Software

OPENMP* ANALYSIS IN INTEL® VTUNE™ AMPLIFIER XE - TALKING TO A USER ABOUT OPENMP* PERFORMANCE IN THE LANGUAGE THE PROGRAM WAS WRITTEN IN, WITH A LITTLE WANDER INTO VECTORIZATION TOO

Material from Dmitry Prohorov, VTune HPC lead
Zakhar Matveev (Intel® Advisor Architect) and Alex Shinsel
Presented by Jim Cownie

Agenda

VTune Amplifier XE OpenMP* Analysis: answering customers' questions about performance in the same language their program was written in

- Concepts, metrics and technology inside
- VTune Amplifier XE OpenMP Analysis Workflow

A short introduction to Intel® Advisor's Roofline Analysis

Summary

Typical customer questions on parallelization efficiency of OpenMP* applications

"I put pragmas but why is my speed up so poor?"

- Parallelization inefficiency

"I ran my app on a system with more cores but it doesn't run as efficiently as on a smaller one"

- Scalability issues

Decomposing the questions

Is the serial time of my application significant in preventing scaling?

How efficient is my OpenMP parallelization?

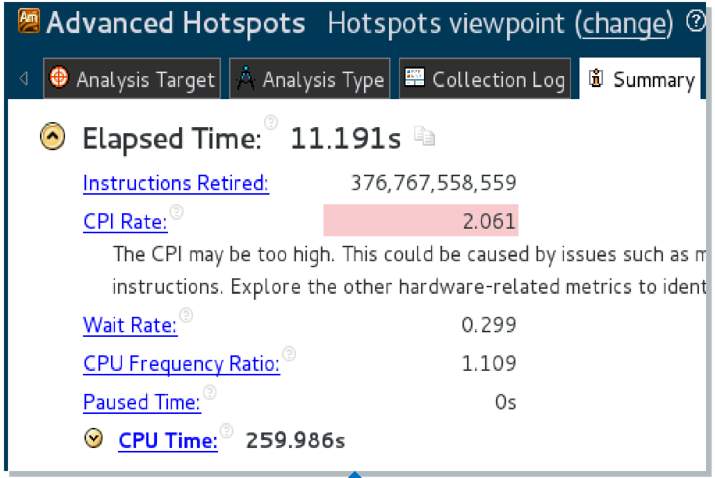
- If inefficient, how much gain can be achieved if I invest in fighting the inefficiencies?

Which OpenMP regions/loops/barriers are worth tuning?

- What are their particular problems?

If Performance Information is OpenMP “Unaware”..

The questions are tied to OpenMP program structure – #pragmas
Answers should be given the same way to be understandable and actionable



Elapsed Time: 11.191s

Instructions Retired: 376,767,558,559

CPI Rate: 2.061

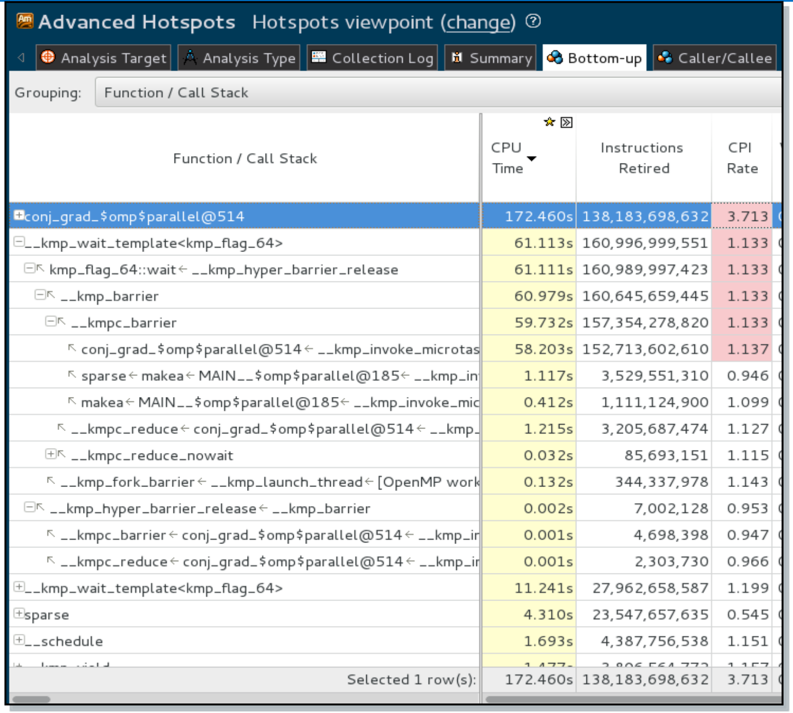
The CPI may be too high. This could be caused by issues such as m
instructions. Explore the other hardware-related metrics to ident

Wait Rate: 0.299

CPU Frequency Ratio: 1.109

Paused Time: 0s

CPU Time: 259.986s



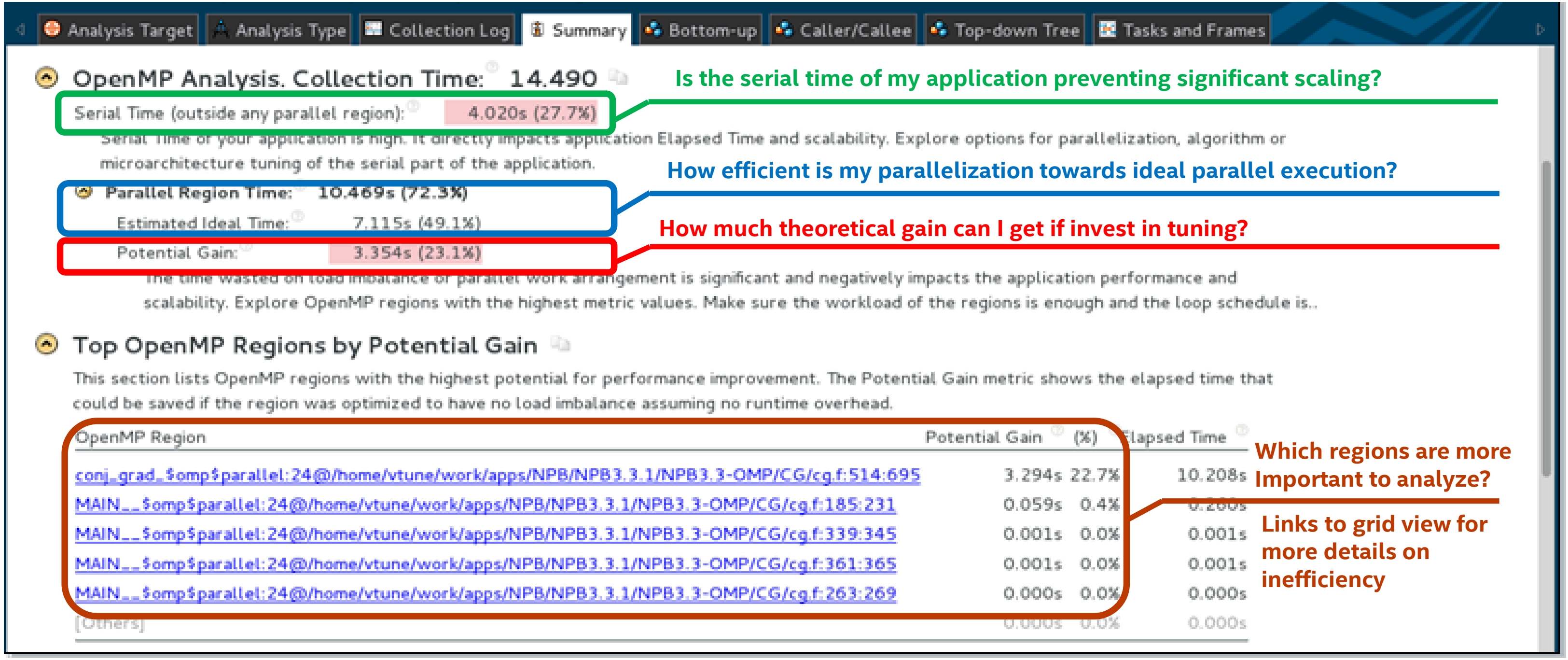
Grouping: Function / Call Stack

Function / Call Stack	CPU Time	Instructions Retired	CPI Rate
conj_grad_ompparallel@514	172.460s	138,183,698,632	3.713
__kmp_wait_template<kmp_flag_64>	61.113s	160,996,999,551	1.133
kmp_flag_64::wait ← __kmp_hyper_barrier_release	61.111s	160,989,997,423	1.133
__kmp_barrier	60.979s	160,645,659,445	1.133
__kmpc_barrier	59.732s	157,354,278,820	1.133
conj_grad_ompparallel@514 ← __kmp_invoke_microtas	58.203s	152,713,602,610	1.137
sparse ← makea ← MAIN__ompparallel@185 ← __kmp_in	1.117s	3,529,551,310	0.946
makea ← MAIN__ompparallel@185 ← __kmp_invoke_mic	0.412s	1,111,124,900	1.099
__kmpc_reduce ← conj_grad_ompparallel@514 ← __kmp_	1.215s	3,205,687,474	1.127
__kmpc_reduce_nowait	0.032s	85,693,151	1.115
__kmp_fork_barrier ← __kmp_launch_thread ← [OpenMP work	0.132s	344,337,978	1.143
__kmp_hyper_barrier_release ← __kmp_barrier	0.002s	7,002,128	0.953
__kmpc_barrier ← conj_grad_ompparallel@514 ← __kmp_ir	0.001s	4,698,398	0.947
__kmpc_reduce ← conj_grad_ompparallel@514 ← __kmp_ir	0.001s	2,303,730	0.966
__kmp_wait_template<kmp_flag_64>	11.241s	27,962,658,587	1.199
sparse	4.310s	23,547,657,635	0.545
__schedule	1.693s	4,387,756,538	1.151
Selected 1 row(s):	172.460s	138,183,698,632	3.713

OpenMP “unaware” views of VTune Amplifier XE
Difficult to detect problems, customers blame the OpenMP runtime seeing CPU time consumption there and not understanding that this is a result of parallelization inefficiency

Optimization Notice

Overview of summary pane



OpenMP Analysis. Collection Time: 14.490

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

Serial Time of your application is high. It directly impacts application Elapsed Time and scalability. Explore options for parallelization, algorithm or microarchitecture tuning of the serial part of the application.

Parallel Region Time: 10.469s (72.3%)

Estimated Ideal Time: 7.115s (49.1%)

Potential Gain: 3.354s (23.1%)

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

Top OpenMP Regions by Potential Gain

This section lists OpenMP regions with the highest potential for performance improvement. The Potential Gain metric shows the elapsed time that could be saved if the region was optimized to have no load imbalance assuming no runtime overhead.

OpenMP Region	Potential Gain	(%)	Elapsed Time
conj_grad_ompparallel:24@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:514:695	3.294s	22.7%	10.208s
MAIN__ompparallel:24@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:185:231	0.059s	0.4%	0.260s
MAIN__ompparallel:24@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:339:345	0.001s	0.0%	0.001s
MAIN__ompparallel:24@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:361:365	0.001s	0.0%	0.001s
MAIN__ompparallel:24@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:263:269	0.000s	0.0%	0.000s
[Others]	0.000s	0.0%	0.000s

Is the serial time of my application preventing significant scaling?

How efficient is my parallelization towards ideal parallel execution?

How much theoretical gain can I get if invest in tuning?

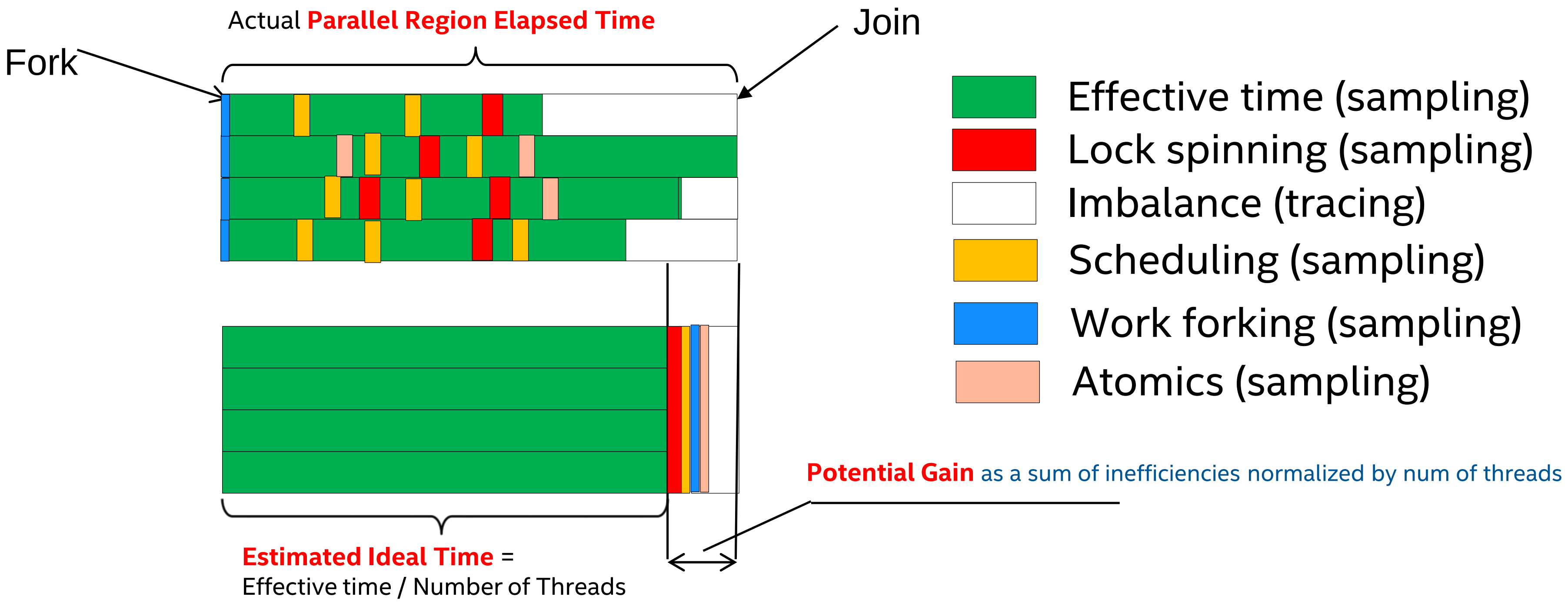
Which regions are more Important to analyze?

Links to grid view for more details on inefficiency

Optimization Notice

Key to OpenMP awareness in VTune – Region based views and metrics

Definition of Region **Potential Gain** (elapsed time metric)



Actual **Parallel Region Elapsed Time**

Fork

Join

- 🟩 Effective time (sampling)
- 🟥 Lock spinning (sampling)
- ⬜ Imbalance (tracing)
- 🟨 Scheduling (sampling)
- 🟦 Work forking (sampling)
- 🟧 Atomics (sampling)

Potential Gain as a sum of inefficiencies normalized by num of threads

Estimated Ideal Time =
Effective time / Number of Threads

Technology used by VTune Amplifier XE OpenMP Analysis

Tracing of OpenMP constructs to provide region/work sharing context and precise imbalance at barriers

- Provided to VTune by LLVM/Intel OpenMP Runtime under profiling
  - Fork-Join points of parallel regions with number of working threads (Intel Compiler 14 and later)
  - OpenMP construct barrier points with imbalance info and OpenMP loop metadata
  - `-parallel-source-info=2` Intel compiler option to embed source file name in region name

Looking at transition to OMPT, working with John M-C on interface enrichments for low overhead analysis

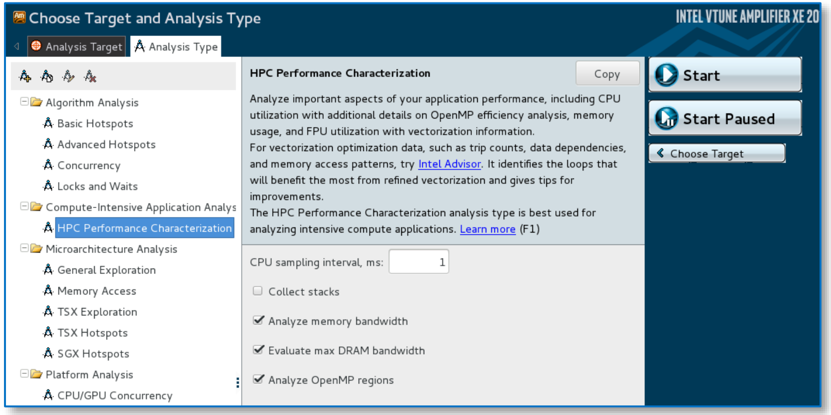
Sampling to define and classify CPU time - user's code and OpenMP RTL work

- Classification: Locking, Scheduling, Work Forking

VTune Amplifier XE OpenMP Analysis Workflow

Start with HPC Performance Characterization analysis

Explore CPU Utilization metrics related to OpenMP in summary, grid, and source views



Choose Target and Analysis Type

Analysis Target | Analysis Type

- Algorithm Analysis
  - Basic Hotspots
  - Advanced Hotspots
  - Concurrency
  - Locks and Waits
- Compute-Intensive Application Analys
  - HPC Performance Characterization
- Microarchitecture Analysis
  - General Exploration
  - Memory Access
  - TSX Exploration
  - TSX Hotspots
  - SGX Hotspots
- Platform Analysis
  - CPU/GPU Concurrency

HPC Performance Characterization

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

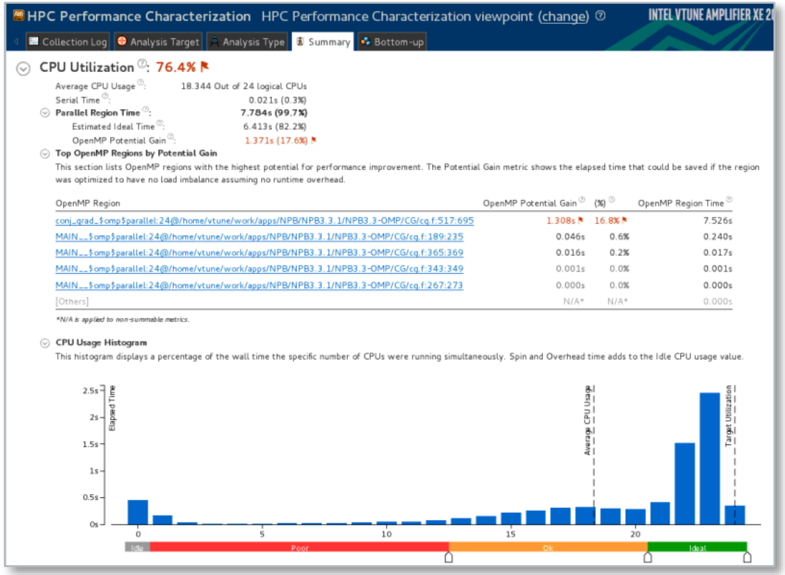
CPU sampling interval, ms: 1

☐ Collect stacks

☑ Analyze memory bandwidth

☑ Evaluate max DRAM bandwidth

☑ Analyze OpenMP regions



HPC Performance Characterization   HPC Performance Characterization viewpoint (change)

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

CPU Utilization: 76.4%

```
Average CPU Usage:        18.344 Out of 24 logical CPUs
Serial Time:                     0.021s (0.3%)
Parallel Region Time:            7.784s (99.7%)
    Estimated Ideal Time:        6.413s (82.2%)
    OpenMP Potential Gain:       1.371s (17.6%)
```

Top OpenMP Regions by Potential Gain

This section lists OpenMP regions with the highest potential for performance improvement. The Potential Gain metric shows the elapsed time that could be saved if the region was optimized to have no load imbalance assuming no runtime overhead.

OpenMP Region	OpenMP Potential Gain	(%)	OpenMP Region Time
conj_grad_ompparallel:24@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:517:695	1.308s	16.8%	7.526s
MAIN__ompparallel:24@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:189:235	0.046s	0.6%	0.240s
MAIN__ompparallel:24@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:365:369	0.016s	0.2%	0.017s
MAIN__ompparallel:24@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:343:349	0.001s	0.0%	0.001s
MAIN__ompparallel:24@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:267:273	0.000s	0.0%	0.000s
[Others]	N/A*	N/A*	0.000s

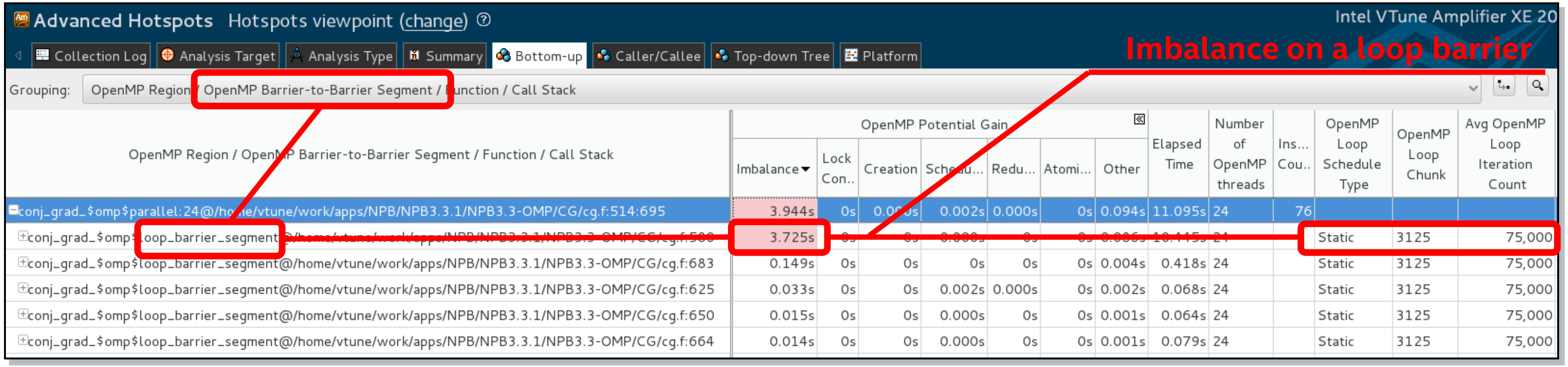
*N/A is applied to non-summable metrics.

CPU Usage Histogram

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

CL: *>amplxe-cl –collect hpc-performance <my_app>*

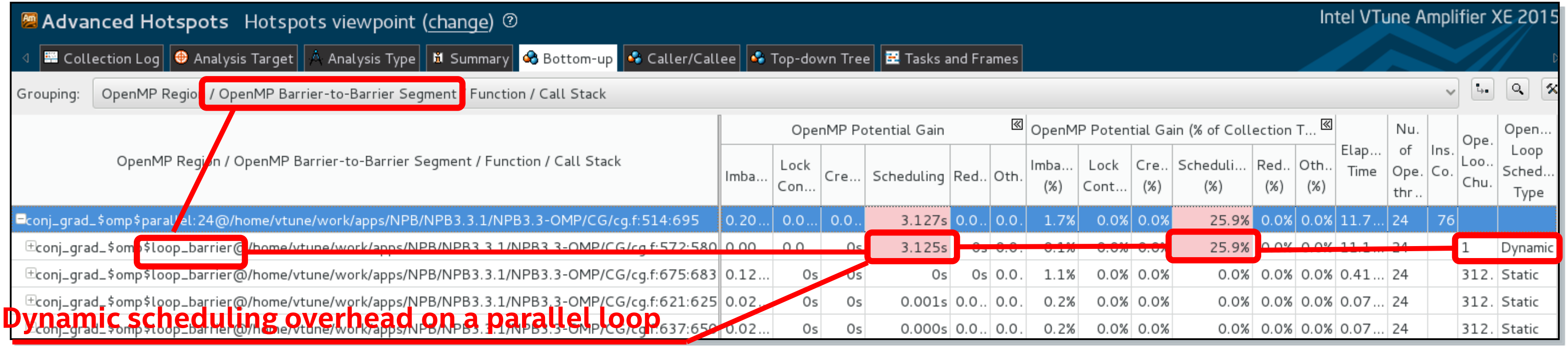
Per-region Details in grid view: inefficiencies in elapsed time are classified and highlighted



Imbalance on a loop barrier

Grouping: OpenMP Region / OpenMP Barrier-to-Barrier Segment / Function / Call Stack

OpenMP Region / OpenMP Barrier-to-Barrier Segment / Function / Call Stack	OpenMP Potential Gain							Elapsed Time	Number of OpenMP threads	Ins... Cou..	OpenMP Loop Schedule Type	OpenMP Loop Chunk	Avg OpenMP Loop Iteration Count
	Imbalance	Lock Con..	Creation	Schedu...	Redu...	Atomi...	Other						
conj_grad_ompparallel:24@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:514:695	3.944s	0s	0.000s	0.002s	0.000s	0s	0.094s	11.095s	24	76			
conj_grad_omploop_barrier_segment@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:580	3.725s	0s	0s	0.000s	0s	0s	0.086s	10.445s	24		Static	3125	75,000
conj_grad_omploop_barrier_segment@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:683	0.149s	0s	0s	0s	0s	0s	0.004s	0.418s	24		Static	3125	75,000
conj_grad_omploop_barrier_segment@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:625	0.033s	0s	0s	0.002s	0.000s	0s	0.002s	0.068s	24		Static	3125	75,000
conj_grad_omploop_barrier_segment@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:650	0.015s	0s	0s	0.000s	0s	0s	0.001s	0.064s	24		Static	3125	75,000
conj_grad_omploop_barrier_segment@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:664	0.014s	0s	0s	0.000s	0s	0s	0.001s	0.079s	24		Static	3125	75,000

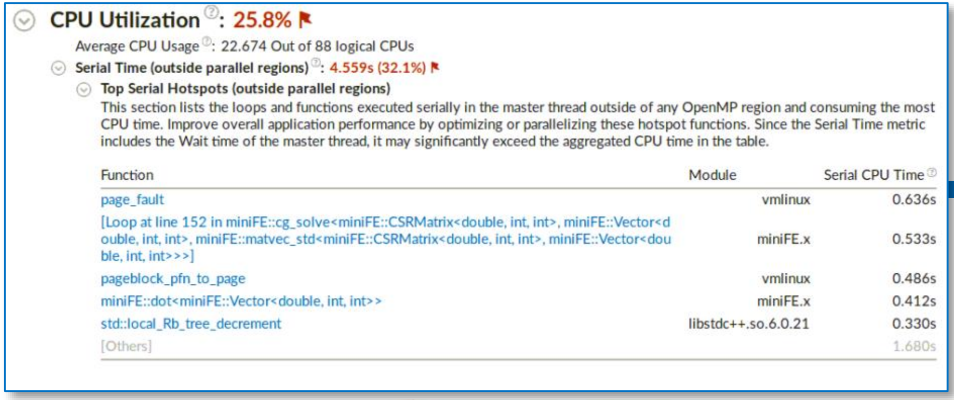


Grouping: OpenMP Region / OpenMP Barrier-to-Barrier Segment / Function / Call Stack

OpenMP Region / OpenMP Barrier-to-Barrier Segment / Function / Call Stack	OpenMP Potential Gain						OpenMP Potential Gain (% of Collection T...						Elap... Time	Nu. of Ope. thr..	Ins. Co.	Ope. Loo.. Chu.	Open... Loop Sched... Type
	Imba...	Lock Con...	Cre...	Scheduling	Red..	Oth.	Imba... (%)	Lock Cont...	Cre.. (%)	Scheduli... (%)	Red.. (%)	Oth.. (%)					
conj_grad_ompparallel:24@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:514:695	0.20...	0.0...	0.0..	3.127s	0.0..	0.0.	1.7%	0.0%	0.0%	25.9%	0.0%	0.0%	11.7...	24	76		
conj_grad_omploop_barrier@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:572:580	0.00	0.0	0s	3.125s	0s	0.0.	0.1%	0.0%	0.0%	25.9%	0.0%	0.0%	11.1...	24		1	Dynamic
conj_grad_omploop_barrier@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:675:683	0.12...	0s	0s	0s	0s	0.0.	1.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.41...	24		312.	Static
conj_grad_omploop_barrier@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:621:625	0.02...	0s	0s	0.001s	0.0..	0.0.	0.2%	0.0%	0.0%	0.0%	0.0%	0.0%	0.07...	24		312.	Static
conj_grad_omploop_barrier@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:637:650	0.02...	0s	0s	0.000s	0.0..	0.0.	0.2%	0.0%	0.0%	0.0%	0.0%	0.0%	0.07...	24		312.	Static

Dynamic scheduling overhead on a parallel loop

Details in Grid View: Serial Time Hotspots



CPU Utilization: 25.8%

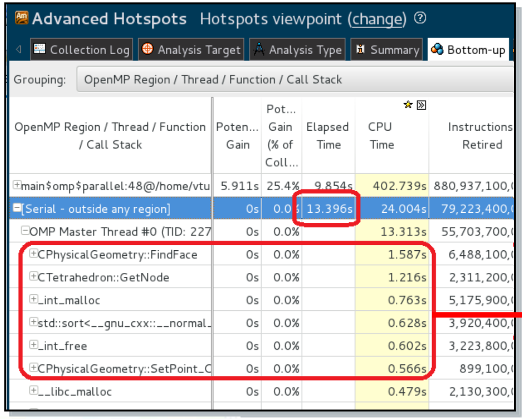
Average CPU Usage: 22.674 Out of 88 logical CPUs

Serial Time (outside parallel regions): 4.559s (32.1%)

Top Serial Hotspots (outside parallel regions)

This section lists the loops and functions executed serially in the master thread outside of any OpenMP region and consuming the most CPU time. Improve overall application performance by optimizing or parallelizing these hotspot functions. Since the Serial Time metric includes the Wait time of the master thread, it may significantly exceed the aggregated CPU time in the table.

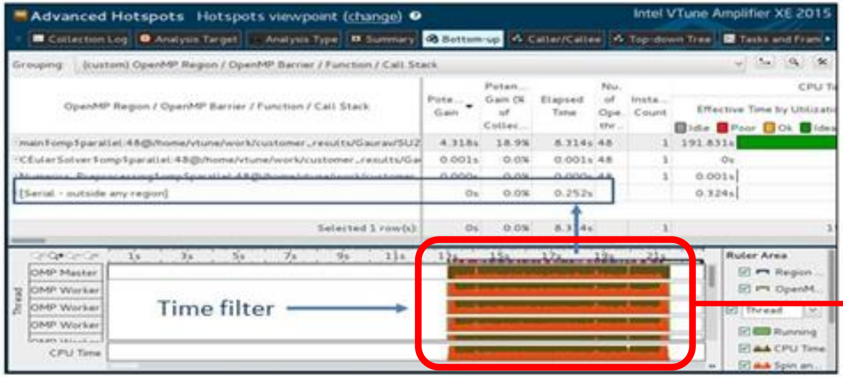
Function	Module	Serial CPU Time
page_fault	vmlinux	0.636s
[Loop at line 152 in miniFE::cg_solve<miniFE::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int>, miniFE::matvec_std<miniFE::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int>>>]	miniFE.x	0.533s
pageblock_pfn_to_page	vmlinux	0.486s
miniFE::dot<miniFE::Vector<double, int, int>>	miniFE.x	0.412s
std::local_Rb_tree_decrement	libstdc++.so.6.0.21	0.330s
[Others]		1.680s



Grouping: OpenMP Region / Thread / Function / Call Stack

OpenMP Region / Thread / Function / Call Stack	Poten... Gain	Pot... Gain (% of Coll...	Elapsed Time	CPU Time	Instructions Retired
main$omp$parallel:48@/home/vtu	5.911s	25.4%	9.854s	402.739s	880,937,100,(
[Serial - outside any region]	0s	0.0	13.396s	24.004s	79,223,400,(
OMP Master Thread #0 (TID: 227	0s	0.0%		13.313s	55,703,700,(
CPhysicalGeometry::FindFace	0s	0.0%		1.587s	6,488,100,(
CTetrahedron::GetNode	0s	0.0%		1.216s	2,311,200,(
_int_malloc	0s	0.0%		0.763s	5,175,900,(
std::sort<__gnu_cxx::__normal_	0s	0.0%		0.628s	3,920,400,(
_int_free	0s	0.0%		0.602s	3,223,800,(
CPhysicalGeometry::SetPoint_C	0s	0.0%		0.566s	899,100,(
__libc_malloc	0s	0.0%		0.479s	2,130,300,(

Serial hotspots under Master Thread



Grouping: (custom) OpenMP Region / OpenMP Barrier / Function / Call Stack

OpenMP Region / OpenMP Barrier / Function / Call Stack	Pote... Gain	Poten... Gain (% of Collec...	Elapsed Time	Nu... of Ope... thr...	Insta... Count	CPU Ti... Effective Time by Utilizatio...
main$omp$parallel:48@/home/vtune/work/customer_results/Gaurav/SU2	4.318s	18.9%	8.314s	48	1	191.831s
CEulerSolver$omp$parallel:48@/home/vtune/work/customer_results/Ga	0.001s	0.0%	0.001s	48	1	0s
Numerics_Preprocessing$omp$parallel:48@/home/vtune/work/customer_	0.000s	0.0%	0.000s	48	1	0.001s
[Serial - outside any region]	0s	0.0%	0.252s			0.324s
Selected 1 row(s):	0s	0.0%	8.314s		1	1

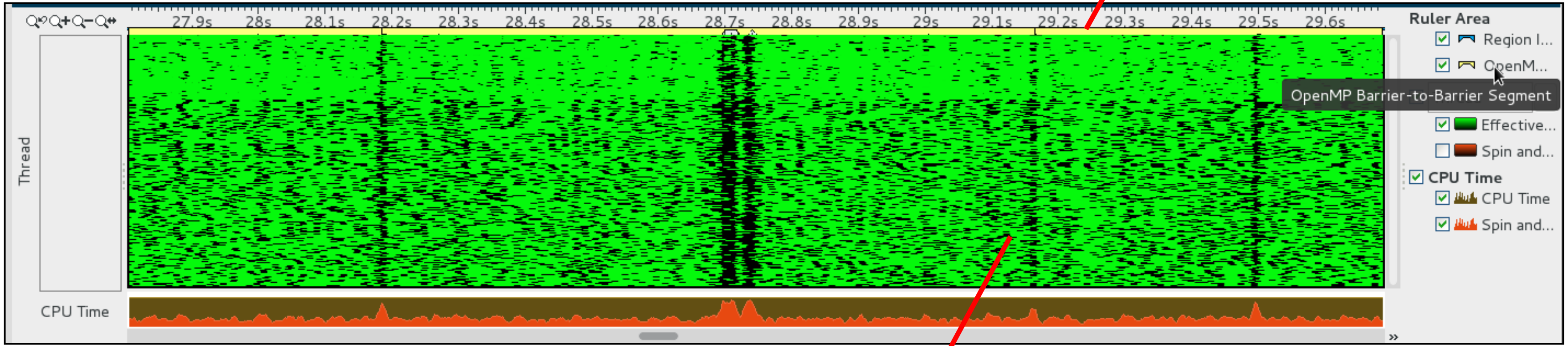
Time Filter to exclude initialization phase

[Optimization Notice](#)

Details on Scalable Timeline

Super tiny timeline display mode – a bird's eye view showing all data without scrolling

Region frames on the ruler

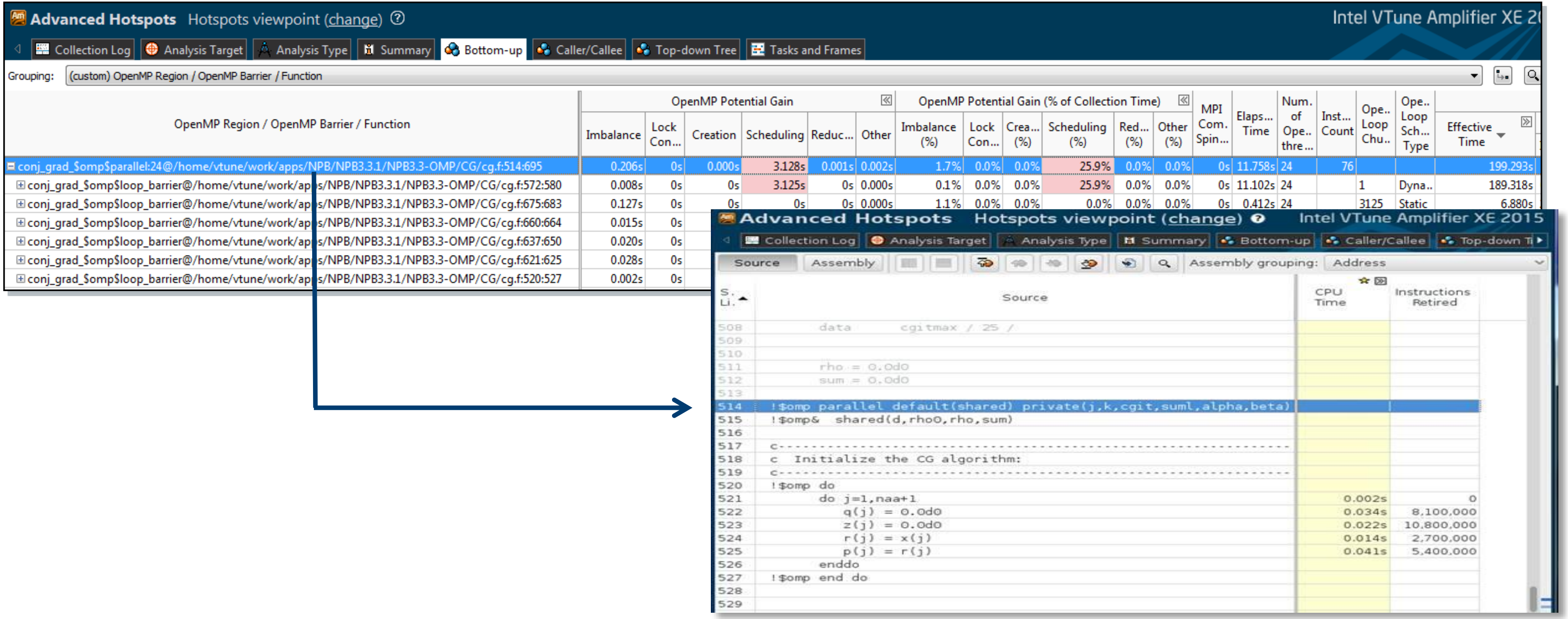


More green => more efficient multithreaded execution

Intel® Xeon Phi™ profiling result with 288 threads

Optimization Notice

Details for a Region at source file level



Grouping: (custom) OpenMP Region / OpenMP Barrier / Function

OpenMP Region / OpenMP Barrier / Function	OpenMP Potential Gain						OpenMP Potential Gain (% of Collection Time)						MPI Com. Spin...	Elaps... Time	Num. of Ope.. thre...	Inst... Count	Ope.. Loop Chu..	Ope.. Loop Sch... Type	Effective Time
	Imbalance	Lock Con...	Creation	Scheduling	Reduc...	Other	Imbalance (%)	Lock Con...	Crea... (%)	Scheduling (%)	Red... (%)	Other (%)							
⊟ conj_grad_ompparallel:24@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:514:695	0.206s	0s	0.000s	3.128s	0.001s	0.002s	1.7%	0.0%	0.0%	25.9%	0.0%	0.0%	0s	11.758s	24	76			199.293s
⊞ conj_grad_omploop_barrier@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:572:580	0.008s	0s	0s	3.125s	0s	0.000s	0.1%	0.0%	0.0%	25.9%	0.0%	0.0%	0s	11.102s	24		1	Dyna..	189.318s
⊞ conj_grad_omploop_barrier@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:675:683	0.127s	0s	0s	0s	0s	0.000s	1.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0s	0.412s	24		3125	Static	6.880s
⊞ conj_grad_omploop_barrier@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:660:664	0.015s	0s																	
⊞ conj_grad_omploop_barrier@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:637:650	0.020s	0s																	
⊞ conj_grad_omploop_barrier@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:621:625	0.028s	0s																	
⊞ conj_grad_omploop_barrier@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:520:527	0.002s	0s																	

Assembly grouping: Address

```
S.Li.  Source                                                              CPU Time   Instructions Retired
508          data      cgitmax / 25 /
509
510
511          rho = 0.0d0
512          sum = 0.0d0
513
514    !$omp parallel default(shared) private(j,k,cgit,suml,alpha,beta)
515    !$omp&  shared(d,rho0,rho,sum)
516
517    c---------------------------------------------------------------------
518    c  Initialize the CG algorithm:
519    c---------------------------------------------------------------------
520    !$omp do
521          do j=1,naa+1                                                   0.002s              0
522             q(j) = 0.0d0                                                0.034s      8,100,000
523             z(j) = 0.0d0                                                0.022s     10,800,000
524             r(j) = x(j)                                                 0.014s      2,700,000
525             p(j) = r(j)                                                 0.041s      5,400,000
526          enddo
527    !$omp end do
528
529
```

Intel® Vtune Amplifier Summary

VTune Amplifier XE OpenMP analysis answers customers' questions about performance in the language of OpenMP constructs

The analysis scales well for many-core systems with good balance of tracing and sampling collection technologies

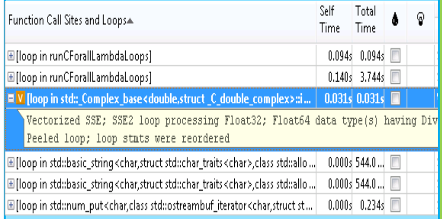
The full feature set is available in VTune Amplifier XE with Intel OpenMP and Intel MPI runtimes as a part of Intel® Parallel Studio XE

ADD OPENMP SIMD WITH INTEL® (VECTOR) ADVISOR

Slides by Zakhar Matveev, Intel® Advisor Architect

Intel Advisor: 5 tools for Efficient Vectorization and Memory utilization

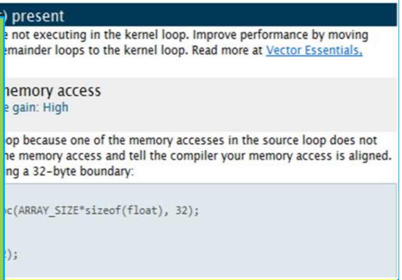
1. Compiler diagnostics + Performance Data + SIMD efficiency information



Function Call Sites and Loops▲	Self Time	Total Time		
⊞[loop in runCForallLambdaLoops]	0.094s	0.094s	□	
⊞[loop in runCForallLambdaLoops]	0.140s	3.744s	□	
⊟ V [loop in std::_Complex_base<double,struct _C_double_complex>::i...	0.031s	0.031s	□	
⊞[loop in std::basic_string<char,struct std::char_traits<char>,class std::allo...	0.000s	544.0 ...	□	
⊞[loop in std::basic_string<char,struct std::char_traits<char>,class std::allo...	0.000s	544.0 ...	□	
⊞[loop in std::num_put<char,class std::ostreambuf_iterator<char,struct st...	0.000s	0.234s	□	

```
Vectorized SSE; SSE2 loop processing Float32; Float64 data type(s) having Div
Peeled loop; loop stmts were reordered
```

2. Guidance: detect problem and recommend how to fix it



⚠ 2 Issue: Peeled/Remainder loop(s) present

e not executing in the kernel loop. Improve performance by moving emainder loops to the kernel loop. Read more at Vector Essentials.

nemory access

e gain: High

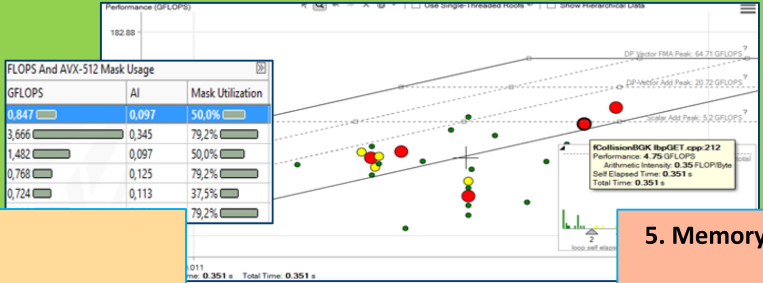
op because one of the memory accesses in the source loop does not he memory access and tell the compiler your memory access is aligned. ng a 32-byte boundary:

```
c(ARRAY_SIZE*sizeof(float), 32);

);
```

3. "Precise" Trip Counts & FLOPs. Roofline analysis.

Characterize your application.

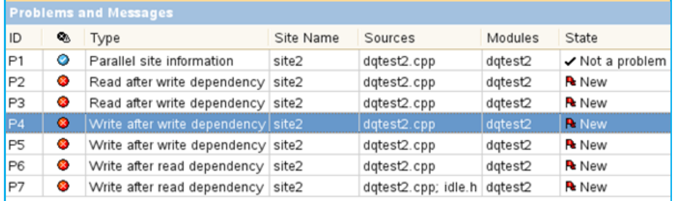


FLOPS And AVX-512 Mask Usage

GFLOPS	AI	Mask Utilization
0,847	0,097	50,0%
3,666	0,345	79,2%
1,482	0,097	50,0%
0,768	0,125	79,2%
0,724	0,113	37,5%
		79,2%

fCollisionBGK lbpGET.cpp:212
Performance: 4.75 GFLOPS
Arithmetic Intensity: 0.35 FLOP/Byte
Self Elapsed Time: 0.351 s
Total Time: 0.351 s

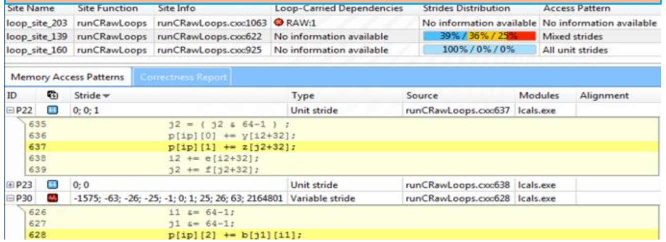
4. Loop-Carried Dependency Analysis



Problems and Messages

ID		Type	Site Name	Sources	Modules	State
P1		Parallel site information	site2	dqtest2.cpp	dqtest2	✓ Not a problem
P2		Read after write dependency	site2	dqtest2.cpp	dqtest2	New
P3		Read after write dependency	site2	dqtest2.cpp	dqtest2	New
P4		Write after write dependency	site2	dqtest2.cpp	dqtest2	New
P5		Write after write dependency	site2	dqtest2.cpp	dqtest2	New
P6		Write after read dependency	site2	dqtest2.cpp	dqtest2	New
P7		Write after read dependency	site2	dqtest2.cpp; idle.h	dqtest2	New

5. Memory Access Pattern Analysis



Site Name	Site Function	Site Info	Loop-Carried Dependencies	Strides Distribution	Access Pattern
loop_site_203	runCRawLoops	runCRawLoops.cxx:1063	RAW:1	No information available	No information available
loop_site_139	runCRawLoops	runCRawLoops.cxx:622	No information available	39% / 36% / 25%	Mixed strides
loop_site_160	runCRawLoops	runCRawLoops.cxx:925	No information available	100% / 0% / 0%	All unit strides

Memory Access Patterns | Correctness Report

ID		Stride▼	Type	Source	Modules	Alignment
⊟P22		0; 0; 1	Unit stride	runCRawLoops.cxx:637	lcals.exe	
⊞P23		0; 0	Unit stride	runCRawLoops.cxx:638	lcals.exe	
⊟P30		-1575; -63; -26; -25; -1; 0; 1; 25; 26; 63; 2164801	Variable stride	runCRawLoops.cxx:628	lcals.exe	

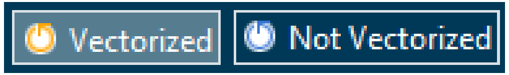
P22:

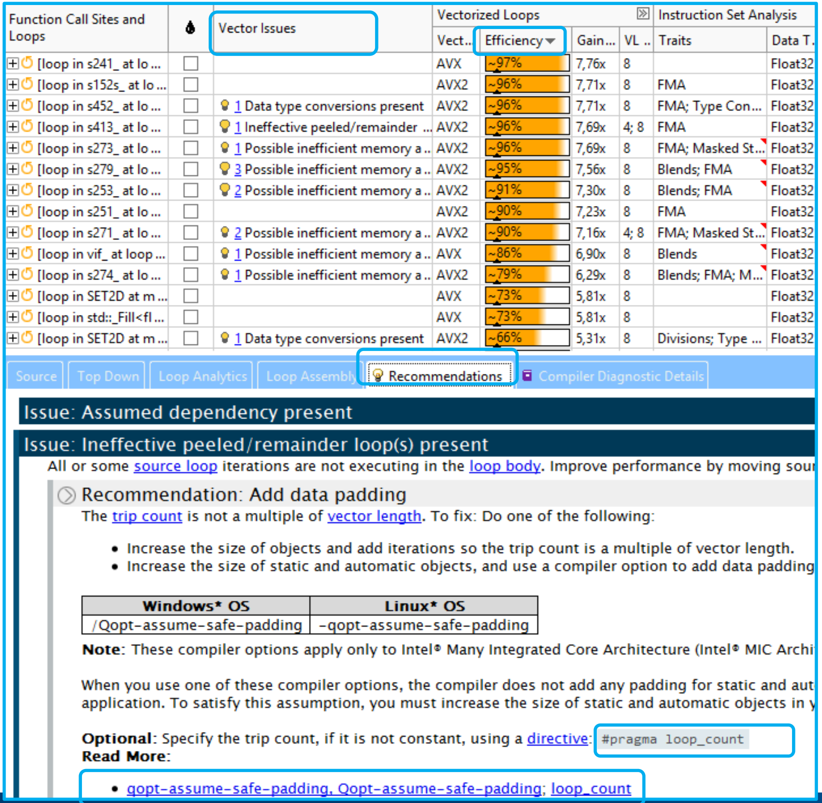
```
j2 = ( j2 & 64-1 ) ;
p[ip][0] += y[i2+32];
p[ip][1] += z[j2+32];
i2 += e[i2+32];
j2 += f[j2+32];
```

P30:

```
i1 &= 64-1;
j1 &= 64-1;
p[ip][2] += b[j1][i1];
```

Advisor Survey: vectorize and improve SIMD code performance!





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

Issue: Assumed dependency present

Issue: Ineffective peeled/remainder loop(s) present

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

Recommendation: Add data padding

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

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

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

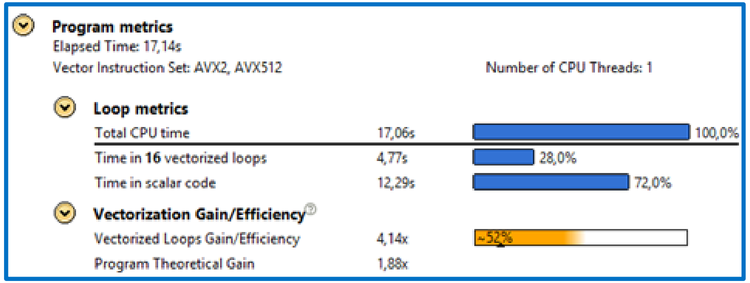
Note: These compiler options apply only to Intel® Many Integrated Core Architecture (Intel® MIC Archi

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

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

Read More:

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



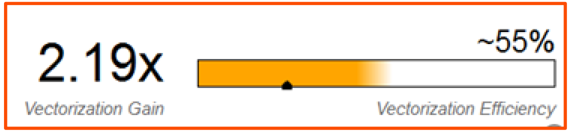
Program metrics
Elapsed Time: 17,14s
Vector Instruction Set: AVX2, AVX512 Number of CPU Threads: 1

Loop metrics

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

Vectorization Gain/Efficiency

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



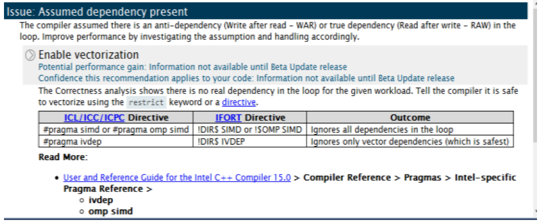
- **Efficiency** – my performance thermometer
- **Recommendations** – get tips on how to improve performance, in particular using OpenMP 4.* and later!

Advisor Dependencies: The Answer to Tough SIMD/Threading Question #1!

Is it safe to force the compiler to vectorize?

```
DO I = 1, N
   A(I) = A(I-1) * B(I)
ENDDO
```

```
void scale(int *a, int *b)
{
   for (int i = 0; i < 1000; i++)
      b[i] = z * a[i];
}
```



Issue: Assumed dependency present

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

Enable vectorization

Potential performance gain: Information not available until Beta Update release

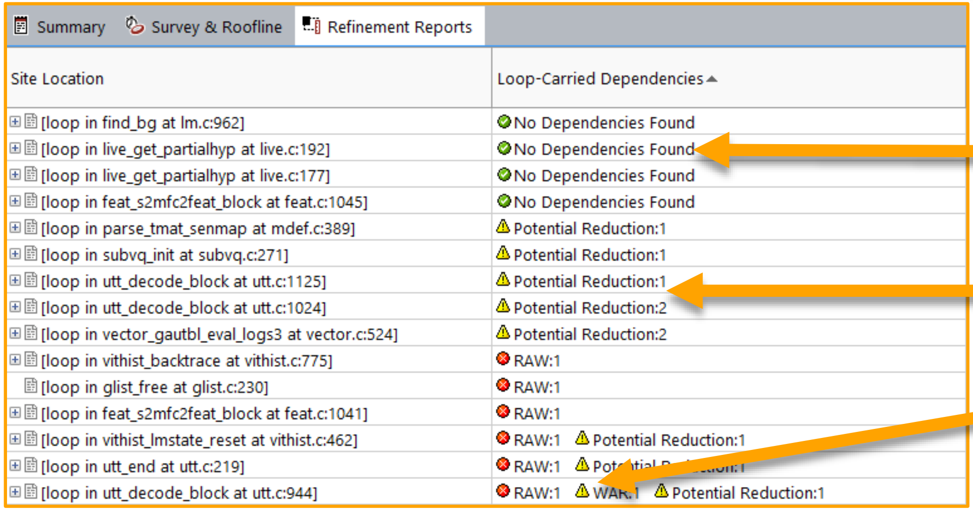
Confidence this recommendation applies to your code: Information not available until Beta Update release

The Correctness 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.

ICL/ICC/ICPC Directive	IFORT Directive	Outcome
#pragma simd or #pragma omp simd	!DIR$ SIMD or !$OMP SIMD	Ignores all dependencies in the loop
#pragma ivdep	!DIR$ IVDEP	Ignores only vector dependencies (which is safest)

Read More:

- User and Reference Guide for the Intel C++ Compiler 15.0 **> Compiler Reference > Pragmas > Intel-specific Pragma Reference >**
  - **ivdep**
  - **omp simd**



Site Location	Loop-Carried Dependencies
[loop in find_bg at lm.c:962]	No Dependencies Found
[loop in live_get_partialhyp at live.c:192]	No Dependencies Found
[loop in live_get_partialhyp at live.c:177]	No Dependencies Found
[loop in feat_s2mfc2feat_block at feat.c:1045]	No Dependencies Found
[loop in parse_tmat_senmap at mdef.c:389]	Potential Reduction:1
[loop in subvq_init at subvq.c:271]	Potential Reduction:1
[loop in utt_decode_block at utt.c:1125]	Potential Reduction:1
[loop in utt_decode_block at utt.c:1024]	Potential Reduction:2
[loop in vector_gautbl_eval_logs3 at vector.c:524]	Potential Reduction:2
[loop in vithist_backtrace at vithist.c:775]	RAW:1
[loop in glist_free at glist.c:230]	RAW:1
[loop in feat_s2mfc2feat_block at feat.c:1041]	RAW:1
[loop in vithist_lmstate_reset at vithist.c:462]	RAW:1 Potential Reduction:1
[loop in utt_end at utt.c:219]	RAW:1 Potential Reduction:1
[loop in utt_decode_block at utt.c:944]	RAW:1 WAR:1 Potential Reduction:1

Safe to vectorize (at least for given workload), use **OMP SIMD**!

Use **OMP reduction** (also for threading)!

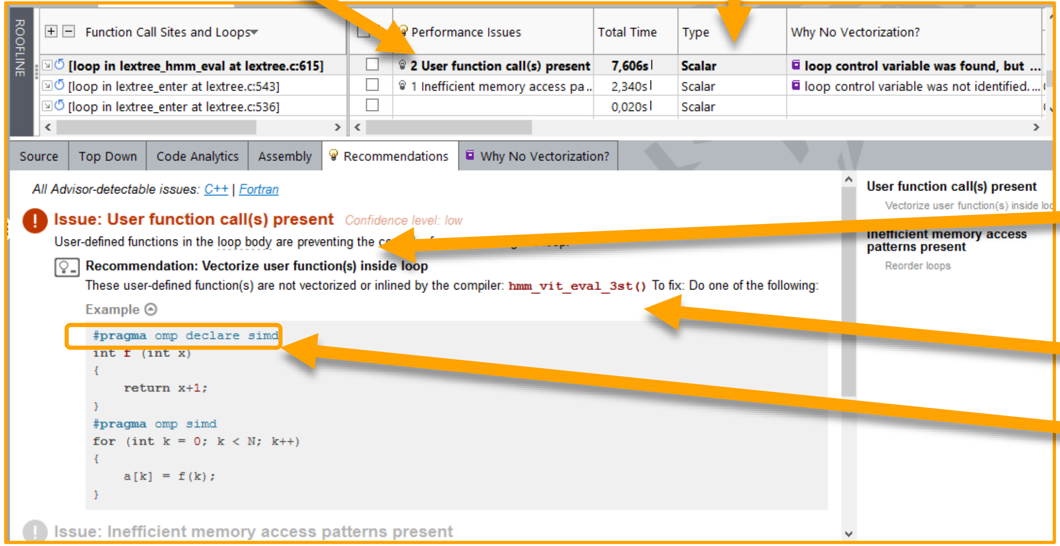
True dependence proven, not way to parallelize without extra work

Optimization Notice

Intel Advisor gives recommendations to guide you where and how to add OpenMP SIMD!

What issues must to be fixed?

Which loops are vectorized?



Function Call Sites and Loops	□	Performance Issues	Total Time	Type	Why No Vectorization?
[loop in lextree_hmm_eval at lextree.c:615]	□	**2 User function call(s) present**	**7,606s**	**Scalar**	**loop control variable was found, but ...**
[loop in lextree_enter at lextree.c:543]	□	1 Inefficient memory access pa..	2,340s	Scalar	loop control variable was not identified....
[loop in lextree_enter at lextree.c:536]	□		0,020s	Scalar	

All Advisor-detectable issues: C++ | Fortran

Issue: User function call(s) present *Confidence level: low*

User-defined functions in the loop body are preventing the c

Recommendation: Vectorize user function(s) inside loop

These user-defined function(s) are not vectorized or inlined by the compiler: `hmm_vit_eval_3st()` To fix: Do one of the following:

Example

```
#pragma omp declare simd
int f (int x)
{
    return x+1;
}
#pragma omp simd
for (int k = 0; k < N; k++)
{
    a[k] = f(k);
}
```

Issue: Inefficient memory access patterns present

User function call(s) present

Vectorize user function(s) inside loo

Inefficient memory access patterns present

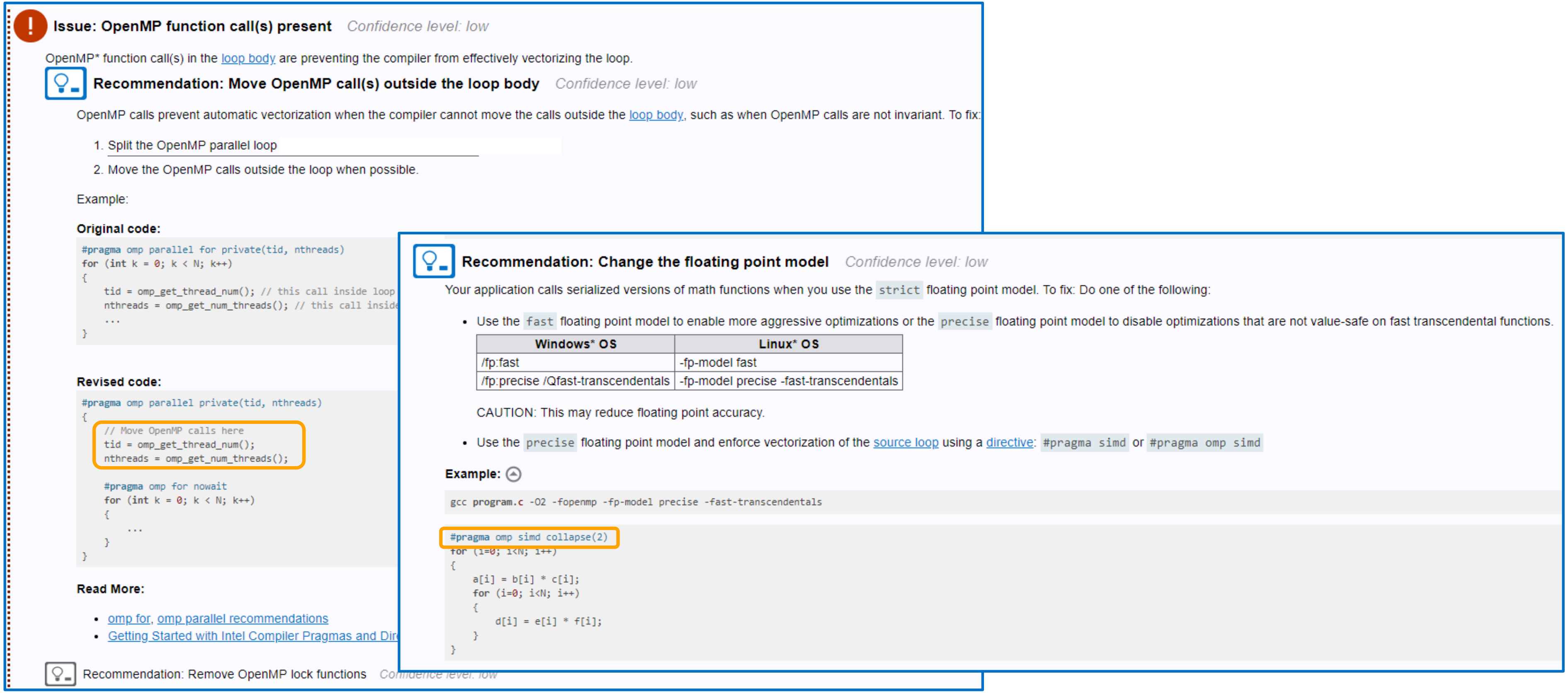
Reorder loops

Read more about the issue and **what to modify in your code** to fix it (to enable vector parallelism!)

In more detail: which function is causing the problem?

Add OpenMP SIMD: reference example

More examples: also threading aware..



Issue: OpenMP function call(s) present *Confidence level: low*

OpenMP* function call(s) in the loop body are preventing the compiler from effectively vectorizing the loop.

Recommendation: Move OpenMP call(s) outside the loop body *Confidence level: low*

OpenMP calls prevent automatic vectorization when the compiler cannot move the calls outside the loop body, such as when OpenMP calls are not invariant. To fix:

1. Split the OpenMP parallel loop
2. Move the OpenMP calls outside the loop when possible.

Example:

Original code:

```
#pragma omp parallel for private(tid, nthreads)
for (int k = 0; k < N; k++)
{
    tid = omp_get_thread_num(); // this call inside loop
    nthreads = omp_get_num_threads(); // this call inside
    ...
}
```

Revised code:

```
#pragma omp parallel private(tid, nthreads)
{
    // Move OpenMP calls here
    tid = omp_get_thread_num();
    nthreads = omp_get_num_threads();

    #pragma omp for nowait
    for (int k = 0; k < N; k++)
    {
        ...
    }
}
```

Read More:

- omp for, omp parallel recommendations
- Getting Started with Intel Compiler Pragmas and Dir

Recommendation: Remove OpenMP lock functions *Confidence level: low*

Recommendation: Change the floating point model *Confidence level: low*

Your application calls serialized versions of math functions when you use the `strict` floating point model. To fix: Do one of the following:

- Use the `fast` floating point model to enable more aggressive optimizations or the `precise` floating point model to disable optimizations that are not value-safe on fast transcendental functions.

Windows* OS	Linux* OS
/fp:fast	-fp-model fast
/fp:precise /Qfast-transcendentals	-fp-model precise -fast-transcendentals

  CAUTION: This may reduce floating point accuracy.

- Use the `precise` floating point model and enforce vectorization of the source loop using a directive: `#pragma simd` or `#pragma omp simd`

Example:

```
gcc program.c -O2 -fopenmp -fp-model precise -fast-transcendentals
```

```
#pragma omp simd collapse(2)
for (i=0; i<N; i++)
{
    a[i] = b[i] * c[i];
    for (i=0; i<N; i++)
    {
        d[i] = e[i] * f[i];
    }
}
```

Optimization Notice

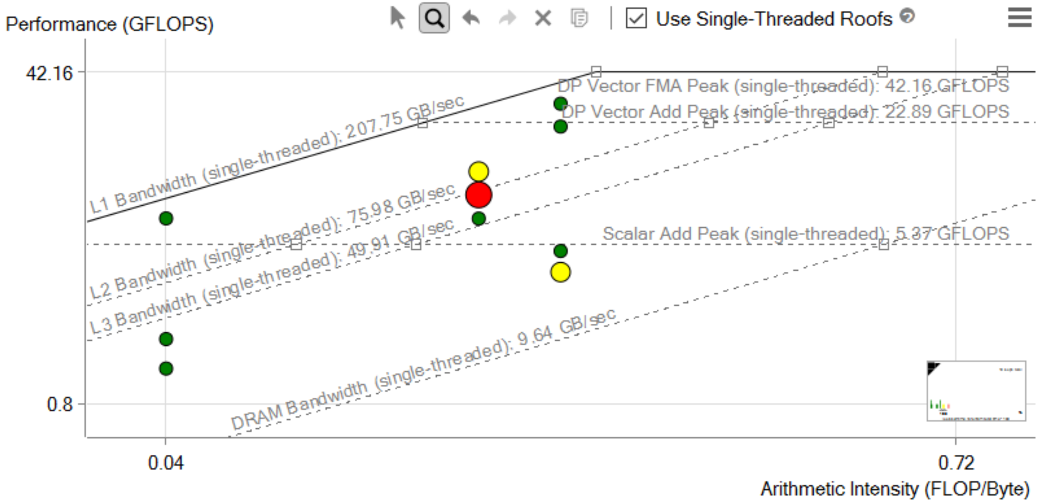
ROOFLINE IN INTEL® ADVISOR

Slides by Alex Shinsel

What is a Roofline Chart?

A Roofline Chart plots application performance against hardware limitations.

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

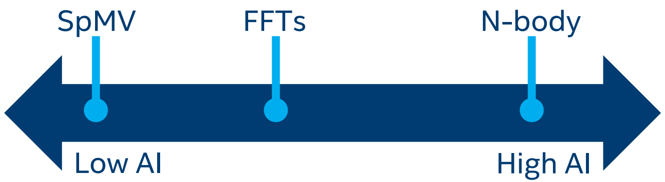


Roofline first proposed by University of California at Berkeley:
Roofline: An Insightful Visual Performance Model for Multicore Architectures, 2009
Cache-aware variant proposed by University of Lisbon:
Cache-Aware Roofline Model: Upgrading the Loft, 2013

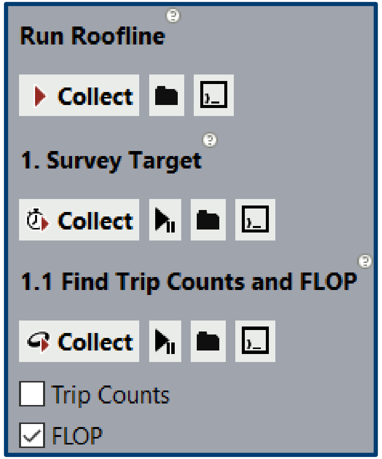
Roofline Metrics

Roofline is based on FLOPS and Arithmetic Intensity (AI).

- **FLOPS**: **Fl**oating-Point **Op**erations / **S**econd
- **Arithmetic Intensity**: FLOP / Byte Accessed



Collecting this information in Intel® Advisor requires two analyses.



Run Roofline

1. Survey Target

1.1 Find Trip Counts and FLOP

□ Trip Counts

☑ FLOP

Shortcut to run Survey followed by Trip Counts + FLOPs

Runs system benchmarks and collects timing data.

Collects memory traffic and FLOP data.
Must be run separately due to higher overhead that would interfere with timing measurements.

Classic vs. Cache-Aware Roofline

Intel® Advisor uses the Cache-Aware Roofline model, which has a different definition of Arithmetic Intensity than the original ("Classic") model.

Classical Roofline

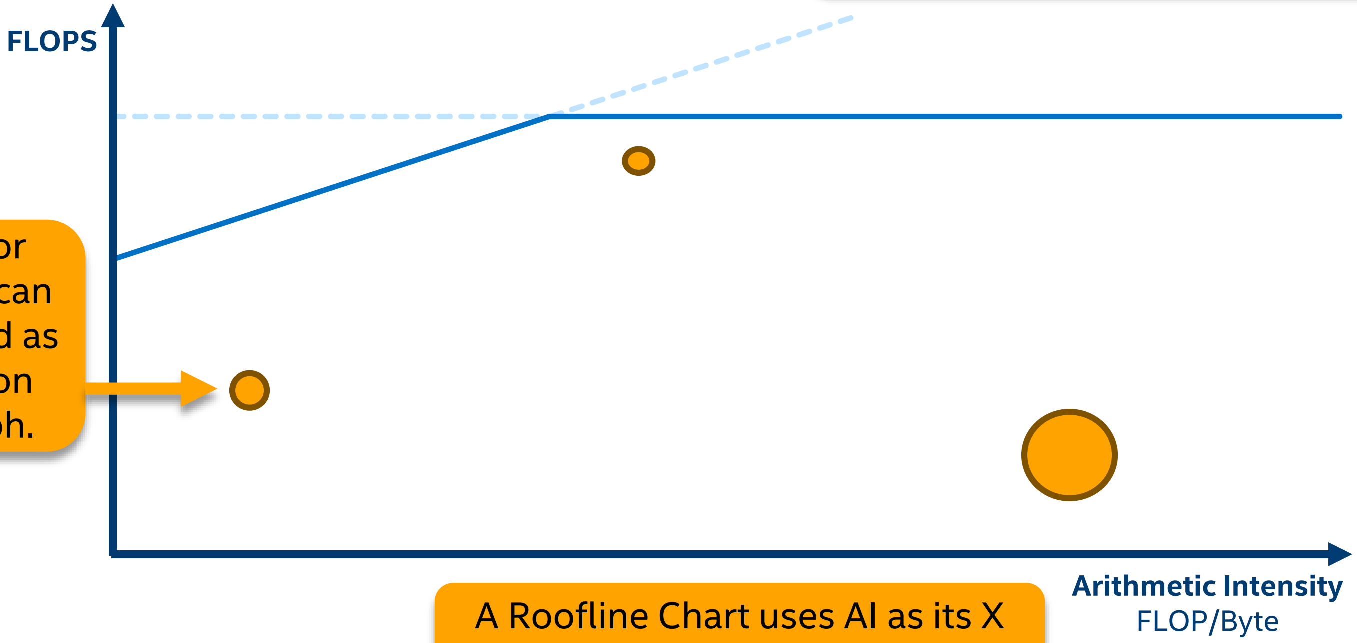
- Traffic measured from one level of memory (usually DRAM)
- AI may change with data set size
- AI changes as a result of memory optimizations

Cache-Aware Roofline

- Traffic measured from all levels of memory
- AI is tied to the algorithm and will not change with data set size
- Optimization does not change AI*, only the performance

**Compiler optimizations may modify the algorithm, which may change the AI.*

Plotting a Roofline Chart



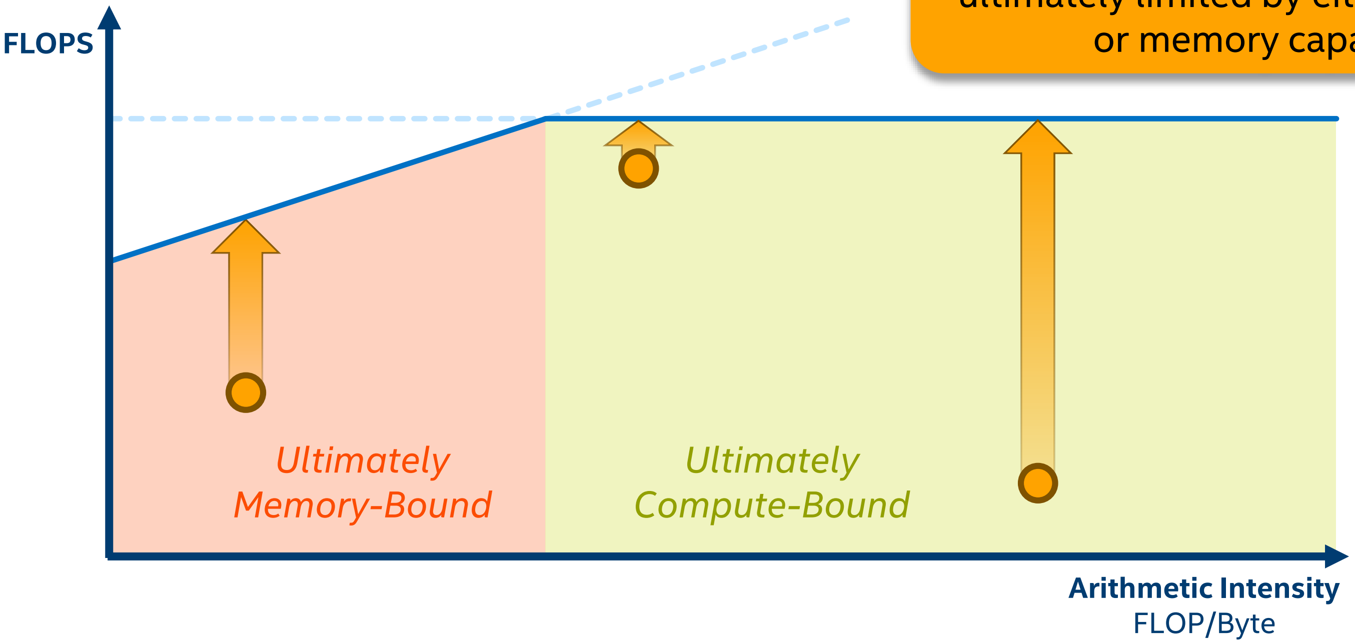
The maximum FLOPS as a product of ops/byte (AI) and maximum bytes supplied per second is a diagonal line.

The CPU's maximum FLOPS can be plotted as a horizontal line.

A loop or function can be plotted as a point on the graph.

A Roofline Chart uses AI as its X axis and FLOPS as its Y axis.

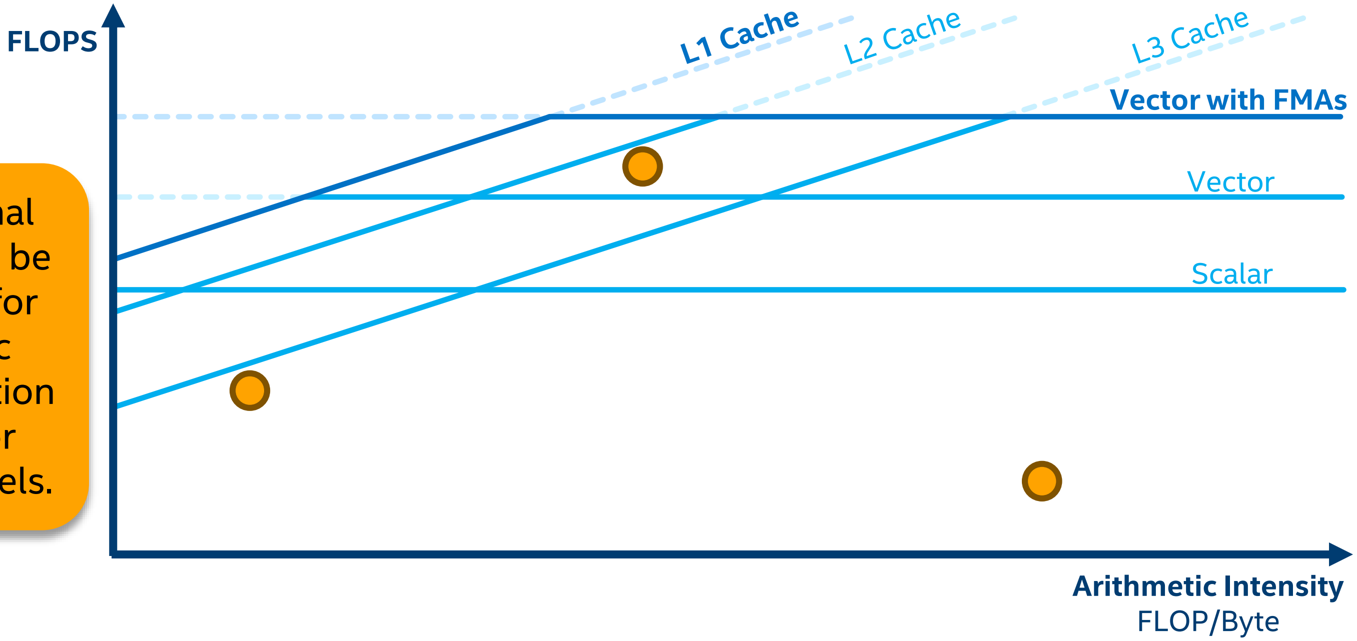
Ultimate Performance Limits



Performance cannot exceed the machine's capabilities, so each loop is ultimately limited by either compute or memory capacity.

Sub-Roofs and Current Limits

Additional roofs can be plotted for specific computation types or cache levels.



These sub-roofs can be used to help diagnose bottlenecks.

Optimization Notice

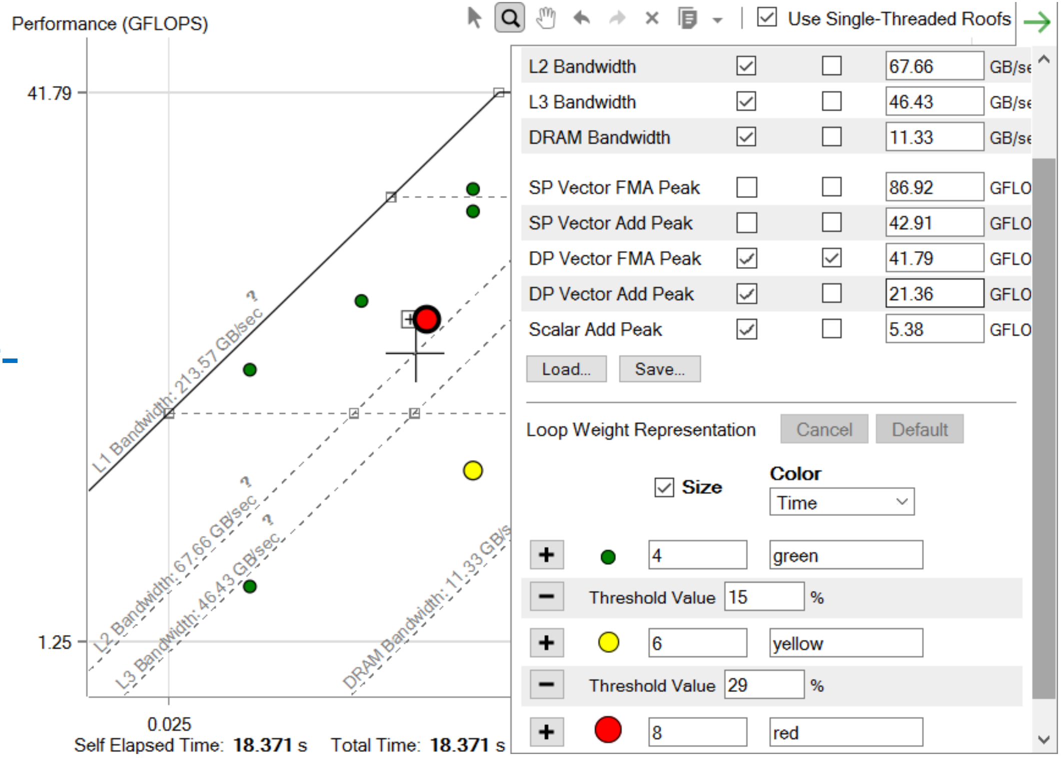
The Intel® Advisor Roofline Interface

Roofs are based on benchmarks run before the application.

- Roofs can be hidden, highlighted, or adjusted.

Intel® Advisor has size- and colour-coding for dots.

- Colour code by duration or vectorization status
- Categories, cutoffs, and visual style can be modified.


Performance (GFLOPS)
41.79
1.25
0.025
L1 Bandwidth: 213.57 GB/sec
L2 Bandwidth: 67.66 GB/sec
L3 Bandwidth: 46.43 GB/sec
DRAM Bandwidth: 11.33 GB/s

Self Elapsed Time: **18.371** s Total Time: **18.371** s

☑ Use Single-Threaded Roofs

L2 Bandwidth	☑	□	67.66	GB/se
L3 Bandwidth	☑	□	46.43	GB/se
DRAM Bandwidth	☑	□	11.33	GB/se
SP Vector FMA Peak	□	□	86.92	GFLO
SP Vector Add Peak	□	□	42.91	GFLO
DP Vector FMA Peak	☑	☑	41.79	GFLO
DP Vector Add Peak	☑	□	21.36	GFLO
Scalar Add Peak	☑	□	5.38	GFLO

Load... Save...

Loop Weight Representation Cancel Default

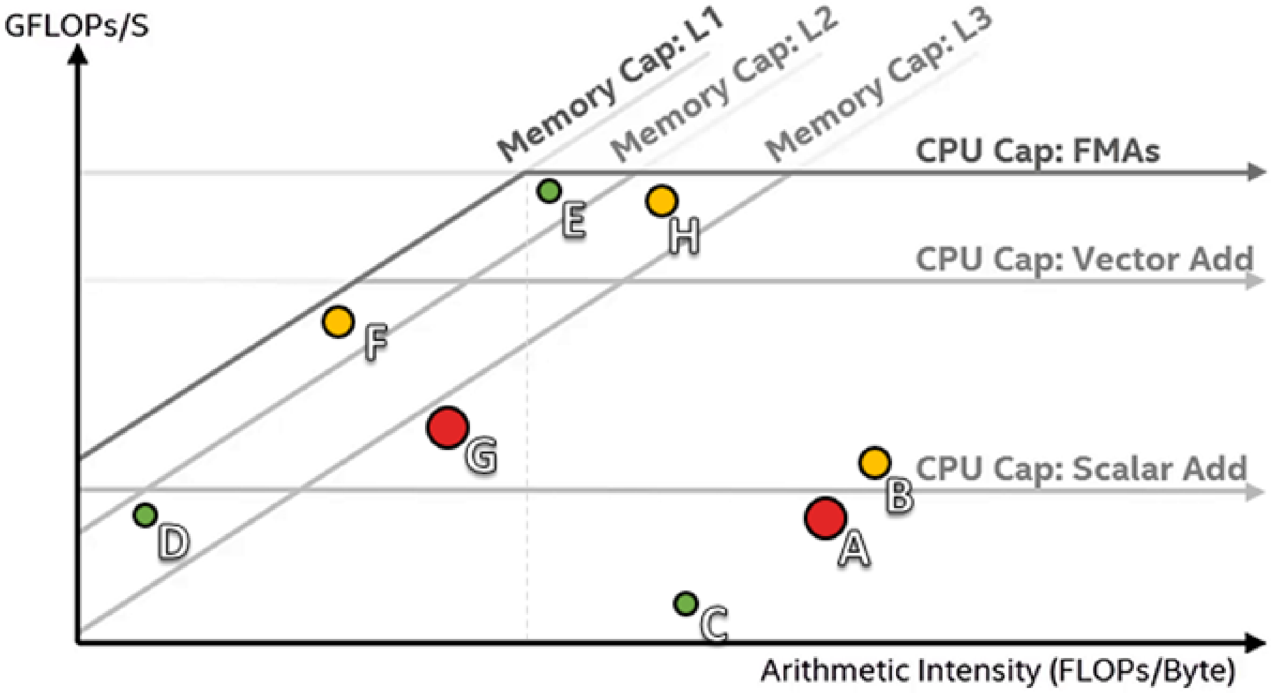
☑ **Size** **Color** Time

+	●	4	green
−	Threshold Value	15	%
+	●	6	yellow
−	Threshold Value	29	%
+	●	8	red

Identifying Good Optimization Candidates

Focus optimization effort where it makes the most difference.

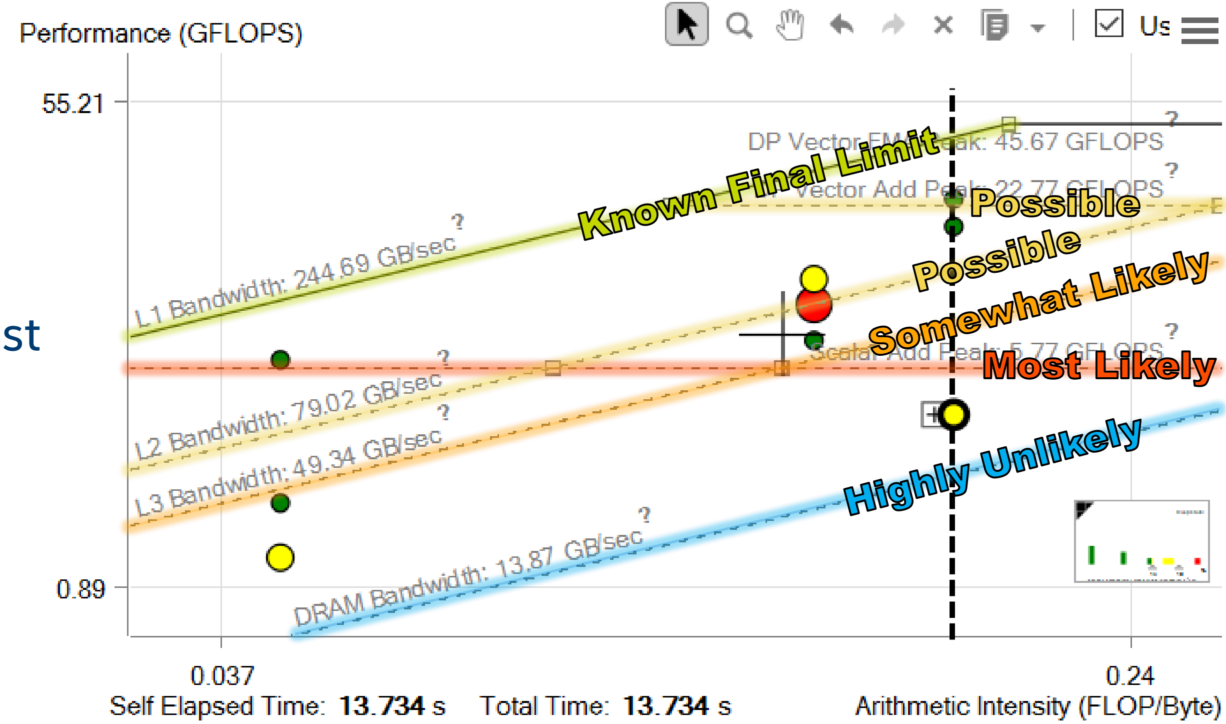
- Large, red loops have the most impact.
- Loops far from the upper roofs have more room to improve.



Identifying Potential Bottlenecks

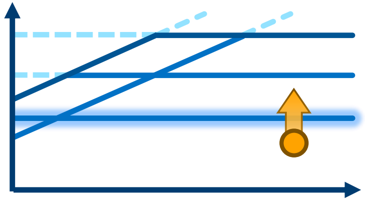
Final roofs *do* apply; sub-roofs *may* apply.

- Roofs above indicate *potential* bottlenecks
- Closer roofs are the most likely suspects
- Roofs below may contribute but are generally not primary bottlenecks



Feature Synergy

Overcoming the Scalar Add Peak



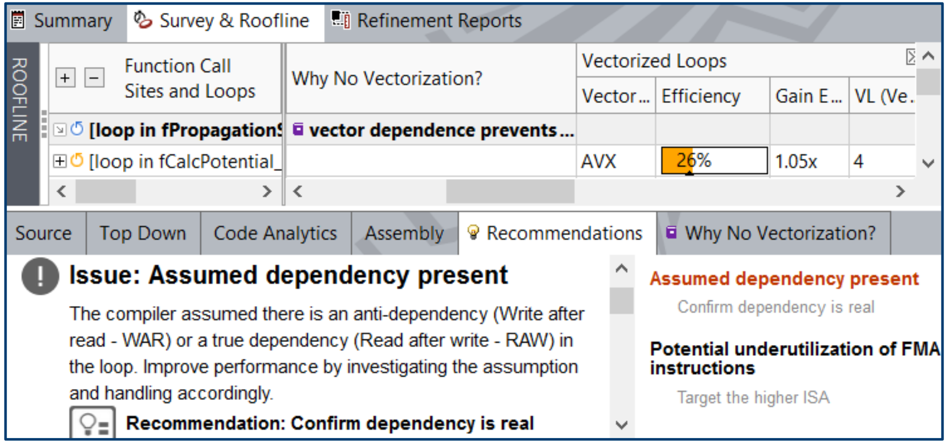
Survey and Code Analytics tabs indicate vectorization status with colored icons.

↺ = Scalar ↺ = Vectorized

“Why No Vectorization” tab and column in Survey explain what prevented vectorization.

Recommendations tab may help you vectorize the loop.

Dependencies determines if it’s safe to force vectorization.



Function Call Sites and Loops	Why No Vectorization?	Vector...	Efficiency	Gain E...	VL (Ve...
[loop in fPropagation	vector dependence prevents ...				
[loop in fCalcPotential_		AVX	26%	1.05x	4

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.

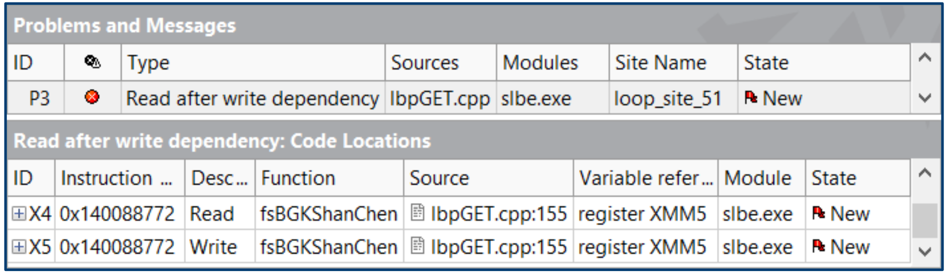
Recommendation: Confirm dependency is real

Assumed dependency present

Confirm dependency is real

Potential underutilization of FMA instructions

Target the higher ISA



Problems and Messages

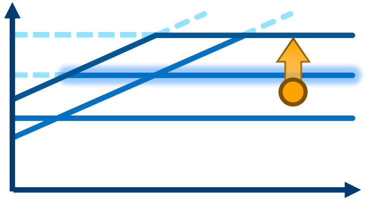
ID		Type	Sources	Modules	Site Name	State
P3	●	Read after write dependency	lbpGET.cpp	slbe.exe	loop_site_51	New

Read after write dependency: Code Locations

ID	Instruction ...	Desc...	Function	Source	Variable refer...	Module	State
X4	0x140088772	Read	fsBGKShanChen	lbpGET.cpp:155	register XMM5	slbe.exe	New
X5	0x140088772	Write	fsBGKShanChen	lbpGET.cpp:155	register XMM5	slbe.exe	New

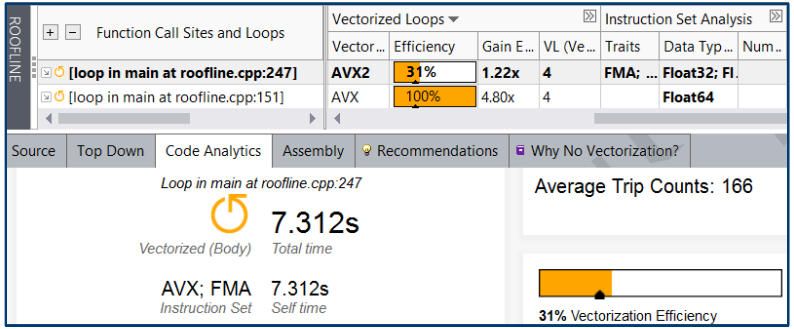
Feature Synergy

Overcoming the Vector Add Peak



Survey and Code Analytics display the vector efficiency and presence of FMAs.

- Recommendations may help improve efficiency or induce FMA usage.



ROOFLINE

+ − Function Call Sites and Loops	Vectorized Loops Vector...	 Efficiency	 Gain E...	 VL (Ve...	Instruction Set Analysis Traits	 Data Typ...	 Num...
[loop in main at roofline.cpp:247]	**AVX2**	**31%**	**1.22x**	**4**	**FMA; ...**	**Float32; Fl.**	
[loop in main at roofline.cpp:151]	AVX	100%	4.80x	4		**Float64**	

Loop in main at roofline.cpp:247

7.312s *Total time*

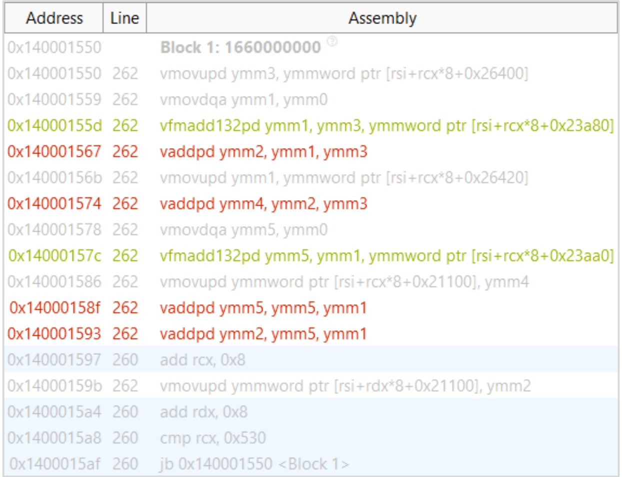
Vectorized (Body)

AVX; FMA *Instruction Set*

7.312s *Self time*

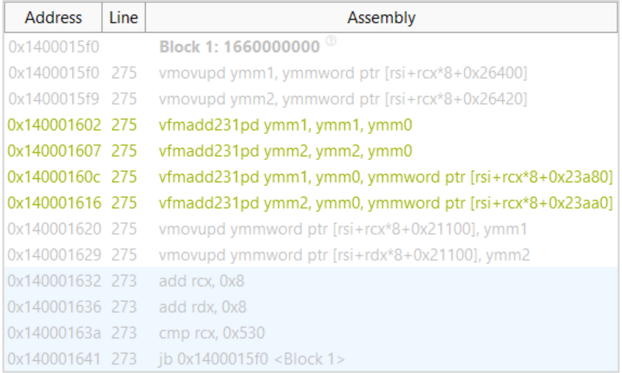
Average Trip Counts: 166

31% Vectorization Efficiency



Address	Line	Assembly
0x140001550		**Block 1: 1660000000**
0x140001550	262	vmovupd ymm3, ymmword ptr [rsi+rcx*8+0x26400]
0x140001559	262	vmovdqa ymm1, ymm0
0x14000155d	262	vfmadd132pd ymm1, ymm3, ymmword ptr [rsi+rcx*8+0x23a80]
0x140001567	262	vaddpd ymm2, ymm1, ymm3
0x14000156b	262	vmovupd ymm1, ymmword ptr [rsi+rcx*8+0x26420]
0x140001574	262	vaddpd ymm4, ymm2, ymm3
0x140001578	262	vmovdqa ymm5, ymm0
0x14000157c	262	vfmadd132pd ymm5, ymm1, ymmword ptr [rsi+rcx*8+0x23aa0]
0x140001586	262	vmovupd ymmword ptr [rsi+rcx*8+0x21100], ymm4
0x14000158f	262	vaddpd ymm5, ymm5, ymm1
0x140001593	262	vaddpd ymm2, ymm5, ymm1
0x140001597	260	add rcx, 0x8
0x14000159b	262	vmovupd ymmword ptr [rsi+rdx*8+0x21100], ymm2
0x1400015a4	260	add rdx, 0x8
0x1400015a8	260	cmp rcx, 0x530
0x1400015af	260	jb 0x140001550 <Block 1>

The Assembly tab* is useful for determining how well you are making use of FMAs.



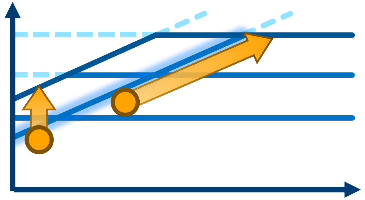
Address	Line	Assembly
0x1400015f0		**Block 1: 1660000000**
0x1400015f0	275	vmovupd ymm1, ymmword ptr [rsi+rcx*8+0x26400]
0x1400015f9	275	vmovupd ymm2, ymmword ptr [rsi+rcx*8+0x26420]
0x140001602	275	vfmadd231pd ymm1, ymm1, ymm0
0x140001607	275	vfmadd231pd ymm2, ymm2, ymm0
0x14000160c	275	vfmadd231pd ymm1, ymm0, ymmword ptr [rsi+rcx*8+0x23a80]
0x140001616	275	vfmadd231pd ymm2, ymm0, ymmword ptr [rsi+rcx*8+0x23aa0]
0x140001620	275	vmovupd ymmword ptr [rsi+rcx*8+0x21100], ymm1
0x140001629	275	vmovupd ymmword ptr [rsi+rdx*8+0x21100], ymm2
0x140001632	273	add rcx, 0x8
0x140001636	273	add rdx, 0x8
0x14000163a	273	cmp rcx, 0x530
0x140001641	273	jb 0x1400015f0 <Block 1>

**Color coding added for clarity.*

Optimization Notice

Feature Synergy

Overcoming the Memory Bandwidth Roofs

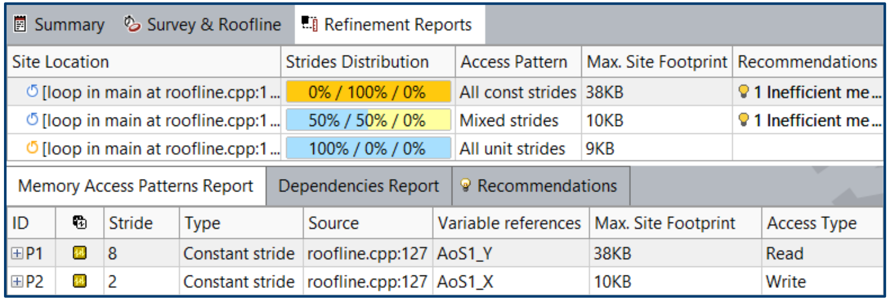


Memory Access Patterns (MAP) identifies inefficient access patterns.

Intel® SIMD Data Layout Templates (Intel® SDLT) allows code written as AOS to be stored as efficient SOA.

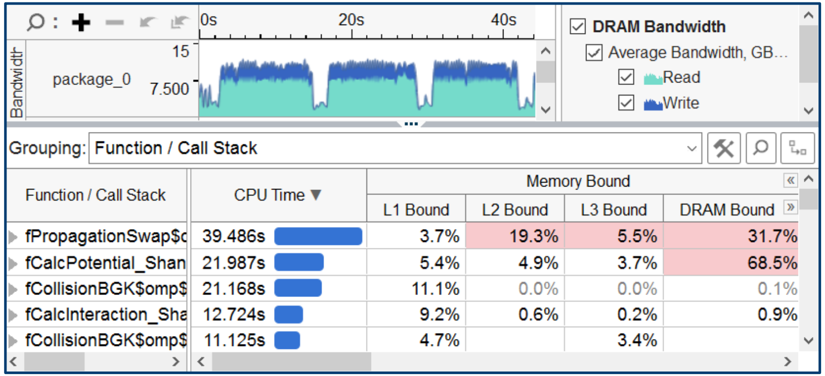
Intel® VTune™ Amplifier can be used to further optimize cache usage.

If cache usage cannot be improved, try re-working the algorithm to increase the AI (and slide up the roof)



Site Location	Strides Distribution	Access Pattern	Max. Site Footprint	Recommendations
[loop in main at roofline.cpp:1...	0% / 100% / 0%	All const strides	38KB	1 Inefficient me...
[loop in main at roofline.cpp:1...	50% / 50% / 0%	Mixed strides	10KB	1 Inefficient me...
[loop in main at roofline.cpp:1...	100% / 0% / 0%	All unit strides	9KB	

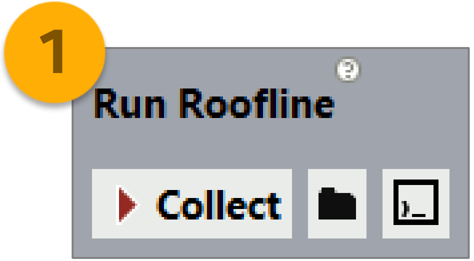
ID		Stride	Type	Source	Variable references	Max. Site Footprint	Access Type
⊞P1		8	Constant stride	roofline.cpp:127	AoS1_Y	38KB	Read
⊞P2		2	Constant stride	roofline.cpp:127	AoS1_X	10KB	Write

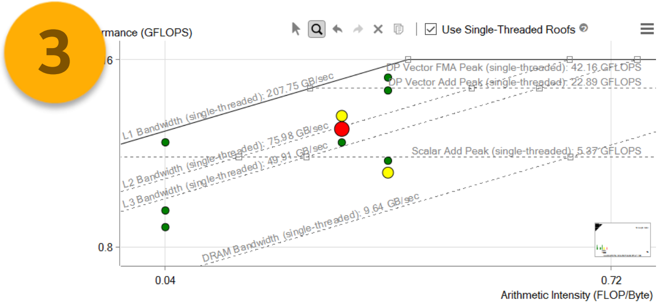


Grouping: Function / Call Stack

Function / Call Stack	CPU Time ▼	Memory Bound			
		L1 Bound	L2 Bound	L3 Bound	DRAM Bound
fPropagationSwap$c	39.486s	3.7%	19.3%	5.5%	31.7%
fCalcPotential_Shan	21.987s	5.4%	4.9%	3.7%	68.5%
fCollisionBGKomp	21.168s	11.1%	0.0%	0.0%	0.1%
fCalcInteraction_Sha	12.724s	9.2%	0.6%	0.2%	0.9%
fCollisionBGKomp	11.125s	4.7%		3.4%	

Intel® Advisor Roofline Summary





☑ Use Single-Threaded Roofs

Intel® Advisor's Roofline Chart is highly customizable and easy to generate.

Lets you identify the best optimization candidates by focusing on low, large loops.

Use the chart to identify the most likely bottlenecks.

Intel® Advisor's many other features allow deep analysis of suspected problems and provide advice on how to overcome them.

Overall Conclusions:

Tuning requires analysis of what’s really happening in the code

- Our guesses are frequently wrong

We also need to understand the potential for improvement

Intel tools (Vtune Amplifier, Advisor and Inspector) can help

Download Intel Parallel Studio XE (includes all the tools and other goodies) and try it.

- Free for students, teachers, Open Source contributors.
- Free evaluation licenses for everyone else.
- Or you can even pay money!

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

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

intel®

Software