Tutorial: Finding Hotspots on an Android* Platform

Intel® VTune™ Amplifier for Systems (Linux* OS version)

C++ Sample Application Code

Legal Information

Important

This document was last updated for the Intel VTune Amplifier 2017 product release. If you are using this tutorial with a newer version of VTune Amplifier, you may see differences in analysis type names and user interface design.

Contents

Legal Information

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors which may cause deviations from published specifications. Current characterized errata are available on request.

Intel, the Intel logo, Intel Atom, Intel Core, Intel Xeon Phi, VTune and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Microsoft, Windows, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Java is a registered trademark of Oracle and/or its affiliates.

OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos.

Overview


Discover how to use Basic Hotspots Analysis with the Intel® VTune™ Amplifier for Systems to understand where your Android* application is spending time. By identifying *hotspots* - the most time-consuming program units - you can understand how effectively your code is using available cores and other system resources. By identifying the causes of ineffective performance and utilization you can correct them and improve your application performance.

This document applies to the analysis of Android applications using the VTune Amplifier in the Intel® System Studio. If you are using a different Intel Studio, please refer to that Studio-specific supplemental documentation in <install-dir>/<Intel_studio>/documentation/.

About This Tutorial	This tutorial uses the sample `tachyon` and guides you through the basic steps required to use the GUI to analyze the code for hotspots by means of remote data collection.
Estimated Duration	• 10 minutes: Preparing your target device for use • 10 minutes: Preparing your sample application and analyzing it
Learning Objectives	After you complete this tutorial, you will be able to: • Set up the Android system for analysis • Build the `tachyon` package and create a new project • Run Basic Hotspots Analysis • Interpret result data • Compare results
More Resources	• The Intel® Developer Zone is a site devoted to software development tools, resources, forums, blogs, and knowledge bases, see http://software.intel.com • The Intel Software Documentation Library is part of the Intel Developer Zone and is an online collection of Release Notes, User and Reference Guides, White Papers, Help, and Tutorials for Intel software products http://software.intel.com/en-us/intel-software-technical-documentation

Start Here

Navigation Quick Start

1

Intel® VTune™ Amplifier for Systems provides information on code performance for users developing serial and multithreaded applications for Windows*, Android*, and Linux* operating systems. VTune Amplifier helps you analyze algorithm choices and identify where and how your application can benefit from available hardware resources.

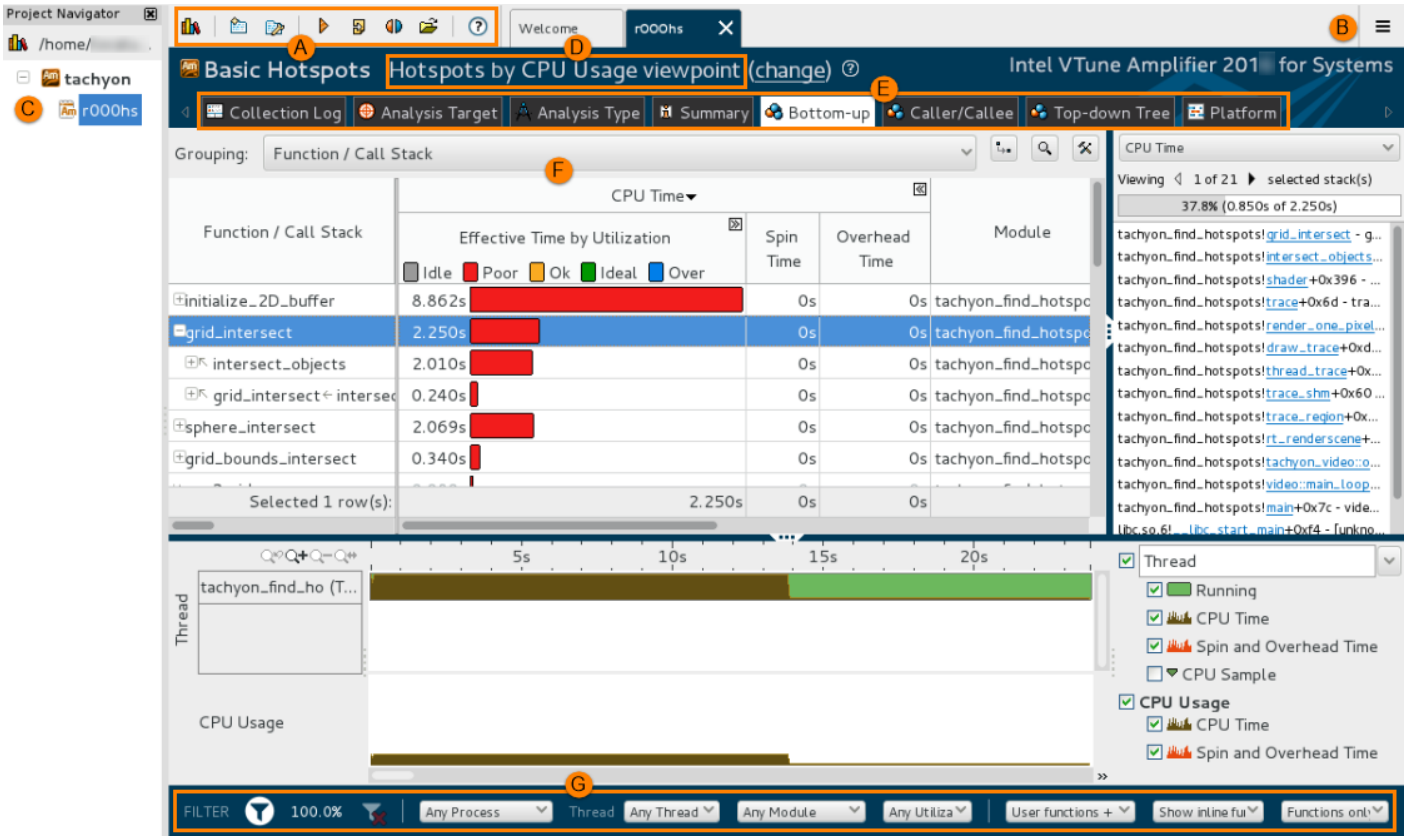
VTune Amplifier Access

VTune Amplifier installation includes shell scripts that you can run in your terminal window to set up environment variables:

1. From the installation directory, type `source amplxe-vars.sh` for bourne or korn or bash shells, or `source amplxe-vars.csh` if you are using the C shell.

   This script sets the PATH environment variable that specifies locations of the product graphical user interface utility and command line utility.

   The default installation directory is `/opt/intel/system_studio_201n/vtune_amplifier_201n_for_systems`.
2. Type `amplxe-gui` to launch the product graphical interface.



A — Configure and manage projects and results, and launch new analyses from the primary toolbar. Click the **Configure Project** button on this toolbar to manage result file locations. Newly completed and opened analysis results along with result comparisons appear in the results tab for easy navigation.

B — Use the VTune Amplifier menu to control result collection, define and view project properties, and set various options.

C — The **Project Navigator** provides an iconic representation of your projects and analysis results. Click the **Project Navigator** button on the toolbar to enable/disable the **Project Navigator**.

D — Click the **(change)** link to select a *viewpoint*, a preset configuration of windows/panes for an analysis result. For each analysis type, you can switch among several viewpoints to focus on particular performance metrics. Click the question mark icon to read the viewpoint description.

E — Switch between window tabs to explore the analysis type configuration options and collected data provided by the selected viewpoint.

F — Use the **Grouping** drop-down menu to choose a granularity level for grouping data in the grid.

G — Use the filter toolbar to filter out the result data according to the selected categories.

Next Step

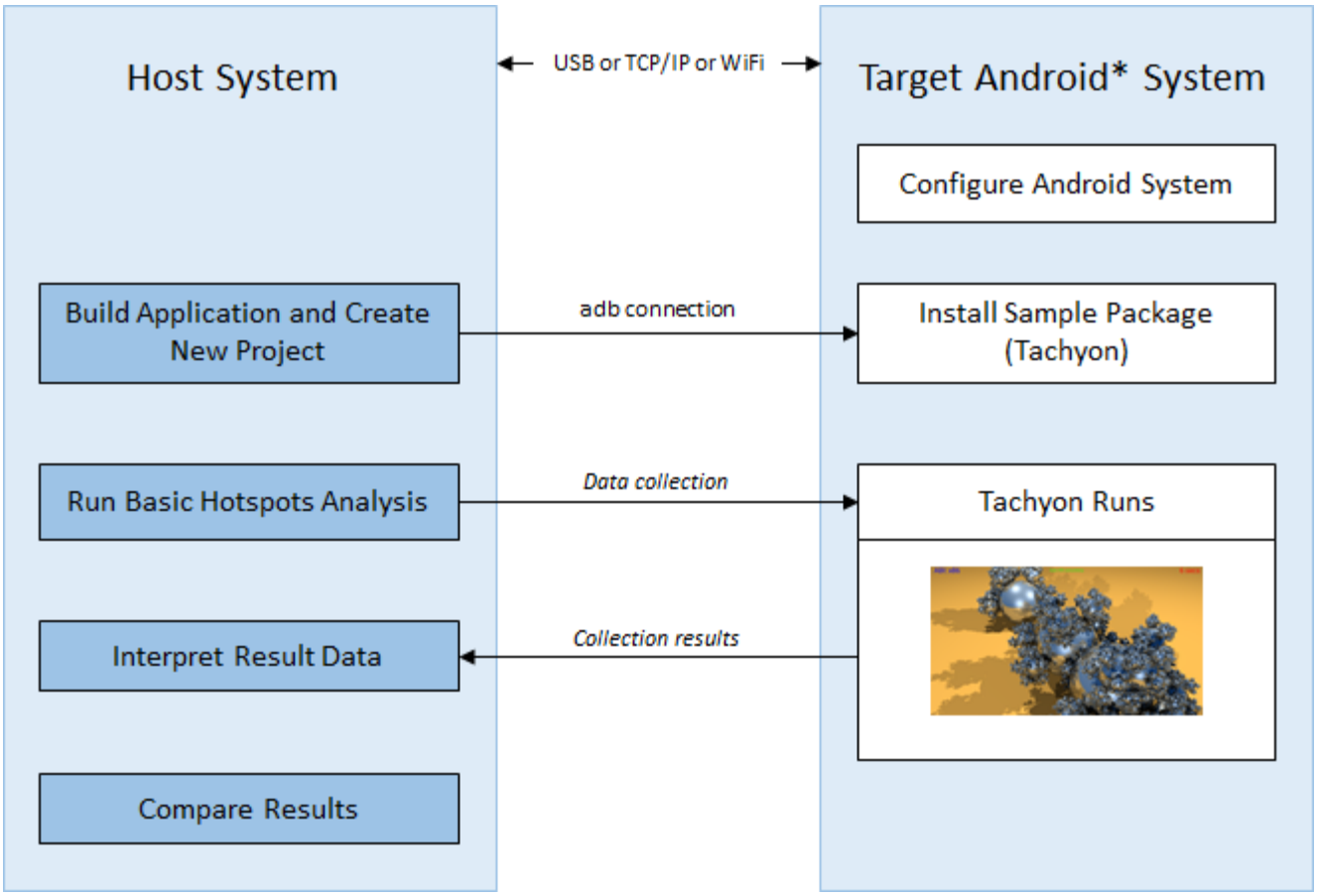
Finding Hotspots

Finding Hotspots

2

You can use the Intel® VTune™ Amplifier for Systems installed on your Linux* host to identify and analyze hotspot functions in your serial or parallel Android* application by performing a series of steps in a workflow. This tutorial guides you through these workflow steps while using a sample ray-tracer application named `tachyon`.

VTune Amplifier for Systems supports this remote usage mode, using the VTune Amplifier target package and ADB communication:



1	Setup the Android System	**1.** Configure your Android device for analysis. **2.** Gain `adb` access through a USB connection, WiFi, or TCP/IP.
2	Build the Application and Create a New Project	**1.** Prepare your Android application for analysis. **2.** Specify your analysis target and remote system. **3.** Specify binary and source search directories.
3	Run Basic Hotspots Analysis	**1.** Choose the Basic Hotspots analysis type. **2.** Run the analysis from the host.

4	Interpret Result Data	1. View the collected data on the host. 2. Consider project views and reports to identify bottlenecks or unexpected application behavior. 3. Analyze your code.
5	Compare Results	1. View performance results for a single-threaded application. 2. Examine differences between results.

Next Step

Set up Android System for Analysis

Set Up the Android* System for Analysis

Configure your Android* device for analysis. For this tutorial you do not need to install an Intel® VTune™ Amplifier device driver, but must have `adb` installed and configured on your host system.

> **NOTE**
> For information on obtaining a supported device on which to test your own Android application or enabling Java Performance Analysis on Android devices:
>
> - Enabling Java Performance Analysis on Android Devices Using Intel® VTune™ Amplifier for Systems at https://software.intel.com/en-us/articles/enabling-java-analysis-on-android-using-vtune-amplifier-2014-for-systems.
> - Intel Mobile Development Kit for Android at https://software.intel.com/en-us/intel-mobile-development-kit-for-android.

Allow a debug connection to enable `adb` access:

1. On your Android device, select **Settings > About <*device*>**
2. Tap **Build number** seven times to enable the **Developer Options** tab
3. Select the **Settings > Developer Options** and enable the **USB debugging** option

> **NOTE**
> The path to **Developer Options** may vary depending on the manufacturer of your device and system version.

Occasionally it is difficult or impossible get `adb` access to a device over USB. You may also get `adb` over Ethernet or WiFi.

For example, to connect via WiFi access point, do the following:

1. Connect the device via USB and select **Always Accept** when the device asks for permission to allow connections.
2. Find the IP Address of the target. The IP address is available in **Settings > About <*device*>> Status**.
3. Make sure `adb` is enabled on the target device. If not enabled, go to a Terminal application (of your choice with root access) on the device and type:

```
$ su
$ setprop service.adb.tcp.port 5555
$ stop adbd; start adbd
```

> **NOTE**
> If you do not have a terminal application with root access, you can run the command via `adb` when connected to the device via USB.

4. Connect `adb` on the host to the remote device. In the Command Prompt or the Terminal on the host, type:

```
$ adb connect <IPAddress>:5555
```

While you do not require root mode `adb` access to the Android device to run this tutorial, you do require it to:

- Change the port and start the service for access via WiFi
- Enable the Android device to support Java analysis
- Install and load the drivers needed for hardware event-based sampling
- Run hardware event-based sampling analysis

> **NOTE**
> For instructions on how to gain a root mode `adb` access to your device, see the *Intel VTune Amplifier for Systems Installation Guide* or "Preparing a Target Android System for Remote Analysis" in the VTune Amplifier online help. On some versions of Android systems, including most of the Intel-supplied reference builds for software development vehicles (SDVs), the required drivers are pre-installed in `/lib/modules` or `/system/lib/modules`. If the drivers are not pre-installed in any of these directories, you need to build them manually from the command line. You do not need to install the Android target package manually since it is automatically installed when you start collection.

Next Step

Build Application and Create New Project

Build Application and Create New Project

Follow these steps before analyzing your sample `tachyon` application target for hotspots.

Get Software Tools

If you do not have them already, get and install these software tools:

- Android* SDK and NDK
- Intel® VTune™ Amplifier for Systems
- Eclipse* or other development environment (optional)

  or
- Apache Ant* (to compile outside of Eclipse or other development environment)

Optionally, update your PATH environment variable to include the location of the Android SDK `platform-tools`, `tools`, and `ndk` directories and the location of your Apache Ant `bin` directory, if applicable.

> **NOTE**
> The following steps use Eclipse Mars.1 Release (4.5.1) and Apache Ant 1.9.6. Other development environments may require alternate steps.

Build the Application

1. Extract the `/opt/intel/vtune_amplifier_<version>_for_systems/samples/en/C++/tachyon_vtune_amp_xe.tar.gz` file to a writeable location on your host system. This file contains the `tachyon\android` directory, which includes the sample application source files used in this tutorial.
2. Specify the location of your Android NDK and SDK directories.

a. Navigate to the `tachyon\android` directory and open the `local.properties` file with a text editor.
b. Add the location of your Android SDK and Android NDK directories. For example:

```
sdk.dir=/home/uid/Android/Sdk
ndk.dir=/opt/android/android-ndk-r10e
```

c. Save and close the file.

3. Compile the C++ code and create the binaries.
   a. Open a command prompt and navigate to the `tachyon\android` directory.
   b. Run the following command: `ndk-build`
4. Build the application `.apk` file.

- Using Eclipse:
  1. Open the `Tachyon` project as an **Android Project from Existing Code** in Eclipse.
  2. Click **Run**.
  3. Select your Android device from the **Choose a running Android device list** and click **OK**.

     The `Tachyon.apk` file is built in the `tachyon\android\bin` directory and installed on the device.
  4. Exit Eclipse.
- Using a command prompt and Apache Ant:
  1. Run the following command from the `tachyon\android` directory: `ant clean`
  2. Run the following command to create the `.apk` file in the `tachyon\android\bin` directory: `ant debug`
  3. Install the `Tachyon-debug.apk` file on the Android device using the following `adb` command:

     ```
     adb install bin\Tachyon-debug.apk
     ```

     You can confirm that the package has installed by either running `Tachyon sample` on the device, or by listing out the installed packages:

     ```
     adb shell pm list packages | grep tachyon
     ```

Create a Project

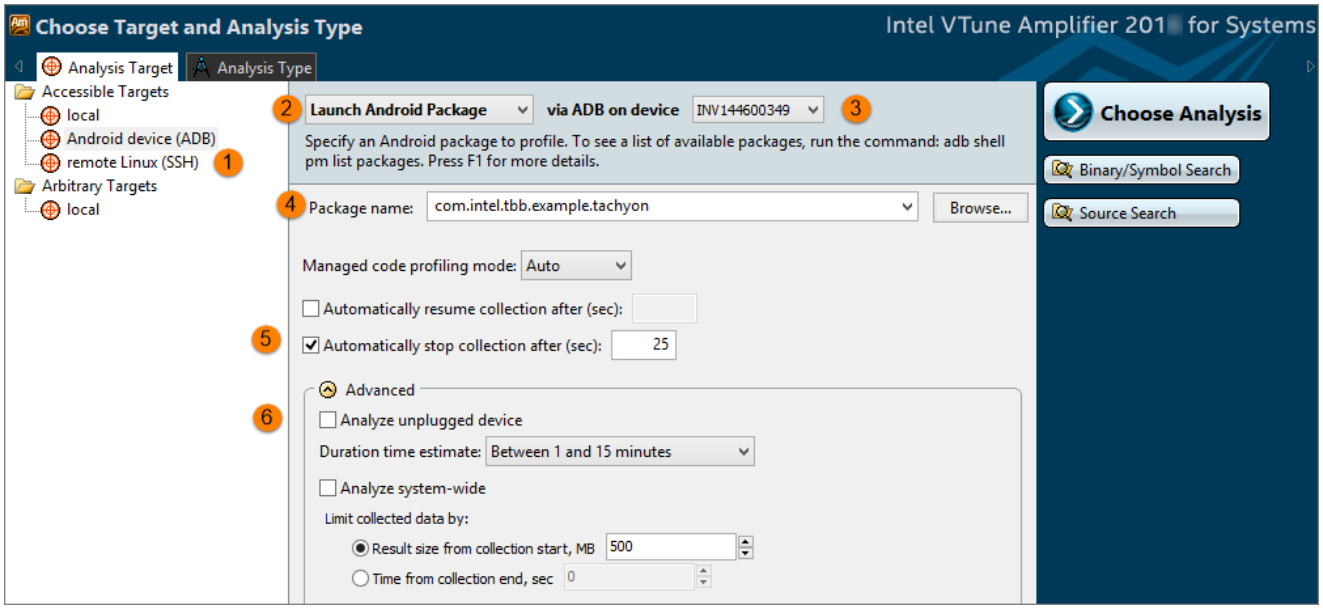
1. From the `<install_dir>/bin32` directory (for IA-32 architecture) or from the `<install_dir>/bin64` directory (for Intel 64 architecture), run the `amplxe-gui` script to launch the VTune Amplifier GUI.

   NOTE

   - For super-users: `/opt/intel/system_studio_`*`version`*`/vtune_amplifier_for_systems`
   - For ordinary users: `$HOME/intel/system_studio_`*`version`*`/vtune_amplifier_for_systems`

2. Create a new project via **New > Project...** or using the new project action  from the toolbar. The **Create a Project** dialog box opens.
3. Specify a project name such as `tachyon` that will be used as the project directory name. Click **Create Project**.

   VTune Amplifier creates the `tachyon` project directory under the `$HOME/projects/` directory or the location you specified and opens the **New Amplifier Result** page with the **Analysis Target** tab active.
4. In the **Analysis Target** tab, specify your target as follows:



1	Target system: **Accessible Targets > Android device (ADB)**
2	Target type: **Launch Android Package**
3	Via ADB on device: Select device from drop-down list. In the example, the Android device is identified as INV144600349.
4	Package name: Either click **Browse** and select the package named `com.intel.tbb.example.tachyon` from the list, or enter it directly as the **Package Name**.
5	Select **Automatically stop collection after (sec)** to a value in seconds. The sample shows 25. You can also choose to click **Stop** after a single iteration of the `tachyon` application. You will see a 3-D fractal image appear, blank out, and start to reappear again.
6	[Optional] Select **Analyze unplugged device** to run the analysis while the device is disconnected from the host system. Data collection begins when the device is disconnected and the results are finalized and displayed in VTune Amplifier after the device is connected again.

5. Click the **Binary/Symbol Search** button, add the directories listed below for the binary with symbols to the search paths, and click **OK**.

   The symbols are created when Android applications are built, but are stripped off before you place the package on the device.

NOTE
Use the `x86` directory when running on a 32-bit Android device and the `x86_64` directory when running on a 64-bit Android device.

Add the following directories to see the relevant symbol information:

- `\tachyon\android\obj\local\<x86 or x86_64>`
- `\tachyon\android\obj\local\<x86 or x86_64>\objs-debug\abi\jni`
- `\tachyon\android\obj\local\<x86 or x86_64>\objs-debug\jni-engine\jni`
- `\tachyon\android\obj\local\<x86 or x86_64>\objs-debug\jni-engine\src\tachyon_src`
- `\tachyon\android\obj\local\<x86 or x86_64>\objs-debug\jni-engine\src\tbb\common\gui`

NOTE
Only the symbol information for functions called by the `tachyon` sample application are included in this list. As a result, any functions called by operations other than those in the `tachyon` sample, including functions called by the Android operating system, will not be mapped in VTune Amplifier. These functions will appear as `funct@<value>` rather than with the function name and the call stack information will not be available.

6. Click the **Source Search** button, add the directory for the binary with symbols to the search paths, and click **OK**.

   Add the following directories to see the relevant search information:

   - `\tachyon\android\src\tachyon_src`
   - `\tachyon\android\src\tachyon_src\gui`
   - `\tachyon\android\src\tachyon_src\tbb\common\gui`
   - `\tachyon\android\src\tachyon_src\tbb\src`
   - `\tachyon\android\src\tbb\common\gui`

   NOTE
   Only the source information for functions called by the `tachyon` sample application are included in this list. As a result, any functions called by operations other than those in the `tachyon` sample, including functions called by the Android operating system, will not be mapped in VTune Amplifier. These functions will appear as `funct@<value>` rather than with the function name and the call stack information will not be available.

7. Click **Choose Analysis** to apply the settings and switch to the **Analysis Type** tab.

Next Step

Run Basic Hotspots Analysis

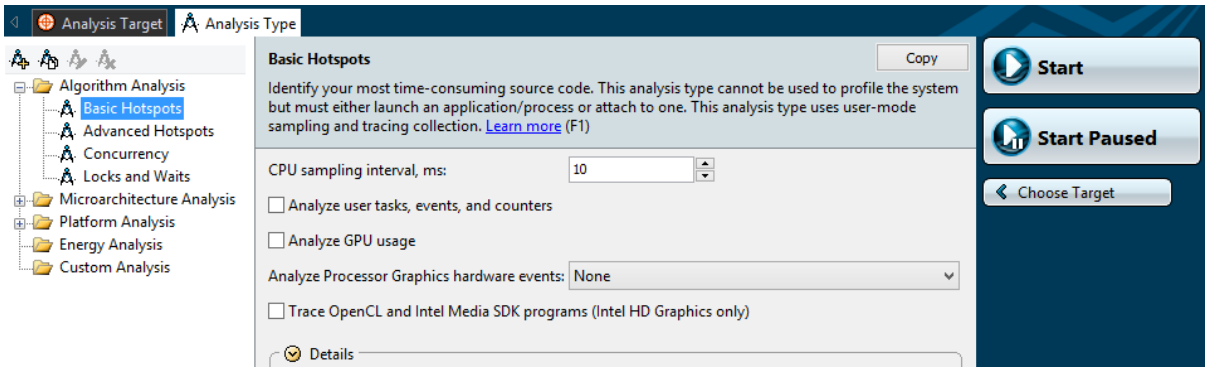
Run Basic Hotspots Analysis


Before running an analysis, choose a configuration level to regulate Intel® VTune™ Amplifier analysis scope and running time, including the size of the data collection files. In this tutorial, you run the Basic Hotspots Analysis to identify the hotspots that took the most time to execute.

Basic Hotspots analysis requires no sampling drivers, but sampling drivers are required for hardware event-based analysis. Refer to the *Intel VTune Amplifier for Systems Installation Guide* or the "Preparing a Target Android* System for Remote Analysis" help topic for more information.

Run an Analysis

1. If it is not already displayed, select the **Analysis Type** tab.
2. On the left pane of the **Analysis Type** window, locate the analysis tree and select **Algorithm Analysis > Basic Hotspots** . The right pane is updated with the predefined settings for the **Basic Hotspots** analysis.
3. Click the **Start** button on the right command bar.



NOTE
After you click the **Start** button, VTune Amplifier first installs the remote collector on the Android device before running `tachyon` and collecting data.

4. If you selected the **Analyze unplugged device** option when you configured the analysis target, unplug the device when prompted either by VTune Amplifier on your host system or by the VTune Amplifier remote collector on your Android device.

Data Collection

VTune Amplifier launches the `tachyon` application which displays a 3-D fractal image and its own execution time, and will continue running until you press **Stop** or until the duration of the sampling session that you set up in seconds has been reached. If you ran the analysis on a disconnected device, reconnect the device when prompted. The VTune Amplifier finalizes the collected results and opens the analysis results in the **Hotspots by CPU Usage** viewpoint.

To make sure the performance of the application is repeatable, go through the entire tuning process on the same system with a minimal amount of other software executing.

NOTE
By using advanced features of the VTune Amplifier you can include Android framework events on your results timeline. This allows you to draw correlations between the code that is running on your CPU and events that are spawned by the Android framework, such as *jank* - dropped frames which make apps smoother. For details on how to set up these events in your Basic Hotspots Analysis refer to Using Android Framework Events.

Next Step

Interpret Result Data

Interpret Result Data

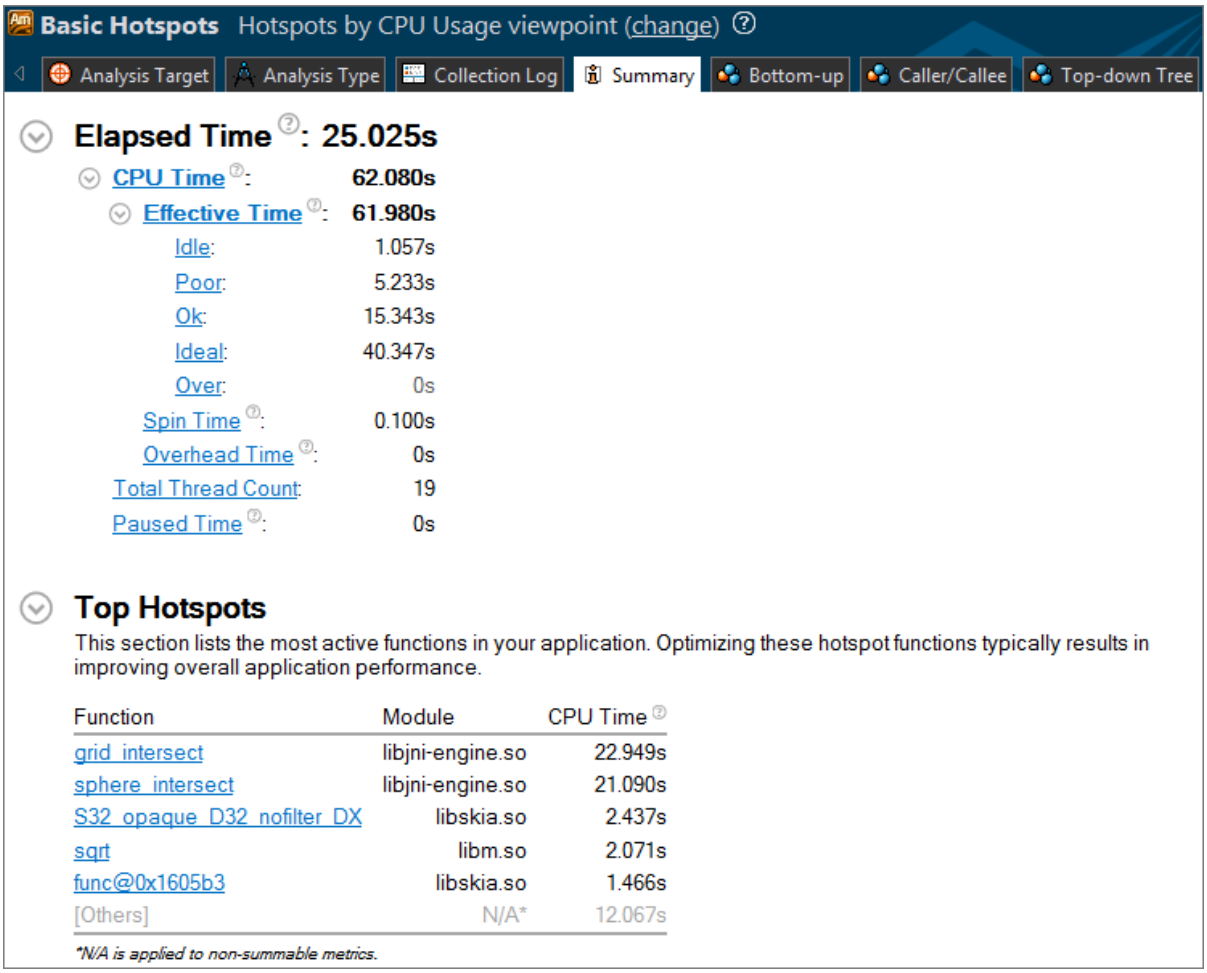
The Android* application `tachyon` runs on the Android device, rendering the 3-D fractal image in cycles. When `tachyon` exits, the Intel® VTune™ Amplifier finalizes the results and opens the **Hotspots by CPU Usage** viewpoint where each window or pane is configured to display code regions that consumed a lot of CPU time. To interpret the data on the sample code performance, do the following:

- Understand the basic performance metrics provided by the Basic Hotspots analysis.
- Analyze the most time-consuming functions and CPU usage.
- Analyze timeline data.

NOTE
The screenshots and execution time data provided in this tutorial are created on a system with four CPU cores with an Intel Atom™ processor running Android version 5.1.1 (Lollipop). Your data may vary depending on the number and type of CPU cores on your Android device.

Understand the Basic Hotspots Metrics

Start analysis with the **Summary** window. Hover over the question mark icons to read the pop-up help and better understand what each performance metric means.

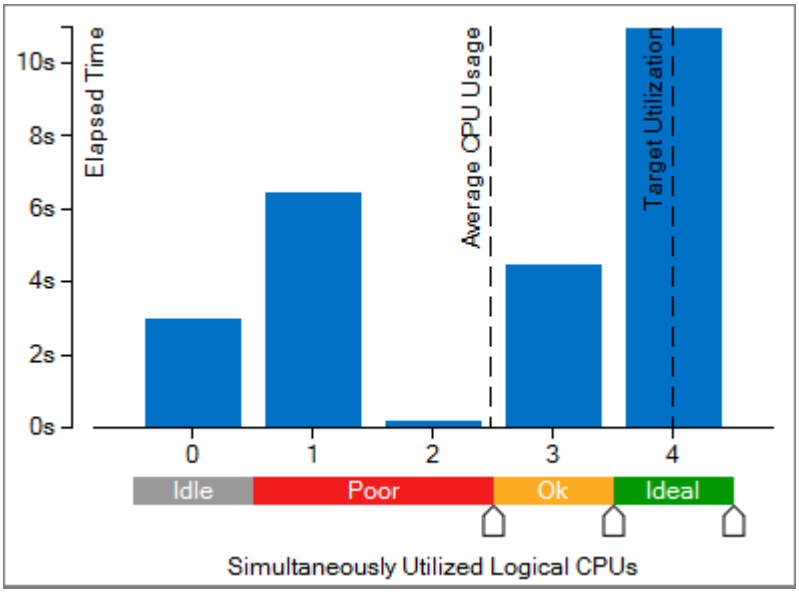


The **CPU Time** for the `tachyon` sample application is equal to 62.080 seconds. It is the sum of **CPU Time** for all application threads. **Total Thread Count** is 19, so multiple threads ran during data collection.

The **Top Hotspots** section provides data on the most time consuming functions (hotspot functions) sorted by **CPU Time** spent on their execution.

- The `grid_intersect` function, which took 22.949 seconds to execute, shows up at the top of the list as the hottest function.
- The second hottest function is `sphere_intersect`, which took 21.090 seconds to execute.
- The `S32_opaque_D32_nofilter_DX` function is part of the underlying Android operating system and is not relevant to the `tachyon` sample application analysis. When testing your own application, watch for functions that may be system processes running behind the scenes that may impact performance.
- The `funct@0x1605b3` function appears with a pseudo name because the source file was not provided during the analysis target setup.
- The `[Others]` entry at the bottom shows the sum of CPU time for all functions not listed in the table, at 12.067 seconds.

The **CPU Usage Histogram** represents the elapsed time and usage level for the available logical processors.



The `tachyon` sample application ran on one, three, or four logical CPUs most of the time. If you hover over each bar, a tooltip appears listing the total amount of time spent with each CPU grouping. You can reset the Poor, Ok, and Ideal values using the slider tabs below the histogram. For example, you may consider running three or four logical CPUs to be ideal usage.

To understand which functions caused the device to spend time in Poor utilization, explore the **Bottom-up** pane.

Analyze the Most Time-consuming Functions and CPU Usage

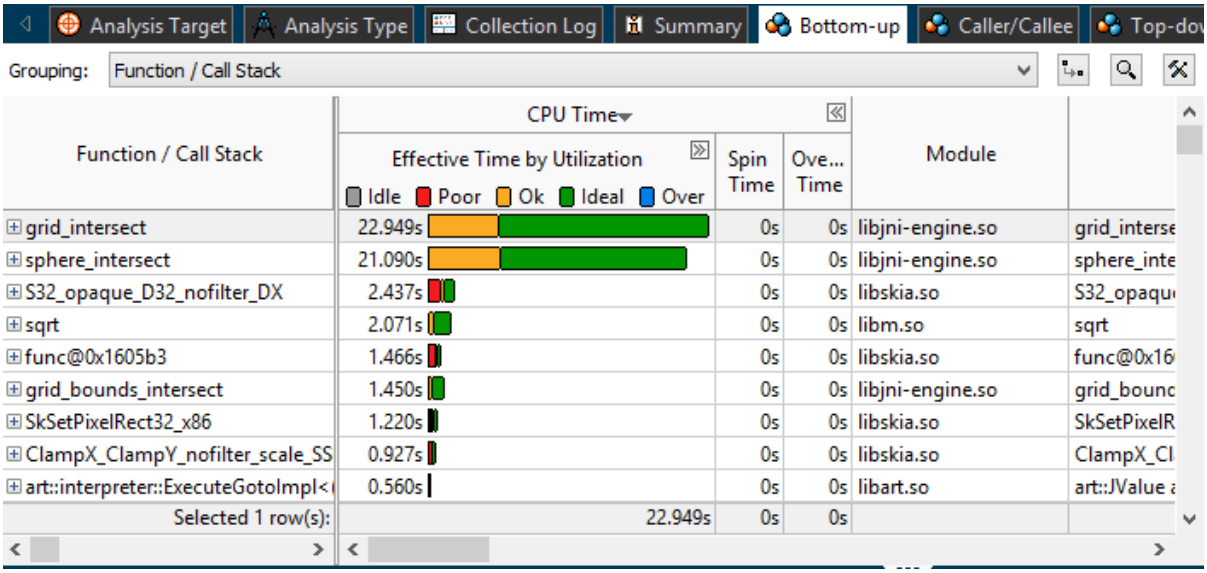
Click the **Bottom-up** tab. Change the grouping level using the **Grouping** drop-down menu at the top of the grid to **Function / Call Stack**.

Examine the **Effective Time by Utilization** column. Functions that took the most CPU time to execute are listed on top.

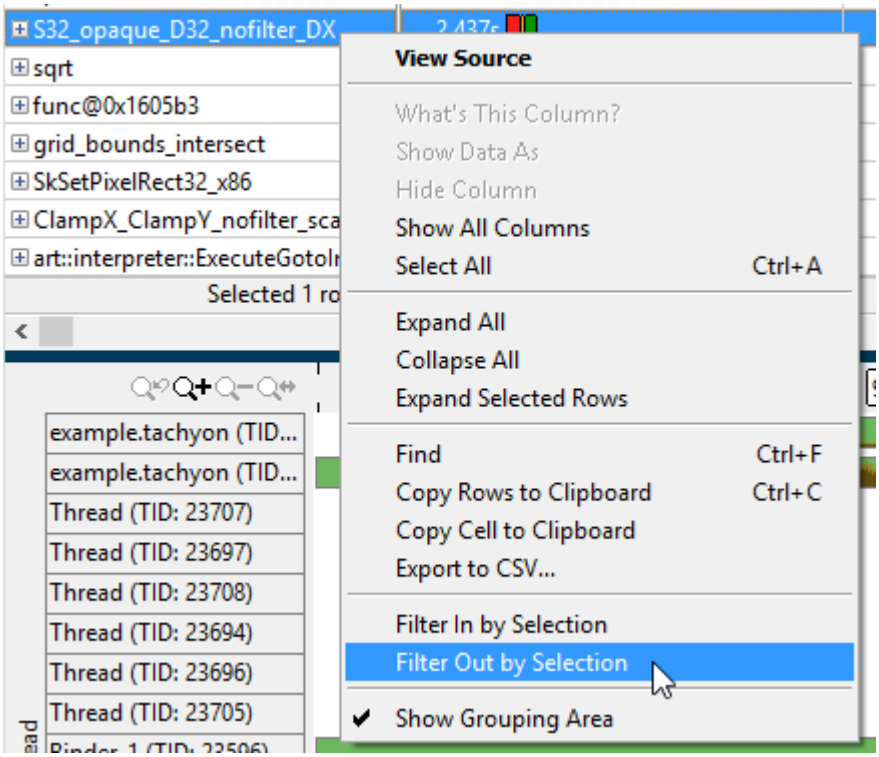
The `grid_intersect` function took the maximum time to execute, 22.949 seconds, and had either OK or Ideal CPU utilization. The `S32_opaque_D32_nofilter_DX` function had primarily Ideal or Poor CPU utilization (red bars). Poor CPU utilization means that the processor cores were not well utilized most of the time spent on executing this function.

NOTE

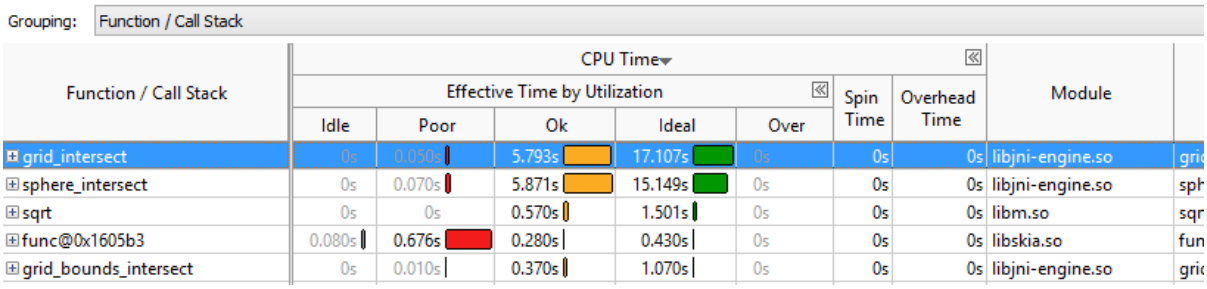
You may change the stack representation to a "tree" style by clicking the **View stacks as a tree** button located in the upper right of the menu.



Since the `S32_opaque_D32_nofilter_DX` function is not related to the `tachyon` sample application, the data can be filtered to remove this function. Right-click on the function name in the grid and select **Filter Out by Selection** . The window refreshes and the `S32_opaque_D32_nofilter_DX` function is removed from the grid and timeline.

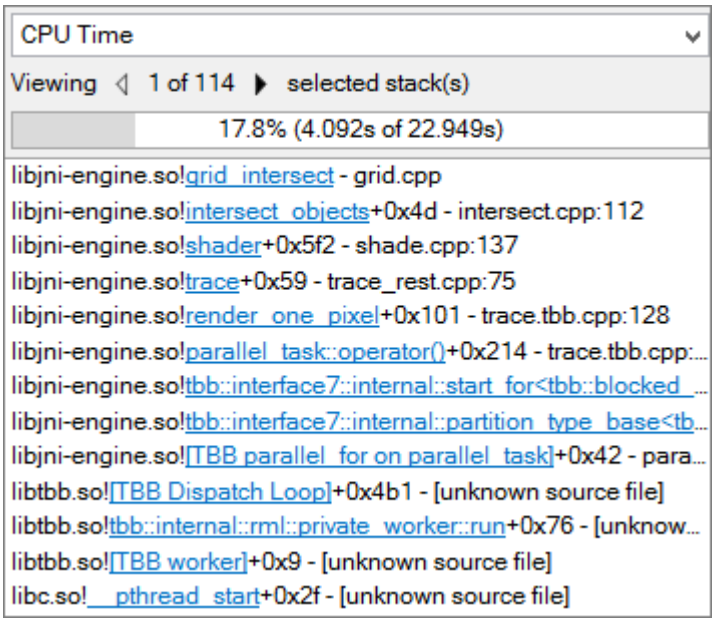


To see the detailed CPU usage information per function, use the Expand button in the **Bottom-up** pane to expand the **Effective Time by Utilization** column.



Select the `grid_intersect` function in the grid and explore the data provided in the pane on the right. The **Call Stack** pane displays full stack data for each hotspot function, enabling you to navigate between function call stacks and understand the impact of each stack to the function CPU time. The stack functions in the **Call Stack** pane are represented in the following format.

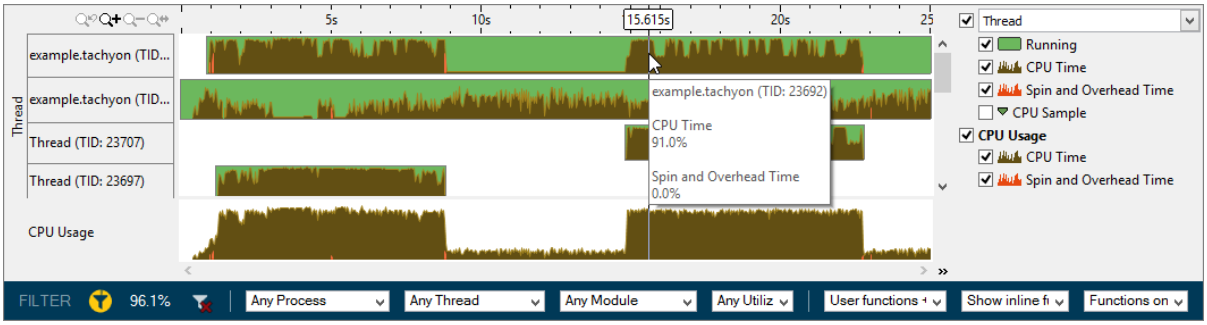
```
<module>!<function> - <source code file>
```



For the sample application, the hottest function `grid_intersect` is located in the `libjni-engine` module in the `grid.cpp` source file. You can double click the module to view the source code.

Analyze Timeline Data

To get detailed information on the thread performance, explore the **Timeline** pane.



The threads area shows the distribution of CPU time utilization per thread. Hover over a bar to see the CPU time utilization in percent for this thread at each moment of time. Green zones the times threads are active. The time that has passed since the application was launched is displayed at the top of the timeline.

The CPU Usage area shows the distribution of CPU time utilization for the whole application. Hover over a bar to see the application-level CPU time utilization in percent at each moment of time.

The application caused a spike in CPU activity at regular intervals during collection. Click and drag to select an area of interest and select the **Zoom In and Filter In by Selection** action to view one of these areas in more detail. Use the **Filter** bar at the bottom of the window to explore data related to a certain process, thread, or module. In this example, only 96.1% of data is shown because the `S32_opaque_D32_nofilter_DX` was removed.

Next Step

Compare Results

Compare Results

Intel® VTune™ Amplifier provides a comparison mechanism that helps identify performance improvements or degradation on subsequent analysis runs. Typically you would make a change to optimize the code and run a second analysis to determine the degree to which the optimization improved application performance. Rather than optimizing the `tachyon` sample code, you can run a second Basic Hotspots Analysis on the `tachyon` package running on a single thread to reveal a different analysis result.

Run Basic Hotspots Analysis

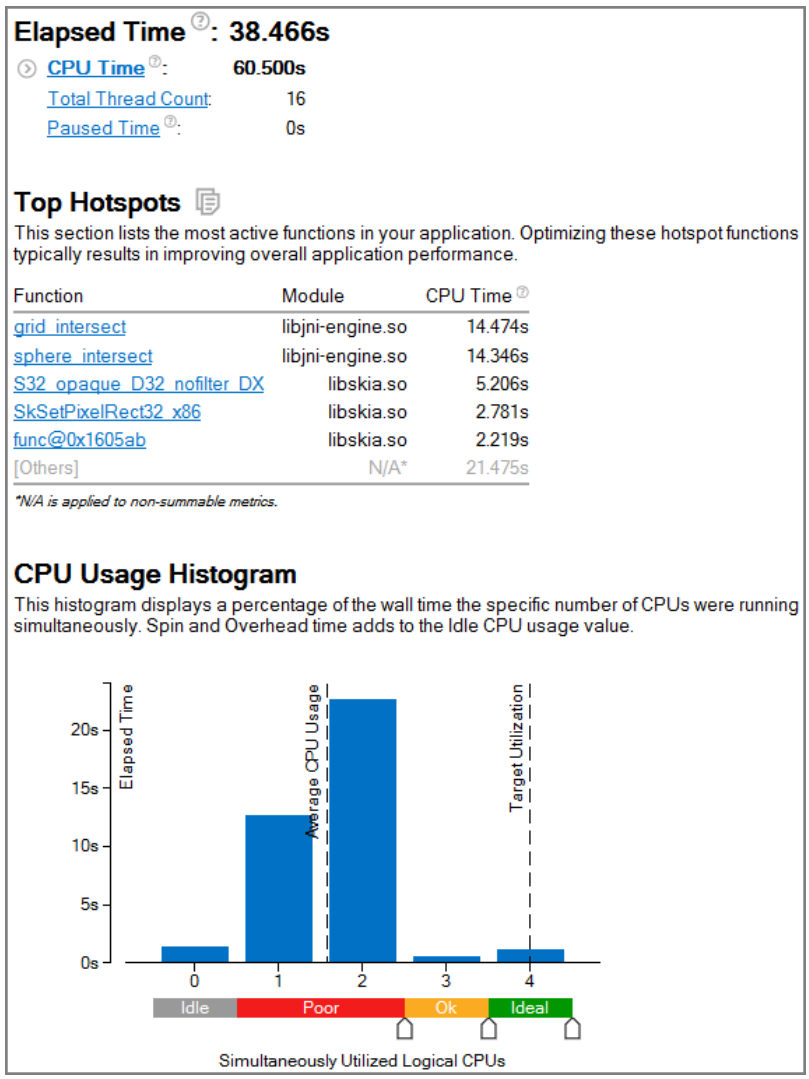
1. Run the Basic Hotspots analysis again on the same Android* device.

   NOTE
   Consider removing the option to automatically stop collection after a certain interval and stopping the collection manually to allow the application to render the image fully. To do this, switch to the **Analysis Target** tab and uncheck the **Automatically stop collection after (sec)** option.

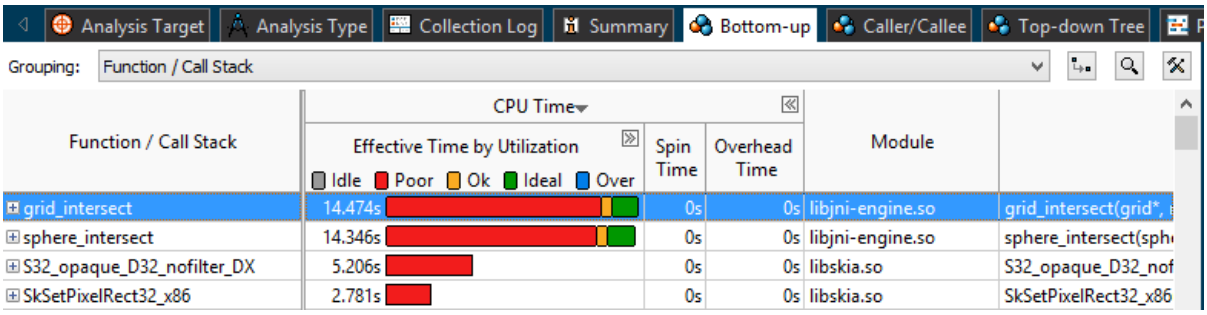
2. When the application launches, tap the top of the device screen and select **1 THREAD** to run the application on a single thread.

Review the Results

In the single-thread version of the application, the hotspots remain the same, but the **CPU Usage Histogram** shows that the sample application ran on one or two logical CPUs rather than three or four when run on multiple threads.



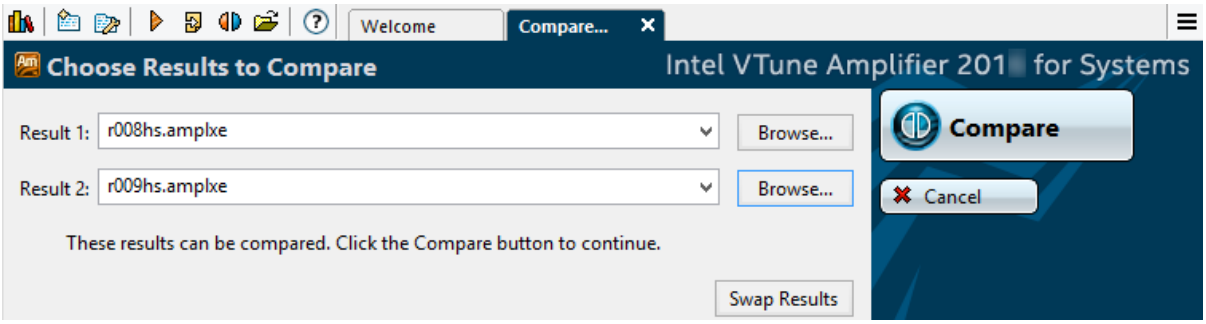
Exploring the **Bottom-up** tab shows that the `grid_intersect` and `sphere_intersect` functions spent the most time with Poor CPU utilization (red bars).



Compare to Previous Result

Compare the new result run on a single thread with the original result run on multiple threads.

1. Close the analysis result tabs if they are open.
2. Click the **Compare Results** button on the toolbar.
3. Specify the results you want to compare.



4. Click **Compare**.

   A **Summary** window opens that provides the statistics for the differences between the collected results.

Examine the data on the **Summary** window to see the differences in hotspot and CPU usage between runs. The list of **Top Hotspots** shows the differences in CPU time for the top functions for each result. The first number is for the first run using the automatic number of threads. The second number is for the second collection using a single thread. The **Top Hotspots by Difference** list shows the hotspots that had the greatest performance difference between runs.

Elapsed Time: 25.025s - 38.466s = -13.441s

CPU Time:	62.080s - 60.500s = 1.580s
Total Thread Count:	19 - 16 = 3
Paused Time:	Not changed, 0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
grid_intersect	libjni-engine.so	22.949s - 14.474s = 8.475s
sphere_intersect	libjni-engine.so	21.090s - 14.346s = 6.744s
S32_opaque_D32_nofilter_DX	libskia.so	2.437s - 5.206s = -2.768s
sqrt	libm.so	2.071s - 1.499s = 0.572s
func@0x1605b3	libskia.so	1.466s - [Unknown] = 1.466s
[Others]	N/A*	12.067s - 24.976s = -12.909s

*N/A is applied to non-summable metrics.

Top Hotspots by Difference

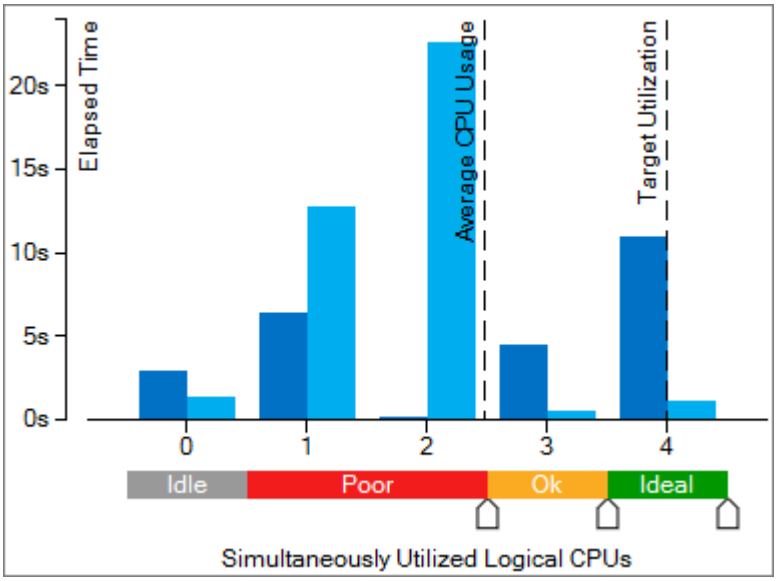
This section displays the performance difference between two selected results for the most active functions in your application.

Function	Module	CPU Time, sorted by abs. difference
grid_intersect	libjni-engine.so	22.949s - 14.474s = 8.475s
sphere_intersect	libjni-engine.so	21.090s - 14.346s = 6.744s
S32_opaque_D32_nofilter_DX	libskia.so	2.437s - 5.206s = -2.768s
func@0x1605ab	libskia.so	[Unknown] - 2.219s = -2.219s
SkSetPixelRect32_x86	libskia.so	1.220s - 2.781s = -1.561s
[Others]	N/A*	14.383s - 21.475s = -7.092s

*N/A is applied to non-summable metrics.

In this example, the first analysis result was collected when the application was using multiple threads and the preset 25 second elapsed time allowed the application to fully render the image twice. Allowing the image to render twice resulted in a higher CPU time for the functions. In the second analysis result the image only rendered once even though the analysis result had a longer elapsed time. When analyzing your own application, you would typically begin by making code changes to improve the most CPU-intensive functions and therefore would hope that the difference between results would show performance improvement.

In this example, the **CPU Usage Histogram** shows that the initial analysis (darker blue) spent the most time running with three or four logical CPUs and the second analysis that was run on a single thread (lighter blue) spent the most time running on two logical CPUs. Switch to the **Bottom-up** window to view details about the CPU usage of each function.



When making changes to your own application, opening a comparison of two results can help identify areas that had performance differences. Evaluate whether your change impacted the highest hotspot and ensure that it did not cause a performance degradation elsewhere in the application. From there, you can begin to reduce other hotspots until your application is fully optimized.

Next Step

Summary

Summary

3

You have completed the Finding Hotspots tutorial. Here are some important things to remember when using the Intel® VTune™ Amplifier for Systems to analyze your Android* application code for hotspots:

Step	Tutorial Recap	Key Tutorial Take-aways
1. Set up the Android* System for Analysis	You enabled Debug connections to enable `adb` access. You also saw an example of how to setup WiFi `adb` access in the case that USB access is not possible.	• You must have `adb` installed on your host to access your Android device and to run Basic Hotspots Analysis. • Root mode access is only required for hardware event-based sampling and Java* analysis.
2. Build the application and Create a New Project	You installed the `tachyon_vtune_amp_xe.tar.gz` package on your Android device. You launched the GUI and configured the project on your Linux* host.	• Use a fully optimized production build with symbols enabled. • Be sure to complete Binary/Symbol Search and Source Search paths to see function names and drill down to source.
3. Run Basic Hotspots Analysis	You ran the analysis from the host system on the target device.	• You can launch analysis from the GUI or from the command line. • Analysis can be run while the device is connected or disconnected via USB.
4. Interpret Result Data	You identified the function `grid_intersect` as the function using the most CPU time.	• Use **Basic Hotspots** analysis to examine the CPU usage and thread utilization for an application.
5. Compare Results	You viewed the results of a single-threaded version of the `tachyon` application to see how **Basic Hotspots** analysis results for single-threaded code compares to the original results you obtained.	• Compare subsequent **Basic Hotspots** analysis results to make sure changes to your code improve its performance.

Key Terms

4


The following list of Key Terms and definitions is included for easy reference while using this tutorial.

baseline : A performance metric used as a basis for comparison of the application versions before and after optimization. Baseline should be measurable and reproducible.

CPU time : The amount of time a thread spends executing on a logical processor. For multiple threads, the CPU time of the threads is summed. The application CPU time is the sum of the CPU time of all the threads that run the application.

CPU usage: A performance metric when the VTune Amplifier identifies a processor utilization scale, calculates the target CPU usage, and defines default utilization ranges depending on the number of processor cores.

Utilization Type	Default color	Description
Idle		All CPUs are waiting - no threads are running.
Poor		Poor usage. By default, poor usage is when the number of simultaneously running CPUs is less than or equal to 50% of the target CPU usage.
OK		Acceptable (OK) usage. By default, OK usage is when the number of simultaneously running CPUs is between 51-85% of the target CPU usage.
Ideal		Ideal usage. By default, Ideal usage is when the number of simultaneously running CPUs is between 86-100% of the target CPU usage.

Elapsed time :The total time your target ran, calculated as follows: **Wall clock time at end of application – Wall clock time at start of application**.

finalization : A process during which the Intel® VTune™ Amplifier converts the collected data to a database, resolves symbol information, and pre-computes data to make further analysis more efficient and responsive.

host system: The Linux* system on which you have installed and are running the Intel VTune Performance Analyzer for Systems.

hotspot: A section of code that took a long time to execute. Some hotspots may indicate bottlenecks and can be removed, while other hotspots inevitably take a long time to execute due to their nature.

Basic Hotspots analysis: An analysis type used to understand the application flow and identify hotspots. VTune Amplifier creates a list of functions in your application ordered by the amount of time spent in a function. It also detects the call stacks for each of these functions so you can see how the hot functions are called. VTune Amplifier uses a low overhead (about 5%) user-mode sampling and tracing collection that gets you the information you need without slowing down the application execution significantly.

target : A *target* is an executable file you analyze using the Intel® VTune™ Amplifier.

target system :The Android* device on which your application and the `tachyon` sample code run. The target system is connection to the host system by means of a USB connection, WiFi, or TCP/IP.

viewpoint : A preset result tab configuration that filters out the data collected during a performance analysis and enables you to focus on specific performance problems. When you select a viewpoint, you select a set of performance metrics the VTune Amplifier shows in the windows/panes of the result tab. To select the required viewpoint, click the **(change)** link and use the drop-down menu at the top of the result tab.

Index