

# AerialVision Performance Visualizer

## Version 1.0

Aaron Ariel, Wilson W. L. Fung, Tor M. Aamodt  
University of British Columbia

October 27, 2009

### Table of Contents

1. What is AerialVision .....	2
2. Installation.....	2
3. Usage .....	3
3.A Walkthrough Example .....	3
3.B Detailed Description of AerialVision Features.....	9
Startup Screen .....	9
Visualizer Tab – Creating New Figure .....	10
Visualizer Tab – Customizing an Existing Figure.....	12
Manage Files.....	17
Source Code Viewer Tab – Performance Metric Selection .....	18
Source Code Viewer Tab – Navigating Through Source Code with Performance Metrics.....	20
4. Description of Available Performance Metrics .....	21
5. Extending AerialVision – Adding Variables .....	25

## 1. What is AerialVision

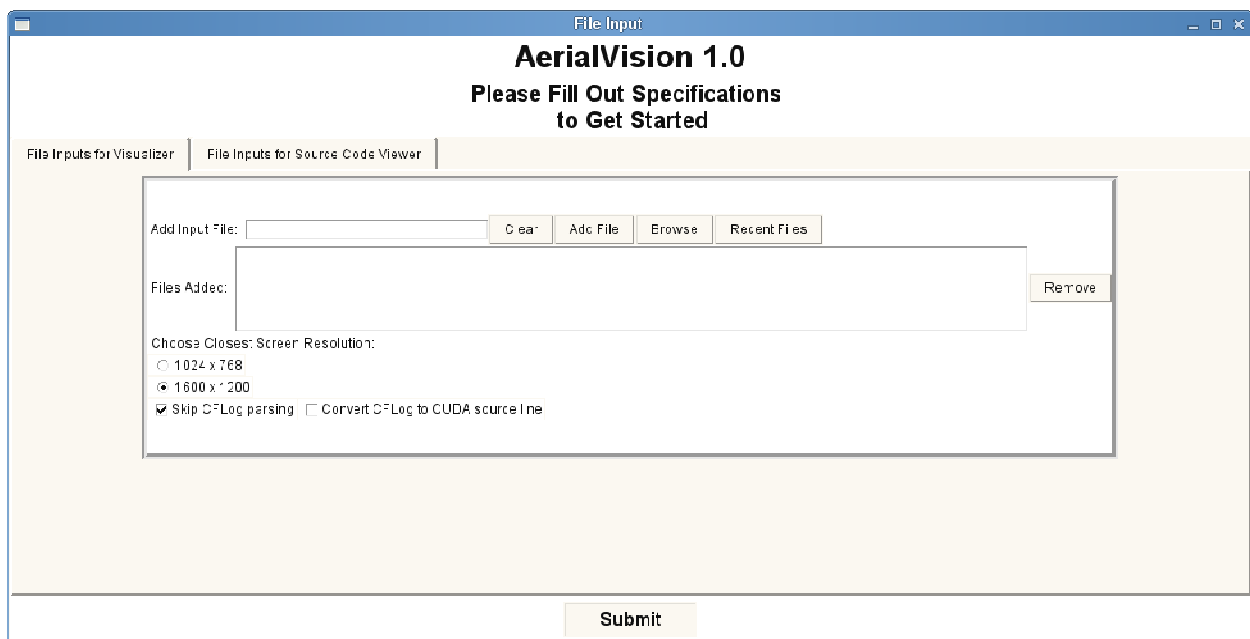
AerialVision is a GPU performance analysis tool. Its main features are a **Visualizer** where various metrics can be graphed as a function of cycle count, and a **Source Code Viewer** where various metrics are displayed in correlation to the corresponding lines of PTX or CUDA.

## 2. Installation

The set of python scripts that implements AerialVision can be found inside the “aerialvision” subdirectory of the GPGPU-Sim distribution. AerialVision has been tested to work using Python 2.6. Earlier distributions of python are not compatible with AerialVision.

To install AerialVision, run the command: “python installscript.py” inside the “aerialvision” subdirectory, which will download all modules required to run AerialVision using wget (which must be installed on your system, along with python). These modules may take up to 200MB of disk space in your \$HOME directory. After that, follow the instruction from the install script to modify these environment variables: PYTHONPATH, CPLUS\_INCLUDE\_PATH, PKG\_CONFIG\_PATH. The AerialVision launcher script “aerialvision.py” can be found inside the “bin” subdirectory of the GPGPU-Sim distribution. Add this “bin” subdirectory (\$GPGPUSIM\_ROOT/bin) to your PATH environment variable. You may need to edit the first line of “aerialvision.py” to point to your local installation of python (if it is not /usr/bin/python).

To begin using AerialVision, run the command: “aerialvision.py”. This should load up the startup screen as in Figure 1.



**Figure 1: Startup Screen in AerialVision**

### 3. Usage

This section is divided into two main parts. The first section gives a brief walkthrough example, so that the user may get started using AerialVision features as quickly as possible. The second section covers all AerialVision features in detail.

#### 3.A Walkthrough Example

We first present a walkthrough example of various AerialVision features. In this manual, we have used GPGPU-Sim trace files corresponding to the MUMmerGPU CUDA application however users may follow along using whichever application they choose.

We begin by loading the startup screen by typing ‘python AerialVision.py’ into the command line. Once the startup screen loads up, the next step is to submit all of the relevant files.

By default, you should now be looking at the fields inside the ‘**File Inputs for Visualizer**’ tab. In this tab, we submit files that will enable us to use the time lapse visualizing capabilities of the Visualizer. These files are by default in the form ‘gpgpusim\_visualizer\_\_\*.log.gz’. We submit files by clicking the ‘**Browse**’ button (if you’ve submitted the file before you can click on the ‘**Recent Files**’ button), and then clicking ‘**Add File**’ once the file’s path is in the ‘**Add Input File**’ text field. Finally, choose the appropriate resolution and parsing options (described in another section of this manual). Figure 2 depicts what your screen should look like at this point. You should note that you can submit numerous files for visualizing into this tab; however, for the purposes of this walkthrough we have limited it to one.

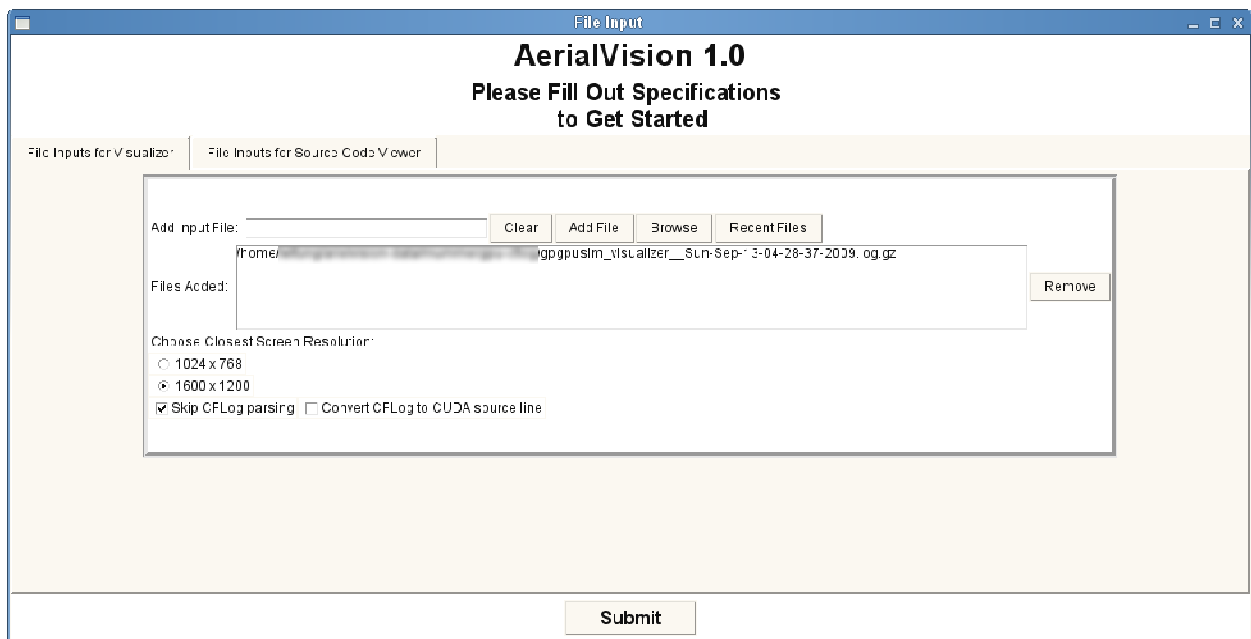


Figure 2 File Inputs for Visualizer

Now click on the **'File Inputs for Source Code Viewer'** tab. In this tab we submit files that present statistics corresponding to each line of PTX or CUDA source. Before clicking the **'Add Files'** button, it is necessary to insert the file paths to three distinct files required by this part of AerialVision. The file that goes in the **'Add CUDA Source Code File'** text field is the appropriate CUDA kernel source code file. This file should end with a **'\*.cu'** extension. In the case of MUMmerGPU, this file is named **'mummergpu\_kernel.cu'** (Not to be confused with **'mummergpu.cu'**). The file that goes in the **'Add Corresponding PTX File'** text field is the appropriate PTX file generated by the CUDA compiler. In the case of MUMmerGPU, this file is named **'mummergpu.ptx'**. Finally, the file that goes in the **'Add Corresponding Stat File'** is generated by the GPGPU-Sim and is by default named **'gpgpu\_inst\_stats.txt'**. This file should be located in the same folder as the one you launched GPGPU-Sim from. Once you have filled the three text fields, click the green **'Add Files'** button. Figure 3 depicts what your screen should look like at this point. You can now launch AerialVision by clicking the **'Submit'** button at the bottom. It should be noted that for your own purposes, it is not necessary to fill both the **'File Inputs for Visualizer'** and **'File Inputs for Source Code Viewer'** tabs as both parts of AerialVision can be used independently of the other.

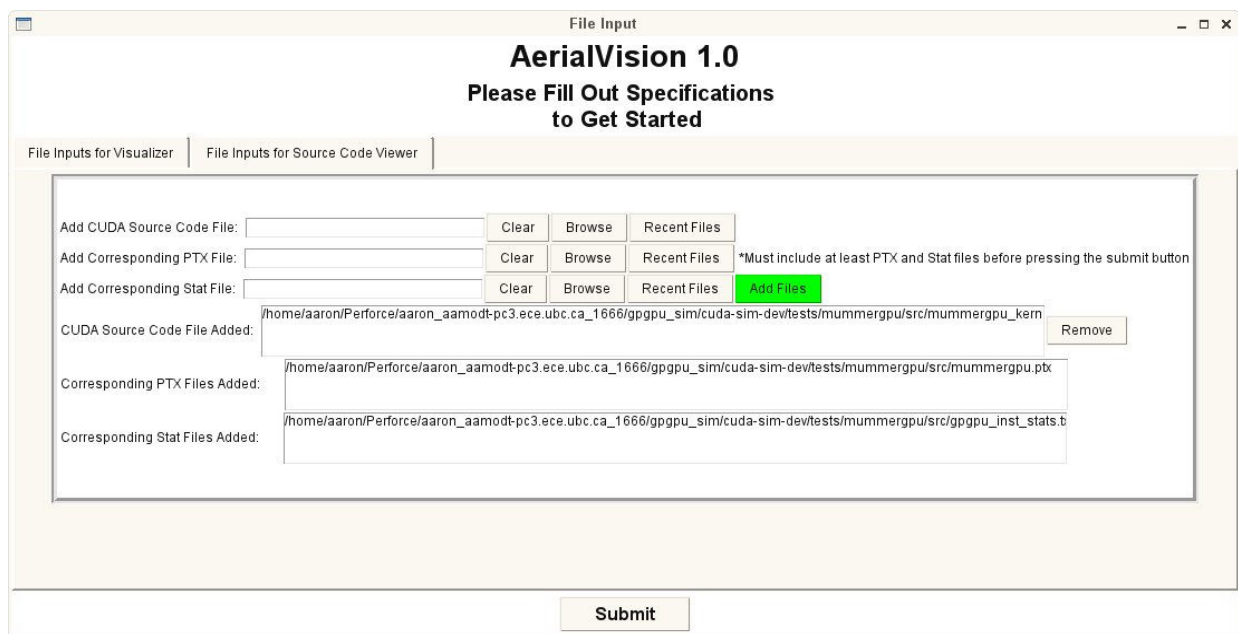


Figure 3 File Inputs for Source Code Viewer

Once the **'Submit'** button has been clicked, AerialVision will parse the input files for the data inside. This process may take a minute to several minutes depending on the length of the trace files submitted to AerialVision. Your screen should now look something like Figure 7.

We will now create our first plot using AerialVision. It should be noted that the plots that we produce in this tutorial will obviously differ from the ones that the user produces as we are simulating a different application (unless of course the user is simulating MUMmerGPU as well). For starters, let's investigate whether off chip memory latency effects performance for our application.

We first need to **'Choose a File'** by double clicking on the trace file that we want to extra data from. These files can be found in box 'A' in Figure 7. Double clicking on one of the files should turn the appropriate section of the **'Options Chosen'** list green.

From this file the first thing we'd like to plot is Instruction Per Cycle (IPC). We do this by selecting **'globalInsn'** from the list of **'Y Vars'** (Table 3 and Table 4 in this manual contains a list of all variables and their descriptions), and then clicking the **'dy/dx'** checkbox. We click the **'dy/dx'** checkbox because the **'globalInsn'** variable contains a running count of instructions executed, and in order to plot IPC we must take the derivative. For this particular plot, we will select **'Line'** from the options available for **'Type of Graph'** (Table 3 and Table 4 depicts the recommended type of plots for all of the available performance metrics).

We now have the option of selecting the number of subplots that we'd like to add. For this particular example, we will choose '2'. In order to do this, slide the **'Add Subplot'** slider to the '2' position and click **'Submit'**.

In the **'Subplot0'** tab we will once again be plotting IPC, only this time on a per shader basis. We do this by first selecting a file as we did earlier. The **'Y Vars'** that we need to select this time is named **'shaderInsn'** so double click on it and once again check the **'dy/dx'** checkbox. This time, the **'Type of Graph'** that we are going to select is the **'Parallel Intensity Plot'**. You have now chosen all of the necessary options for **'Subplot0'**.

In the **'Subplot1'** tab we will be plotting the memory latency. Once again select the appropriate file as we did earlier. The **'Y Vars'** that we need to select this time is named **'averagemflatency'**. This time do not click on the **'dy/dx'** button and choose the **'Line'** plot from the **'Type of Graph'** options. You may now press the green **'GraphMe!'** button. If you have followed this walkthrough correctly, all of the fields in the **'Option Chosen'** list should be green. After clicking the green **'GraphMe!'** button, your screen should now look something like Figure 4.

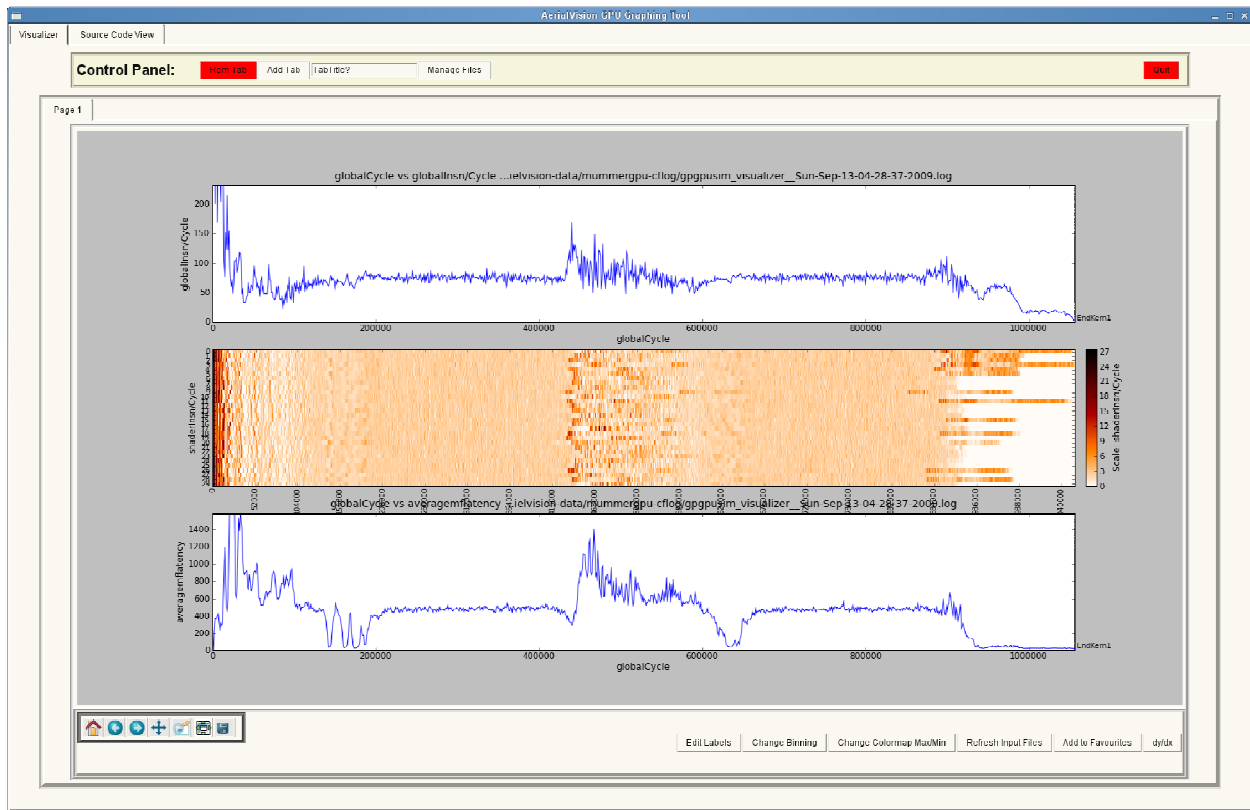


Figure 4 Time Lapse Plots

We now see how the latency affects the IPC. The top plot in your screen is the IPC for all of the shaders added up, the middle plot displays the IPC for each individual shader, and lastly the bottom plot displays the average latency per cycle.

We will now demonstrate how to use the 'Source Code View' of AerialVision. After clicking on the 'Source Code View' tab your screen should look like Figure 5.

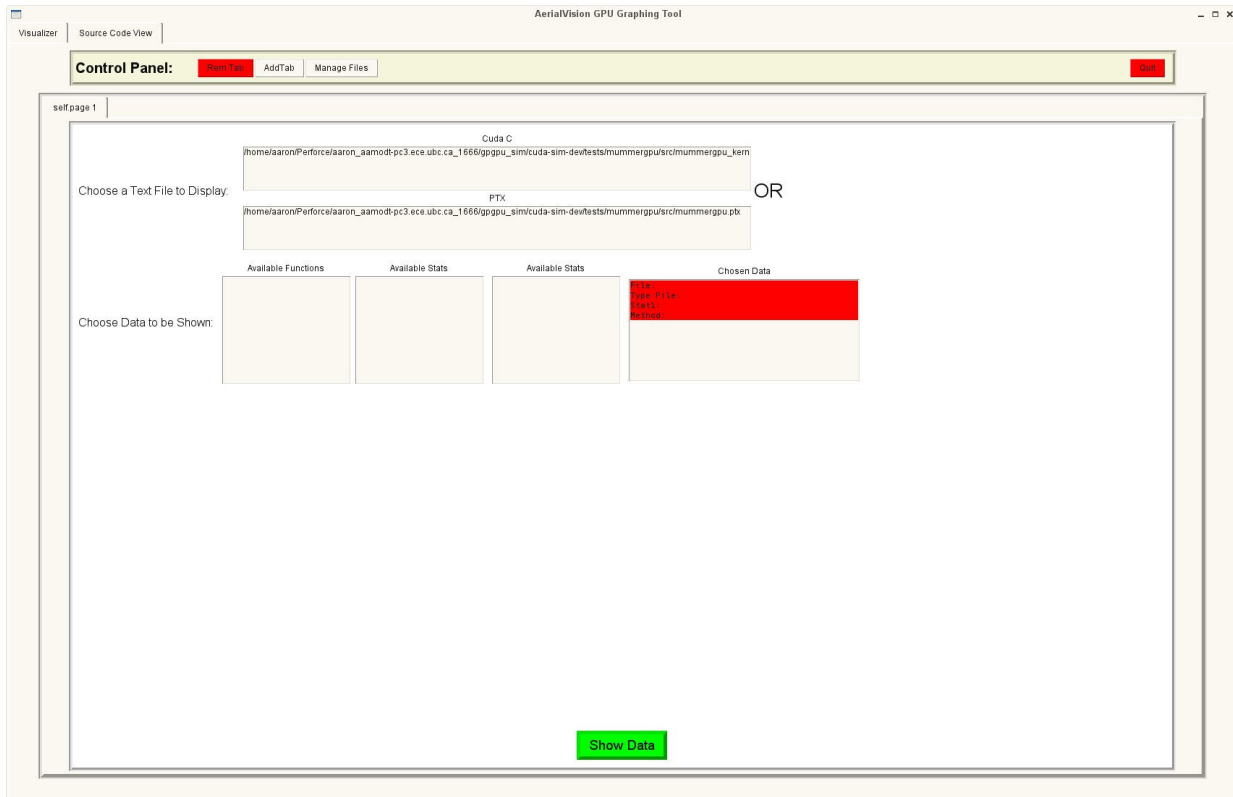


Figure 5 Source Code View

For this example, we will plot CUDA source line statistics rather than PTX line statistics. Therefore first we must choose the appropriate CUDA source file, do this by clicking the appropriate file under the **'Cuda C'** header. This should turn the **'File: '** under **'Chosen Data'** from red to green.

Next, we will need to choose the appropriate PTX statistic aggregation method from under **'Available Functions'** (this is explained in another section of this manual), choose the **'Sum'** option, and then choose **'exposed\_latency'** from the **'Available Stats'** list. If you have followed the instruction correctly, all fields under **'Chosen Data'** should be green and you are now ready to plot. Click the green **'Show Data'** button at the bottom. Your screen should now look something like Figure 6.

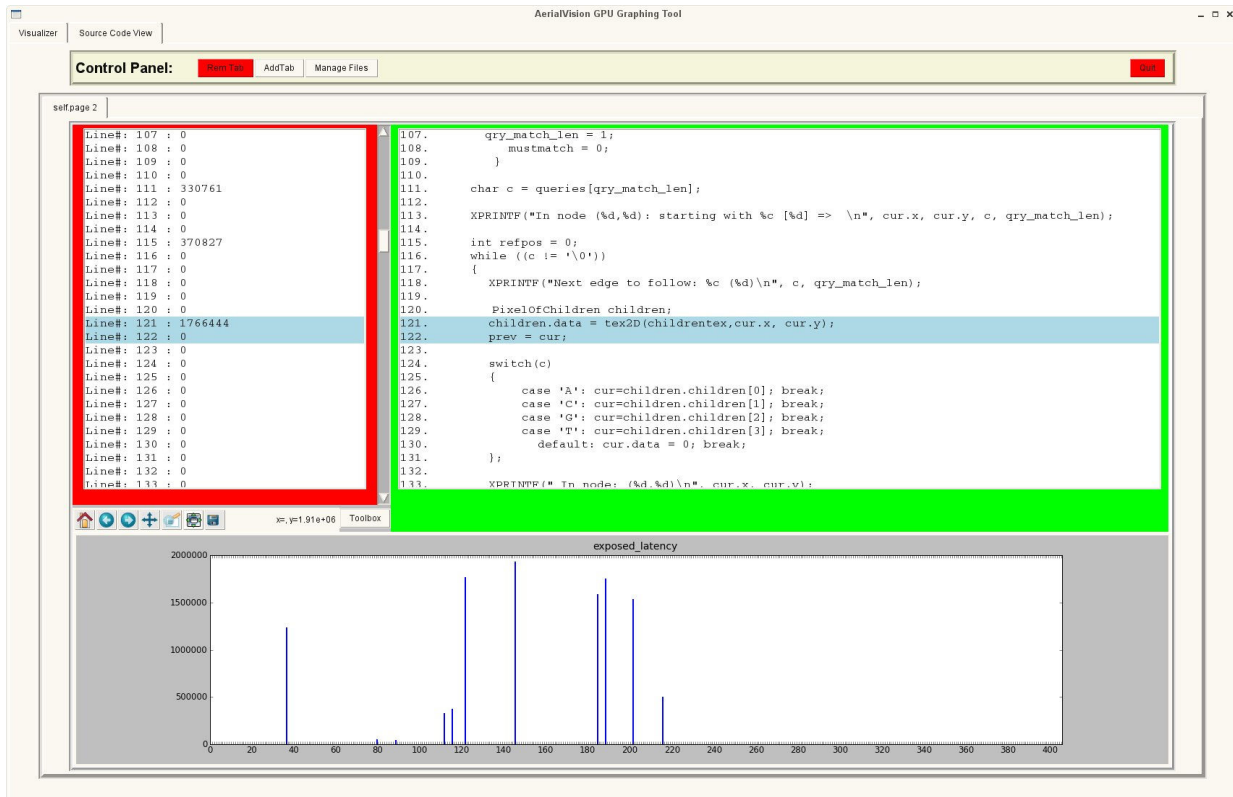


Figure 6 Statistics Per Line of Source Code

You can now explore which lines of CUDA source are causing the greatest amount of thread stalls due to waiting time on memory requests. Right click on the largest bar to scroll the text to that specific line. In our particular example, after right clicking on the bar corresponding to line 121, the text scrolls down to that line and we see that this large amount of exposed latency is being cause by a memory texture access.

This walkthrough is not intended to be a comprehensive presentation of all AerialVision features. It is simply intended to introduce the user to some of the basic AerialVision functionality. A far more detailed understanding can be gained by reading the rest of this manual.



## **3.B Detailed Description of AerialVision Features**

### **Startup Screen**

In the startup screen, you can select all relevant input files to either visualizer or source code viewer parts of AerialVision by switching between the tabs found in the upper left. To select a file, you may type the path to the input file or click 'Browse' for a file selection dialog box to fill in the path. Click 'Add File' to add the file to the list of input files.

The visualizer takes in trace files generated by GPGPU-Sim. The default naming scheme for the trace files is: `gpgpusim_visualizer_*.log.gz` (can be changed with the option `-visualizer_outputfile` in GPGPU-Sim). The source code viewer requires the user to input a CUDA source file and a PTX source file (generated from the CUDA source file) and a file generated by GPGPU-Sim containing the statistics for each line of PTX source code. The default name for this statistics file is `gpgpu_inst_stats.txt` (can be changed with the option `-ptx_line_stats_filename` in GPGPU-Sim).

The user may also select the screen resolution of your desktop (for AerialVision to optimize the widget layout). The two checkboxes at the bottom enable AerialVision to parse CFLog data (a visualization of how threads traverse through a program over time) and convert the parsed data from PTX line number to CUDA source code line numbers (requires a PTX file to be selected in the 'File Input for Source Code Viewer' tab). Clicking the 'Skip CFLOG parsing' checkbox will decrease the load time of AerialVision however this abovementioned functionality would no longer be available. Clicking the 'Convert CFLOG to CUDA source line' checkbox will increase the load time of AerialVision however the user will now be able to visualize this metric with respect to CUDA source line numbers instead of ptx.

Once you have selected all the input files and specified all the options, press the 'Submit' button at the bottom and AerialVision will start after it finishes parsing the input files. It should be noted that the more files that the user inputs (and the larger the files) the longer it will take for AerialVision to parse.

## Visualizer Tab – Creating New Figure

Figure 7 depicts the Visualizer tab in the middle of creating a new figure. This is what you will see by default when AerialVision starts. To create additional plots, click 'Add Tab' in the Control Panel (Region J in Figure 7). A description of each boxed section is on the next page.

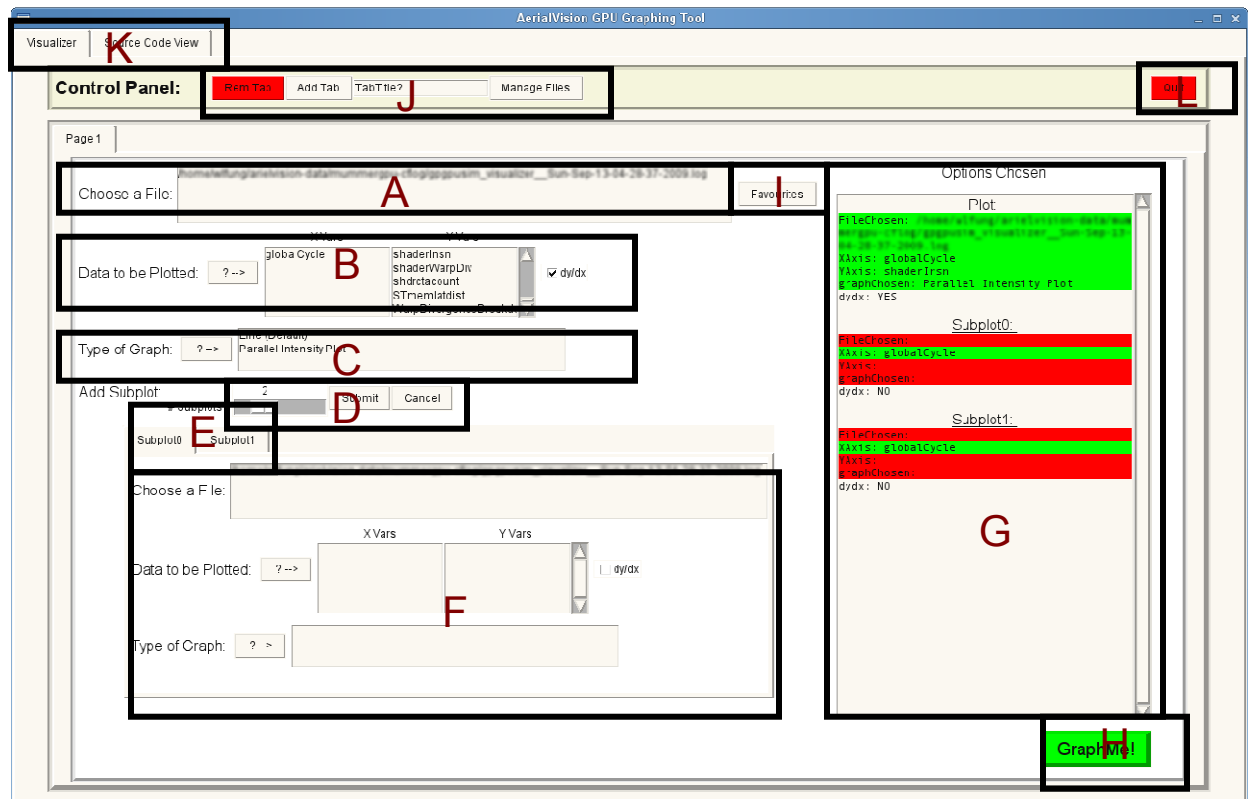


Figure 7: Visualizer Tab (New Plot Creation)

To create a single plot in the new figure, specify the options in regions A, B, and C. Use the widgets in regions D, E, and F to attach additional subplots to the main plot. The text box in region G displays the options chosen in green. Once all the options are set, click 'GraphMe!' button in region H to generate the new figure. Here is the description of each labeled region in Figure 7:

**A:** A list of loaded traces files. Choose one by double clicking on it. A list of statistics that are available for graphing in the trace file will then appear in the boxes below.

**B:** A list of statistics available for plotting should now appear under the 'X Vars' and 'Y Vars' headings. 'X Vars' currently only consist of global clock cycle, whereas 'Y Vars' consist of different performance counters. Choose one of each by double clicking on them. Use the 'dy/dx' button to take the derivative of the y-axis variable before it is graphed.

**C:** Display the types of plots available for the chosen variables. The following types of plots are supported by this release of AerialVision:

1. Line Plot – A standard line plot.
2. The Parallel Intensity Plot – Essentially a color map. This plot is particularly useful for displaying performance counters that are collected for multiple units (e.g. instruction count for each shader core).
3. Stacked Bar – A standard stacked bar chart that shows the component breakdown of each sample of a performance counter.

**D:** Use this slider to select the number of subplots that you would like to graph along with the main plot. Press 'Submit' when you've selected the desired number. The 'Cancel' button is used to remove the subplots that have already been specified.

**E:** Use these tabs to switch between the options available for each subplot.

**F:** Select the configuration for each subplot here. The options are exactly the same as those that were available for the main plot above (A,B,C).

Important note: In this documentation, we refer to labels (A,B,C) as defining the configuration for the PLOT while label (F) defines the configuration for the SUBPLOTS.

**G:** This list displays the options you have chosen. A green highlight is representative of an option that has been selected. A red highlight is representative of an option that still needs to be chosen. You will not be able to graph anything until there is no red in this list.

**H:** click this button to generate the figure. Once there are no red fields in the options chosen list (directly to the above this button),

**I:** The 'favourites' button is useful when there are certain combinations of plots that you use frequently. The button is only enabled after you have selected a trace file for the figure (which unfortunately limits the favourites to a combination of plots from a single trace only). Clicking the button will show you're a list of favourites available. Choose to generate a favorite figure by double clicking it. Instructions describing how to save a plot combination to the 'favourites' can be found in section 4b.

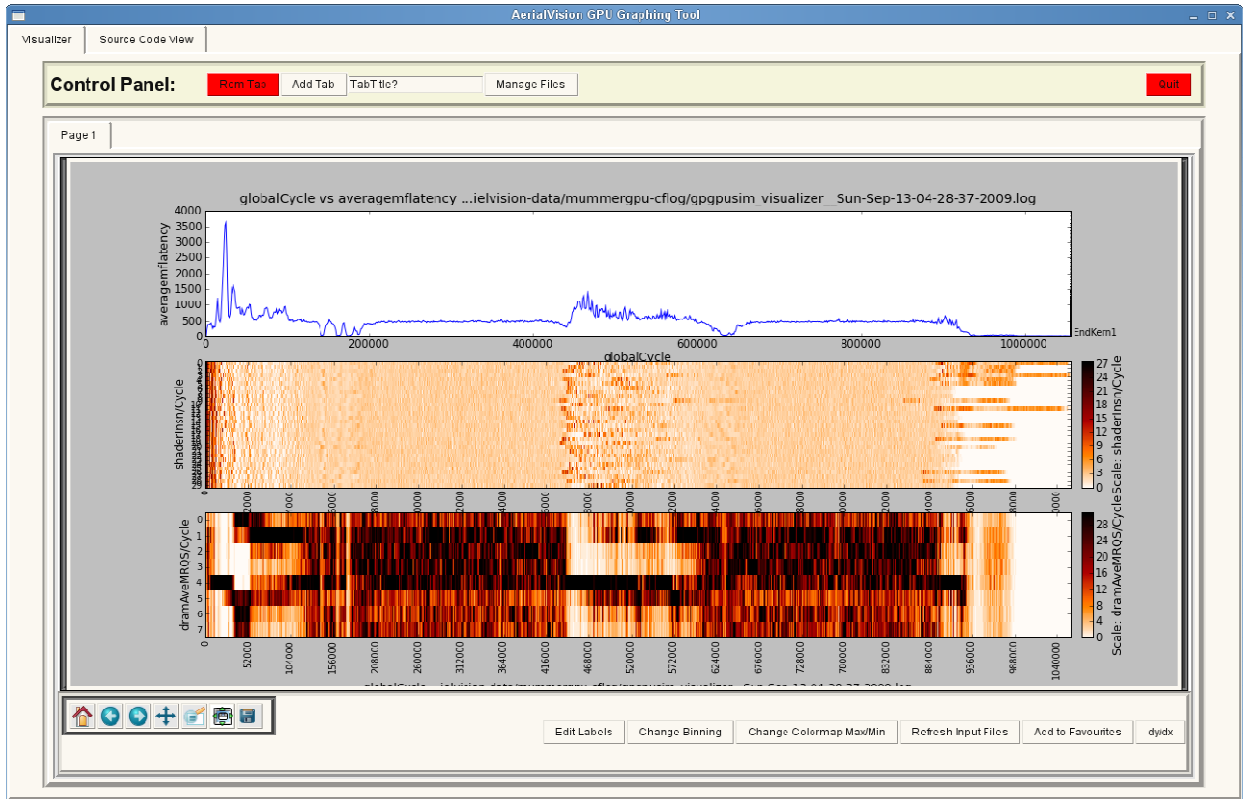
**J:** In the control panel there are three buttons:

1. Rem Tab: Deletes the current tab you are viewing
2. Add Tab: Adds a new tab with a new plot. You may define a name for this tab by entering a string of characters in the field beside this button
3. Manage Files: Use this feature to refresh, remove, or add a file to the visualizer.

**K:** Use these tabs to switch between **Visualizer** and **Source Code Viewer**.

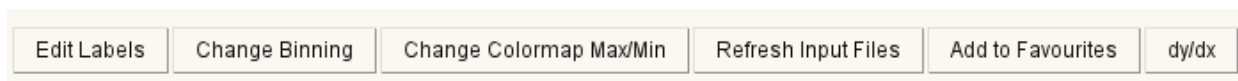
**L:** Exits the program.

## Visualizer Tab – Customizing an Existing Figure

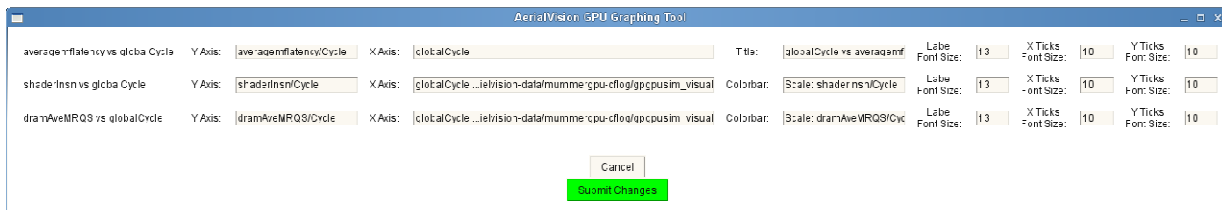


**Figure 8: Visualizer Tab with a Generated Figure**

Figure 8 shows AerialVision with a generated figure. The buttons below the figure canvas provides ways for the user to customize a figure after it is generated. Here is a zoom-in view of the buttons at the bottom right of the window, followed by the description of the functionality of each button:

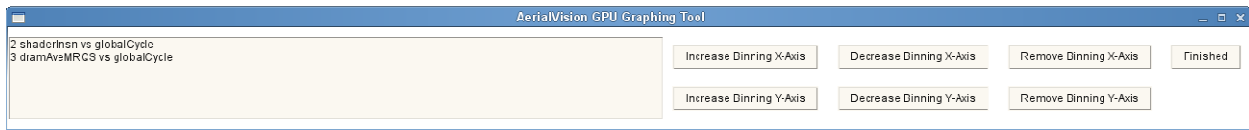


### 1. Edit Labels



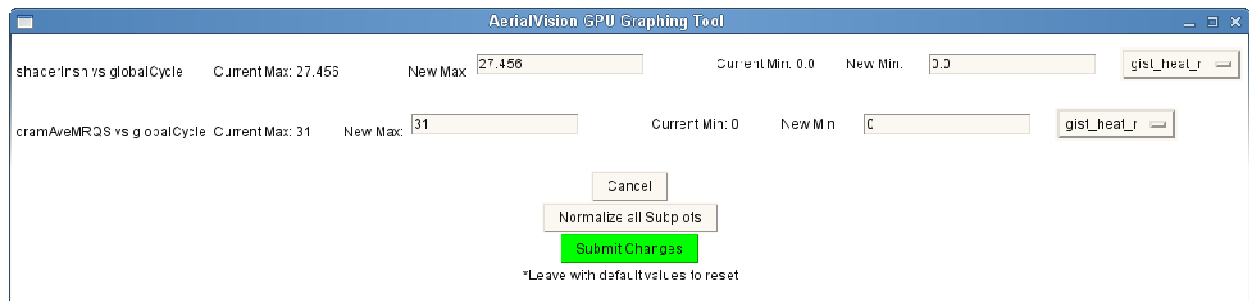
Clicking the 'Edit Labels' button opens up the above dialog box. The user can modify the axis label, the title (color map label for parallel intensity plots as they have no title), their font sizes, and the font sizes of the axes tick labels.

## 2. Change Binning (for Parallel Intensity Plots only)



Clicking the 'Change Binning' button opens up the above dialog box. The user can use this dialog box to increase or decrease the density of y-axis or x-axis tick labels and can only be used on Parallel Intensity Plots. First, select the plot that you would like to edit from the list on the left. Then, click on the appropriate button on the right to perform the modification. Click 'Increase Binning' and 'Decrease Binning' to increase or decrease the density of the tick labels. Click 'Remove Binning' to remove all the tick labels completely from the specific axis of the plot.

## 3. Change Colormap Max/Min



Clicking the 'Change Colormap Max/Min' button opens up the above dialog box. The user can use this dialog box to specify the max/min value that maps to the extremas of the color map in each Parallel Intensity Plot in the figure. This feature is useful if the user would like to increase the color resolution over a particular range of data. The dropping lists on the right allow the user to specify the color map for each Parallel Intensity Plot. Click 'Submit Changes' to apply the new color mapping settings.

Or, the user can normalize the color mapping among all the plots in a figure (so the the same value maps to the same color between different plots) with the 'Normalize all Subplots' button. By default, the visualizer sets the extremas of the color map to be the max/min value of the plotted data (regardless of the previous settings). So, to return to the original color mapping, simply submit the default values.

## 4. Refresh Input Files

The 'Refresh Input Files' button reloads the plotted data of a figure from the trace files and redraws the whole figure to reflect any changes in the data. This allows the user to reload trace files into the visualizer as the GPGPU-Sim writes to them, visualizing the behaviour of a benchmark run even before the simulation finishes.

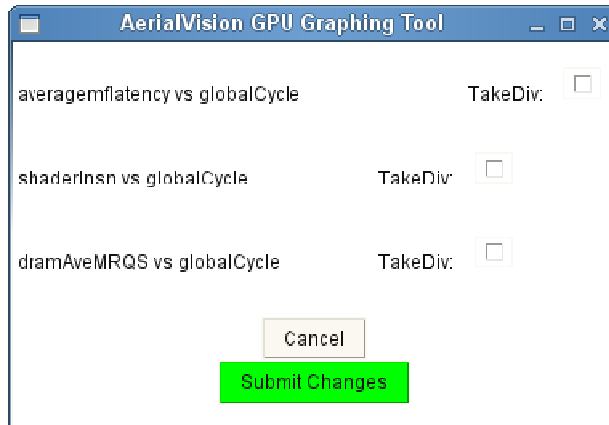
Note: Only the trace files that are currently being plotted will be reloaded using this feature. Instead, use the Manage Files feature described in another section of this documentation.

## 5. Add to Favourites



Use this feature to save the configuration of commonly used plot combinations for quick reuse at a later time (potentially with a different trace file). Fill in the two entry fields in the above dialog box to give this combination of plots a title and a description, then click 'Submit' to save the configuration as a favourite that can be recalled with the 'Favourite' button.

## 6. Dy/Dx



Use this feature to take the derivative of the data in a plot after the figure is generated. In the dialog box (shown above), check the particular plot with data that needs to be differentiated and re-plotted and click 'Submit Changes'.

## 7. Current Constraints When Customizing an Existing Feature

When modifying an existing figure, it is necessary to make changes to the figure in the following order:

1. If desired, modify the colormap configuration first.
2. If desired, modify the binning configuration second.
3. If desired, modify the edit labels configuration last.

The reason for this requirement is that when modifying one of the features out of order, making a change may set the configuration of a different feature to its default value. Modifying your figure in the order specified will ensure that the process of customizing an existing feature will go as smoothly as possible. This constraint will be resolved in a future release of AerialVision.

## 8. Navigation Tool Bar

The following image depicts the navigation tool bar at the bottom-left corner of the figure canvas once the figure has been generated. The functionality of each button is outlined in this section.



### A. Home

Use this button to return all plots in the figure to their default views.

### B. Go Back/Go Forward

Use this button to go back/forward to the last view of the most recently changed plot.

### C. Pan

To pan around a particular plot, click the button to switch to pan mode, then hold the left-mouse button down on the plot and drag in the direction you would like to view.

### D. Zoom

To zoom-in on a particular plot, click this button to switch to zoom mode and then make a box around the area with the left-mouse button that you would like to enlarge. Use the right-mouse button for zoom-out.

### E. Change Spacing

Clicking this button will show the following dialog box.

Use the sliders to adjust the spacing for all plots in the current figure.

Left: Adjust the distance from the left border of the figure canvas.

Bottom: Adjust the distance from the bottom border of the figure canvas.

Right: Adjust the distance from the right border of the figure canvas.

Top: Adjust the distance from the top border of the figure canvas.

Wspace: Currently has no effect.

Hspace: Adjust the spacing between plots.

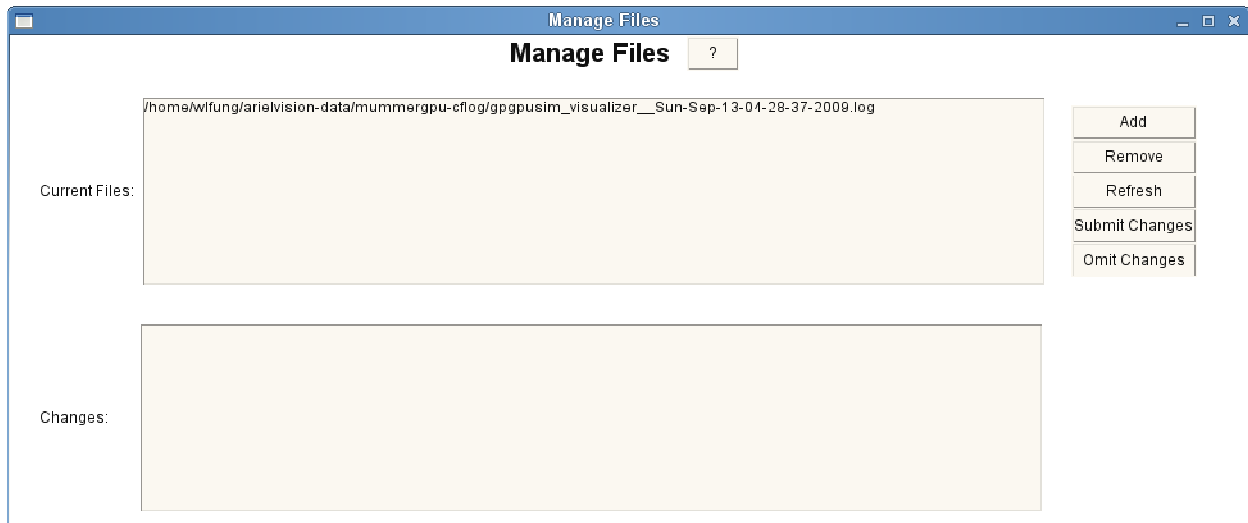
#### F. Save

Save the current figure as an image. Various extensions are available.



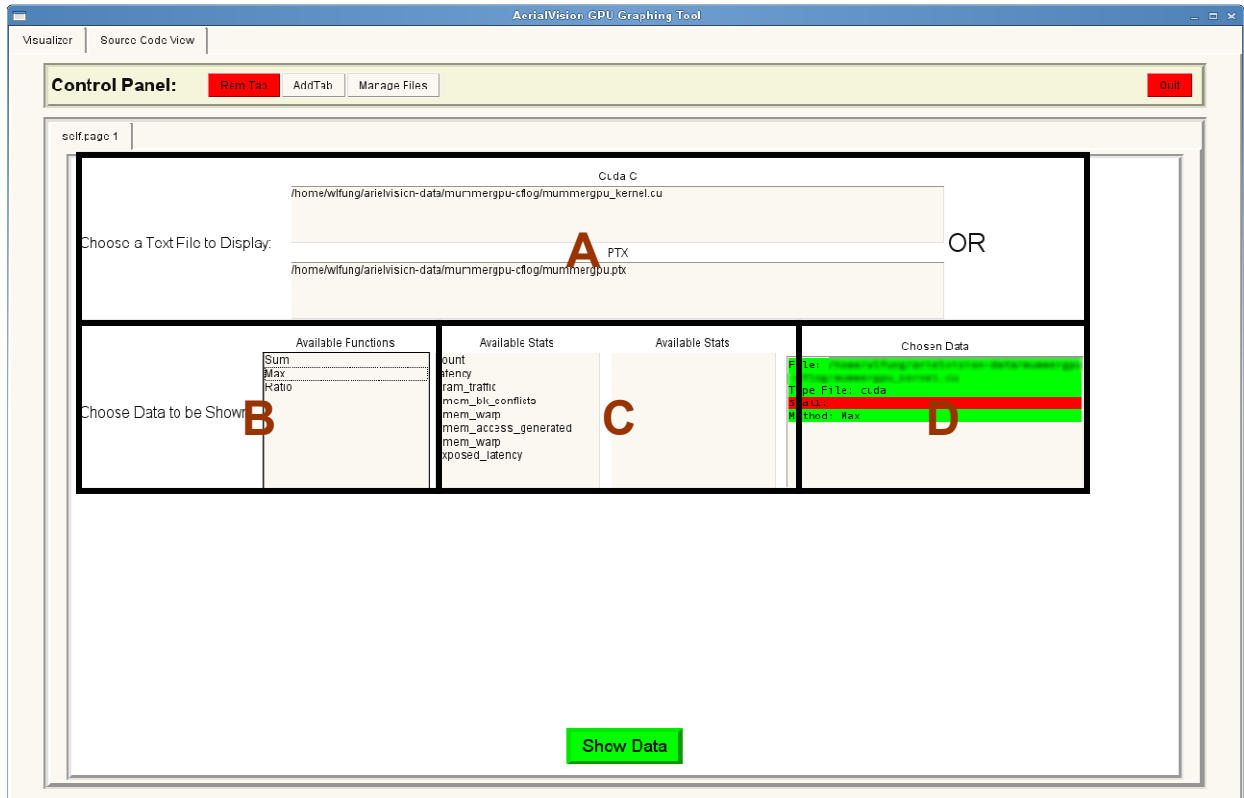
## Manage Files

The following dialog box shows up when the 'Manage Files' button is clicked. It can be used to manage (Add, Remove, Refresh) the current set of trace files that are available for visualization in AerialVision. All changes are added to the list on the bottom and will not take effect until the 'Submit Changes' button is clicked. If you are unsatisfied with the list of changes created, simply click the 'Omit Changes' button to discard all.



## Source Code Viewer Tab – Performance Metric Selection

Figure 9 depicts the startup configuration for the ‘Source Code Viewer’ tab. The user can choose the source code file to be viewed and the performance metric to be displayed along with it. A description of each labeled region is included below.



**Figure 9: Source Code Viewer (Performance Metric Selection)**

**A:** Select either a PTX or CUDA C++ source file to be viewed with the source code viewer.

**B:** In GPGPU-Sim, performance metrics are collected per line of PTX and then mapped to the appropriate line of CUDA source code. Therefore the user needs to specify how the data per line of PTX should be combined to form the data for a line of CUDA source code. For instance, the execution count for each line of CUDA source code should be the maximum execution count among its corresponding set of PTX source code. On the other hand, the total dram traffic generated by each line of CUDA source code should be the sum of the dram traffic generated by its corresponding set of PTX source code. In the current release of AerialVision, the ‘ratio’ option takes the ratio of the ‘sum’ of the two performance metrics chosen. A future release of AerialVision will provide the flexibility of choosing whether the ‘max’ or ‘sum’ should be used for a performance metric when calculating the ratios. Table 1 depicts the recommended configurations for the available performance metrics. Table 2 depicts a few example ratios that may be insightful to the user.

**C:** Choose performance metrics to be displayed along side with the selected source code.

**D:** This list displays which options you have chosen. A green highlight is representative of an option that has been selected. A red highlight is representative of an option that still needs to be chosen. You will not be able to plot anything until there is no red in this list.

**Table 1 Recommended Statistic Amalgamation Method for Performance Metrics**

<b>Performance Metric</b>	<b>Recommended Operation on Particular Statistic</b>
Count	Max
Latency	Sum
Dram_Traffic	Sum
Smem_bk_conflicts	Sum
Smem_warp	Sum
Gmem_access_generated	Sum
Gmem_warp	Sum
Exposed_latency	Sum

**Table 2 Examples of Insightful Ratios**

<b>Recommended Ratios</b>
$\text{Smem\_bk\_conflicts} / \text{Smem\_warp}$
$\text{Gmem\_access\_generated} / \text{Gmem\_warp}$
$\text{Gmem\_access\_generated} / \text{Dram\_Traffic}$

## Source Code Viewer Tab – Navigating Through Source Code with Performance Metrics

Figure 10 shows the Source Code Viewer tab with a CUDA source file opened. The user can view the source code (labeled region B) along side with the chosen performance metric (labeled region A). The viewer includes a navigation graph that plots performance metric among each line of CUDA source code (labeled region C). The user can quickly traverse to the line of source code with the data of interest by right-clicking on the graph. The navigation graph also has a tool bar (labeled region D) that allows the user to customize the looks of the graph as it does with figures in the Visualizer tab.

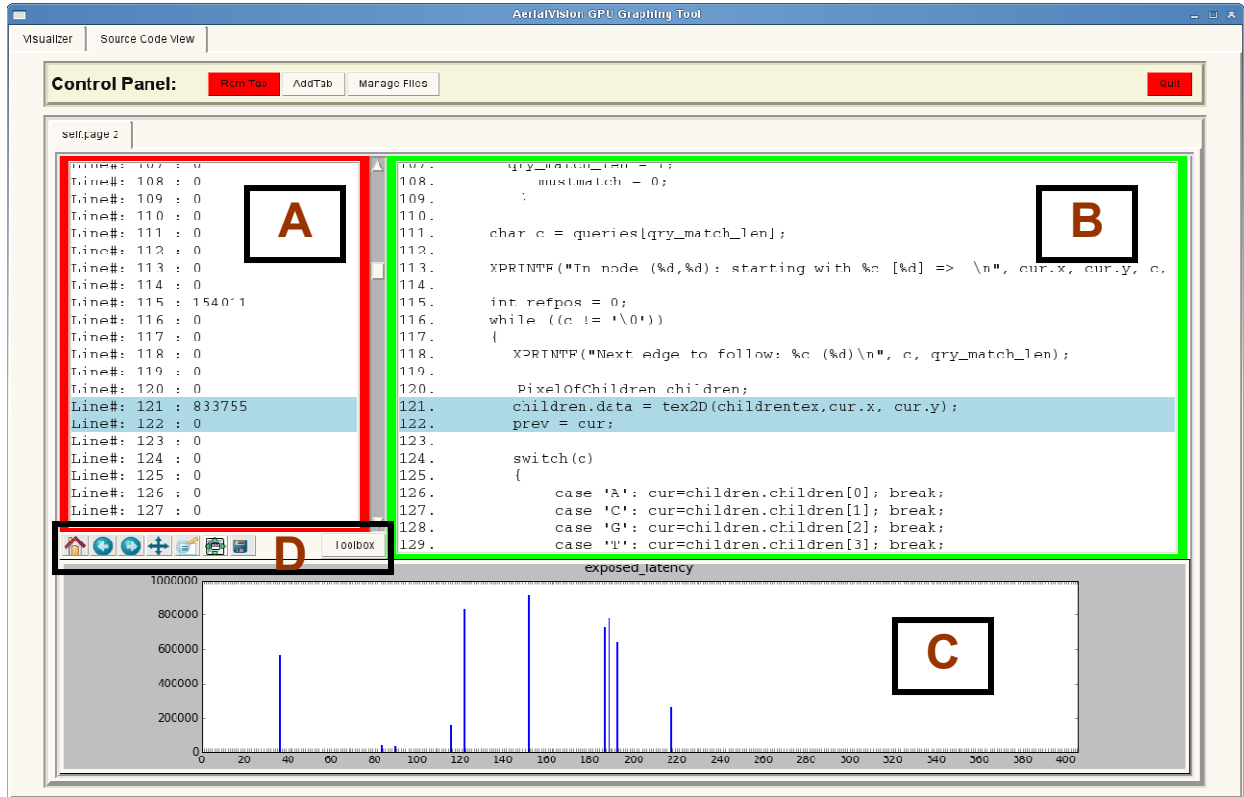


Figure 10: Source Code Viewer

## 4. Description of Available Performance Metrics

The current release of AerialVision contains numerous performance metrics that can be plotted in a variety of ways. We aim here to give the user a comprehensive understanding of each of the performance metrics. A deeper understanding of the available performance metrics should greatly increase the efficiency of the user and consequently speed up the analytical process.

Upon loading the program, there are few principal performance metrics that should be plotted first. These principal performance metrics illustrate the very top-level behavior of the benchmark (i.e. whether the benchmark was memory bound or not). A list of these top-level performance metrics can be found in Table 3. A description of each of them can be found directly thereafter.

There are many other performance metrics in AerialVision. Table 4 contains a list of these performance metrics and is also followed by a description of each of them.

It's important to note that in the current release of AerialVision, all variables are plotted as a function of cycle count.

**Table 3 General Behavior Performance Metrics.** [dy/dx = Select dy/dx for this metric.](#) [PI Plot = View this metric with Parallel Intensity Plot](#)

	General Behavior Performance Metrics	dy/dx	PI Plot
1	averagemflatency	No	No
2	dramUtil	No	Yes
3	globalInsn	Yes	No
4	Ldmemlatdist	NA	NA
5	shaderInsn	Yes	Yes
6	STmemlatdist	NA	NA
7	WarpDivergenceBreakdown	NA	NA

1. **averagemflatency** – The average amount of cycles that threads waiting for off-chip memory requests have been stalled for.
2. **dramUtil** – The percent of the full capacity that each DRAM channel is being utilized.
3. **globalInsn** – A running count of the total number of instructions that have executed. Taking the 'dy/dx' of this variable (described in previous sections of this manual) results in the more commonly known **instructions per cycle** metric.
4. **Ldmemlatdist** – A breakdown of memory read access (load) latency to reflect the amount of time that the memory accesses are spending on each part of the GPU microarchitecture during the sampling period. Here is the description of each component in the breakdown:
  - FQPUSHED – Time spent in the memory fetch queue (currently unused)
  - ICNT\_PUSHED – Time spent in interconnect buffer to DRAM
  - ICNT\_INJECTED – Time spent on traversing inside the interconnect to DRAM
  - ICNT\_AT\_DEST – Time spent waiting inside the interconnect output buffer

- DRAMQ – Time spent inside DRAM controller to have the access processed
  - DRAM\_OUTQ – Time spent at the DRAM output queue (currently unused)
  - 2SH\_ICNT\_PUSHED – Time spent in the interconnect buffer to the Shader Cores
  - 2SH\_ICNT\_INJECTED – Time spent on traversing inside the interconnect back to the Shader Cores
  - 2SH\_ICNT\_AT\_DEST – Time spent waiting inside the interconnection output buffer
  - 2SH\_FQ\_POP – Time spent at the memory interface into the Shader Cores
  - RETURN\_Q – Time spent at the return queue to writeback stage inside a Shader Core
  - WRITEBACK – Time spent waiting for the writeback to register file (always 0)
5. **shaderlnsn** – Corresponds to a running count of the total number of instructions that have executed in each shader. This performance metric is best viewed using the ‘Parallel Intensity Plot’ which improves the clarity of visualization for this metric significantly. Taking the ‘dy/dx’ of this variable (described in previous sections of this manual) results in the more commonly known **instruction per cycle** metric, viewed on a per shader basis.
  6. **STmemlatdist** – A breakdown of memory write access (store) latency to reflect the amount of time that the memory accesses are spending on each part of the GPU microarchitecture during the sampling period. See **Ldmemlatdist** for the description of each component in the breakdown.
  7. **WarpDivergenceBreakdown** – A breakdown of the number of warps warp issued for execution during the sampling period according to the number of active threads in the warp. For example, component W1:4 includes all warps with one to four active threads. Category W0 denotes the idle cycles in each SM when all the warps in the SM are waiting for data from off-chip memory; whereas category Fetch Stalled denotes the cycles when the fetch stage of an SM stalls, preventing any warp from being issued that cycle.

Table 4 Other Performance Metrics. dy/dx = Select dy/dx for this metric. PI Plot = View this metric with Parallel Intensity Plot

	Other Performance Metrics	dy/dx	PI Plot			dy/dx	PI Plot
8	cacheMissRate_constL1_all	Yes	Yes	25	dramNPRE	No	Yes
9	cacheMissRate_constL1_noMgHt	Yes	Yes	26	dramNREQ	No	Yes
10	cacheMissRate_globalL1_all	Yes	Yes	27	dramtexture_acc_r	Yes	Yes
11	cacheMissRate_globalL1_noMgHt	Yes	Yes	28	globalCompletedThreads	Yes	No
12	cacheMissRate_textureL1_all	Yes	Yes	29	globalProcessedWrites	Yes	No
13	cacheMissRate_textureL1_noMgHt	Yes	Yes	30	globalSentWrites	Yes	No
14	CFLOG	No	Yes	31	gpu_stall_by_MSHRwb	Yes	No
15	dramAveMRQS	No	Yes	32	L1ConstMiss	Yes	No
16	dramCMD	No	Yes	33	L1ReadMiss	Yes	No
17	dramconst_acc_r	Yes	Yes	34	L1TextMiss	Yes	No
18	dramEff	No	Yes	35	L1WriteMiss	Yes	No
19	dramglobal_acc_r	Yes	Yes	36	L2ReadHit	Yes	No
20	dramglobal_acc_w	Yes	Yes	37	L2ReadMiss	Yes	No
21	dramlocal_acc_r	Yes	Yes	38	L2WriteHit	Yes	No
22	dramlocal_acc_w	Yes	Yes	39	L2WriteMiss	Yes	No
23	dramNACT	No	Yes	40	shaderWarpDiv	Yes	Yes
24	dramNOP	No	Yes	41	shdrctacount	No	Yes

8. **cacheMissRate\_constL1\_all** – Cache miss rate of the L1 constant cache in each shader core. All accesses that miss the cache are accounted.
9. **cacheMissRate\_constL1\_noMgHt** – Cache miss rate of the L1 constant cache in each shader core. This cache miss rate discards any misses that are merged into an in-flight access and therefore not generating extra memory accesses.
10. **cacheMissRate\_globalL1\_all** – Cache miss rate of the L1 data cache (serves global and local memory space) in each shader core. All accesses that miss the cache are accounted.
11. **cacheMissRate\_globalL1\_noMgHt** – Cache miss rate of the L1 data cache in each shader core. This cache miss rate discards any misses that are merged into an in-flight access and therefore not generating extra memory accesses.
12. **cacheMissRate\_textureL1\_all** – Cache miss rate of the L1 texture cache in each shader core. All accesses that miss the cache are accounted.
13. **cacheMissRate\_textureL1\_noMgHt** – Cache miss rate of the L1 texture cache in each shader core. This cache miss rate discards any misses that are merged into an in-flight access and therefore not generating extra memory accesses.
14. **CFLOG** – All performance metrics that begin with ‘CFLOG’ correspond to visualizing how many threads are executing each PTX instruction or line of CUDA source. This variable can be used to associate observed performance behavior with lines of code.
15. **dramAveMRQS** – The average number of requests inside the memory controller of each DRAM channel. This metric is best visualized using the parallel intensity map.
16. **dramCMD** – The maximum number of command the memory controller of each DRAM channel can send in each sampling period.
17. **dramconst\_acc\_r** – The number of memory read accesses sent to each DRAM channel that are generated by access to constant memory space. This metric is best visualized using the parallel intensity map. When plotting using the parallel intensity plot, the ticks on the y-axis are in the form <#<sub>1</sub>,#<sub>2</sub>>. The first number corresponds to the particular DRAM channel and the second number corresponds to the particular memory bank in that DRAM channel.
18. **dramEff** – The percent of the full capacity of each DRAM channel is utilized when there is a pending request at the DRAM channel.
19. **dramglobal\_acc\_r** – The number of memory read accesses sent to each DRAM channel that are generated by access to global memory space.
20. **dramglobal\_acc\_w** – The number of memory write accesses sent to each DRAM channel that are generated by access to global memory space.
21. **dramlocal\_acc\_r** – The number of memory read accesses sent to each DRAM channel that are generated by access to local memory space.
22. **dramlocal\_acc\_w** – The number of memory write accesses sent to each DRAM channel that are generated by access to local memory space.
23. **dramNACT** – The total number of row activation command send by the memory controller of each DRAM channel in each sampling period.
24. **dramNOP** – The total number of NOP command send by the memory controller of each DRAM channel in each sampling period.

25. **dramNPRE** – The total number of precharge command send by the memory controller of each DRAM channel in each sampling period.
26. **dramNREQ** – The total number of read/write requests command send by the memory controller of each DRAM channel in each sampling period.
27. **dramtexture\_acc\_r** – The number of memory read accesses sent to each DRAM channel that are generated by access to texture memory space.
28. **globalCompletedThreads** – The total number of threads that have finished executing.
29. **globalProcessedWrites** – The total number of memory writes processed by the DRAM memory subsystem.
30. **globalSentWrites** – The total number of memory writes sent by the shader cores.
31. **gpu\_stall\_by\_MSHRwb** – The total number of pipeline stalls caused by yielding to MSHR writebacks (i.e. writing data from DRAM to register file).
32. **L1ConstMiss** – The total number of L1 constant cache misses across all shader cores.
33. **L1ReadMiss** – The total number of L1 data cache (serves global and local memory space) read misses across all shader cores.
34. **L1TextMiss** – The total number of L1 texture cache misses across all shader cores.
35. **L1WriteMiss** – The total number of L1 data cache (serves global and local memory space) write misses across all shader cores.
36. **L2ReadHit** – The total number of read hits across all L2 cache banks.
37. **L2ReadMiss** – The total number of read misses across all L2 cache banks.
38. **L2WriteHit** – The total number of write hits across all L2 cache banks.
39. **L2WriteMiss** – The total number of write misses across all L2 cache banks.
40. **shaderWarpDiv** – The total number of warp divergence occurred in each shader core.
41. **shdrctacount** – The total number of CTA's assigned to each shader core.



## **5. Extending AerialVision – Adding Variables**

1. Open the file `lexyacc.py`
  1. Scroll down to the comment "**Section 1.1 for adding a variable**". This section of the file has a long list of `<variablename> = vc.variable(<#1>,<#2>)`. Here, `<variablename>` will be the python object corresponding to the new data that you are entering into the visualizer. This object is initialized with two class variables that are explained below.
    1. `<#1>`: This number can currently be between 1 and 3 and in order to make these instruction easier to read we'll call this number the variable "type"
      1. Put a 1 if: Variable that is being submitted to the visualizer is a plot with one 'y' value for every 'x' value (X0,Y0).
      2. Put a 2 if: Variable that is being submitted to the visualizer is a plot with multiple 'y' values for every 'x' value (X0,Y0,Y1,Y2, Y3...YN).
      3. Put a 3 if: Variable that is being submitted to the visualizer will be a stacked bar plot.
    2. `<#2>`: This number is a boolean
      1. Put a 1 if: Variable that is being submitted to the visualizer resets every kernal.
      2. Put a 0 if: Variable that is being submitted to the visualizer does not reset every kernal.
  2. Scroll down to the comment "**Section 1.2 for adding a variable**". This section of the file has a long list of if/elif statements
    1. 2 scenarios here:
      1. If variable is of type 1, copy the format of BOX1 below.
      2. If variable is of type 2, copy the format of BOX2 below.
    3. Scroll down to the comment "**Section 1.3 for adding a variable**". This is at the very bottom of the file.
      1. Add to the end of the variables dictionary..., '`<variablename>`':  
`<variablename>`
2. Open the file `organizedata.py`
  1. Starting at the comment "**Section 2.1 for adding a variable**", there are 3 scenarios here:
    1. If variable is of type 1, copy the format of BOX3 below.
    2. If variable is of type 2 or 3, and is not a dram stat. copy the format of BOX4 below.
    3. If variable is of type 2 and is a Dram Stat, make sure with Aaron that you've entered the variable correctly into the simulator and follow the format of BOX5 below.

**BOX1:**

```
elif p[1].lower() == "<how variable appears in file>":  
    for x in num:  
        <variablename>.data.append(int(x))
```

**BOX2:**

```
elif p[1].lower() == "<how variable appears in file>":  
    for x in num:  
        <variablename>.data.append(int(x))  
        <variablename>.data.append("NULL")
```

**BOX3:**

```
if vars[files].has_key('<variablename>'):  
    vars[files]['<variablename>'].data = [0] + [int(x) for x in vars[files]['<variablename>'].data]
```

**BOX4:**

```
if vars[files].has_key('<variablename>'):  
    vars[files]['<variablename>'].data = nullOrganizedShader(vars[files]['<variablename>'].data)
```

**BOX5:**

```
if vars[files].has_key('<variablename>'):  
    vars[files]['<variablename>'].data = nullOrganizedDram(vars[files]['<variablename>'].data)
```