Intel® VTune™ Profiler

Vladimir Tsymbal, Technical Consulting Engineer, Intel

intel®

Agenda

A short intro to the Intel® VTune Profiler

Collecting Hotspots with and w/o EBS

Analysis types overview

Microarchitecture Exploration

HPC Performance Characterization

Memory Analysis

What is VTune Profiler, and what it is not?

- The Intel® VTune™ Profiler (aka VTune) if one of the many **dynamic** performance profiling tools (WPA on Windows, Linux Perf, gprof) for native (C,C++, C#,Fortran, and to some extent jit or interpreted languages, Java, Python, OpenCL, Sycl/DPC++, Go)
  - Highest awareness of Intel CPU, GPU, and FPGA microarchitecture
  - Hight awareness of parallel runtimes: native/Posix threads, OpenMP, TBB, MPI, Intel's C/C++ Parallel Language Extensions
  - Fully integrated and self-contained tool

- Nots (Buts):
  - Not a static code analyzer, but…
  - Not a code/system debugger, but…
  - Not a system tracing tool like strace, but…
  - Not a comprehensive advisor for code change, but…

VTune is all the following

- System wide profiling
- Application-level profiling
- Profiling hardware events
- Detecting CPU/GPU microarchitecture issues
- Heap/stack memory analysis
- System I/O analysis
- Statistical and Instrumented measurement
- Averaged measurements and precise tracing

Tools diversity in the VTune Zoo

- VTune Profiler

  `vtune, vtune-gui, vtune-backend`

- APS – Application Performance Snapshot tool

  `aps, aps-report`

- VPP – VTune Platform Profiler

  `vpp-server, vpp-collect`

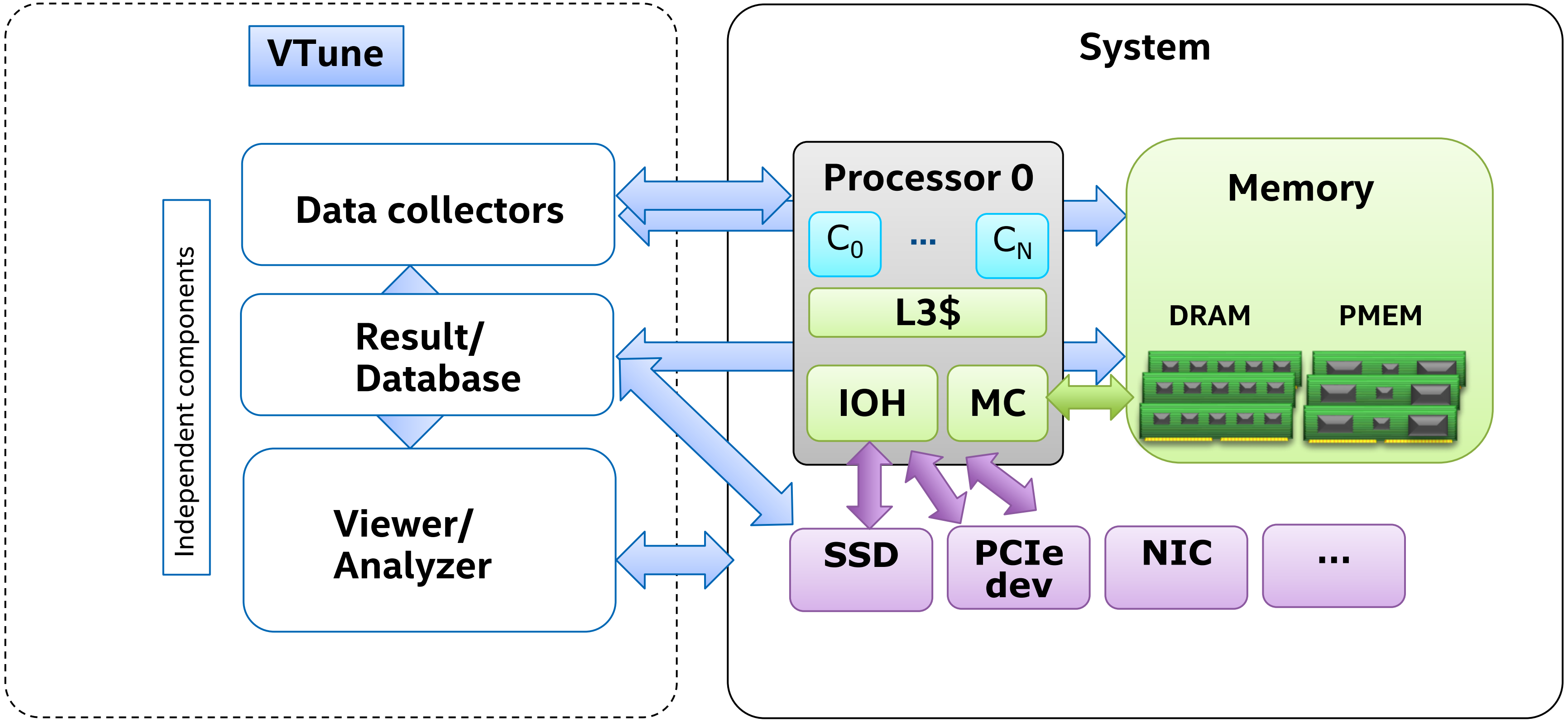
- Utilities

  `vtune-self-checker, prepare-debugfs`

- Standalone data collectors

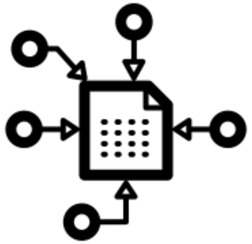
  `sep, pin, emon`

Visible to a user VTune (standalone) structure



More complex structure with remote and server usage

Data collectors



- Software data collector (Hotspots, Threading, etc.)
  - No restrictions on virtual environments
  - No driver
  - Instrumentation based (overhead)
  - No administrative rights required
- Hardware collector (Hotspots, Microarchitecture, Memory Analysis, I/O, Accelerators, etc.)
  - VTune driver (sep) or driverless mode (Linux Perf), some admin's help is required
  - For stack collection yet another driver (vtss) or limited Perf stacks
  - All Hardware PMU events and Uncore events
  - Hight resolution (down to 0.1 ms sampling interval)
  - VMs need to virtualize PMU MSRs
- System collectors
  - Windows (ETW), Linux (strace)

Profiling result directory

- A self-contained config, logs, raw traces, and resolved results data base
- Created automatically in a project or current dir, or on path defined by a user
- Auto-numeration and pre- post-fixing for the analysis type
- CL option: -result-dir <path-to-my-result>
- Can be shared and viewed by other users
- Usually asked by Intel support engineers or developers (most system and config info is there)
- Can be accompanied with profiled app binaries. CL option: -archive -result-dir <path-to-my-result>
- Does not contain user source code by default (but can be added in a scope of opened results)
- Raw data traces obtained by collectors (sep, perf) can be imported/added to a result dir

Viewer/Analyzer interface

GUI

- The most comfortable and useful method of analysis
- Good visual experience (complex data, scrolling, filtering, zooming in/out, etc.)
- Can be launched locally (on an analyzed machine) or remotely

Command Line

- Good for collection or analysis procedures automation
- Can be launched locally on a machine without GUI or remotely
- No memory overhead for weak machines (try GNOME 3 Desktop on a machine with 4GB RAM)

HTML GUI via Web Server

- Super powerful method for machines management, remote computing and work collaboration
- Requires VTune web server deployment (not a big deal, though)

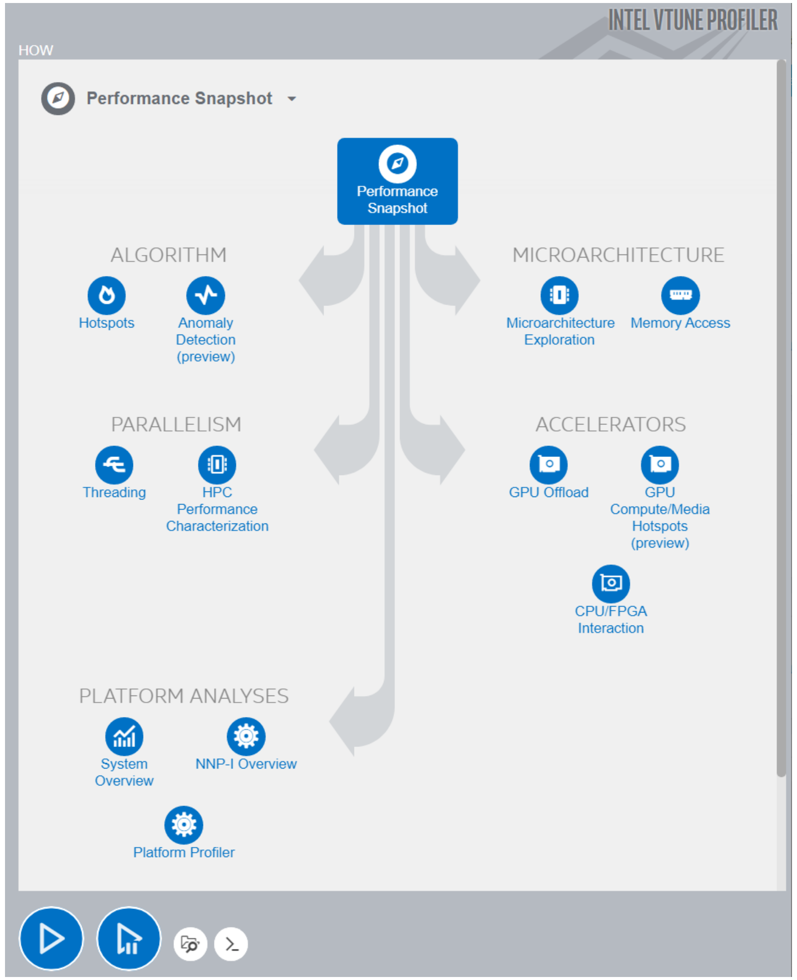
Performance Profiling with Intel® VTune™ Profiler

Intel® VTune™ Profiler

A full set of performance analysis types

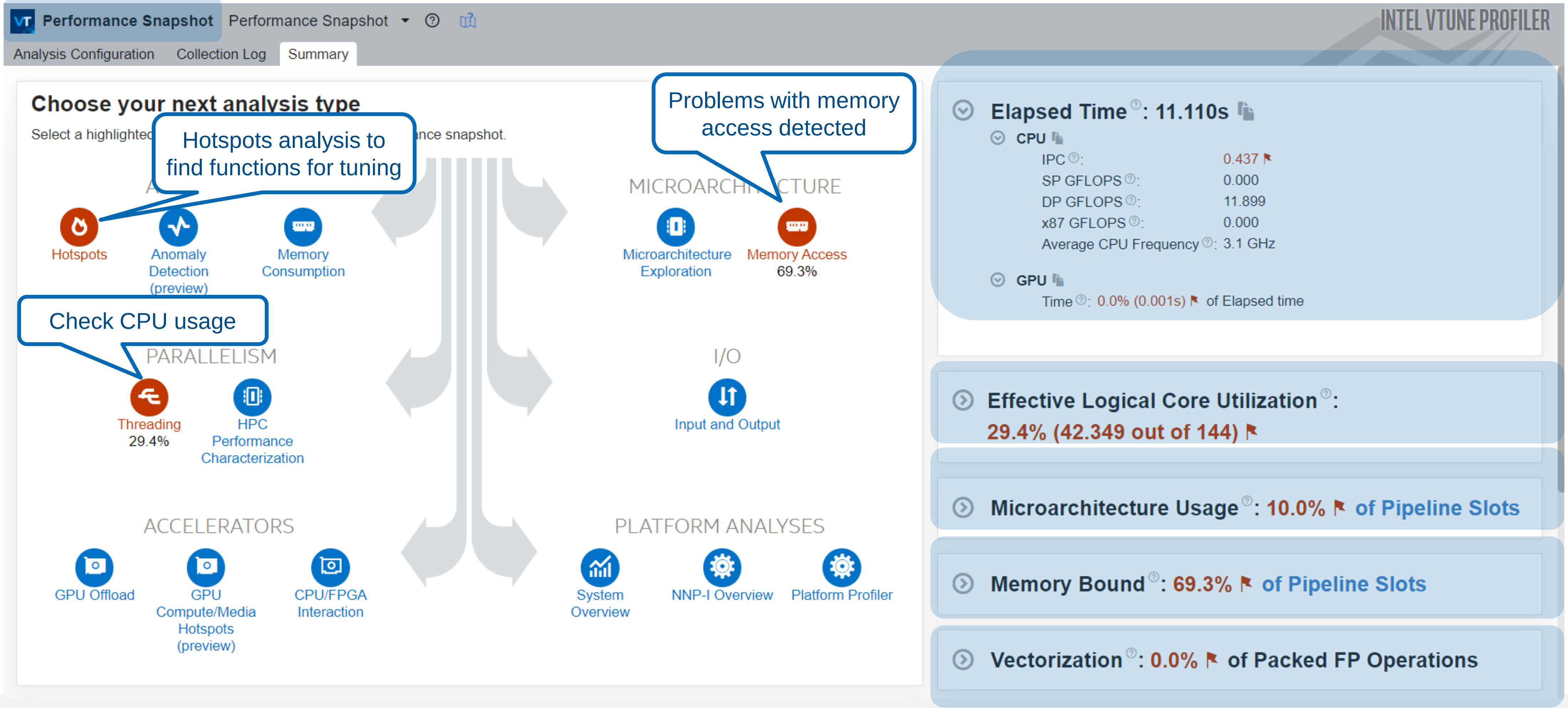
Which one to start with?

- Start with Performance Snapshot if you want to explore performance weakness of your app and get a recommendation how to continue
- Run Platform Profiler for analyzing long running apps on the whole platform (CPU, Memory, Disks, Ethernet, etc.)
- With Input-Output analysis make a snapshot of communication interfaces performance (PCIe, QPI, DRAM, SATA)
- The HPC Performance Characterization helps when your app is parallelized with OpenMP or MPI runtimes
- Dive deeper into a microarchitecture level inefficiencies of application execution with the Microarchitecture Exploration
- Investigate application memory bandwidth and latency problems with the Memory Access



Performance Snapshot

All application weaknesses in one snapshot



HPC Performance Characterization

HPC Performance Characterization | HPC Performance Characterization | INTEL VTUNE PROFILER

Analysis Configuration | Collection Log | Summary | Bottom-up

Characterize OpenMP or MPI application

Effective Physical Core Utilization: 54.7% (39.378 out of 72)

Effective Logical Core Utilization: 34.8% (50.169 out of 144)

Serial Time (outside parallel regions): 0.191s (1.9%)

Parallel Region Time: 9.744s (98.1%)

Estimated Ideal Time: 7.810s (78.6%)

OpenMP Potential Gain: 1.933s (19.5%)

Top OpenMP Regions by Potential Gain

This section lists OpenMP regions with the highest potential for performance improvement. The Potential Gain metric shows the elapsed time that could be saved if the region was optimized to have no load imbalance assuming no runtime overhead.

OpenMP Region	OpenMP Potential Gain	(%)	OpenMP Region Time
multiply1ompparallel:64@unknown:179:180	1.933s	19.5%	9.744s

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

Effective CPU Utilization Histogram

Vectorization: 0.0% of Packed FP Operations

Instruction Mix:

SP FLOPs:	0.0%	of uOps
DP FLOPs:	22.3%	of uOps
Packed:	0.0%	from DP FP
Scalar:	100.0%	from DP FP
x87 FLOPs:	0.0%	of uOps
Non-FP:	77.7%	of uOps

FP Arith/Mem Rd Instr. Ratio: 0.887

FP Arith/Mem Wr Instr. Ratio: 1.890

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 182 in multiply1ompparallel@179]	493.126s	24.6%	0.0%	100.0%		Body

Parallelization: more details

Now many cores were really used in a system

```
Addr of buf1 = 0x7f51f38b2010
Offs of buf1 = 0x7f51f38b2180
Addr of buf2 = 0x7f51eb8b1010
Offs of buf2 = 0x7f51eb8b11c0
Addr of buf3 = 0x7f51e38b0010
Offs of buf3 = 0x7f51e38b0100
Addr of buf4 = 0x7f51db8af010
Offs of buf4 = 0x7f51db8af140
Threads #: 64 OpenMP threads
Matrix size: 4096
Using multiply kernel: multiply1
Execution time = 7.114 seconds
```

Effective Physical Core Utilization: 54.7% (39.378 out of 72)

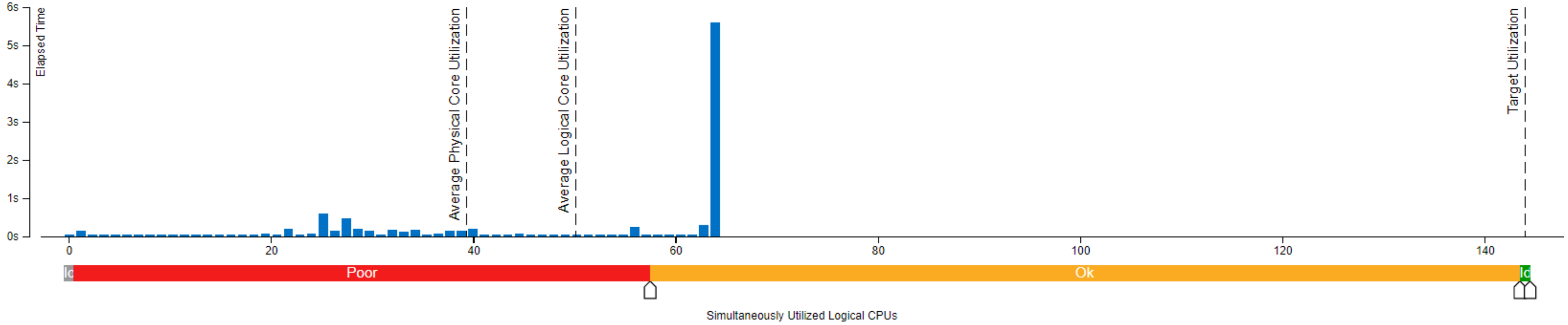
Effective Logical Core Utilization: 34.8% (50.169 out of 144)

Serial Time (outside parallel regions): 0.191s (1.9%)

Parallel Region Time: 9.744s (98.1%)

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.



Threading analysis, TBB based apps

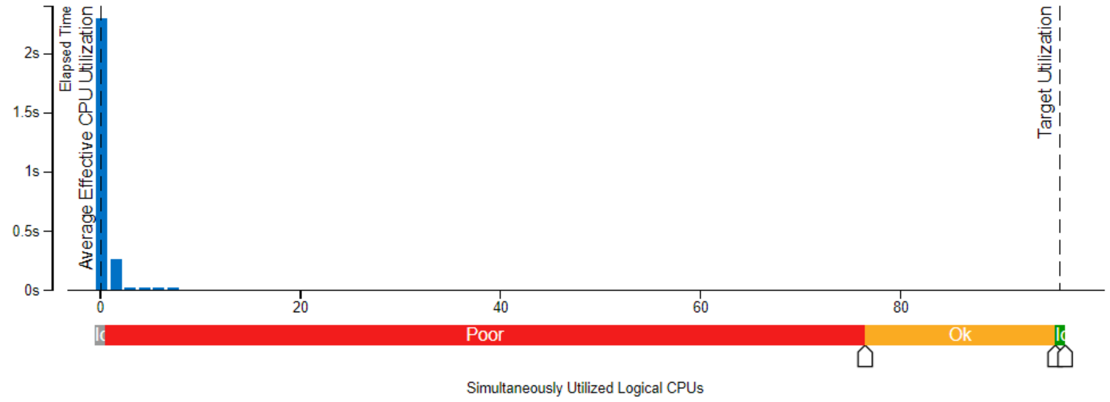
Threading | INTEL VTUNE PROFILER

Analysis Configuration | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Flame Graph | Platform

Effective CPU Utilization: 0.1% (0.126 out of 96 logical CPUs)

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.



Wait Time with poor CPU Utilization: 6.325s (100.0% of Wait Time)

Top Waiting Objects

This section lists the objects that spent the most time waiting in your application. Objects can wait on specific calls, such as sleep() or I/O, or on contended synchronizations. A significant amount of Wait time associated with a synchronization object reflects high contention for that object and, thus, reduced parallelism.

Sync Object	Wait Time with poor CPU Utilization	(% from Object Wait Time)	Wait Count
Futex 0xb2cbe538	6.294s	100.0%	913
Futex 0xbcf65048	0.031s	100.0%	10
Stream 0x59b268b6	0.000s	100.0%	1
Stream /proc/cpuinfo 0x1ca6ac0a	0.000s	100.0%	1
Stream /proc/meminfo 0xc313ae18	0.000s	100.0%	1
[Others]	0.000s	100.0%	3

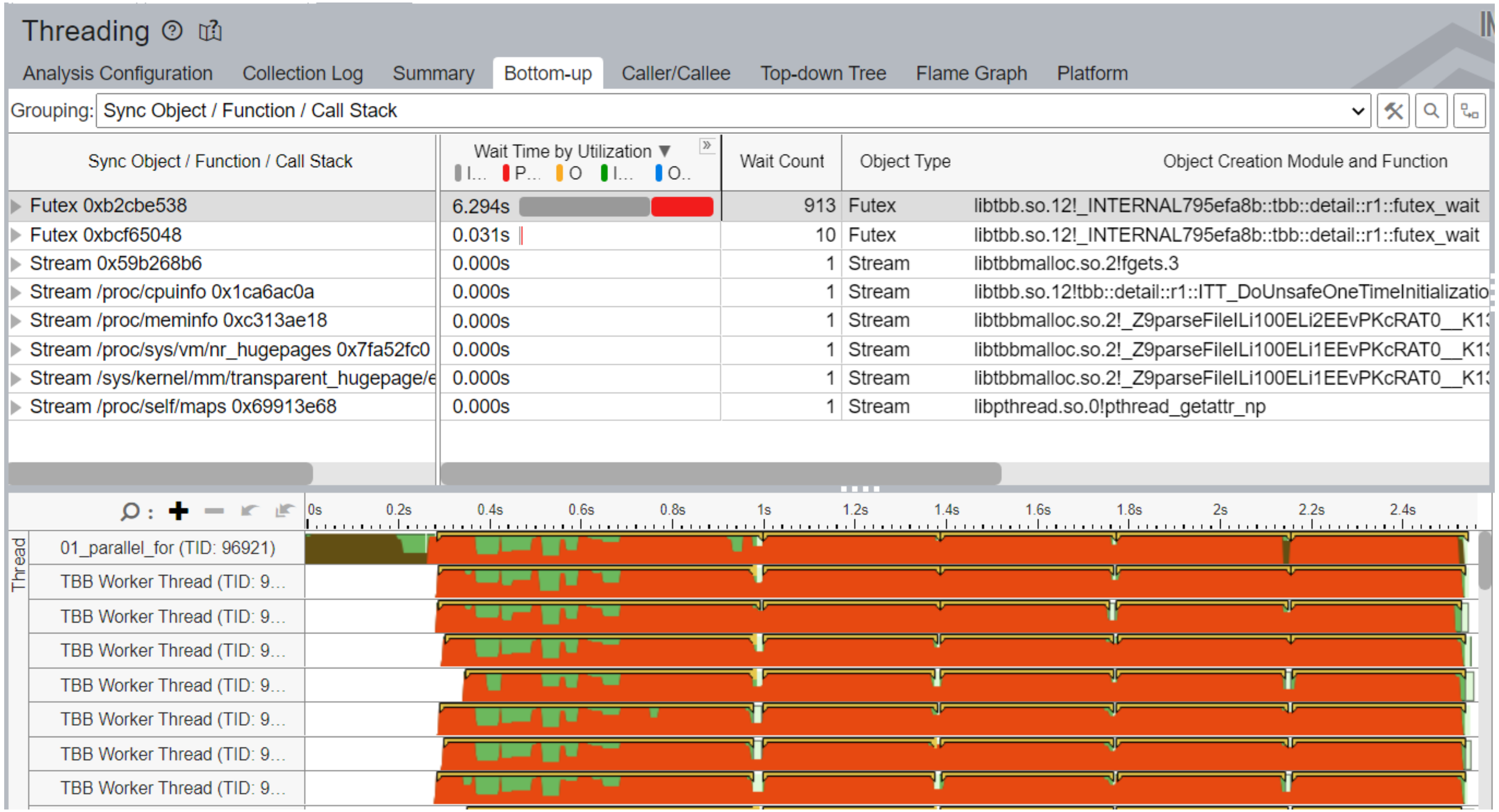
Spin and Overhead Time: 188.376s (99.8% of CPU Time)

Top Functions with Spin or Overhead Time

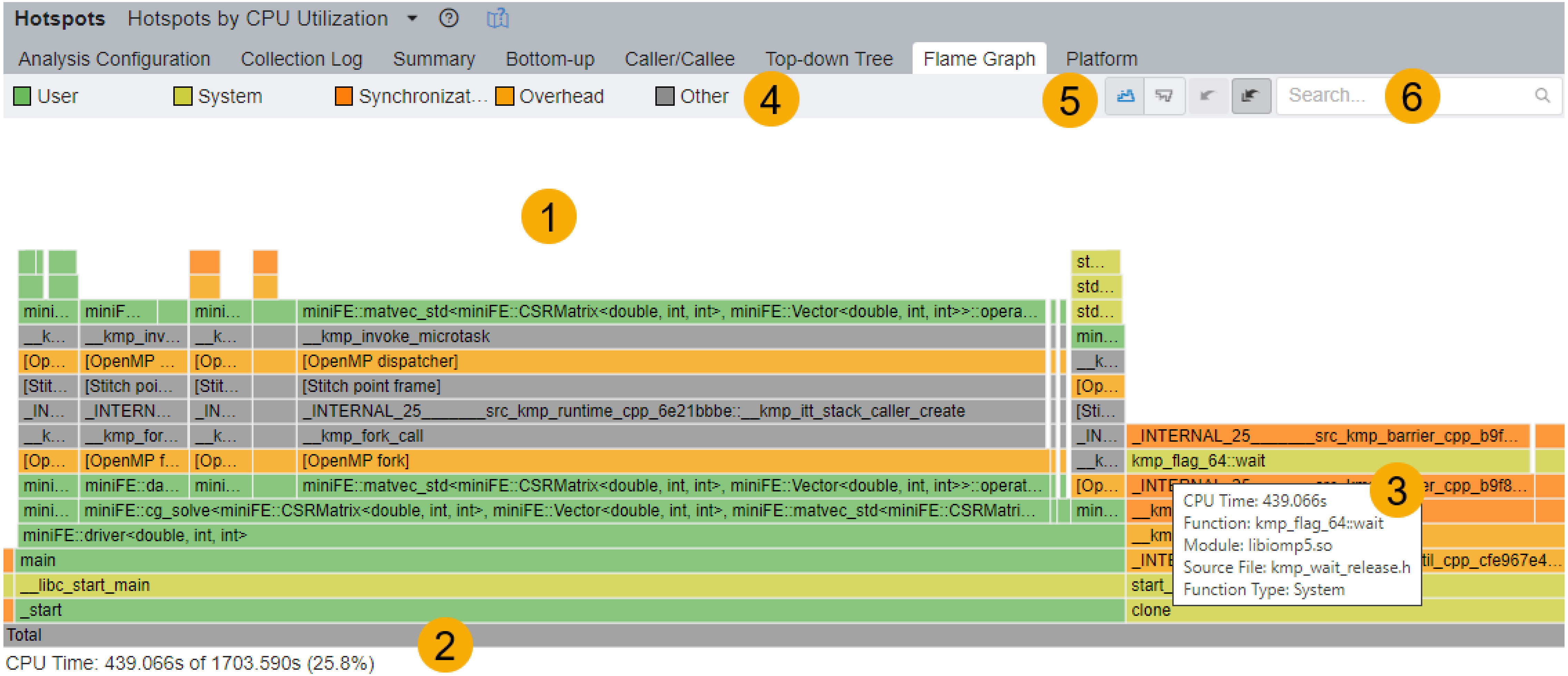
The section lists top functions in your application with the most spin and overhead time.

Function	Module	Spin and Overhead Time	(% from CPU Time)
tbb::detail::d1::start_for<tbb::detail::d1::blocked_range<unsigned long>, gemm<float>(void, tbb::detail::d1::blocked_range<unsigned long>*, tbb::detail::d1::blocked_range<unsigned long> const*, tbb::detail::d1::blocked_range<unsigned long> const*, unsigned long, unsigned long, unsigned long)::{lambda(tbb::detail::d1::blocked_range<unsigned long> const&)#1}, tbb::detail::d1::auto_partitioner const>::execute	01_parallel_for_solution	183.663s	97.3%
tbb::detail::r1::market::create_one_job	libtbb.so.12	4.678s	2.5%
sched_yield	libc.so.6	0.010s	0.0%

What were the TBB internals that keep waits

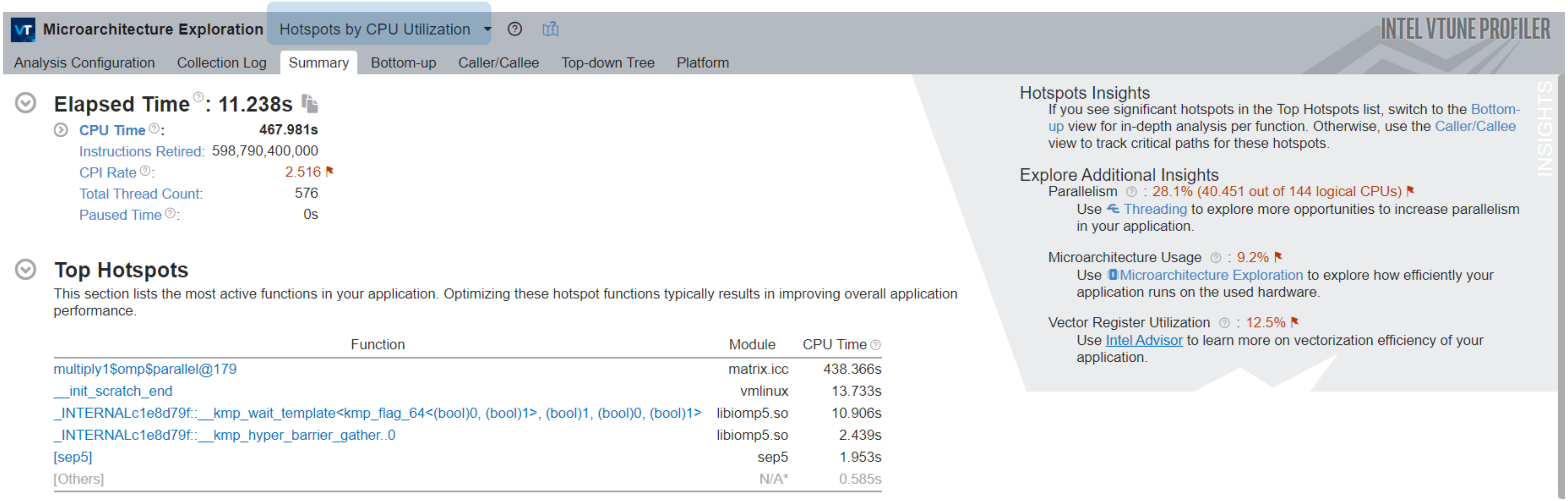


Flame Graph


Hotspots Hotspots by CPU Utilization
Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Flame Graph
Platform
User
System
Synchronizat...
Overhead
Other
Search...
1
2
3
4
5
6
miniFE::matvec_std<miniFE::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int>>::opera...
__kmp_invoke_microtask
[OpenMP dispatcher]
[Stitch point frame]
_INTERNAL_25______src_kmp_runtime_cpp_6e21bbbe::__kmp_itt_stack_caller_create
__kmp_fork_call
[OpenMP fork]
miniFE::matvec_std<miniFE::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int>>::operat...
miniFE::cg_solve<miniFE::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int>, miniFE::matvec_std<miniFE::CSRMatri...
miniFE::driver<double, int, int>
main
__libc_start_main
_start
Total
_INTERNAL_25______src_kmp_barrier_cpp_b9f...
kmp_flag_64::wait
clone
CPU Time: 439.066s
Function: kmp_flag_64::wait
Module: libiomp5.so
Source File: kmp_wait_release.h
Function Type: System
CPU Time: 439.066s of 1703.590s (25.8%)

Hotspots and Optimization Insights

Get additional insights on execution efficiency



Compiler Optimization Options

Use vectorization switches

Linux*, OS X*: -x

- Might enable Intel processor specific optimizations
- Processor-check added to "main" routine: Application errors in case SIMD feature missing or non-Intel processor with appropriate/informative message

    Example: -xCORE-AVX512

Special switch for Linux*, OS X*: -xHost

- Compiler checks SIMD features of current host processor (where built on) and makes use of latest SIMD feature available
- Code only executes on processors with same SIMD feature or later as on build host

gcc options:

-march=*cpu-type*

'icelake-server'

Help Compiler for Better Parallelization

Remove dependencies between loop iterations for vectorization

- Improving unit stride (at fastest index walk)
- Loop interchange – a known trick (leading to an excessive data transfers)
- Give compiler a hint like ivdev pragma when arrays are passed via pointers
- If possible, align data allocation to 64 Bytes (for AVX-512)

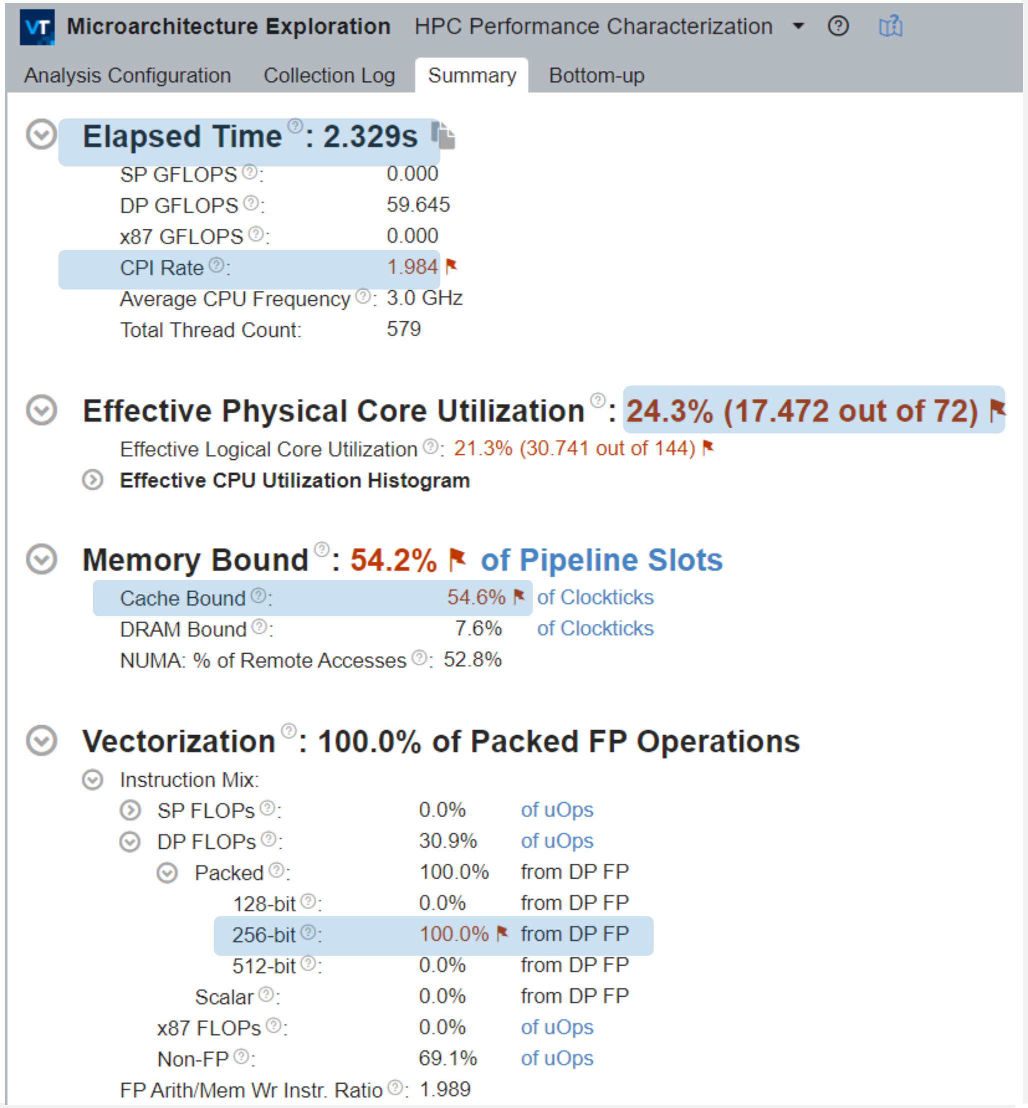
Increase level of parallelization and work balance

- Utilize all CPU physical or logical Cores
- Adjust OMP compactness for OpenMP workloads
- Provide enough data size for workload balancing

First Optimization Results

Implemented loop interchange, data alignment and vectorization options

- Significantly improved Elapsed Time
- CPI Rate is only slightly better
- Still underutilizing Cores
- Cache bound memory accesses
- Vectorization with 256-bit registers, which is AVX only



Compiler Vectorization

Why not AVX-512?

Address ▲	Source Line	Assembly	CPU Time
0x402770		**Block 16:**	
0x402770	66	vmovupdy (%r12,%r10,8), %ymm2	40.037s
0x402776	66	vfmadd213pdy (%r15,%r10,8), %ymm1, %ymm2	8.752s
0x40277c	66	vmovupdy %ymm2, (%r15,%r10,8)	6.690s
0x402782	65	add $0x4, %r10	0.025s
0x402786	65	cmp %rax, %r10	0.003s
0x402789	65	jb 0x402770 <Block 16>	7.972s

Check Compiler's Opt report

-qopt-report=2 -qopt-report-phase=vec

```
    LOOP BEGIN at ../src/multiply.c(65,4)
       remark #15300: LOOP WAS VECTORIZED
       remark #26013: Compiler has chosen to target XMM/YMM vector.
Try using -qopt-zmm-usage=high to override
    LOOP END
```

Force Compiler for ZMM vector?

-qopt-zmm-usage=high

Check Opt report again

```
    LOOP BEGIN at ../src/multiply.c(65,4)
       remark #15300: LOOP WAS VECTORIZED
    LOOP END
```

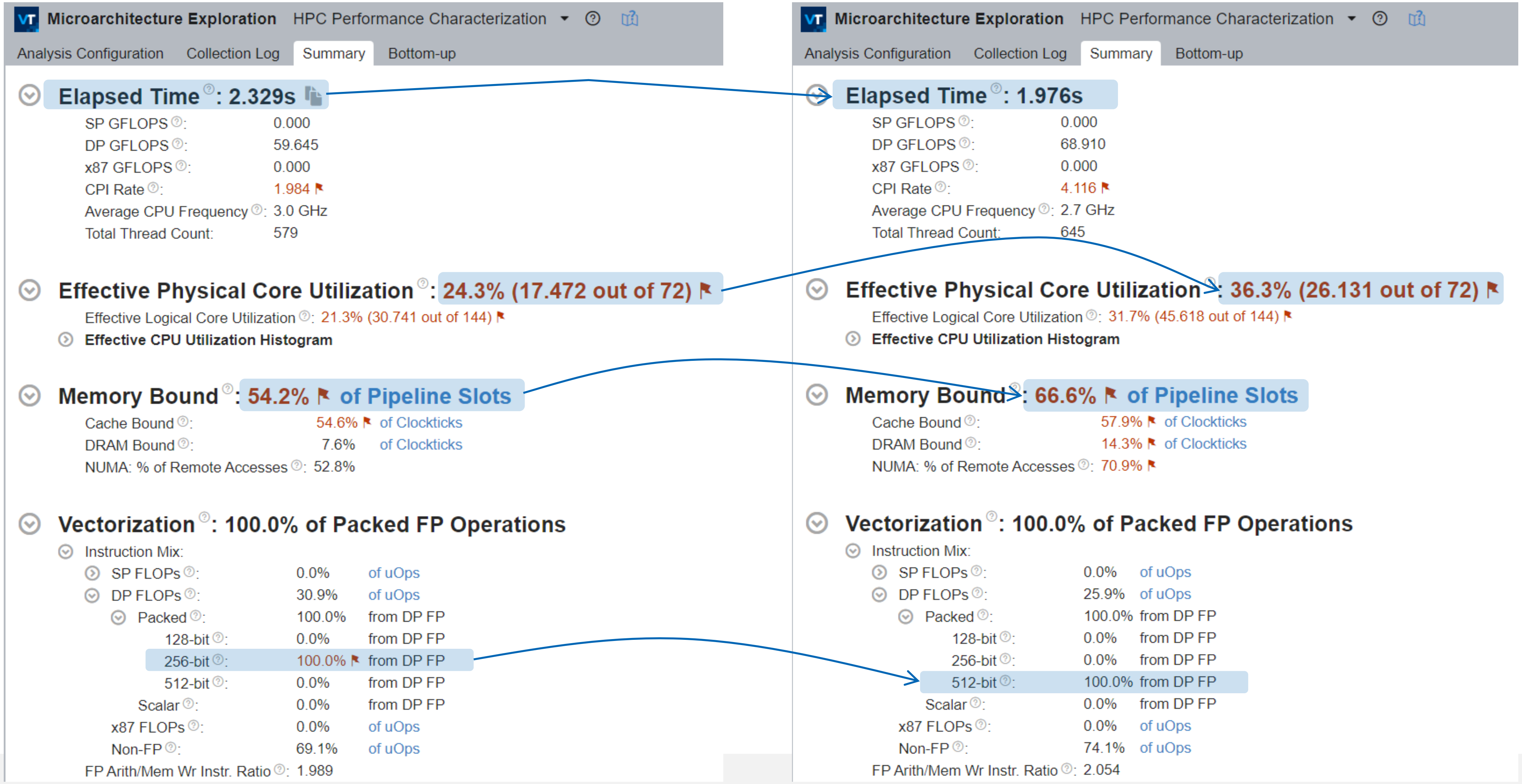
Collect VTune profile and compare

Address ▲	Source Line	Assembly	CPU Time
0x402871		**Block 16:**	
0x402871	66	vmovupsz (%r12,%rbx,8), %zmm4	63.988s
0x402878	66	vfmadd213pdz (%r9,%rbx,8), %zmm3, %zmm4	23.717s
0x40287f	66	vmovupdz %zmm4, (%r9,%rbx,8)	3.327s
0x402886	65	add $0x8, %rbx	0.026s
0x40288a	65	cmp %rax, %rbx	0.004s
0x40288d	65	jb 0x402871 <Block 16>	3.583s

Why Performance Results are not much better?

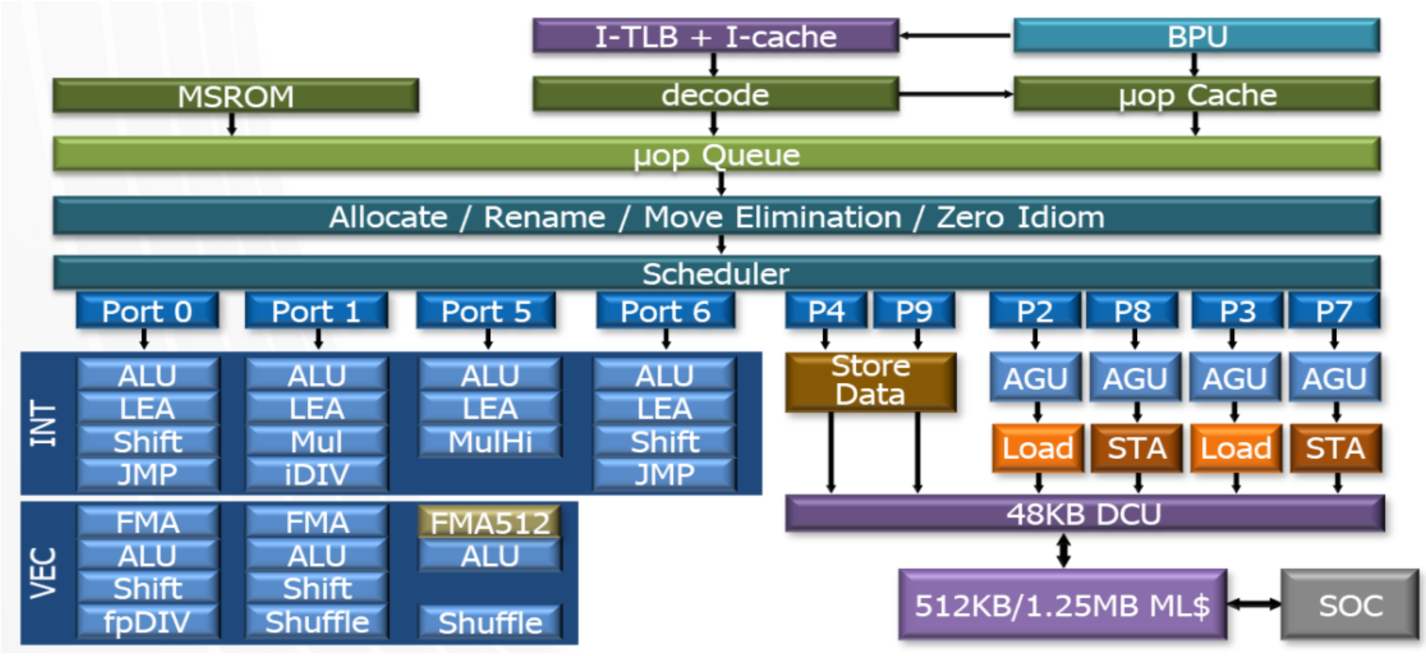
xmm/ymm registers

zmm registers



Top-Down Method for Performance Analysis

Ice Lake Core Microarchitecture

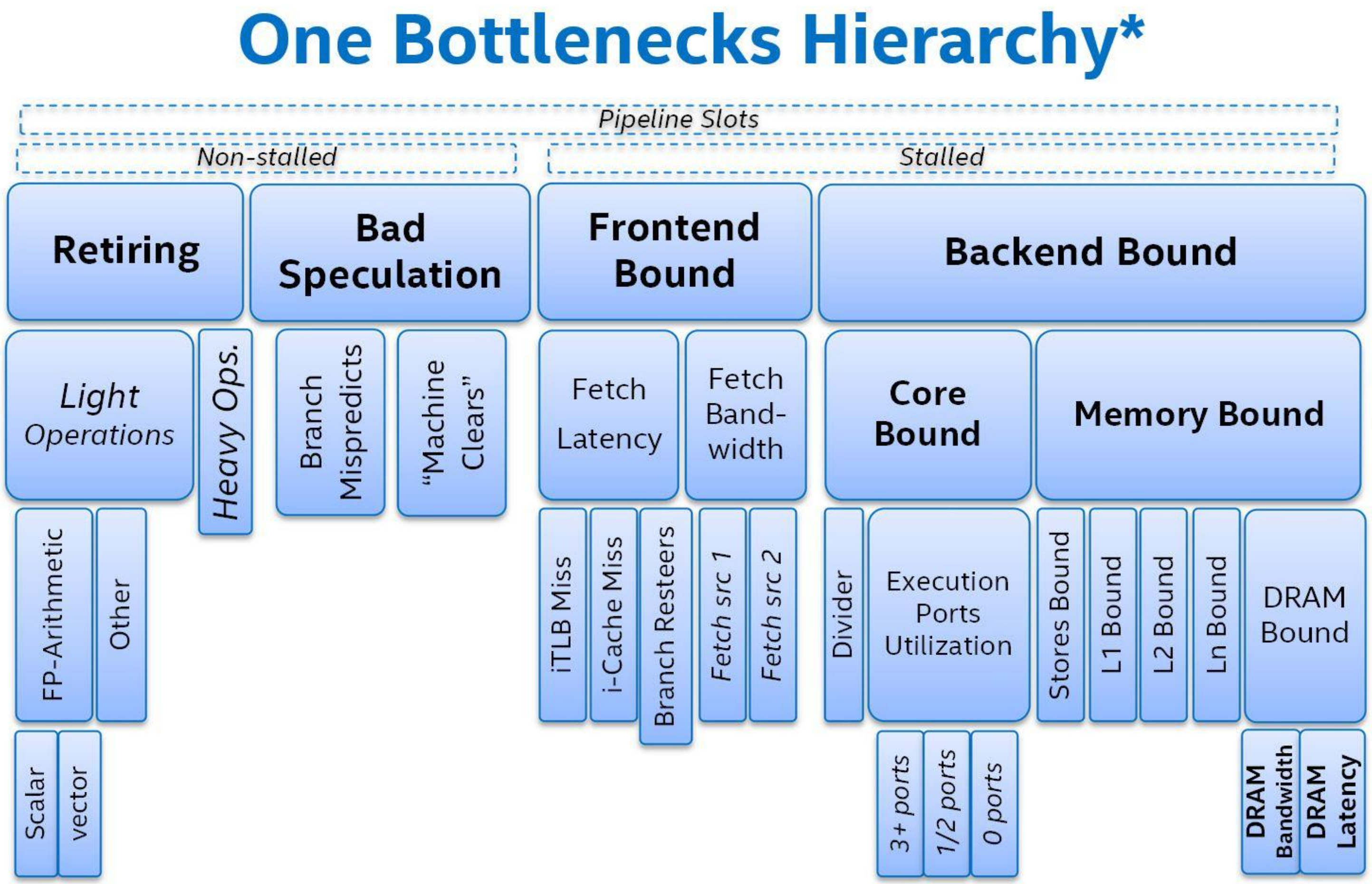


	Cascade Lake (per core)	Ice Lake (per core)
Out-of-order Window	224	384
In-flight Loads + Stores	72 + 56	128 + 72
Scheduler Entries	97	160
Register Files – Integer + FP	180 + 168	280 +224
Allocation Queue	64/thread	70/thread; 140/1 thread
L1D Cache (KB)	32	48
L1D BW (B/Cyc) – Load + Store	128 + 64	128 + 64
L2 Unified TLB	1.5K	2K
Mid-level Cache (MB)	1	1.25

- Wider and deeper machine: wider allocation and execution resources + larger structures
  Improved Front-end: higher capacity and improved branch predictor
- Enhancements in TLBs, single thread execution, prefetching
- Server enhancements – larger Mid-level Cache (L2) + second FMA

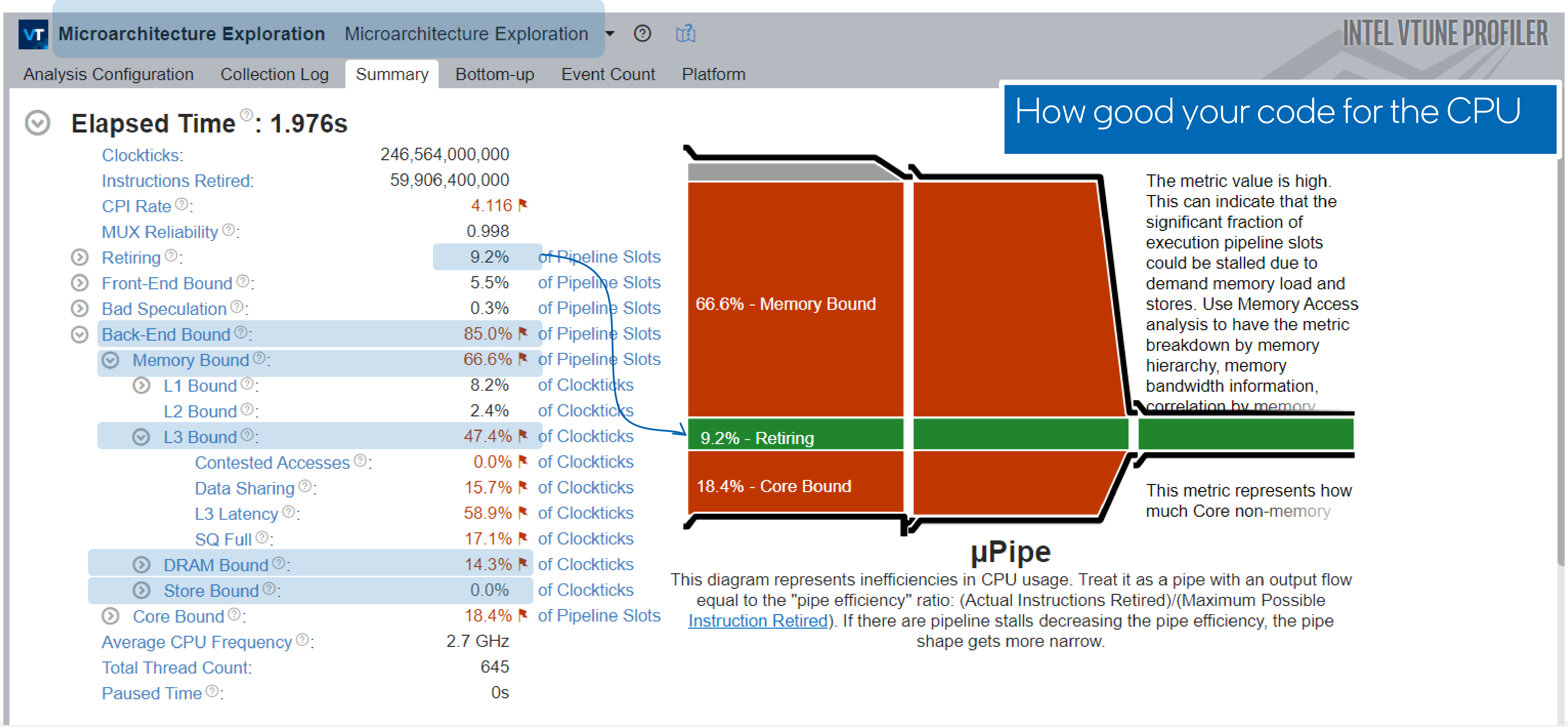
Increased instruction level parallelism

Top-Down Method for Performance Analysis



*Reference paper: A. Yasin, "A Top-Down Method for Performance Analysis and Counters Architecture", ISPASS 2014

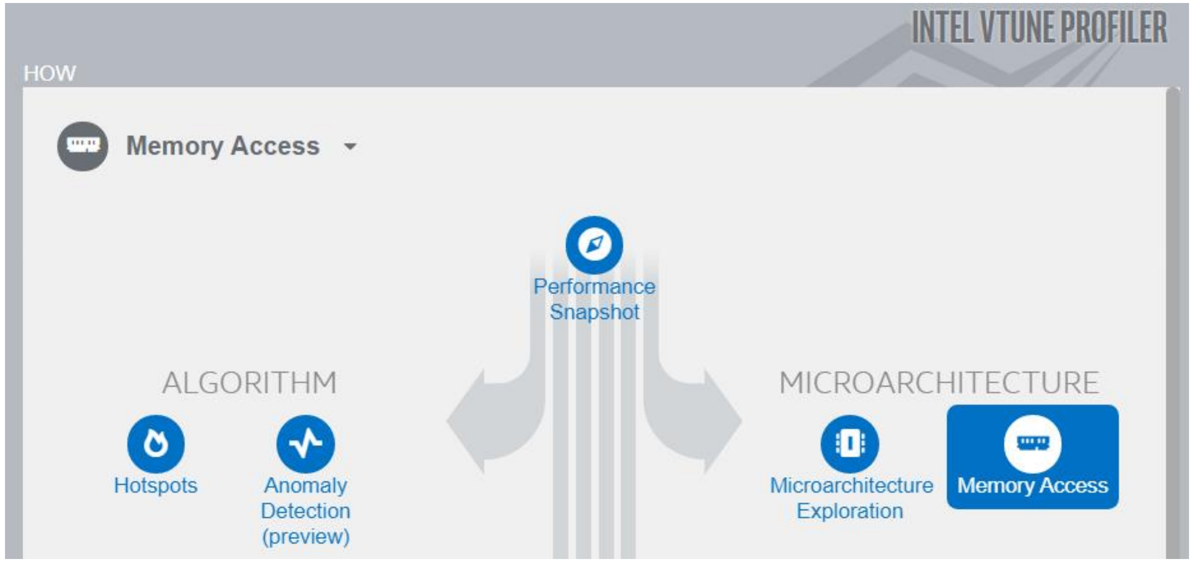
Microarchitecture Exploration



Intel® VTune™ Profiler Memory Access Analysis

VTune™ Profiler Memory Access

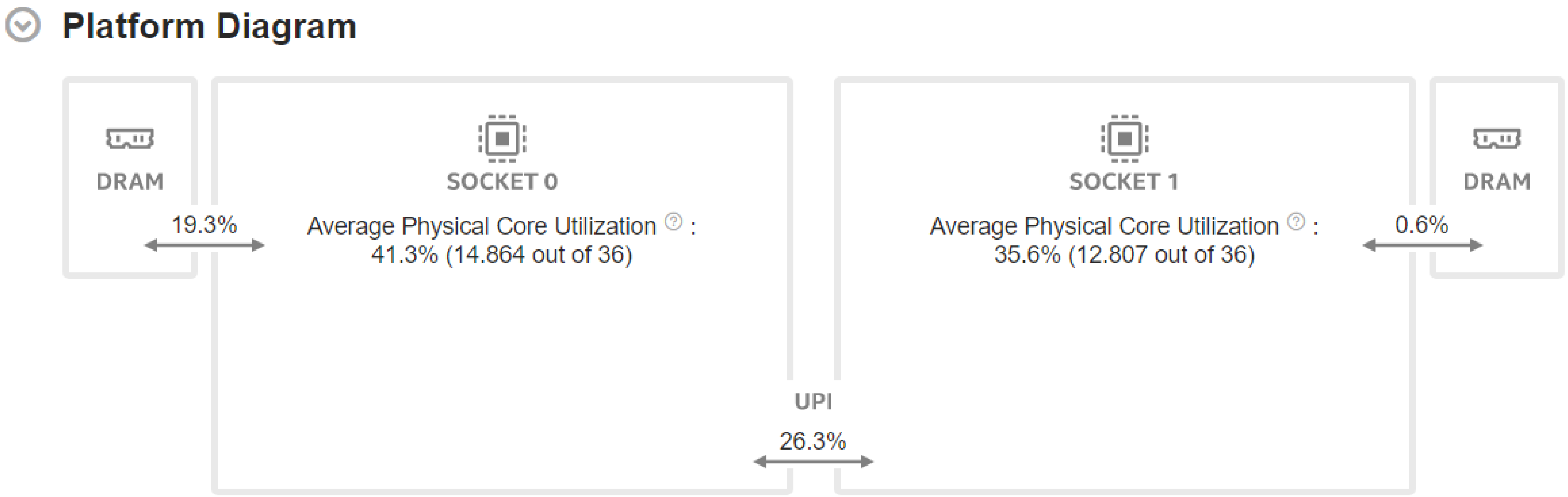
Both problems: Memory Latency and Memory Bandwidth are estimated



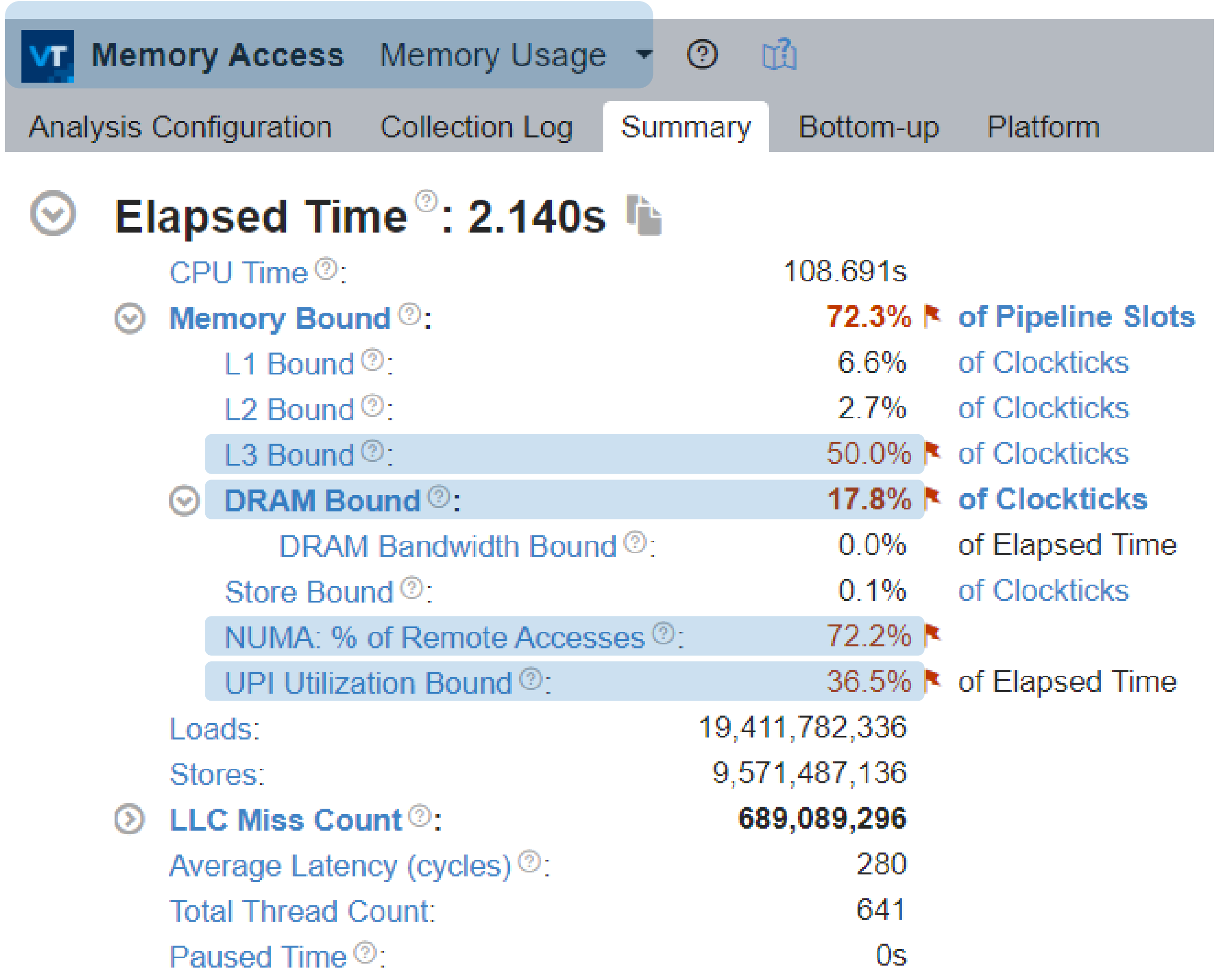
- Latency problem estimation against Code and Memory Objects, and the Memory Level involved
- Bandwidth measurements are system wide (no code attribution, but Time stamps attribution)
- Like with any other type of analysis, investigate the Summary for your application and then focus on smaller range: functions or loops

Platform Diagram

A high-level view on data flow in a system



Memory Issues Hierarchy

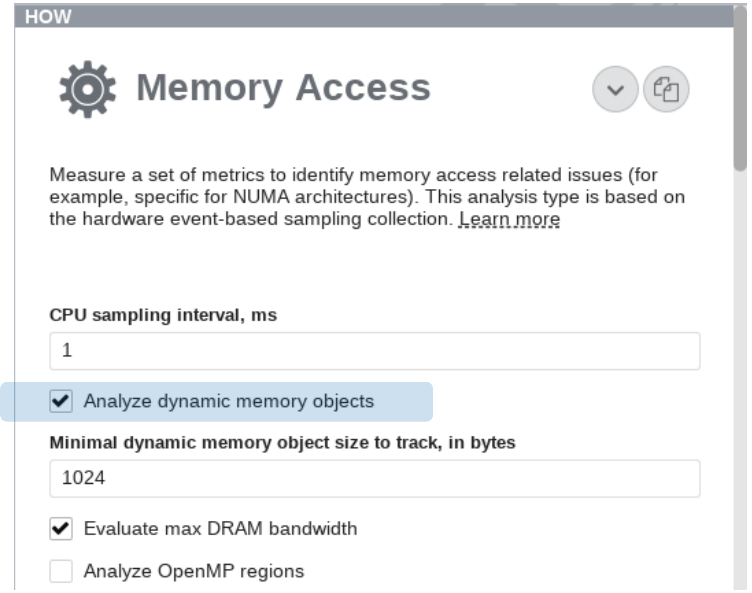


Discover inefficient data access

- Finding which memory level is providing data with highest latency
- How much in % the problem is affecting performance
- If it's DRAM, then quickly identify if there is a NUMA problem (unbalanced access to remote DRAM)
- Go to the functions level (Bottom-Up Tab) for more details of responsible code
- Collect Memory Objects for more details on responsible data

Memory Objects

Instrument memory allocation with a single option



- Finding allocation place for memory objects that are "responsible" for CPU stalls
- Memory objects are identified by allocation source line and call stack
- Enables allocations instrumentation (off by default) and threshold for objects size (1024 by default)
- Don't use it when not needed (due to overhead)

Memory Objects Decomposition

Grouping: Memory Object / Function / Allocation Stack

Memory Object / Function…	Loads	Stores	LLC Miss Count	Average Latency (cycles)
[Unknown]	42,001,260	12,600,378	0	2
matrix.c:126 (128 MB)	11,200,336	18,565,956,962	0	0
matrix.c:121 (128 MB)	19,659,389,764	0	8,998,429,846	805
matrix.c:116 (128 MB)	17,520,125,588	0	70,004,900	46

Load operations

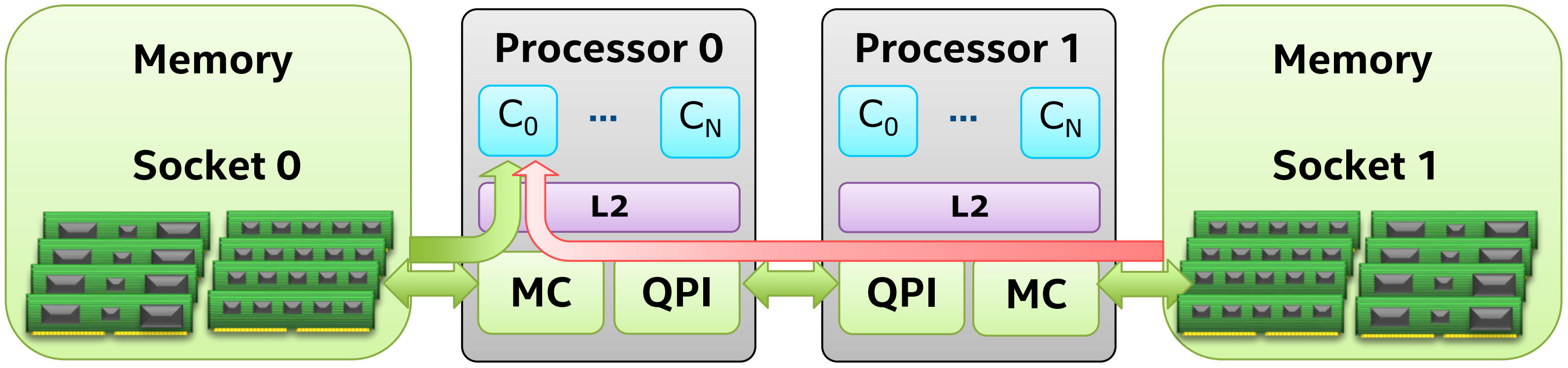
Store operations

Most of LLC Miss

Biggest latencies

- Helps when you do not know data layout
- Helps when the same code line operates at many arrays
- “Memory Object” Grouping helps to find significant data objects first

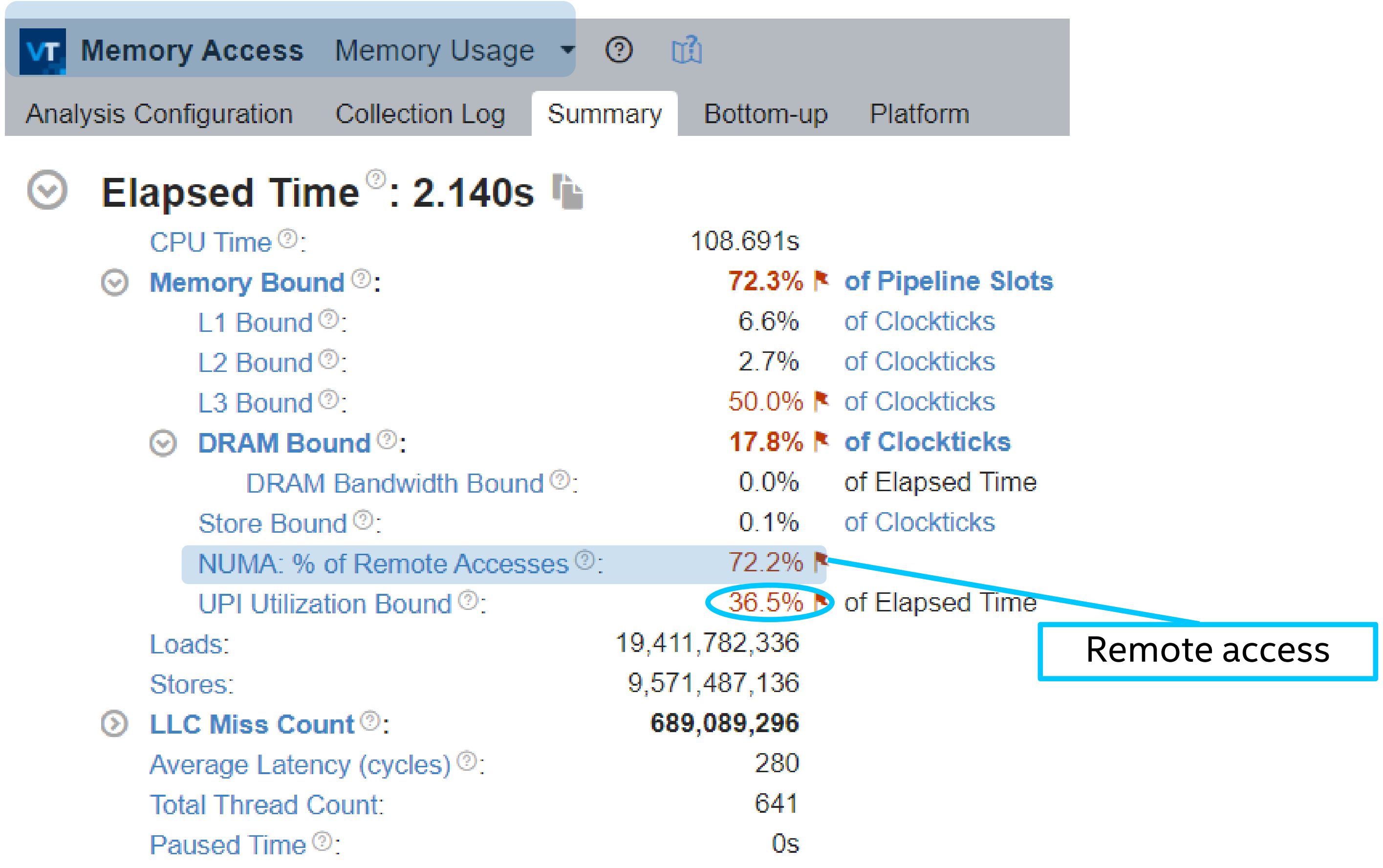
Dual Socket (-EP) System and NUMA Effects



The shortest data path

The longest data path

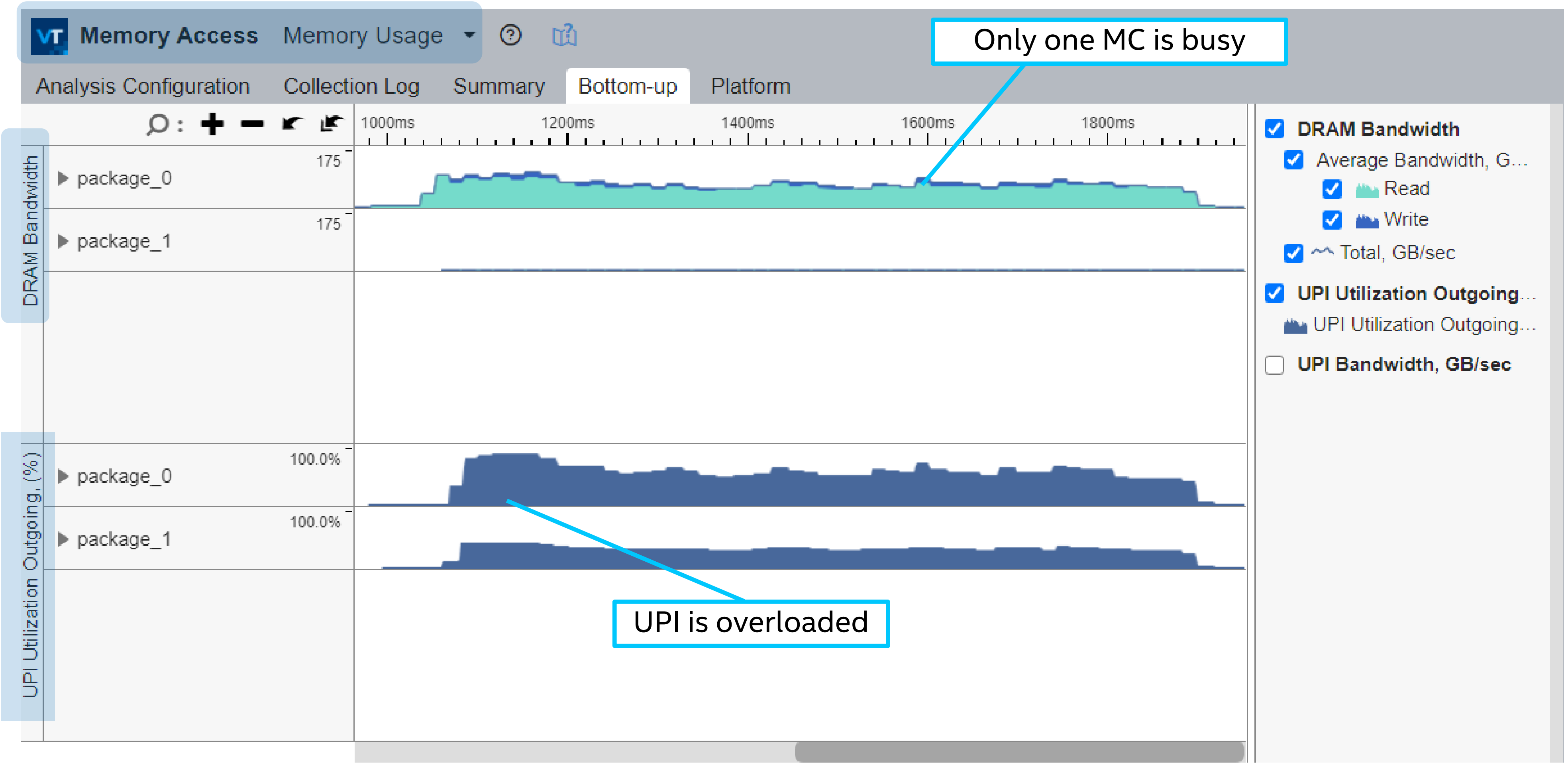
Measuring Remote Memory Access



- Remote DRAM access has biggest effect
- Remote memory access **latency** ~1.7x greater than local memory
- Local memory **bandwidth** can be up to ~2x greater than remote
- Remote Cache access bares its penalty as well

Refer to: Local and Remote Memory: Memory in a Linux/NUMA System

Visualizing NUMA Problems



Refer to: [Optimizing Applications for NUMA]

Summary

- VTune Profiler is a comprehensive set that contains tools covering all aspects of platforms performance, from system wide and long app runs down to small execution kernels and microarchitecture specific on all intel platforms
- VTune Profiler helps to address algorithmic, multithreading, microarchitecture and memory issues
- VTune Profiler infrastructure provides flexible means for data collection, analysis, storage and team collaboration

Quick References

Intel® VTune™ Profiler – Performance Profiler

- Product page – overview, features, FAQs…
- Training materials – Cookbooks, User Guide, Processor Tuning Guides
- Support Forum
- Online Service Center - Secure Priority Support
- What's New?

Additional Analysis Tools

- Intel® Advisor – Design and optimize for efficient vectorization, threading, memory usage, and accelerator offload. Roofline and flow graph analysis.
- Intel® Inspector – memory and thread checker/ debugger
- Intel® Trace Analyzer and Collector - MPI Analyzer and Profiler

Additional Development Products

- Intel® Software Development Products

Notices & Disclaimers

Intel technologies may require enabled hardware, software or service activation. Learn more at intel.com or from the OEM or retailer.

Your costs and results may vary.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

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. https://software.intel.com/en-us/articles/optimization-notice

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.

Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. See backup for configuration details. For more complete information about performance and benchmark results, visit www.intel.com/benchmarks.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.

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.