OPTIMIZATION WORKSHOP

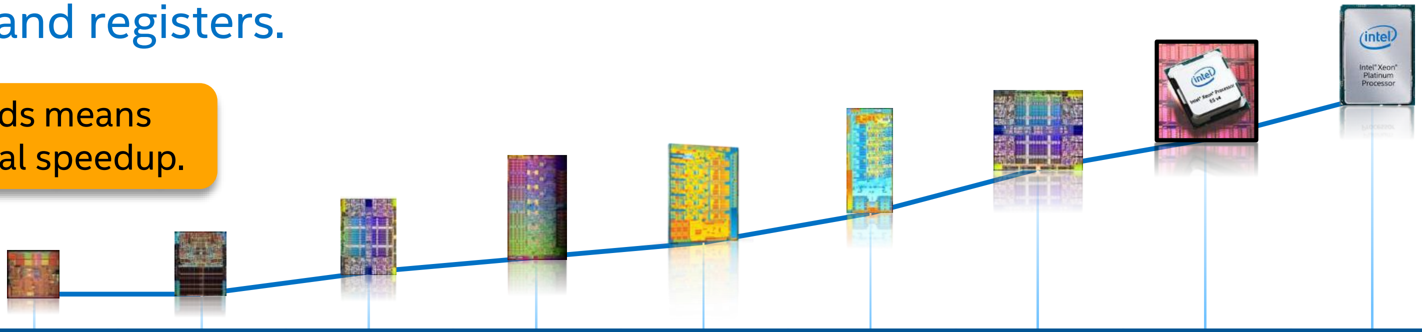
Intel® VTune™ Amplifier and Intel® Advisor

Kevin O'Leary, Technical Consulting Engineer

Changing Hardware Affects Software Development

More cores and wider vector registers mean more threads and more maximum performance! … but you need to need to write software that takes advantage of those cores and registers.

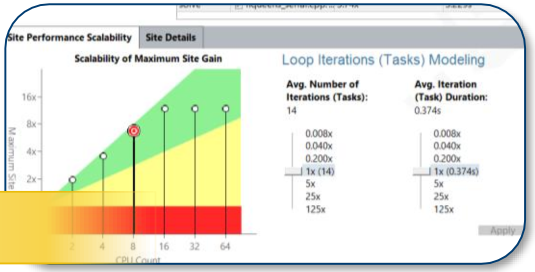
More threads means more potential speedup.



Intel® Xeon® Processor	64-bit	5100 series	5500 series	5600 series	E5-2600	E5-2600 V2	E5-2600 V3	E5-2600 V4	Platinum 8180
Cores	1	2	4	6	8	12	18	22	28
Threads	2	2	8	12	16	24	36	44	56
SIMD Width	128	128	128	128	256	256	256	256	512

The Agenda

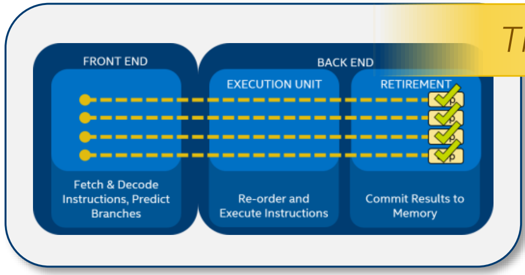
Optimization 101


Site Performance Scalability
Site Details
Scalability of Maximum Site Gain
16x
8x
4x
2x
Maximum Site
2
4
8
16
32
64
CPU Count
Apply

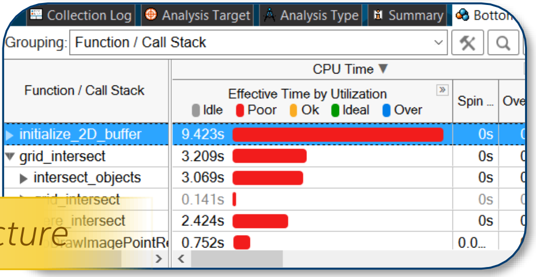
Loop Iterations (Tasks) Modeling

Avg. Number of Iterations (Tasks):	Avg. Iteration (Task) Duration:
14	0.374s
0.008x	0.008x
0.040x	0.040x
0.200x	0.200x
1x (14)	1x (0.374s)
5x	5x
25x	25x
125x	125x

Threading


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

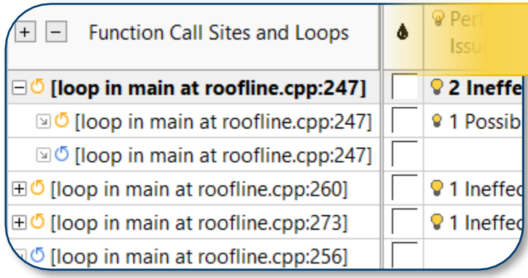
The uOp Pipeline



Grouping: Function / Call Stack

Function / Call Stack	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Spin ...	Ove
initialize_2D_buffer	9.423s	0s	0
grid_intersect	3.209s	0s	0
intersect_objects	3.069s	0s	0
grid_intersect	0.141s	0s	0
...re_intersect	2.424s	0s	0
...awImagePointR	0.752s	0.0...	

Tuning to the Architecture



Function Call Sites and Loops		Perf Issu
[loop in main at roofline.cpp:247]		2 Ineffe
[loop in main at roofline.cpp:247]		1 Possib
[loop in main at roofline.cpp:247]		
[loop in main at roofline.cpp:260]		1 Ineffec
[loop in main at roofline.cpp:273]		1 Ineffec
[loop in main at roofline.cpp:256]		

Vectorization

Q & A

Optimization 101

Take advantage of compiler optimizations with the right flags.

Linux*	Windows*	Description
-xCORE-AVX512	/QxCORE-AVX512	Optimize for Intel® Xeon® Scalable processors, including AVX-512.
-xCOMMON-AVX512	/QxCOMMON-AVX512	Alternative, if the above does not produce expected speedup.
-fma	/Qfma	Enables fused multiply-add instructions. *(Warning: affects rounding!)*
-O2	/O2	Optimize for speed (enabled by default).
-g	*/Zi*	*Generate debug information for use in performance profiling tools.*

Use optimized libraries, like Intel® Math Kernel Library (MKL).

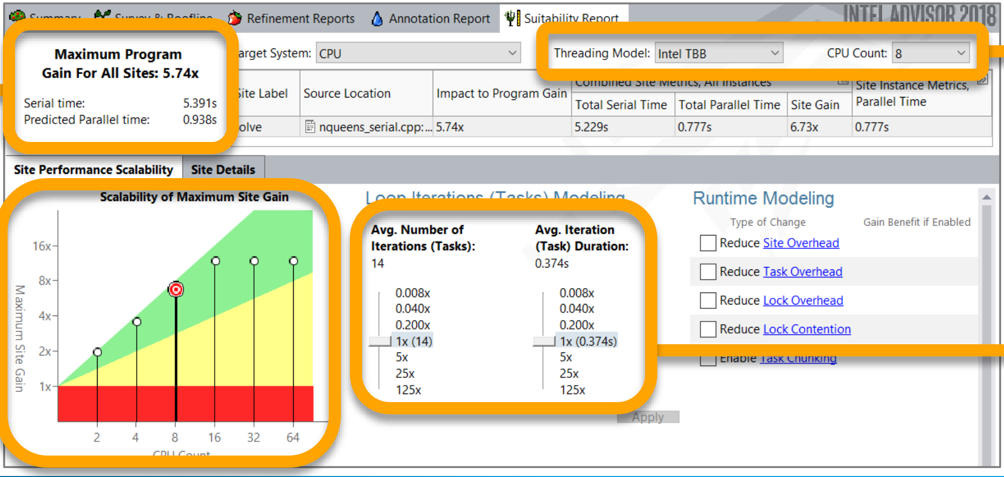
Linear Algebra	Fast Fourier Transforms	Vector Math	Summary Statistics	Deep Neural Networks	And More...
• BLAS • LAPACK • ScaLAPACK • Sparse BLAS • Sparse Solvers • Iterative • PARDISO* • Cluster Sparse Solver	• Multidimensional • FFTW interfaces • Cluster FFT	• Trigonometric • Hyperbolic • Exponential • Log • Power • Root • Vector RNGs	• Kurtosis • Variation coefficient • Order statistics • Min/max • Variance-covariance	• Convolution • Pooling • Normalization • ReLU • Softmax	• Splines • Interpolation • Trust Region • Fast Poisson Solver

Adding Threading with Intel® Advisor

- Find good threading sites with the **Survey** analysis, then annotate the code to tell Advisor how to simulate threading and locking.
- Use the **Suitability** analysis to predict threading performance and the **Dependencies** analysis to check for correctness problems.

Predicted program speedup

See how each parallel site would scale on a given number of CPUs.



Maximum Program Gain For All Sites: 5.74x

Serial time:	5.391s
Predicted Parallel time:	0.938s

Target System: CPU

Threading Model: Intel TBB   CPU Count: 8

Site Label	Source Location	Impact to Program Gain	Combined Site Metrics, All Instances			Site Instance Metrics, Parallel Time
			Total Serial Time	Total Parallel Time	Site Gain	
olve	nqueens_serial.cpp:...	5.74x	5.229s	0.777s	6.73x	0.777s

Loop Iterations (Tasks) Modeling

Avg. Number of Iterations (Tasks):	Avg. Iteration (Task) Duration:
14	0.374s
0.008x	0.008x
0.040x	0.040x
0.200x	0.200x
1x (14)	1x (0.374s)
5x	5x
25x	25x
125x	125x

Runtime Modeling

Type of Change	Gain Benefit if Enabled
☐ Reduce Site Overhead	
☐ Reduce Task Overhead	
☐ Reduce Lock Overhead	
☐ Reduce Lock Contention	
☐ Enable Task Chunking	

Set hypothetical environment details to see effects.

Experiment with what would happen if you changed the number or duration of parallel tasks without re-running the analysis.

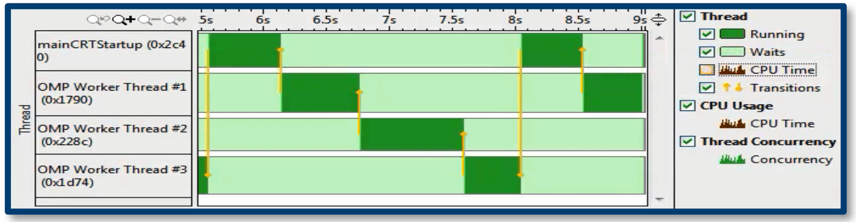
Using Intel® VTune™ Amplifier for Threading Optimization

Use **Threading** analysis to see how well your program is using its threads.

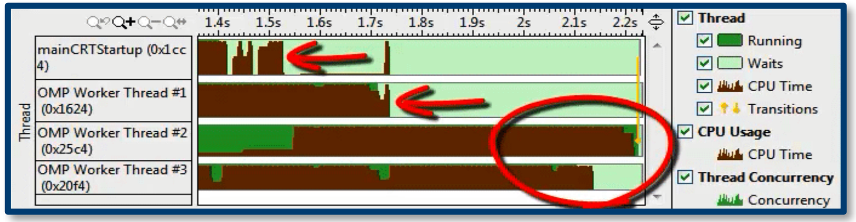
Each thread is displayed on the timeline, with color coded activity.

- Coarse-grain locks indicate that your program is effectively single threaded.
- Thread imbalance is when the application isn't using all the threads all the time.
- Lock contention means your program is spending more time swapping threads out than actually working.

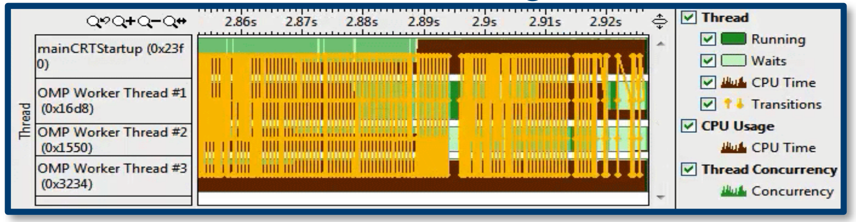
Coarse-Grain Locks



Thread Imbalance

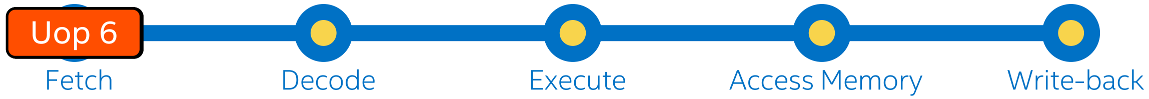


High Lock Contention



What is the uop Pipeline?

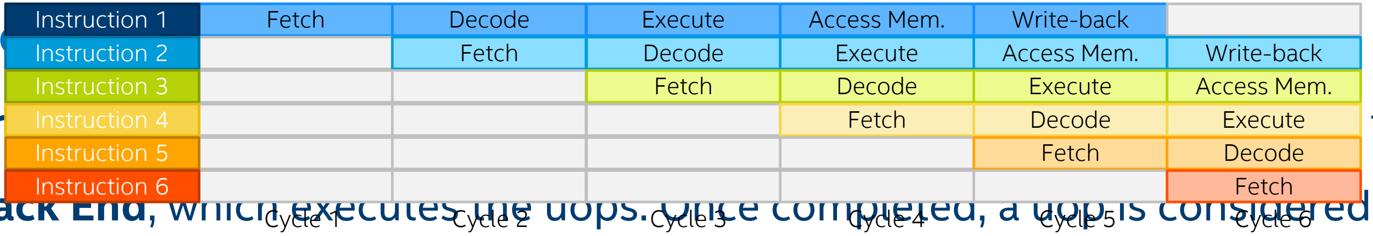
There are multiple steps to executing an instruction.



Modern CPUs **pipeline** instructions rather than performing all the steps for one instruction before beginning the next instruction.

The pipe

- The **Fr** to…
- The **Back End**, which executes the uops. Once completed, a uop is considered "retired."



Instruction 1	Fetch	Decode	Execute	Access Mem.	Write-back	
Instruction 2		Fetch	Decode	Execute	Access Mem.	Write-back
Instruction 3			Fetch	Decode	Execute	Access Mem.
Instruction 4				Fetch	Decode	Execute
Instruction 5					Fetch	Decode
Instruction 6						Fetch
	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6

A **Pipeline Slot** is a representation of the hardware needed to process a uop.

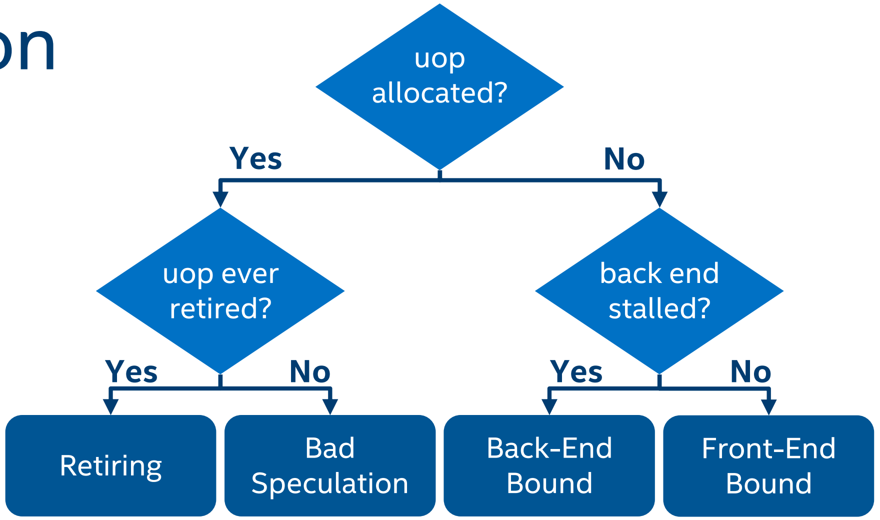
The Front End can only allocate (and the Back End retire) so many uops per cycle. This determines the number of Pipeline Slots. In general, there are four.

Pipeline Slot Categorization

Pipeline slots can be sorted into four categories on each cycle.

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

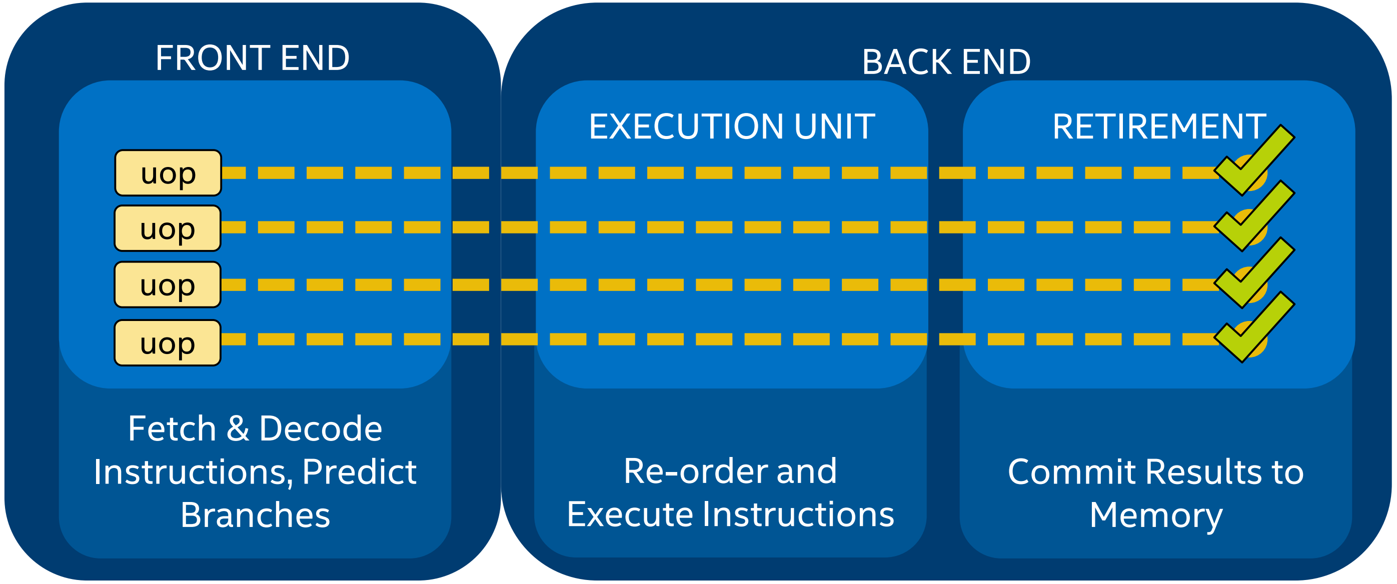
Each category has an expected range of values in a well tuned application.



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

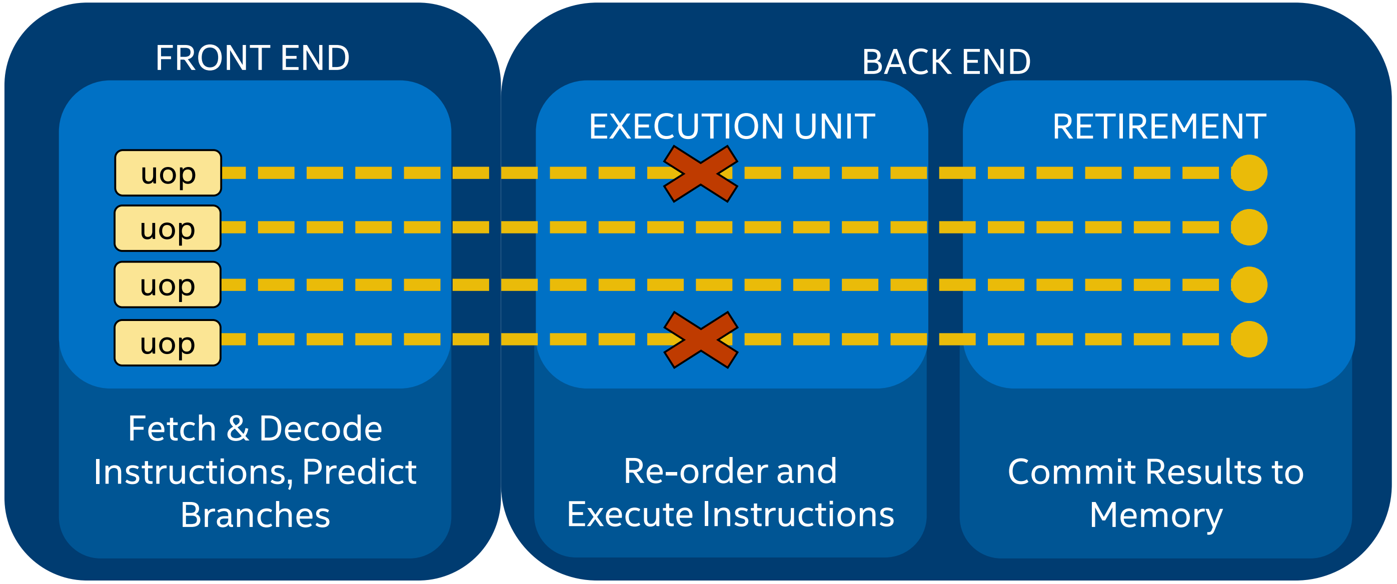
Pipeline Slot Categorization: Retiring

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


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

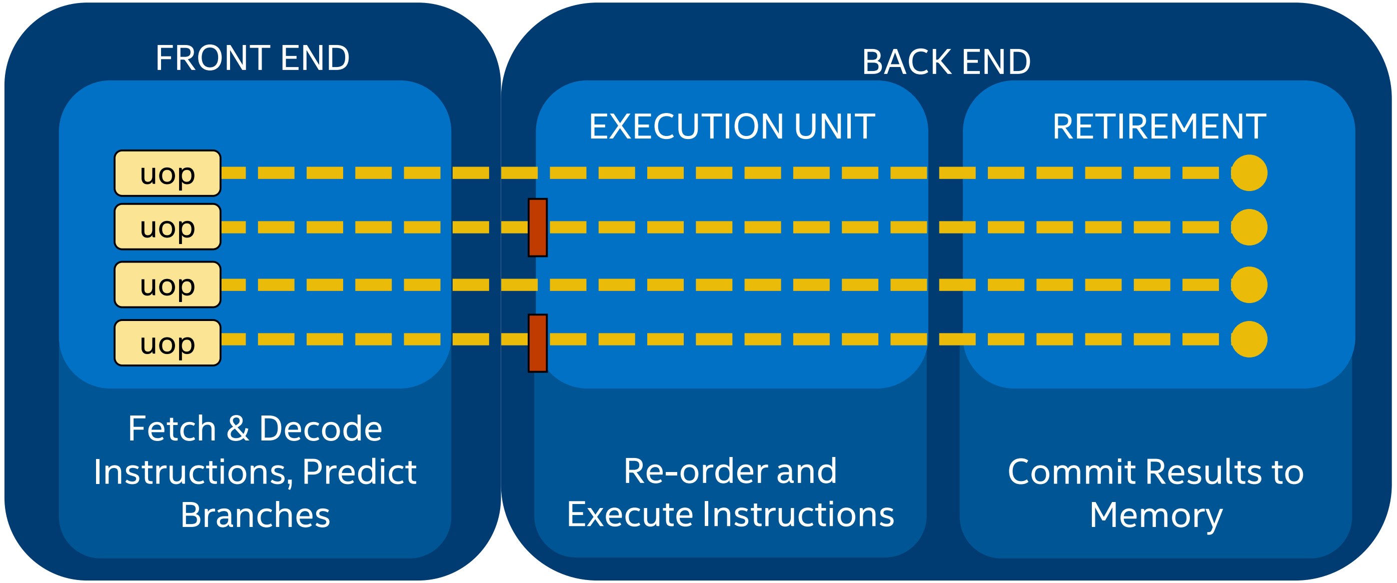
Pipeline Slot Categorization: Bad Speculation

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


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

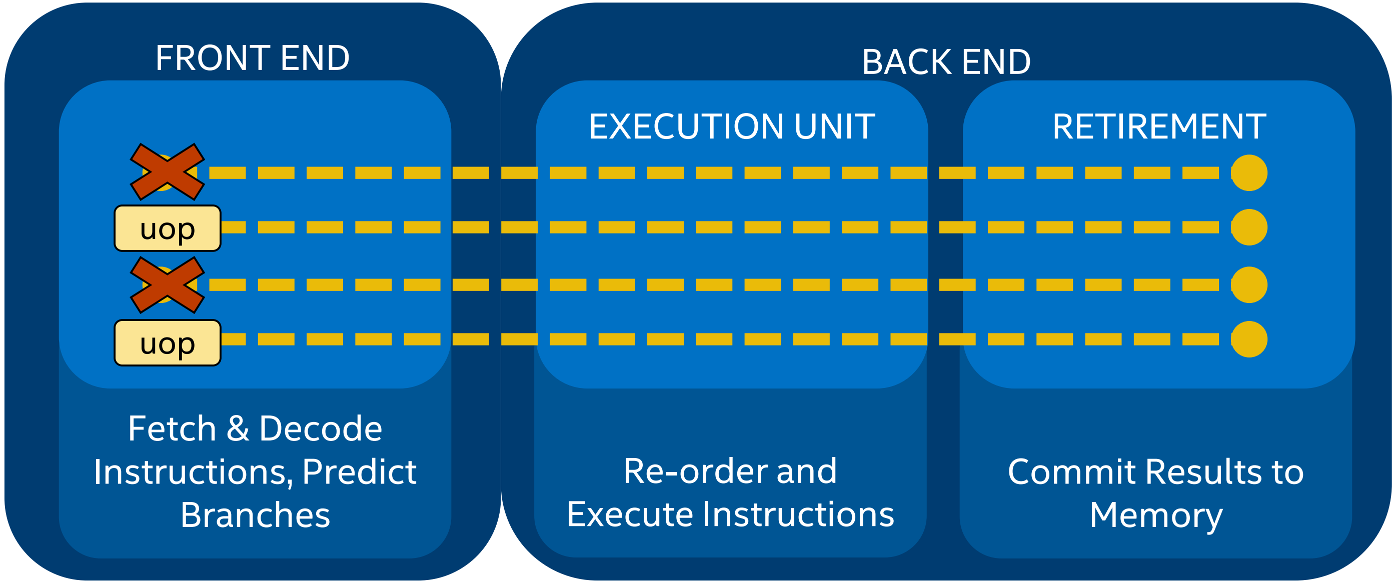
Pipeline Slot Categorization: Back End Bound

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


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

Pipeline Slot Categorization: Front End Bound

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


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

The Tuning Process

Find Hotspots

Intel® VTune™ Amplifier

Hotspots Analysis

For Each Hotspot

Determine Efficiency

Intel® VTune™ Amplifier

Microarchitecture Exploration

If Inefficient:

Diagnose Bottleneck

Intel® VTune™ Amplifier

Microarchitecture Exploration

Memory Access Analysis

Intel® Advisor

Vectorization Advisor

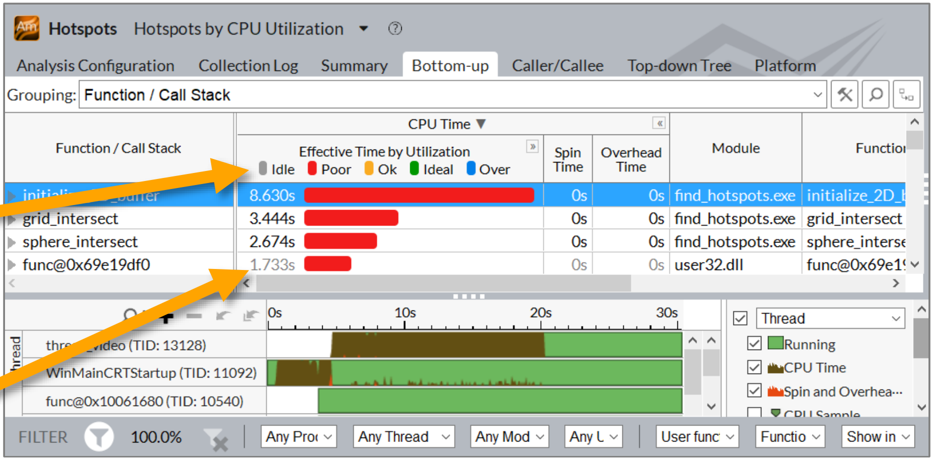
Implement Solution

Finding Hotspots

Find Hotspots
Determine Efficiency
Diagnose Bottleneck
Implement Solution

Use **Hotspots** analysis. Find where your program is spending the most time to ensure optimizations have a bigger impact.

- The Summary tab shows a high-level overview of the most important data.
- The Bottom-up tab provides more detailed analysis results.
  - The total amount of time spent in a function is divided up by how many CPUs were active during the time the function was running.
  - Low confidence metrics are grayed out: VTune uses statistical sampling and may miss extremely small, fast portions of the program.



Function / Call Stack	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	CPU Time: Spin Time	CPU Time: Overhead Time	Module	Functio
initialize_2D_buffer	8.630s	0s	0s	find_hotspots.exe	initialize_2D_b
grid_intersect	3.444s	0s	0s	find_hotspots.exe	grid_intersect
sphere_intersect	2.674s	0s	0s	find_hotspots.exe	sphere_interse
func@0x69e19df0	1.733s	0s	0s	user32.dll	func@0x69e1

Thread:
thread_video (TID: 13128)
WinMainCRTStartup (TID: 11092)
func@0x10061680 (TID: 10540)

Determining Efficiency

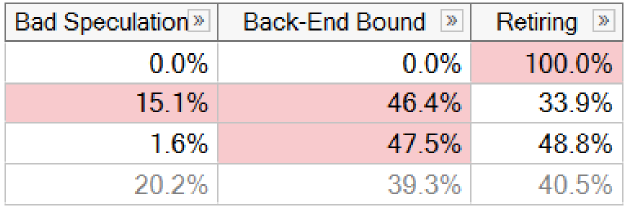
Find Hotspots → **Determine Efficiency** → Diagnose Bottleneck → Implement Solution

Use **Microarchitecture Exploration** analysis. It's preconfigured with:

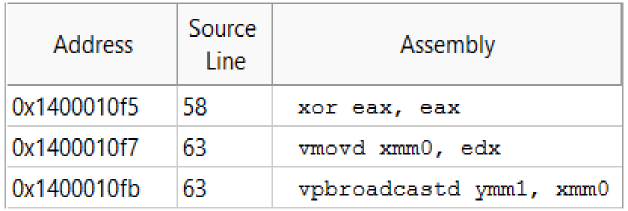
- appropriate events and metric formulae for the architecture
- hardware-specific thresholds for highlighting potential problems in pink

Inefficiency can be caused by:

- Not retiring enough necessary instructions.
  - Look for retiring rate lower than expected value.
- Retiring too many unnecessary instructions.
  - Look for underuse of AVX or FMA instructions.



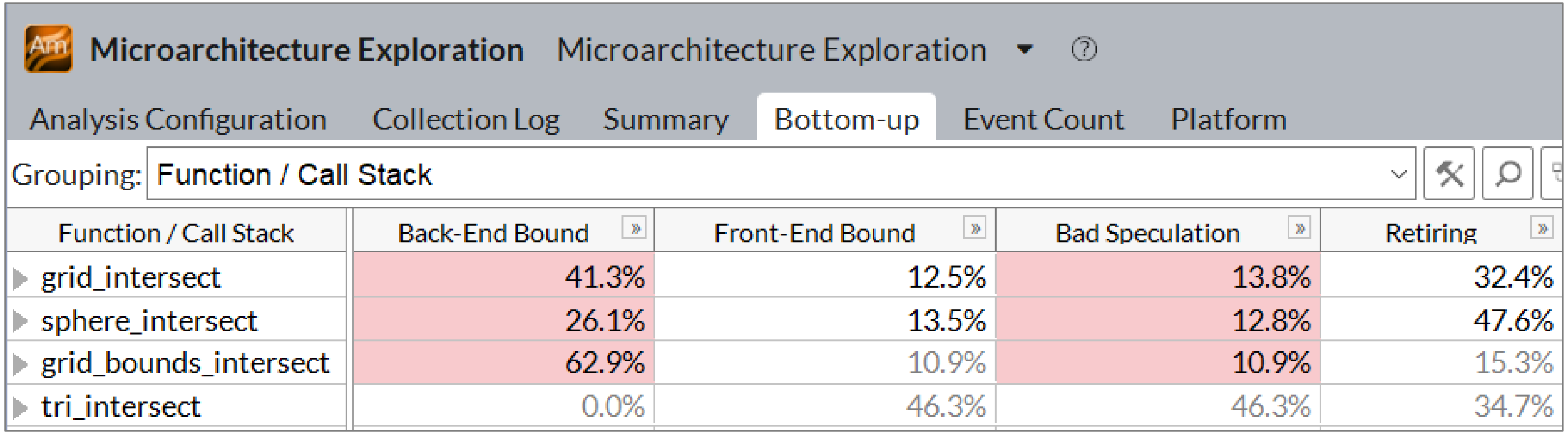
Bad Speculation	Back-End Bound	Retiring
0.0%	0.0%	100.0%
15.1%	46.4%	33.9%
1.6%	47.5%	48.8%
20.2%	39.3%	40.5%



Address	Source Line	Assembly
0x1400010f5	58	`xor eax, eax`
0x1400010f7	63	`vmovd xmm0, edx`
0x1400010fb	63	`vpbroadcastd ymm1, xmm0`

Diagnosing the Bottleneck

Intel® VTune™ Amplifier has hierarchical expanding metrics categorized by the four slot types. You can follow the pink highlights down the subcategories to identify the root cause. You can hover over a column to see a helpful tooltip.



Grouping: Function / Call Stack

Function / Call Stack	Back-End Bound	Front-End Bound	Bad Speculation	Retiring
grid_intersect	41.3%	12.5%	13.8%	32.4%
sphere_intersect	26.1%	13.5%	12.8%	47.6%
grid_bounds_intersect	62.9%	10.9%	10.9%	15.3%
tri_intersect	0.0%	46.3%	46.3%	34.7%

We can't cover all solutions today, but there's more information in the Tuning Guides: https://software.intel.com/en-us/articles/processor-specific-performance-analysis-papers

Solutions Sampler

Find Hotspots → Determine Efficiency → Diagnose Bottleneck → Implement Solution

Back End Bound

Core Bound

Divider

- Use reciprocal-multiplication where possible.

Memory Bound

Contested Access/Data Sharing

- Solve false sharing by padding variables to cache line boundaries.
- Try to reduce actual sharing requirements.

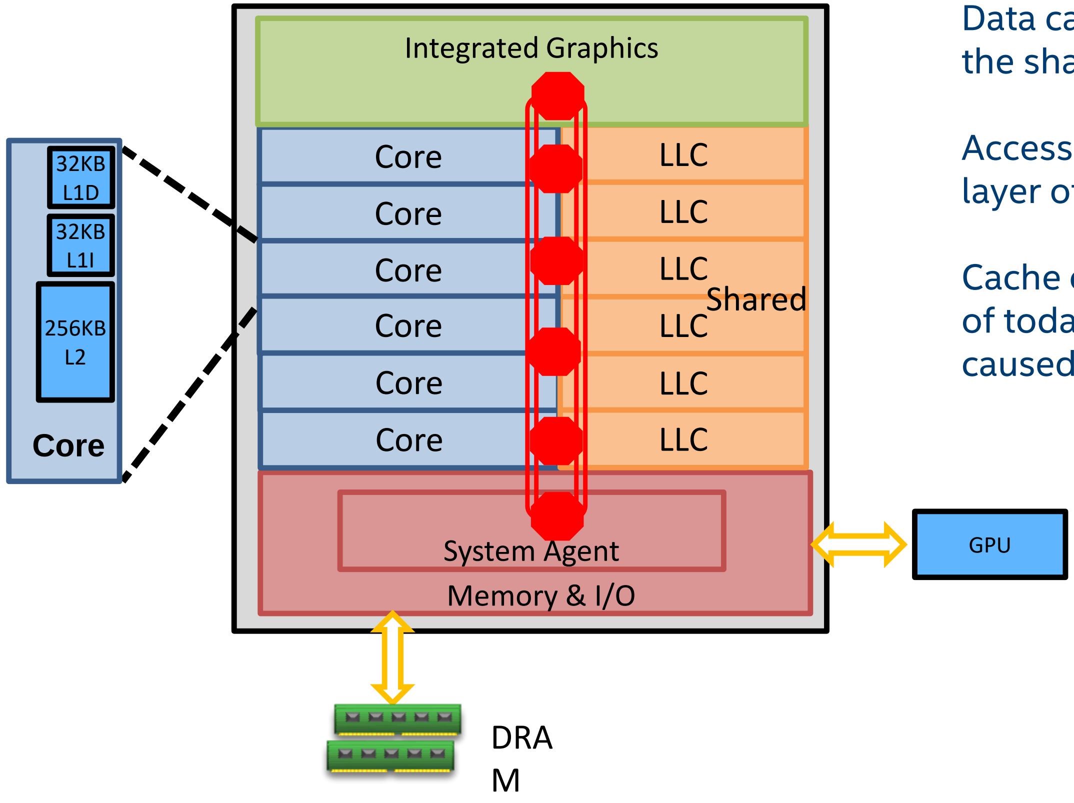
Remote Memory Access

- Affinitize/pin threads to cores.
- Use NUMA-efficient thread schedulers like Intel® Threading Building Blocks.
- Test whether performance improves using Sub-NUMA Cluster Mode.

Cache Misses

- Block your data.
- Use software prefetches.
- Consider Intel® Optane™ DC Persistent Memory.

Understanding the Memory Hierarchy



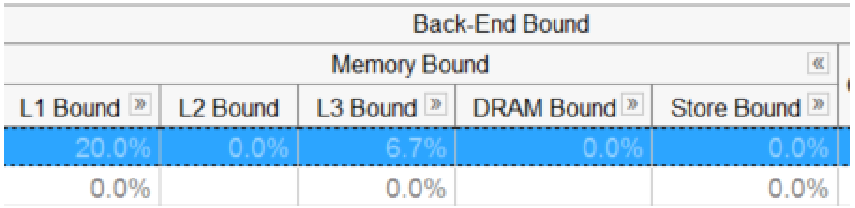
Data can be in any level of any core's cache, or in the shared L3, DRAM, or on disk.

Accessing data from another core adds another layer of complexity

Cache coherence protocols – beyond the scope of today's lecture. But we will cover some issues caused by this.

Cache Misses

Why: Cache misses raise the CPI of an application. Focus on long-latency data accesses coming from 2nd and 3rd level misses



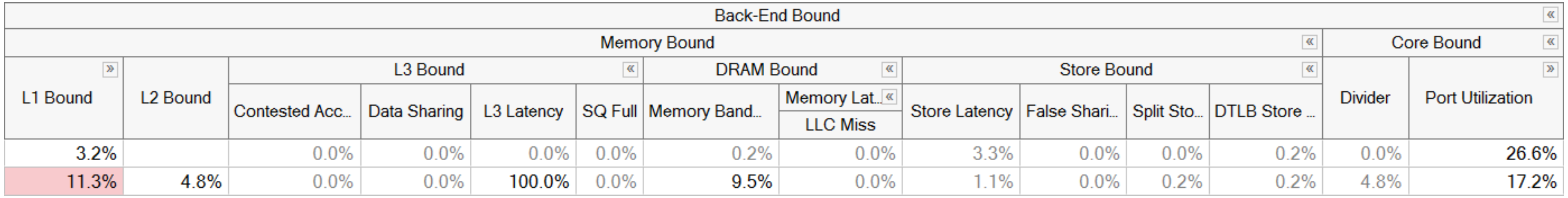
Back-End Bound				
Memory Bound				
L1 Bound	L2 Bound	L3 Bound	DRAM Bound	Store Bound
20.0%	0.0%	6.7%	0.0%	0.0%
0.0%		0.0%		0.0%

"<memory level> Bound" = Percentage of cycles when the CPU is stalled, waiting for data to come back from <memory level>

What Now: If either metric is highlighted for your hotspot, consider reducing misses:

- Change your algorithm to reduce data storage
- Block data accesses to fit into cache
- Check for sharing issues (See Contested Accesses)
- Align data for vectorization (and tell your compiler)
- Use streaming stores
- Use software prefetch instructions

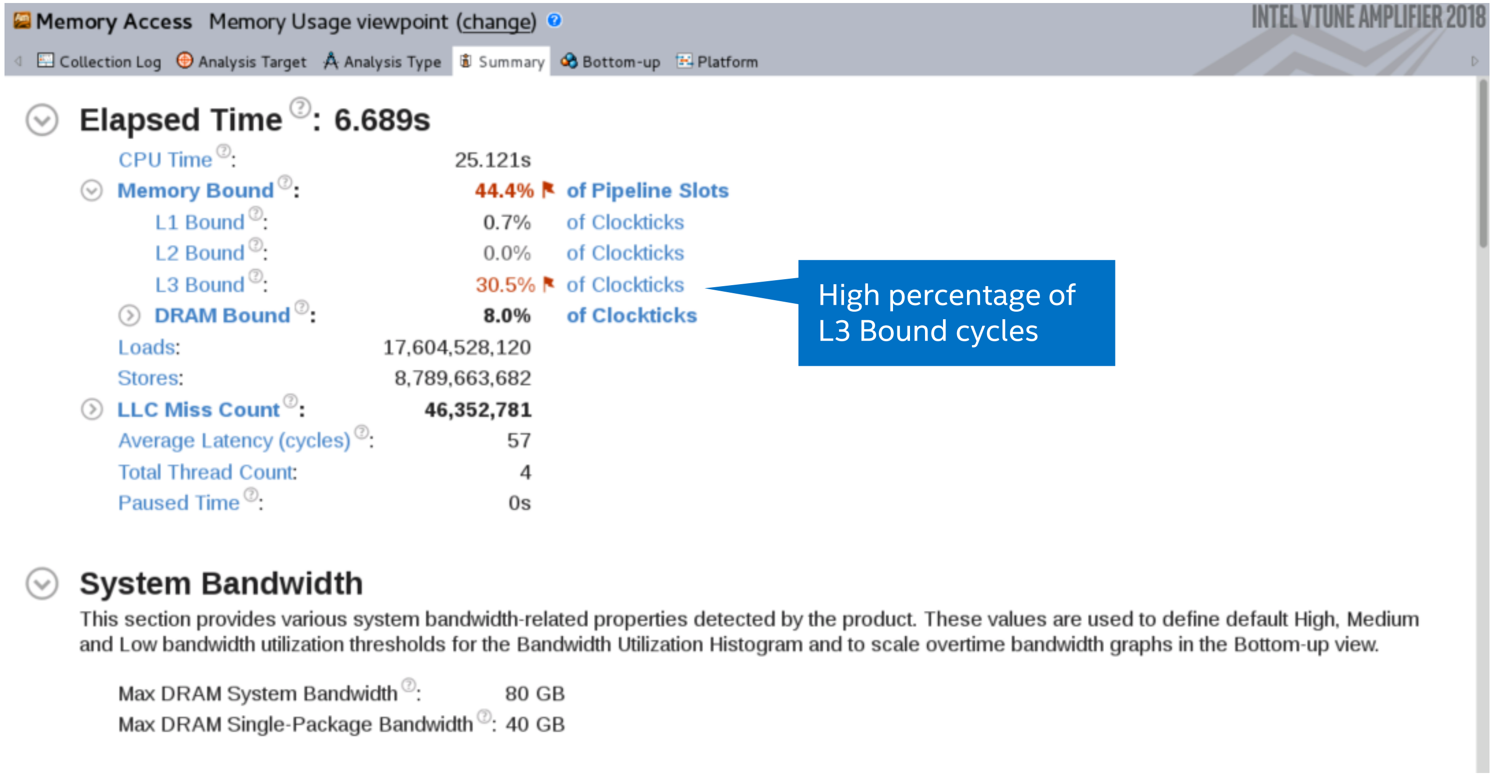
Categorizing Inefficiencies in the Memory Subsystem



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

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

VTune Amplifier Workflow Example- Summary View


Memory Access Memory Usage viewpoint (change)
INTEL VTUNE AMPLIFIER 2018
Collection Log
Analysis Target
Analysis Type
Summary
Bottom-up
Platform

Elapsed Time: 6.689s

CPU Time:	25.121s	
Memory Bound:	44.4%	**of Pipeline Slots**
L1 Bound:	0.7%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	30.5%	of Clockticks
DRAM Bound:	**8.0%**	**of Clockticks**
Loads:	17,604,528,120	
Stores:	8,789,663,682	
LLC Miss Count:	**46,352,781**	
Average Latency (cycles):	57	
Total Thread Count:	4	
Paused Time:	0s	

High percentage of L3 Bound cycles

System Bandwidth

This section provides various system bandwidth-related properties detected by the product. These values are used to define default High, Medium and Low bandwidth utilization thresholds for the Bandwidth Utilization Histogram and to scale overtime bandwidth graphs in the Bottom-up view.

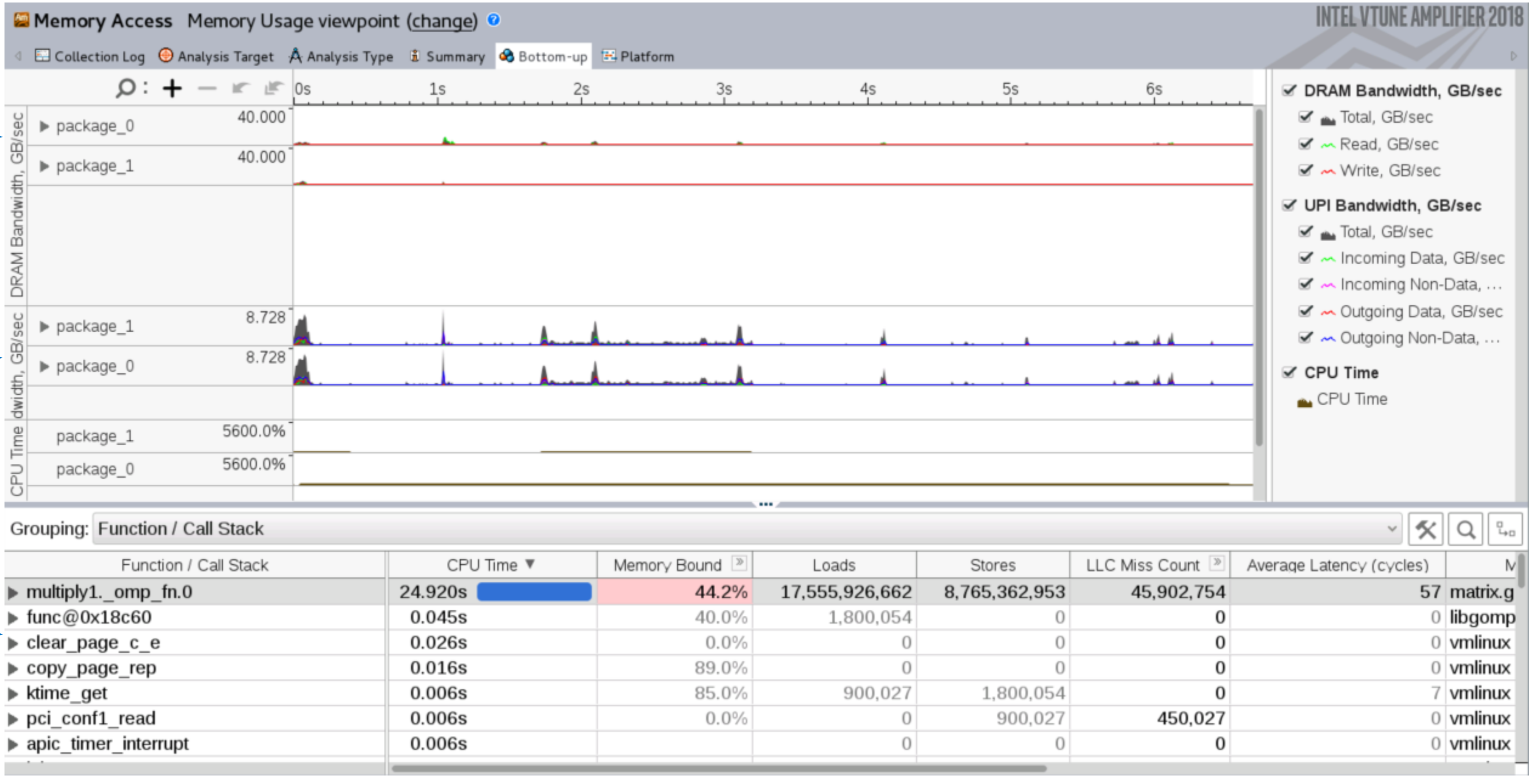
Max DRAM System Bandwidth:	80 GB
Max DRAM Single-Package Bandwidth:	40 GB

VTune Amplifier Workflow Example- Bottom-Up View

Over-Time DRAM Bandwidth

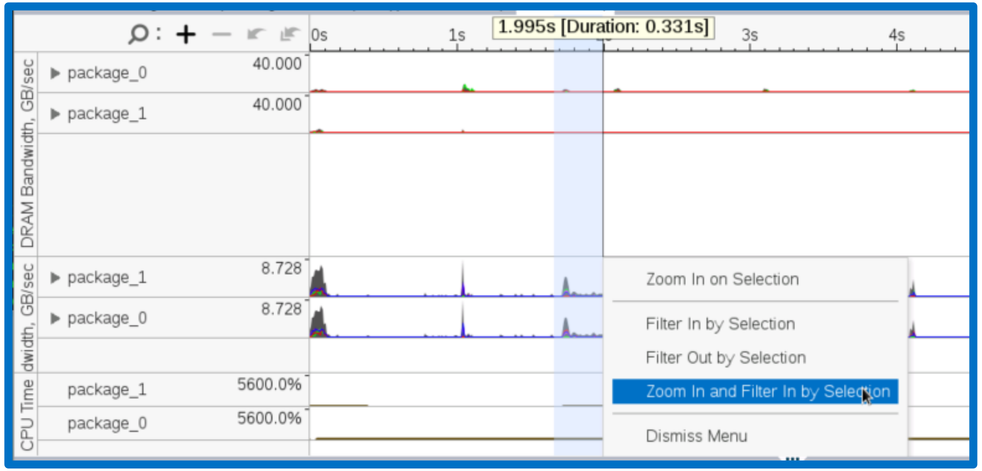
Over-Time QPI/UPI Bandwidth

Grid Breakdown by Function (configurable)



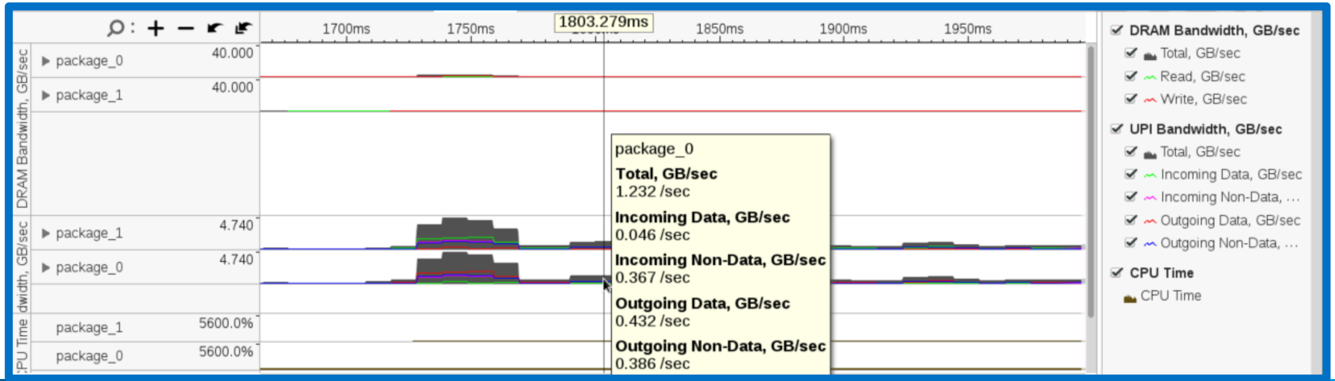
Function / Call Stack	CPU Time ▼	Memory Bound	Loads	Stores	LLC Miss Count	Average Latency (cycles)	M
▶ multiply1._omp_fn.0	24.920s	44.2%	17,555,926,662	8,765,362,953	45,902,754	57	matrix.g
▶ func@0x18c60	0.045s	40.0%	1,800,054	0	0	0	libgomp
▶ clear_page_c_e	0.026s	0.0%	0	0	0	0	vmlinux
▶ copy_page_rep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

VTune Amplifier Workflow Example- Bottom-Up View



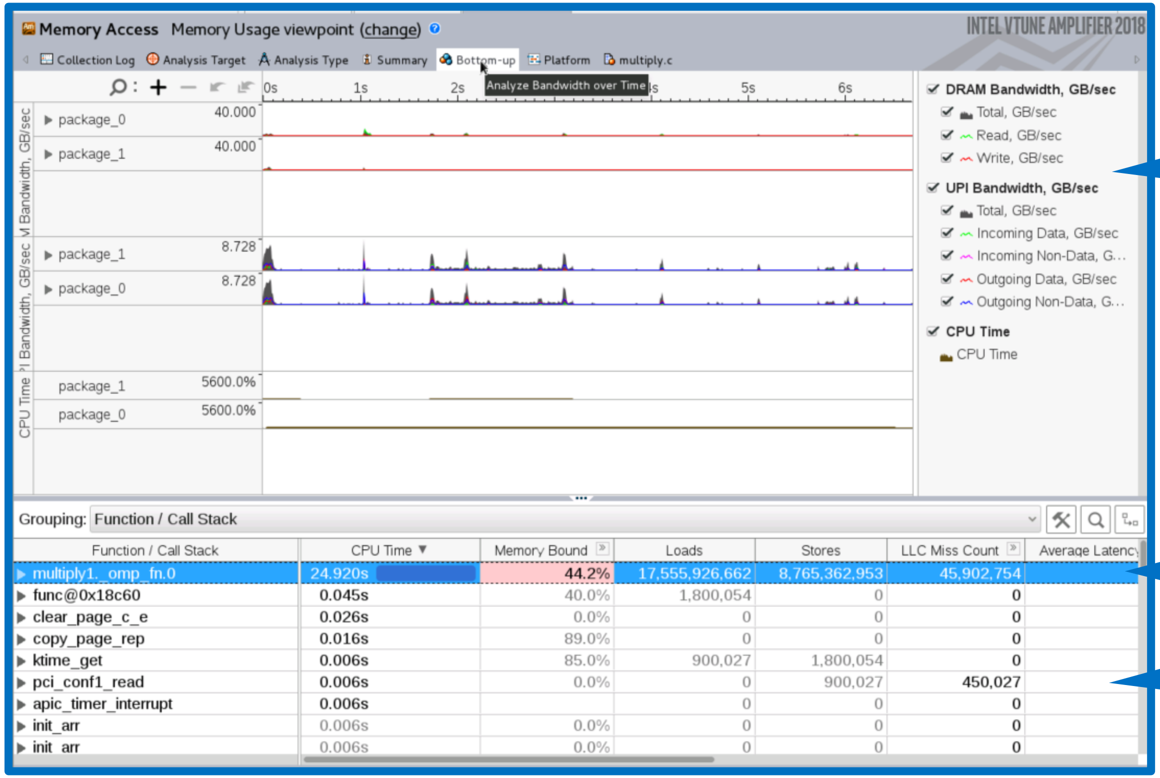
Focus on areas of interest with "Zoom In and Filter"

Fine-grained details in Zoomed-in view



package_0
Total, GB/sec
1.232 /sec
Incoming Data, GB/sec
0.046 /sec
Incoming Non-Data, GB/sec
0.367 /sec
Outgoing Data, GB/sec
0.432 /sec
Outgoing Non-Data, GB/sec
0.386 /sec

VTune Amplifier Workflow Example- Bottom-Up View



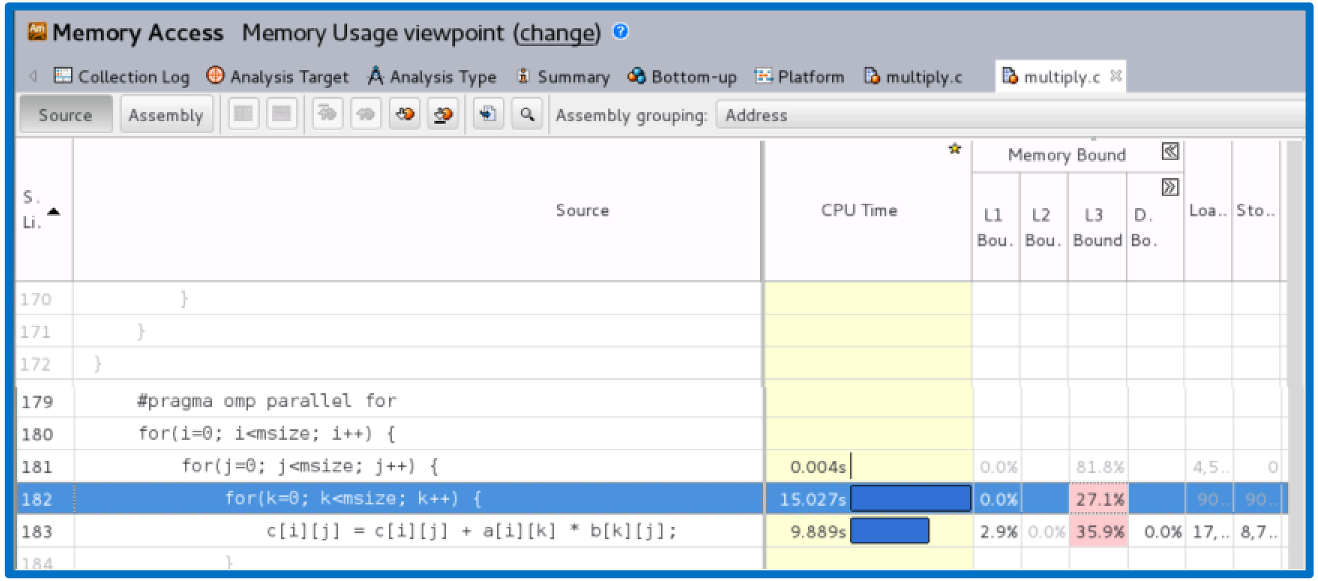
Function / Call Stack	CPU Time ▼	Memory Bound	Loads	Stores	LLC Miss Count	Average Latency
multiply1._omp_fn.0	24.920s	44.2%	17,555,926,662	8,765,362,953	45,902,754	
func@0x18c60	0.045s	40.0%	1,800,054	0	0	
clear_page_c_e	0.026s	0.0%	0	0	0	
copy_page_rep	0.016s	89.0%	0	0	0	
ktime_get	0.006s	85.0%	900,027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

DRAM and UPI Bandwidth are low.

Memory Bound function. 44% of pipeline slots are stalled.

Double-click a function for source view.

VTune Amplifier Workflow Example- Source View



Memory Access Memory Usage viewpoint (change)

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Platform | multiply.c | multiply.c

Source | Assembly | Assembly grouping: Address

S. Li.	Source	CPU Time	Memory Bound				Loa..	Sto..
			L1 Bou.	L2 Bou.	L3 Bound	D. Bo.		
170	`}`							
171	`}`							
172	`}`							
179	`#pragma omp parallel for`							
180	`for(i=0; i<msize; i++) {`							
181	`for(j=0; j<msize; j++) {`	0.004s	0.0%		81.8%		4,5..	0
182	`for(k=0; k<msize; k++) {`	15.027s	0.0%		27.1%		90..	90..
183	`c[i][j] = c[i][j] + a[i][k] * b[k][j];`	9.889s	2.9%	0.0%	35.9%	0.0%	17,..	8,7..
184	`}`							

Metrics at a source line granularity

Inefficient array access pattern in nested loop

Intel® Optane™ DC Persistent Memory

Determine whether your application can benefit from Intel® Optane™ DC Persistent Memory without the hardware using **Memory Consumption** analysis. Identify frequently accessed objects using a **Memory Access** analysis.

Memory Mode	App Direct Mode
Requires no special programming. Just turn it on and see if it helps!	Requires the use of an API to manually control memory allocation.
Not actually persistent. Acts like an extra layer of cache between DRAM and disk.	Comes in Volatile (non-persistent) and Non-Volatile (persistent) modes.
Hottest data should remain in DRAM while the rest goes to persistent memory instead of disk.	Hottest and/or store-heavy objects should reside in DRAM and the rest in persistent memory.

Non-Volatile Persistent Memory may not behave as expected. Errors can be detected early using **Intel® Inspector – Persistence Inspector.**

***Note:** Memory Consumption analysis is not currently available on Windows* operating systems.*

Solutions Sampler

Find Hotspots → Determine Efficiency → Diagnose Bottleneck → Implement Solution

Back End Bound

Core Bound

Divider

- Use reciprocal-multiplication where possible.

Memory Bound

Contested Access/Data Sharing

- Solve false sharing by padding variables to cache line boundaries.
- Try to reduce actual sharing requirements.

Remote Memory Access

- Affinitize/pin threads to cores.
- Use NUMA-efficient thread schedulers like Intel® Threading Building Blocks.
- Test whether performance improves using Sub-NUMA Cluster Mode.

Cache Misses

- Block your data.
- Use software prefetches.
- Consider Intel® Optane™ DC Persistent Memory.

Front End Bound

Front End Latency

- Use switches to reduce code size, such as `/O1` or `/Os`.
- Use Profile-Guided Optimization (PGO) with the compiler.
- For dynamically generated code, try co-locating hot code, reducing code size, and avoiding indirect calls.

Bad Speculation

Branch Mispredicts

- Avoid unnecessary branching.
- Hoist popular branch targets.
- Use PGO with the compiler.

Machine Clears

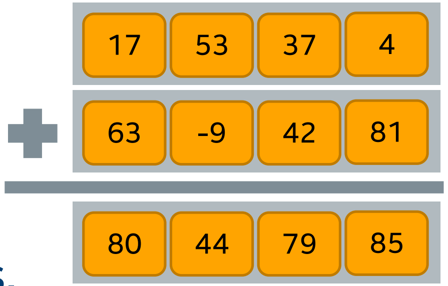
- Check for lock contention or 4k aliasing.

Retiring

You're doing more work than you need to.

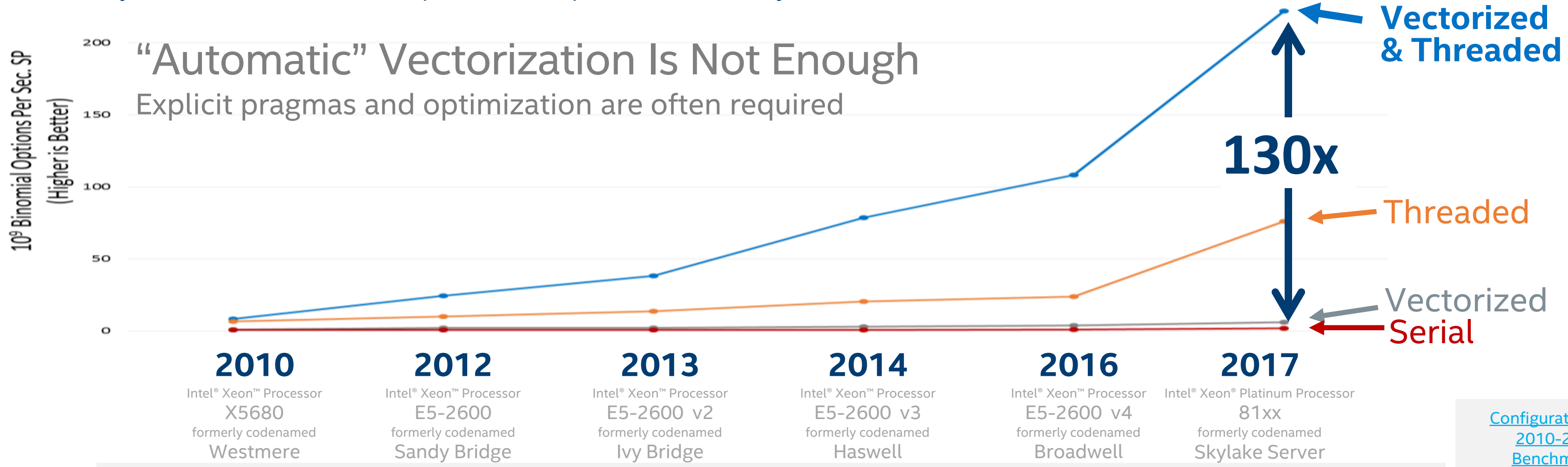
- Use FMAs. Compile with `–fma` or `/Qfma` and the appropriate `–x` or `/Qx` option.
- Take advantage of vectorization with AVX-512!

Vectorization 101



Vector registers and SIMD (Single Instruction Multiple Data) instructions allow a CPU to do multiple operations at once.

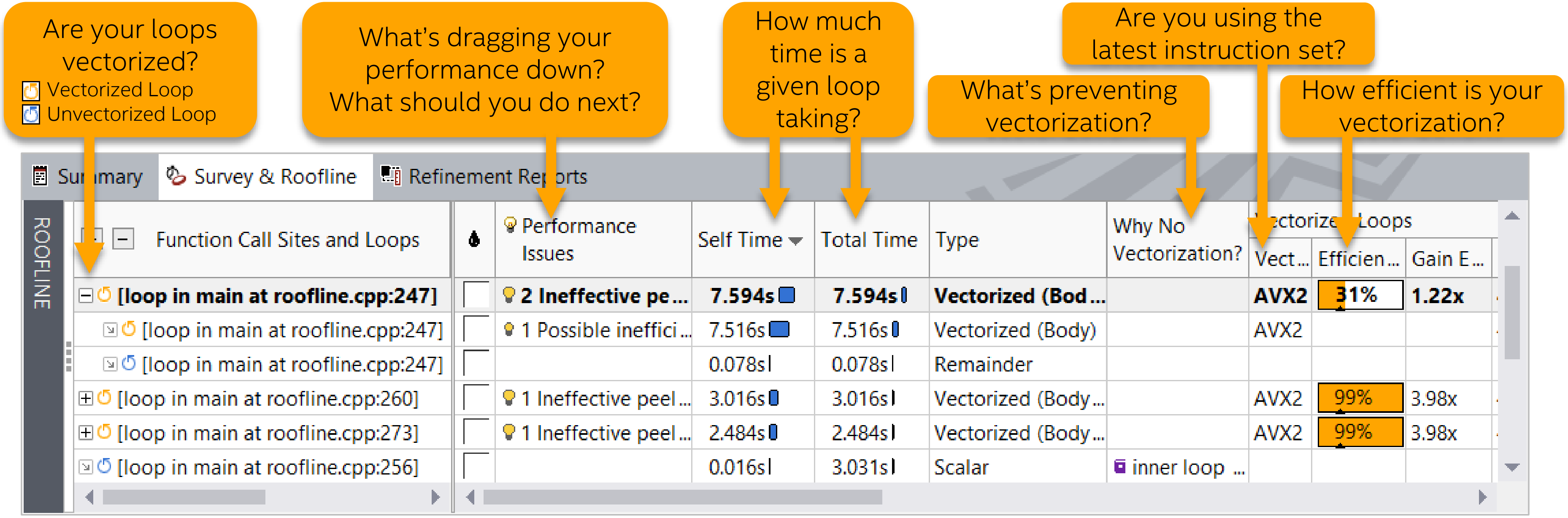
- Use `/QxCORE-AVX512` or `-xCORE-AVX512` compiler flags.
  - *If you don't see the expected improvement, try `COMMON-AVX512` instead.*



Configurations for 2010-2017 Benchmarks at the end of this presentation

Intel® Advisor

Intel® Advisor is a thread prototyping and vectorization optimization tool. Start with a **Survey** analysis.



Are your loops vectorized?
Vectorized Loop
Unvectorized Loop

What's dragging your performance down? What should you do next?

How much time is a given loop taking?

What's preventing vectorization?

Are you using the latest instruction set?

How efficient is your vectorization?

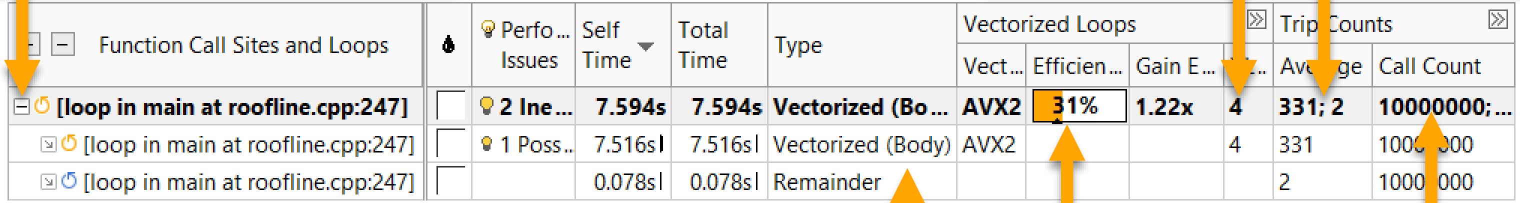
Function Call Sites and Loops		Performance Issues	Self Time	Total Time	Type	Why No Vectorization?	ctorize Loops		
							Vect...	Efficien...	Gain E...
[loop in main at roofline.cpp:247]		**2 Ineffective pe...**	**7.594s**	**7.594s**	**Vectorized (Bod...**		**AVX2**	**31%**	**1.22x**
[loop in main at roofline.cpp:247]		1 Possible ineffici...	7.516s	7.516s	Vectorized (Body)		AVX2		
[loop in main at roofline.cpp:247]			0.078s	0.078s	Remainder				
[loop in main at roofline.cpp:260]		1 Ineffective peel...	3.016s	3.016s	Vectorized (Body...		AVX2	99%	3.98x
[loop in main at roofline.cpp:273]		1 Ineffective peel...	2.484s	2.484s	Vectorized (Body...		AVX2	99%	3.98x
[loop in main at roofline.cpp:256]			0.016s	3.031s	Scalar	inner loop ...			

Trip Counts…

Trip Counts analysis shows you loop trip counts and call counts. High call counts amplify the importance of tuning a loop. Scalar trip counts that aren't divisible by vector length cause remainder loops.

Loops with peels and/or remainders can be expanded.

This loop's scalar trip count was 1326, which doesn't divide evenly by 4.
1326/4=331.5



Function Call Sites and Loops		Perfo… Issues	Self Time	Total Time	Type	Vectorized Loops				Trip Counts	
						Vect…	Efficien…	Gain E…	…	Av ge	Call Count
[loop in main at roofline.cpp:247]		**2 Ine…**	**7.594s**	**7.594s**	**Vectorized (Bo…**	**AVX2**	**31%**	**1.22x**	**4**	**331; 2**	**10000000; …**
[loop in main at roofline.cpp:247]		1 Poss…	7.516s	7.516s	Vectorized (Body)	AVX2			4	331	100 000
[loop in main at roofline.cpp:247]			0.078s	0.078s	Remainder					2	1000 000

You can see which component loops are what type in this column.

Poor efficiency + high call count = major performance penalty!

This is especially important with the long vector registers of AVX-512!

… and FLOPS

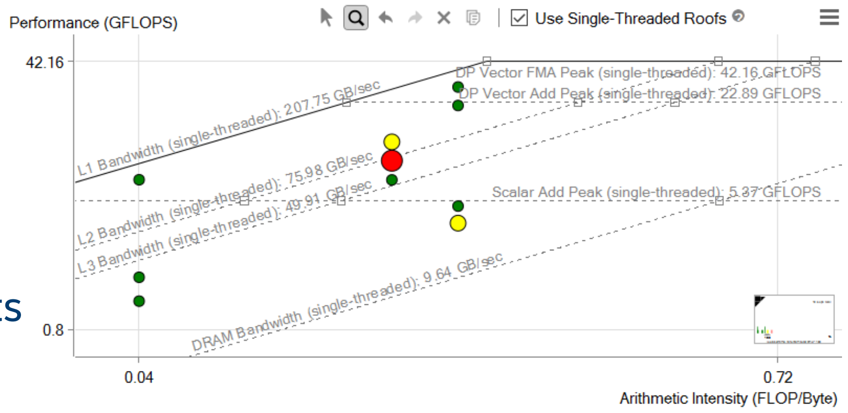
Roofline first proposed by University of California at Berkeley:
[Roofline: An Insightful Visual Performance Model for Multicore Architectures](), 2009
Cache-aware variant proposed by University of Lisbon:
[Cache-Aware Roofline Model: Upgrading the Loft](), 2013

Trip Counts analysis can also collect FLOP and Mask Utilization data.

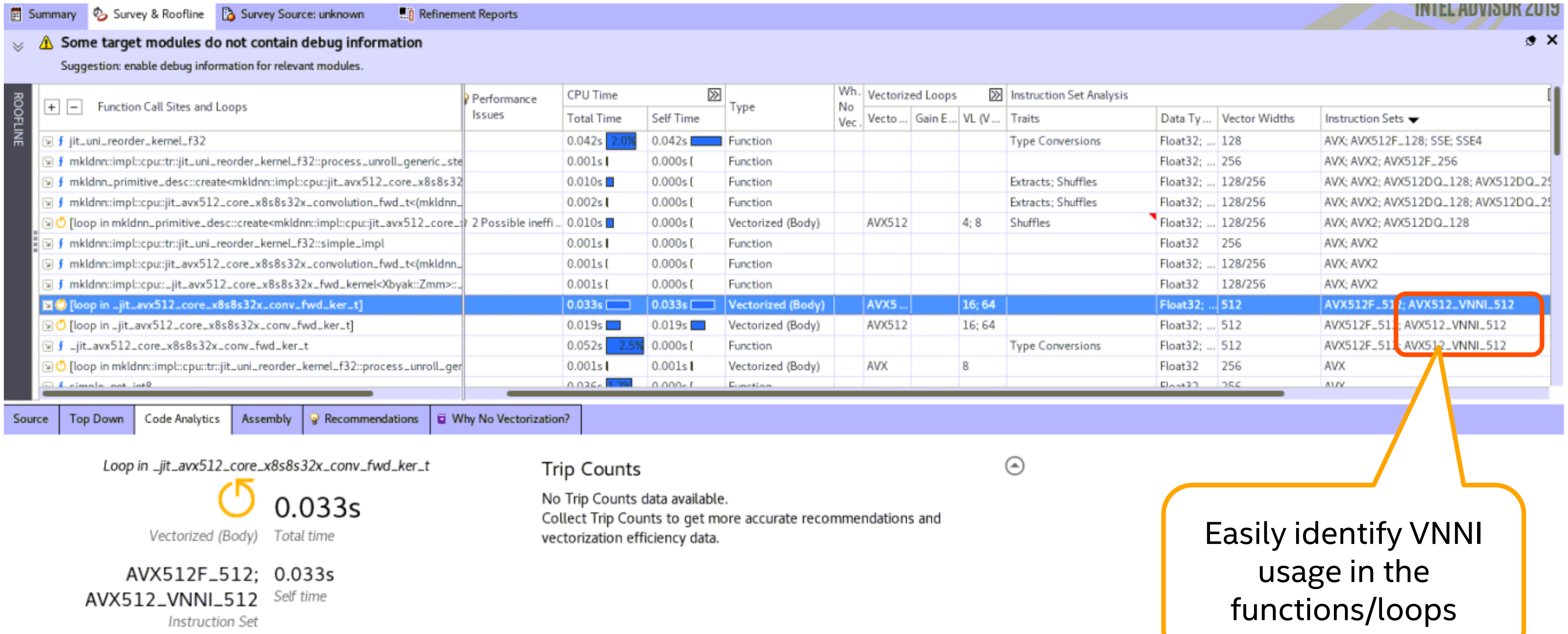
- **Fl**oating-point **O**perations are used to calculate FLOPS (**Fl**oating Point **O**perations **P**er **S**econd)… but Integer operations are also supported!

FLOPS and IntOPS are computation-specific performance measurements. Collecting them produces a Roofline chart, a visual representation of performance relative to hardware limits.

- The horizontal axis is Arithmetic Intensity, a measurement of FLOPs per byte accessed. The vertical axis is performance.
- The dots are loops. The lines are hardware limitations; horizontal lines are compute limits and diagonal lines are memory limits.



VNNI usage verification by Intel Advisor



Some target modules do not contain debug information

Suggestion: enable debug information for relevant modules.

Function Call Sites and Loops	Performance Issues	CPU Time		Type	Wh. No Vec.	Vectorized Loops			Instruction Set Analysis			
		Total Time	Self Time			Vecto ...	Gain E...	VL (V...	Traits	Data Ty...	Vector Widths	Instruction Sets
jit_uni_reorder_kernel_f32		0.042s 2.0%	0.042s	Function					Type Conversions	Float32; ...	128	AVX; AVX512F_128; SSE; SSE4
mkldnn::impl::cpu::tr::jit_uni_reorder_kernel_f32::process_unroll_generic_ste		0.001s	0.000s	Function						Float32; ...	256	AVX; AVX2; AVX512F_256
mkldnn_primitive_desc::create<mkldnn::impl::cpu::jit_avx512_core_x8s8s32		0.010s	0.000s	Function					Extracts; Shuffles	Float32; ...	128/256	AVX; AVX2; AVX512DQ_128; AVX512DQ_2
mkldnn::impl::cpu::jit_avx512_core_x8s8s32x_convolution_fwd_t<(mkldnn_		0.002s	0.000s	Function					Extracts; Shuffles	Float32; ...	128/256	AVX; AVX2; AVX512DQ_128; AVX512DQ_2
[loop in mkldnn_primitive_desc::create<mkldnn::impl::cpu::jit_avx512_core_	2 Possible ineffi...	0.010s	0.000s	Vectorized (Body)		AVX512		4; 8	Shuffles	Float32; ...	128/256	AVX; AVX2; AVX512DQ_128
mkldnn::impl::cpu::tr::jit_uni_reorder_kernel_f32::simple_impl		0.001s	0.000s	Function						Float32	256	AVX; AVX2
mkldnn::impl::cpu::jit_avx512_core_x8s8s32x_convolution_fwd_t<(mkldnn_		0.001s	0.000s	Function						Float32; ...	128/256	AVX; AVX2
mkldnn::impl::cpu::_jit_avx512_core_x8s8s32x_fwd_kernel<Xbyak::Zmm>::_		0.001s	0.000s	Function						Float32	128/256	AVX; AVX2
[loop in _jit_avx512_core_x8s8s32x_conv_fwd_ker_t]		0.033s	0.033s	Vectorized (Body)		AVX5 ...		16; 64		Float32; ...	512	AVX512F_512; AVX512_VNNI_512
[loop in _jit_avx512_core_x8s8s32x_conv_fwd_ker_t]		0.019s	0.019s	Vectorized (Body)		AVX512		16; 64		Float32; ...	512	AVX512F_512; AVX512_VNNI_512
_jit_avx512_core_x8s8s32x_conv_fwd_ker_t		0.052s 2.5%	0.000s	Function					Type Conversions	Float32; ...	512	AVX512F_512; AVX512_VNNI_512
[loop in mkldnn::impl::cpu::tr::jit_uni_reorder_kernel_f32::process_unroll_ger		0.001s	0.001s	Vectorized (Body)		AVX		8		Float32	256	AVX

Loop in _jit_avx512_core_x8s8s32x_conv_fwd_ker_t

0.033s
Vectorized (Body) *Total time*

AVX512F_512; AVX512_VNNI_512
Instruction Set

0.033s
Self time

Trip Counts

No Trip Counts data available.
Collect Trip Counts to get more accurate recommendations and vectorization efficiency data.

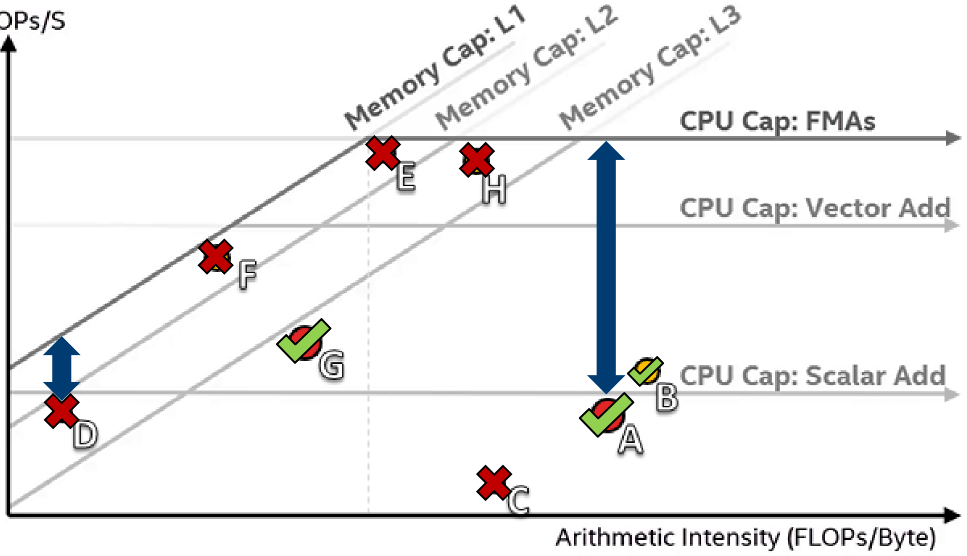
Easily identify VNNI usage in the functions/loops

Roofline

Roofline Video: https://software.intel.com/en-us/videos/roofline-analysis-in-intel-advisor-2017
Roofline Article: https://software.intel.com/en-us/articles/intel-advisor-roofline

The Roofline chart can be an effective means of identifying bottlenecks, and determining what optimizations to make where, for maximum effect.
It is a good indicator of:

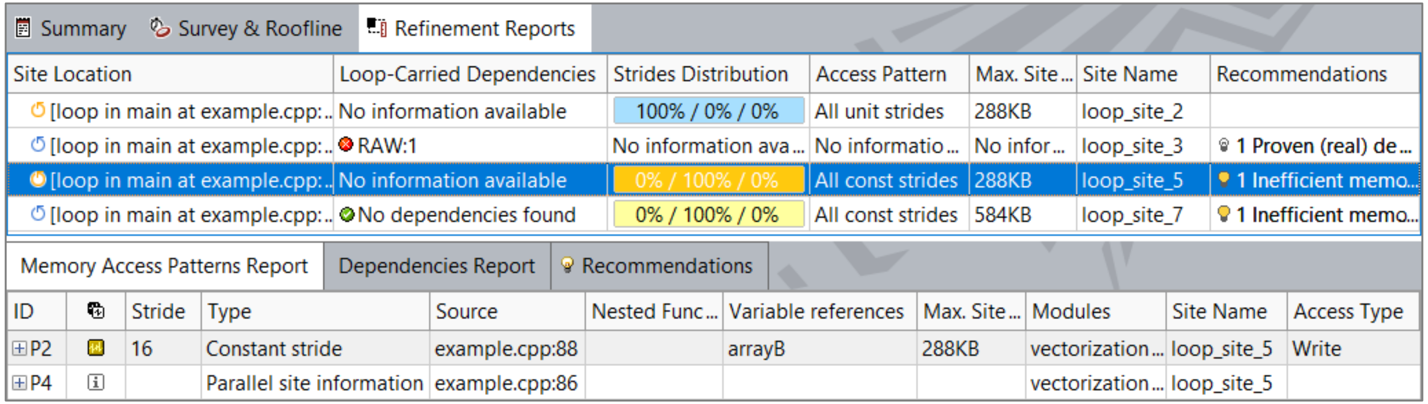
- How much performance is left on the table
- Which loops take the most time
- Which loops are worth optimizing
- Likely causes of performance bottlenecks
- What to investigate next



Memory Access Patterns & Dependencies

Memory Access Patterns (MAP) and **Dependencies** are specialized analysis types. Use them when Advisor recommends.

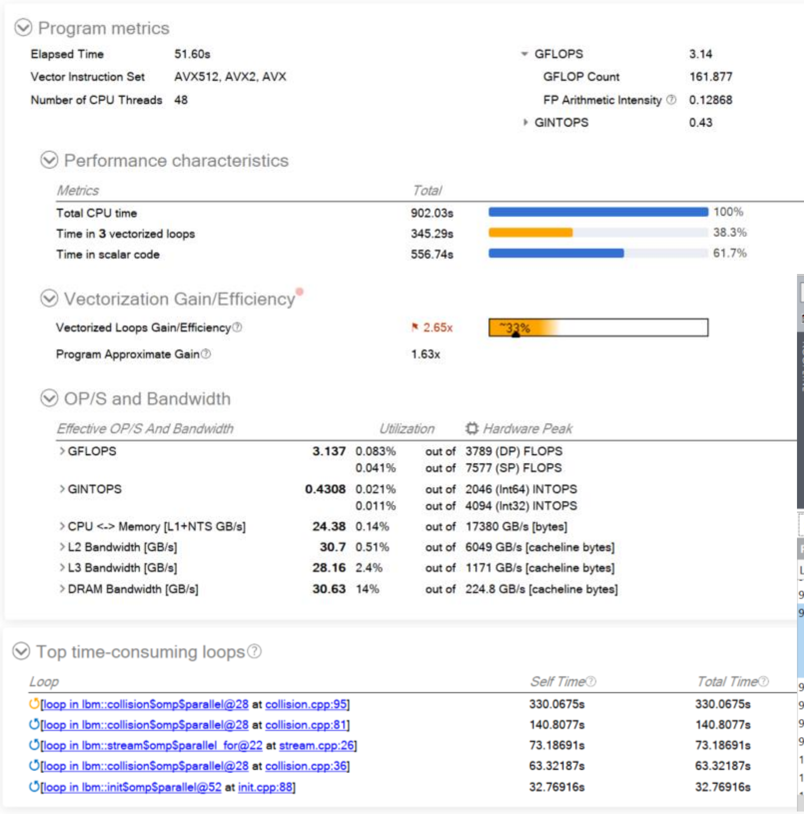
- MAP detects inefficient strides and mask utilization information.
- Dependencies determines whether it's safe to force vectorization in a loop that was left scalar due to the compiler detecting a potential dependency.



Site Location	Loop-Carried Dependencies	Strides Distribution	Access Pattern	Max. Site...	Site Name	Recommendations
[loop in main at example.cpp:...	No information available	100% / 0% / 0%	All unit strides	288KB	loop_site_2	
[loop in main at example.cpp:...	RAW:1	No information ava...	No informatio...	No infor...	loop_site_3	1 Proven (real) de...
[loop in main at example.cpp:...	No information available	0% / 100% / 0%	All const strides	288KB	loop_site_5	1 Inefficient memo...
[loop in main at example.cpp:...	No dependencies found	0% / 100% / 0%	All const strides	584KB	loop_site_7	1 Inefficient memo...

ID		Stride	Type	Source	Nested Func...	Variable references	Max. Site...	Modules	Site Name	Access Type
⊞P2		16	Constant stride	example.cpp:88		arrayB	288KB	vectorization...	loop_site_5	Write
⊞P4			Parallel site information	example.cpp:86				vectorization...	loop_site_5	

Intel® Advisor GUI



Program metrics

Elapsed Time	51.60s	▾ GFLOPS	3.14
Vector Instruction Set	AVX512, AVX2, AVX	GFLOP Count	161.877
Number of CPU Threads	48	FP Arithmetic Intensity	0.12868
		▸ GINTOPS	0.43

Performance characteristics

Metrics	*Total*	
Total CPU time	902.03s	100%
Time in 3 vectorized loops	345.29s	38.3%
Time in scalar code	556.74s	61.7%

Vectorization Gain/Efficiency

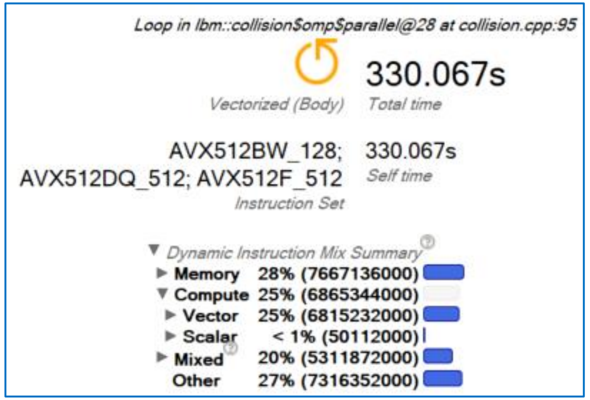
Vectorized Loops Gain/Efficiency	2.65x	~33%
Program Approximate Gain	1.63x	

OP/S and Bandwidth

Effective OP/S And Bandwidth		*Utilization*		*Hardware Peak*
› GFLOPS	3.137	0.083%	out of	3789 (DP) FLOPS
		0.041%	out of	7577 (SP) FLOPS
› GINTOPS	0.4308	0.021%	out of	2046 (Int64) INTOPS
		0.011%	out of	4094 (Int32) INTOPS
› CPU <-> Memory [L1+NTS GB/s]	24.38	0.14%	out of	17380 GB/s [bytes]
› L2 Bandwidth [GB/s]	30.7	0.51%	out of	6049 GB/s [cacheline bytes]
› L3 Bandwidth [GB/s]	28.16	2.4%	out of	1171 GB/s [cacheline bytes]
› DRAM Bandwidth [GB/s]	30.63	14%	out of	224.8 GB/s [cacheline bytes]

Top time-consuming loops

Loop	*Self Time*	*Total Time*
[loop in lbm::collision$omp$parallel@28 at collision.cpp:95]	330.0675s	330.0675s
[loop in lbm::collision$omp$parallel@28 at collision.cpp:81]	140.8077s	140.8077s
[loop in lbm::stream$omp$parallel_for@22 at stream.cpp:26]	73.18691s	73.18691s
[loop in lbm::collision$omp$parallel@28 at collision.cpp:36]	63.32187s	63.32187s
[loop in lbm::init$omp$parallel@52 at init.cpp:88]	32.76916s	32.76916s

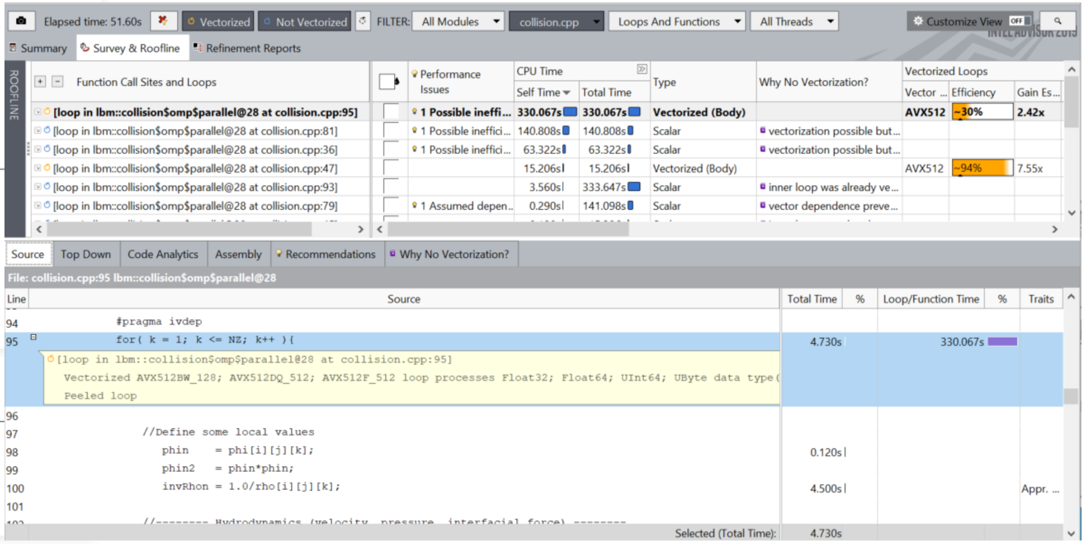


Loop in lbm::collision$omp$parallel@28 at collision.cpp:95

Vectorized (Body)	330.067s *Total time*
AVX512BW_128; AVX512DQ_512; AVX512F_512 *Instruction Set*	330.067s *Self time*

▼ *Dynamic Instruction Mix Summary*

▶ Memory	28% (7667136000)
▼ Compute	25% (6865344000)
▶ Vector	25% (6815232000)
▶ Scalar	< 1% (50112000)
▶ Mixed	20% (5311872000)
Other	27% (7316352000)



Function Call Sites and Loops	Performance Issues	CPU Time		Type	Why No Vectorization?	Vectorized Loops		
		Self Time	Total Time			Vector ...	Efficiency	Gain Es...
[loop in lbm::collision$omp$parallel@28 at collision.cpp:95]	**1 Possible ineffi...**	**330.067s**	**330.067s**	**Vectorized (Body)**		**AVX512**	**~30%**	**2.42x**
[loop in lbm::collision$omp$parallel@28 at collision.cpp:81]	1 Possible ineffici...	140.808s	140.808s	Scalar	vectorization possible but...			
[loop in lbm::collision$omp$parallel@28 at collision.cpp:36]	1 Possible ineffici...	63.322s	63.322s	Scalar	vectorization possible but...			
[loop in lbm::collision$omp$parallel@28 at collision.cpp:47]		15.206s	15.206s	Vectorized (Body)		AVX512	~94%	7.55x
[loop in lbm::collision$omp$parallel@28 at collision.cpp:93]		3.560s	333.647s	Scalar	inner loop was already ve...			
[loop in lbm::collision$omp$parallel@28 at collision.cpp:79]	1 Assumed depen...	0.290s	141.098s	Scalar	vector dependence preve...			

File: collision.cpp:95 lbm::collision$omp$parallel@28

Line	Source	Total Time	%	Loop/Function Time	%	Traits
94	`#pragma ivdep`					
95	`for( k = 1; k <= NZ; k++ ){`	4.730s		330.067s		
	`[loop in lbm::collision$omp$parallel@28 at collision.cpp:95]` `Vectorized AVX512BW_128; AVX512DQ_512; AVX512F_512 loop processes Float32; Float64; UInt64; UByte data type(` `Peeled loop`					
96						
97	`//Define some local values`					
98	`phin     = phi[i][j][k];`	0.120s				
99	`phin2    = phin*phin;`					
100	`invRhon = 1.0/rho[i][j][k];`	4.500s				Appr. ...
101						

Selected (Total Time): 4.730s

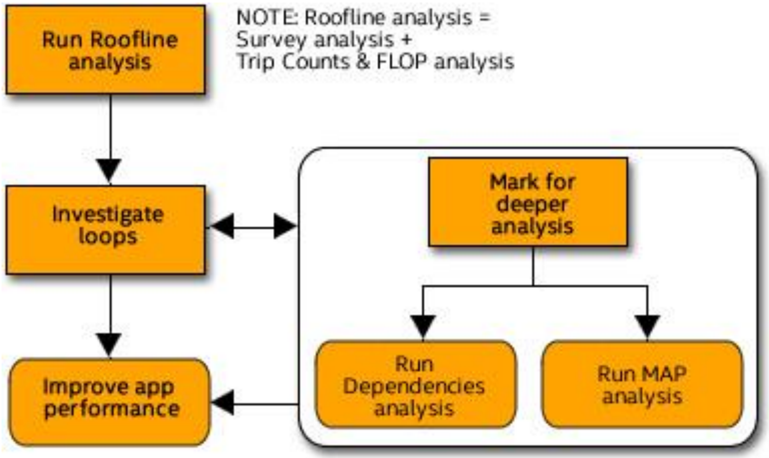
Typical Vectorization Optimization Workflow

There is no need to recompile or relink the application, but the use of -g is recommended.

In a rush: Collect Survey data and analyze loops iteratively

Looking for detail:

1. Collect survey and tripcounts data **[Roofline]**
   - Investigate application place within roofline model
   - Determine vectorization efficiency and opportunities for improvement
2. Collect memory access pattern data
   - Determine data structure optimization needs
3. Collect dependencies
   - Differentiate between real and assumed issues blocking vectorization

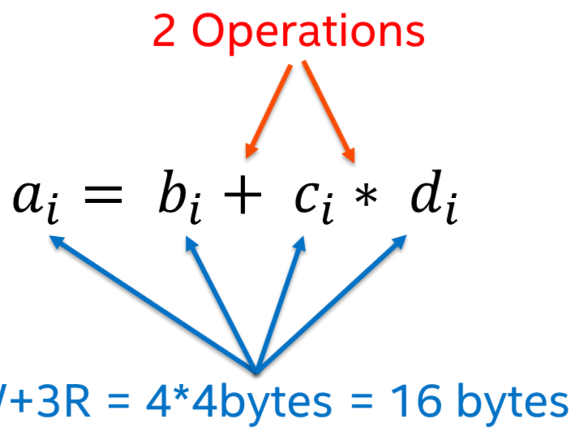


What is the Roofline Model?

Characterization of your application performance in the context of the hardware

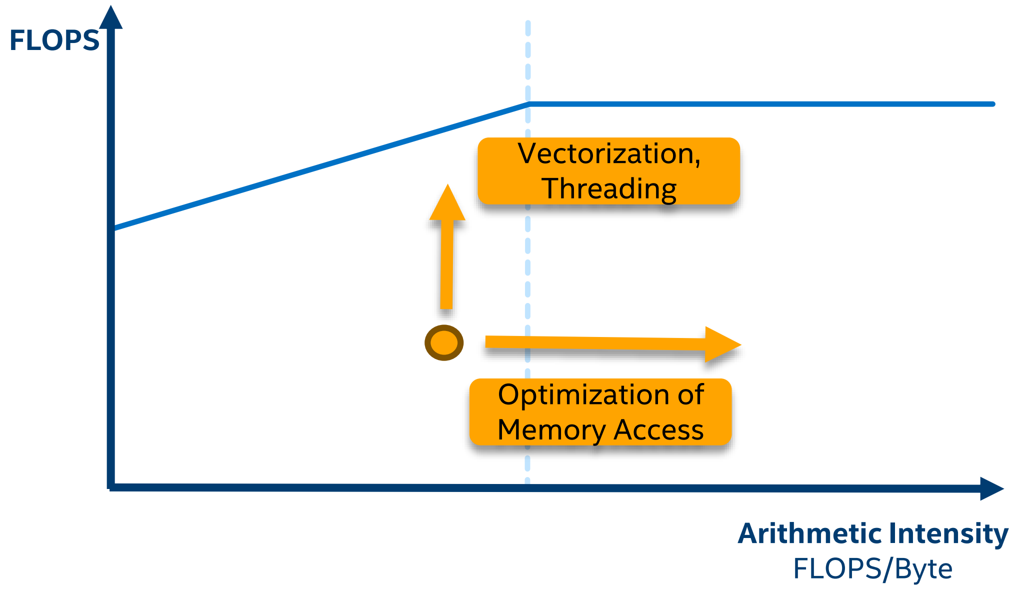
It uses two simple metrics

- Flop count
- Bytes transferred



$$a_i = b_i + c_i * d_i$$

1W+3R = 4*4bytes = 16 bytes



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

Roofline Model in Intel® Advisor

Intel® Advisor implements a Cache Aware Roofline Model (CARM)

- "Algorithmic", "Cumulative (L1+L2+LLC+DRAM)" traffic-based
- Invariant for the given code / platform combination

How does it work ?

- Counts every memory movement
- Instrumentation - Bytes and Flops
- Sampling - Time

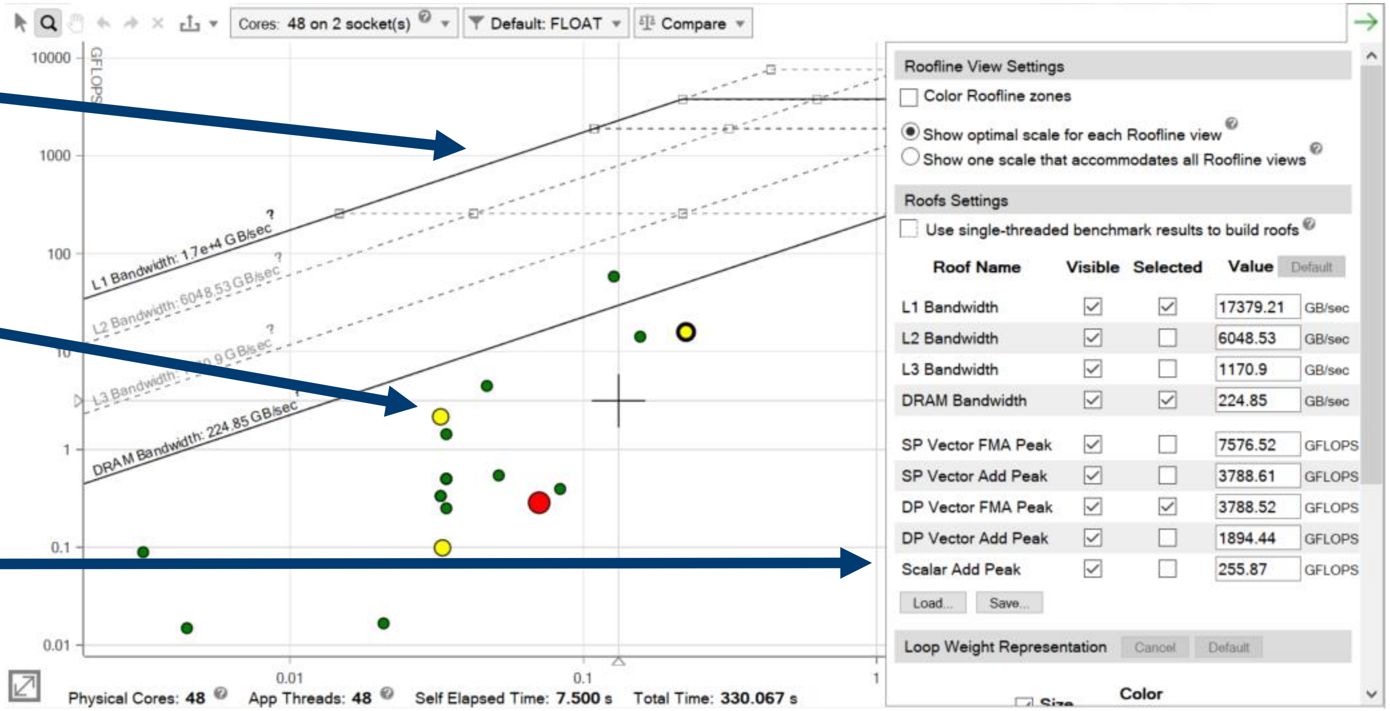
Advantage of CARM	Disadvantage of CARM
No Hardware counters	Only vertical movements !
Affordable overhead (at worst =~10x)	Difficult to interpret
Algorithmic (cumulative L1/L2/LLC)	How to improve performance ?

Roofline Chart in Intel® Advisor

Roof values are **measured**

Dots represent profiled loops and functions

High level of customization


Cores: 48 on 2 socket(s)
Default: FLOAT
Compare
GFLOPS
10000
1000
100
10
1
0.1
0.01
L1 Bandwidth: 1.7e+4 GB/sec
L2 Bandwidth: 6048.53 GB/sec
L3 Bandwidth: 1170.9 GB/sec
DRAM Bandwidth: 224.85 GB/sec
0.01
0.1
1

Physical Cores: 48 App Threads: 48 Self Elapsed Time: 7.500 s Total Time: 330.067 s

Roofline View Settings

- [ ] Color Roofline zones
- [✓ Show optimal scale for each Roofline view]
- Show one scale that accommodates all Roofline views

Roofs Settings

- [ ] Use single-threaded benchmark results to build roofs

Roof Name	Visible	Selected	Value	
L1 Bandwidth	✓	✓	17379.21	GB/sec
L2 Bandwidth	✓		6048.53	GB/sec
L3 Bandwidth	✓		1170.9	GB/sec
DRAM Bandwidth	✓	✓	224.85	GB/sec
SP Vector FMA Peak	✓		7576.52	GFLOPS
SP Vector Add Peak	✓		3788.61	GFLOPS
DP Vector FMA Peak	✓	✓	3788.52	GFLOPS
DP Vector Add Peak	✓		1894.44	GFLOPS
Scalar Add Peak	✓		255.87	GFLOPS

Load... Save...

Loop Weight Representation Cancel Default

Size Color

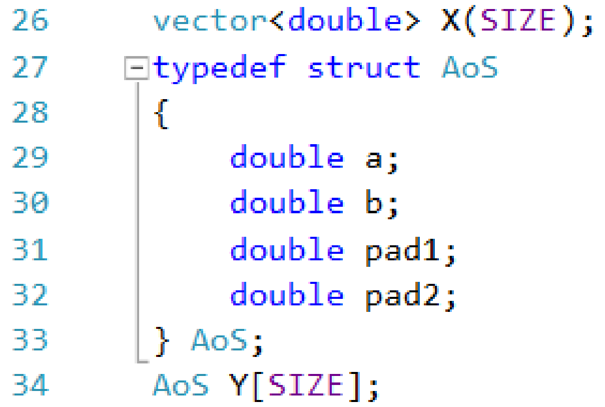
TUNING A SMALL EXAMPLE WITH ROOFLINE

A Short Walk Through the Process

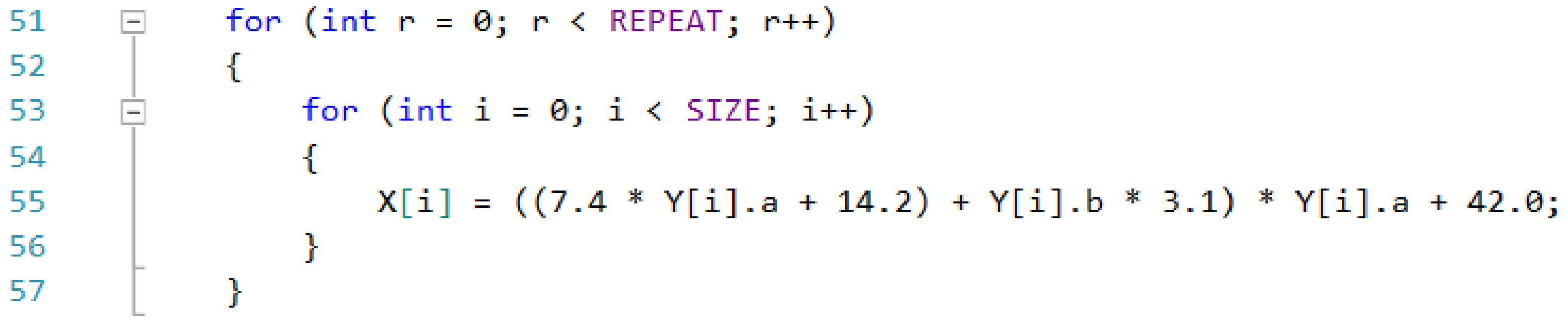
Example Code

A Short Walk Through the Process

The example loop runs through an array of structures and does some generic math on some of its elements, then stores the results into a vector. It repeats this several times to artificially pad the short run time of the simple example.



```
vector<double> X(SIZE);
typedef struct AoS
{
    double a;
    double b;
    double pad1;
    double pad2;
} AoS;
AoS Y[SIZE];
```

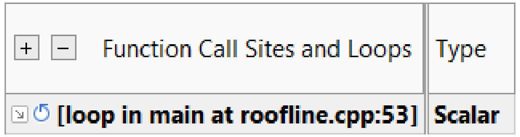


```
for (int r = 0; r < REPEAT; r++)
{
    for (int i = 0; i < SIZE; i++)
    {
        X[i] = ((7.4 * Y[i].a + 14.2) + Y[i].b * 3.1) * Y[i].a + 42.0;
    }
}
```

Finding the Initial Bottleneck

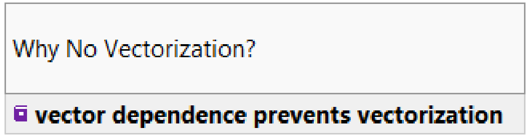
A Short Walk Through the Process

The loop is initially under the Scalar Add Peak. The Survey confirms the loop is not vectorized.

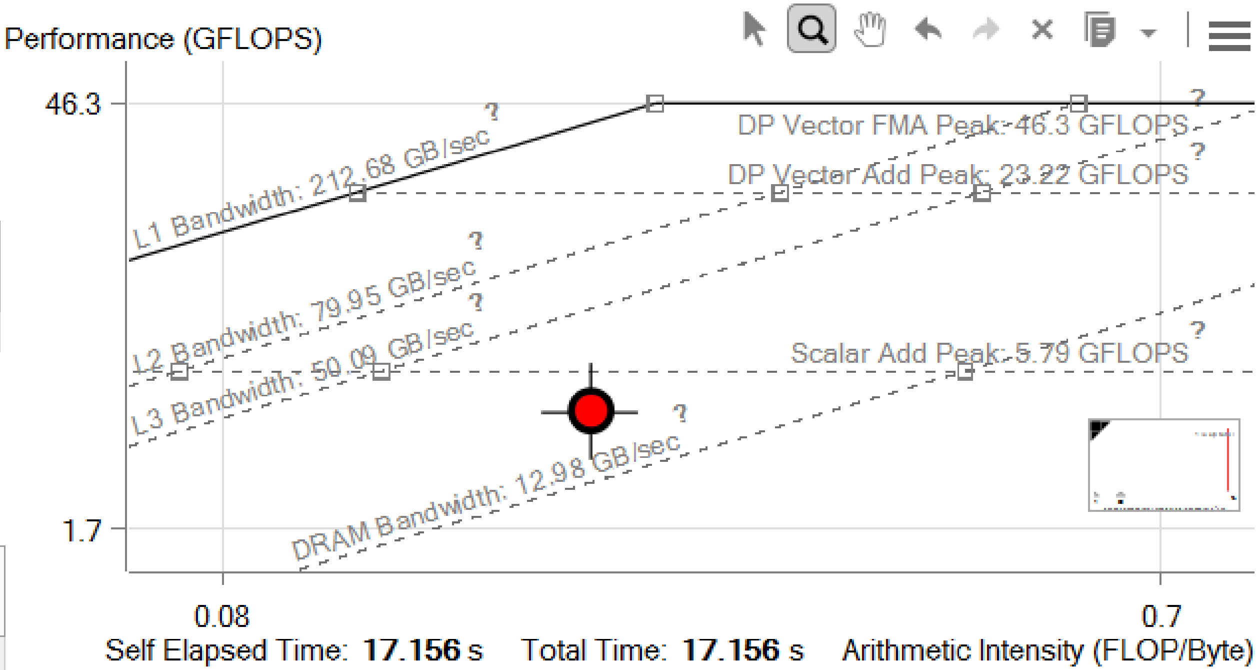


Function Call Sites and Loops	Type
[loop in main at roofline.cpp:53]	Scalar

The “Why No Vectorization?” column reveals why.



Why No Vectorization?
vector dependence prevents vectorization

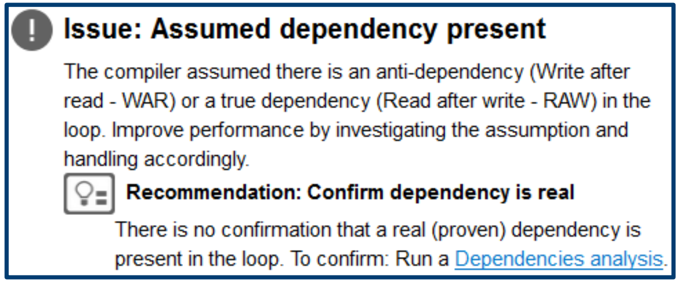


Self Elapsed Time: **17.156** s Total Time: **17.156** s

Overcoming the Initial Bottleneck

A Short Walk Through the Process

The recommendations tab elaborates: the dependency is only assumed.



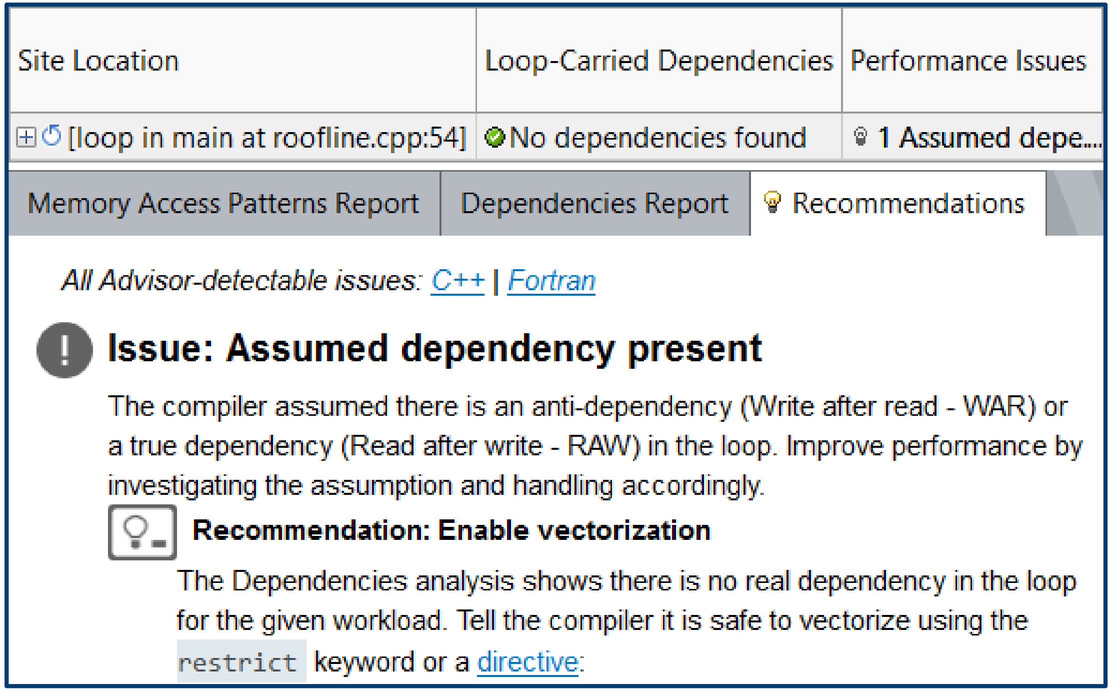
Issue: Assumed dependency present

The compiler assumed there is an anti-dependency (Write after read - WAR) or a true dependency (Read after write - RAW) in the loop. Improve performance by investigating the assumption and handling accordingly.

Recommendation: Confirm dependency is real

There is no confirmation that a real (proven) dependency is present in the loop. To confirm: Run a Dependencies analysis.

Running a Dependencies analysis confirms that it's false, and recommends forcing vectorization with a pragma.



Site Location	Loop-Carried Dependencies	Performance Issues
[loop in main at roofline.cpp:54]	No dependencies found	1 Assumed depe...

All Advisor-detectable issues: C++ | Fortran

Issue: Assumed dependency present

The compiler assumed there is an anti-dependency (Write after read - WAR) or a true dependency (Read after write - RAW) in the loop. Improve performance by investigating the assumption and handling accordingly.

Recommendation: Enable vectorization

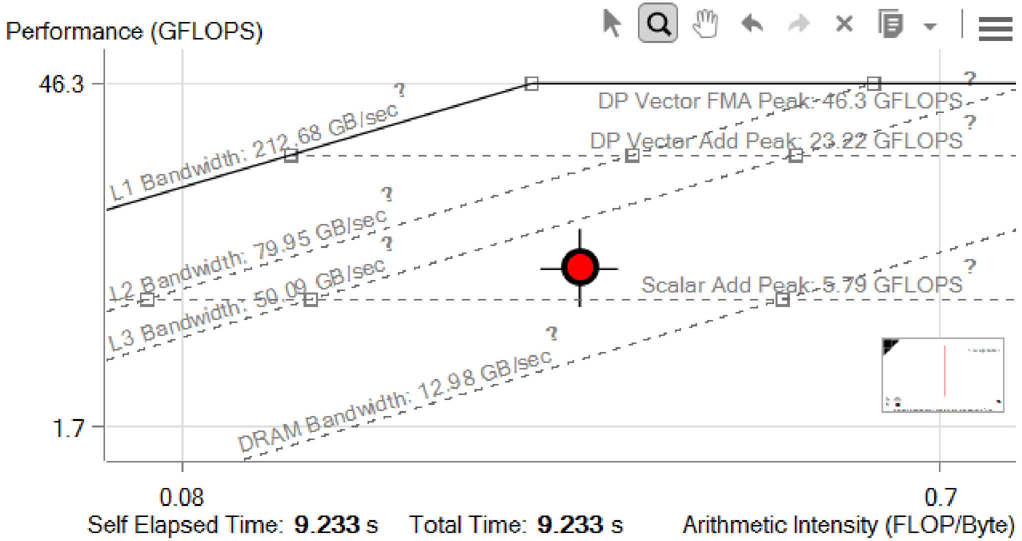
The Dependencies analysis shows there is no real dependency in the loop for the given workload. Tell the compiler it is safe to vectorize using the `restrict` keyword or a directive:

The Second Bottleneck

A Short Walk Through the Process

Adding a pragma to force the loop to vectorize successfully overcomes the Scalar Add Peak. It is now below L3 Bandwidth.

The compiler is not making the same algorithmic optimizations, so the AI has also changed.

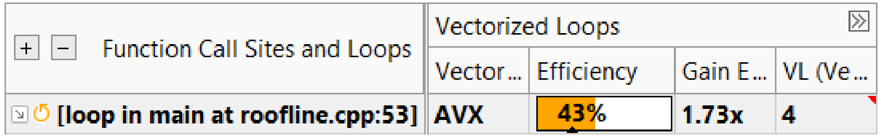


```
for (int r = 0; r < REPEAT; r++)
{
    #pragma omp simd
    for (int i = 0; i < SIZE; i++)
    {
        X[i] = ((7.4 * Y[i].a + 14.2) + Y[i].b * 3.1) * Y[i].a + 42.0;
    }
}
```

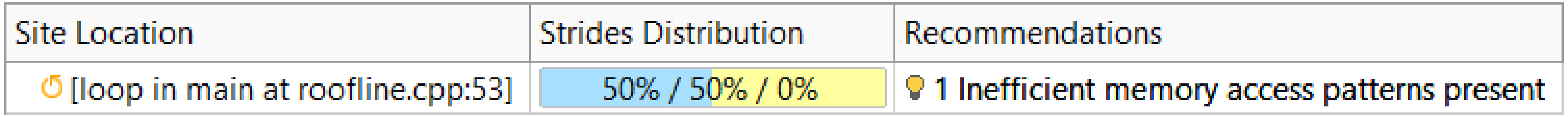
Diagnosing Inefficiency

A Short Walk Through the Process

While the loop is now vectorized, it is inefficient. Inefficient vectorization and excessive cache traffic both often result from poor access patterns, which can be confirmed with a MAP analysis.

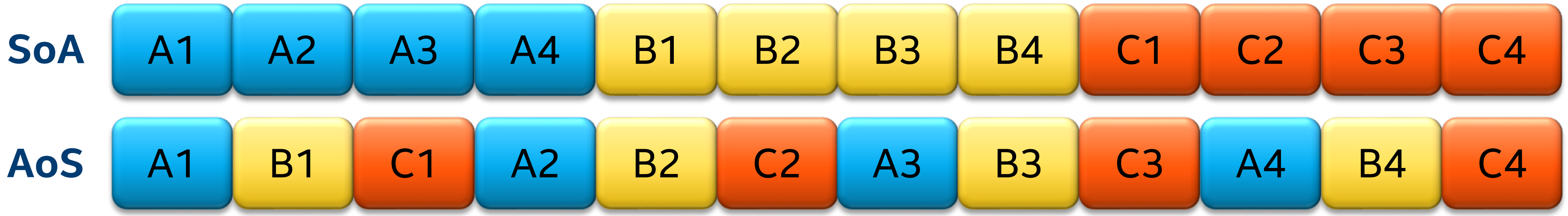


Function Call Sites and Loops	Vectorized Loops			
	Vector...	Efficiency	Gain E...	VL (Ve...
[loop in main at roofline.cpp:53]	AVX	43%	1.73x	4



Site Location	Strides Distribution	Recommendations
[loop in main at roofline.cpp:53]	50% / 50% / 0%	1 Inefficient memory access patterns present

Array of Structures is an inefficient data layout, particularly for vectorization.

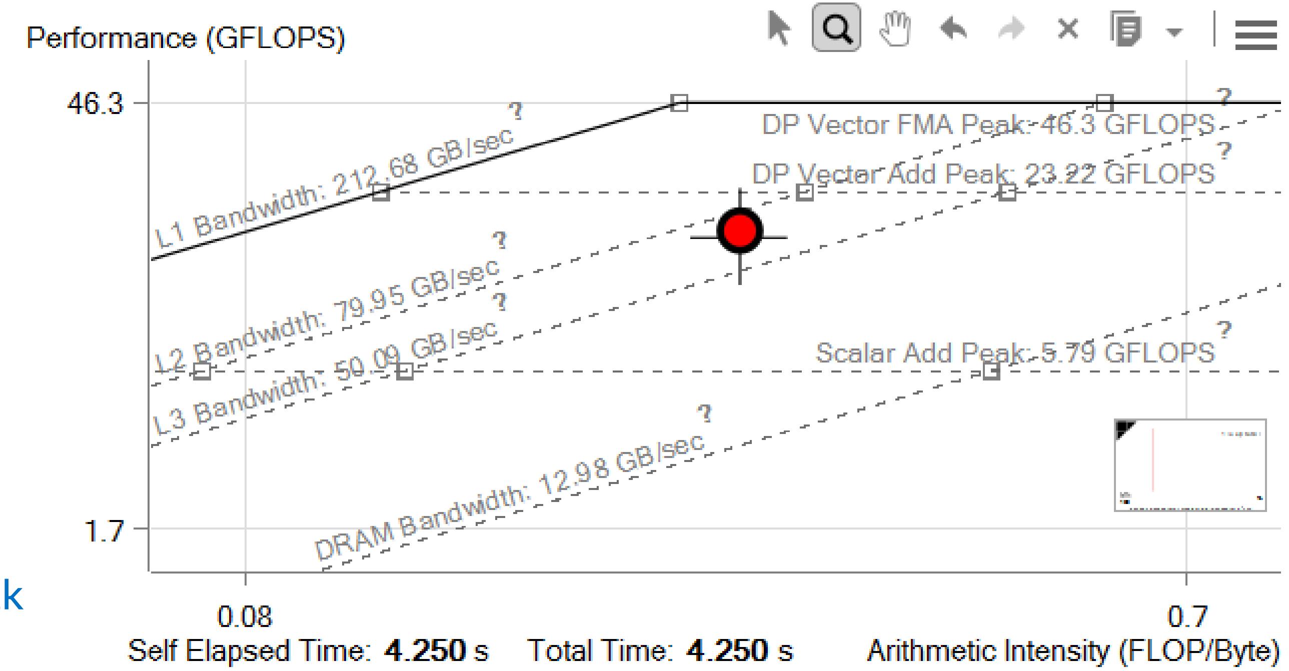


A New Data Layout

A Short Walk Through the Process

Changing Y to SoA layout moved performance up again.

```
vector<double> X(SIZE);
typedef struct SoA
{
double a[SIZE];
double b[SIZE];
double pad1[SIZE];
double pad2[SIZE];
} SoA;
SoA Y;
```

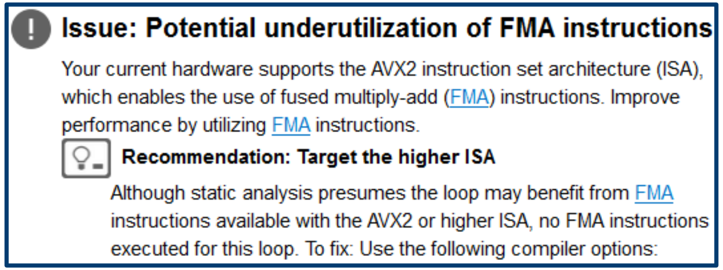


Either the Vector Add Peak or L2 Bandwidth could be the problem now.

Improving the Instruction Set

A Short Walk Through the Process

Because it's so close to an intersection, it's hard to tell whether the Bandwidth or Computation roof is the bottleneck. Checking the Recommendations tab guides us to recompile with a flag for AVX2 vector instructions.

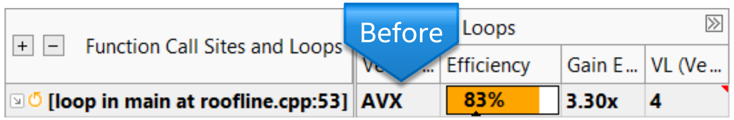


Issue: Potential underutilization of FMA instructions

Your current hardware supports the AVX2 instruction set architecture (ISA), which enables the use of fused multiply-add (FMA) instructions. Improve performance by utilizing FMA instructions.

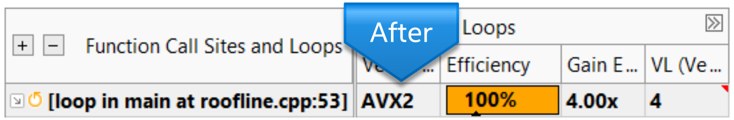
Recommendation: Target the higher ISA

Although static analysis presumes the loop may benefit from FMA instructions available with the AVX2 or higher ISA, no FMA instructions executed for this loop. To fix: Use the following compiler options:



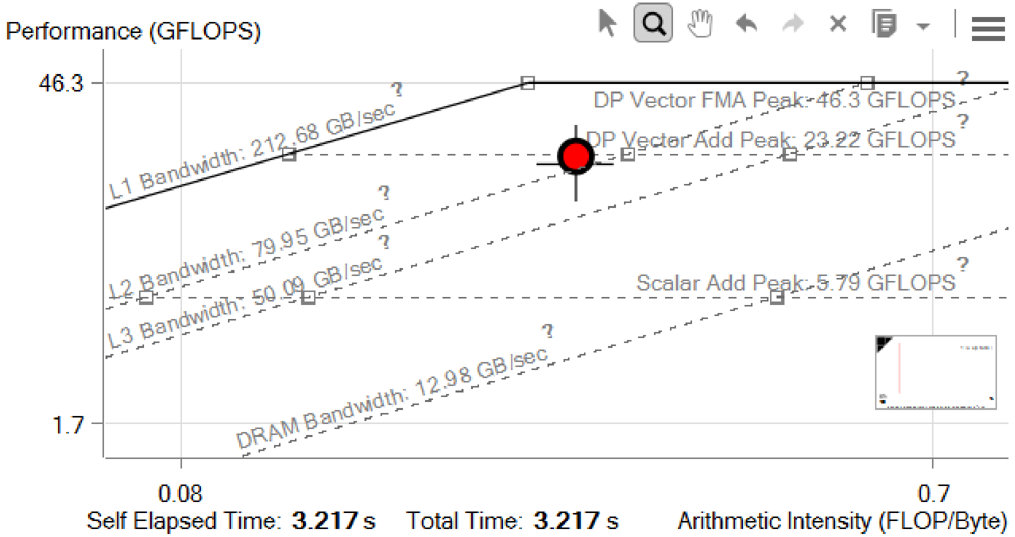
Before

Function Call Sites and Loops	Ve...	Efficiency	Gain E...	VL (Ve...
[loop in main at roofline.cpp:53]	AVX	83%	3.30x	4



After

Function Call Sites and Loops	Ve...	Efficiency	Gain E...	VL (Ve...
[loop in main at roofline.cpp:53]	AVX2	100%	4.00x	4



Self Elapsed Time: 3.217 s Total Time: 3.217 s

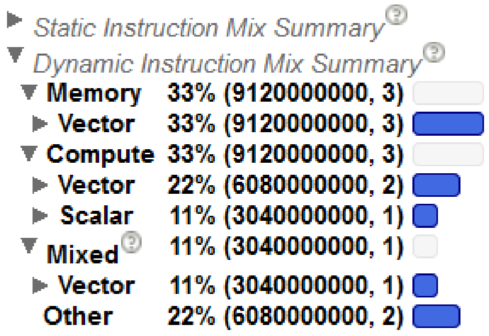
Assembly Detective Work

A Short Walk Through the Process

The dot is now sitting directly on the Vector Add Peak, so it is meeting but not exceeding the machine's vector capabilities. The next roof is the FMA peak. The Assembly tab shows that the loop is making good use of FMAs, too.

The Code Analytics tab reveals an unexpectedly high percentage of scalar compute instructions.

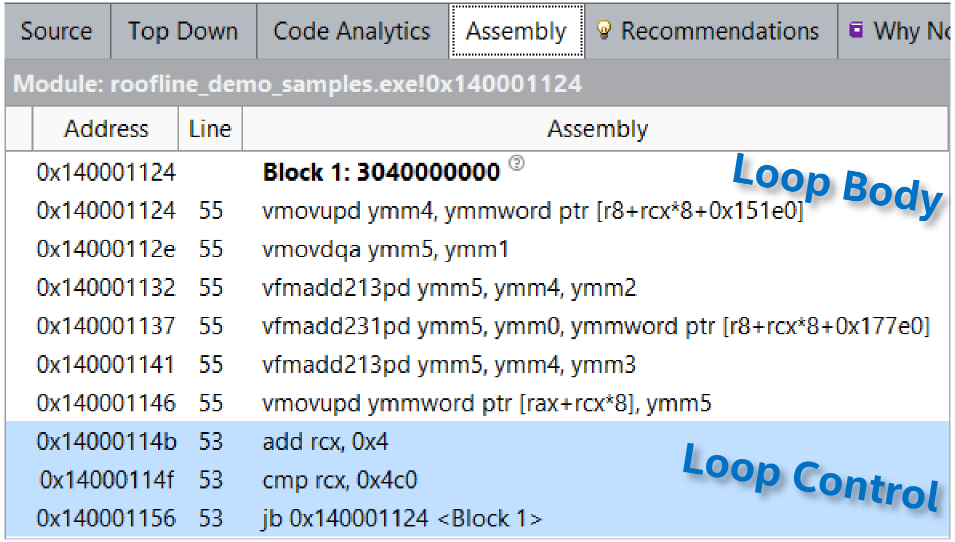
The only scalar math op present is in the loop control.



▶ *Static Instruction Mix Summary*

▼ *Dynamic Instruction Mix Summary*

▼ **Memory**	**33% (9120000000, 3)**
▶ **Vector**	**33% (9120000000, 3)**
▼ **Compute**	**33% (9120000000, 3)**
▶ **Vector**	**22% (6080000000, 2)**
▶ **Scalar**	**11% (3040000000, 1)**
▼ **Mixed**	**11% (3040000000, 1)**
▶ **Vector**	**11% (3040000000, 1)**
Other	**22% (6080000000, 2)**



Module: roofline_demo_samples.exe!0x140001124

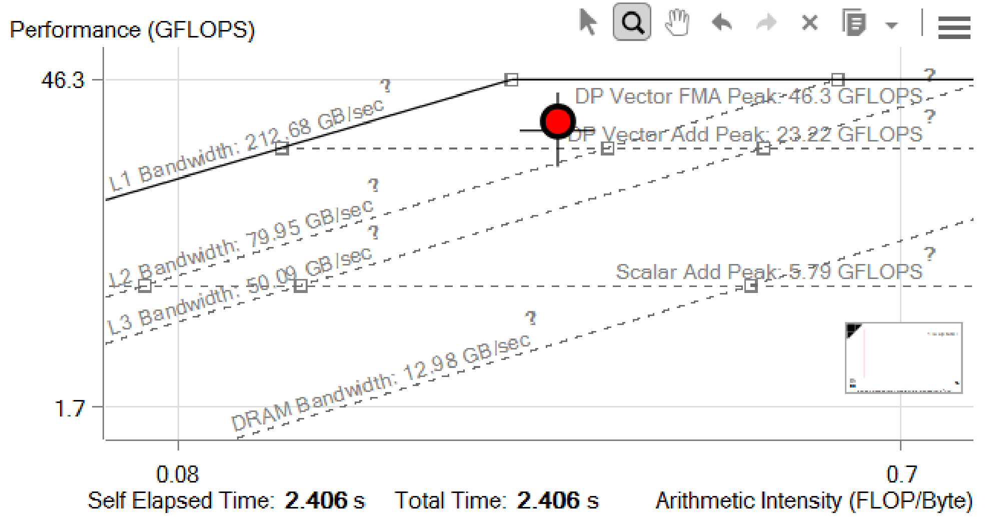
Address	Line	Assembly
0x140001124		**Block 1: 3040000000**
0x140001124	55	vmovupd ymm4, ymmword ptr [r8+rcx*8+0x151e0]
0x14000112e	55	vmovdqa ymm5, ymm1
0x140001132	55	vfmadd213pd ymm5, ymm4, ymm2
0x140001137	55	vfmadd231pd ymm5, ymm0, ymmword ptr [r8+rcx*8+0x177e0]
0x140001141	55	vfmadd213pd ymm5, ymm4, ymm3
0x140001146	55	vmovupd ymmword ptr [rax+rcx*8], ymm5
0x14000114b	53	add rcx, 0x4
0x14000114f	53	cmp rcx, 0x4c0
0x140001156	53	jb 0x140001124 <Block 1>

One More Optimization

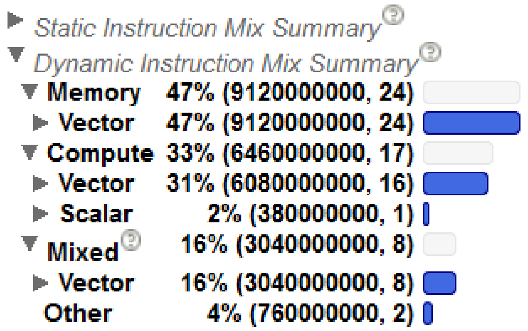
A Short Walk Through the Process

Scalar instructions in the loop control are slowing the loop down.

Unrolling a loop duplicates its body multiple times per iteration, so control makes up proportionately less of the loop.

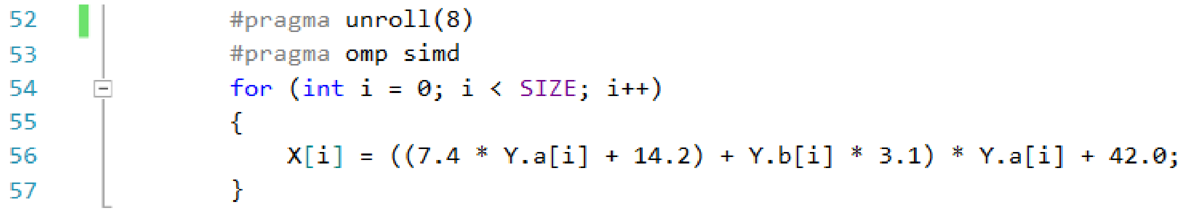


Self Elapsed Time: 2.406 s   Total Time: 2.406 s



▶ *Static Instruction Mix Summary*
▼ *Dynamic Instruction Mix Summary*

▼ Memory	47% (9120000000, 24)
▶ Vector	47% (9120000000, 24)
▼ Compute	33% (6460000000, 17)
▶ Vector	31% (6080000000, 16)
▶ Scalar	2% (380000000, 1)
▼ Mixed	16% (3040000000, 8)
▶ Vector	16% (3040000000, 8)
Other	4% (760000000, 2)



```
#pragma unroll(8)
#pragma omp simd
for (int i = 0; i < SIZE; i++)
{
    X[i] = ((7.4 * Y.a[i] + 14.2) + Y.b[i] * 3.1) * Y.a[i] + 42.0;
}
```

Recap

A Short Walk Through the Process

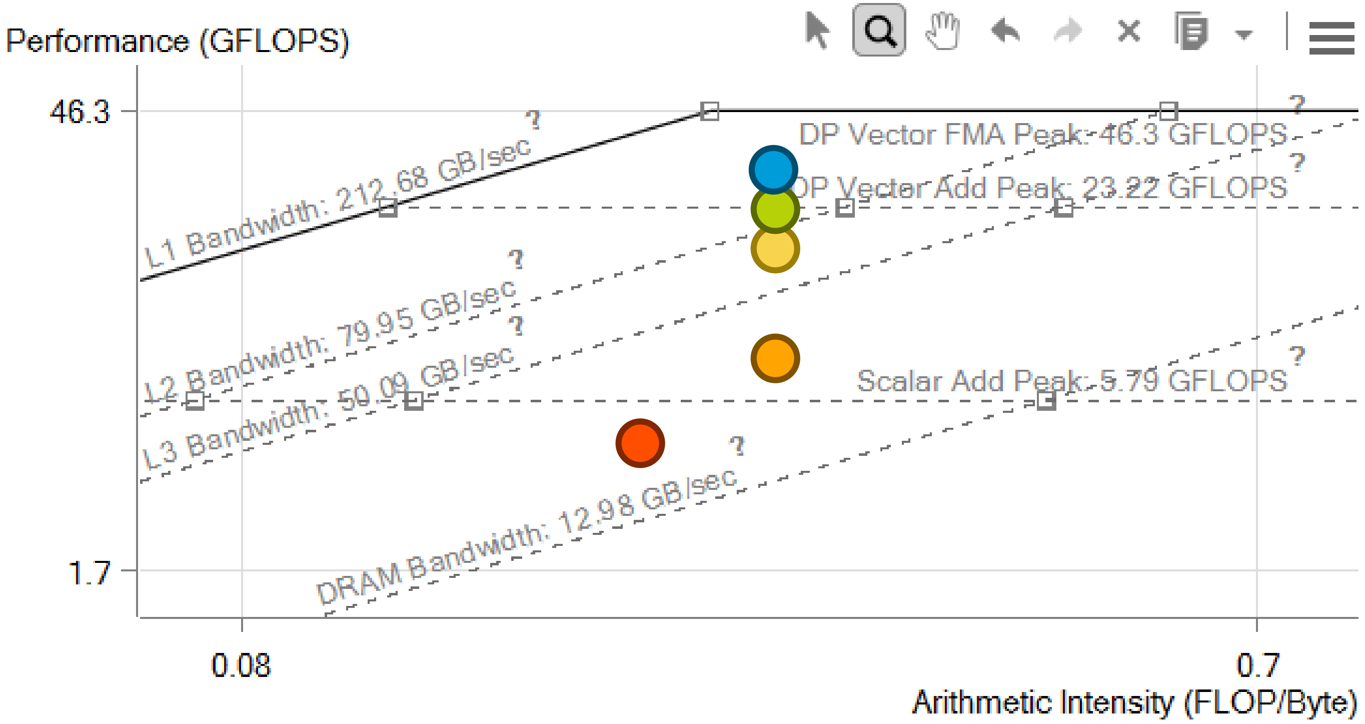
17.156s
Original scalar loop.

9.233s
Vectorized with a pragma.

4.250s
Switched from AoS to SoA.

3.217s
Compiled for AVX2.

2.406s
Unrolled with a pragma.

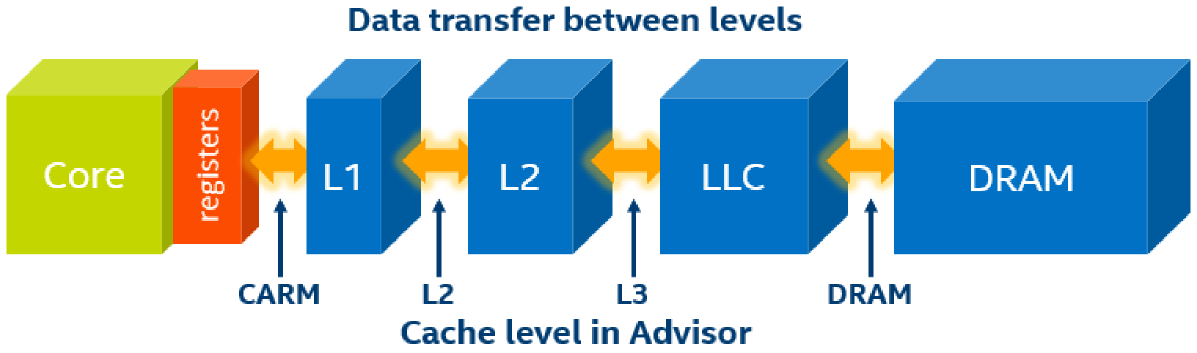


INTEGRATED ROOFLINE

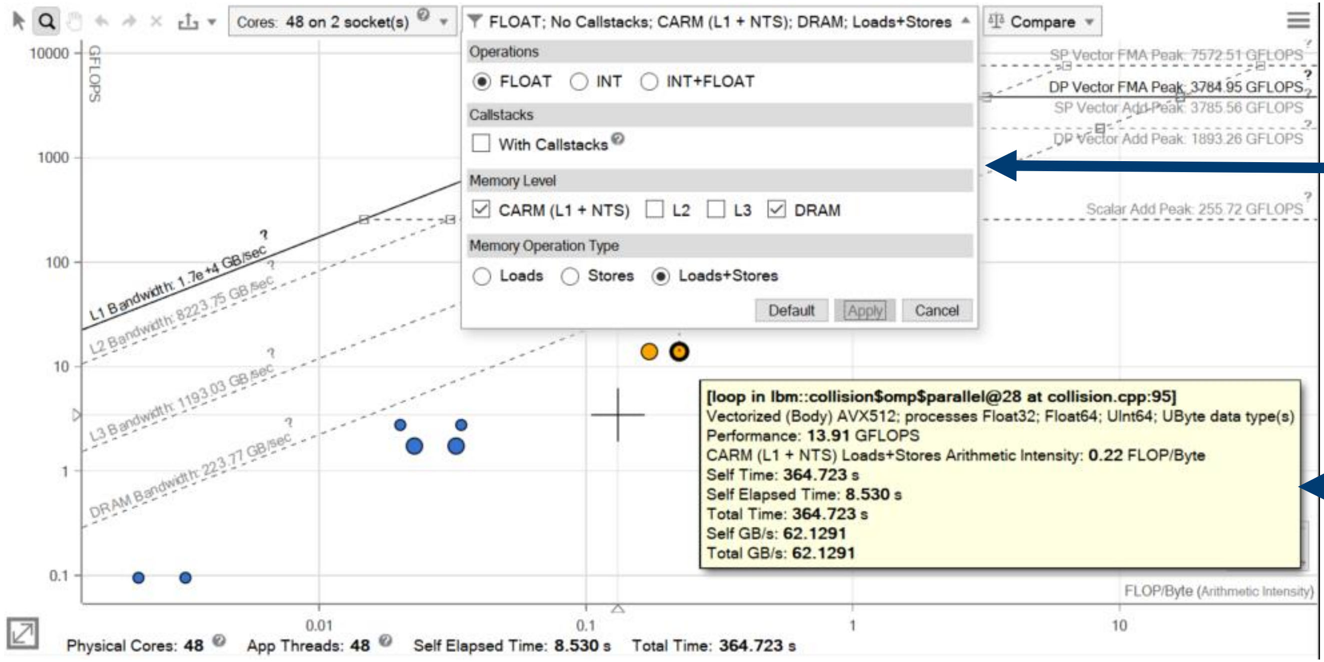
Beyond CARM: Integrated Roofline

New capability in Intel® Advisor: use simulation based method to estimate specific traffic across memory hierarchies.

- Record load/store instructions
- Use knowledge of processor cache structure and size
- Produce estimates of traffic generated at each level by individuals loops/functions



Integrated Roofline Representation


Cores: 48 on 2 socket(s)
FLOAT; No Callstacks; CARM (L1 + NTS); DRAM; Loads+Stores
Compare
GFLOPS
10000
1000
100
10
1
0.1
0.01
0.1
1
10
FLOP/Byte (Arithmetic Intensity)
SP Vector FMA Peak: 7572.51 GFLOPS
DP Vector FMA Peak: 3784.95 GFLOPS
SP Vector Add Peak: 3785.56 GFLOPS
DP Vector Add Peak: 1893.26 GFLOPS
Scalar Add Peak: 255.72 GFLOPS
L1 Bandwidth: 1.7e+4 GB/sec
L2 Bandwidth: 8223.75 GB/sec
L3 Bandwidth: 1193.03 GB/sec
DRAM Bandwidth: 223.77 GB/sec
Default
Apply
Cancel

Operations

[✓ FLOAT] INT INT+FLOAT

Callstacks

☐ With Callstacks

Memory Level

☑ CARM (L1 + NTS) ☐ L2 ☐ L3 ☑ DRAM

Memory Operation Type

Loads Stores [✓ Loads+Stores]

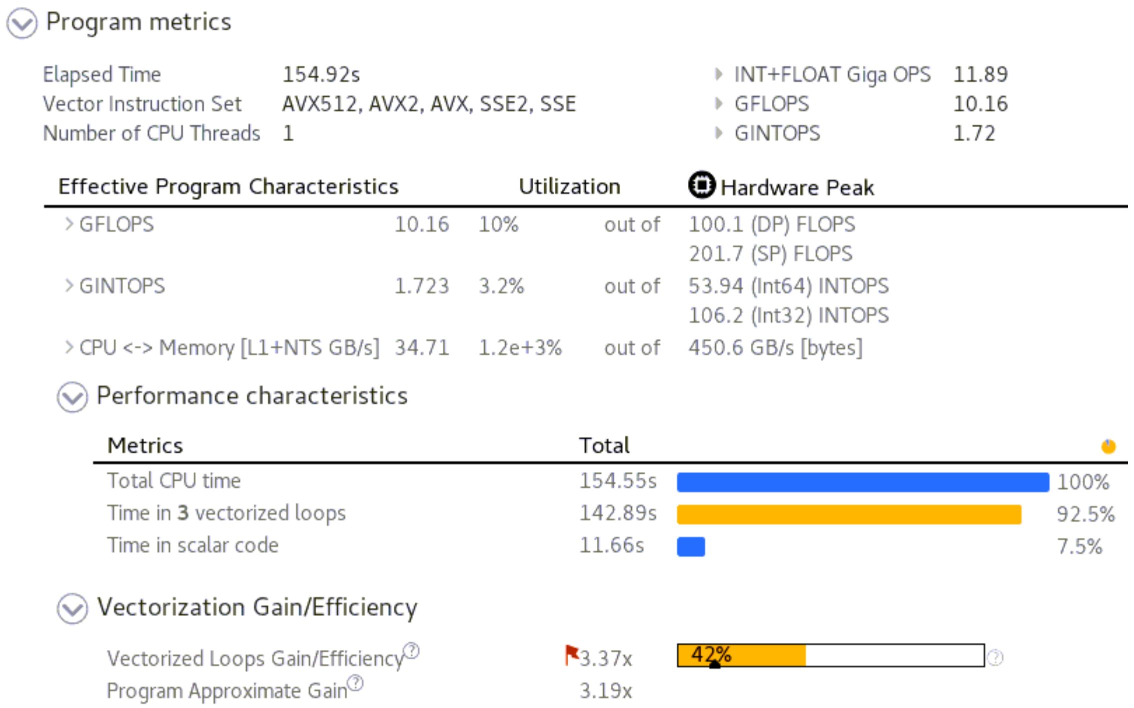
[loop in lbm::collision$omp$parallel@28 at collision.cpp:95]
Vectorized (Body) AVX512; processes Float32; Float64; UInt64; UByte data type(s)
Performance: **13.91** GFLOPS
CARM (L1 + NTS) Loads+Stores Arithmetic Intensity: **0.22** FLOP/Byte
Self Time: **364.723** s
Self Elapsed Time: **8.530** s
Total Time: **364.723** s
Self GB/s: **62.1291**
Total GB/s: **62.1291**

Physical Cores: **48** App Threads: **48** Self Elapsed Time: **8.530** s Total Time: **364.723** s

Choose memory level

Hover for details

New and improved summary



Program metrics

Elapsed Time	154.92s
Vector Instruction Set	AVX512, AVX2, AVX, SSE2, SSE
Number of CPU Threads	1

INT+FLOAT Giga OPS	11.89
GFLOPS	10.16
GINTOPS	1.72

Effective Program Characteristics		Utilization		Hardware Peak
GFLOPS	10.16	10%	out of	100.1 (DP) FLOPS 201.7 (SP) FLOPS
GINTOPS	1.723	3.2%	out of	53.94 (Int64) INTOPS 106.2 (Int32) INTOPS
CPU <-> Memory [L1+NTS GB/s]	34.71	1.2e+3%	out of	450.6 GB/s [bytes]

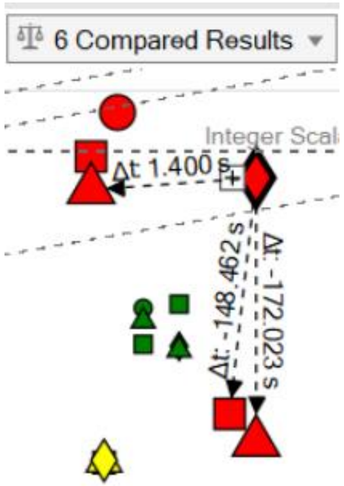
Performance characteristics

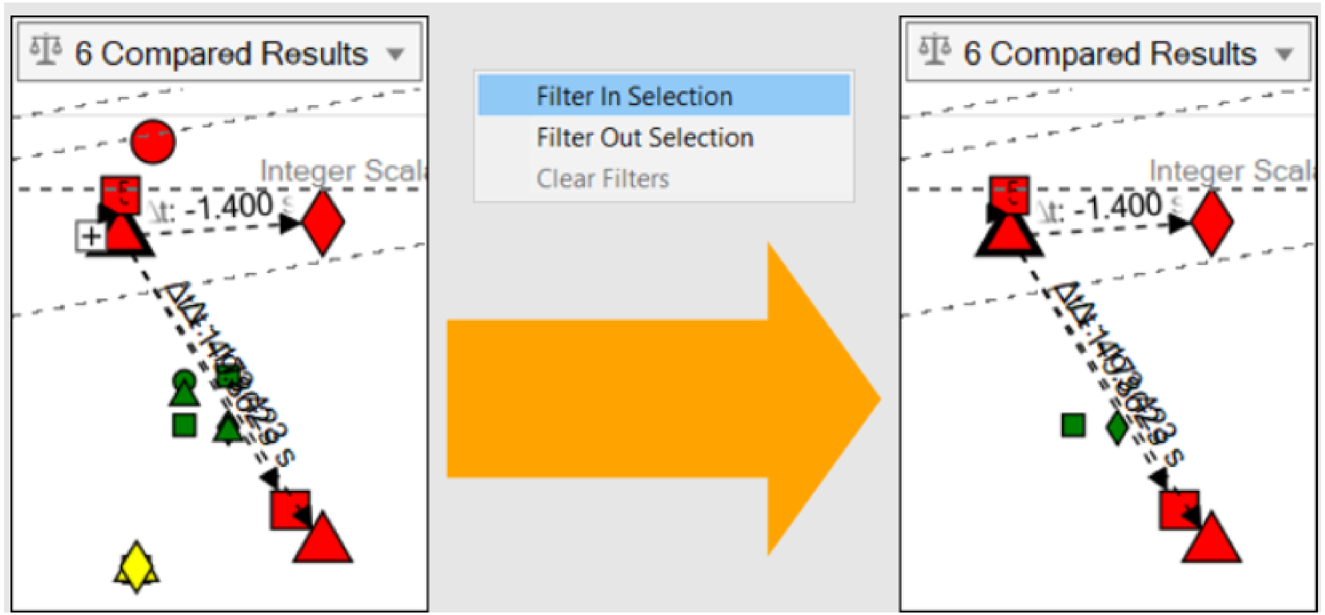
Metrics	Total	
Total CPU time	154.55s	100%
Time in **3** vectorized loops	142.89s	92.5%
Time in scalar code	11.66s	7.5%

Vectorization Gain/Efficiency

Vectorized Loops Gain/Efficiency	3.37x	42%
Program Approximate Gain	3.19x	

Roofline compare





FLOW GRAPH ANALYZER

Flow Graph Analyzer

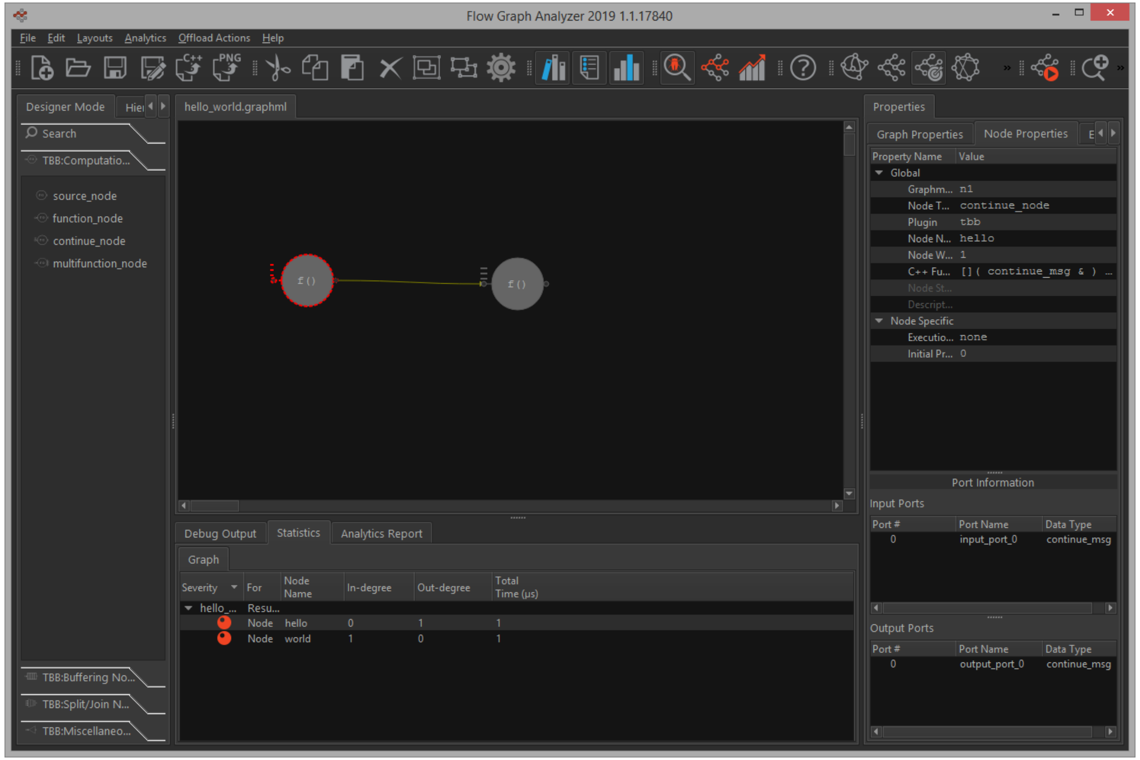
Workflows: Create, Debug, Visualize and Analyze

Design mode

- Allows you to create a graph topology interactively
- Validate the graph and explore what-if scenarios
- Add C/C++ code to the node body
- Export C++ code using Threading Building Blocks (TBB) flow graph API

Analysis mode

- Compile your application (with tracing enabled)
- Capture execution traces during the application run
- Visualize/analyze in Flow Graph Analyzer
- Works with TBB and OpenMP



Property Name	Value
Global	
Graphm...	n1
Node T...	continue_node
Plugin	tbb
Node N...	hello
Node W...	1
C++ Fu...	[] (continue_msg &) ...
Node St...	
Descript...	
Node Specific	
Executio...	none
Initial Pr...	0

Input Ports

Port #	Port Name	Data Type
0	input_port_0	continue_msg

Output Ports

Port #	Port Name	Data Type
0	output_port_0	continue_msg

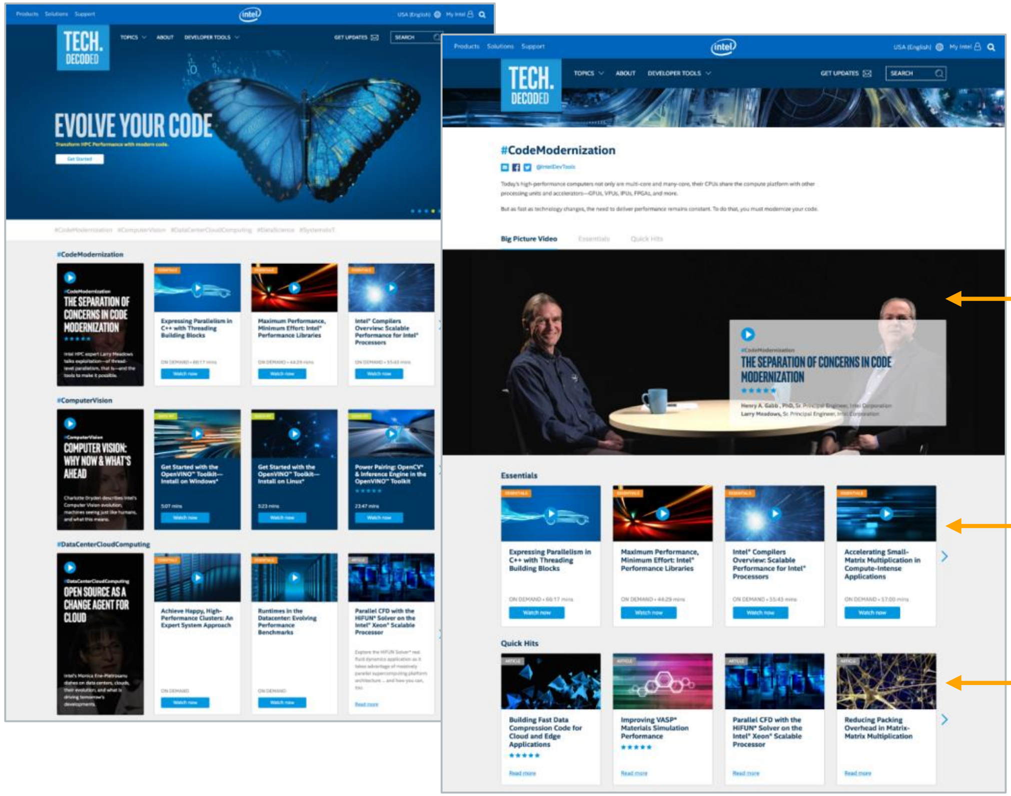
Severity	For	Node Name	In-degree	Out-degree	Total Time (µs)
hello_...	Resu...				
●	Node	hello	0	1	1
●	Node	world	1	0	1

Summary

2nd gen Intel® Xeon® Scalable processors have more performance capacity than ever before, but code needs to be written to take advantage of it!

- Build a good foundation
  - Use the right compiler flags and libraries
  - Write your application to make good use of multithreading
    - Use **Intel® Advisor** to plan your threading
    - Use Intel® VTune™ Amplifier's **Threading** analysis to optimize your threading
- Tune to the architecture with performance profiling tools.
  - Find your hotspots with VTune™ Amplifier's **Hotspots** analysis type.
  - Diagnose your bottlenecks with the **Microarchitecture Exploration** analysis type
    - Dig deeper with a **Memory Access** analysis or **Intel® Advisor**
  - Implement solutions based on your findings
    - Use **Intel® Inspector** to make good use of Intel® Optane™ DC Persistent Memory

Get the Most from Your Code Today with Intel® Tech.Decoded



#CodeModernization

@IntelDevTools

Today's high-performance computers not only are multi-core and many-core, their CPUs share the compute platform with other processing units and accelerators—GPUs, VPUs, IPUs, FPGAs, and more.

But as fast as technology changes, the need to deliver performance remains constant. To do that, you must modernize your code.

#CodeModernization
THE SEPARATION OF CONCERNS IN CODE MODERNIZATION
Henry A. Gabb, PhD, Sr. Principal Engineer, Intel Corporation
Larry Meadows, Sr. Principal Engineer, Intel Corporation

Essentials

- Expressing Parallelism in C++ with Threading Building Blocks — ON DEMAND • 66:17 mins
- Maximum Performance, Minimum Effort: Intel® Performance Libraries — ON DEMAND • 44:29 mins
- Intel® Compilers Overview: Scalable Performance for Intel® Processors — ON DEMAND • 55:43 mins
- Accelerating Small-Matrix Multiplication in Compute-Intense Applications — ON DEMAND • 17:00 mins

Quick Hits

- Building Fast Data Compression Code for Cloud and Edge Applications
- Improving VASP* Materials Simulation Performance
- Parallel CFD with the HiFUN* Solver on the Intel® Xeon® Scalable Processor
- Reducing Packing Overhead in Matrix-Matrix Multiplication

Visit **TechDecoded.intel.io** to learn how to put key optimization strategies into practice with Intel development tools.

Big Picture Videos

Discover Intel's vision for key development areas.

Essential Webinars

Gain strategies, practices and tools to optimize application and solution performance.

Quick Hit How-To Videos

Learn how to do specific programming tasks using Intel® tools.

TOPICS:

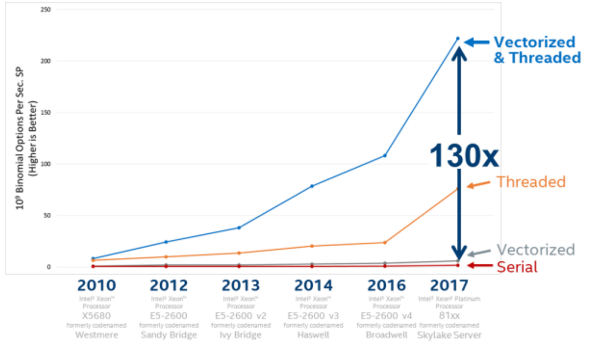
- Visual Computing
- Code Modernization
- Systems & IoT
- Data Science
- Data Center & Cloud

Legal Disclaimer & Optimization Notice

Configurations for 2010-2017 Benchmarks

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Performance measured in Intel Labs by Intel employees



Platform Hardware and Software Configuration

	Platform	Unscaled Core Frequency	Cores/ Socket	Num Sockets	L1 Data Cache	L2 Cache	L3 Cache	Memory	Memory Frequency	Memory Access	H/W Prefetchers Enabled	HT Enabled	Turbo Enabled	C States	O/S Name	Operating System	Compiler Version
WSM	Intel® Xeon™ X5680 Processor	3.33 GHZ	6	2	32K	256K	12 MB	48 MB	1333 MHz	NUMA	Y	Y	Y	Disabled	Fedora 20	3.11.10-301.fc20	icc version 17.0.2
SNB	Intel® Xeon™ E5 2690 Processor	2.9 GHZ	8	2	32K	256K	20 MB	64 GB	1600 MHz	NUMA	Y	Y	Y	Disabled	Fedora 20	3.11.10-301.fc20	icc version 17.0.2
IVB	Intel® Xeon™ E5 2697v2 Processor	2.7 GHZ	12	2	32K	256K	30 MB	64 GB	1867 MHz	NUMA	Y	Y	Y	Disabled	RHEL 7.1	3.10.0-229.el7.x86_64	icc version 17.0.2
HSW	Intel® Xeon™ E5 2600v3 Processor	2.2 GHz	18	2	32K	256K	46 MB	128 GB	2133 MHz	NUMA	Y	Y	Y	Disabled	Fedora 20	3.15.10-200.fc20.x86_64	icc version 17.0.2
BDW	Intel® Xeon™ E5 2600v4 Processor	2.3 GHz	18	2	32K	256K	46 MB	256 GB	2400 MHz	NUMA	Y	Y	Y	Disabled	RHEL 7.0	3.10.0-123. el7.x86_64	icc version 17.0.2
BDW	Intel® Xeon™ E5 2600v4 Processor	2.2 GHz	22	2	32K	256K	56 MB	128 GB	2133 MHz	NUMA	Y	Y	Y	Disabled	CentOS 7.2	3.10.0-327. el7.x86_64	icc version 17.0.2
SKX	Intel® Xeon® Platinum 81xx Processor	2.5 GHz	28	2	32K	1024K	40 MB	192 GB	2666 MHz	NUMA	Y	Y	Y	Disabled	CentOS 7.3	3.10.0-514.10.2.el7.x86_64	icc version 17.0.2

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance

TECH.
DECODED

intel
Software