1 cache	(l2_subp0_read_sector_queries + l2_subp1_read_sector_queries) * 32 / gputime
l2_tex_read_throughput	Memory read throughput at L2 cache for read requests from texture cache	(l2_subp0_read_tex_sector_queries + l2_subp1_read_tex_sector_queries) * 32 / gputime
local_memory_overhead	Ratio of local memory traffic to total memory traffic between L1 and L2	100 * (2 * l1_local_load_miss * 128) / ((l2_subp0_read_requests + l2_subp1_read_requests +l2_subp0_write_requests + l2_subp1_write_requests) * 32)

Devices with compute capability greater than or equal to 3.0 implement the metrics shown in the following table.

Table 5 Capability 3.x Metrics

Metric Name	Description	Formula
sm_efficiency	The ratio of the time at least one warp is active on a multiprocessor to the total time	100 * (active_cycles / #SM) / elapsed_clocks

Metric Name	Description	Formula
achieved_occupancy	Ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor	100 * (active_warps / active_cycles) / max_warps_per_sm
ipc	Instructions executed per cycle	(inst_executed / #SM) / elapsed_clocks
branch_efficiency	Ratio of non-divergent branches to total branches	100 * (branch - divergent_branch) / branch
warp_execution_efficiency	Ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor (not available for compute capability 3.0)	thread_inst_executed / (inst_executed * warp_size)
inst_replay_overhead	Percentage of instruction issues due to memory replays	100 * ((inst_issued_1 + inst_issued_2 * 2) - inst_executed) / (inst_issued_1 + inst_issued_2 * 2)
shared_replay_overhead	Percentage of instruction issues due to replays for shared memory conflicts	100 * (shared_load_bank_conflict + shared_store_bank_conflict) / (inst_issued_1 + inst_issued_2 * 2)
global_replay_overhead	Percentage of instruction issues due to replays for non-coherent global memory accesses	100 * (global_ld_mem_divergence_replays + global_st_mem_divergence_replays) / (inst_issued_1 + inst_issued_2 * 2)
global_cache_replay_overhead	Percentage of instruction issues due to replays for global memory cache misses	100 * global_load_miss / (inst_issued_1 + inst_issued_2 * 2)
local_replay_overhead	Percentage of instruction issues due to replays for local memory cache misses	100 * (local_load_miss + local_store_miss) / (inst_issued_1 + inst_issued_2 * 2)

Metric Name	Description	Formula
gld_efficiency	Ratio of requested global memory load throughput to actual global memory load throughput	100 * gld_requested_throughput / gld_throughput
gst_efficiency	Ratio of requested global memory store throughput to actual global memory store throughput	100 * gst_requested_throughput / gst_throughput
shared_efficiency	Ratio of shared memory loads and stores executed to shared memory transactions required for those loads and stores	100 * (shared_load + shared_store) / (shared_ld_transactions + shared_st_transactions)
gld_throughput	Global memory load throughput	((128 * global_load_hit) + (l2_subp0_read_requests + l2_subp1_read_requests + l2_subp2_read_requests + l2_subp3_read_requests) * 32 - (l1_local_ld_miss * 128)) / gputime
gst_throughput	Global memory store throughput	(l2_subp0_write_requests + l2_subp1_write_requests + l2_subp2_write_requests + l2_subp3_write_requests) * 32 - (l1_local_ld_miss * 128)) / gputime
gld_requested_throughput	Requested global memory load throughput	(gld_inst_8bit + 2 * gld_inst_16bit + 4 * gld_inst_32bit + 8 * gld_inst_64bit + 16 * gld_inst_128bit) / gputime
gst_requested_throughput	Requested global memory store throughput	(gst_inst_8bit + 2 * gst_inst_16bit + 4 * gst_inst_32bit + 8 * gst_inst_64bit + 16 * gst_inst_128bit) / gputime
nc_gld_requested_throughput	Requested throughput for global memory loaded via non-coherent texture cache (not available for compute capability 3.0)	(ldg_inst_8bit + 2 * ldg_inst_16bit + 4 * ldg_inst_32bit + 8 * ldg_inst_64bit + 16 * ldg_inst_128bit) / gputime

Metric Name	Description	Formula
dram_read_throughput	DRAM read throughput	(fb_subp0_read + fb_subp1_read) * 32 / gputime
dram_write_throughput	DRAM write throughput	(fb_subp0_write + fb_subp1_write) * 32 / gputime
l1_cache_global_hit_rate	Hit rate in L1 cache for global loads	100 * l1_global_ld_hit / (l1_global_ld_hit + l1_global_ld_miss)
l1_cache_local_hit_rate	Hit rate in L1 cache for local loads and stores	100 * (l1_local_ld_hit + l1_local_st_hit) / (l1_local_ld_hit + l1_local_ld_miss + l1_local_st_hit + l1_local_st_miss)
tex_cache_hit_rate	Texture cache hit rate	100 * (tex0_cache_sector_queries + tex1_cache_sector_queries + tex2_cache_sector_queries + tex3_cache_sector_queries - tex0_cache_misses - tex1_cache_misses - tex2_cache_misses - tex3_cache_misses) / (tex0_cache_sector_queries + tex1_cache_sector_queries + tex2_cache_sector_queries + tex3_cache_sector_queries)
tex_cache_throughput	Texture cache throughput	(tex0_cache_sector_queries + tex1_cache_sector_queries + tex2_cache_sector_queries + tex3_cache_sector_queries) * 32 / gputime
sm_efficiency_instance	The ratio of the time at least one warp is active on a multiprocessor to the total time	100 * active_cycles / elapsed_clocks
ipc_instance	Instructions executed per cycle	inst_executed / elapsed_clocks
l2_l1_read_hit_rate	Hitrate at L2 cache for read requests from L1 cache	100 * (l2_subp0_read_hit_sectors + l2_subp1_read_hit_sectors + l2_subp2_read_hit_sectors + l2_subp3_read_hit_sectors) / (l2_subp0_read_sector_queries +

Metric Name	Description	Formula
		l2_subp1_read_sector_queries + l2_subp2_read_sector_queries + l2_subp3_read_sector_queries)
l2_tex_read_hit_rate	Hitrate at L2 cache for read requests from texture cache	100 * (l2_subp0_read_tex_hit_sectors + l2_subp1_read_tex_hit_sectors + l2_subp2_read_tex_hit_sectors + l2_subp3_read_tex_hit_sectors) / (l2_subp0_read_tex_sector_queries + l2_subp1_read_tex_sector_queries + l2_subp2_read_tex_sector_queries + l2_subp3_read_tex_sector_queries)
l2_l1_read_throughput	Memory read throughput at L2 cache for read requests from L1 cache	(l2_subp0_read_sector_queries + l2_subp1_read_sector_queries + l2_subp2_read_sector_queries + l2_subp3_read_sector_queries) * 32 / gputime
l2_tex_read_throughput	Memory read throughput at L2 cache for read requests from texture cache	(l2_subp0_read_tex_sector_queries + l2_subp1_read_tex_sector_queries + l2_subp2_read_tex_sector_queries + l2_subp3_read_tex_sector_queries) * 32 / gputime
local_memory_overhead	Ratio of local memory traffic to total memory traffic between L1 and L2	100 * (2 * l1_local_load_miss * 128) / ((l2_subp0_read_requests + l2_subp1_read_requests +l2_subp2_read_requests + l2_subp3_read_requests + l2_subp0_write_requests + l2_subp1_write_requests + l2_subp2_write_requests + l2_subp3_write_requests) * 32)