INTEL® VTUNE™ AMPLIFIER

PERFORMANCE PROFILER

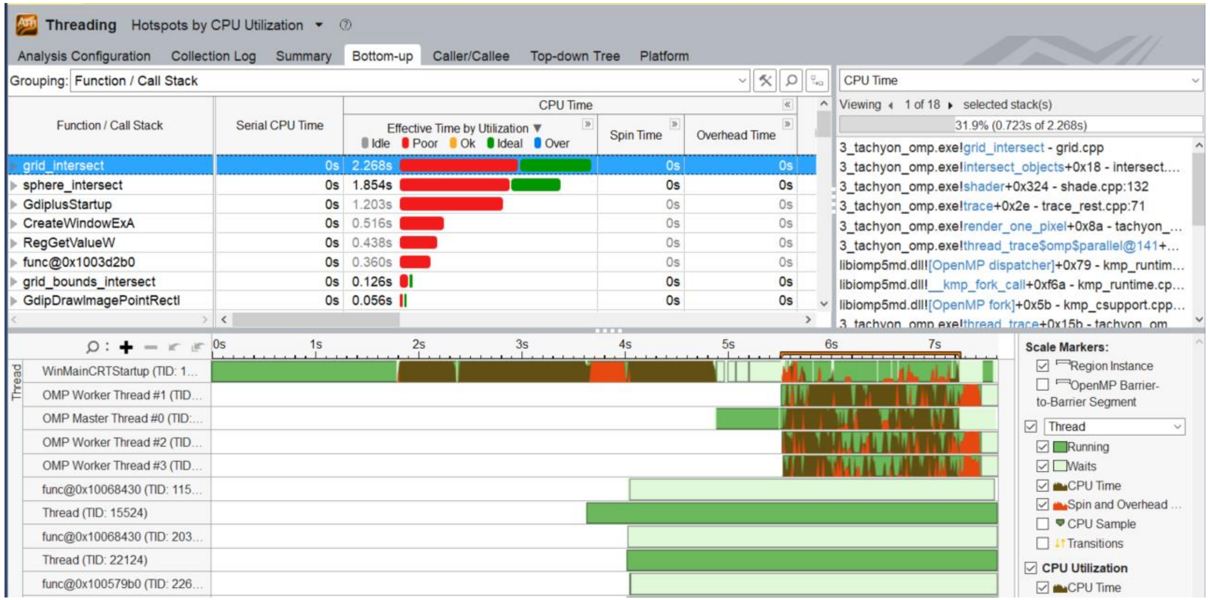
Kevin O'Leary – Intel Developer Products Division

Intel® VTune™ Amplifier

Tune Applications for Scalable Multicore Performance

Agenda

- Introduction
- Data Collection – Rich set of performance data
- Data Analysis - Find answers fast
- Flexible workflow –
  - User i/f and command line
  - Compare results
  - Remote collection
- Performance Analysis Details
- Summary


Threading
Hotspots by CPU Utilization
Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Platform
Grouping: Function / Call Stack
CPU Time
Scale Markers:
Region Instance
OpenMP Barrier-to-Barrier Segment
Thread
Running
Waits
CPU Time
Spin and Overhead ...
CPU Sample
Transitions
CPU Utilization
CPU Time
Thread
WinMainCRTStartup (TID: 1...
OMP Worker Thread #1 (TID...
OMP Master Thread #0 (TID:...
OMP Worker Thread #2 (TID...
OMP Worker Thread #3 (TID...
func@0x10068430 (TID: 115...
Thread (TID: 15524)
func@0x10068430 (TID: 203...
Thread (TID: 22124)
func@0x100579b0 (TID: 226...
0s
1s
2s
3s
4s
5s
6s
7s

Function / Call Stack	Serial CPU Time	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Spin Time	Overhead Time
grid_intersect	0s	2.268s	0s	0s
sphere_intersect	0s	1.854s	0s	0s
GdiplusStartup	0s	1.203s	0s	0s
CreateWindowExA	0s	0.516s	0s	0s
RegGetValueW	0s	0.438s	0s	0s
func@0x1003d2b0	0s	0.360s	0s	0s
grid_bounds_intersect	0s	0.126s	0s	0s
GdipDrawImagePointRectI	0s	0.056s	0s	0s

Viewing 1 of 18 selected stack(s)
31.9% (0.723s of 2.268s)

```
3_tachyon_omp.exe!grid_intersect - grid.cpp
3_tachyon_omp.exe!intersect_objects+0x18 - intersect....
3_tachyon_omp.exe!shader+0x324 - shade.cpp:132
3_tachyon_omp.exe!trace+0x2e - trace_rest.cpp:71
3_tachyon_omp.exe!render_one_pixel+0x8a - tachyon_...
3_tachyon_omp.exe!thread_trace$omp$parallel@141+...
libiomp5md.dll![OpenMP dispatcher]+0x79 - kmp_runtim...
libiomp5md.dll!__kmp_fork_call+0xf6a - kmp_runtime.cp...
libiomp5md.dll![OpenMP fork]+0x5b - kmp_csupport.cpp...
3_tachyon_omp.exe!thread_trace+0x15b - tachyon_om
```

Faster, Scalable Code, Faster

Intel® VTune™ Amplifier Performance Profiler

Accurate Data - Low Overhead

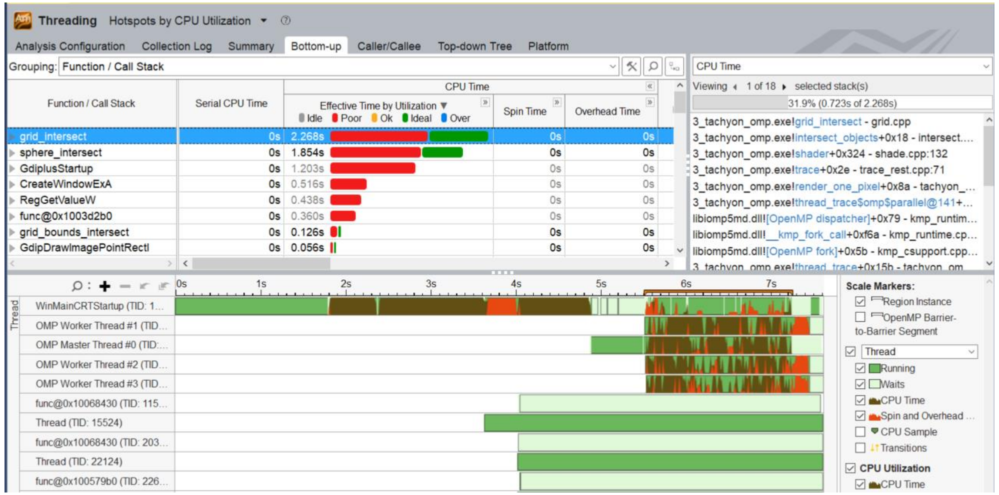
- CPU, GPU, FPU, threading, bandwidth…

Meaningful Analysis

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

Easy

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



Function / Call Stack	Serial CPU Time	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	CPU Time: Spin Time	CPU Time: Overhead Time
grid_intersect	0s	2.268s	0s	0s
sphere_intersect	0s	1.854s	0s	0s
GdiplusStartup	0s	1.203s	0s	0s
CreateWindowExA	0s	0.516s	0s	0s
RegGetValueW	0s	0.438s	0s	0s
func@0x1003d2b0	0s	0.360s	0s	0s
grid_bounds_intersect	0s	0.126s	0s	0s
GdipDrawImagePointRectI	0s	0.056s	0s	0s

CPU Time
Viewing 1 of 18 selected stack(s)
31.9% (0.723s of 2.268s)

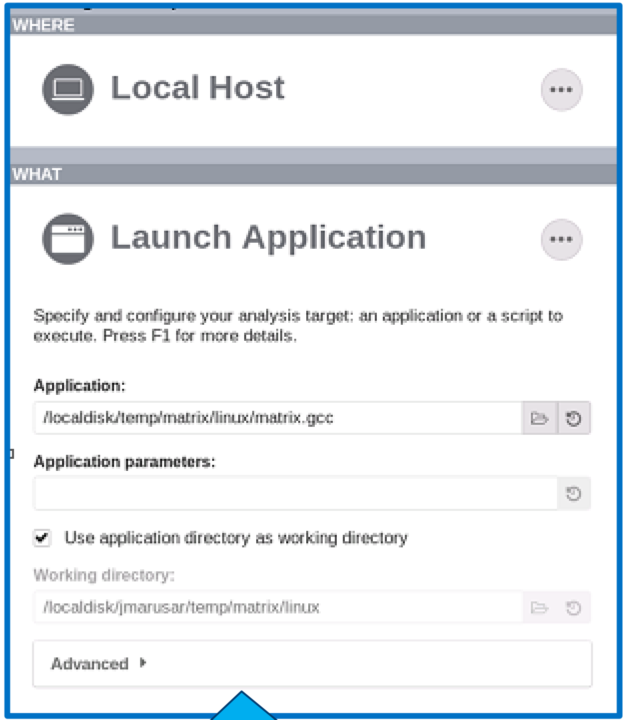
```
3_tachyon_omp.exe!grid_intersect - grid.cpp
3_tachyon_omp.exe!intersect_objects+0x18 - intersect.…
3_tachyon_omp.exe!shader+0x324 - shade.cpp:132
3_tachyon_omp.exe!trace+0x2e - trace_rest.cpp:71
3_tachyon_omp.exe!render_one_pixel+0x8a - tachyon_…
3_tachyon_omp.exe!thread_trace$omp$parallel@141+…
libiomp5md.dll![OpenMP dispatcher]+0x79 - kmp_runtim…
libiomp5md.dll!__kmp_fork_call+0xf6a - kmp_runtime.cp…
libiomp5md.dll![OpenMP fork]+0x5b - kmp_csupport.cpp…
3_tachyon_omp.exe!thread_trace+0x15b - tachyon_om
```

0s
1s
2s
3s
4s
5s
6s
7s
Thread
WinMainCRTStartup (TID: 1…
OMP Worker Thread #1 (TID…
OMP Master Thread #0 (TID:…
OMP Worker Thread #2 (TID…
OMP Worker Thread #3 (TID…
func@0x10068430 (TID: 115…
Thread (TID: 15524)
func@0x10068430 (TID: 203…
Thread (TID: 22124)
func@0x100579b0 (TID: 226…
Scale Markers:
Region Instance
OpenMP Barrier-to-Barrier Segment
Thread
Running
Waits
CPU Time
Spin and Overhead…
CPU Sample
Transitions
CPU Utilization
CPU Time

"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.

Setting up a profile is easy



WHERE

Local Host

WHAT

Launch Application

Specify and configure your analysis target: an application or a script to execute. Press F1 for more details.

Application: /localdisk/temp/matrix/linux/matrix.gcc

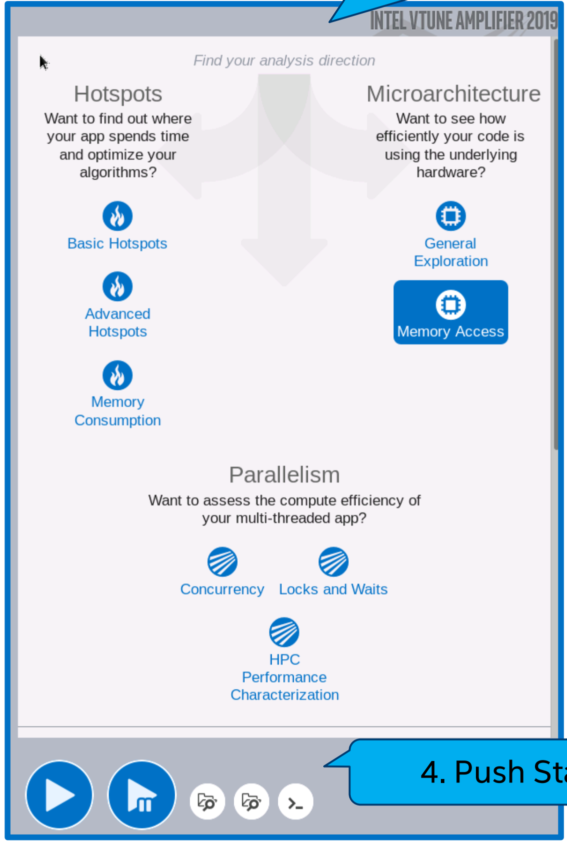
Application parameters:

☑ Use application directory as working directory

Working directory: /localdisk/jmarusar/temp/matrix/linux

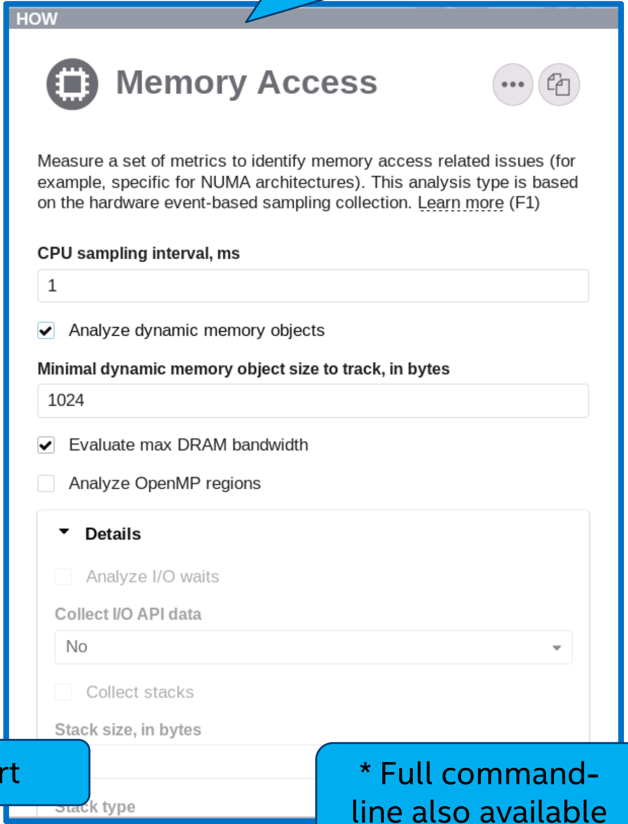
Advanced

1. What/where to profile



INTEL VTUNE AMPLIFIER 2019
Find your analysis direction
Hotspots
Want to find out where your app spends time and optimize your algorithms?
Basic Hotspots
Advanced Hotspots
Memory Consumption
Microarchitecture
Want to see how efficiently your code is using the underlying hardware?
General Exploration
Memory Access
Parallelism
Want to assess the compute efficiency of your multi-threaded app?
Concurrency
Locks and Waits
HPC Performance Characterization

2. Choose Analysis Type



HOW

Memory Access

Measure a set of metrics to identify memory access related issues (for example, specific for NUMA architectures). This analysis type is based on the hardware event-based sampling collection. Learn more (F1)

CPU sampling interval, ms: 1

☑ Analyze dynamic memory objects

Minimal dynamic memory object size to track, in bytes: 1024

☑ Evaluate max DRAM bandwidth

☐ Analyze OpenMP regions

Details

☐ Analyze I/O waits

Collect I/O API data: No

☐ Collect stacks

Stack size, in bytes

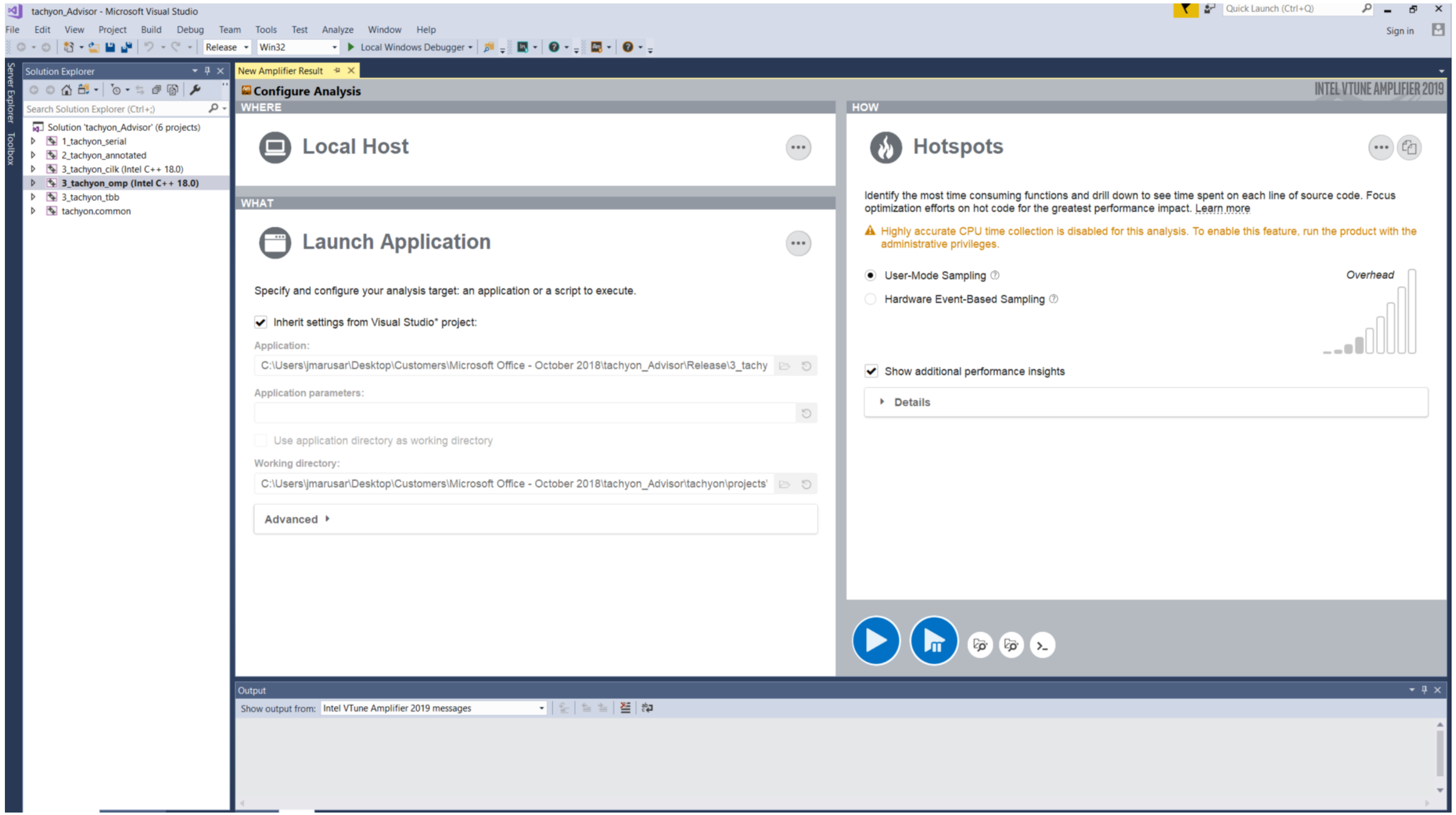
Stack type

3. Collection options

4. Push Start

* Full command-line also available

Full Visual Studio* Integration



tachyon_Advisor - Microsoft Visual Studio

File Edit View Project Build Debug Team Tools Test Analyze Window Help

Release | Win32 | Local Windows Debugger

Solution Explorer

Solution 'tachyon_Advisor' (6 projects)
- 1_tachyon_serial
- 2_tachyon_annotated
- 3_tachyon_cilk (Intel C++ 18.0)
- **3_tachyon_omp (Intel C++ 18.0)**
- 3_tachyon_tbb
- tachyon.common

New Amplifier Result

Configure Analysis

INTEL VTUNE AMPLIFIER 2019

WHERE

Local Host

WHAT

Launch Application

Specify and configure your analysis target: an application or a script to execute.

[✓] Inherit settings from Visual Studio* project:

Application: C:\Users\jmarusar\Desktop\Customers\Microsoft Office - October 2018\tachyon_Advisor\Release\3_tachy

Application parameters:

[ ] Use application directory as working directory

Working directory: C:\Users\jmarusar\Desktop\Customers\Microsoft Office - October 2018\tachyon_Advisor\tachyon\projects'

Advanced

HOW

Hotspots

Identify the most time consuming functions and drill down to see time spent on each line of source code. Focus optimization efforts on hot code for the greatest performance impact. Learn more

⚠ Highly accurate CPU time collection is disabled for this analysis. To enable this feature, run the product with the administrative privileges.

[● User-Mode Sampling]

○ Hardware Event-Based Sampling

Overhead

[✓] Show additional performance insights

Details

Output

Show output from: Intel VTune Amplifier 2019 messages

Two Great Ways to Collect Data

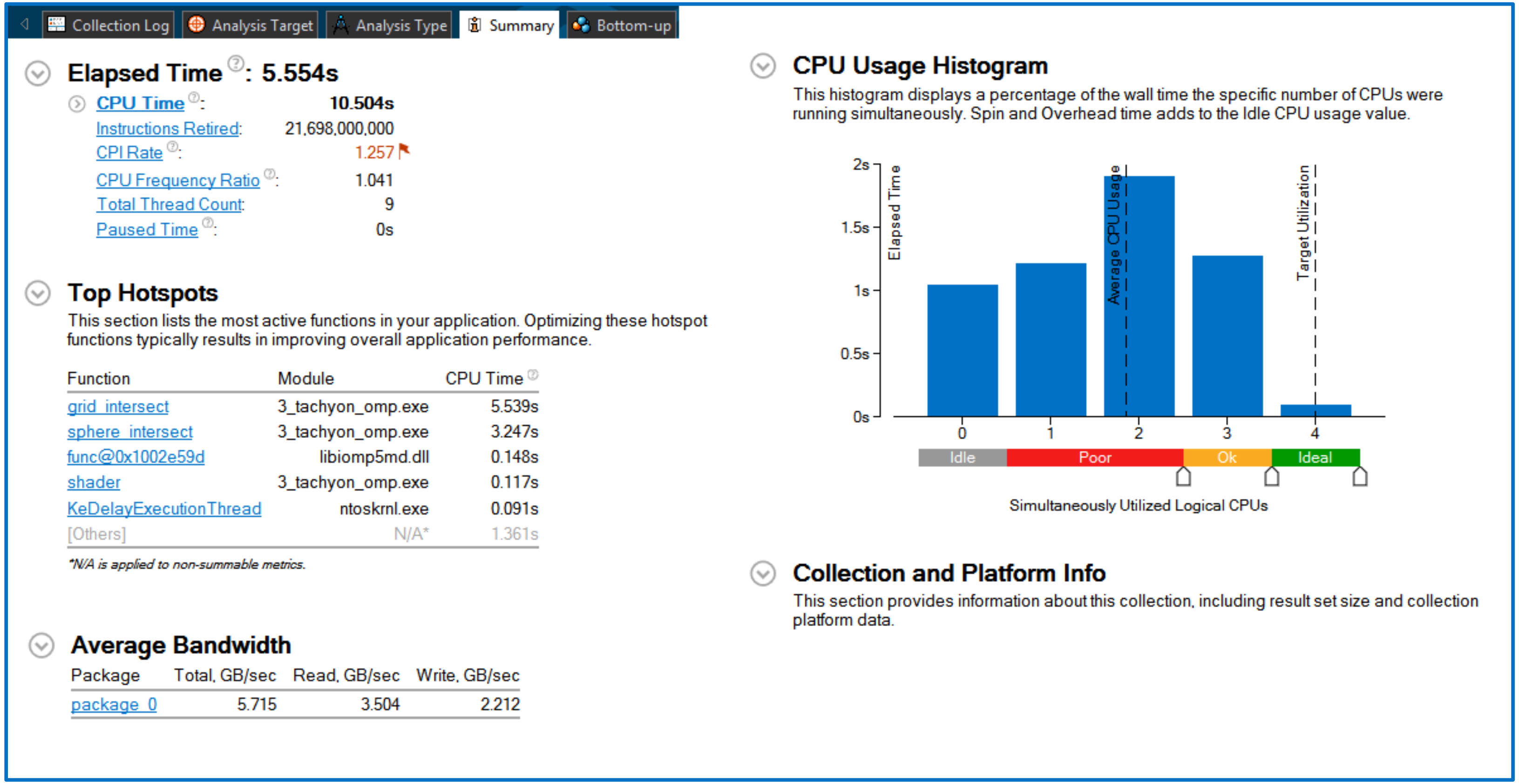
Intel® VTune™ Amplifier

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

Example: Hotspots Analysis

Summary View



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

Elapsed Time: 5.554s

CPU Time:	10.504s
Instructions Retired:	21,698,000,000
CPI Rate:	1.257
CPU Frequency Ratio:	1.041
Total Thread Count:	9
Paused Time:	0s

Top Hotspots

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

Function	Module	CPU Time
grid_intersect	3_tachyon_omp.exe	5.539s
sphere_intersect	3_tachyon_omp.exe	3.247s
func@0x1002e59d	libiomp5md.dll	0.148s
shader	3_tachyon_omp.exe	0.117s
KeDelayExecutionThread	ntoskrnl.exe	0.091s
[Others]	N/A*	1.361s

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

Average Bandwidth

Package	Total, GB/sec	Read, GB/sec	Write, GB/sec
package_0	5.715	3.504	2.212

CPU Usage Histogram

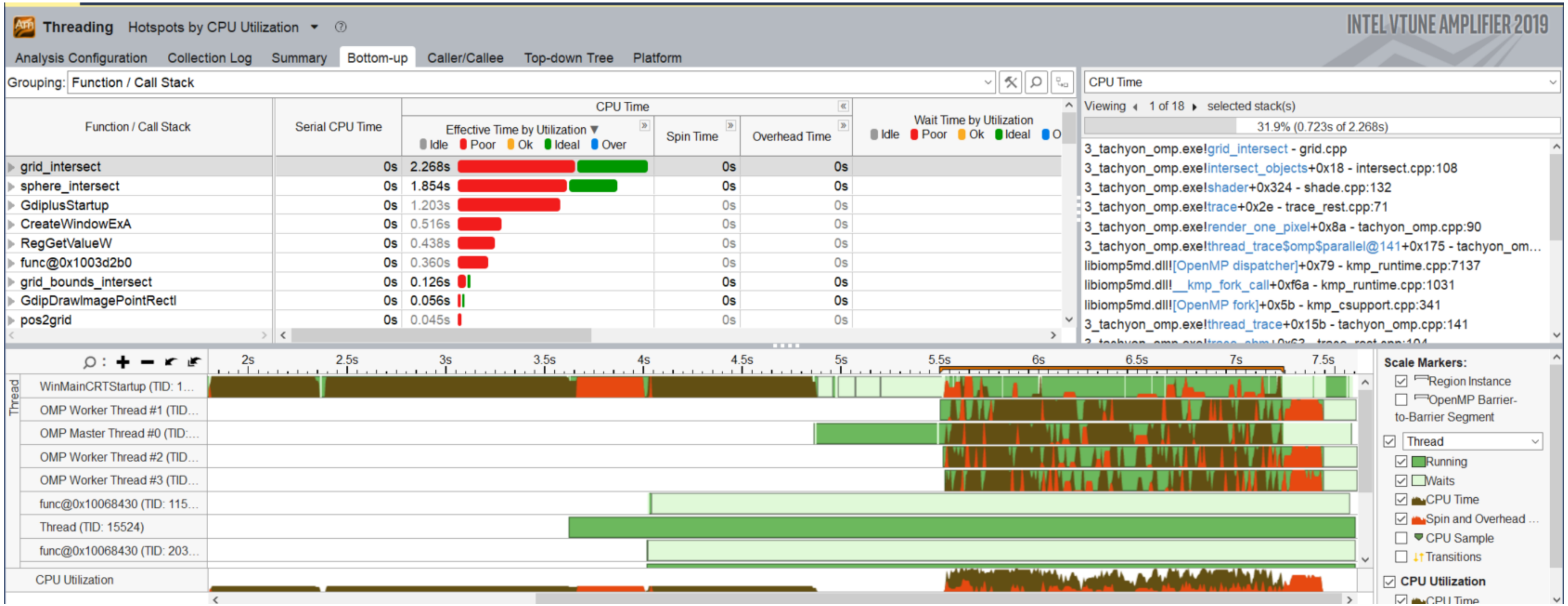
This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.

Collection and Platform Info

This section provides information about this collection, including result set size and collection platform data.

Example: Threading Analysis

Bottom-up View



Grouping: Function / Call Stack

Function / Call Stack	Serial CPU Time	CPU Time: Effective Time by Utilization (🟫 Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over)	Spin Time	Overhead Time	Wait Time by Utilization (🟫 Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 O)
grid_intersect	0s	2.268s	0s	0s	
sphere_intersect	0s	1.854s	0s	0s	
GdiplusStartup	0s	1.203s	0s	0s	
CreateWindowExA	0s	0.516s	0s	0s	
RegGetValueW	0s	0.438s	0s	0s	
func@0x1003d2b0	0s	0.360s	0s	0s	
grid_bounds_intersect	0s	0.126s	0s	0s	
GdipDrawImagePointRectI	0s	0.056s	0s	0s	
pos2grid	0s	0.045s	0s	0s	

CPU Time

Viewing 1 of 18 selected stack(s)

31.9% (0.723s of 2.268s)

```
3_tachyon_omp.exe!grid_intersect - grid.cpp
3_tachyon_omp.exe!intersect_objects+0x18 - intersect.cpp:108
3_tachyon_omp.exe!shader+0x324 - shade.cpp:132
3_tachyon_omp.exe!trace+0x2e - trace_rest.cpp:71
3_tachyon_omp.exe!render_one_pixel+0x8a - tachyon_omp.cpp:90
3_tachyon_omp.exe!thread_trace$omp$parallel@141+0x175 - tachyon_om...
libiomp5md.dll![OpenMP dispatcher]+0x79 - kmp_runtime.cpp:7137
libiomp5md.dll!__kmp_fork_call+0xf6a - kmp_runtime.cpp:1031
libiomp5md.dll![OpenMP fork]+0x5b - kmp_csupport.cpp:341
3_tachyon_omp.exe!thread_trace+0x15b - tachyon_omp.cpp:141
```

2s
2.5s
3s
3.5s
4s
4.5s
5s
5.5s
6s
6.5s
7s
7.5s
Thread
WinMainCRTStartup (TID: 1...
OMP Worker Thread #1 (TID...
OMP Master Thread #0 (TID:...
OMP Worker Thread #2 (TID...
OMP Worker Thread #3 (TID...
func@0x10068430 (TID: 115...
Thread (TID: 15524)
func@0x10068430 (TID: 203...
CPU Utilization
Scale Markers:
Region Instance
OpenMP Barrier-to-Barrier Segment
Thread
Running
Waits
CPU Time
Spin and Overhead ...
CPU Sample
Transitions
CPU Utilization
CPU Time

Optimization Notice

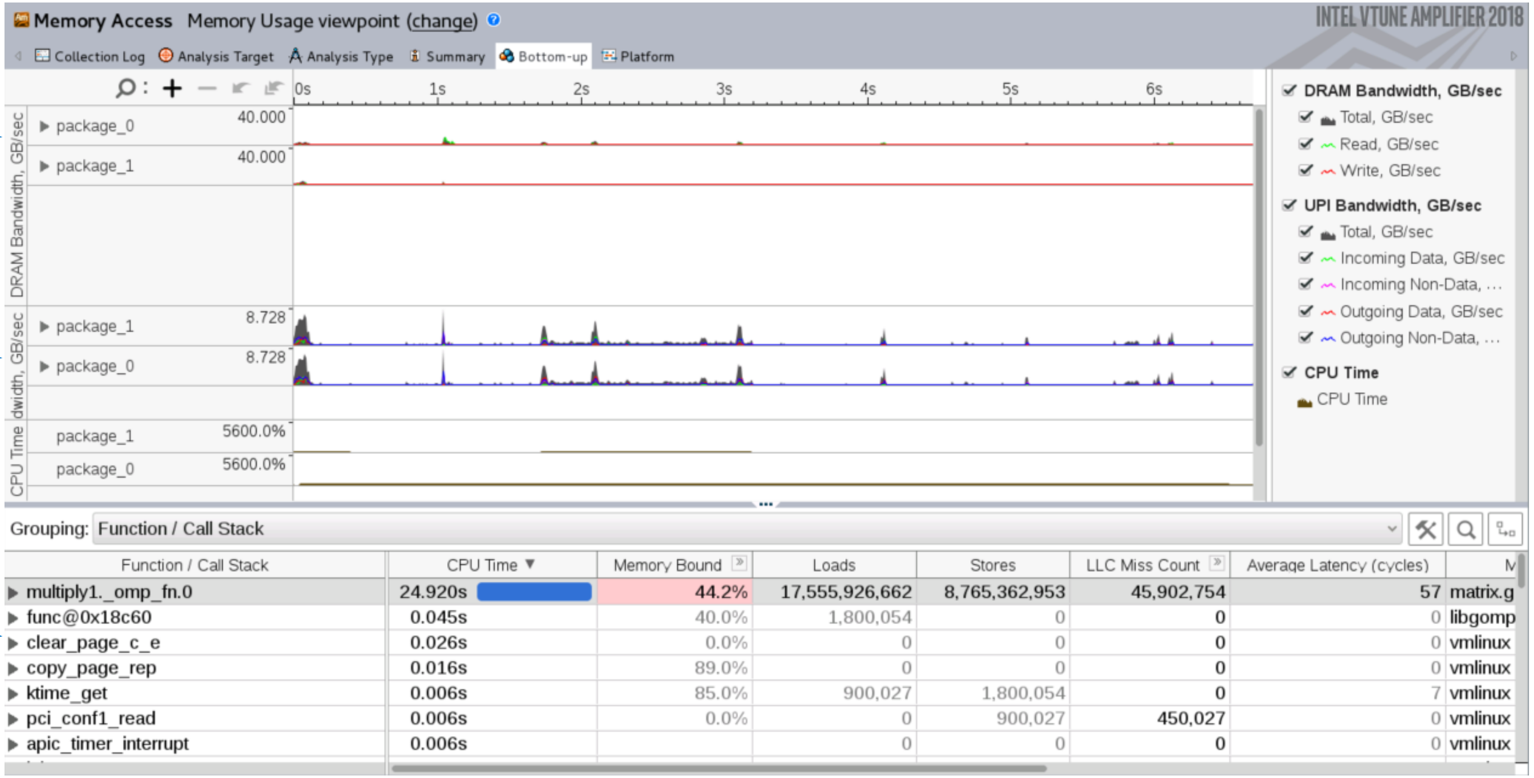
Example: Memory Access Analysis

Bottom-up View

Over-Time DRAM Bandwidth

Over-Time QPI/UPI Bandwidth

Grid Breakdown by Function (configurable)



Function / Call Stack	CPU Time	Memory Bound	Loads	Stores	LLC Miss Count	Average Latency (cycles)	M
multiply1._omp_fn.0	24.920s	44.2%	17,555,926,662	8,765,362,953	45,902,754	57	matrix.g
func@0x18c60	0.045s	40.0%	1,800,054	0	0	0	libgomp
clear_page_c_e	0.026s	0.0%	0	0	0	0	vmlinux
copy_page_rep	0.016s	89.0%	0	0	0	0	vmlinux
ktime_get	0.006s	85.0%	900,027	1,800,054	0	7	vmlinux
pci_conf1_read	0.006s	0.0%	0	900,027	450,027	0	vmlinux
apic_timer_interrupt	0.006s		0	0	0	0	vmlinux

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

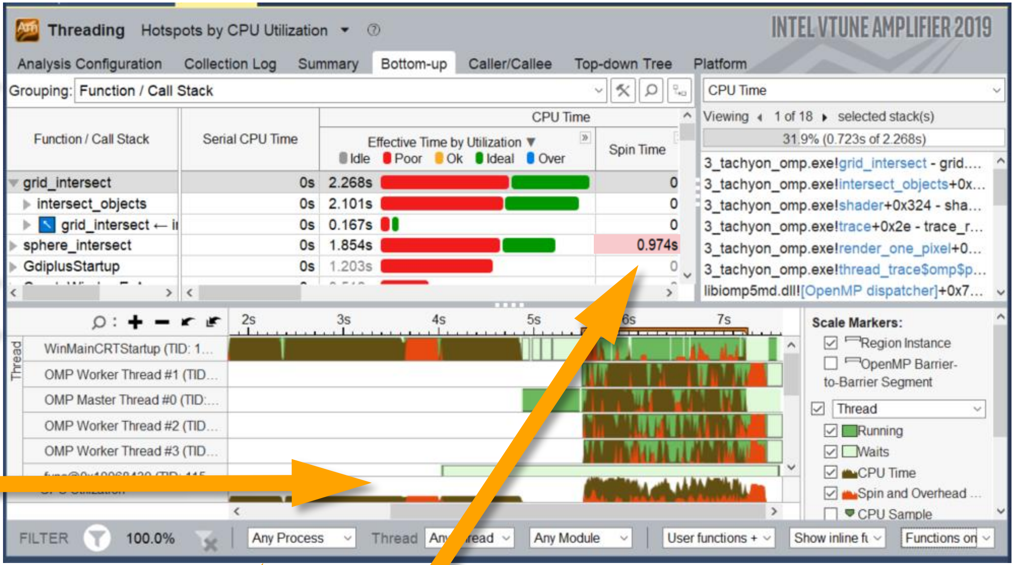
... (Partial list shown)

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



Grouping: Function / Call Stack

CPU Time

Function / Call Stack	Serial CPU Time	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Spin Time
grid_intersect	0s	2.268s	0
intersect_objects	0s	2.101s	0
grid_intersect ← i	0s	0.167s	0
sphere_intersect	0s	1.854s	0.974s
GdiplusStartup	0s	1.203s	0

Viewing 1 of 18 selected stack(s)
31.9% (0.723s of 2.268s)
3_tachyon_omp.exe!grid_intersect - grid....
3_tachyon_omp.exe!intersect_objects+0x...
3_tachyon_omp.exe!shader+0x324 - sha...
3_tachyon_omp.exe!trace+0x2e - trace_r...
3_tachyon_omp.exe!render_one_pixel+0...
3_tachyon_omp.exe!thread_trace$omp$p...
libiomp5md.dll![OpenMP dispatcher]+0x7...

Filter by Process & Other Controls

Tuning Opportunities Shown in Pink. Hover for Tips

Optimization Notice

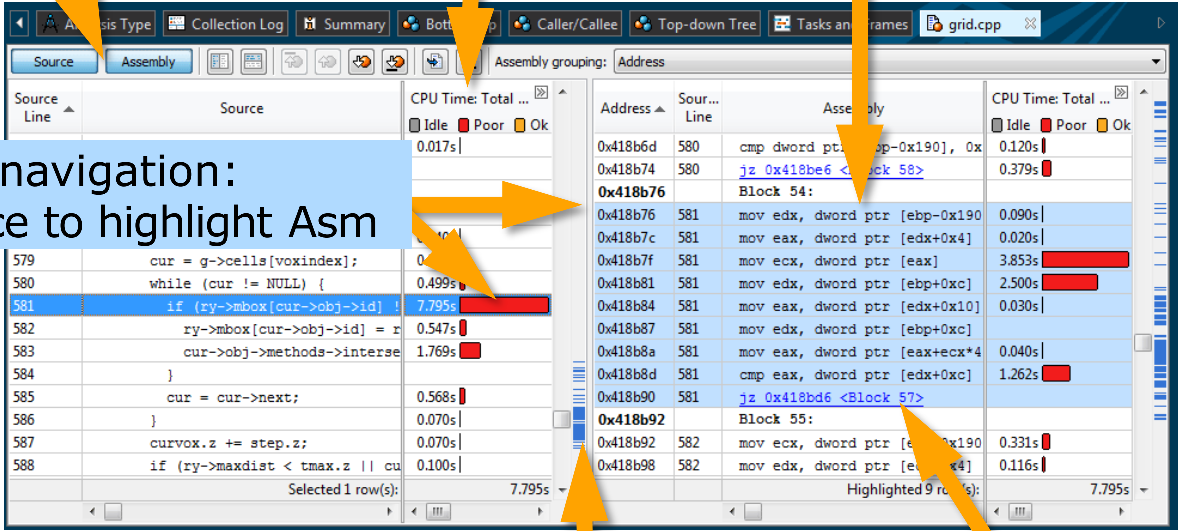
See Profile Data On Source / Asm

Double Click from Grid or Timeline

View Source / Asm or both

CPU Time

Right click for instruction reference manual



Source Line	Source	CPU Time: Total ... (Idle / Poor / Ok)
		0.017s
579	cur = g->cells[voxindex];	
580	while (cur != NULL) {	0.499s
581	if (ry->mbox[cur->obj->id] !	7.795s
582	ry->mbox[cur->obj->id] = r	0.547s
583	cur->obj->methods->interse	1.769s
584	}	
585	cur = cur->next;	0.568s
586	}	0.070s
587	curvox.z += step.z;	0.070s
588	if (ry->maxdist < tmax.z || cu	0.100s
	Selected 1 row(s):	7.795s

Address	Sour... Line	Assembly	CPU Time: Total ... (Idle / Poor / Ok)
0x418b6d	580	cmp dword ptr [ebp-0x190], 0x	0.120s
0x418b74	580	jz 0x418be6 < ock 58>	0.379s
0x418b76		Block 54:	
0x418b76	581	mov edx, dword ptr [ebp-0x190	0.090s
0x418b7c	581	mov eax, dword ptr [edx+0x4]	0.020s
0x418b7f	581	mov ecx, dword ptr [eax]	3.853s
0x418b81	581	mov edx, dword ptr [ebp+0xc]	2.500s
0x418b84	581	mov eax, dword ptr [edx+0x10]	0.030s
0x418b87	581	mov edx, dword ptr [ebp+0xc]	
0x418b8a	581	mov eax, dword ptr [eax+ecx*4	0.040s
0x418b8d	581	cmp eax, dword ptr [edx+0xc]	1.262s
0x418b90	581	jz 0x418bd6 <Block 57>	
0x418b92		Block 55:	
0x418b92	582	mov ecx, dword ptr [e x190	0.331s
0x418b98	582	mov edx, dword ptr [e x4]	0.116s
		Highlighted 9 r s):	7.795s

Quick Asm navigation:
Select source to highlight Asm

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

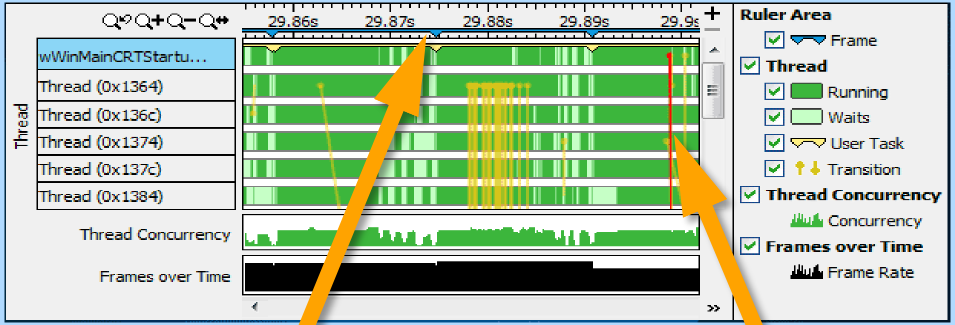
Click jump to scroll Asm

Timeline Visualizes Thread Behavior

Intel® VTune™ Amplifier

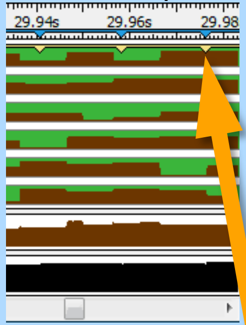
Transitions

Locks & Waits

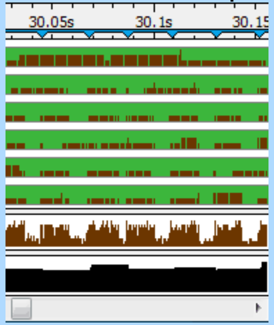


CPU Time

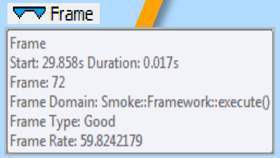
Basic Hotspots



Advanced Hotspots

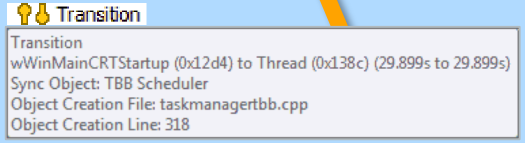


Hovers:



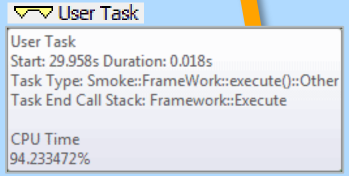
Frame

Frame
Start: 29.858s Duration: 0.017s
Frame: 72
Frame Domain: Smoke::Framework::execute()
Frame Type: Good
Frame Rate: 59.8242179



Transition

Transition
wWinMainCRTStartup (0x12d4) to Thread (0x138c) (29.899s to 29.899s)
Sync Object: TBB Scheduler
Object Creation File: taskmanagertbb.cpp
Object Creation Line: 318



User Task

User Task
Start: 29.958s Duration: 0.018s
Task Type: Smoke::FrameWork::execute()::Other
Task End Call Stack: Framework::Execute

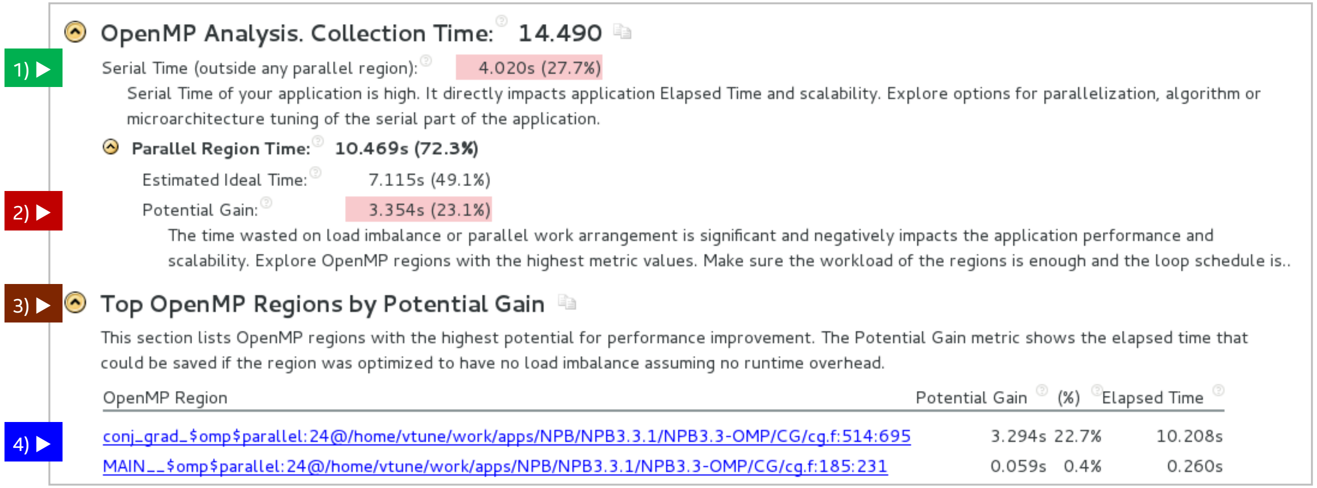
CPU Time
94.233472%

Optional: Use API to mark frames and user tasks Frame User Task

Optional: Add a mark during collection Mark Timeline

Tune OpenMP for Efficiency and Scalability

Fast Answers: Is My OpenMP Scalable? How Much Faster Could It Be?



1) ▶ 2) ▶ 3) ▶ 4) ▶

OpenMP Analysis. Collection Time: 14.490

1) ▶ Serial Time (outside any parallel region): 4.020s (27.7%)

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: 10.469s (72.3%)

Estimated Ideal Time: 7.115s (49.1%)

2) ▶ Potential Gain: 3.354s (23.1%)

The time wasted on load imbalance or parallel work arrangement is significant and negatively impacts the application performance and scalability. Explore OpenMP regions with the highest metric values. Make sure the workload of the regions is enough and the loop schedule is..

3) ▶ 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	Potential Gain	(%)	Elapsed Time
conj_grad_ompparallel:24@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:514:695	3.294s	22.7%	10.208s
MAIN__ompparallel:24@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:185:231	0.059s	0.4%	0.260s

4) ▶

The summary view shown above gives fast answers to four important OpenMP tuning questions:

1) Is the serial time of my application significant enough to prevent scaling?
2) How much performance can be gained by tuning OpenMP?
3) Which OpenMP regions / loops / barriers will benefit most from tuning?
4) What are the inefficiencies with each region? (click the link to see details)

Command Line Interface

Automate analysis

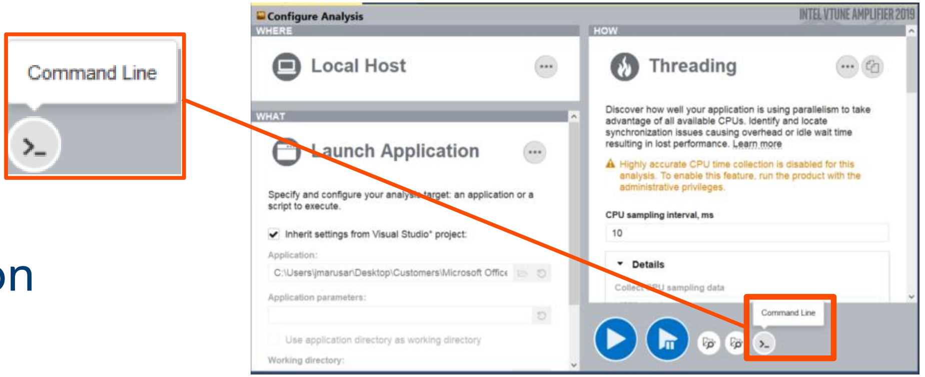
amplxe-cl is the command line:

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



Discover how well your application is using parallelism to take advantage of all available CPUs. Identify and locate synchronization issues causing overhead or idle wait time resulting in lost performance. Learn more

⚠ Highly accurate CPU time collection is disabled for this analysis. To enable this feature, run the product with the administrative privileges.

CPU sampling interval, ms: 10

Collect CPU sampling data

Specify and configure your analysis target: an application or a script to execute.

✓ Inherit settings from Visual Studio* project:

Application: C:\Users\jmarusar\Desktop\Customers\Microsoft Office

Application parameters:

Use application directory as working directory

Working directory:

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

Compare Results Quickly - Sort By Difference

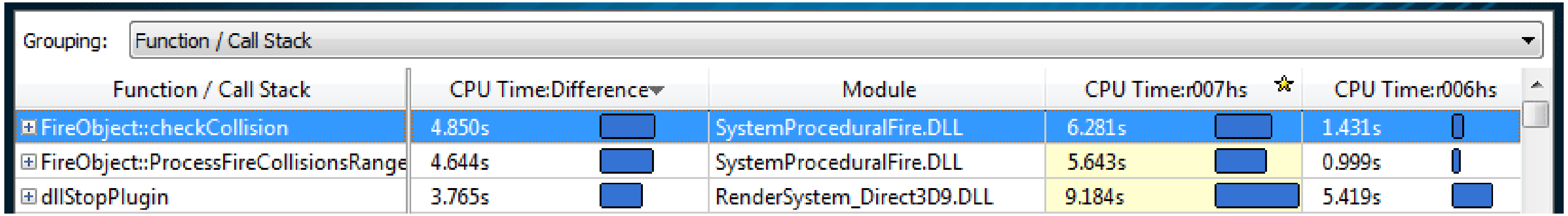
Intel® VTune™ Amplifier

Quickly identify cause of regressions.

- Run a command line analysis daily
- Identify the function responsible so you know who to alert

Compare 2 optimizations – What improved?

Compare 2 systems – What didn't speed up as much?



Grouping: Function / Call Stack

Function / Call Stack	CPU Time:Difference	Module	CPU Time:r007hs	CPU Time:r006hs
⊞ FireObject::checkCollision	4.850s	SystemProceduralFire.DLL	6.281s	1.431s
⊞ FireObject::ProcessFireCollisionsRange	4.644s	SystemProceduralFire.DLL	5.643s	0.999s
⊞ dllStopPlugin	3.765s	RenderSystem_Direct3D9.DLL	9.184s	5.419s

Optimize Memory Access

Memory Access Analysis - Intel® VTune™ Amplifier 2017

Tune data structures for performance

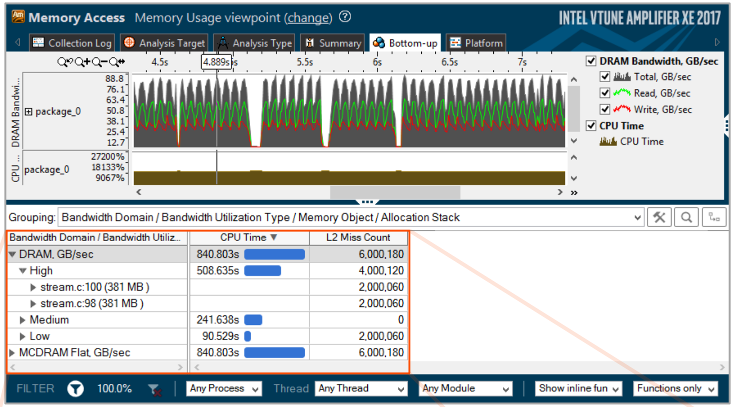
- Attribute cache misses to data structures (not just the code causing the miss)
- Support for custom memory allocators

Optimize NUMA latency & scalability

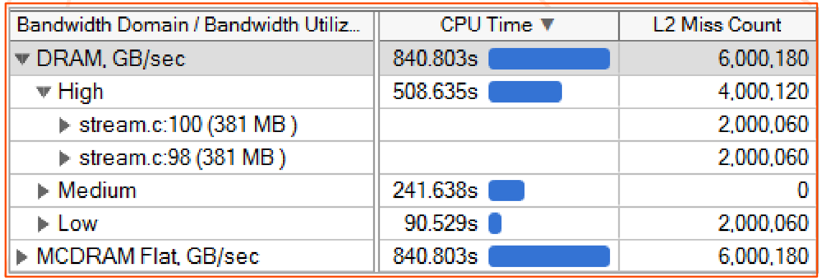
- True & false sharing optimization
- Auto detect max system bandwidth
- Easier tuning of inter-socket bandwidth

Easier install, Latest processors

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

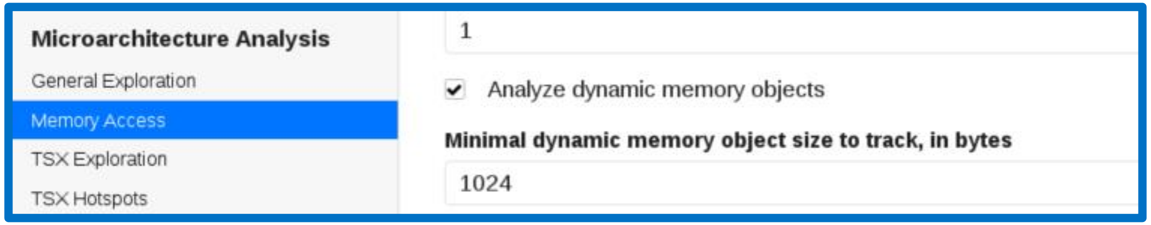


Grouping: Bandwidth Domain / Bandwidth Utilization Type / Memory Object / Allocation Stack



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 Object Identification



Microarchitecture Analysis

- General Exploration
- Memory Access
- TSX Exploration
- TSX Hotspots

1

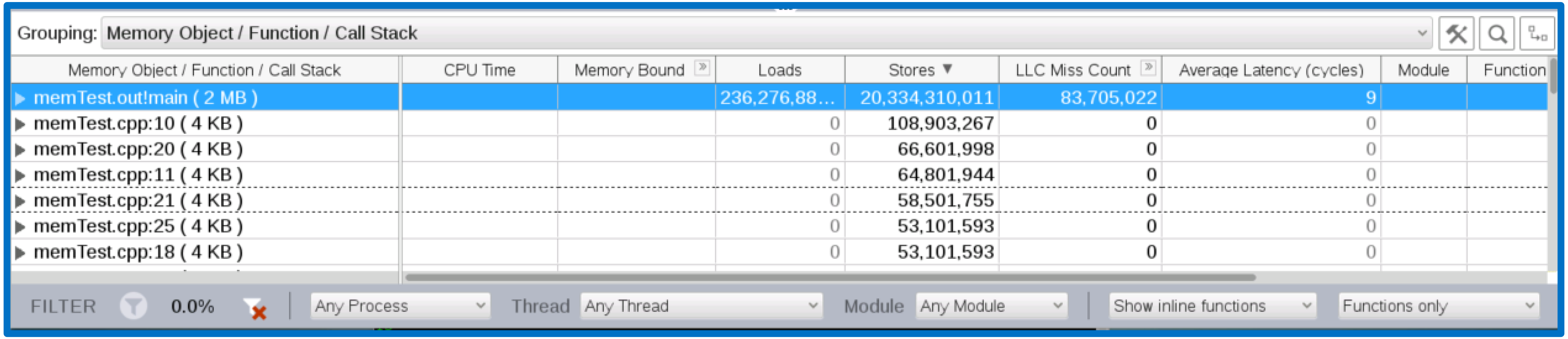
☑ Analyze dynamic memory objects

Minimal dynamic memory object size to track, in bytes

1024

View allocated objects

Sort by LLC Miss Count

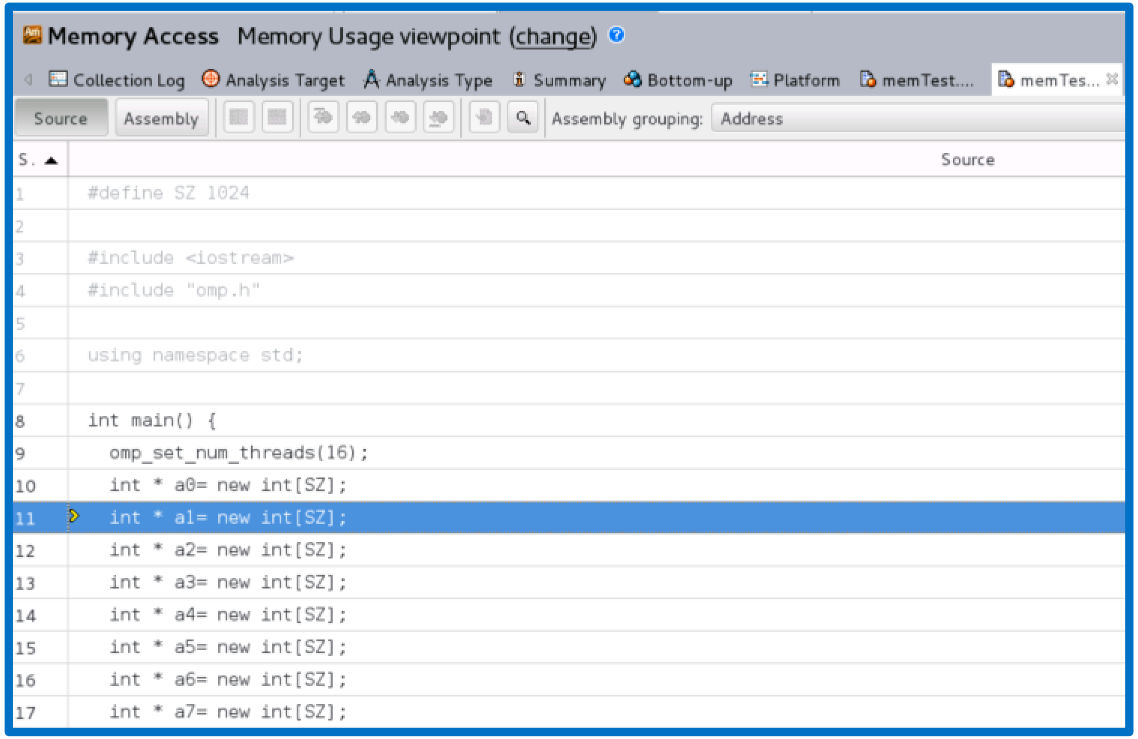


Grouping: Memory Object / Function / Call Stack

Memory Object / Function / Call Stack	CPU Time	Memory Bound	Loads	Stores ▼	LLC Miss Count	Average Latency (cycles)	Module	Function
▶ memTest.out!main (2 MB)			236,276,88…	20,334,310,011	83,705,022	9		
▶ memTest.cpp:10 (4 KB)			0	108,903,267	0	0		
▶ memTest.cpp:20 (4 KB)			0	66,601,998	0	0		
▶ memTest.cpp:11 (4 KB)			0	64,801,944	0	0		
▶ memTest.cpp:21 (4 KB)			0	58,501,755	0	0		
▶ memTest.cpp:25 (4 KB)			0	53,101,593	0	0		
▶ memTest.cpp:18 (4 KB)			0	53,101,593	0	0		

FILTER 0.0% Any Process Thread Any Thread Module Any Module Show inline functions Functions only

Memory Object Identification



```
#define SZ 1024

#include <iostream>
#include "omp.h"

using namespace std;

int main() {
  omp_set_num_threads(16);
  int * a0= new int[SZ];
  int * a1= new int[SZ];
  int * a2= new int[SZ];
  int * a3= new int[SZ];
  int * a4= new int[SZ];
  int * a5= new int[SZ];
  int * a6= new int[SZ];
  int * a7= new int[SZ];
```

Assembly view also available

Double-click to see allocation site in source view

Storage Device Analysis (HDD, SATA or NVMe SSD)

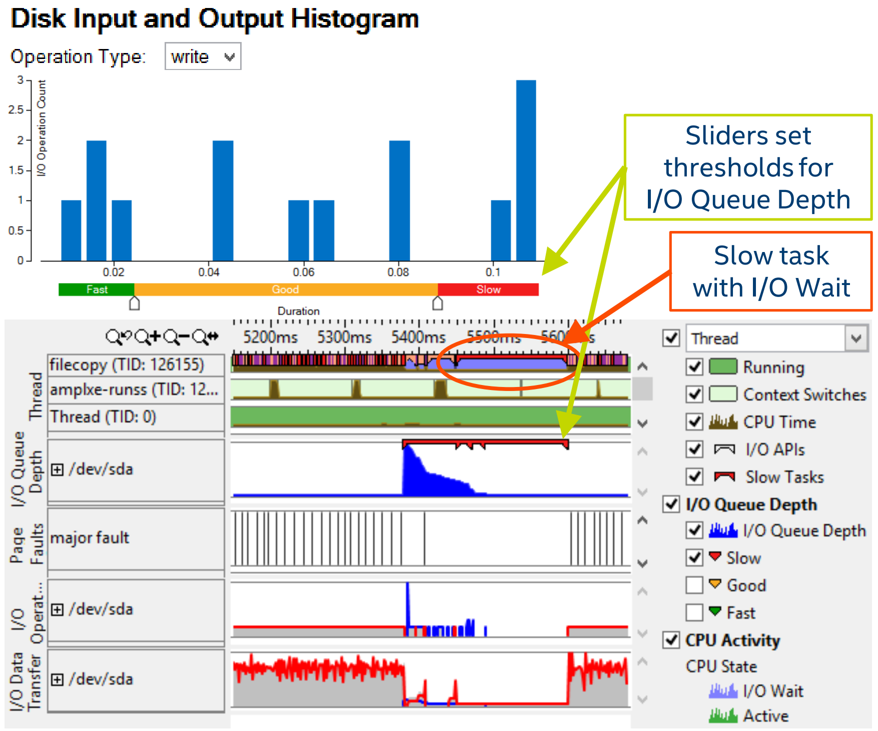
Intel® VTune™ Amplifier

Are You I/O Bound or CPU Bound?

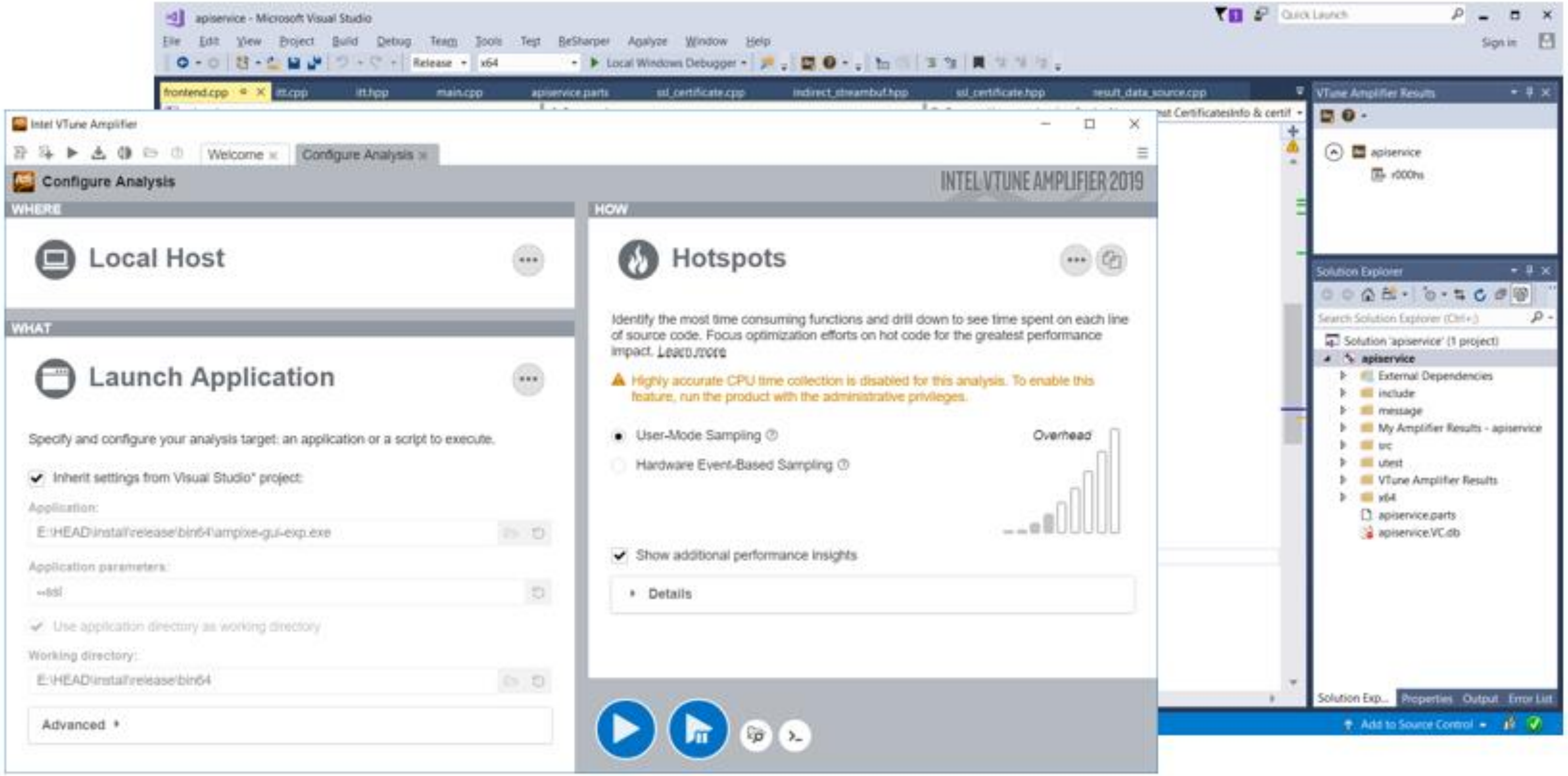
- Explore imbalance between I/O operations (async & sync) and compute
- Storage accesses mapped to the source code
- See when CPU is waiting for I/O
- Measure bus bandwidth to storage

Latency analysis

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



A Quick Question for the Audience


apiservice - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test ReSharper Analyze Window Help
Release x64 Local Windows Debugger
frontend.cpp itt.cpp itt.hpp main.cpp apiservice.parts ssl_certificate.cpp indirect_streambuf.hpp ssl_certificate.hpp result_data_source.cpp
VTune Amplifier Results
apiservice
r000hs
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'apiservice' (1 project)
apiservice
External Dependencies
include
message
My Amplifier Results - apiservice
src
utest
VTune Amplifier Results
x64
apiservice.parts
apiservice.VC.db
Solution Exp... Properties Output Error List
Add to Source Control
Intel VTune Amplifier
Welcome
Configure Analysis
INTEL VTUNE AMPLIFIER 2019

Configure Analysis

WHERE

Local Host

WHAT

Launch Application

Specify and configure your analysis target: an application or a script to execute.

✓ Inherit settings from Visual Studio* project:

Application:
E:\HEAD\install\release\bin64\amplxe-gui-exp.exe

Application parameters:
--ssl

✓ Use application directory as working directory

Working directory:
E:\HEAD\install\release\bin64

Advanced

HOW

Hotspots

Identify the most time consuming functions and drill down to see time spent on each line of source code. Focus optimization efforts on hot code for the greatest performance impact. Learn more

⚠ Highly accurate CPU time collection is disabled for this analysis. To enable this feature, run the product with the administrative privileges.

● User-Mode Sampling
○ Hardware Event-Based Sampling

Overhead

✓ Show additional performance insights

Details

TANGENT ON A COUPLE OTHER TOOLS

INTEL® ADVISOR 2019

VECTORIZATION OPTIMIZATION AND THREAD PROTOTYPING

- Vectorization Advisor
- Threading Advisor
- Flow Graph Analyzer

Get Faster Code Faster! Intel® Advisor

Thread Prototyping

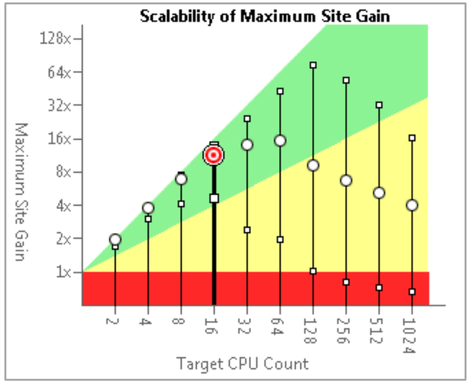
Have you:

- Threaded an app, but seen little benefit?
- Hit a "scalability barrier"?
- Delayed release due to sync. errors?

Data Driven Threading Design:

- Quickly prototype multiple options
- Project scaling on larger systems
- Find synchronization errors before implementing threading
- Design without disrupting development

Add Parallelism with Less Effort, Less Risk and More Impact



"**Intel® Advisor** has allowed us to quickly prototype ideas for parallelism, saving developer time and effort"

Simon Hammond
Senior Technical Staff
Sandia National Laboratories

http://intel.ly/advisor-xe

Get Faster Code Faster! Intel® Advisor

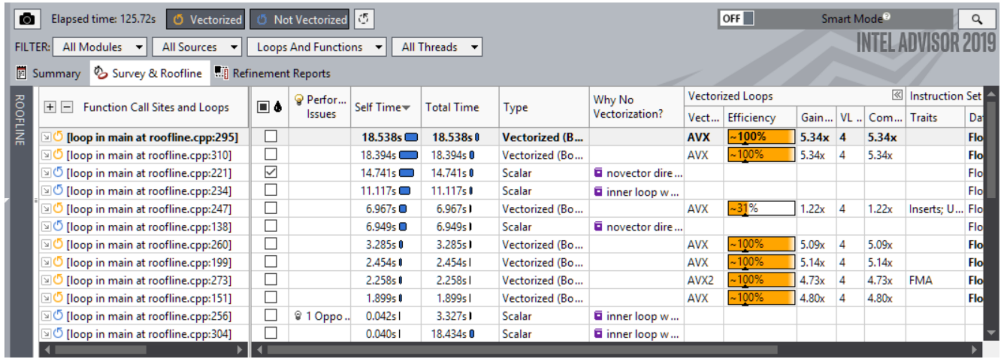
Vectorization Optimization

Have you:

- Recompiled for AVX2 with little gain
- Wondered where to vectorize?
- Recoded intrinsics for new arch.?
- Struggled with compiler reports?

Data Driven Vectorization:

- What vectorization will pay off most?
- What's blocking vectorization? Why?
- Are my loops vector friendly?
- Will reorganizing data increase performance?
- Is it safe to just use pragma simd?



Function Call Sites and Loops		Perfor... Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops					Instruction Set	
							Vect...	Efficiency	Gain...	VL ..	Com...	Traits	Da
[loop in main at roofline.cpp:295]	☐		**18.538s**	**18.538s**	**Vectorized (B...**		**AVX**	**~100%**	**5.34x**	**4**	**5.34x**		**Flo**
[loop in main at roofline.cpp:310]	☐		18.394s	18.394s	Vectorized (Bo...		AVX	~100%	5.34x	4	5.34x		Flo
[loop in main at roofline.cpp:221]	☑		14.741s	14.741s	Scalar	novector dire...							Flo
[loop in main at roofline.cpp:234]	☐		11.117s	11.117s	Scalar	inner loop w...							Flo
[loop in main at roofline.cpp:247]	☐		6.967s	6.967s	Vectorized (Bo...		AVX	~31%	1.22x	4	1.22x	Inserts; U...	Flo
[loop in main at roofline.cpp:138]	☐		6.949s	6.949s	Scalar	novector dire...							Flo
[loop in main at roofline.cpp:260]	☐		3.285s	3.285s	Vectorized (Bo...		AVX	~100%	5.09x	4	5.09x		Flo
[loop in main at roofline.cpp:199]	☐		2.454s	2.454s	Vectorized (Bo...		AVX	~100%	5.14x	4	5.14x		Flo
[loop in main at roofline.cpp:273]	☐		2.258s	2.258s	Vectorized (Bo...		AVX2	~100%	4.73x	4	4.73x	FMA	Flo
[loop in main at roofline.cpp:151]	☐		1.899s	1.899s	Vectorized (Bo...		AVX	~100%	4.80x	4	4.80x		Flo
[loop in main at roofline.cpp:256]	☐	1 Oppo..	0.042s	3.327s	Scalar	inner loop w...							
[loop in main at roofline.cpp:304]	☐		0.040s	18.434s	Scalar	inner loop w...							

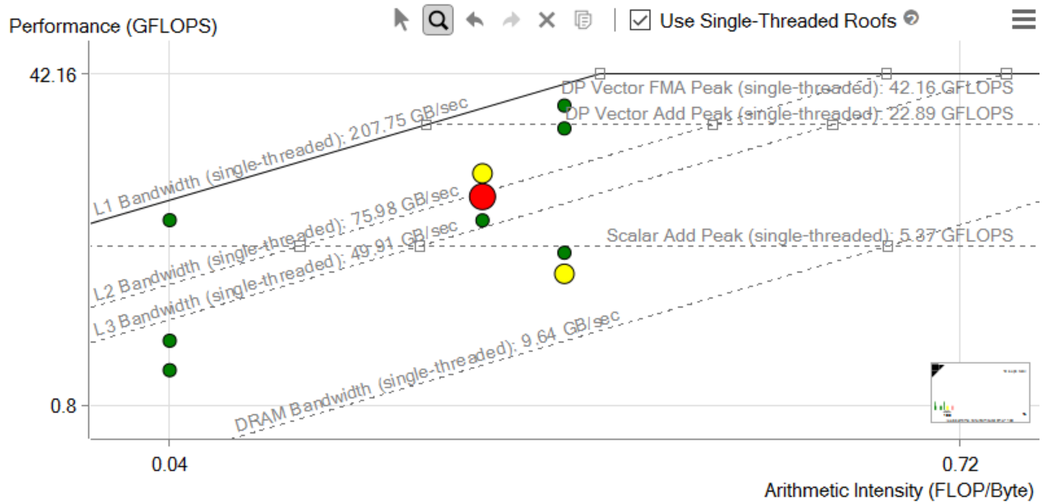
"Intel® Advisor's Vectorization Advisor permitted me to focus my work where it really mattered. When you have only a limited amount of time to spend on optimization, it is invaluable."

Gilles Civario
Senior Software Architect
Irish Centre for High-End Computing

What is a Roofline Chart?

A Roofline Chart plots application performance against hardware limitations.

- Where are the bottlenecks?
- How much performance is being left on the table?
- Which bottlenecks can be addressed, and which *should* be addressed?
- What's the most likely cause?
- What are the next steps?



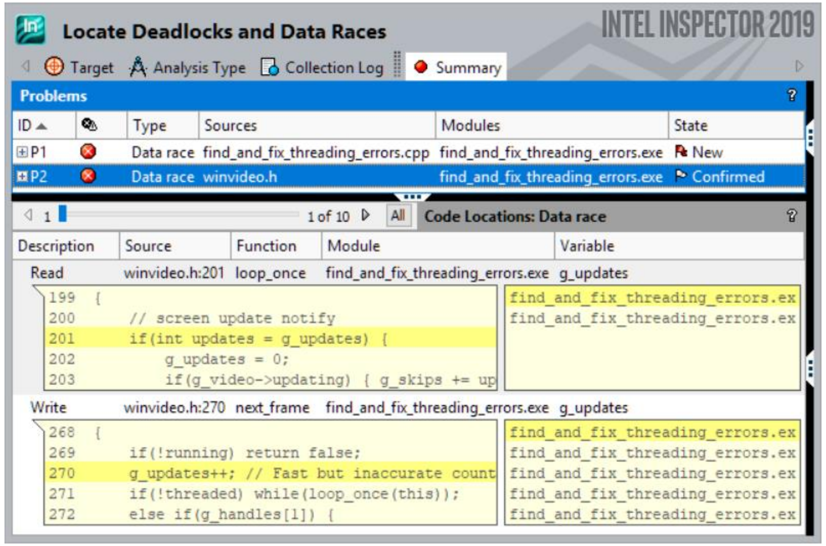
Roofline first proposed by University of California at Berkeley:
Roofline: An Insightful Visual Performance Model for Multicore Architectures, 2009
Cache-aware variant proposed by University of Lisbon:
Cache-Aware Roofline Model: Upgrading the Loft, 2013

INTEL® INSPECTOR 2019

MEMORY & THREAD DEBUGGER

Debug Memory & Threading with Intel® Inspector

Find & Debug Memory Leaks, Corruption, Data Races, Deadlocks



Problems

ID		Type	Sources	Modules	State
P1	●	Data race	find_and_fix_threading_errors.cpp	find_and_fix_threading_errors.exe	New
P2	●	Data race	winvideo.h	find_and_fix_threading_errors.exe	Confirmed

1 1 of 10 All Code Locations: Data race

Description	Source	Function	Module	Variable
Read	winvideo.h:201	loop_once	find_and_fix_threading_errors.exe	g_updates

```
199   {
200       // screen update notify
201       if(int updates = g_updates) {
202           g_updates = 0;
203           if(g_video->updating) { g_skips += up
```

```
find_and_fix_threading_errors.ex
find_and_fix_threading_errors.ex
```

Description	Source	Function	Module	Variable
Write	winvideo.h:270	next_frame	find_and_fix_threading_errors.exe	g_updates

```
268   {
269       if(!running) return false;
270       g_updates++; // Fast but inaccurate count
271       if(!threaded) while(loop_once(this));
272       else if(g_handles[1]) {
```

```
find_and_fix_threading_errors.ex
find_and_fix_threading_errors.ex
find_and_fix_threading_errors.ex
find_and_fix_threading_errors.ex
find_and_fix_threading_errors.ex
```

Correctness Tools Increase ROI by 12%-21%[1]

- Errors found earlier are less expensive to fix
- Races & deadlocks not easily reproduced
- Memory errors are hard to find without a tool

Debugger Integration Speeds Diagnosis

- Breakpoint set just before the problem
- Examine variables and threads with the debugger

What's New in 2019 Release

Find Persistent Memory Errors

- Missing / redundant cache flushes
- Missing store fences
- Out-of-order persistent memory stores
- PMDK transaction redo logging errors

Learn More: bit.ly/intel-inspector

[1]**Cost Factors – Square Project Analysis -** *CERT: U.S. Computer Emergency Readiness Team, and Carnegie Mellon CyLab NIST: National Institute of Standards & Technology: Square Project Results*

Debug Memory & Threading Errors

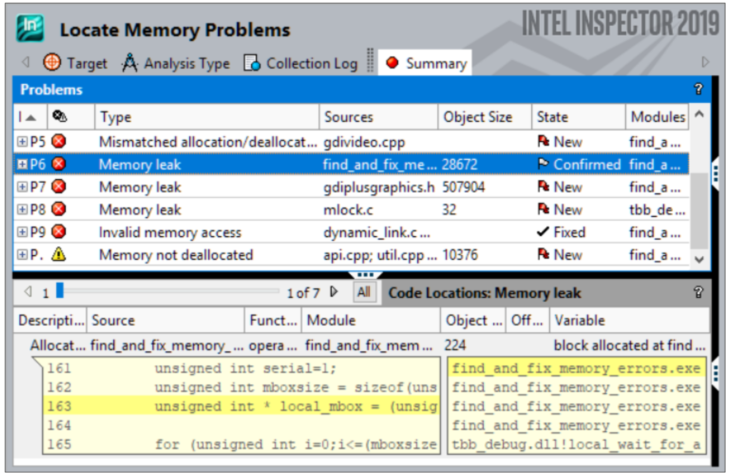
Intel® Inspector

Find and eliminate errors

- Memory leaks, invalid access…
- Races & deadlocks
- C, C++ and Fortran (or a mix)

Simple, Reliable, Accurate

- No special recompiles
  Use any build, any compiler[1]
- Analyzes dynamically generated or linked code
- Inspects 3rd party libraries without source
- Productive user interface + debugger integration
- Command line for automated regression analysis



I		Type	Sources	Object Size	State	Modules
P5	⊗	Mismatched allocation/deallocat…	gdivideo.cpp		New	find_a…
P6	⊗	Memory leak	find_and_fix_me…	28672	Confirmed	find_a…
P7	⊗	Memory leak	gdiplusgraphics.h	507904	New	find_a…
P8	⊗	Memory leak	mlock.c	32	New	tbb_de…
P9	⊗	Invalid memory access	dynamic_link.c…		✓ Fixed	find_a…
P.	⚠	Memory not deallocated	api.cpp; util.cpp…	10376	New	find_a…

Descripti…	Source	Funct…	Module	Object …	Off…	Variable
Allocat…	find_and_fix_memory_…	opera…	find_and_fix_mem…	224		block allocated at find…

```
161         unsigned int serial=1;
162         unsigned int mboxsize = sizeof(uns
163         unsigned int * local_mbox = (unsig
164
165         for (unsigned int i=0;i<=(mboxsize
```

```
find_and_fix_memory_errors.exe
find_and_fix_memory_errors.exe
find_and_fix_memory_errors.exe
find_and_fix_memory_errors.exe
tbb_debug.dll!local_wait_for_a
```

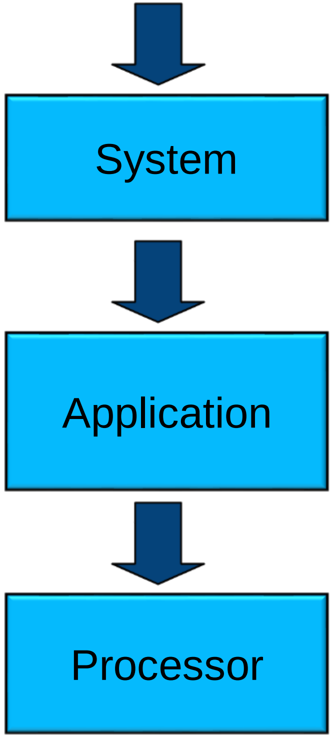
Clicking an error instantly displays source code snippets and the call stack

Fits your existing process

[1]That follows common OS standards.

DIVING DEEPER INTO ANALYSIS

Introduction to Performance Tuning

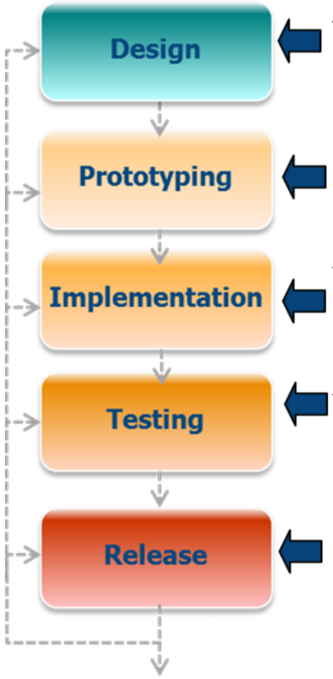


H/W tuning:
- BIOS (TB, HT)
- Memory
- Network I/O
- Disk I/O

OS tuning:
- Page size
- Swap file
- RAM Disk
- Power settings
- Network protocols

Better application design:
- Parallelization
- Fast algorithms / data bases
- Programming language and RT libs
- Performance libraries
- Driver tuning

Tuning for Microarchitecture:
- Compiler settings/Vectorization
- Memory/Cache usage
- CPU pitfalls



Think performance wise (app/sys level)

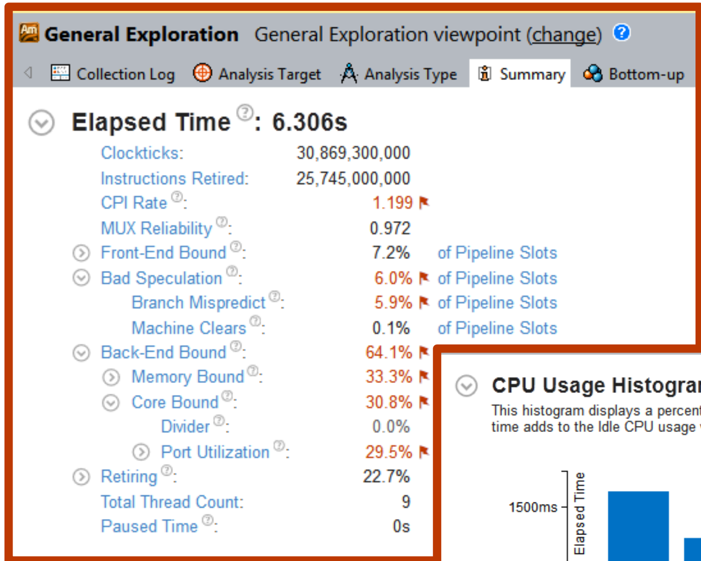
Choose performance. effective solutions

Apply performance optimization and check results

Add performance regressions to test stage

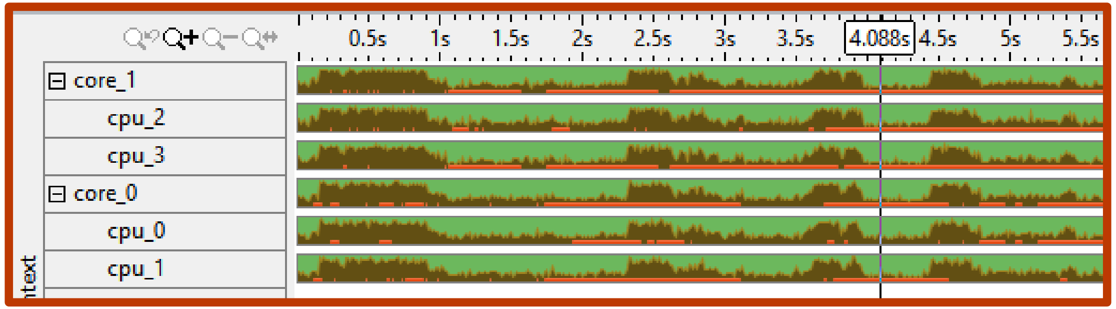
Collect and analyze performance related issues from users

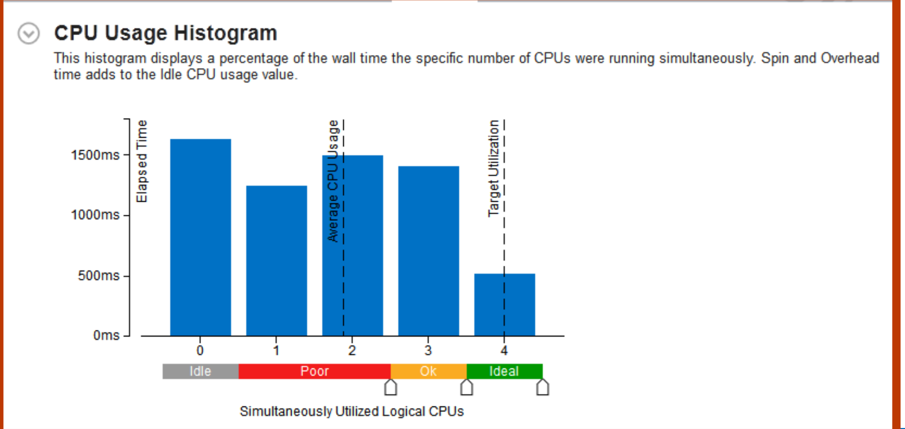
System-Level Profiling – High-level Overviews



Elapsed Time: 6.306s

Clockticks:	30,869,300,000	
Instructions Retired:	25,745,000,000	
CPI Rate:	1.199	
MUX Reliability:	0.972	
Front-End Bound:	7.2%	of Pipeline Slots
Bad Speculation:	6.0%	of Pipeline Slots
Branch Mispredict:	5.9%	of Pipeline Slots
Machine Clears:	0.1%	of Pipeline Slots
Back-End Bound:	64.1%	
Memory Bound:	33.3%	
Core Bound:	30.8%	
Divider:	0.0%	
Port Utilization:	29.5%	
Retiring:	22.7%	
Total Thread Count:	9	
Paused Time:	0s	

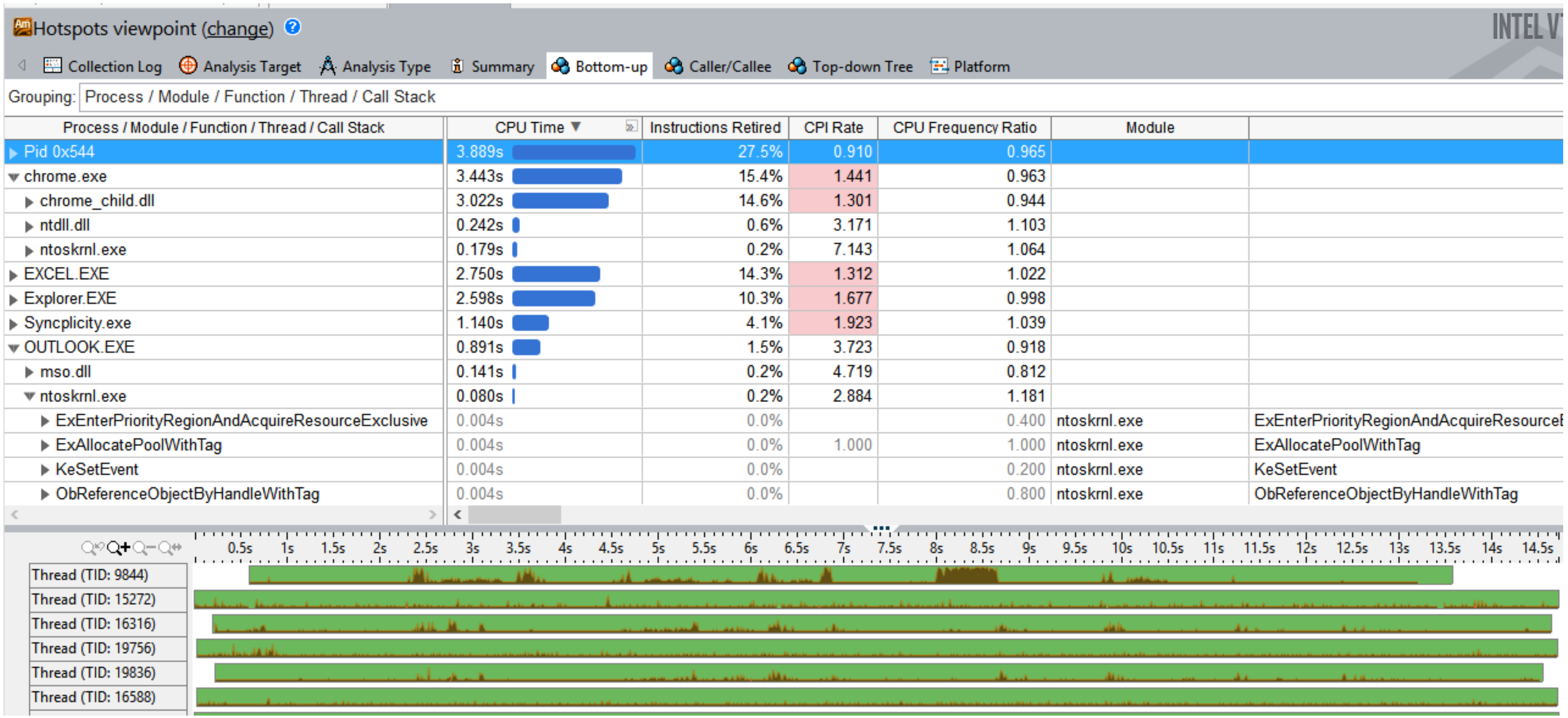




CPU Usage Histogram

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.

System-Level Profiling – Process/Module Breakdowns



Grouping: Process / Module / Function / Thread / Call Stack

Process / Module / Function / Thread / Call Stack	CPU Time	Instructions Retired	CPI Rate	CPU Frequency Ratio	Module	
Pid 0x544	3.889s	27.5%	0.910	0.965		
chrome.exe	3.443s	15.4%	1.441	0.963		
chrome_child.dll	3.022s	14.6%	1.301	0.944		
ntdll.dll	0.242s	0.6%	3.171	1.103		
ntoskrnl.exe	0.179s	0.2%	7.143	1.064		
EXCEL.EXE	2.750s	14.3%	1.312	1.022		
Explorer.EXE	2.598s	10.3%	1.677	0.998		
Syncplicity.exe	1.140s	4.1%	1.923	1.039		
OUTLOOK.EXE	0.891s	1.5%	3.723	0.918		
mso.dll	0.141s	0.2%	4.719	0.812		
ntoskrnl.exe	0.080s	0.2%	2.884	1.181		
ExEnterPriorityRegionAndAcquireResourceExclusive	0.004s	0.0%		0.400	ntoskrnl.exe	ExEnterPriorityRegionAndAcquireResourceE
ExAllocatePoolWithTag	0.004s	0.0%	1.000	1.000	ntoskrnl.exe	ExAllocatePoolWithTag
KeSetEvent	0.004s	0.0%		0.200	ntoskrnl.exe	KeSetEvent
ObReferenceObjectByHandleWithTag	0.004s	0.0%		0.800	ntoskrnl.exe	ObReferenceObjectByHandleWithTag

0.5s 1s 1.5s 2s 2.5s 3s 3.5s 4s 4.5s 5s 5.5s 6s 6.5s 7s 7.5s 8s 8.5s 9s 9.5s 10s 10.5s 11s 11.5s 12s 12.5s 13s 13.5s 14s 14.5s

Thread (TID: 9844)
Thread (TID: 15272)
Thread (TID: 16316)
Thread (TID: 19756)
Thread (TID: 19836)
Thread (TID: 16588)

System-Level Profiling – Disk I/O Analysis

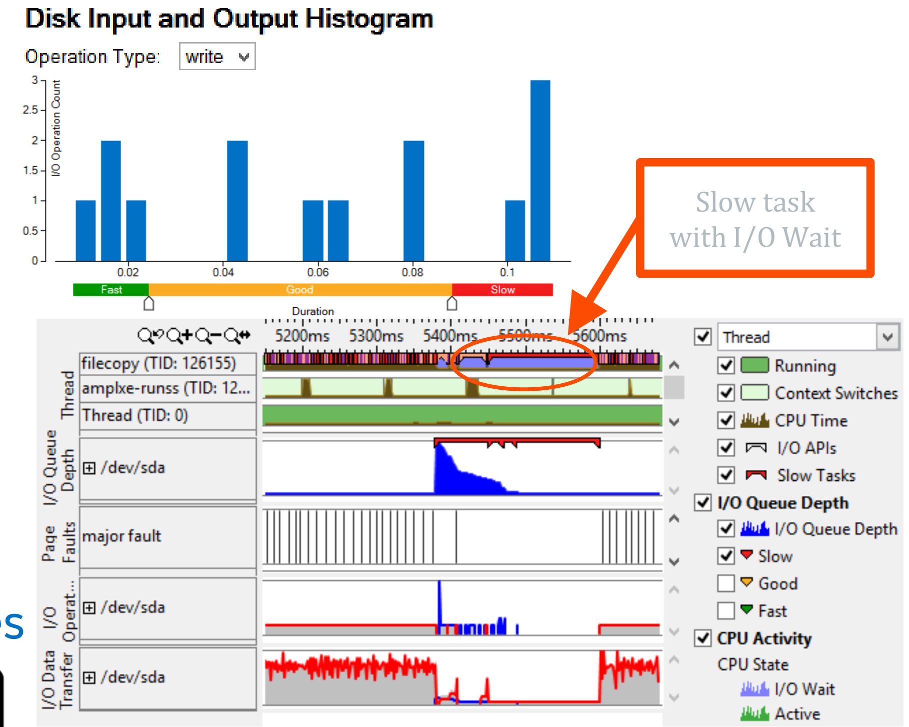
Are You I/O Bound or CPU Bound?

- Explore imbalance between I/O opera (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
```

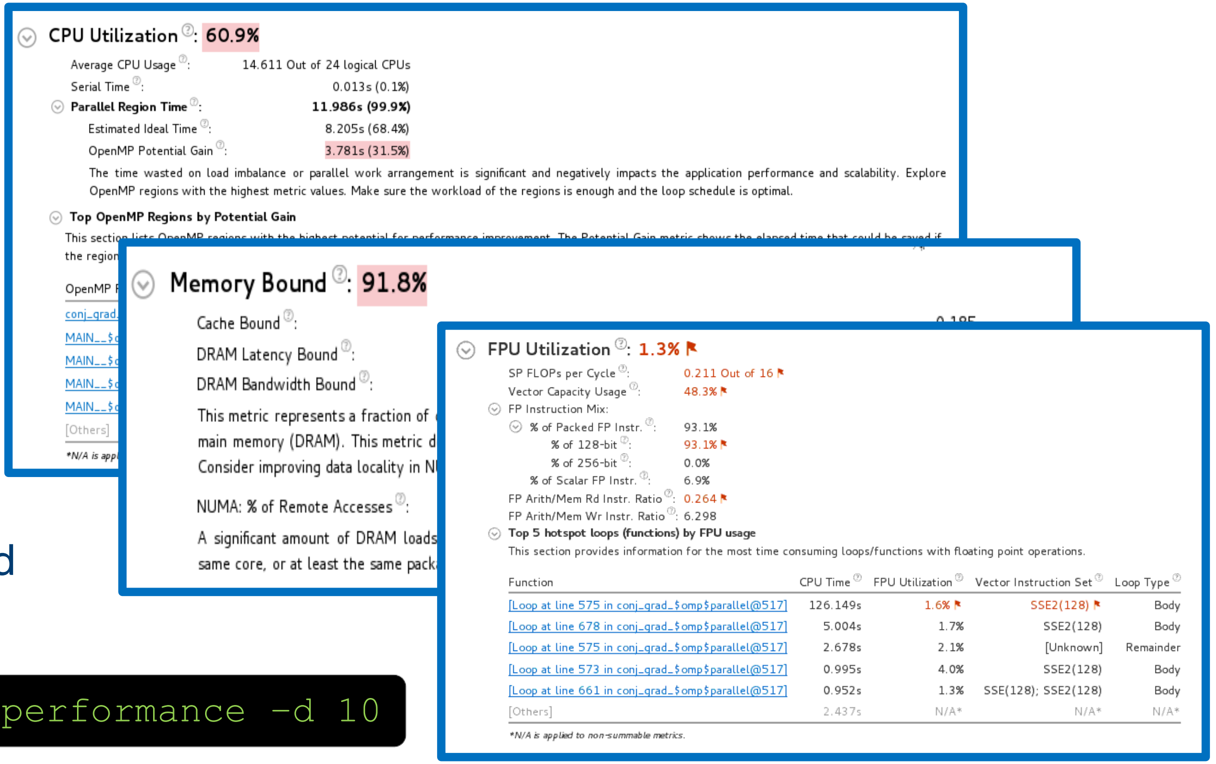


System-Level Profiling – HPC Characterizaton

Three Metric Classes

- CPU Utilization
  - Logical core % usage
  - Includes parallelism and OpenMP information
- Memory Bound
  - Break down each level of the memory hierarchy
- FPU Utilization
  - Floating point GFLOPS and density

```
> amplxe-cl -collect hpc-performance -d 10
```



CPU Utilization: 60.9%

Average CPU Usage:	14.611 Out of 24 logical CPUs
Serial Time:	0.013s (0.1%)
Parallel Region Time:	**11.986s (99.9%)**
Estimated Ideal Time:	8.205s (68.4%)
OpenMP Potential Gain:	3.781s (31.5%)

The time wasted on load imbalance or parallel work arrangement is significant and negatively impacts the application performance and scalability. Explore OpenMP regions with the highest metric values. Make sure the workload of the regions is enough and the loop schedule is optimal.

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

OpenMP R

conj_grad

MAIN__$c

MAIN__$c

MAIN__$c

MAIN__$c

[Others]

*N/A is app

Memory Bound: 91.8%

Cache Bound:

DRAM Latency Bound:

DRAM Bandwidth Bound:

This metric represents a fraction of c
main memory (DRAM). This metric d
Consider improving data locality in NU

NUMA: % of Remote Accesses:

A significant amount of DRAM loads
same core, or at least the same pack

0.185

FPU Utilization: 1.3%

SP FLOPs per Cycle:	0.211 Out of 16
Vector Capacity Usage:	48.3%
FP Instruction Mix:	
% of Packed FP Instr.:	93.1%
% of 128-bit:	93.1%
% of 256-bit:	0.0%
% of Scalar FP Instr.:	6.9%
FP Arith/Mem Rd Instr. Ratio:	0.264
FP Arith/Mem Wr Instr. Ratio:	6.298

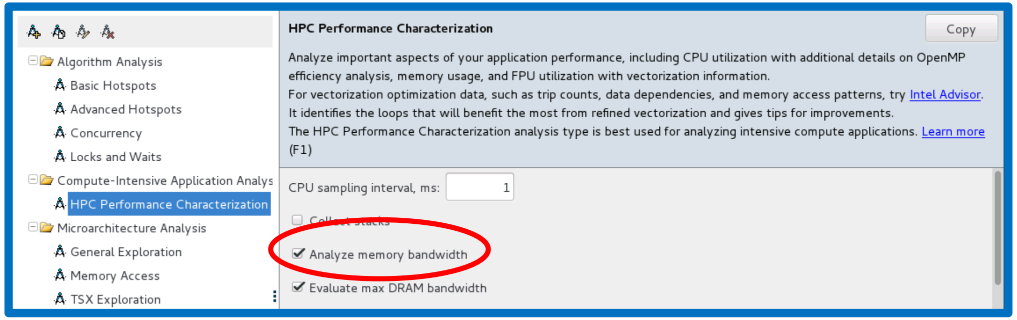
Top 5 hotspot loops (functions) by FPU usage

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

Function	CPU Time	FPU Utilization	Vector Instruction Set	Loop Type
[Loop at line 575 in conj_grad_ompparallel@517]	126.149s	1.6%	SSE2(128)	Body
[Loop at line 678 in conj_grad_ompparallel@517]	5.004s	1.7%	SSE2(128)	Body
[Loop at line 575 in conj_grad_ompparallel@517]	2.678s	2.1%	[Unknown]	Remainder
[Loop at line 573 in conj_grad_ompparallel@517]	0.995s	4.0%	SSE2(128)	Body
[Loop at line 661 in conj_grad_ompparallel@517]	0.952s	1.3%	SSE(128); SSE2(128)	Body
[Others]	2.437s	N/A*	N/A*	N/A*

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

System-Level Profiling – Memory Bandwidth



HPC Performance Characterization

Analyze important aspects of your application performance, including CPU utilization with additional details on OpenMP efficiency analysis, memory usage, and FPU utilization with vectorization information.
For vectorization optimization data, such as trip counts, data dependencies, and memory access patterns, try Intel Advisor.
It identifies the loops that will benefit the most from refined vectorization and gives tips for improvements.
The HPC Performance Characterization analysis type is best used for analyzing intensive compute applications. Learn more (F1)

CPU sampling interval, ms: 1

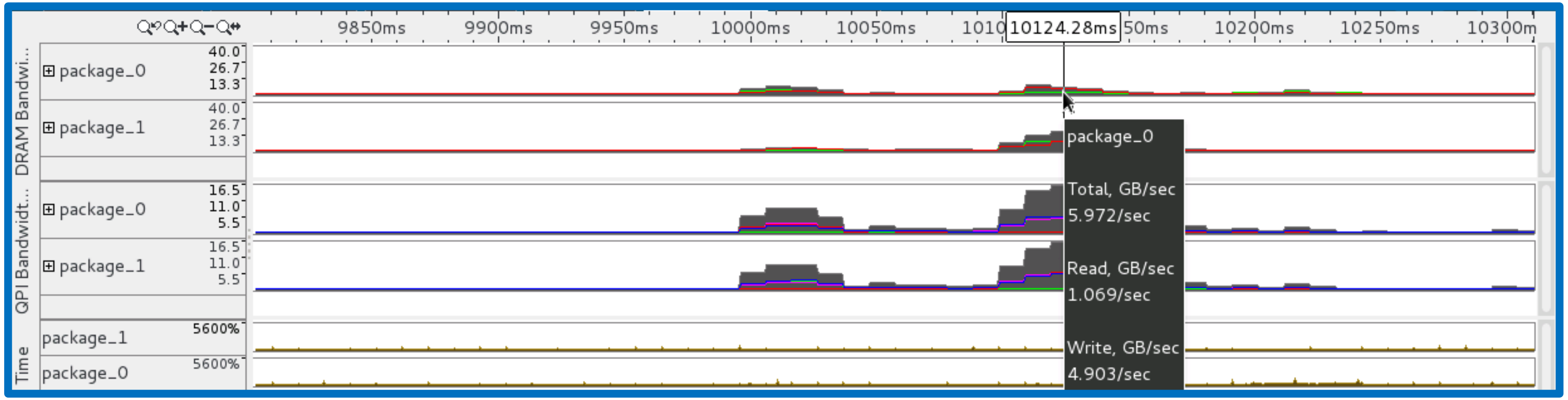
[ ] Collect stacks

[✓] Analyze memory bandwidth

[✓] Evaluate max DRAM bandwidth

Find areas of high and low bandwidth usage. Compare to max system bandwidth based on Stream benchmarks.

```
-knob collect-memory-bandwidth=true
```



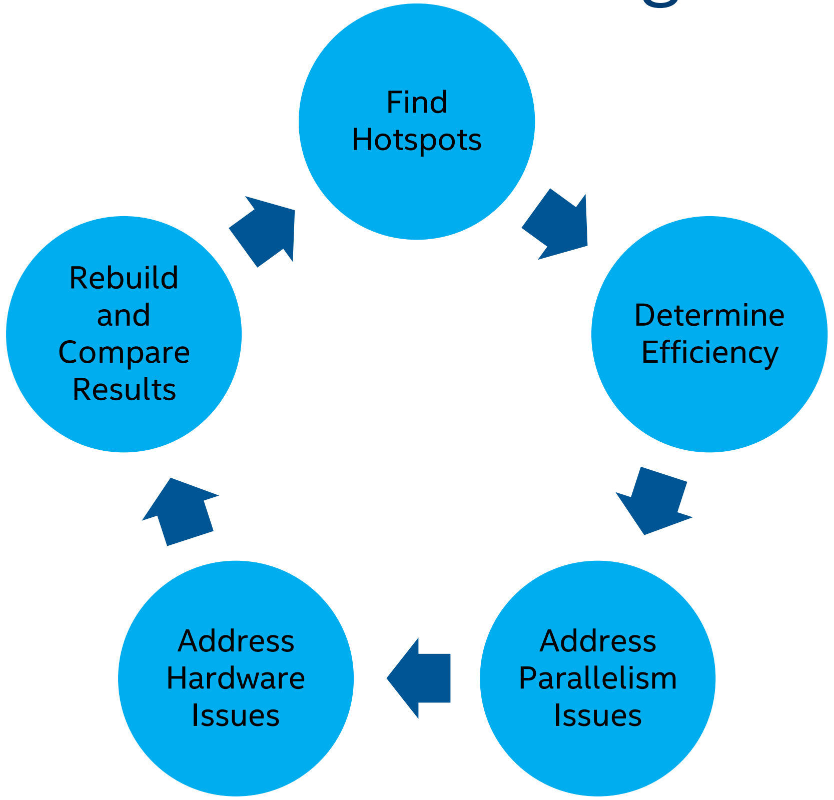
package_0

Total, GB/sec 5.972/sec

Read, GB/sec 1.069/sec

Write, GB/sec 4.903/sec

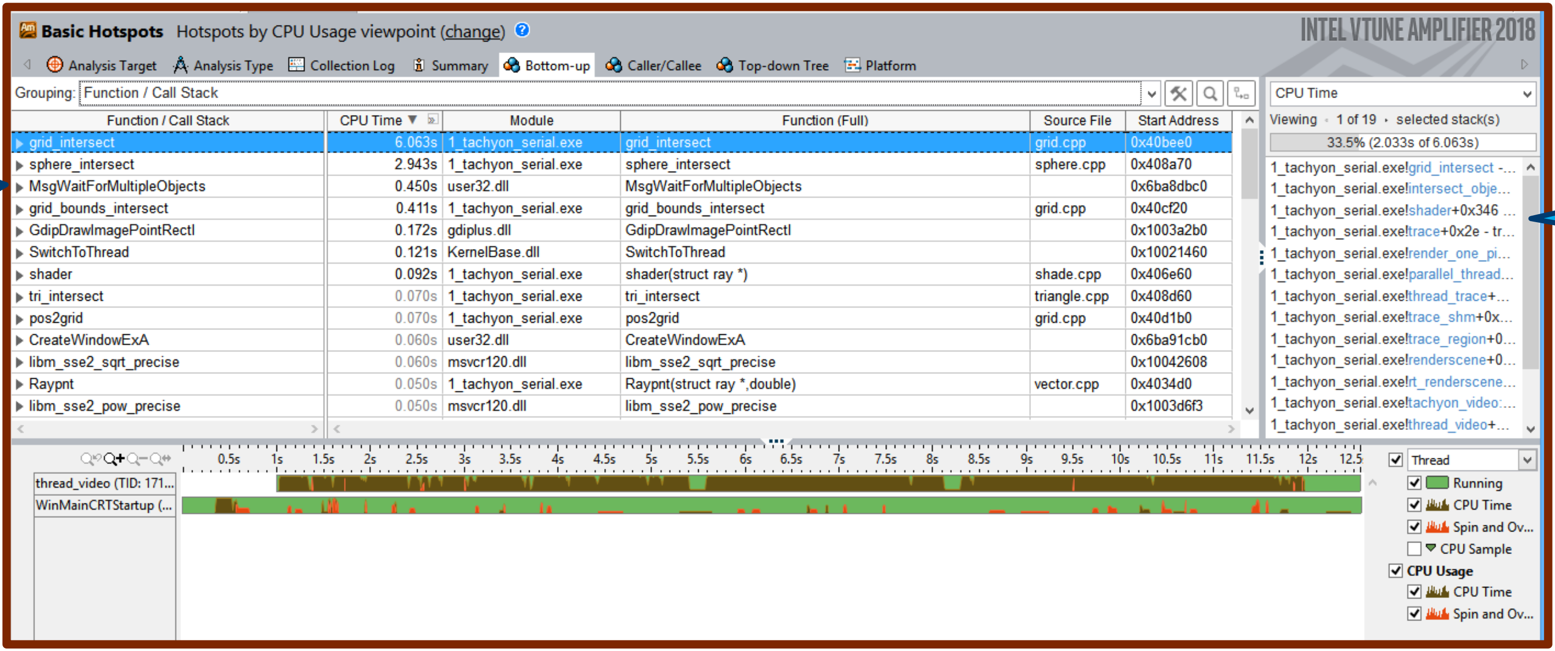
Application Performance Tuning Process



Find Hotspots

Functions

Call Stacks


Basic Hotspots Hotspots by CPU Usage viewpoint (change)
INTEL VTUNE AMPLIFIER 2018
Analysis Target
Analysis Type
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Platform
Grouping: Function / Call Stack
CPU Time
Viewing 1 of 19 selected stack(s)
33.5% (2.033s of 6.063s)
0.5s 1s 1.5s 2s 2.5s 3s 3.5s 4s 4.5s 5s 5.5s 6s 6.5s 7s 7.5s 8s 8.5s 9s 9.5s 10s 10.5s 11s 11.5s 12s 12.5s
thread_video (TID: 171...
WinMainCRTStartup (...
Thread
Running
CPU Time
Spin and Ov...
CPU Sample
CPU Usage
CPU Time
Spin and Ov...

Function / Call Stack	CPU Time	Module	Function (Full)	Source File	Start Address
grid_intersect	6.063s	1_tachyon_serial.exe	grid_intersect	grid.cpp	0x40bee0
sphere_intersect	2.943s	1_tachyon_serial.exe	sphere_intersect	sphere.cpp	0x408a70
MsgWaitForMultipleObjects	0.450s	user32.dll	MsgWaitForMultipleObjects		0x6ba8dbc0
grid_bounds_intersect	0.411s	1_tachyon_serial.exe	grid_bounds_intersect	grid.cpp	0x40cf20
GdipDrawImagePointRectI	0.172s	gdiplus.dll	GdipDrawImagePointRectI		0x1003a2b0
SwitchToThread	0.121s	KernelBase.dll	SwitchToThread		0x10021460
shader	0.092s	1_tachyon_serial.exe	shader(struct ray *)	shade.cpp	0x406e60
tri_intersect	0.070s	1_tachyon_serial.exe	tri_intersect	triangle.cpp	0x408d60
pos2grid	0.070s	1_tachyon_serial.exe	pos2grid	grid.cpp	0x40d1b0
CreateWindowExA	0.060s	user32.dll	CreateWindowExA		0x6ba91cb0
libm_sse2_sqrt_precise	0.060s	msvcr120.dll	libm_sse2_sqrt_precise		0x10042608
Raypnt	0.050s	1_tachyon_serial.exe	Raypnt(struct ray *,double)	vector.cpp	0x4034d0
libm_sse2_pow_precise	0.050s	msvcr120.dll	libm_sse2_pow_precise		0x1003d6f3

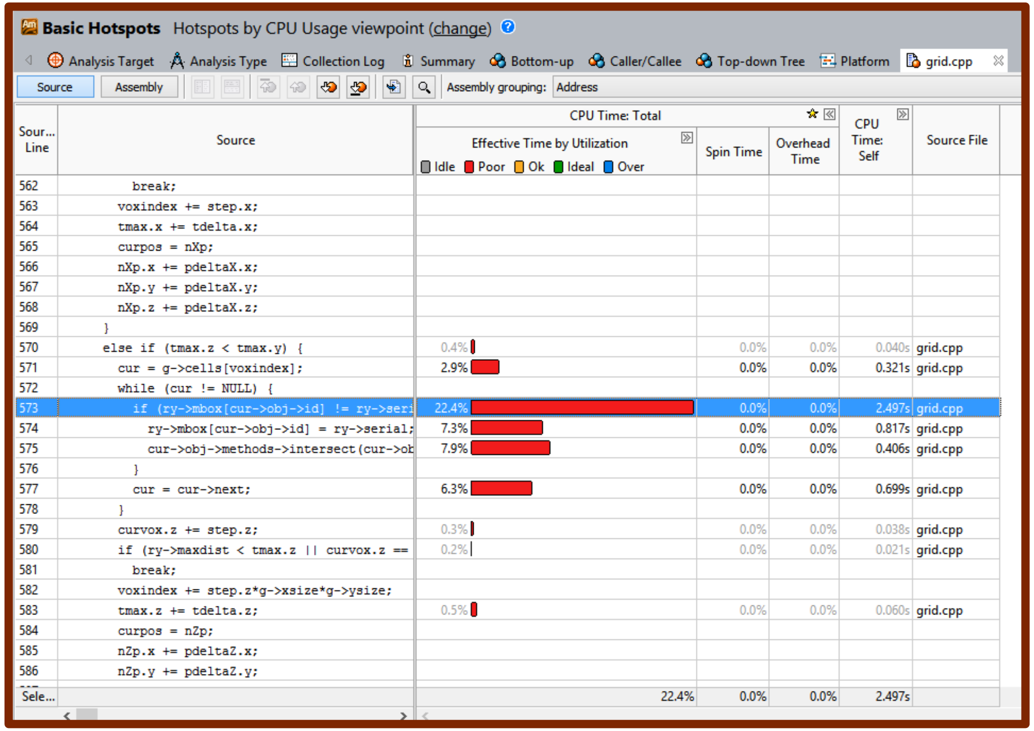
1_tachyon_serial.exe!grid_intersect -...
1_tachyon_serial.exe!intersect_obje...
1_tachyon_serial.exe!shader+0x346 ...
1_tachyon_serial.exe!trace+0x2e - tr...
1_tachyon_serial.exe!render_one_pi...
1_tachyon_serial.exe!parallel_thread...
1_tachyon_serial.exe!thread_trace+...
1_tachyon_serial.exe!trace_shm+0x...
1_tachyon_serial.exe!trace_region+0...
1_tachyon_serial.exe!renderscene+0...
1_tachyon_serial.exe!rt_renderscene...
1_tachyon_serial.exe!tachyon_video:...
1_tachyon_serial.exe!thread_video+...

```
> amplxe-cl -collect hotspots -- ./myapp.out
```

Find Hotspots

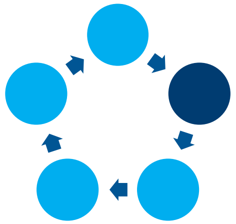
- Drill to source or assembly
- Hottest areas easy to ID
- Is this the expected behavior
- Pay special attention to loops and memory accesses

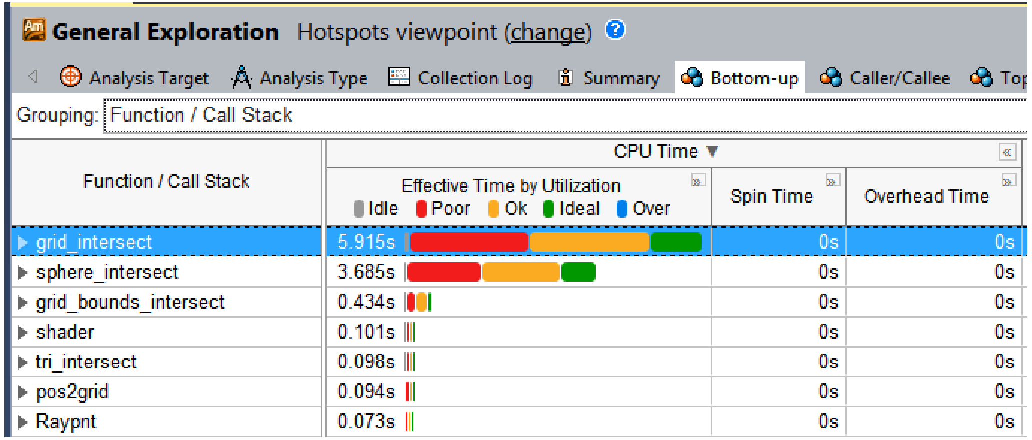
- Learn how your code behaves
- What did the compiler generate
- What are the expensive statements



Sour... Line	Source	CPU Time: Total Effective Time by Utilization ▨ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over	Spin Time	Overhead Time	CPU Time: Self	Source File
562	`break;`					
563	`voxindex += step.x;`					
564	`tmax.x += tdelta.x;`					
565	`curpos = nXp;`					
566	`nXp.x += pdeltaX.x;`					
567	`nXp.y += pdeltaX.y;`					
568	`nXp.z += pdeltaX.z;`					
569	`}`					
570	`else if (tmax.z < tmax.y) {`	0.4%	0.0%	0.0%	0.040s	grid.cpp
571	`cur = g->cells[voxindex];`	2.9%	0.0%	0.0%	0.321s	grid.cpp
572	`while (cur != NULL) {`					
573	`if (ry->mbox[cur->obj->id] != ry->seri`	22.4%	0.0%	0.0%	2.497s	grid.cpp
574	`ry->mbox[cur->obj->id] = ry->serial;`	7.3%	0.0%	0.0%	0.817s	grid.cpp
575	`cur->obj->methods->intersect(cur->ob`	7.9%	0.0%	0.0%	0.406s	grid.cpp
576	`}`					
577	`cur = cur->next;`	6.3%	0.0%	0.0%	0.699s	grid.cpp
578	`}`					
579	`curvox.z += step.z;`	0.3%	0.0%	0.0%	0.038s	grid.cpp
580	`if (ry->maxdist < tmax.z || curvox.z ==`	0.2%	0.0%	0.0%	0.021s	grid.cpp
581	`break;`					
582	`voxindex += step.z*g->xsize*g->ysize;`					
583	`tmax.z += tdelta.z;`	0.5%	0.0%	0.0%	0.060s	grid.cpp
584	`curpos = nZp;`					
585	`nZp.x += pdeltaZ.x;`					
586	`nZp.y += pdeltaZ.y;`					
Sele...		22.4%	0.0%	0.0%	2.497s	

Determine Efficiency



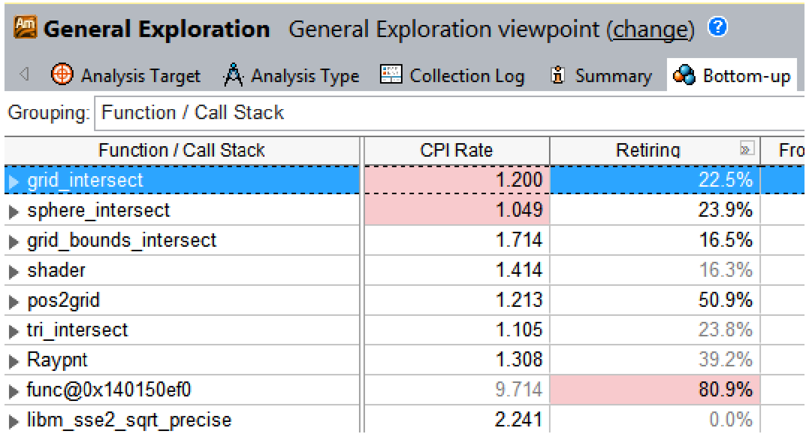


General Exploration Hotspots viewpoint (change)

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

Grouping: Function / Call Stack

Function / Call Stack	CPU Time: Effective Time by Utilization (▮ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over)	CPU Time: Spin Time	CPU Time: Overhead Time
grid_intersect	5.915s 🟥🟧🟩	0s	0s
sphere_intersect	3.685s 🟥🟧🟩	0s	0s
grid_bounds_intersect	0.434s 🟥🟧🟩	0s	0s
shader	0.101s	0s	0s
tri_intersect	0.098s	0s	0s
pos2grid	0.094s	0s	0s
Raypnt	0.073s	0s	0s



General Exploration General Exploration viewpoint (change)

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

Grouping: Function / Call Stack

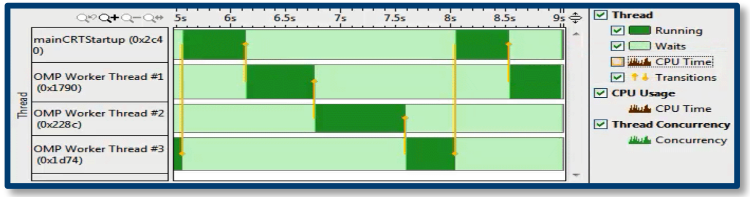
Function / Call Stack	CPI Rate	Retiring	Fro
grid_intersect	1.200	22.5%	
sphere_intersect	1.049	23.9%	
grid_bounds_intersect	1.714	16.5%	
shader	1.414	16.3%	
pos2grid	1.213	50.9%	
tri_intersect	1.105	23.8%	
Raypnt	1.308	39.2%	
func@0x140150ef0	9.714	80.9%	
libm_sse2_sqrt_precise	2.241	0.0%	

Look for Parallelism, Cycles-per-Instruction (CPI), and Retiring %

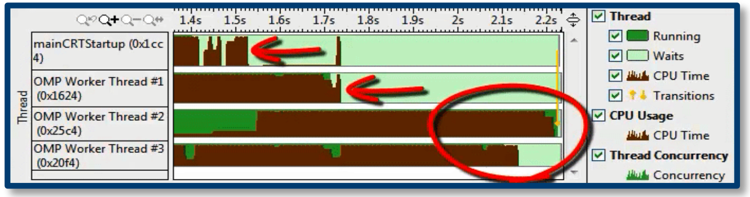
Address Parallelism Issues

- Use Concurrency Analysis to ensure you're using all your threads as often as possible.
- Common concurrency problems can often be diagnosed in the timeline.
- Switch to the Locks And Waits viewpoint or run a Locks and Waits analysis to investigate contention.

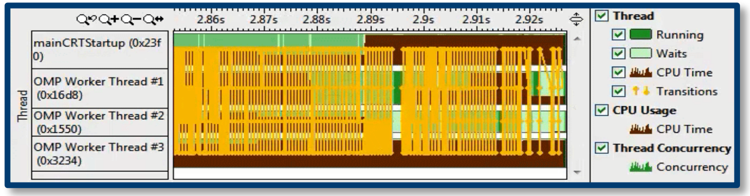
Coarse-Grain Locks



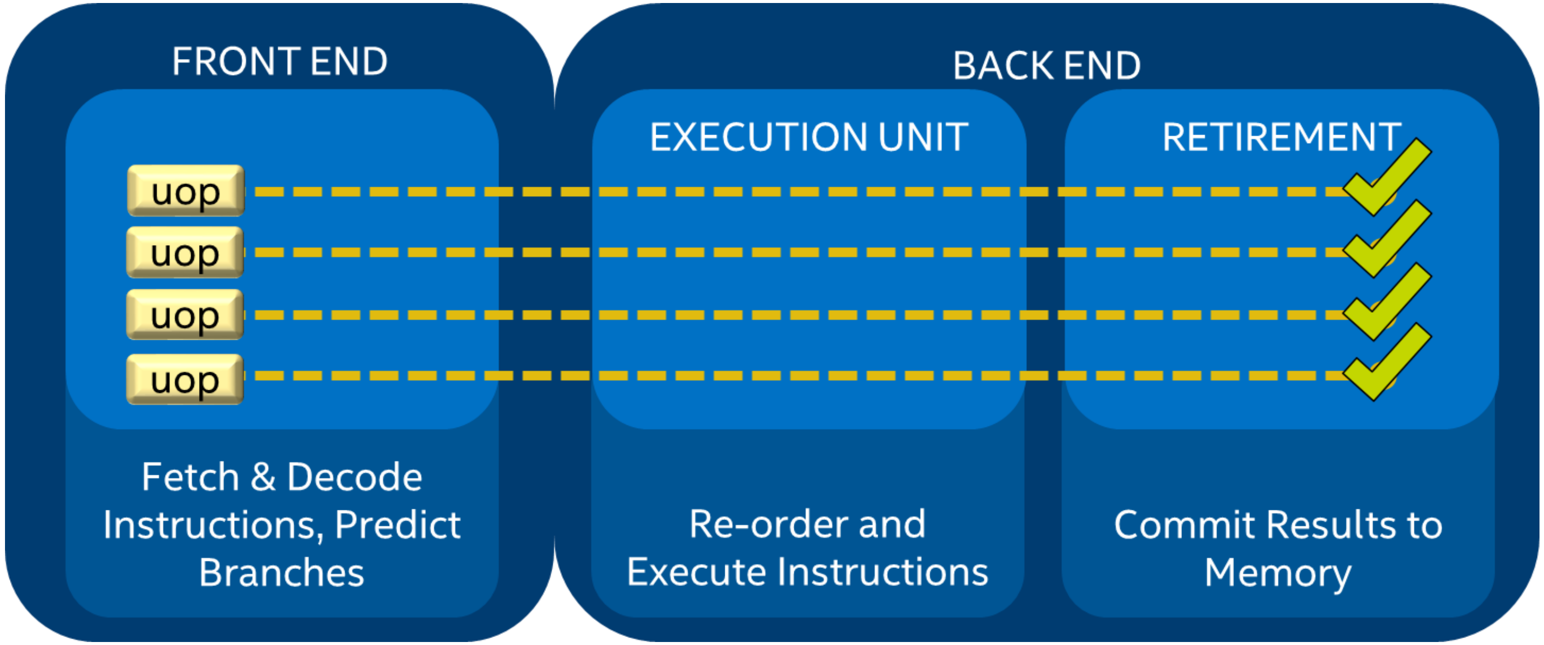
Thread Imbalance



High Lock Contention



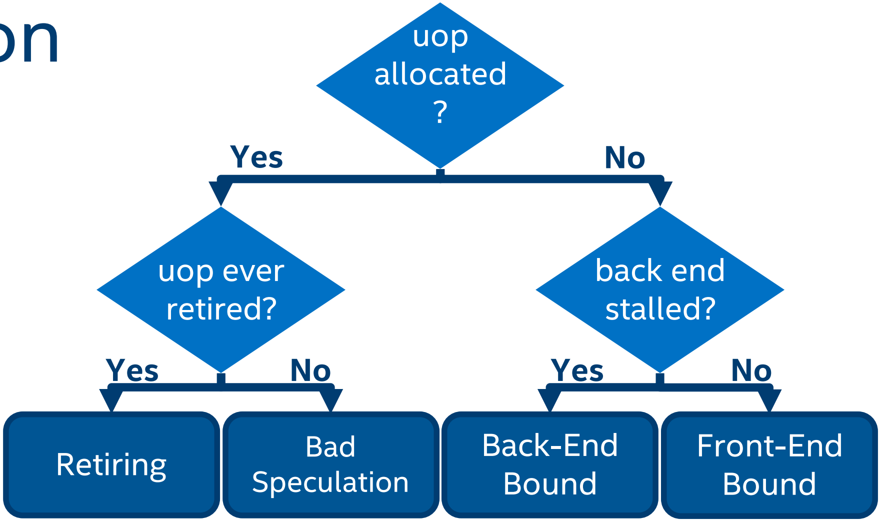
Address Hardware Issues


FRONT END
BACK END
EXECUTION UNIT
RETIREMENT
uop
uop
uop
uop
Fetch & Decode Instructions, Predict Branches
Re-order and Execute Instructions
Commit Results to Memory

The X86 Processor Pipeline (simplified)

Pipeline Slot Categorization

- Pipeline slots can be sorted into one of four categories on a given cycle by what happens to the uop in that slot.
  - Retiring
  - Bad Speculation
  - Back End Bound
  - Front End Bound
- Each category has an expected range of values in a well tuned application.



Category \ App. Type:	Client/Desktop	Server/Database/ Distributed	High Performance Computing
Retiring ▲	20-50%	10-30%	30-70%
Bad Speculation ▼	5-10%	5-10%	1-5%
Front End Bound ▼	5-10%	10-25%	5-10%
Back End Bound ▼	20-40%	20-60%	20-40%

The uop Pipeline

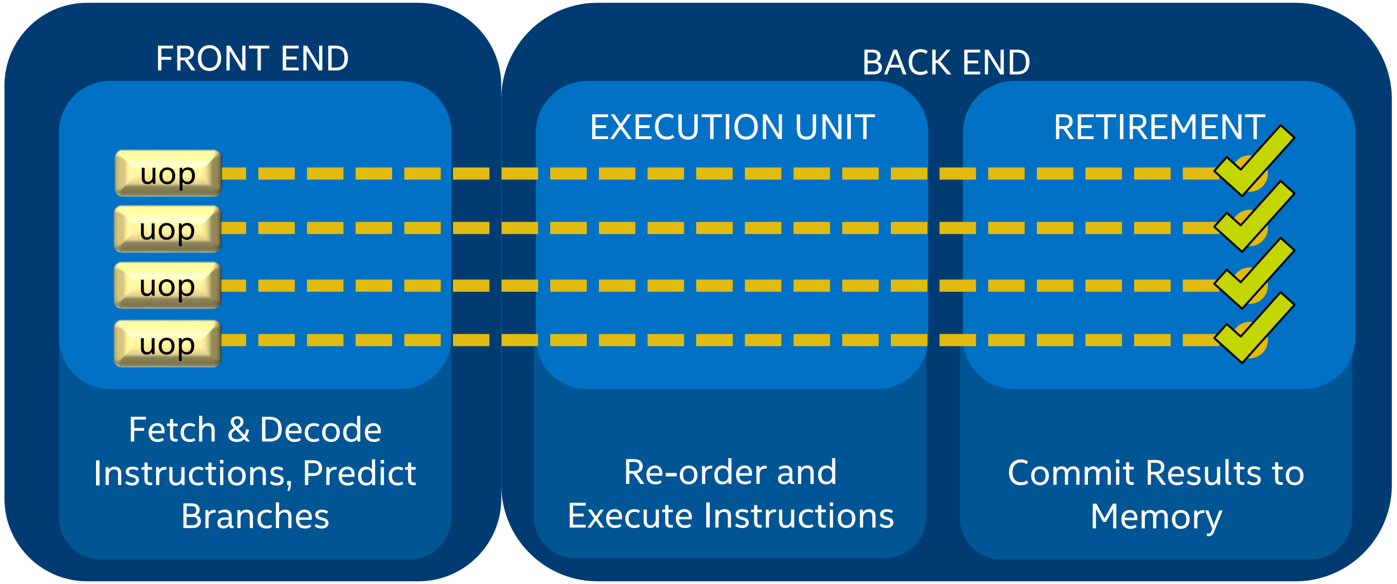
Categorizing the hotspots

- Modern CPUs “pipeline” instructions. This pipeline can be generally divided into two sections.
  - The Front End fetches instructions, decodes them into uops, and allocates them to…
  - The Back End, which is responsible for executing the uops. Once successfully completed, a uop is considered “retired”.
- A Pipeline Slot is an abstract representation of the hardware resources needed to process a uop.
- The front end can only allocate so many uops per cycle, and the same is true of the back end and retiring them. This determines the number of Pipeline Slots. As a general rule, this number is four.

Pipeline Slot Categorization

Retiring

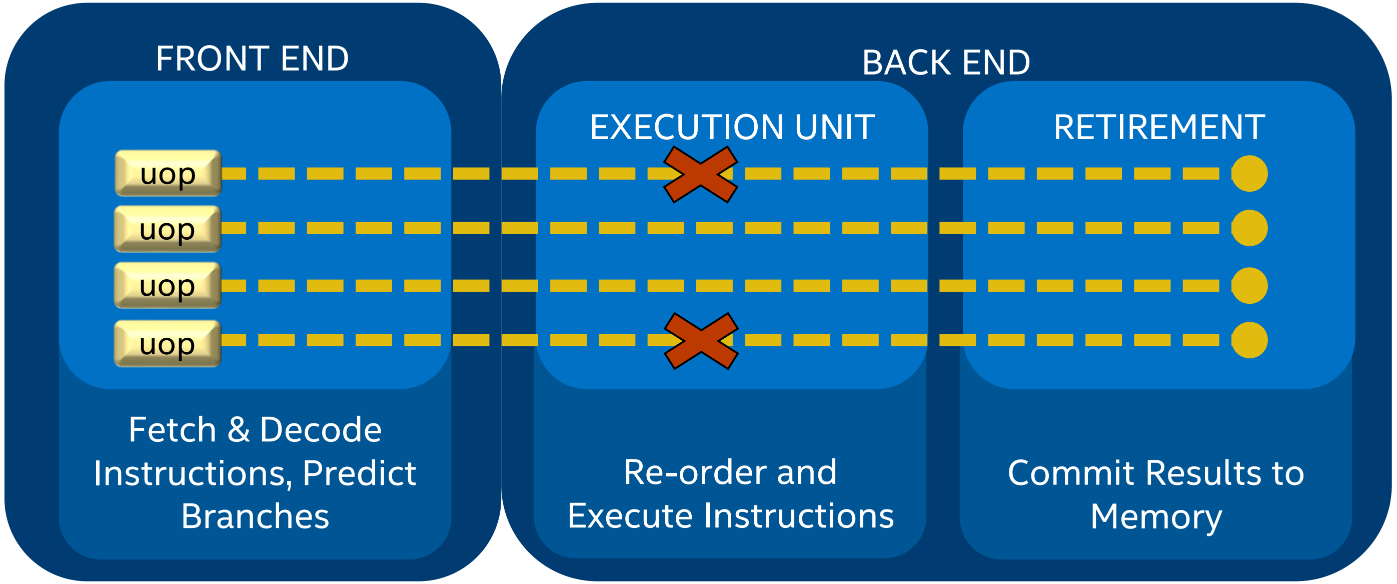
This is the good category! You want as many of your slots in this category as possible. However, even here there may be room for optimization.


FRONT END
BACK END
EXECUTION UNIT
RETIREMENT
uop
uop
uop
uop
Fetch & Decode Instructions, Predict Branches
Re-order and Execute Instructions
Commit Results to Memory

Pipeline Slot Categorization

Bad Speculation

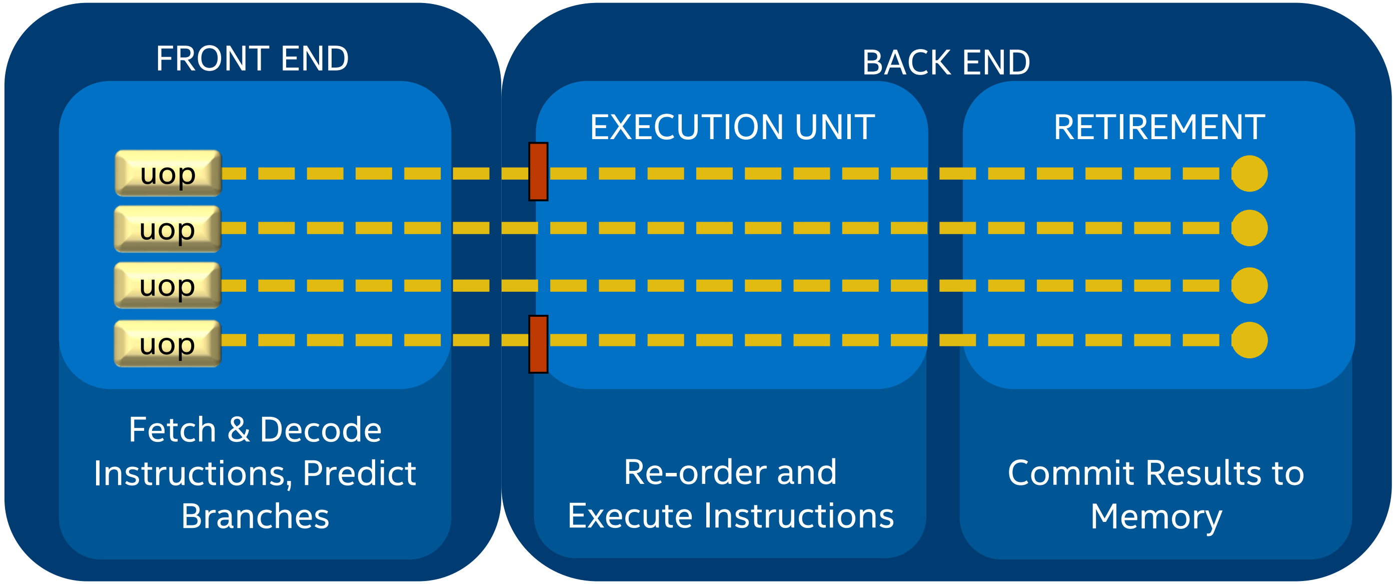
This occurs when a uop is removed from the back end without retiring; effectively, it's cancelled, most often because a branch was mispredicted.


FRONT END
BACK END
EXECUTION UNIT
RETIREMENT
uop
uop
uop
uop
Fetch & Decode Instructions, Predict Branches
Re-order and Execute Instructions
Commit Results to Memory

Pipeline Slot Categorization

Back End Bound

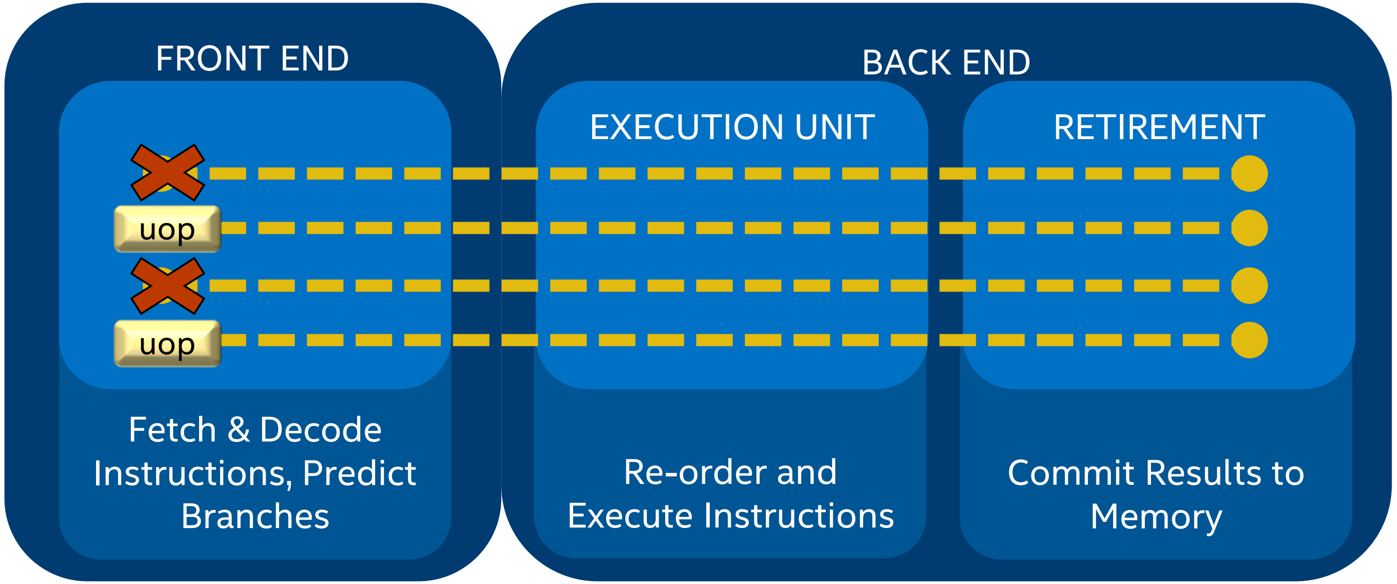
This is when the back end can't accept uops, even if the front end can send them, because it already contains uops waiting on data or long execution.


FRONT END
BACK END
EXECUTION UNIT
RETIREMENT
uop
uop
uop
uop
Fetch & Decode Instructions, Predict Branches
Re-order and Execute Instructions
Commit Results to Memory

Pipeline Slot Categorization

Front End Bound

This is when the front end can't deliver uops even though the back end can take them, usually due to delays in fetching code or decoding instructions.


FRONT END
BACK END
EXECUTION UNIT
RETIREMENT
uop
uop
Fetch & Decode Instructions, Predict Branches
Re-order and Execute Instructions
Commit Results to Memory

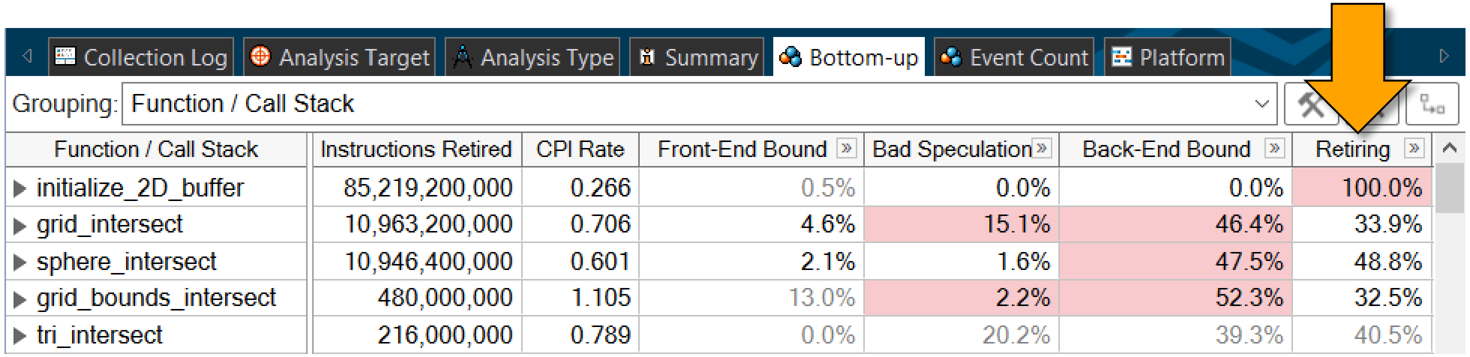
Identifying and Diagnosing Inefficiency

Microarchitecture Analysis

```
> amplxe-cl -collect uarch-exploration -- ./myapp.out
```

- Microarchitecture Exploration (previously General Exploration) is a hardware events analysis. It is preconfigured to sample the appropriate events on your architecture and calculates the proper metrics from them.
- Potential tuning opportunities are highlighted in pink.
- To check the efficiency of a hotspot, look at the Retiring metric. If it's less than the expected number for your application type, it's probably inefficient.
  - Hotspots with high retiring values may still have room for improvement.

App Type	Expected
Client/ Desktop	20-50%
Server/ Database/ Distributed	10-30%
HPC	30-70%



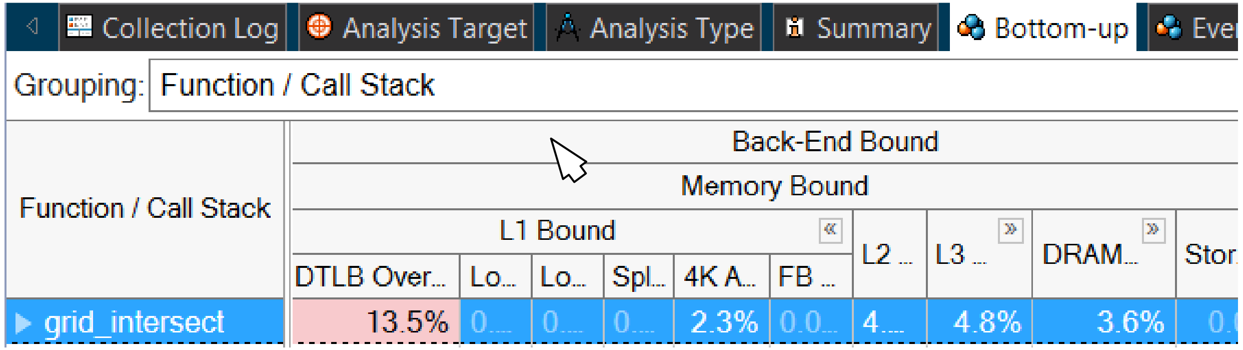
Grouping: Function / Call Stack

Function / Call Stack	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound	Retiring
initialize_2D_buffer	85,219,200,000	0.266	0.5%	0.0%	0.0%	100.0%
grid_intersect	10,963,200,000	0.706	4.6%	15.1%	46.4%	33.9%
sphere_intersect	10,946,400,000	0.601	2.1%	1.6%	47.5%	48.8%
grid_bounds_intersect	480,000,000	1.105	13.0%	2.2%	52.3%	32.5%
tri_intersect	216,000,000	0.789	0.0%	20.2%	39.3%	40.5%

Categorizing and Correcting Inefficiencies

Microarchitecture Exploration Analysis

- Intel® VTune™ Amplifier has hierarchical expanding metrics categorized by the four slot types.
- You can expand your way down, following the hotspot, to identify the root cause of the inefficiency.
  - Sub-metrics highlight pink on their own merits, just like top level metrics.
- Hovering over a metric produces a helpful, detailed tooltip (not shown).
  - There are tooltips on Summary tabs too: hover over any ⓘ icon.



Grouping: Function / Call Stack

Function / Call Stack	Back-End Bound										
	Memory Bound										
	L1 Bound						L2 ...	L3 ...	DRAM...	Stor	
	DTLB Over...	Lo...	Lo...	Spl...	4K A...	FB ...					
grid_intersect	13.5%	0....	0....	0....	2.3%	0.0...	4....	4.8%	3.6%	0.	

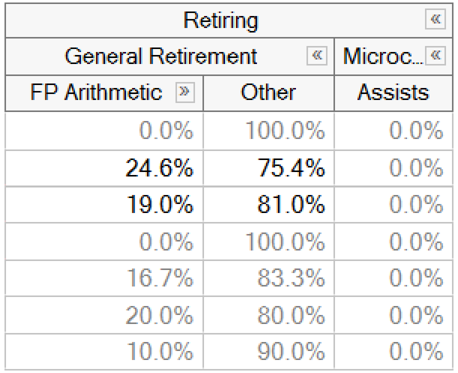
Categorizing and Correcting Inefficiencies

Retiring: *Microarchitecture Exploration Analysis, Intel® Advisor*

- High Retiring percentage is generally good, but may be inefficient if you're doing work that doesn't need to be done at all, or could be done faster.
- Retiring can be split based on whether the uops being retired came from the microcode sequencer or not.
  - **Yes?** Try reworking code to avoid microcode assists.
  - **No?** Make sure the code is well vectorized.

Tip:

Use Vectorization Advisor to fine-tune your vectorization.



Retiring		
General Retirement		Microc...
FP Arithmetic	Other	Assists
0.0%	100.0%	0.0%
24.6%	75.4%	0.0%
19.0%	81.0%	0.0%
0.0%	100.0%	0.0%
16.7%	83.3%	0.0%
20.0%	80.0%	0.0%
10.0%	90.0%	0.0%

HPC Characterization: FPU Utilization

FPU utilization

% of FPU load (100% - FPU is fully loaded, threshold 50%)

Calculation based on PMU events representing scalar and packed single and double precision SIMD instructions

Metrics in FPU utilization section

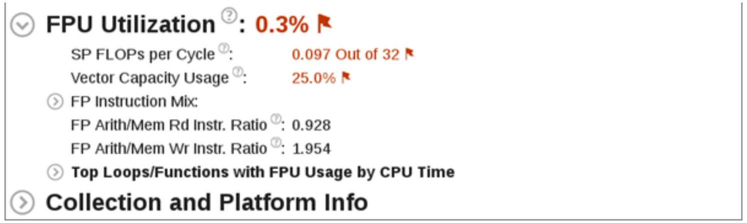
FLOPs broken down by scalar and packed

Instruction Mix

Top 5 loops/functions by FPU usage

- Detected with static binary analysis

Vectorized vs. Non-vectorized, ISA, and characterization detected by static analysis and Intel Compiler diagnostics

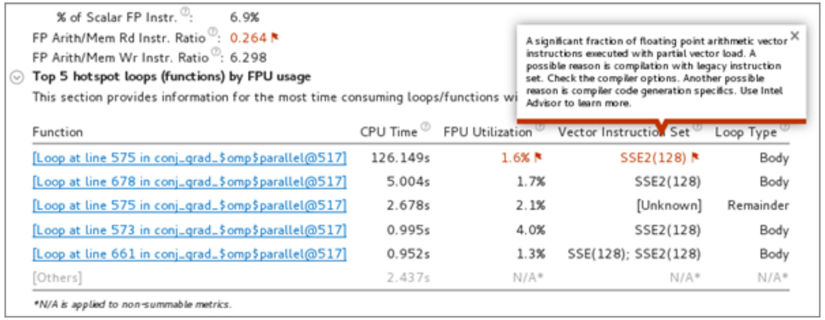


FPU Utilization: 0.3%

SP FLOPs per Cycle:	0.097 Out of 32
Vector Capacity Usage:	25.0%
FP Instruction Mix:	
FP Arith/Mem Rd Instr. Ratio:	0.928
FP Arith/Mem Wr Instr. Ratio:	1.954

Top Loops/Functions with FPU Usage by CPU Time

Collection and Platform Info



% of Scalar FP Instr.:	6.9%
FP Arith/Mem Rd Instr. Ratio:	0.264
FP Arith/Mem Wr Instr. Ratio:	6.298

Top 5 hotspot loops (functions) by FPU usage

This section provides information for the most time consuming loops/functions wi

A significant fraction of floating point arithmetic vector instructions executed with partial vector load. A possible reason is compilation with legacy instruction set. Check the compiler options. Another possible reason is compiler code generation specifics. Use Intel Advisor to learn more.

Function	CPU Time	FPU Utilization	Vector Instruction Set	Loop Type
[Loop at line 575 in conj_grad_ompparallel@517]	126.149s	1.6%	SSE2(128)	Body
[Loop at line 678 in conj_grad_ompparallel@517]	5.004s	1.7%	SSE2(128)	Body
[Loop at line 575 in conj_grad_ompparallel@517]	2.678s	2.1%	[Unknown]	Remainder
[Loop at line 573 in conj_grad_ompparallel@517]	0.995s	4.0%	SSE2(128)	Body
[Loop at line 661 in conj_grad_ompparallel@517]	0.952s	1.3%	SSE(128); SSE2(128)	Body
[Others]	2.437s	N/A*	N/A*	N/A*

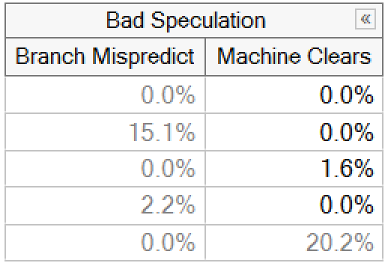
*N/A is applied to non-summable metrics.

HARDWARE IS BECOMING MORE VECTORIZED, SO SHOULD YOU!

Categorizing and Correcting Inefficiencies

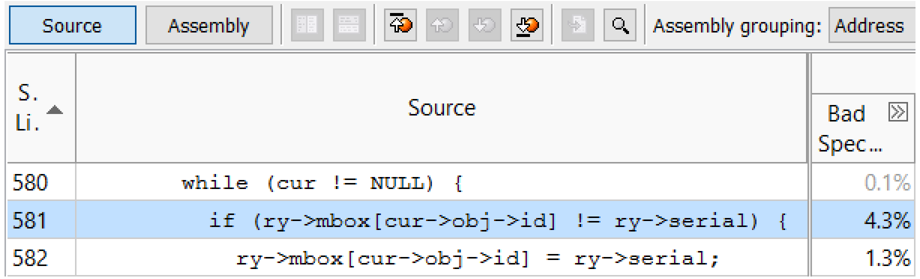
Bad Speculation: *Microarchitecture Exploration Analysis*

- Bad Speculation is caused by either Machine Clears or Branch Mispredicts.
  - **Machine Clears** can be caused by self-modifying code, etc.
  - **Branch mispredicts** are more common. These occur when the paths taken by `if`, `switch`, `for`, `do-while`, and other conditional branches are incorrectly predicted and the uops have to be thrown out.



Bad Speculation	
Branch Mispredict	Machine Clears
0.0%	0.0%
15.1%	0.0%
0.0%	1.6%
2.2%	0.0%
0.0%	20.2%

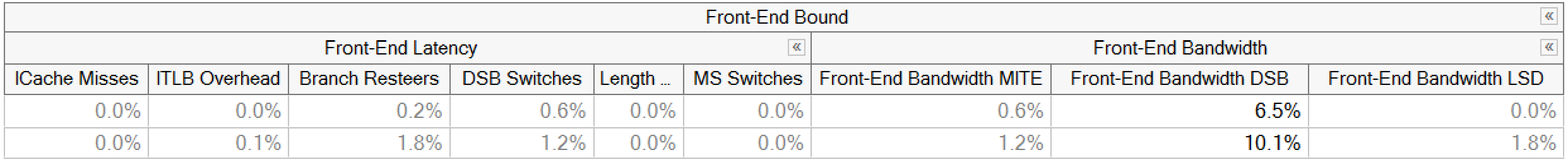
- Use Intel® VTune™ Amplifier's Source Viewer to identify problematic branches.
- Avoid unnecessary branching:
  - Remove branches entirely if possible
  - Move branches outside of loops if possible.



S. Li.	Source	Bad Spec...
580	`while (cur != NULL) {`	0.1%
581	`if (ry->mbox[cur->obj->id] != ry->serial) {`	4.3%
582	`ry->mbox[cur->obj->id] = ry->serial;`	1.3%

Categorizing and Correcting Inefficiencies

Front End: *Microarchitecture Exploration Analysis*

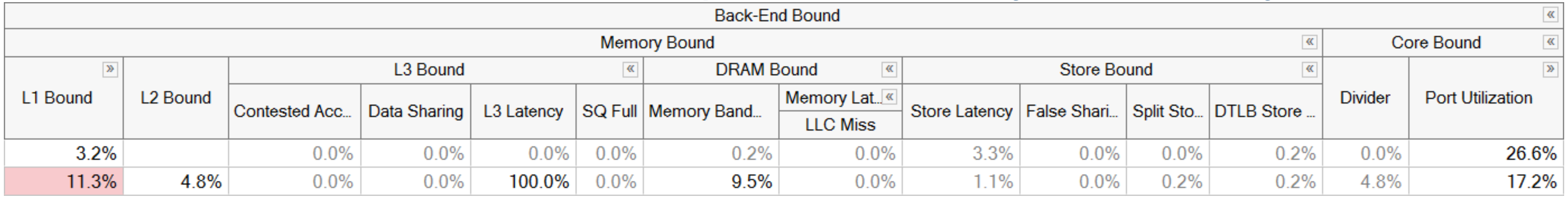


Front-End Bound								
Front-End Latency						Front-End Bandwidth		
ICache Misses	ITLB Overhead	Branch Resteers	DSB Switches	Length ...	MS Switches	Front-End Bandwidth MITE	Front-End Bandwidth DSB	Front-End Bandwidth LSD
0.0%	0.0%	0.2%	0.6%	0.0%	0.0%	0.6%	6.5%	0.0%
0.0%	0.1%	1.8%	1.2%	0.0%	0.0%	1.2%	10.1%	1.8%

- Front End Bound pipeline slots are common in JIT or interpreted code.
- Front End Bound can be bandwidth or latency:
  - **Bandwidth** issues are caused by inefficient instruction decoding, or restrictions in caching decoded instructions, etc.
  - **Latency** is caused by instruction cache misses, delays in instruction fetching after branch mispredicts, switching to the microcode sequencer too often, etc.

Categorizing and Correcting Inefficiencies

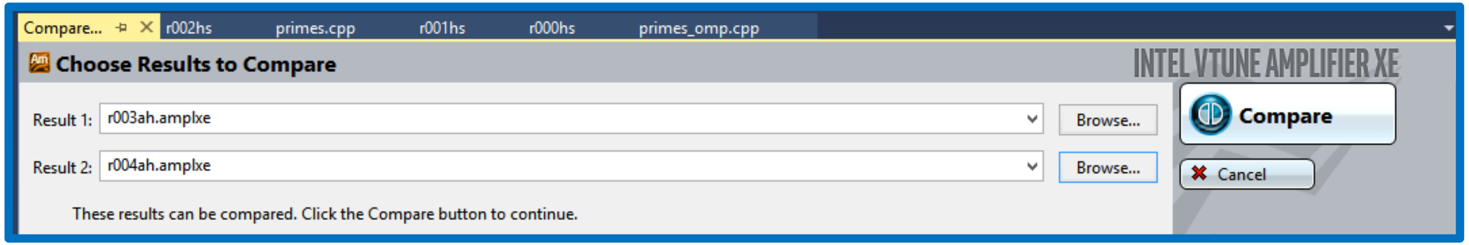
Back End: *Microarchitecture Exploration Analysis, Memory Bandwidth*



Back-End Bound													
Memory Bound												Core Bound	
L1 Bound	L2 Bound	L3 Bound				DRAM Bound		Store Bound				Divider	Port Utilization
		Contested Acc...	Data Sharing	L3 Latency	SQ Full	Memory Band...	Memory Lat...	Store Latency	False Shari...	Split Sto...	DTLB Store ...		
							LLC Miss						
3.2%		0.0%	0.0%	0.0%	0.0%	0.2%	0.0%	3.3%	0.0%	0.0%	0.2%	0.0%	26.6%
11.3%	4.8%	0.0%	0.0%	100.0%	0.0%	9.5%	0.0%	1.1%	0.0%	0.2%	0.2%	4.8%	17.2%

- Back End bound is the most common bottleneck type for most applications.
- It can be split into Core Bound and Memory Bound
  - **Core Bound** includes issues like not using execution units effectively and performing too many divides.
  - **Memory Bound** involves cache misses, inefficient memory accesses, etc.
    - Store Bound is when load-store dependencies are slowing things down.
    - The other sub-categories involve caching issues and the like. Memory Access Analysis may provide additional information for resolving this performance bottleneck.

Rebuild and Compare Results

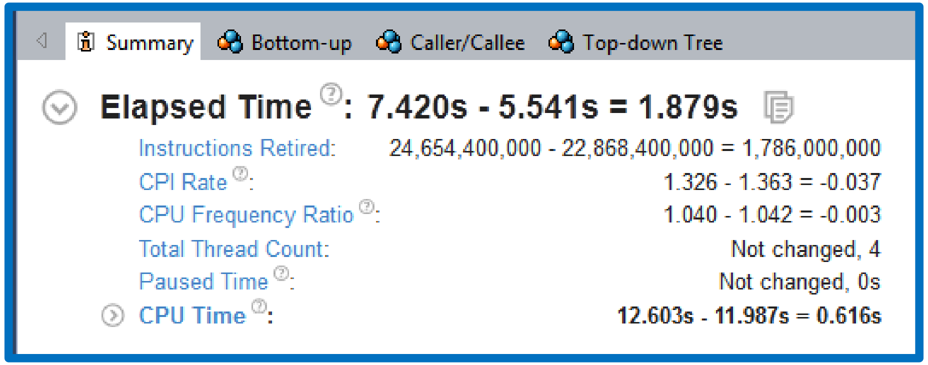


Choose Results to Compare

Result 1: r003ah.amplxe

Result 2: r004ah.amplxe

These results can be compared. Click the Compare button to continue.

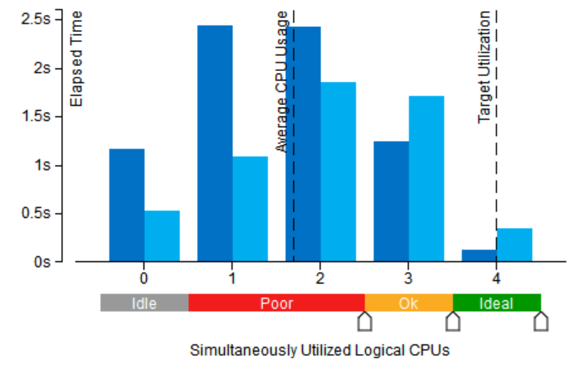


Elapsed Time: 7.420s - 5.541s = 1.879s

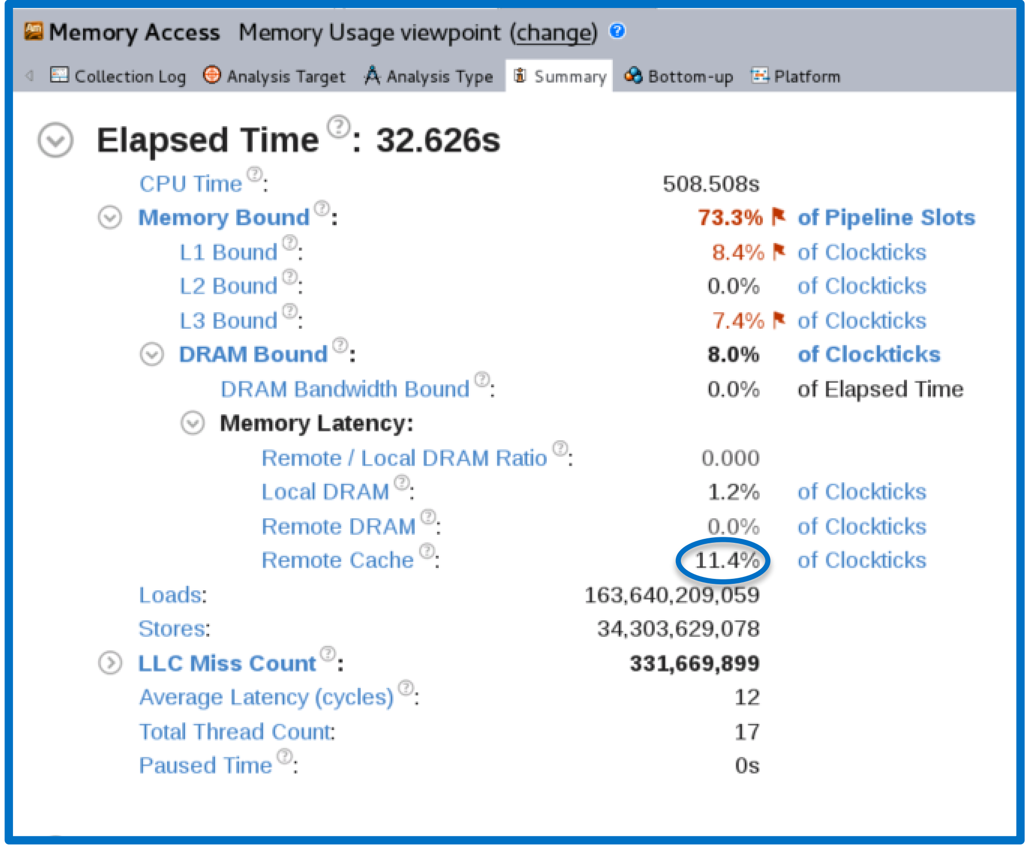
Instructions Retired:	24,654,400,000 - 22,868,400,000 = 1,786,000,000
CPI Rate:	1.326 - 1.363 = -0.037
CPU Frequency Ratio:	1.040 - 1.042 = -0.003
Total Thread Count:	Not changed, 4
Paused Time:	Not changed, 0s
CPU Time:	12.603s - 11.987s = 0.616s

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running



Example: Poor NUMA Utilization



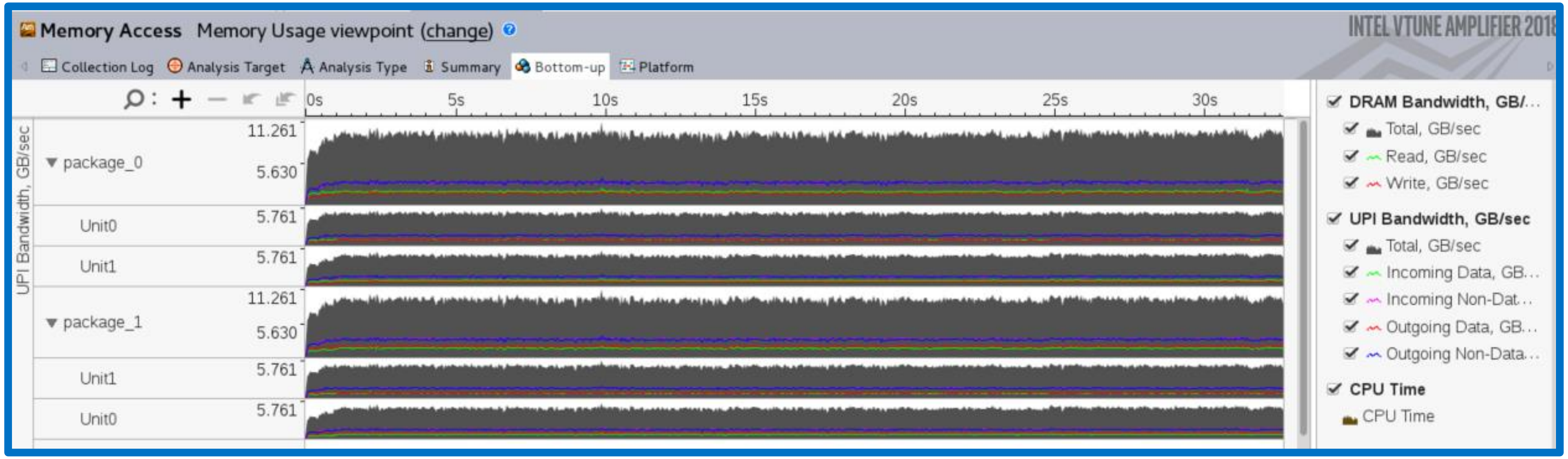
Elapsed Time: 32.626s

CPU Time:	508.508s	
Memory Bound:	**73.3%**	**of Pipeline Slots**
L1 Bound:	8.4%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	7.4%	of Clockticks
DRAM Bound:	**8.0%**	**of Clockticks**
DRAM Bandwidth Bound:	0.0%	of Elapsed Time
Memory Latency:		
Remote / Local DRAM Ratio:	0.000	
Local DRAM:	1.2%	of Clockticks
Remote DRAM:	0.0%	of Clockticks
Remote Cache:	11.4%	of Clockticks
Loads:	163,640,209,059	
Stores:	34,303,629,078	
LLC Miss Count:	**331,669,899**	
Average Latency (cycles):	12	
Total Thread Count:	17	
Paused Time:	0s	

If Memory Bound is high and local caches are not the problem

Focus on "Remote" metrics

Example: Poor NUMA Utilization



Look for areas of high QPI/UPI bandwidth

QPI/UPI BANDWIDTH IS COMMUNICATION BETWEEN THE SOCKETS. THIS MAY INDICATE SOME SORT OF NUMA ISSUE.

EXAMPLE: POOR NUMA UTILIZATION

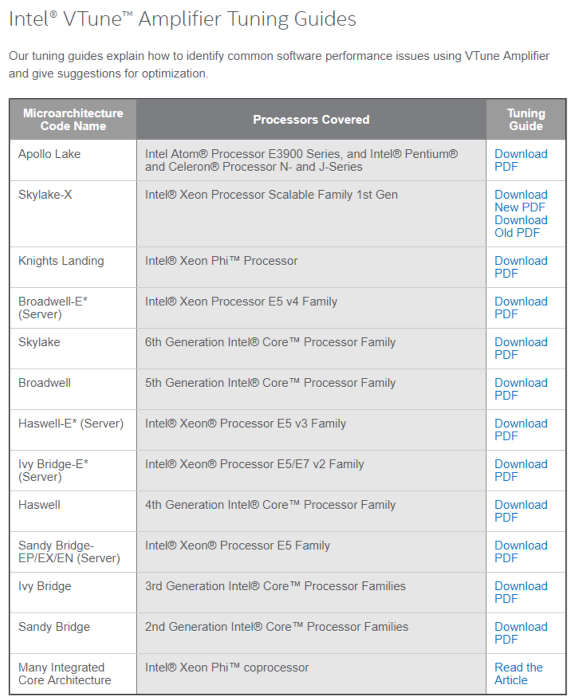
Common causes of poor NUMA utilization

- Allocation vs. first touch memory location
- False sharing of cache lines
  - Use padding when necessary
- Arbitrary array accesses
- Poor thread affinity

WHERE IS YOUR MEMORY ALLOCATED AND WHERE ARE YOUR THREADS RUNNING?

Tuning Guides Available Online

- http://intel.com/vtune-tuning-guides

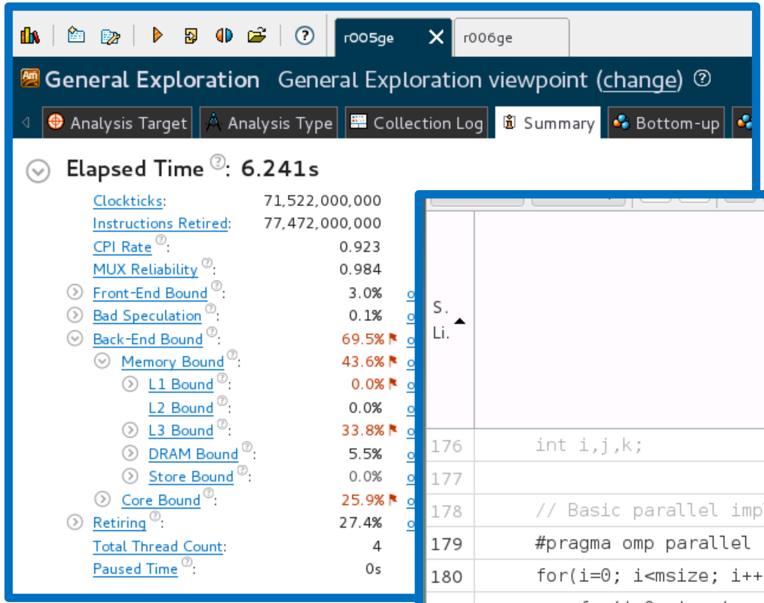


Intel® VTune™ Amplifier Tuning Guides

Our tuning guides explain how to identify common software performance issues using VTune Amplifier and give suggestions for optimization.

Microarchitecture Code Name	Processors Covered	Tuning Guide
Apollo Lake	Intel Atom® Processor E3900 Series, and Intel® Pentium® and Celeron® Processor N- and J-Series	Download PDF
Skylake-X	Intel® Xeon Processor Scalable Family 1st Gen	Download New PDF Download Old PDF
Knights Landing	Intel® Xeon Phi™ Processor	Download PDF
Broadwell-E* (Server)	Intel® Xeon Processor E5 v4 Family	Download PDF
Skylake	6th Generation Intel® Core™ Processor Family	Download PDF
Broadwell	5th Generation Intel® Core™ Processor Family	Download PDF
Haswell-E* (Server)	Intel® Xeon® Processor E5 v3 Family	Download PDF
Ivy Bridge-E* (Server)	Intel® Xeon® Processor E5/E7 v2 Family	Download PDF
Haswell	4th Generation Intel® Core™ Processor Family	Download PDF
Sandy Bridge-EP/EX/EN (Server)	Intel® Xeon® Processor E5 Family	Download PDF
Ivy Bridge	3rd Generation Intel® Core™ Processor Families	Download PDF
Sandy Bridge	2nd Generation Intel® Core™ Processor Families	Download PDF
Many Integrated Core Architecture	Intel® Xeon Phi™ coprocessor	Read the Article

Example 1 – Matrix Multiply



r005ge | r006ge

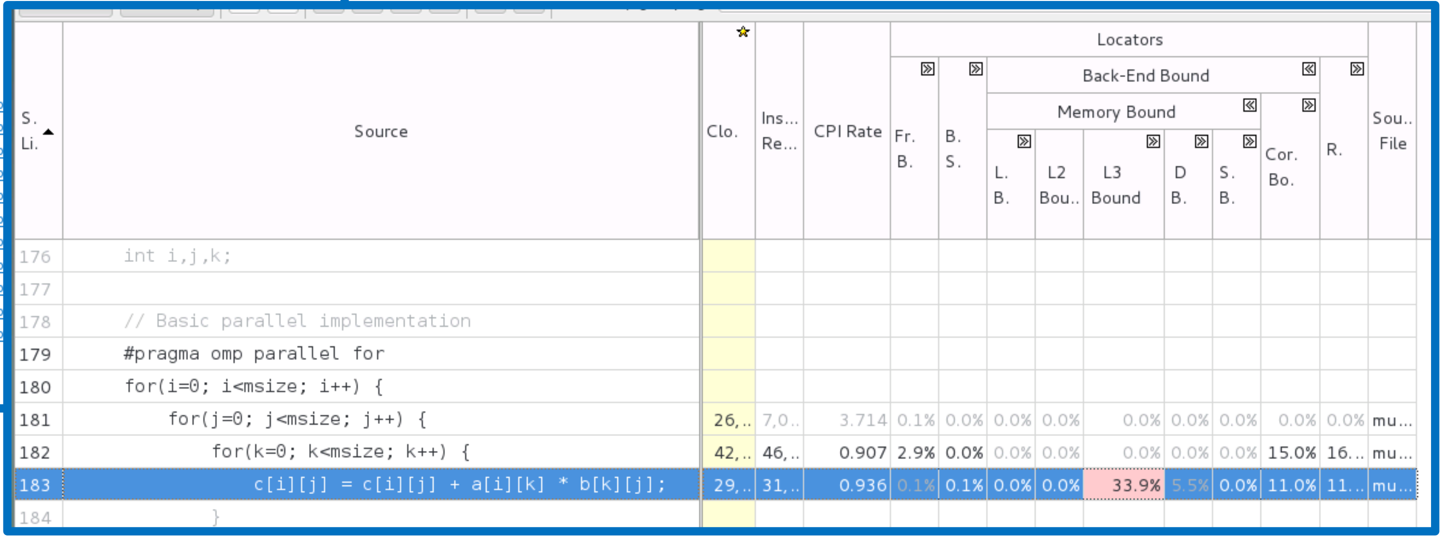
General Exploration General Exploration viewpoint (change)

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

Elapsed Time: 6.241s

Clockticks:	71,522,000,000
Instructions Retired:	77,472,000,000
CPI Rate:	0.923
MUX Reliability:	0.984
Front-End Bound:	3.0%
Bad Speculation:	0.1%
Back-End Bound:	69.5% ⚑
Memory Bound:	43.6% ⚑
L1 Bound:	0.0% ⚑
L2 Bound:	0.0%
L3 Bound:	33.8% ⚑
DRAM Bound:	5.5%
Store Bound:	0.0%
Core Bound:	25.9% ⚑
Retiring:	27.4%
Total Thread Count:	4
Paused Time:	0s

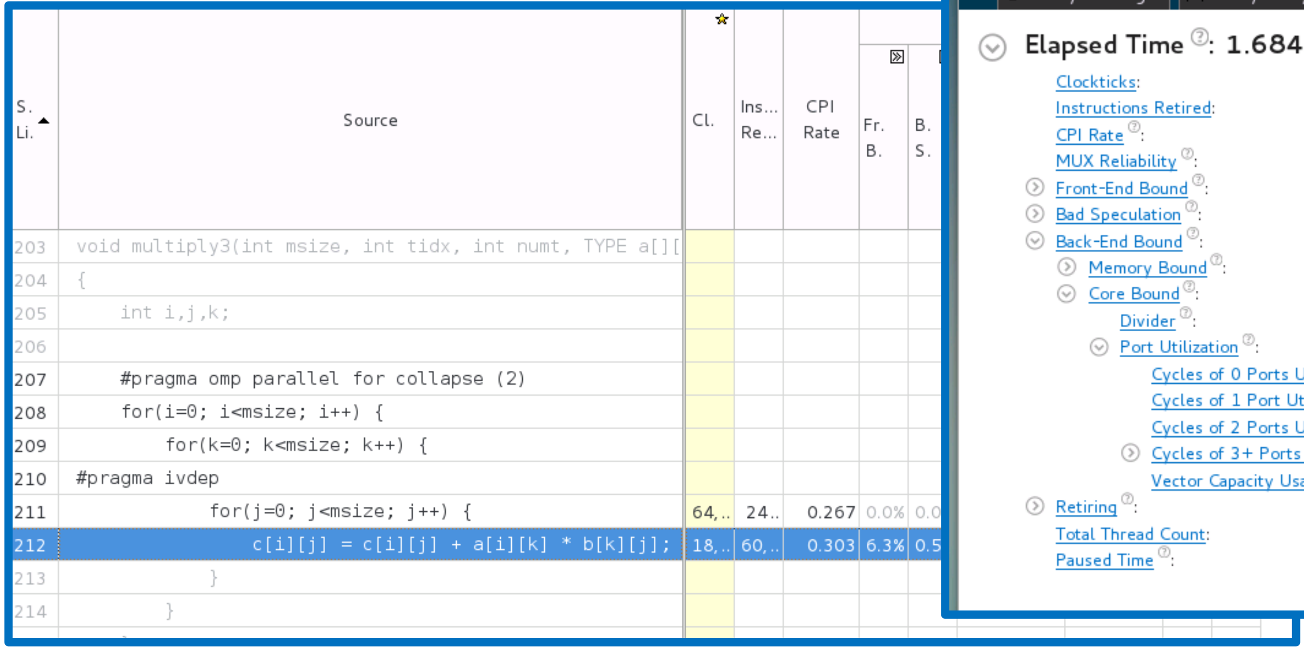
Bound by the L3 Cache while reading arrays



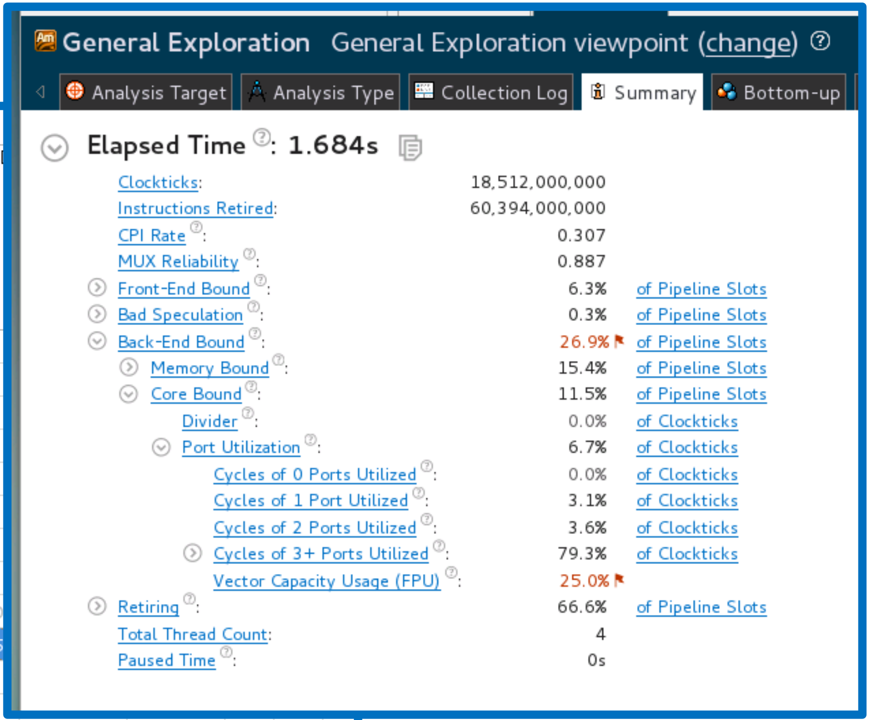
S. Li.	Source	Clo.	Ins... Re...	CPI Rate	Locators									Sou.. File
							Back-End Bound							
							Memory Bound							
					Fr. B.	B. S.	L. B.	L2 Bou..	L3 Bound	D B.	S. B.	Cor. Bo.	R.	
176	int i,j,k;													
177														
178	// Basic parallel implementation													
179	#pragma omp parallel for													
180	for(i=0; i<msize; i++) {													
181	for(j=0; j<msize; j++) {	26,..	7,0..	3.714	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	mu...
182	for(k=0; k<msize; k++) {	42,..	46,..	0.907	2.9%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	15.0%	16...	mu...
183	c[i][j] = c[i][j] + a[i][k] * b[k][j];	29,..	31,..	0.936	0.1%	0.1%	0.0%	0.0%	33.9%	5.5%	0.0%	11.0%	11...	mu...
184	}													

Example 1 – Matrix Multiply

Interchange loop indices and collapse loops



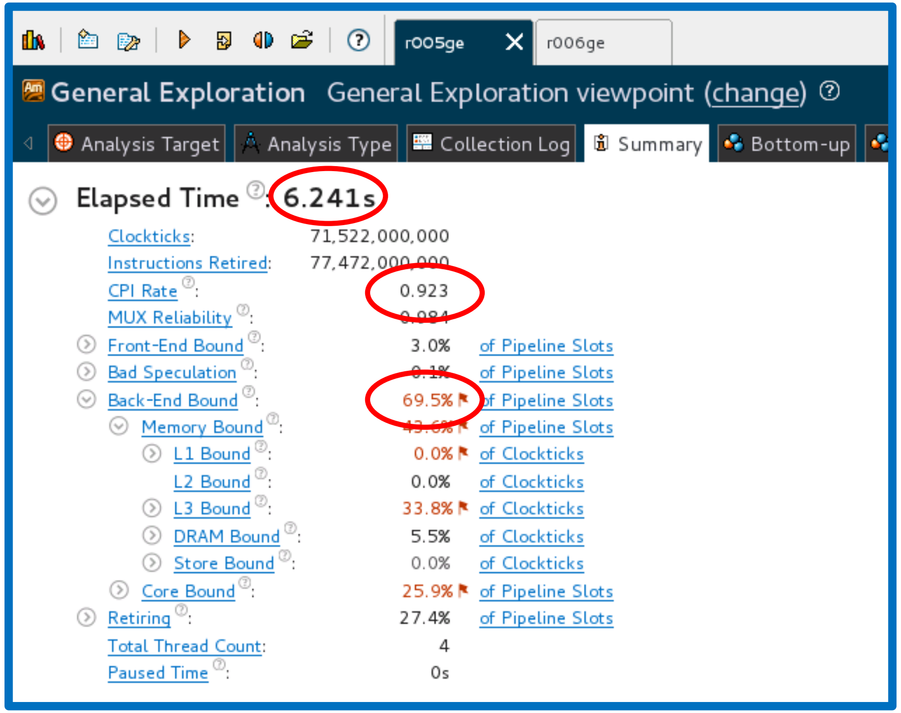
S. Li.	Source	Cl.	Ins... Re...	CPI Rate	Fr. B.	B. S.
203	`void multiply3(int msize, int tidx, int numt, TYPE a[][`					
204	`{`					
205	`    int i,j,k;`					
206						
207	`    #pragma omp parallel for collapse (2)`					
208	`    for(i=0; i<msize; i++) {`					
209	`        for(k=0; k<msize; k++) {`					
210	`#pragma ivdep`					
211	`            for(j=0; j<msize; j++) {`	64,..	24..	0.267	0.0%	0.0
212	`                c[i][j] = c[i][j] + a[i][k] * b[k][j];`	18,..	60,..	0.303	6.3%	0.5
213	`            }`					
214	`        }`					



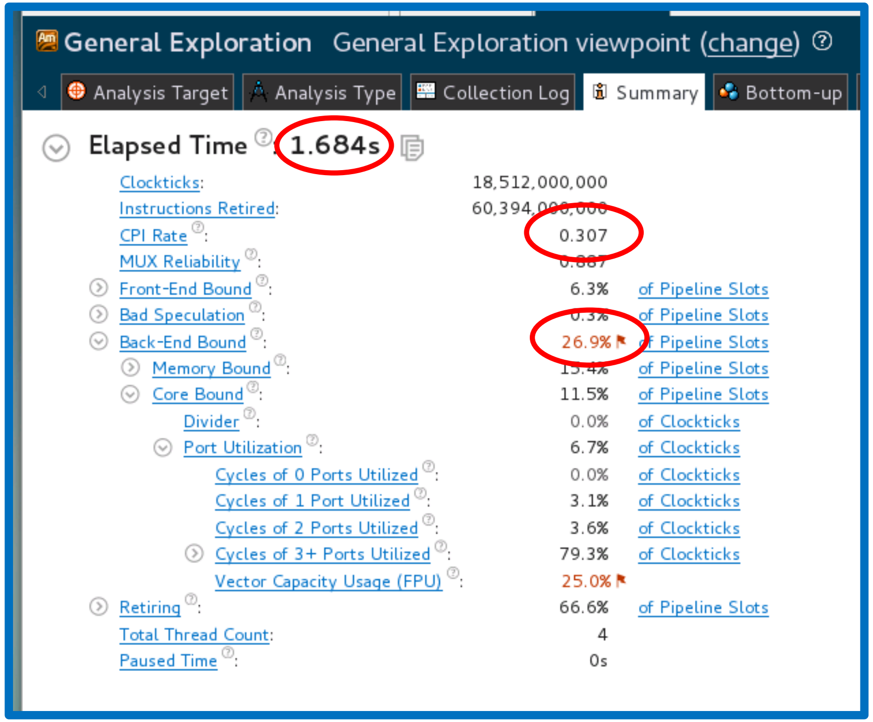
Elapsed Time: 1.684s

Clockticks:	18,512,000,000	
Instructions Retired:	60,394,000,000	
CPI Rate:	0.307	
MUX Reliability:	0.887	
Front-End Bound:	6.3%	of Pipeline Slots
Bad Speculation:	0.3%	of Pipeline Slots
Back-End Bound:	26.9%	of Pipeline Slots
Memory Bound:	15.4%	of Pipeline Slots
Core Bound:	11.5%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	6.7%	of Clockticks
Cycles of 0 Ports Utilized:	0.0%	of Clockticks
Cycles of 1 Port Utilized:	3.1%	of Clockticks
Cycles of 2 Ports Utilized:	3.6%	of Clockticks
Cycles of 3+ Ports Utilized:	79.3%	of Clockticks
Vector Capacity Usage (FPU):	25.0%	
Retiring:	66.6%	of Pipeline Slots
Total Thread Count:	4	
Paused Time:	0s	

Example 1 – Matrix Multiply



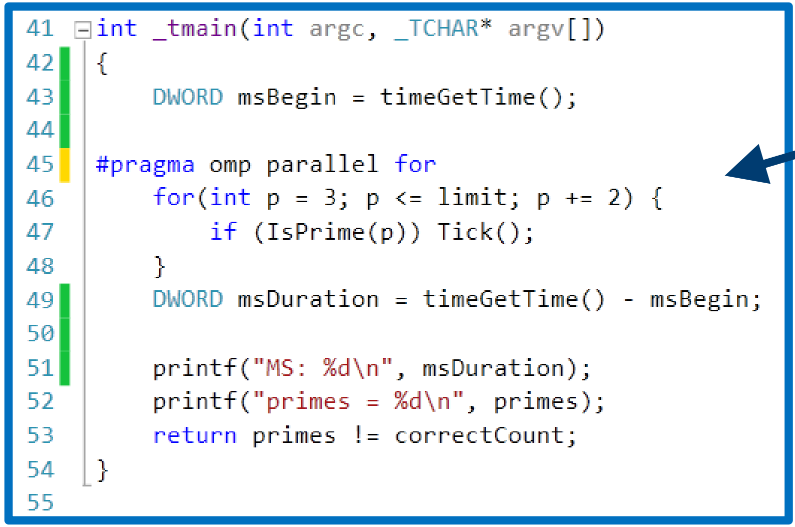
Elapsed Time:	6.241s	
Clockticks:	71,522,000,000	
Instructions Retired:	77,472,000,000	
CPI Rate:	0.923	
MUX Reliability:	0.984	
Front-End Bound:	3.0%	of Pipeline Slots
Bad Speculation:	0.1%	of Pipeline Slots
Back-End Bound:	69.5% ⚑	of Pipeline Slots
Memory Bound:	43.6% ⚑	of Pipeline Slots
L1 Bound:	0.0% ⚑	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	33.8% ⚑	of Clockticks
DRAM Bound:	5.5%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	25.9% ⚑	of Pipeline Slots
Retiring:	27.4%	of Pipeline Slots
Total Thread Count:	4	
Paused Time:	0s	



Elapsed Time:	1.684s	
Clockticks:	18,512,000,000	
Instructions Retired:	60,394,000,000	
CPI Rate:	0.307	
MUX Reliability:	0.887	
Front-End Bound:	6.3%	of Pipeline Slots
Bad Speculation:	0.3%	of Pipeline Slots
Back-End Bound:	26.9% ⚑	of Pipeline Slots
Memory Bound:	15.4%	of Pipeline Slots
Core Bound:	11.5%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	6.7%	of Clockticks
Cycles of 0 Ports Utilized:	0.0%	of Clockticks
Cycles of 1 Port Utilized:	3.1%	of Clockticks
Cycles of 2 Ports Utilized:	3.6%	of Clockticks
Cycles of 3+ Ports Utilized:	79.3%	of Clockticks
Vector Capacity Usage (FPU):	25.0% ⚑	
Retiring:	66.6%	of Pipeline Slots
Total Thread Count:	4	
Paused Time:	0s	

Optimization Notice

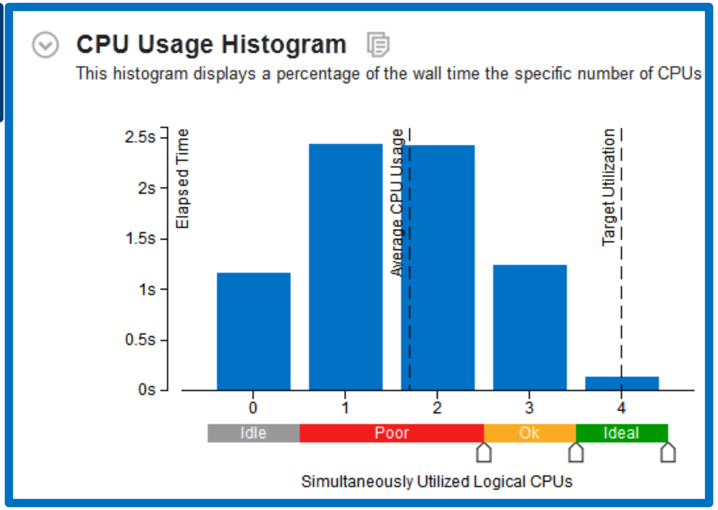
Example 2 – Calculating Prime Numbers



```
int _tmain(int argc, _TCHAR* argv[])
{
    DWORD msBegin = timeGetTime();

#pragma omp parallel for
    for(int p = 3; p <= limit; p += 2) {
        if (IsPrime(p)) Tick();
    }
    DWORD msDuration = timeGetTime() - msBegin;

    printf("MS: %d\n", msDuration);
    printf("primes = %d\n", primes);
    return primes != correctCount;
}
```

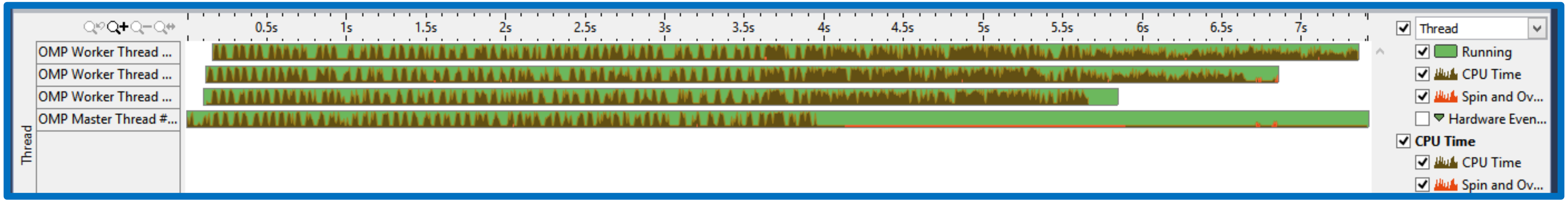
OpenMP uses Static Scheduling

Load Imbalance

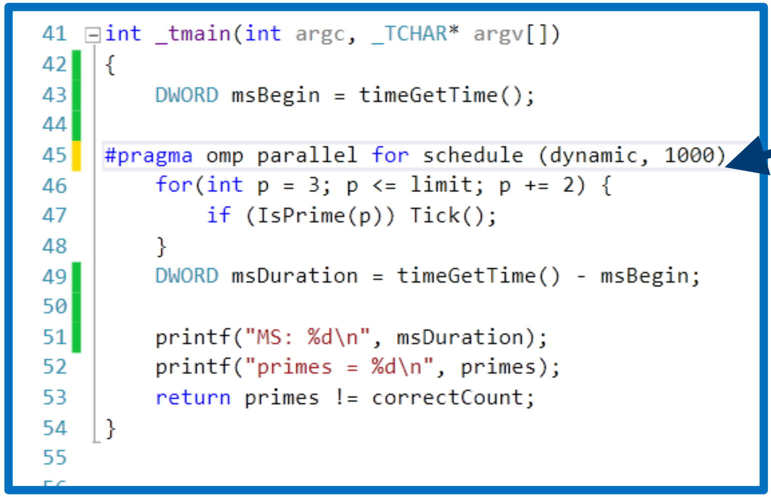


CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs

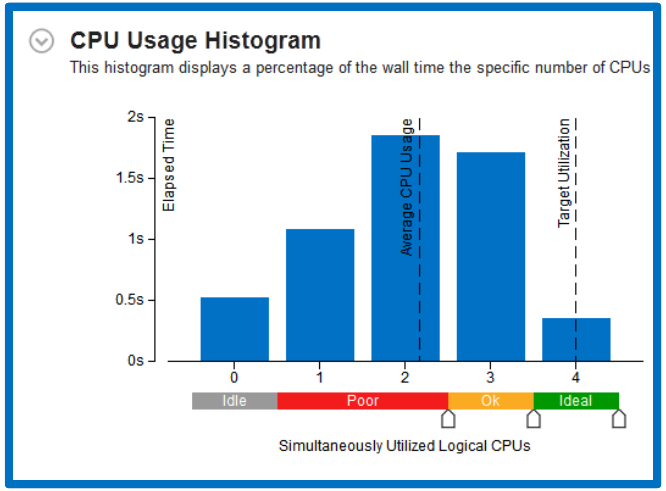


Example 2 – Calculating Prime Numbers

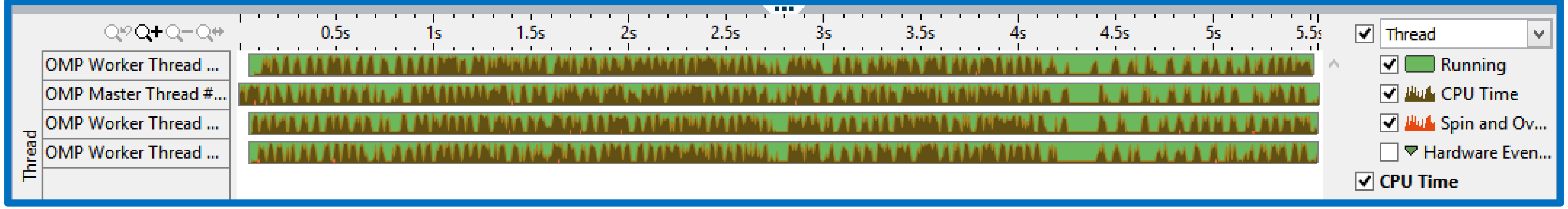


```
int _tmain(int argc, _TCHAR* argv[])
{
    DWORD msBegin = timeGetTime();

#pragma omp parallel for schedule (dynamic, 1000)
    for(int p = 3; p <= limit; p += 2) {
        if (IsPrime(p)) Tick();
    }
    DWORD msDuration = timeGetTime() - msBegin;

    printf("MS: %d\n", msDuration);
    printf("primes = %d\n", primes);
    return primes != correctCount;
}
```

Switch to Dynamic Scheduling

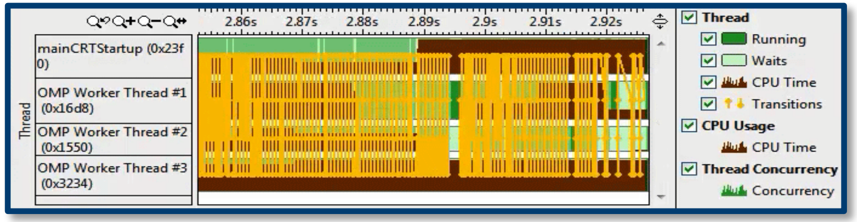


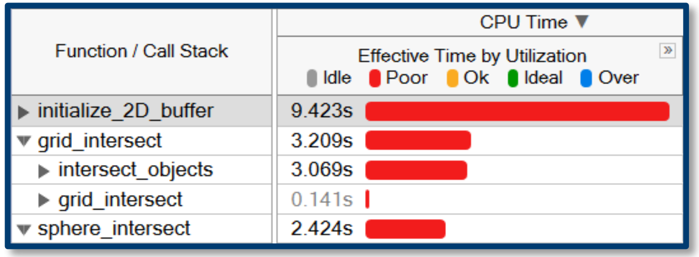
More Balanced Threads



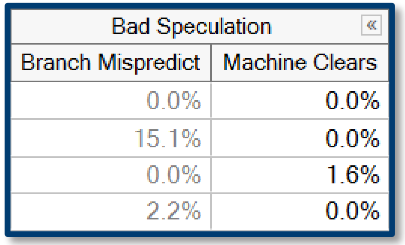
Summary: Top Down Tuning Method

- Make system-level optimizations
- Make algorithmic optimizations
  - Use Threading Advisor to add threading
  - Use *Concurrency Analysis* and *Locks & Waits Analysis* to tune threading
- Make microarchitectural optimizations
  - Find your hotspots
    - Use *Hotspots Analysis* or *Advanced Hotspots Analysis*
  - For each hotspot, determine efficiency.
    - Use *General Exploration Analysis* to identify inefficient hotspots.
    - If inefficient: Categorize the bottleneck, identify the cause, and optimize it!
      - Hierarchical metrics in *General Exploration Analysis* focus your attention where it's needed most and allow you to easily identify the issue.
      - *Memory Access Analysis* can help with Back End Bound code.
      - Vectorization Advisor can help improve the efficiency of Retiring code.





Function / Call Stack	CPU Time ▼ Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)
▶ initialize_2D_buffer	9.423s
▼ grid_intersect	3.209s
▶ intersect_objects	3.069s
▶ grid_intersect	0.141s
▼ sphere_intersect	2.424s



Bad Speculation	
Branch Mispredict	Machine Clears
0.0%	0.0%
15.1%	0.0%
0.0%	1.6%
2.2%	0.0%

Intel® VTune™ Amplifier

Faster, Scalable Code Faster

Get the Data You Need

- Hotspot (Statistical call tree), Call counts (Statistical)
- Thread Profiling – Concurrency and Lock & Waits Analysis
- Cache miss, Bandwidth analysis…[1]
- GPU Offload and OpenCL™ Kernel Tracing

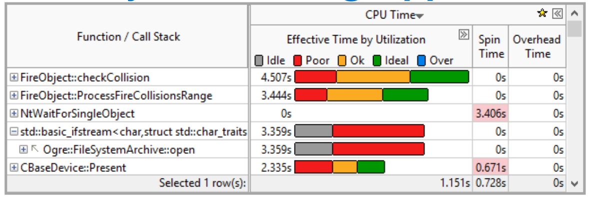
Find Answers Fast

- View Results on the Source / Assembly
- OpenMP Scalability Analysis, Graphical Frame Analysis
- Filter Out Extraneous Data – Organize Data with Viewpoints
- Visualize Thread & Task Activity on the Timeline

Easy to Use

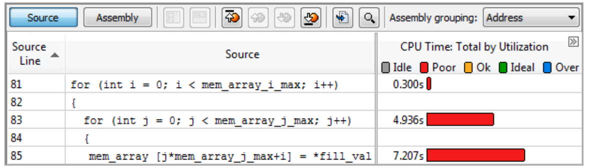
- No Special Compiles – C, C++, C#, Fortran, Java, ASM
- Visual Studio* Integration or Stand Alone
- Local & Remote Data Collection, Command Line
- Analyze Windows* & Linux* data on OS X*[2]

Quickly Find Tuning Opportunities



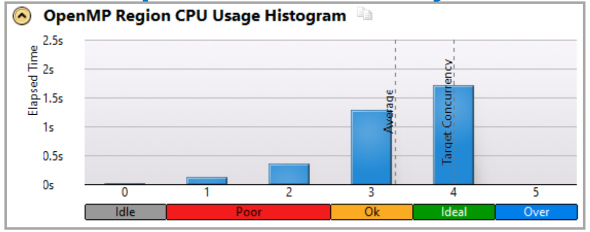
Function / Call Stack	CPU Time: Effective Time by Utilization (🟫 Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over)	Spin Time	Overhead Time
⊞ FireObject::checkCollision	4.507s	0s	0s
⊞ FireObject::ProcessFireCollisionsRange	3.444s	0s	0s
⊞ NtWaitForSingleObject	0s	3.406s	0s
⊟ std::basic_ifstream<char,struct std::char_traits	3.359s	0s	0s
⊞ ↖ Ogre::FileSystemArchive::open	3.359s	0s	0s
⊞ CBaseDevice::Present	2.335s	0.671s	0s
Selected 1 row(s):	1.151s	0.728s	0s

See Results On The Source Code

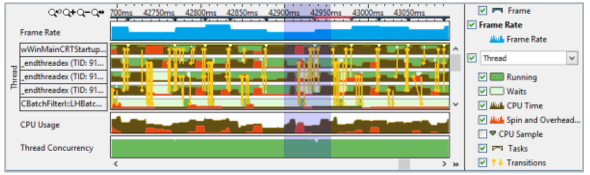


Source Line	Source	CPU Time: Total by Utilization (🟫 Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over)
81	`for (int i = 0; i < mem_array_i_max; i++)`	0.300s
82	`{`	
83	`  for (int j = 0; j < mem_array_j_max; j++)`	4.936s
84	`  {`	
85	`   mem_array [j*mem_array_j_max+i] = *fill_val`	7.207s

Tune OpenMP Scalability



Visualize & Filter Data



[1] Events vary by processor. [2] No data collection on OS X*

Legal Disclaimer & Optimization Notice