Tutorial: Analyze Common Performance Bottlenecks with Intel® VTune™ Profiler - Linux*

Contents

Notices and Disclaimers

Tutorial: Analyze Common Performance Bottlenecks with Intel® VTune™Profiler - C++ Sample Code (Linux* OS)

1

Discover how to use Intel® VTune™Profiler for Linux* OS to identify algorithm or hardware utilization issues that can cause your applications to spend large amounts of time performing tasks and underutilize available hardware resources.

About This Tutorial	This tutorial guides you through the steps required to analyze and optimize a sample `matrix` application that performs multiplication of large matrices. It introduces you to the main concepts of VTune Profiler and the iterative process of analyzing and optimizing an application. The tutorial was last updated for the Intel VTune Profiler 2021 product release.
Estimated Duration	20-30 minutes.
Learning Objectives	After you complete this tutorial, you should be able to: • Open the pre-configured `matrix` sample project in VTune Profiler. • Run the Performance Snapshot analysis to locate the main problem areas in the `matrix` sample application and identify next steps for optimization. • Run the Hotspots and Memory Access analyses to better understand the main bottleneck and determine next steps. • Navigate the source code from inside VTune Profiler to locate the lines of code with memory access bottlenecks. • Use the HPC Performance Characterization analysis to identify microarchitecture underutilization issues related to lack of proper vectorization. • Compare results before and after optimization.
More Resources	• Other Intel VTune Profiler tutorials (HTML, PDF) • Intel VTune Profiler Cookbook • Additional Intel VTune Profiler documentation • Intel Software Product Support Page

Start Here

Use Case and Prerequisites

2

You can use Intel® VTune™Profiler to identify and analyze performance bottlenecks 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`.

Prerequisites

This tutorial requires you to install several Intel software tools. You can download and use these tools for free.

- Intel® VTune™Profiler 2021 or later
- Intel® C++ Compiler Classic

Follow these links to download the components:

- Intel® C++ Compiler Classic
- Standalone Intel® VTune™Profiler

NOTE

- This tutorial uses the Intel® C++ Compiler Classic to establish a common baseline for analysis and performance gain tracking. Your results and workflow may be different depending on the compiler you use.

Workflow

Follow these steps to identify and fix the most prominent performance issues in the sample `matrix` application.

1. Establish the application performance baseline
   a. Run Performance Snapshot analysis
   b. Interpret the Performance Snapshot analysis result
2. Identify main bottleneck in the matrix application
   a. Run Hotspots analysis and interpret data
   b. Run Memory Access analysis and interpret data
3. Eliminate the memory access bottleneck
   a. Fix memory issue and recompile application
4. Assess the performance improvement
   a. Run Performance Snapshot analysis and interpret result
5. Address the vectorization problem
   a. Recompile the application and run the HPC Performance Characterization analysis
   b. Recompile with different compiler options
6. Identify next steps
   a. Run and interpret the Microarchitecture Exploration analysis
7. Visualize the performance gain
   a. Compare results before and after optimization

Run Performance Snapshot Analysis 3

Performance Snapshot Analysis

For most software developers, the goal of performance optimization is to get the highest possible performance gain with the least possible investment of time and effort.

The Performance Snapshot analysis type helps you achieve this goal by highlighting the main problem areas in your application and providing metrics to estimate their severity. This enables you to focus on the most acute problems, solving which can yield the highest performance gain. This analysis type also offers other analysis types for deeper investigation into each performance problem.

Open Matrix Sample Project

The first step towards analyzing an application in VTune Profiler is to create a project. A project is a container that holds analysis target configuration and data collection results.

VTune Profiler provides a sample matrix project pre-configured to work with the pre-built matrix sample application.

Begin by opening the pre-configured matrix project:

1. Launch the VTune Profiler GUI:
   a. Run the following script to set the appropriate environment variables:
      - For bash users:

        ```
        source <install-dir>/env/vars.sh
        ```
      - For tch/tsch users:

        ```
        source <install-dir>/env/vars.csh
        ```

      For VTune Profiler, the default <install-dir> is:

      ```
      /opt/intel/oneapi/vtune/<version>
      ```
   b. Launch the vtune-gui binary located in the <install-dir>/bin64/ directory.

> **NOTE**
> You may need to run VTune Profiler as root to use certain analysis types.

2. The VTune Profiler welcome screen is displayed after the product launches.

   The sample (matrix) project should already be open in the **Project Navigator**. If so, no further action is required.

   If the sample (matrix) project is not available from the **Project Navigator**, open the project manually:

   a. Click the **Menu** button and select **Open > Project...** to open an existing project.
   b. Browse to the matrix project on your local machine and click **Open**.

      By default, it is located in this directory:

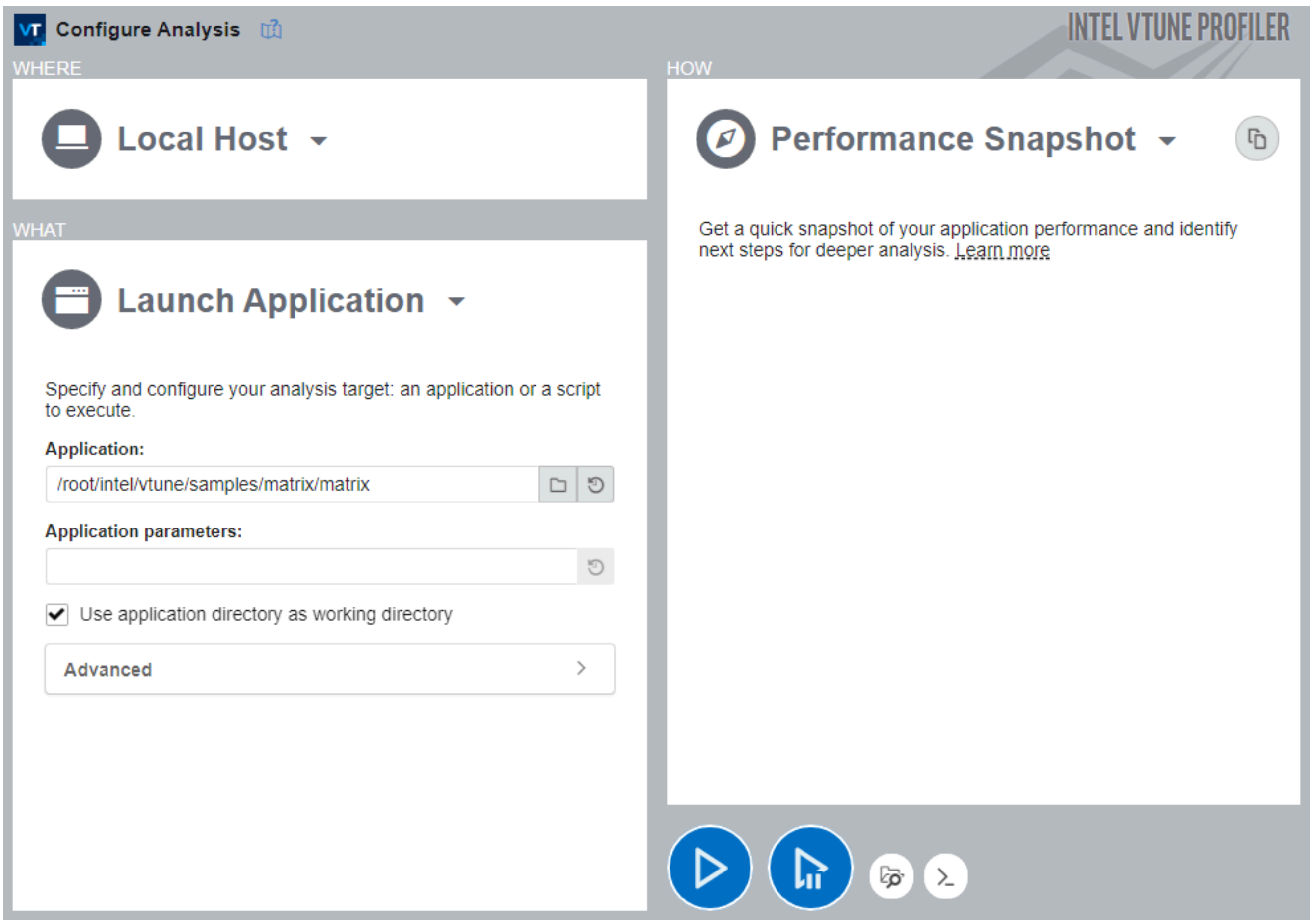
      ```
      $HOME/intel/vtune/projects/sample (matrix)
      ```

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

NOTE

- This tutorial uses the pre-built `matrix` sample application. When you analyze your own application, make sure to build it in the Release mode with full optimizations and establish a performance baseline before running a full analysis. For more information on preparing a Linux* target, see the Linux Targets section of the User Guide.
- To make sure that the performance data is accurate and repeatable, it is recommended to run the analysis while the system is running a minimal amount of other software.

Run Performance Snapshot Analysis



To start the Performance Snapshot analysis for the `matrix` sample application:

1. Click the **Configure Analysis** button to begin a new analysis. The default analysis is pre-configured for the **Performance Snapshot** analysis to profile the `matrix` application on the local system.
2. Click the **Start** button to run the analysis.

VTune Profiler the `matrix` application that calculates multiplication of large matrices before exiting. VTune Profiler finalizes the collected results and opens the **Summary** viewpoint of the Performance Snapshot analysis.

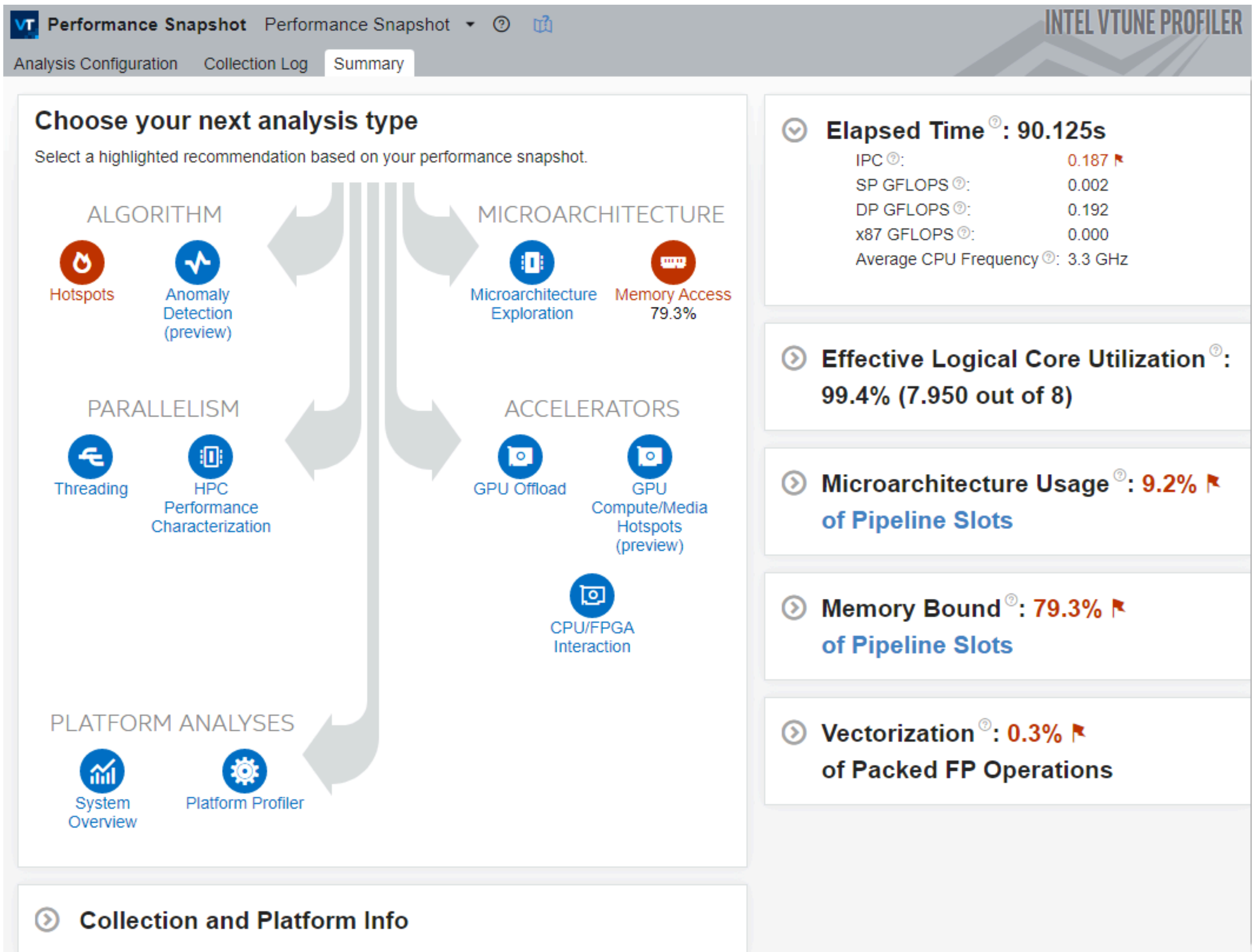
NOTE
This tutorial explains how to run an analysis from the VTune Profiler graphical user interface (GUI). You can also use the VTune Profiler command-line interface (`vtune` command) to run an analysis. A simple way to get the appropriate command syntax is by clicking the **Command Line** button at the bottom of the window. For more details, check the Command Line Interface chapter of the VTune Profiler User Guide.

Next step: Interpret Performance Snapshot Analysis.

Interpret Performance Snapshot Result Data 4

When the sample application exits, Intel® VTune™ Profiler finalizes the result and opens the **Summary** tab of the Performance Snapshot analysis result.

Understand the Performance Snapshot Summary Tab



The Performance Snapshot result **Summary** tab shows the following:

- **Analysis tree:** Performance Snapshot offers other analysis types that may be useful for a deeper investigation into the performance issues found in your application. Analysis types that are related to performance problems detected in your application are highlighted in red.

  You can estimate the severity of each problem by studying the metric values.

  Hover over an analysis type icon to understand how an analysis type is related to your performance problem.
- **Metrics Panes:** these panes show the high-level metrics that contribute most to estimating application performance. Problematic areas are highlighted in red. You can expand each pane to get more information on each problem area and to see the lower-level metrics that contributed to the verdict.

Hover over each metric to see the metric description.

- **Collection and Platform Info:** this pane shows the information about the system on which this particular result was collected. It is useful when opening results collected on a different hardware platform.

Identify Problem Areas

In this case, observe these main indicators that highlight the performance bottlenecks:

- The **Elapsed Time** for this application is very high.
- The **Memory Bound** metric is high, indicating a memory access problem. Due to this, Performance Snapshot highlights the **Memory Access** analysis as a potential starting point and indicates that this performance bottleneck is the most severe and contributes most to the total **Elapsed Time**.
- The **IPC (Instructions per Cycle)** metric value is very low for a modern superscalar processor, indicating that the processor is stalled for most of the time.
- The Performance Snapshot analysis highlights the **Hotspots** analysis as a good starting point. In general, the **Hotspots** analysis is a good candidate for a first in-depth analysis. It highlights hotspots, or areas of code that contributed most to the elapsed time.

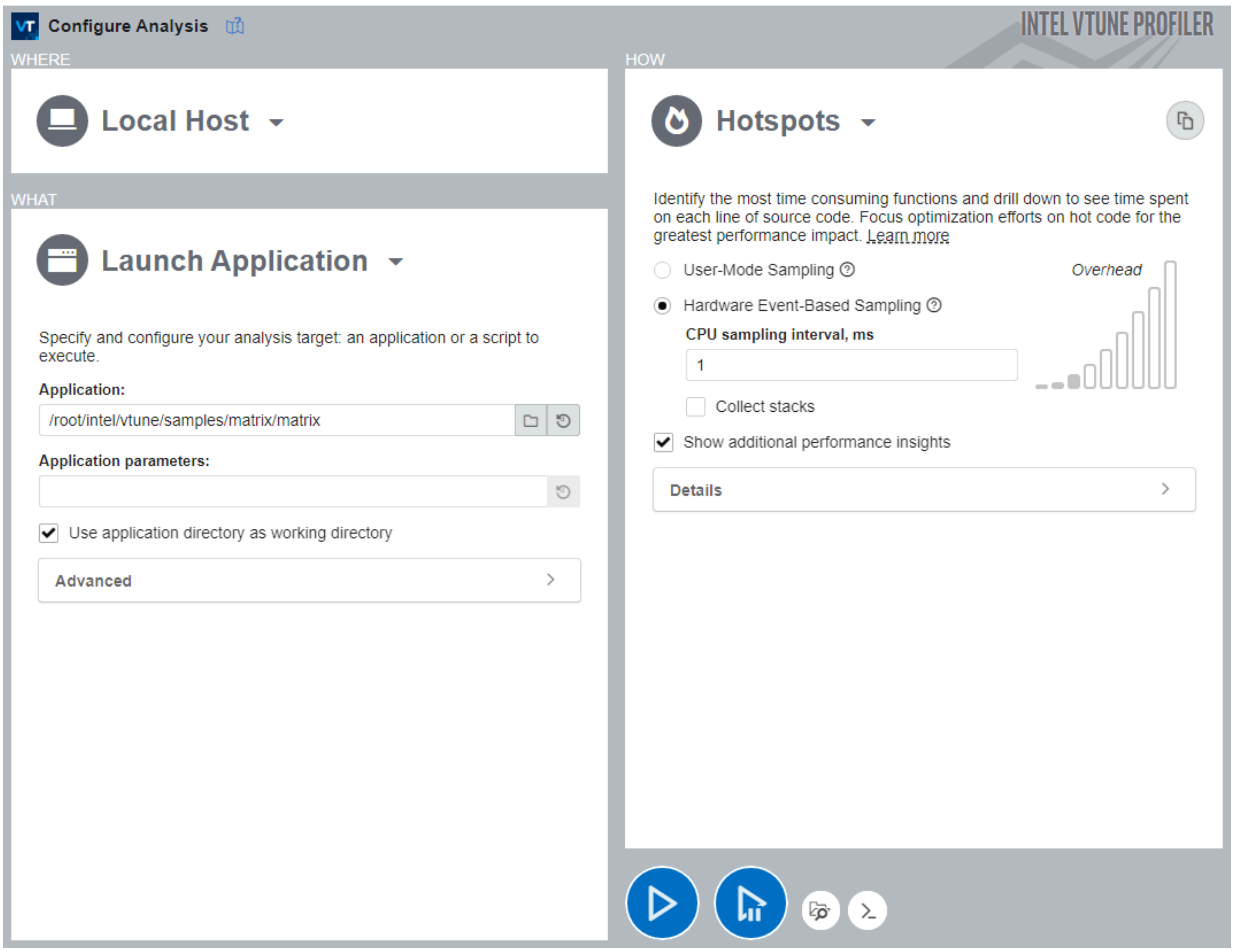
Start with the **Hotspots** analysis to see which area of code in the `matrix` application contributes most to the performance problem.

Next step: Run and Interpret Hotspots Analysis.

Run and Interpret Hotspots Analysis 5

In this part of the tutorial, you run the Hotspots analysis to locate hotspots, or sections of code that contribute most to the total elapsed time of the application.

Run Hotspots Analysis



To run the Hotspots analysis from the Performance Snapshot **Summary** window:

1. Click the **Hotspots** icon in the **Analysis tree**.

   The **Configure Analysis** window opens.
2. In the **WHERE** pane, select **Local Host**.
3. If you're using the pre-provided `sample (matrix)` project, the **WHAT** pane should already be configured.

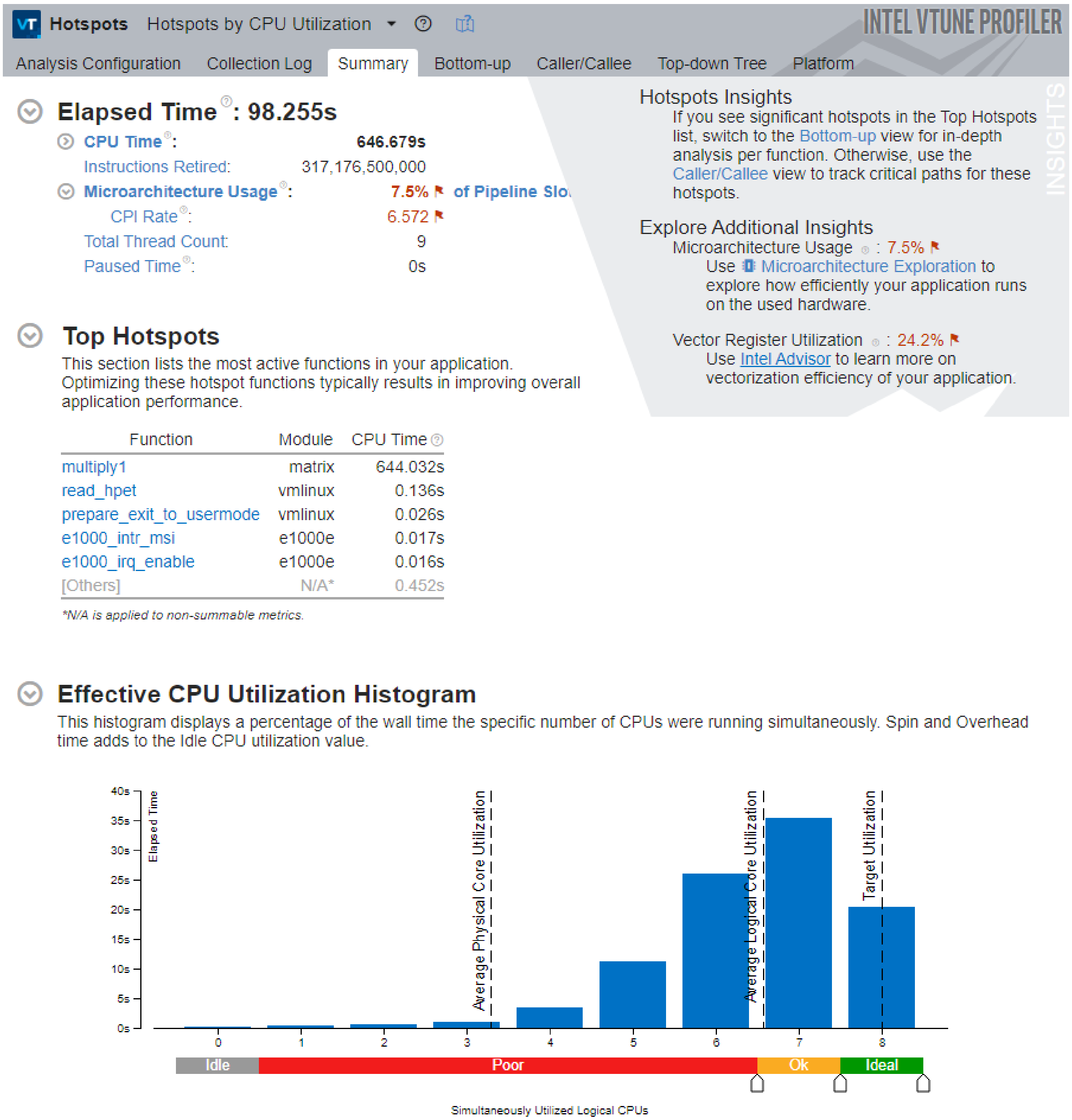
   If not, provide the path to the application in the **Application** textbox.
4. In the **HOW** pane, the **Hotspots** analysis is pre-selected.

For the collection mode, you can choose between **User-Mode Sampling** and **Hardware Event-Based Sampling**. These sampling methods are different, but, typically, it is better to use Hardware Event-Based Sampling when possible, since it provides greater detail with lower overhead.

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

Interpret Hotspots Result Data

Once the sample application exits, Intel® VTune™ Profiler finalizes the result and opens the **Summary** viewpoint.



This viewpoint offers multiple metrics. Hover over the question mark icons to get a detailed description of each metric.

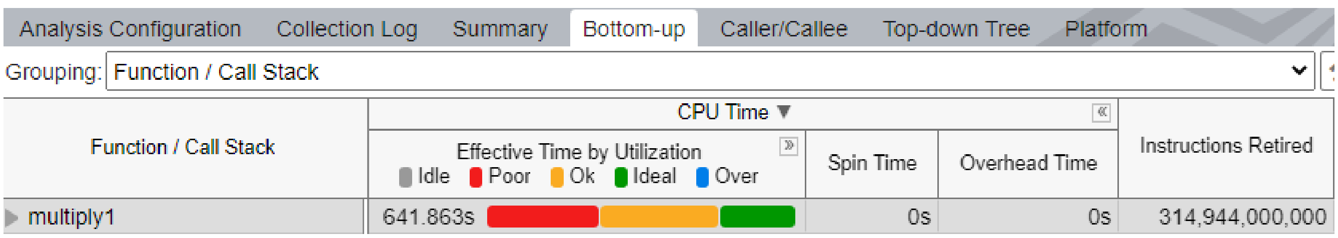
Note that the total **CPU Time** for the application is equal to about 644 seconds. It is the sum of CPU time for all threads in the application. The **Total Thread Count** is 9, so the application is multi-threaded.

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 roughly 640 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.

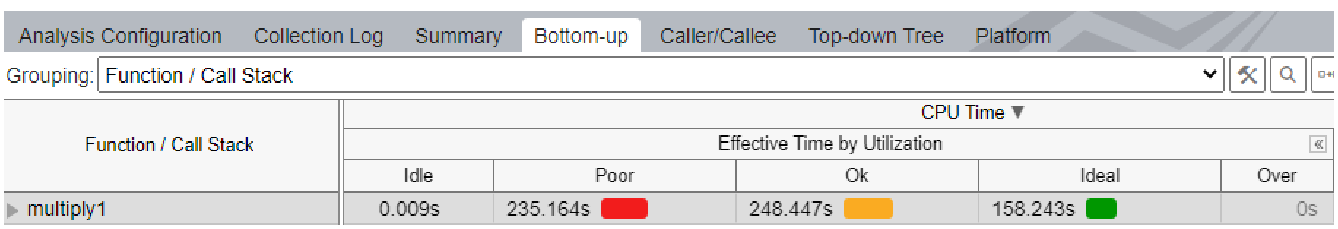
Identify Most Time-Consuming Code Areas

To get a per-function view of the code, switch to the **Bottom-up** tab. By default, the data in the grid is grouped by function. You can change the grouping level using the **Grouping** menu at the top of the grid.

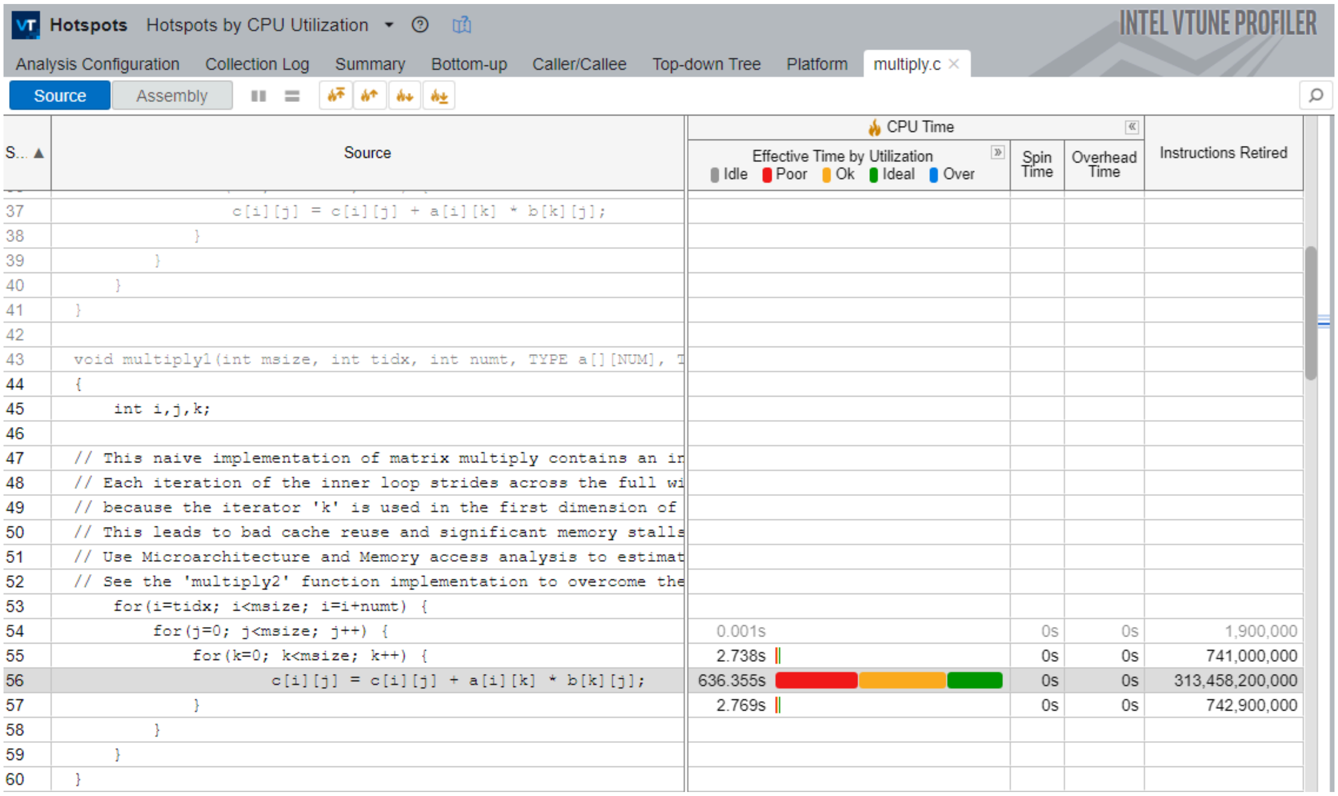


The `multiply1` function took the most time to execute, roughly 640 seconds, and shows a poor CPU utilization.

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.



Double-click the `multiply1` function on the **Bottom-up** grid to open the **Source** window.



Note that the most time-consuming line is attributed to the loop that performs the matrix multiplication in the `multiply1` function.

To analyze the behavior of this loop in relation to memory, run the **Memory Access** analysis.

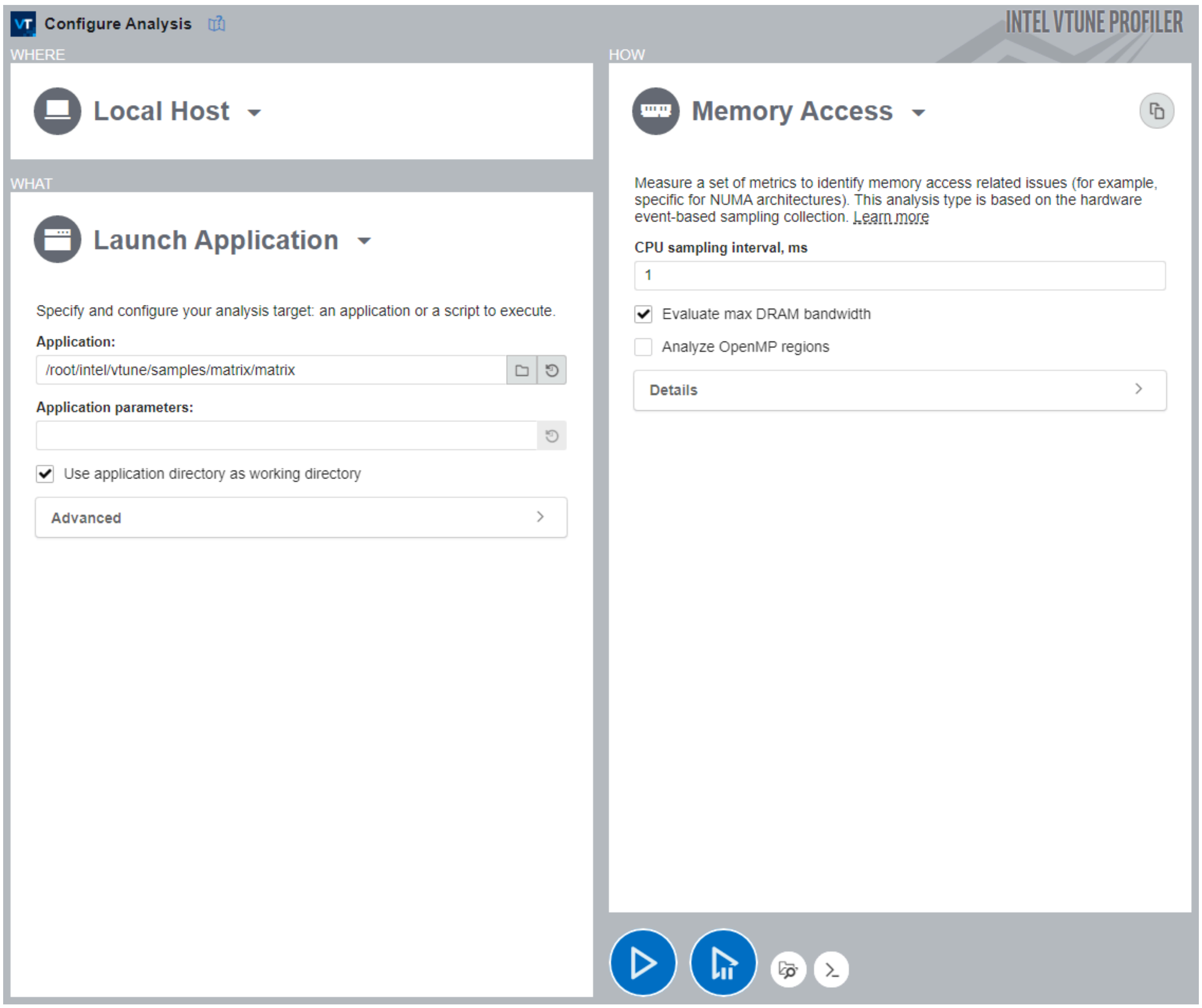
Next step: Analyze Memory Access.

Analyze Memory Access

6

To understand the exact mechanics behind the memory access problems in the `multiply1` loop, run the **Memory Access** analysis.

Run Memory Access Analysis



To run the **Memory Access** analysis:

1. Click the **Memory Access** icon in the previously collected Performance Snapshot result or click the **Configure Analysis** button in the main toolbar.
2. If you clicked the **Memory Access** analysis icon, the Memory Access analysis should be pre-selected. If not, select this analysis type in the HOW pane.
3. In the **HOW** pane, disable the **Analyze OpenMP regions** option as it is not required for this application.

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

Interpret Memory Access Data

Once the sample application exits, Intel® VTune™ Profiler finalizes the result and opens the **Summary** viewpoint.

Memory Access | Memory Usage | INTEL VTUNE PROFILER

Analysis Configuration | Collection Log | Summary | Bottom-up | Platform

Elapsed Time: 102.308s

CPU Time:	634.334s	
Memory Bound:	**84.5%**	**of Pipeline Slots**
L1 Bound:	1.9%	of Clockticks
L2 Bound:	0.5%	of Clockticks
L3 Bound:	3.9%	of Clockticks
DRAM Bound:	**82.3%**	**of Clockticks**
DRAM Bandwidth Bound:	12.7%	of Elapsed Time
Store Bound:	0.0%	of Clockticks
Loads:	155,107,053,072	
Stores:	17,872,136,148	
LLC Miss Count:	7,876,151,292	
Average Latency (cycles):	39	
Total Thread Count:	10	
Paused Time:	0s	

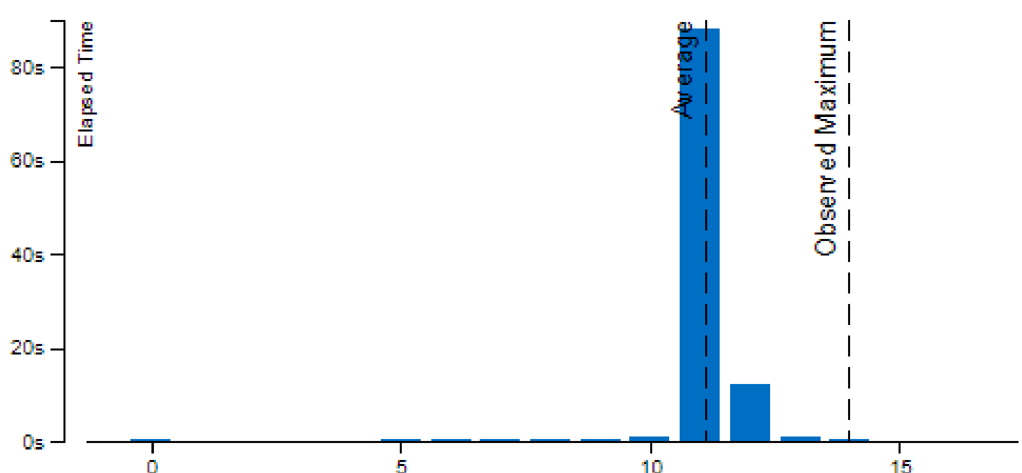
Bandwidth Utilization Histogram

Explore bandwidth utilization over time using the histogram and identify memory objects or functions with maximum contribution to the high bandwidth utilization.

Bandwidth Domain: DRAM, GB/sec

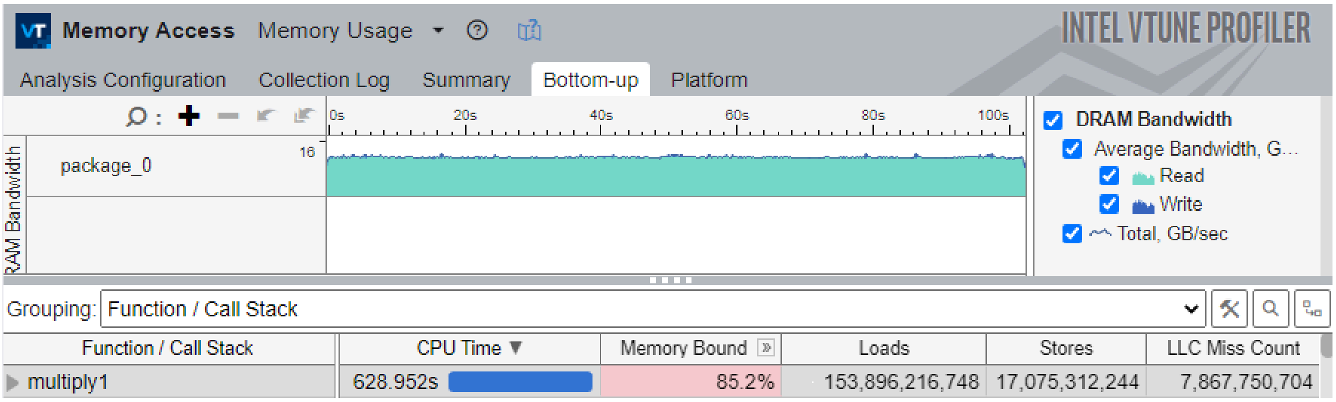
Bandwidth Utilization Histogram

This histogram displays the wall time the bandwidth was utilized by certain value. Use sliders at the bottom of the histogram to define thresholds for Low, Medium and High utilization levels. You can use these bandwidth utilization types in the Bottom-up view to group data and see all functions executed during a particular utilization type. To learn bandwidth capabilities, refer to your system specifications or run appropriate benchmarks to measure them; for example, Intel Memory Latency Checker can provide maximum achievable DRAM and Interconnect bandwidth.



Once again, note that the application is severely bound by memory accesses. The fact that the system is not bound by the **DRAM Bandwidth** alone indicates that the application is bound by frequent, but small, requests to memory, rather than by the saturated physical DRAM Bandwidth.

Switch to the **Bottom-up** tab to see the exact metrics for the `multiply1` function.



The `multiply1` function is at the top of the grid with the highest **CPU Time** and high **Memory Bound** metric values.

Note that the **LLC Miss Count** metric is very high. This indicates that the application uses a cache-unfriendly memory access pattern, which causes the processor to frequently miss the LLC and request data from DRAM, which is expensive in terms of latency.

A good way to resolve this issue is to apply the loop interchange technique, which, in this case, changes the way the rows and columns of the matrices are addressed in the main loop. This way, the inefficient memory access pattern is eliminated, enabling the processor to make better use of the LLC.

Next step: Resolve Memory Access Issue.

Resolve Memory Access Issue

7

NOTE

- Across this tutorial, the Intel® C++ Compiler Classic is used. Your results and workflow may vary depending on the compiler that you use.
- In this stage of the tutorial, you will be instructed to set the Optimization Level of the compiler to Maximum Optimization (Favor Size) (-O1) as opposed to Maximum Optimization (Favor Speed) (-O2).

  While it makes sense to perform performance profiling with maximum optimizations that favor speed enabled, we will use this as an example to demonstrate how Intel® VTune™ Profiler can help detect issues related to unobvious behavior of compiler options. In case of the Intel® C++ Compiler Classic, the -O1 option disables automatic vectorization.

  Such issues can occur in real, larger projects, with reasons that range from something as simple as a typo, to something more complicated, such as the lack of awareness of how particular compiler options influence performance.

  For example, some compilers, such as `gcc`, do not attempt vectorization at -O2 level, unless instructed to do so using the `-ftree-vectorize` option, and will only perform automatic vectorization at the -O3 level.

Follow these steps to edit and recompile the code using the Intel® oneAPI DPC++/C++ Compiler:

1. In the `/opt/intel/oneapi/compiler/latest/env` folder, run this command to set compiler environment variables:

```
source env.vars
```

2. Locate the matrix sample application folder on your machine. By default, it is placed in:

   `$HOME/intel/vtune/samples/matrix`
3. Using a text editor of your choice, open the `Makefile` located in the `../matrix/linux/` folder.
4. Change line 42 from:

```
CFLAGS  = -g -O3 -fno-asm
```

   To:

```
CFLAGS  = -g -O1
```

5. Change line 43 from:

```
OPTFLAGS = -xSSE3
```

   To:

```
OPTFLAGS =
```

6. Save and close the Makefile.
7. Open the `multiply.h` header file located in `../matrix/src` folder with a text editor.
8. Change line 36 from:

```
#define MULTIPLY multiply1
```

   To:

```
#define MULTIPLY multiply2
```

   This changes the program to use the `multiply2` function from the `multiply.c` source file, which implements the loop interchange technique that resolves the memory access problem.
9. Save and close the `multiply.h` file.

10. Navigate to the `../matrix/linux` folder and use this command to recompile the application:

```
make icc
```

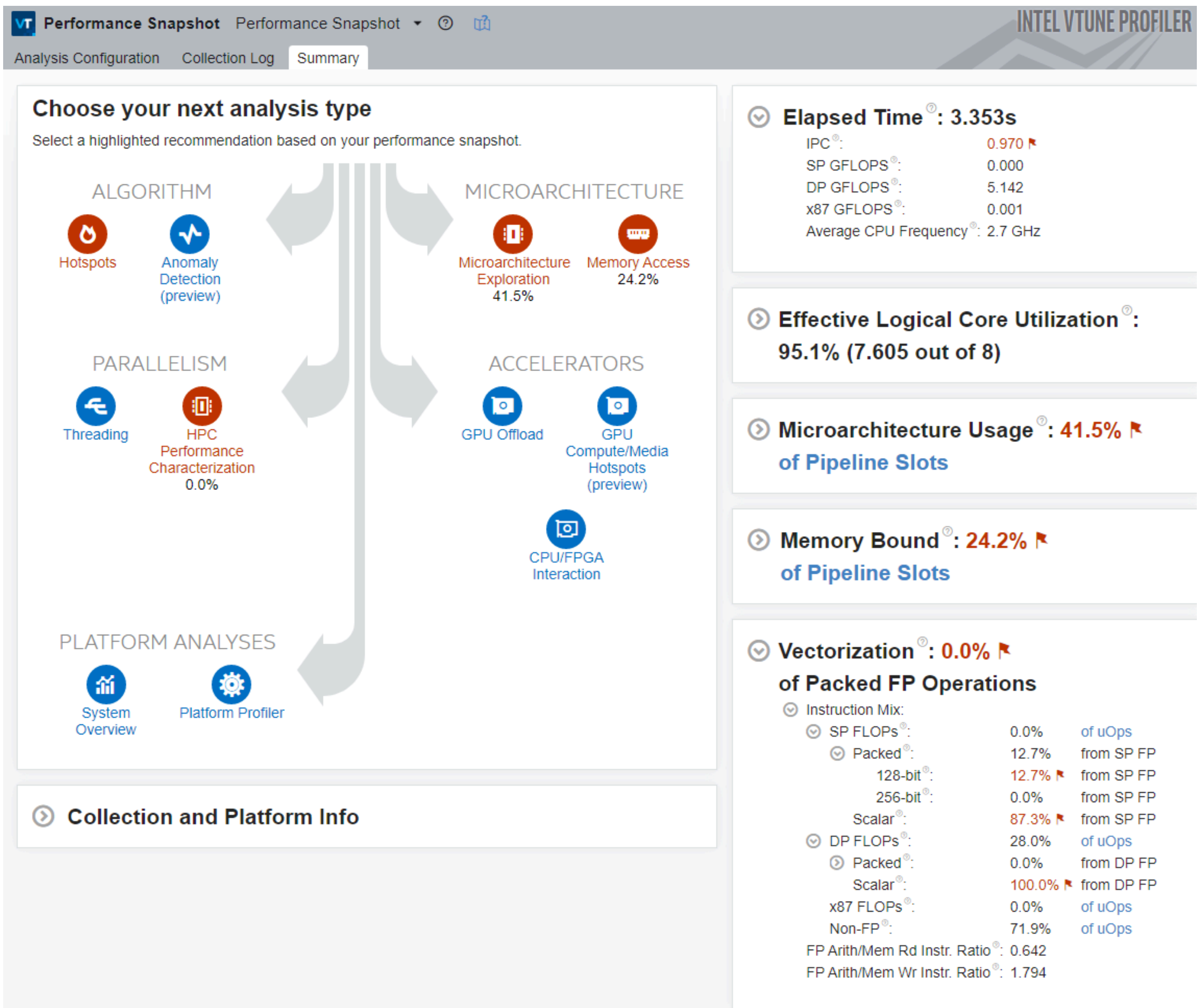
Next step: Analyze Performance After Optimization.

Analyze Performance After Optimization 8

To see the improvement provided by using the loop interchange technique, run the Performance Snapshot analysis again.

> **NOTE**
> Depending on your compiler and IDE, when configuring the analysis, you may need to browse to a different executable that was generated during recompilation in the previous step.

Once the sample application finishes, the Performance Snapshot **Summary** window opens.



Observe these main indicators:

- The **Elapsed Time** for the application is significantly reduced. This improvement is mainly the result of the eliminated memory access bottleneck, which caused the processor to frequently miss the cache and request data from the DRAM, which is very expensive in terms of latency.
- The **Vectorization** metric is equal to 0.0%, which means that the code was not vectorized. Due to this, Performance Snapshot highlights the HPC Performance Characterization analysis as a potential next step.

In this case, the code was not vectorized because the Intel® oneAPI DPC++/C++ Compiler does not perform vectorization when compiling with binary size favored (-O1).

To enable automatic vectorization by the compiler, follow these steps:

1. Open the `Makefile` located in `../matrix/linux` folder with a text editor.
2. Change line 42 from:

```
CFLAGS  = -g -O1
```

   To:

```
CFLAGS  = -g -O2
```

3. Run the following command to recompile the application:

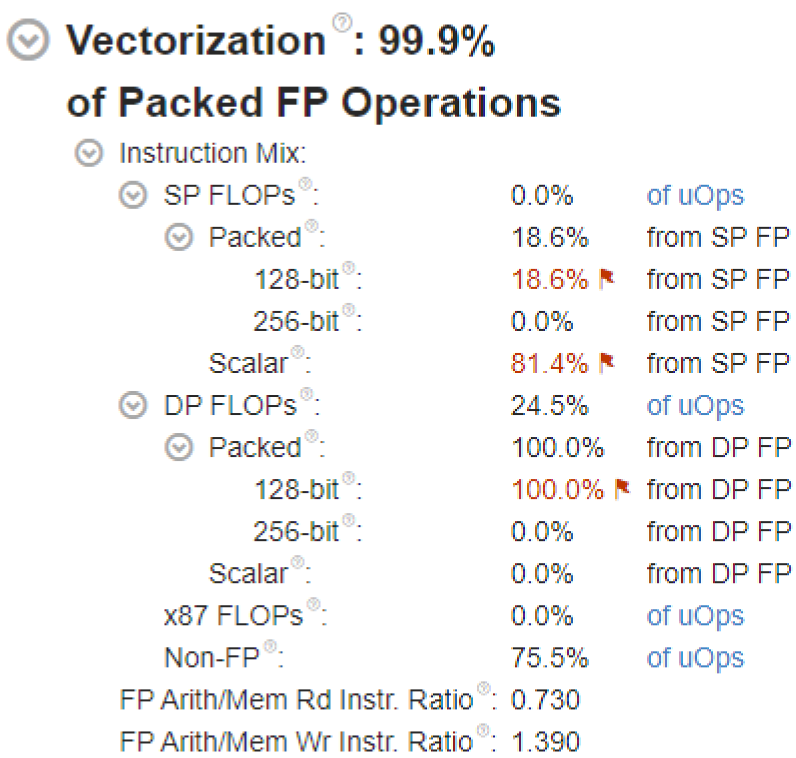
```
make icc
```

Next step: Analyze Vectorization Efficiency.

Analyze Vectorization Efficiency 9

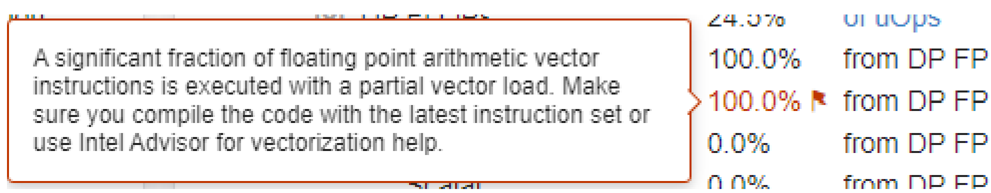
Once you recompile the application with the -O2 level enabled, run the Performance Snapshot analysis again to analyze vectorization efficiency.

Once the analysis is complete, see the **Vectorization** pane of the **Summary** window.



Observe these main indicators:

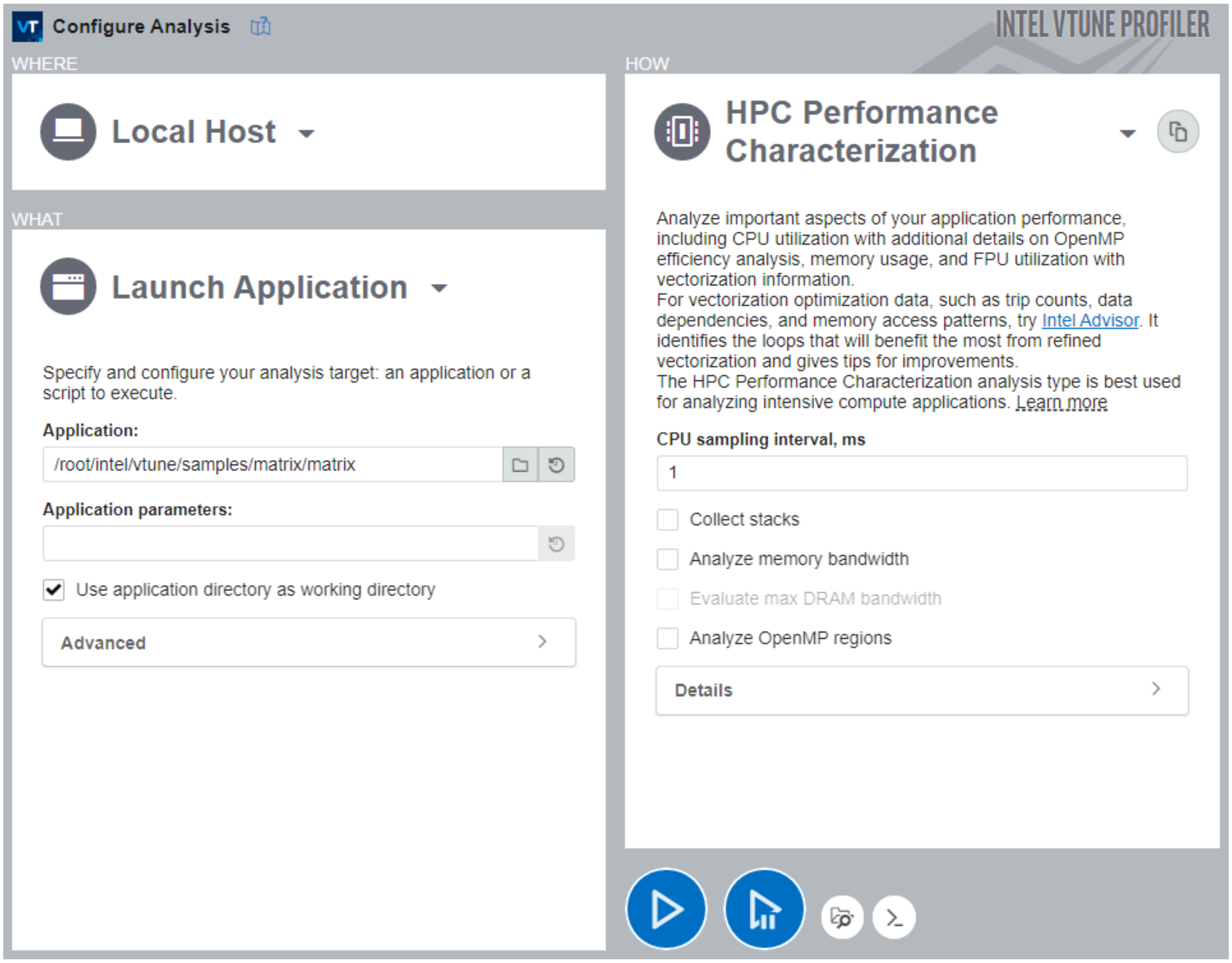
1. The overall **Vectorization** metric is equal to 99.9%, which indicates that the code was vectorized.
2. However, there are red flags next to the **128-bit Packed FLOPs** metrics. Hover over the red flag icon or the metric value to get a description of the issue.

   In this case, Intel® VTune™ Profiler indicates that a significant portion of floating-point instructions is executed with partial vector load.

Since the analysis was performed on a machine based on an Intel processor capable of using the AVX2 instruction set, the fact that all instructions were executed using only the 128-bit registers means that the 256-bit wide AVX2 registers were not utilized at all. Therefore, VTune Profiler flags the 100.0% utilization of 128-bit vector registers as an issue.

To understand what vector instruction set is actually used, run the **HPC Performance Characterization** analysis.



To run the analysis:

1. Click the **HPC Performance Characterization** analysis icon from the **analysis tree**.
2. Disable the **Collect stacks**, **Analyze Memory bandwidth** and **Analyze OpenMP regions** options as they are not required for vectorization analysis.
3. Click the **Start** button to run the analysis.

Once the data collection is complete, VTune Profiler opens the default **Summary** window of the HPC Performance Characterization Analysis.

Vectorization: 99.8% of Packed FP Operations

Instruction Mix:

SP FLOPs:	0.0%	of uOps
DP FLOPs:	31.5%	of uOps
Packed:	99.8%	from DP FP
128-bit:	99.8%	from DP FP
256-bit:	0.0%	from DP FP
Scalar:	0.2%	from DP FP
x87 FLOPs:	0.0%	of uOps
Non-FP:	68.5%	of uOps

FP Arith/Mem Rd Instr. Ratio: 0.925

FP Arith/Mem Wr Instr. Ratio: 1.852

Top Loops/Functions with FPU Usage by CPU Time

This section provides information for the most time consuming loops/functions with floating point operations.

Function	CPU Time	% of FP Ops	FP Ops: Packed	FP Ops: Scalar	Vector Instruction Set	Loop Type
[Loop at line 71 in multiply2]	16.764s	31.7%	100.0%	0.0%	SSE2(128)	
[Loop at line 70 in multiply2]	0.059s	29.4%	100.0%	0.0%	SSE2(128)	
[Loop at line 50 in init_arr]			0.0%	100.0%		

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

Focus on the **Vectorization** section of the **Summary** window.

Note that the main loop of the `multiply2` function was vectorized using the older **SSE2** instruction set, while compilation and analysis were performed on an AVX2-capable processor. Therefore, a portion of hardware resources remains underutilized.

Next step: Enable Platform-Appropriate Vectorization.

Enable Platform-Appropriate Vectorization 10

NOTE

- For an in-depth exploration of vectorization, try Intel® Advisor. It is a performance analysis tool that offers deep insights into vectorization opportunities, vectorization efficiency, dependencies, and much more.
- In this section, you will be instructed to use the -xHost option compile the application with the best instruction set extension out of the ones that your processor performing the compilation supports. To generate multiple code paths that enable your software to run on a variety of microarchitectures, see the ax, Qax option of the Intel® oneAPI DPC++/C++ Compiler.

Enable Full Vectorization

To enable the use of a vector instruction set appropriate for the platform, one possible way is to instruct the compiler to use the same vector extension as the best one available in the processor performing the compilation.

Follow these steps to enable platform-appropriate vectorization:

1. Open the `Makefile` located in `../matrix/linux` with a text editor.
2. Change line 43 from:

```
OPTFLAGS =
```

   To:

```
OPTFLAGS = -xHost
```

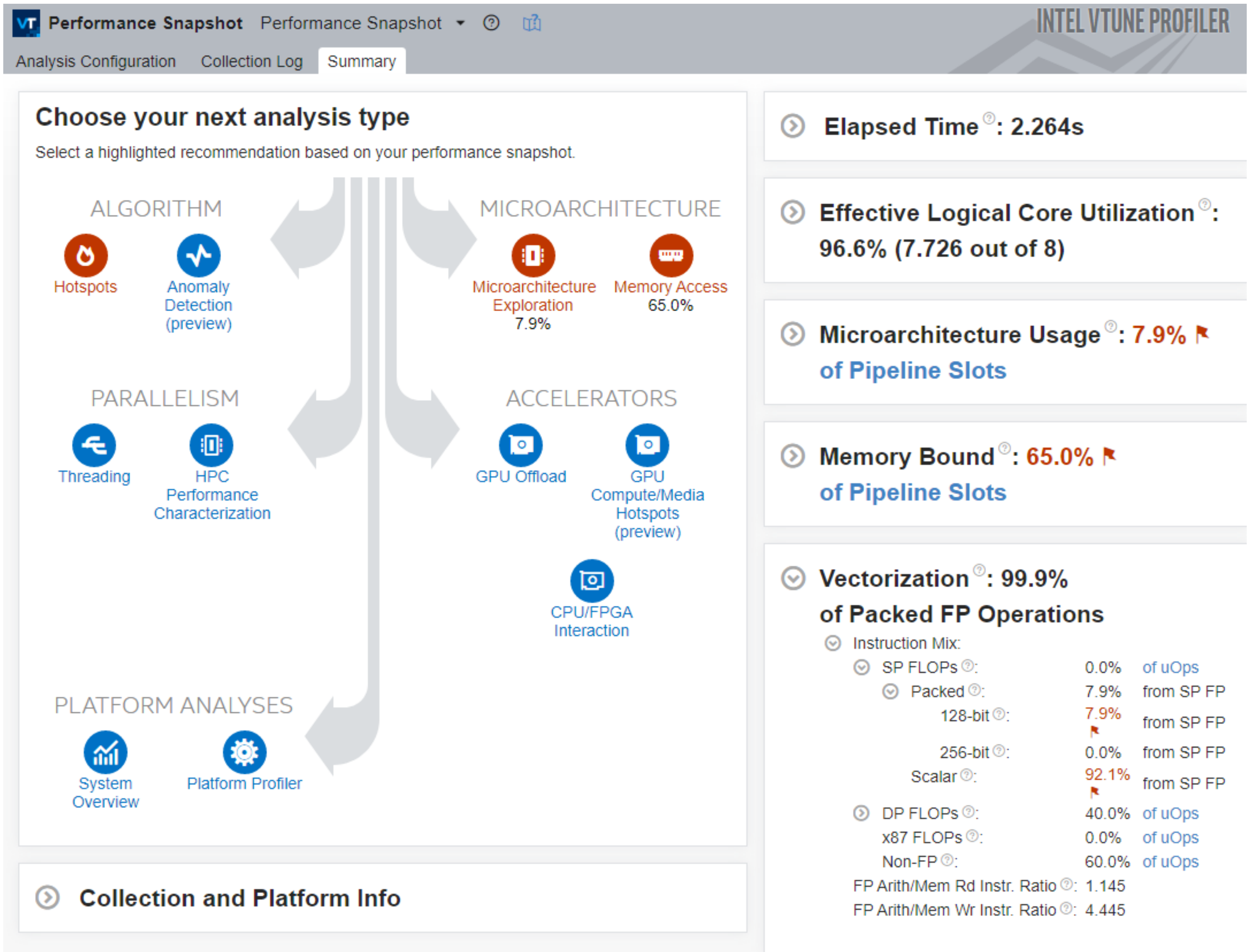
   This option instructs the compiler to use the best instruction set extension that the processor performing the compilation supports.
3. Save and close the `Makefile` and recompile the application using command:

```
make icc
```

Check Vectorization with Performance Snapshot

Run the Performance Snapshot analysis to ensure that the application is properly vectorized.

Once the application exits, Intel® VTune™ Profiler opens the Performance Snapshot **Summary** window.



Observe these main indicators:

- The **Elapsed Time** for the application has slightly decreased.
- The **Vectorization** metric equals to 99.9%, so the code was fully vectorized.
- A total 100.0% of **Packed DP FLOP** instructions were executed using the 256-bit registers. Therefore, even without running the HPC Performance Characterization analysis, the conclusion is that the AVX2 vector extensions were fully utilized.
- VTune Profiler highlights the **Microarchitecture Usage** metric and offers to use the **Microarchitecture Exploration** analysis to understand how exactly the application is underutilizing the microarchitecture.

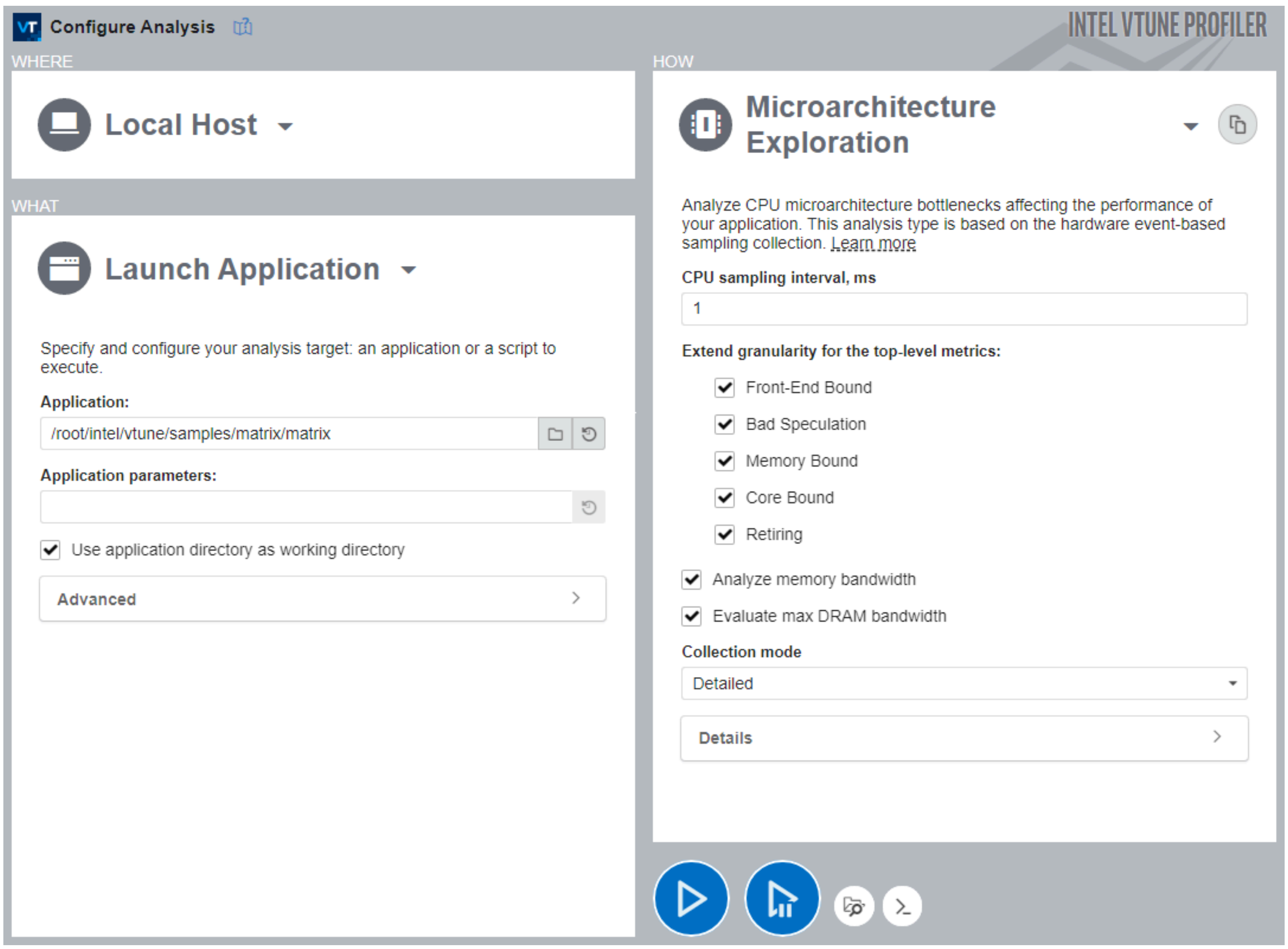
Next step: Analyze Microarchitecture Usage.

Analyze Microarchitecture Usage 11

While the previous optimizations resulted in great benefit to the total elapsed time of the application, there are still areas for improvement. The Performance Snapshot analysis has highlighted that the microarchitecture is not utilized well.

Run the Microarchitecture Exploration analysis to identify opportunities for improvement.

Run Microarchitecture Exploration Analysis

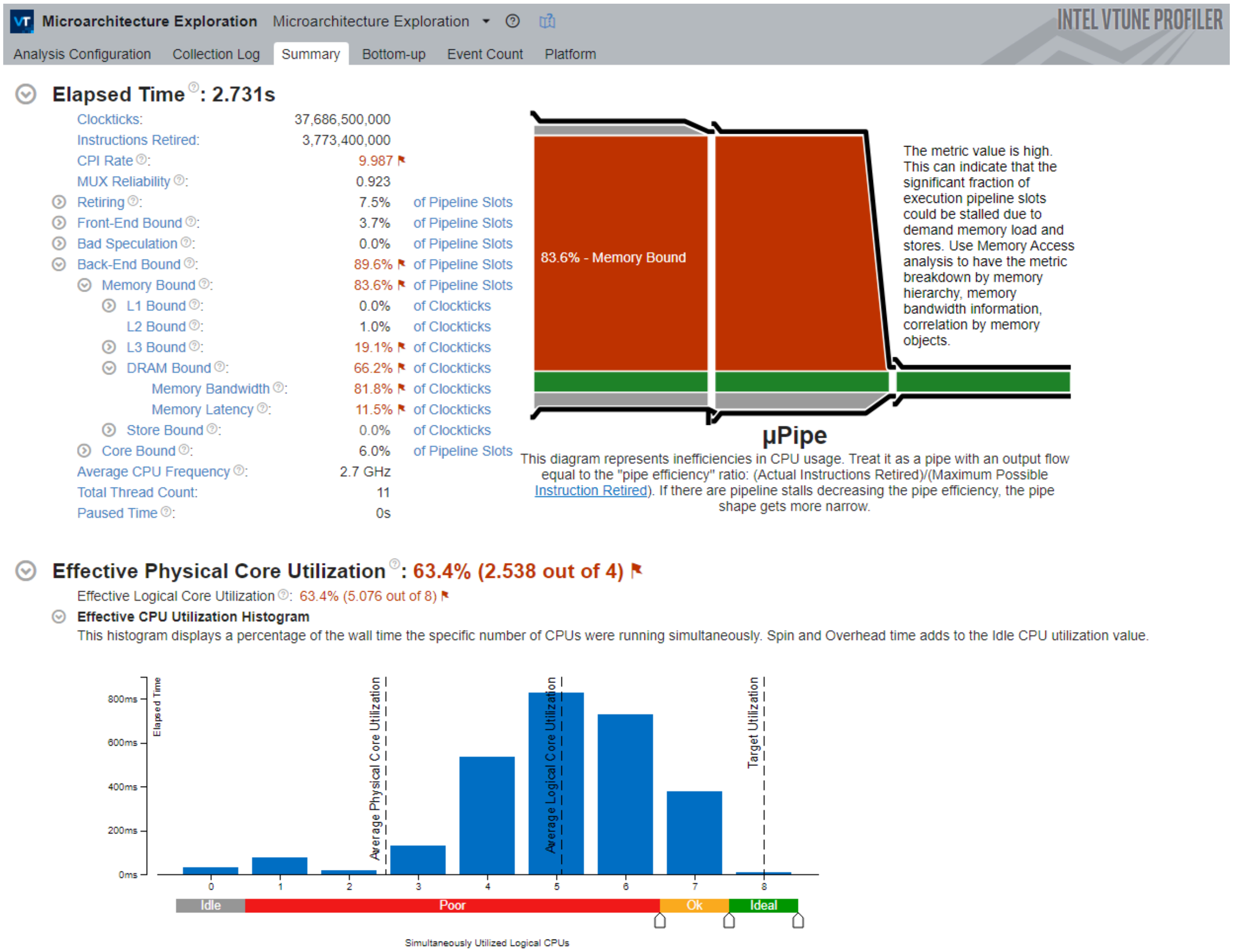


To run the Microarchitecture Exploration analysis:

1. In the Performance Snapshot **analysis tree**, click the **Microarchitecture Exploration** analysis icon.
2. In the **HOW** pane, enable all extra options.
3. Click the **Start** button to run the analysis.

Interpret Microarchitecture Exploration Result Data

Once the application exits, Intel® VTune™ Profiler opens the default **Summary** window.



This view shows the following:

- **Elapsed Time section:** this section shows metrics related to hardware utilization levels for your hardware. Hover over the flagged metrics to get a description of the issues, possible causes, and suggestions for resolving the issue.

  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. Intel® 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.
- **µPipe Diagram:** the µPipe, or Microarchitecture pipe, 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 User Guide for a more detailed explanation of the µPipe.
- **Effective CPU Utilization Histogram:** this histogram 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.

In this case, observe the following indicators:

- The **Memory Bound** metric is high, so the application is bound by memory access.
- The **Memory Bandwidth** and **Memory Latency** metrics are high.

Considering these factors together, the conclusion is that the application has a memory access issue. However, this issue is slightly different in nature from the memory access issue previously resolved using the loop interchange technique.

Before the introduction of the loop interchange, the application was mainly bound by the cache-unfriendly memory access pattern, which resulted in a large number of LLC (Last-Level Cache) misses. This, in turn, resulted in frequent requests to the DRAM.

In this case, the fact that the **Memory Bandwidth** metric is high means that the application has saturated the bandwidth limits of the DRAM. While nothing can be done to increase the physical capabilities of the DRAM, the application can be modified to make even better use of the Last-Level Cache and to reduce the number of loads from the DRAM even further.

(Optional) Improve Cache Reuse

In general, most developers stop further optimizing their application when they have reached their desired performance goal. The performance improvement gained by optimizing the `matrix` application has resulted in a decrease of application wall time from roughly 90 seconds to roughly 2.5 seconds.

If you wish to experiment further, you can modify the code to implement the **cache blocking** technique. Cache blocking is an approach for rearranging data access in such a way that blocks of data get loaded into the cache and are reused for as long as they are needed, greatly reducing the number of DRAM accesses.

To modify the code to use the cache blocking technique:

1. In the `multiply.h` header file, change line 36:

```
#define MULTIPLY multiply2
```

   To:

```
#define MULTIPLY multiply4
```

2. Save changes and recompile the application.

This modifies the code to use the `multiply4` function from the `multiply.c` source file, which implements the cache blocking technique.

Once the application is recompiled, you can run an analysis of your choice to determine the performance improvement.

Next step: Compare with Previous Result.

Compare with Previous Result 12

Over the course of the Tutorial, you've applied multiple changes to improve the performance of the `matrix` sample application.

To get a detailed view of the performance improvement, you can use the **Compare Results** feature of Intel® VTune™ Profiler.

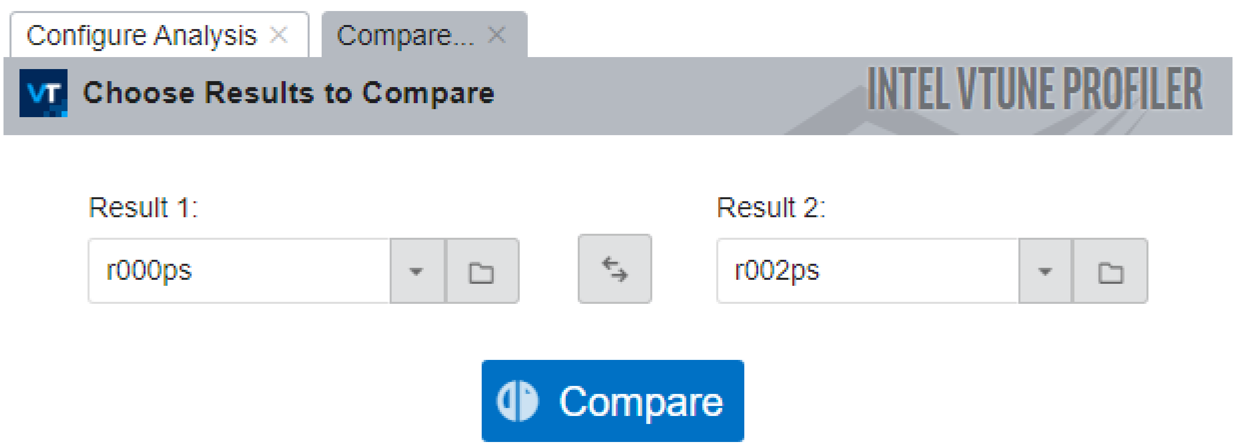
Compare Performance Before and After Optimization

You can compare results collected with VTune Profiler to better see the changes in performance.

While you can compare results from different analysis types (such as Hotspots and Performance Snapshot), only the metrics that are applicable to both analysis types simultaneously are shown.

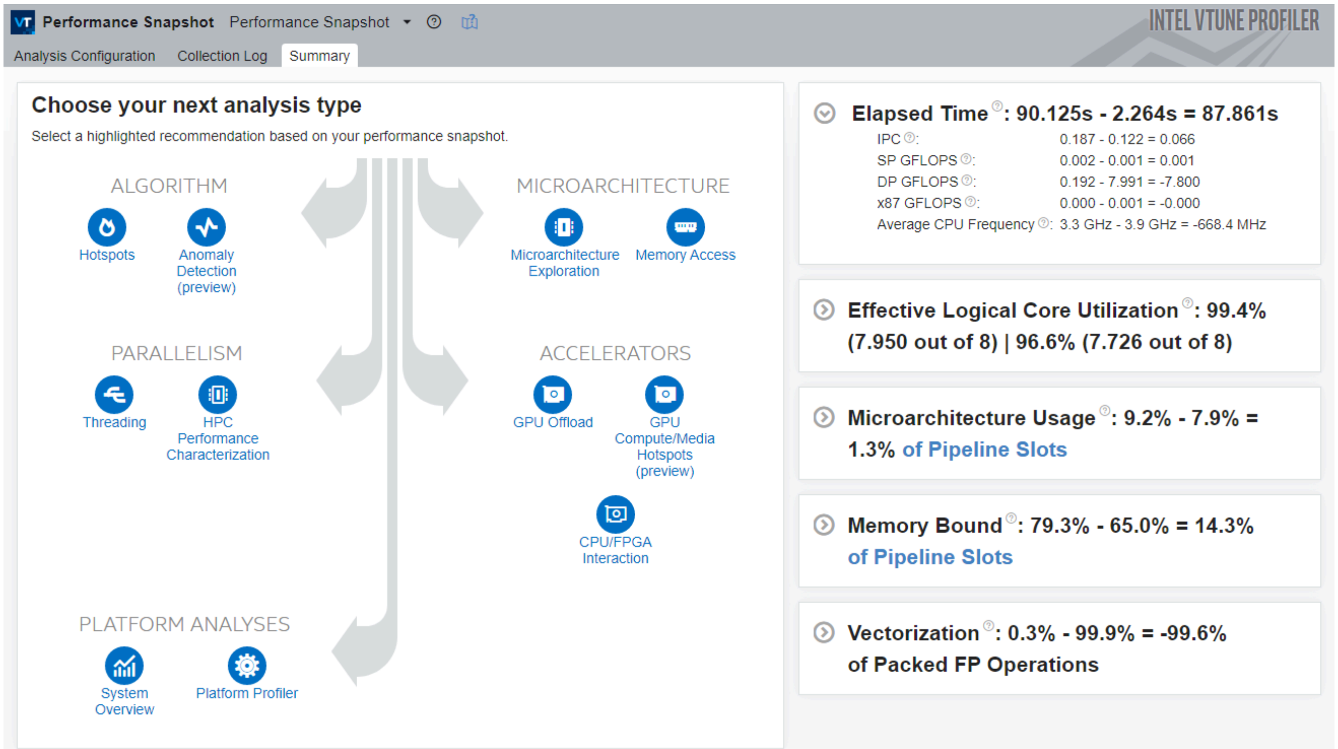
To compare results:

1. Click the **Compare Results** button in the **Main Toolbar**.
2. Select the results that you want to compare.



3. Click the **Compare** button.

VTune Profiler profiler calculates the differences between metrics and opens the default **Summary** window.



You can expand any metric pane and see the difference between all metrics that are applicable to both results.

For example, for the `matrix` sample application, the **Elapsed Time** was reduced by almost 88 seconds.

Summary

13

You have completed the Finding Common Bottlenecks 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 Takeaways
1. Find the bottleneck	You started with Performance Snapshot to determine main limiting factors and next steps for optimization: 1. Using the Hotspots analysis to isolate problem to a specific code area. 2. Using the Memory Access analysis to understand the exact mechanics behind the bottleneck.	• When you first analyze an application, it is a good idea to start with the Performance Snapshot analysis to determine main problem areas and next steps. • Use the Hotspots analysis to isolate the performance issue to a specific area of code. Click the hotspot function name in the **Bottom-up** window to see the code lines responsible for bottleneck. • Use the Memory Access analysis to determine issues related to inefficient DRAM accesses, one of the most common limiting factors in software.
2. Resolve issue and recompile application	You edited the code and recompiled the application to eliminate the cache-unfriendly DRAM access pattern. This has resulted in a great decrease of application running time. You've set compiler options to use a different optimization level to see how compiler options can influence vectorization.	• Using efficient, cache-friendly DRAM access patterns can result in a significant increase in performance. • Compiler options can influence the behavior of the application in unobvious ways, especially when multiple different compilers are used. VTune Profiler can help identify issues related to the application being vectorized improperly, which underutilizes available hardware resources.
3. Resolve vectorization issues	You recompiled the application with a different optimization level, and the code was vectorized. However, while using Performance Snapshot, you've noticed that only the 128-bit vector registers were utilized, while the 256-bit registers were not utilized at all. By using the HPC Performance Characterization analysis, you've noticed that the vector instruction set extension SSE2 was used, which is an older instruction set extension. A portion of hardware resources remained underutilized. You've recompiled the application again with different options to ensure vectorization was performed according to full platform capability.	• Both the Performance Snapshot and the HPC Performance Characterization analysis types can help identify issues related to improper vectorization. • While compiler options are well-documented and their behavior is known, it is easy to miss a peculiarity of an option. This can lead to not compiling an application to make the best use of hardware resources straight away, no matter what compiler is used. VTune Profiler can help catch such issues on all stages of development.

Step	Tutorial Recap	Key Tutorial Takeaways
4. Analyze Microarchitecture Usage	As recommended by Performance Snapshot, you used the Microarchitecture Exploration analysis to identify next optimization steps. Using this analysis type, you saw that the best way to further optimize the application was the cache blocking technique.	• VTune Profiler provides a large number of microarchitecture metrics tuned by Intel architects to enable you to make an informed optimization decision. • You used the metrics and the **μPipe** diagram to make the next optimization decision.
5. Check your work	You used the **Compare Results** feature to compare the performance of the application at different optimization stages.	Perform regular regression testing by comparing analysis results before and after optimization. From the 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.

See Also

Explore the User Guide

Tuning and configuration recipes in the VTune Profiler Cookbook

More tutorials with associated sample code