Tutorial: Finding Hotspots with Intel® VTune™ Profiler - Linux*

Contents

Notices and Disclaimers

Use Case and Prerequisites

1

You can use the Intel® VTune™Profiler to identify and analyze hotspot functions and microarchitecture usage issues in your serial or parallel application by performing a series of steps in a workflow. This tutorial guides you through these workflow steps while using a sample matrix multiplication application named `matrix`.

Use Case Workflow

1. Create a project and find hotspots.
   a. Run Hotspots analysis
   b. Interpret Hotspots Result Data
2. Find hardware usage bottlenecks.
   a. Run Microarchitecture Exploration analysis
   b. Interpret Microarchitecture Exploration Analysis Result Data
3. Eliminate memory access performance problems.

   Resolve the Performance Issue
4. Check your work.

   Compare with Previous Result

Prerequisites

You need the following tools to try the tutorial steps yourself using the pre-built `matrix` sample application:

- Intel® VTune™Profiler 2020 or later (standalone or packaged with Intel® Parallel Studio XE or Intel® System Studio)

  Tip
  If you do not already have access to the VTune Profiler, you can download a copy from the Intel VTune Profiler page.

- Supported compiler (see the VTune Profiler Release Notes for more information); optionally, Intel® C++ Compiler

Next Step

Run Hotspots Analysis

Run Hotspots Analysis

2

In this part of the tutorial, you open the Matrix sample project and run the Hotspots analysis with user-mode sampling to identify the hotspots that took too much time to execute.

Open Matrix Sample Project

To analyze your target in the VTune Profiler, you need to create or open a project, which is a container for an analysis target configuration and data collection results. VTune Profiler provides a sample project pre-configured to work with the pre-built `matrix` sample application.

1. If working in a bash shell, set the `EDITOR` or `VISUAL` environment variable to associate your source files with the code editor (like emacs, vi, vim, gedit, and so on). For example:

   ```
   $ export EDITOR=gedit
   ```
2. Launch Intel VTune Profiler GUI.

   a. Run the shell script to set the PATH environment variable that specifies locations of the product graphical user interface utility and command line utility:

      - for csh/tcsh users: `source <install_dir>/env/vars.csh`
      - for bash users: `source <install_dir>/env/vars.sh`

      where default installation directory is:

      - for root users: `/opt/intel/vtune_profiler_version`
      - for non-root users: `$HOME/intel/vtune_profiler_version`

   b. Type `vtune-gui` to launch the product graphical interface.

   The VTune Profiler **Welcome** screen is displayed after the product launches. The Matrix project and r000hs result file may already be open in the **Project Navigator**. If so, no further action is required.

   If the Matrix project is not available from the **Project Navigator**, click the menu button and select **Open > Project...** to open an existing project.
3. Browse to the Matrix project on your local system and click **Open**. By default, this is located in the `/home/<user>/intel/vtune/projects/sample (matrix)` directory.

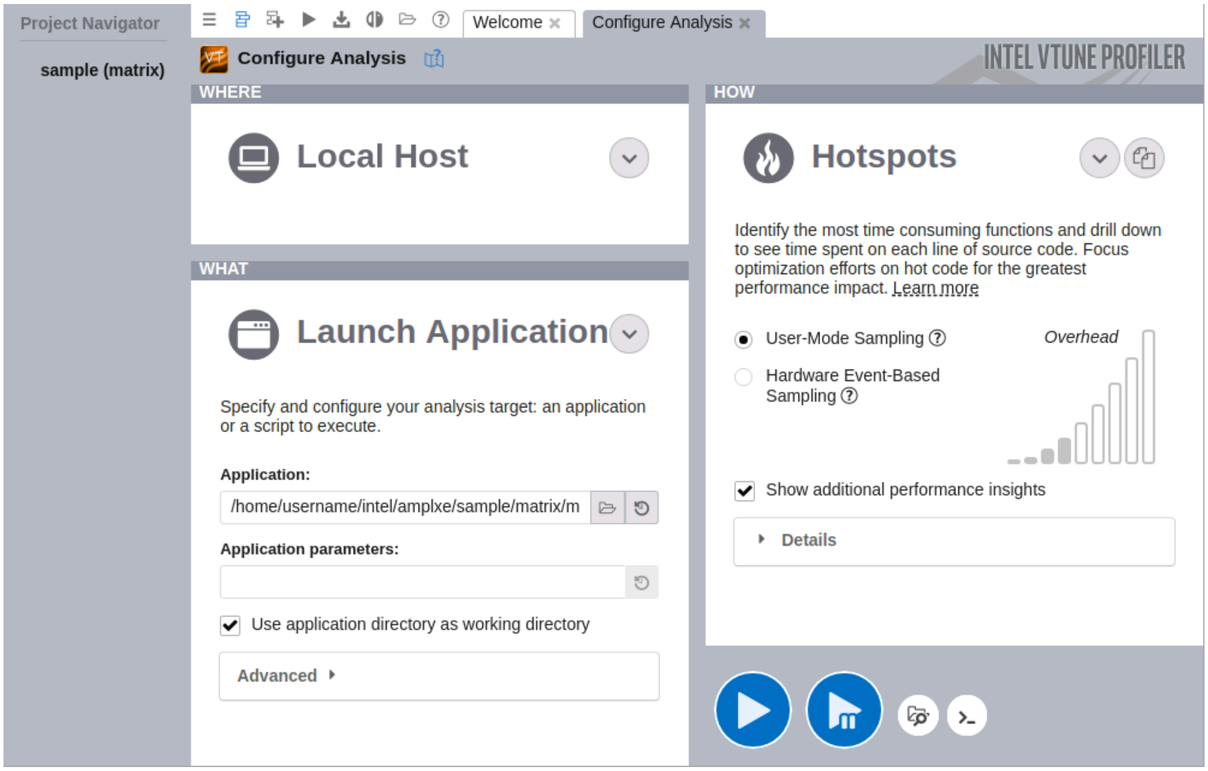
   VTune Profiler opens the Matrix project in the **Project Navigator**.

> **Tip**
> This tutorial uses the pre-built `matrix` sample application. When you use your own application for analysis, be sure to build the application in the Release mode with full optimizations and establish a performance baseline before running a full analysis. For more information, see the VTune Profiler User Guide.

Run Hotspots Analysis

1. Click **Configure Analysis** button to begin a new analysis.

   The default analysis is pre-configured for the entry-level **Hotspots** analysis to profile the `matrix` application on the local system.



2. Click the **Start** button to run the analysis.

VTune Profiler launches the `matrix` application that calculates matrix multiplication before exiting. VTune Profiler finalizes the collected results and opens the **Hotspots by CPU Utilization** 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
This tutorial explains how to run an analysis from the VTune Profiler graphical user interface (GUI). You can also use the VTune Profiler command-line interface (`vtune` command) to run an analysis. A simple way to get the appropriate command syntax is by clicking the **Command Line** button at the bottom of the window. For more details, check the *Intel VTune Profiler Command Line Interface* section of the VTune Profiler User Guide.

Next Step

Interpret Result Data

Interpret Hotspots Result Data 3

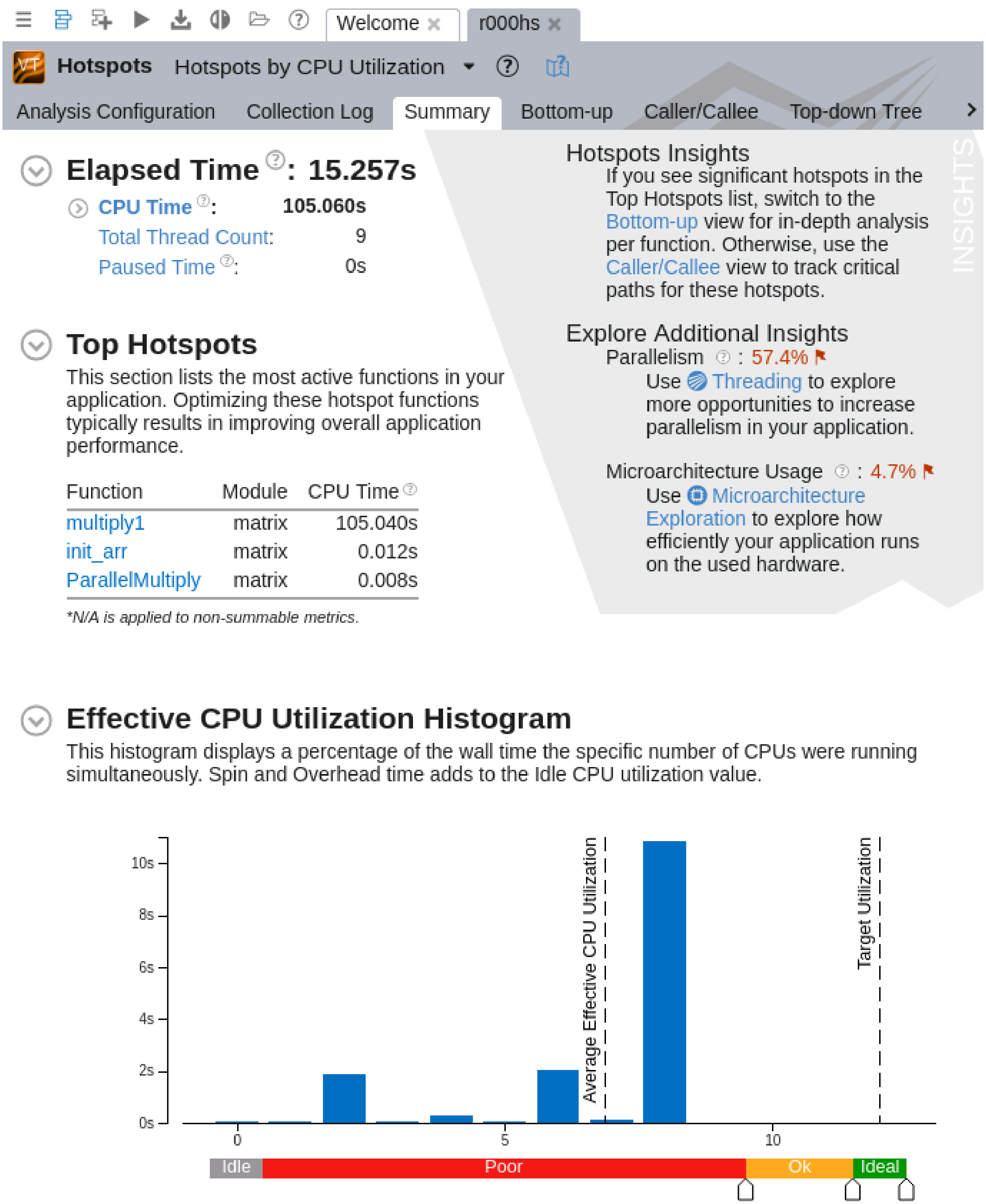
When the sample application exits, the Intel® VTune™Profiler finalizes the results and opens the **Hotspots by CPU Utilization** viewpoint where each window or pane is configured to display code regions that consumed a lot of CPU time. To interpret the data on the sample code performance, do the following:

1. Understand the basic performance metrics provided by the Hotspots analysis.
2. Analyze the most time-consuming functions and CPU utilization.
3. Analyze performance per thread.
4. View the source code for the most time-consuming function.

> **NOTE**
> The screenshots and execution time data provided in this tutorial are created on a system with 6 cores and 12 threads. Your data may vary depending on the number and type of CPU cores on your system.

Understand the Hotspots Metrics

Start your investigation with the **Summary** window. To learn more about a particular metric, hover over the question mark icons to read the pop-up help and better understand what each performance metric means.


Welcome
r000hs
Hotspots
Hotspots by CPU Utilization
Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Elapsed Time: 15.257s
CPU Time: 105.060s
Total Thread Count: 9
Paused Time: 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
multiply1 matrix 105.040s
init_arr matrix 0.012s
ParallelMultiply matrix 0.008s
*N/A is applied to non-summable metrics.
Hotspots Insights
If you see significant hotspots in the Top Hotspots list, switch to the Bottom-up view for in-depth analysis per function. Otherwise, use the Caller/Callee view to track critical paths for these hotspots.
Explore Additional Insights
Parallelism: 57.4%
Use Threading to explore more opportunities to increase parallelism in your application.
Microarchitecture Usage: 4.7%
Use Microarchitecture Exploration to explore how efficiently your application runs on the used hardware.
INSIGHTS
Effective CPU Utilization Histogram
This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.
10s
8s
6s
4s
2s
0s
Average Effective CPU Utilization
Target Utilization
0
5
10
Idle
Poor
Ok
Ideal

Note that **CPU Time** for the sample application is equal to about 105 seconds. It is the sum of CPU time for all application threads. **Total Thread Count** is 9, so the sample application is multi-threaded. Note that VTune Profiler flagged an issue with threading in the **Insights** pane. You can use the Threading analysis to explore threading optimization opportunities.

The **Top Hotspots** section of the **Summary** window provides data on the most time-consuming functions (hotspot functions) sorted by CPU time spent on their execution. For the sample application, the `multiply1` function, which took 105.040 seconds to execute, shows up at the top of the list as the hottest function.

The **Effective CPU Utilization Histogram** lower on the **Summary** window represents the Elapsed Time and usage level for the available logical processors and provides a graphical look at how many logical processors were used during the application execution. Ideally, the highest bar of your chart should match the Target Utilization level. The `matrix` application ran mostly on all logical CPUs.

The **Insights** pane highlights on the most critical issues with the application and provides recommendations based on the collected results. In this case, it recommends reviewing the per-function performance statistics on the **Bottom-up** pane for the identified hotspots, such as the `multiply1` function.

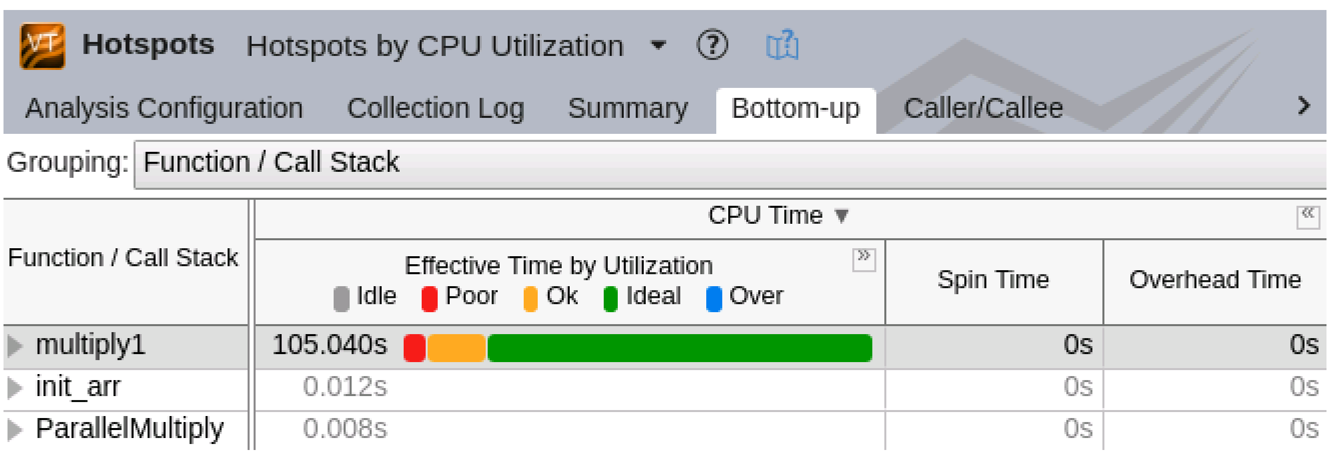
As an additional insight, VTune Profiler flagged an issue with the Microarchitecture Usage. The metric value is below the threshold, which indicates low code efficiency on this hardware platform. Possible causes of low performance can include memory stalls, instruction starvation, branch misprediction, or long latency instructions. After analyzing or resolving the algorithm issues for hotspot functions, run the **Microarchitecture Exploration** analysis type to identify the root cause of the Microarchitecture Usage issues.

Analyze the Most Time-Consuming Functions

To view per-function hotspots analysis, switch to the **Bottom-up** tab. By default, the data in the grid is sorted by Function. You may change the grouping level using the **Grouping** drop-down menu at the top of the grid.

Analyze the **CPU Time** column values. Functions that took most CPU time to execute are listed on top.

The `multiply1` function took the maximum time to execute, 105.040 seconds, and had the longest poor CPU utilization (red bars). This means that the processor cores were underutilized during a portion of the time spent executing this function.



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

Grouping: Function / Call Stack

Function / Call Stack	CPU Time ▼					
	Effective Time by Utilization					Spin Time
	Idle	Poor	Ok	Ideal	Over	
multiply1	0s	4.924s	13.247s	86.869s	0s	0s
init_arr	0s	0.012s	0s	0s	0s	0s
ParallelMultiply	0s	0.008s	0s	0s	0s	0s

Select the `multiply1` function in the grid and explore the data provided in the **Call Stack** pane on the right. The **Call Stack** pane displays full stack data for each hotspot function, which enables 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> - <file>:<line number>, where the line number corresponds to the line calling the next function in the stack.

CPU Time ▼

Viewing 1 of 1 selected stack(s)

100.0% (105.040s of 105.040s)

matrix!multiply1 - multiply.c
matrix!ThreadFunction+0x5a - thrmodel.c:48
libpthread.so.0!start_thread+0xda - pthread_cr…
libc.so.6!clone+0x3e - clone.S:95

For the sample application, the hottest function `multiply1` is called at line 48 of the `ThreadFunction` function in the `thrmodel.c` file.

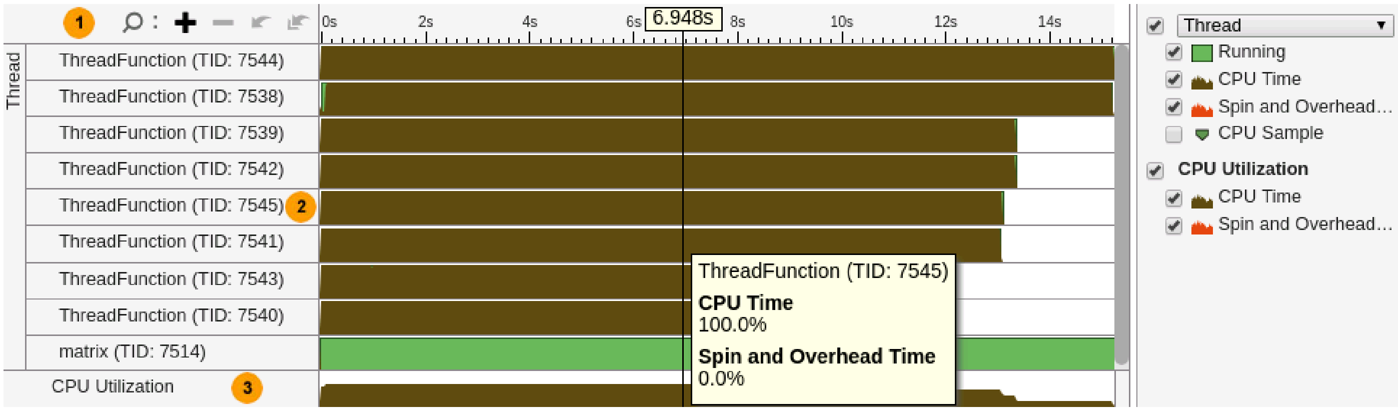
Analyze Performance per Thread

If you change the grouping level in the **Bottom-up** pane from **Function/Call Stack** to **Thread/Function/Call Stack**, you see that the `multiply1` function belongs to the `ThreadFunction` thread.

Grouping: Thread / Function / Call Stack

Thread / Function / Call Stack	CPU Time ▼					
	Effective Time by Utilization					Spin Time
	Idle	Poor	Ok	Ideal	Over	
ThreadFunction (TID: 7544)	0s	2.180s	2.180s	10.860s	0s	0s
ThreadFunction (TID: 7538)	0s	2.138s	2.113s	10.859s	0s	0s
ThreadFunction (TID: 7539)	0s	0.288s	2.182s	10.860s	0s	0s
ThreadFunction (TID: 7542)	0s	0.288s	2.182s	10.860s	0s	0s
ThreadFunction (TID: 7545)	0s	0.030s	2.180s	10.860s	0s	0s
ThreadFunction (TID: 7541)	0s	0s	2.180s	10.860s	0s	0s
ThreadFunction (TID: 7543)	0s	0s	0.140s	10.850s	0s	0s
ThreadFunction (TID: 7540)	0s	0s	0.090s	10.860s	0s	0s
matrix (TID: 7514)	0s	0.020s	0s	0s	0s	0s

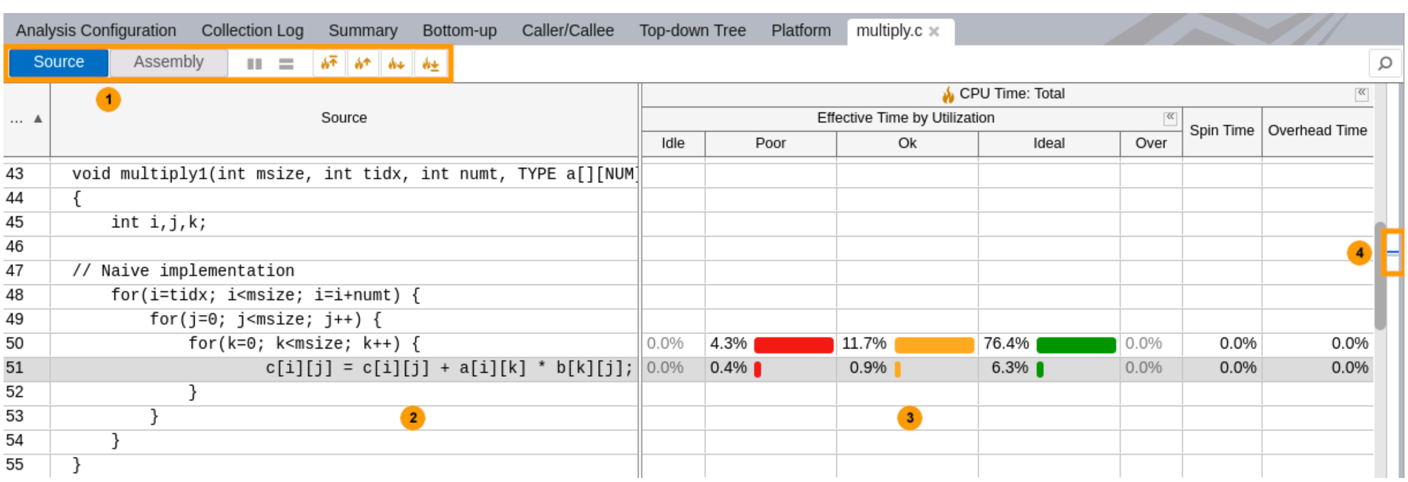
To get detailed information on the thread performance, explore the **Timeline** pane.



1	**Timeline** area. When you hover over the graph element, the timeline tooltip displays the time passed since the application has been launched.
2	**Threads** area that 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 show the time threads are active.
3	**CPU Utilization** area that 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. VTune Profiler calculates the overall **CPU Utilization** metric as the sum of CPU time per each thread of the **Threads** area. Maximum **CPU Utilization** value is equal to `[number of processor cores] x 100%`.

View Source Code

Double-click the `multiply1` function on the **Bottom-up** pane grid to open the **Source** window and analyze the source code.



The table below explains some of the features available in the **Source** window.

1	Source window toolbar. Use the hotspot navigation buttons to switch between most performance-critical code lines. Use the **Source**/**Assembly** buttons to toggle the **Source**/**Assembly** panes (if both of them are available) on/off.

2	**Source** pane displaying the source code of the application if the function symbol information is available. The hottest code line in the function is highlighted. The source code in the **Source** pane is not editable. If the function symbol information is not available, the **Assembly** pane opens displaying assembler instructions for the selected hotspot function. To enable the **Source** pane, make sure to build the target properly.
3	Processor time attributed to a particular code line. If the hotspot is a system function, its time, by default, is attributed to the user function that called this system function. Drag-and-drop the columns to organize the view for your convenience. VTune Profiler remembers your settings and restores them each time you open the viewpoint.
4	Heat map markers to quickly identify performance-critical code lines (hotspots). The bright blue markers indicate hot lines for the function you selected for analysis. Light blue markers indicate hot lines for other functions. Scroll to a marker to locate the hot code line it identifies.

By default, when you double-click the hotspot in the **Bottom-up** pane, VTune Profiler opens the source file positioned at the most time-consuming code line of this function. For the `multiply1` function, this is line 51, which operates over three arrays: a, b, and c.

> **NOTE**
> Depending on the sample code version, your source line numbers may slightly differ from the numbers provided in this tutorial.

According to the **Insights** data on the **Summary** pane, the `matrix` application may use microarchitecture resources ineffectively. To learn more about possible issues, run the **Microarchitecture Exploration** analysis and identify the affected part of the core pipeline.

Next Step

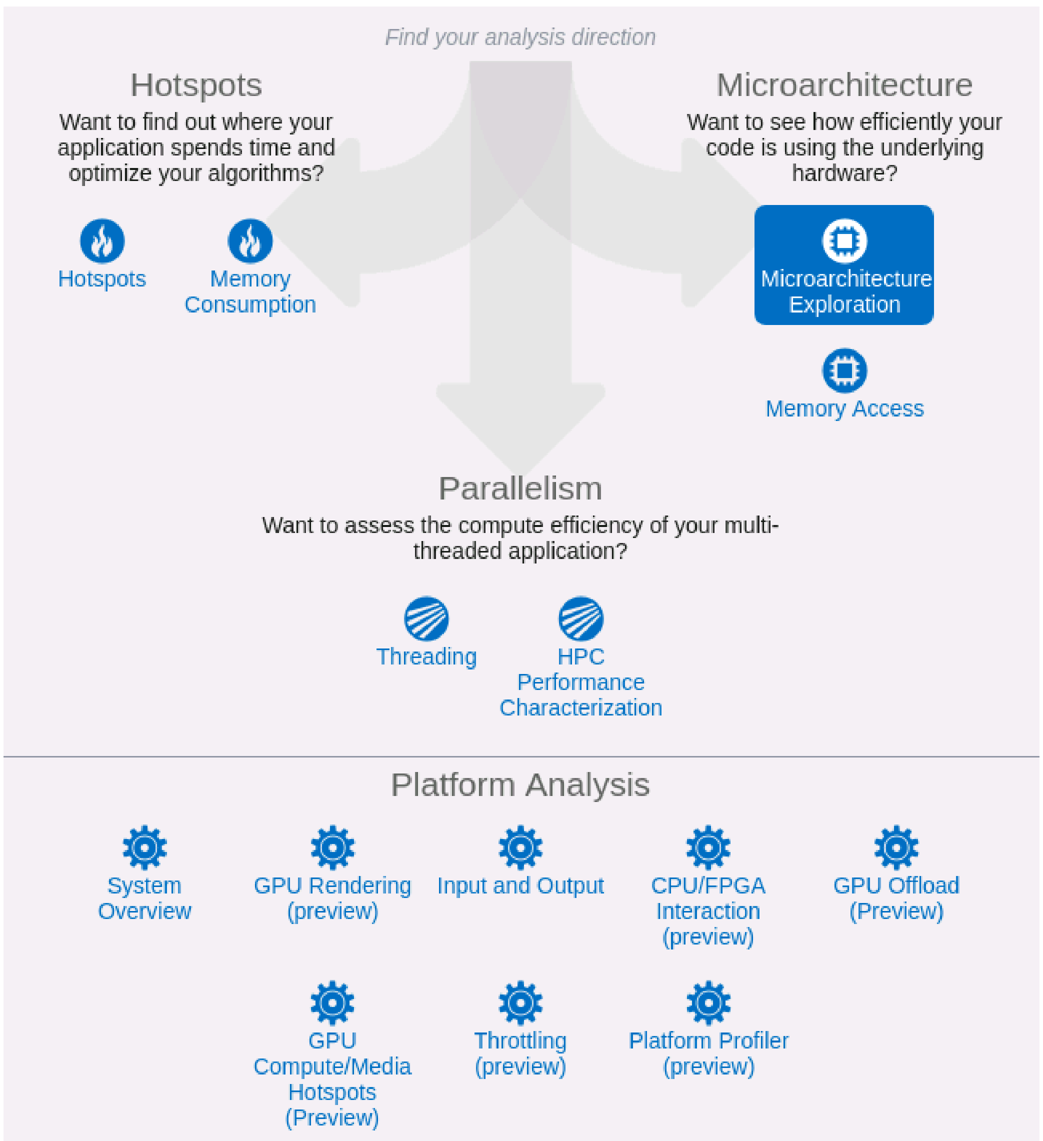
Run Microarchitecture Exploration Analysis

Run Microarchitecture Exploration Analysis 4

After running the Hotspots analysis on the `matrix` sample, microarchitecture usage inefficiency was highlighted as the top issue in the **Insights** pane.

Once you have determined hotspots in your code, perform Microarchitecture Exploration analysis to understand how efficiently your code is passing through the core pipeline. During Microarchitecture Exploration analysis, Intel® VTune™Profiler collects a selected list of hardware events for analyzing a typical user application. It calculates a set of predefined hardware metrics and facilitates identifying hardware-level performance problems.

1. Click **Configure Analysis** to begin a new analysis.
2. In the **HOW** pane, click the Browse button and select the **Microarchitecture Exploration** analysis type.



3. Click the **Start** button to run the analysis.

VTune Profiler launches the `matrix` application, runs the analysis, and finalizes the collected results. The results are shown in the **Microarchitecture Exploration** viewpoint.

Next Step

Interpret Microarchitecture Exploration Analysis Result Data

Interpret Microarchitecture Exploration Analysis Result Data

5

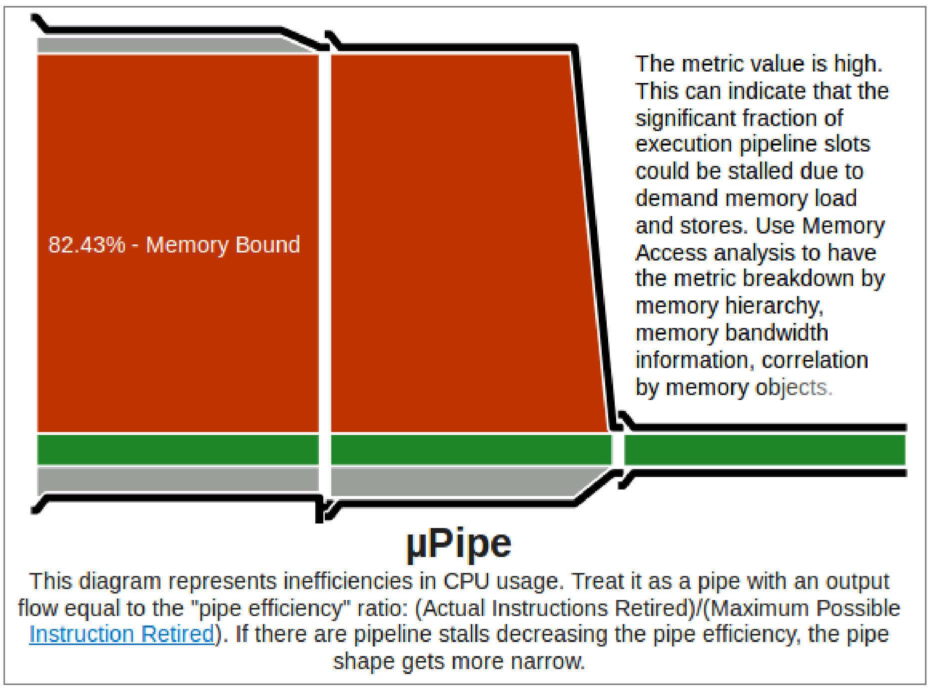
When the sample application exits, the Intel® VTune™Profiler finalizes the results and opens the **Microarchitecture Exploration** viewpoint, which provides a high-level performance overview of the interaction between the application and the available hardware.

To interpret the data on the sample code performance, do the following:

1. Understand the Event-based Metrics
2. Identify Hardware Usage Bottlenecks
3. Analyze Code

Understand the Event-based Metrics

Start with the **Summary** pane for an overview of application performance.



The μPipe diagram provides a graphical representation of CPU microarchitecture metrics showing inefficiencies in hardware usage. Treat the diagram as a pipe with an output flow equal to the ratio: **Actual Instructions Retired/Possible Maximum Instruction Retired** (pipe efficiency). The μPipe is based on CPU pipeline slots that represent hardware resources needed to process one micro-operation. Usually there are several pipeline slots available on each cycle (pipeline width). If a pipeline slot does not retire, this is considered a stall and the μPipe diagram represents this as an obstacle making the pipe narrow.

See the Microarchitecture Pipe page of the online User Guide for a more detailed explanation of the μPipe.

In this case, the **Memory Bound** metric is high, so only a small fraction (approximately 3%) of pipeline slots are being retired. Hover over each section for a description and percentage of the total pipeline or refer to the metrics on the left.

The hierarchy of event-based metrics in the Microarchitecture Exploration viewpoint depends on your hardware architecture. Each metric is an event ratio defined by Intel architects and has its own predefined threshold. VTune Profiler analyzes a ratio value for each aggregated program unit (for example, function). When this value exceeds the threshold, it signals a potential performance problem.

Elapsed Time: 20.438s

Clockticks:	515,136,600,000	
Instructions Retired:	69,237,300,000	
CPI Rate:	7.440	
MUX Reliability:	0.997	
Filled Pipeline Slots:		
Retiring:	3.4%	of Pipeline Slots
Bad Speculation:	0.1%	of Pipeline Slots
Unfilled Pipeline Slots (Stalls):		
Back-End Bound:	96.3%	of Pipeline Slots
Memory Latency:		
LLC Load Misses Serviced By Remote DRAM:	0.1%	of Clockticks
LLC Miss:	20.6%	of Clockticks
LLC Hit:	11.0%	of Clockticks
DTLB Overhead:	55.7%	of Clockticks
Contested Accesses:	0.0%	of Clockticks
Data Sharing:	3.3%	of Clockticks
Memory Replacements:		
L1D Replacement Percentage:	100.0%	
L2 Replacement Percentage:	100.0%	
LLC Replacement Percentage:	100.0%	
Memory Reissues:		
Loads Blocked by Store Forwarding:	0.0%	of Clockticks
Split Loads:	0.0%	of Clockticks
Split Stores:	0.3%	of Clockticks
4K Aliasing:	0.2%	of Clockticks
Divider:	0.0%	of Clockticks
Flags Merge Stalls:	0.0%	of Clockticks
Slow LEA Stalls:	0.0%	of Clockticks
Front-End Bound:	0.2%	of Pipeline Slots
Average CPU Frequency:	3.6 GHz	
Total Thread Count:	9	
Paused Time:	0s	

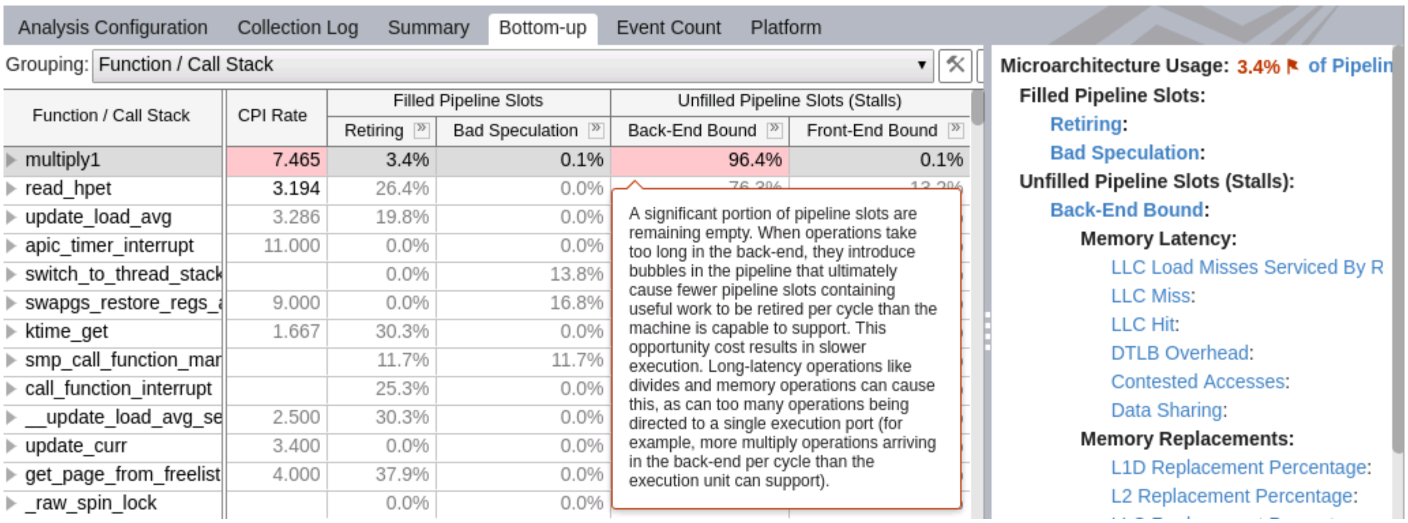
The **Elapsed Time** section shows metrics related to hardware event ratios for your hardware. Hover over the flagged metrics to get a description of the issues, possible causes, and suggestions for resolving the issue. This result shows issues with both **CPI Rate** (Clockticks per Instructions Retired rate) and **Back-End Bound**. Both issues were identified as possible causes for slow execution by the original **Hotspots** analysis.

In the expanded **Back-End Bound** section, there are issues with the application being **Memory Bound**, which matches the µPipe diagram. The **Bottom-up** pane can help identify the program units responsible for the memory issues.

Identify Hardware Usage Bottlenecks

Switch to the **Bottom-up** pane to see how each program unit performs against the event-based metrics. Each row represents a program unit and percentage of the CPU cycles used by this unit. Program units that take more than 5% of the CPU time are considered hotspots.

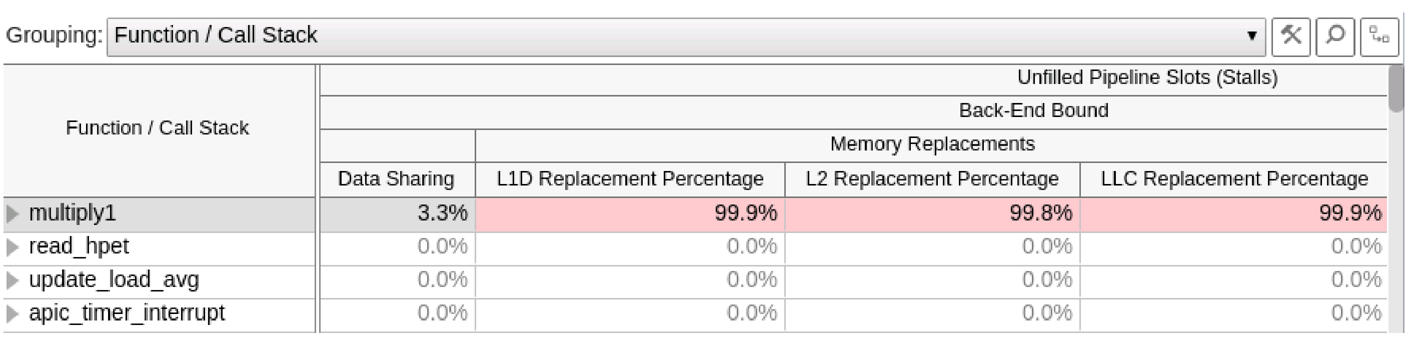
By default, the VTune Profiler sorts data in the descending order by CPU Time and provides the hotspots at the top of the list. The metric values for event ratios show up as numbers and/or bars.



As was identified when running the **Hotspots** analysis, the `multiply1` function is the most obvious hotspot in the `matrix` application. It has the highest event count (**Clockticks** and **Instructions Retired** events) and most of the hardware issues were also detected during the execution of this function.

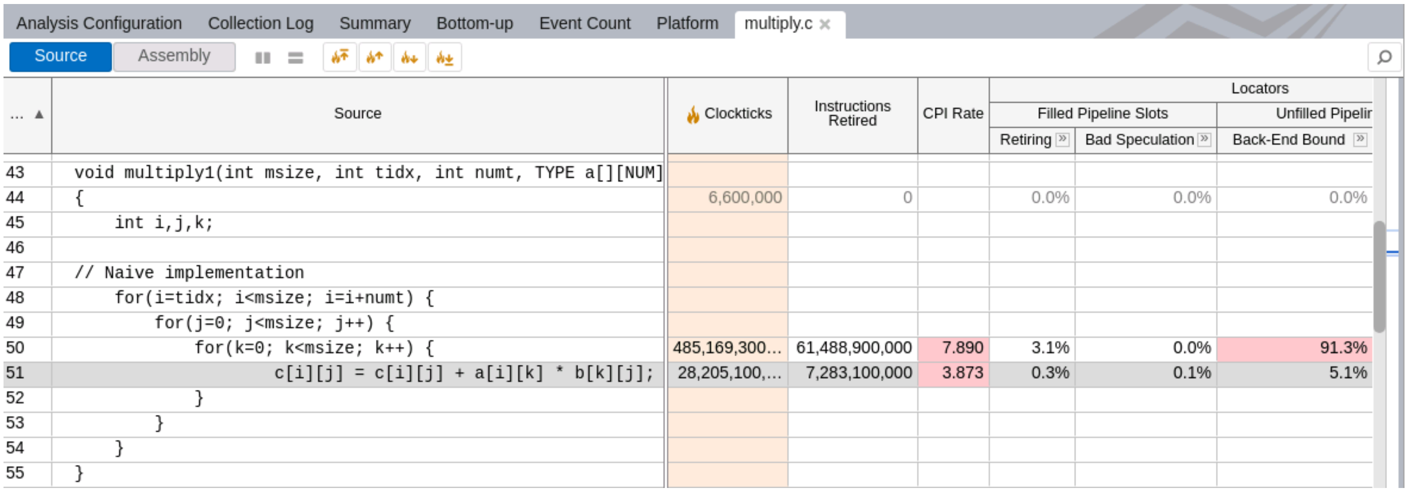
The **Back-End Bound** metric describes a portion of the pipeline where the out-of-order scheduler dispatches ready µOps into their respective execution units, and, once completed, these µOps get retired according to program order. Identify slots where no µOps are delivered due to a lack of required resources for accepting more µOps in the bad-end of the pipeline. Stalls due to data-cache misses or stalls due to the overloaded divider unit are examples of back-end bound issues.

Expand the **Back-End Bound** column to discover that the code is memory bound with the most percentage of stalls occurring on the main memory (DRAM). Hover over the highlighted cells to learn more about optimization opportunities.



Analyze Code

Double-click the `multiply1` function to open the **Source** window and analyze the source code.



When you drill-down from the grid to the source view, the VTune Profiler automatically highlights the code line that has the highest event count. In the **Source** pane for the `multiply1` function, you see that line 51 took the most of the Clockticks event samples during execution and was also highlighted as the top hotspot line in the Hotspots result. This code section multiplies matrices in the loop but ineffectively accesses the memory. Expand the **Back-End Bound** column to learn more. Focus on this section and try to reduce the memory issues.

Tip

For advanced users looking for a different way to identify and diagnose memory issues in your application, try running the **Memory Access** analysis type. An example of how to define which data structure induces inefficient memory access is available from the VTune Profiler Cookbook.

Next Step

Resolve Issue

Resolve Issue

6

In the **Source** window, the `multiply1` function was identified as a Memory Bound hotspot. To solve this issue, do the following:

> **NOTE**
> The proposed solution is one of the multiple ways to optimize memory access and is used for demonstration purposes only.

1. Open the `multiply.h` file from the sample code source files. You can find the source files in the following location: `/home/intel/vtune/sample/matrix/src`.

For this sample, the `multiply.h` file is used to define the functions used in the `multiply.c` file.

```
33
34 // Select which multiply kernel to use via the following macro so that the
35 // kernel being used can be reported when the test is run.
36 #define MULTIPLY multiply1
37
```

2. In line 36, replace the `multiply1` function name with the `multiply2` function.

This new function uses the loop interchange mechanism that optimizes the memory access in the code, which can be seen in the `multiply.c` file.

```

// Step 2: Loop interchange
    for(i=tidx; i<msize; i=i+numt) {
        for(k=0; k<msize; k++) {
            for(j=0; j<msize; j++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }
}

```

3. Save files and rebuild the project using the compiler of your choice.

For example, run Intel® C++ Compiler from the `/home/intel/vtune/sample/matrix/linux` directory using the following command: .

The matrix application is automatically built with the Intel Compiler (as `matrix.icc`) and stored in the source directory.

Next Step

Compare with Previous Result

Optimization Notice
optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Compare with Previous Result 7

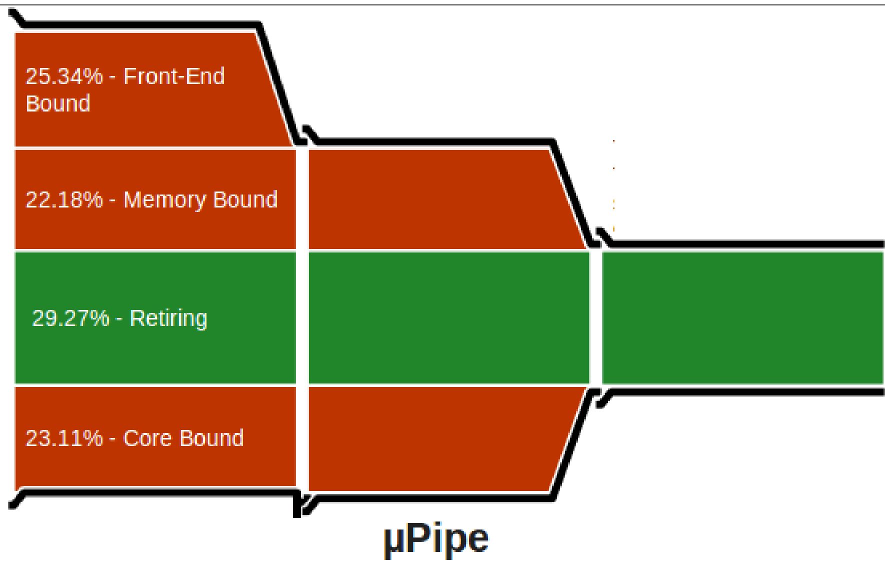
You optimized your code to apply a loop interchange mechanism. To understand whether you got rid of the memory bound issue and what kind of optimization you got per function, re-run the Microarchitecture Exploration analysis on the optimized code and compare results:

1. Collect results after optimization.
2. Compare results before and after optimization.
3. Identify the performance gain.

Analyze Results After Optimization

Run the Microarchitecture Exploration analysis on the modified code. VTune Profiler automatically opens the **Microarchitecture Exploration** viewpoint.

The optimized result shows an improvement in the μPipe diagram with about 30% of pipeline slots retiring compared to only 3% before optimization. The uPipe diagram also shows further opportunities for optimization of this application.

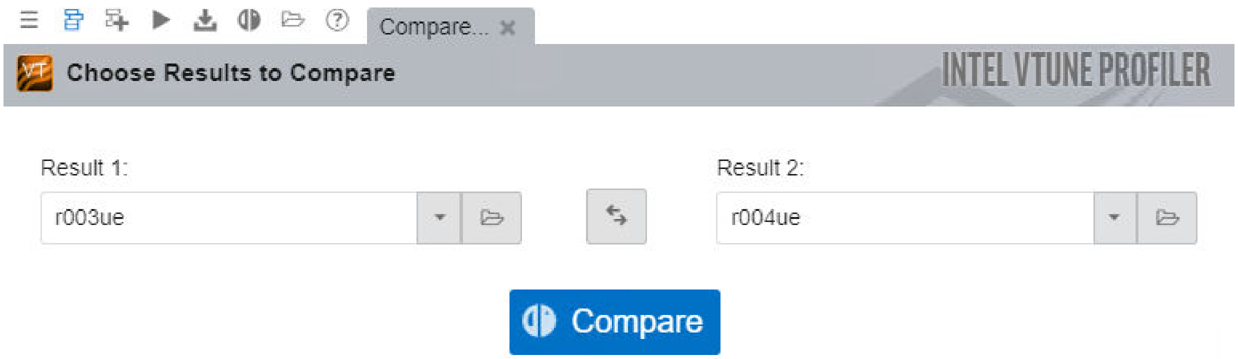


This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Compare Results Before and After Optimization

1. Close the new result file.
2. Select the result in the **Project Navigator**, right-click, and choose **Compare Results** from the context menu.

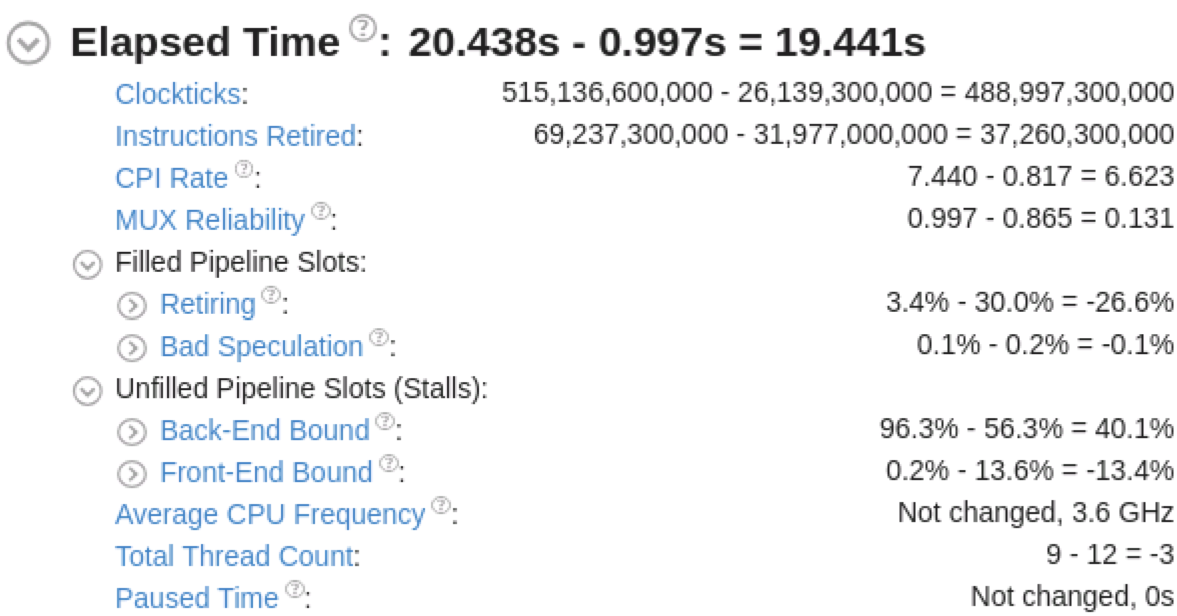
   The **Compare Results** window opens.
3. Specify the analysis results you want to compare and click the **Compare** button.



The **Summary** window opens, providing a high-level picture of performance improvements in the following format: `<result 1 value> - <result 2 value>`.

Identify the Performance Gain

In the **Summary** window, you see that the Elapsed Time shows 19.411 seconds of optimization for the whole application execution and an improvement from 3.4% of instructions retired to 30.0% of instructions retired. The Back-End Bound metric improved by 40.1%.



Switch to the **Bottom-up** window to compare the two results and see the differences per metrics side by side.

See Also

Summary

Summary

8

You have completed the Finding Hotspots tutorial. Here are some important things to remember when using the Intel® VTune™Profiler to analyze your code for hotspots and hardware issues:

Step	Tutorial Recap	Key Tutorial Take-aways
1. Find hotspots	You launched the Hotspots data collection that analyzes function calls and CPU time spent in each program unit of your application and identified the following hotspots: • Identified a function that took the most CPU time and could be a good candidate for algorithm tuning. • Identified the code section that took the most CPU time to execute.	• Start analyzing the performance of your application from the **Summary** window to explore the performance metrics for the whole application. Then, move to the **Bottom-up** window to analyze the performance per function. Focus on the hotspots - functions that took the most CPU time. By default, they are located at the top of the table. • Double-click the hotspot function in the **Bottom-up** pane or **Call Stack** pane to open its source code and identify the code line that took the most CPU time.
2. Discover hardware usage bottlenecks	You ran the Microarchitecture Exploration analysis that monitors how your application performs against a set of event-based hardware metrics as follows: • Analyzed the data provided in the **Microarchitecture Exploration** viewpoint, explored the event-based metrics, identified the areas where your sample application had hardware issues, and found the exact function with poor performance per metrics that could be a good candidate for further analysis. • Analyzed the code for the hotspot function identified in the **Bottom-up** window and located the hotspot line that generated a high number of CPU Clockticks.	See the **Details** section of the Microarchitecture Exploration configuration section to get the list of processor events used for this analysis type.
3. Resolve detected issues	You solved the memory access issue for the sample application by interchanging the loops and sped up the execution time. You also considered using the Intel C++ Compiler to enable instruction vectorization.	• Start analyzing the performance of your application from the **Summary** window to explore the event-based performance metrics for the whole application. Mouse over the help icons to read the metric descriptions. Use the Elapsed time value as your performance baseline. • Move to the **Bottom-up** window and analyze the performance per function. Analyze the hardware issues detected for the hotspot functions (functions with the highest Clockticks). Hardware issues

Step	Tutorial Recap	Key Tutorial Take-aways
		are highlighted in pink. Mouse over a highlighted value to read the issues description and see the threshold formula. • Double-click the hotspot function in the **Bottom-up** pane to open its source code and identify the code line that took the highest Clockticks event count. • Consider using Intel C++ Compiler to vectorize instructions. Explore the compiler documentation for more details.
4. Check your work	You ran Microarchitecture Exploration analysis on the optimized code and compared the results before and after optimization using the Compare mode of the VTune Profiler. Compare analysis results regularly to look for regressions and to track how incremental changes to the code affect its performance.	Perform regular regression testing by comparing analysis results before and after optimization. From GUI, click the **Compare Results** button on the VTune Profiler toolbar. From command line, use the `vtune` command.

Next step: Prepare your own application(s) for analysis. Then use the VTune Profiler to find and eliminate performance problems.

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

See Also

Tuning and configuration recipes in the VTune Profiler Cookbook

More tutorials with associated sample code