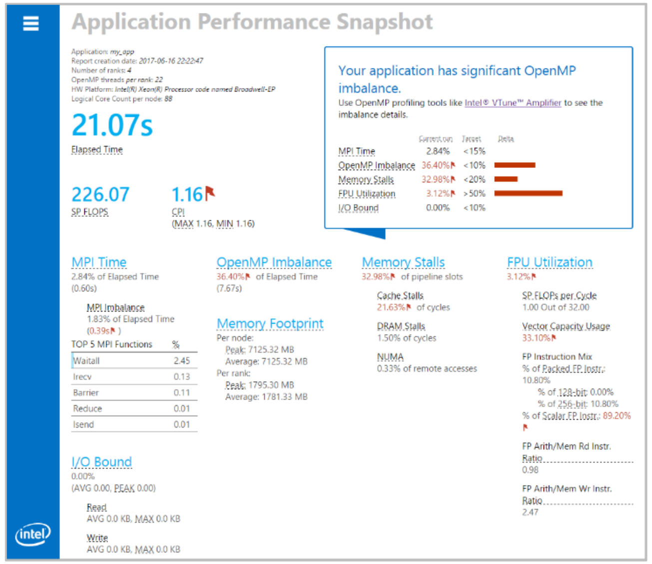
OPTIMIZING THREADED CODE PERFORMANCE AND SCALABILITY

Intel Software Developer Conference – London, 2017

AGENDA

- **Which tools should I use for threading and scalability ?**
- Intel® Performance Snapshot
- Intel® VTune™ Amplifier
- Some examples and solutions
- What's new in 2018 ?



WHICH TOOL SHOULD I USE FOR THREADING AND SCALABILITY?

- Intel® Performance Snapshots
  - Provides high level and easy to understand metrics
  - Highlight the main bottlenecks
  - Can be easily integrated in the build chain to provide feedback to developers
- Intel® Vtune™ Amplifier
  - Go deeper, get detailed information about source lines
  - Dedicated analysis to target a specific aspect (threading, memory, etc)

AGENDA

- Which tools should I use for threading and scalability ?
- **Intel® Performance Snapshot**
- Intel® VTune™ Amplifier
- Some examples and solutions
- What's new in 2018 ?

BEFORE DIVE TO A PARTICULAR TOOL..

- How to assess that I have potential in performance tuning?
- Which tool should I use first?
- What to use on big scale not be overwhelmed with huge trace size, post processing time and collection overhead?
  - On a KNL cluster customers can end-up with more than 1000 ranks on just 8 nodes
- How to quickly evaluate environment settings or incremental code changes?
- **Answer: try Application Performance Snapshot 2018**

APPLICATION PERFORMANCE SNAPSHOT (APS)

- High-level overview of application performance
- Identify primary optimization areas and next steps in analysis
- Easy to use
- Detailed reports available via command line
- Scales to large jobs
- Multiple methods to obtain
  - Part of Intel® Parallel Studio XE 2018
  - Separate free download from Performance Snapshot page

APS HTML REPORT

Application Performance Snapshot

Application:heart_demo_avx_2
Number of ranks:22
Used statistics:/home/vtune/dprohoro/apps/Cardiac/Cardiac/build/stat_20170605
Creation date:2017-06-05 21:33:32

20.22s
Elapsed Time

60.81
SP FLOPS

1.12
CPI
(MAX 1.13, MIN 1.12)

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. Use MPI profiling tools like Intel® Trace Analyzer and Collector to explore performance bottlenecks.

	Current run	Target	Delta
MPI Time	62.60%	<15%	
OpenMP Imbalance	4.03%	<10%	
Memory Stalls	23.33%	<20%	
FPU Utilization	0.90%	>50%	
I/O Bound	0.00%	<10%	

MPI Time
62.60% of Elapsed Time
(12.66s)

MPI Imbalance
53.16% of Elapsed Time
(10.75s)

TOP 5 MPI Functions	%
Waitall	55.30
Barrier	5.80
Isend	0.28
Irecv	0.15
Scatterv	0.01

OpenMP Imbalance
4.03% of Elapsed Time
(0.81s)

Memory Footprint
Per node: AVG 11055.40 MB, PEAK 11055.40 MB
Per rank: AVG 502.52 MB, PEAK 610.43 MB

Memory Stalls
23.33% of pipeline slots

Cache Stalls
24.45% of cycles

DRAM Stalls
0.01% of cycles

NUMA
16.03% of remote accesses

FPU Utilization
0.90%

SP FLOPs per Cycle
0.28 Out of 32.00

Vector Capacity Usage
26.21%

FP Instruction Mix
% of Packed FP Instr.: 3.06%
% of 128-bit: 2.11%
% of 256-bit: 0.96%
% of Scalar FP Instr.: 96.94%

FP Arith/Mem Rd Instr. Ratio
0.44

APS USAGE

Setup Environment

- source <APS_Install_dir>/apsvars.sh

Run Application

- mpirun <mpi options> aps.sh <application and args>

Generate Report on Results

- aps.sh –report <result folder>

Generate advanced CL reports on Results

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

AGENDA

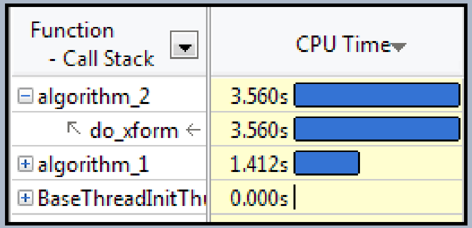
- Which tools should I use for threading and scalability ?
- Intel® Performance Snapshot
- **Intel® VTune™ Amplifier**
- Some examples and solutions
- What's new in 2018 ?

INTEL® VTUNE™ AMPLIFIER XE

Performance Profiler

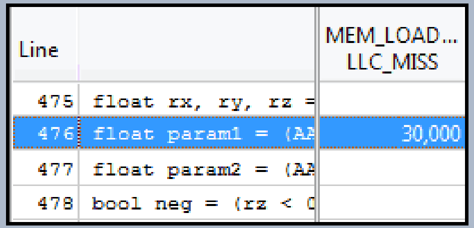
Where is my application…

Spending Time?



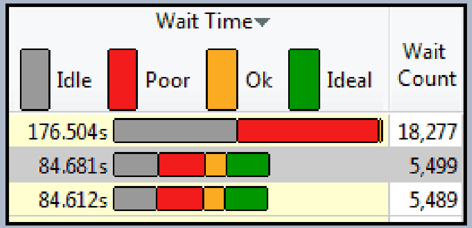
- Focus tuning on functions taking time
- See call stacks
- See time on source

Wasting Time?



- See cache misses on your source
- See functions sorted by # of cache misses

Waiting Too Long?



- See locks by wait time
- Red/Green for CPU utilization during wait

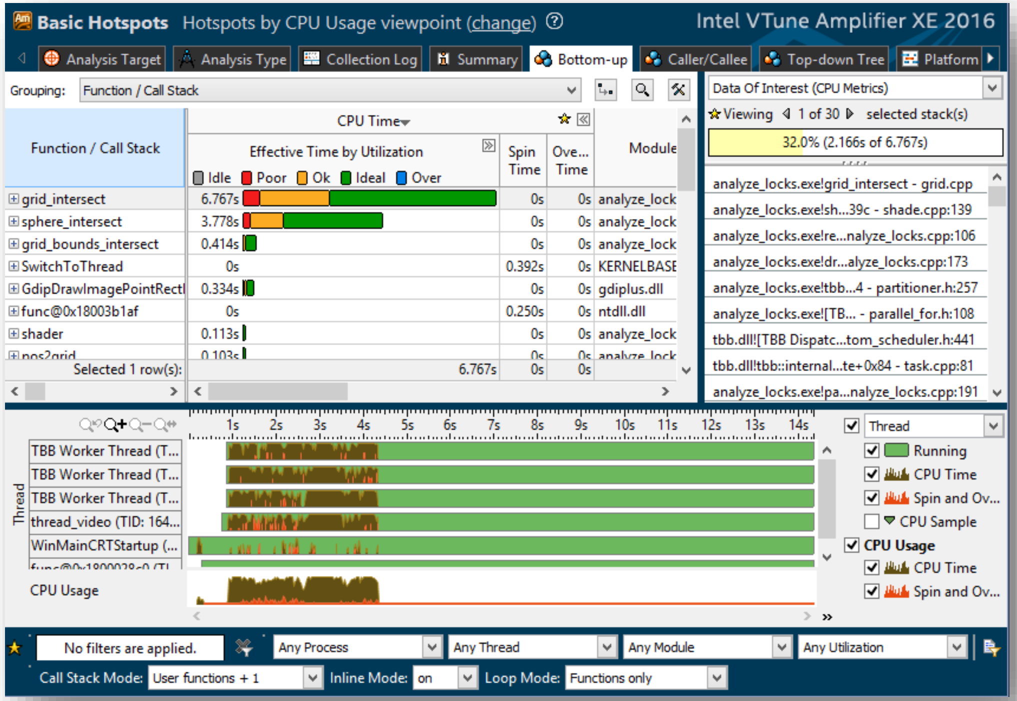
- Windows & Linux
- Low overhead
- No special recompiles

Advanced Profiling For Scalable Multicore Performance

INTEL® VTUNE™ AMPLIFIER XE

Tune Applications for Scalable Multicore Performance

- **Fast, Accurate Performance Profiles**
  - Hotspot (Statistical call tree)
  - Call counts (Statistical)
  - Hardware-Event Sampling
- **Thread Profiling**
  - Visualize thread interactions on timeline
  - Balance workloads
- **Easy set-up**
  - Pre-defined performance profiles
  - Use a normal production build
- **Find Answers Fast**
  - Filter extraneous data
  - View results on the source / assembly
- **Compatible**
  - Microsoft, GCC, Intel compilers
  - C/C++, Fortran, Assembly, .NET, Java
  - Latest Intel® processors and compatible processors[1]
- **Windows or Linux**
  - Visual Studio Integration (Windows)
  - Standalone user i/f and command line
  - 32 and 64-bit



[1] IA32 and Intel® 64 architectures.
Many features work with compatible processors.
Event based sampling requires a genuine Intel® Processor.

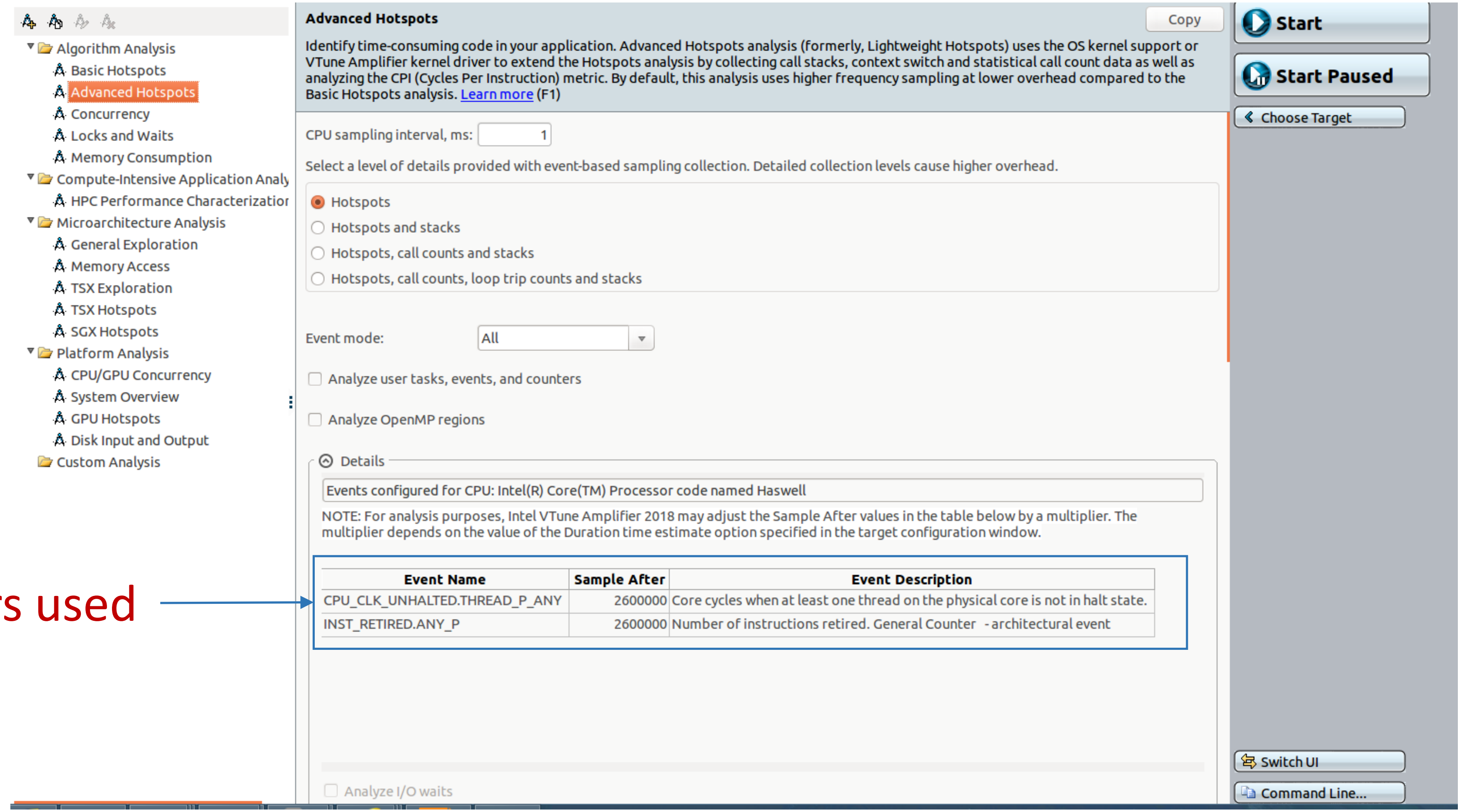
INTEL® VTUNE™ AMPLIFIER XE

Analysis Types (based on technology)

Software Collector Any x86 processor, any virtual, no driver	**Hardware Collector** Higher res., lower overhead, system wide
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** Dig deep to tune bandwidth, cache misses, access contention, etc.

INTEL® VTUNE™ AMPLIFIER XE

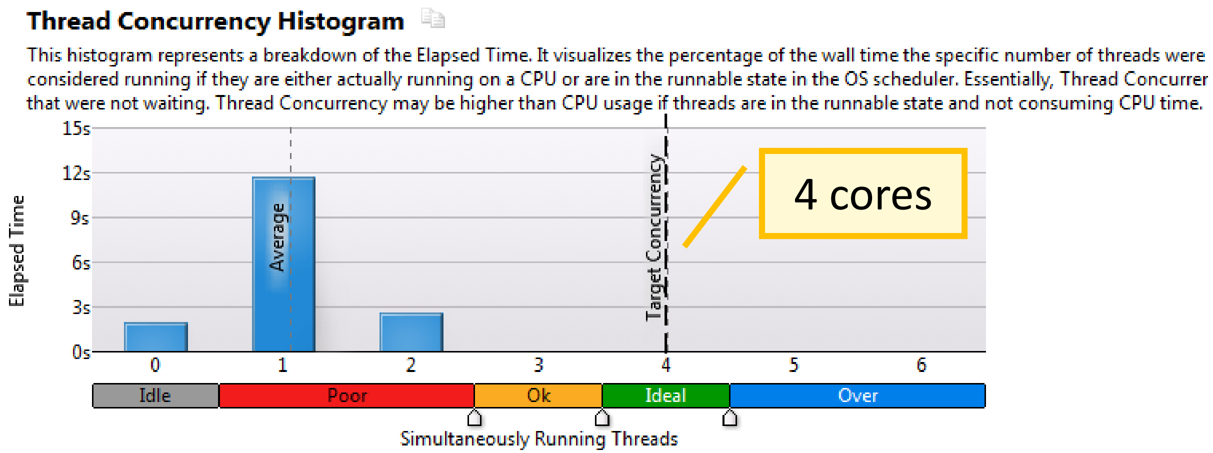
Software or hardware collector ?

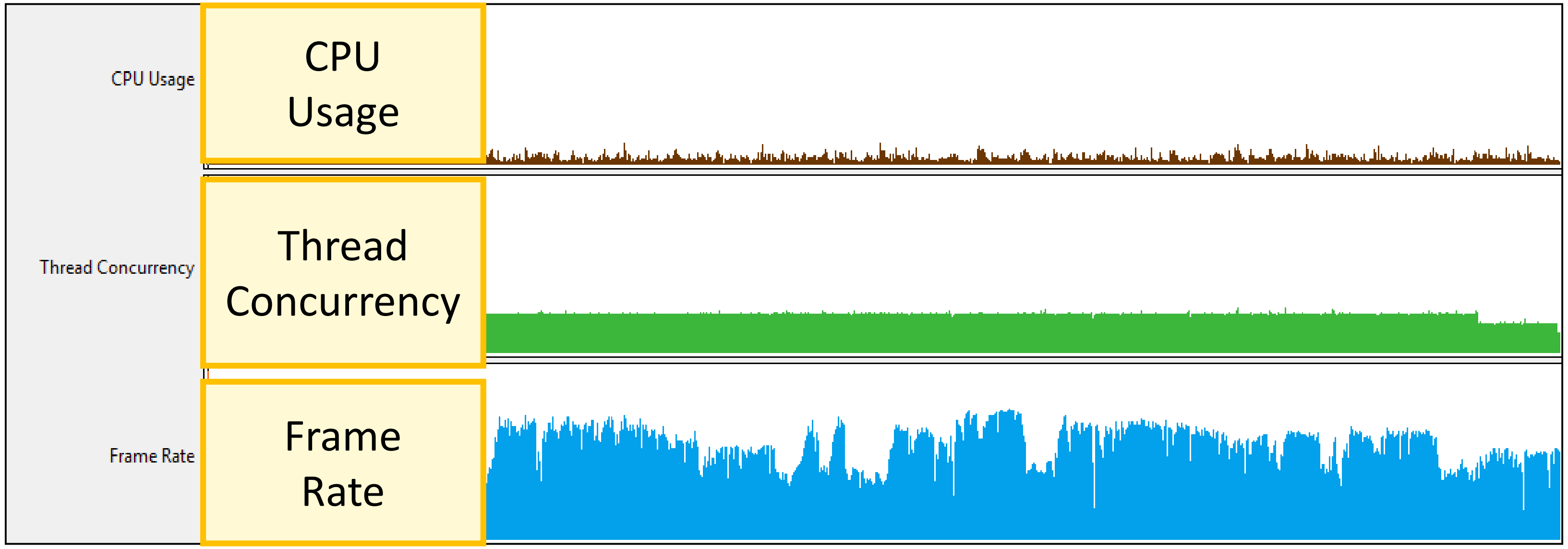


List of hardware counters used

INTEL® VTUNE™ AMPLIFIER XE

Get a quick snapshot





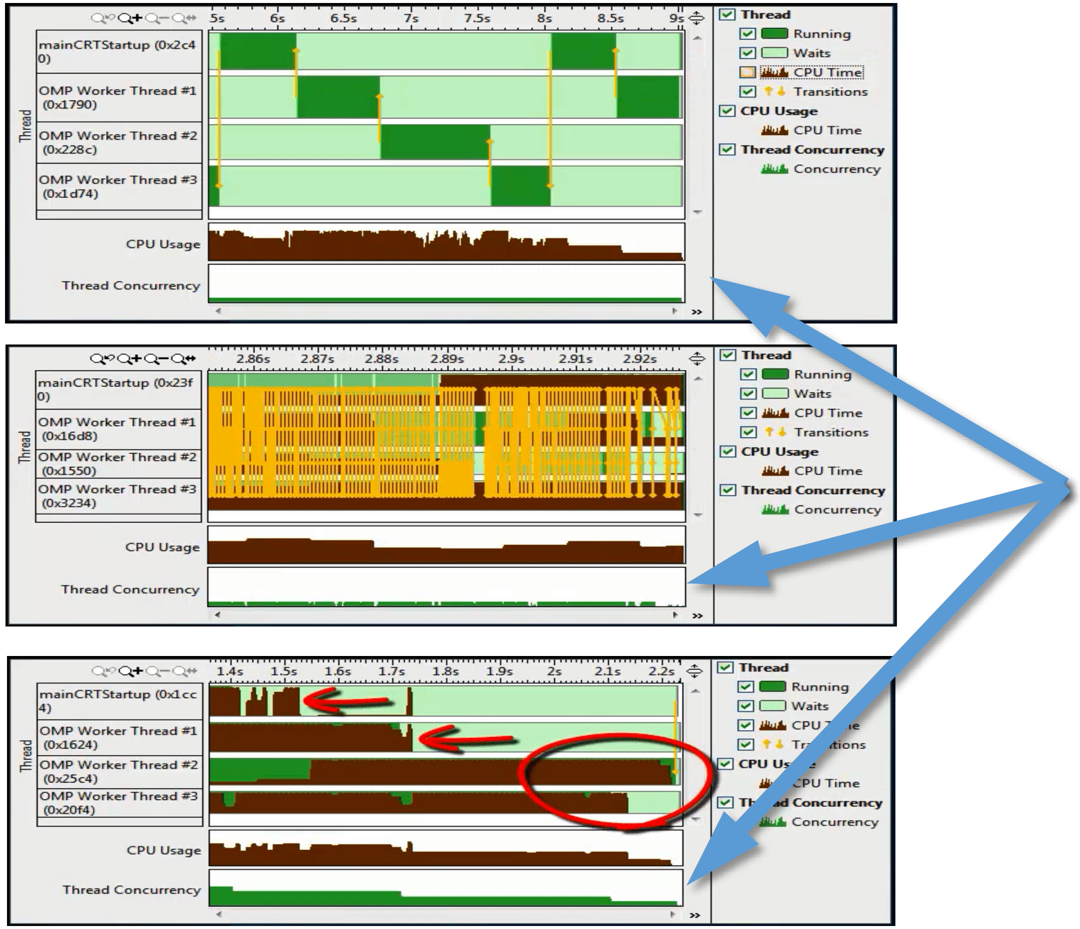
INTEL® VTUNE™ AMPLIFIER XE

Look for Common Patterns

Coarse Grain Locks

High Lock Contention

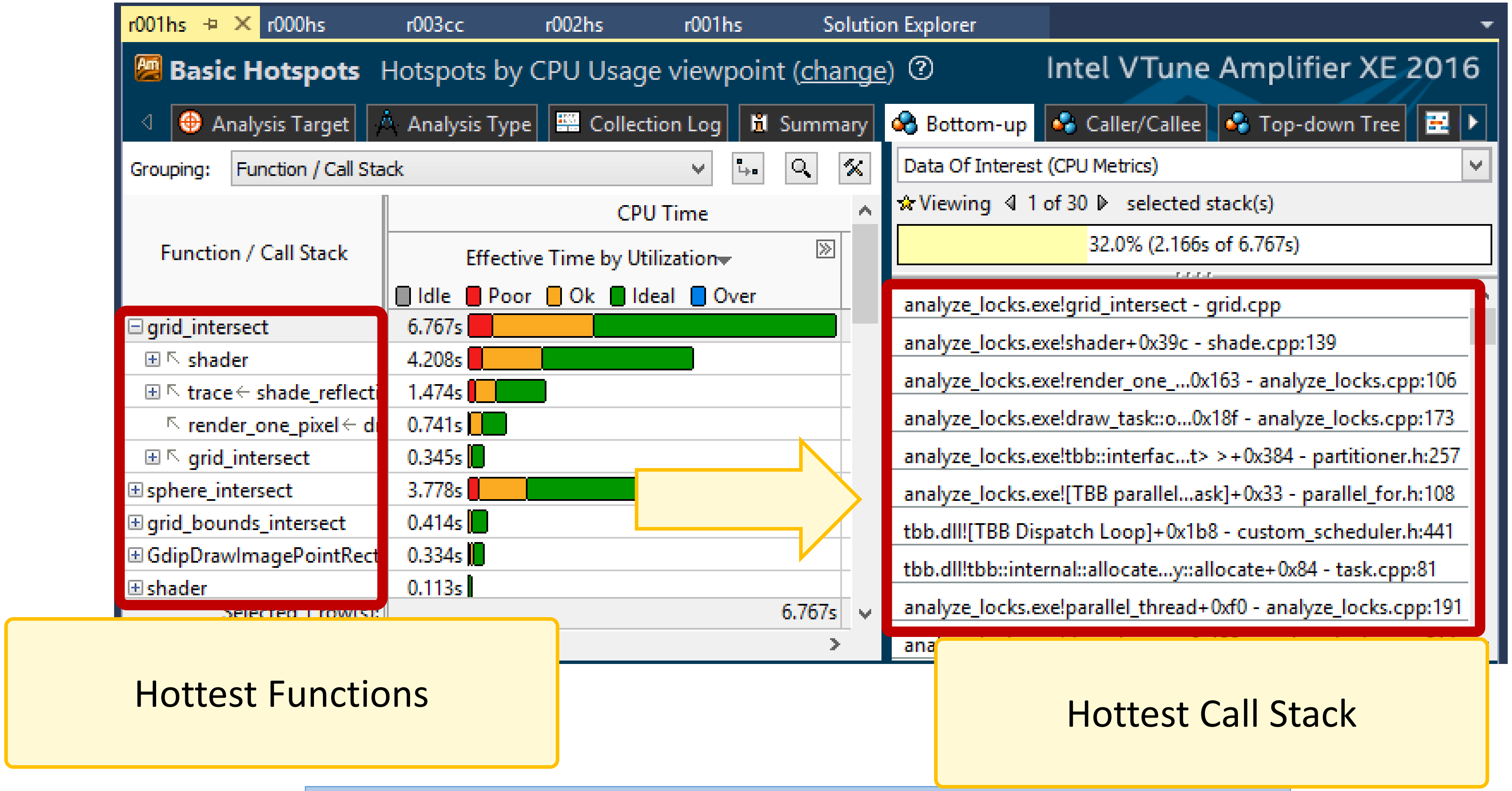
Load Imbalance



Low Concurrency

INTEL® VTUNE™ AMPLIFIER XE

Identify hotspots



Quickly identify what is important

INTEL® VTUNE™ AMPLIFIER XE

Find Answers Fast

Adjust Data Grouping

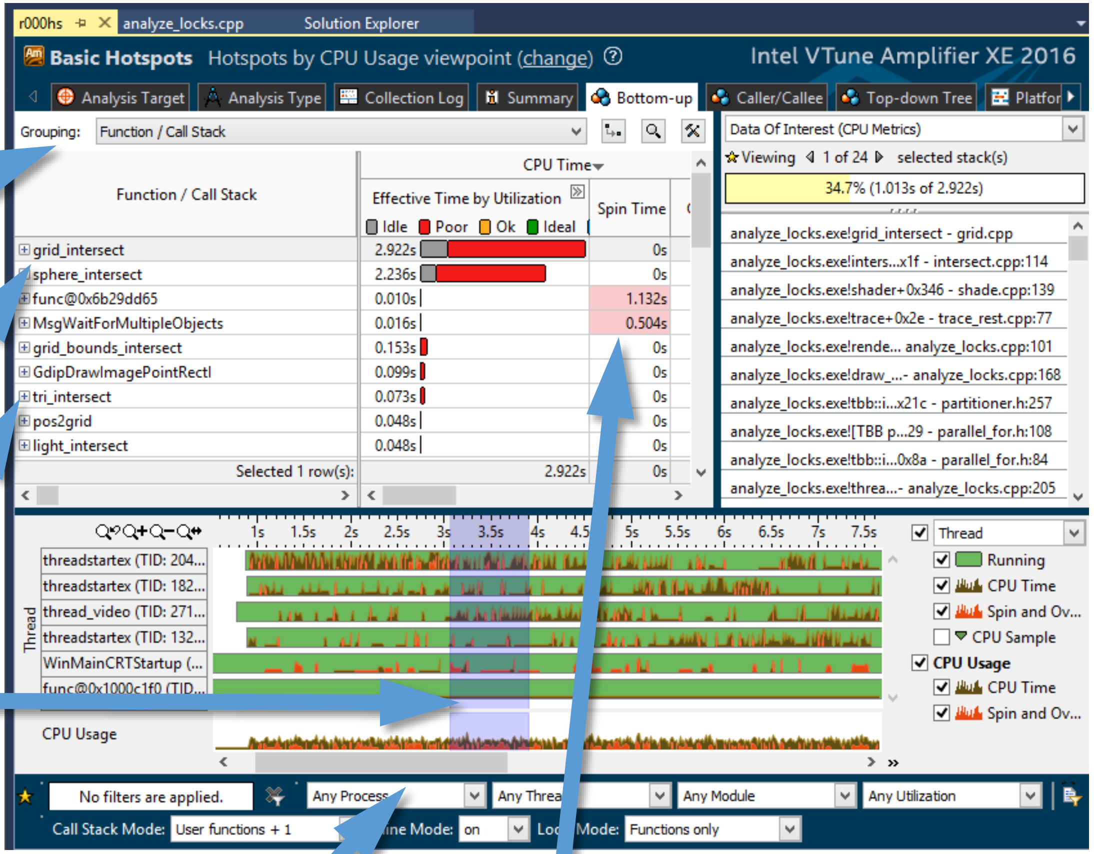
Function - Call Stack
Module - Function - Call Stack
Source File - Function - Call Stack
Thread - Function - Call Stack
... (Partial list shown)

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

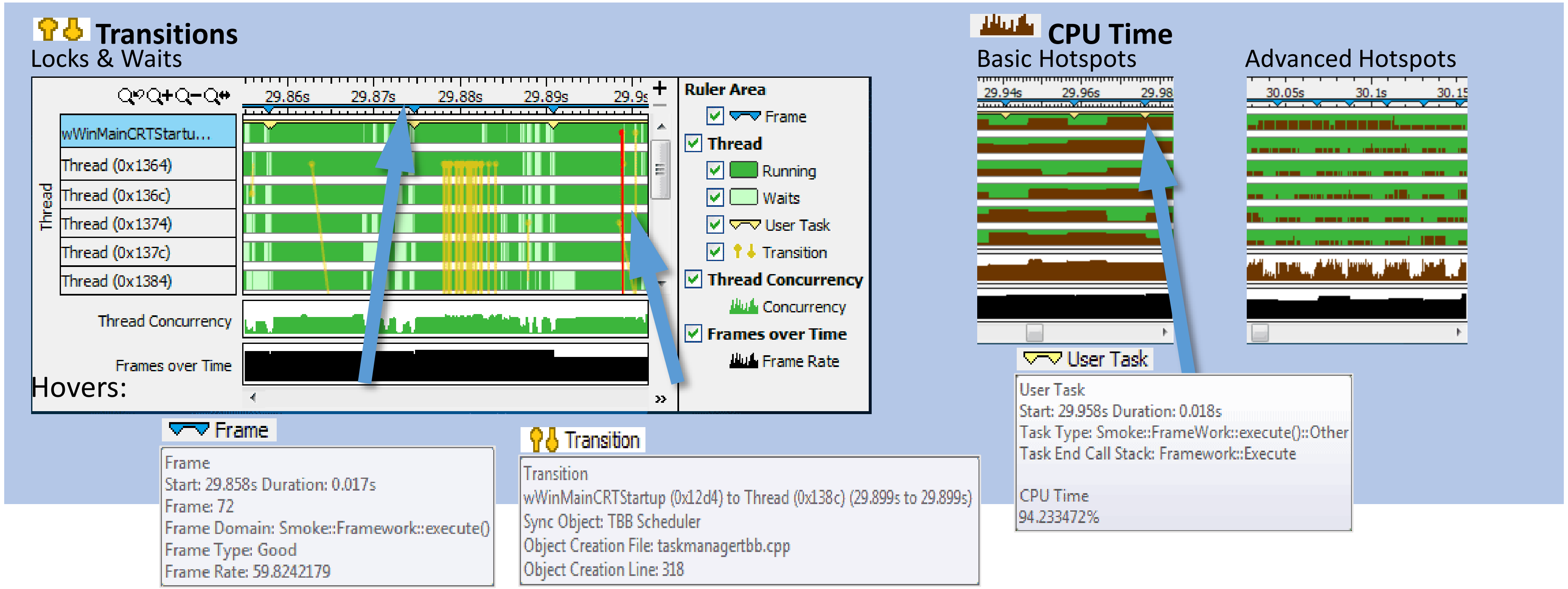


Filter by Process & Other Controls

Tuning Opportunities Shown in Pink. Hover for Tips

INTEL® VTUNE™ AMPLIFIER XE

Timeline Visualizes Thread Behavior



- Optional: Use API to mark frames and user tasks
- Optional: Add a mark during collection

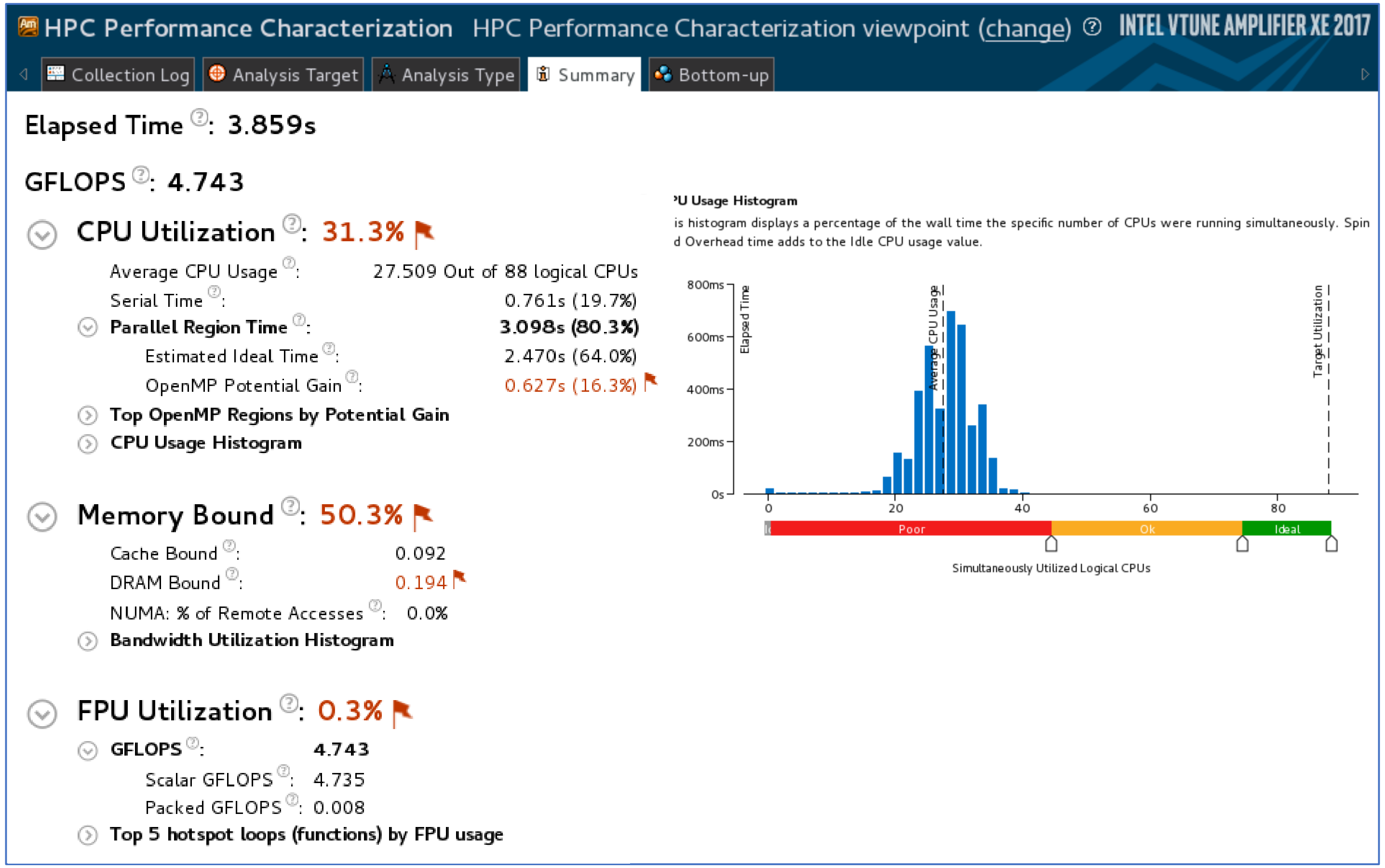
For more complete information about compiler optimizations, see our Optimization Notice.

New!

THREE KEYS TO HPC PERFORMANCE

Threading, Memory Access, Vectorization – Intel VTune™ Amplifier

- **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



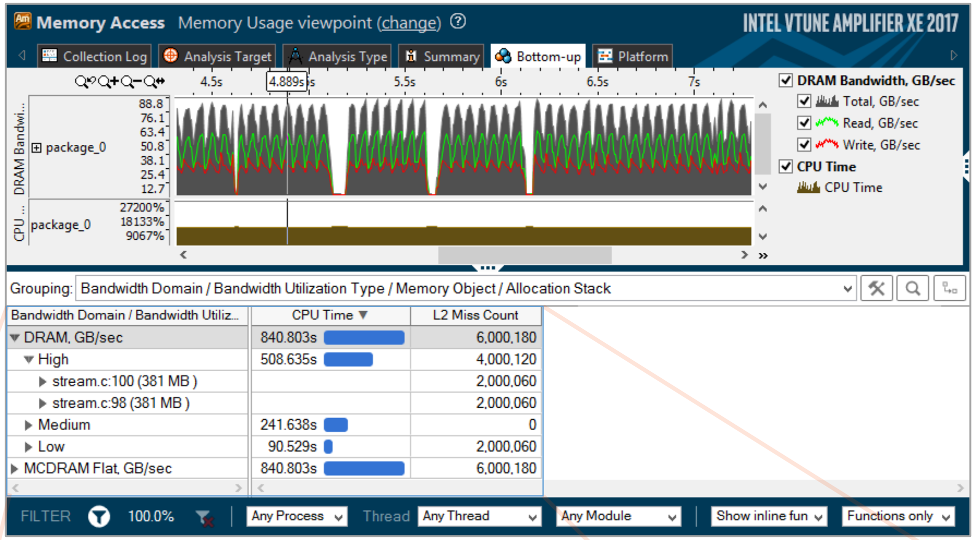
† For 3rd, 5th, 6th Generation Intel® Core™ processors and second generation Intel® Xeon Phi™ processor code named Knights Landing.

Improved!

OPTIMIZE MEMORY ACCESS

Memory Access Analysis - Intel® VTune™ Amplifier 2017

- 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

User API

Enable you to

- control collection
- set marks during the execution of the specific code
- specify custom synchronization primitives implemented without standard system APIs

To use the user APIs, do the following:

- Include ittnotify.h, located at <install_dir>/include
- Insert __itt_* notifications in your code
- Link to the libittnotify.lib file located at <install_dir>/lib

User API

Collection control and threads naming

Collection Control APIs

void __**itt_pause** (void)	Run the application without collecting data. VTune™ Amplifier XE reduces the overhead of collection, by collecting only critical information, such as thread and process creation.
void __**itt_resume** (void)	Resume data collection. VTune™ Amplifier XE resumes collecting all data.

Thread naming APIs

void __**itt_thread_set_name** (const __itt_char *name)	Set thread name using char or Unicode string, where *name* is the thread name.
void __**itt_thread_ignore** (void)	Indicate that this thread should be ignored from analysis. It will not affect the concurrency of the application. It will not be visible in the Timeline pane.

User API

Collection Control Example

```
int main(int argc, char* argv[])
{
        doSomeInitializationWork();

        __itt_resume();
        while(gRunning) {
                doSomeDataParallelWork();
        }
        __itt_pause();

        doSomeFinalizationWork();
        return 0;
}
```

AGENDA

- Which tools for threading and scalability ?
- Intel® Performance Snapshot
- Intel® VTune™ Amplifier
- **Some examples and solutions**
- What’s new in 2018 ?

Fibonacci and scheduling

Thread scheduling issue with OMP

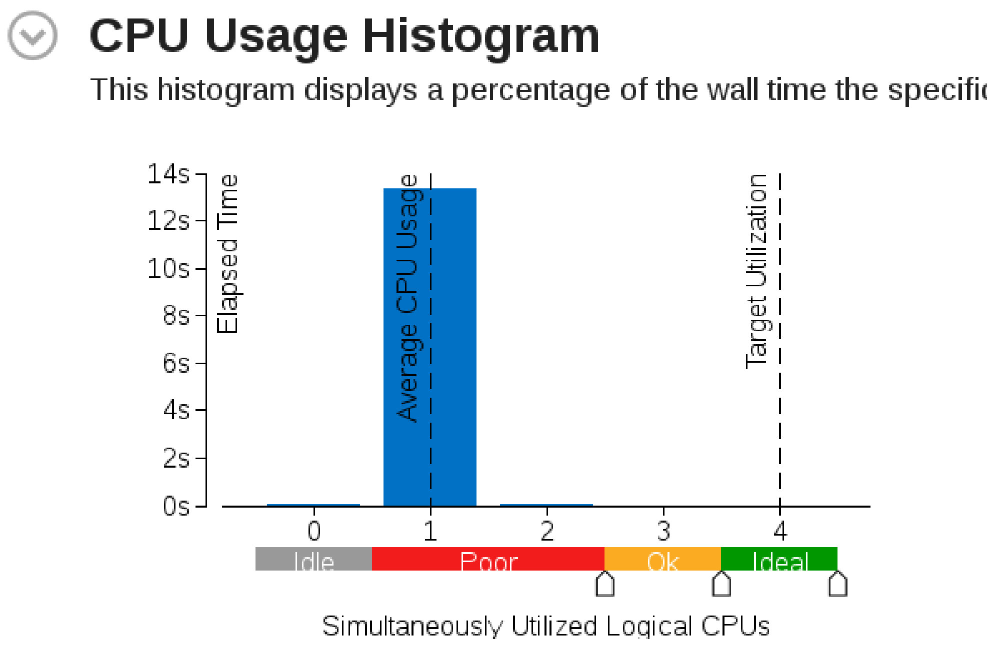
- Very naïve implementation (just want to show a common pattern)
  - We want to fill an array with numbers from the Fibonacci suite

```
#pragma omp parallel for
for(int i=0; i<SIZE; i++){
  fib_array[i] = fib(i);
}
```

By default, OMP uses a static scheduling.
Each thread will do the same number of iterations

```
int fib(int i){
        if(i==0) return 0;
        if(i==1) return 1;
        return fib(i-1) + fib(i-2);
}
```

Thread scheduling issue with OMP



Very poor threading
Fib(0) is much faster to compute than Fib(50) !!!!
A static scheduling creates a very high Load imbalance.

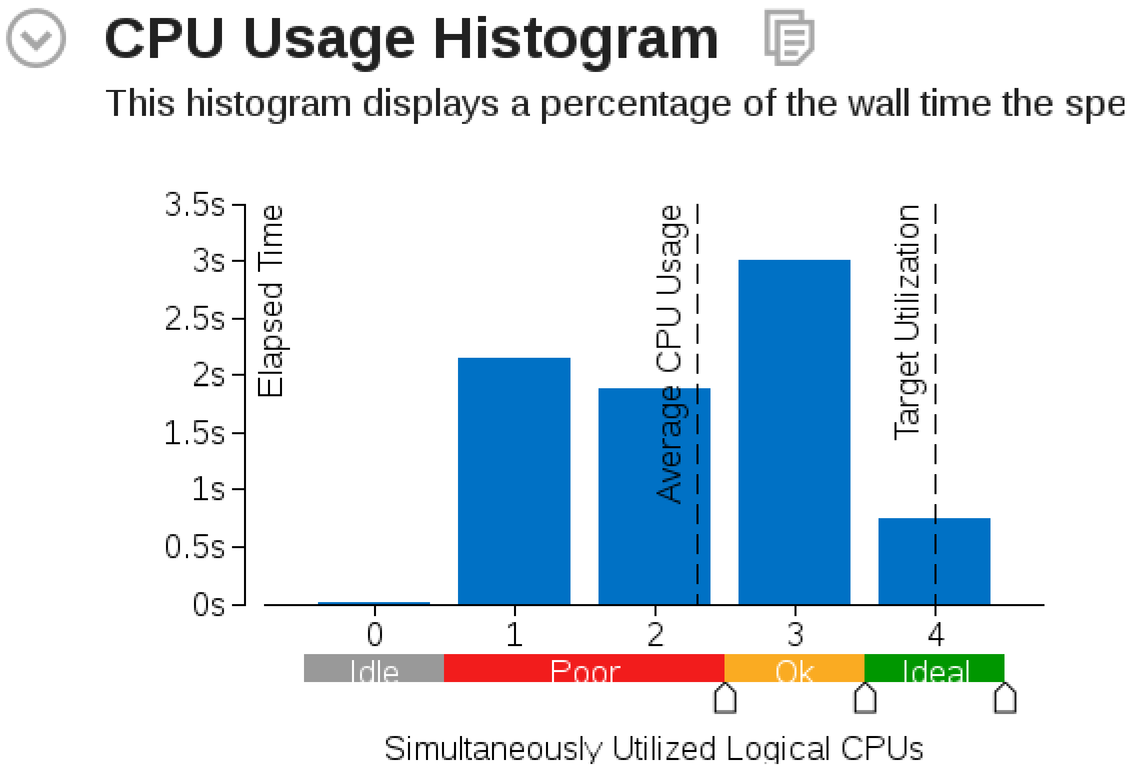
Thread scheduling issue with OMP

- Very naïve implementation (just want to show a common pattern)
  - We want to fill an array with numbers from the Fibonacci suite

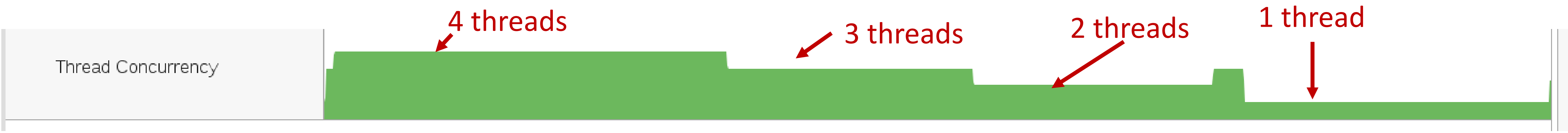
```
#pragma omp parallel for schedule(guided)
for(int i=0; i<SIZE; i++){
  fib_array[i] = fib(i);
}
```

```
int fib(int i){
        if(i==0) return 0;
        if(i==1) return 1;
        return fib(i-1) + fib(i-2);
}
```

Thread scheduling issue with OMP



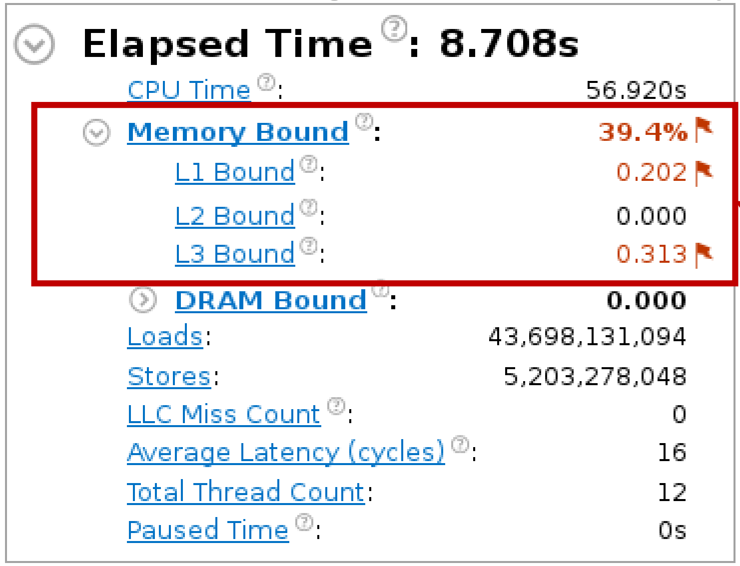
Just changing the scheduling provides an important speedup, around 2x for Fib(50)



Linear regression and false sharing identification

What is false sharing ?

- 2 or more threads reading/writing the same cache line
  - At least 1 thread is writing data
  - Other threads want to read another data in the same cache line
- Linear regression sample (available in Vtune's package)



Running the memory analysis shows a bottleneck on the L1 cache system.

What is false sharing ?

1- Look for memory object responsible for latency

Top Memory Objects by Latency

Memory Object	Total Latency	Loads	Stores	LLC Miss Count
linear_regression_pthread .c:136 (512 B)	64.4%	14,058,042,174	4,998,074,970	0
[Unknown]	28.7%	19,104,057,312	202,803,042	0
linear_regression_pthread .c:118 (54 MB)	6.9%	10,536,031,608	0	0
[Stack]	0.0%	0	2,400,036	0

2- Identify allocation site, object size and average latency

Grouping: Memory Object / Function / Allocation Stack

Memory Object / Function / Allocation Stack	Loads ▼	Stores	Average Latency (cyc...
[Unknown]	19,104,057,312	202,803,042	8
linear_regression_pthread.c:136 (512 B)	14,058,042,174	4,998,074,970	37
linear_regression_pthread.c:118 (54 MB)	10,536,031,608	0	8
[Stack]	0	2,400,036	0

3- Look into the code

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

Source | Assembly | Assembly grouping: Address

```
S ▲  Source
135      req_units = n / num_threads;
136      tid_args = (lreg_args *)calloc(sizeof(lreg_args), num_procs);
137
138        // Assign a portion of the points for each thread
139      for(i = 0; i < num_threads; i++)
140      {
141          tid_args[i].points = &points[i*req_units];
142          tid_args[i].num_elems = req_units;
143          if(i == (num_threads - 1))
144              tid_args[i].num_elems = n - i*req_units;
145
146          CHECK_ERROR(pthread_create(&tid_args[i].tid, &attr, linear_regressi
```

```
typedef struct
{
    pthread_t tid;
    POINT_T *points;
    int num_elems;
    long long SX;
    long long SY;
    long long SXX;
    long long SYY;
    long long SXY;
} lreg_args;
```

This structure seems to be responsible

What is false sharing ?

Cache line

SX	SY	SXX	SYY	tid	*points	Num_elem	SX	SY	SXX	SYY	SXY	tid

Thread 0

New struct starts here

Thread 1

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

Source | Assembly | Assembly grouping: Address

S	Source	Loads
76	// ADD UP RESULTS	
77	for (i = 0; i < args->num_elems; i++)	1,242,003,726
78	{	
79	//Compute SX, SY, SYY, SXX, SXY	
80	args->SX += args->points[i].x;	714,002,142
81	args->SXX += args->points[i].x*args->points[i].x;	2,754,008,262
82	args->SY += args->points[i].y;	1,830,005,490
83	args->SYY += args->points[i].y*args->points[i].y;	3,912,011,736
84	args->SXY += args->points[i].x*args->points[i].y;	3,606,010,818
85	}	

Here the structure is 64bytes (same as cache line)

But depending on alignment, 2 lreg_args objects can Share the same cache line.

```
typedef struct
{
    pthread_t tid;
    POINT_T *points;
    int num_elems;
    long long SX;
    long long SY;
    long long SXX;
    long long SYY;
    long long SXY;
} lreg_args;
```

What is false sharing ?

- To solve the false sharing, we can add an array that will pad our structure and avoid having data of 2 lreg_args objects sharing the same cache line.

```
typedef struct
{
    char pad[80];
    pthread_t tid;
    POINT_T *points;
    int num_elems;
    long long SX;
    long long SY;
    long long SXX;
    long long SYY;
    long long SXY;
} lreg_args;
```

Bonus, not explained in the sample !

In this test, aligning the data to a 64 bytes boundary can also solve the problem !

AGENDA

- Which tools for threading and scalability ?
- Intel® Performance Snapshot
- Intel® VTune™ Amplifier
- Some examples and solutions
- **What's new in 2018 ?**

APPLICATION PERFORMANCE SNAPSHOT ADDS MPI

All the data in one place: MPI + OpenMP + Memory + Floating Point

Quick & easy performance overview

- Does the app need performance tuning?

MPI and non-MPI Apps†

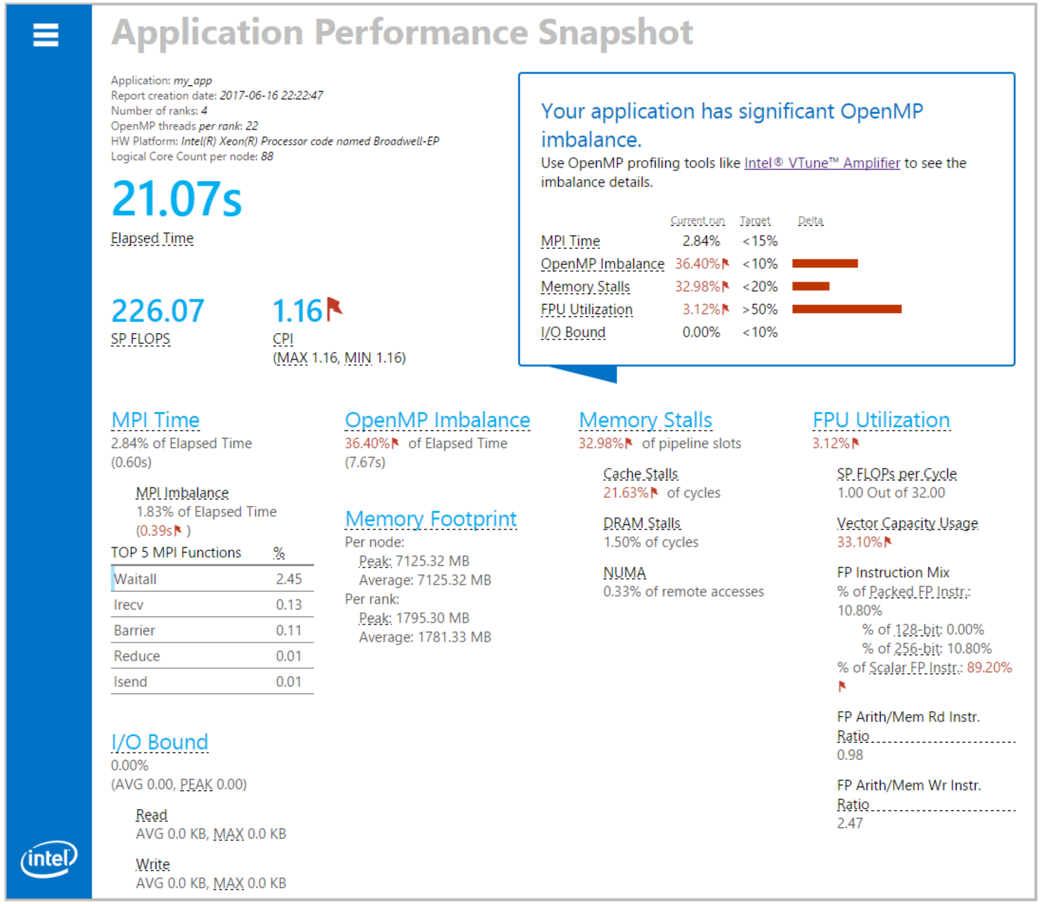
- Distributed MPI with or without threading
- Shared memory applications

Popular MPI implementations supported

- Intel® MPI
- MPICH and Cray MPI

Richer metrics on computation efficiency

- CPU (processor stalls, memory access)
- FPU (vectorization metrics)



† MPI supported only on Linux*.

MORE COMPLETE HPC PERFORMANCE OVERVIEW

MPI metrics added to HPC analysis

MPI Imbalance Metric

- Metric for performance of rank on critical path
- Computational bottlenecks and outlier rank behavior now available in VTune Amplifier
- For communication pattern problems between ranks use Intel® Trace Analyzer and Collector (ITAC)

Threading: CPU Utilization

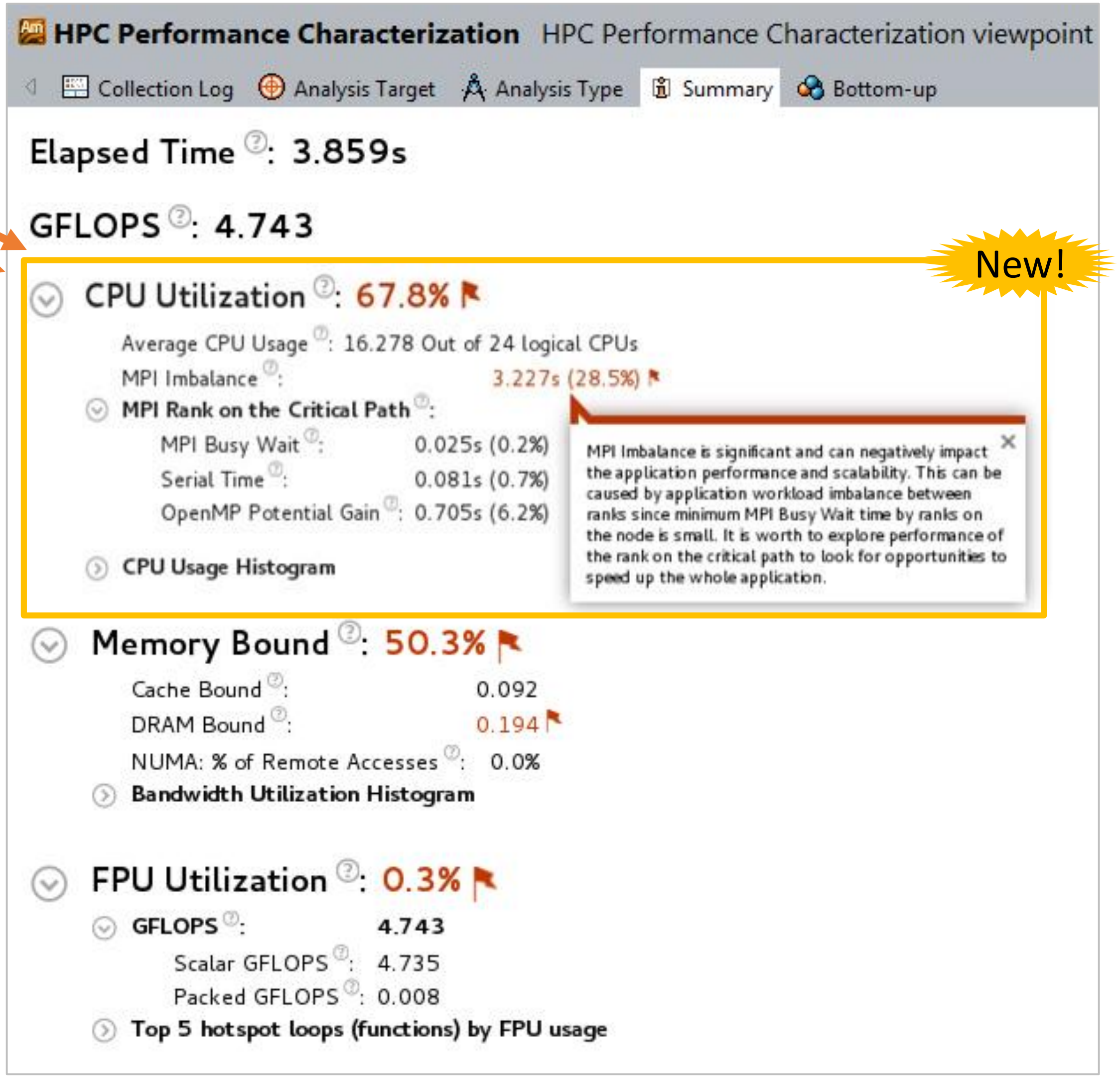
- Serial vs. Parallel time
- Top OpenMP regions by potential gain
- Tip: Use hotspot OpenMP region analysis for more detail

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.

WHAT'S USING ALL THE MEMORY?

Memory Consumption Analysis

Preview!

See What Is Allocating Memory

- Lists top memory consuming functions and objects
- View source to understand cause
- Filter by time using the memory consumption timeline

• Standard & Custom Allocators

- Recognizes libc malloc/free, memkind and jemalloc libraries
- Use custom allocators after markup with ITT Notify API

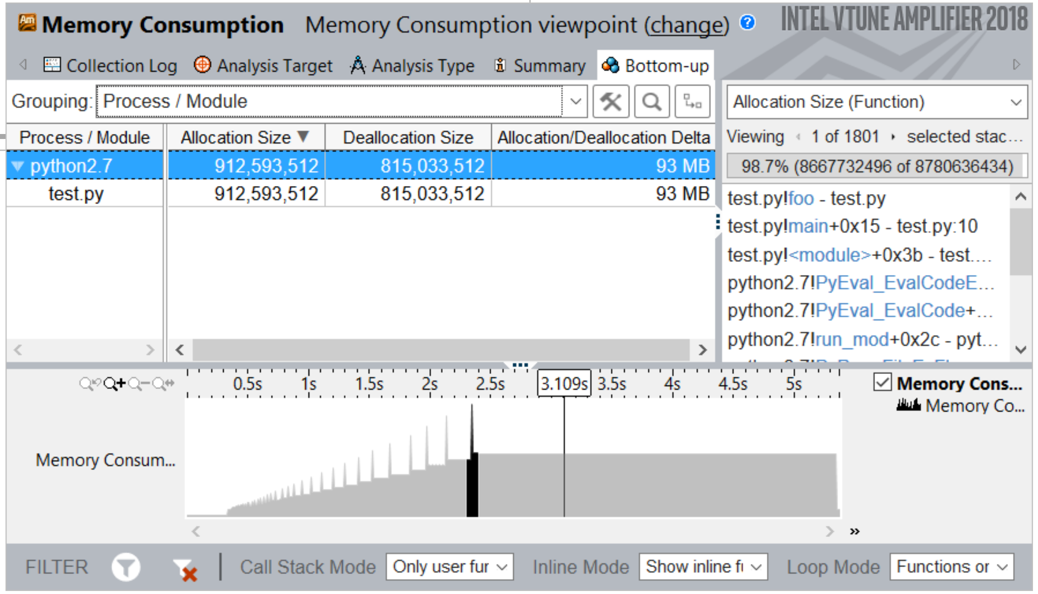
Languages

- Python*
- Linux*: Native C, C++, Fortran

Top Memory-Consuming Objects

This section lists the most memory-consuming objects in your application. Optimizing these objects results in improving an overall application memory consumption.

Memory Object	Memory Consumption
dictobject.c:632 (768 B)	768 B
filedoalloc.c:120 (4 KB)	4 KB
iofopen.c:76 (568 B)	568 B
msort.c:224 (1 KB)	
dictobject.c:632 (3 KB)	
[Others]	



Native language support is not currently available for Windows*

OPTIMIZE PRIVATE CLOUD-BASED APPLICATIONS

Profile native & Java apps in containers

Profile Enterprise Applications

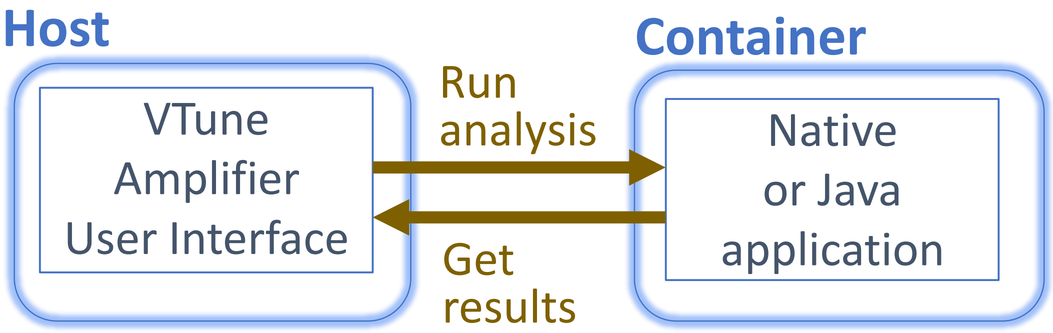
- Native C, C++, Fortran
- Attach to running Java services (e.g., Mail)
- Profile Java daemons without restart

• Accurate low-overhead data collection

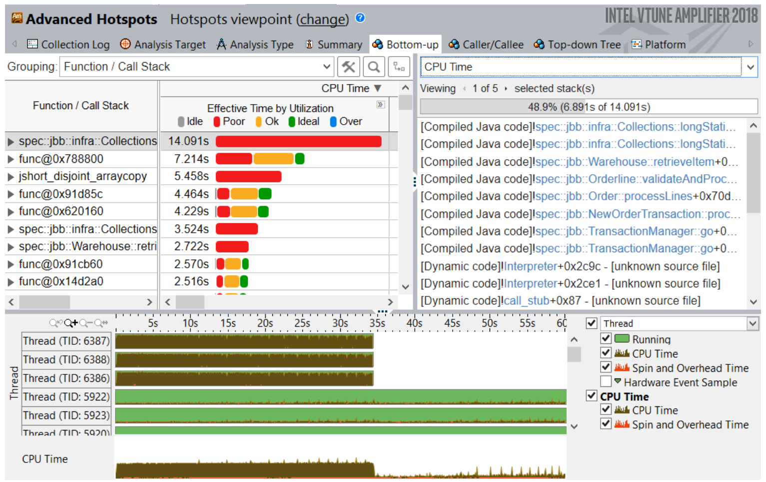
- Advanced hotspots and hardware events
- Memory analysis
- Accurate stack information for Java and HHVM

Popular containers supported

- Docker* ▪ Mesos* ▪ LXC*



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



Software collectors (e.g. Locks & Waits) and Python profiling are not currently available for containers.

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.
-

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