# COMPUTE VISUAL PROFILER

DU-05162-001_v04 | May 2011

**User Guide**

# DOCUMENT CHANGE HISTORY

DU-05162-001_v04

| Version | Date | Authors | Description of Change |
|---|---|---|---|
| 01 | August 26, 2010 | SS, RG, AL, RP, VS | Initial release. |
| 02 | October 29, 2010 | SS, VS | Corrected path (note the additional 3.2) specified in section entitled, "Running the Compute Visual Profiler". |
| 03 | March 05, 2011 | SR, SS | Added Kernel Analysis feature.<br>Updated current features to incorporate new UI. |
| 04 | May 3, 2011 | SS, SR, VS | Added the command line profile document, "Compute_Profiler.txt" to this User Guide as a separate chapter entitled, "Command Line Profiler".<br>Note: Compute_Profiler.txt will not be supported as a separate document.<br>• Added description of the new analysis feature under a separate chapter entitled, "Compute Application Analysis". |

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# OVERVIEW

This document is intended for users of Compute Visual Profiler for NVIDIA® CUDA™ technology. Compute Visual Profiler is a graphical user interface based profiling tool that can be used to measure performance and find potential opportunities for optimization in order to achieve maximum performance from NVIDIA® GPUs.

Compute Visual Profiler provides metrics in the form of plots and counter values presented in tables and as graphs. It tracks events with hardware counters on signals in the chip; this is explained in detail in the chapter entitled, "Compute Visual Profiler Counters." This document should be used in conjunction with the *CUDA C Programming Best Practices Guide* from NVIDIA.

Note that in CUDA version 3.1 onwards, NVIDIA's CUDA Visual Profiler and OpenCL Visual Profiler have been integrated into a single application called–Compute Visual Profiler.

Compute Visual Profiler 4.0 provides a new analysis feature that provides performance analysis of the application based on the profiling data. The feature also provides various optimization hints to improve application performance. This is described in further detail in the section entitled, "Compute Application Analysis".

## GETTING STARTED

In order to run Compute Visual Profiler you need the following:

- CUDA compatible NVIDIA graphics card
- NVIDIA CUDA Toolkit, and
- NVIDIA Display Driver (latest version)

Refer to the *Getting Started Guide* for your operating system for help with installation.

## Installation and Setup

### Windows

If you do not have Microsoft Visual C++ 2008 or Microsoft Visual C++ 2008 Redistributable Package installed you will need to install the Microsoft Visual C++ 2008 Redistributable Package by running **`vcredist_x86.exe`** available under the "**`<CudaToolkitDir>\computeprof\bin`**" directory.

Note that if the correct versions of Microsoft Visual C++ DLLs are not available when you run Compute Visual Profiler, the following error is displayed:

```
Application failed to start because side-by-side configuration is
incorrect.
```

### Linux

The installation is part of the CUDA toolkit installation. The files are installed under "**`<CudaToolkitDir>/computeprof`**" where **`<CudaToolkitDir>`** is the directory under which the CUDA Toolkit is installed.

Setup **`LD_LIBRARY PATH`** to include the ComputeVisualProfiler bin directory:

```
> export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:<CudaToolkitDir>/computeprof/bin
```

### MacOS X

The installation is part of the CUDA toolkit installation. The files are installed under "**`<CudaToolkitDir>\computeprof`**" where **`<CudaToolkitDir>`** is the directory under which the CUDA Toolkit is installed.

## Running the Compute Visual Profiler

### Windows

To run the Compute Visual Profiler, go to:

**Start→All Programs→NVIDIA Corporation→CUDA Toolkit→3.2→Compute Visual Profiler**

Sample pre-run projects and online help are available as follows:

- Directory containing sample Compute Visual Profiler projects:
  **`<CudaToolkitDir>\computeprof\projects`**
- Directory containing files for online help and user documentation:
  **`<CudaToolkitDir>\computeprof\doc`**

### Linux

> **`<CudaToolkitDir>/computeprof/bin/computeprof &`**

### MacOS X

From Finder run:
"**`<CudaToolkitDir>\computeprof\computeprof.app`**"

### Xterm

From Terminal run:
"**`<CudaToolkitDir>\computeprof\computeprof.app\Contents\MacOS\computeprof`**"

When Visual Profiler is launched, a welcome screen (shown in Figure 1) is displayed.

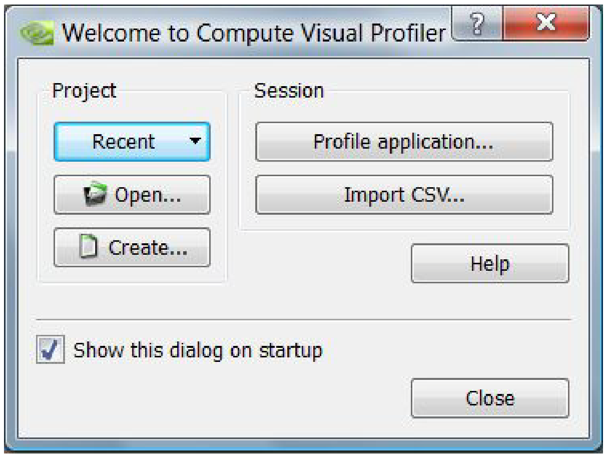


Figure 1. Welcome Screen

This dialog box allows first time users to navigate Visual Profiler more effectively. It allows you to:

- Open recently saved projects.
- Open projects that are **`<CudaToolkitDir>\computeprof\projects`** folder
- Import previously saved .csv data from previous profiler runs

Click on **Help** to go directly to online help information about Compute Visual Profiler.

For future launches of the Compute Visual Profiler, this dialog box may be disabled by un-checking the "**Show this dialog on startup**" check box.

## CUDA™ AND OPENCL™ SUPPORT

The Compute Visual Profiler supports profiling for both NVIDIA CUDA™ and OpenCL™ applications. The Session settings dialog box shows options in the CUDA terminology. Most of the options are common and supported for both CUDA and OpenCL except for the following:

- **dynsmemperblock**
  The kernel option 'dynsmemperblock' is supported only for CUDA. The warning **NV_Warning: Ignoring the invalid profiler config option: dynsmemperblock** is displayed after each profiling run if this option is selected for OpenCL.
- **localworkgroupsize**
  The kernel option 'localworkgroupsize' is valid only for OpenCL. If this option is selected for a CUDA program a column 'localblocksize' is added to the Profiler Table, but this column is hidden by default.

The type of session—CUDA or OPENCL, is shown within square parentheses after the session name:
**Context_0 [CUDA] or Context_1 [OPENCL]**
The column names in the Profiler Table or the summary table for a context are displayed based on the compute language for the context. In the CUDA context, CUDA terminology is used and in the OpenCL context OpenCL terminology is used.

A project can consist of sessions containing a mix of CUDA program profiling sessions and OpenCL program profiling sessions. To distinguish such projects from old projects, a new project file extension **.cvp** is used. Support for old projects is provided and you can open an old CUDA project (file extension '.cpj') or an old OpenCL project (file extension '.oclpj'). However, when you save existing .cpj or .oclpj projects the old projects are saved in the new format with the .cvp file extension.

Table 1 shows how CUDA terminology maps to OpenCL™ terminology:

Table 1. NVIDIA® CUDA™ and OpenCL™ Terminology

| CUDA Term | OpenCL™ Term |
|---|---|
| Thread | Work-item |
| Thread block or CTA (Cooperative Thread Array) | Work-group |
| Grid size | Range size |
| Shared memory | Local memory |
| Local memory | Private memory |

## COMPUTE VISUAL PROFILER FILES AND SETTINGS

Profiling is automatically enabled by Compute Visual Profiler at the start of the application. For a long running application profiling can be interactively enabled or disabled while the application is running. Profiling can be enabled or disabled before launching the application either using the main menu option, tool bar option or through the checkbox on the Session settings dialog. After the application is launched and it is running profiling can be enabled or disabled using the main menu option or the tool bar option. The width plot shows idle time gaps on the time line for the periods when profiling is disabled.

Each program run is referred to as a **session**. It is recommended that you save profiling data for multiple sessions for useful analysis of your program. A group of sessions is called a **project**.

Compute Visual Profiler saves the following files:

- Compute Visual Profiler project file:
  **`<project-name>.cvp`**
- Compute Visual Profiler data file for a context in a session:
  **`<project-name>_<session-name>_Context_<context-number>.csv`**
- CUDA API trace data:
  **`<project-name>_<session-name>_Context_<context-number>.trc`**

The following list of Compute Visual Profiler settings are saved across different Compute Visual Profiler sessions.

- Last opened project path
- Method Colors
- Recent files list
- Recent programs
- Recent work Dirs
- Show Output window
- Demangle Method Names
- Main Window/Size
- Main Window/Maximized
- Global view dialog box/Size
- Session view dialog box/Size
- Horizontal Splitter/Sizes
- Vertical Splitter/Sizes
- Profiler Table/Hide Zero Columns
- Summary Table/Show Average
- Summary Plot/Average
- Displayed Summary Plot/Percentage
- Displayed Height Plot/Fit in window
- Height Plot/Show CPU Time
- Height Plot/Show Legend
- Height Plot/Use global scale
- Width Plot/Enable time stamp
- Width Plot/Fit in window
- Width Plot/Maximum bar width
- Width Plot/Show CPU Time
- Width Plot/Show legend
- Width Plot/Start time stamp at zero
- Width Plot/Type

On Windows, these settings are saved in the system registry at the location: HKEY_CURRENT_USER\Software\NVIDIA Corporation\computeprof

On Linux systems these settings are saved to the file: **`$HOME/.config/NVIDIA Corporation/computeprof.conf`**

The Compute Visual Profiler Help cache is saved in the folder shown below:

- Windows : **`C:\Documents and Settings\<username>\Local Settings\Application Data\NVIDIA Corporation\computeprof`**
- Linux : **`$HOME/.local/share/data/NVIDIA Corporation/computeprof`**

There is a separate sub-directory for each version.

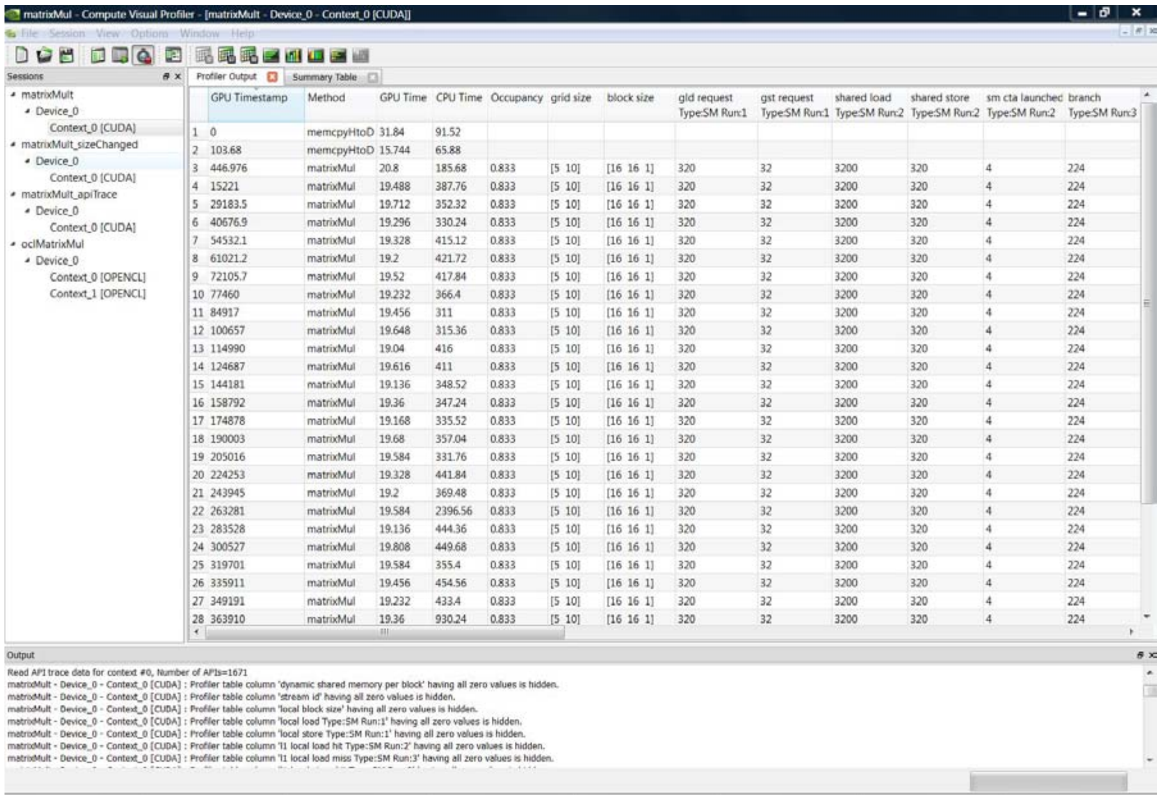
## COMPUTE VISUAL PROFILER USAGE

A brief overview of the graphical user interface (GUI) is included to help explore saved sample projects and to create new projects for profiling.

### Graphical User Interface (GUI) at a Glance

Figure 2 shows the graphical user interface after you launch Compute Visual Profiler and open an existing project. Sample projects are available in the **projects** folder under the **computeprof** folder.

A **project**, saved as a .cvp file, may contain multiple sessions. Multiple sessions can be saved in a single project file and analyzed at a later point in time. Counter data is saved in .csv files and trace data is stored in the .trc files.



| | GPU Timestamp | Method | GPU Time | CPU Time | Occupancy | grid size | block size | gld request Type:SM Run:1 | gst request Type:SM Run:1 | shared load Type:SM Run:2 | shared store Type:SM Run:2 | sm cta launched Type:SM Run:2 | branch Type:SM Run:3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | memcpyHtoD | 31.84 | 91.52 | | | | | | | | | |
| 2 | 103.68 | memcpyHtoD | 15.744 | 65.88 | | | | | | | | | |
| 3 | 446.976 | matrixMul | 20.8 | 185.68 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 4 | 15221 | matrixMul | 19.488 | 387.76 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 5 | 29183.5 | matrixMul | 19.712 | 352.32 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 6 | 40676.9 | matrixMul | 19.296 | 330.24 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 7 | 54532.1 | matrixMul | 19.328 | 415.12 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 8 | 61021.2 | matrixMul | 19.2 | 421.72 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 9 | 72105.7 | matrixMul | 19.52 | 417.84 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 10 | 77460 | matrixMul | 19.232 | 366.4 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 11 | 84917 | matrixMul | 19.456 | 311 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 12 | 100657 | matrixMul | 19.648 | 315.36 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 13 | 114990 | matrixMul | 19.04 | 416 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 14 | 124687 | matrixMul | 19.616 | 411 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 15 | 144181 | matrixMul | 19.136 | 348.52 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 16 | 158792 | matrixMul | 19.36 | 347.24 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 17 | 174878 | matrixMul | 19.168 | 335.52 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 18 | 190003 | matrixMul | 19.68 | 357.04 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 19 | 205016 | matrixMul | 19.584 | 331.76 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 20 | 224253 | matrixMul | 19.328 | 441.84 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 21 | 243945 | matrixMul | 19.2 | 369.48 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 22 | 263281 | matrixMul | 19.584 | 2396.56 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 23 | 283528 | matrixMul | 19.136 | 444.36 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 24 | 300527 | matrixMul | 19.808 | 449.68 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 25 | 319701 | matrixMul | 19.584 | 355.4 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 26 | 335911 | matrixMul | 19.456 | 454.56 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 27 | 349191 | matrixMul | 19.232 | 433.4 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |
| 28 | 363910 | matrixMul | 19.36 | 930.24 | 0.833 | [5 10] | [16 16 1] | 320 | 32 | 3200 | 320 | 4 | 224 |

```
Read API trace data for context #0, Number of APIs=1671
matrixMult - Device_0 - Context_0 [CUDA] : Profiler table column 'dynamic shared memory per block' having all zero values is hidden.
matrixMult - Device_0 - Context_0 [CUDA] : Profiler table column 'stream id' having all zero values is hidden.
matrixMult - Device_0 - Context_0 [CUDA] : Profiler table column 'local block size' having all zero values is hidden.
matrixMult - Device_0 - Context_0 [CUDA] : Profiler table column 'local load Type:SM Run:1' having all zero values is hidden.
matrixMult - Device_0 - Context_0 [CUDA] : Profiler table column 'local store Type:SM Run:1' having all zero values is hidden.
matrixMult - Device_0 - Context_0 [CUDA] : Profiler table column 'l1 local load hit Type:SM Run:2' having all zero values is hidden.
matrixMult - Device_0 - Context_0 [CUDA] : Profiler table column 'l1 local load miss Type:SM Run:3' having all zero values is hidden.
```

Figure 2. Compute Visual Profiler GUI

## Session Frame (Left)

A session is associated with a program run. A group of sessions is called a project. The frame on the left lists all the sessions in the current project as a tree with three levels:

- Sessions at the top level
  **Session** profiles a CUDA program run with certain options; profiled data capture from this run is then presented as tables, counters, graphs, and plots. The default session names (Session1, Session2 etc.) may be customized by right-clicking on the default Session name and selecting **Rename**.
- Devices under a session at the second level
  **Device** corresponds to each physical GPU in the system. For multi-GPU systems multiple devices are displayed. The **Device** which is a child of a **Session** is named as **Device_< device_number** >, with **device_number** starting at 0, for example: **Device_0**.
- Contexts under a device at the third level
  **Context** corresponds to a CUDA context which in turn is analogous to a CPU process. For a multi-context application, multiple contexts can be seen under the **device** tab. The *Context* which is a child of a Device is named as **Context_<context_number> [CUDA|OPENCL]** with **context_number** starting at 0 for example: Context_0 [CUDA] or Context_1 [OPENCL].
  The type of session- CUDA or OPENCL is shown within square parentheses.

Right-clicking on a **session** item in the tree view (left frame) displays the following context sensitive menu items related to customizing the session:

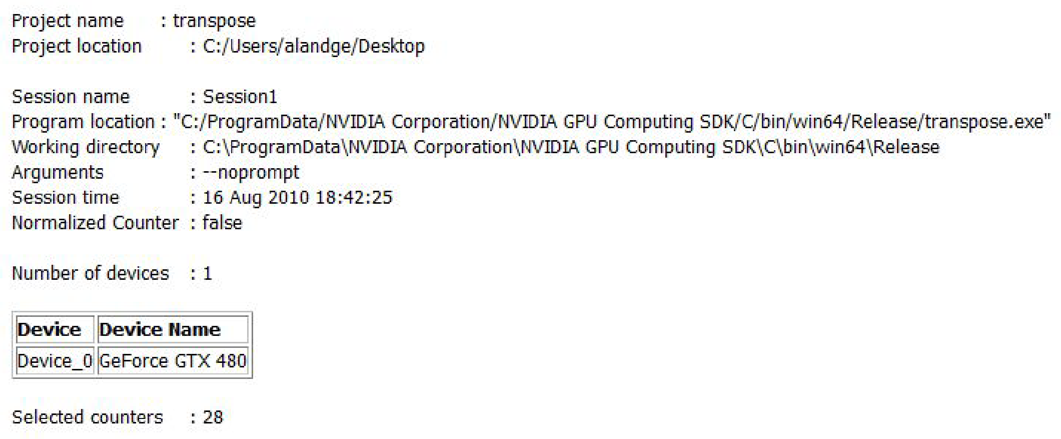
- Rename: Rename the current session.
- Delete: Delete the current session.
- Copy settings to current: Copy settings for the current session as the session settings to be used for a new profiling session.
- Session level summary plot: Displays the GPU Utilization Plot.

## Workspace Frame (Right)

### Session

When a session is selected in the tree view in the left frame, the right frame displays a summary of the information related to the session.

Project and device related information is displayed as shown in Figure 3.



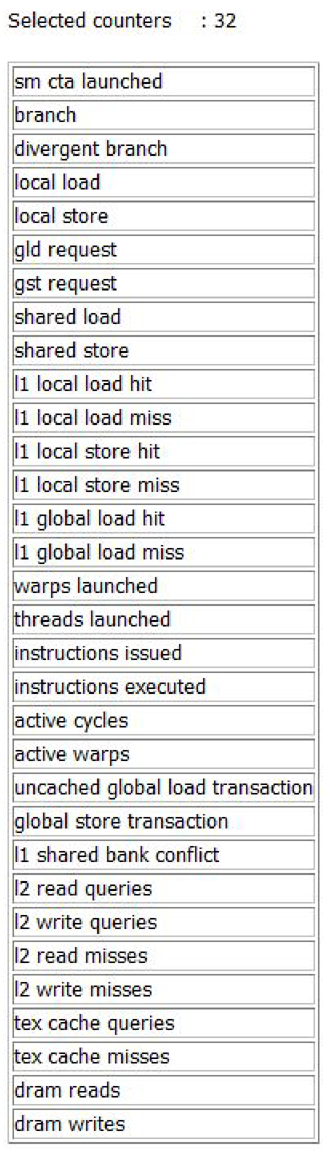
```
Project name     : transpose
Project location     : C:/Users/alandge/Desktop

Session name         : Session1
Program location : "C:/ProgramData/NVIDIA Corporation/NVIDIA GPU Computing SDK/C/bin/win64/Release/transpose.exe"
Working directory    : C:\ProgramData\NVIDIA Corporation\NVIDIA GPU Computing SDK\C\bin\win64\Release
Arguments            : --noprompt
Session time         : 16 Aug 2010 18:42:25
Normalized Counter : false

Number of devices    : 1
```

| Device | Device Name |
|---|---|
| Device_0 | GeForce GTX 480 |

```
Selected counters    : 28
```

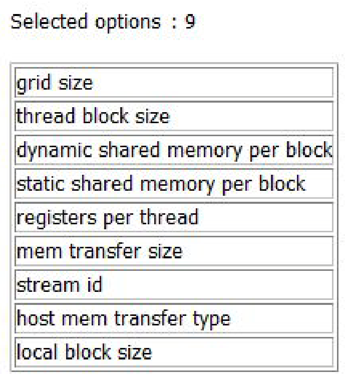
Figure 3. Session Properties- Project Related

In addition, counter information and selected options are displayed as shown in Figure 4 and Figure 5 respectively.


Selected counters : 32

| |
|---|
| sm cta launched |
| branch |
| divergent branch |
| local load |
| local store |
| gld request |
| gst request |
| shared load |
| shared store |
| l1 local load hit |
| l1 local load miss |
| l1 local store hit |
| l1 local store miss |
| l1 global load hit |
| l1 global load miss |
| warps launched |
| threads launched |
| instructions issued |
| instructions executed |
| active cycles |
| active warps |
| uncached global load transaction |
| global store transaction |
| l1 shared bank conflict |
| l2 read queries |
| l2 write queries |
| l2 read misses |
| l2 write misses |
| tex cache queries |
| tex cache misses |
| dram reads |
| dram writes |

Figure 4. Session Properties- Counters



Selected options : 9

|  |
|---|
| grid size |
| thread block size |
| dynamic shared memory per block |
| static shared memory per block |
| registers per thread |
| mem transfer size |
| stream id |
| host mem transfer type |
| local block size |

Figure 5. Session Properties- Selected Options

## Device

When a **device** is selected in the tree view (left frame), the right frame displays a summary of the information related to the device. The following information is displayed:

- Device name
- Number of Contexts
- Table listing: Context | Type | Number of rows

Right-clicking on a *Device* (level below *Session*) item in the tree view displays the Device Summary Plot.

## Context

Right-clicking on a **context** item (level below device) in the tree view displays the following context sensitive menu items:

1. Summary table (See section entitled, ″Summary Table”)

- Kernel table (See section entitled, ″Kernel Table”)
- Memcopy table (See section entitled, ″Memcopy Table”)
- GPU time summary plot (See section entitled, ″GPU Time Summary Plot”)
- GPU time height plot (See section entitled, ″GPU Time Height Plot”)
- GPU time width plot (See section entitled, ″GPU Time Width Plot”)
- Comparison summary Plot (See section entitled, ″Comparison Summary Plot”)
- CUDA API Trace (See section entitled, “CUDA API Trace”)

The workspace to the right of the tree view frame contains tabbed windows for each session, each device in a session and for each context for a device. The different windows for each context are shown as different tabs:

- Profiler Output table
- Summary table
- Kernel table
- Memcopy table
- GPU Time height plot
- GPU Time width plot
- GPU Time Summary Plot
- Profiler counter plot
- Column plot
- Comparison Summary plot
- CUDA API Trace

Right-clicking on the table headers of the Profiler Output table and Summary table allows you to customize the columns displayed.

**Go to the Profiler Output (or Summary) Table →Right-click on any cell table header.**

The following options are displayed:

- Hide
- Hide zero columns
- Show all columns

## Output Frame (Bottom)

The Output frame displays at the bottom of the Compute Visual Profiler GUI screen (Figure 1). The Output frame contains standard error information and any status messages associated with the Compute program (CUDA or OpenCL program) you are running.

## Exploring a Saved Project

This example illustrates how you can explore the various sessions and view settings and options to obtain the tables and plots of your choice. Several sample projects are available in the **<CudaToolkitDir>\computeprof\projects** folder when the CUDA toolkit and SDK are installed.

1. Open the project saved in the **projects** folder (**<CudaToolkitDir>\computeprof\projects**) using the main menu option **File→Open**. The **Profiler Output** table is displayed.
2. Right-click **Session1→Device_0→Context_0** in the session tree to display the various tables and plots available (See Figure 6.)

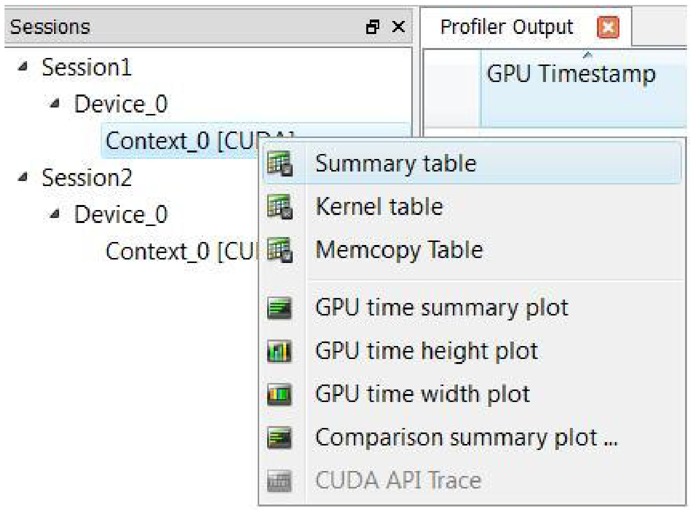


Figure 6. Displaying Tables and Plots for a Saved Project

3. Select settings for a new session by using the main menu option **Session→Session settings**.
4. Browse and select the Compute program to profile.
5. Change the working directory if it is different from the program directory.
6. Execute the Compute program by clicking **Launch** in the Session settings dialog box**.** If the Compute program is correctly executed the profiler output is displayed. Compare the profiler output for Session1 and Session2.
7. Right-click on the appropriate row or column in the profiler output or summary table for a session to try the Profiler counter plot and Column plot.
8. Exit Compute Visual Profiler using the main menu option **File→Exit** from the main menu.

## Creating a New Project

Profiling is automatically enabled by Compute Visual Profiler. Use the following procedure to create a new project and display the various tables and plots of your choice.

1. **Open** the project ( **.cvp** file) using main menu option **File→New** or Click on the **File Open Project** icon in the toolbar .
2. **Select** the project name and project directory where the project files will be saved.
3. **Select** the session settings through the dialog box. See section entitled, "Session" for details.
4. **Browse** and select the Compute program to profile.
5. **Change** the working directory if it is different from the program directory.
6. **Click** on **Session settings** on the **Session** menu**.** Select options for maximum program execution time, profiler counters, kernel and memory transfer options using the tabbed options.
   See Figure 11, Figure 12, and Figure 13 for details on the **Session→Session settings** tabs.
7. **Execute** the Compute program by clicking the Start button of the Session settings dialog box or through the main menu option **Session→Start**. If the Compute program executes correctly, the profiler output is displayed.
8. **Right-click** on **Session1→Device_0→Context_0** to display the summary table in the session tree.
9. **Choose** the Summary table option or use the Summary table toolbar option.
10. **Right-click** on **Session1→Device_0→Context_0** in the session tree and choose the GPU Time Summary Plot option to display the GPU Time summary plot. You may also use the GPU Time Summary Plot toolbar option. The Profiler Output and GPU Time Summary plot windows can be viewed by scrolling, resizing and or repositioning.
11. **Save** the project by using the main menu option **File→Save** or the toolbar icon.
12. **Exit** Compute Visual Profiler using the main menu option **File→Exit** from the main menu.

# COMPUTE VISUAL PROFILER GRAPHICAL USER INTERFACE (GUI)

## MAIN MENU BAR

Figure 7 shows the main toolbar for the CUDA™ Compute Visual Profiler.

Most operations can be conducted using the File pull down menu or the toolbar situated right below the Menu bar.

The Menu bar consists of the main menu options: **File**, **Session**, **View**, **Options**, **Window,** and **Help**. See the description below for details on the menu options.

The Toolbar icons fall into four main groups: **File, Profile, Session View Settings,** and **Tables and Plots.** They provide options for file and project management, session settings, and the various output formats.

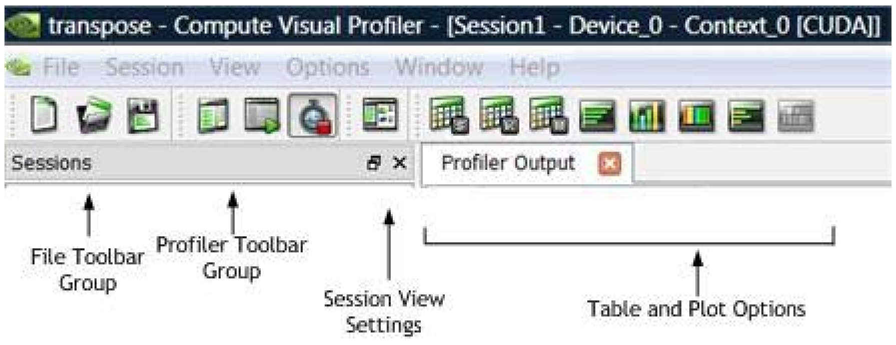


Figure 7. Compute Visual Profiler Menu Bar and Toolbar

## File

**On the main menu→Click on File.** Figure 8. shows the **File** pull down menu.

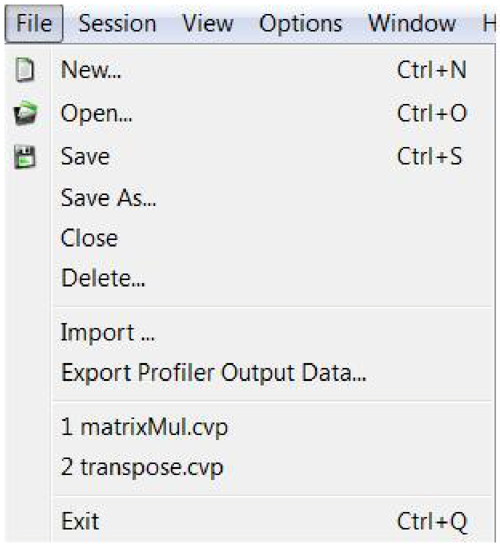


Figure 8. File Pull Down Menu

The **File** toolbar group consists of the following self-explanatory options:

- **New** : Creates a new project. The New project dialog box is opened to choose the project name and project directory. On **OK** the **Session settings** dialog box is opened.
- **Open** : Opens an existing project. The Open project dialog box is opened to select the profiler project to be opened. On Open the project data for all sessions is loaded and the profiler data table is displayed.
- **Save** : Saves the current project. The profiler data for the current open project is saved to the disk.
- **Save As** : Saves the current project as a new project. The project name and directory can be selected. The profiler data for the current open project is saved to the disk.
- **Close** : Closes the current project. The current open project is closed. All profiler session data is deleted from memory and all open windows are closed.
- **Delete** : Deletes the project. File dialog box is opened to select the project. It deletes the selected project file (.cvp) and related data files(.csv) files.
- **Import**: Imports Compute Visual Profiler output in comma-separated format (CSV). A new session is created in the current project and imported data is loaded.
- **Export**: Exports Compute Visual Profiler output for the current session to a file in the comma-separated format (CSV).
- List of recently opened profiler projects.
- **Exit**: Exits the Compute Visual Profiler program.

## New Project Dialog Box

The **File→New** project dialog box has two boxes as shown in Figure 9.

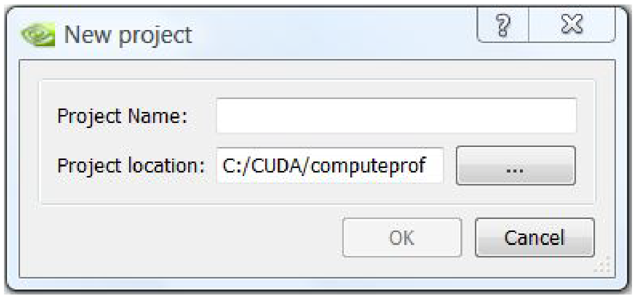


Figure 9. File->New Project Dialog Box

- Project Name: Name of the profiler project.
- Project location: Directory where the project files are saved.

## Session

**On the main menu→Click on Session**
Figure 10 shows the Session pull down menu.

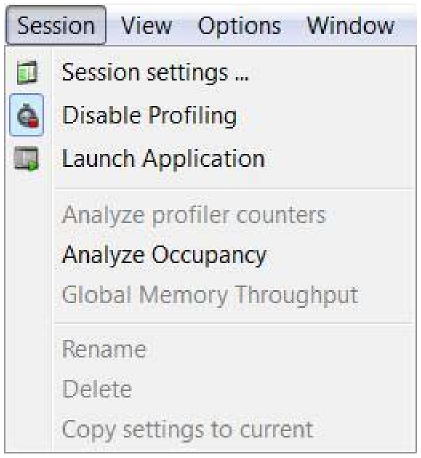


Figure 10. Session Pull Down Menu

The Session menu consists of:

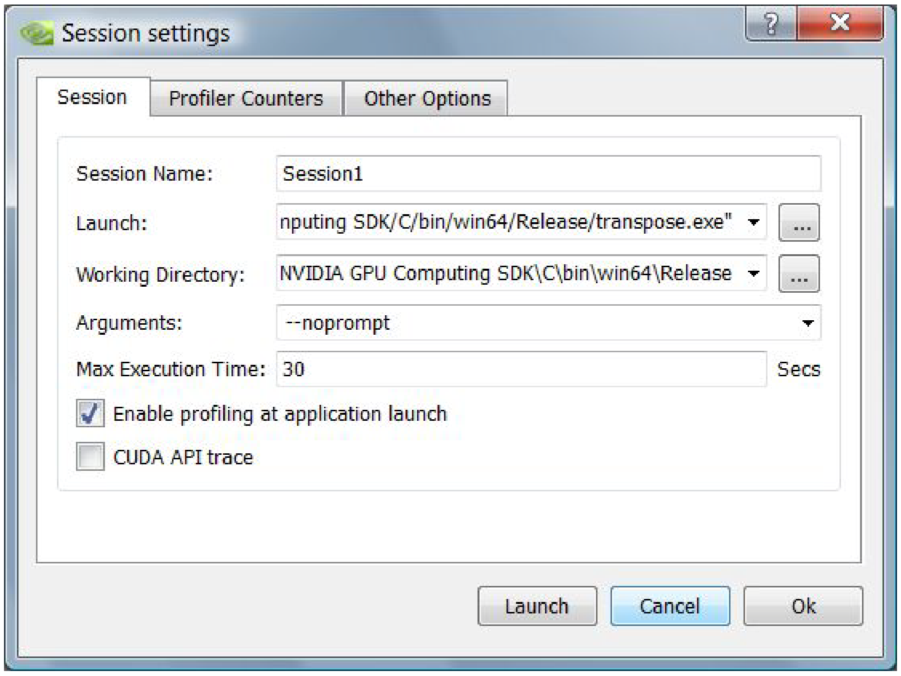
- **Session settings**: Change session settings.
- **Disable Profiling**: Option to disable profiling.

- **Global Memory Throughput**: Display overall application level global memory read throughput, global memory write throughput and overall global memory throughput.
- **Rename**: Rename the current session.
- **Delete**: Delete the current session. This is same as the Session context menu Delete option.
- **Copy settings to current**: Copy settings for the current session as the session settings to be used for a new profiling session.

## Session settings

As shown in Figure 11, the Session setting dialog has three tabs:

- Session
- Profiler Counters
- Other Options



Session Name: Session1
Launch: nputing SDK/C/bin/win64/Release/transpose.exe"
Working Directory: NVIDIA GPU Computing SDK\C\bin\win64\Release
Arguments: --noprompt
Max Execution Time: 30 Secs
[✓] Enable profiling at application launch
[ ] CUDA API trace

Figure 11. Session->Session settings->Session (tab)

## Session Tab

**On the main menu→Click on Session→Click on Session settings→Click on the Session tab**

Using this tab customizes your session; the following options are available:

- **Session Name**: Name of the profiler session. By default a new session name is chosen (Session1, Session2 ...). This can be changed by the user.
- **Launch**: Select the Compute program to be profiled.
- **Working Directory**: Select the working directory to be used for running the Compute program.
- **Arguments**: Command line arguments to be passed to the Compute program.
  - Multiple command line arguments should be separated by one or more spaces.
  - Arguments containing spaces should be enclosed within double quotes.
  - Double quotes can be used within an argument; there is no need to use a backslash followed by double quote.
  - Each backslash is replaced by two backslash characters and there is no other special handling for backslash.
- **Max. execution time (in seconds)**: Select maximum time to wait for Compute program execution completion. After this cutoff time the program is aborted.
- **Run in separate window**: This option is useful for console applications which accept some keyboard input. In this case the Compute program is run from a separate window. The standard output and standard error for the Compute program is shown in this separate window.

**Note:** Currently this option is supported only on Linux and a new xterm window is opened.

- **CUDA API trace**: This option is used to collect CUDA driver API call information.

**Note:** Currently this option is not supported on MacOS X.

## Profiler Counters Tab

**On the main menu→Click on Session→Click on Session settings→Click on the Profiler Counters tab**

Profiler Counters are logically grouped based on their functions. Since only a few of the selected profiler counters can be collected for a single program run - the Compute program should be run multiple times.

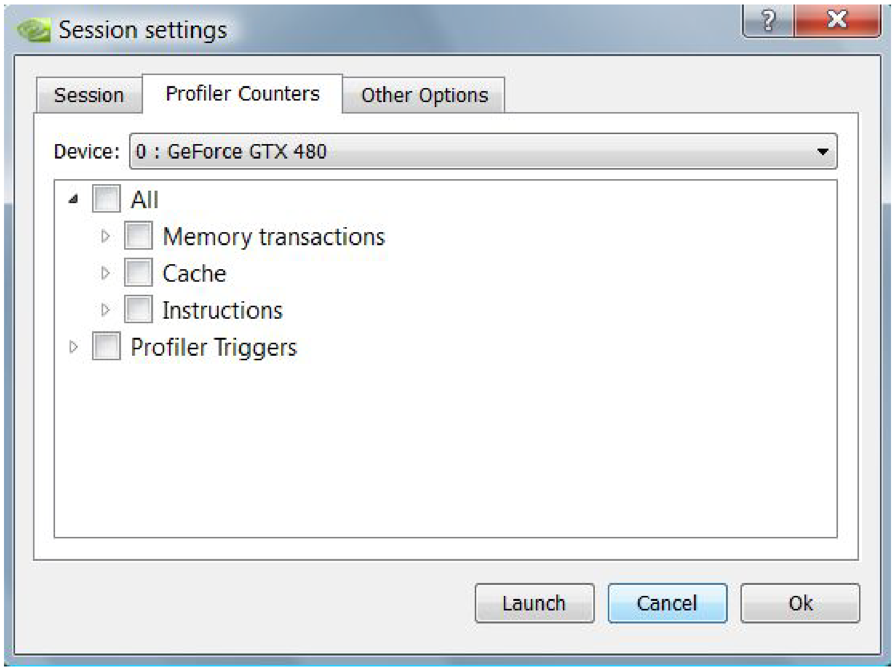


Figure 12. Session->Session settings->Profiler Counters (tab)

Using this tab, customize the profile counters of interest; the following options are available:

- **Device**: Selection of a device in this option displays the list of counters that are supported on that device. The user can then select the desired counters from this list. If device 0 is selected in device selection then only profiler counters supported on device 0 are listed for selection. If multi-device option is selected then all the counters supported on all devices (device 0, device 1...) are selected. In this case device specific counters are ignored for contexts which are run on other devices. The following warning message is displayed in the output window:

```
NV_Warning: Ignoring the invalid profiler config option:
gld_incoherent.
```

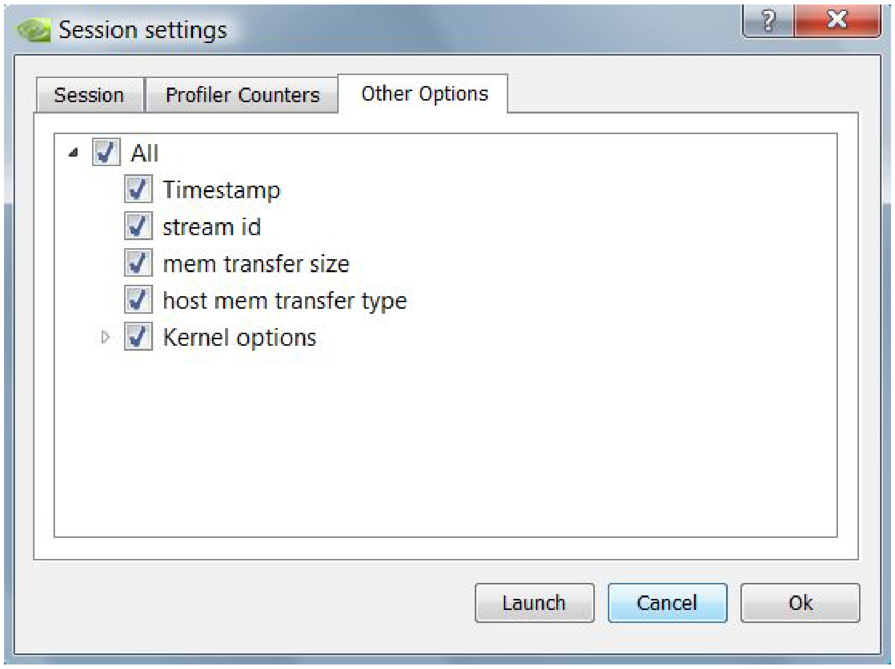
**Note:** Selecting a device from the Session tab does not run the program on the device selected; the user has to handle the device selection in the program.

- You can select or de-select all counters by using the Select All Counters check box.
- You can also select any sub-set of specific counters using the check boxes for each counter.
- You can enable or disable normalization of counter values by using the Normalize counters check box.

Profiler counters are available only with CUDA toolkit version 1.1 or later.

## Other Options Tab

**On the main menu→Click on Session→Click on Session settings→Click on the Other Options tab**



Session settings

Session | Profiler Counters | Other Options

- [✓] All
  - [✓] Timestamp
  - [✓] stream id
  - [✓] mem transfer size
  - [✓] host mem transfer type
  - [✓] Kernel options

Launch | Cancel | Ok

Figure 13. Session->Session settings->Other Options (tab)

Using this tab, customize other metrics of interest for the session; the following options are available:

- **Timestamp**: Enable option to include time stamps for kernel/method launching. GPU timestamp is the time when a method starts execution on the GPU. GPU timestamps are shifted in origin, to make the minimum GPU timestamp zero, across all devices and all contexts in a session.
- **Stream id**: Enable option to include stream id for kernel/method. This feature is available only with CUDA toolkit version 1.1 or later.

- **Memory Transfer Size**: It is to be enabled for describing the size of memory transfer. It outputs the total size in bytes at the Memcopy table when profiling was done with this option enabled.
- **Host Memory Transfer Type**: It specifies if the host memory from/to which data is transferred, is pageable or page-locked.
- **Kernel Options**: This is a group of the following options:
    - **Grid Size** : It is to be enabled to get dimensions of grid in terms of blocks (3 dimensional) in Kernel table.
    - **Thread Block Size**: It is to be enabled to get dimensions of a block in terms of threads (3 dimensional).
    - **Dynamic shared memory size**: It is to be enabled to get Dynamic shared memory size.
    - **Static shared memory size**: It is to be enabled to get Static shared memory size.
    - **Register per thread**: It is to be enabled to get Register count per thread.
    - **Local Block Size**: If workgroupsize has been specified by the user, this option would be 1, otherwise it would be 0(used only for OpenCL.)

## View

**On the main menu→ Click on View**
Figure 14 shows the View pull down menu.

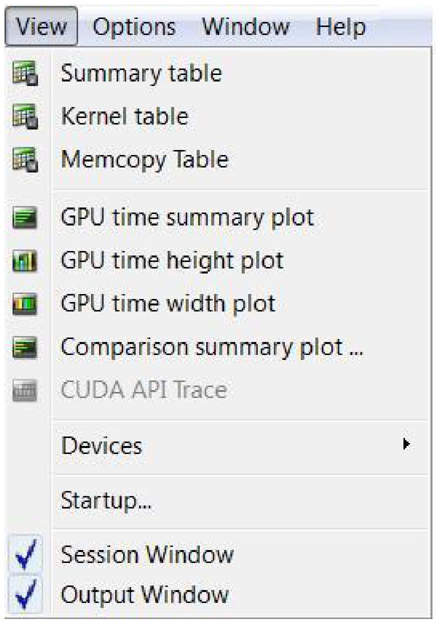


Figure 14. View Pull Down Menu

## Summary Table

For a selected context in the left frame (Sessions tree view):

**On the main menu→ Click on View→ Click on Summary table**

The summary table for the selected context within the current session is displayed in the right frame. The rows in the table are sorted in decreasing order of total GPU time and memcopy is shown as the last row.

The summary table has the following columns:

- Method: method name.
- #Calls: number of calls.
- GPU usec: total GPU time in micro seconds.
- CPU usec: total CPU time in micro seconds (column is hidden by default.)
- %GPU time: Percentage of total GPU time across all methods.
- Cumulative count column for each available profiler counter (columns are hidden by default.)
- Derived statistics:
    - glob mem read throughput
    - glob mem write throughput
    - glob mem overall throughput
    - gld efficiency
    - gst efficiency
    - instruction throughput
    - retire ipc
    - active warps/active cycles
    - l1 gld hit rate
    - texture hit rate %

For a description of the derived statistics please refer to the section entitled "Supported Derived Statistics".

## Kernel Table

For a selected context in the left frame (Sessions tree view):

**On the main menu→ Click on View→ Click on Kernel table**

The Kernel table with the following properties is displayed:

- method: method name
- #calls: number of times the kernel is called
- Grid Size (x,y,z dimensions)
- Thread Block Size (x,y,z dimensions)
- Dynamic Shared Memory per Block
- Static Shared Memory per Block
- Registers per Thread

## Memcopy Table

**On the main menu→ Click on View→ Click on Memcopy Table**

The memcopy table with the following properties is displayed:

- Method
- #calls
- Host mem transfer type
- Memory Transfer Size

## GPU Time Summary plot

**On the main menu→ Click on View→ Click on GPU time summary plot**

The GPU time summary plot for the current session is displayed. This is same as selecting the GPU Time Summary plot option from the Session context menu.

## GPU Time Height plot

**On the main menu→ Click on View→ Click on GPU time height plot**

The GPU time height plot for the selected context in the current session is displayed. This is same as selecting the **GPU time height plot** option from the Session context menu.

## GPU Time Width plot

**On the main menu→ Click on View→ Click on GPU time width plot**

The GPU time width plot for the selected context in the current session is displayed. This is same as selecting the **GPU time width plot** option from the Session context menu.

### Comparison plot

For a selected context in the current session:

**On the main menu→ Click on View→ Click on Comparison summary plot**

## Options

**On the main menu→ Click on Options.** Figure 15 shows the Options pull down menu.

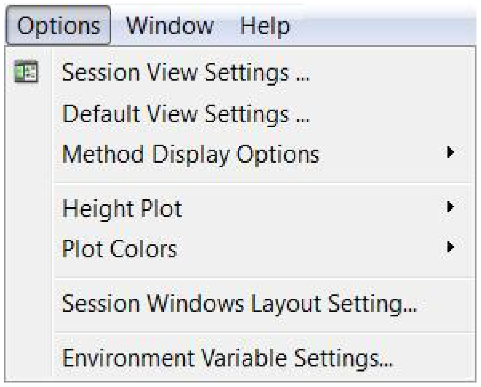

Options
Window
Help
Session View Settings ...
Default View Settings ...
Method Display Options
Height Plot
Plot Colors
Session Windows Layout Setting...
Environment Variable Settings...

Figure 15. Options Pull Down Menu

The Options menu consists of:

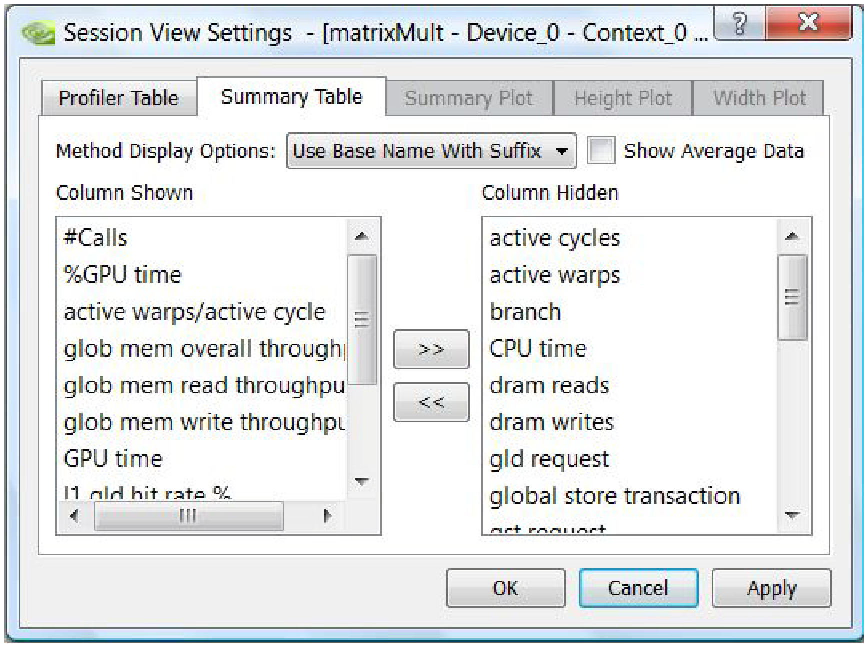
- Session view settings: Change session view settings for the current session.
- Default view settings: Change the default view settings to be used for new sessions.
- Method Display Options: One of the following options to display method names :
  - Use Full Name : Full Mangled name is displayed.
  - Use Base Name : Only base name is displayed.
  - Use Base Name with Suffix : Full Mangled name with suffix is displayed.
- Height plot: Change global GPU time height plot options.
  - Use Global Scale: Enable/Disable option to use a common global scale across multiple sessions.
- Plot Colors: Select colors for plots.
  - Method Colors: Pop ups a color dialog box which can be used to select colors used for different methods in plots. The colors are saved on application exit and so they can be used across Compute Visual Profiler sessions.

- Session Windows Layout Setting: Change settings for display of multiple session windows. The choices are:
  - Maximize
  - Cascade
  - Tiles
- Environment Variable Settings: Change environment variable settings used by the Compute program.

## Options->Session View Settings Dialog Box

**On the main menu→ Click on Options→ Click on Session View Settings**

This dialog box can be invoked using the main menu option **Options→Session View Settings** or the toolbar. This dialog box allows the changing of settings for the different views for the current session. There is a separate tab for different views. The dialog box is opened with the tab corresponding to the current view. Only tabs for currently created views can be selected.


Session View Settings - [matrixMult - Device_0 - Context_0 ...

Profiler Table | Summary Table | Summary Plot | Height Plot | Width Plot

Method Display Options: Use Base Name With Suffix

Show Average Data

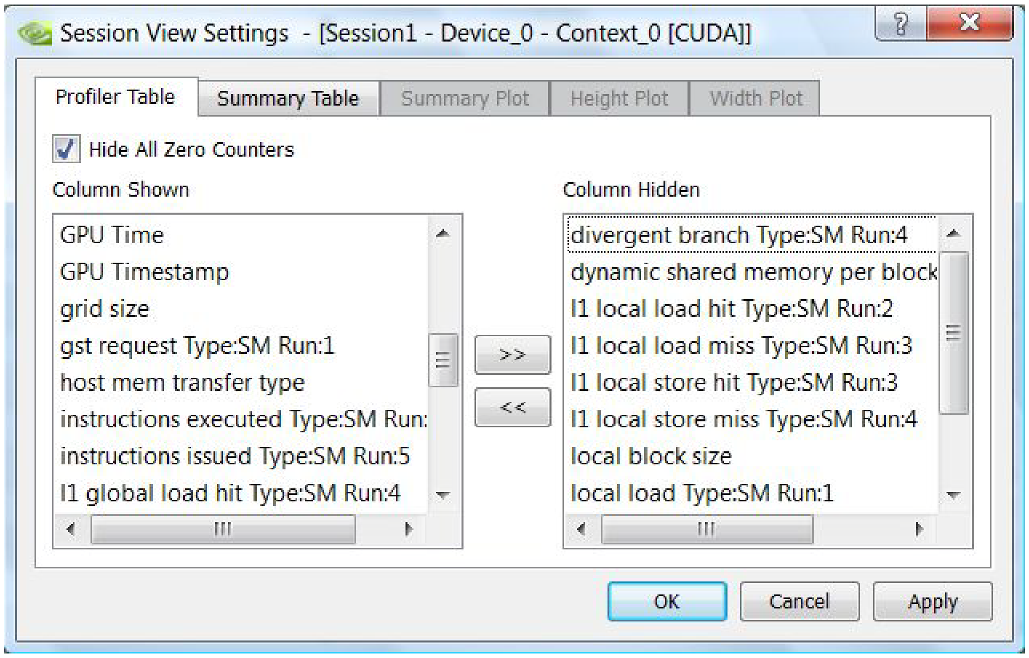
Column Shown:
#Calls
%GPU time
active warps/active cycle
glob mem overall throughp
glob mem read throughpu
glob mem write throughpu
GPU time
l1 gld hit rate %

Column Hidden:
active cycles
active warps
branch
CPU time
dram reads
dram writes
gld request
global store transaction
gst request

Figure 16. Options->Session View Settings Dialog

## Profiler Table Tab

**On the main menu→ Click on Options→ Click on Session View Settings→ Select the Profiler Table tab**



☑ Hide All Zero Counters

| Column Shown | Column Hidden |
|---|---|
| GPU Time | divergent branch Type:SM Run:4 |
| GPU Timestamp | dynamic shared memory per block |
| grid size | l1 local load hit Type:SM Run:2 |
| gst request Type:SM Run:1 | l1 local load miss Type:SM Run:3 |
| host mem transfer type | l1 local store hit Type:SM Run:3 |
| instructions executed Type:SM Run: | l1 local store miss Type:SM Run:4 |
| instructions issued Type:SM Run:5 | local block size |
| l1 global load hit Type:SM Run:4 | local load Type:SM Run:1 |

Figure 17. Options->Session View Settings->Profiler Table Tab

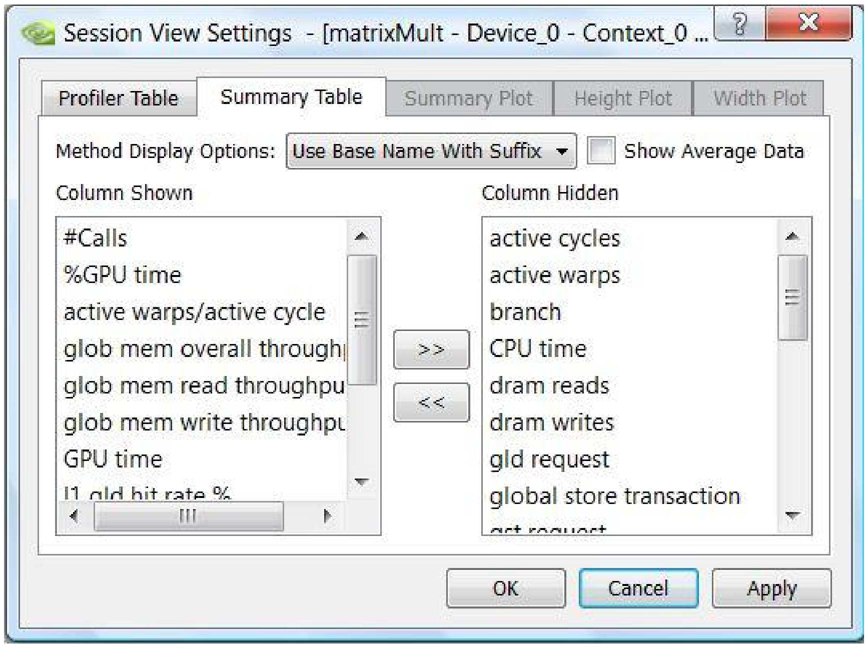
Customize the Profiler table output using the following options:

- Hide All Zero Counters: Enable /disable hiding of counter columns having all zero values. This is enabled by default.
- Columns Shown: Lists columns which are to be shown. Can select and move columns from hidden list to shown list using **<<**.
- Columns Hidden: Lists columns which are to be hidden. Can select and move columns from shown list to hidden list using **>>**.

## Summary Table Tab

**On the main menu→ Click on Options→ Click on Session View Settings→ Select the Summary Table tab**

Customize the Summary table output using the following options:



Method Display Options: Use Base Name With Suffix
Show Average Data

Column Shown

#Calls
%GPU time
active warps/active cycle
glob mem overall through
glob mem read throughpu
glob mem write throughpu
GPU time
l1 gld hit rate %

Column Hidden

active cycles
active warps
branch
CPU time
dram reads
dram writes
gld request
global store transaction
gst request

Figure 18. Options->Session View Settings->Summary Table Tab

- Method Display Options: One of the following options to display method names :
  - Use Full Name: Full Mangled name is displayed.
  - Use Base Name: Only base name is displayed.
  - Use Base Name with suffix: Full Mangled name with suffix is displayed.
- Show Average Data: Enable/Disable showing average data values. When this option is disabled the sum total across all the calls for a method are shown. When this option is enabled the total value is divided by the number of times the method is called and this average value for a method is displayed. This option is disabled by default.
- Column Shown: Lists columns which are to be shown. Can select and move columns from hidden list to shown list using **<<**.
- Column Hidden: Lists columns which are to be hidden. Can select and move columns from shown list to hidden list using >>. The CPU usec and all counter columns are hidden by default.

## Summary Plot Tab

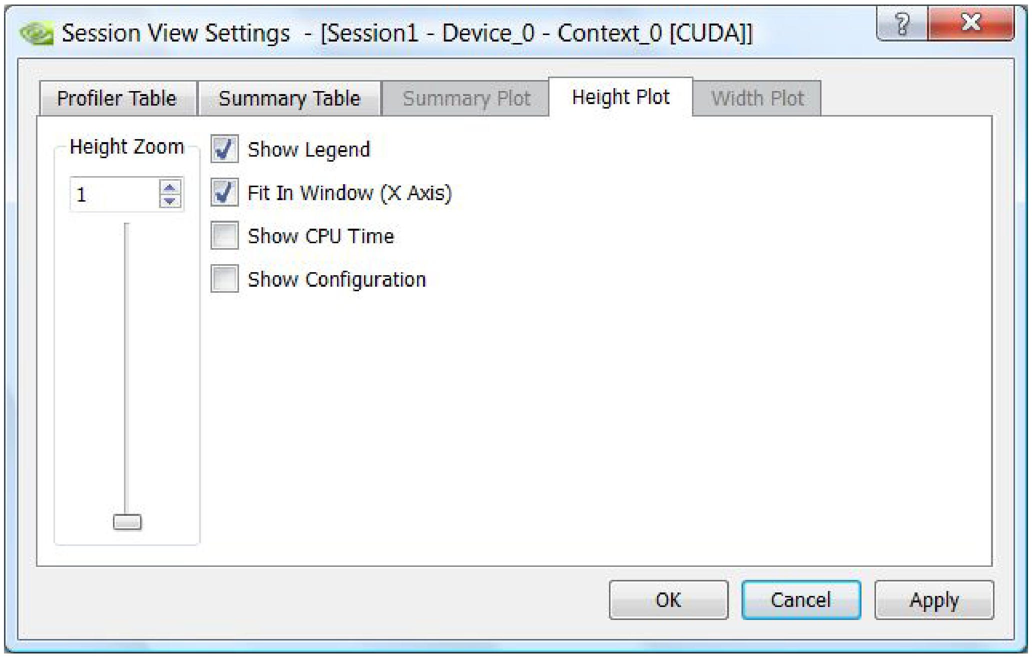
**On the main menu→ Click on Options→ Click on Session View Settings→ Select the Summary Plot tab**

Customize the Summary plot using the following options:

- Method Display Options: One of the following options to display method names :
  - Use Full Name: Full Mangled name is displayed.
  - Use Base Name: Only base name is displayed.
  - Use Base Name with suffix: Full Mangled name with suffix is displayed.
- Percentage Displayed: Enable/disable displaying percentage values. When this option is disabled total values are shown. This option is enabled by default.
- Average Displayed: Enable/disable using average data values. When this option is disabled total values are used. This option is disabled by default.
- Timestamp based Total: Enable/disable calculation of total using initial and final timestamps. If enabled, one extra bar showing GPU Idle with total no of method calls is presented in a different color.

## Height Plot Tab

**On the main menu→ Click on Options→ Click on Session View Settings→ Select the Height Plot tab**


Session View Settings - [Session1 - Device_0 - Context_0 [CUDA]]

Height Zoom: 1

[✓] Show Legend
[✓] Fit In Window (X Axis)
[ ] Show CPU Time
[ ] Show Configuration

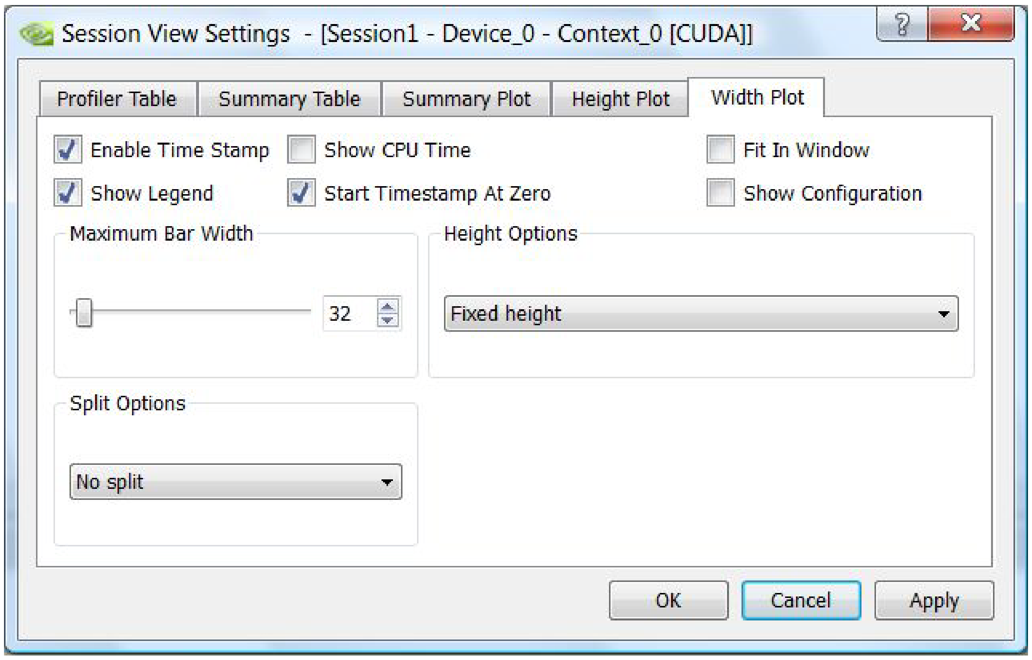
Figure 19. Height Plot Options

Customize the Height plot using the following options:

- **Show legend**: Enable/Disable display of GPU Time plot legend
- **Fit in window**: Enable/Disable option to fit the GPU plot in the window. When fit is enabled multiple bars can overlap.
- **Show CPU Time**: Enable/Disable option to show CPU time.
- **Show Configuration**: Enable/Disable option to show the plot configuration in the plot view.

## Width Plot Tab

**On the main menu→ Click on Options→ Click on Session View Settings→ Select the Width Plot tab**


Session View Settings - [Session1 - Device_0 - Context_0 [CUDA]]

Profiler Table | Summary Table | Summary Plot | Height Plot | Width Plot

- [✓] Enable Time Stamp
- [ ] Show CPU Time
- [ ] Fit In Window
- [✓] Show Legend
- [✓] Start Timestamp At Zero
- [ ] Show Configuration

Maximum Bar Width: 32

Height Options: Fixed height

Split Options: No split

Figure 20. Width Plot Options

Customize the Width plot using the following options:

- Enable Time Stamp: Enable/Disable option to use time stamps.
- Show CPU Time: Enable/Disable option to show CPU time.
- Fit in window: Enable/Disable option to fit the plot in the window.
- Show legend: Enable/Disable display of GPU Time plot legend.
- Start Timestamp at Zero.
- Show Configuration.

- Max Bar Width: Maximum width of a bar in pixels. For this option the plot display is immediately updated and so one can interactively choose an appropriate value.
- Height Options: Choose option to use for bar height.
- Split Options- choose between **No Split** or **Show all devices**.
  - No Split: A single horizontal group of bars is displayed. Even in case of multiple streams or multiple devices the data is displayed in a single group.
  - Split on Device: In case of multiple devices one separate horizontal group of bars is displayed for each device.
  - Split on Stream: In case of multiple devices one separate horizontal group of bars is displayed for each stream.

Apply and OK change the view properties temporarily and permanently, respectively.

### Default View Settings Dialog Box

**On the main menu→ Click on Options→Select Default View Settings**

The **Default View Settings** dialog box allows you to change the default settings which are used for subsequent new session views. The tabs displayed in this window are similar to the tabs displayed in the **Options→Session View Settings** dialog box (see Figure 16, Figure 17, and Figure 18.)

## Window

**On the main menu→ Click on Window**. Figure 21 shows the Window pull down menu.

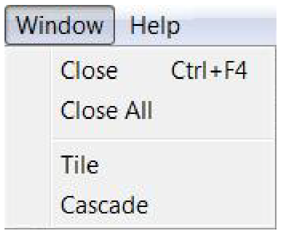


Figure 21. Window Pull Down Menu

The Window menu consists of the following self-explanatory window-related options for the right frame:

- **Close**: Close active window
- **Close All:** Close all open windows
- **Tile:** Tile all open windows
- **Cascade:** Cascade all open windows

## Help

**On the main menu→ Click on Help**
Figure 22 shows the Help pull down menu.

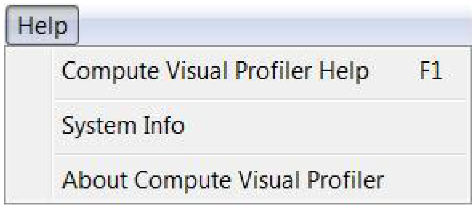

Help
Compute Visual Profiler Help F1
System Info
About Compute Visual Profiler

Figure 22. Help Pull Down Menu

The Help menu consists of:

- **Compute Visual Profiler Help**: Shows the Help for Compute Visual Profiler.

**Note:** This is currently not supported on Mac OS

- **System Info**: Shows the Host system machine configuration information.
- **About Compute Visual Profiler**: Display Compute Visual Profiler program version and copyright information.

## MAIN TOOLBAR

The first row in the top frame shows the main menu options:
File, Session, View, Options, Window, and Help.

As illustrated in Figure 23, the second row in the top frame has four groups of toolbar icons.

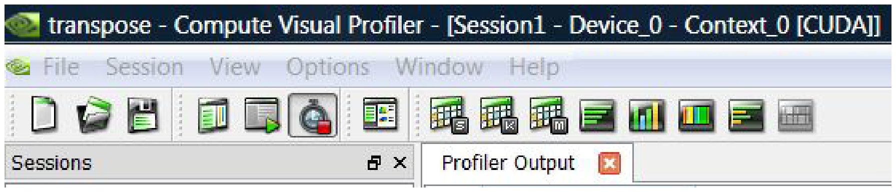

transpose - Compute Visual Profiler - [Session1 - Device_0 - Context_0 [CUDA]]
File
Session
View
Options
Window
Help
Sessions
Profiler Output

Figure 23. Toolbar Icons

### File Toolbar Group

File toolbar group has the following three icons (listed from left to right):

- Create a new project: The behavior is same as the **File→New** menu option.
- Open an existing project: The behavior is same as the **File→Open** menu option.
- Save the current project: The behavior is same as the **File→Save** menu option.

### Profile Toolbar Group

Profile toolbar group has the following three icons (listed from left to right):

- Session settings: The behavior is same as the **Session→Session settings** menu option
- Launch/Abort application: Abort the program.
- Enable/Diasable profiling: The behavior is same as the **Session→Start** menu option

### Session Toolbar Group

The Session toolbar group has the following four icons (listed from left to right)

- Summary table: The behavior is same as the **View→Summary table** menu option
- Summary plot: The behavior is same as the **View→Summary plot** menu option
- Kernel table: The behavior is same as the View->Kernel table menu option.
- Memcopy table: The behavior is same as the View->memcopy table menu option.
- GPU time height plot: The behavior is same as the **View→GPU time height plot** menu option

- GPU time width plot: The behavior is same as the **View→ GPU time width plot** menu option
- CUDA API trace:  The behavior is same as the View->CUDA API trace table menu option

## View Options Toolbar Group

- Session view settings: The behavior is same as the **Options→Session View Settings** menu option

Note that in order to customize your working environment you may enable or disable certain toolbar buttons. **Right-click** anywhere on toolbar for a pop-up that allows you to enable/disable toolbar buttons that fall under the File and Profiler toolbar category.

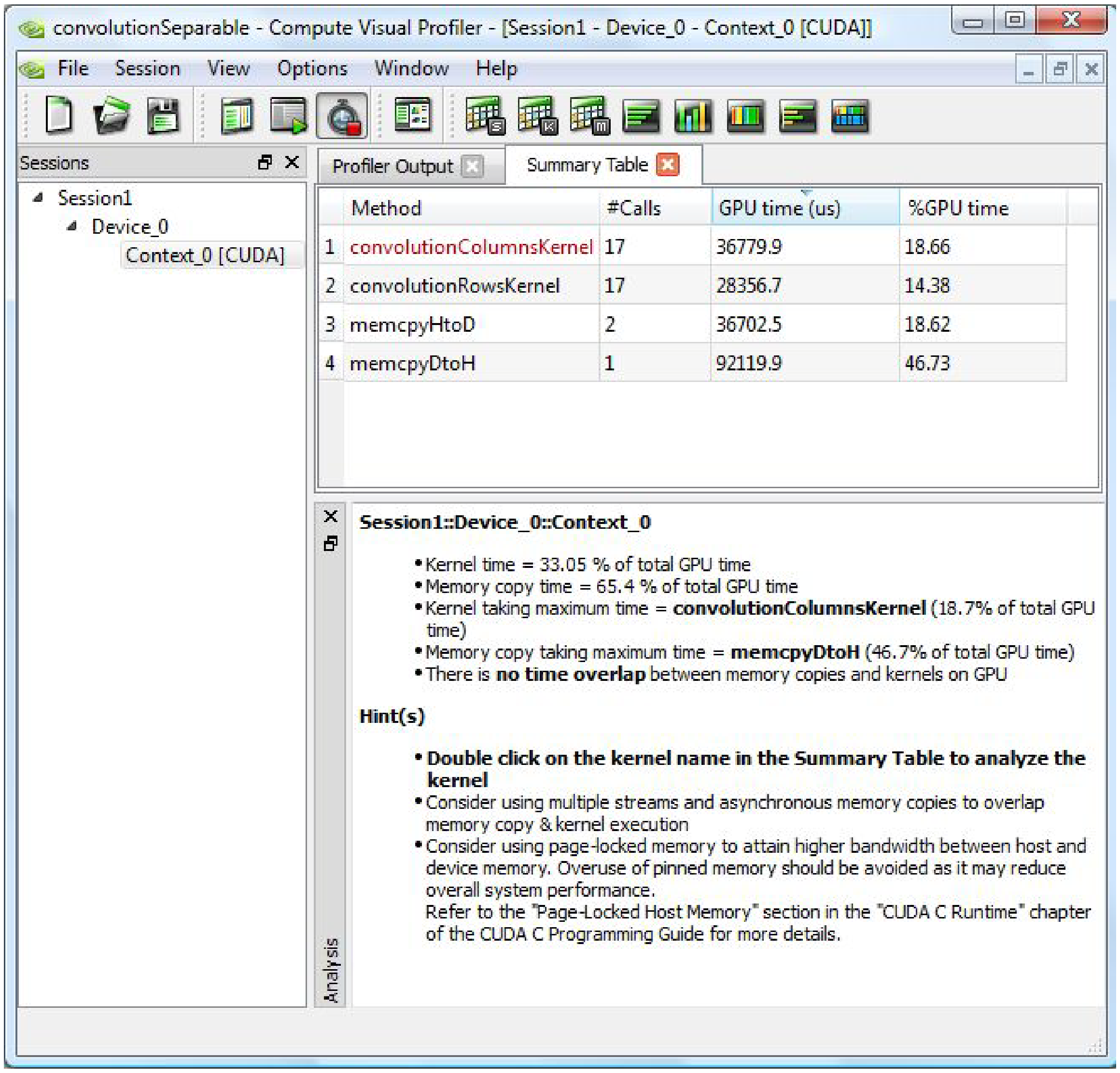
# COMPUTE APPLICATION ANALYSIS

The Visual Profiler contains a powerful analysis feature that provides performance analysis of the application at the context level, kernel level, session level and device level.

## CONTEXT LEVEL ANALYSIS

An analysis of GPU utilization for the CUDA context is carried out at this level and appropriate hints are provided, for example, usage of streams to improve overlap between kernel execution and memory copies.

- Click on the context in **Sessions** tree to get context level analysis in the analysis window.



| | Method | #Calls | GPU time (us) | %GPU time |
|---|---|---|---|---|
| 1 | convolutionColumnsKernel | 17 | 36779.9 | 18.66 |
| 2 | convolutionRowsKernel | 17 | 28356.7 | 14.38 |
| 3 | memcpyHtoD | 2 | 36702.5 | 18.62 |
| 4 | memcpyDtoH | 1 | 92119.9 | 46.73 |

**Session1::Device_0::Context_0**

- Kernel time = 33.05 % of total GPU time
- Memory copy time = 65.4 % of total GPU time
- Kernel taking maximum time = **convolutionColumnsKernel** (18.7% of total GPU time)
- Memory copy taking maximum time = **memcpyDtoH** (46.7% of total GPU time)
- There is **no time overlap** between memory copies and kernels on GPU

**Hint(s)**

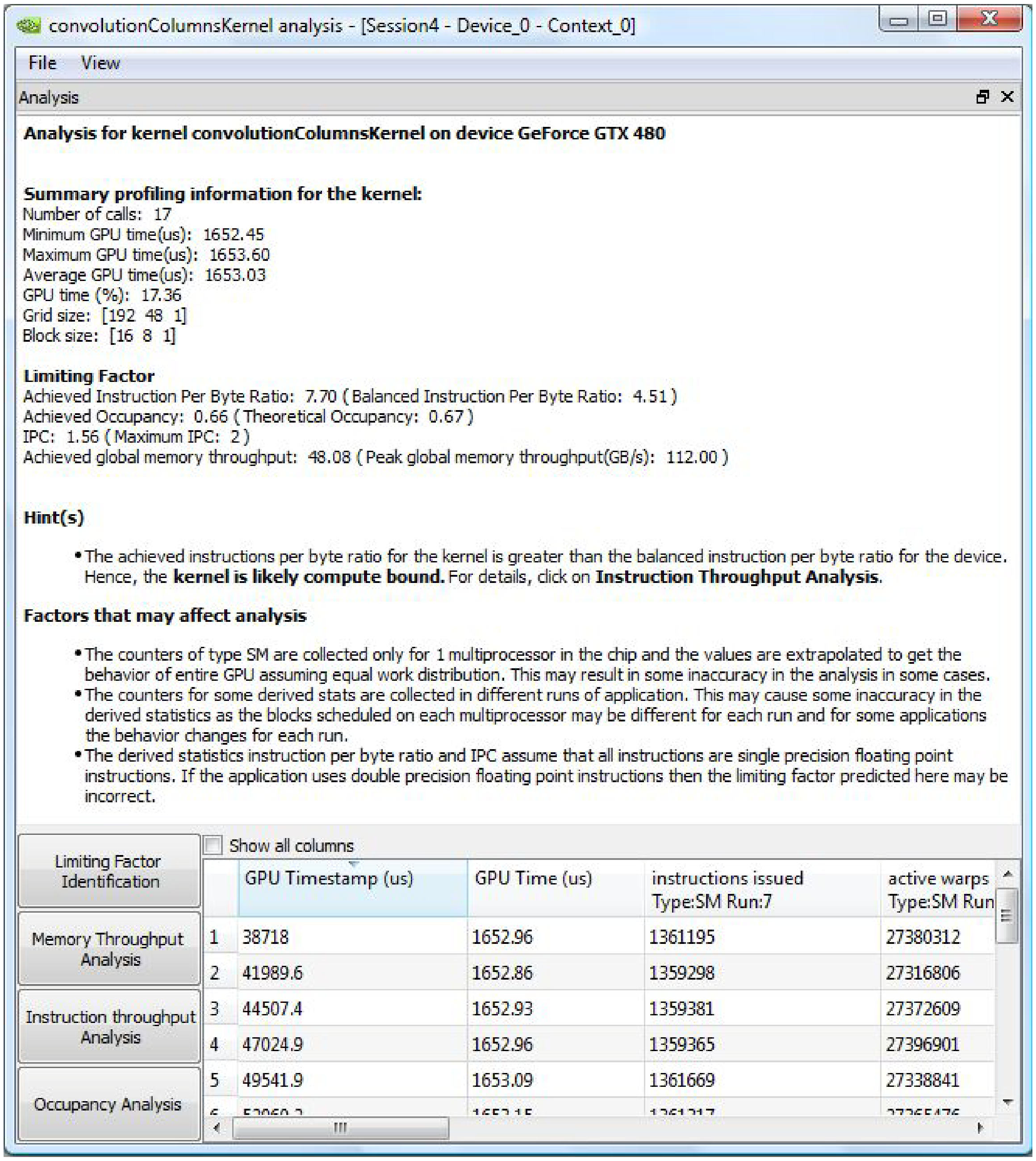
- **Double click on the kernel name in the Summary Table to analyze the kernel**
- Consider using multiple streams and asynchronous memory copies to overlap memory copy & kernel execution
- Consider using page-locked memory to attain higher bandwidth between host and device memory. Overuse of pinned memory should be avoided as it may reduce overall system performance.
  Refer to the "Page-Locked Host Memory" section in the "CUDA C Runtime" chapter of the CUDA C Programming Guide for more details.

Figure 24: Context Level Analysis

# KERNEL LEVEL ANALYSIS

To view the kernel analysis for any kernel, double click the kernel name in the summary table. A new pop up window analyzes that particular kernel in greater detail as mentioned below:

## Limiting Factor Identification Tab


convolutionColumnsKernel analysis - [Session4 - Device_0 - Context_0]
File
View
Analysis
Limiting Factor Identification
Memory Throughput Analysis
Instruction throughput Analysis
Occupancy Analysis
Show all columns

**Analysis for kernel convolutionColumnsKernel on device GeForce GTX 480**

**Summary profiling information for the kernel:**
Number of calls: 17
Minimum GPU time(us): 1652.45
Maximum GPU time(us): 1653.60
Average GPU time(us): 1653.03
GPU time (%): 17.36
Grid size: [192 48 1]
Block size: [16 8 1]

**Limiting Factor**
Achieved Instruction Per Byte Ratio: 7.70 ( Balanced Instruction Per Byte Ratio: 4.51 )
Achieved Occupancy: 0.66 ( Theoretical Occupancy: 0.67 )
IPC: 1.56 ( Maximum IPC: 2 )
Achieved global memory throughput: 48.08 ( Peak global memory throughput(GB/s): 112.00 )

**Hint(s)**

- The achieved instructions per byte ratio for the kernel is greater than the balanced instruction per byte ratio for the device. Hence, the **kernel is likely compute bound.** For details, click on **Instruction Throughput Analysis.**

**Factors that may affect analysis**

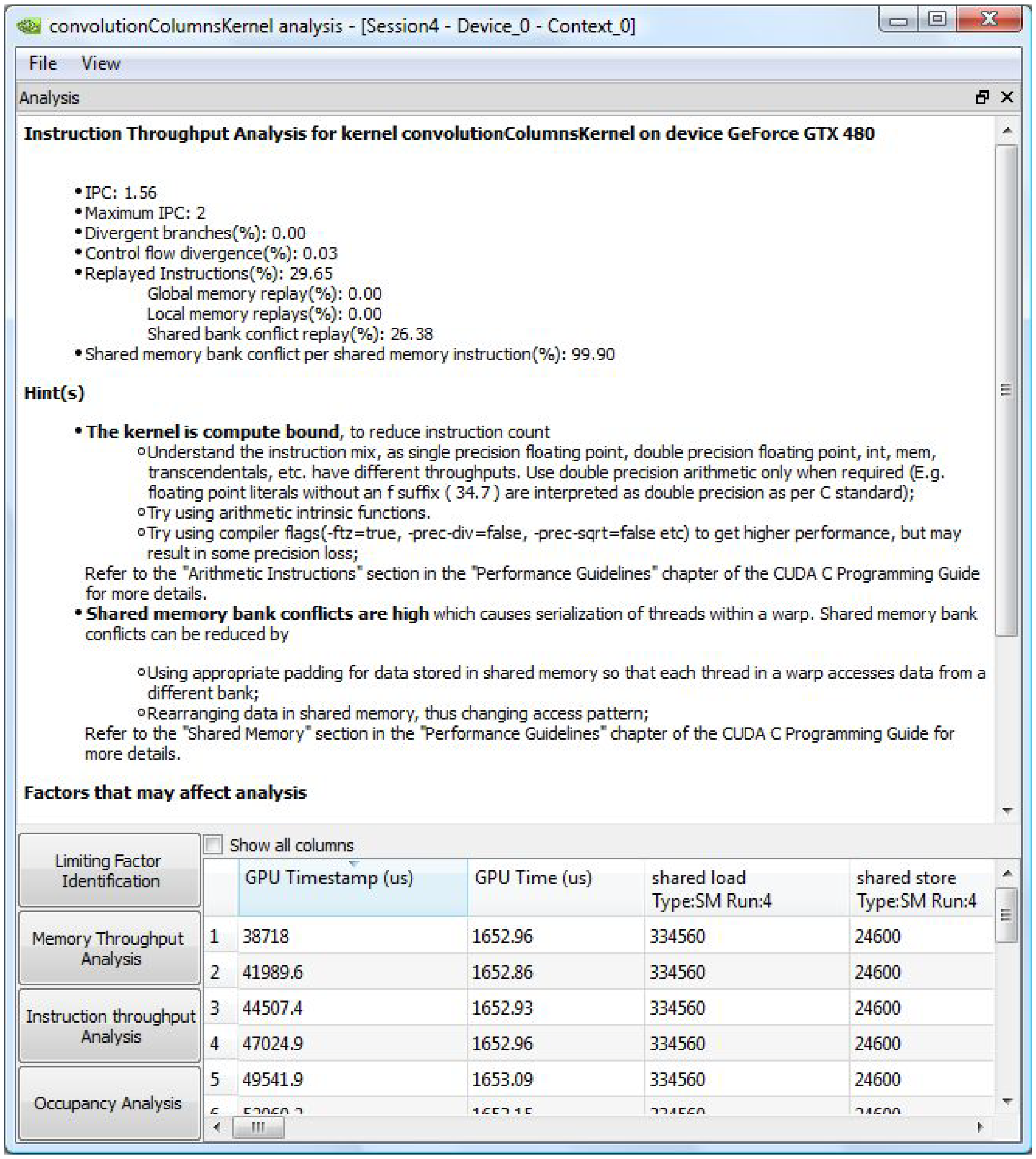
- The counters of type SM are collected only for 1 multiprocessor in the chip and the values are extrapolated to get the behavior of entire GPU assuming equal work distribution. This may result in some inaccuracy in the analysis in some cases.
- The counters for some derived stats are collected in different runs of application. This may cause some inaccuracy in the derived statistics as the blocks scheduled on each multiprocessor may be different for each run and for some applications the behavior changes for each run.
- The derived statistics instruction per byte ratio and IPC assume that all instructions are single precision floating point instructions. If the application uses double precision floating point instructions then the limiting factor predicted here may be incorrect.

| | GPU Timestamp (us) | GPU Time (us) | instructions issued Type:SM Run:7 | active warps Type:SM Run |
|---|---|---|---|---|
| 1 | 38718 | 1652.96 | 1361195 | 27380312 |
| 2 | 41989.6 | 1652.86 | 1359298 | 27316806 |
| 3 | 44507.4 | 1652.93 | 1359381 | 27372609 |
| 4 | 47024.9 | 1652.96 | 1359365 | 27396901 |
| 5 | 49541.9 | 1653.09 | 1361669 | 27338841 |

Figure 25: Limiting Factor Identification tab

- Limiting Factor Identification – In the Analysis window, this default tab displays important statistics for the kernel for example the min/max/avg gpu time for kernel at each call and block/grid dimensions amongst others. It shows
  - The performance limiting factor for the kernel which indicates if the application is more compute bound or memory bandwidth bound.
  - The key parameters for example IPC (Instructions per Cycle), Memory throughput and occupancy of the kernel and compares them with the corresponding peak values for that device which helps in identifying the limiting factor for the kernel.

## Instruction Throughput Analysis Tab


convolutionColumnsKernel analysis - [Session4 - Device_0 - Context_0]
File
View
Analysis
Limiting Factor Identification
Memory Throughput Analysis
Instruction throughput Analysis
Occupancy Analysis
Show all columns

**Instruction Throughput Analysis for kernel convolutionColumnsKernel on device GeForce GTX 480**

- IPC: 1.56
- Maximum IPC: 2
- Divergent branches(%): 0.00
- Control flow divergence(%): 0.03
- Replayed Instructions(%): 29.65
  - Global memory replay(%): 0.00
  - Local memory replays(%): 0.00
  - Shared bank conflict replay(%): 26.38
- Shared memory bank conflict per shared memory instruction(%): 99.90

**Hint(s)**

- **The kernel is compute bound**, to reduce instruction count
  - Understand the instruction mix, as single precision floating point, double precision floating point, int, mem, transcendentals, etc. have different throughputs. Use double precision arithmetic only when required (E.g. floating point literals without an f suffix ( 34.7 ) are interpreted as double precision as per C standard);
  - Try using arithmetic intrinsic functions.
  - Try using compiler flags(-ftz=true, -prec-div=false, -prec-sqrt=false etc) to get higher performance, but may result in some precision loss;

  Refer to the "Arithmetic Instructions" section in the "Performance Guidelines" chapter of the CUDA C Programming Guide for more details.
- **Shared memory bank conflicts are high** which causes serialization of threads within a warp. Shared memory bank conflicts can be reduced by
  - Using appropriate padding for data stored in shared memory so that each thread in a warp accesses data from a different bank;
  - Rearranging data in shared memory, thus changing access pattern;

  Refer to the "Shared Memory" section in the "Performance Guidelines" chapter of the CUDA C Programming Guide for more details.

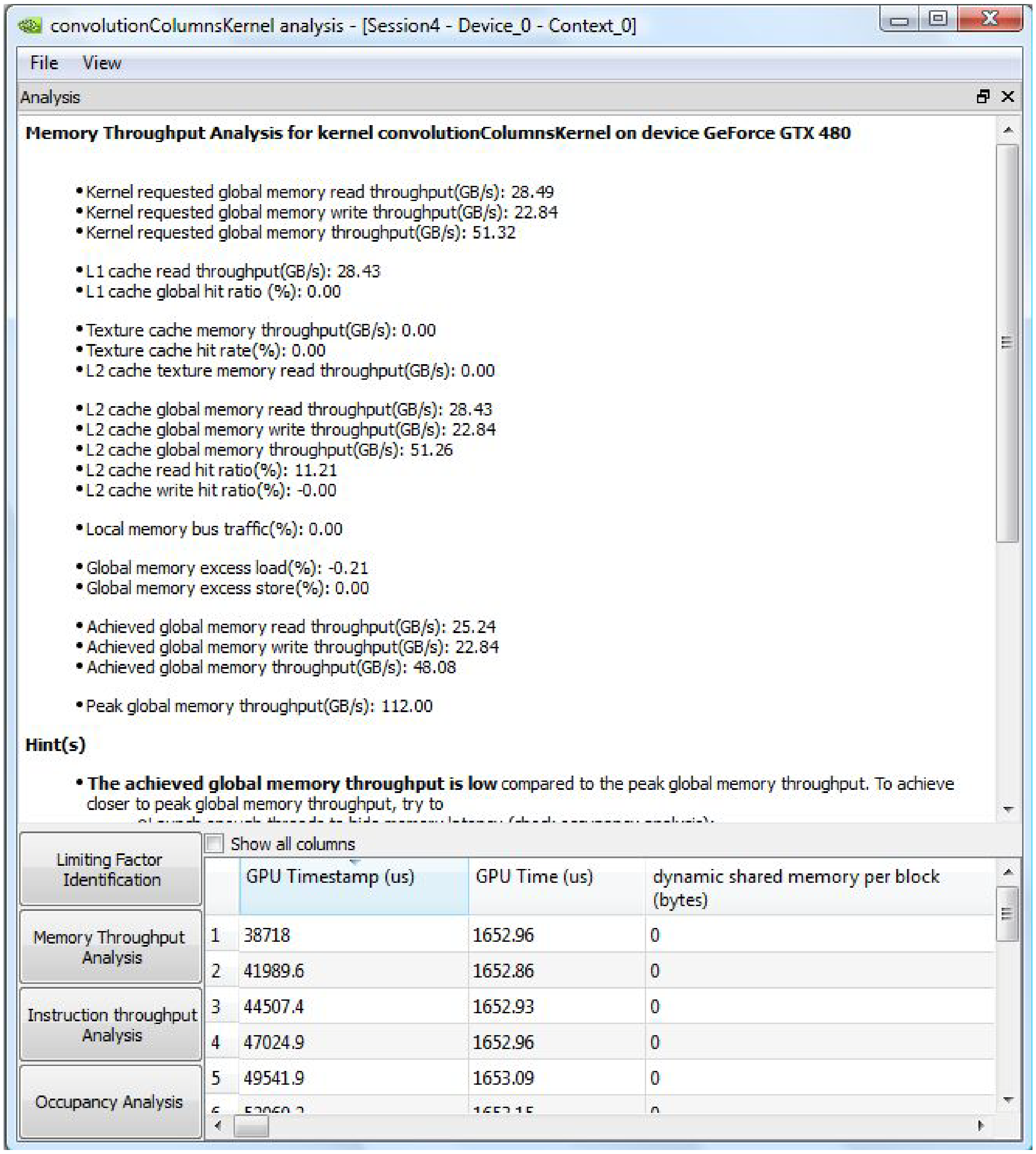
**Factors that may affect analysis**

| | GPU Timestamp (us) | GPU Time (us) | shared load Type:SM Run:4 | shared store Type:SM Run:4 |
|---|---|---|---|---|
| 1 | 38718 | 1652.96 | 334560 | 24600 |
| 2 | 41989.6 | 1652.86 | 334560 | 24600 |
| 3 | 44507.4 | 1652.93 | 334560 | 24600 |
| 4 | 47024.9 | 1652.96 | 334560 | 24600 |
| 5 | 49541.9 | 1653.09 | 334560 | 24600 |

Figure 26: Instruction Throughput Analysis tab

- Instruction Throughput Analysis – Gives instruction throughput analysis. It tries to identify the amount of divergence and serialization in the kernel by analyzing control flow divergence and the reasons for instructions replayed. It also gives hints to reduce serialization and improve IPC.

## Memory Throughput Analysis Tab



convolutionColumnsKernel analysis - [Session4 - Device_0 - Context_0]

File View

Analysis

**Memory Throughput Analysis for kernel convolutionColumnsKernel on device GeForce GTX 480**

- Kernel requested global memory read throughput(GB/s): 28.49
- Kernel requested global memory write throughput(GB/s): 22.84
- Kernel requested global memory throughput(GB/s): 51.32

- L1 cache read throughput(GB/s): 28.43
- L1 cache global hit ratio (%): 0.00

- Texture cache memory throughput(GB/s): 0.00
- Texture cache hit rate(%): 0.00
- L2 cache texture memory read throughput(GB/s): 0.00

- L2 cache global memory read throughput(GB/s): 28.43
- L2 cache global memory write throughput(GB/s): 22.84
- L2 cache global memory throughput(GB/s): 51.26
- L2 cache read hit ratio(%): 11.21
- L2 cache write hit ratio(%): -0.00

- Local memory bus traffic(%): 0.00

- Global memory excess load(%): -0.21
- Global memory excess store(%): 0.00

- Achieved global memory read throughput(GB/s): 25.24
- Achieved global memory write throughput(GB/s): 22.84
- Achieved global memory throughput(GB/s): 48.08

- Peak global memory throughput(GB/s): 112.00

**Hint(s)**

- **The achieved global memory throughput is low** compared to the peak global memory throughput. To achieve closer to peak global memory throughput, try to

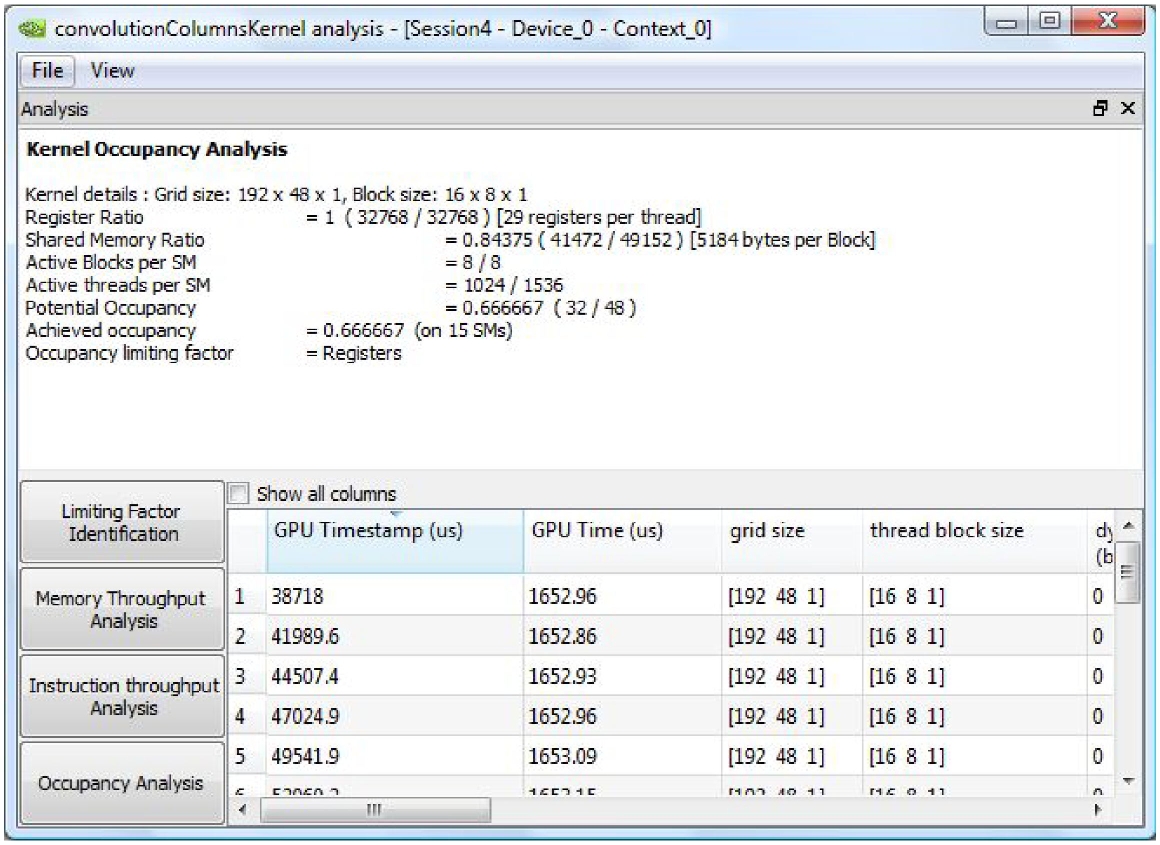
Limiting Factor Identification
Memory Throughput Analysis
Instruction throughput Analysis
Occupancy Analysis
Show all columns

| | GPU Timestamp (us) | GPU Time (us) | dynamic shared memory per block (bytes) |
|---|---|---|---|
| 1 | 38718 | 1652.96 | 0 |
| 2 | 41989.6 | 1652.86 | 0 |
| 3 | 44507.4 | 1652.93 | 0 |
| 4 | 47024.9 | 1652.96 | 0 |
| 5 | 49541.9 | 1653.09 | 0 |

Figure 27: Memory Throughput Analysis

- Memory Throughput Analysis - Gives memory throughput analysis. This gives derived statistics at all the levels in memory hierarchy for example throughput at each level L1 cache, L2 cache, Texture cache and global memory, the hit ratio, and extra memory fetched/store due to coalescing issues. It also provides hints about how to increase the memory throughput and remove some other issues in kernel like register spilling.

## Occupancy Analysis


convolutionColumnsKernel analysis - [Session4 - Device_0 - Context_0]
File
View
Analysis
Limiting Factor Identification
Memory Throughput Analysis
Instruction throughput Analysis
Occupancy Analysis
Show all columns

**Kernel Occupancy Analysis**

```
Kernel details : Grid size: 192 x 48 x 1, Block size: 16 x 8 x 1
Register Ratio              = 1  ( 32768 / 32768 ) [29 registers per thread]
Shared Memory Ratio                   = 0.84375 ( 41472 / 49152 ) [5184 bytes per Block]
Active Blocks per SM                  = 8 / 8
Active threads per SM                 = 1024 / 1536
Potential Occupancy                   = 0.666667  ( 32 / 48 )
Achieved occupancy          = 0.666667  (on 15 SMs)
Occupancy limiting factor   = Registers
```

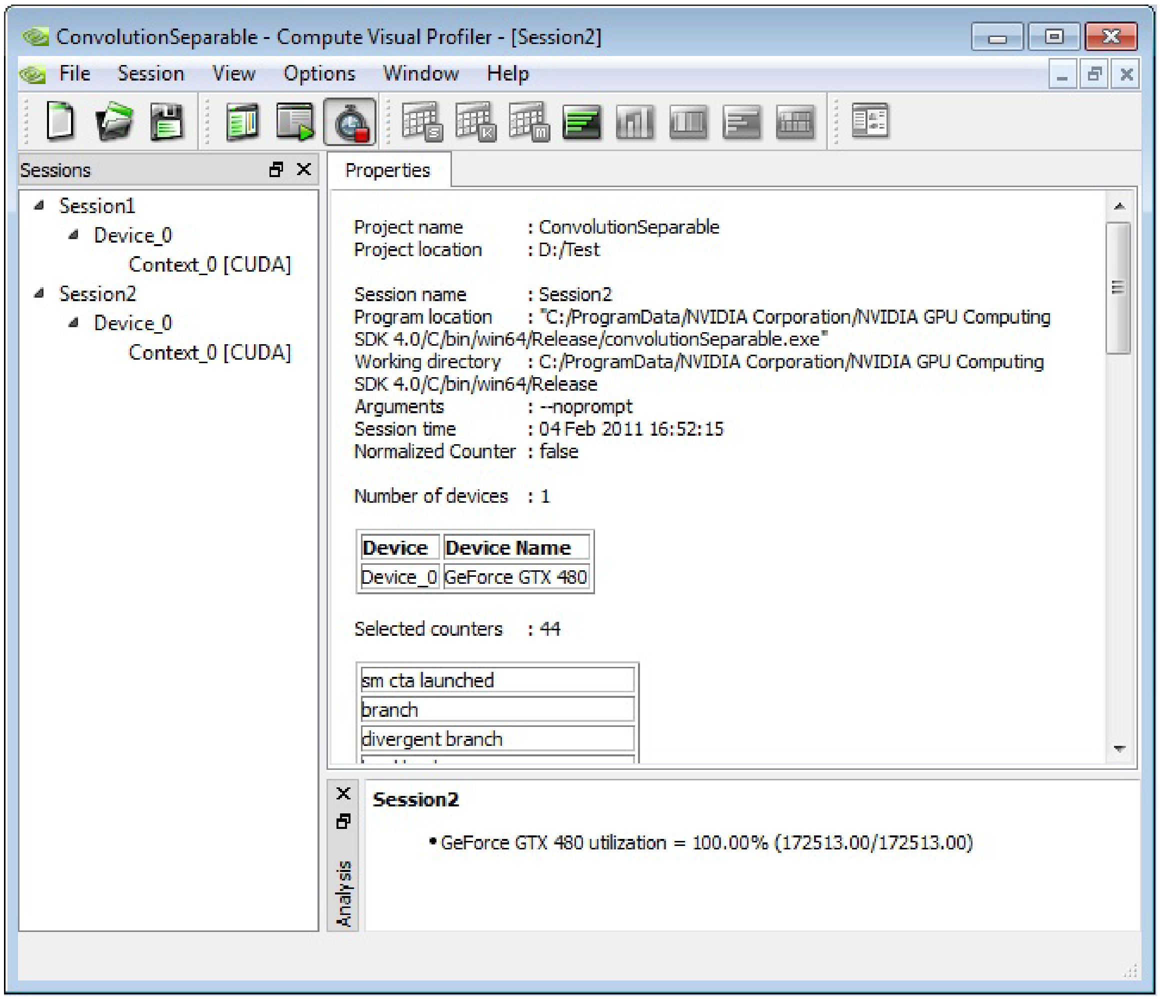
| | GPU Timestamp (us) | GPU Time (us) | grid size | thread block size | dy (b |
|---|---|---|---|---|---|
| 1 | 38718 | 1652.96 | [192 48 1] | [16 8 1] | 0 |
| 2 | 41989.6 | 1652.86 | [192 48 1] | [16 8 1] | 0 |
| 3 | 44507.4 | 1652.93 | [192 48 1] | [16 8 1] | 0 |
| 4 | 47024.9 | 1652.96 | [192 48 1] | [16 8 1] | 0 |
| 5 | 49541.9 | 1653.09 | [192 48 1] | [16 8 1] | 0 |

Figure 28: Kernel Occupancy Analysis

- Occupancy Analysis – This gives the theoretical kernel occupancy and identifies the limiting factor for occupancy. It is calculated using the static parameters of the kernel like launch configuration, shared memory, and register usage.
- The table shown in kernel analysis window displays derived statistics and raw counters for each call for the kernel for respective analysis tab. Clicking **Show all columns** displays all the columns that are available in the profiler table for that kernel.
- Use File->Export table to export the profiler table in csv format, filtered for the kernel.

## SESSION LEVEL ANALYSIS

- Click on the Session name in the **Sessions** tree. This displays the session level analysis in the analysis window.



Project name : ConvolutionSeparable
Project location : D:/Test

Session name : Session2
Program location : "C:/ProgramData/NVIDIA Corporation/NVIDIA GPU Computing SDK 4.0/C/bin/win64/Release/convolutionSeparable.exe"
Working directory : C:/ProgramData/NVIDIA Corporation/NVIDIA GPU Computing SDK 4.0/C/bin/win64/Release
Arguments : --noprompt
Session time : 04 Feb 2011 16:52:15
Normalized Counter : false

Number of devices : 1

| Device | Device Name |
|---|---|
| Device_0 | GeForce GTX 480 |

Selected counters : 44

| |
|---|
| sm cta launched |
| branch |
| divergent branch |

**Session2**

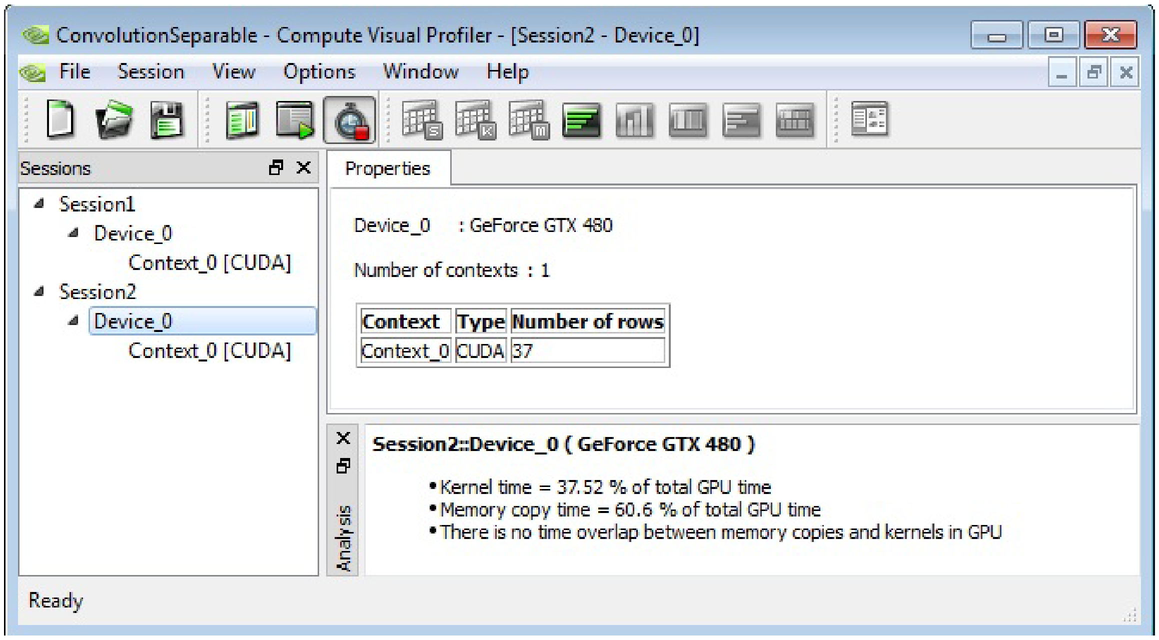
- GeForce GTX 480 utilization = 100.00% (172513.00/172513.00)

Figure 29: Session Level Analysis

- It shows GPU utilization for all the GPUs for that session and provides suitable optimization hints.

## DEVICE LEVEL ANALYSIS

- Click on the device in the **Sessions** tree. This displays device level analysis the analysis window.



Device_0 : GeForce GTX 480

Number of contexts : 1

| Context | Type | Number of rows |
|---|---|---|
| Context_0 | CUDA | 37 |

**Session2::Device_0 ( GeForce GTX 480 )**

- Kernel time = 37.52 % of total GPU time
- Memory copy time = 60.6 % of total GPU time
- There is no time overlap between memory copies and kernels in GPU

Figure 30: Device Level Analysis

- It shows GPU utilization for the device by showing the distribution of GPU time over kernel execution and memory copy and it also gives the overlap time between memory copy and kernel execution. It also provides suitable hints towards improving the application performance.

# COMPUTE VISUAL PROFILER TABLES

## PROFILER OUTPUT TABLE

Whenever a CUDA program is run with profiling enabled, Compute Visual Profiler produces a Profiler Output table by default. Table 2 shows the composition of a Profiler output table.

Table 2. Profiler Output Table

| GPU Timestamp | Method | GPU Time | CPU Time | Stream Id | Columns for kernel options (See Table 3) | Columns for memcopy options (See Table 4) | Columns for Profile Counters |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

Profiler Output Table columns are described below:

- **GPU Timestamp:** Start time stamp.
- **Method:** GPU Method name. This is either memcpy* for memory copies or the name of a GPU kernel. Memory copies have a suffix that describes the type of a memory transfer, e.g. memcpyDToHasync means an asynchronous transfer from Device memory to Host memory.
- **GPU Time:** Execution time for the method on the GPU.
- **CPU Time:** Sum of GPU time and CPU overhead to launch the GPU Method. At the driver generated data level, the CPU Time is only the CPU overhead to launch the Method for non-blocking Methods. For blocking methods it is the sum of GPU time and CPU overhead. All kernel launches by default are non-blocking. But if any of the

profiler counters are enabled kernel launches are blocking. Asynchronous memory copy requests in different streams are non-blocking.

- **Stream Id** : Identification number for the stream
- **Kernel Options Columns:** The columns are described as follows:

Table 3 shows the columns that are displayed for kernel methods.

## Table 3. Kernel Options Columns

| Occupancy | Profiler Counters | GridSize [X, Y, Z] | Thread Block Size [X, Y, Z] | Dyn smem per block | Sta smem per block | Reg per thread |
|---|---|---|---|---|---|---|
| | | | | | | |
| | | | | | | |

- **Occupancy** : Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of active warps.
- **Profiler counters**: Refer to the Interpreting Profiler Counters section for a list of counters supported.
- **GridSize[X, Y, Z]**: Number of blocks in the grid along dimensions X, Y and Z displayed as [num_blocks_X, num_blocks_Y, num_blocks_Z] in a single column.
- **Block size[X, Y, Z]**: Number of threads in a block along dimensions X, Y, and Z displayed as [num_threads_X, num_threads_Y, num_threads_Z] in a single column.
- **dyn smem per block**: Dynamic shared memory size per block in bytes.
- **sta smem per block**: Static shared memory size per block in bytes.
- **reg per thread**: Number of registers per thread.

Table 4 shows the columns that are displayed for **memcopy** options.

## Table 4. memcopy Options Columns

| Method | #Calls | Host mem transfer type |
|---|---|---|
| | | |
| | | |
| | | |

## Profiler Table Context Sensitive Menu

Right-clicking anywhere in the Profiler Output table window brings up a menu with the following options:

- **Profiler counter plot**: Display the profiler counter plot for the method in the current row.
- **Column plot**: Display the column plot for the current column.
- **Export**: Export the profiler data to a CSV format file.
- **Copy**: Copy the selected table cells to the clipboard.

## SUMMARY TABLE

The Summary table menu is described in the section entitled, "Summary Table".

A typical summary table is shown in Table 5. See the section entitled, "Summary Table Tab" on how to select columns to be displayed.

Table 5. Summary Table

| Method | # Calls | GPU usec | CPU usec | %GPU time | glob mem read throughput (GB/s) | glob mem write throughput (GB/s) | glob mem overall throughput (GB/s) | instruction throughput | retired ipc | Warps per cycle | l1 global load hit rate |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | |
| | | | | | | | | | | | |
| | | | | | | | | | | | |

# COMPUTE VISUAL PROFILER PLOTS

For a selected context in the left frame (Sessions tree view):

**On the main menu→ Click on View**
Various plots supported by the Compute Visual Profiler are displayed. Compute Visual Profiler supports the following plots:

- Summary plot
  - GPU time summary plot
  - Device level summary plot
  - Session level summary plot
- GPU time height plot
- GPU Time Width plot
- Comparison Summary plot
- CUDA API Trace

## GPU Time Summary Plot

For a selected context in the left frame (Sessions tree view):

**On the main menu→ Click on View→ Click on GPU time summary plot**

The Summary profiling data bar plot has one bar for each method. The bars are sorted in decreasing GPU time and the bar length is proportional to cumulative GPU time for a method

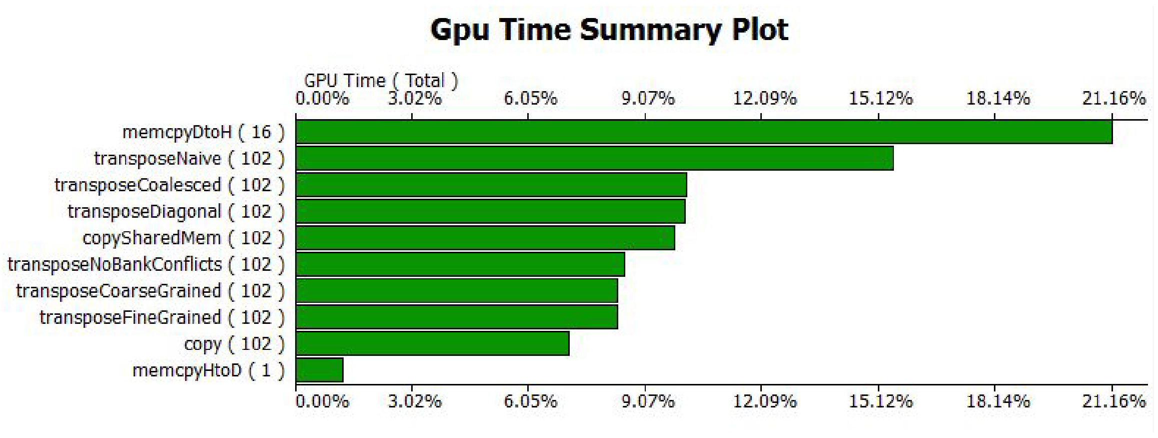


Figure 31. Summary Plot

## Device Level Summary Plot

For a selected device in the left frame (Sessions tree view):

**On the main menu→ Click on View→ Click on Device level summary plot**

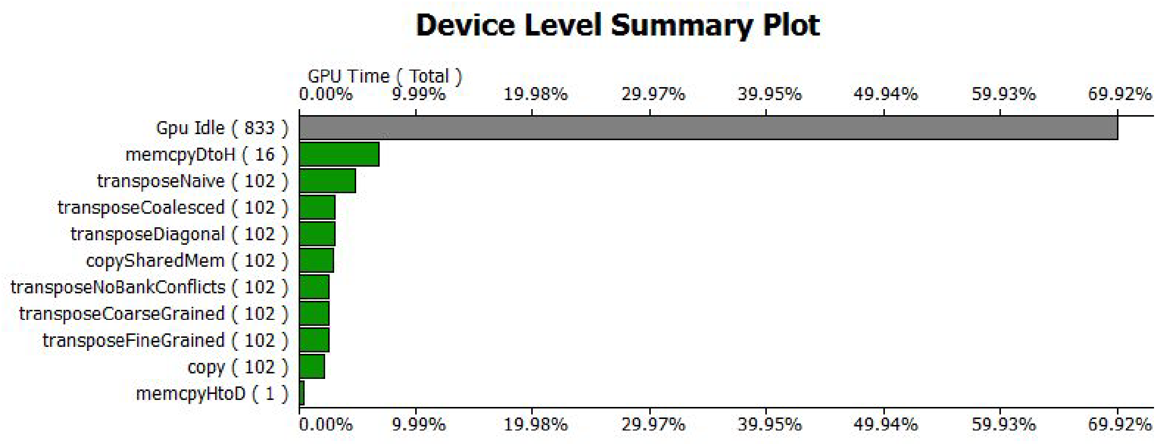


Figure 32. Device Level Summary Plot

The Device level summary plot has one bar for each method. Bars are sorted in decreasing GPU time. The bar length is proportional to the cumulative GPU time for a method across all contexts for a device.

## Session Level Summary Plot

For a selected session in the left frame (Sessions tree view):

**On the main menu→ Click on View→ Click on Session level summary plot**

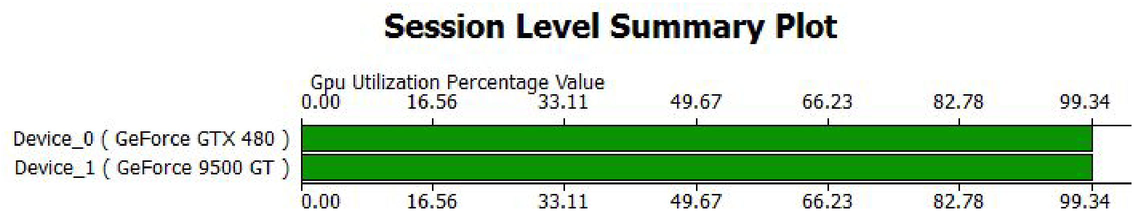


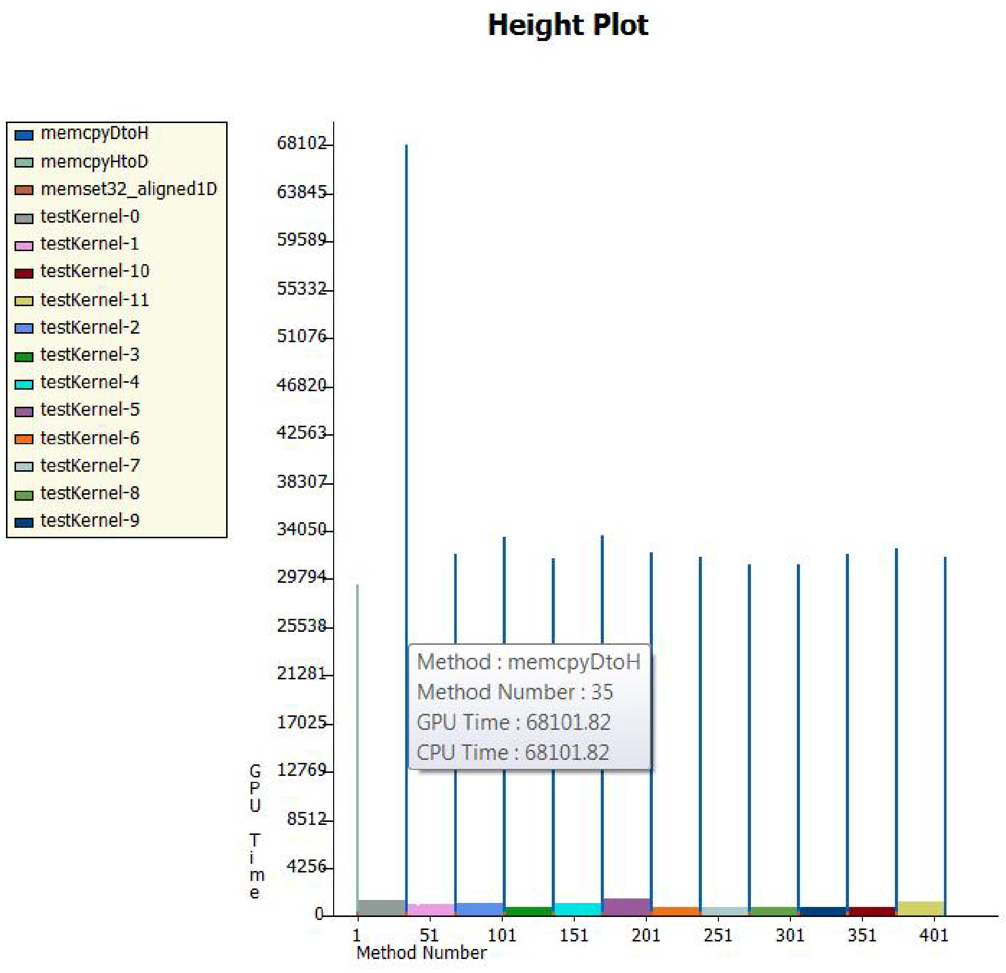
Figure 33. Session Level Summary Plot

The Session level summary plot has one bar for each device used. The bar length is proportional to GPU utilization which is the proportion of time that GPU spent on the execution of a particular method to the total time interval from GPU start to end. The values are presented in percentage format.

## GPU Time Height Plot

For a selected context in the left frame (Sessions tree view):

**On the main menu→ Click on View→ Click on GPU time height plot**



Method : memcpyDtoH
Method Number : 35
GPU Time : 68101.82
CPU Time : 68101.82

Figure 34. GPU Time Height Plot

The GPU time height plot is a bar diagram in which the height of each bar is proportional to the GPU time for a method; a different bar color is assigned for each method. The width of each bar is fixed and the bars are displayed in the order in which the methods are executed. When the Fit In Window (**Options→Session View Settings→**Click on **Height Plot tab→**Check **Fit In Window** box) option is enabled the display is adjusted so as to fit all the bars in the displayed window width. In this case bars for multiple methods can overlap. The overlapped bars are displayed in decreasing order of height so that all the different bars are visible. When the Show CPU Time option (**Options**→**Session View Settings**→Click on **Height Plot** tab→Check the **Show CPU Time** box) is enabled the CPU time is shown as a bar in a different color on top of the GPU time bar. The height of this bar is proportional to the difference of CPU time and GPU time for the method.

A legend which shows the color assignment for different methods is displayed if the Show Legend box is checked.

The plot can customized as described in the section entitled, "Height Plot Tab"; the dialog box with options is shown in Figure 19.

## GPU Time Width Plot

For a selected context in the left frame (Sessions tree view):

**On the main menu→ Click on View→ Click on GPU time width plot**

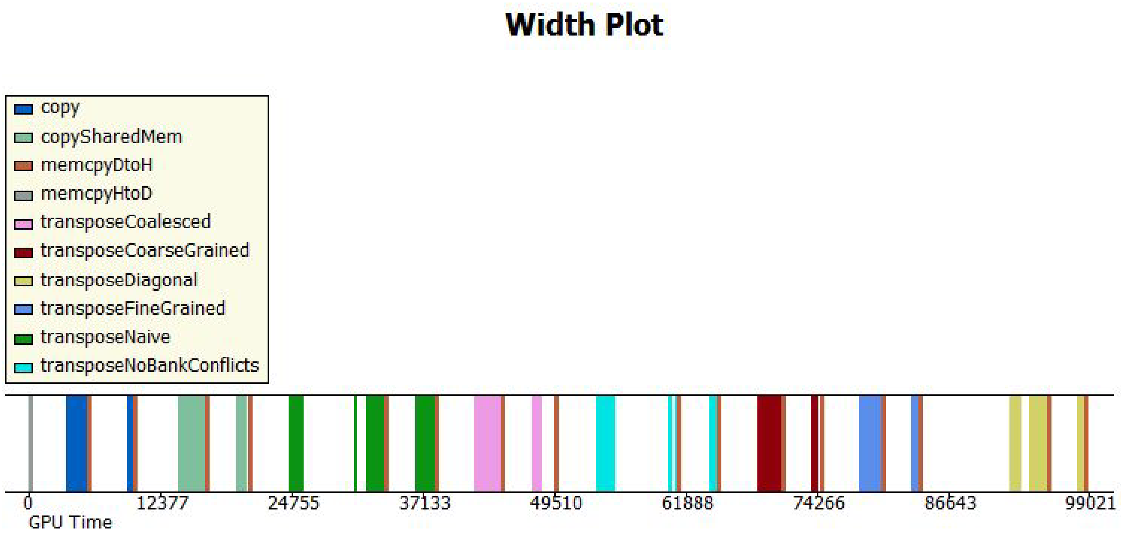


Figure 35. GPU Time Width Plot

The GPU time width plot is a bar diagram in which the width of each bar is proportional to the GPU time for a method. A different bar color is assigned for each method. A legend which shows the color assignment for different methods is displayed. The bars are displayed in the order in which the methods are executed. When time stamps are enabled the bars are positioned based on the time stamp. The height of each bar is based on the option chosen.

The plot can customized as described in the section entitled, "Width Plot Tab"; the dialog box with options is shown in Figure 20.

## Profiler Counter Bar Plot

**Go to the Profiler Output Tab →Right-click on any cell in the Profiler Table *except cells in the Method column* →Select Profiler Counter Plot**

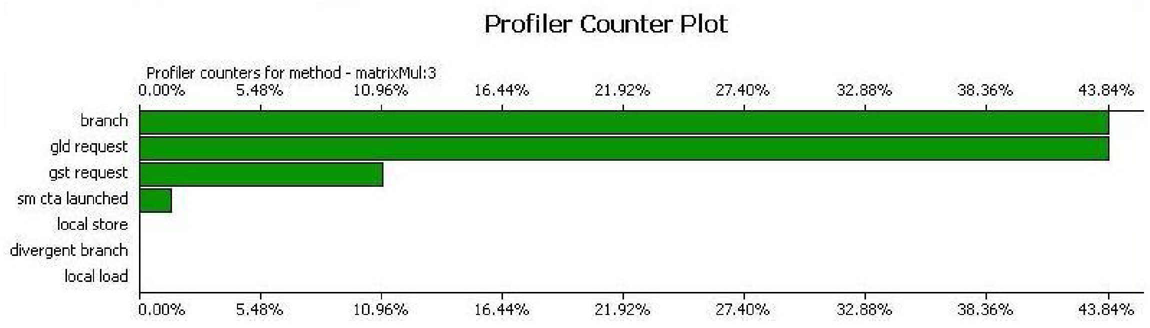


Figure 36. Profiler Counter Plot

The Profiler Counter bar plot displays profiler counter values for a GPU Method from the profiler output table or the summary table. There is one bar for each profiler counter. Bars are sorted in decreasing profiler counter value. The bar length is proportional to profiler counter value.

## Profiler Output Table Column Bar Plot

**Go to the Profiler Output Tab →Right-click on any cell in the Profiler Table→Select Column Plot**

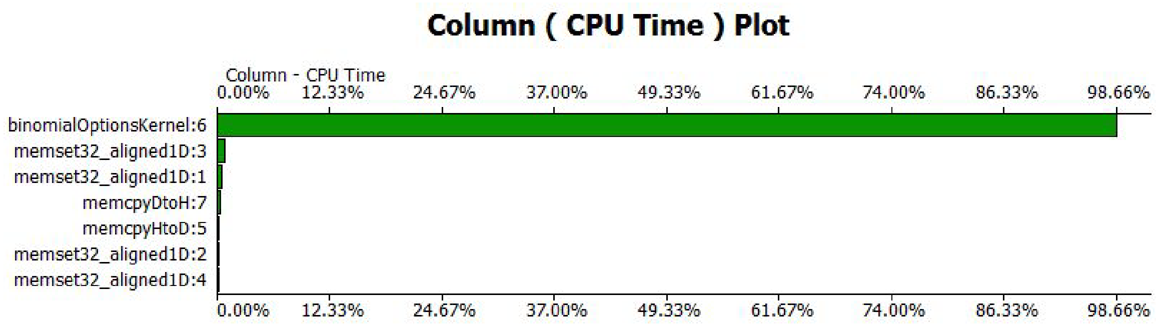


Figure 37. Profiler Output Column Plot

The Profiler output table column plot displays a bar graph of the selected column of values from the profiler output table or summary table. There is one bar for each row in the table. Bars are sorted in decreasing column value. The bar length is proportional to column value. Figure 37 displays the CPU time since a cell on the CPU time was selected.

## Comparison Summary Plot

The Comparison Summary plot can be used to compare GPU time summary data for two sessions: a **base session** and a **compare session**. The base session is the session with respect to which comparison is done. The other session which is selected for comparison is called the compare Session.

As shown in Figure 38, the dialog box **Select Device of Compare Session** is presented for selecting the device on which the sessions are compared, if multiple devices are present.

Note that in case of a single device the **Select Device of Compare Session** dialog box will not appear.

**Select the device→Click on OK.**

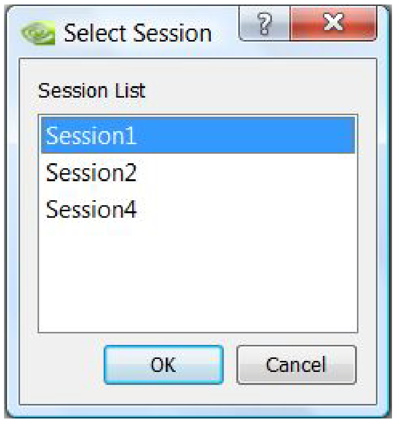


Figure 38. Select Device

Next, a dialog box as shown in Figure 39, allows you to select the columns that may be used for comparison.

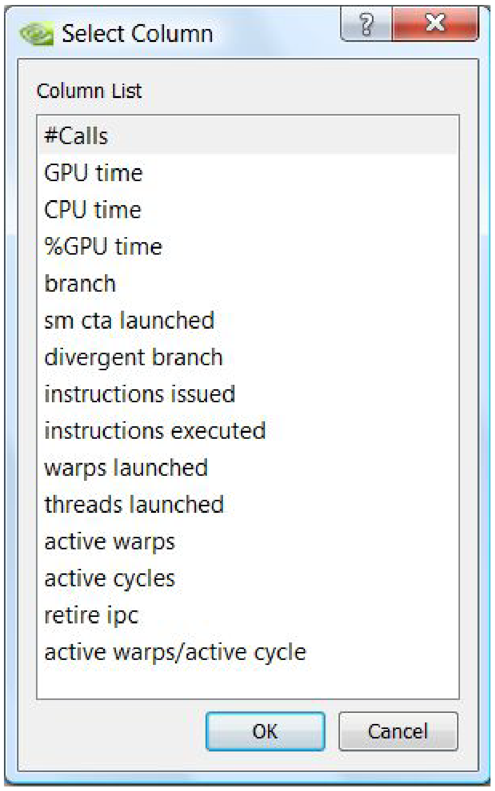


Figure 39. Select Column Screen for Comparison Summary Plot

Figure 40 shows the Comparison Summary Plot. Selected columns for matching kernels from the two sessions are grouped together. For each matched kernel from the compare session, a percentage increment or decrement with respect to base session is displayed at the right end of the bar. In addition to the matched pairs, the unmatched kernels' column values are shown. At the bottom of the plot two bars with total column values for the two sessions are shown.

If multiple contexts exist, a context selection dialog is presented along with a column selection dialog. Based on these selections, the comparison summary plot is displayed. The plot groups matching methods from two contexts (chosen from base and compare sessions) and plots the values of the selected columns together. In addition, non-matching methods are plotted separately. Finally the total values are compared at the bottom.

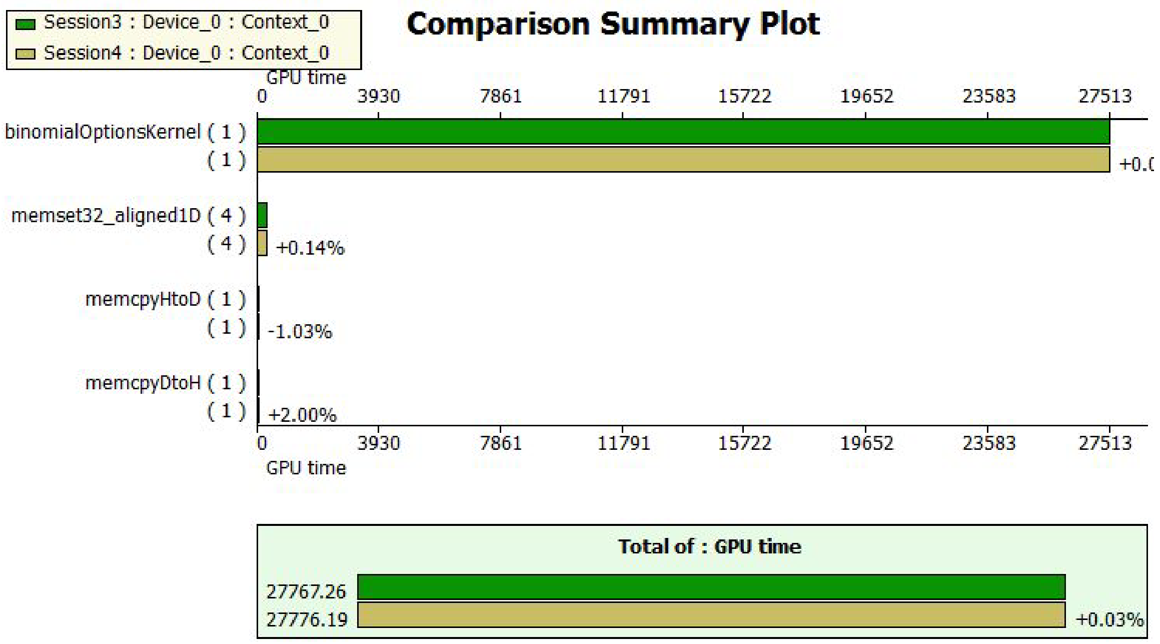


Figure 40. Comparison Summary Plot

## CUDA API Trace

Figure 41 shows a sample CUDA API Trace view.

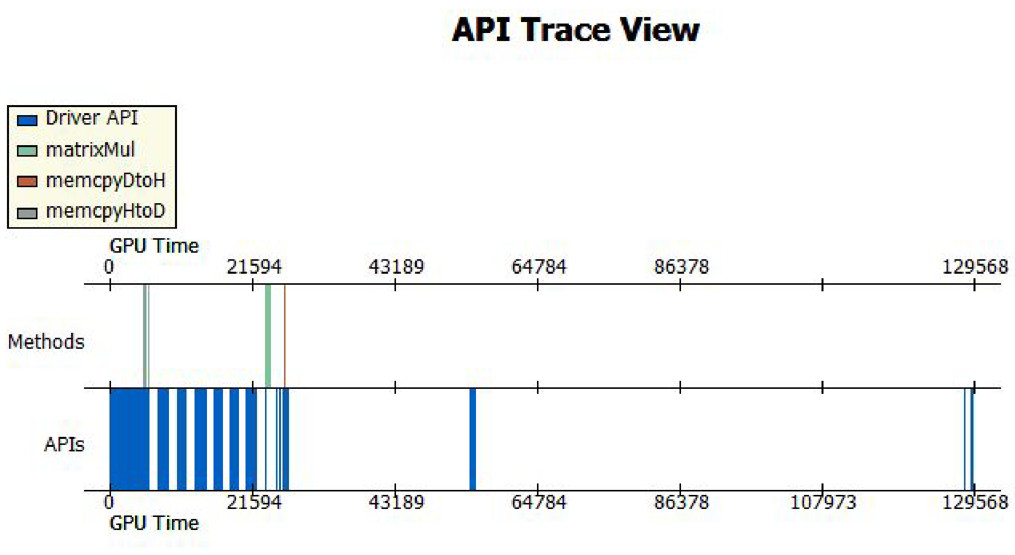


Figure 41. CUDA API Trace

The CUDA API trace helps the user to understand the CPU side overhead for CUDA driver API calls and specifically to understand the overhead involved for each kernel launch and memory transfer request.. Capturing of CUDA Driver API calls can be enabled by selecting **API trace** in the **Session settings** dialog.

To view CUDA API Trace for a context:

**On the main menu→ Click on View →Select CUDA API trace**

Or,

**Go to the left frame Sessions tree view → Right-click on context→Select CUDA API trace**

The API trace view displays two horizontal rows of bars. The top row of bars shows the GPU methods and the bottom row of bars shows the CUDA driver API functions. Each GPU method or API is represented by a bar with a width proportional to the time of execution. The bars are displayed in time order along the horizontal direction based on the start time. A different color is assigned to each GPU method and all APIs are shown in the same color. Consult the legend for the color used for different GPU methods and for APIs.

The attributes for a GPU method or an API can be viewed by pointing the cursor on the bar. The following attributes are displayed for a CUDA driver API:

- **API name**: Name of CUDA driver API function
- **Context ID**: GPU context ID
- **Thread ID**: CPU thread ID
- **Process ID**: CPU process ID
- **Stream ID**: GPU steam ID
- **Return value**: API call return value
- **Start time stamp**: Start time of an API call in micro seconds
- **Time duration**: Time duration for execution of a API in micro seconds

# COMPUTE VISUAL PROFILER COUNTERS

## INTERPRETING COUNTER VALUES

Counter values obtained from the Compute Visual Profiler are not the same as numbers obtained by inspecting kernel code. Compute Visual Profiler values are best used to identify relative performance differences between un-optimized and optimized code. For example, if for the initial version of the program the profiler reports **`N`** non-coalesced global loads, it is easy to see if the optimized code produces less than **`N`** non-coalesced loads. In most cases, the goal is to make **`N`** go to **`0`**, so the counter value is useful for tracking progress toward this goal.

Performance counter values represent events within a thread warp; they do not correspond to individual thread activity. For example, a divergent branch within a thread warp will increment the **`divergent_branch`** counter by one. Therefore the final counter value contains information for ***all*** divergent branches in ***all*** warps. In addition, the profiler can only target one of the multiprocessors in the GPU, so the counter values will not correspond to the total number of warps launched for a particular kernel. For this reason, when using the performance counter options in the profiler the user should always launch enough threads blocks to ensure that the target multiprocessor is given a consistent percentage of the total work. In practice for consistent results, it is best to launch at least 2 times as many blocks as there are multiprocessors in the device on which you are profiling.

Note that the counter values for the same application can be different across different runs even on the same setup since it depends on the number of thread blocks which are executed on each multiprocessor. For consistent results it is best to have number of blocks for each kernel launched to be at least equal to or a multiple of the total number of multiprocessors on a compute device. In other words when profiling the grid configuration should be chosen such that all the multiprocessors are uniformly loaded i.e. the number of blocks launched on each multiprocessor is same and also the amount of work of interest per block is the same. This will result in better accuracy of extrapolated counts, such as memory and instruction throughput, and will also provide more consistent results from one run to the next run.

In every application run only a few counter values can be collected. The number of counters depends on the specific counters selected. Compute Visual Profiler executes the application multiple times to collect all the counter values. Note that in case the number blocks in a kernel is less than or not a multiple of the number of multiprocessors the counters values across multiple runs will not be consistent.

Refer to the Best Practices Guides for CUDA and OpenCL for further details.

## PROFILER COUNTERS FOR A SINGLE MULTIPROCESSOR (SM)

These counter values are a cumulative count for all thread blocks which were run on multiprocessor zero. Note that the multiprocessor single-instruction multi-thread unit (SIMT) creates, manages, schedules, and executes threads in groups of 32 threads called warps. These counters are incremented by one for each warp.

## PROFILER COUNTERS FOR ALL MULTIPROCESSORS IN A TPC

Profiler counter values for all multiprocessors in a Texture Processing Cluster (TPC) are a cumulative count for all thread blocks which were run on multiprocessors within TPC zero. There are two multiprocessors per TPC on compute devices with compute capability less than 1.3, there are three multiprocessors per TPC on compute devices with compute capability 1.3 and one multiprocessor per TPC on compute devices with compute capability 2.0. The number of multiprocessors per TPC is not dependent on compute capability.

A **coalesced access** is said to occur when simultaneous global memory accesses by threads in a half-warp, during the execution of a single read or write instruction, can be combined into a single memory transaction of 32, 64, or 128 bytes.

If the global memory access by all threads of a half-warp does not fulfill the coalescing requirements it is called a **non-coalesced access** and a separate memory transaction is issued for each thread and throughput is significantly reduced. The coalescing requirements on devices with compute capability 1.2 and higher are different from devices with compute capability 1.0 or 1.1. Refer to the *CUDA C Programming Guide* for details. The profiler counters related to global memory count the number of global memory accesses or memory transactions and they are not per warp. They provide counts for all global memory requests initiated by warps running on a TPC.

## NORMALIZED COUNTER VALUES

When the "Normalize Counters" option is selected (see Figure 12) all counter values are normalized and per block counts are shown. This option is currently supported only for compute devices with compute capability less than 2.0.

For single multiprocessor counters the counter value is divided by the number of thread blocks which were run on multiprocessor 0. The profiler counter "**`sm_cta_launched`**" is used to count thread blocks which were run on multiprocessor 0.

For TPC counters the counter value is divided by the number of thread blocks which were run on TPC 0. The profiler counter "**`cta_launched`**" is used to count thread blocks which were run on multiprocessors in TPC 0.

The counter value is set to zero in the following cases:

- The number of blocks launched on the multiprocessor(s) being profiled is zero. This can happen when the number of blocks launched for a kernel is less than the total number of multiprocessors on a compute device.
- The counter value is less than the number of blocks launched on the multiprocessor(s) being profiled. The normalized fractional value less than one is truncated to zero.

If any counter value is set to zero a warning is displayed at the end of the application profiling.

Enabling the "**Normalize Counters**" option results in the following:

- more number of application runs are required to collect all counter values as compared to when the option is disabled.
- the "**`cta_launched`**" and "**`sm_cta_launched`**" columns are not shown in the profiler table.

## PROFILER COUNTERS

Table 6 lists the profiler counters supported for different multiprocessor configurations and compute capabilities.

The "**Type**" column specifies one of the following types of counters:

- **SM**: Streaming Multiprocessor
  Counters of this type provide accumulated values for all thread blocks which were run on multiprocessor zero.
- **TPC**: Texture Processing Cluster
  Counters of this type provide accumulated values for all thread blocks which were run on multiprocessors within TPC 0.
- **FB**: Frame Buffer for GPU DRAM or Device Memory
  DRAM and L2 cache counters are categorized as FB type counters. Counters of this type provide accumulated values for all instances of the unit available on the GPU. e.g. the L2 counters provide accumulated values for all the L2 cache units and DRAM counters provide accumulated values for all the DRAM units available on the GPU. When values of these counters are compared with the SM counter values, the SM counter values need to be extrapolated for the total number of thread blocks for the kernel.
  e.g.  ((128 * 'l1 global load miss' * 'grid size')/'sm cta launched') should be approximately equal to ('l2 read requests' * 32) assuming uniform work load across all the multiprocessors.
- **SW**: Counter values obtained by performing code instrumentation on the device code
  SW counters include the counters that count different sizes of memory requests in the kernel. Since these counters are collected for all thread blocks of the kernel, when values of these counters are compared with the SM counter values, SM counter values should be extrapolated for the total number of thread blocks for the kernel.
  e..g. ('gld inst 8bit' + 2 * 'gld inst 16bit' + 4 * 'gld inst 32bit' + 8 * 'gld inst 64bit' + 16 * 'gld inst 128bit') should be approximately equal to ((128 * (l1_global_load_hit + l1_global_load_miss) * 'grid size') / 'sm cta launched') if the memory access pattern is coalesced and assuming uniform work load across all multiprocessors.

In addition to the Compute Visual Profiler, a command line profiling tool called the Compute Command Line Profiler is also supported for compute application profiling. Both tools support similar counters with slightly different nomenclature.

Profiler counters for the command line profiler are indicated in a different font (`Courier`) in Table 6 below.

High level counters supported only in the Visual Profiler are computed in terms of the command line profiler low level counters. Formulas (in `Courier font`) for such high level counters are listed in the **"Description"** column in Table 6 below.

Details regarding the command line profiler low level counters are provided in Table 9 in the section entitled, "Command Line Profiler Counters".

## Table 6. Profiler Counters

| **Visual Profiler Counter Name**<br>**`Command Line Profiler Counter Name`** | **Description** | **Type**<br>**SM= Streaming Multiprocessor**<br>**TPC=Texture Processing Cluster**<br>**FB =Frame Buffer (GPU DRAM or Device Memory)**<br>**SW= Software** | **Compute Capability Support**<br>**Y= Yes**<br>**N= No** | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | **1.0** | **1.1** | **1.2** | **1.3** | **2.0** | **2.1** |
| Branch<br>`branch` | Number of branches taken by threads executing a kernel. This counter is incremented by one if at least one thread in a warp takes the branch. Note that barrier instructions (__syncThreads()) also get counted as branches. | SM | Y | Y | Y | Y | Y | Y |
| divergent branch<br>`divergent_branch` | Number of divergent branches within a warp. This counter is incremented by one if at least one thread in a warp diverges (that is, follows a different execution path) via a data dependent conditional branch. The counter is incremented by one at each point of divergence in a warp. | SM | Y | Y | Y | Y | Y | Y |
| instructions<br>`instructions` | Number of instructions executed. | SM | Y | Y | Y | Y | N | N |
| warp serialize<br>`warp_serialize` | If two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized. This counter gives the number of thread warps that serialize on address conflicts to either shared or constant memory. | SM | Y | Y | Y | Y | N | N |
| sm cta launched<br>`sm_cta_launched` | Number of threads blocks launched on a multiprocessor. | SM | Y | Y | Y | Y | Y | Y |
| gld uncoalesced<br>`gld_incoherent` | Number of non-coalesced global memory loads. | TPC | Y | Y | N | N | N | N |
| gld coalesced<br>`gld_coherent` | Number of coalesced global memory loads. | TPC | Y | Y | N | N | N | N |

| **Visual Profiler Counter Name** **`Command Line Profiler Counter Name`** | **Description** | **Type** **SM= Streaming Multiprocessor** **TPC=Texture Processing Cluster** **FB =Frame Buffer (GPU DRAM or Device Memory)** **SW= Software** | **Compute Capability Support** **Y= Yes** **N= No** | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | **1.0** | **1.1** | **1.2** | **1.3** | **2.0** | **2.1** |
| gld request `gld_request` | Number of global memory load requests. On devices with compute capability 1.3 enabling this counter will result in increased counts for the "instructions" and "branch" counter values if they are also enabled in the same application run. | TPC | N | N | Y | Y | Y | Y |
| gld 32 byte `gld_32b` | Number of 32 byte global memory load transactions; incremented by 1 for each 32 byte transaction. | TPC | N | N | Y | Y | N | N |
| gld 64 byte `gld_64b` | Number of 64 byte global memory load transactions; incremented by 1 for each 64 byte transaction. | TPC | N | N | Y | Y | N | N |
| gld 128 byte `gld_128b` | Number of 128 byte global memory load transactions; incremented by 1 for each 128 byte transaction. | TPC | N | N | Y | Y | N | N |
| gst coalesced `gst_coherent` | Number of coalesced global memory stores. | TPC | Y | Y | N | N | N | N |
| gst request `gst_request` | Number of global memory store requests. On devices with compute capability 1.3 enabling this counter will result in increased counts for the "instructions" and "branch" counter values if they are also enabled in the same application run. | TPC | N | N | Y | Y | Y | Y |
| gst 32 byte `gst_32b` | Number of 32 byte global memory store transactions; incremented by 2 for each 32 byte transaction. | TPC | N | N | Y | Y | N | N |
| gst 64 byte `gst_64b` | Number of 64 byte global memory store transactions; incremented by 4 for each 64 byte transaction. | TPC | N | N | Y | Y | N | N |
| gst 128 byte `gst_128b` | Number of 128 byte global memory store transactions; incremented by 8 for each 128 byte transaction. | TPC | N | N | Y | Y | N | N |
| local load `local_load` | Number of local memory load transactions. Each local load request will generate one transaction irrespective of the size of the transaction. | TPC | Y | Y | Y | Y | Y | Y |
| local store `local_store` | Number of local memory store transactions; incremented by 2 for each 32-byte transaction, by 4 for each 64-byte transaction and by 8 for each 128-byte transaction for compute devices having compute capability 1.x. It is incremented by 1 irrespective of the size of the transaction for compute devices having compute capability 2.0. | TPC | Y | Y | Y | Y | Y | Y |

| **Visual Profiler Counter Name**<br>**`Command Line Profiler Counter Name`** | **Description** | **Type**<br>**SM= Streaming Multiprocessor**<br>**TPC=Texture Processing Cluster**<br>**FB =Frame Buffer (GPU DRAM or Device Memory)**<br>**SW= Software** | **Compute Capability Support**<br>**Y= Yes**<br>**N= No** | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | **1.0** | **1.1** | **1.2** | **1.3** | **2.0** | **2.1** |
| cta launched<br>`cta_launched` | Number of threads blocks launched on a TPC. | TPC | Y | Y | Y | Y | N | N |
| texture cache hit<br>`tex_cache_hit` | Number of texture cache hits. | TPC | Y | Y | Y | Y | N | N |
| texture cache miss<br>`tex_cache_miss` | Number of texture cache misses. | TPC | Y | Y | Y | Y | N | N |
| prof triggers<br>`prof_trigger_00 : prof_trigger_07` | There are 8 such triggers that user can profile. Those are generic and can be inserted in any place of the code to collect the related information. | TPC | Y | Y | Y | Y | Y | Y |
| shared load/<br>`shared_load` | Number of executed shared load instructions per warp on a multiprocessor. | SM | N | N | N | N | Y | Y |
| shared store/<br>`shared_store` | Number of executed shared store instructions per warp on a multiprocessor. | SM | N | N | N | N | Y | Y |
| instructions issued | Number of instructions issued including replays.<br>This is calculated as:<br>`inst_issued`<br>OR<br>`(inst_issued1_0 + 2*inst_issued2_0 + inst_issued1_1 + 2*inst_issued2_1)` | SM | N | N | N | N | Y | Y |
| instructions executed<br>`inst_executed` | Number of instructions executed, do not include replays. | SM | N | N | N | N | Y | Y |
| threads instruction executed | Number of instructions executed by all threads. This does not include replays. For each instruction it increments by the number of threads in the warp that execute the instruction.<br>This is calculated as:<br>`(thread_inst_executed_0 + thread_inst_executed_1`<br>`[+ thread_inst_executed_2 + thread_inst_executed_3])` | SM | N | N | N | N | Y | Y |
| warps launched<br>`warps_launched` | Number of warps launched on a multiprocessor. | SM | N | N | N | N | Y | Y |
| threads launched<br>`threads launched` | Number of threads launched on a multiprocessor. | SM | N | N | N | N | Y | Y |
| active cycles<br>`active_cycles` | Number of cycles a multiprocessor has at least one active warp. | SM | N | N | N | N | Y | Y |
| active warps<br>`active_warps` | Accumulated number of active warps per cycle. For every cycle it increments by the number of active warps in the cycle which can be in the range 0 to 48. | SM | N | N | N | N | Y | Y |

| **Visual Profiler Counter Name**<br>**Command Line Profiler Counter Name** | **Description** | **Type**<br>**SM= Streaming Multiprocessor**<br>**TPC=Texture Processing Cluster**<br>**FB =Frame Buffer (GPU DRAM or Device Memory)**<br>**SW= Software** | **Compute Capability Support**<br>**Y= Yes**<br>**N= No** | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | **1.0** | **1.1** | **1.2** | **1.3** | **2.0** | **2.1** |
| l1 global load hit<br>`l1_global_load_hit` | Number of global load hits in L1 cache. | SM | N | N | N | N | Y | Y |
| l1 global load miss<br>`l1_global_load_miss` | Number of global load misses in L1 cache. | SM | N | N | N | N | Y | Y |
| l1 local load hit<br>`l1_local_load_hit` | Number of local load hits in L1 cache. | SM | N | N | N | N | Y | Y |
| l1 local load miss<br>`l1 local load miss` | Number of local load misses in L1 cache | SM | N | N | N | N | Y | Y |
| l1 local store hit<br>`l1_local_store_hit` | Number of local store hits in L1 cache. | SM | N | N | N | N | Y | Y |
| l1 local store miss<br>`l1_local_store_miss` | Number of local store misses in L1 cache. | SM | N | N | N | N | Y | Y |
| l1 shared bank conflicts<br>`l1 shared bank conflicts` | Number of shared bank conflicts. | SM | N | N | N | N | Y | Y |
| uncached global load transaction<br>`uncached_global_load_transaction` | Number of uncached global load transactions; incremented by 1 per transaction. Transaction size can be 32/64/128 bytes. | SM | N | N | N | N | Y | Y |
| global store transaction<br>`global_store_transaction` | Number of global store transactions; incremented by 1 per transaction. Transaction size can be 32/64/128 bytes. | SM | N | N | N | N | Y | Y |
| l2 read requests | Number of read requests from L1 to L2 cache; incremented by 1 for each 32-byte access.<br>This is calculated as:<br>`l2_subp0_read_sector_queries [+ l2_subp1_read_sector_queries]` | FB | N | N | N | N | Y | Y |
| l2 read texture requests | Number of read requests from texture cache to L2 cache; incremented by 1 for each 32-byte access.<br>This is calculated as:<br>`l2_subp0_read_tex_sector_queries [+ l2_subp1_read_tex_sector_queries]` | FB | N | N | N | N | Y | Y |
| l2 write requests | Number of write requests from L1 to L2 cache; incremented by 1 for each 32-byte access.<br>This is calculated as:<br>`l2_subp0_write_sector_queries [+ l2_subp1_write_sector_queries]` | FB | N | N | N | N | Y | Y |

| **Visual Profiler Counter Name** **Command Line Profiler Counter Name** | **Description** | **Type** **SM= Streaming Multiprocessor** **TPC=Texture Processing Cluster** **FB =Frame Buffer (GPU DRAM or Device Memory)** **SW= Software** | **Compute Capability Support** **Y= Yes** **N= No** | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | **1.0** | **1.1** | **1.2** | **1.3** | **2.0** | **2.1** |
| l2 read misses | Number of read misses in L2 cache; incremented by 1 for each 32-byte access. This is calculated as: `l2_subp0_read_sector_misses [+ l2_subp1_read_sector_misses]` | FB | N | N | N | N | Y | Y |
| l2 write misses | Number of write misses in L2 cache; incremented by 1 for each 32-byte access. This is calculated as: `l2_subp0_write_sector_misses [+ l2_subp1_write_sector_misses]` | FB | N | N | N | N | Y | Y |
| dram reads | Number of read requests to DRAM; incremented by 1 for each 32-byte access. This is calculated as: `(fb_subp0_read_sectors + fb_subp1_read_sectors)` OR `(fb0_subp0_read_sectors + fb0_subp1_read_sectors + fb1_subp0_read_sectors + fb1_subp1_read_sectors)` | FB | N | N | N | N | Y | Y |
| dram writes | Number of write requests to DRAM; incremented by 1 for each 32-byte access. This is calculated as: `(fb_subp0_write_sectors + fb_subp1_write_sectors)` OR `(fb0_subp0_write_sectors + fb0_subp1_write_sectors + fb1_subp0_write_sectors + fb1_subp1_write_sectors)` | FB | N | N | N | N | Y | Y |
| tex cache requests | Number of texture cache requests; incremented by 1 for each 32-byte access. This is calculated as: `tex0_cache_sector_queries [+ tex1_cache_sector_queries]` | SM | N | N | N | N | Y | Y |
| tex cache misses | Number of texture cache misses; incremented by 1 for each 32-byte access. This is calculated as: `tex0_cache_sector_misses [+ tex1_cache_sector_misses]` | SM | N | N | N | N | Y | Y |
| gld instruction 8bit `gld_inst_8bit` | Total number of 8-bit global load instructions that are executed by all the threads across all thread blocks. | SW | N | N | N | N | Y | Y |

| **Visual Profiler Counter Name**<br>**`Command Line Profiler Counter Name`** | **Description** | **Type**<br>**SM= Streaming Multiprocessor**<br>**TPC=Texture Processing Cluster**<br>**FB =Frame Buffer (GPU DRAM or Device Memory)**<br>**SW= Software** | **Compute Capability Support**<br>**Y= Yes**<br>**N= No** | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | **1.0** | **1.1** | **1.2** | **1.3** | **2.0** | **2.1** |
| gld instruction 16bit<br>`gld_inst_16bit` | Total number of 16-bit global load instructions that are executed by all the threads across all thread blocks. | SW | N | N | N | N | Y | Y |
| gld instruction 32bit<br>`gld_inst_32bit` | Total number of 32-bit global load instructions that are executed by all the threads across all thread blocks. | SW | N | N | N | N | Y | Y |
| gld instruction 64bit<br>`gld_inst_64bit` | Total number of 64-bit global load instructions that are executed by all the threads across all thread blocks. | SW | N | N | N | N | Y | Y |
| gld instruction 128bit<br>`gld_inst_128bit` | Total number of 128-bit global load instructions that are executed by all the threads across all thread blocks. | SW | N | N | N | N | Y | Y |
| gst instruction 8bit<br>`gst_inst_8bit` | Total number of 8-bit global store instructions that are executed by all the threads across all thread blocks. | SW | N | N | N | N | Y | Y |
| gst instruction 16bit<br>`gst_inst_16bit` | Total number of 16-bit global store instructions that are executed by all the threads across all thread blocks. | SW | N | N | N | N | Y | Y |
| gst instruction 32bit<br>`gst_inst_32bit` | Total number of 32-bit global store instructions that are executed by all the threads across all thread blocks. | SW | N | N | N | N | Y | Y |
| gst instruction 64bit<br>`gst_inst_64bit` | Total number of 64-bit global store instructions that are executed by all the threads across all thread blocks. | SW | N | N | N | N | Y | Y |
| gst instruction 128bit<br>`gst_inst_128bit` | Total number of 128-bit global store instructions that are executed by all the threads across all thread blocks. | SW | N | N | N | N | Y | Y |

# SUPPORTED DERIVED STATISTICS

Visual Profiler supports derived statistics for different multiprocessor configurations and compute capabilities. A description of the statistics that are derived from the profiler counter values is provided in Table 7. The **Compute Capability** columns provide theoretical valid ranges for the derived statistics.

- * indicates that the range for this derived statistic varies from one device to another and depends on factors such as memory bus width and memory clock.
- NA indicates that the derived statistic is not available for the specific compute capability.

**Note:** The derived statistics displayed in the Summary Table as well as in the Analysis window of the Kernel Analysis feature for a particular kernel are the average values taken over all the invocations of that kernel.

## Table 7. Supported Derived Statistics

| Derived Statistic | Description | Compute Capability | | | | | |
|---|---|---|---|---|---|---|---|
| | | 1.0 | 1.1 | 1.2 | 1.3 | 2.0 | 2.1 |
| glob mem read throughput | Global memory read throughput in gigabytes per second. For compute capability < 2.0 this is calculated as: (((gld_32*32) + (gld_64*64) + (gld_128*128)) * TPC) / (gputime * 1000) For compute capability >= 2.0 this is calculated as: ((DRAM reads) * 32) / (gputime * 1000) This derived statistic is also shown as '**Achieved global memory read throughput (GB/s)**' in the kernel analysis window for Fermi. | * | * | * | * | * | * |
| glob mem write throughput | Global memory write throughput in gigabytes per second. For compute capability < 2.0 this is calculated as: (((gst_32*32) + (gst_64*64) + (gst_128*128)) * TPC) / (gputime * 1000) For compute capability >= 2.0 this is calculated as: ((DRAM writes) * 32) / (gputime * 1000) This derived statistic is also shown as **'Achieved global memory write throughput (GB/s)**' in the kernel analysis window for Fermi. | * | * | * | * | * | * |
| glob mem overall throughput | Global memory overall throughput in gigabytes per second. This is calculated as: Global memory read throughput + Global memory write throughput This derived statistic is also shown as '**Achieved global memory throughput (GB/s)**' in the kernel analysis window for Fermi. | * | * | * | * | * | * |
| gld efficiency | Global load efficiency | NA | NA | 0-1 | 0-1 | NA | NA |
| gst efficiency | Global store efficiency | NA | NA | 0-1 | 0-1 | NA | NA |
| instruction throughput | This is the ratio of achieved instruction rate to peak single issue instruction rate. The achieved instruction rate is calculated using the profiler counter "instructions". The peak instruction rate is calculated based on the GPU clock speed. In the case of instruction dual-issue coming into play, this ratio shoots up to greater than 1. This is calculated as: (instructions) / (gpu_time * clock_frequency) | 0-1 | 0-1 | 0-1 | 0-1 | NA | NA |
| active warps/active cycles | The average number of warps that are active on a multiprocessor per cycle. This is calculated as: (active warps) / (active cycles). | NA | NA | NA | NA | 0-48 | 0-48 |
| l1 gld hit rate | This is calculated as: 100 * (l1 global load hit count) / ((l1 global load hit count) + (l1 global load miss count)) | NA | NA | NA | NA | 0-100 | 0-100 |
| texture hit rate % | This is calculated as: 100 * (tex_cache_requests - tex_cache_misses) / (tex_cache_requests) | NA | NA | NA | NA | 0-100 | 0-100 |

| Derived Statistic | Description | Compute Capability | | | | | |
|---|---|---|---|---|---|---|---|
| | | 1.0 | 1.1 | 1.2 | 1.3 | 2.0 | 2.1 |
| Ideal Instruction/Byte ratio | This is a ratio of the peak instruction throughput and the peak memory throughput of the CUDA device.<br>This is a property of the device and is independent of the kernel. | NA | NA | NA | NA | * | * |
| instruction/byte | This is the ratio of the total number of instructions issued by the kernel and the total number of bytes accessed by the kernel from global memory.<br>If this ratio is greater than the Ideal instruction/byte ratio, then the kernel is compute bound and if it's less, then the kernel is memory bound. This is calculated as:<br>(32 * instructions issued * #SM)/ {32 * (l2 read requests + l2 write requests + l2 read texture requests)} | NA | NA | NA | NA | * | * |
| Achieved Kernel Occupancy | This ratio provides the actual occupancy of the kernel based on the number of warps executing per cycle on the SM. It is the ratio of active warps and active cycles divided by the max number of warps that can execute on an SM.<br>This is calculated as:<br>(active warps/active cycles)/48 | NA | NA | NA | NA | 0-1 | 0-1 |
| Kernel requested global memory read throughput (GB/s) | This is the actual number of bytes requested in terms of loads by the kernel from global memory divided by the kernel execution time.<br>These requests are made in terms of global load instructions which can be of varying word sizes of 8, 16, 32, 64 or 128 bits. This is calculated as:<br>(gld instructions 8bit + 2 * gld instructions 16bit + 4 * gld instructions 32bit + 8 * gld instructions 64bit + 16 * gld instructions 128bit) / (gpu time * 1000) | NA | NA | NA | NA | * | * |
| Kernel requested global memory write throughput (GB/s) | This is the actual number of bytes requested in terms of stores by the kernel from global memory divided by the kernel execution time.<br>These requests are made in terms of global store instructions which can be of varying word sizes of 8, 16, 32, 64 or 128 bits. This is calculated as:<br>(gst instructions 8bit + 2 * gst instructions 16bit + 4 * gst instructions 32bit + 8 * gst instructions 64bit + 16 * gst instructions 128bit) / (gpu time * 1000) | NA | NA | NA | NA | * | * |
| Kernel requested global memory throughput (GB/s) | This is the combined kernel requested read and write memory throughput. This is calculated as:<br>(Kernel requested global memory read throughput + Kernel requested global memory write throughput) | NA | NA | NA | NA | * | * |
| L1 cache read throughput (GB/s) | This gives the throughput achieved while accessing data from L1 cache. This is calculated as:<br>[(l1 global load hit + l1 local load hit) * 128 * #SM + l2 read requests * 32] / (gpu time * 1000) | NA | NA | NA | NA | * | * |
| L1 cache global hit ratio (%) | Percentage of hits that occur in L1 cache while accessing global memory. This statistic will be zero when L1 cache is disabled. This is calculated as:<br>(100 * l1 global load hit)/(l1 global load hit + l1 global load miss ) | NA | NA | NA | NA | 0-100 | 0-100 |

| **Derived Statistic** | **Description** | **Compute Capability** | | | | | |
|---|---|---|---|---|---|---|---|
| | | **1.0** | **1.1** | **1.2** | **1.3** | **2.0** | **2.1** |
| Texture cache memory throughput (GB/s) | This gives the memory throughput achieved while reading data from texture memory. This statistic will be zero when texture memory is not used. This is calculated as: (#SM * tex cache sector queries * 32) / (gpu time * 1000) | NA | NA | NA | NA | * | * |
| Texture cache hit rate (%) | Percentage of hits that occur in texture cache while accessing data from texture memory. This statistic will be zero when texture memory is not used. This is calculated as: 100 * (tex cache requests – tex cache misses)/tex cache requests | NA | NA | NA | NA | 0-100 | 0-100 |
| L2 cache texture memory read throughput (GB/s) | This gives the throughput achieved while reading data from L2 cache when a request for data residing in texture memory is made. This is calculated as: (l2 read tex requests * 32)/(gpu time *1000) | NA | NA | NA | NA | * | * |
| L2 cache global memory read throughput (GB/s) | This gives the throughput achieved while reading data from L2 cache when a request for data residing in global memory is made by L1. This is calculated as: (l2 read requests * 32)/(gpu time * 1000) | NA | NA | NA | NA | * | * |
| L2 cache global memory write throughput (GB/s) | This gives the throughput achieved while writing data to L2 cache when a request to store data in global memory is made by L1. This is calculated as: (l2 write requests * 32)/(gpu time * 1000) | NA | NA | NA | NA | * | * |
| L2 cache global memory throughput (GB/s) | This is the combined L2 cache read and write memory throughput. This is calculated as: (L2 cache global memory read throughput + L2 cache global memory write throughput) | NA | NA | NA | NA | * | * |
| L2 cache read hit ratio (%) | Percentage of hits that occur in L2 cache while reading from global memory. This is calculated as: 100 * (L2 cache global memory read throughput - glob mem read throughput)/( L2 cache global memory read throughput) | NA | NA | NA | NA | 0-100 | 0-100 |
| L2 cache write hit ratio (%) | Percentage of hits that occur in L2 cache while writing to global memory. This is calculated as: 100 * (L2 cache global memory write throughput - glob mem write throughput)/( L2 cache global memory write throughput) | NA | NA | NA | NA | 0-100 | 0-100 |
| Local memory bus traffic (%) | Percentage of bus traffic caused due to accesses to local memory. This is calculated as: (2 * l1 local load miss * 128 * 100)/((l2 read requests + l2 write requests)* 32 / #SMs) | NA | NA | NA | NA | 0-100 | 0-100 |

| Derived Statistic | Description | Compute Capability | | | | | |
|---|---|---|---|---|---|---|---|
| | | 1.0 | 1.1 | 1.2 | 1.3 | 2.0 | 2.1 |
| Global memory excess load (%) | This shows the percentage of excess data that is fetched while making global memory load transactions. Ideally 0% excess loads will be achieved when kernel requested global memory read throughput is equal to the L2 cache read throughput i.e. the number of bytes requested by the kernel in terms of reads are equal to the number of bytes actually fetched by the hardware during kernel execution to service the kernel. If this statistic is high, it implies that the access pattern for fetch is not coalesced, many extra bytes are getting fetched while serving the threads of the kernel. This is calculated as:<br>100 – (100 * kernel requested global memory read throughput / l2 read throughput) | NA | NA | NA | NA | 0-100 | 0-100 |
| Global memory excess store (%) | This shows the percentage of excess data that is accessed while making global memory store transactions. Ideally 0% excess stores will be achieved when kernel requested global memory write throughput is equal to the L2 cache write throughput i.e. the number of bytes requested by the kernel in terms of stores are equal to the number of bytes actually accessed by the hardware during kernel execution to service the kernel. If this statistic is high, it implies that the access pattern for store is not coalesced, many extra bytes are getting accessed while execution of the threads of the kernel. This is calculated as:<br>100 - (100 * kernel requested global memory write throughput / l2 write throughput) | NA | NA | NA | NA | 0-100 | 0-100 |
| Peak global memory throughput (GB/s) | This is the peak memory throughput or bandwidth that can be achieved on the present CUDA device. This is a device property and the kernel achieved memory throughput should be as close as possible to this peak. | * | * | * | * | * | * |
| IPC - Instructions/Cycle | This gives the number of instructions issued per cycle. This should be compared to maximum IPC possible for the device. The range provided is for single precision floating point instructions. This is calculated as: (instructions issued/active cycles) | NA | NA | NA | NA | 0-2 | 0-4 |
| Divergent branches (%) | The percentage of branches that are causing divergence within a warp amongst all the branches present in the kernel. Divergence within a warp causes serialization in execution. This is calculated as:<br>(100*divergent branch)/(divergent branch + branch) | 0-100 | 0-100 | 0-100 | 0-100 | 0-100 | 0-100 |
| Control flow divergence (%) | Control flow divergence gives the percentage of thread instructions that were not executed by all threads in the warp, hence causing divergence. This should be as low as possible. This is calculated as:<br>100 * ((32 * instructions executed) – threads instruction executed)/(32* instructions executed) | NA | NA | NA | NA | 0-100 | 0-100 |

| **Derived Statistic** | **Description** | **Compute Capability** | | | | | |
|---|---|---|---|---|---|---|---|
| | | **1.0** | **1.1** | **1.2** | **1.3** | **2.0** | **2.1** |
| Replayed Instructions (%) | This gives the percentage of instructions replayed during kernel execution. Replayed instructions are the difference between the numbers of instructions that are actually issued by the hardware to the number of instructions that are to be executed by the kernel. Ideally this should be zero. This is calculated as:<br>100 * (instructions issued - instruction executed) /instruction issued | NA | NA | NA | NA | 0-100 | 0-100 |
| Global memory replay (%) | Percentage of replayed instructions caused due to global memory accesses. This is calculated as:<br>100 * (l1 global load miss)/ instructions issued | NA | NA | NA | NA | 0-100 | 0-100 |
| Local memory replay (%) | Percentage of replayed instructions caused due to local memory accesses. This is calculated as:<br>100 * (l1 local load miss + l1 local store miss)/ instructions issued | NA | NA | NA | NA | 0-100 | 0-100 |
| Shared bank conflict replay (%) | Percentage of replayed instructions caused due to shared memory bank conflicts. This is calculated as:<br>100 * (l1 shared conflict)/ instructions issued | NA | NA | NA | NA | 0-100 | 0-100 |
| Shared memory bank conflict per shared memory instruction (%) | This gives an indication of the number of bank conflicts caused per shared memory instruction. This may exceed 100% if there are n-way bank conflicts or the data accessed is double precision. This is calculated as:<br>100 * (l1 shared bank conflict)/(shared load + shared store) | NA | NA | NA | NA | 0-100 | 0-100 |
| SM activity (%) | Percentage of multiprocessor utilization. This is calculated as:<br>100 * (active cycles)/ elapsed clocks | NA | NA | NA | NA | 0-100 | 0-100 |

# COMMAND LINE PROFILER

The command line profiler allows users to gather timing information about kernel execution and memory transfer operations for CUDA and OpenCL applications. 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.

## 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 can 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. If there is no log path specified, the profiler will log data to "cuda_profile_%d.log" in case of a CUDA context and "opencl_profile_%d.log" in case of a OpenCL 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 enabling performance counters in the GPU.

Configuration details are covered in a subsequent section.

The old environment variables, which were used specifically for CUDA/OpenCL are still supported. The old environment variables for the above functionalities are:

**CUDA_PROFILE/OPENCL_PROFILE**

**CUDA_PROFILE_LOG/OPENCL_PROFILE_LOG**

**CUDA_PROFILE_CSV/OPENCL_PROFILE_CSV**

**CUDA_PROFILE_CONFIG/OPENCL_PROFILE_CONFIG**

If **CUDA_PROFILE** or **OPENCL_PROFILE** are explicitly set and the **COMPUTE_PROFILE** environment variable is not set, the profiler outputs only the corresponding contexts. If both are set, the **COMPUTE_PROFILE** environment variables take precedence over **CUDA_PROFILE/OPENCL_PROFILE** environment variable.

## 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. The profiler configuration options are same for CUDA and OpenCL contexts, though they differ in their terminology. Refer to Table 1 for the terminology mapping between CUDA and OpenCL.

### Command Line Profiler Options

Table 8 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 8.
- In most cases the column name is the same as the option name; the exceptions are listed in Table 8.
- In most cases the column values are 32-bit integers in decimal format; the exceptions are listed in Table 8.

## Table 8. Command Line Profiler Options

| Option | Description |
|---|---|
| timestamp | Time stamps for kernel launches and memory transfers. This can be used for timeline analysis.<br>The column values are single precision floating point value in microseconds. |
| gpustarttimestamp | Time stamp when kernel starts execution in GPU.<br>The column values are 64-bit unsigned value in nanoseconds in hexadecimal format. |
| gpuendtimestamp | Time stamp when kernel ends execution in GPU.<br>The column values are 64-bit unsigned value in nanoseconds in hexadecimal format. |
| gridsize | Number of blocks in a grid along the X and Y dimensions for a kernel launch.<br>This option outputs the following two columns:<br>CUDA:<br>• gridsizeX<br>• gridsizeY<br>OpenCL:<br>• ndrangesizeX<br>• ndrangesizeY |
| gridsize3d | Number of blocks in a grid along the X, Y and Z dimensions for a kernel launch.<br>This option outputs the following three columns:<br>CUDA:<br>• gridsizeX<br>• gridsizeY<br>• gridsizeZ<br>OpenCL:<br>• ndrangesizeX<br>• ndrangesizeY<br>• ndrangesizeZ |
| threadblocksize | Number of threads in a block along the X, Y and Z dimensions for a kernel launch.<br>This option outputs the following three columns:<br>CUDA:<br>• threadblocksizeX<br>• threadblocksizeY<br>• threadblocksizeZ<br>OpenCL:<br>• workgroupsizeX<br>• workgroupsizeY<br>• workgroupsizeZ |
| dynsmemperblock | Size of dynamically allocated shared memory per block in bytes for a kernel launch. (Only CUDA) |

| Option | Description |
|---|---|
| stasmemperblock | Size of statically allocated shared memory per block in bytes for a kernel launch.<br>This option outputs the following columns:<br>CUDA:<br>• stasmemperblock<br>OpenCL:<br>• stasmemperworkgroup |
| regperthread | Number of registers used per thread for a kernel launch.<br>This option outputs the following columns:<br>CUDA:<br>• regperthread<br>OpenCL:<br>• regperworkitem |
| 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). |
| memtransferhostmem type | 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. |
| localblocksize | If workgroupsize has been specified by the user, this option would be 1, otherwise it would be 0.(Only OpenCL).<br>This option outputs the following column:<br>• localworkgroupsize |
| cacheconfigrequested | Requested cache configuration option for a kernel launch:<br>• 0 CU_FUNC_CACHE_PREFER_NONE - no preference for shared memory or L1 (default)<br>• 1 CU_FUNC_CACHE_PREFER_SHARED - prefer larger shared memory and smaller L1 cache<br>• 2 CU_FUNC_CACHE_PREFER_L1 - prefer larger L1 cache and smaller shared memory<br>• 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. |

## Command Line Profiler Counters

The command line profiler supports logging of counters during kernel execution. Table 9 lists only counters specific to the command line profiler. Refer to Table 6 for counters which are common to both command line profiler and Visual Profiler. Table 6 also contains formulas for calculating some higher level counters provided in Visual Profiler which in turn are calculated using the low level counters supported by the command line profiler. In every application run only a few counter values can be collected. The number of counters depends on the specific counters selected.

Table 9. Command Line Profiler Counters

| Command Line Profiler Counter Name | Description | Type<br>SM= Single Multiprocessor<br>FB = Frame Buffer (GPU DRAM or Device Memory) | Compute Capability Support<br>Y= Yes<br>N= No | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 1.0 | 1.1 | 1.2 | 1.3 | 2.0 | 2.1 |
| inst_issued | Number of instructions issued including replays. | SM | N | N | N | N | Y | N |
| inst_issued1_0 | Number of cycles that issue one instruction for instruction pipeline 0 | SM | N | N | N | N | N | Y |
| inst_issued2_0 | Number of cycles that issue two instructions for instruction pipeline 0 | SM | N | N | N | N | N | Y |
| inst_issued1_1 | Number of cycles that issue one instruction for instruction pipeline 1 | SM | N | N | N | N | N | Y |
| inst_issued2_1 | Number of cycles that issue two instructions for instruction pipeline 1 | SM | N | N | N | N | N | Y |
| thread_inst_executed_0 | Number of instructions executed by all threads. This does not include replays. For each instruction it increments by the number of threads in the warp that execute the instruction in pipeline 0. | SM | N | N | N | N | Y | Y |
| thread_inst_executed_1 | Number of instructions executed by all threads. This does not include replays. For each instruction it increments by the number of threads in the warp that execute the instruction in pipeline 1. | SM | N | N | N | N | Y | Y |
| thread_inst_executed_2 | Number of instructions executed by all threads. This does not include replays. For each instruction it increments by the number of threads in the warp that execute the instruction in pipeline 2. | SM | N | N | N | N | Y | Y |

| **Command Line Profiler Counter Name** | **Description** | **Type**<br>**SM= Single Multiprocessor**<br>**FB = Frame Buffer (GPU DRAM or Device Memory)** | **Compute Capability Support**<br>**Y= Yes**<br>**N= No** | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | **1.0** | **1.1** | **1.2** | **1.3** | **2.0** | **2.1** |
| thread_inst_executed_3 | Number of instructions executed by all threads. This does not include replays. For each instruction it increments by the number of threads in the warp that execute the instruction in pipeline 3. | SM | N | N | N | N | Y | Y |
| l2_subp0_read_sector_queries | Accumulated read sector queries from L1 to L2 cache for slice 0 of all the L2 cache units | FB | N | N | N | N | Y | Y |
| l2_subp1_read_sector_queries | Accumulated read sector queries from L1 to L2 cache for slice 1 of all the L2 cache units | FB | N | N | N | N | Y | Y* |
| l2_subp0_read_tex_sector_queries | Accumulated read sector queries from texture cache to L2 cache for slice 0 of all the L2 cache units | FB | N | N | N | N | Y | Y |
| l2_subp1_read_tex_sector_queries | Accumulated read sector queries from texture cache to L2 cache for slice 1 of all the L2 cache units | FB | N | N | N | N | Y | Y* |
| l2_subp0_write_sector_queries | Accumulated write sector queries from L1 to L2 cache for slice 0 of all the L2 cache units | FB | N | N | N | N | Y | Y |
| l2_subp1_write_sector_queries | Accumulated write sector queries from L1 to L2 cache for slice 1 of all the L2 cache units | FB | N | N | N | N | Y | Y* |
| l2_subp0_read_sector_misses | Accumulated read sectors misses from L2 cache for slice 0 for all the L2 cache units | FB | N | N | N | N | Y | Y |
| l2_subp1_read_sector_misses | Accumulated read sectors misses from L2 cache for slice 1 for all the L2 cache units | FB | N | N | N | N | Y | Y* |
| l2_subp0_write_sector_misses | Accumulated write sector misses from L2 cache for slice 0 for all the L2 cache units | FB | N | N | N | N | Y | Y |
| l2_subp1_write_sector_misses | Accumulated write sectors misses from L2 cache for slice 1 for all the L2 cache units | FB | N | N | N | N | Y | Y* |
| fb_subp0_read_sectors | Number of read requests sent to sub-partition 0 of all the DRAM units | FB | N | N | N | N | Y | Y |
| fb_subp1_read_sectors | Number of read requests sent to sub-partition 1 of all the DRAM units | FB | N | N | N | N | Y | Y |
| fb0_subp0_read_sectors | Number of read requests sent to sub-partition 0 of DRAM unit 0 | FB | N | N | N | N | Y | Y |
| fb0_subp1_read_sectors | Number of read requests sent to sub-partition 1 of DRAM unit 0 | FB | N | N | N | N | Y | Y |
| fb1_subp0_read_sectors | Number of read requests sent to sub-partition 0 of DRAM unit 1 | FB | N | N | N | N | Y | Y |
| fb1_subp1_read_sectors | Number of read requests sent to sub-partition 1 of DRAM unit 1 | FB | N | N | N | N | Y | Y |

| Command Line Profiler Counter Name | Description | Type SM= Single Multiprocessor FB = Frame Buffer (GPU DRAM or Device Memory) | Compute Capability Support Y= Yes N= No | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 1.0 | 1.1 | 1.2 | 1.3 | 2.0 | 2.1 |
| fb_subp0_write_sectors | Number of write requests sent to sub-partition 0 of all the DRAM units | FB | N | N | N | N | Y | Y |
| fb_subp1_write_sectors | Number of read requests sent to sub-partition 1 of all the DRAM units | FB | N | N | N | N | Y | Y |
| fb0_subp0_write_sectors | Number of write requests sent to sub-partition 0 of DRAM unit 0 | FB | N | N | N | N | N | Y* |
| fb0_subp1_write_sectors | Number of write requests sent to sub-partition 1 of DRAM unit 0 | FB | N | N | N | N | N | Y* |
| fb1_subp0_write_sectors | Number of write requests sent to sub-partition 0 of DRAM unit 1 | FB | N | N | N | N | N | Y* |
| fb1_subp1_write_sectors | Number of write requests sent to sub-partition 1 of DRAM unit 1 | FB | N | N | N | N | N | Y* |
| tex0_cache_sector_queries | Number of texture cache sector queries for texture unit 0 | SM | N | N | N | N | Y | Y |
| tex1_cache_sector_queries | Number of texture cache sector queries for texture unit 1 | SM | N | N | N | N | N | Y |
| tex0_cache_sector_misses | Number of texture cache sector misses for texture unit 0 | SM | N | N | N | N | Y | Y |
| tex1_cache_sector_misses | Number of texture cache sector misses for texture unit 1 | SM | N | N | N | N | N | Y |

## 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. The profiler determines dynamically whether the context is CUDA or OpenCL, and produces the output log accordingly.

The default log syntax shown in Example 1 is part of the profiler log for a CUDA application with no profiler configuration file specified.

### Example 1. CUDA Default Profiler Log- No Options or Counters Enabled

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE_NAME 0 GeForce GTX 280
timestamp,method,gputime,cputime,occupancy
timestamp=[ 2155.302 ] method=[ _Z10fhaar1dwtdiPf ] gputime=[ 7.808 ]
cputime=[ 74.730 ]
occupancy=[ 1.000 ]
timestamp=[ 2421.886 ] method=[ memcopy ] gputime=[ 4.864 ] cputime=[
238.159 ]
timestamp=[ 2706.140 ] method=[ _Z10ihaar1dwtdiPf ] gputime=[ 7.296 ]
cputime=[ 59.295 ]
occupancy=[ 1.000 ]
timestamp=[ 2876.413 ] method=[ memcopy ] gputime=[ 4.608 ] cputime=[
224.679 ]
```

The log above in Example 1 shows data for memory copies and a few different kernel launches. The '**method'** label specifies which GPU function was executed by the driver. The '**gputime'** and '**cputime'** labels specify the actual chip execution time and the driver execution time (including **gputime**), respectively. Note that timestamp, **gputime** and **cputime** are in microseconds. The 'occupancy' label gives the warp occupancy - percentage of the maximum warp count in the GPU - for a particular method launch.

Example 2 shows the profiler log of a matrix multiplication application. There are a few options and counters enabled in this example using the profiler configuration file:

```
gridsize
threadblocksize
memtransfersize
memtransferdir
instructions
branch
cta_launched
```

## Example 2. CUDA Profiler Log- Options and Counters Enabled

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE_NAME 0 GeForce GTX 280
timestamp,method,gputime,cputime,gridsizeX,gridsizeY,threadblocksizeX,t
hreadblocksizeY,
threadblocksizeZ,occupancy,instructions,branch,cta_launched,memtransfer
size,memtransferdir
timestamp=[ 6492.515 ] method=[ _Z10dmatrixmulPfiiS_iiS_ ] gputime=[
25.472 ] cputime=[ 203.797 ]
gridSize=[ 2, 1 ] threadblocksize=[ 32, 8, 8 ] occupancy=[ 0.333 ]
instructions=[ 2261 ]
branch=[ 312 ] cta_launched=[ 2 ]
timestamp=[ 7031.061 ] method=[ memcopy ] gputime=[ 8.896 ] cputime=[
230.686 ]
memtransfersize=[ 8192 ] memtransferdir=[ 1 ]
```

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.

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

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_PROFILE_CSV 1
# CUDA_DEVICE_NAME 0 GeForce GTX 280
timestamp,method,gputime,cputime,gridsizeX,gridsizeY,threadblocksizeX,t
hreadblocksizeY,
threadblocksizeZ,occupancy,cta_launched,branch,instructions,memtransfer
size,memtransferdir
6390.687,_Z10dmatrixmulPfiiS_iiS_,25.184,203.168,2,1,32,8,8,0.333,312,3
12,2261
6946.483,memcopy,8.928,240.673,,,,,,,,,,8192,1
```

The following examples are for OpenCL applications. Example 4 is part of the log from a test of the scan application without any counters enabled.

### Example 4. OpenCL Default Profiler Log- No Options or Counters Enabled

```
# OPENCL_PROFILE_LOG_VERSION 2.0
# OPENCL_DEVICE 0 GeForce GTX 280
# TIMESTAMPFACTOR 114aa119a0c9d7d2
timestamp,gpustarttimestamp,gpuendtimestamp,method,gputime,cputime,occu
pancy
timestamp=[ 7791621.500 ] gpustarttimestamp=[ 114ab721d9c649e0 ]
gpuendtimestamp=[ 114ab721da1a0be0 ]
method=[ workgroupScanInclusive ] gputime=[ 5489.152 ] cputime=[
5842.782 ] occupancy=[ 1.000 ]
timestamp=[ 7802433.500 ] gpustarttimestamp=[ 114ab721da6aaaa0 ]
gpuendtimestamp=[ 114ab721da6b5500 ]
method=[ workgroupScanExclusive ] gputime=[ 43.616 ] cputime=[ 387.270
] occupancy=[ 1.000 ]
timestamp=[ 7804496.500 ] gpustarttimestamp=[ 114ab721da894480 ]
gpuendtimestamp=[ 114ab721dacecc00 ]
method=[ uniformUpdate ] gputime=[ 4556.672 ] cputime=[ 4915.150 ]
occupancy=[ 1.000 ]
```

This log shows data for memory copies and a few different kernel launches. The '**method**' label specifies which GPU function was executed by the driver. The '**gputime**' and '**cputime**' labels specify the actual chip execution time and the driver execution time (including **gputime**), respectively. The **gpustarttimestamp** and **gpuendtimestamp** indicate the start and end timestamps of the kernel being executed on the GPU.

Note that timestamp, **gputime** and **cputime** are in microseconds, and **gpustarttimestamp** and **gpuendtimestamp** are in nanoseconds. The 'occupancy' label gives the warp occupancy - percentage of the maximum warp count in the GPU - for a particular method launch. An occupancy of 1.000 means the chip is completely full.

Example 5 shows the profiler log for the matrix multiplication application. There are some options and counters enabled using the same configuration file as for Example 2:

### Example 5. OpenCL Profiler Log- Options and Counters Enabled

```
# OPENCL_PROFILE_LOG_VERSION 2.0
# OpenCL_DEVICE_NAME 0 GeForce GTX 280
# TIMESTAMPFACTOR 12bae765a4r9c521
timestamp,method,gputime,cputime,ndrangesizeX,ndrangesizeY,workgroupsiz
eX,workgroupsizeY,
workgroupsizeZ,occupancy,instructions,branch,cta_launched,memtransfersi
ze,memtransferdir
timestamp=[ 7205451.000 ] method=[ matrixMul ] gputime=[ 92695.133 ]
cputime=[ 93108.766 ]
NDRangesize=[ 50, 100 ] workgroupsize=[ 16, 16, 1 ] occupancy=[ 1.000 ]
instructions=[ 18204777 ]
```

```
branch=[ 1479119 ] cta_launched=[ 500 ]
timestamp=[ 7423482.500 ] method=[ memcopy ] gputime=[ 8.896 ]
cputime=[ 230.686 ]
memtransfersize=[ 8192 ] memtransferdir=[ 1 ]
```

When **COMPUTE_PROFILE_CSV** is set to 1, this same test produces the following output:

### Example 6. OpenCL Profiler Log- Options and Counters Enabled in CSV Format

```
# OPENCL_PROFILE_LOG_VERSION 1.0
# OPENCL_PROFILE_CSV 1
# OpenCL_DEVICE_NAME 0 GeForce GTX 280
# TIMESTAMPFACTOR 1e4231f54a45c645
timestamp,method,gputime,cputime,ndrangesizeX,ndrangesizeY,workgroupsiz
eX,workgroupsizeY,
workgroupsizeZ,occupancy,instructions,branch,
cta_launched,memtransfersize,memtransferdir
7535422.000,matrixMul,91935.766,93031.500,50,100,16, 16,
1,1.000,18204777,1479119,500
7754673.000,memcopy,8.536,241.342,,,,,,,,,,8192,1
```