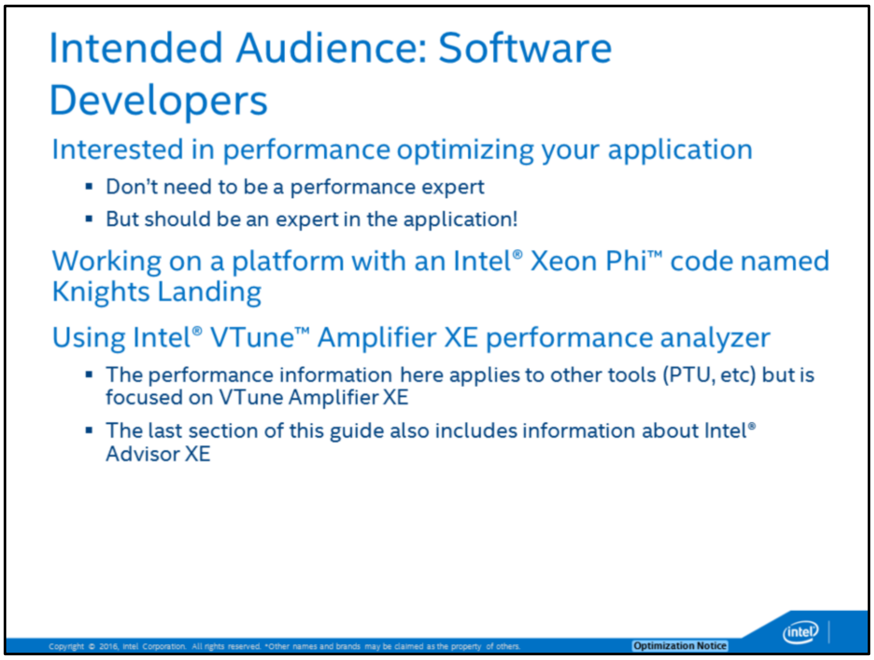


Intended Audience: Software Developers

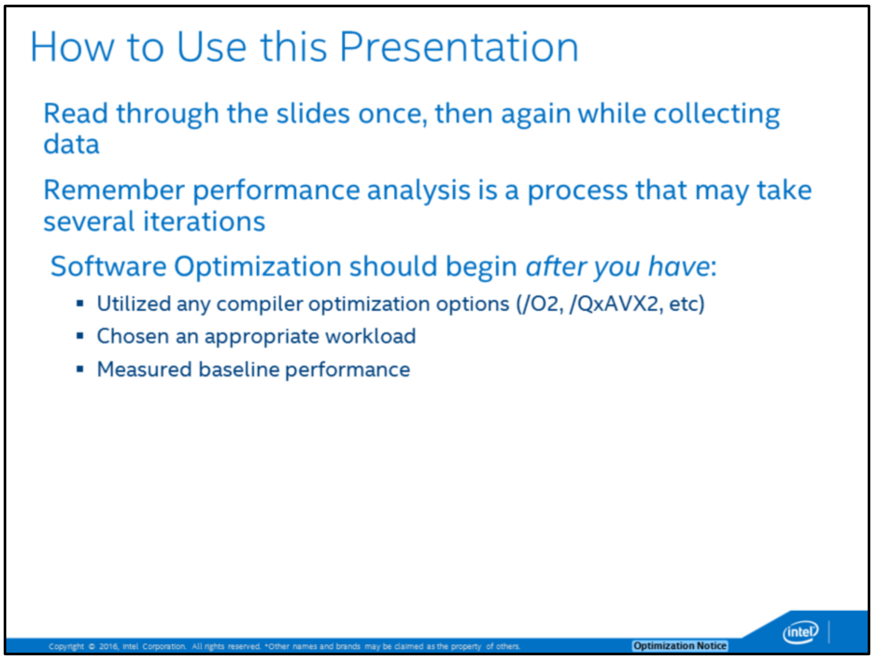
Interested in performance optimizing your application

- Don't need to be a performance expert
- But should be an expert in the application!

Working on a platform with an Intel® Xeon Phi™ code named Knights Landing

Using Intel® VTune™ Amplifier XE performance analyzer

- The performance information here applies to other tools (PTU, etc) but is focused on VTune Amplifier XE
- The last section of this guide also includes information about Intel® Advisor XE



How to Use this Presentation

Read through the slides once, then again while collecting data

Remember performance analysis is a process that may take several iterations

Software Optimization should begin *after you have*:

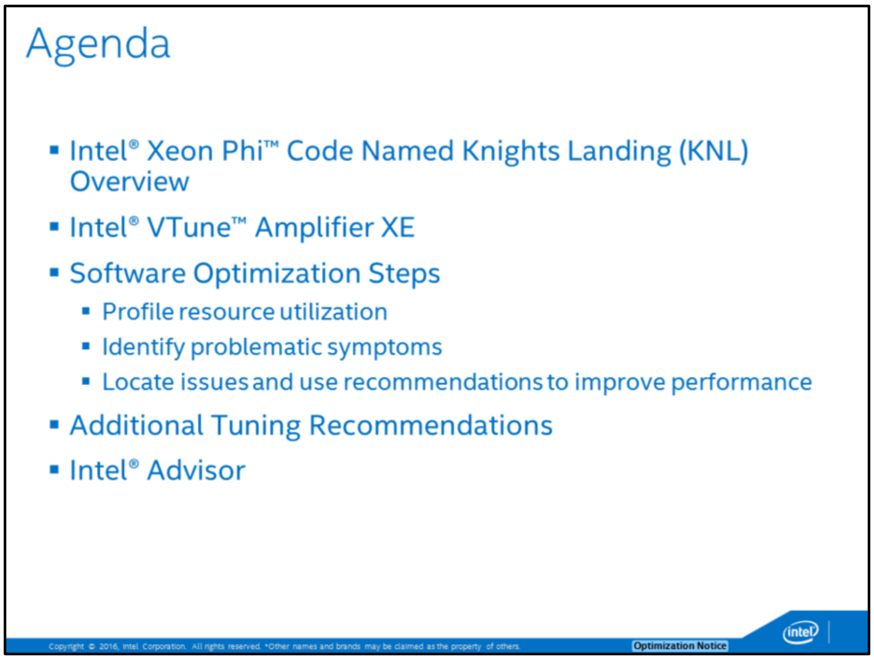
- Utilized any compiler optimization options (/O2, /QxAVX2, etc)
- Chosen an appropriate workload
- Measured baseline performance

Using Intel® VTune™ Amplifier XE to Tune Software on the Intel® Xeon Phi™ code named Knights Landing (KNL)

Software and Services Group

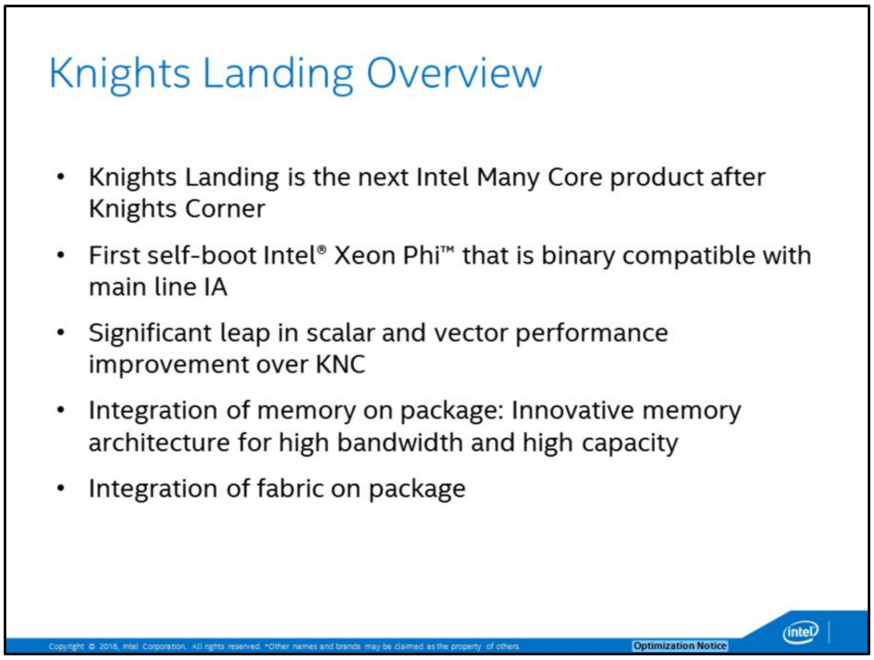
Ver. 1.1

Optimization Notice



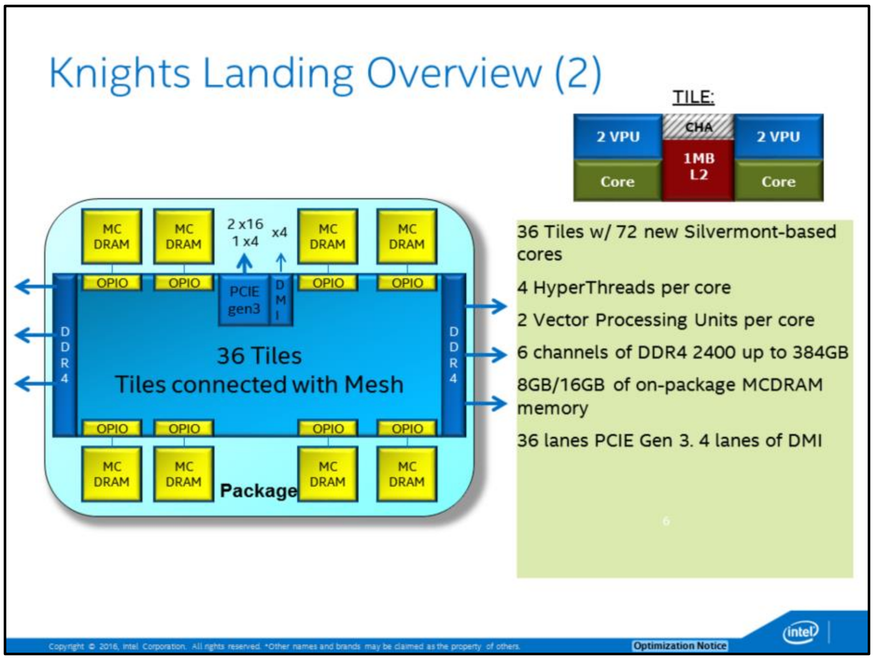
Agenda

- Intel® Xeon Phi™ Code Named Knights Landing (KNL) Overview
- Intel® VTune™ Amplifier XE
- Software Optimization Steps
  - Profile resource utilization
  - Identify problematic symptoms
  - Locate issues and use recommendations to improve performance
- Additional Tuning Recommendations
- Intel® Advisor

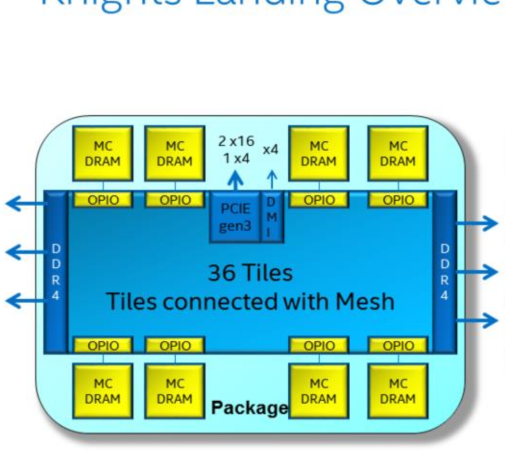


Knights Landing Overview

- Knights Landing is the next Intel Many Core product after Knights Corner
- First self-boot Intel® Xeon Phi™ that is binary compatible with main line IA
- Significant leap in scalar and vector performance improvement over KNC
- Integration of memory on package: Innovative memory architecture for high bandwidth and high capacity
- Integration of fabric on package



Knights Landing Overview (2)



36 Tiles w/ 72 new Silvermont-based cores

4 HyperThreads per core

2 Vector Processing Units per core

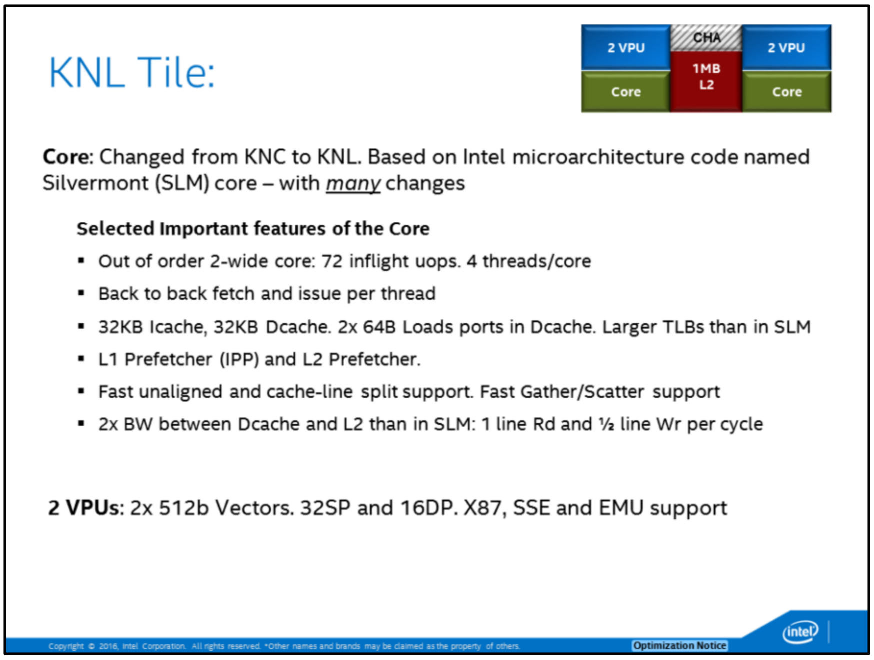
6 channels of DDR4 2400 up to 384GB

8GB/16GB of on-package MCDRAM memory

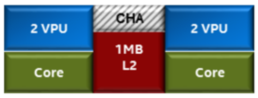
36 lanes PCIE Gen 3. 4 lanes of DMI

Optimization Notice

KNL is a highly-parallel architecture with large vector units. To get the most performance out of this platform, it is imperative to take advantage of these strengths.



KNL Tile:

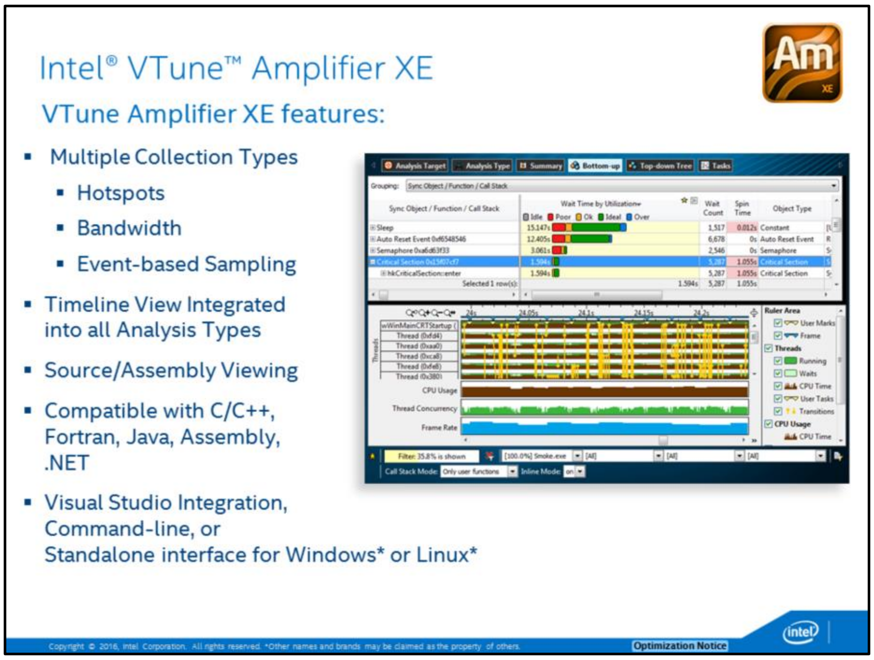


Core: Changed from KNC to KNL. Based on Intel microarchitecture code named Silvermont (SLM) core – with *many* changes

Selected Important features of the Core

- Out of order 2-wide core: 72 inflight uops. 4 threads/core
- Back to back fetch and issue per thread
- 32KB Icache, 32KB Dcache. 2x 64B Loads ports in Dcache. Larger TLBs than in SLM
- L1 Prefetcher (IPP) and L2 Prefetcher.
- Fast unaligned and cache-line split support. Fast Gather/Scatter support
- 2x BW between Dcache and L2 than in SLM: 1 line Rd and ½ line Wr per cycle

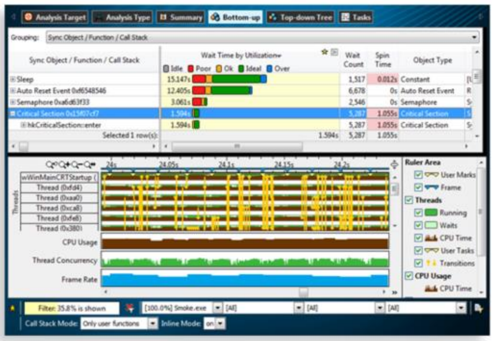
2 VPUs: 2x 512b Vectors. 32SP and 16DP. X87, SSE and EMU support



Intel® VTune™ Amplifier XE

VTune Amplifier XE features:

- Multiple Collection Types
  - Hotspots
  - Bandwidth
  - Event-based Sampling
- Timeline View Integrated into all Analysis Types
- Source/Assembly Viewing
- Compatible with C/C++, Fortran, Java, Assembly, .NET
- Visual Studio Integration, Command-line, or Standalone interface for Windows* or Linux*



Grouping: Sync Object / Function / Call Stack

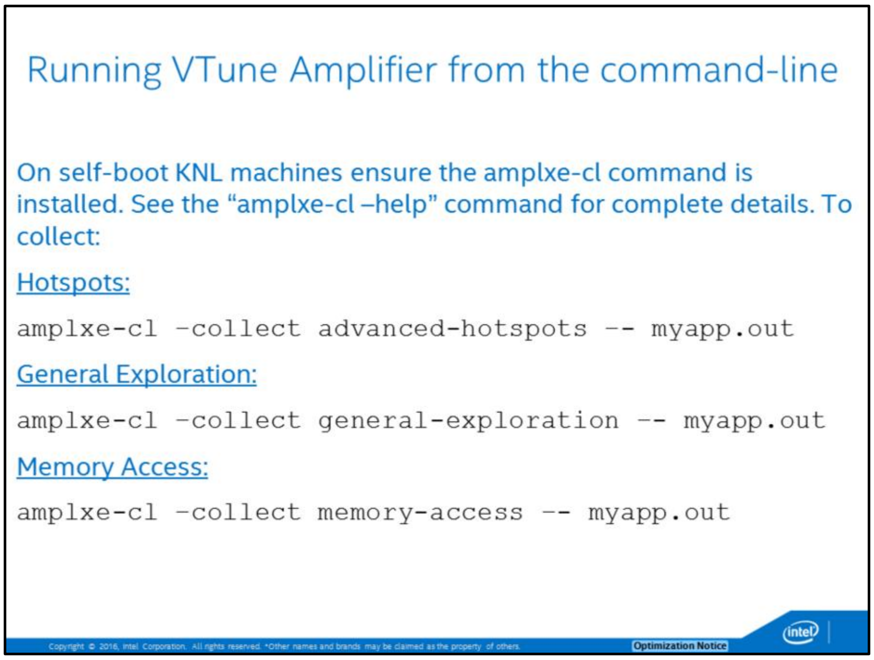
Sync Object / Function / Call Stack	Wait Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Wait Count	Spin Time	Object Type
Sleep	15.147s	1,517	0.012s	Constant
Auto Reset Event 0xf6548546	12.405s	6,678	0s	Auto Reset Event
Semaphore 0xa6d63f33	3.061s	2,546	0s	Semaphore
Critical Section 0x15f07cf7	1.594s	5,287	1.055s	Critical Section
hkCriticalSection::enter	1.594s	5,287	1.055s	Critical Section
Selected 1 row(s):	1.594s	5,287	1.055s	

Filter: 35.8% is shown | [100.0%] Smoke.exe | [All] | [All] | [All]

Call Stack Mode: Only user functions | Inline Mode: on

Optimization Notice

Most screenshots in this presentation were taken from Intel® VTune™ Amplifier XE 2016 Update 4. This is the first public version with KNL support. Screenshots from different versions of the tool may have minor differences.



Running VTune Amplifier from the command-line

On self-boot KNL machines ensure the amplxe-cl command is installed. See the "amplxe-cl –help" command for complete details. To collect:

Hotspots:

```
amplxe-cl -collect advanced-hotspots -- myapp.out
```

General Exploration:

```
amplxe-cl -collect general-exploration -- myapp.out
```

Memory Access:

```
amplxe-cl -collect memory-access -- myapp.out
```

Optimization Notice

Results will be created in a directory named r###ah, r###ge, or r###macc

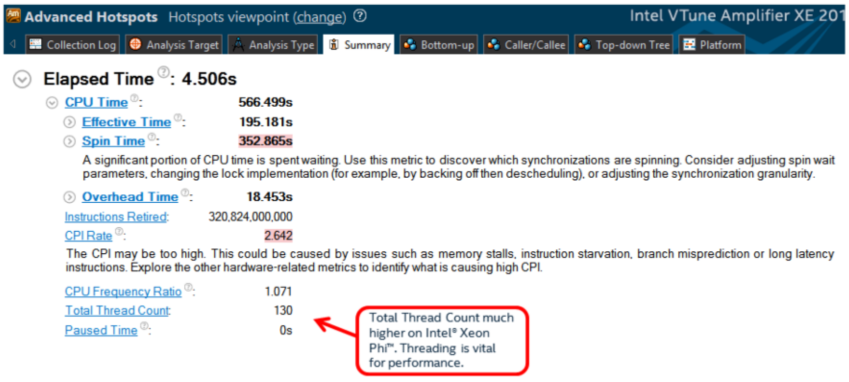
Results can be viewed from the command-line or GUI on the KNL machine, but it is generally more efficient to copy results to another machine with the GUI installed for analysis.

It is also recommended to add the –no-auto-finalize flag to collections that will be creating large results. The finalization step is compute intensive and runs serially which may take a long time on the KNL. Finalization can be done on another machine after copying the results off of the KNL.

The data collected may be very large for longer runs with many threads active. If you find that you are reaching the data limit, use the flag -data-limit=<integer>. The default limit is 500MB. The integer specifies the size in MB. Use 0 for no limit.

Advanced Hotspots Analysis

- Supports OpenMP* analysis
- Stack-sampling is enabled. However, call counts and trip counts are not supported.



Elapsed Time: 4.506s

CPU Time:	566.499s
Effective Time:	195.181s
Spin Time:	352.865s

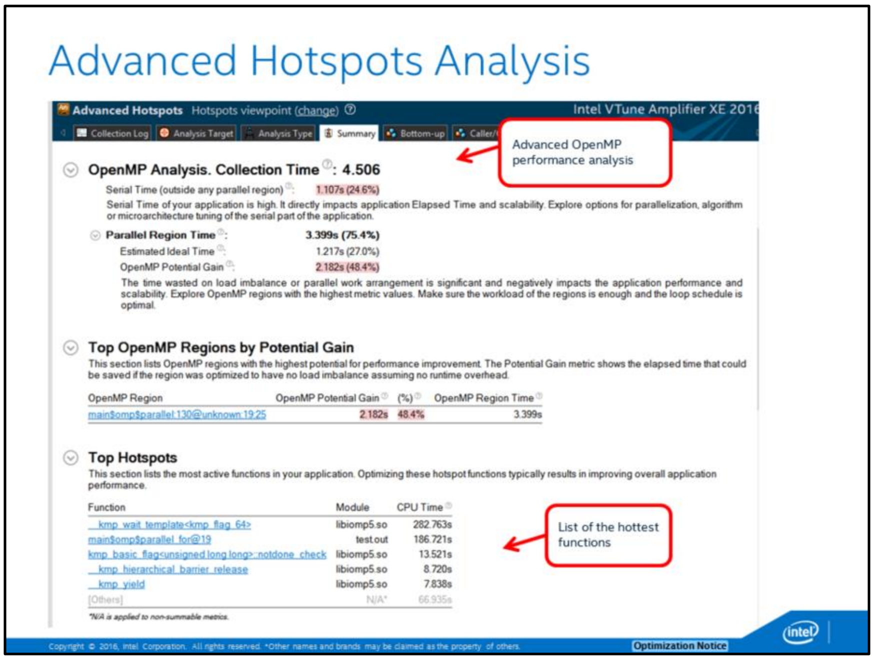
A significant portion of CPU time is spent waiting. Use this metric to discover which synchronizations are spinning. Consider adjusting spin wait parameters, changing the lock implementation (for example, by backing off then descheduling), or adjusting the synchronization granularity.

Overhead Time:	18.453s
Instructions Retired:	320,824,000,000
CPI Rate:	2.642

The CPI may be too high. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or long latency instructions. Explore the other hardware-related metrics to identify what is causing high CPI.

CPU Frequency Ratio:	1.071
Total Thread Count:	130
Paused Time:	0s

Total Thread Count much higher on Intel® Xeon Phi™. Threading is vital for performance.



Advanced Hotspots Analysis

Advanced Hotspots Hotspots viewpoint (change)
Intel VTune Amplifier XE 2016
Collection Log
Analysis Target
Analysis Type
Summary
Bottom-up
Caller/
Advanced OpenMP performance analysis

OpenMP Analysis. Collection Time: 4.506

Serial Time (outside any parallel region):	1.107s (24.6%)

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:	**3.399s (75.4%)**
Estimated Ideal Time:	1.217s (27.0%)
OpenMP Potential Gain:	2.182s (48.4%)

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 was optimized to have no load imbalance assuming no runtime overhead.

OpenMP Region	OpenMP Potential Gain	(%)	OpenMP Region Time
main$omp$parallel:130@unknown:19:25	2.182s	48.4%	3.399s

Top Hotspots

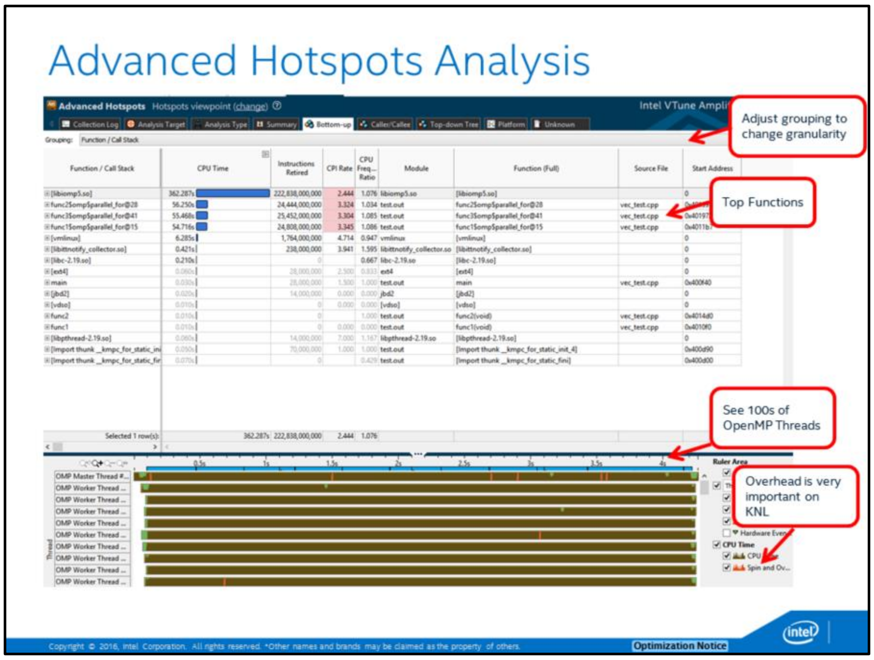
This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
__kmp_wait_template<kmp_flag_64>	libiomp5.so	282.763s
main$omp$parallel_for@19	test.out	186.721s
kmp_basic_flag<unsigned long long>::notdone_check	libiomp5.so	13.521s
__kmp_hierarchical_barrier_release	libiomp5.so	8.720s
__kmp_yield	libiomp5.so	7.838s
[Others]	N/A*	66.935s

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

Optimization Notice

The Advanced Hotspots Analysis will show where your application is spending its time, including information related to OpenMP parallelism. Ensure that the OpenMP runtime library used in the application (e.g. libiomp5.so) is available on the system doing the analysis. This is required to accurately analyze OpenMP overhead.



Advanced Hotspots Analysis

Advanced Hotspots Hotspots viewpoint (change)

Grouping: Function / Call Stack

Function / Call Stack	CPU Time	Instructions Retired	CPI Rate	CPU Freq... Ratio	Module	Function (Full)	Source File	Start Address
[libiomp5.so]	362.287s	222,838,000,000	2.444	1.076	libiomp5.so	[libiomp5.so]		0
func2$omp$parallel_for@28	56.250s	24,444,000,000	3.324	1.034	test.out	func2$omp$parallel_for@28	vec_test.cpp	[illegible]
func3$omp$parallel_for@41	55.468s	25,452,000,000	3.304	1.085	test.out	func3$omp$parallel_for@41	vec_test.cpp	0x40197
func1$omp$parallel_for@15	54.716s	24,808,000,000	3.345	1.086	test.out	func1$omp$parallel_for@15	vec_test.cpp	0x4011b7
[vmlinux]	6.285s	1,764,000,000	4.714	0.947	vmlinux	[vmlinux]		0
[libittnotify_collector.so]	0.421s	238,000,000	3.941	1.595	libittnotify_collector.so	[libittnotify_collector.so]		0
[libc-2.19.so]	0.210s	0		0.667	libc-2.19.so	[libc-2.19.so]		0
[ext4]	0.060s	28,000,000	2.500	0.833	ext4	[ext4]		0
main	0.030s	28,000,000	1.500	1.000	test.out	main	vec_test.cpp	0x400f40
[jbd2]	0.020s	14,000,000	0.000	0.000	jbd2	[jbd2]		0
[vdso]	0.010s	0	0.000	0.000	[vdso]	[vdso]		0
func2	0.010s	0		1.000	test.out	func2(void)	vec_test.cpp	0x4014d0
func1	0.010s	0	0.000	0.000	test.out	func1(void)	vec_test.cpp	0x4010f0
[libpthread-2.19.so]	0.060s	14,000,000	7.000	1.167	libpthread-2.19.so	[libpthread-2.19.so]		0
[Import thunk __kmpc_for_static_ini	0.050s	70,000,000	1.000	1.000	test.out	[Import thunk __kmpc_for_static_init_4]		0x400d90
[Import thunk __kmpc_for_static_fir	0.070s	0		0.429	test.out	[Import thunk __kmpc_for_static_fini]		0x400d00
Selected 1 row(s):	362.287s	222,838,000,000	2.444	1.076				

Adjust grouping to change granularity

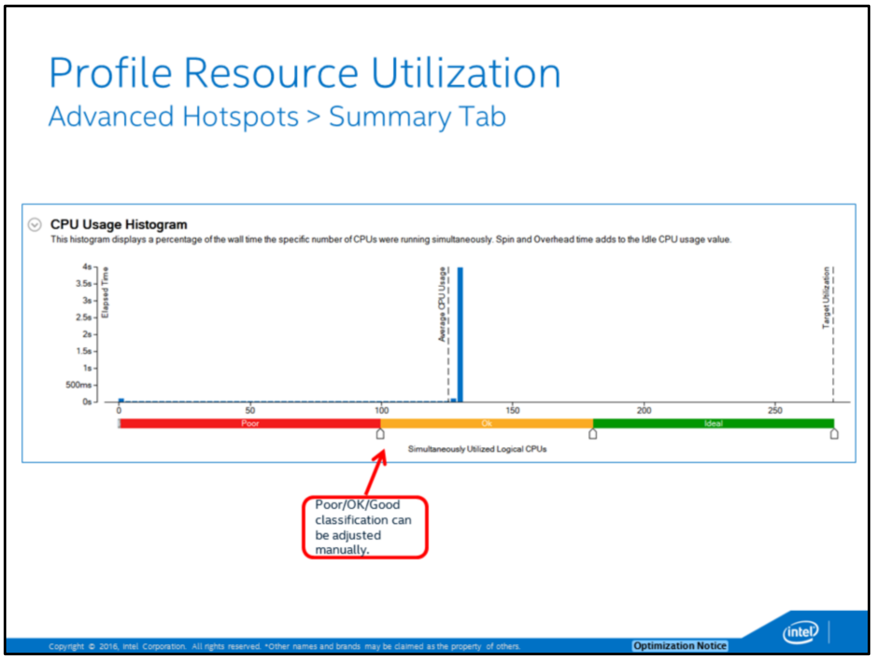
Top Functions

See 100s of OpenMP Threads

Overhead is very important on KNL

Optimization Notice

Use the Bottom-up view to see time spent at various granularities; for example Function or Module granularities. This can be changed in the Grouping drop-down menu. Focus tuning efforts on the hot portions of your application.



Profile Resource Utilization

Advanced Hotspots > Summary Tab

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.

Poor/OK/Good classification can be adjusted manually.

Optimization Notice

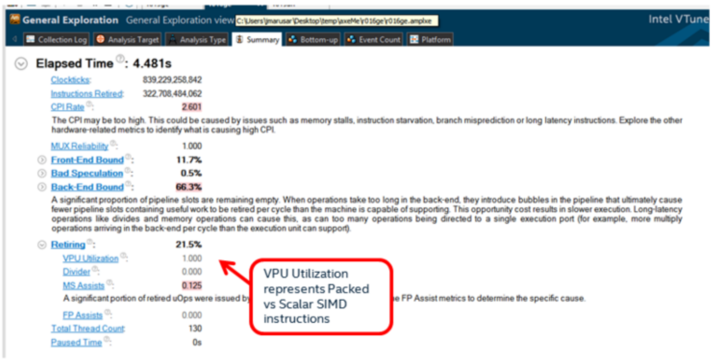
To get the best performance from KNL, it is important to have highly threaded parallel applications. The CPU Usage Histogram in the Summary shows how much time was spent with various numbers of logical cores active. As a general guideline, the vast majority of time should be spent with more than 50% of all available logical cores active. Because each KNL core has 4 HyperThreads, it isn’t always beneficial to have all logical cores active if the bottleneck is the execution core, which is shared between HyperThreads. If memory accesses are the bottlenecks, more threads may alleviate the problem.

Memory Bandwidth may not be helped by more threads, but Memory Latency can.

To identify memory latency as the issue look at L2 misses. If L2 misses are high and bandwidth is high, bandwidth may be the bottleneck. If L2 misses are high, but bandwidth is low, latency may be the issue, and more threads may help.

Vectorization Usage

General Exploration > Summary Tab



General Exploration General Exploration view C:\Users\jmarusar\Desktop\temp\axeMe\r016ge\r016ge.amplxe Intel VTune

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

Elapsed Time: 4.481s

Clockticks:	839,229,258,842
Instructions Retired:	322,708,484,062
CPI Rate:	2.601

The CPI may be too high. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or long latency instructions. Explore the other hardware-related metrics to identify what is causing high CPI.

MUX Reliability:	1.000
Front-End Bound:	11.7%
Bad Speculation:	0.5%
Back-End Bound:	66.3%

A significant proportion of pipeline slots are remaining empty. When operations take too long in the back-end, they introduce bubbles in the pipeline that ultimately cause fewer pipeline slots containing useful work to be retired per cycle than the machine is capable of supporting. This opportunity cost results in slower execution. Long-latency operations like divides and memory operations can cause this, as can too many operations being directed to a single execution port (for example, more multiply operations arriving in the back-end per cycle than the execution unit can support).

Retiring:	21.5%
VPU Utilization:	1.000
Divider:	0.000
MS Assists:	0.125

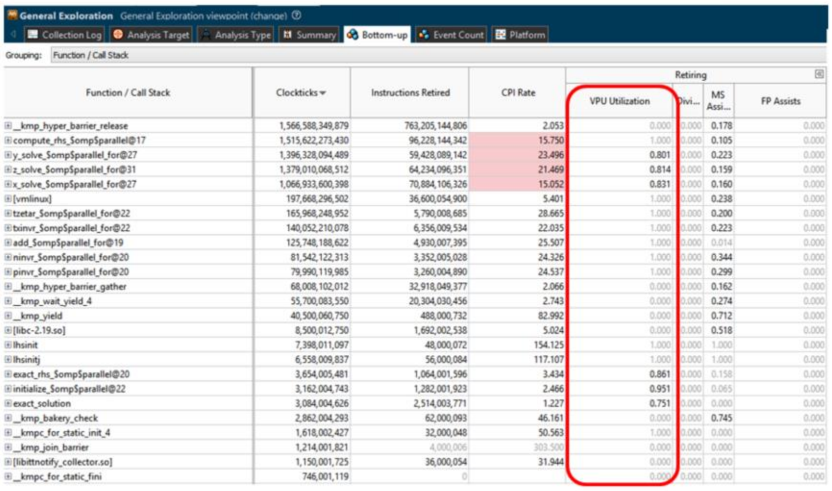
A significant portion of retired uOps were issued by e FP Assist metrics to determine the specific cause.

FP Assists:	0.000
Total Thread Count:	130
Paused Time:	0s

VPU Utilization represents Packed vs Scalar SIMD instructions

KNL supports 512 bit vector instructions. To optimize for KNL, an application should take advantage of these large vector units with heavily vectorized code. Look at the metric VPU Utilization to determine the areas of high and low vectorization in your application.

Vectorization Usage

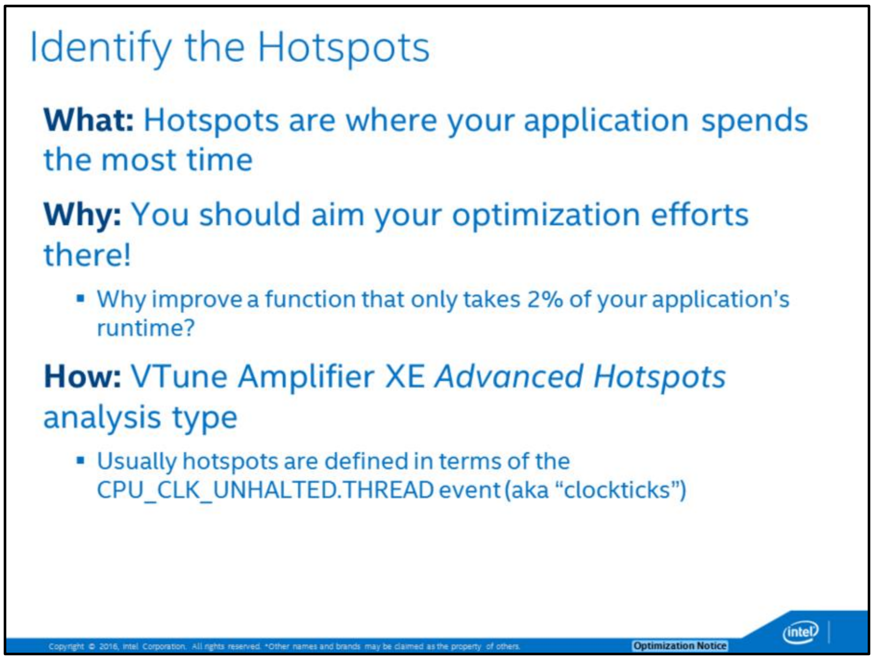


Grouping: Function / Call Stack

Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	Retiring			
				VPU Utilization	Divi...	MS Assi...	FP Assists
__kmp_hyper_barrier_release	1,566,588,349,879	763,205,144,806	2.053	0.000	0.000	0.178	0.000
compute_rhs_ompparallel@17	1,515,622,273,430	96,228,144,342	15.750	1.000	0.000	0.105	0.000
y_solve_ompparallel_for@27	1,396,328,094,489	59,428,089,142	23.496	0.801	0.000	0.223	0.000
z_solve_ompparallel_for@31	1,379,010,068,512	64,234,096,351	21.469	0.814	0.000	0.159	0.000
x_solve_ompparallel_for@27	1,066,933,600,398	70,884,106,326	15.052	0.831	0.000	0.160	0.000
[vmlinux]	197,668,296,502	36,600,054,900	5.401	1.000	0.000	0.238	0.000
tzetar_ompparallel_for@22	165,968,248,952	5,790,008,685	28.665	1.000	0.000	0.200	0.000
txinvr_ompparallel_for@22	140,052,210,078	6,356,009,534	22.035	1.000	0.000	0.223	0.000
add_ompparallel_for@19	125,748,188,622	4,930,007,395	25.507	1.000	0.000	0.014	0.000
ninvr_ompparallel_for@20	81,542,122,313	3,352,005,028	24.326	1.000	0.000	0.344	0.000
pinvr_ompparallel_for@20	79,990,119,985	3,260,004,890	24.537	1.000	0.000	0.299	0.000
__kmp_hyper_barrier_gather	68,008,102,012	32,918,049,377	2.066	0.000	0.000	0.162	0.000
__kmp_wait_yield_4	55,700,083,550	20,304,030,456	2.743	0.000	0.000	0.274	0.000
__kmp_yield	40,500,060,750	488,000,732	82.992	0.000	0.000	0.712	0.000
[libc-2.19.so]	8,500,012,750	1,692,002,538	5.024	0.000	0.000	0.518	0.000
lhsinit	7,398,011,097	48,000,072	154.125	1.000	0.000	1.000	0.000
lhsinitj	6,558,009,837	56,000,084	117.107	1.000	0.000	1.000	0.000
exact_rhs_ompparallel@20	3,654,005,481	1,064,001,596	3.434	0.861	0.000	0.158	0.000
initialize_ompparallel@22	3,162,004,743	1,282,001,923	2.466	0.951	0.000	0.065	0.000
exact_solution	3,084,004,626	2,514,003,771	1.227	0.751	0.000	0.000	0.000
__kmp_bakery_check	2,862,004,293	62,000,093	46.161	0.000	0.000	0.745	0.000
__kmpc_for_static_init_4	1,618,002,427	32,000,048	50.563	1.000	0.000	0.000	0.000
__kmp_join_barrier	1,214,001,821	4,000,006	303.500	0.000	0.000	0.000	0.000
[libittnotify_collector.so]	1,150,001,725	36,000,054	31.944	0.000	0.000	0.000	0.000
__kmpc_for_static_fini	746,001,119	0		0.000	0.000	0.000	0.000

Optimization Notice

The VPU Utilization metric is also available in the Bottom-up view of the General Exploration viewpoint. Locate hotspots with low VPU Utilization and try to improve their usage of the AVX512 capabilities.



Identify the Hotspots

What: Hotspots are where your application spends the most time

Why: You should aim your optimization efforts there!

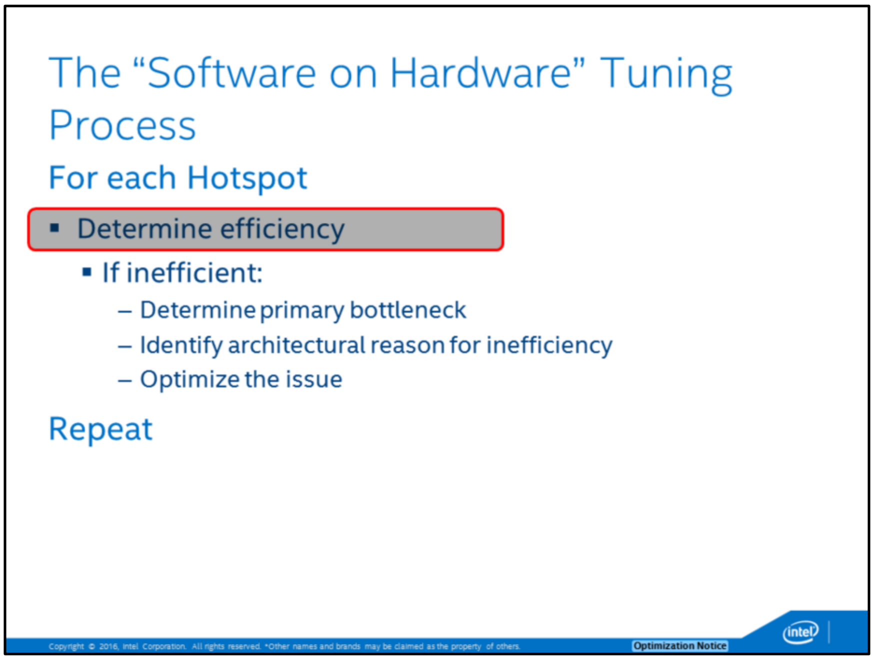
- Why improve a function that only takes 2% of your application's runtime?

How: VTune Amplifier XE *Advanced Hotspots* analysis type

- Usually hotspots are defined in terms of the CPU_CLK_UNHALTED.THREAD event (aka "clockticks")

Optimization Notice

For this processor, the CPU_CLK_UNHALTED.THREAD counter measures unhalted clockticks on a per hardware thread basis. The CPU_CLK_UNHALTED.THREAD counter allows you to see where cycles are being spent on each individual hardware thread.
There is also a CPU_CLK_UNHALTED.REF counter, which counts unhalted clockticks per thread, at the reference frequency for the CPU. In other words, the CPU_CLK_UNHALTED.REF counter should not increase or decrease as a result of frequency changes due to throttling. This counter can be useful for removing the variance introduced due to throttling when comparing multiple analyses.

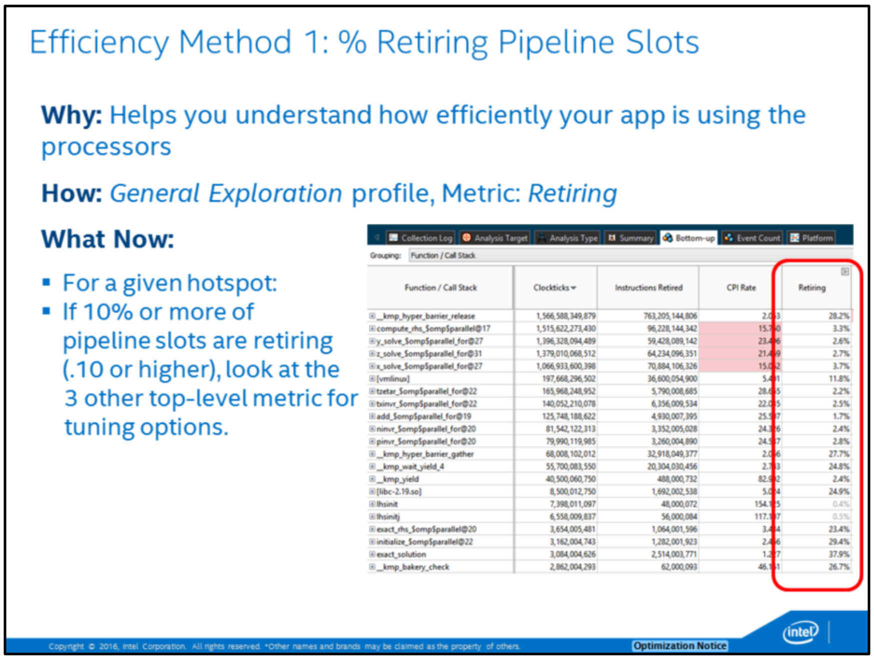


The “Software on Hardware” Tuning Process

For each Hotspot

- Determine efficiency
  - If inefficient:
    - Determine primary bottleneck
    - Identify architectural reason for inefficiency
    - Optimize the issue

Repeat



Efficiency Method 1: % Retiring Pipeline Slots

Why: Helps you understand how efficiently your app is using the processors

How: *General Exploration* profile, Metric: *Retiring*

What Now:

- For a given hotspot:
- If 10% or more of pipeline slots are retiring (.10 or higher), look at the 3 other top-level metric for tuning options.

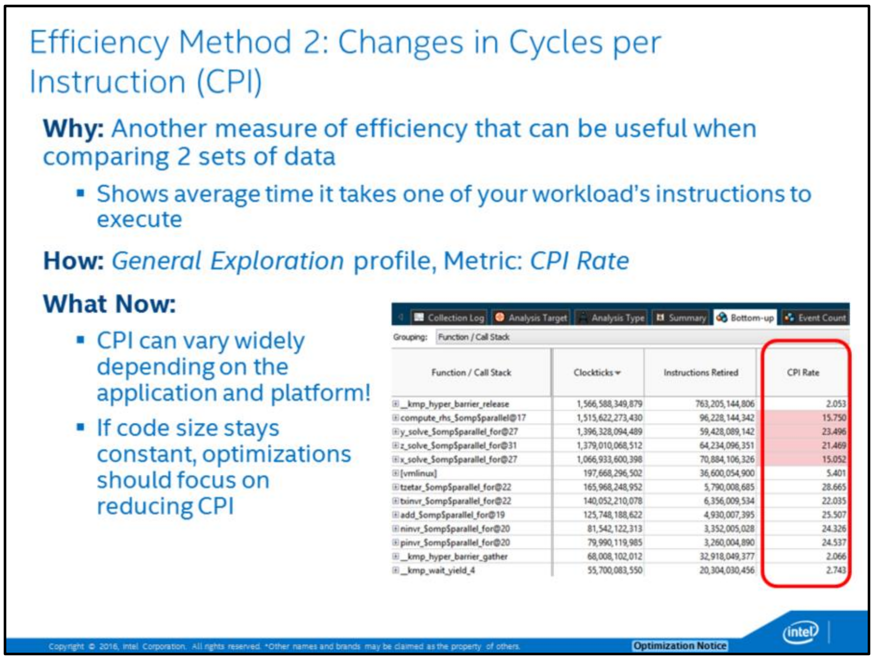
Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	Retiring
__kmp_hyper_barrier_release	1,566,588,349,879	763,205,144,806	2.0[illegible]3	28.2%
compute_rhs_ompparallel@17	1,515,622,273,430	96,228,144,342	15.7[illegible]0	3.3%
y_solve_ompparallel_for@27	1,396,328,094,489	59,428,089,142	23.4[illegible]6	2.6%
z_solve_ompparallel_for@31	1,379,010,068,512	64,234,096,351	21.4[illegible]9	2.7%
x_solve_ompparallel_for@27	1,066,933,600,398	70,884,106,326	15.0[illegible]2	3.7%
[vmlinux]	197,668,296,502	36,600,054,900	5.4[illegible]1	11.8%
tzetar_ompparallel_for@22	165,968,248,952	5,790,008,685	28.6[illegible]5	2.2%
txinvr_ompparallel_for@22	140,052,210,078	6,356,009,534	22.0[illegible]5	2.5%
add_ompparallel_for@19	125,748,188,622	4,930,007,395	25.5[illegible]7	1.7%
ninvr_ompparallel_for@20	81,542,122,313	3,352,005,028	24.3[illegible]6	2.4%
pinvr_ompparallel_for@20	79,990,119,985	3,260,004,890	24.5[illegible]7	2.8%
__kmp_hyper_barrier_gather	68,008,102,012	32,918,049,377	2.0[illegible]6	27.7%
__kmp_wait_yield_4	55,700,083,550	20,304,030,456	2.7[illegible]3	24.8%
__kmp_yield	40,500,060,750	488,000,732	82.9[illegible]2	2.4%
[libc-2.19.so]	8,500,012,750	1,692,002,538	5.0[illegible]4	24.9%
lhsinit	7,398,011,097	48,000,072	154.1[illegible]5	0.4%
lhsinitj	6,558,009,837	56,000,084	117.1[illegible]7	0.5%
exact_rhs_ompparallel@20	3,654,005,481	1,064,001,596	3.4[illegible]4	23.4%
initialize_ompparallel@22	3,162,004,743	1,282,001,923	2.4[illegible]6	29.4%
exact_solution	3,084,004,626	2,514,003,771	1.2[illegible]7	37.9%
__kmp_bakery_check	2,862,004,293	62,000,093	46.1[illegible]1	26.7%

Optimization Notice

Formula:
(UOPS_RETIRED.ALL/ (2*CPU_CLK_UNHALTED.THREAD))

Thresholds: Investigate if -
% Retiring < .10

This metric is based on the fact that when operating at peak performance, the pipeline on this CPU should be able to retire 2 micro-operations per clock cycle (or "clocktick"). The formula looks at "slots" in the pipeline for each core, and sees if the slots are filled, and if so, whether they contained a micro-op that retired.



Efficiency Method 2: Changes in Cycles per Instruction (CPI)

Why: Another measure of efficiency that can be useful when comparing 2 sets of data

- Shows average time it takes one of your workload's instructions to execute

How: *General Exploration* profile, Metric: *CPI Rate*

What Now:

- CPI can vary widely depending on the application and platform!
- If code size stays constant, optimizations should focus on reducing CPI

Grouping: Function / Call Stack

Function / Call Stack	Clockticks ▾	Instructions Retired	CPI Rate
⊞ __kmp_hyper_barrier_release	1,566,588,349,879	763,205,144,806	2.053
⊞ compute_rhs_ompparallel@17	1,515,622,273,430	96,228,144,342	15.750
⊞ y_solve_ompparallel_for@27	1,396,328,094,489	59,428,089,142	23.496
⊞ z_solve_ompparallel_for@31	1,379,010,068,512	64,234,096,351	21.469
⊞ x_solve_ompparallel_for@27	1,066,933,600,398	70,884,106,326	15.052
⊞ [vmlinux]	197,668,296,502	36,600,054,900	5.401
⊞ tzetar_ompparallel_for@22	165,968,248,952	5,790,008,685	28.665
⊞ txinvr_ompparallel_for@22	140,052,210,078	6,356,009,534	22.035
⊞ add_ompparallel_for@19	125,748,188,622	4,930,007,395	25.507
⊞ ninvr_ompparallel_for@20	81,542,122,313	3,352,005,028	24.326
⊞ pinvr_ompparallel_for@20	79,990,119,985	3,260,004,890	24.537
⊞ __kmp_hyper_barrier_gather	68,008,102,012	32,918,049,377	2.066
⊞ __kmp_wait_yield_4	55,700,083,550	20,304,030,456	2.743

Optimization Notice

Formula:
CPU_CLK_UNHALTED.THREAD/INST_RETIRED.ANY

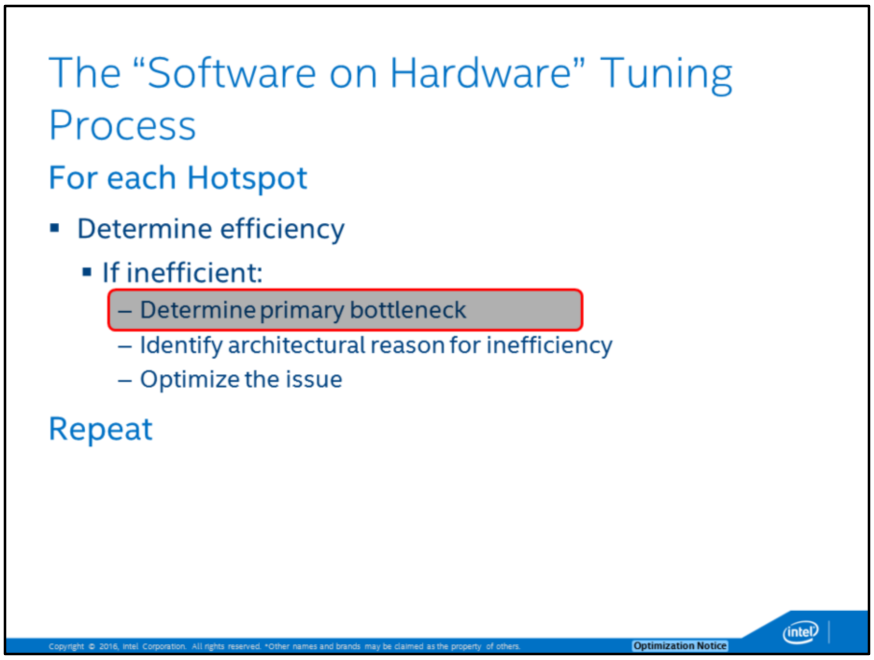
Threshold:
In the interface, CPI will be highlighted if it is greater than 6. This is a very general rule based on the fact that many tuned applications should be able to get below this threshold. However, many applications will naturally have a CPI of over 6 – it is very dependent on workload and platform. It is best used as a comparison factor – know your app's CPI and see if over time it is moving upward (that is bad) or reducing (good!).

Note that CPI is a ratio! Cycles per instruction. So if the code size changes for a binary, CPI will change. In general, if CPI reduces as a result of optimizations, that is good, and if it increases, that is bad. However there are exceptions. Some code can have a very low CPI but still be inefficient because more instructions are executed than are needed.

Additionally, CPI can be affected if using Intel® Hyper-threading. In a serial workload, or a workload with Intel® Hyper-threading disabled the theoretical best CPI on a hardware thread is 0.5 because the core can allocate and retire 2 instructions per cycle. In a workload with Intel® Hyper-threading enabled which utilizes all 4 hardware threads effectively, the ideal CPI per-thread would be 2 instead of 0.5. This is because the hardware threads share allocation and retirement resources on the core.

Note: Optimized code (e.g. with AVX512 instructions) may actually increase the CPI, and increase stall % – but improve the performance. This is because a single vector instruction will generally take more cycles than a single scalar instruction, but it also often performs more work. For example, a vector instruction may take twice as many cycles, but perform the work of four scalar instructions. In that case, the average CPI will increase, but the application will still be running faster.

CPI is just a general efficiency metric – the real measure of efficiency is work taking less time.

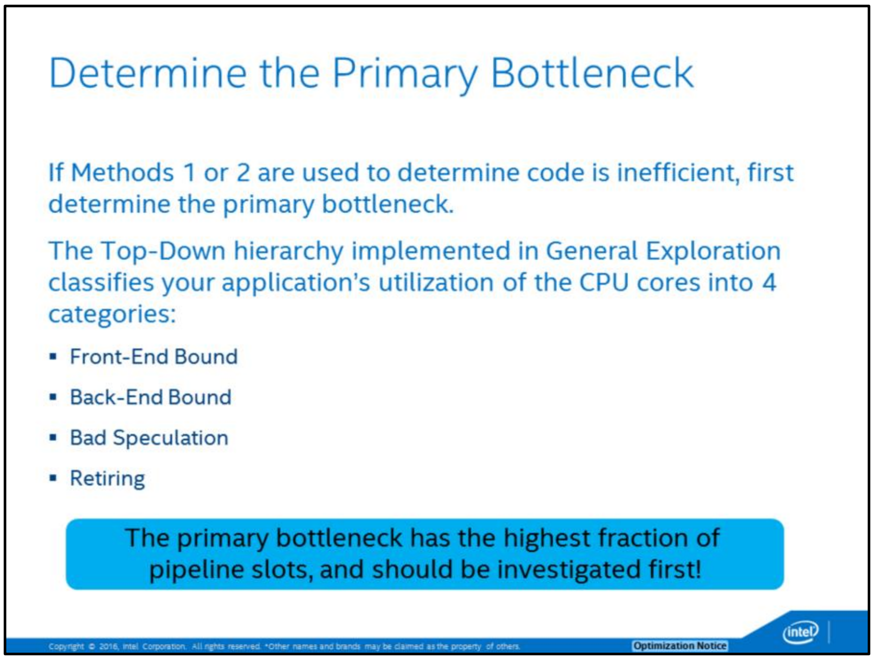


The "Software on Hardware" Tuning Process

For each Hotspot

- Determine efficiency
  - If inefficient:
    - – Determine primary bottleneck
    - – Identify architectural reason for inefficiency
    - – Optimize the issue

Repeat



Determine the Primary Bottleneck

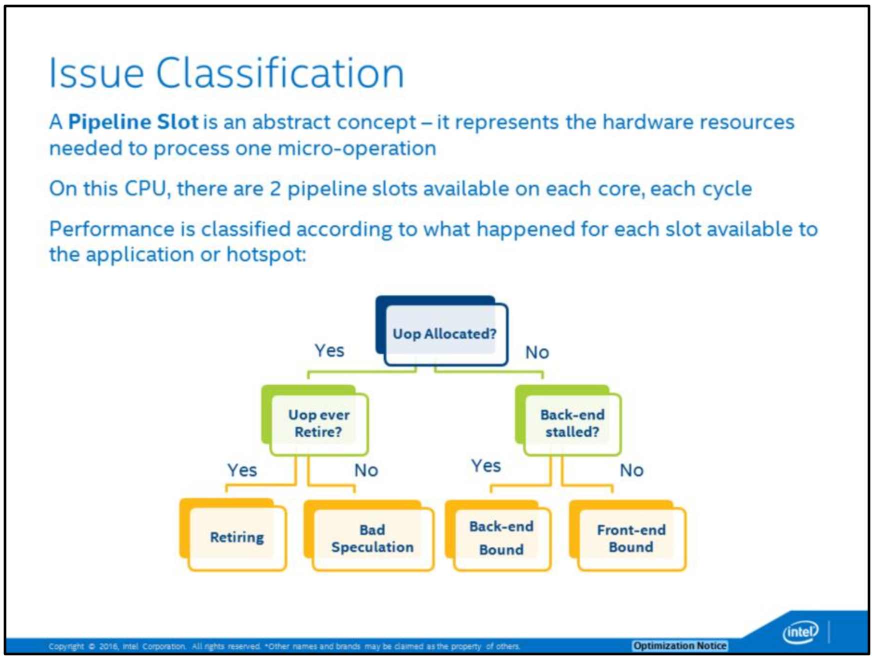
If Methods 1 or 2 are used to determine code is inefficient, first determine the primary bottleneck.

The Top-Down hierarchy implemented in General Exploration classifies your application's utilization of the CPU cores into 4 categories:

- Front-End Bound
- Back-End Bound
- Bad Speculation
- Retiring

The primary bottleneck has the highest fraction of pipeline slots, and should be investigated first!

For a hotspot that is inefficient, determining the primary bottleneck is the first step. Optimizing code to fix issues outside the primary bottleneck category may not boost performance – the biggest boost will come from resolving the biggest bottleneck. Generally, if Retiring is the primary bottleneck, that is good. See next slides.



Issue Classification

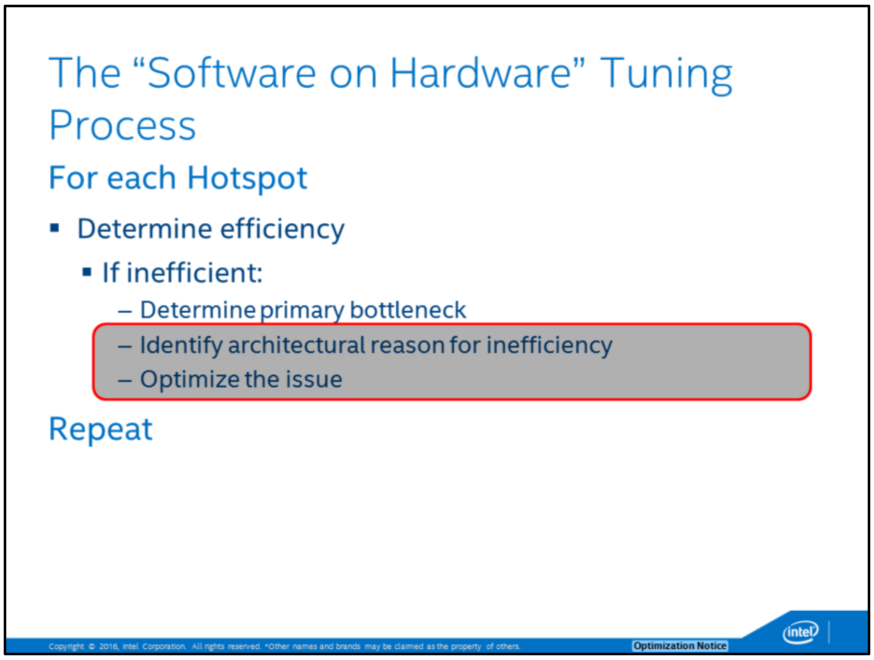
A **Pipeline Slot** is an abstract concept – it represents the hardware resources needed to process one micro-operation

On this CPU, there are 2 pipeline slots available on each core, each cycle

Performance is classified according to what happened for each slot available to the application or hotspot:

Uop Allocated?
Yes
No
Uop ever Retire?
Back-end stalled?
Yes
No
Yes
No
Retiring
Bad Speculation
Back-end Bound
Front-end Bound

Note the way that this methodology allows us to classify what percentage of all pipeline slots end up in each category, for each cycle and for each core. It is possible that for a given dataset, there may be a significant percentage of pipeline slots in multiple categories that merit investigation. Start with the category with the highest percentage of pipeline slots. Ideally a large percentage of slots will fall into the “Retiring” category, but even then, it may be possible to make your code more efficient.



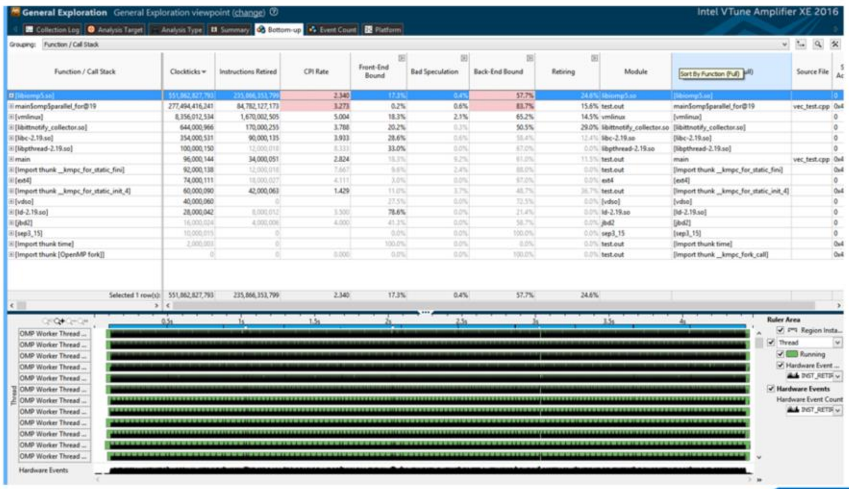
The "Software on Hardware" Tuning Process

For each Hotspot

- Determine efficiency
  - If inefficient:
    - Determine primary bottleneck
    - Identify architectural reason for inefficiency
    - Optimize the issue

Repeat

General Exploration Analysis



General Exploration General Exploration viewpoint (change)

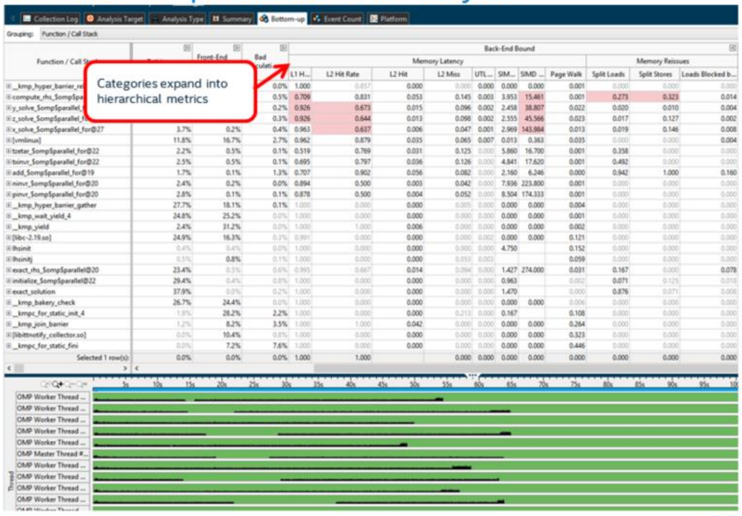
Intel VTune Amplifier XE 2016

Grouping: Function / Call Stack

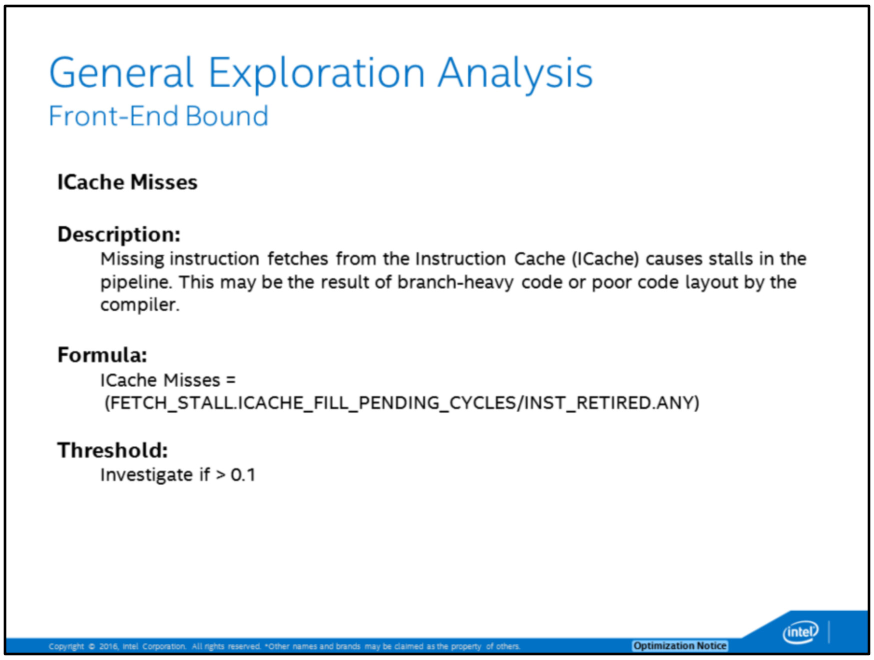
Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound	Retiring	Module	Sort By Function (Full)	Source File	S Ac
[libiomp5.so]	551,862,827,793	235,866,353,799	2.340	17.3%	0.4%	57.7%	24.6%	libiomp5.so	[libiomp5.so]		0
main$omp$parallel_for@19	277,494,416,241	84,782,127,173	3.273	0.2%	0.6%	83.7%	15.6%	test.out	main$omp$parallel_for@19	vec_test.cpp	0x4
[vmlinux]	8,356,012,534	1,670,002,505	5.004	18.3%	2.1%	65.2%	14.5%	vmlinux	[vmlinux]		0
[libittnotify_collector.so]	644,000,966	170,000,255	3.788	20.2%	0.3%	50.5%	29.0%	libittnotify_collector.so	[libittnotify_collector.so]		0
[libc-2.19.so]	354,000,531	90,000,135	3.933	28.6%	0.6%	58.4%	12.4%	libc-2.19.so	[libc-2.19.so]		0
[libpthread-2.19.so]	100,000,150	12,000,018	8.333	33.0%	0.0%	67.0%	0.0%	libpthread-2.19.so	[libpthread-2.19.so]		0
main	96,000,144	34,000,051	2.824	18.3%	9.2%	61.0%	11.5%	test.out	main	vec_test.cpp	0x4
[Import thunk __kmpc_for_static_fini]	92,000,138	12,000,018	7.667	9.6%	2.4%	88.0%	0.0%	test.out	[Import thunk __kmpc_for_static_fini]		0x4
[ext4]	74,000,111	18,000,027	4.111	3.0%	0.0%	97.0%	0.0%	ext4	[ext4]		0
[Import thunk __kmpc_for_static_init_4]	60,000,090	42,000,063	1.429	11.0%	3.7%	48.7%	36.7%	test.out	[Import thunk __kmpc_for_static_init_4]		0x4
[vdso]	40,000,060	0		27.5%	0.0%	72.5%	0.0%	[vdso]	[vdso]		0
[ld-2.19.so]	28,000,042	8,000,012	3.500	78.6%	0.0%	21.4%	0.0%	ld-2.19.so	[ld-2.19.so]		0
[jbd2]	16,000,024	4,000,006	4.000	41.3%	0.0%	58.7%	0.0%	jbd2	[jbd2]		0
[sep3_15]	10,000,015	0		0.0%	0.0%	100.0%	0.0%	sep3_15	[sep3_15]		0
[Import thunk time]	2,000,003	0		100.0%	0.0%	0.0%	0.0%	test.out	[Import thunk time]		0x4
[Import thunk [OpenMP fork]]	0	0	0.000	0.0%	0.0%	100.0%	0.0%	test.out	[Import thunk __kmpc_fork_call]		0x4
Selected 1 row(s):	551,862,827,793	235,866,353,799	2.340	17.3%	0.4%	57.7%	24.6%				

Optimization Notice

General Exploration Analysis



Function / Call St		Front-End	Bad culati	Back-End Bound										
				Memory Latency								Memory Reissues		
				L1 H...	L2 Hit Rate	L2 Hit	L2 Miss	UTL...	SIM...	SIMD ...	Page Walk	Split Loads	Split Stores	Loads Blocked b...
__kmp_hyper_barrier_re			0.0%	1.000	0.857	0.000	0.000	0.000	0.000	0.000	0.001	0.000	0.000	0.000
compute_rhs_omppa			0.5%	0.709	0.831	0.053	0.145	0.003	3.953	15.461	0.001	0.273	0.323	0.014
y_solve_ompparallel_f			0.2%	0.926	0.673	0.015	0.096	0.002	2.458	38.807	0.022	0.020	0.010	0.004
z_solve_ompparallel_fo			0.3%	0.926	0.644	0.013	0.098	0.002	2.555	45.566	0.023	0.017	0.127	0.002
x_solve_ompparallel_for@27	3.7%	0.2%	0.4%	0.963	0.637	0.006	0.047	0.001	2.969	143.984	0.013	0.019	0.146	0.008
[vmlinux]	11.8%	16.7%	2.7%	0.962	0.879	0.035	0.065	0.007	0.013	0.363	0.035	0.000	0.000	0.004
tzetar_ompparallel_for@22	2.2%	0.5%	0.1%	0.519	0.769	0.031	0.125	0.000	5.860	16.700	0.001	0.358	0.000	0.000
txinvr_ompparallel_for@22	2.5%	0.5%	0.1%	0.695	0.797	0.036	0.126	0.000	4.841	17.620	0.001	0.492	0.000	0.000
add_ompparallel_for@19	1.7%	0.1%	1.3%	0.707	0.902	0.056	0.082	0.000	2.160	6.246	0.000	0.942	1.000	0.160
ninvr_ompparallel_for@20	2.4%	0.2%	0.0%	0.894	0.500	0.003	0.042	0.000	7.936	223.800	0.001	0.000	0.000	0.000
pinvr_ompparallel_for@20	2.8%	0.1%	0.1%	0.878	0.500	0.004	0.052	0.000	8.504	174.333	0.001	0.000	0.000	0.000
__kmp_hyper_barrier_gather	27.7%	18.1%	0.1%	1.000	0.000	0.000	0.005	0.000	0.000	0.000	0.004	0.000	0.000	0.000
__kmp_wait_yield_4	24.8%	25.2%	0.0%	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.000	0.000	0.000
__kmp_yield	2.4%	31.2%	0.0%	1.000	1.000	0.006	0.000	0.000	0.000	0.000	0.002	0.000	0.000	0.000
[libc-2.19.so]	24.9%	16.3%	0.3%	0.991	0.000	0.000	0.000	0.002	0.000	0.000	0.121	0.000	0.000	0.000
lhsinit	0.4%	0.4%	0.0%	1.000	0.000	0.000	0.000	0.000	4.750		0.152	0.000	0.000	0.000
lhsinitj	0.5%	0.8%	0.1%	1.000	0.000	0.000	0.053	0.003			0.059	0.000	0.000	0.000
exact_rhs_ompparallel@20	23.4%	0.5%	0.6%	0.995	0.667	0.014	0.094	0.000	1.427	274.000	0.031	0.167	0.000	0.078
initialize_ompparallel@22	29.4%	0.4%	0.8%	1.000	0.000	0.000	0.000	0.000	0.963		0.002	0.071	0.125	0.018
exact_solution	37.9%	0.0%	0.2%	1.000	0.000	0.000	0.000	0.000	1.470		0.000	0.876	0.071	0.008
__kmp_bakery_check	26.7%	24.4%	0.0%	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.006	0.000	0.000	0.000
__kmpc_for_static_init_4	1.9%	28.2%	2.2%	1.000	0.000	0.000	0.213	0.000	0.167		0.108	0.000	0.000	0.000
__kmp_join_barrier	1.2%	8.2%	3.5%	1.000	1.000	0.042	0.000	0.000	0.000	0.000	0.264	0.000	0.000	0.000
[libittnotify_collector.so]	0.0%	10.4%	0.8%	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.323	0.000	0.000	0.000
__kmpc_for_static_fini	0.0%	7.2%	7.6%	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.446	0.000	0.000	0.000
Selected 1 row(s):	0.0%	0.0%	0.0%	1.000	1.000		0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000



General Exploration Analysis

Front-End Bound

ICache Misses

Description:

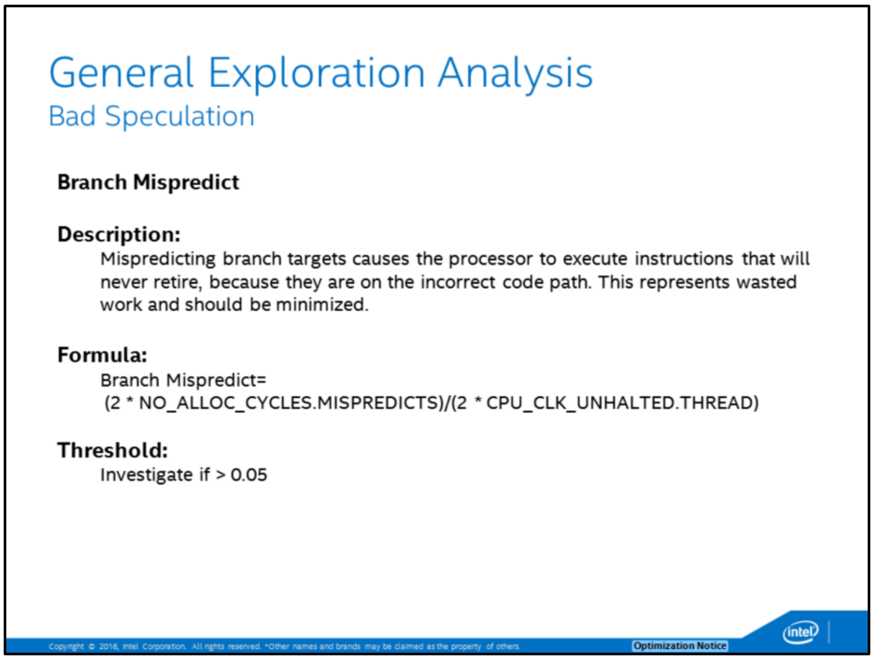
Missing instruction fetches from the Instruction Cache (ICache) causes stalls in the pipeline. This may be the result of branch-heavy code or poor code layout by the compiler.

Formula:

ICache Misses =
(FETCH_STALL.ICACHE_FILL_PENDING_CYCLES/INST_RETIRED.ANY)

Threshold:

Investigate if > 0.1



General Exploration Analysis
Bad Speculation

Branch Mispredict

Description:

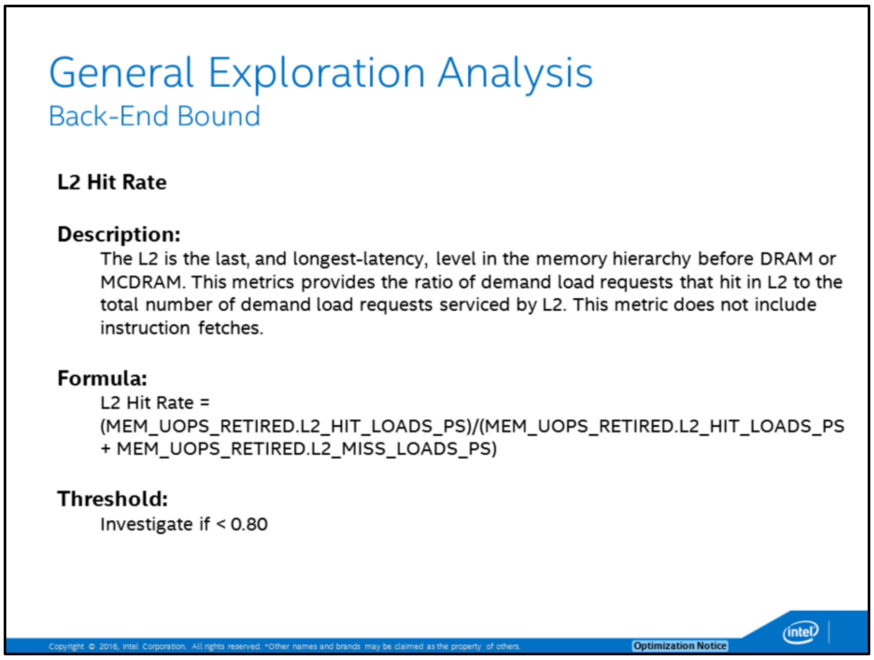
Mispredicting branch targets causes the processor to execute instructions that will never retire, because they are on the incorrect code path. This represents wasted work and should be minimized.

Formula:

Branch Mispredict=
(2 * NO_ALLOC_CYCLES.MISPREDICTS)/(2 * CPU_CLK_UNHALTED.THREAD)

Threshold:

Investigate if > 0.05



General Exploration Analysis

Back-End Bound

L2 Hit Rate

Description:

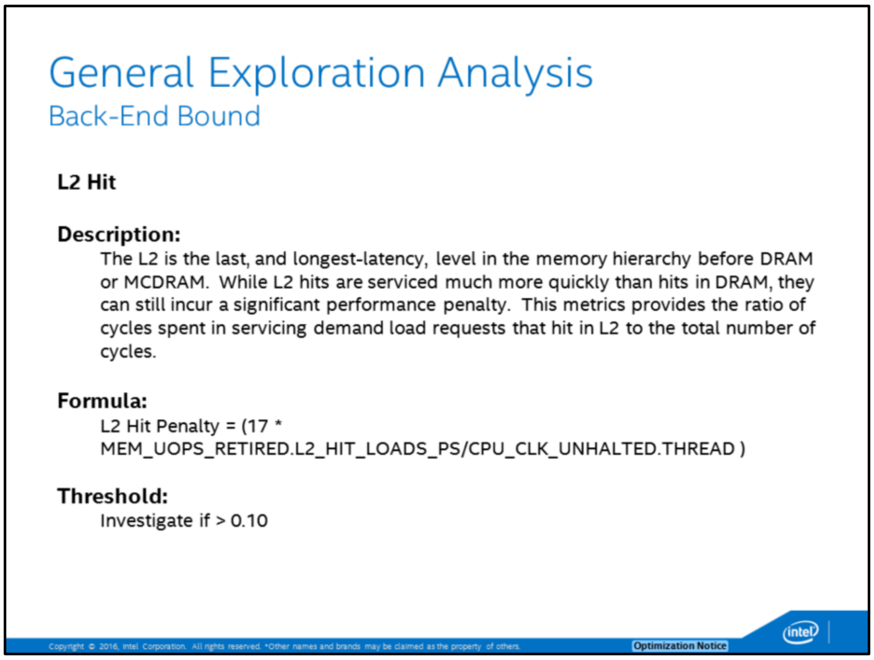
The L2 is the last, and longest-latency, level in the memory hierarchy before DRAM or MCDRAM. This metrics provides the ratio of demand load requests that hit in L2 to the total number of demand load requests serviced by L2. This metric does not include instruction fetches.

Formula:

L2 Hit Rate =
(MEM_UOPS_RETIRED.L2_HIT_LOADS_PS)/(MEM_UOPS_RETIRED.L2_HIT_LOADS_PS + MEM_UOPS_RETIRED.L2_MISS_LOADS_PS)

Threshold:

Investigate if < 0.80



General Exploration Analysis

Back-End Bound

L2 Hit

Description:

The L2 is the last, and longest-latency, level in the memory hierarchy before DRAM or MCDRAM. While L2 hits are serviced much more quickly than hits in DRAM, they can still incur a significant performance penalty. This metrics provides the ratio of cycles spent in servicing demand load requests that hit in L2 to the total number of cycles.

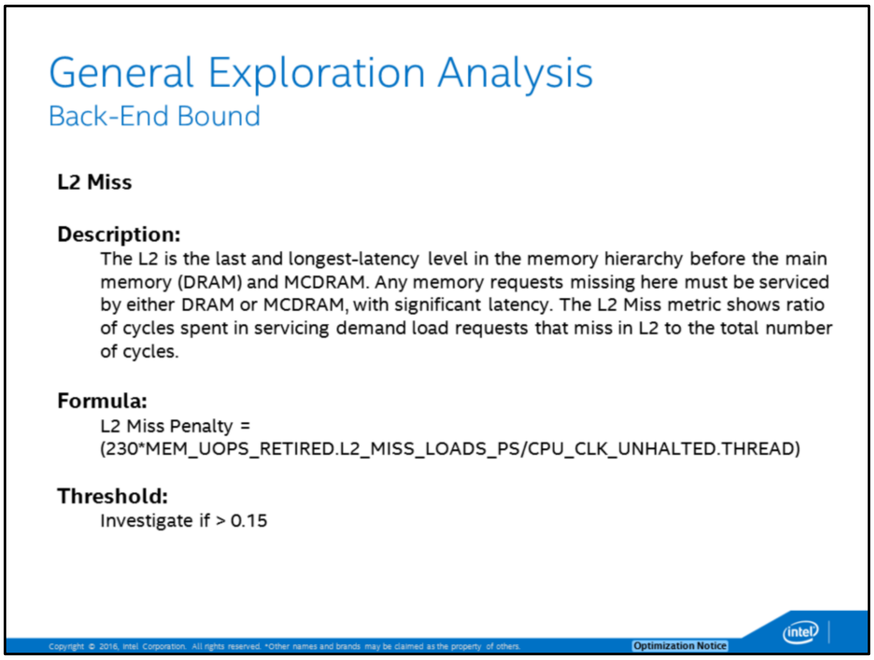
Formula:

L2 Hit Penalty = (17 * MEM_UOPS_RETIRED.L2_HIT_LOADS_PS/CPU_CLK_UNHALTED.THREAD)

Threshold:

Investigate if > 0.10

Optimization Notice



General Exploration Analysis

Back-End Bound

L2 Miss

Description:

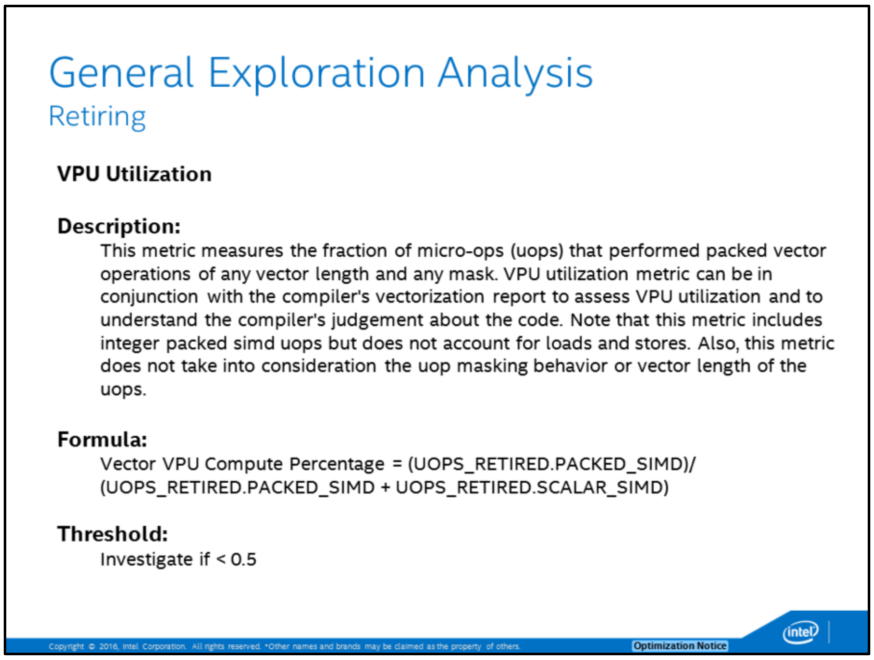
The L2 is the last and longest-latency level in the memory hierarchy before the main memory (DRAM) and MCDRAM. Any memory requests missing here must be serviced by either DRAM or MCDRAM, with significant latency. The L2 Miss metric shows ratio of cycles spent in servicing demand load requests that miss in L2 to the total number of cycles.

Formula:

L2 Miss Penalty =
(230*MEM_UOPS_RETIRED.L2_MISS_LOADS_PS/CPU_CLK_UNHALTED.THREAD)

Threshold:

Investigate if > 0.15



General Exploration Analysis

Retiring

VPU Utilization

Description:

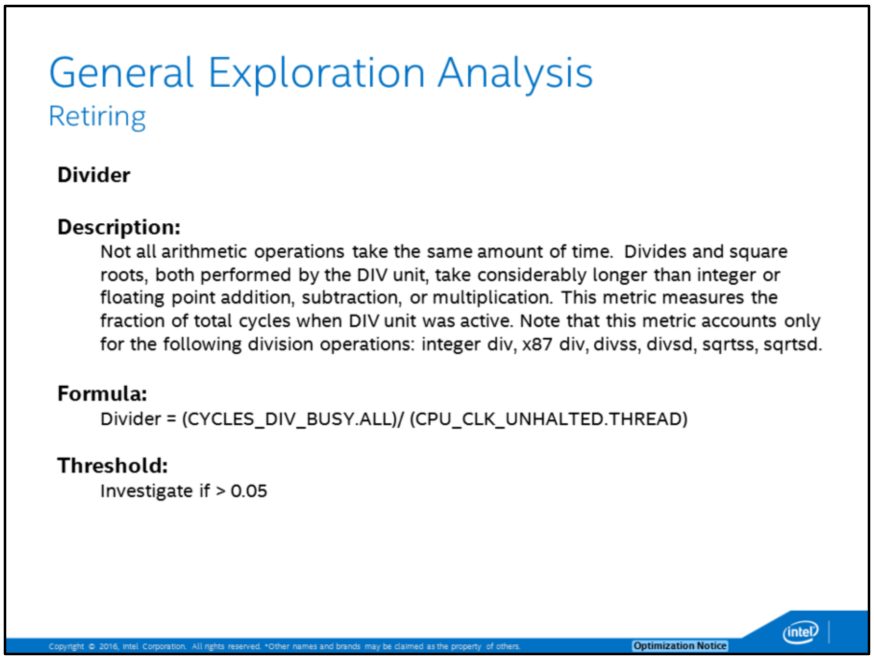
This metric measures the fraction of micro-ops (uops) that performed packed vector operations of any vector length and any mask. VPU utilization metric can be in conjunction with the compiler's vectorization report to assess VPU utilization and to understand the compiler's judgement about the code. Note that this metric includes integer packed simd uops but does not account for loads and stores. Also, this metric does not take into consideration the uop masking behavior or vector length of the uops.

Formula:

Vector VPU Compute Percentage = (UOPS_RETIRED.PACKED_SIMD)/(UOPS_RETIRED.PACKED_SIMD + UOPS_RETIRED.SCALAR_SIMD)

Threshold:

Investigate if < 0.5



General Exploration Analysis

Retiring

Divider

Description:

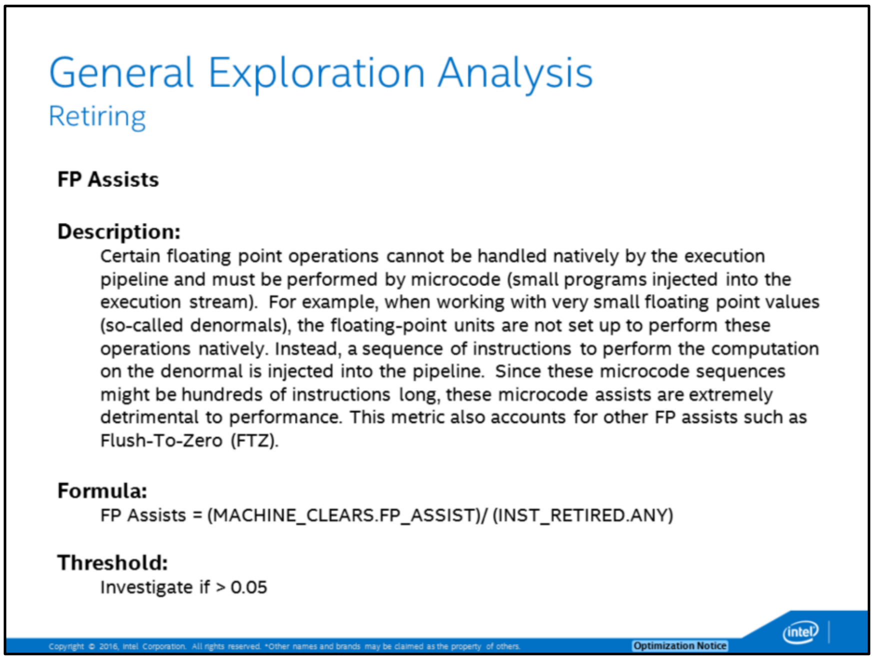
Not all arithmetic operations take the same amount of time. Divides and square roots, both performed by the DIV unit, take considerably longer than integer or floating point addition, subtraction, or multiplication. This metric measures the fraction of total cycles when DIV unit was active. Note that this metric accounts only for the following division operations: integer div, x87 div, divss, divsd, sqrtss, sqrtsd.

Formula:

Divider = (CYCLES_DIV_BUSY.ALL)/ (CPU_CLK_UNHALTED.THREAD)

Threshold:

Investigate if > 0.05



General Exploration Analysis

Retiring

FP Assists

Description:

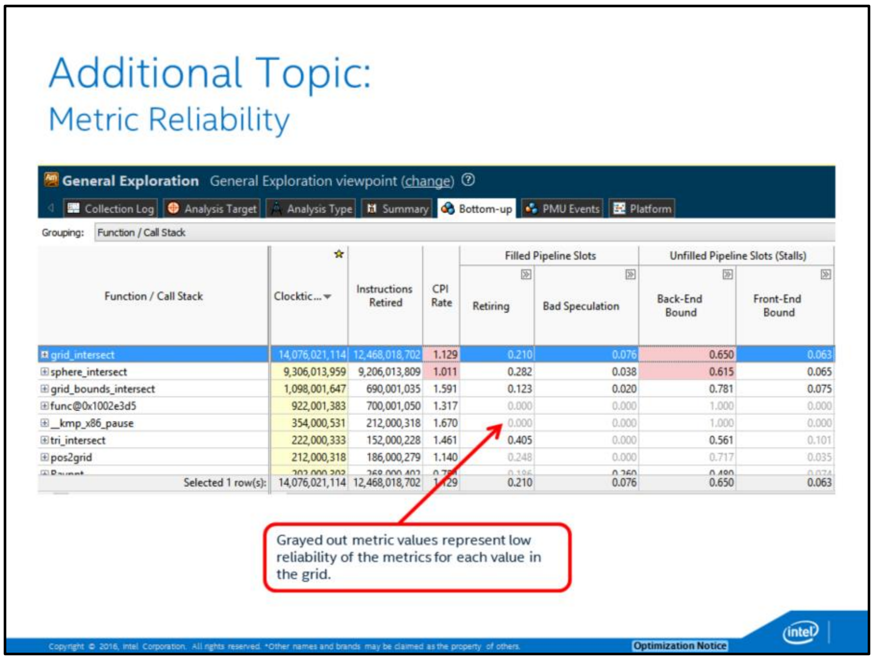
Certain floating point operations cannot be handled natively by the execution pipeline and must be performed by microcode (small programs injected into the execution stream). For example, when working with very small floating point values (so-called denormals), the floating-point units are not set up to perform these operations natively. Instead, a sequence of instructions to perform the computation on the denormal is injected into the pipeline. Since these microcode sequences might be hundreds of instructions long, these microcode assists are extremely detrimental to performance. This metric also accounts for other FP assists such as Flush-To-Zero (FTZ).

Formula:

FP Assists = (MACHINE_CLEARS.FP_ASSIST)/ (INST_RETIRED.ANY)

Threshold:

Investigate if > 0.05



Additional Topic:
Metric Reliability

General Exploration General Exploration viewpoint (change)

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

Grouping: Function / Call Stack

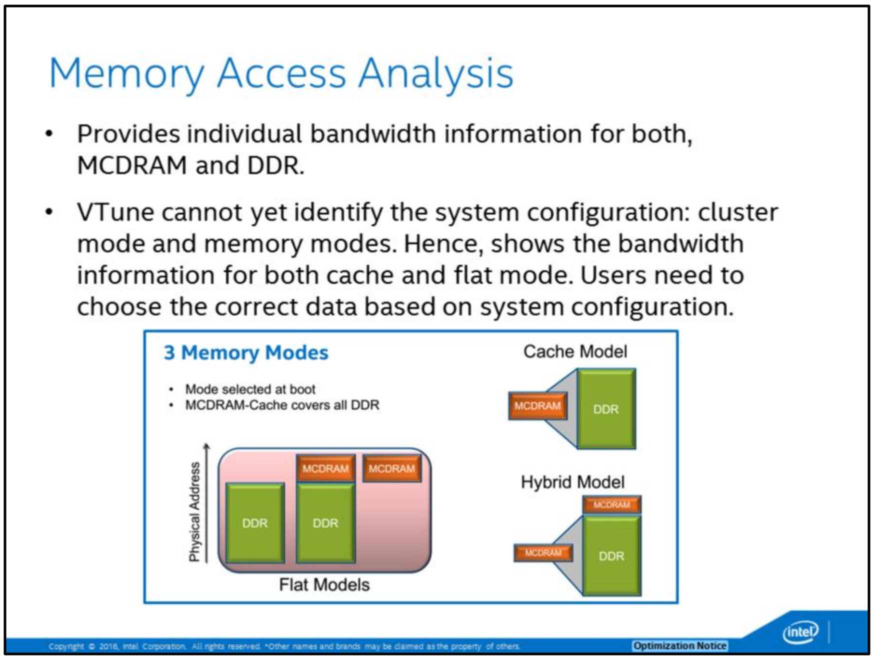
Function / Call Stack	Clocktic...	Instructions Retired	CPI Rate	Filled Pipeline Slots		Unfilled Pipeline Slots (Stalls)	
				Retiring	Bad Speculation	Back-End Bound	Front-End Bound
grid_intersect	14,076,021,114	12,468,018,702	1.129	0.210	0.076	0.650	0.063
sphere_intersect	9,306,013,959	9,206,013,809	1.011	0.282	0.038	0.615	0.065
grid_bounds_intersect	1,098,001,647	690,001,035	1.591	0.123	0.020	0.781	0.075
func@0x1002e3d5	922,001,383	700,001,050	1.317	0.000	0.000	1.000	0.000
__kmp_x86_pause	354,000,531	212,000,318	1.670	0.000	0.000	1.000	0.000
tri_intersect	222,000,333	152,000,228	1.461	0.405	0.000	0.561	0.101
pos2grid	212,000,318	186,000,279	1.140	0.248	0.000	0.717	0.035
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0.260	0.480	[illegible]
Selected 1 row(s):	14,076,021,114	12,468,018,702	1.129	0.210	0.076	0.650	0.063

Grayed out metric values represent low reliability of the metrics for each value in the grid.

Optimization Notice

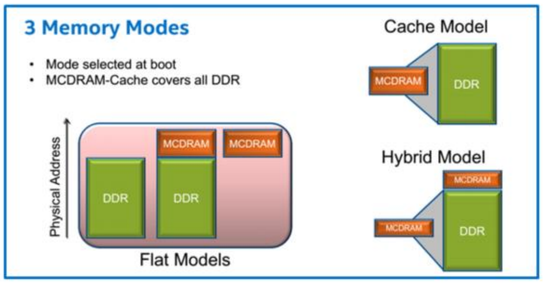
The General Exploration analysis type multiplexes hardware events during collection, which can result in imprecise results if too few samples are collected. The GUI will gray out metrics if the reliability is low based on the number of samples collected. If a metric is grayed out for your area of interest, consider increasing the runtime of the analysis or allowing multiple runs via the project properties.

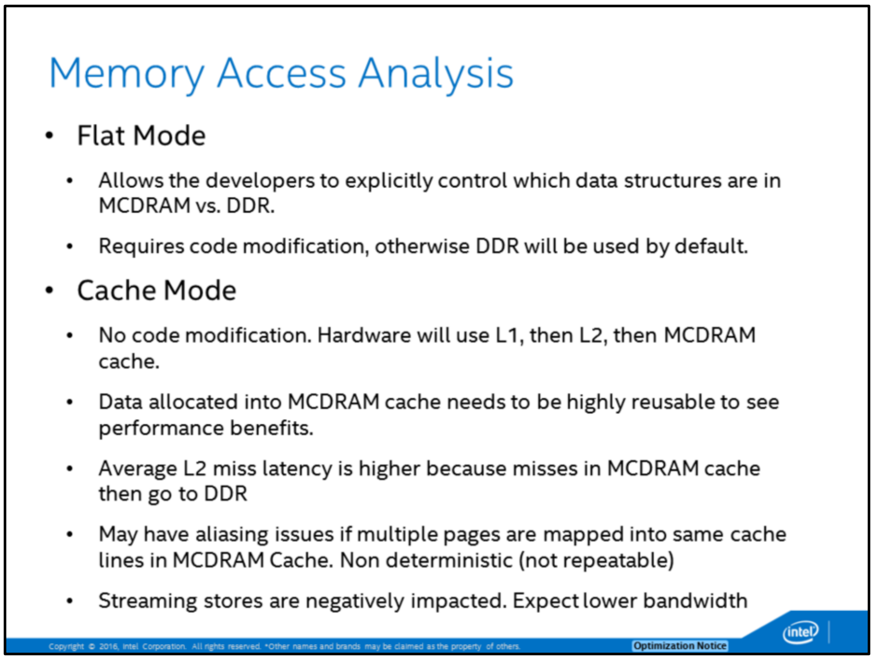
Previous versions of the tool used a MUX Reliability metric for each row, however this was unable to distinguish between different metrics on the same row.



Memory Access Analysis

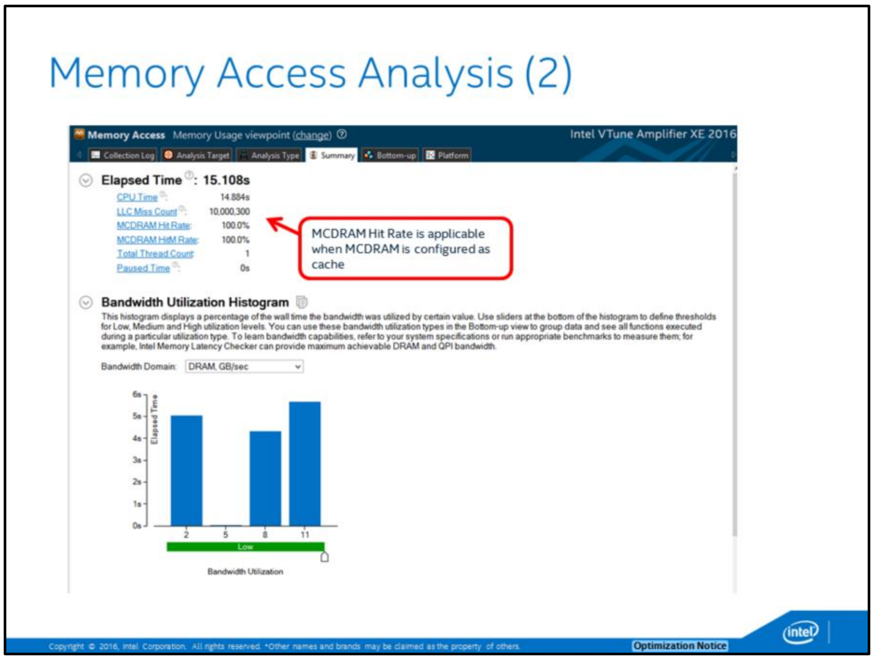
- Provides individual bandwidth information for both, MCDRAM and DDR.
- VTune cannot yet identify the system configuration: cluster mode and memory modes. Hence, shows the bandwidth information for both cache and flat mode. Users need to choose the correct data based on system configuration.





Memory Access Analysis

- Flat Mode
  - Allows the developers to explicitly control which data structures are in MCDRAM vs. DDR.
  - Requires code modification, otherwise DDR will be used by default.
- Cache Mode
  - No code modification. Hardware will use L1, then L2, then MCDRAM cache.
  - Data allocated into MCDRAM cache needs to be highly reusable to see performance benefits.
  - Average L2 miss latency is higher because misses in MCDRAM cache then go to DDR
  - May have aliasing issues if multiple pages are mapped into same cache lines in MCDRAM Cache. Non deterministic (not repeatable)
  - Streaming stores are negatively impacted. Expect lower bandwidth



Memory Access Analysis (2)

Elapsed Time: 15.108s

CPU Time:	14.884s
LLC Miss Count:	10,000,300
MCDRAM Hit Rate:	100.0%
MCDRAM HitM Rate:	100.0%
Total Thread Count:	1
Paused Time:	0s

MCDRAM Hit Rate is applicable when MCDRAM is configured as cache

Bandwidth Utilization Histogram

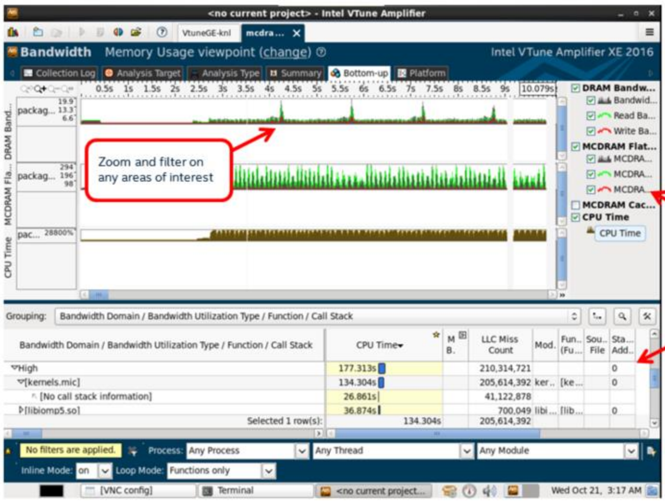
This histogram displays a percentage of the wall time the bandwidth was utilized by certain value. Use sliders at the bottom of the histogram to define thresholds for Low, Medium and High utilization levels. You can use these bandwidth utilization types in the Bottom-up view to group data and see all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

Bandwidth Domain: DRAM, GB/sec

Optimization Notice

1. Notes about MCDRAM Hit Rate
    1. This rate counts loads and streaming stores, e.g. vmovnt (non-temporal), but not stores/writebacks
    2. If you have streaming stores, your MCDRAM Hit Rate may be lower than expected because streaming stores are expected to miss MCDRAM Cache

Memory Access Analysis (3)



<no current project> - Intel VTune Amplifier
VtuneGE-knl
mcdra...
Bandwidth Memory Usage viewpoint (change)
Intel VTune Amplifier XE 2016
Collection Log
Analysis Target
Analysis Type
Summary
Bottom-up
Platform
DRAM Band...
MCDRAM Fla...
CPU Time
DRAM Bandw...
Bandwid...
Read Ba...
Write Ba...
MCDRAM Flat...
MCDRA...
MCDRAM Cac...
CPU Time
Zoom and filter on any areas of interest
Select Flat or Cache mode for MCDRAM
Memory information at source code level.

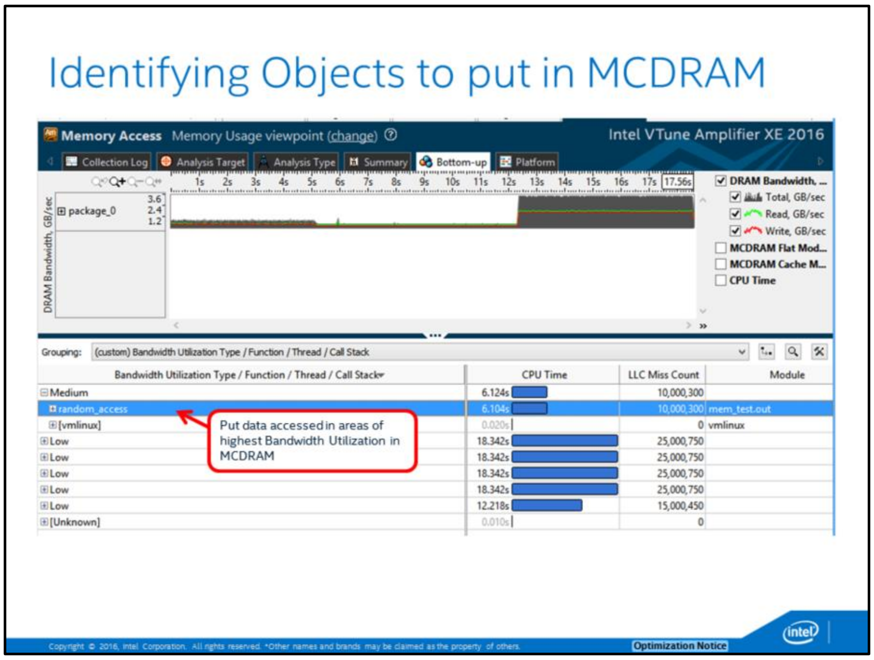
Grouping: Bandwidth Domain / Bandwidth Utilization Type / Function / Call Stack

Bandwidth Domain / Bandwidth Utilization Type / Function / Call Stack	CPU Time	M B.	LLC Miss Count	Mod.	Fun.. (Fu...	Sou.. File	Sta... Add..
High	177.313s		210,314,721				0
[kernels.mic]	134.304s		205,614,392	ker..	[ke...		0
[No call stack information]	26.861s		41,122,878				
[libiomp5.so]	36.874s		700,049	libi...	[lib...		0
Selected 1 row(s):	134.304s		205,614,392				

No filters are applied. Process: Any Process | Any Thread | Any Module
Inline Mode: on | Loop Mode: Functions only

[VNC config] Terminal <no current project... Wed Oct 21, 3:17 AM

Optimization Notice



Identifying Objects to put in MCDRAM

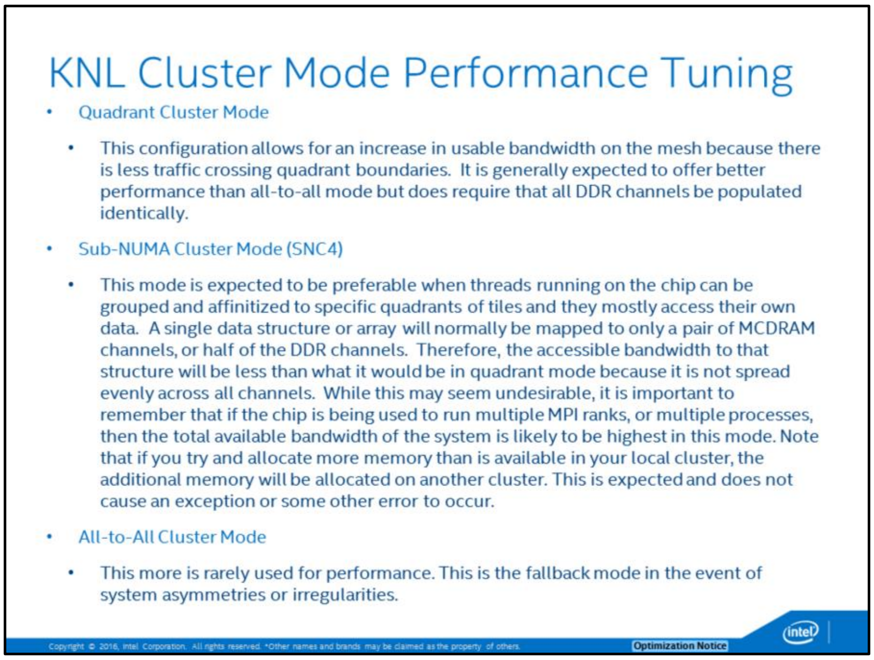
Grouping: (custom) Bandwidth Utilization Type / Function / Thread / Call Stack

Bandwidth Utilization Type / Function / Thread / Call Stack	CPU Time	LLC Miss Count	Module
Medium	6.124s	10,000,300	
random_access	6.104s	10,000,300	mem_test.out
[vmlinux]	0.020s	0	vmlinux
Low	18.342s	25,000,750	
Low	18.342s	25,000,750	
Low	18.342s	25,000,750	
Low	18.342s	25,000,750	
Low	12.218s	15,000,450	
[Unknown]	0.010s	0	

Put data accessed in areas of highest Bandwidth Utilization in MCDRAM

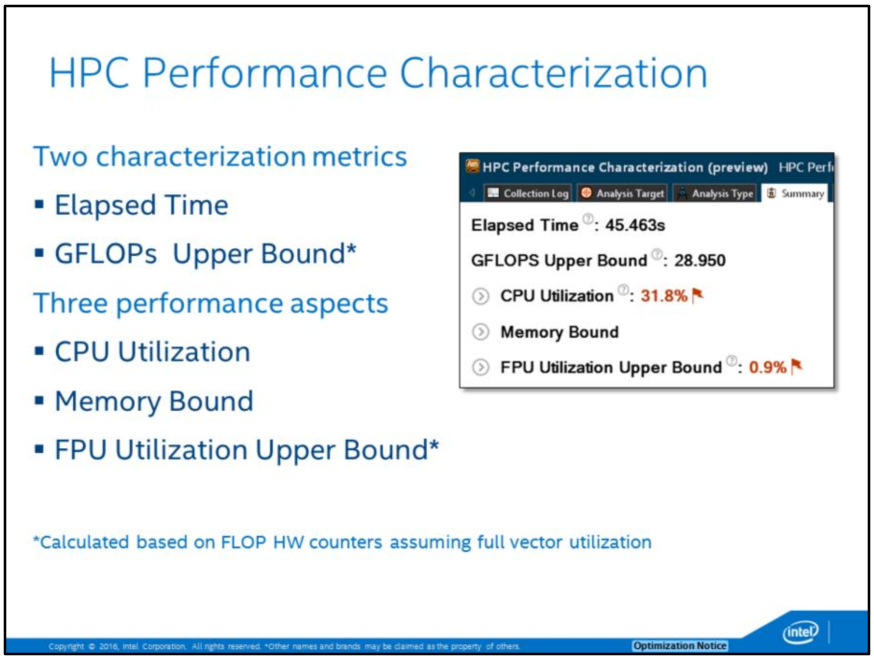
Optimization Notice

Run a Memory Access analysis with MCDRAM configures in flat mode and all allocations occurring in DDR (not using MCDRAM).
Create a custom grouping in the Memory Access analysis to see functions causing Medium or High bandwidth utilization. Objects accessed within these functions may be candidates to move into MCDRAM.



KNL Cluster Mode Performance Tuning

- Quadrant Cluster Mode
  - This configuration allows for an increase in usable bandwidth on the mesh because there is less traffic crossing quadrant boundaries. It is generally expected to offer better performance than all-to-all mode but does require that all DDR channels be populated identically.
- Sub-NUMA Cluster Mode (SNC4)
  - This mode is expected to be preferable when threads running on the chip can be grouped and affinitized to specific quadrants of tiles and they mostly access their own data. A single data structure or array will normally be mapped to only a pair of MCDRAM channels, or half of the DDR channels. Therefore, the accessible bandwidth to that structure will be less than what it would be in quadrant mode because it is not spread evenly across all channels. While this may seem undesirable, it is important to remember that if the chip is being used to run multiple MPI ranks, or multiple processes, then the total available bandwidth of the system is likely to be highest in this mode. Note that if you try and allocate more memory than is available in your local cluster, the additional memory will be allocated on another cluster. This is expected and does not cause an exception or some other error to occur.
- All-to-All Cluster Mode
  - This more is rarely used for performance. This is the fallback mode in the event of system asymmetries or irregularities.



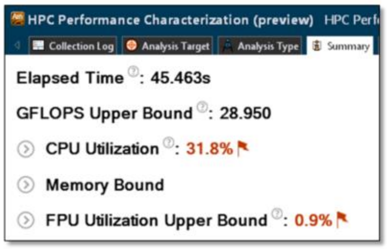
HPC Performance Characterization

Two characterization metrics

- Elapsed Time
- GFLOPs Upper Bound*

Three performance aspects

- CPU Utilization
- Memory Bound
- FPU Utilization Upper Bound*



Elapsed Time: 45.463s

GFLOPS Upper Bound: 28.950

CPU Utilization: 31.8%

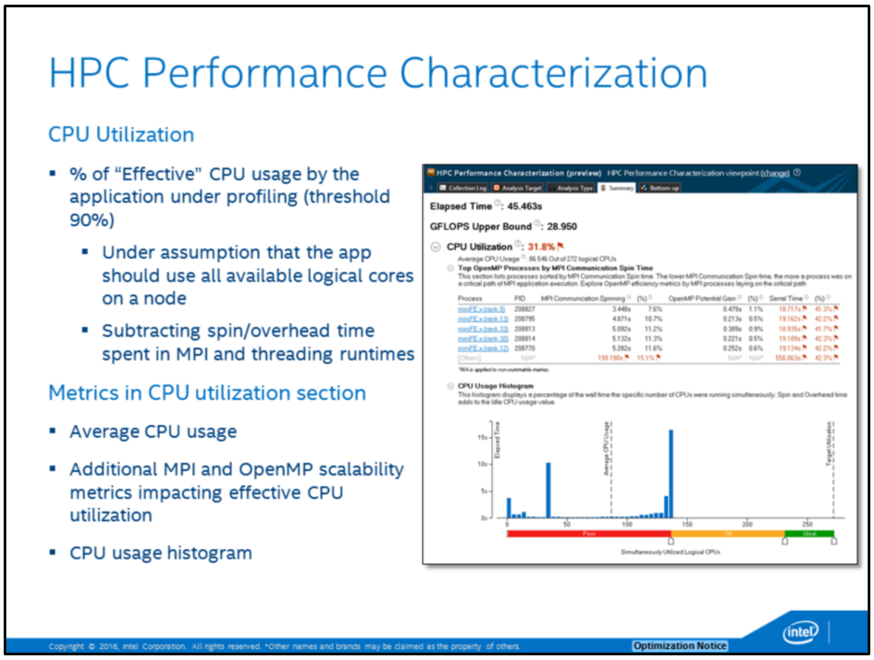
Memory Bound

FPU Utilization Upper Bound: 0.9%

*Calculated based on FLOP HW counters assuming full vector utilization

Optimization Notice

HPC Performance Characterization provides performance information that is especially important for High Performance Computing (HPC) applications. This analysis can be run from the GUI or using the command line flag "–collect hpc-performance"



HPC Performance Characterization

CPU Utilization

- % of "Effective" CPU usage by the application under profiling (threshold 90%)
  - Under assumption that the app should use all available logical cores on a node
  - Subtracting spin/overhead time spent in MPI and threading runtimes

Metrics in CPU utilization section

- Average CPU usage
- Additional MPI and OpenMP scalability metrics impacting effective CPU utilization
- CPU usage histogram

HPC Performance Characterization (preview) HPC Performance Characterization viewpoint (change)

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

Elapsed Time: 45.463s

GFLOPS Upper Bound: 28.950

CPU Utilization: 31.8%

Average CPU Usage: 86.546 Out of 272 logical CPUs

Top OpenMP Processes by MPI Communication Spin Time

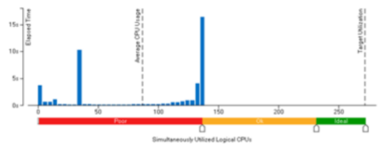
This section lists processes sorted by MPI Communication Spin time. The lower MPI Communication Spin time, the more a process was on a critical path of MPI application execution. Explore OpenMP efficiency metrics by MPI processes laying on the critical path

Process	PID	MPI Communication Spinning	(%)	OpenMP Potential Gain	(%)	Serial Time	(%)
miniFE.x (rank 0)	208827	3.448s	7.6%	0.479s	1.1%	18.717s	41.3%
miniFE.x (rank 13)	208795	4.871s	10.7%	0.213s	0.5%	19.162s	42.2%
miniFE.x (rank 33)	208813	5.092s	11.2%	0.389s	0.9%	18.935s	41.7%
miniFE.x (rank 30)	208814	5.132s	11.3%	0.221s	0.5%	19.189s	42.3%
miniFE.x (rank 12)	208770	5.282s	11.6%	0.252s	0.6%	19.134s	42.2%
[Others]	N/A*	199.190s	15.1%	N/A*	N/A*	556.063s	42.3%

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

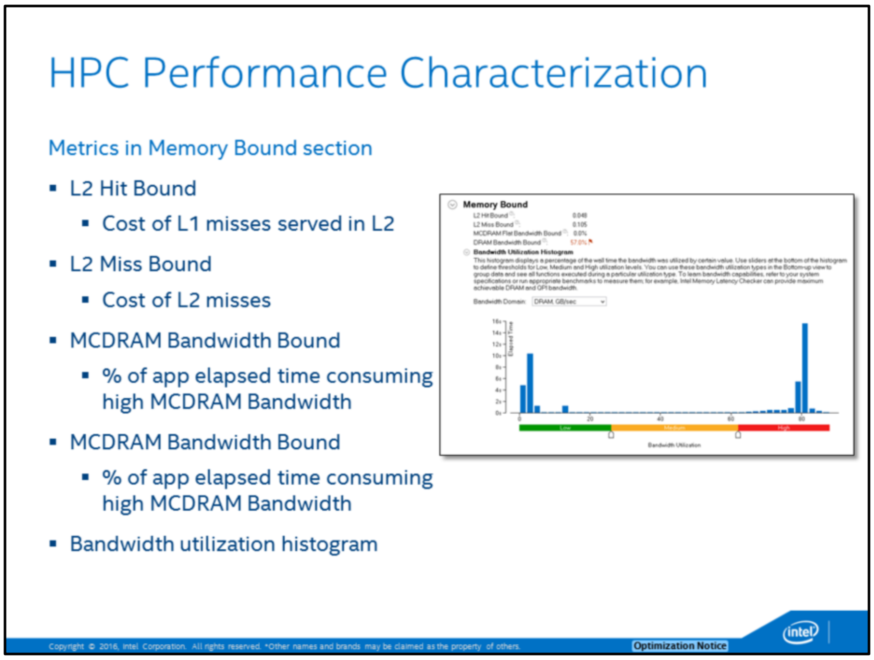
CPU Usage Histogram

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



Optimization Notice

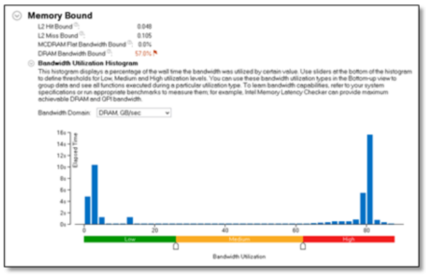
The CPU Utilization metrics provide another way to determine how busy all of the cores are during the performance analysis.



HPC Performance Characterization

Metrics in Memory Bound section

- L2 Hit Bound
  - Cost of L1 misses served in L2
- L2 Miss Bound
  - Cost of L2 misses
- MCDRAM Bandwidth Bound
  - % of app elapsed time consuming high MCDRAM Bandwidth
- MCDRAM Bandwidth Bound
  - % of app elapsed time consuming high MCDRAM Bandwidth
- Bandwidth utilization histogram



Memory Bound

L2 Hit Bound:	0.048
L2 Miss Bound:	0.105
MCDRAM Flat Bandwidth Bound:	0.0%
DRAM Bandwidth Bound:	57.0%

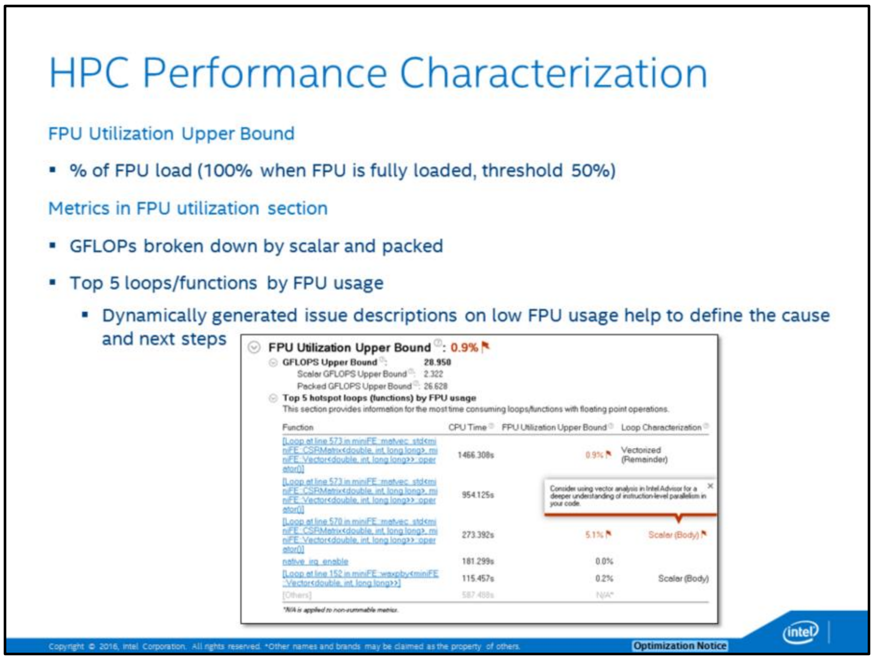
Bandwidth Utilization Histogram

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

Bandwidth Domain: DRAM, GB/sec

Optimization Notice

The Memory Bound metrics provide information about how the application is utilizing, and possibly bottlenecked by, the memory subsystem. If issues are exposed here, the Memory Access analysis may provide even more detailed information.



HPC Performance Characterization

FPU Utilization Upper Bound

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

Metrics in FPU utilization section

- GFLOPs broken down by scalar and packed
- Top 5 loops/functions by FPU usage
  - Dynamically generated issue descriptions on low FPU usage help to define the cause and next steps

FPU Utilization Upper Bound: 0.9%

GFLOPS Upper Bound: 28.950

Scalar GFLOPS Upper Bound: 2.322

Packed GFLOPS Upper Bound: 26.628

Top 5 hotspot loops (functions) by FPU usage

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

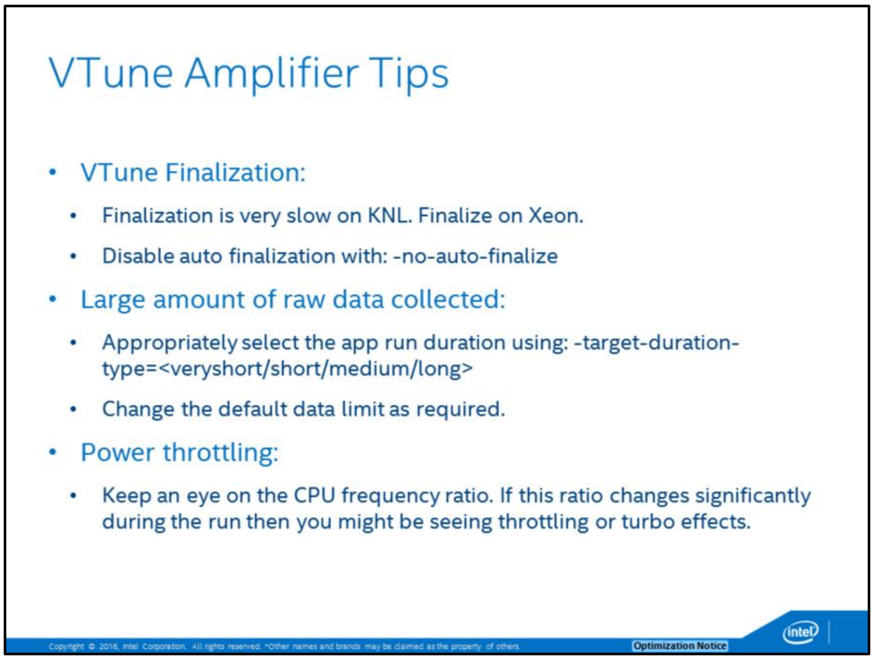
Function	CPU Time	FPU Utilization Upper Bound	Loop Characterization
[Loop at line 573 in miniFE::matvec_std<miniFE::CSRMatrix<double, int, long long>, miniFE::Vector<double, int, long long>>::operator()]	1466.308s	0.9%	Vectorized (Remainder)
[Loop at line 573 in miniFE::matvec_std<miniFE::CSRMatrix<double, int, long long>, miniFE::Vector<double, int, long long>>::operator()]	954.125s		
[Loop at line 570 in miniFE::matvec_std<miniFE::CSRMatrix<double, int, long long>, miniFE::Vector<double, int, long long>>::operator()]	273.392s	5.1%	Scalar (Body)
native_irq_enable	181.299s	0.0%	
[Loop at line 152 in miniFE::waxpby<miniFE::Vector<double, int, long long>>]	115.457s	0.2%	Scalar (Body)
[Others]	587.488s	N/A*	

Consider using vector analysis in Intel Advisor for a deeper understanding of instruction-level parallelism in your code.

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

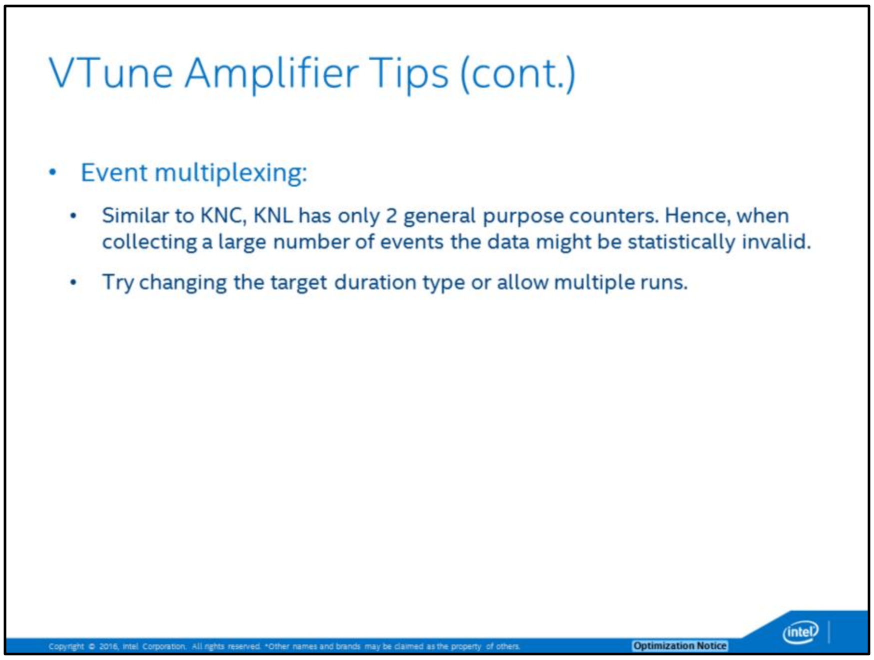
Optimization Notice

The Floating Point Units (FPU) on KNL are important for getting the best performance. The HPC Performance Characterization provides an upper bound estimate of the utilization. This is an upper bound because the events used are not able to account for masking, and the metric assumes all vector lanes are used in each instruction.



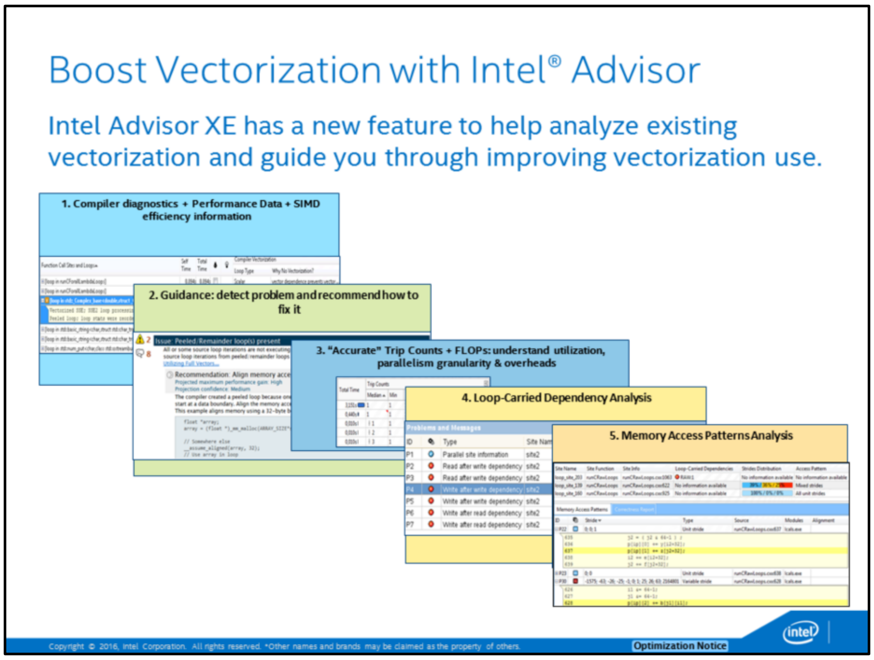
VTune Amplifier Tips

- VTune Finalization:
  - Finalization is very slow on KNL. Finalize on Xeon.
  - Disable auto finalization with: -no-auto-finalize
- Large amount of raw data collected:
  - Appropriately select the app run duration using: -target-duration-type=<veryshort/short/medium/long>
  - Change the default data limit as required.
- Power throttling:
  - Keep an eye on the CPU frequency ratio. If this ratio changes significantly during the run then you might be seeing throttling or turbo effects.



VTune Amplifier Tips (cont.)

- Event multiplexing:
  - Similar to KNC, KNL has only 2 general purpose counters. Hence, when collecting a large number of events the data might be statistically invalid.
  - Try changing the target duration type or allow multiple runs.



Boost Vectorization with Intel® Advisor

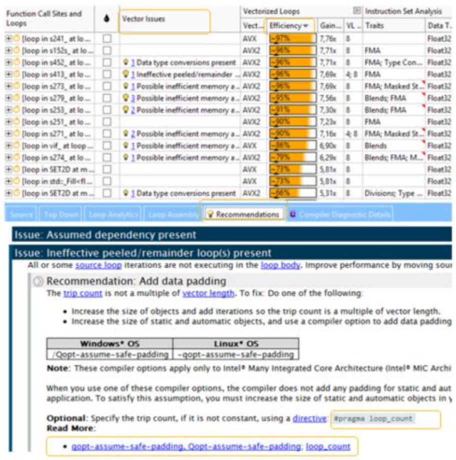
Intel Advisor XE has a new feature to help analyze existing vectorization and guide you through improving vectorization use.

1. Compiler diagnostics + Performance Data + SIMD efficiency information
2. Guidance: detect problem and recommend how to fix it
3. "Accurate" Trip Counts + FLOPs: understand utilization, parallelism granularity & overheads
4. Loop-Carried Dependency Analysis
5. Memory Access Patterns Analysis

Optimization Notice

Use this 5 step process to determine how well you are vectorizing and where you can improve.
Intel® Advisor is available at: https://software.intel.com/en-us/intel-advisor-xe

Survey Analysis



Function Call Sites and Loops		Vector Issues	Vectorized Loops				Instruction Set Analysis	
			Vect...	Efficiency	Gain...	VL ...	Traits	Data T...
[loop in s241_ at lo...	☐		AVX	~97%	7,76x	8		Float32
[loop in s152s_ at lo...	☐		AVX2	~96%	7,71x	8	FMA	Float32
[loop in s452_ at lo...	☐	1 Data type conversions present	AVX2	~96%	7,71x	8	FMA; Type Con...	Float32
[loop in s413_ at lo...	☐	1 Ineffective peeled/remainder ...	AVX2	~96%	7,69x	4; 8	FMA	Float32
[loop in s273_ at lo...	☐	1 Possible inefficient memory a...	AVX2	~96%	7,69x	8	FMA; Masked St...	Float32
[loop in s279_ at lo...	☐	3 Possible inefficient memory a...	AVX2	~95%	7,56x	8	Blends; FMA	Float32
[loop in s253_ at lo...	☐	2 Possible inefficient memory a...	AVX2	~91%	7,30x	8	Blends; FMA	Float32
[loop in s251_ at lo...	☐		AVX2	~90%	7,23x	8	FMA	Float32
[loop in s271_ at lo...	☐	2 Possible inefficient memory a...	AVX2	~90%	7,16x	4; 8	FMA; Masked St...	Float32
[loop in vif_ at loop ...	☐	1 Possible inefficient memory a...	AVX	~86%	6,90x	8	Blends	Float32
[loop in s274_ at lo...	☐	1 Possible inefficient memory a...	AVX2	~79%	6,29x	8	Blends; FMA; M...	Float32
[loop in SET2D at m...	☐		AVX	~73%	5,81x	8		Float32
[loop in std::_Fill<fl...	☐		AVX	~73%	5,81x	8		Float32
[loop in SET2D at m...	☐	1 Data type conversions present	AVX2	~66%	5,31x	8	Divisions; Type ...	Float32

Issue: Assumed dependency present

Issue: Ineffective peeled/remainder loop(s) present

All or some source loop iterations are not executing in the loop body. Improve performance by moving sou

Recommendation: Add data padding

The trip count is not a multiple of vector length. To fix: Do one of the following:

- Increase the size of objects and add iterations so the trip count is a multiple of vector length.
- Increase the size of static and automatic objects, and use a compiler option to add data padding

Windows* OS	Linux* OS
/Qopt-assume-safe-padding	-qopt-assume-safe-padding

Note: These compiler options apply only to Intel® Many Integrated Core Architecture (Intel® MIC Archi

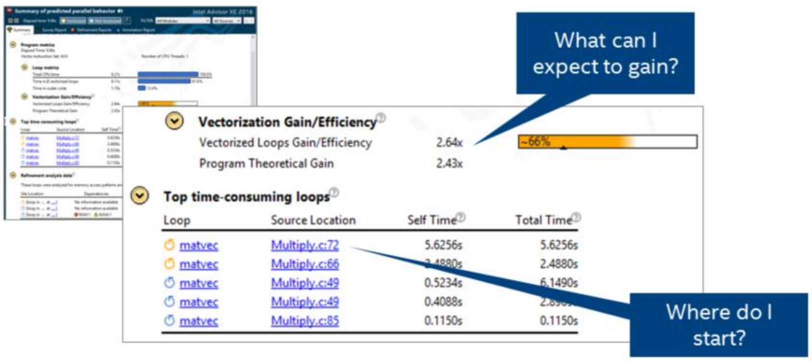
When you use one of these compiler options, the compiler does not add any padding for static and aut application. To satisfy this assumption, you must increase the size of static and automatic objects in y

Optional: Specify the trip count, if it is not constant, using a directive: `#pragma loop_count`
Read More:

- qopt-assume-safe-padding, Qopt-assume-safe-padding; loop_count

- **Sort** – Look at your hottest vectorized loops
- **Efficiency** – use as a performance thermometer
- **Recommendations** – get tips on how to improve performance

Summary View: Plan Your Next Steps

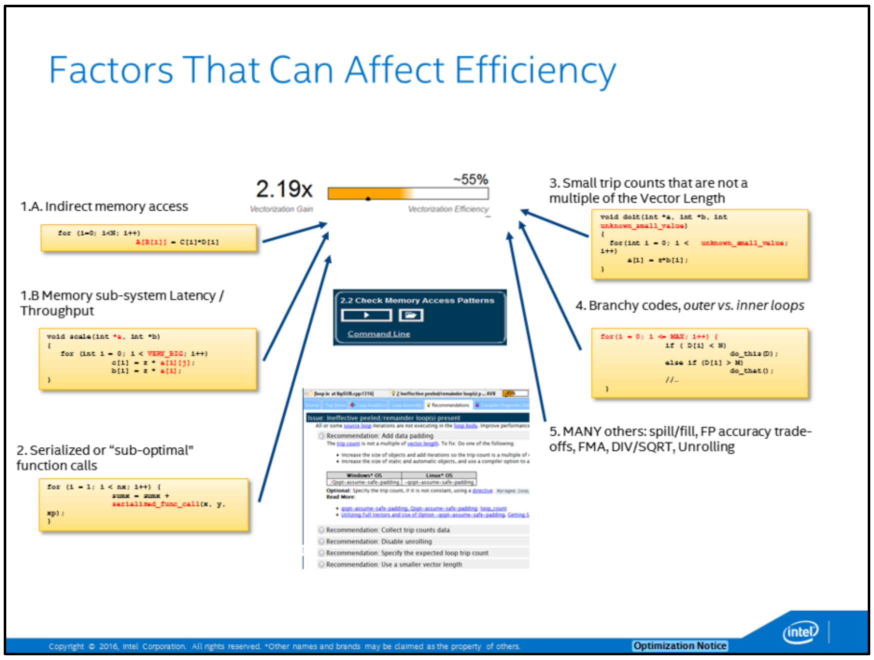


Vectorization Gain/Efficiency

Vectorized Loops Gain/Efficiency	2.64x	~66%
Program Theoretical Gain	2.43x	

Top time-consuming loops

Loop	Source Location	Self Time	Total Time
matvec	Multiply.c:72	5.6256s	5.6256s
matvec	Multiply.c:66	2.4880s	2.4880s
matvec	Multiply.c:49	0.5234s	6.1490s
matvec	Multiply.c:49	0.4088s	2.89[illegible]
matvec	Multiply.c:85	0.1150s	0.1150s



Factors That Can Affect Efficiency

2.19x Vectorization Gain

~55% Vectorization Efficiency

1.A. Indirect memory access

```
for (i=0; i<N; i++)
            A[B[i]] = C[i]*D[i]
```

1.B Memory sub-system Latency / Throughput

```
void scale(int *a, int *b)
{
   for (int i = 0; i < VERY_BIG; i++)
            c[i] = z * a[i][j];
            b[i] = z * a[i];
}
```

2. Serialized or "sub-optimal" function calls

```
for (i = 1; i < nx; i++) {
            sumx = sumx +
            serialized_func_call(x, y,
xp);
}
```

3. Small trip counts that are not a multiple of the Vector Length

```
void doit(int *a, int *b, int
unknown_small_value)
{
  for(int i = 0; i <   unknown_small_value;
i++)
      a[i] = z*b[i];
}
```

4. Branchy codes, *outer vs. inner loops*

```
for(i = 0; i <= MAX; i++) {
            if ( D[i] < N)
                        do_this(D);
            else if (D[i] > M)
                        do_that();
            //…
 }
```

5. MANY others: spill/fill, FP accuracy trade-offs, FMA, DIV/SQRT, Unrolling

Issue: Ineffective peeled/remainder loop(s) present

All or some source loop iterations are not executing in the loop body. Improve performance

Recommendation: Add data padding

The trip count is not a multiple of vector length. To fix: Do one of the following:

- Increase the size of objects and add iterations so the trip count is a multiple of
- Increase the size of static and automatic objects, and use a compiler option to a

Windows* OS	Linux* OS
/Qopt-assume-safe-padding	-qopt-assume-safe-padding

Optional: Specify the trip count, if it is not constant, using a directive: #pragma loop

Read More:

- qopt-assume-safe-padding, Qopt-assume-safe-padding; loop_count
- Utilizing Full Vectors and Use of Option -qopt-assume-safe-padding, Getting S

Recommendation: Collect trip counts data

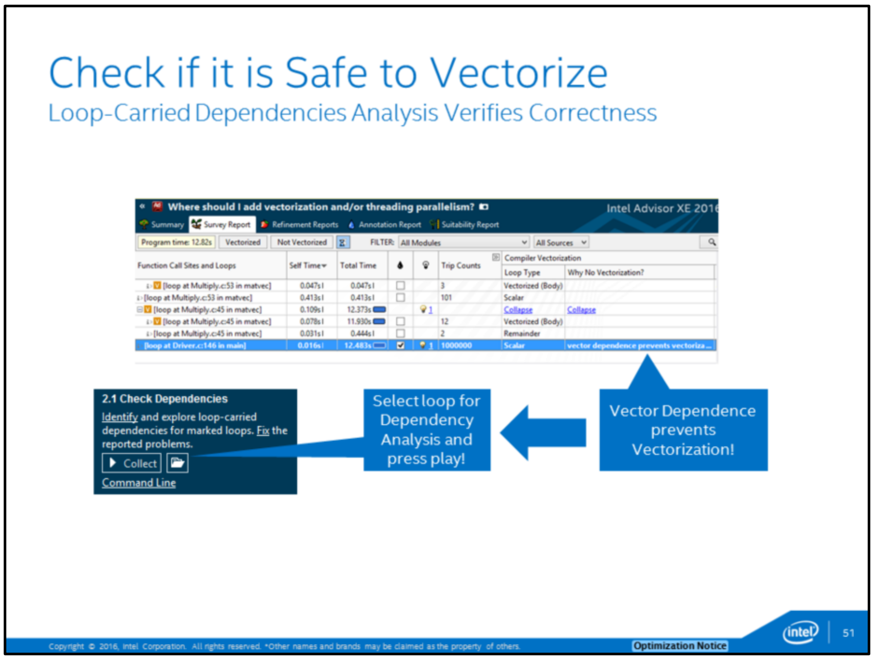
Recommendation: Disable unrolling

Recommendation: Specify the expected loop trip count

Recommendation: Use a smaller vector length

Optimization Notice

Analyze the hot loops for the common issues that can impact vectorization. Use the Memory Access Patterns Analysis and Recommendations to identify problematic behaviors and ways to correct them.



Check if it is Safe to Vectorize

Loop-Carried Dependencies Analysis Verifies Correctness

Where should I add vectorization and/or threading parallelism? Intel Advisor XE 2016

Summary | Survey Report | Refinement Reports | Annotation Report | Suitability Report

Program time: 12.82s | Vectorized | Not Vectorized | FILTER: All Modules | All Sources

Function Call Sites and Loops	Self Time	Total Time			Trip Counts	Compiler Vectorization	
						Loop Type	Why No Vectorization?
[loop at Multiply.c:53 in matvec]	0.047s	0.047s	☐		3	Vectorized (Body)	
[loop at Multiply.c:53 in matvec]	0.413s	0.413s	☐		101	Scalar	
[loop at Multiply.c:45 in matvec]	0.109s	12.373s		1		Collapse	Collapse
[loop at Multiply.c:45 in matvec]	0.078s	11.930s	☐		12	Vectorized (Body)	
[loop at Multiply.c:45 in matvec]	0.031s	0.444s	☐		2	Remainder	
[loop at Driver.c:146 in main]	0.016s	12.483s	☑	1	1000000	Scalar	vector dependence prevents vectoriza…

2.1 Check Dependencies

Identify and explore loop-carried dependencies for marked loops. Fix the reported problems.

Collect

Command Line

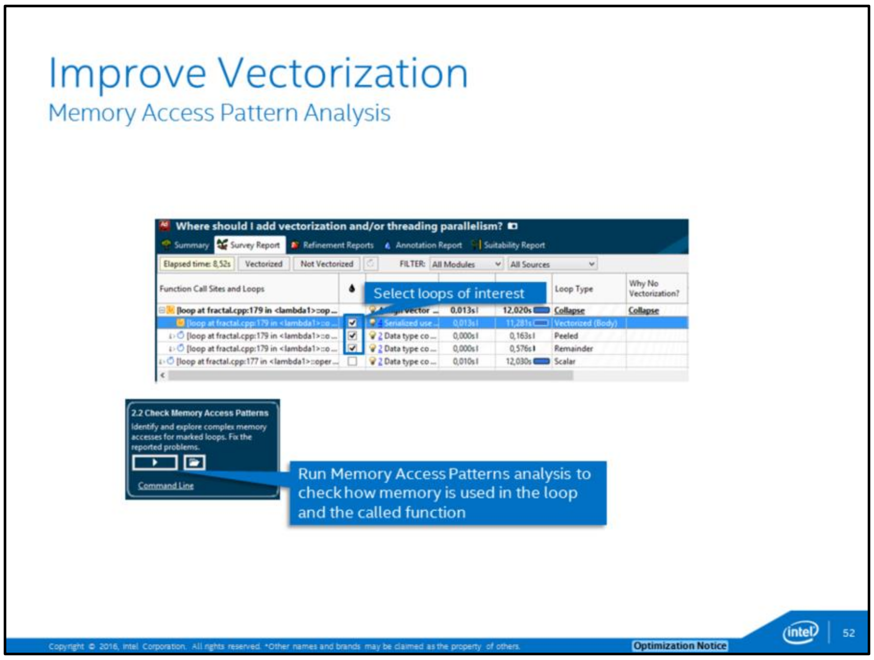
Select loop for Dependency Analysis and press play!

Vector Dependence prevents Vectorization!

Data dependencies between loop iterations make it difficult for the compiler to vectorize a loop. For example:

```
for (i = 1; i < N; i++) {
   A[i] = A[i-1] + C[i];
}
```

Each iteration is dependent on the value calculated in the previous iteration. Use Advisor to detect these dependencies.



Improve Vectorization

Memory Access Pattern Analysis

Where should I add vectorization and/or threading parallelism?

Summary | Survey Report | Refinement Reports | Annotation Report | Suitability Report

Elapsed time: 8,52s | Vectorized | Not Vectorized | FILTER: All Modules | All Sources

Function Call Sites and Loops	●				Loop Type	Why No Vectorization?
[loop at fractal.cpp:179 in <lambda1>::op ...		[illegible] vector ...	0,013s	12,020s	Collapse	Collapse
[loop at fractal.cpp:179 in <lambda1>::o ...	✓	4 Serialized use ...	0,013s	11,281s	Vectorized (Body)	
[loop at fractal.cpp:179 in <lambda1>::o ...	✓	2 Data type co ...	0,000s	0,163s	Peeled	
[loop at fractal.cpp:179 in <lambda1>::o ...	✓	2 Data type co ...	0,000s	0,576s	Remainder	
[loop at fractal.cpp:177 in <lambda1>::oper ...	☐	2 Data type co ...	0,010s	12,030s	Scalar	

Select loops of interest

2.2 Check Memory Access Patterns

Identify and explore complex memory accesses for marked loops. Fix the reported problems.

Command Line

Run Memory Access Patterns analysis to check how memory is used in the loop and the called function

Optimization Notice

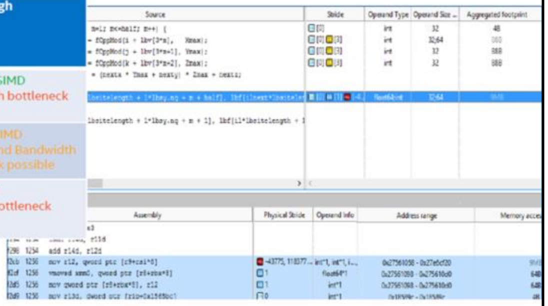
52

The strides of memory accesses can affect vectorization. Determine the patterns to learn which loops may be difficult to vectorize.

Memory Analysis Is Critical

Determine Possible Bandwidth or Latency Issues

Footprint / Access Pattern	Small enough	Big enough
Unit Stride	Effective SIMD No Latency and BW bottlenecks	Effective SIMD Bandwidth bottleneck
Const stride	Medium SIMD Latency bottleneck possible	Medium SIMD Latency and Bandwidth bottleneck possible
Irregular Access, Gather/Scatter	Bad SIMD Latency bottleneck possible	Bad SIMD Latency bottleneck

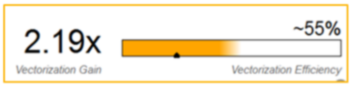


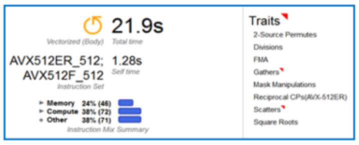
Source	Stride	Operand Type	Operand Size	Aggregated footprint
m=1; m<=half; m++) {	▢ [0]	int	32	48
= fOppMod(i + lbv[3*m], Xmax);	▢ [0] ▢ [3]	int	32;64	[illegible]
= fOppMod(j + lbv[3*m+1], Ymax);	▢ [0] ▢ [3]	int	32	888
= fOppMod(k + lbv[3*m+2], Zmax);	▢ [0] ▢ [3]	int	32	888
= (nextx * Ymax + nexty) * Zmax + nextz;				
lbsitelength + l*lbsy.nq + m + half], lbf[ilnext*lbsitele	▢ [0] ▢ [1] ▢ [-4	float64;int	32;64	[illegible]
lbsitelength + l*lbsy.nq + m + 1], lbf[il*lbsitelength + l				

Assembly	Physical Stride	Operand Info	Address range	Memory acce
x3				
r15d				
1298 1254 add r14d, r12d				
12cb 1256 mov r12, qword ptr [r9+rsi*8]	-43775, 118377...	int*1, int*1, i...	0x27561058 - 0x27e6cf20	[illegible]
12cf 1256 vmovsd xmm0, qword ptr [r8+rbx*8]	1	float64*1	0x27561098 - 0x275610d0	64B
12d5 1256 mov qword ptr [r8+rbx*8], r12	1	int*1	0x27561098 - 0x275610d0	64B
12d9 1256 mov r13d, dword ptr [rip+0x1565bc]	0	int*1	[illegible]	4B

AVX-512 Specifics for KNL

1. **Native AVX-512 profiling** on KNL
2. Precise **FLOPs and Mask Utilization** profiler
3. AVX-512 **Advice and Traits**
4. AVX-512 **Gather/Scatter Profiler**


2.19x
Vectorization Gain
~55%
Vectorization Efficiency

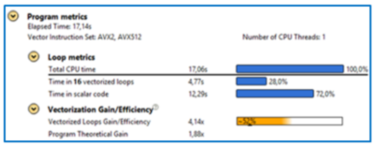

21.9s
Vectorized (Body) Total time

AVX512ER_512; AVX512F_512 1.28s
Instruction Set Self time

▸ Memory	24% (46)	▬
▸ Compute	38% (72)	▬
• Other	38% (71)	▬

Instruction Mix Summary

Traits
- 2-Source Permutes
- Divisions
- FMA
- Gathers
- Mask Manipulations
- Reciprocal CPs(AVX-512ER)
- Scatters
- Square Roots

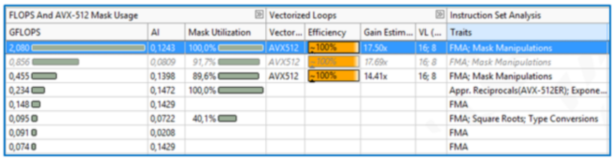

Program metrics
Elapsed Time: 17,14s
Vector Instruction Set: AVX2, AVX512 Number of CPU Threads: 1

Loop metrics

Total CPU time	17,06s	100,0%
Time in **16** vectorized loops	4,77s	28,0%
Time in scalar code	12,29s	72,0%

Vectorization Gain/Efficiency

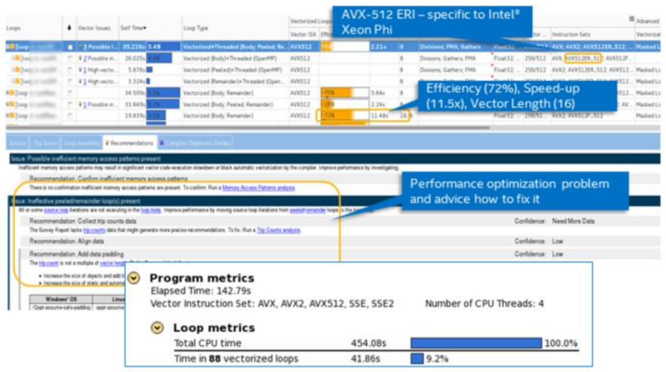
Vectorized Loops Gain/Efficiency	4,14x	~52%
Program Theoretical Gain	1,88x	



FLOPS And AVX-512 Mask Usage			Vectorized Loops				Instruction Set Analysis
GFLOPS	AI	Mask Utilization	Vector...	Efficiency	Gain Estim...	VL (...	Traits
2,080	0,1243	100,0%	AVX512	~100%	17.50x	16; 8	FMA; Mask Manipulations
0,856	0,0809	91,7%	AVX512	~100%	17.69x	16; 8	FMA; Mask Manipulations
0,455	0,1398	89,6%	AVX512	~100%	14.41x	16; 8	FMA; Mask Manipulations
0,234	0,1472	100,0%					Appr. Reciprocals(AVX-512ER); Expone...
0,148	0,1429						FMA
0,095	0,0722	40,1%					FMA; Square Roots; Type Conversions
0,091	0,0208						FMA
0,074	0,1429						FMA

Vectorization Advisor on KNL AVX-512

See the Intel Advisor tutorials and documentation to learn how to analyze your KNL application.

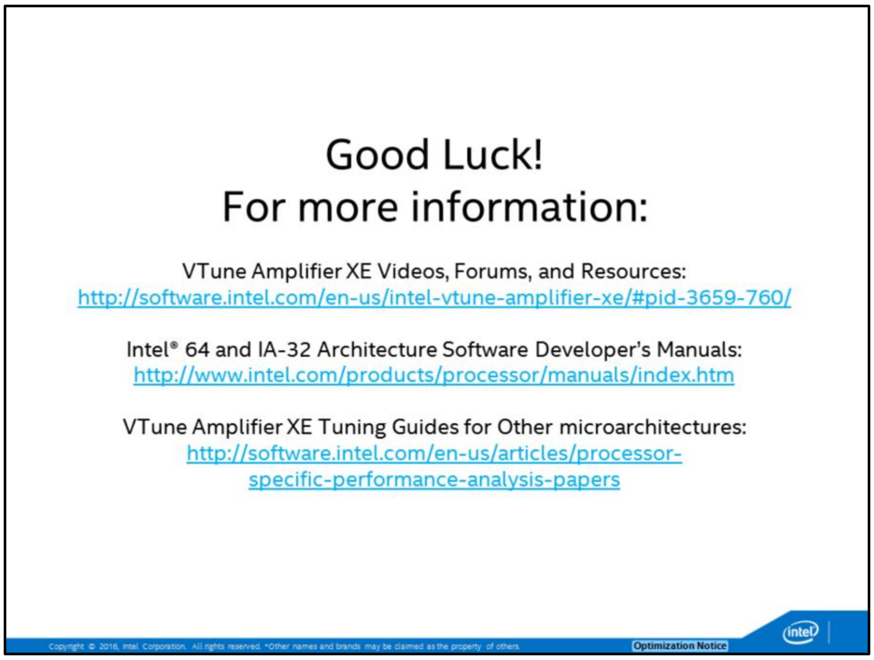

AVX-512 ERI – specific to Intel® Xeon Phi
Efficiency (72%), Speed-up (11.5x), Vector Length (16)
Performance optimization problem and advice how to fix it

Program metrics
Elapsed Time: 142.79s
Vector Instruction Set: AVX, AVX2, AVX512, SSE, SSE2 Number of CPU Threads: 4

Loop metrics

Total CPU time	454.08s	100.0%
Time in **88** vectorized loops	41.86s	9.2%

Optimization Notice

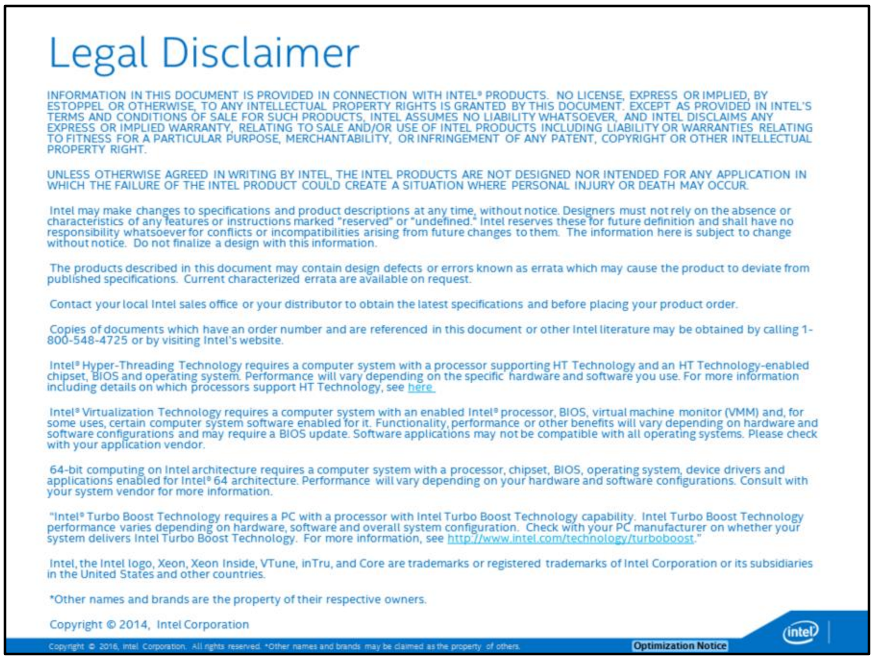


Good Luck!
For more information:

VTune Amplifier XE Videos, Forums, and Resources:
http://software.intel.com/en-us/intel-vtune-amplifier-xe/#pid-3659-760/

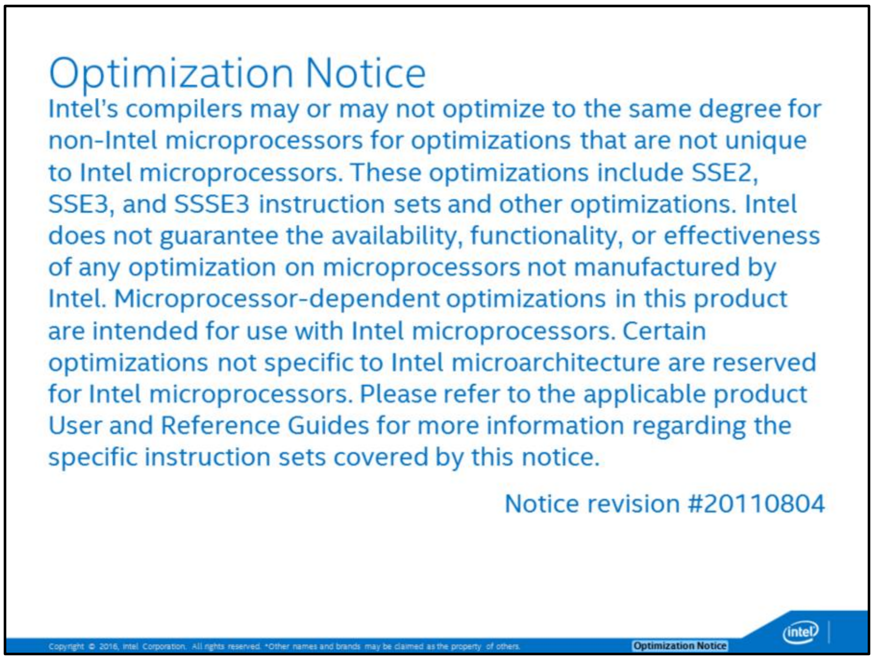
Intel® 64 and IA-32 Architecture Software Developer's Manuals:
http://www.intel.com/products/processor/manuals/index.htm

VTune Amplifier XE Tuning Guides for Other microarchitectures:
http://software.intel.com/en-us/articles/processor-specific-performance-analysis-papers



Legal Disclaimer

Optimization Notice



Optimization Notice

Optimization Notice