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

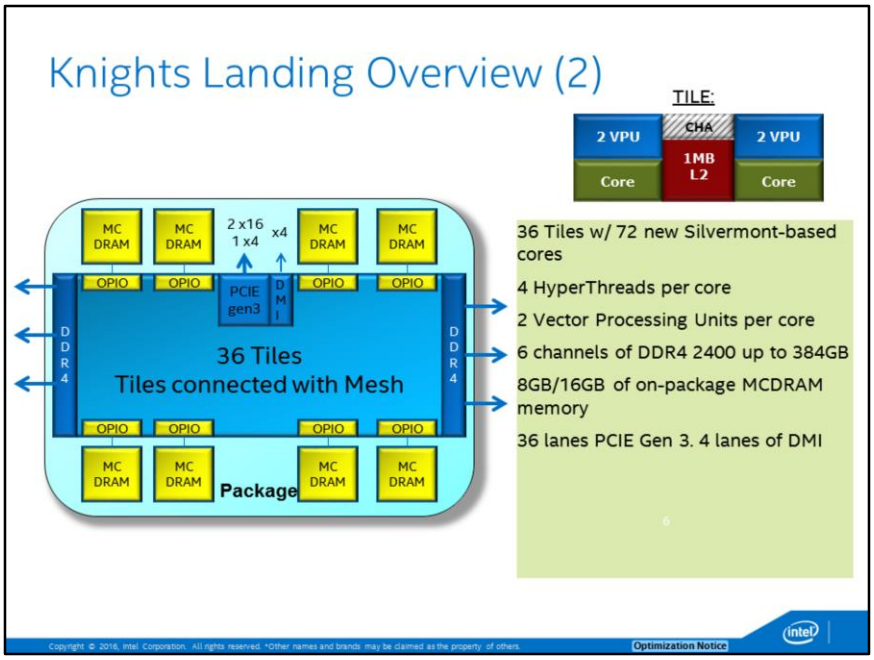
Ver. 1.1

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



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

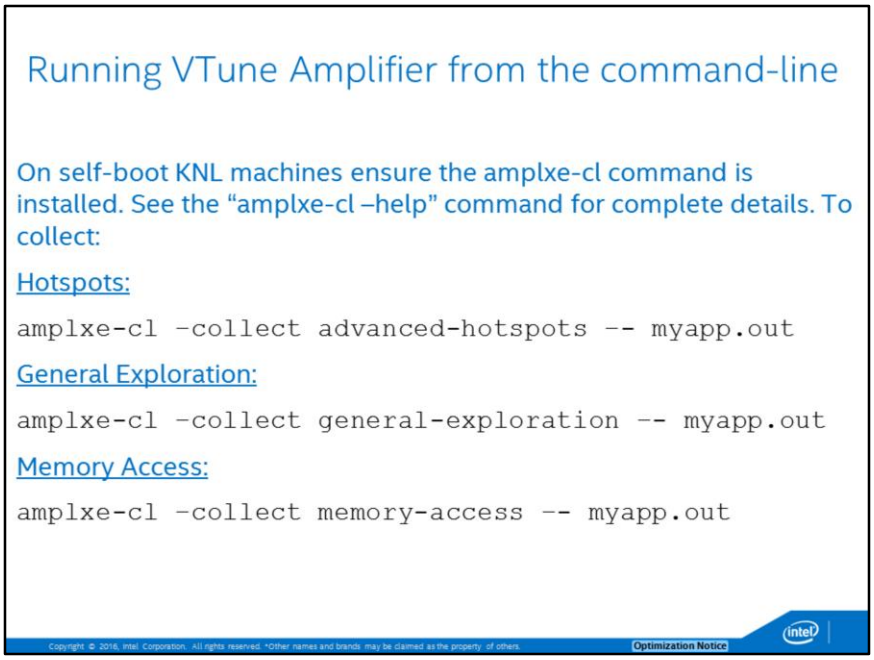
2 VPUs: 2x 512b Vectors. 32SP and 16DP. X87, SSE and EMU support

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*

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.



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

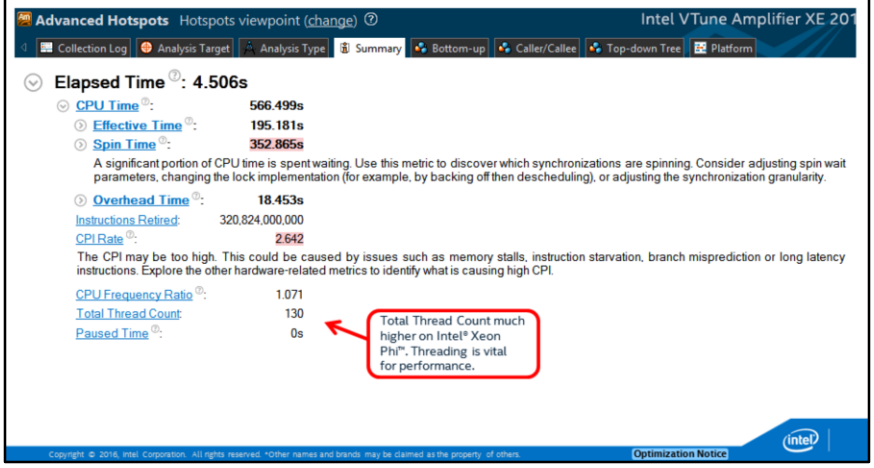
Results can be viewed from the command-line or GUI on the KNL machine, but it is generally more efficient to copy results to another machine with the GUI installed for analysis.

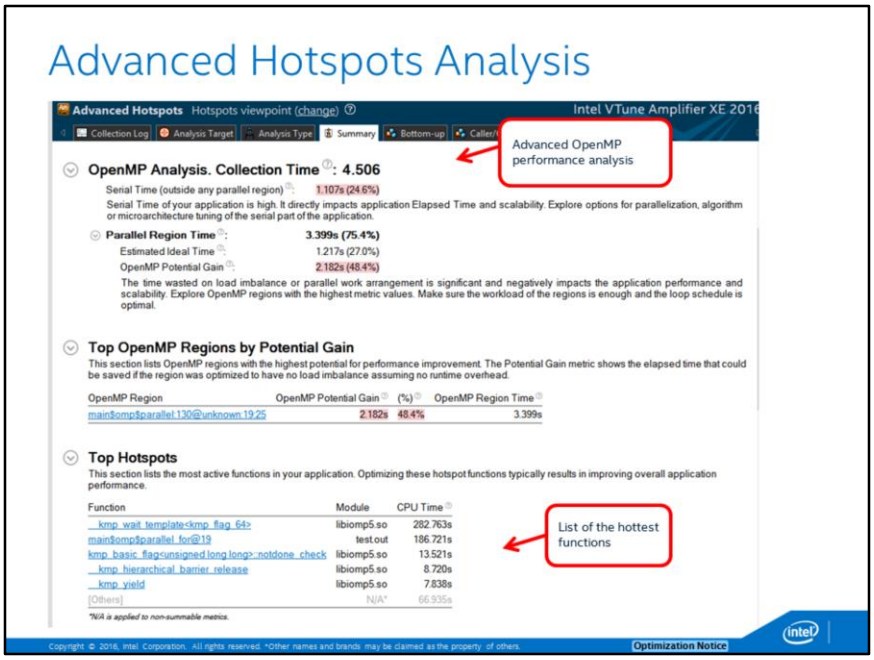
It is also recommended to add the –no-auto-finalize flag to collections that will be creating large results. The finalization step is compute intensive and runs serially which may take a long time on the KNL. Finalization can be done on another machine after copying the results off of the KNL.

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.

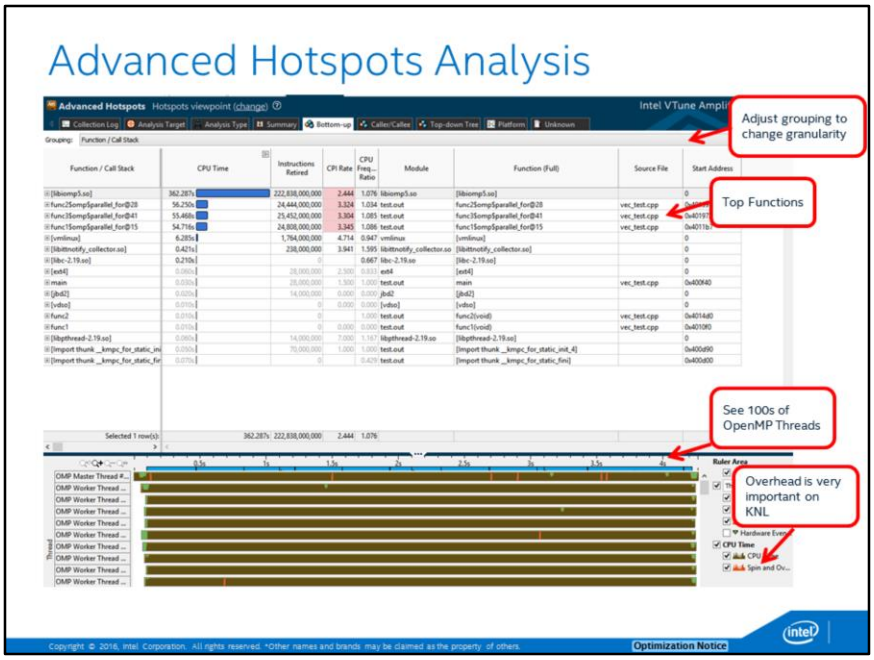
Advanced Hotspots Analysis

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

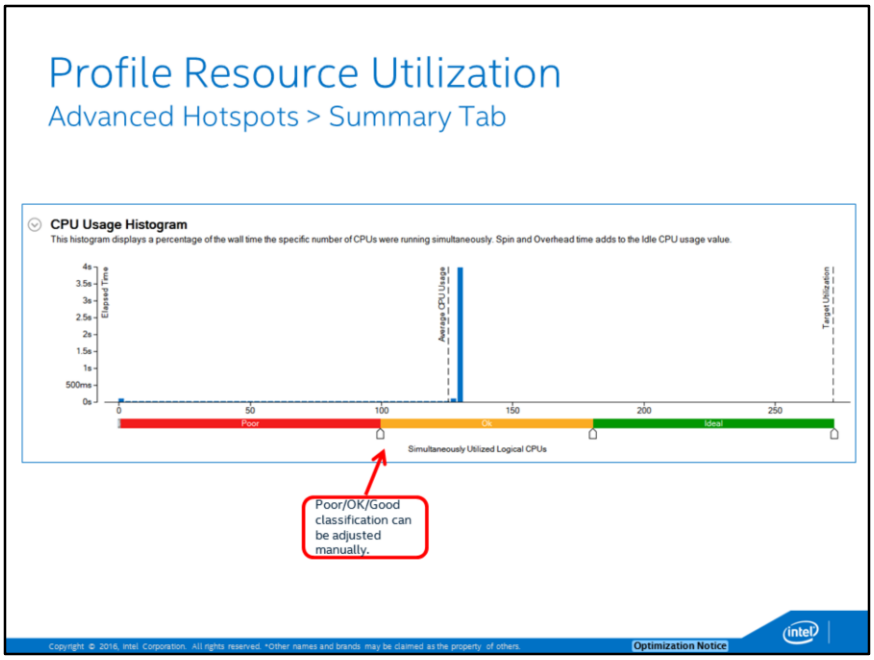




The Advanced Hotspots Analysis will show where your application is spending its time, including information related to OpenMP parallelism. Ensure that the OpenMP runtime library used in the application (e.g. libiomp5.so) is available on the system doing the analysis. This is required to accurately analyze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



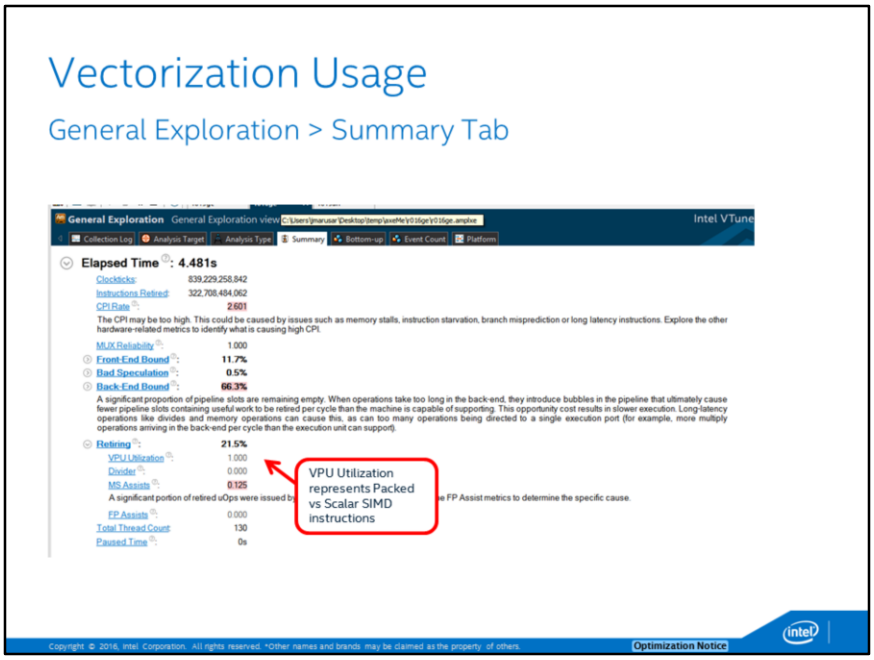
Use the Bottom-up view to see time spent at various granularities; for example Function or Module granularities. This can be changed in the Grouping drop-down menu. Focus tuning efforts on the hot portions of your application.



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.

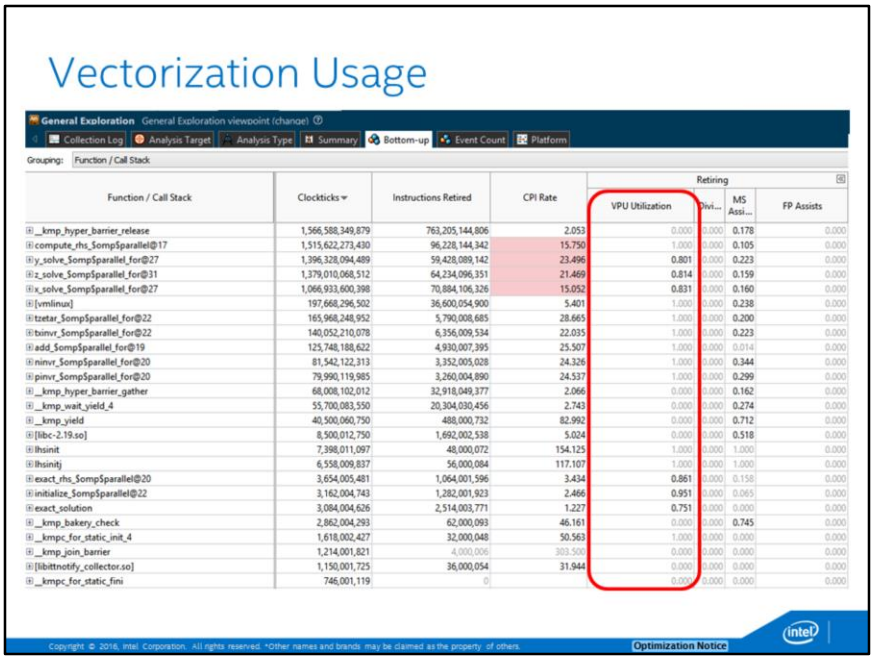
Memory Bandwidth may not be helped by more threads, but Memory Latency can.

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.



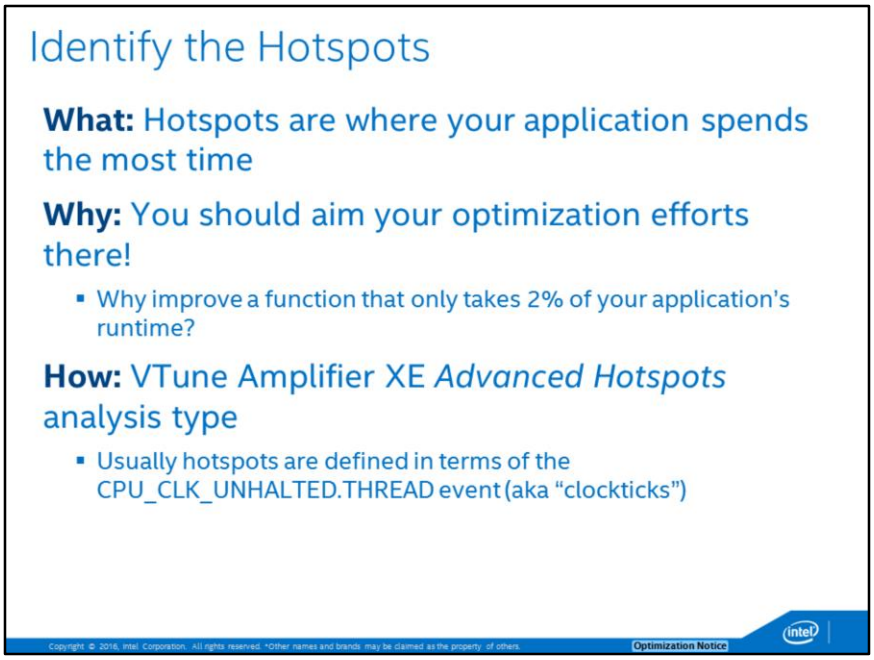
KNL supports 512 bit vector instructions. To optimize for KNL, an application should take advantage of these large vector units with heavily vectorized code. Look at the metric VPU Utilization to determine the areas of high and low vectorization in your application.

Vectorization Usage



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

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.



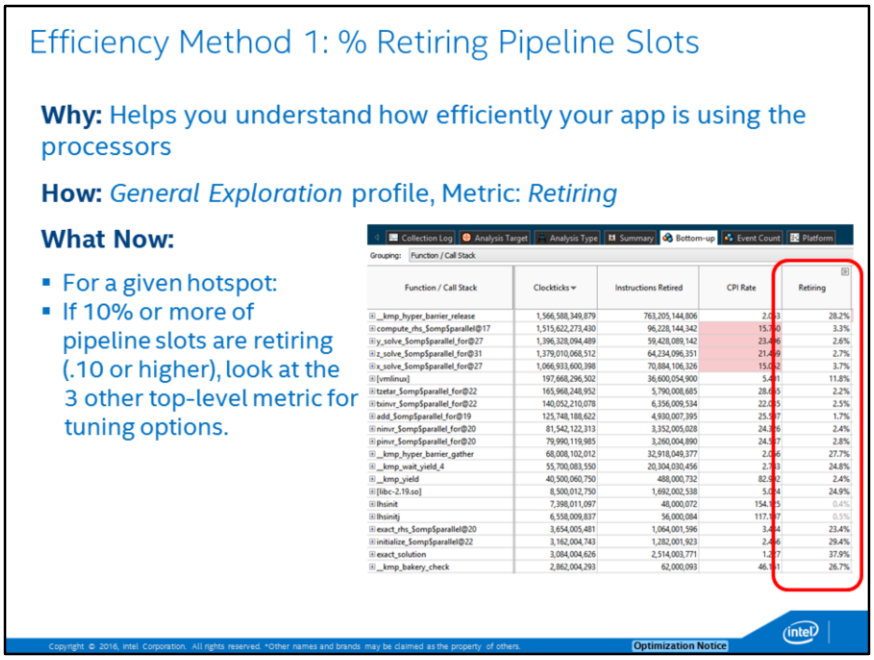
For this processor, the CPU_CLK_UNHALTED.THREAD counter measures unhalted clockticks on a per hardware thread basis. The CPU_CLK_UNHALTED.THREAD counter allows you to see where cycles are being spent on each individual hardware thread.
There is also a CPU_CLK_UNHALTED.REF counter, which counts unhalted 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

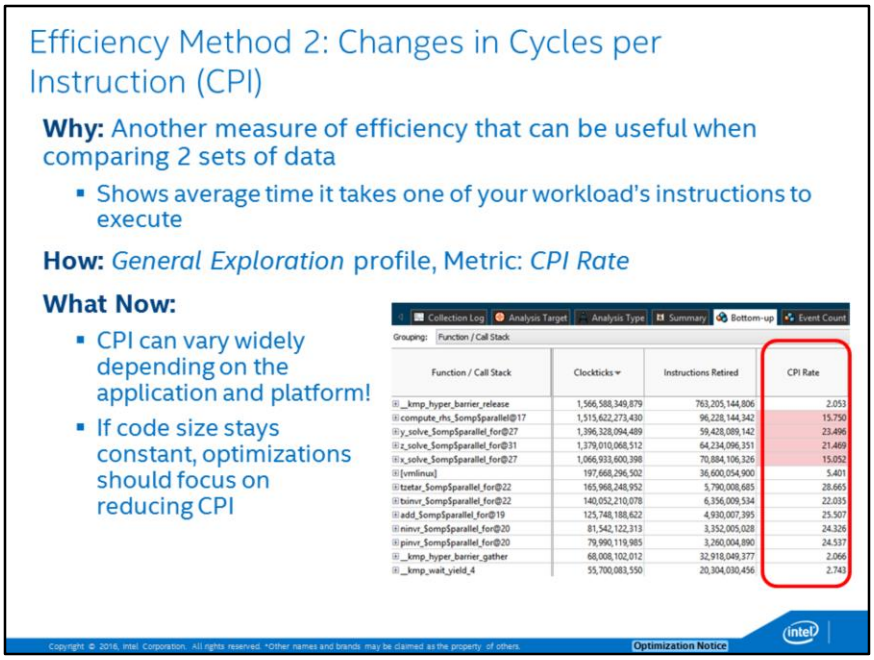
Repeat



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.



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.

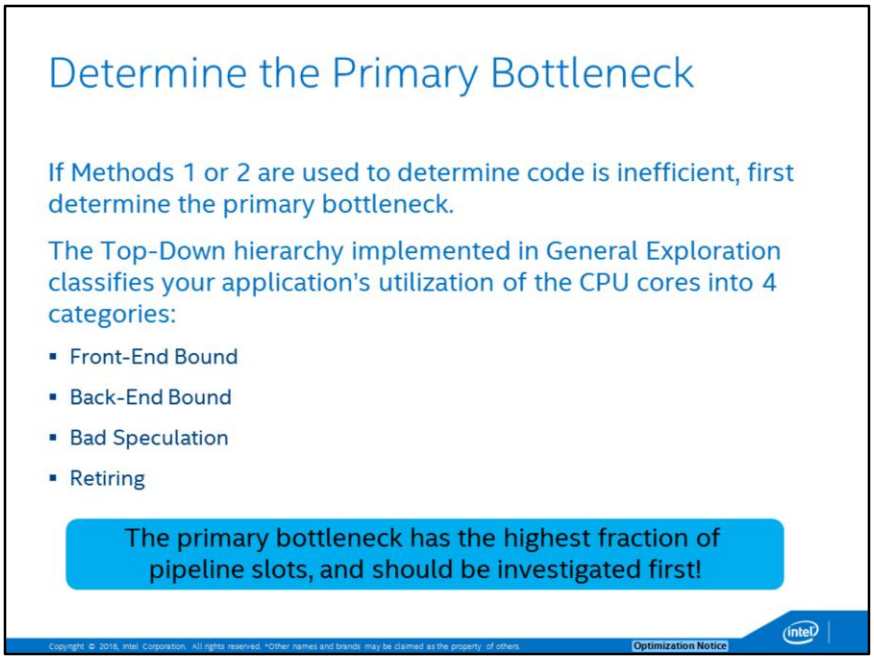
CPI is just a general efficiency metric – the real measure of efficiency is work taking less time.

The "Software on Hardware" Tuning Process

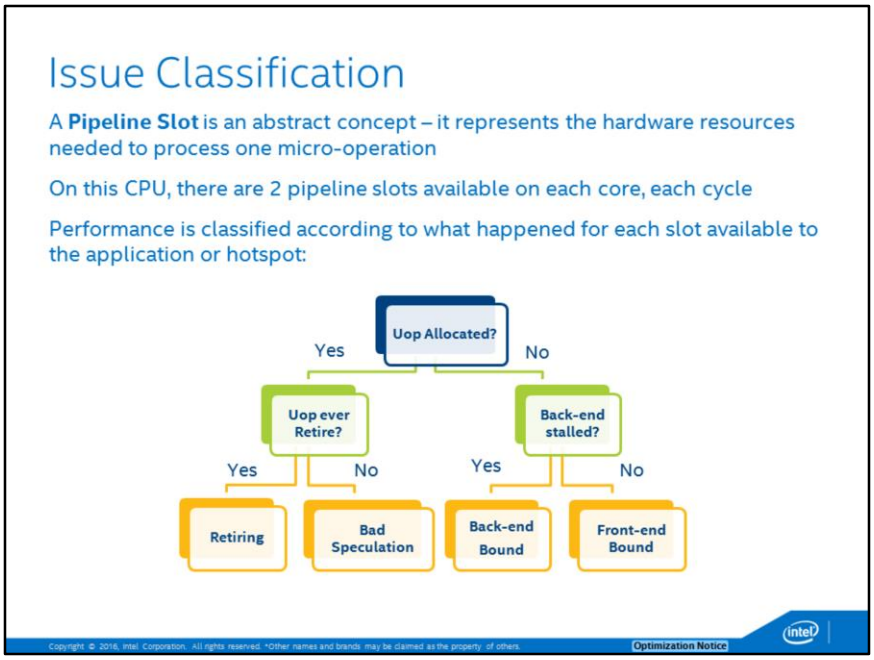
For each Hotspot

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

Repeat



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.



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.

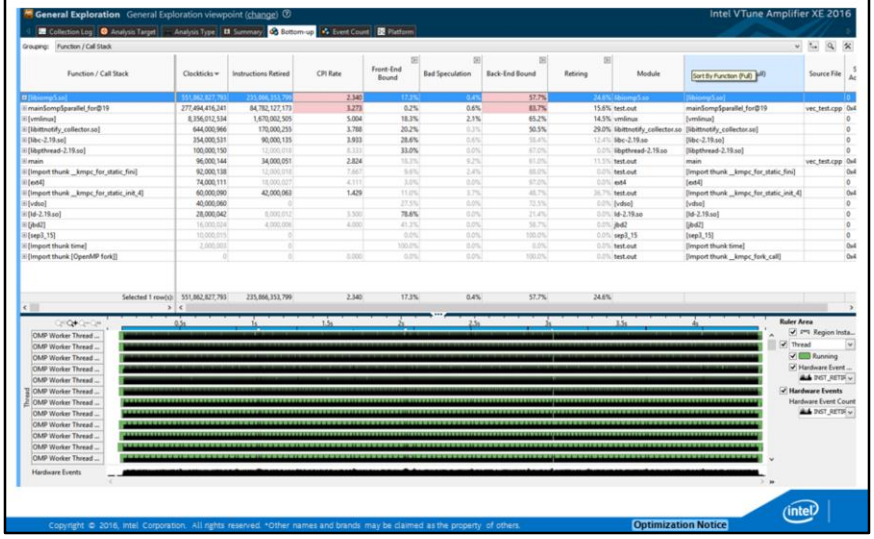
The "Software on Hardware" Tuning Process

For each Hotspot

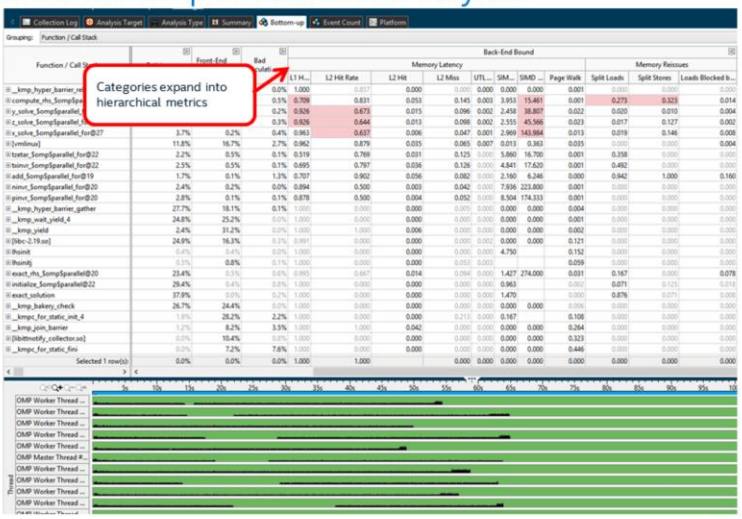
- Determine efficiency
  - If inefficient:
    - Determine primary bottleneck
    - Identify architectural reason for inefficiency
    - Optimize the issue

Repeat

General Exploration Analysis



General Exploration Analysis



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

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:

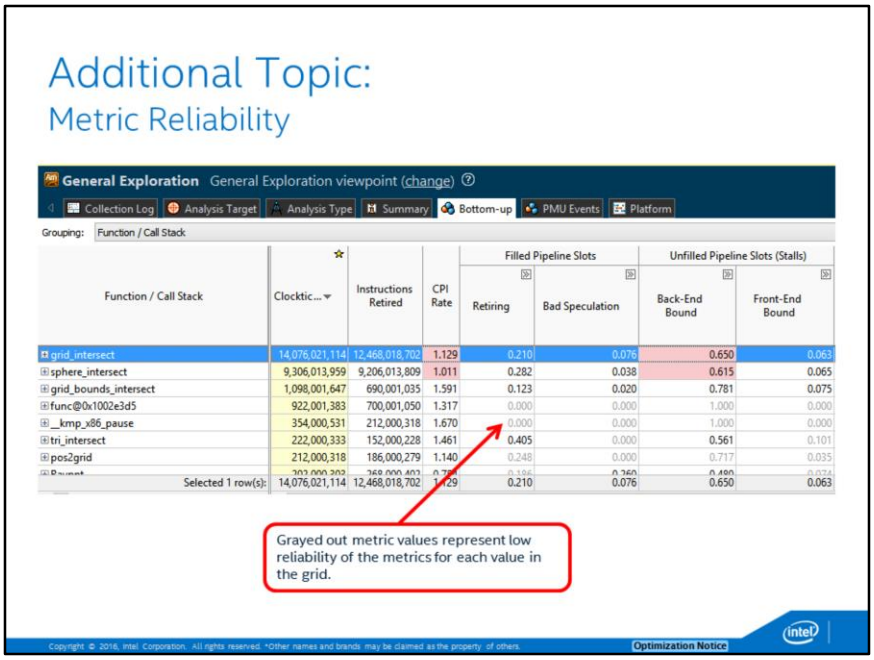
Certain floating point operations cannot be handled natively by the execution pipeline and must be performed by microcode (small programs injected into the execution stream). For example, when working with very small floating point values (so-called denormals), the floating-point units are not set up 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

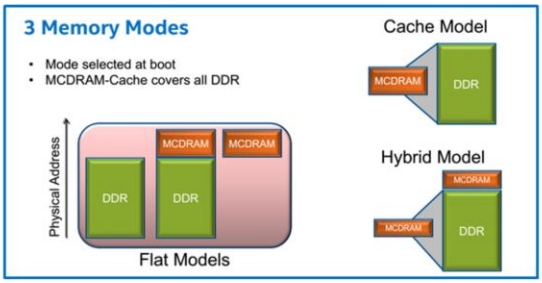


The General Exploration analysis type multiplexes hardware events during collection, which can result in imprecise results if too few samples are collected. The GUI will gray out metrics if the reliability is low based on the number of samples collected. If a metric is grayed out for your area of interest,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

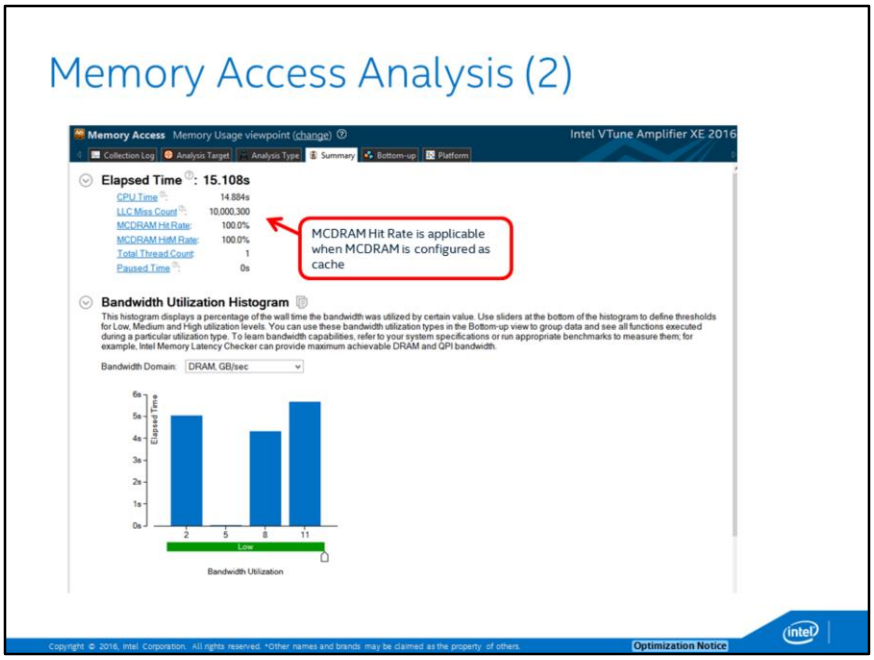
Memory Access Analysis

- 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.



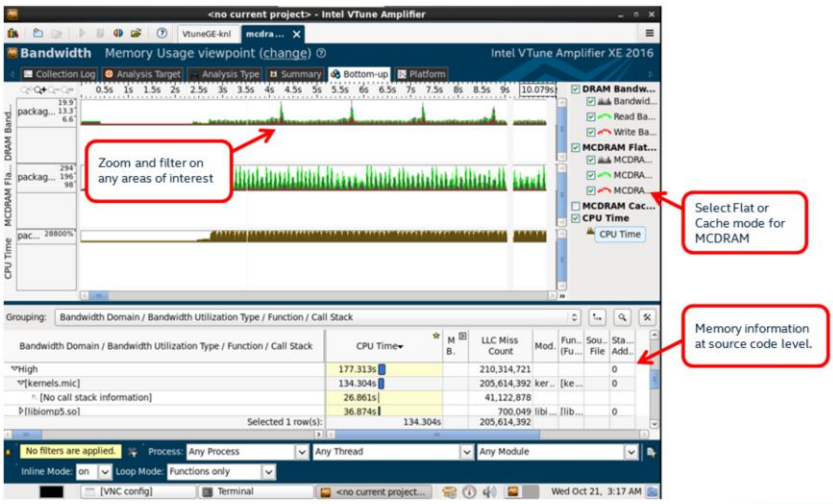
Memory Access Analysis

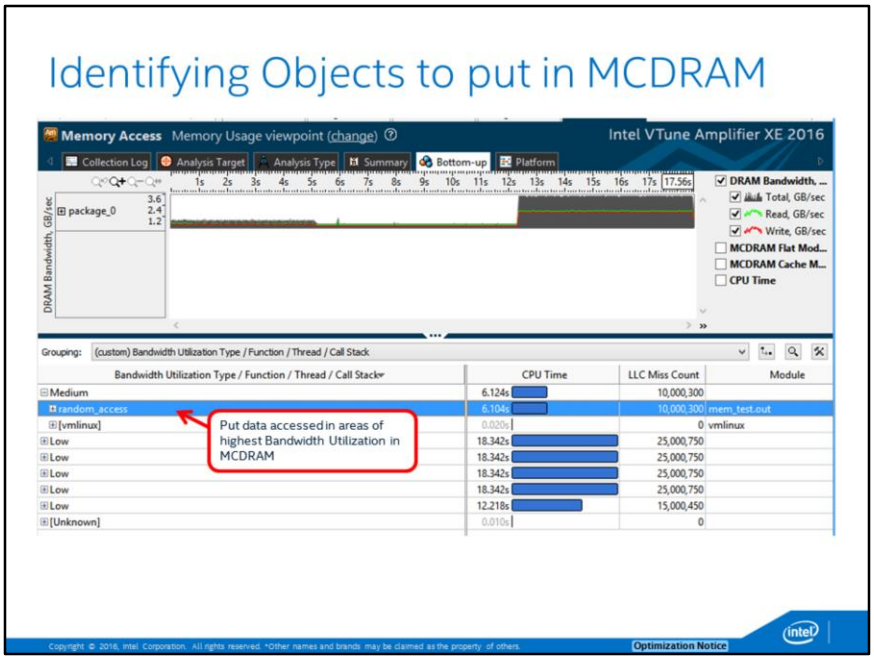
- Flat Mode
  - Allows the developers to explicitly control which data structures are in MCDRAM vs. DDR.
  - Requires code modification, otherwise DDR will be used by default.
- Cache Mode
  - No code modification. Hardware will use L1, then L2, then MCDRAM cache.
  - Data allocated into MCDRAM cache 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



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)

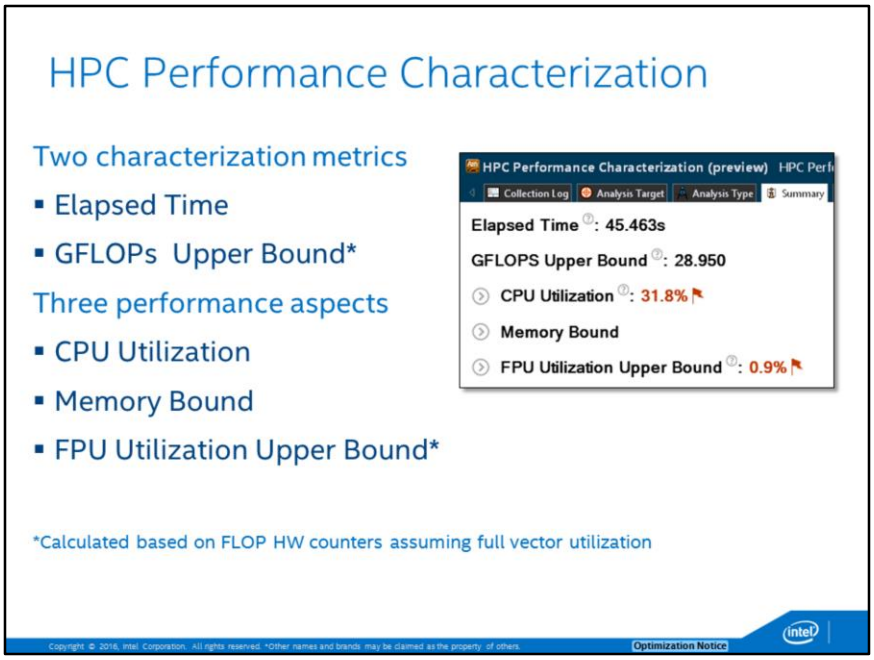




Run a Memory Access analysis with MCDRAM configures in flat mode and all allocations occurring in DDR (not using MCDRAM).
Create a custom grouping in the Memory Access analysis to see functions causing Medium or High bandwidth utilization. Objects accessed within these functions may be candidates to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

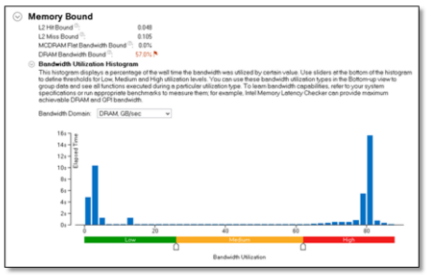
- Average CPU usage
- Additional MPI and OpenMP scalability metrics impacting effective CPU utilization
- CPU usage histogram

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

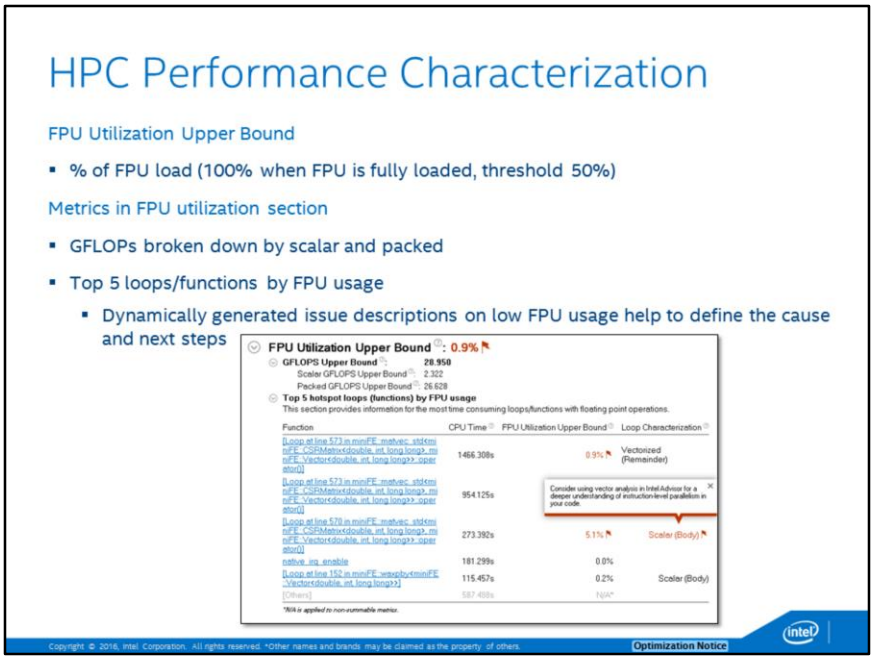
HPC Performance Characterization

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



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.



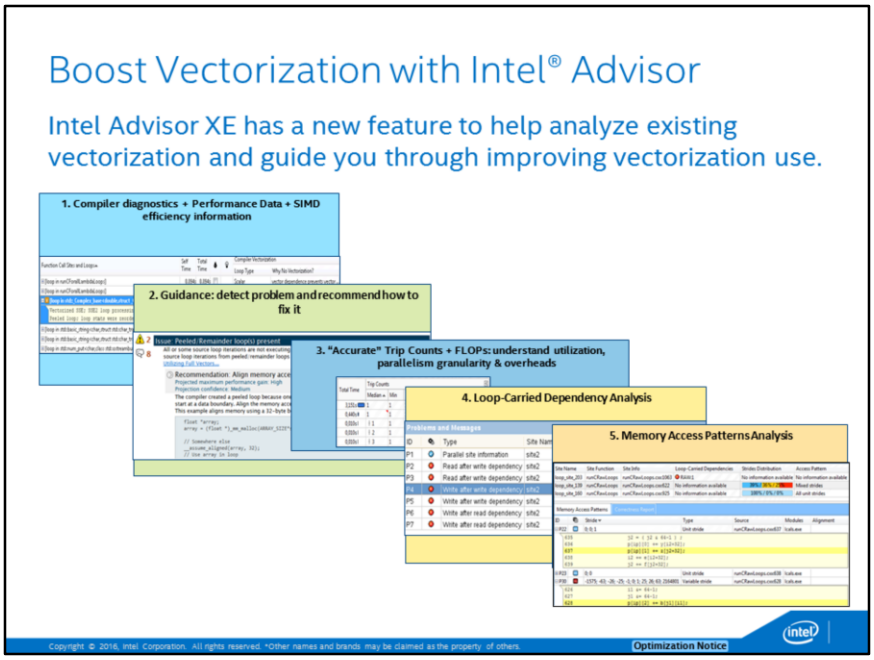
The Floating Point Units (FPU) on KNL are important for getting the best performance. The HPC Performance Characterization provides an upper bound estimate of the utilization. This is an upper bound because the events used are not able to account for masking, and the metric assumes all vector lanes are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

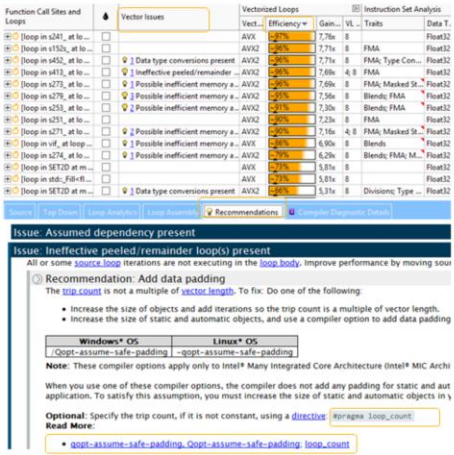
VTune Amplifier Tips (cont.)

- 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.



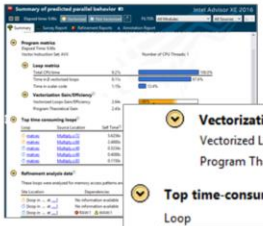
Use this 5 step process to determine how well you are vectorizing and where you can improve.
Intel® Advisor is available at: https://software.intel.com/en-us/intel-advisor-xe

Survey Analysis



- **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



What can I expect to gain?

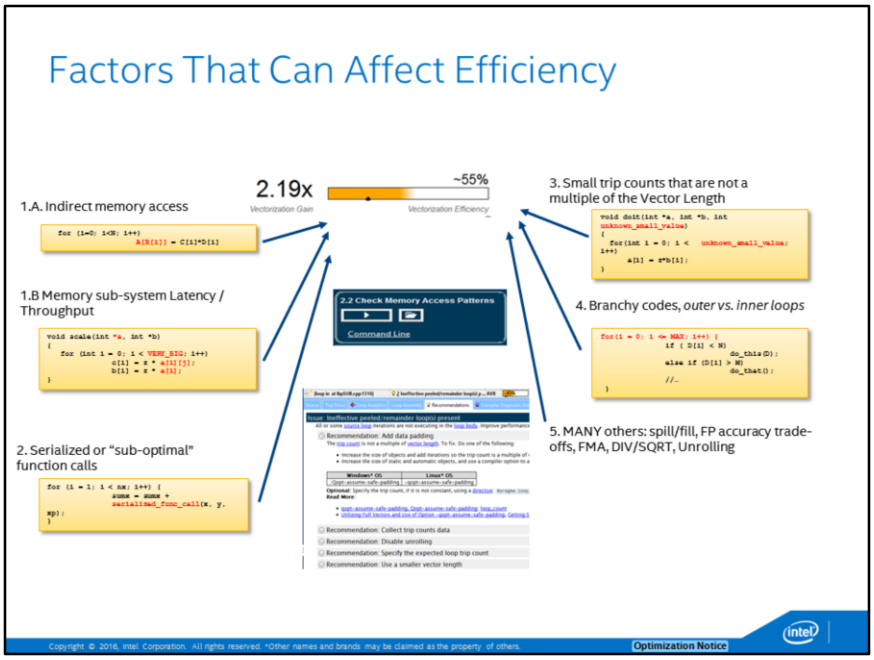
Vectorization Gain/Efficiency

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

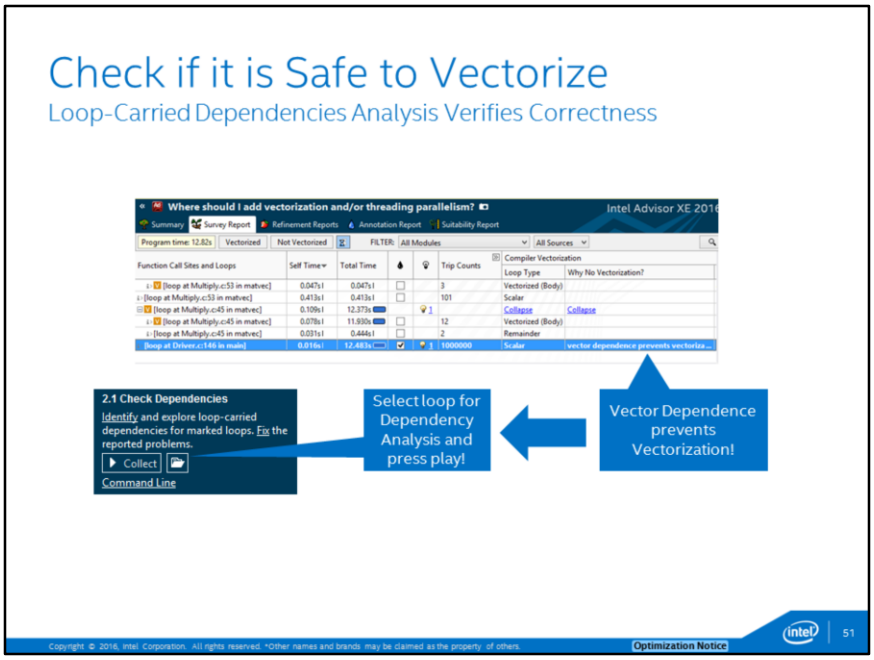
Top time-consuming loops

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	2.8[illegible]
matvec	Multiply.c:85	0.1150s	0.1150s

Where do I start?



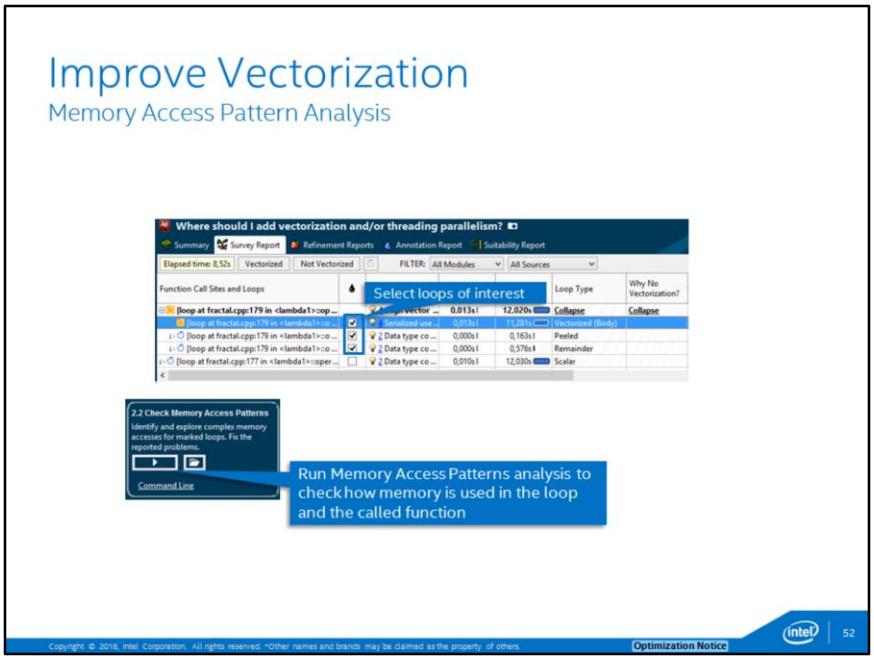
Analyze the hot loops for the common issues that can impact vectorization. Use the Memory Access Patterns Analysis and Recommendations to identify problematic behaviors and ways to correct them.



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.

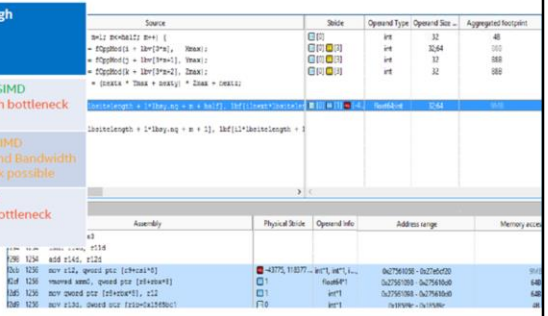


The strides of memory accesses can affect vectorization. Determine the patterns to learn which loops may be difficult to vectorize.

Memory Analysis Is Critical

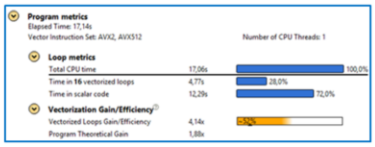
Determine Possible Bandwidth or Latency Issues

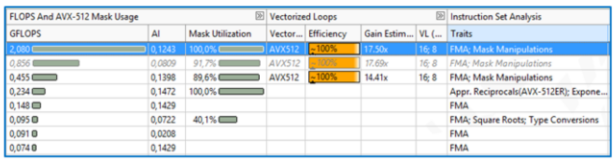
Footprint / Access Pattern	Small enough	Big enough
Unit Stride	Effective SIMD No Latency and BW bottlenecks	Effective SIMD Bandwidth bottleneck
Const stride	Medium SIMD Latency bottleneck possible	Medium SIMD Latency and Bandwidth bottleneck possible
Irregular Access, Gather/Scatter	Bad SIMD Latency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



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**

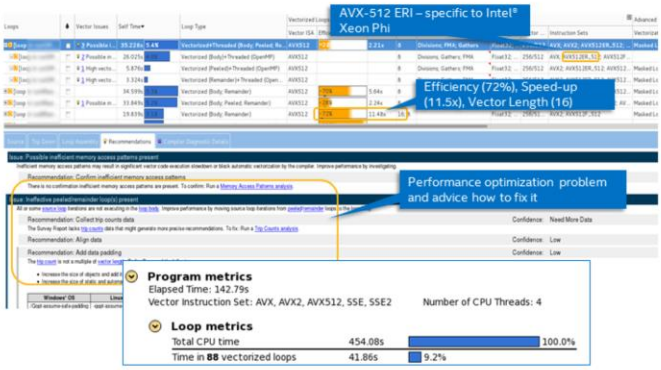




FLOPS And AVX-512 Mask Usage			Vectorized Loops				Instruction Set Analysis
GFLOPS	AI	Mask Utilization	Vector...	Efficiency	Gain Estim...	VL (...	Traits
2,080	0,1243	100,0%	AVX512	100%	17.50x	16; 8	FMA; Mask Manipulations
0,856	0,0809	91,7%	AVX512	100%	17.69x	16; 8	FMA; Mask Manipulations
0,455	0,1398	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.



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

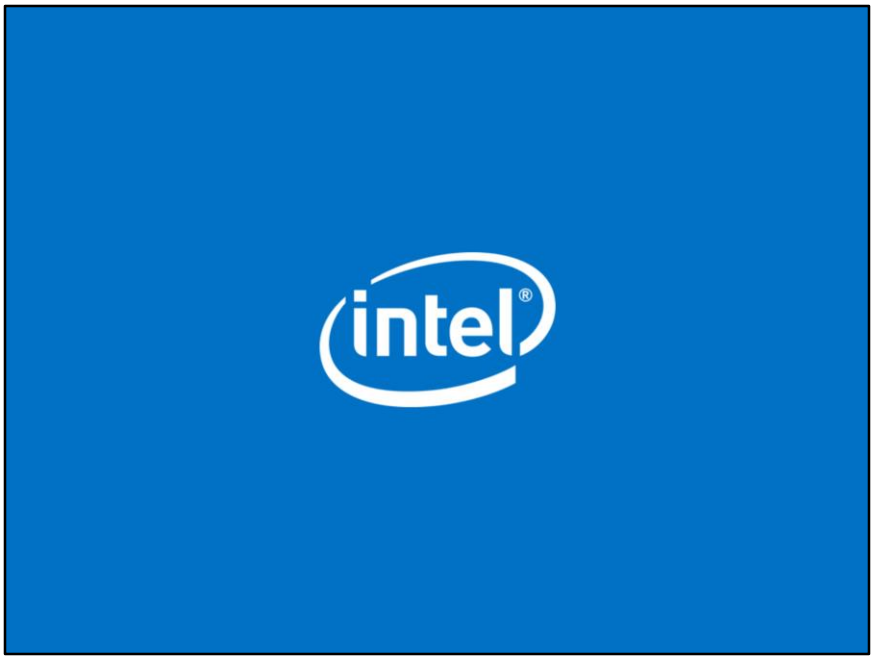
VTune Amplifier XE Tuning Guides for Other microarchitectures:
http://software.intel.com/en-us/articles/processor-specific-performance-analysis-papers

Legal Disclaimer

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804


intel