Intel® VTune™ Amplifier

Performance Profiler

Faster, Scalable Code, Faster

Intel® VTune™ Amplifier Performance Profiler

Accurate Data - Low Overhead

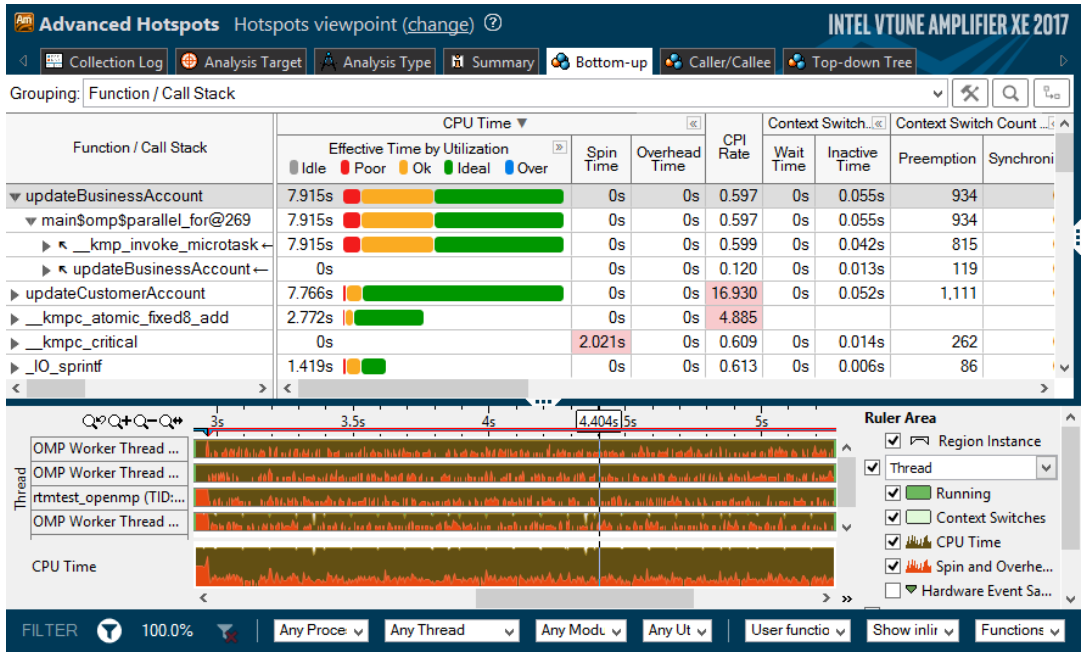
- CPU, GPU, FPU, threading, bandwidth…

Meaningful Analysis

- Threading, OpenMP region efficiency
- Memory access, storage device

Easy

- Data displayed on the source code
- Easy set-up, no special compiles



"Last week, Intel® VTune™ Amplifier helped us find almost 3X performance improvement. This week it helped us improve the performance another 3X."

Claire Cates
Principal Developer
SAS Institute Inc.

http://intel.ly/vtune-amplifier-xe

Two Great Ways to Collect Data

Intel® VTune™ Amplifier

Software Collector	Hardware Collector
Uses OS interrupts	Uses the on chip Performance Monitoring Unit (PMU)
Collects from a single process tree	Collect system wide or from a single process tree.
~10ms default resolution	~1ms default resolution (finer granularity - finds small functions)
Either an Intel® or a compatible processor	Requires a genuine Intel® processor for collection
Call stacks show calling sequence	Optionally collect call stacks
Works in virtual environments	Works in a VM only when supported by the VM (e.g., vSphere*, KVM)
No driver required	Requires a driver - Easy to install on Windows - Linux requires root (or use default perf driver)

No special recompiles - C, C++, C#, Fortran, Java, Assembly

A Rich Set of Performance Data

Intel® VTune™ Amplifier

Software Collector	Hardware Collector
Basic Hotspots Which functions use the most time?	**Advanced Hotspots** Which functions use the most time? Where to inline? – Statistical call counts
Concurrency Tune parallelism. Colors show number of cores used.	**General Exploration** Where is the biggest opportunity? Cache misses? Branch mispredictions?
Locks and Waits Tune the #1 cause of slow threaded performance: – waiting with idle cores.	**Advanced Analysis** Memory-access, HPC Characterization, etc…
Any IA86 processor, any VM, no driver	Higher res., lower overhead, system wide

No special recompiles - C, C++, C#, Fortran, Java, Assembly

Find Answers Fast

Intel® VTune™ Amplifier

Adjust Data Grouping

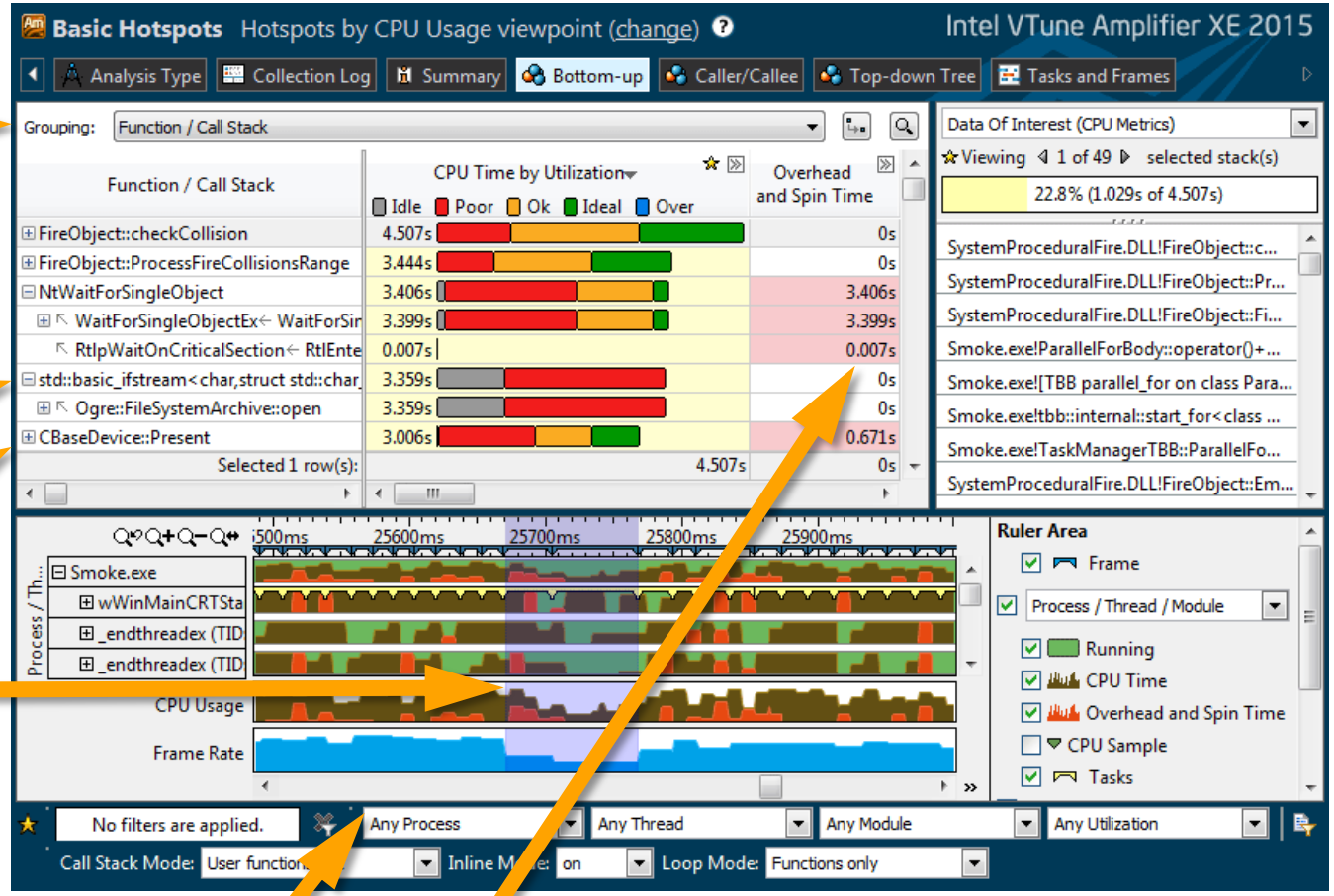
Function - Call Stack
Module - Function - Call Stack
Source File - Function - Call Stack
Thread - Function - Call Stack
… (Partial list shown)

Double Click Function to View Source

Click [+] for Call Stack

Filter by Timeline Selection (or by Grid Selection)

Zoom In And Filter On Selection
Filter In by Selection
Remove All Filters



Filter by Process & Other Controls

Tuning Opportunities Shown in Pink. Hover for Tips

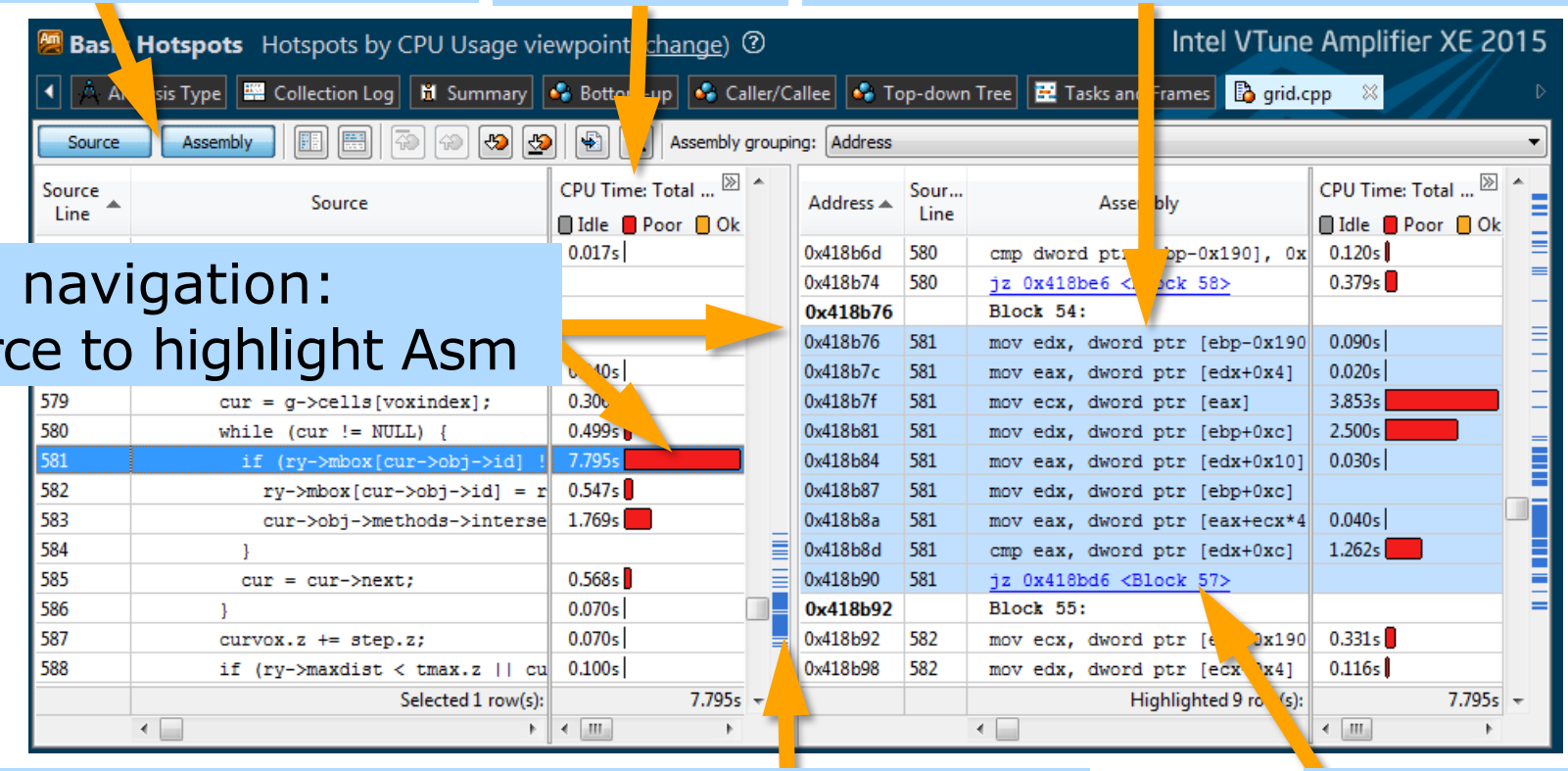
See Profile Data On Source / Asm

Double Click from Grid or Timeline

View Source / Asm or both

CPU Time

Right click for instruction reference manual



Quick Asm navigation:
Select source to highlight Asm

Scroll Bar “Heat Map” is an overview of hot spots

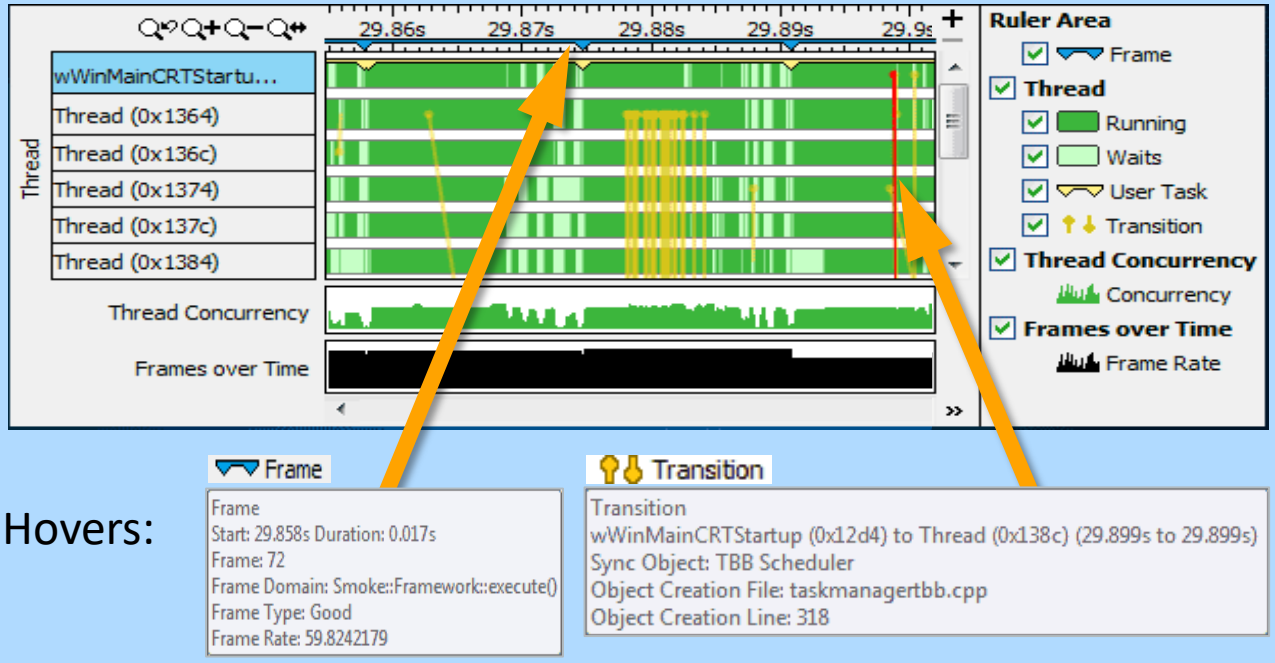
Click jump to scroll Asm

Timeline Visualizes Thread Behavior

Intel® VTune™ Amplifier

Transitions

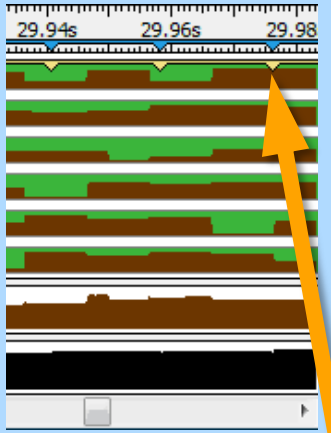
Locks & Waits



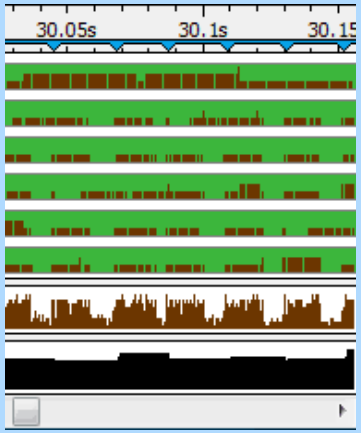
Hovers:

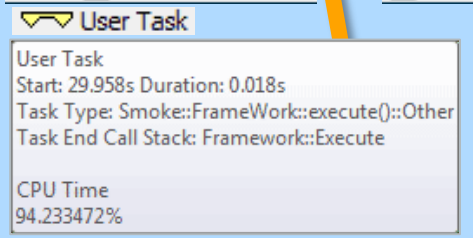
CPU Time

Basic Hotspots



Advanced Hotspots





Optional: Use API to mark frames and user tasks

Optional: Add a mark during collection

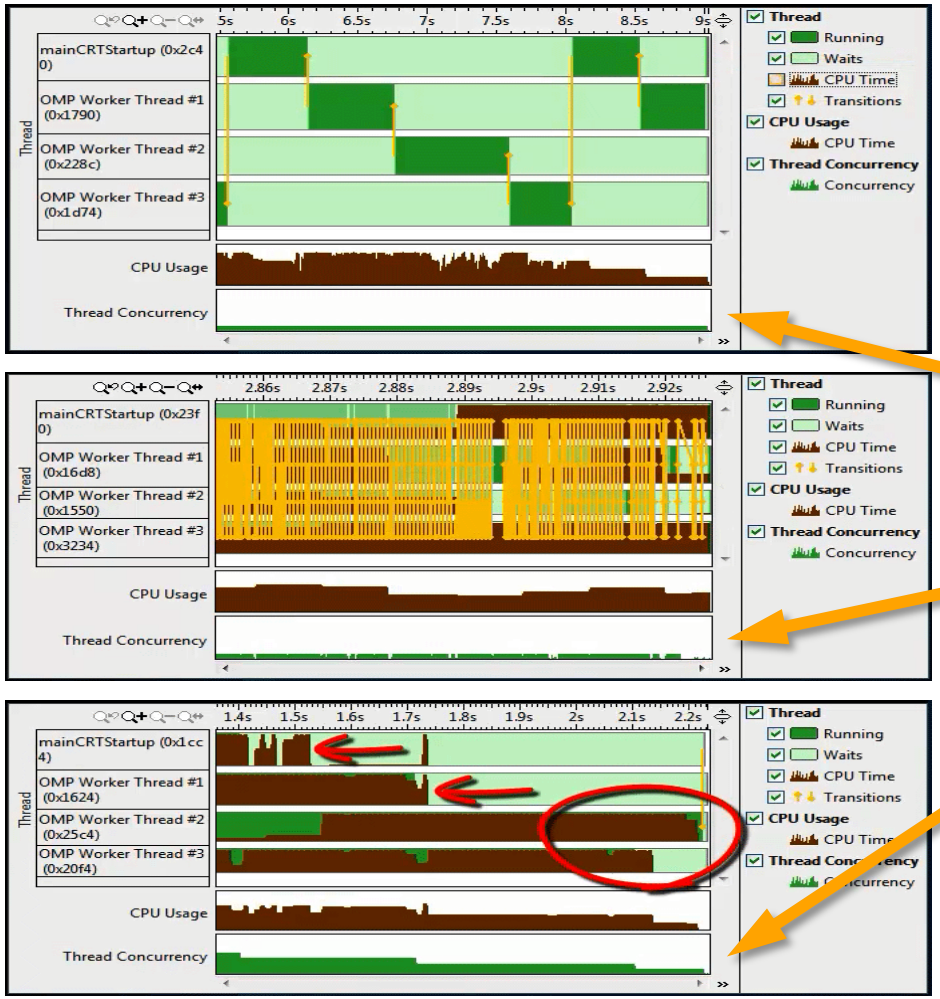
Visualize Parallel Performance Issues

Look for Common Patterns

Coarse Grain Locks

High Lock Contention

Load Imbalance



Low Concurrency

Command Line Interface

Automate analysis

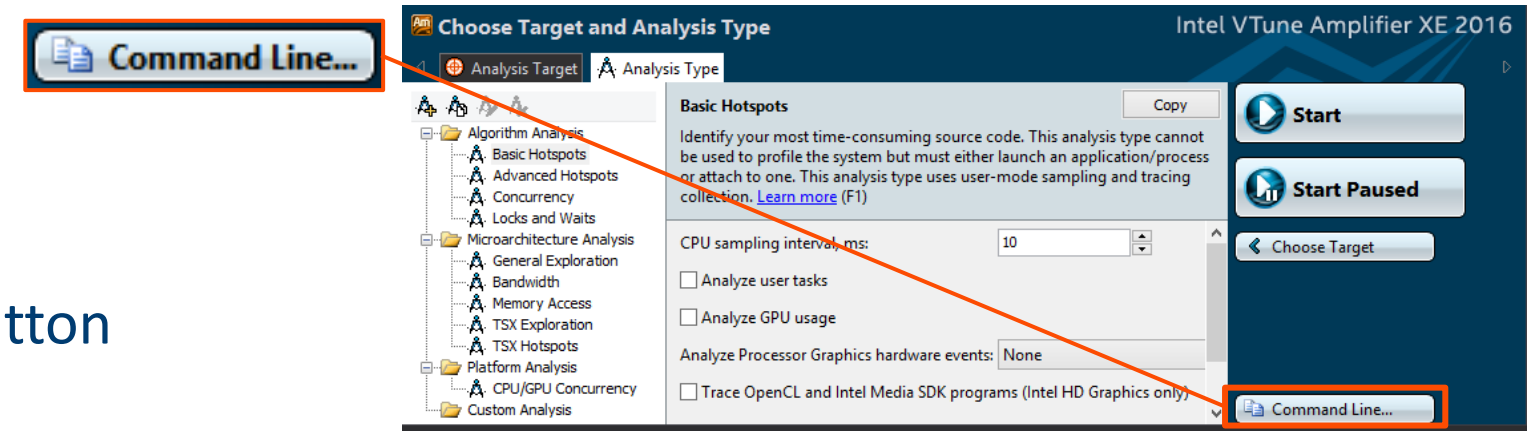
amplxe-cl is the command line:

- **Windows:** `C:\Program Files (x86)\Intel\VTune Amplifier XE \bin[32|64]\amplxe-cl.exe`
- **Linux:** `/opt/intel/vtune_amplifier_xe/bin[32|64]/amplxe-cl`

Help: `amplxe-cl -help`

Use UI to setup

1) Configure analysis in UI
2) Press “Command Line…” button
3) Copy & paste command



Great for regression analysis – send results file to developer
Command line results can also be opened in the UI

MPI Analysis

Command line:

```
> mpirun -n 16 –ppn 4 –l amplxe-cl -collect advanced-hotspots -
trace-mpi -result-dir my_result -- my_app.a
```

Or use gtool:

```
> mpirun -gtool "amplxe-cl -collect memory-access -result-dir
my_result:7,5" my_app.a
```

Each process data is presented for each node they were running on:

```
my_result.host_name1 (rank 0-3)
my_result.host_name2 (rank 4-7)
my_result.host_name3 (rank 8-11)
my_result.host_name4 (rank 12-15)
```

Advanced Hotspots Hotspots viewpoint (change)

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform

Elapsed Time: 120.101s

- CPU Time: **7085.857s**
  - Effective Time: **2227.628s**
  - Spin Time: **4589.471s**

    A significant portion of CPU time is spent waiting. Use this metric to discover which synchronizations are spinning. Consider adjusting spin wait parameters, changing the lock implementation (for example, by backing off then descheduling), or adjusting the synchronization granularity.
  - Overhead Time: **268.758s**

Instructions Retired: 3,333,315,033,755
CPI Rate: 2.757
Wait Rate: 191.043
CPU Frequency Ratio: 1.000

- Context Switch Time: **609.769s**

Total Thread Count: 139
Paused Time: 60.056s

OpenMP Analysis. Collection Time: 60.046

Serial Time (outside any parallel region): 43.097s (71.8%)

Serial Time of your application is high. It directly impacts application Elapsed Time and scalability. Explore options for parallelization, algorithm or microarchitecture tuning of the serial part of the application.

- **Parallel Region Time: 16.949s (28.2%)**

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 assu

OpenMP Region	OpenMP Potential Gain	(%)	OpenMP Region Time
allocGauge_Dompparallel:128@/panfs/projects/milc_wg/ruizi/QPHIX/mbench/ks_long_dslash.cpp:231:234	0.340s	0.6%	1.033s
qphix_ks_dslash_Dompparallel:128@/panfs/projects/milc_wg/ruizi/QPHIX/mbench/ks_long_dslash.cpp:541:627	0.301s	0.5%	3.573s
allocGauge18_Dompparallel:128@/panfs/projects/milc_wg/ruizi/QPHIX/mbench/ks_long_dslash.cpp:231:234	0.278s	0.5%	0.894s
ks_congrad_parity_qphix_Dompparallel:128@/panfs/projects/milc_wg/ruizi/milc_qcd_1/milc_qcd/ks_imp_rhmc/../generic_ks/d_congrad5_fn_qphix_P.c:188:191	0.159s	0.3%	0.713s
ks_congrad_parity_qphix_Dompparallel:128@/panfs/projects/milc_wg/ruizi/milc_qcd_1/milc_qcd/ks_imp_rhmc/../generic_ks/d_congrad5_fn_qphix_P.c:240:243	0.144s	0.2%	0.741s
[Others]	N/A*	N/A*	9.996s

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

Advanced Hotspots Hotspots viewpoint (change) Intel VTune Amplifier XE

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform

Top Hotspots

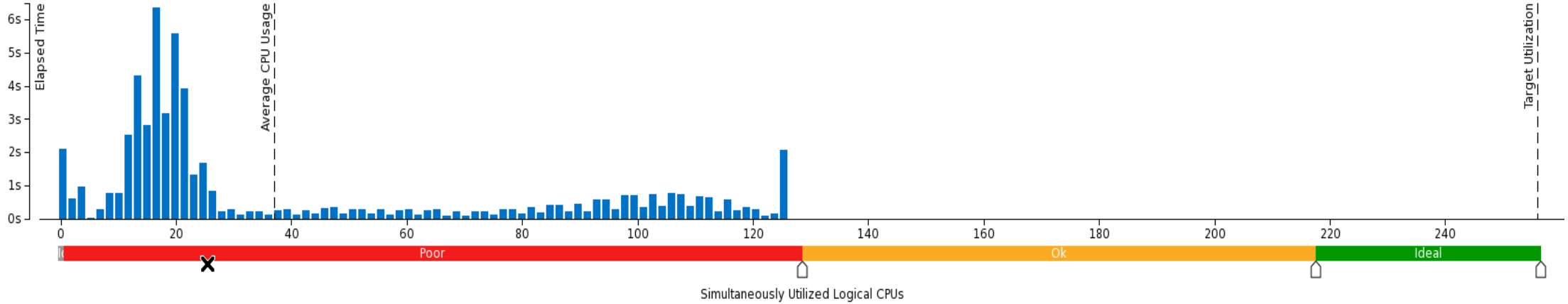
This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
__kmp_hyper_barrier_release	libiomp5.so	3791.856s
[vmlinux]	vmlinux	671.755s
__kmp_wait_template<kmp_flag_64>	libiomp5.so	425.421s
ks_long_dslash_vec_noinline	su3_rhmc_hisq_d_knl_mpi	360.775s
__kmp_release_template<kmp_flag_64>	libiomp5.so	263.734s
[Others]	N/A*	1572.316s

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

CPU Usage 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 usage value.



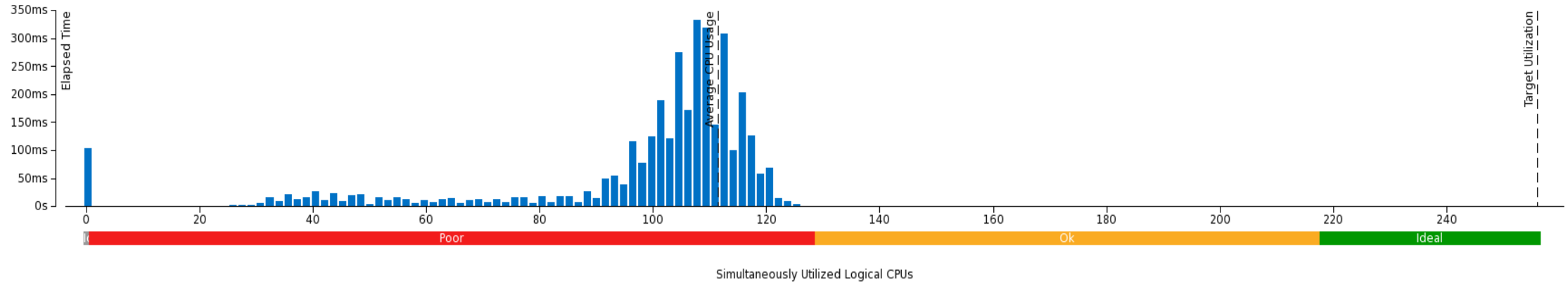
Advanced Hotspots Hotspots viewpoint (change) ⓘ Intel VTune Amplifier XE 20

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform

OpenMP Region CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously in an OpenMP region. Spin and Overhead time adds to the Idle CPU usage value. OpenMP regions in the drop-down list are sorted by Potential Gain (Elapsed Time) so it is recommended to start exploration from the top.

OpenMP Region: qphix_ks_dslash_Dompparallel:128@/panfs/projects/milc_wg/ruizi/QPHIX/mbench/ks_long_dslash.cpp:541:627

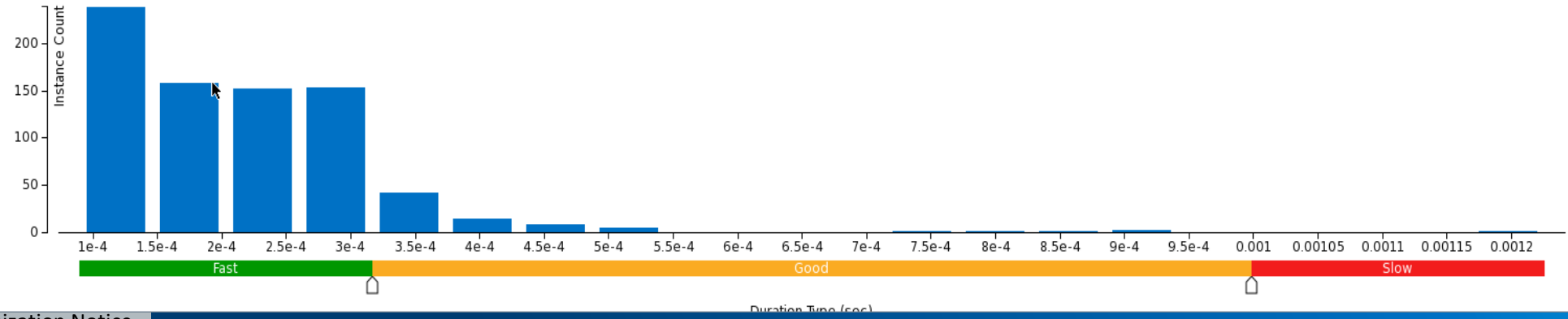


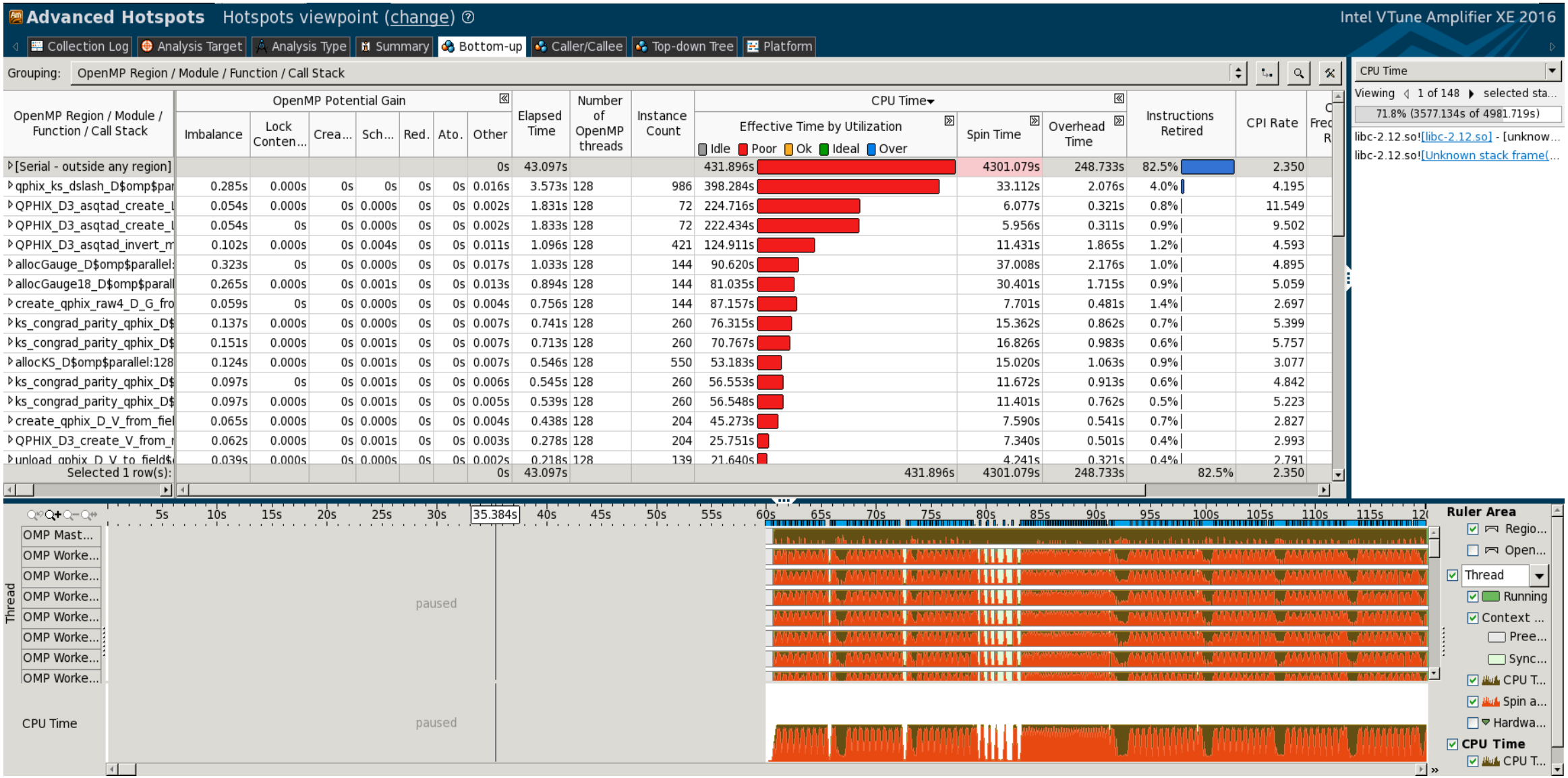
OpenMP Region Duration Histogram

This histogram shows the total number of region instances in your application executed with a specific duration. High number of slow instances may signal a performance bottleneck. Explore the data provided in the Bottom-up, Top-down Tree, and Timeline panes to identify code regions with the slow duration.

8/18/2016 11:15 AM - ScreenClipping

OpenMP Region: declare_strided_gatherompparallel:128@/panfs/projects/milc_wg/ruizi/milc_qcd_1/milc_qcd/ks_imp_rhmc/../generic/com_mpi.c:1755:1757





Memory Access Analysis

Tune data structures for performance

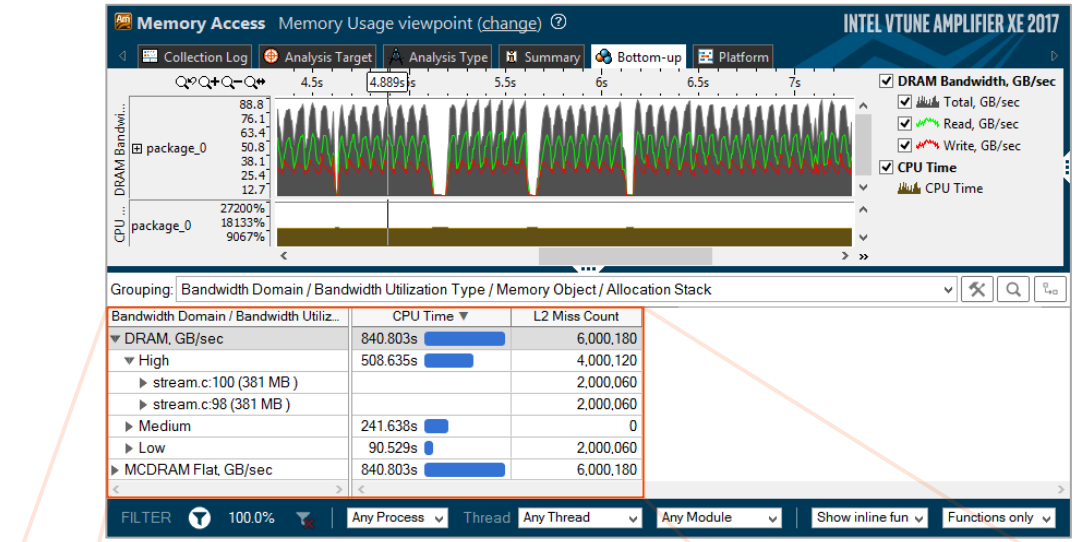
- Attribute cache misses to data structures (not just the code causing the miss)
- Support for custom memory allocators

Optimize NUMA latency & scalability

- True & false sharing optimization
- Auto detect max system bandwidth
- Easier tuning of inter-socket bandwidth

Easier install, Latest processors

- No special drivers required on Linux*
- Intel® Xeon Phi™ processor MCDRAM (high bandwidth memory) analysis



Bandwidth Domain / Bandwidth Utiliz...	CPU Time ▼	L2 Miss Count
DRAM, GB/sec	840.803s	6,000,180
High	508.635s	4,000,120
stream.c:100 (381 MB)		2,000,060
stream.c:98 (381 MB)		2,000,060
Medium	241.638s	0
Low	90.529s	2,000,060
MCDRAM Flat, GB/sec	840.803s	6,000,180

Memory Access Memory Usage viewpoint (change)

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Platform

Elapsed Time: 3.821s

CPU Time:	282.064s	
Memory Bound:	**84.6%**	**of Pipeline Slots**
L1 Bound:	30.5%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	27.1%	of Clockticks
DRAM Bound:	**15.0%**	**of Clockticks**
DRAM Bandwidth Bound:	0.0%	of Elapsed Time
Memory Latency:		
Remote / Local memory Ratio:	0.000	
Local DRAM:	0.0%	of Clockticks
Remote DRAM:	0.0%	of Clockticks
Remote Cache:	24.4%	of Clockticks
Loads:	43,544,206,287	
Stores:	5,444,263,323	
LLC Miss Count:	**95,405,724**	
Average Latency (cycles):	39	
Total Thread Count:	116	
Paused Time:	0s	

Top Memory Objects by Latency

This section lists memory objects that introduced the highest latency to the overall application execution.

Memory Object	Total Latency	Loads	Stores	LLC Miss Count
[Unknown]	56.4%	26,823,504,681	2,909,787,291	59,403,564
linear_regression_pthread.c:136 (7 KB)	40.4%	6,193,985,814	2,146,564,395	36,002,160
[Stack]	1.7%	5,949,178,470	91,802,754	0
lreg-pthread!main (54 MB)	1.1%	4,015,020,447	0	0
[vmlinux]	0.3%	502,215,066	296,108,883	0
[Others]	0.0%	60,301,809	0	0

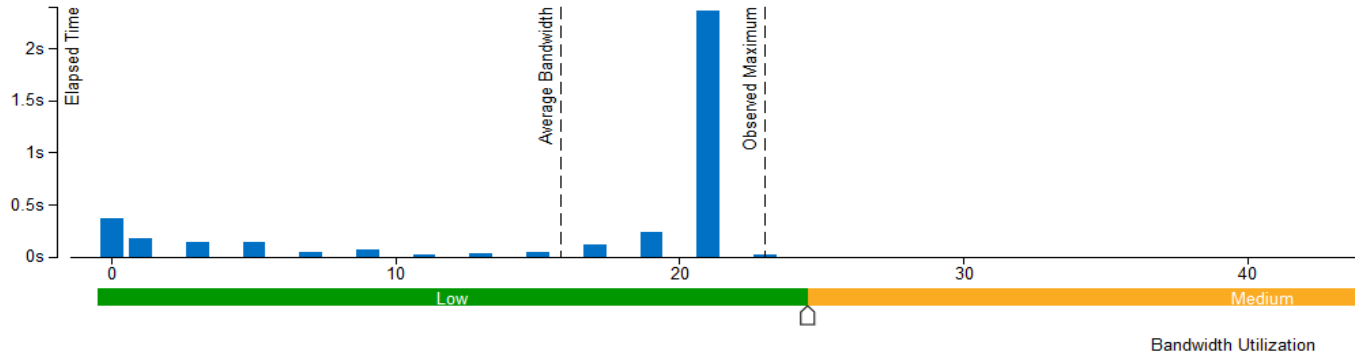
Bandwidth Utilization Histogram

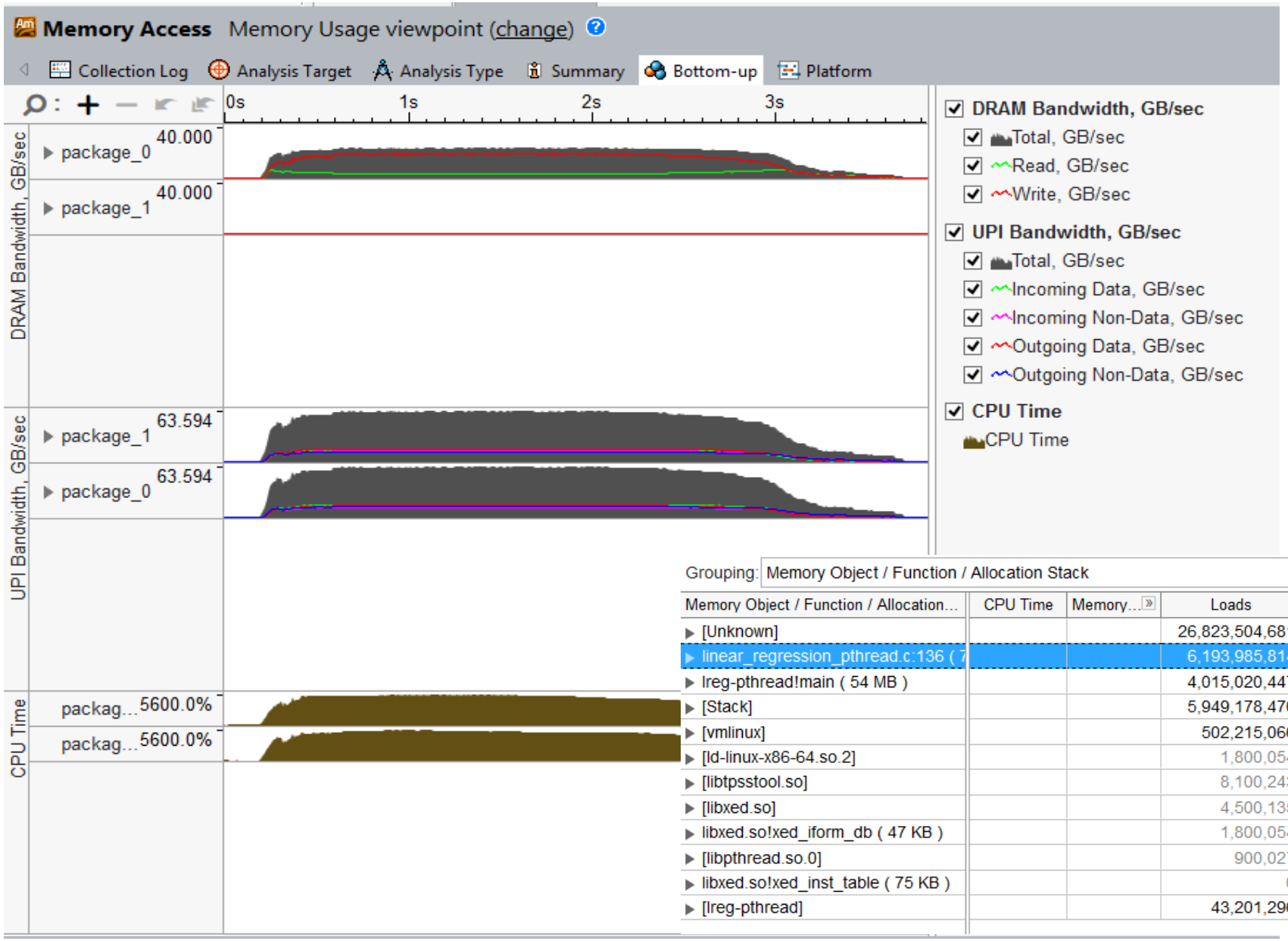
Explore bandwidth utilization over time using the histogram and identify memory objects or functions with maximum contribution to the high bandwidth utilization.

Bandwidth Domain: DRAM, GB/sec

Bandwidth Utilization Histogram

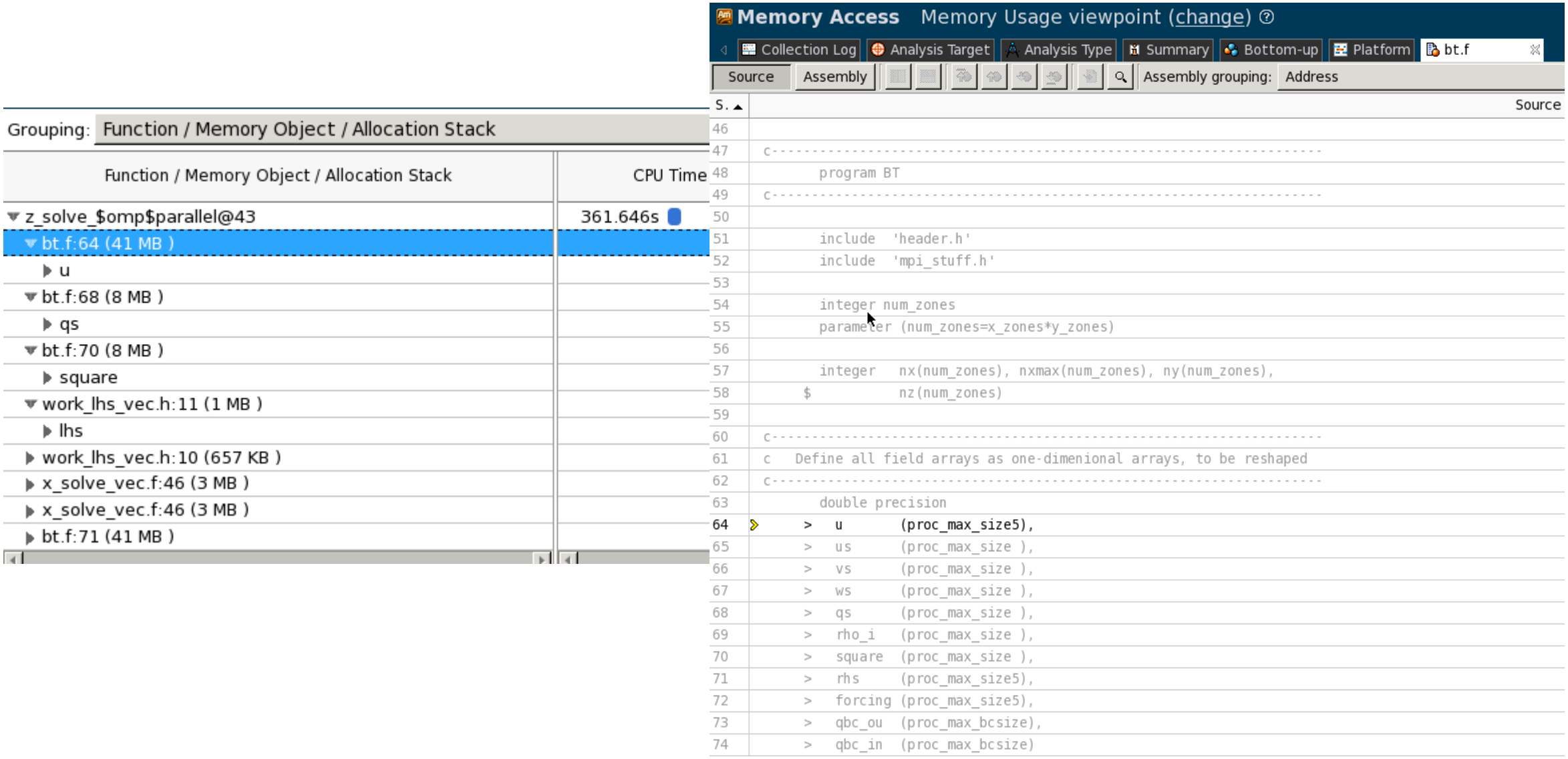
This histogram displays the wall time the bandwidth was utilized by certain value. Use sliders at the bottom of the histogram to define thresholds for Low, Medium an functions executed during a particular utilization type. To learn bandwidth capabilities, refer to your system specifications or run appropriate benchmarks to measure





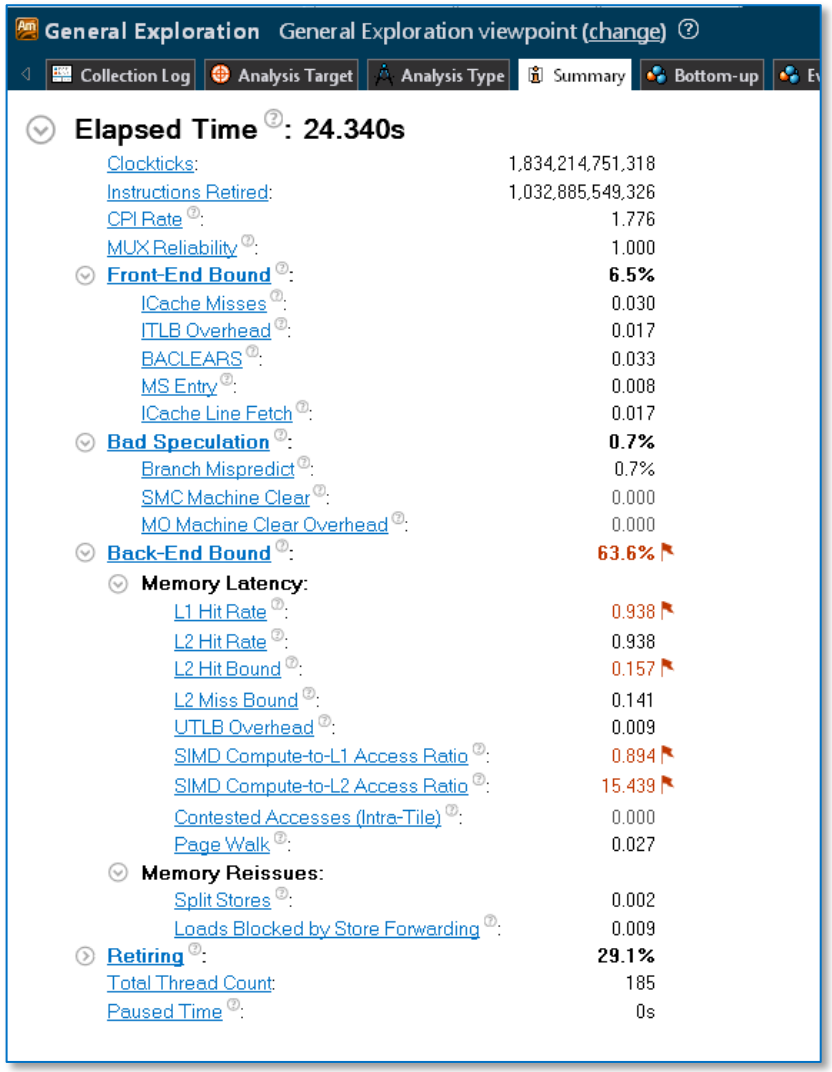
Grouping: Memory Object / Function / Allocation Stack

Memory Object / Function / Allocation...	CPU Time	Memory...	Loads	Stores	LLC Miss...	Average Latency (cycles)
[Unknown]			26,823,504,681	2,909,787,291	62.3%	29
linear_regression_pthread.c:136 (7			6,193,985,814	2,146,564,395	37.7%	147
lreg-pthread!main (54 MB)			4,015,020,447	0	0.0%	8
[Stack]			5,949,178,470	91,802,754	0.0%	8
[vmlinux]			502,215,066	296,108,883	0.0%	10
[ld-linux-x86-64.so.2]			1,800,054	0	0.0%	35
[libtpsstool.so]			8,100,243	0	0.0%	7
[libxed.so]			4,500,135	0	0.0%	0
libxed.so!xed_iform_db (47 KB)			1,800,054	0	0.0%	0
[libpthread.so.0]			900,027	0	0.0%	0
libxed.so!xed_inst_table (75 KB)			0	0	0.0%	6
[lreg-pthread]			43,201,296	0	0.0%	0



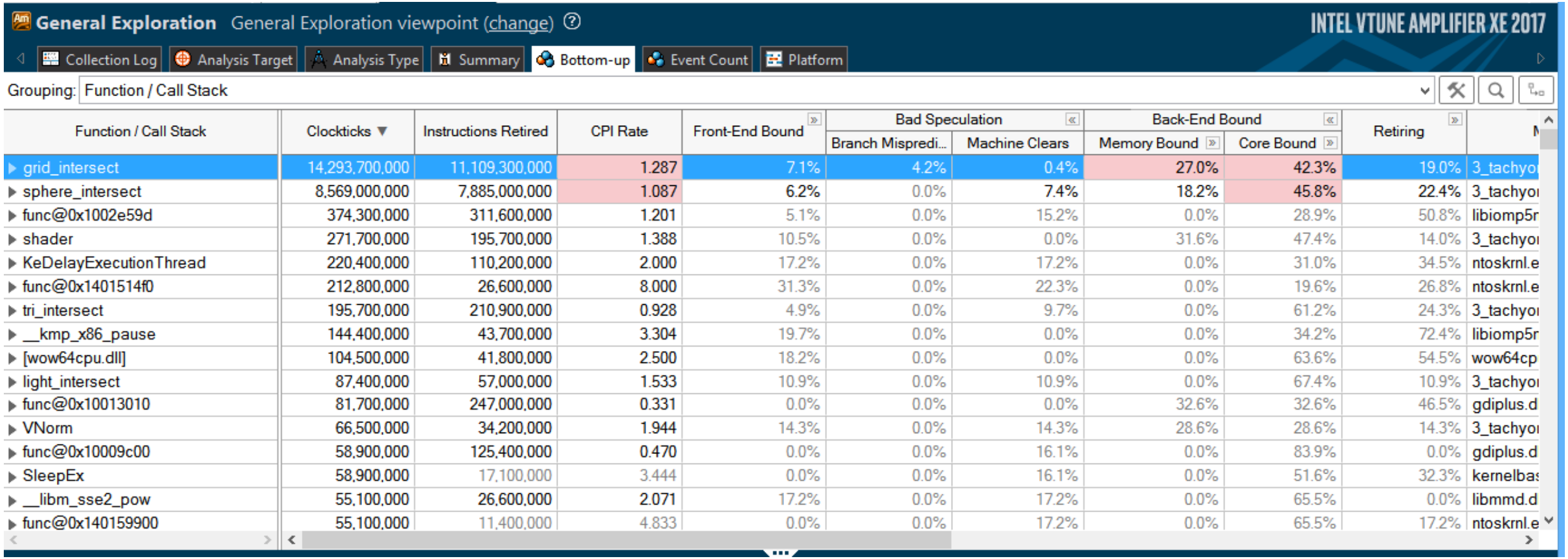
General Exploration Analysis

- Predefined Analysis Type that collects different types of CPU performance events
- Good for first look at whether any CPU event categories are affecting performance
- GUI highlights those events and functions that have performance problems
- Execution pipeline slots distribution by Retiring, Front-End, Back-End, Bad Speculation
- Second level metrics for each aspect of execution pipeline to understand the reason of stalls



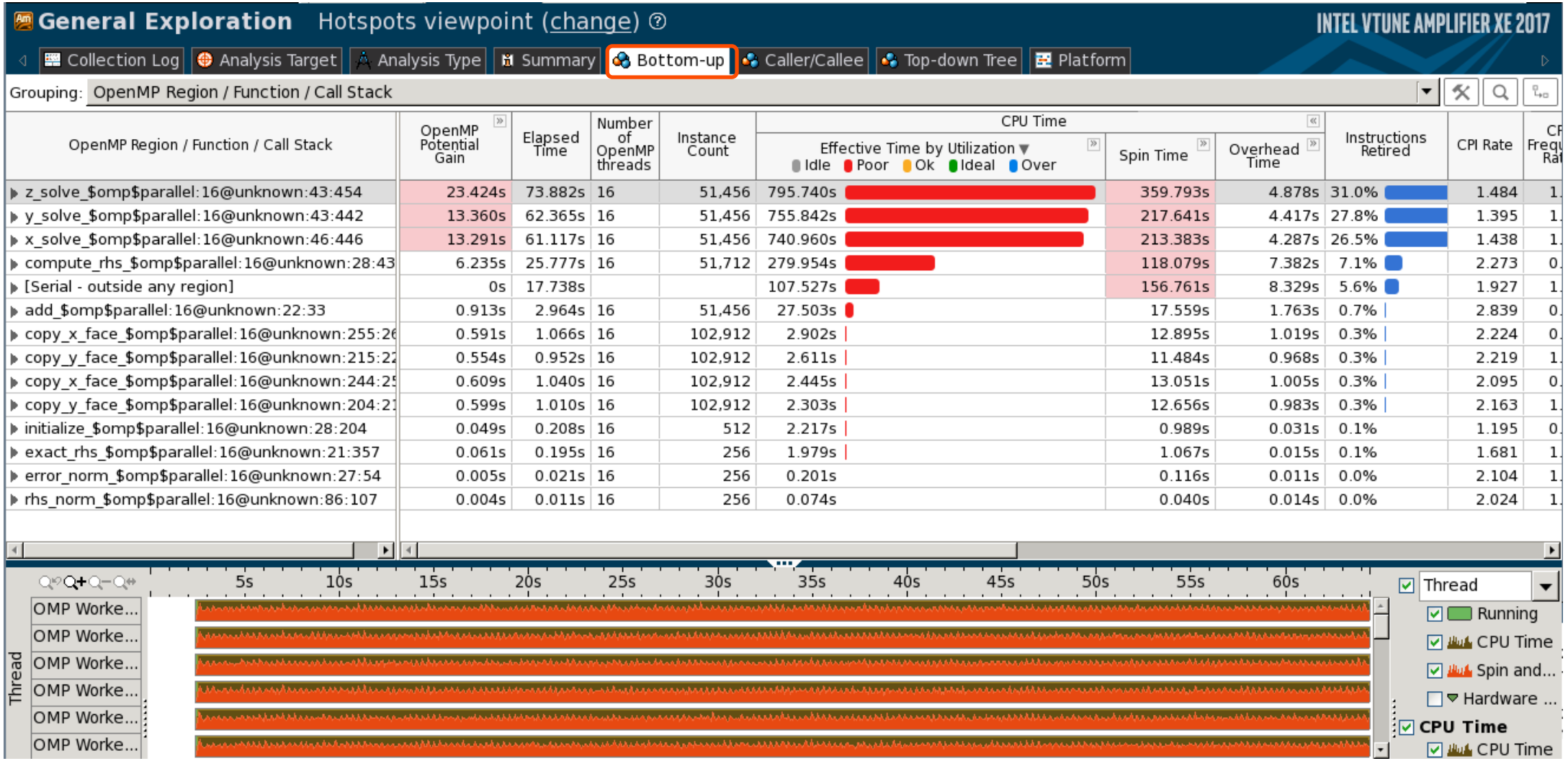
Example: General Exploration

Bottom-up View

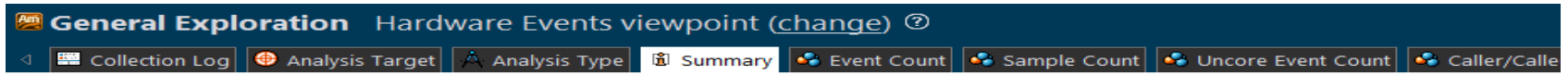


Function / Call Stack	Clockticks ▼	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation		Back-End Bound		Retiring	
					Branch Mispredi...	Machine Clears	Memory Bound	Core Bound		
grid_intersect	14,293,700,000	11,109,300,000	1.287	7.1%	4.2%	0.4%	27.0%	42.3%	19.0%	3_tachyo
sphere_intersect	8,569,000,000	7,885,000,000	1.087	6.2%	0.0%	7.4%	18.2%	45.8%	22.4%	3_tachyo
func@0x1002e59d	374,300,000	311,600,000	1.201	5.1%	0.0%	15.2%	0.0%	28.9%	50.8%	libiomp5r
shader	271,700,000	195,700,000	1.388	10.5%	0.0%	0.0%	31.6%	47.4%	14.0%	3_tachyo
KeDelayExecutionThread	220,400,000	110,200,000	2.000	17.2%	0.0%	17.2%	0.0%	31.0%	34.5%	ntoskrnl.e
func@0x1401514f0	212,800,000	26,600,000	8.000	31.3%	0.0%	22.3%	0.0%	19.6%	26.8%	ntoskrnl.e
tri_intersect	195,700,000	210,900,000	0.928	4.9%	0.0%	9.7%	0.0%	61.2%	24.3%	3_tachyo
__kmp_x86_pause	144,400,000	43,700,000	3.304	19.7%	0.0%	0.0%	0.0%	34.2%	72.4%	libiomp5r
[wow64cpu.dll]	104,500,000	41,800,000	2.500	18.2%	0.0%	0.0%	0.0%	63.6%	54.5%	wow64cp
light_intersect	87,400,000	57,000,000	1.533	10.9%	0.0%	10.9%	0.0%	67.4%	10.9%	3_tachyo
func@0x10013010	81,700,000	247,000,000	0.331	0.0%	0.0%	0.0%	32.6%	32.6%	46.5%	gdiplus.d
VNorm	66,500,000	34,200,000	1.944	14.3%	0.0%	14.3%	28.6%	28.6%	14.3%	3_tachyo
func@0x10009c00	58,900,000	125,400,000	0.470	0.0%	0.0%	16.1%	0.0%	83.9%	0.0%	gdiplus.d
SleepEx	58,900,000	17,100,000	3.444	0.0%	0.0%	16.1%	0.0%	51.6%	32.3%	kernelbas
__libm_sse2_pow	55,100,000	26,600,000	2.071	17.2%	0.0%	17.2%	0.0%	65.5%	0.0%	libmmd.d
func@0x140159900	55,100,000	11,400,000	4.833	0.0%	0.0%	17.2%	0.0%	65.5%	17.2%	ntoskrnl.e

Find real CPU stalls due to cache misses, instruction fetch misses, branch misprediction, and a lot more.



Example – Hardware Events Viewpoint



Hardware Events

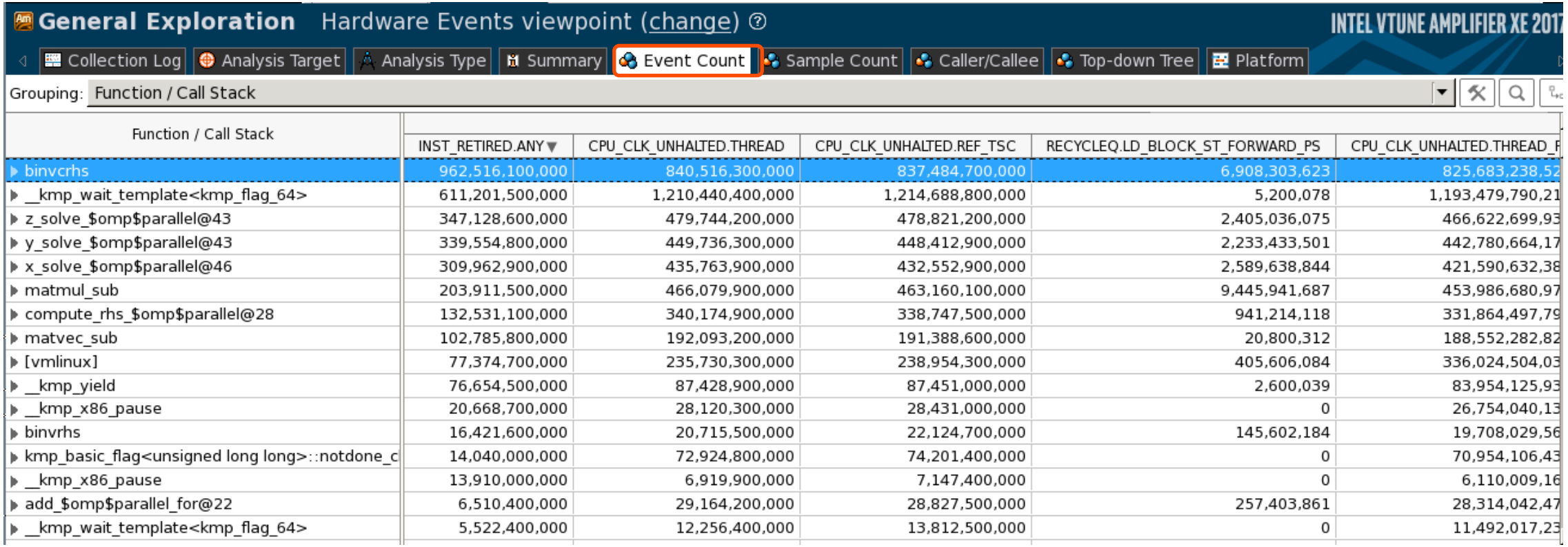
Hardware Event Type	Hardware Event Count	Hardware Event Sample Count	Events Per Sample
BACLEARS.ALL	141,602,124	59	200003
CPU_CLK_UNHALTED.REF_TSC	73,650,110,475	36,825	2000003
CPU_CLK_UNHALTED.THREAD	79,282,118,923	39,641	2000003
CPU_CLK_UNHALTED.THREAD_P	79,248,118,872	3,302	2000003
CYCLES_DIV_BUSY.ALL	0	0	2000003
FETCH_STALL.ICACHE_FILL_PENDING_CYCLES	285,604,284	119	200003
INST_RETIRED.ANY	10,042,015,063	5,021	2000003
MACHINE_CLEARS.FP_ASSIST	0	0	200003
MACHINE_CLEARS.MEMORY_ORDERING	0	0	200003
MACHINE_CLEARS.SMC	0	0	200003
MEM_UOPS_RETIRED.ALL_LOADS	2,913,643,704	1,214	200003
MEM_UOPS_RETIRED.ALL_STORES	974,414,616	406	200003
MEM_UOPS_RETIRED.HITM	0	0	200003
MEM_UOPS_RETIRED.L1_MISS_LOADS	1,048,815,732	437	200003
MEM_UOPS_RETIRED.L2_HIT_LOADS_PS	993,614,904	414	200003
MEM_UOPS_RETIRED.L2_MISS_LOADS_PS	28,802,016	24	100007
MEM_UOPS_RETIRED.UTLB_MISS_LOADS	40,800,612	17	200003
MS_DECODED.MS_ENTRY	4,800,072	2	200003
NO_ALLOC_CYCLES.MISPREDICTS	19,805,097,072	8,252	200003
NO_ALLOC_CYCLES.NOT_DELIVERED	7,183,307,748	2,993	200003
PAGE_WALKS.CYCLES	271,204,068	113	200003
PAGE_WALKS.I_SIDE_CYCLES	36,000,540	15	200003
REHABQ.LD_SPLITS_PS	840,012,600	350	200003
UOPS_RETIRED.ALL	10,320,015,480	430	2000003
UOPS_RETIRED.MS	96,000,144	4	2000003
UOPS_RETIRED.PACKED_SIMD	4,437,666,564	1,849	200003
UOPS_RETIRED.SCALAR_SIMD	124,801,872	52	200003

Uncore Event Count

Uncore Event Type	Uncore Event Count
UNC_M_CAS_COUNT.RD[UNIT0]	121,667,400
UNC_M_CAS_COUNT.RD[UNIT1]	121,277,947
UNC_M_CAS_COUNT.RD[UNIT2]	122,910,148
UNC_M_CAS_COUNT.RD[UNIT3]	122,185,974
UNC_M_CAS_COUNT.RD[UNIT4]	120,999,173
[Others]	285,397,113

44% of packed and scalar SIMD over all Uops.retired

Example – Hardware Events Viewpoint



Function / Call Stack	INST_RETIRED.ANY ▼	CPU_CLK_UNHALTED.THREAD	CPU_CLK_UNHALTED.REF_TSC	RECYCLEQ.LD_BLOCK_ST_FORWARD_PS	CPU_CLK_UNHALTED.THREAD_F
binvcrhs	962,516,100,000	840,516,300,000	837,484,700,000	6,908,303,623	825,683,238,52
__kmp_wait_template<kmp_flag_64>	611,201,500,000	1,210,440,400,000	1,214,688,800,000	5,200,078	1,193,479,790,21
z_solve_ompparallel@43	347,128,600,000	479,744,200,000	478,821,200,000	2,405,036,075	466,622,699,93
y_solve_ompparallel@43	339,554,800,000	449,736,300,000	448,412,900,000	2,233,433,501	442,780,664,17
x_solve_ompparallel@46	309,962,900,000	435,763,900,000	432,552,900,000	2,589,638,844	421,590,632,38
matmul_sub	203,911,500,000	466,079,900,000	463,160,100,000	9,445,941,687	453,986,680,97
compute_rhs_ompparallel@28	132,531,100,000	340,174,900,000	338,747,500,000	941,214,118	331,864,497,79
matvec_sub	102,785,800,000	192,093,200,000	191,388,600,000	20,800,312	188,552,282,82
[vmlinux]	77,374,700,000	235,730,300,000	238,954,300,000	405,606,084	336,024,504,03
__kmp_yield	76,654,500,000	87,428,900,000	87,451,000,000	2,600,039	83,954,125,93
__kmp_x86_pause	20,668,700,000	28,120,300,000	28,431,000,000	0	26,754,040,13
binvrhs	16,421,600,000	20,715,500,000	22,124,700,000	145,602,184	19,708,029,56
kmp_basic_flag<unsigned long long>::notdone_c	14,040,000,000	72,924,800,000	74,201,400,000	0	70,954,106,43
__kmp_x86_pause	13,910,000,000	6,919,900,000	7,147,400,000	0	6,110,009,16
add_ompparallel_for@22	6,510,400,000	29,164,200,000	28,827,500,000	257,403,861	28,314,042,47
__kmp_wait_template<kmp_flag_64>	5,522,400,000	12,256,400,000	13,812,500,000	0	11,492,017,23

Example – Gflops and FPU utilization

Elapsed Time: 64.488s

GFLOPS Upper Bound: 92.897

CPU Utilization: 16.5%

Average CPU Usage: 42.213 Out of 256 logical CPUs

Top OpenMP Processes by MPI Communication Spin Time

This section lists processes sorted by MPI Communication Spin time. The lower MPI Communication Spin time, the mc
efficiency metrics by MPI processes laying on the critical path

Process	PID	OpenMP Potential Gain	(%)	Serial Time	(%)
bt-mz.C.4 (rank 0)	238538	15.057s	24.3%	3.962s	6.4%
bt-mz.C.4 (rank 2)	238540	13.994s	22.5%	4.492s	7.2%
bt-mz.C.4 (rank 3)	238542	14.726s	23.7%	4.558s	7.3%
bt-mz.C.4 (rank 1)	238550	15.920s	25.6%	4.726s	7.6%

CPU Usage Histogram

Back-End Bound: 39.6%

L2 Hit Bound: 0.080

L2 Miss Bound: 0.167

FPU Utilization Upper Bound: 3.7%

GFLOPS Upper Bound: 92.897

Scalar GFLOPS Upper Bound: 24.134

Packed GFLOPS Upper Bound: 68.763

Top 5 hotspot loops (functions) by FPU usage

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

Function	CPU Time	FPU Utilization Upper Bound	Loop Characterization
binvcrhs	645.711s	3.8%	
z_solve_ompparallel@43	369.177s	3.2%	
matmul_sub	357.102s	13.6%	
y_solve_ompparallel@43	345.732s	3.3%	
x_solve_ompparallel@46	333.504s	3.2%	
[Others]	675.905s	N/A*	

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

The HPC Performance Characterization Analysis

Threading: CPU Utilization

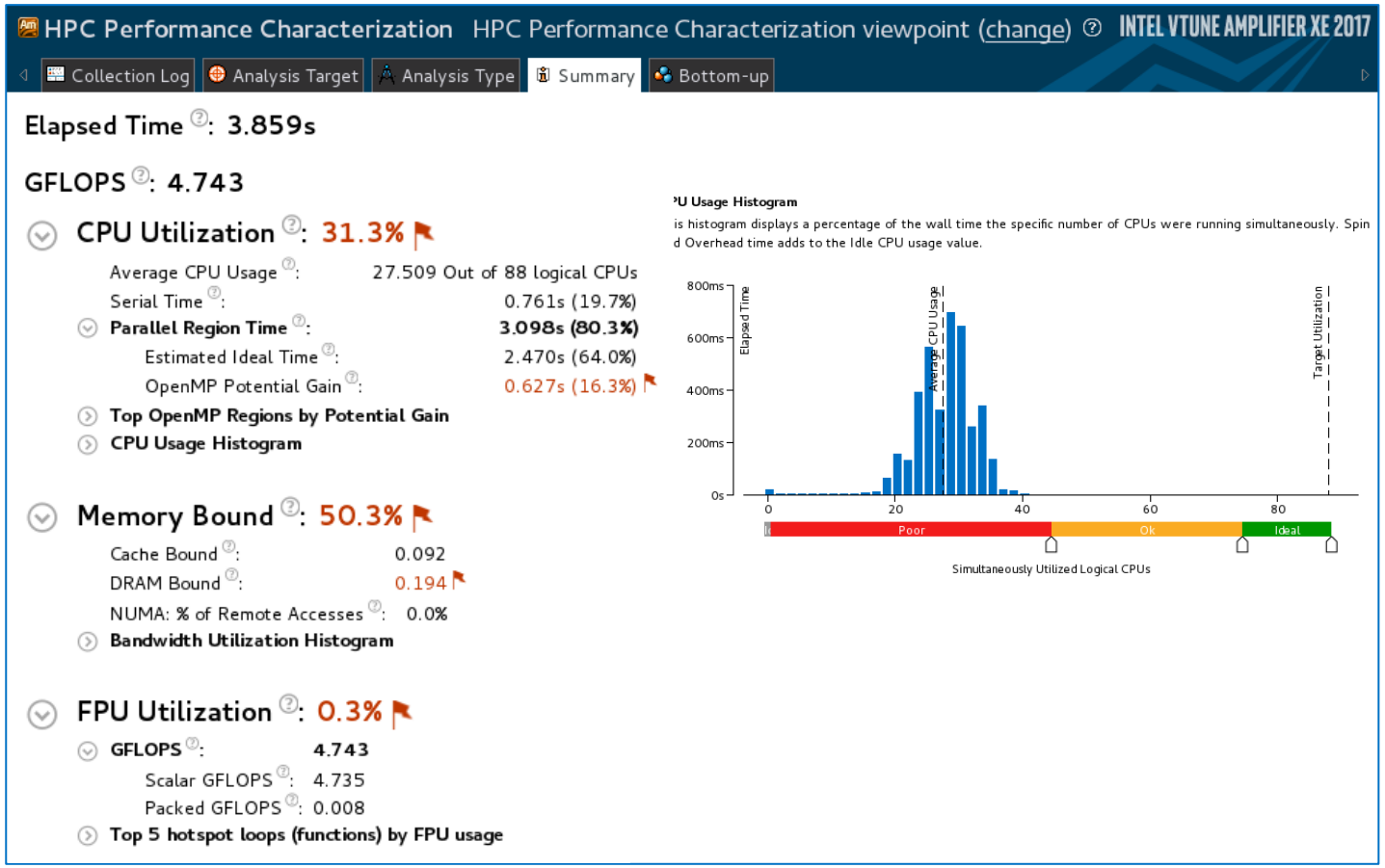
- Serial vs. Parallel time
- Top OpenMP regions by potential gain
- Tip: Use hotspot OpenMP region analysis for more detail

Memory Access Efficiency

- Stalls by memory hierarchy
- Bandwidth utilization
- Tip: Use Memory Access analysis

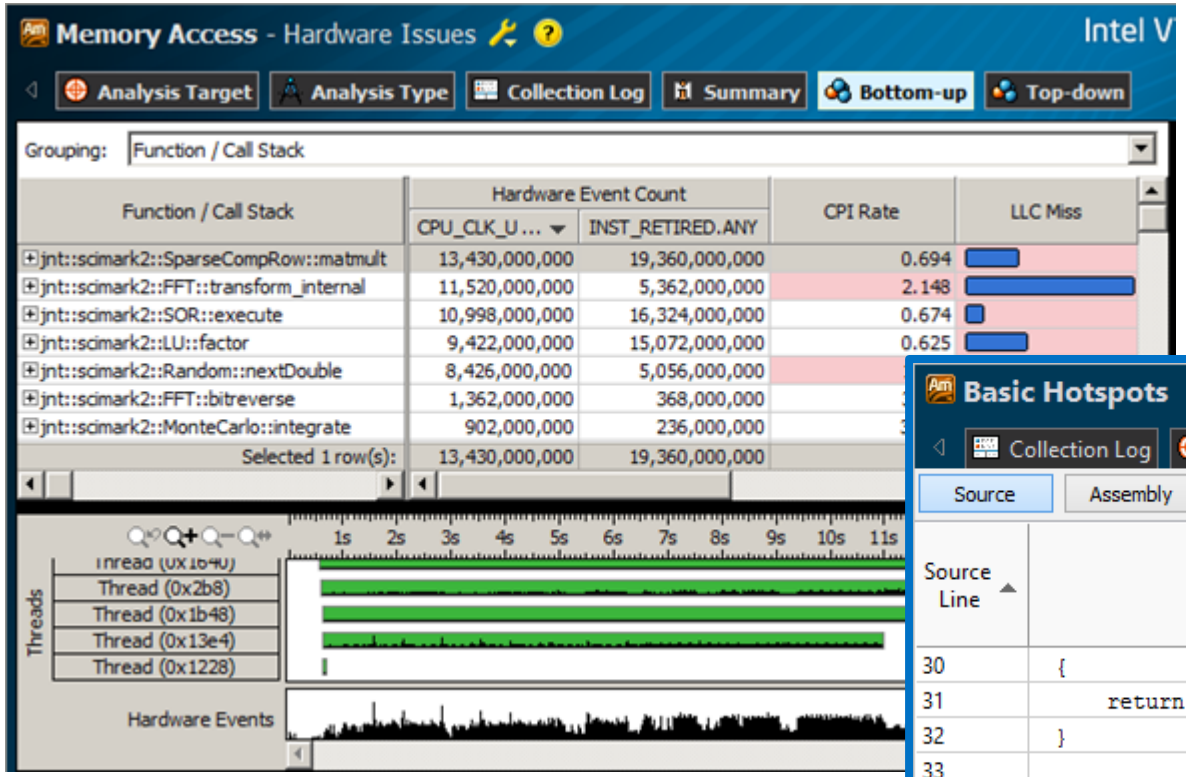
Vectorization: FPU Utilization

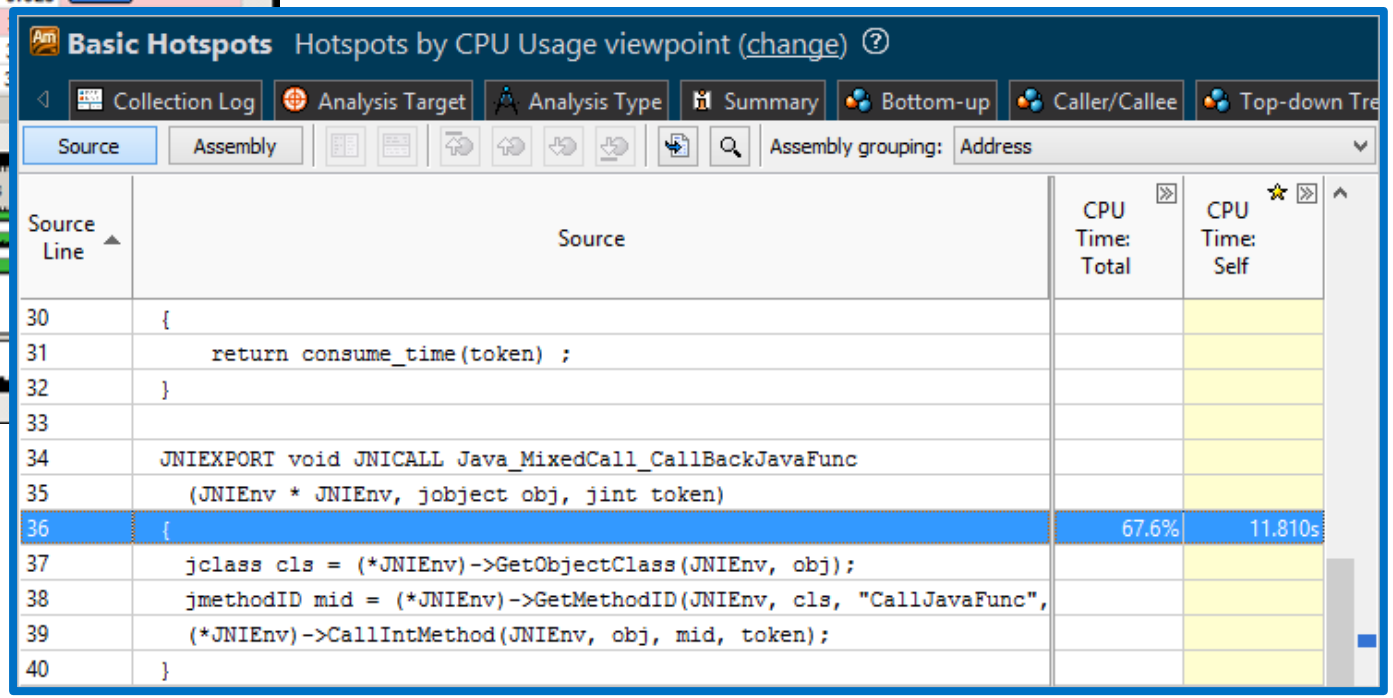
- FLOPS † estimates from sampling
- Tip: Use Intel Advisor for precise metrics and vectorization optimization



† For 3rd, 5th, 6th Generation Intel® Core™ processors and second generation Intel® Xeon Phi™ processor code named Knights Landing.

Java Analysis





- Multiple simultaneous JVMs
- Sampling is fast / unobtrusive
- Mixed Java / C++ / Fortran
- See results on the Java source

Optimize Private Cloud-Based Applications

Profile Enterprise Applications

- Native C, C++, Fortran*
- Attach to running Java* services (e.g., Mail)
- Profile Java daemons without restart

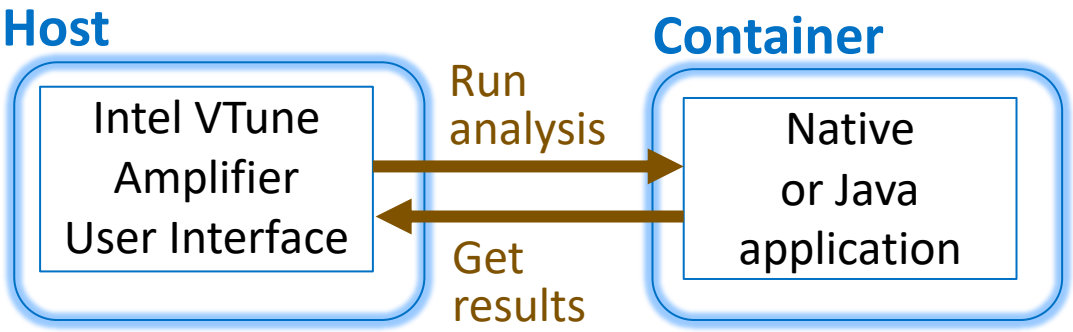
Accurate, Low-Overhead Data Collection

- Advanced hotspots and hardware events
- Memory analysis
- Accurate stack information for Java and HHVM*

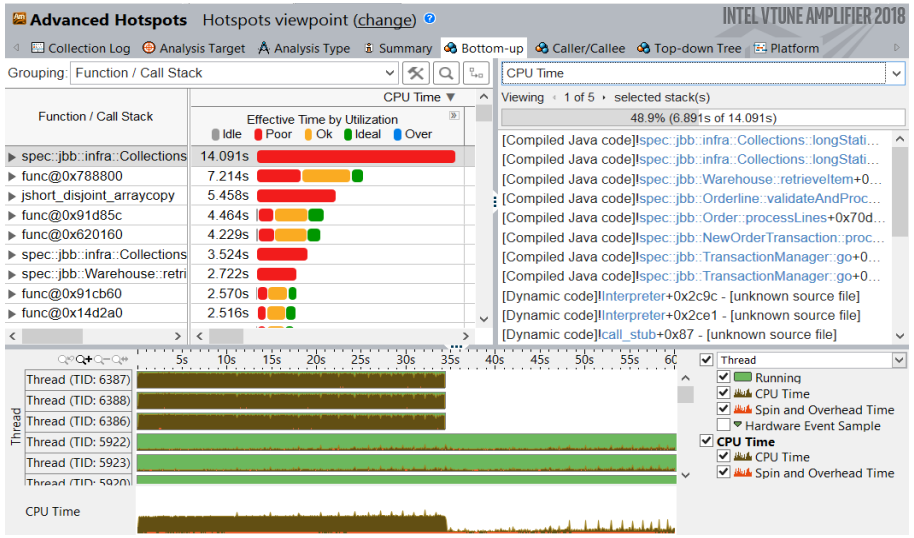
Popular Containers Supported

- Docker*
- Mesos*

Software collectors (e.g., locks & waits) and Python* profiling are not currently available for containers.



- No container configuration required
- Detection of the container is automatic



Storage Device Analysis (HDD, SATA or NVMe SSD)

Intel® VTune™ Amplifier

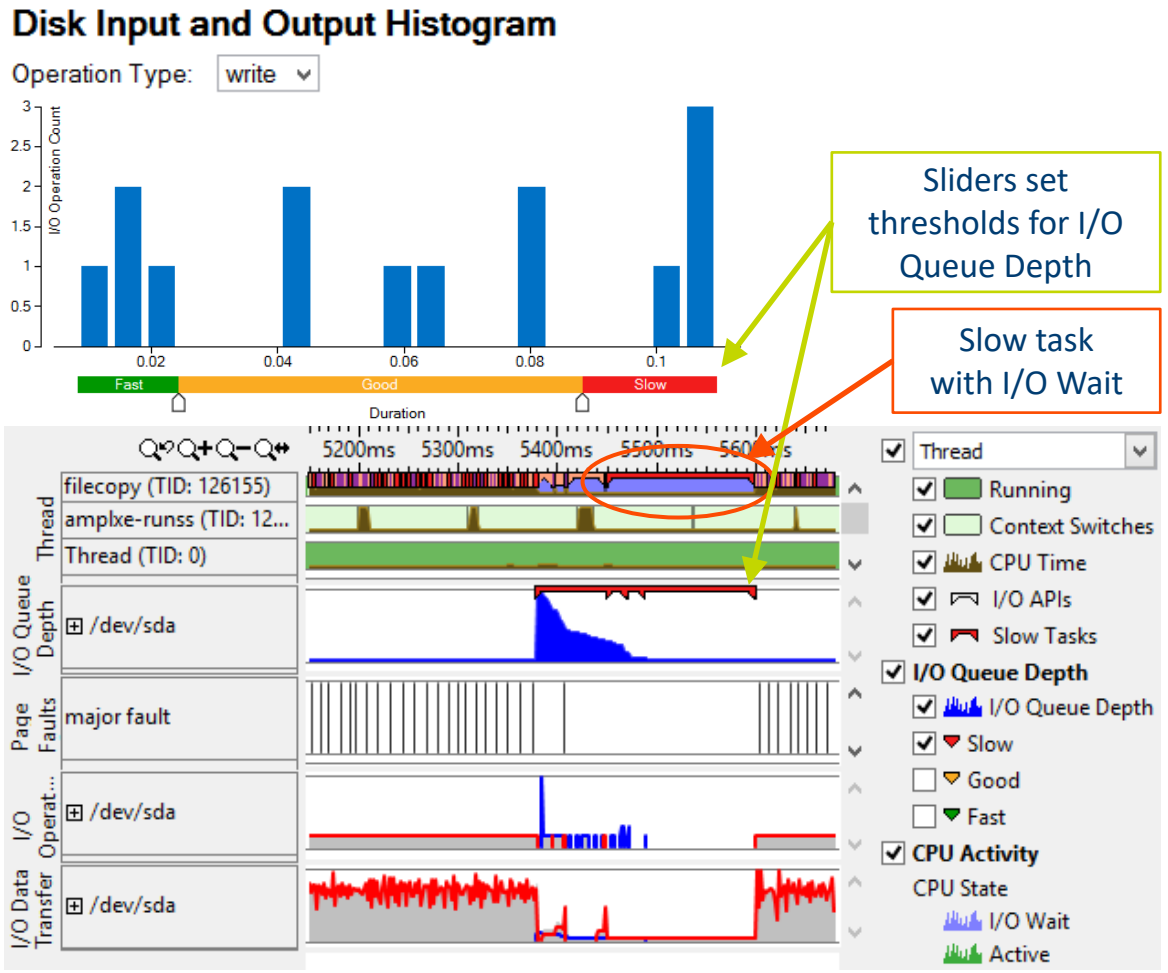
Are You I/O Bound or CPU Bound?

- Explore imbalance between I/O operations (async & sync) and compute
- Storage accesses mapped to the source code
- See when CPU is waiting for I/O
- Measure bus bandwidth to storage

Latency analysis

- Tune storage accesses with latency histogram
- Distribution of I/O over multiple devices

```
> amplxe-cl -collect disk-io –d 10
```



Application Performance Snapshot (APS)

High-level **overview** of application performance

Identify primary optimization areas and **next steps** in analysis

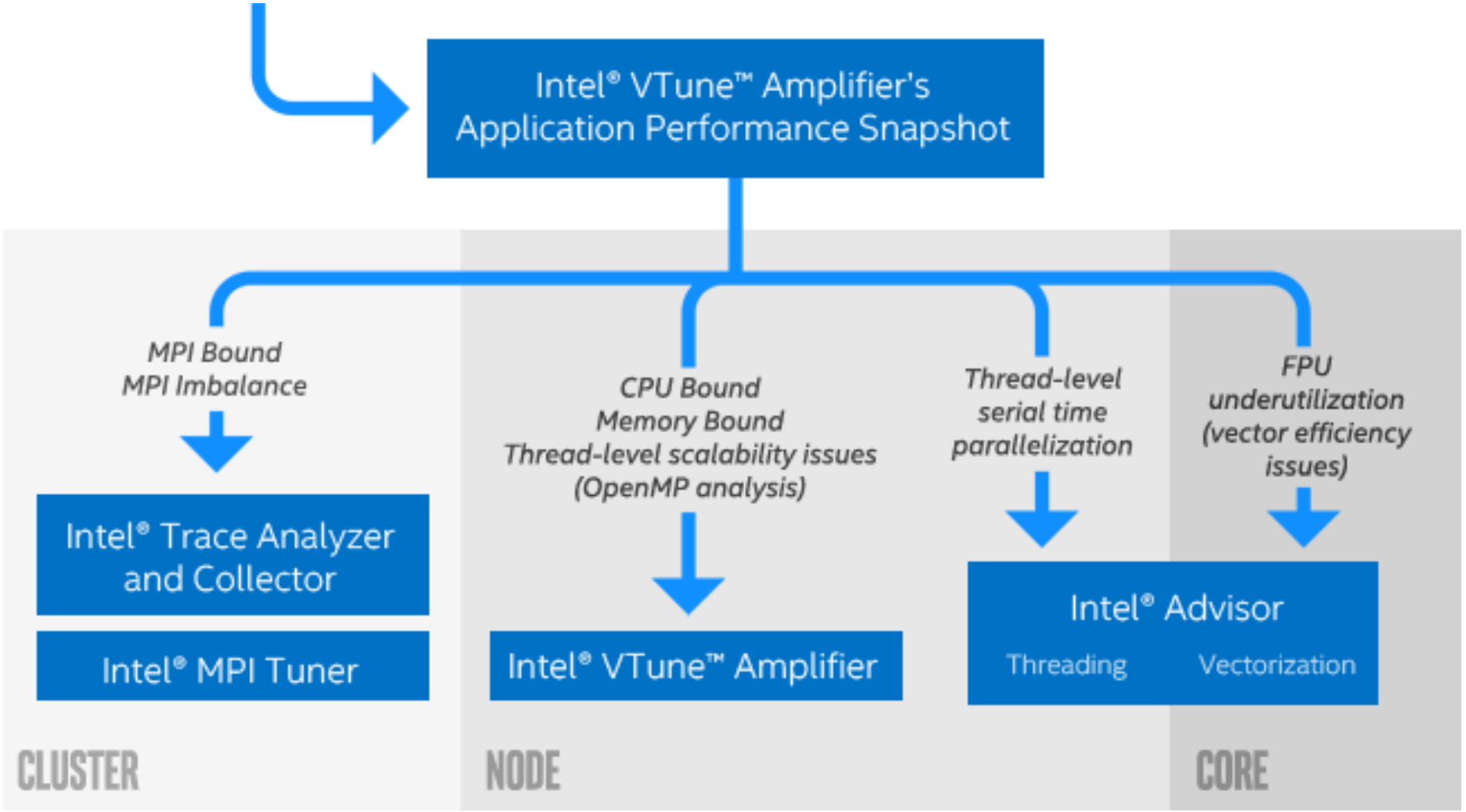
Easy to install, run, explore results with CL or HTML reports

Scales to large jobs

Multiple methods to obtain

- Part of Intel® Parallel Studio XE 2018
  - http://intel.ly/intel-parallel-studio-xe-2018-beta
- Separate **free** download (110Mb) from APS page
  - https://software.intel.com/sites/products/snapshots/application-snapshot/

Performance Optimization Workflow based on APS



APS Usage

Setup Environment

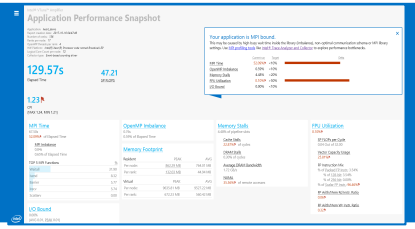
- >source <APS_Install_dir>/apsvars.sh

Run Application

- >aps <application and args>
- MPI: >mpirun <mpi options> aps <application and args>

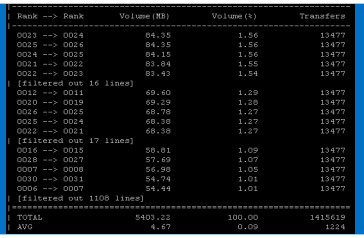
Generate Report on Result Folder

- >aps –report <result folder>

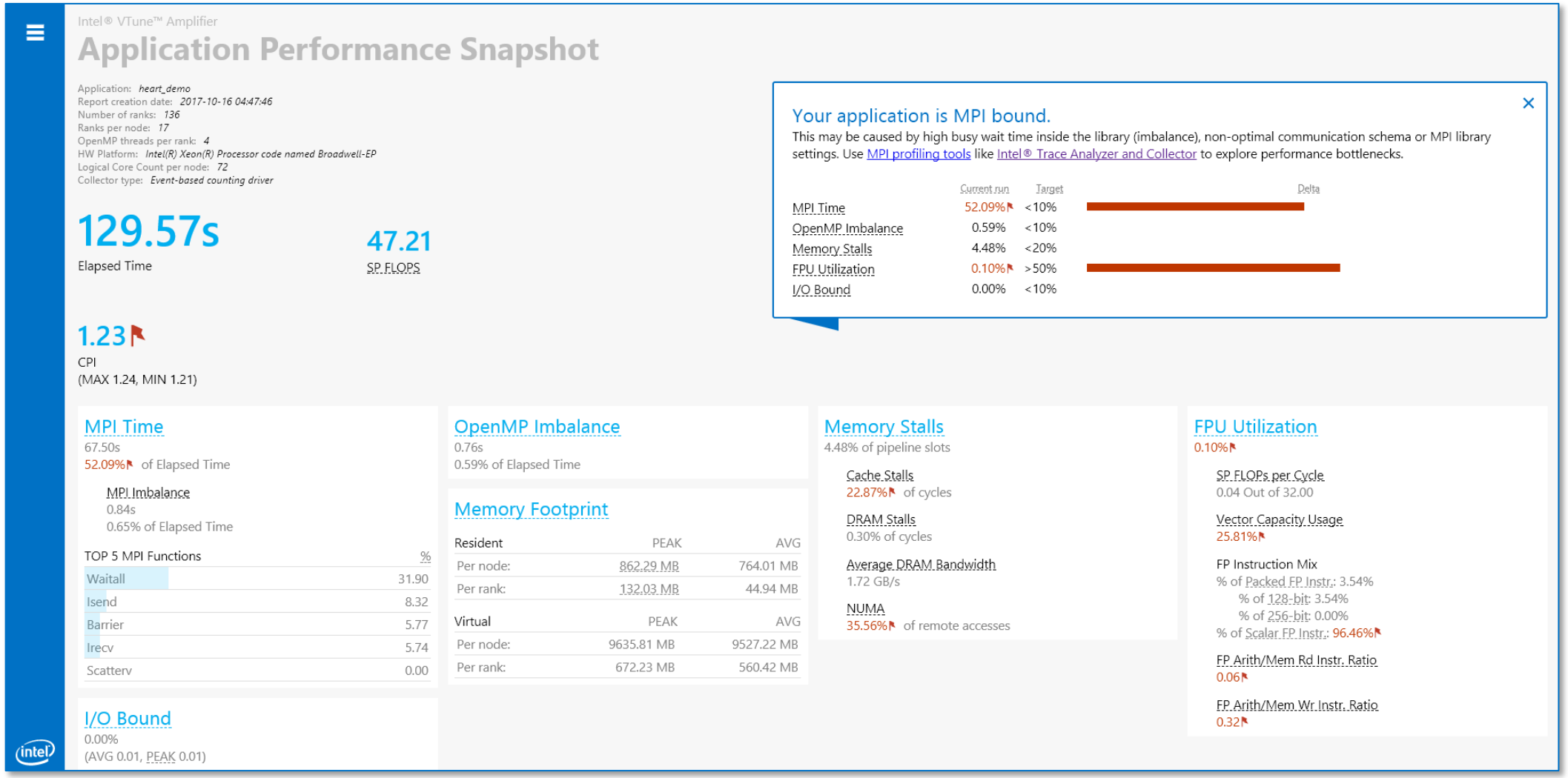


Generate CL reports with detailed MPI statistics on Result Folder

- aps-report –<option> <result folder>



APS HTML Report



APS Command Line Reports - Summary

```
| Summary information
|-----------------------------------------------------------------
  Application      : heart_demo.test02
  Number of ranks  : 8
  Used statistics  : stat_20170502/
  Creation date    : 2017-05-02 11:44:27
|
  Your application has significant OpenMP imbalance. Use OpenMP profiling tools like Intel(R) VTune(TM) Amplifier
 to see the imbalance details.
|
  Elapsed time:              73.19 sec
  CPI Rate:                   4.01
| The CPI value may be too high.
| This could be caused by such issues as memory stalls, instruction starvation,
| branch misprediction, or long latency instructions.
| Use Intel(R) VTune(TM) Amplifier General Exploration analysis to specify
| particular reasons of high CPI.
  MPI Time:                  11.48 sec           15.69%
| Your application is MPI bound. This may be caused by high busy wait time
| inside the library (imbalance), non-optimal communication schema or MPI
| library settings. Explore the MPI Imbalance metric if it is available or use
| MPI profiling tools like Intel(R) Trace Analyzer and Collector to explore
| possible performance bottlenecks.
    MPI Imbalance:            3.36 sec            4.59%
  OpenMP Imbalance:          22.52 sec           30.77%
| The metric value can indicate significant time spent by threads waiting at
| barriers. Consider using dynamic work scheduling to reduce the imbalance where
| possible. Use Intel(R) VTune(TM) Amplifier HPC Performance Characterization
| analysis to review imbalance data distributed by barriers of different lexical
| regions.
```

Tip:

>aps –report=<my_result_dir> | grep –v “|”
eliminating verbose descriptions

APS Command Line Reports – Advanced MPI statistics (1/3)

- MPI Time per rank
  - aps-report –t <result>

MPI Time per Rank

Rank	LifeTime(sec)	MPI Time(sec)	MPI Time(%)	Imbalance(sec)	Imbalance(%)
0007	72.52	14.31	19.74	4.84	6.67
0004	72.53	11.57	15.96	3.26	4.50
0005	72.52	11.40	15.72	3.20	4.42
0006	72.51	11.11	15.32	3.17	4.37
0000	72.49	11.08	15.29	4.33	5.97
0001	72.52	10.95	15.10	3.01	4.15
0002	72.49	10.79	14.88	2.57	3.55
0003	72.50	10.64	14.68	2.50	3.45
TOTAL	580.07	91.86	15.84	26.88	4.63
AVG	72.51	11.48	15.84	3.36	4.63

APS Command Line Reports – Advanced MPI statistics (2/3)

Message Size Summary by all ranks

- aps-report –m <result>

Message Sizes summary for all ranks

Message size(B)	Volume(MB)	Volume(%)	Transfers	Time(sec)	Time(%)
8	1.49	0.09	195206	27.79	37.93
176	0.41	0.02	2420	27.67	37.78
4	0.00	0.00	1150	15.55	21.22
100264	115.89	6.94	1212	0.27	0.37
98400	113.74	6.81	1212	0.19	0.26
66256	38.29	2.29	606	0.17	0.23
[filtered out 57 lines]					
TOTAL	1670.60	100.00	265160	73.25	100.00

APS Command Line Reports – Advanced MPI statistics (3/3)

Data Transfers for Rank-to-Rank Communication

- aps-report –x <result>

And many others – check

- aps-report -help

Rank --> Rank	Volume(MB)	Volume(%)	Transfers
0023 --> 0024	84.35	1.56	13477
0025 --> 0026	84.35	1.56	13477
0024 --> 0025	84.15	1.56	13477
0021 --> 0022	83.84	1.55	13477
0022 --> 0023	83.43	1.54	13477
[filtered out 16 lines]			
0012 --> 0011	69.60	1.29	13477
0020 --> 0019	69.29	1.28	13477
0026 --> 0025	68.78	1.27	13477
0025 --> 0024	68.38	1.27	13477
0022 --> 0021	68.38	1.27	13477
[filtered out 17 lines]			
0016 --> 0015	58.81	1.09	13477
0028 --> 0027	57.69	1.07	13477
0007 --> 0008	56.98	1.05	13477
0030 --> 0031	54.74	1.01	13477
0006 --> 0007	54.44	1.01	13477
[filtered out 1108 lines]			
TOTAL	5403.22	100.00	1415619
AVG	4.67	0.09	1224

Collection Control API

Since 2018 Update 2

To measure a particular application phase or exclude initialization/finalization phases use:

MPI:

- Pause: MPI_Pcontrol(0)
- Resume: MPI_Pcontrol(1)

MPI or Shared memory applications:

- Pause: __itt_pause()
- Resume: __itt_resume()
  - See how to configure the build of your application to use itt API

Tip: use aps “-start-paused” option allows to start application without profiling and skip initialization phase

Python

Profiling Python is straightforward in VTune™ Amplifier, as long as one does the following:

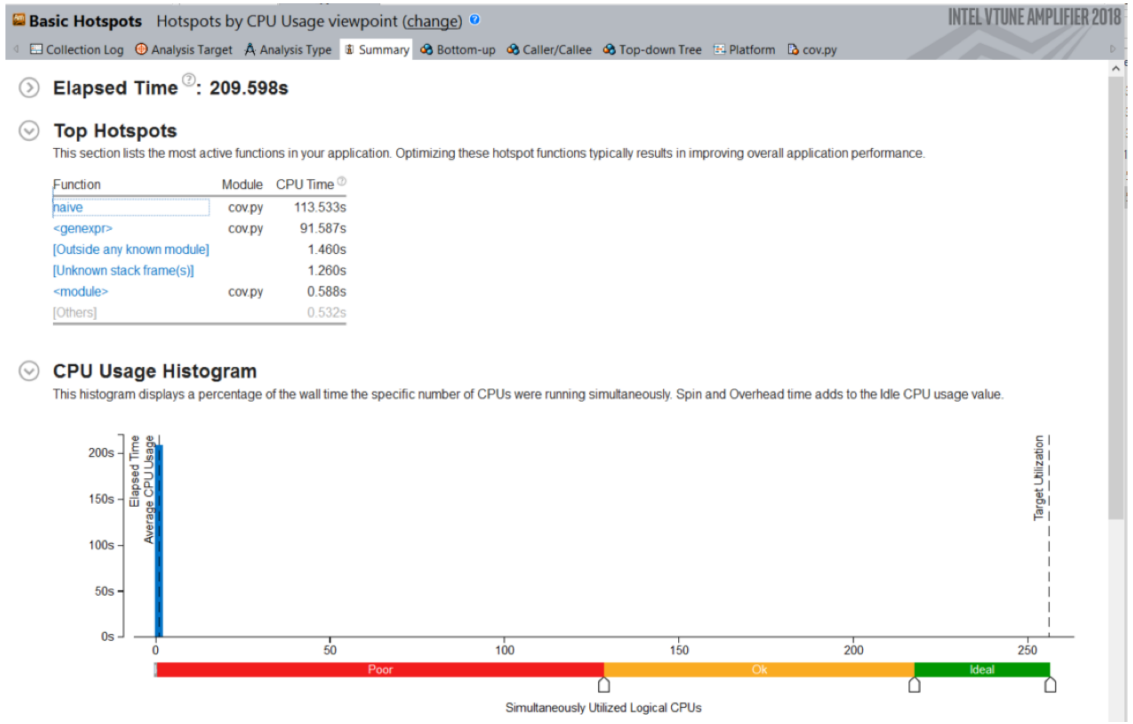
- The “application” should be the full path to the python interpreter used
- The python code should be passed as “arguments” to the “application”

Example:

```
mpirun -n 1 -N 1 amplxe-cl -c hotspots -r res_dir \
                  -- /usr/bin/python3 mycode.py myarguments
```

Simple Python Example

```
mpirun -n 1 -N 1 amplxe-cl -c hotspots -r vt_pytest \
                           -- /usr/bin/python ./cov.py naive 100 1000
```



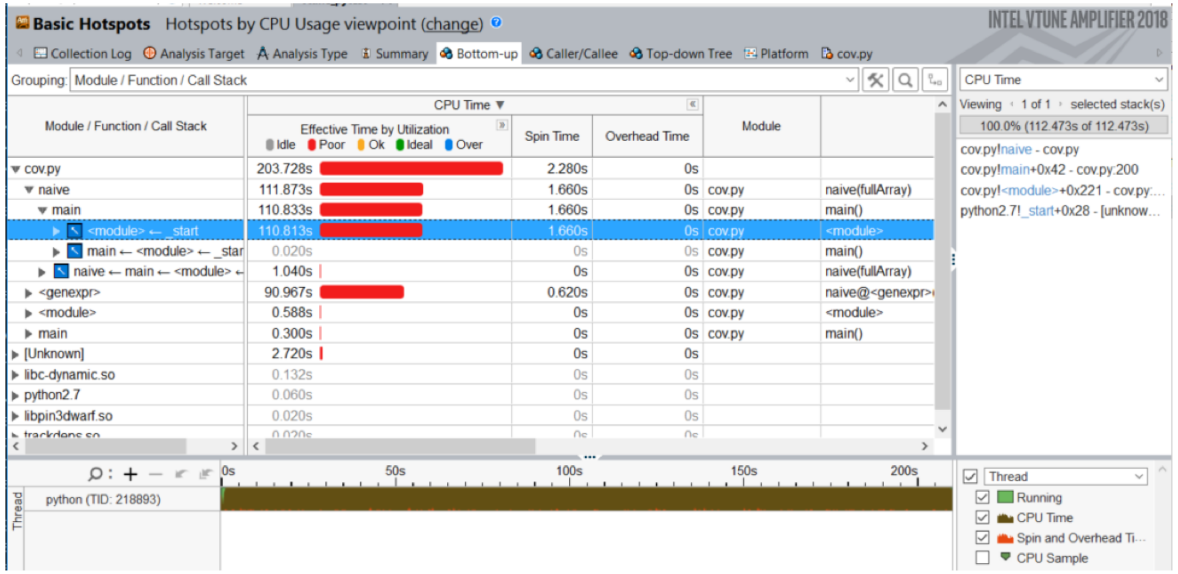
Naïve implementation of the calculation of a covariance matrix

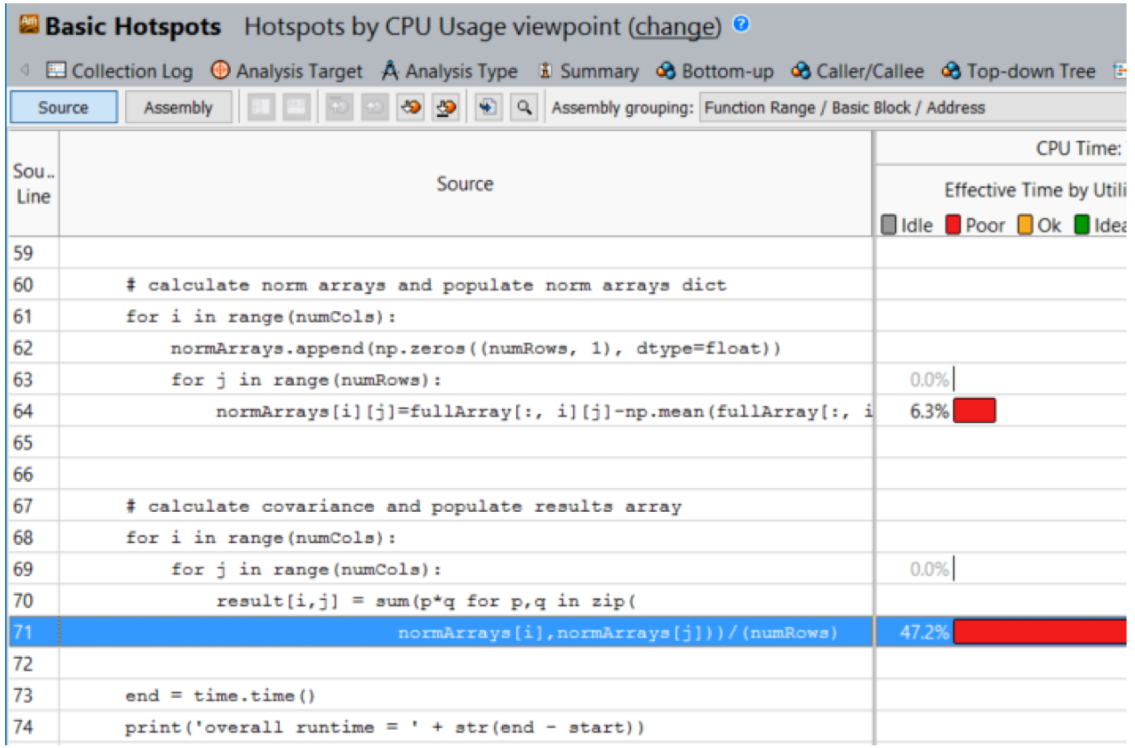
Summary shows:

- Single thread execution
- Top function is “naive”

Click on top function to go to Bottom-up view

Bottom-up View and Source Code





Inefficient array multiplication found quickly
We could use numpy to improve on this

Note that for mixed Python/C code a Top-Down view can often be helpful to drill down into the C kernels

VTune Command Line Analysis Configuration

How to Run VTune on MPI Applications

><mpi_launcher> – n N <vtune_command_line> ./app_to_run

- >srun –n 48 -N 16 amplxe-cl –collect memory-access **–trace-mpi** –r result_dir ./my_mpi_app
- >mpirun –n 48 -ppn 16 amplxe-cl –collect advanced-hotspots –r result_dir ./my_mpi_app

- Encapsulates ranks to per-node result directories suffixed with hostname
  - result_dir.hostname1 with 0-15, result_dir.hostname2 with 16-31, result_dir.hostname3 with 32-47

→ Add **–trace-mpi** option for VTune CL to enable per-node result directories for non-Intel MPIs

- Works for software and Intel driver-based collectors

VTune Command Line Analysis Configuration

Selective Rank Profiling

Superposition of application to launch and VTune command line for selective ranks to reduce trace size

Example: profile rank 1 from 0-15:

>mpirun -n 1 ./my_app : -n 1 <vtune_command_line> -- ./my_app : -n 14 ./my_app

- In the case of Intel MPI launcher –gtool option can be used:

Example: profile ranks 3, 7, 11-13 from 0-15:

>mpirun –gtool “amplxe-cl –collect advanced-hotspots –r result_dir:3,7,11-13” ./my_app

Analysis Workflow

Result finalization and viewing on KNL target might be slow

Use the recommended workflow:

1. Run collection on KNL deferring finalization to host:

   >amplxe-cl –collect memory-access –no-auto-finalize –r <my_result_dir> ./my_app

2. Finalize the result on the host

   - Provide search directories to the binaries of interest for resolving with –search-dir option

   >amplxe-cl –finalize –r <my_result_dir> –search-dir <my_binary_dir>

3. Generate reports, work with GUI

   >amplxe-cl –report hotspots –r <my_result_dir>

Legal Disclaimer & Optimization Notice

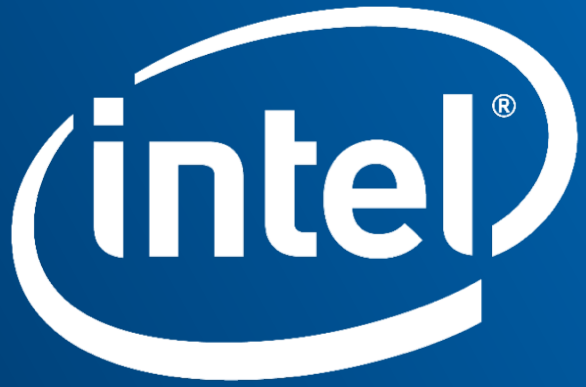
INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

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. For more complete information visit www.intel.com/benchmarks.

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



Software