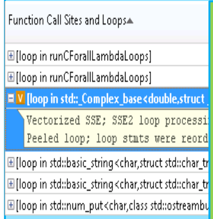
INTEL (VECTORIZATION AND ROOFLINE) ADVISOR

Intel Software and Services, 2017

Zakhar Matveev, PhD, Product architect

5 Steps to Efficient Vectorization and Memory utilization: Intel Advisor 2017

1. Compiler diagnostics + Performance Data + SIMD efficiency information



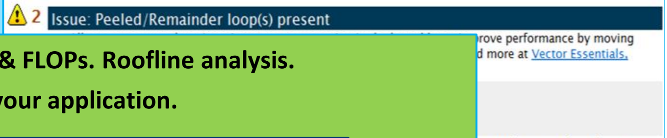
Function Call Sites and Loops▲

- ⊞ [loop in runCForallLambdaLoops]
- ⊞ [loop in runCForallLambdaLoops]
- ⊟ V [loop in std::_Complex_base<double,struct

```
Vectorized SSE; SSE2 loop processi
Peeled loop; loop stmts were reord
```

- ⊞ [loop in std::basic_string<char,struct std::char_tr
- ⊞ [loop in std::basic_string<char,struct std::char_tr
- ⊞ [loop in std::num_put<char,class std::ostreambu

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



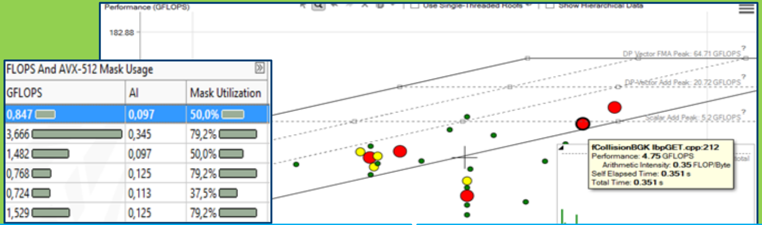
⚠ 2 Issue: Peeled/Remainder loop(s) present

rove performance by moving
d more at Vector Essentials.

es in the source loop does not
r your memory access is aligned.

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

Characterize your application.

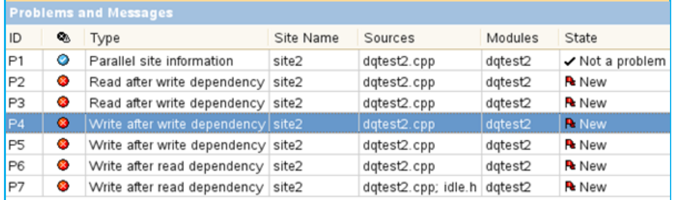

Performance (GFLOPS)
Use Single-Threaded Roofs
Show Hierarchical Data
182.88
DP Vector FMA Peak: 64.71 GFLOPS
DP Vector Add Peak: 20.72 GFLOPS
Scalar Add Peak: 5.2 GFLOPS

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

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%
1,529	0,125	79,2%

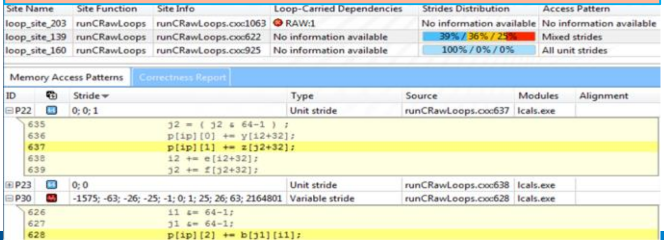
4. Loop-Carried Dependency Analysis



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 Patterns 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	

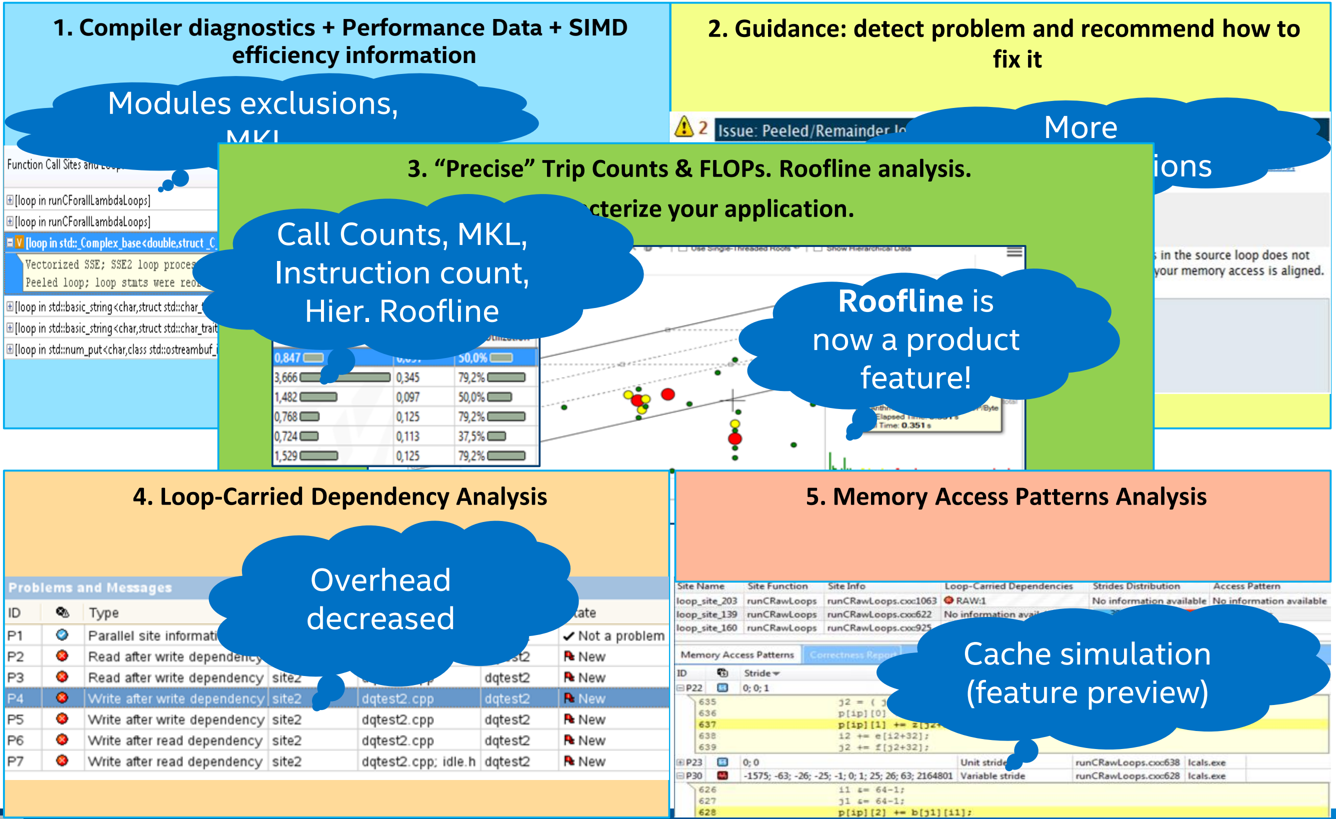
```
j2 = ( j2 & 64-1 ) ;
p[ip][0] += y[i2+32];
p[ip][1] += z[j2+32];
i2 += e[i2+32];
j2 += f[j2+32];
```

ID		Stride▾	Type	Source	Modules	Alignment
⊞ 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	

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

Optimization Notice

What’s new in “2018” release



1. Compiler diagnostics + Performance Data + SIMD efficiency information

Function Call Sites and Loop

- [loop in runCForallLambdaLoops]
- [loop in runCForallLambdaLoops]
- V [loop in std::_Complex_base<double,struct _C

```
Vectorized SSE; SSE2 loop proces
Peeled loop; loop stmts were reo
```

- [loop in std::basic_string<char,struct std::char_
- [loop in std::basic_string<char,struct std::char_trait
- [loop in std::num_put<char,class std::ostreambuf_

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

⚠ 2 Issue: Peeled/Remainder lo

...s in the source loop does not
...your memory access is aligned.

3. “Precise” Trip Counts & FLOPs. Roofline analysis.

...cterize your application.

0,847	[illegible]	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%
1,529	0,125	79,2%

4. Loop-Carried Dependency Analysis

Problems and Messages

ID		Type				tate
P1	●	Parallel site informati				✓ Not a problem
P2	●	Read after write dependency			t2	New
P3	●	Read after write dependency	site2		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 Patterns 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 avai		
loop_site_160	runCRawLoops	runCRawLoops.cxx:925			

Memory Access Patterns | Correctness Report

ID		Stride			
P22		0; 0; 1			
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

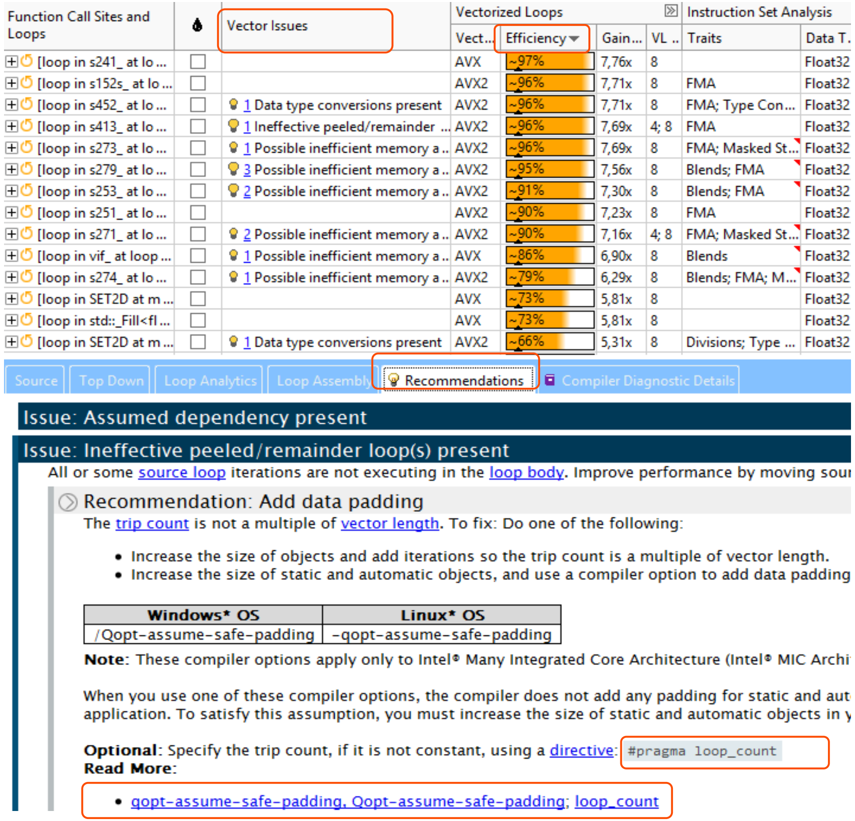
```
j2 = ( j
p[ip][0]
p[ip][1] += z[j2+
i2 += e[i2+32];
j2 += f[j2+32];
```

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

Optimization Notice

Advisor Survey: **Focus** + **Characterize.**

Focus and order **vectorized loops**



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

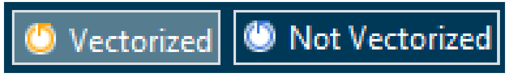
Windows* OS	Linux* OS
/Qopt-assume-safe-padding	-qopt-assume-safe-padding

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



- **Efficiency** – my performance thermometer
- **Recommendations** – get tip on how to improve performance
  - (also apply to scalar loops)

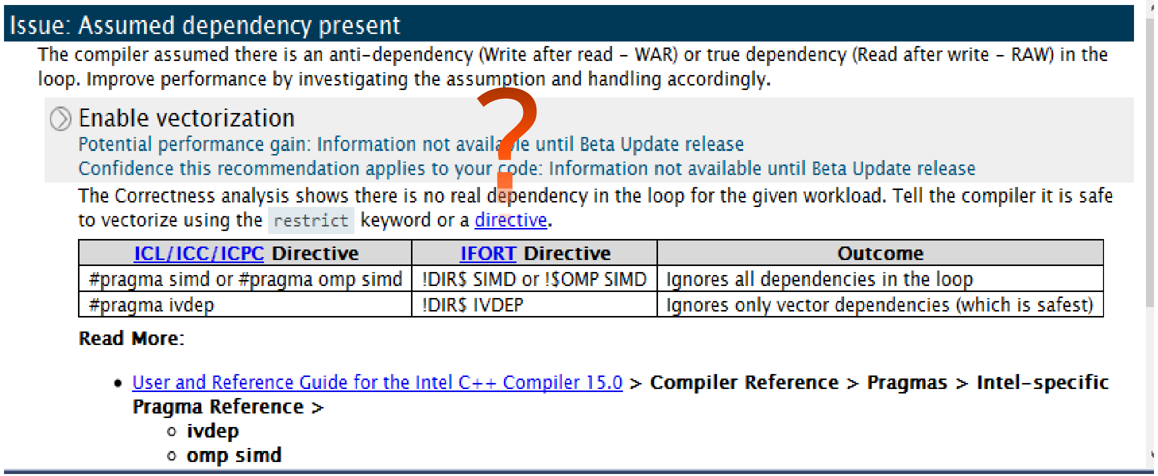
Data Dependencies – Tough Problem #1

Is it safe to force the compiler to vectorize?

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

or

```
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**

Optimization Notice

Advisor Memory Access Pattern (MAP):
know your access pattern

Unit-Stride access

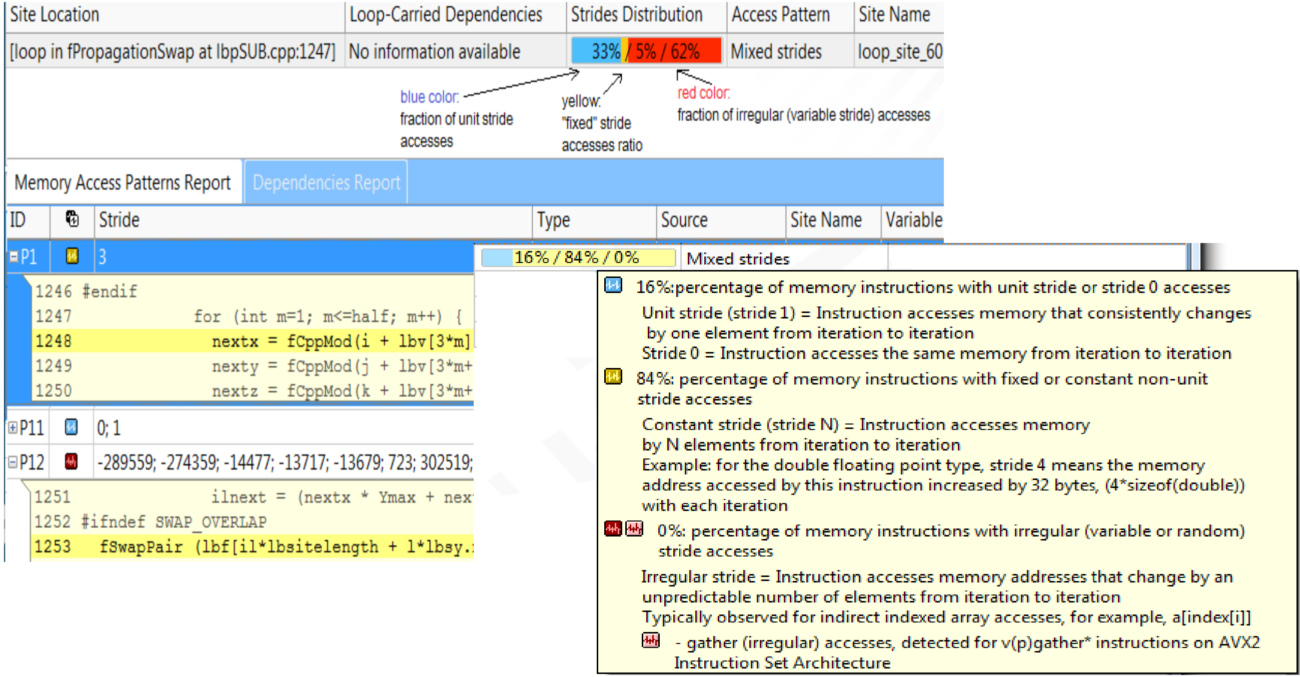
```
for (i=0; i<N; i++)
A[i] = C[i]*D[i]
```

Constant stride access

```
for (i=0; i<N; i++)
point[i].x = x[i]
```

Variable stride access

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



Site Location	Loop-Carried Dependencies	Strides Distribution	Access Pattern	Site Name
[loop in fPropagationSwap at lbpSUB.cpp:1247]	No information available	33% / 5% / 62%	Mixed strides	loop_site_60

blue color: fraction of unit stride accesses

yellow: "fixed" stride accesses ratio

red color: fraction of irregular (variable stride) accesses

ID		Stride	Type	Source	Site Name	Variable
⊟P1	▨	3	16% / 84% / 0%	Mixed strides		
⊞P11	▨	0; 1				
⊟P12	▨	-289559; -274359; -14477; -13717; -13679; 723; 302519;				

```
1246 #endif
1247                 for (int m=1; m<=half; m++) {
1248                   nextx = fCppMod(i + lbv[3*m]
1249                   nexty = fCppMod(j + lbv[3*m+
1250                   nextz = fCppMod(k + lbv[3*m+
```

```
1251                   ilnext = (nextx * Ymax + nex
1252 #ifndef SWAP_OVERLAP
1253   fSwapPair (lbf[il*lbsitelength + l*lbsy.
```

16%:percentage of memory instructions with unit stride or stride 0 accesses

Unit stride (stride 1) = Instruction accesses memory that consistently changes by one element from iteration to iteration

Stride 0 = Instruction accesses the same memory from iteration to iteration

84%: percentage of memory instructions with fixed or constant non-unit stride accesses

Constant stride (stride N) = Instruction accesses memory by N elements from iteration to iteration

Example: for the double floating point type, stride 4 means the memory address accessed by this instruction increased by 32 bytes, (4*sizeof(double)) with each iteration

0%: percentage of memory instructions with irregular (variable or random) stride accesses

Irregular stride = Instruction accesses memory addresses that change by an unpredictable number of elements from iteration to iteration

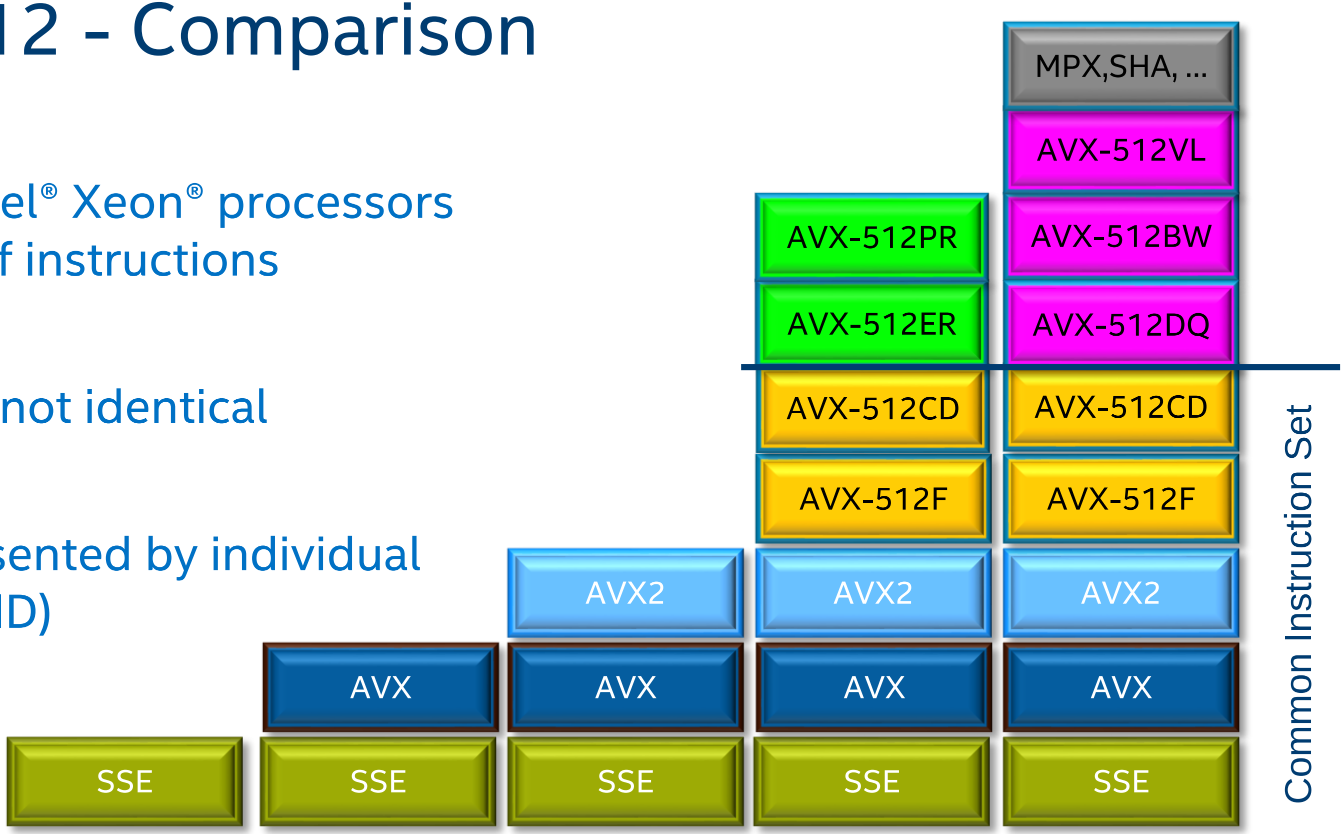
Typically observed for indirect indexed array accesses, for example, a[index[i]]

- gather (irregular) accesses, detected for v(p)gather* instructions on AVX2 Instruction Set Architecture

AVX-512 PROFILING WITH INTEL ADVISOR

Intel® AVX-512 - Comparison

- KNL and future Intel® Xeon® processors share a large set of instructions
- But some sets are not identical
- Subsets are represented by individual feature flags (CPUID)

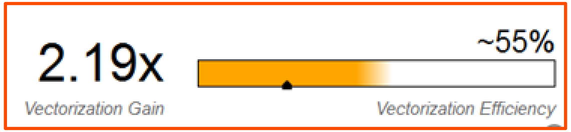


Intel® microarchitecture code name …	NHM	SNB	HSW	KNL	Next Intel Xeon
					MPX,SHA, …
					AVX-512VL
				AVX-512PR	AVX-512BW
				AVX-512ER	AVX-512DQ
				AVX-512CD	AVX-512CD
				AVX-512F	AVX-512F
			AVX2	AVX2	AVX2
		AVX	AVX	AVX	AVX
	SSE	SSE	SSE	SSE	SSE

Common Instruction Set

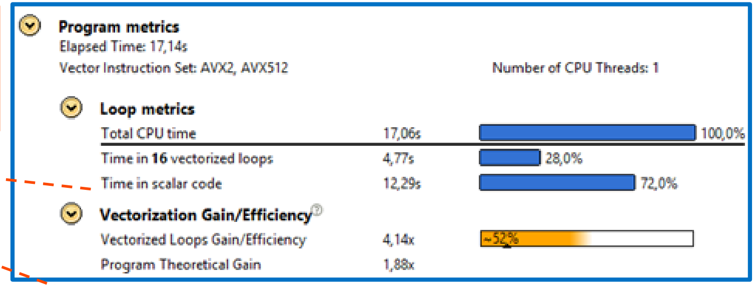
Optimization Notice

Advisor 2017: **AVX-512** specific performance insights



2.19x
Vectorization Gain

~55%
Vectorization Efficiency



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

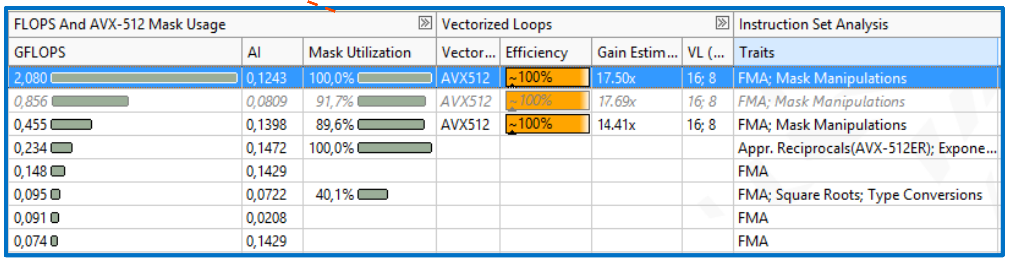
Loop metrics

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

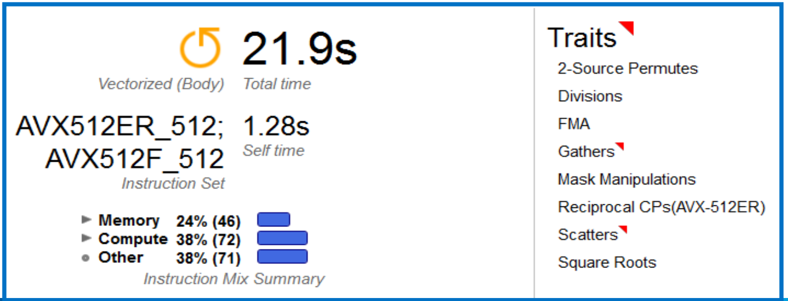
Vectorization Gain/Efficiency

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

- **Native AVX-512 profiling** on KNL
- Precise **FLOPs and Mask Utilization** profiler
- AVX-512 **Advices and "Traits"**
- And more..
  - Performance **Summary for** AVX-512 codes
  - **AVX-512 Gather/Scatter Profiler**
- No access to AVX-512 Hardware yet?
  - Explore AVX-512 code with –axcode flags and new Advisor Survey capability!



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



Vectorized (Body)

21.9s
Total time

AVX512ER_512; AVX512F_512
Instruction Set

1.28s
Self time

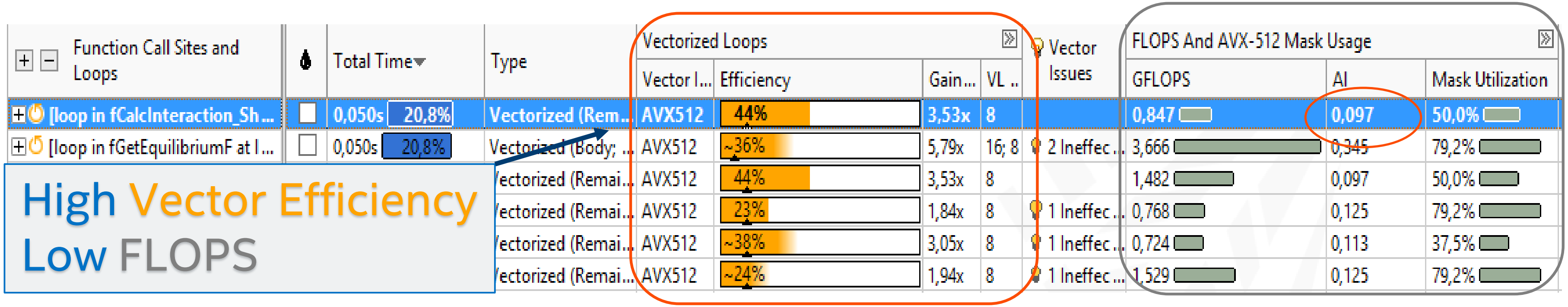
▶ Memory	24% (46)	
▶ Compute	38% (72)	
● Other	38% (71)	

Instruction Mix Summary

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

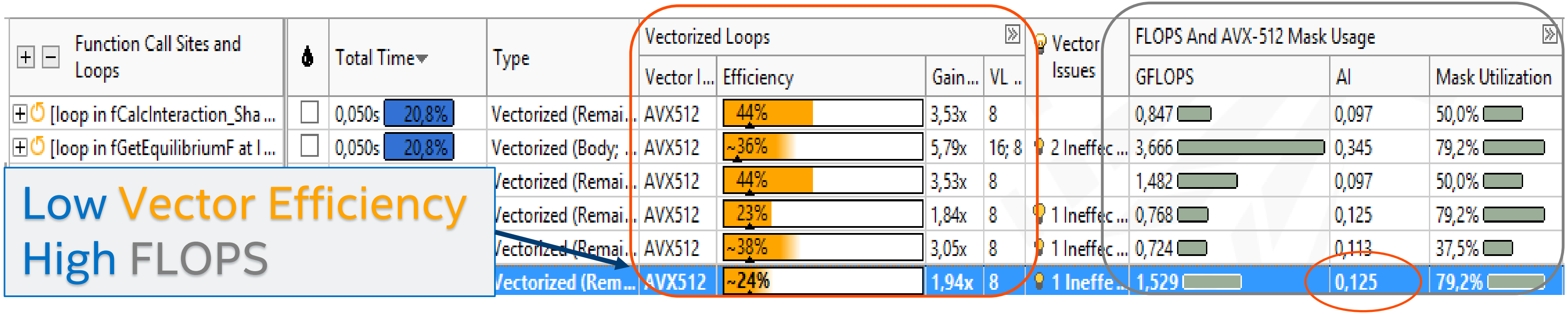
Optimization Notice

General efficiency (*FLOPS*) vs. VPU-centric efficiency (Vector Efficiency)



	Function Call Sites and Loops		Total Time	Type	Vectorized Loops				Vector Issues	FLOPS And AVX-512 Mask Usage		
					Vector I...	Efficiency	Gain...	VL ..		GFLOPS	AI	Mask Utilization
⊞	[loop in fCalcInteraction_Sh...	□	0,050s 20,8%	Vectorized (Rem...	AVX512	44%	3,53x	8		0,847	0,097	50,0%
⊞	[loop in fGetEquilibriumF at l...	□	0,050s 20,8%	Vectorized (Body; ...	AVX512	~36%	5,79x	16; 8	2 Ineffec...	3,666	0,345	79,2%
				ectorized (Remai...	AVX512	44%	3,53x	8		1,482	0,097	50,0%
				ectorized (Remai...	AVX512	23%	1,84x	8	1 Ineffec...	0,768	0,125	79,2%
				ectorized (Remai...	AVX512	~38%	3,05x	8	1 Ineffec...	0,724	0,113	37,5%
				ectorized (Remai...	AVX512	~24%	1,94x	8	1 Ineffec...	1,529	0,125	79,2%

High Vector Efficiency
Low FLOPS



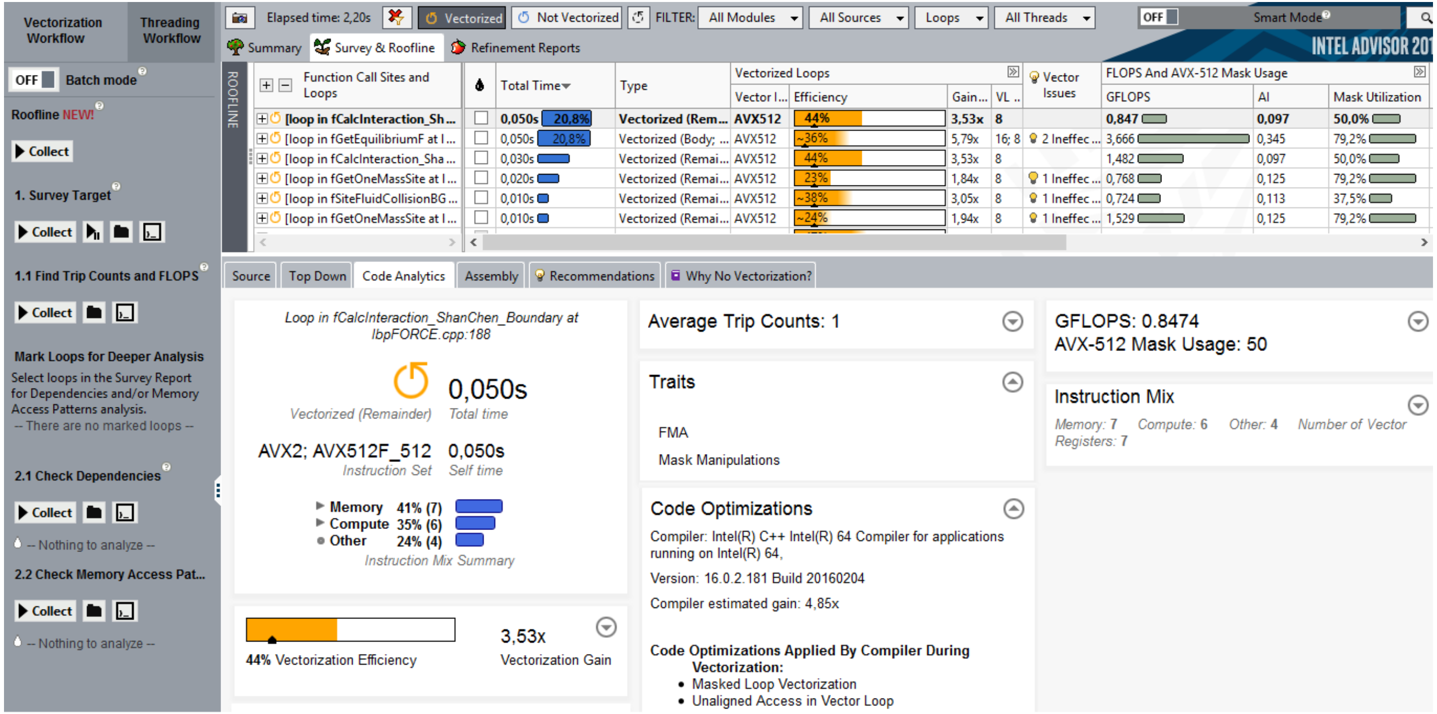
	Function Call Sites and Loops		Total Time	Type	Vectorized Loops				Vector Issues	FLOPS And AVX-512 Mask Usage		
					Vector I...	Efficiency	Gain...	VL ..		GFLOPS	AI	Mask Utilization
⊞	[loop in fCalcInteraction_Sha...	□	0,050s 20,8%	Vectorized (Remai...	AVX512	44%	3,53x	8		0,847	0,097	50,0%
⊞	[loop in fGetEquilibriumF at l...	□	0,050s 20,8%	Vectorized (Body; ...	AVX512	~36%	5,79x	16; 8	2 Ineffec...	3,666	0,345	79,2%
				ectorized (Remai...	AVX512	44%	3,53x	8		1,482	0,097	50,0%
				ectorized (Remai...	AVX512	23%	1,84x	8	1 Ineffec...	0,768	0,125	79,2%
				ectorized (Remai...	AVX512	~38%	3,05x	8	1 Ineffec...	0,724	0,113	37,5%
				ectorized (Rem...	AVX512	~24%	1,94x	8	1 Ineffe...	1,529	0,125	79,2%

Low Vector Efficiency
High FLOPS

Optimization Notice

Survey+FLOPs Report on AVX-512:

FLOP/s, Bytes and AI, Masks and Efficiency



Mark Loops for Deeper Analysis

Select loops in the Survey Report for Dependencies and/or Memory Access Patterns analysis.

-- There are no marked loops --

Function Call Sites and Loops		Total Time	Type	Vectorized Loops				Vector Issues	FLOPS And AVX-512 Mask Usage		
				Vector I...	Efficiency	Gain...	VL ..		GFLOPS	AI	Mask Utilization
[loop in fCalcInteraction_Sh...	□	**0,050s** 20,8%	**Vectorized (Rem...**	**AVX512**	**44%**	**3,53x**	**8**		**0,847**	**0,097**	**50,0%**
[loop in fGetEquilibriumF at l...	□	0,050s 20,8%	Vectorized (Body; ...	AVX512	~36%	5,79x	16; 8	2 Ineffec ...	3,666	0,345	79,2%
[loop in fCalcInteraction_Sha ...	□	0,030s	Vectorized (Remai...	AVX512	44%	3,53x	8		1,482	0,097	50,0%
[loop in fGetOneMassSite at l ...	□	0,020s	Vectorized (Remai...	AVX512	23%	1,84x	8	1 Ineffec ...	0,768	0,125	79,2%
[loop in fSiteFluidCollisionBG ...	□	0,010s	Vectorized (Remai...	AVX512	~38%	3,05x	8	1 Ineffec ...	0,724	0,113	37,5%
[loop in fGetOneMassSite at l ...	□	0,010s	Vectorized (Remai...	AVX512	~24%	1,94x	8	1 Ineffec ...	1,529	0,125	79,2%

Loop in fCalcInteraction_ShanChen_Boundary at lbpFORCE.cpp:188

0,050s *Total time*

Vectorized (Remainder)

AVX2; AVX512F_512 *Instruction Set*

0,050s *Self time*

▸ Memory	41% (7)	
▸ Compute	35% (6)	
● Other	24% (4)	

Instruction Mix Summary

44% Vectorization Efficiency

3,53x Vectorization Gain

Average Trip Counts: 1

Traits

FMA

Mask Manipulations

Code Optimizations

Compiler: Intel(R) C++ Intel(R) 64 Compiler for applications running on Intel(R) 64,

Version: 16.0.2.181 Build 20160204

Compiler estimated gain: 4,85x

Code Optimizations Applied By Compiler During Vectorization:

- Masked Loop Vectorization
- Unaligned Access in Vector Loop

GFLOPS: 0.8474

AVX-512 Mask Usage: 50

Instruction Mix

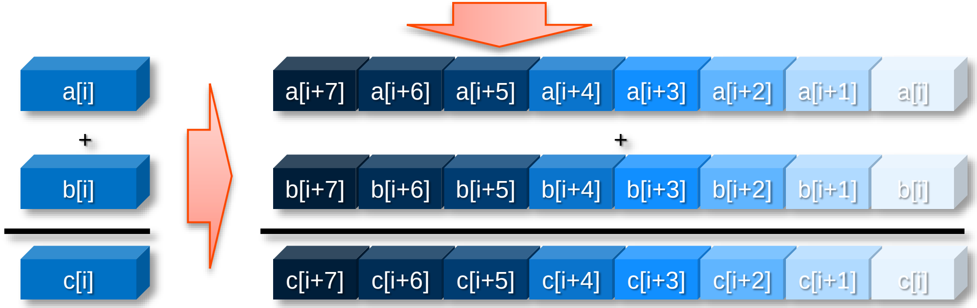
Memory: 7 *Compute:* 6 *Other:* 4 *Number of Vector Registers:* 7

Optimization Notice

Why Mask Utilization Important?

```
for(i = 0; i <= MAX; i++)
   c[i] = a[i] + b[i];
```

100%



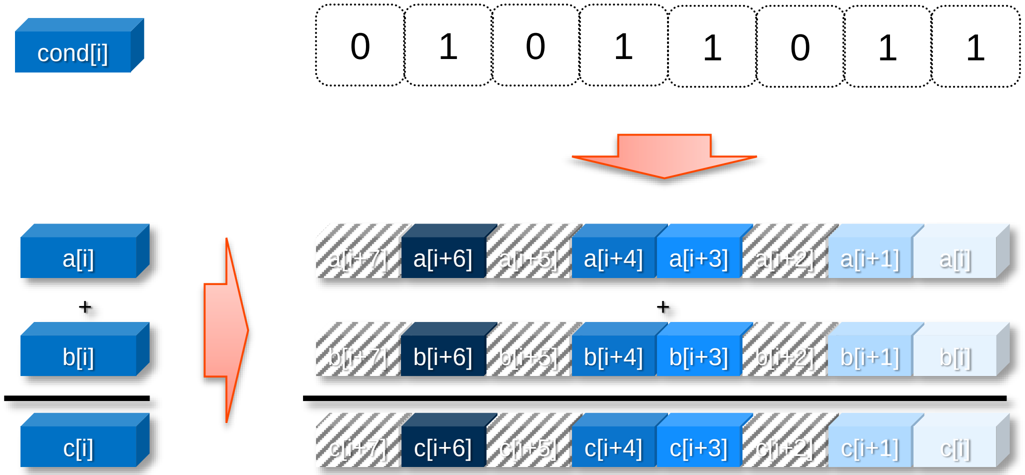
Why Mask Utilization Important?

3 elements suppressed

SIMD Utilization = 5/8

62.5%

```
for(i = 0; i <= MAX; i++)
   if (cond(i))
      c[i] = a[i] + b[i];
```



cond[i]

0	1	0	1	1	0	1	1

a[i]
+
b[i]
c[i]
a[i+7] a[i+6] a[i+5] a[i+4] a[i+3] a[i+2] a[i+1] a[i]
+
b[i+7] b[i+6] b[i+5] b[i+4] b[i+3] b[i+2] b[i+1] b[i]
c[i+7] c[i+6] c[i+5] c[i+4] c[i+3] c[i+2] c[i+1] c[i]

Optimization Notice

AVX-512 Mask Registers

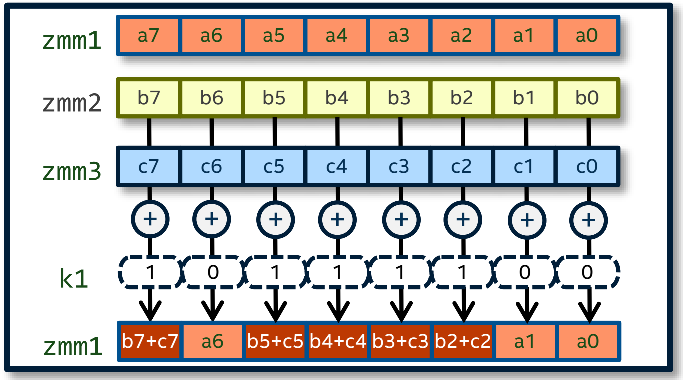
8 Mask registers of size 64-bits

- k1-k7 can be used for predication
  - k0 can be used as a destination or source for mask manipulation operations

4 different mask granularities. For instance, at 512b:

- Packed Integer Byte use mask bits [63:0]
  - `VPADDB zmm1 {k1}, zmm2, zmm3`
- Packed Integer Word use mask bits [31:0]
  - `VPADDW zmm1 {k1}, zmm2, zmm3`
- Packed IEEE FP32 and Integer Dword use mask bits [15:0]
  - `VADDPS zmm1 {k1}, zmm2, zmm3`
- Packed IEEE FP64 and Integer Qword use mask bits [7:0]
  - `VADDPD zmm1 {k1}, zmm2, zmm3`

`VADDPD zmm1 {k1}, zmm2, zmm3`



zmm1: a7 a6 a5 a4 a3 a2 a1 a0

zmm2: b7 b6 b5 b4 b3 b2 b1 b0

zmm3: c7 c6 c5 c4 c3 c2 c1 c0

+ + + + + + + +

k1: 1 0 1 1 1 1 0 0

zmm1: b7+c7 a6 b5+c5 b4+c4 b3+c3 b2+c2 a1 a0

		Vector Length		
		128	256	512
element size	Byte	16	32	64
	Word	8	16	32
	Dword/SP	4	8	16
	Qword/DP	2	4	8

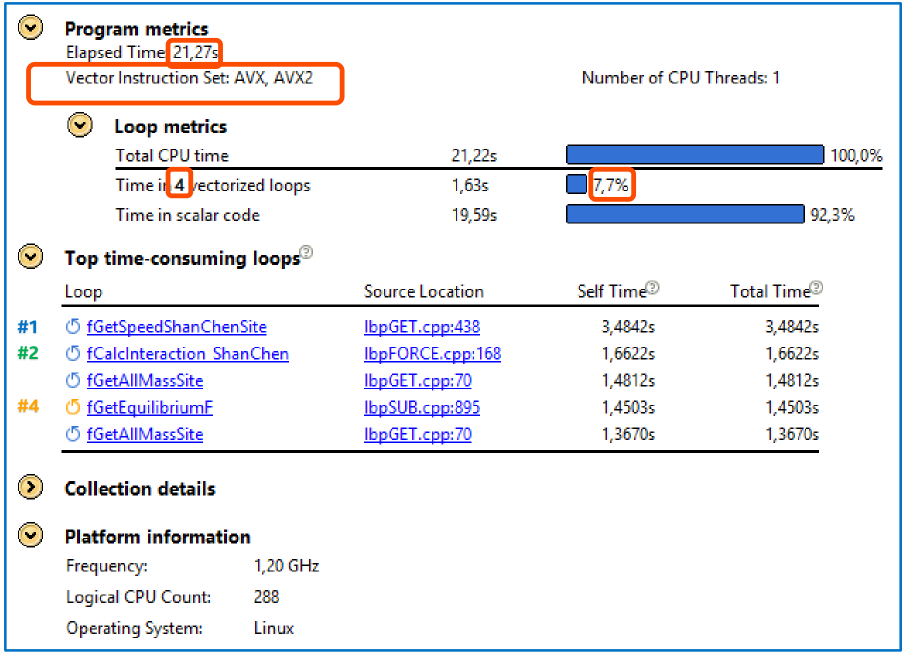
Optimization Notice

Mask Utilization and FLOPS profiler

- Long-waiting in HPC: accurate HW independent FLOPs measurement tool
- Not just count FLOPs. Has following additions:
  - (AVX-512 only) Mask-aware. Masked-Memory/Unmasked-Compute pattern aware
  - Unique capability to correlate FLOPs with performance data (obtained without instrumentation). Gives FLOPs/s.
- Lightweight instrumentation, PIN-based, benefits from “threadchecker tools” and more generally Advisor framework integration.

Characterize and compare

AVX-512 against AVX2 versions (on Xeon Phi)



Program metrics

Elapsed Time: 21,27s

Vector Instruction Set: AVX, AVX2

Number of CPU Threads: 1

Loop metrics

Total CPU time	21,22s	100,0%
Time in 4 vectorized loops	1,63s	7,7%
Time in scalar code	19,59s	92,3%

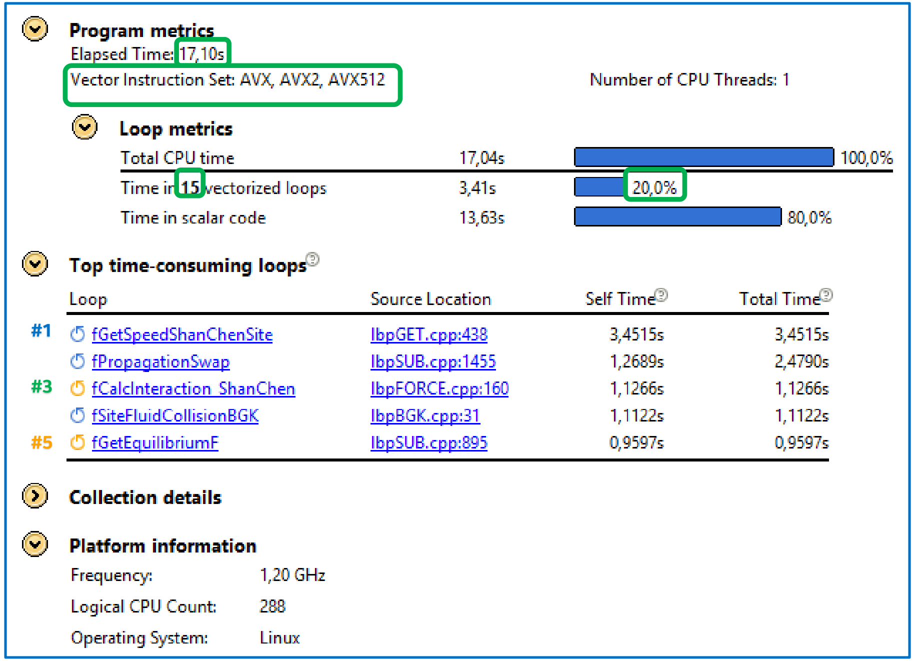
Top time-consuming loops

	Loop	Source Location	Self Time	Total Time
#1	fGetSpeedShanChenSite	lbpGET.cpp:438	3,4842s	3,4842s
#2	fCalcInteraction_ShanChen	lbpFORCE.cpp:168	1,6622s	1,6622s
	fGetAllMassSite	lbpGET.cpp:70	1,4812s	1,4812s
#4	fGetEquilibriumF	lbpSUB.cpp:895	1,4503s	1,4503s
	fGetAllMassSite	lbpGET.cpp:70	1,3670s	1,3670s

Collection details

Platform information

Frequency:	1,20 GHz
Logical CPU Count:	288
Operating System:	Linux



Program metrics

Elapsed Time: 17,10s

Vector Instruction Set: AVX, AVX2, AVX512

Number of CPU Threads: 1

Loop metrics

Total CPU time	17,04s	100,0%
Time in 15 vectorized loops	3,41s	20,0%
Time in scalar code	13,63s	80,0%

Top time-consuming loops

	Loop	Source Location	Self Time	Total Time
#1	fGetSpeedShanChenSite	lbpGET.cpp:438	3,4515s	3,4515s
	fPropagationSwap	lbpSUB.cpp:1455	1,2689s	2,4790s
#3	fCalcInteraction_ShanChen	lbpFORCE.cpp:160	1,1266s	1,1266s
	fSiteFluidCollisionBGK	lbpBGK.cpp:31	1,1122s	1,1122s
#5	fGetEquilibriumF	lbpSUB.cpp:895	0,9597s	0,9597s

Collection details

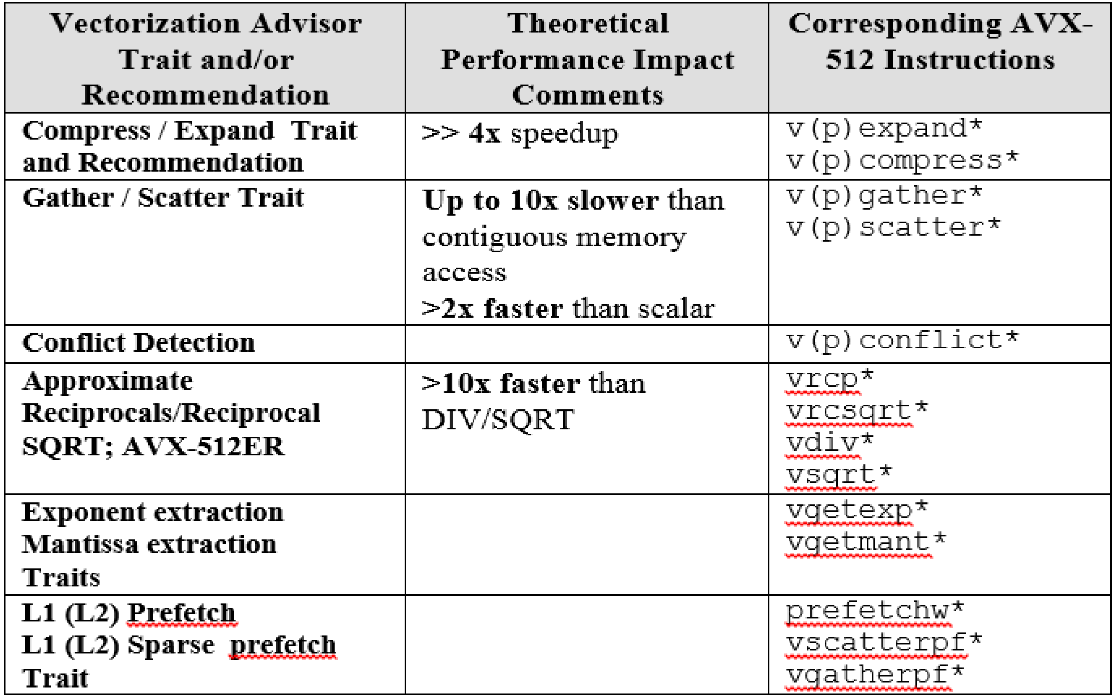
Platform information

Frequency:	1,20 GHz
Logical CPU Count:	288
Operating System:	Linux

Highlight “impactful” AVX-512 instructions.

Survey Static Analysis - AVX-512 ***“Traits”***

Presence of remarkable performance-impactful (negative or positive impact) instructions



Vectorization Advisor Trait and/or Recommendation	**Theoretical Performance Impact Comments**	**Corresponding AVX-512 Instructions**

Compress / Expand Trait and Recommendation	>> **4x** speedup	`v(p)expand*`
		`v(p)compress*`

Gather / Scatter Trait	**Up to 10x slower** than contiguous memory access
	>2x faster than scalar
		`v(p)gather*`
		`v(p)scatter*`

Conflict Detection		`v(p)conflict*`

Approximate Reciprocals/Reciprocal SQRT; AVX-512ER	**>10x faster** than DIV/SQRT	`vrcp*`
		`vrcsqrt*`
		`vdiv*`
		`vsqrt*`

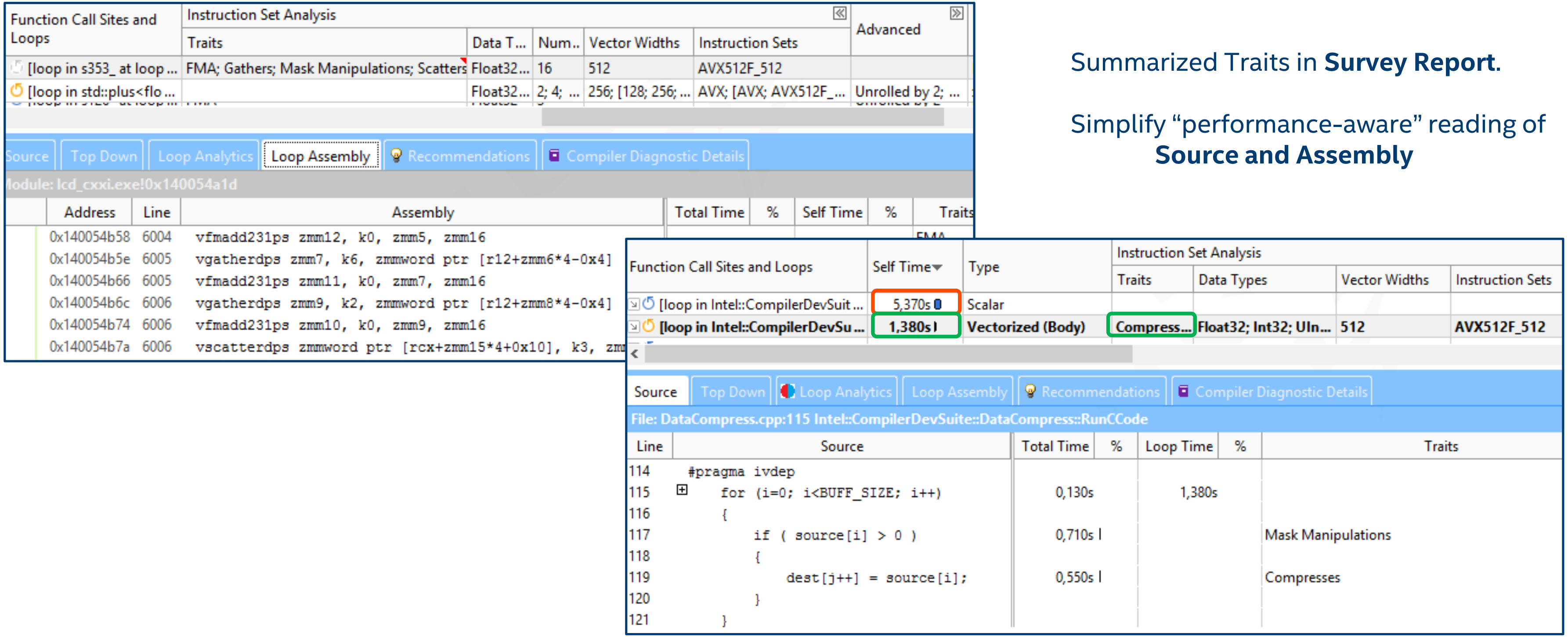
Exponent extraction Mantissa extraction Traits		`vgetexp*` `vgetmant*`

L1 (L2) Prefetch
L1 (L2) Sparse prefetch Trait
		`prefetchw*`
		`vscatterpf*`
		`vgatherpf*`

Optimization Notice

Highlight “impactful” AVX-512 instructions.

Survey Static Analysis - AVX-512 ***“Traits”***



Function Call Sites and Loops	Instruction Set Analysis					Advanced
	Traits	Data T...	Num..	Vector Widths	Instruction Sets	
[loop in s353_ at loop ...	FMA; Gathers; Mask Manipulations; Scatters	Float32...	16	512	AVX512F_512	
[loop in std::plus<flo ...		Float32...	2; 4; ...	256; [128; 256; ...	AVX; [AVX; AVX512F_...	Unrolled by 2; ...

Module: lcd_cxxi.exe!0x140054a1d

Address	Line	Assembly
0x140054b58	6004	`vfmadd231ps zmm12, k0, zmm5, zmm16`
0x140054b5e	6005	`vgatherdps zmm7, k6, zmmword ptr [r12+zmm6*4-0x4]`
0x140054b66	6005	`vfmadd231ps zmm11, k0, zmm7, zmm16`
0x140054b6c	6006	`vgatherdps zmm9, k2, zmmword ptr [r12+zmm8*4-0x4]`
0x140054b74	6006	`vfmadd231ps zmm10, k0, zmm9, zmm16`
0x140054b7a	6006	`vscatterdps zmmword ptr [rcx+zmm15*4+0x10], k3, zm`

Function Call Sites and Loops	Self Time▾	Type	Instruction Set Analysis			
			Traits	Data Types	Vector Widths	Instruction Sets
[loop in Intel::CompilerDevSuit ...	5,370s	Scalar				
[loop in Intel::CompilerDevSu ...	**1,380s**	**Vectorized (Body)**	**Compress...**	**Float32; Int32; UIn...**	**512**	**AVX512F_512**

File: DataCompress.cpp:115 Intel::CompilerDevSuite::DataCompress::RunCCode

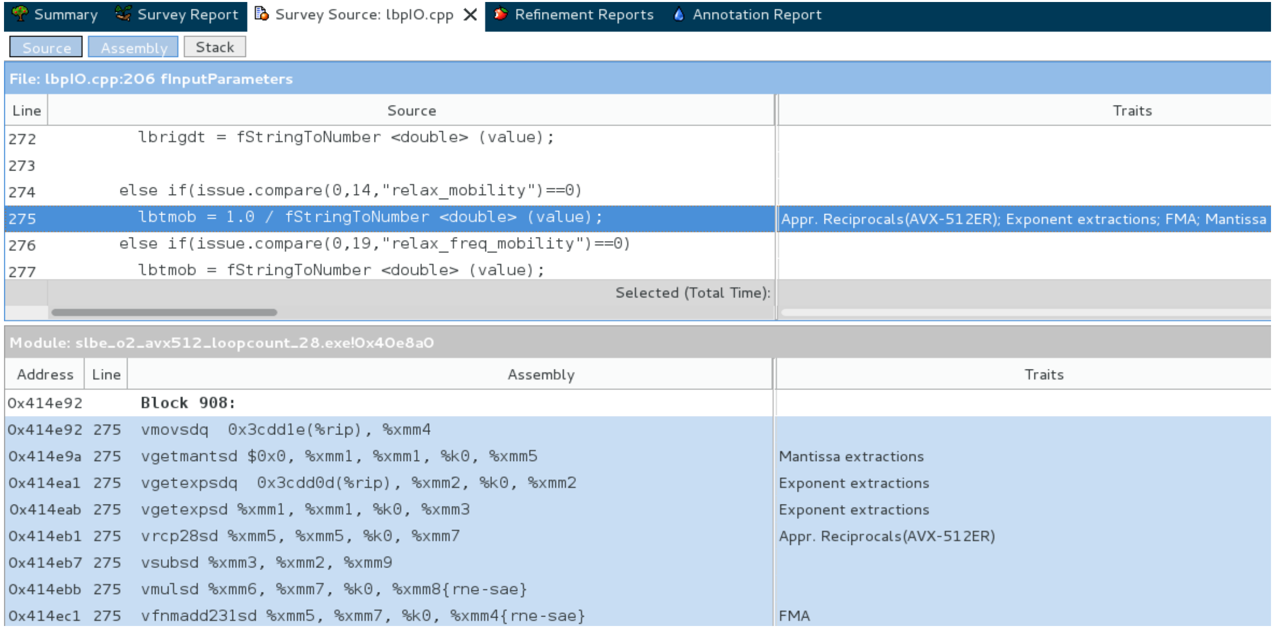
Line	Source	Total Time	%	Loop Time	%	Traits
114	`#pragma ivdep`					
115	`for (i=0; i<BUFF_SIZE; i++)`	0,130s		1,380s		
116	`{`					
117	`if ( source[i] > 0 )`	0,710s				Mask Manipulations
118	`{`					
119	`dest[j++] = source[i];`	0,550s				Compresses
120	`}`					
121	`}`					

Summarized Traits in **Survey Report**.

Simplify “performance-aware” reading of **Source and Assembly**

AVX-512 Also Benefit Scalar Code a lot…

Survey Static Analysis - AVX-512 ***“Traits”***



File: lbpIO.cpp:206 fInputParameters

Line	Source	Traits
272	`      lbrigdt = fStringToNumber <double> (value);`	
273		
274	`    else if(issue.compare(0,14,"relax_mobility")==0)`	
275	`      lbtmob = 1.0 / fStringToNumber <double> (value);`	Appr. Reciprocals(AVX-512ER); Exponent extractions; FMA; Mantissa
276	`    else if(issue.compare(0,19,"relax_freq_mobility")==0)`	
277	`      lbtmob = fStringToNumber <double> (value);`	
	Selected (Total Time):	

Module: slbe_o2_avx512_loopcount_28.exe!0x40e8a0

Address	Line	Assembly	Traits
0x414e92		**Block 908:**	
0x414e92	275	`vmovsdq  0x3cdd1e(%rip), %xmm4`	
0x414e9a	275	`vgetmantsd $0x0, %xmm1, %xmm1, %k0, %xmm5`	Mantissa extractions
0x414ea1	275	`vgetexpsdq  0x3cdd0d(%rip), %xmm2, %k0, %xmm2`	Exponent extractions
0x414eab	275	`vgetexpsd %xmm1, %xmm1, %k0, %xmm3`	Exponent extractions
0x414eb1	275	`vrcp28sd %xmm5, %xmm5, %k0, %xmm7`	Appr. Reciprocals(AVX-512ER)
0x414eb7	275	`vsubsd %xmm3, %xmm2, %xmm9`	
0x414ebb	275	`vmulsd %xmm6, %xmm7, %k0, %xmm8{rne-sae}`	
0x414ec1	275	`vfnmadd231sd %xmm5, %xmm7, %k0, %xmm4{rne-sae}`	FMA

Gather/Scatter Analysis

Motivation

AVX-512 Gather/Scatter-based vectorization.

Much wider usage than before :

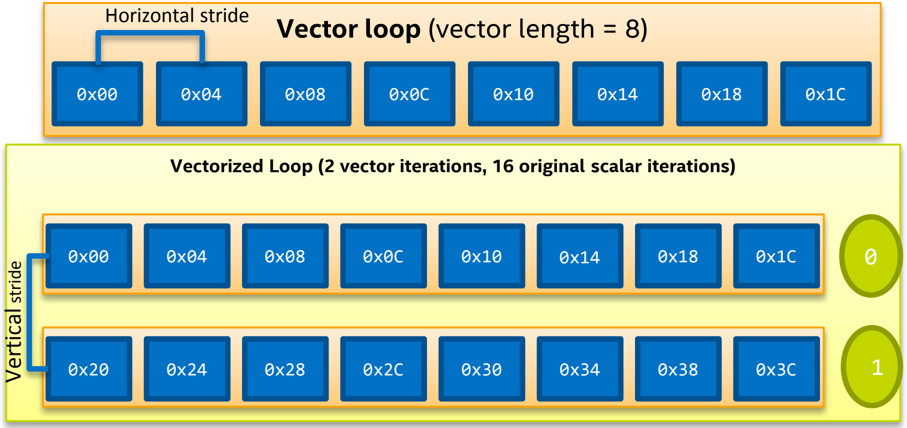
- Makes much more codes (profitably) vectorizable
- Gives good average performance, but often far from optimal.

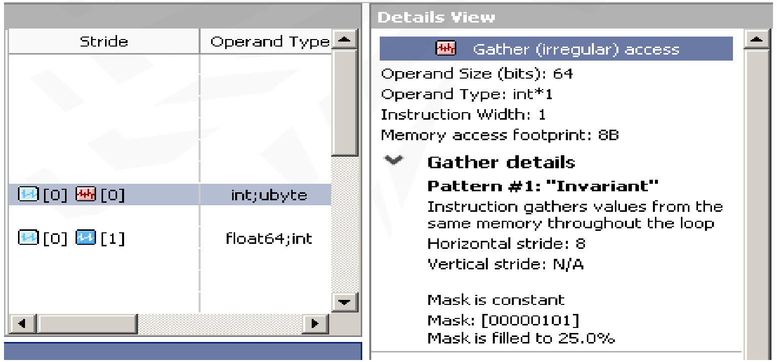
Could be 2x faster than scalar mov

Could be 10x slower than vmovp*

Gather/Scatter Analysis

Advisor MAP detects gather “offset patterns”.





Stride	Operand Type
[0] [0]	int;ubyte
[0] [1]	float64;int

Details View

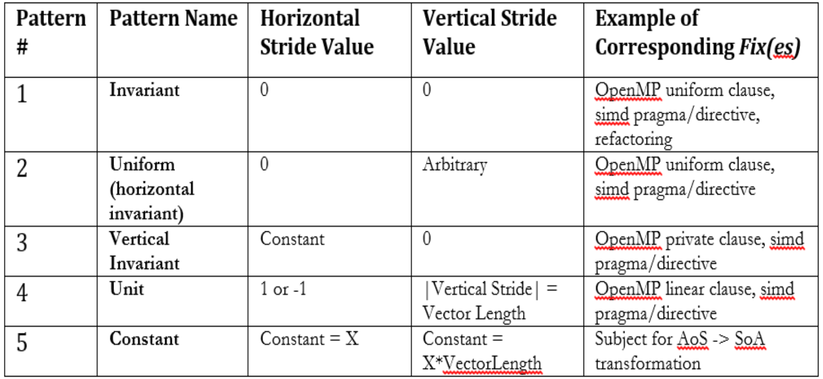
Gather (irregular) access

Operand Size (bits): 64
Operand Type: int*1
Instruction Width: 1
Memory access footprint: 8B

Gather details

Pattern #1: "Invariant"
Instruction gathers values from the same memory throughout the loop
Horizontal stride: 8
Vertical stride: N/A

Mask is constant
Mask: [00000101]
Mask is filled to 25.0%



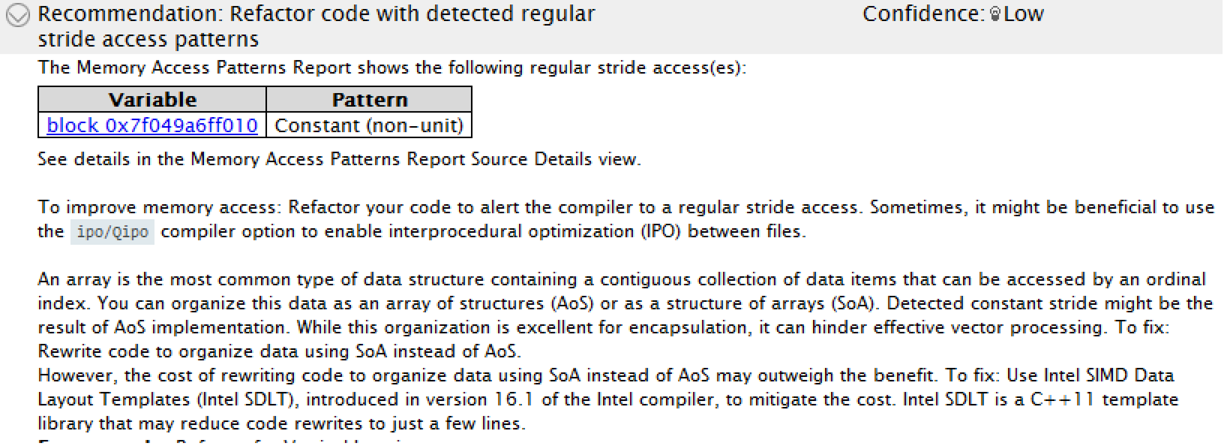
Pattern #	Pattern Name	Horizontal Stride Value	Vertical Stride Value	Example of Corresponding *Fix(es)*
1	Invariant	0	0	OpenMP uniform clause, simd pragma/directive, refactoring
2	Uniform (horizontal invariant)	0	Arbitrary	OpenMP uniform clause, simd pragma/directive
3	Vertical Invariant	Constant	0	OpenMP private clause, simd pragma/directive
4	Unit	1 or -1	|Vertical Stride| = Vector Length	OpenMP linear clause, simd pragma/directive
5	Constant	Constant = X	Constant = X*VectorLength	Subject for AoS -> SoA transformation

Optimization Notice

Gather/scatter issue improvements

Compiler may generate gather/scatter instructions despite regular access pattern. In this case, performance can be improved by refactoring the code.

- Detecting regular patterns taking into account masking instructions
- Added new access pattern for gather profiling – Constant (Non-Unit Stride) with adjusted recommendation to transform AOS to SOA



Recommendation: Refactor code with detected regular stride access patterns   Confidence: Low

The Memory Access Patterns Report shows the following regular stride access(es):

Variable	Pattern
block 0x7f049a6ff010	Constant (non-unit)

See details in the Memory Access Patterns Report Source Details view.

To improve memory access: Refactor your code to alert the compiler to a regular stride access. Sometimes, it might be beneficial to use the `ipo/Qipo` compiler option to enable interprocedural optimization (IPO) between files.

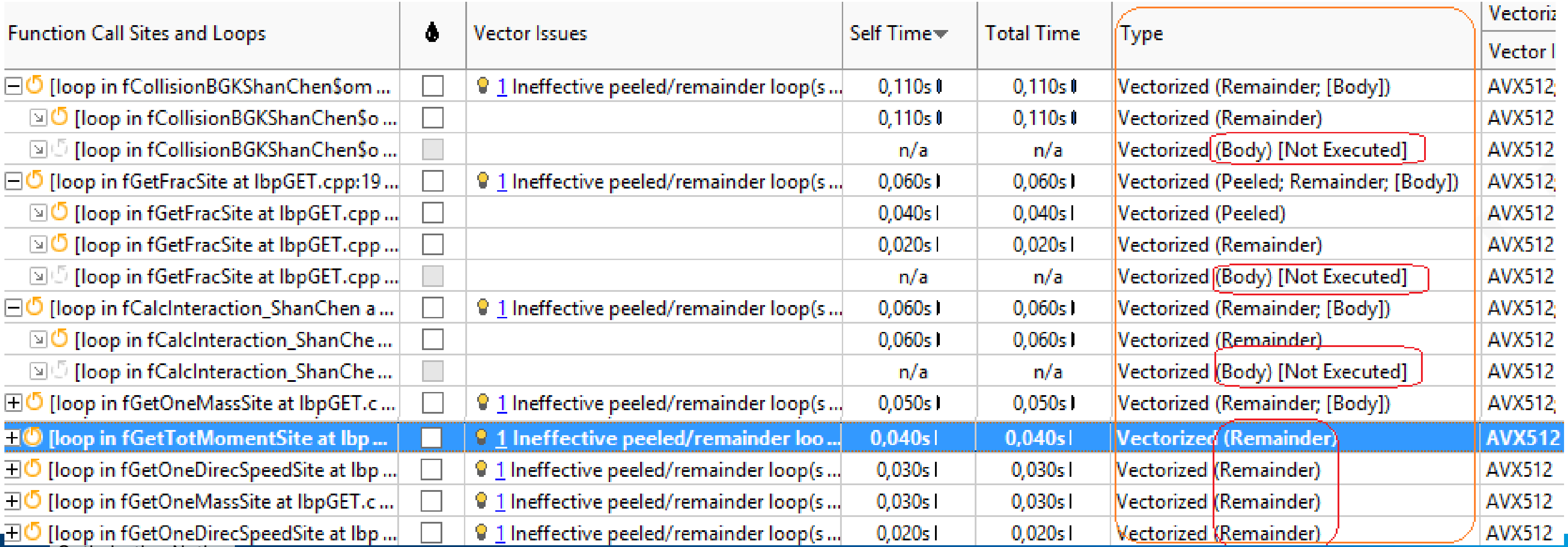
An array is the most common type of data structure containing a contiguous collection of data items that can be accessed by an ordinal index. You can organize this data as an array of structures (AoS) or as a structure of arrays (SoA). Detected constant stride might be the result of AoS implementation. While this organization is excellent for encapsulation, it can hinder effective vector processing. To fix: Rewrite code to organize data using SoA instead of AoS.
However, the cost of rewriting code to organize data using SoA instead of AoS may outweigh the benefit. To fix: Use Intel SIMD Data Layout Templates (Intel SDLT), introduced in version 16.1 of the Intel compiler, to mitigate the cost. Intel SDLT is a C++11 template library that may reduce code rewrites to just a few lines.

AVX-512-specific performance trade-offs

Advisor AVX-512 Recommendations

Increasing Vector Register Size ->

Increase fraction of time spent in Remainders

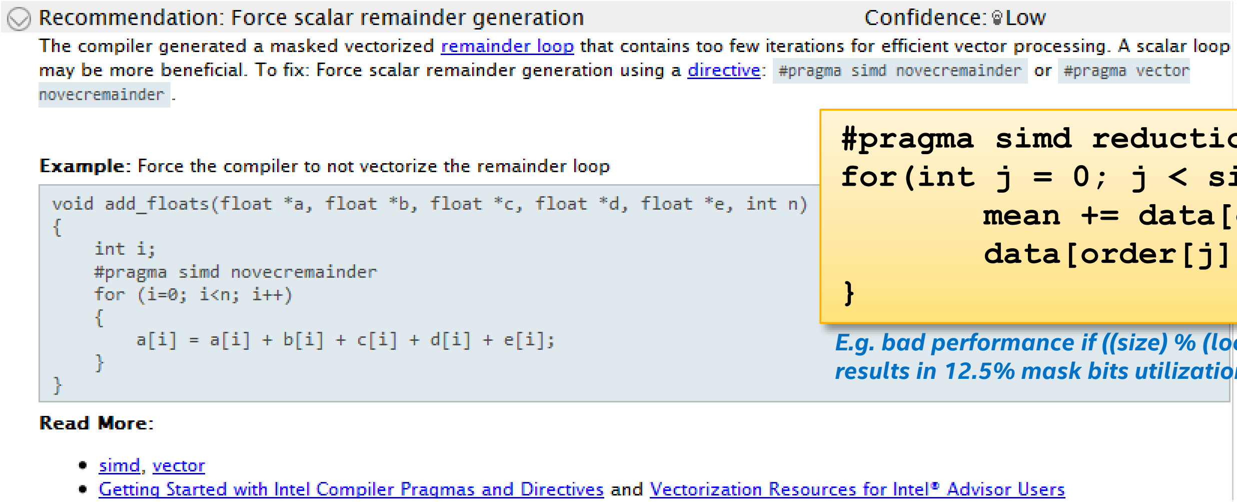


Function Call Sites and Loops		Vector Issues	Self Time	Total Time	Type	Vectoriz Vector I
⊟ [loop in fCollisionBGKShanChen$om ...	☐	1 Ineffective peeled/remainder loop(s ...	0,110s	0,110s	Vectorized (Remainder; [Body])	AVX512;
[loop in fCollisionBGKShanChen$o ...	☐		0,110s	0,110s	Vectorized (Remainder)	AVX512
[loop in fCollisionBGKShanChen$o ...	☐		n/a	n/a	Vectorized (Body) [Not Executed]	AVX512
⊟ [loop in fGetFracSite at lbpGET.cpp:19 ...	☐	1 Ineffective peeled/remainder loop(s ...	0,060s	0,060s	Vectorized (Peeled; Remainder; [Body])	AVX512;
[loop in fGetFracSite at lbpGET.cpp ...	☐		0,040s	0,040s	Vectorized (Peeled)	AVX512
[loop in fGetFracSite at lbpGET.cpp ...	☐		0,020s	0,020s	Vectorized (Remainder)	AVX512
[loop in fGetFracSite at lbpGET.cpp ...	☐		n/a	n/a	Vectorized (Body) [Not Executed]	AVX512
⊟ [loop in fCalcInteraction_ShanChen a ...	☐	1 Ineffective peeled/remainder loop(s ...	0,060s	0,060s	Vectorized (Remainder; [Body])	AVX512;
[loop in fCalcInteraction_ShanChe ...	☐		0,060s	0,060s	Vectorized (Remainder)	AVX512
[loop in fCalcInteraction_ShanChe ...	☐		n/a	n/a	Vectorized (Body) [Not Executed]	AVX512
⊞ [loop in fGetOneMassSite at lbpGET.c ...	☐	1 Ineffective peeled/remainder loop(s ...	0,050s	0,050s	Vectorized (Remainder; [Body])	AVX512;
⊞ [loop in fGetTotMomentSite at lbp ...	☐	1 Ineffective peeled/remainder loo ...	0,040s	0,040s	Vectorized (Remainder)	AVX512
⊞ [loop in fGetOneDirecSpeedSite at lbp ...	☐	1 Ineffective peeled/remainder loop(s ...	0,030s	0,030s	Vectorized (Remainder)	AVX512
⊞ [loop in fGetOneMassSite at lbpGET.c ...	☐	1 Ineffective peeled/remainder loop(s ...	0,030s	0,030s	Vectorized (Remainder)	AVX512
⊞ [loop in fGetOneDirecSpeedSite at lbp ...	☐	1 Ineffective peeled/remainder loop(s ...	0,020s	0,020s	Vectorized (Remainder)	AVX512

Ineffective masked remainder for AVX512 codes

- Compiler generates vector masked remainder due to the number of iterations (trip count) not being divisible by vector length. In case of executing a few iterations, it is ineffective comparing to scalar versions of the loop.

- Using AVX512 mask profiler and trip-counts data to prove the issue.



Recommendation: Force scalar remainder generation　　Confidence: Low

The compiler generated a masked vectorized remainder loop that contains too few iterations for efficient vector processing. A scalar loop may be more beneficial. To fix: Force scalar remainder generation using a directive: `#pragma simd novecremainder` or `#pragma vector novecremainder`.

Example: Force the compiler to not vectorize the remainder loop

```
void add_floats(float *a, float *b, float *c, float *d, float *e, int n)
{
    int i;
    #pragma simd novecremainder
    for (i=0; i<n; i++)
    {
        a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
    }
}
```

Read More:

- simd, vector
- Getting Started with Intel Compiler Pragmas and Directives and Vectorization Resources for Intel® Advisor Users

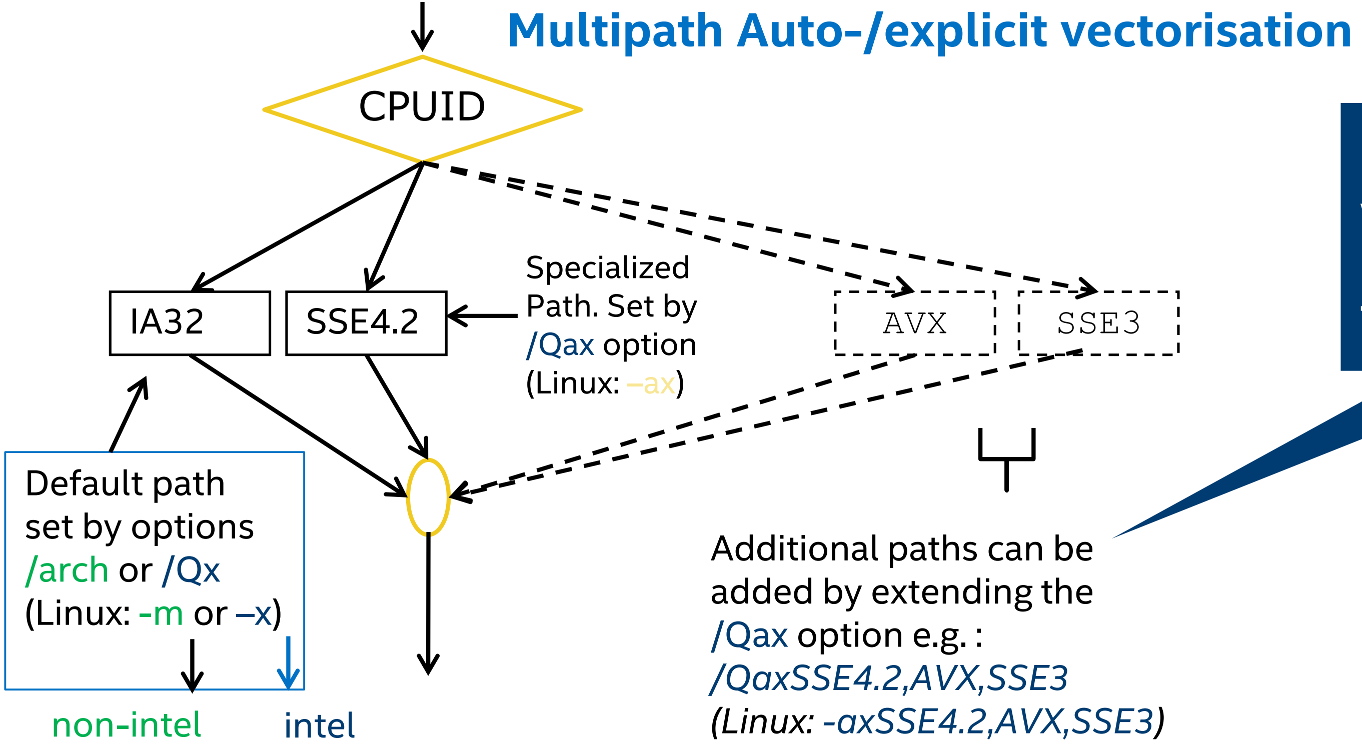
```
#pragma simd reduction(+:mean)
for(int j = 0; j < size; j++) {
        mean += data[order[j]] / N;
        data[order[j]] = 10.f / (j+1);
}
```

E.g. bad performance if ((size) % (loop_body_vl) == 1), in case of float number it results in 12.5% mask bits utilization only, in addition leads to gathers, scatters…

Optimization Notice

Start Tuning for AVX-512 without AVX-512 hardware

Intel® Advisor - Vectorization Advisor “axcode feature”



Specialized Path. Set by /Qax option (Linux: –ax)

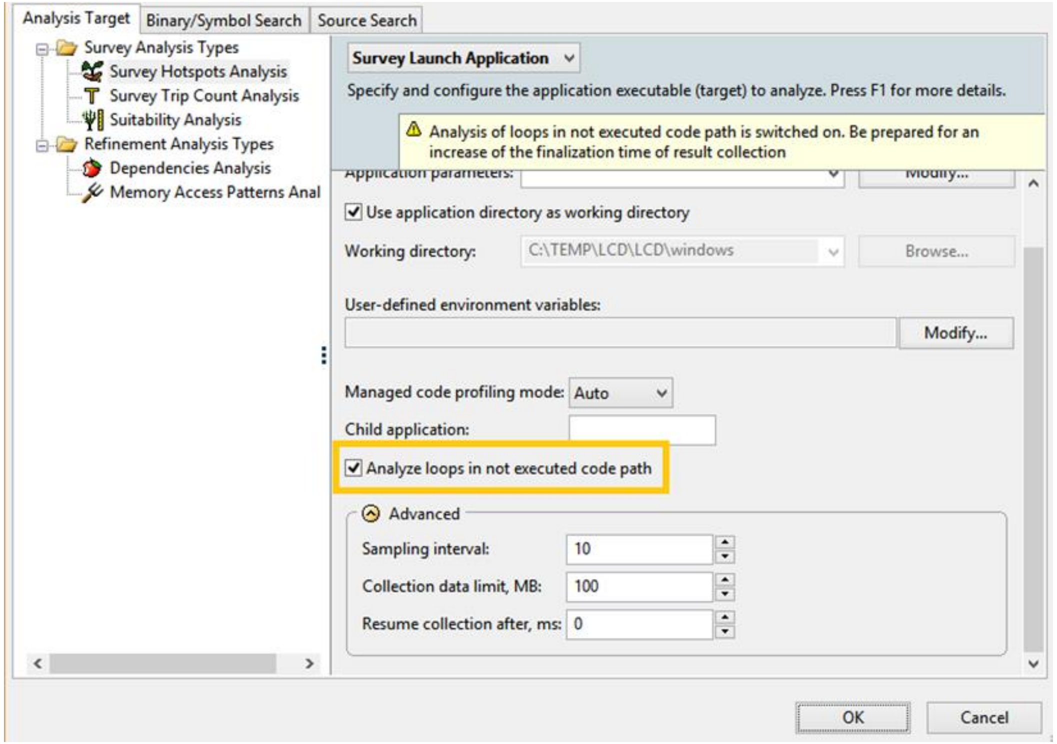
Default path set by options /arch or /Qx (Linux: -m or –x)

Additional paths can be added by extending the /Qax option e.g. : *\/QaxSSE4.2,AVX,SSE3* *(Linux: -axSSE4.2,AVX,SSE3)*

Use -ax option when compiling to create multiple paths through code

Optimization Notice

Viewing non-executed paths



Survey Launch Application

Specify and configure the application executable (target) to analyze. Press F1 for more details.

⚠ Analysis of loops in not executed code path is switched on. Be prepared for an increase of the finalization time of result collection

Application parameters: Modify...

☑ Use application directory as working directory

Working directory: C:\TEMP\LCD\LCD\windows Browse...

User-defined environment variables: Modify...

Managed code profiling mode: Auto

Child application:

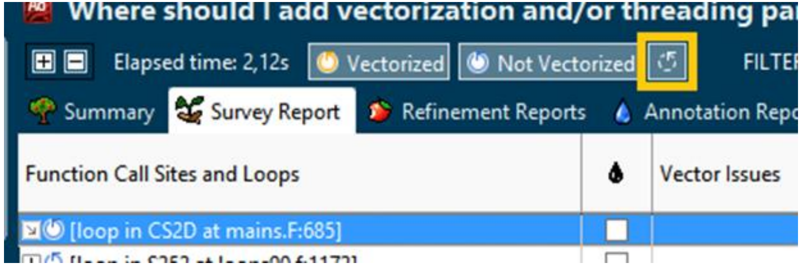
☑ Analyze loops in not executed code path

Advanced

Sampling interval: 10

Collection data limit, MB: 100

Resume collection after, ms: 0



Where should I add vectorization and/or threading pa

[loop in CS2D at mains.F:685]

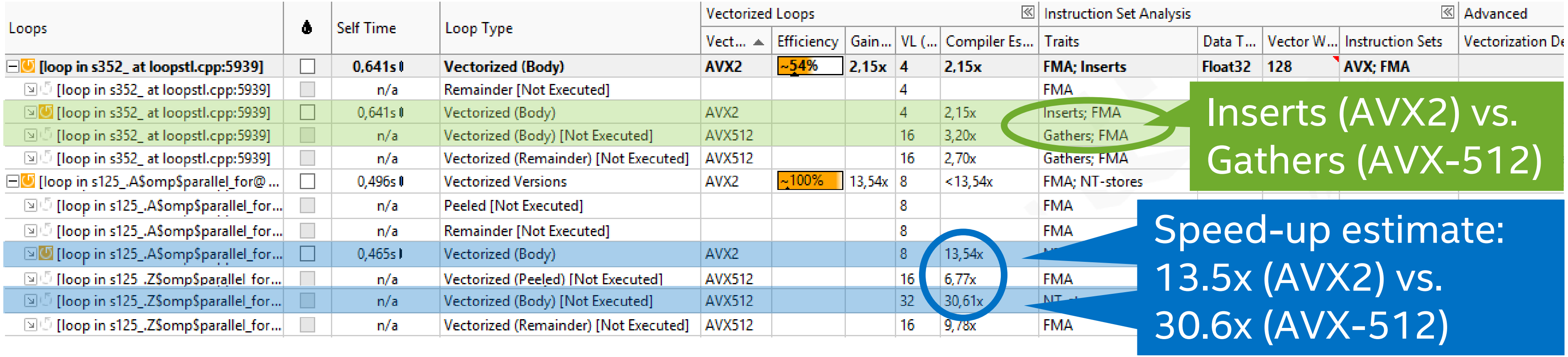
Start Tuning for AVX-512 without AVX-512 hardware

Intel® Advisor - Vectorization Advisor “axcode feature”

Use –axCOMMON-AVX512 –xAVX compiler flags to generate both code-paths

- AVX(2) code path (executed on Haswell and earlier processors)
- AVX-512 code path for newer hardware

Compare AVX and AVX-512 code characteristics with Intel Advisor



Loops	💧	Self Time	Loop Type	Vectorized Loops					Instruction Set Analysis				Advanced
				Vect... ▲	Efficiency	Gain...	VL (...	Compiler Es...	Traits	Data T...	Vector W...	Instruction Sets	Vectorization De
⊟ **[loop in s352_ at loopstl.cpp:5939]**	□	**0,641s**	**Vectorized (Body)**	**AVX2**	**~54%**	**2,15x**	**4**	**2,15x**	**FMA; Inserts**	**Float32**	**128**	**AVX; FMA**	
[loop in s352_ at loopstl.cpp:5939]	□	n/a	Remainder [Not Executed]				4		FMA				
[loop in s352_ at loopstl.cpp:5939]	□	0,641s	Vectorized (Body)	AVX2			4	2,15x	Inserts; FMA				
[loop in s352_ at loopstl.cpp:5939]	□	n/a	Vectorized (Body) [Not Executed]	AVX512			16	3,20x	Gathers; FMA				
[loop in s352_ at loopstl.cpp:5939]	□	n/a	Vectorized (Remainder) [Not Executed]	AVX512			16	2,70x	Gathers; FMA				
⊟ [loop in s125_.A$omp$parallel_for@ ...	□	0,496s	Vectorized Versions	AVX2	~100%	13,54x	8	<13,54x	FMA; NT-stores				
[loop in s125_.A$omp$parallel_for...	□	n/a	Peeled [Not Executed]				8		FMA				
[loop in s125_.A$omp$parallel_for...	□	n/a	Remainder [Not Executed]				8		FMA				
[loop in s125_.A$omp$parallel_for...	□	0,465s	Vectorized (Body)	AVX2			8	13,54x	NT				
[loop in s125_.Z$omp$parallel_for...	□	n/a	Vectorized (Peeled) [Not Executed]	AVX512			16	6,77x	FMA				
[loop in s125_.Z$omp$parallel_for...	□	n/a	Vectorized (Body) [Not Executed]	AVX512			32	30,61x	NT-				
[loop in s125_.Z$omp$parallel_for...	□	n/a	Vectorized (Remainder) [Not Executed]	AVX512			16	9,78x	FMA				

Inserts (AVX2) vs. Gathers (AVX-512)

Speed-up estimate: 13.5x (AVX2) vs. 30.6x (AVX-512)

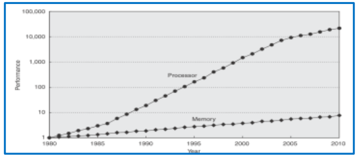
Optimization Notice

ROOFLINE PERFORMANCE MODEL *AUTOMATION*

From “Old HPC principle” to modern performance model

“Old” HPC principles:

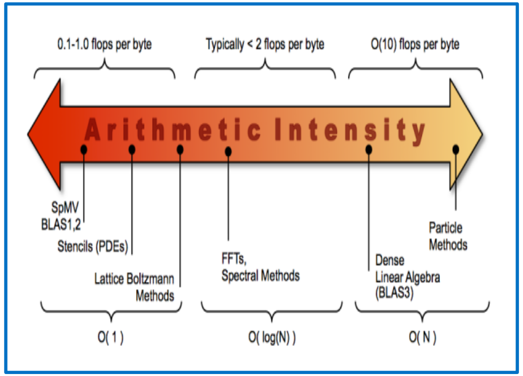
1. “Balance” principle (e.g. Kung 1986) – hw and software parameters altogether
2. “intensity”, “machine balance” - (FLOP/byte or Byte/FLOP ratio for algorithm or hardware). E.g. Kennedy, Carr: 1988, 1994: “Improving the Ratio of Memory operations to Floating-Point Operations in Loops “.

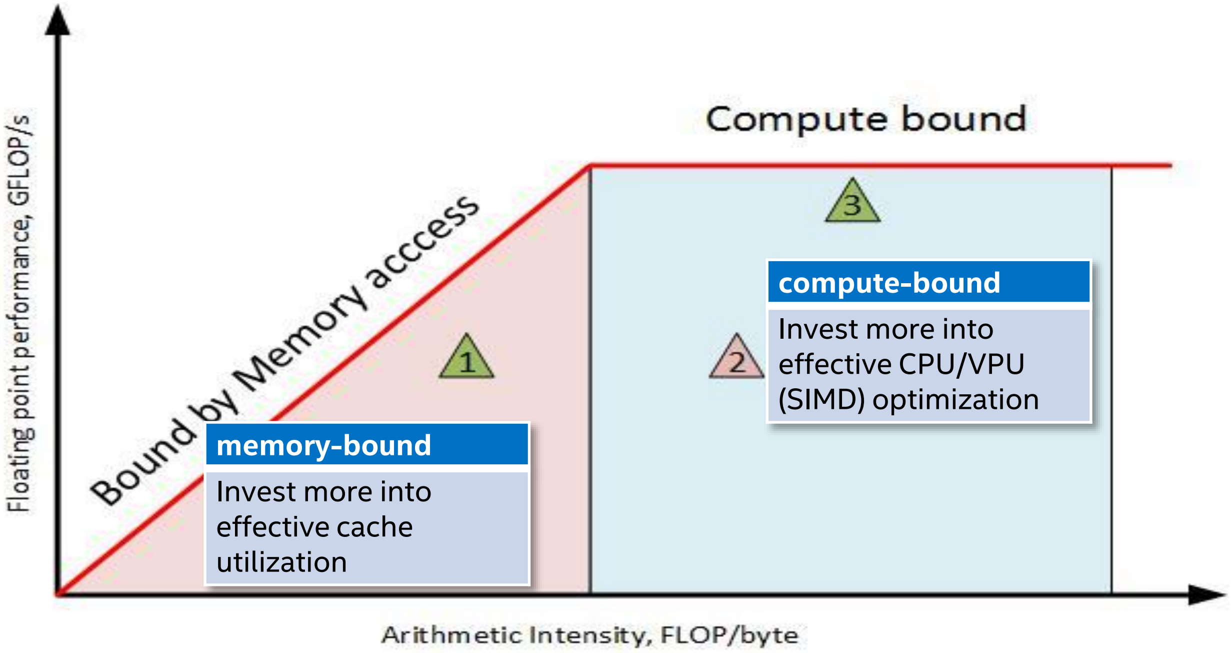


More research catalyzed by memory wall

- 2008, Berkeley: generalized into Roofline Performance Model. Williams, Waterman, Patterson. “Roofline: an insightful visual performance model for multicore”
- 2014: “Cache-aware Roofline model: ” Ilic, Pratas, Sousa. INESC-ID/IST, Technical Uni of Lisbon.

Roofline Performance Model





compute-bound

Invest more into effective CPU/VPU (SIMD) optimization

memory-bound

Invest more into effective cache utilization

Density, Intensity, Machine balance

Arithmetic Intensity = Total Flops computed / *Total Bytes transferred*

OI

~~Arithmetic~~ Operational Intensity = Total Flops computed / Total Bytes transferred between **DRAM (MCDRAM) and LLC**

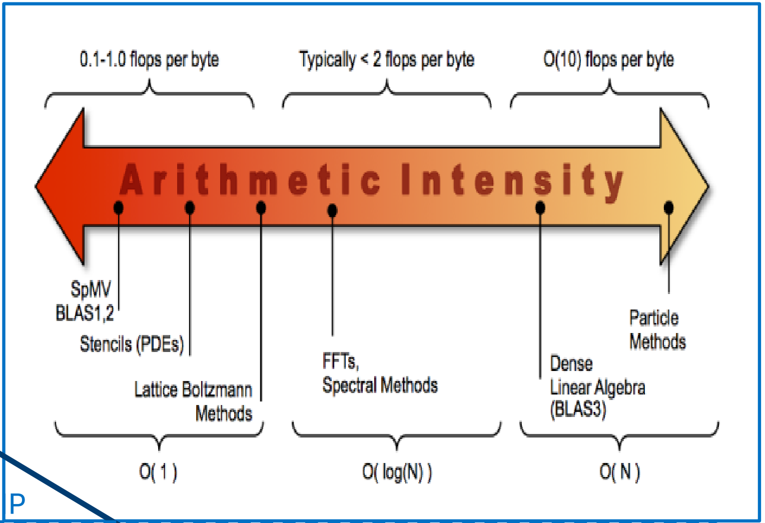
Implemented in 2017 Update 1

AI

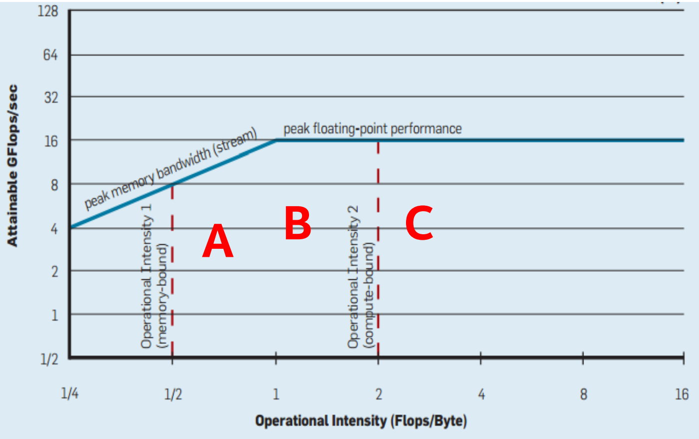
Arithmetic Intensity = Total Flops computed / Total Bytes transferred between **CPU and "memory"**

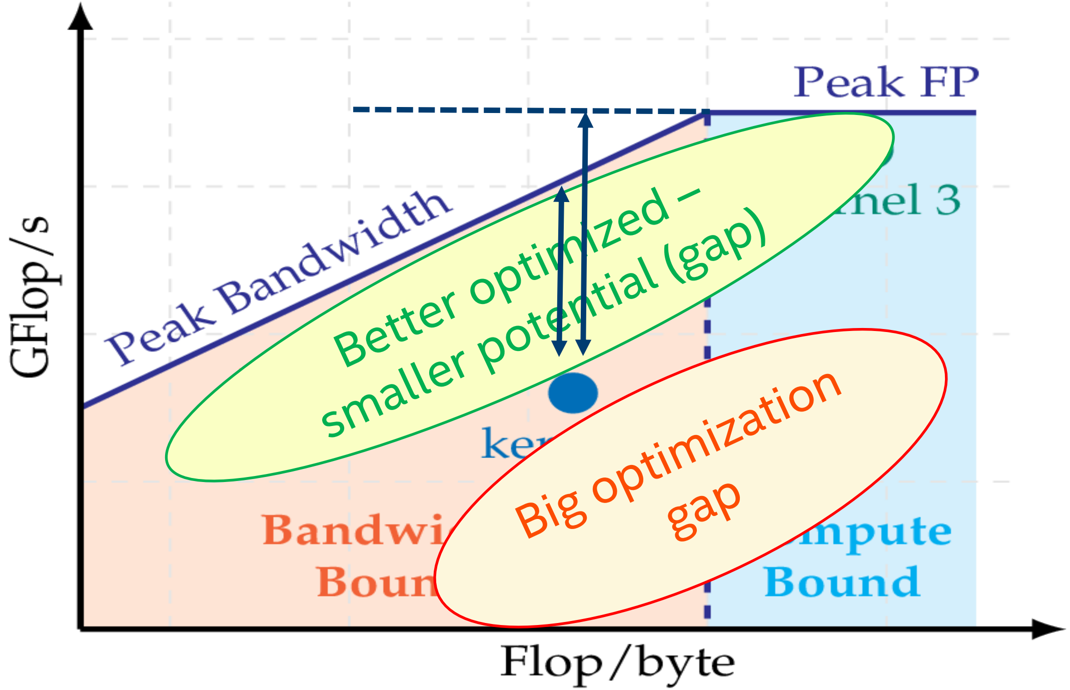
WIP

~~Arithmetic~~ Intensity = Total **Intops+Flops** computed / *Total Bytes transferred* between CPU and "memory"



Roofline model: **Am I bound by VPU/CPU or by Memory?**



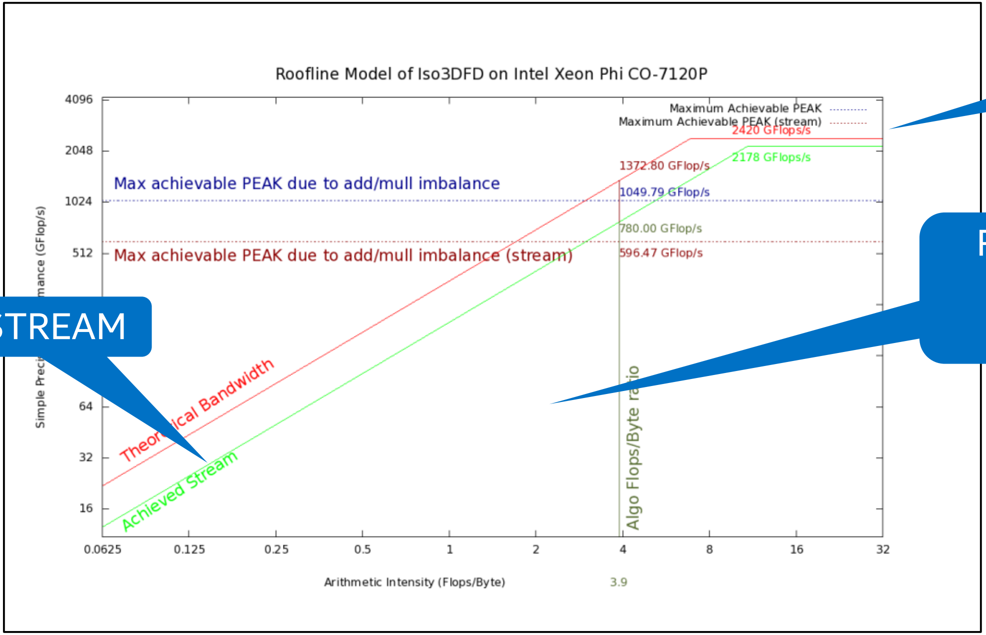


Better optimized – smaller potential (gap)

Big optimization gap

What makes loops **A, B, C** different?

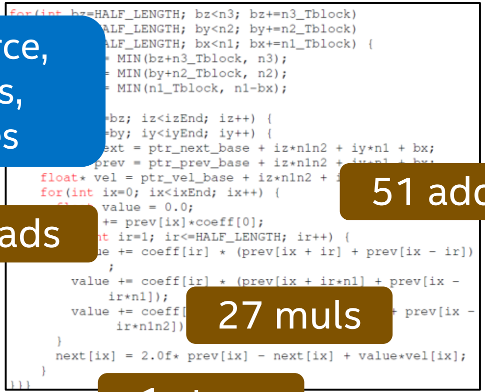
Old approach – pen and paper



Run DGEMM

Run STREAM

Read the source, count FP ops, loads&stores



```
for(int bz=HALF_LENGTH; bz<n3; bz+=n3_Tblock)
           ALF_LENGTH; by<n2; by+=n2_Tblock)
           ALF_LENGTH; bx<n1; bx+=n1_Tblock) {
          = MIN(bz+n3_Tblock, n3);
          = MIN(by+n2_Tblock, n2);
          = MIN(n1_Tblock, n1-bx);

          =bz; iz<izEnd; iz++) {
          =by; iy<iyEnd; iy++) {
          ext = ptr_next_base + iz*n1n2 + iy*n1 + bx;
         prev = ptr_prev_base + iz*n1n2 +
    float* vel = ptr_vel_base + iz*n1n2 + i
    for(int ix=0; ix<ixEnd; ix++) {
           value = 0.0;
           += prev[ix]*coeff[0];
          nt ir=1; ir<=HALF_LENGTH; ir++) {
          ue += coeff[ir] * (prev[ix + ir] + prev[ix - ir])
            ;
        value += coeff[ir] * (prev[ix + ir*n1] + prev[ix -
            ir*n1]);
        value += coeff[                          + prev[ix -
             ir*n1n2])
      }
      next[ix] = 2.0f* prev[ix] - next[ix] + value*vel[ix];
    }
}}}
```

4 loads

51 adds

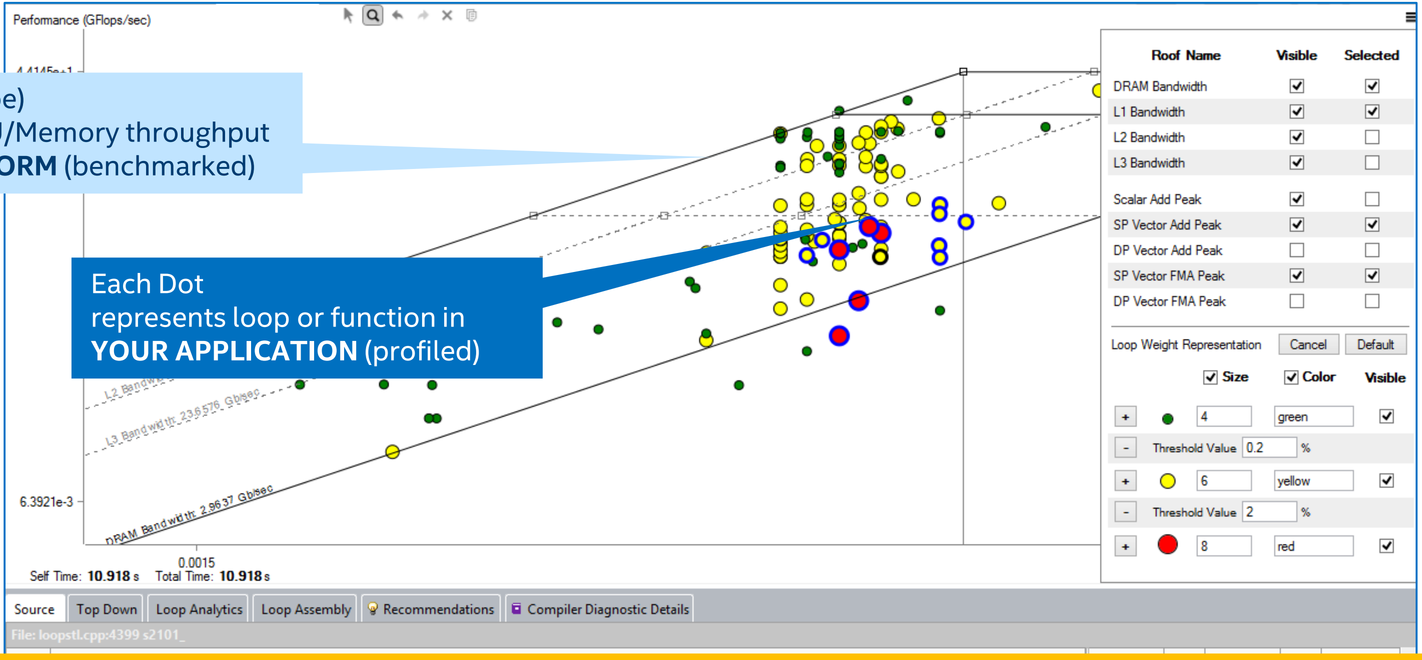
27 muls

1 store

“3D stencil performance evaluation and auto-tuning on multi and many-core computers”, C.Andreolli et.al.

Cumbersome – but people still did it!

Roofline Automation in Intel® Advisor 2017



Self Time: **10.918** s Total Time: **10.918** s

Roof Name	Visible	Selected
DRAM Bandwidth	☑	☑
L1 Bandwidth	☑	☑
L2 Bandwidth	☑	☐
L3 Bandwidth	☑	☐
Scalar Add Peak	☑	☐
SP Vector Add Peak	☑	☑
DP Vector Add Peak	☐	☐
SP Vector FMA Peak	☑	☑
DP Vector FMA Peak	☐	☐

Loop Weight Representation Cancel Default

		☑ Size	☑ Color	Visible
+	●	4	green	☑
-	Threshold Value	0.2	%	
+	●	6	yellow	☑
-	Threshold Value	2	%	
+	●	8	red	☑

File: loopstl.cpp:4399 s2101_

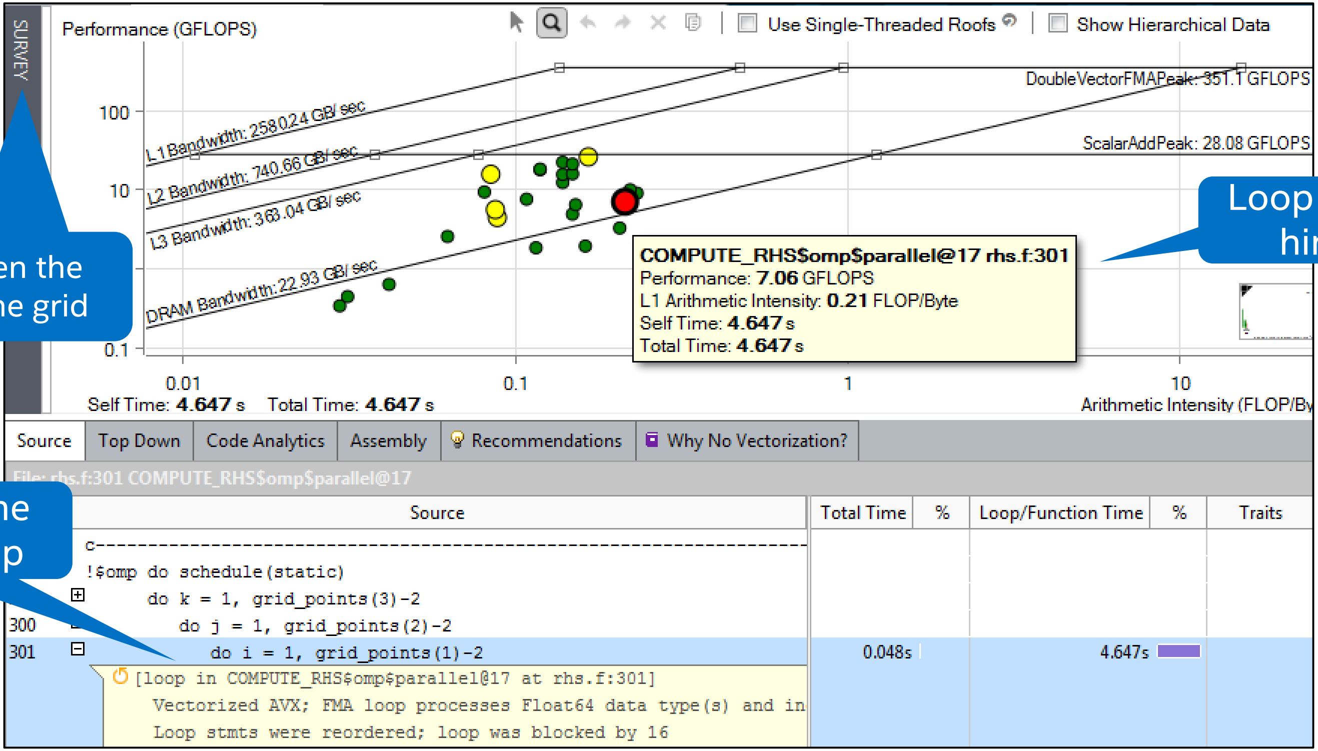
Each Roof (slope)
Gives peak CPU/Memory throughput of your **PLATFORM** (benchmarked)

Each Dot
represents loop or function in **YOUR APPLICATION** (profiled)

Automatic and integrated – first class citizen in Intel® Advisor

Optimization Notice

Roofline in Intel® Advisor


SURVEY
Performance (GFLOPS)
□ Use Single-Threaded Roofs
□ Show Hierarchical Data
DoubleVectorFMAPeak: 351.1 GFLOPS
ScalarAddPeak: 28.08 GFLOPS
L1 Bandwidth: 2580.24 GB/ sec
L2 Bandwidth: 740.66 GB/ sec
L3 Bandwidth: 363.04 GB/ sec
DRAM Bandwidth: 22.93 GB/ sec
100
10
0.1
0.01
0.1
1
10
Arithmetic Intensity (FLOP/By
Self Time: 4.647 s
Total Time: 4.647 s
Source
Top Down
Code Analytics
Assembly
Recommendations
Why No Vectorization?

COMPUTE_RHS$omp$parallel@17 rhs.f:301
Performance: **7.06** GFLOPS
L1 Arithmetic Intensity: **0.21** FLOP/Byte
Self Time: **4.647** s
Total Time: **4.647** s

File: rhs.f:301 COMPUTE_RHS$omp$parallel@17

```
Source                                                  Total Time   %   Loop/Function Time   %   Traits
c------------------------------------------------------------------------
!$omp do schedule(static)
      do k = 1, grid_points(3)-2
         do j = 1, grid_points(2)-2
            do i = 1, grid_points(1)-2                     0.048s                  4.647s
     [loop in COMPUTE_RHS$omp$parallel@17 at rhs.f:301]
      Vectorized AVX; FMA loop processes Float64 data type(s) and in
      Loop stmts were reordered; loop was blocked by 16
```

Switch between the roofline and the grid

Loop data hint

Source for the selected loop

Automatic and integrated – first class citizen in Intel® Advisor

Optimization Notice

Find Effective Optimization Strategies

Intel Advisor: Cache-aware roofline analysis

Roofs Show Platform Limits

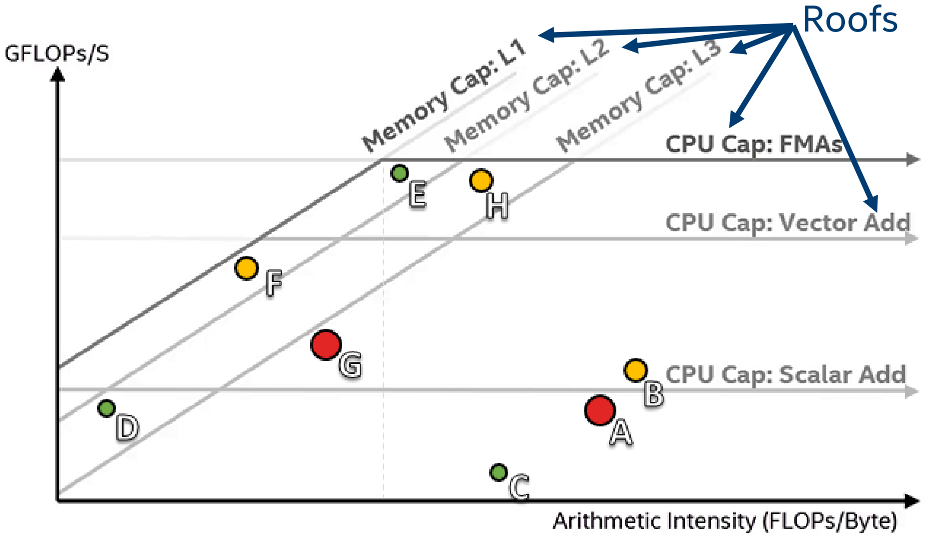
- Memory, cache & compute limits

Dots Are Loops

- Bigger, red dots take more time so optimization has a bigger impact
- Dots farther from a roof have more room for improvement

Higher Dot = Higher GFLOPs/sec

- Optimization moves dots up
- Algorithmic changes move dots horizontally



Which loops should we optimize?

- A and G have the biggest impact & biggest gap
- B has room to improve, but will have less impact
- E and H are perfectly optimized already

Roofline tutorial video

Optimization Notice

Advisor Roofline: under the hood

Roofline application profile:

Axis Y: **FLOP/S** = **#FLOP** (mask aware) / **#Seconds**

Axis X: **AI** = **#FLOP** / **#Bytes**

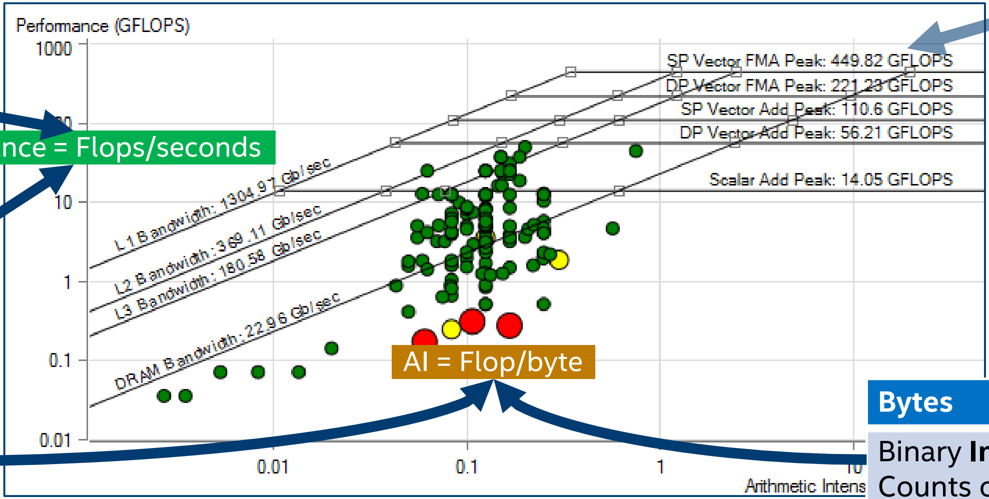
Seconds

User-mode **sampling**

Root access not needed

Roofs

Microbenchmarks
Actual peak for the current configuration



Performance = Flops/seconds

AI = Flop/byte

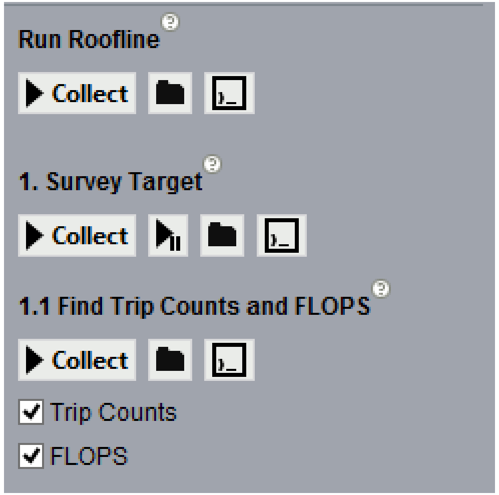
#FLOP

Binary **Instrumentation**
Does not rely on CPU counters

Bytes

Binary **Instrumentation**
Counts operands size (not cachelines)

Getting Roofline data in Intel® Advisor



Run Roofline

1. Survey Target

1.1 Find Trip Counts and FLOPS

☑ Trip Counts

☑ FLOPS

FLOP/S **= #FLOP/Seconds**	**Seconds**	**#FLOP** **- Mask Utilization** **- #Bytes**
Step 1: Survey - Non intrusive. *Representative* - Output: Seconds (+much more)	✓	
Step 2: Trip counts+FLOPS - Precise, instrumentation based - Physically count Num-Instructions - Output: #FLOP, #Bytes		✓

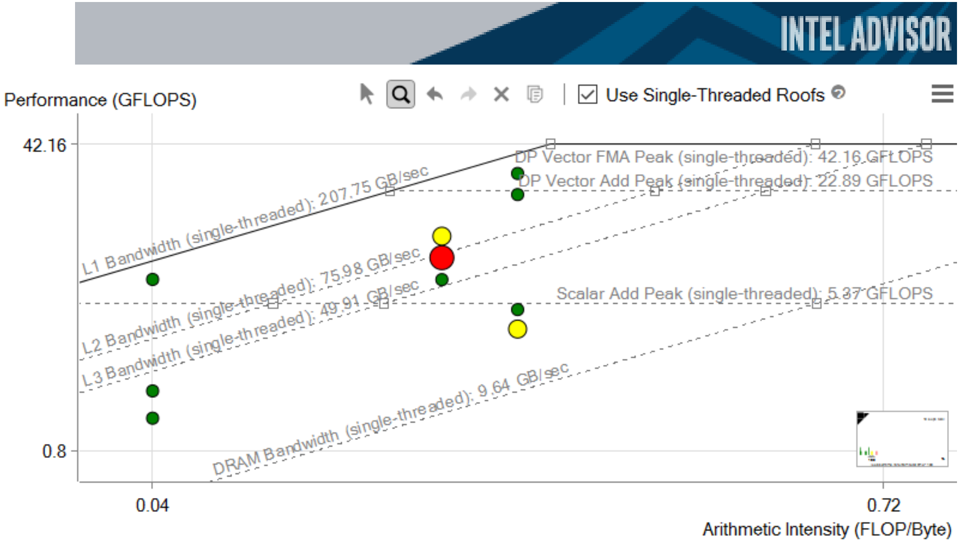
ROOFLINE PERFORMANCE MODEL: *FLAVORS AND INTERPRETATION*

Find Effective Optimization Strategies

Intel Advisor: Cache-aware roofline analysis

Roofline Performance Insights

- Highlights poor performing loops
- Shows performance "headroom" for each loop
  - Which can be improved
  - Which are worth improving
- Shows likely causes of bottlenecks
- Suggests next optimization steps



☑ Use Single-Threaded Roofs

INTEL ADVISOR
Performance (GFLOPS)
42.16
0.8
0.04
0.72
Arithmetic Intensity (FLOP/Byte)
L1 Bandwidth (single-threaded): 207.75 GB/sec
L2 Bandwidth (single-threaded): 75.98 GB/sec
L3 Bandwidth (single-threaded): 49.91 GB/sec
DRAM Bandwidth (single-threaded): 9.64 GB/sec
DP Vector FMA Peak (single-threaded): 42.16 GFLOPS
DP Vector Add Peak (single-threaded): 22.89 GFLOPS
Scalar Add Peak (single-threaded): 5.37 GFLOPS

Classical Roofline vs Cache-Aware Roofline

Classical Roofline Model

AI = # FLOPS / BYTES (DRAM →)

Bytes out of a level in memory hierarchy are measured in AI

AI depends on problem size

AI is platform dependent

AI depends on cache reuse

Cache-Aware Roofline Model

AI = # FLOPS / # BYTES (→ CPU)

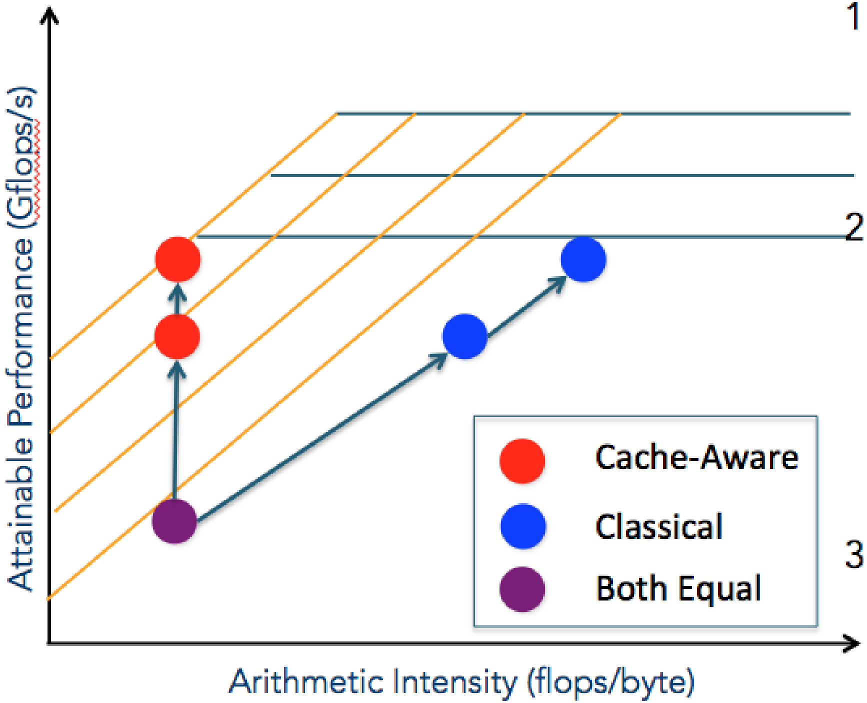
Bytes into the cpu from all levels in memory hierarchy are measured in AI

AI is independent of problem size

AI is independent of platform

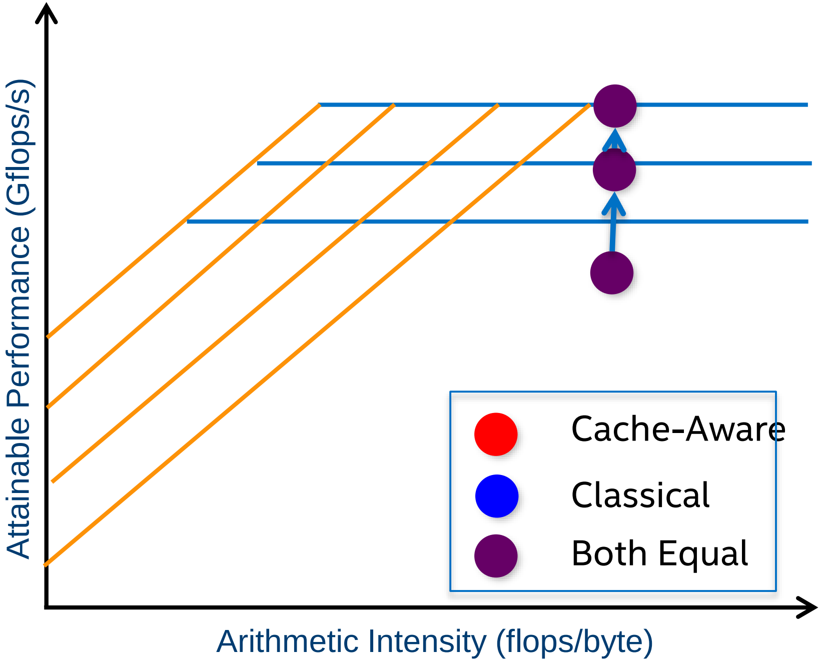
AI is constant for an algorithm

CARM vs. ORM Roofline flavors


Attainable Performance (Gflops/s)
Arithmetic Intensity (flops/byte)
Cache-Aware
Classical
Both Equal

1. Low Ai, "Stream-like" application. Assume it's well vectorized
   - No cache reuse
     → DRAM bandwidth bound
     → DRAM AI = L1 AI
2. Implement L2 cache optimization
   - L2 Cache is fully reused, GFLOPS increase
     → C-A roofline rises up to the L2 bandwidth limit
     → CI roofline moves to the right because we are doing less loads from DRAM.
3. Implement L1 cache optimization
   - See 2.
     → By chance, CI roofline seems bound by the scalar add peak

Example 2: Compute Bound Application

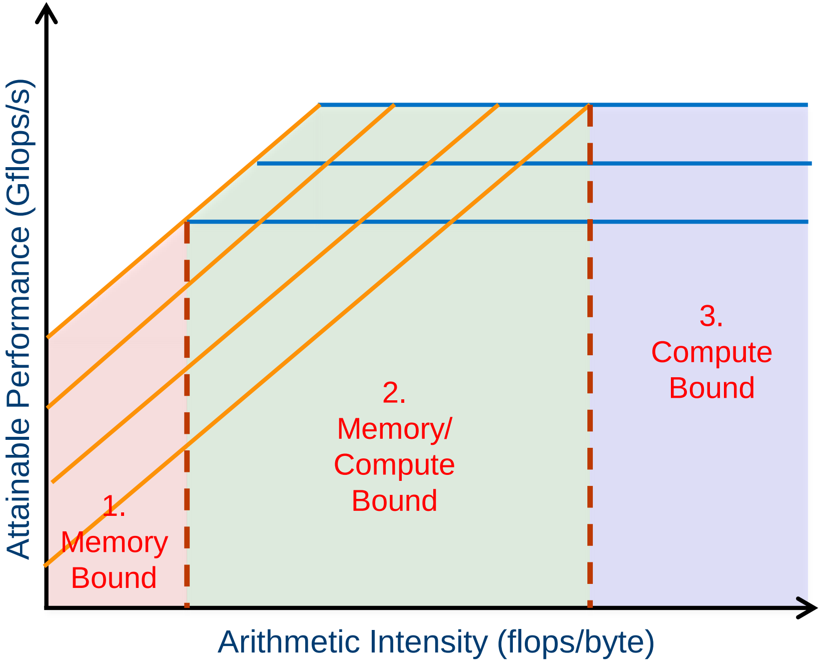


Attainable Performance (Gflops/s)
Arithmetic Intensity (flops/byte)
Cache-Aware
Classical
Both Equal

1. High AI “particle - like” application.
   - No cache reuse again
   - Compute bound but not using vectorization/FMA/both VPUs
2. Implement vectorization
   - Since we are not touching memory, the AI in both C-A and CI roofline does not change
   - We are fully utilizing VPUs → FLOPS increases
3. Implement FMA use

[1] S. Williams et al. *CACM* (2009), crd.lbl.gov/departments/computer-science/PAR/research/roofline

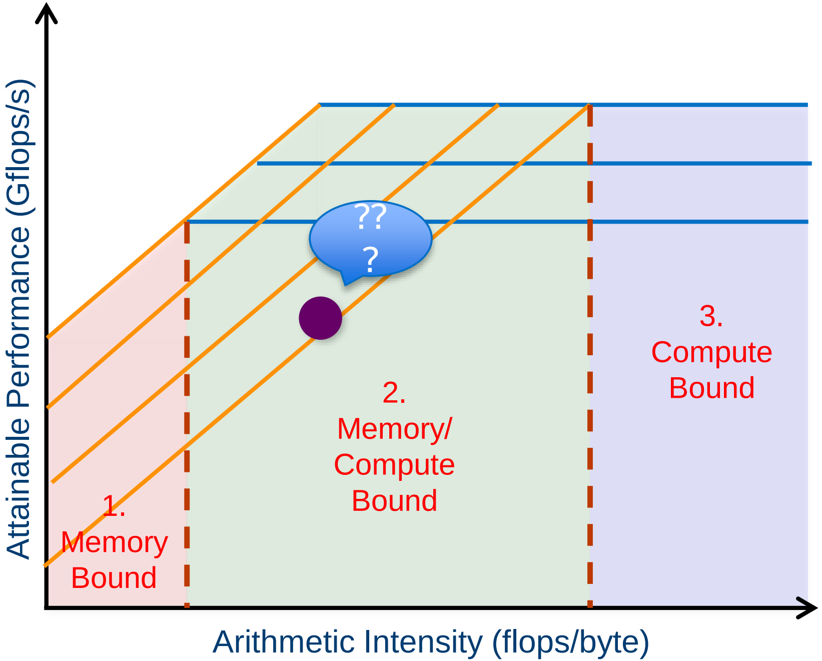
Is My Application Bound by a Memory Bandwidth or a Compute Peak?



Often it’s a combination of the two

- Applications in area 1 are purely memory bandwidth bound
- Applications in area 3 are purely compute bound
- In area 2 we need more information

Ask Yourself "Why am I Here?" and "Where am I going?"


Attainable Performance (Gflops/s)
??
?
1.
Memory
Bound
2.
Memory/
Compute
Bound
3.
Compute
Bound
Arithmetic Intensity (flops/byte)

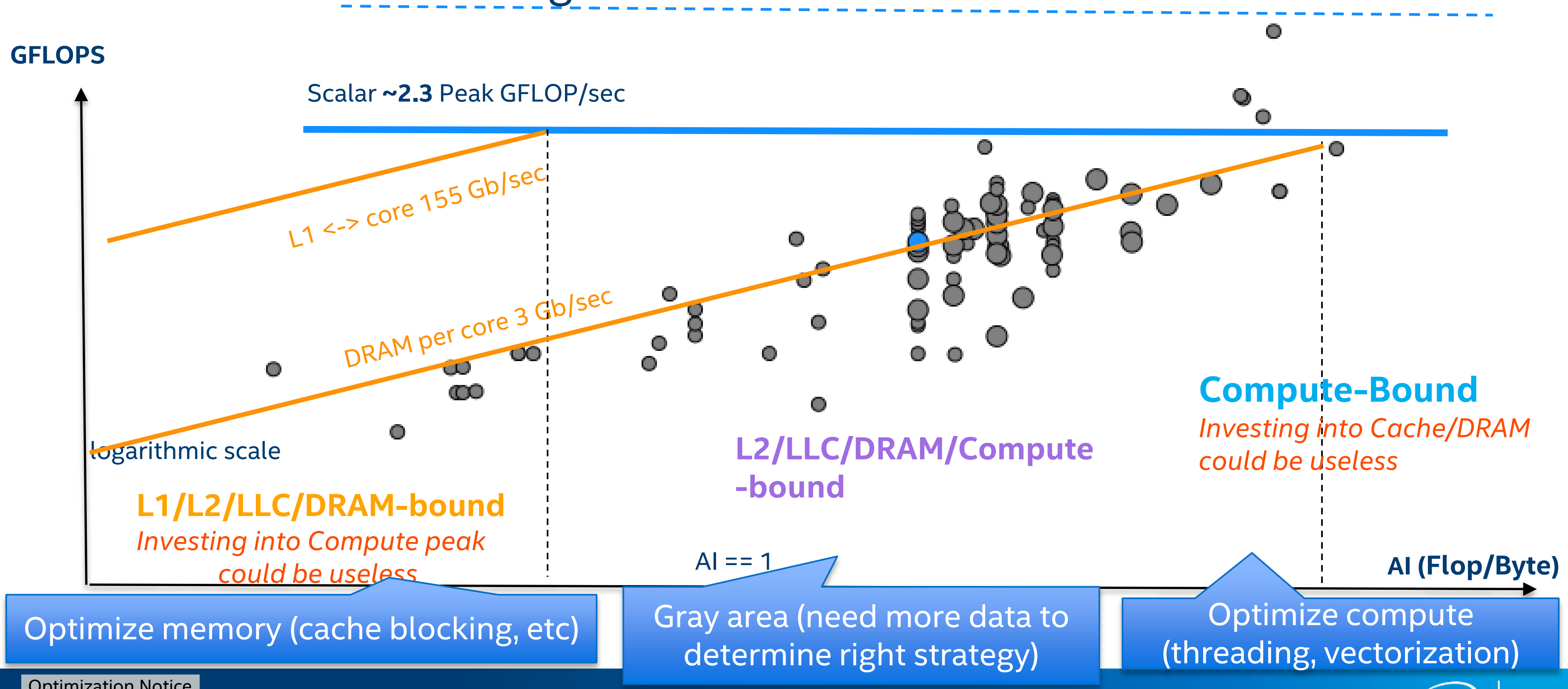
Usually, it is more complicated…

You won't be on any ceiling. Or if you are, it is kind of coincidence.

BUT - asking the questions
"why am I not on a higher ceiling?"
and "what should I do to reach it?"
is always productive.

Perform the right optimization for your region

Roofline: characterization regions



L1/L2/LLC/DRAM-bound

Investing into Compute peak could be useless

L2/LLC/DRAM/Compute -bound

Compute-Bound

Investing into Cache/DRAM could be useless

Optimize memory (cache blocking, etc)

Gray area (need more data to determine right strategy)

Optimize compute (threading, vectorization)

Optimization Notice

Interpreting Roofline Data

Final Limits

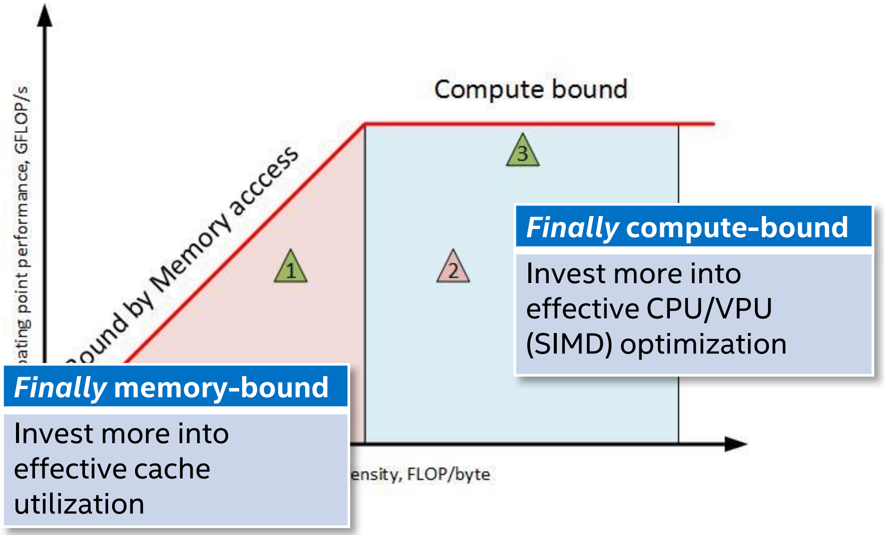
(assuming perfect optimization)

Long-term ROI, optimization strategy

Current Limits

(what are my current bottlenecks)

Next step, optimization tactics



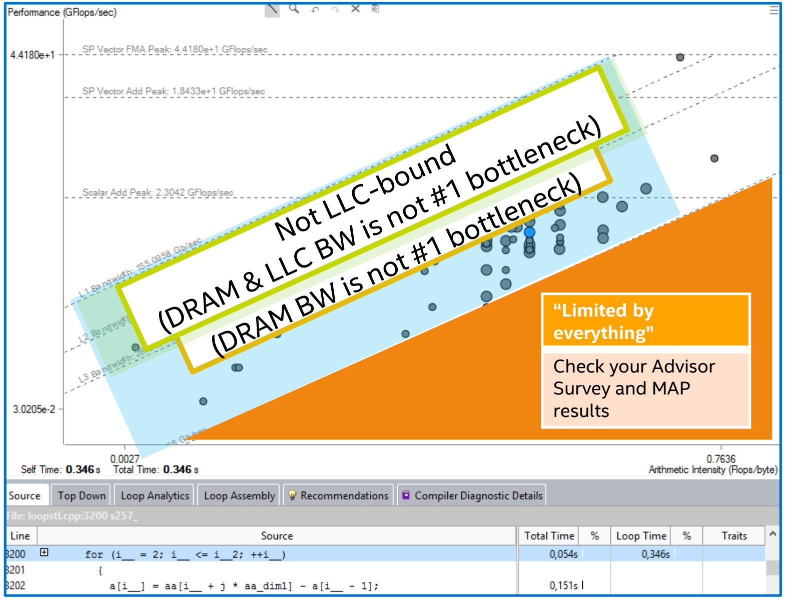
Compute bound
Bound by Memory access
1
2
3
Floating point performance, GFLOP/s
Arithmetic intensity, FLOP/byte

Finally **compute-bound**

Invest more into effective CPU/VPU (SIMD) optimization

Finally **memory-bound**

Invest more into effective cache utilization



Performance (GFlops/sec)
4.4180e+1
SP Vector FMA Peak: 4.4180e+1 GFlops/sec
SP Vector Add Peak: 1.8433e+1 GFlops/sec
Scalar Add Peak: 2.3042 GFlops/sec
L1 Bandwidth: 155.0958 Gb/sec
L2 Bandwidth
L3 Bandwidth
3.0205e-2
0.0027
0.7636
Arithmetic Intensity (Flops/byte)
Source
Top Down
Loop Analytics
Loop Assembly
Recommendations
Compiler Diagnostic Details

Not LLC-bound

(DRAM & LLC BW is not #1 bottleneck)

(DRAM BW is not #1 bottleneck)

"Limited by everything"

Check your Advisor Survey and MAP results

Self Time: **0.346** s Total Time: **0.346** s

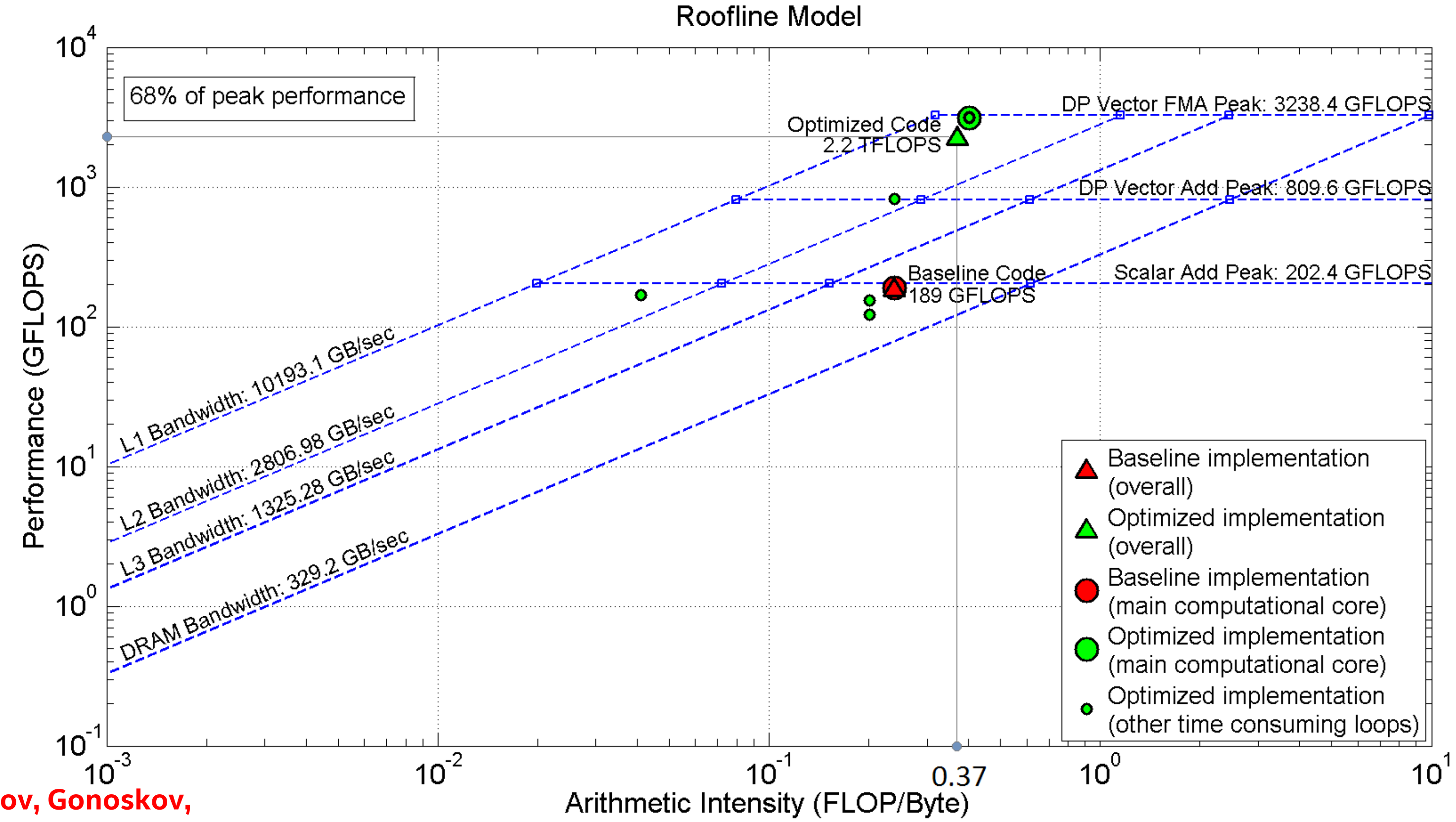
File: loopstl.cpp:3200 s257_

Line	Source	Total Time	%	Loop Time	%	Traits
3200	`for (i__ = 2; i__ <= i__2; ++i__)`	0,054s		0,346s		
3201	`{`					
3202	`a[i__] = aa[i__ + j * aa_dim1] - a[i__ - 1];`	0,151s				

Optimization Notice

ROOFLINE PERFORMANCE MODEL: *CASE STUDIES*

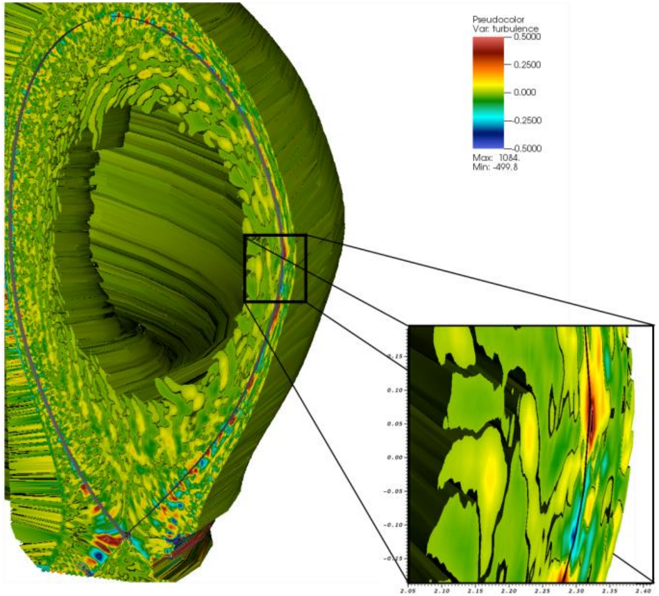
PIC (PICADOR) plasma simulation use case



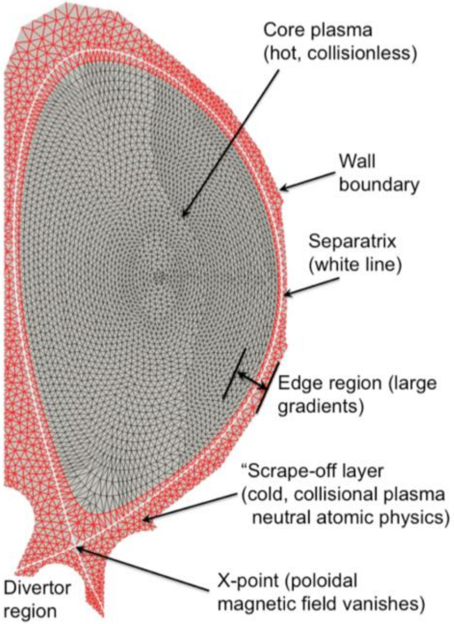
Surmin, Meyerov, Gonoskov,
NN State University & Institute of Applied Physics, Nizhny Novgorod

Optimization Notice

XGC1 is a PIC Code for Tokamak (Edge) Fusion Plasmas (Koskela et all, LBNL, NERSC)



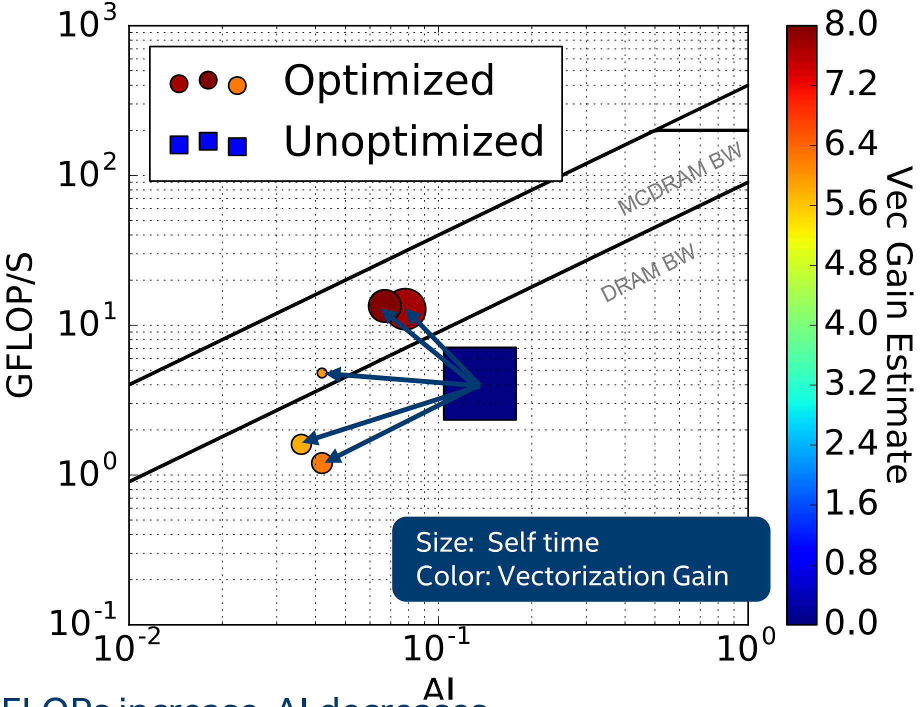
XGC1 Simulation of edge turbulence in the DIII-D tokamak



Unstructured field-aligned mesh in a poloidal domain

Optimization Notice

XGC1: Effect of Optimizations on 1st Order B Interpolation



Size: Self time
Color: Vectorization Gain

- **Single KNL quadcache node 1 rank, 64 threads.**
- **Data collected with Advisor survey + tripcounts**
- **Inner loops over blocks of particles added**
  - Scalar function → vectorized loops
- **Most time-consuming loops above DRAM bandwidth limit**

GFLOPs increase, AI decreases
→ Data alignment should be next optimization target

Total time: 3.5s → 2.1s
Peak GFLOPS: 4.0 → 16.0

Roofline Analysis to Tune an MRI Image Reconstruction Benchmark

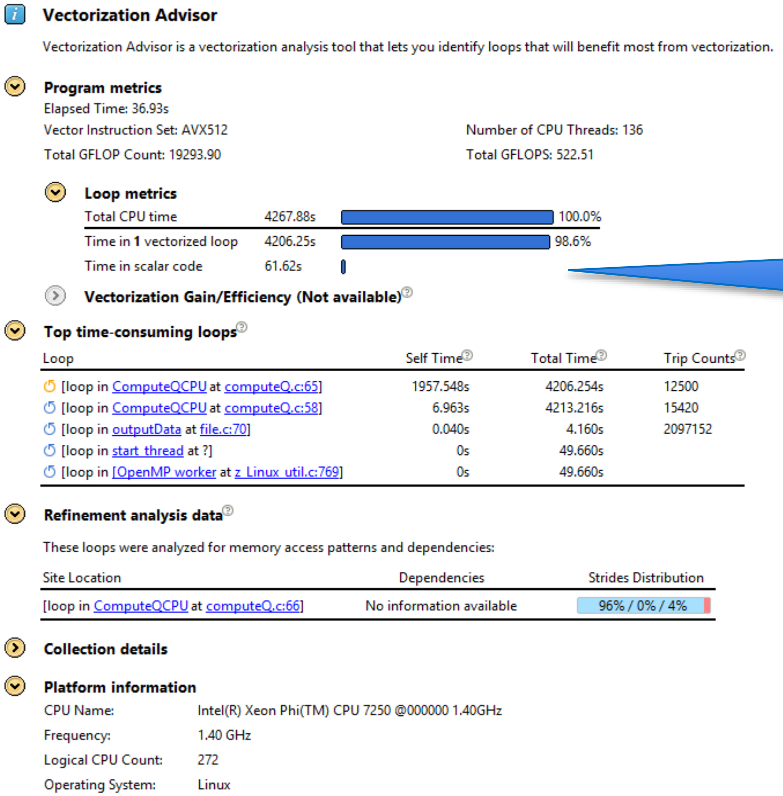
The 514.pomriq SPEC ACCEL Benchmark

An MRI image reconstruction kernel described in Stone et al. (2008). MRI image reconstruction is a conversion from sampled radio responses to magnetic field gradients. The sample coordinates are in the space of magnetic field gradients, or K-space.

The algorithm examines a large set of input, representing the intended MRI scanning trajectory and the points that will be sampled.

The input to 514.pomriq consists of one file containing the number of K-space values, the number of X-space values, and then the list of K-space coordinates, X-space coordinates, and Phi-field complex values for the K-space samples.

Hot loop is vectorized



Vectorization Advisor

Vectorization Advisor is a vectorization analysis tool that lets you identify loops that will benefit most from vectorization.

Program metrics

Elapsed Time: 36.93s

Vector Instruction Set: AVX512 | Number of CPU Threads: 136

Total GFLOP Count: 19293.90 | Total GFLOPS: 522.51

Loop metrics

Total CPU time	4267.88s		100.0%
Time in **1** vectorized loop	4206.25s		98.6%
Time in scalar code	61.62s		

Vectorization Gain/Efficiency (Not available)

Top time-consuming loops

Loop	Self Time	Total Time	Trip Counts
[loop in ComputeQCPU at computeQ.c:65]	1957.548s	4206.254s	12500
[loop in ComputeQCPU at computeQ.c:58]	6.963s	4213.216s	15420
[loop in outputData at file.c:70]	0.040s	4.160s	2097152
[loop in start_thread at ?]	0s	49.660s	
[loop in [OpenMP worker] at z_Linux_util.c:769]	0s	49.660s	

Refinement analysis data

These loops were analyzed for memory access patterns and dependencies:

Site Location	Dependencies	Strides Distribution
[loop in ComputeQCPU at computeQ.c:66]	No information available	96% / 0% / 4%

Collection details

Platform information

CPU Name:	Intel(R) Xeon Phi(TM) CPU 7250 @000000 1.40GHz
Frequency:	1.40 GHz
Logical CPU Count:	272
Operating System:	Linux

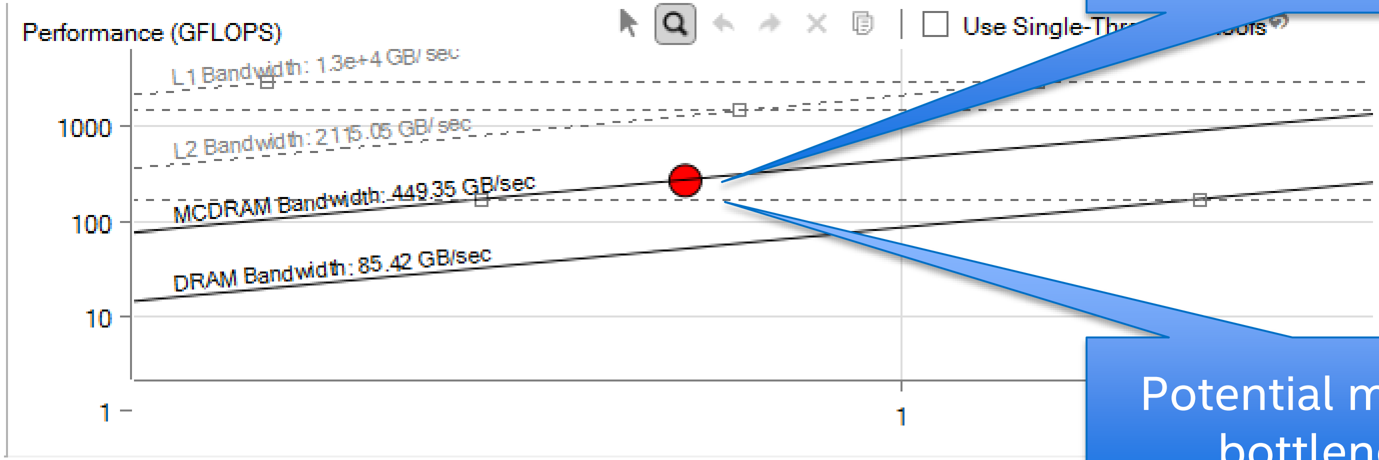
Intel Advisor summary view

1 vectorized loop that we spend 98.8% of our time in

Need more information to see if we can get more performance

What is our performance?

Relative to peak system performance

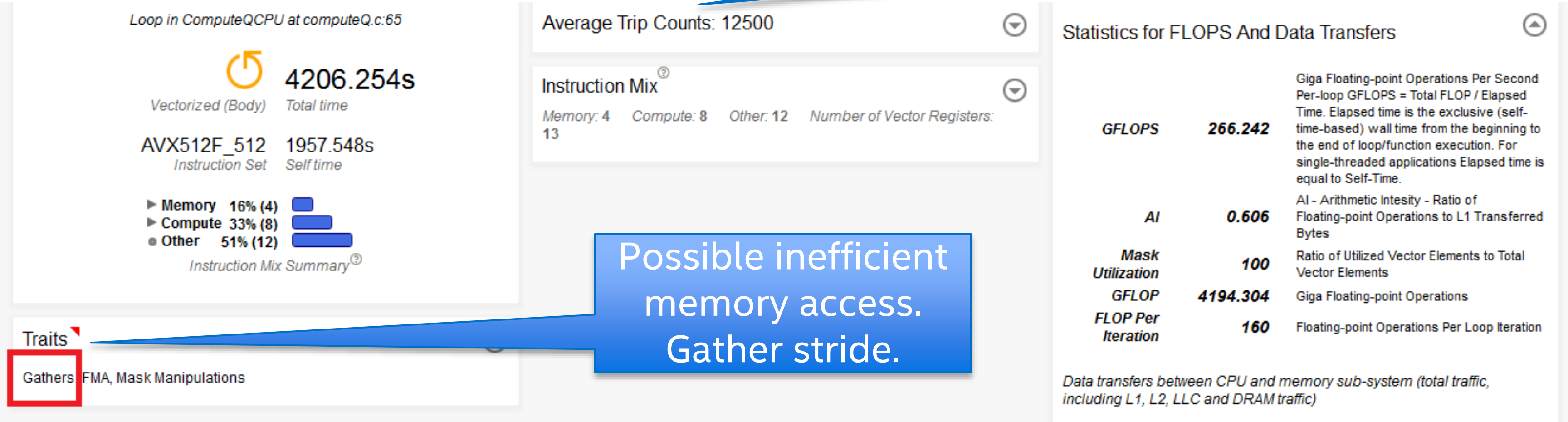


Our hot loop is below the MCDRAM roof

Potential memory bottleneck

Get detailed Advice from intel® Advisor

Intel® Advisor code analytics



Loop in ComputeQCPU at computeQ.c:65

4206.254s *Total time*

Vectorized (Body)

AVX512F_512 *Instruction Set*

1957.548s *Self time*

- ▶ Memory 16% (4)
- ▶ Compute 33% (8)
- ● Other 51% (12)

Instruction Mix Summary

Traits

Gathers FMA, Mask Manipulations

Average Trip Counts: 12500

Instruction Mix

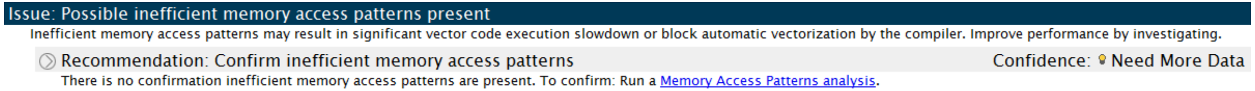
Memory: 4 *Compute:* 8 *Other:* 12 *Number of Vector Registers:* 13

Statistics for FLOPS And Data Transfers

GFLOPS	***266.242***	Giga Floating-point Operations Per Second Per-loop GFLOPS = Total FLOP / Elapsed Time. Elapsed time is the exclusive (self-time-based) wall time from the beginning to the end of loop/function execution. For single-threaded applications Elapsed time is equal to Self-Time.
AI	***0.606***	AI - Arithmetic Intesity - Ratio of Floating-point Operations to L1 Transferred Bytes
Mask Utilization	***100***	Ratio of Utilized Vector Elements to Total Vector Elements
GFLOP	***4194.304***	Giga Floating-point Operations
FLOP Per Iteration	***160***	Floating-point Operations Per Loop Iteration

Data transfers between CPU and memory sub-system (total traffic, including L1, L2, LLC and DRAM traffic)

Possible inefficient memory access. Gather stride.



Issue: Possible inefficient memory access patterns present

Inefficient memory access patterns may result in significant vector code execution slowdown or block automatic vectorization by the compiler. Improve performance by investigating.

Recommendation: Confirm inefficient memory access patterns — Confidence: Need More Data

There is no confirmation inefficient memory access patterns are present. To confirm: Run a Memory Access Patterns analysis.

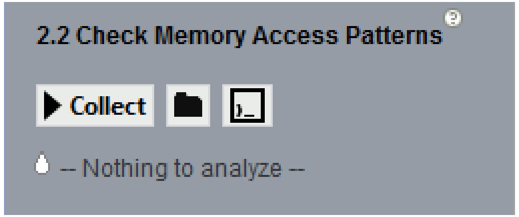
Recommendations – need more information, confirm inefficient memory access

Optimization Notice

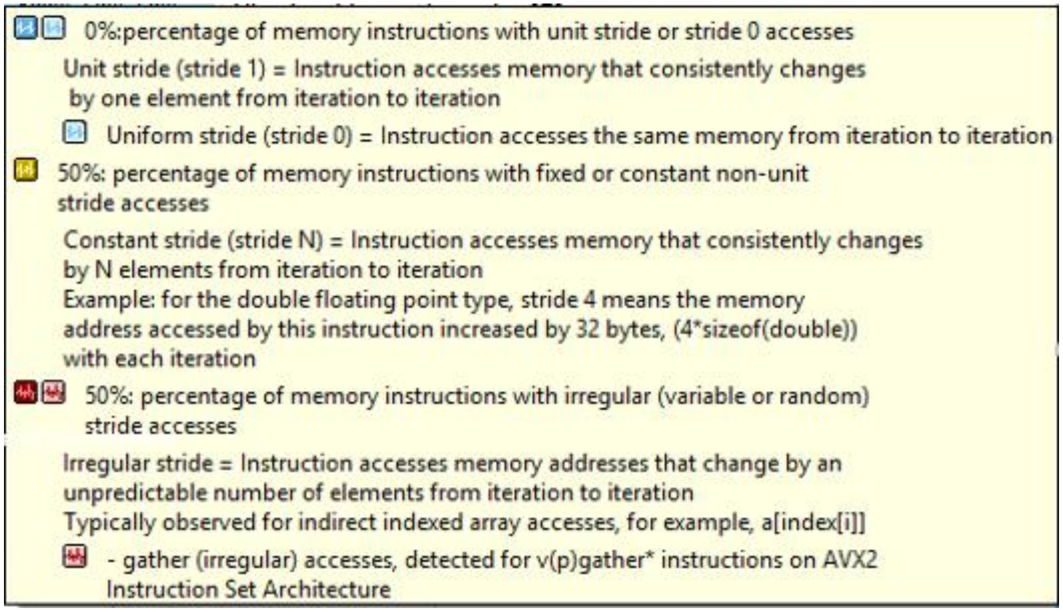
Irregular access patterns decreases performance!

Gather profiling

Run Memory Access Pattern Analysis (MAP)


2.2 Check Memory Access Patterns

-- Nothing to analyze --


0%:percentage of memory instructions with unit stride or stride 0 accesses

Unit stride (stride 1) = Instruction accesses memory that consistently changes by one element from iteration to iteration

Uniform stride (stride 0) = Instruction accesses the same memory from iteration to iteration

50%: percentage of memory instructions with fixed or constant non-unit stride accesses

Constant stride (stride N) = Instruction accesses memory that consistently changes by N elements from iteration to iteration
Example: for the double floating point type, stride 4 means the memory address accessed by this instruction increased by 32 bytes, (4*sizeof(double)) with each iteration

50%: percentage of memory instructions with irregular (variable or random) stride accesses

Irregular stride = Instruction accesses memory addresses that change by an unpredictable number of elements from iteration to iteration
Typically observed for indirect indexed array accesses, for example, a[index[i]]

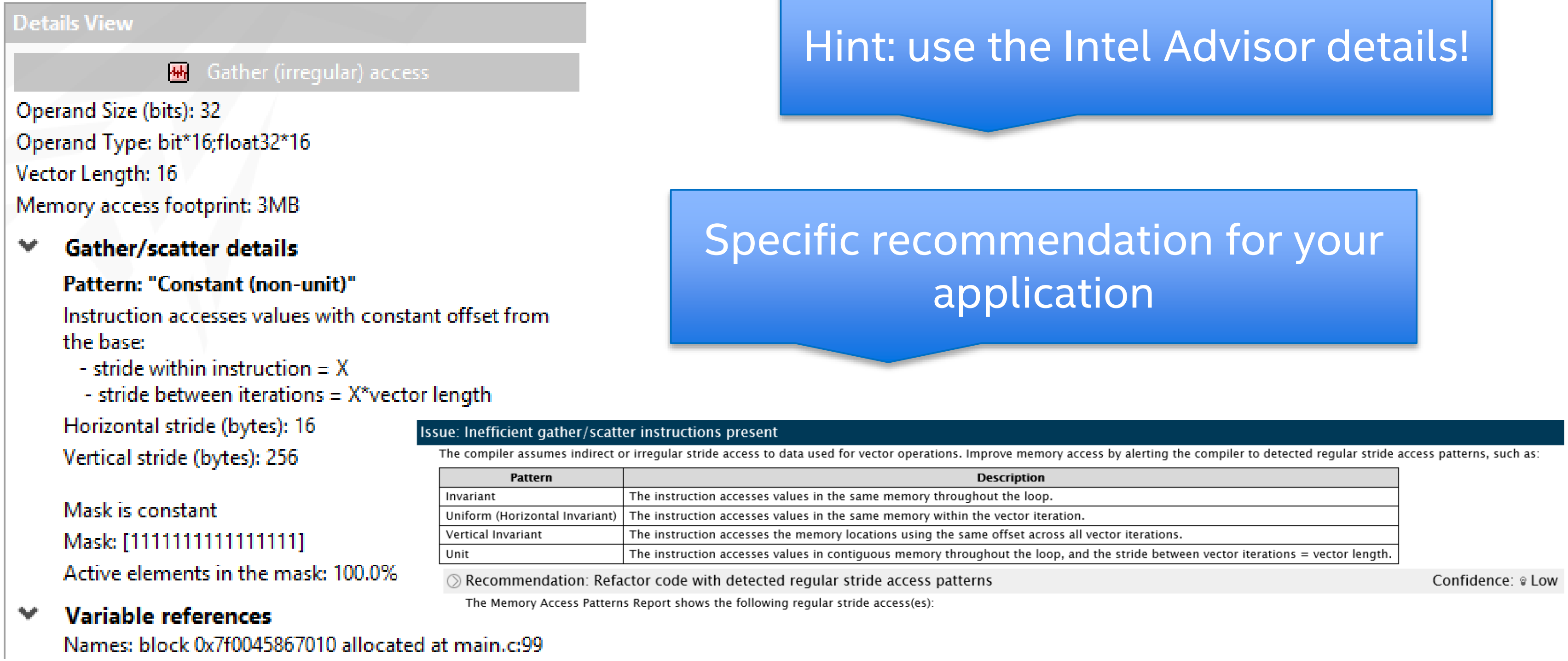
- gather (irregular) accesses, detected for v(p)gather* instructions on AVX2 Instruction Set Architecture

Irregular access patterns

Bad for vectorization performance

Hint: use the Intel Advisor details!

Specific recommendation for your application



Details View

Gather (irregular) access

Operand Size (bits): 32

Operand Type: bit*16;float32*16

Vector Length: 16

Memory access footprint: 3MB

Gather/scatter details

Pattern: "Constant (non-unit)"

Instruction accesses values with constant offset from the base:

- stride within instruction = X
- stride between iterations = X*vector length

Horizontal stride (bytes): 16

Vertical stride (bytes): 256

Mask is constant

Mask: [1111111111111111]

Active elements in the mask: 100.0%

Variable references

Names: block 0x7f0045867010 allocated at main.c:99

Issue: Inefficient gather/scatter instructions present

The compiler assumes indirect or irregular stride access to data used for vector operations. Improve memory access by alerting the compiler to detected regular stride access patterns, such as:

Pattern	Description
Invariant	The instruction accesses values in the same memory throughout the loop.
Uniform (Horizontal Invariant)	The instruction accesses values in the same memory within the vector iteration.
Vertical Invariant	The instruction accesses the memory locations using the same offset across all vector iterations.
Unit	The instruction accesses values in contiguous memory throughout the loop, and the stride between vector iterations = vector length.

Recommendation: Refactor code with detected regular stride access patterns — Confidence: Low

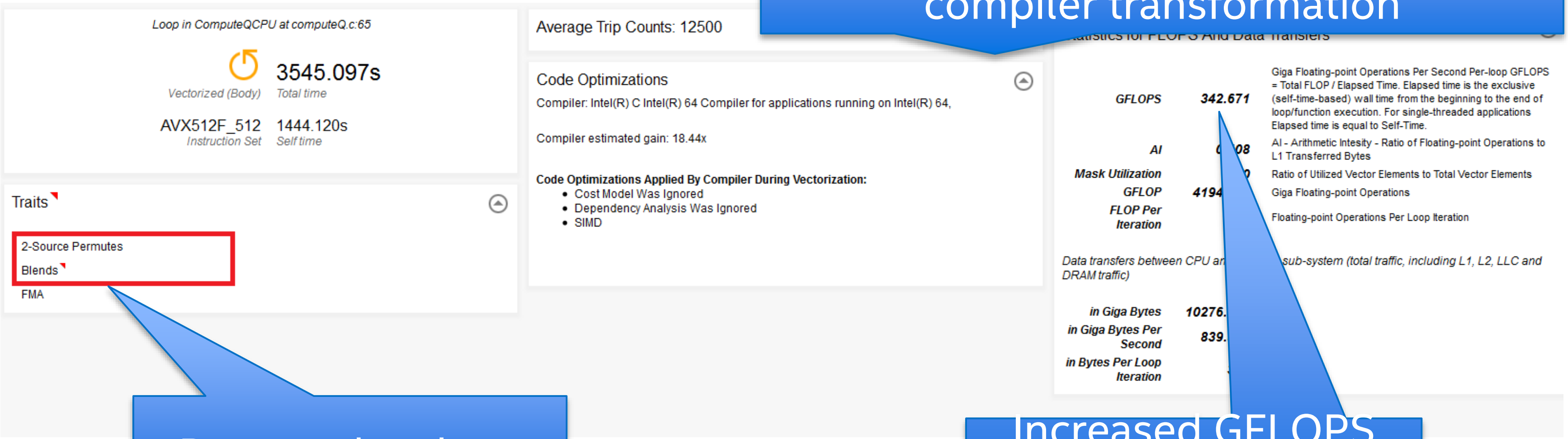
The Memory Access Patterns Report shows the following regular stride access(es):

Optimization Notice

Remove gather instructions

step #1 – use newer version of the intel compiler can recognize the access pattern

Gathers replacement is performed by the “Gather to Shuffle/Permutes” compiler transformation



Loop in ComputeQCPU at computeQ.c:65

3545.097s
Total time

Vectorized (Body)

AVX512F_512
Instruction Set

1444.120s
Self time

Traits

2-Source Permutes
Blends
FMA

Average Trip Counts: 12500

Code Optimizations

Compiler: Intel(R) C Intel(R) 64 Compiler for applications running on Intel(R) 64,

Compiler estimated gain: 18.44x

Code Optimizations Applied By Compiler During Vectorization:

- Cost Model Was Ignored
- Dependency Analysis Was Ignored
- SIMD

atistics for FLOPS And Data Transfers

GFLOPS	342.671	Giga Floating-point Operations Per Second Per-loop GFLOPS = Total FLOP / Elapsed Time. Elapsed time is the exclusive (self-time-based) wall time from the beginning to the end of loop/function execution. For single-threaded applications Elapsed time is equal to Self-Time.
AI	0 08	AI - Arithmetic Intesity - Ratio of Floating-point Operations to L1 Transferred Bytes
Mask Utilization	0	Ratio of Utilized Vector Elements to Total Vector Elements
GFLOP	4194	Giga Floating-point Operations
FLOP Per Iteration		Floating-point Operations Per Loop Iteration

Data transfers between CPU an[illegible] sub-system (total traffic, including L1, L2, LLC and DRAM traffic)

in Giga Bytes	10276.
in Giga Bytes Per Second	839.
in Bytes Per Loop Iteration	

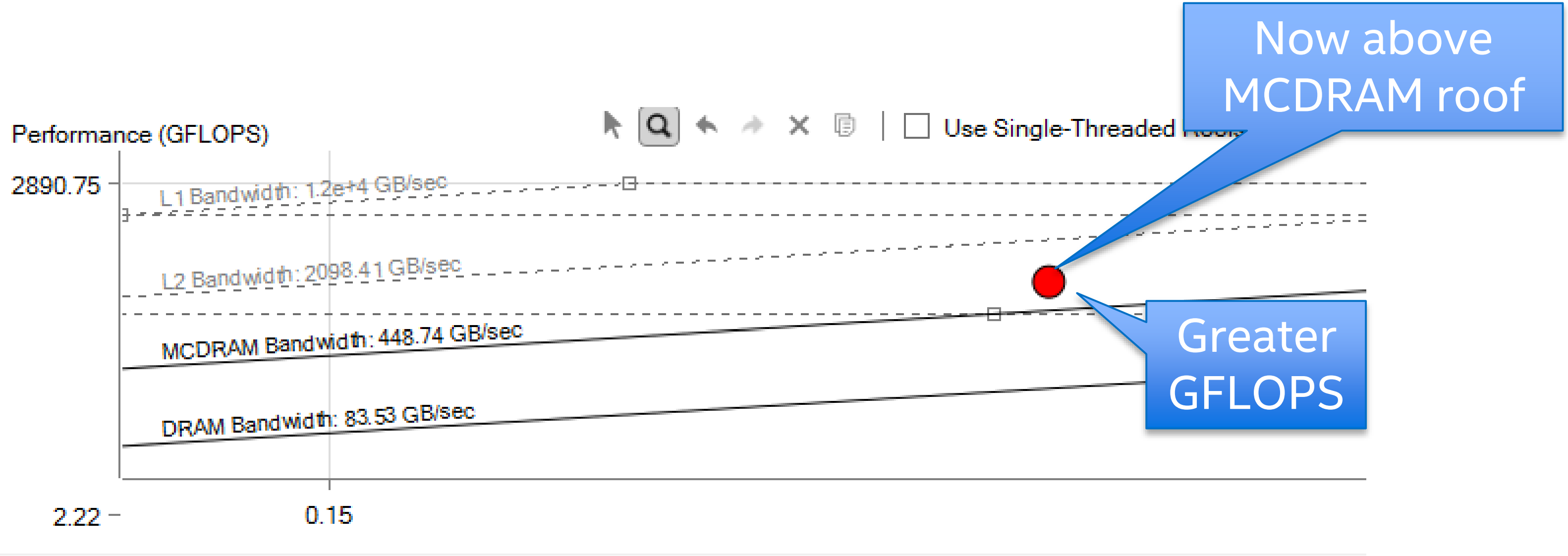
Removed gathers

Increased GFLOPS (from 266.42 to 342.67)

Optimization Notice

Remove gather instructions

step #1 – newer version of the intel compiler can recognize the access pattern



Now above MCDRAM roof

Greater GFLOPS

Remove gather instructions

step #2 - Use structure of arrays instead of array of structures T

```
struct kValues {
  float Kx;
  float Ky;
  float Kz;
  float PhiMag;
};

SDLT_PRIMITIVE(kValues, Kx, Ky, Kz, PhiMag)


sdlt::soa1d_container<kValues> inputKValues(numK);
auto kValues = inputKValues.access();

  for (k = 0; k < numK; k++) {
    kValues [k].Kx() = kx[k];
    kValues [k].Ky() = ky[k];
    kValues [k].Kz() = kz[k];
    kValues [k].PhiMag() = phiMag[k];
  }

auto kVals = inputKValues.const_access();
#pragma omp simd private(expArg, cosArg, sinArg) reduction(+:QrSum, QiSum)
      for (indexK = 0; indexK < numK; indexK++) {
        expArg = PIx2 * (kVals[indexK].Kx() * x[indexX] +
        kVals[indexK].Ky() * y[indexX] +
        kVals[indexK].Kz() * z[indexX]);

        cosArg = cosf(expArg);
        sinArg = sinf(expArg);

        float phi = kVals[indexK].PhiMag();
        QrSum += phi * cosArg;
        QiSum += phi * sinArg;
      }
```

This is a classic vectorization efficiency strategy

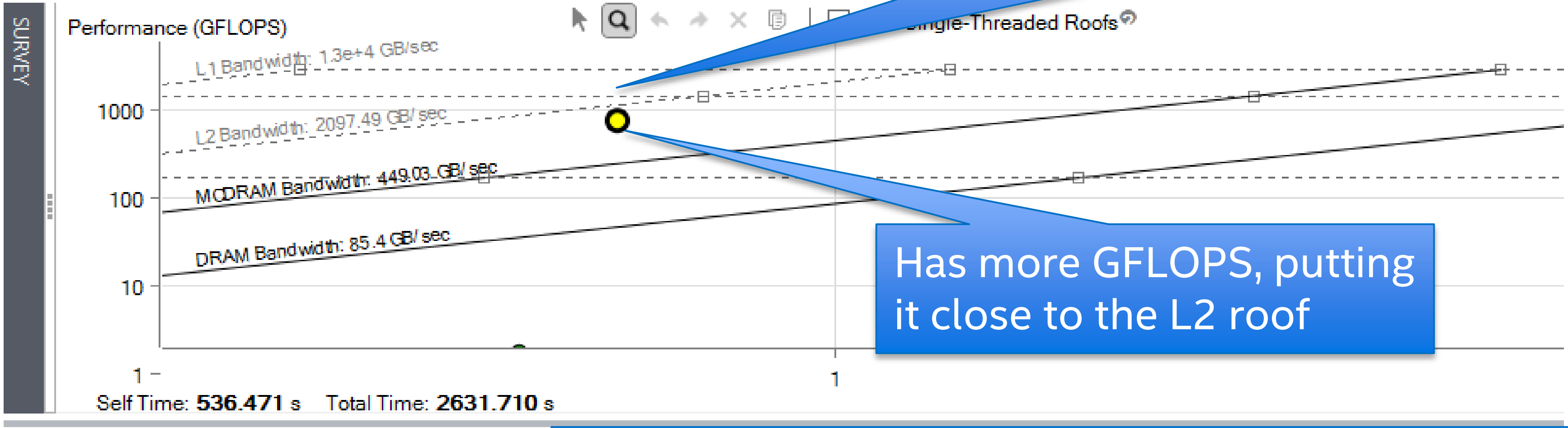
But it can yield poorly designed code

Intel® SIMD Data Layout Templates makes this transformation easy and painless!

Optimization Notice

Remove gather instructions

step #2 - Transform code using the Intel® SIMD Data Layout Templates



SURVEY
Performance (GFLOPS)
Single-Threaded Roofs
L1 Bandwidth: 1.3e+4 GB/sec
L2 Bandwidth: 2097.49 GB/sec
MCDRAM Bandwidth: 449.03 GB/sec
DRAM Bandwidth: 85.4 GB/sec
1000
100
10
1
1

Self Time: **536.471** s Total Time: **2631.710** s

The loop is no longer red. This means it takes less time now

Has more GFLOPS, putting it close to the L2 roof

The total performance improvement is almost 3x for the kernel and 50% for the entire application.

Optimization Notice

ROOFLINE PERFORMANCE MODEL:

HOW-TOS

Roofline access and how-to **command line** example

FLOP/S =
#FLOP/Seconds

```
> source advixe-vars.sh
```

```
> advixe-cl --collect survey --project-dir ./your_project --
<your-executable-with-parameters>
```

1st pass
Obtain "Seconds"
1.1x overhead

```
> advixe-cl --collect tripcounts -flops-and-masks --project-dir
./your_project -- <your-executable-with-parameters>
```

2nd pass
Obtain #FLOP count:
3x-5x overhead

```
> advixe-gui ./your_project
```

Launch GUI

MPI example (slurm)

1st step:

```
srun -n <num-of-ranks> -c <num_of_cores_per_rank> advixe-cl -v -collect
survey -project-dir=<same_dir_name> -data-limit=0 <your_executable>
```

2nd step:

```
srun -n <num-of-ranks> -c <num_of_cores_per_rank> advixe-cl -v -collect
tripcounts -flops-and-masks -project-dir=<same_dir_name> -data-limit=0
<your_executable>
```

Observe slower Survey analysis or “finalization”?

(1.5x analysis slow-down or more)

Change default call stacks processing mode (***especially for Fortran***)

```
advixe-cl -collect survey –stackwalk-mode=online –no-stack-stitching
```

Disable system modules and non-interesting modules processing:

```
advixe-cl -collect survey -module-filter-mode=include -module-filter=foo.so
```

Optimization Notice

Observe slow tripcounts/FLOP analysis ??

(> 8x slower than native and more)

Consider combinations:

1. FLOPS **only**, no TripCounts:

   ```
   advixe-cl -collect tripcounts –flops-and-masks –no-trip-counts
   ```

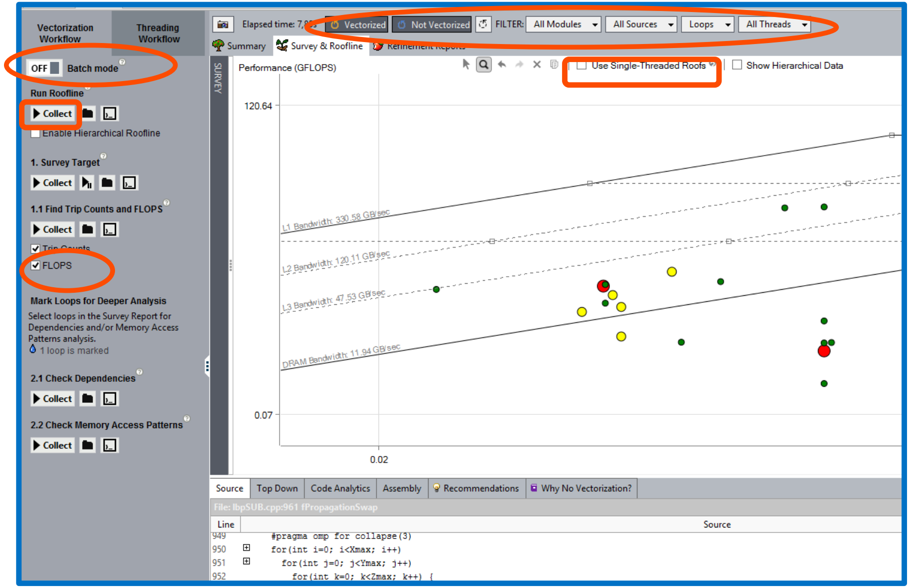
2. no FLOPS , TripCounts **only**, (->No Roofline):

   ```
   advixe-cl -collect tripcounts
   ```

3. FLOPS **and** TripCounts :

   ```
   advixe-cl -collect tripcounts –flops-and-masks
   ```

Roofline GUI access and how-to: **GUI**



OFF Batch mode

Run Roofline

▶ Collect

□ Enable Hierarchical Roofline

1. Survey Target

▶ Collect

1.1 Find Trip Counts and FLOPS

▶ Collect

☑ Trip Counts

☑ FLOPS

Mark Loops for Deeper Analysis

Select loops in the Survey Report for Dependencies and/or Memory Access Patterns analysis.

1 loop is marked

2.1 Check Dependencies

▶ Collect

2.2 Check Memory Access Patterns

▶ Collect

□ Use Single-Threaded Roofs

□ Show Hierarchical Data

File: lbpSUB.cpp:961 fPropagationSwap

Line	Source
949	`#pragma omp for collapse(3)`
950	`for(int i=0; i<Xmax; i++)`
951	`  for(int j=0; j<Ymax; j++)`
952	`    for(int k=0; k<Zmax; k++) {`

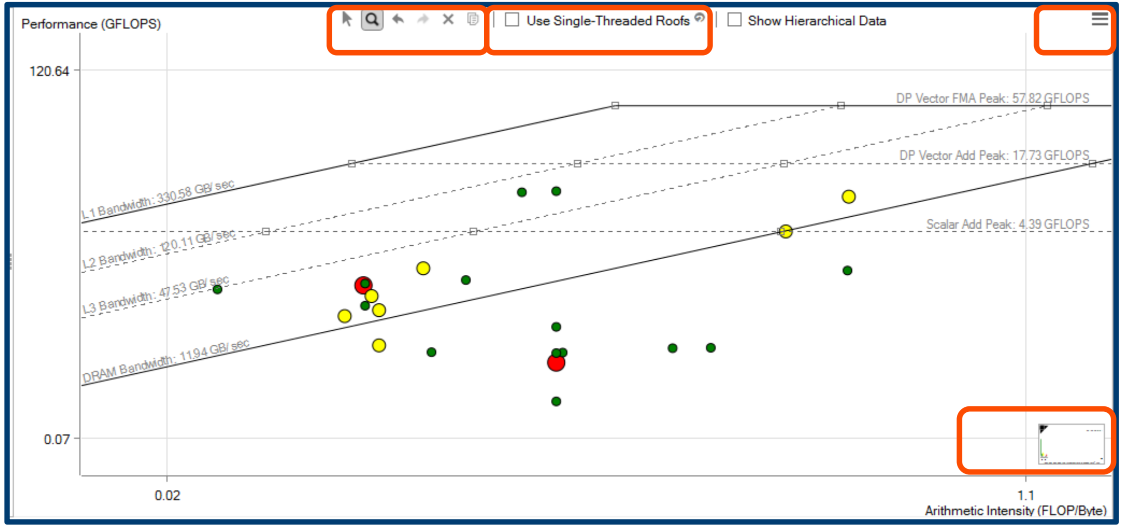
1) **“Run Roofline”** : most automated way.

2) You can also use **two separate runs**:

1. Survey
2. TripCounts (remember to switch **FLOPs** ON)

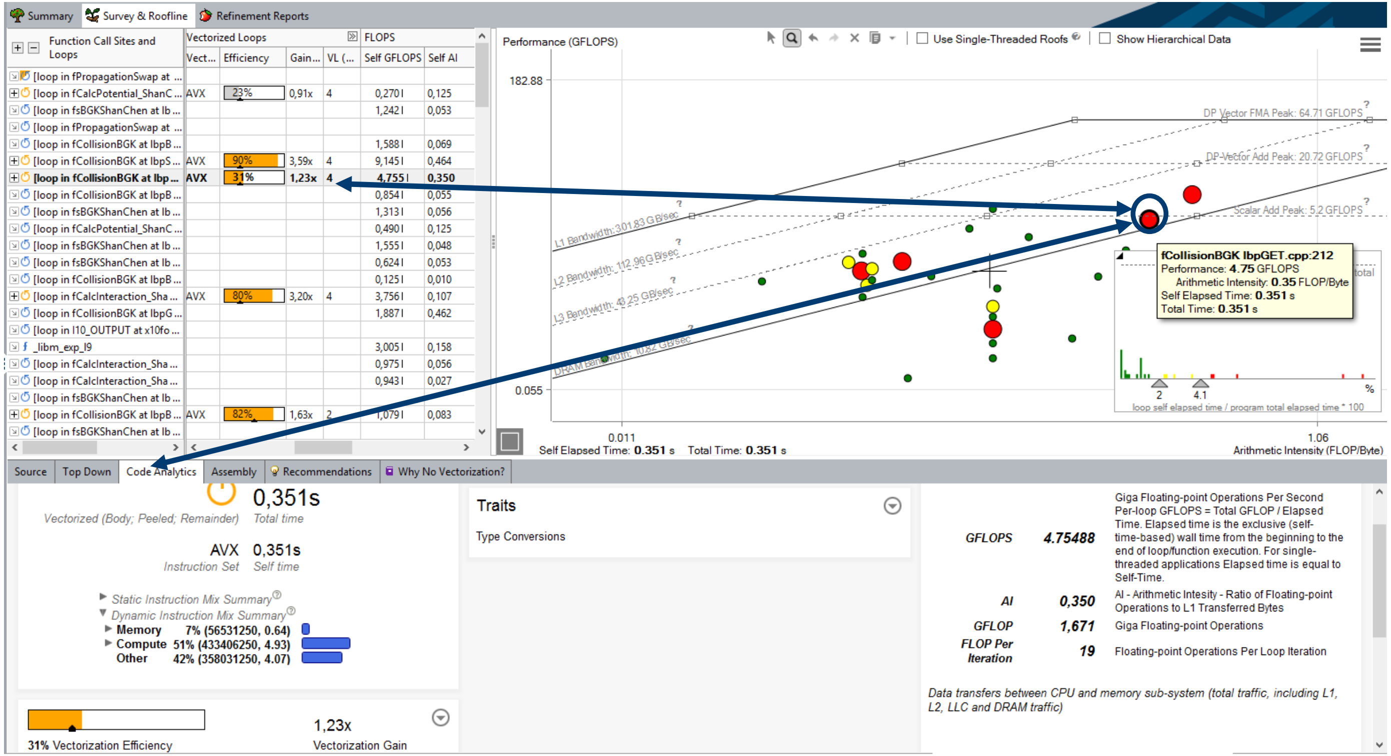
3) **Batch** Mode

Roofline Chart


Performance (GFLOPS)
□ Use Single-Threaded Roofs
□ Show Hierarchical Data
120.64
0.07
0.02
1.1
Arithmetic Intensity (FLOP/Byte)
DP Vector FMA Peak: 57.82 GFLOPS
DP Vector Add Peak: 17.73 GFLOPS
Scalar Add Peak: 4.39 GFLOPS
L1 Bandwidth: 330.58 GB/ sec
L2 Bandwidth: 120.11 GB/ sec
L3 Bandwidth: 47.53 GB/ sec
DRAM Bandwidth: 11.94 GB/ sec

Optimization Notice

Use Vectorization and Roofline views together


Summary
Survey & Roofline
Refinement Reports
Source
Top Down
Code Analytics
Assembly
Recommendations
Why No Vectorization?
Performance (GFLOPS)
□ Use Single-Threaded Roofs
□ Show Hierarchical Data
182.88
0.055
0.011
1.06
Arithmetic Intensity (FLOP/Byte)
DP Vector FMA Peak: 64.71 GFLOPS
DP Vector Add Peak: 20.72 GFLOPS
Scalar Add Peak: 5.2 GFLOPS
L1 Bandwidth: 301.83 GB/sec
L2 Bandwidth: 112.96 GB/sec
L3 Bandwidth: 43.25 GB/sec
DRAM Bandwidth: 10.82 GB/sec
total
2
4.1
%
loop self elapsed time / program total elapsed time * 100

Function Call Sites and Loops	Vectorized Loops				FLOPS	
	Vect...	Efficiency	Gain...	VL (...	Self GFLOPS	Self AI
[loop in fPropagationSwap at ...						
[loop in fCalcPotential_ShanC ...	AVX	23%	0,91x	4	0,270	0,125
[loop in fsBGKShanChen at lb ...					1,242	0,053
[loop in fPropagationSwap at ...						
[loop in fCollisionBGK at lbpB ...					1,588	0,069
[loop in fCollisionBGK at lbpS ...	AVX	90%	3,59x	4	9,145	0,464
[loop in fCollisionBGK at lbp ...	**AVX**	**31%**	**1,23x**	**4**	**4,755**	**0,350**
[loop in fCollisionBGK at lbpB ...					0,854	0,055
[loop in fsBGKShanChen at lb ...					1,313	0,056
[loop in fCalcPotential_ShanC ...					0,490	0,125
[loop in fsBGKShanChen at lb ...					1,555	0,048
[loop in fsBGKShanChen at lb ...					0,624	0,053
[loop in fCollisionBGK at lbpB ...					0,125	0,010
[loop in fCalcInteraction_Sha ...	AVX	80%	3,20x	4	3,756	0,107
[loop in fCollisionBGK at lbpG ...					1,887	0,462
[loop in l10_OUTPUT at x10fo ...						
_libm_exp_l9					3,005	0,158
[loop in fCalcInteraction_Sha ...					0,975	0,056
[loop in fCalcInteraction_Sha ...					0,943	0,027
[loop in fsBGKShanChen at lb ...						
[loop in fCollisionBGK at lbpB ...	AVX	82%	1,63x	2	1,079	0,083
[loop in fsBGKShanChen at lb ...						

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

Self Elapsed Time: **0.351** s Total Time: **0.351** s

0,351s
Vectorized (Body; Peeled; Remainder) *Total time*

AVX 0,351s
Instruction Set *Self time*

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

▶ **Memory**	**7% (56531250, 0.64)**
▶ **Compute**	**51% (433406250, 4.93)**
Other	**42% (358031250, 4.07)**

31% Vectorization Efficiency

1,23x
Vectorization Gain

Traits

Type Conversions

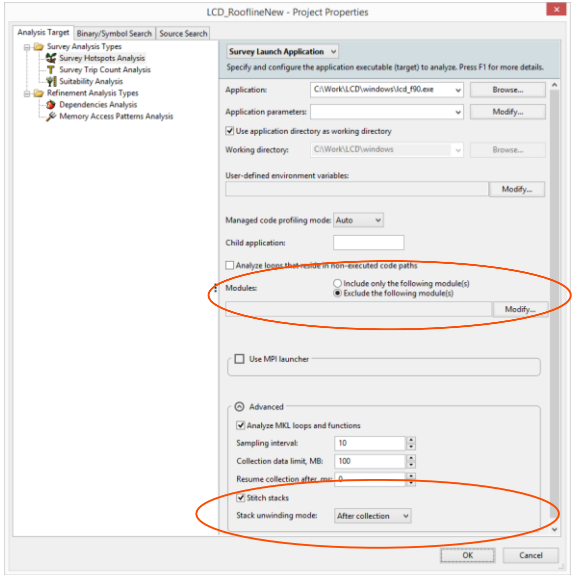
GFLOPS	***4.75488***	Giga Floating-point Operations Per Second Per-loop GFLOPS = Total GFLOP / Elapsed Time. Elapsed time is the exclusive (self-time-based) wall time from the beginning to the end of loop/function execution. For single-threaded applications Elapsed time is equal to Self-Time.
AI	***0,350***	AI - Arithmetic Intesity - Ratio of Floating-point Operations to L1 Transferred Bytes
GFLOP	***1,671***	Giga Floating-point Operations
FLOP Per Iteration	***19***	Floating-point Operations Per Loop Iteration

Data transfers between CPU and memory sub-system (total traffic, including L1, L2, LLC and DRAM traffic)

Observe slower Survey analysis or “finalization”?

(1.5x slower than native run and more)

Configuration via GUI:



LCD_RooflineNew - Project Properties

Specify and configure the application executable (target) to analyze. Press F1 for more details.

Application: C:\Work\LCD\windows\lcd_f90.exe

Application parameters:

☑ Use application directory as working directory

Working directory: C:\Work\LCD\windows

User-defined environment variables:

Managed code profiling mode: Auto

Child application:

□ Analyze loops that reside in non-executed code paths

Modules: ○ Include only the following module(s) ● Exclude the following module(s)

□ Use MPI launcher

Advanced

☑ Analyze MKL loops and functions

Sampling interval: 10

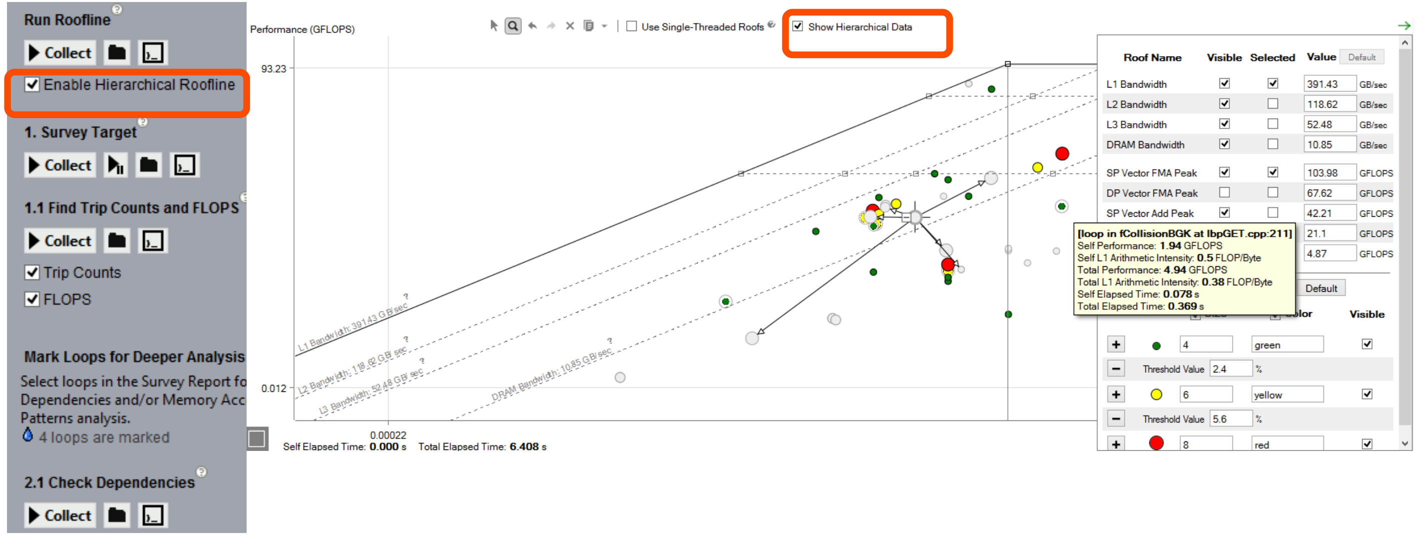
Collection data limit, MB: 100

Resume collection after, ms: 0

☑ Stitch stacks

Stack unwinding mode: After collection

Hierarchical (top-down) Roofline:
new in 2018 release



Run Roofline

▶ Collect

☑ Enable Hierarchical Roofline

1. Survey Target

▶ Collect

1.1 Find Trip Counts and FLOPS

▶ Collect

☑ Trip Counts

☑ FLOPS

Mark Loops for Deeper Analysis

Select loops in the Survey Report fo Dependencies and/or Memory Acc Patterns analysis.

4 loops are marked

2.1 Check Dependencies

▶ Collect

☐ Use Single-Threaded Roofs

☑ Show Hierarchical Data

Performance (GFLOPS)
93.23
0.012
0.00022
L1 Bandwidth: 391.43 GB/sec
L2 Bandwidth: 118.62 GB/sec
L3 Bandwidth: 52.48 GB/sec
DRAM Bandwidth: 10.85 GB/sec

Self Elapsed Time: **0.000** s   Total Elapsed Time: **6.408** s

Roof Name	Visible	Selected	Value	Default
L1 Bandwidth	☑	☑	391.43	GB/sec
L2 Bandwidth	☑	☐	118.62	GB/sec
L3 Bandwidth	☑	☐	52.48	GB/sec
DRAM Bandwidth	☑	☐	10.85	GB/sec
SP Vector FMA Peak	☑	☑	103.98	GFLOPS
DP Vector FMA Peak	☐	☐	67.62	GFLOPS
SP Vector Add Peak	☑	☐	42.21	GFLOPS
			21.1	GFLOPS
			4.87	GFLOPS

[loop in fCollisionBGK at lbpGET.cpp:211]
Self Performance: **1.94** GFLOPS
Self L1 Arithmetic Intensity: **0.5** FLOP/Byte
Total Performance: **4.94** GFLOPS
Total L1 Arithmetic Intensity: **0.38** FLOP/Byte
Self Elapsed Time: **0.078** s
Total Elapsed Time: **0.369** s

Default

			olor	Visible
+	●	4	green	☑
−	Threshold Value	2.4	%	
+	●	6	yellow	☑
−	Threshold Value	5.6	%	
+	●	8	red	☑

```
export ADVIXE_EXPERIMENTAL=roofline_ex
```

Hierarchical Roofline (based on stacks w/ FLOPS)

```
> source advixe-vars.sh

> export ADVIXE_EXPERIMENTAL=roofline_ex



> advixe-cl --collect survey --project-dir ./your_project -- <your-executable-with-
parameters>
```

2nd pass
Obtain #FLOP count:
>>5x overhead

```
> advixe-cl --collect tripcounts -flops-and-masks -callstack-flops --project-dir
./your_project -- <your-executable-with-parameters>


> export ADVIXE_EXPERIMENTAL=roofline_ex

> advixe-gui ./your_project
```

OpenLAB location of Advisor 2017 Update 3

```
Update 3:
```

```
 /oplashare/sw/Intel/advisor_2017_update
```

accessible from openlab machines

Also consider installing advisor on your local laptop. Just copy advisor*.tar.gz from **/oplashare/sw/Intel/advisor_2017_update/** to your laptop, unpack, run advixe-genvars.sh

You'll need to point $INTEL_LICENSE_FILE to license server in openlab

BACK-UP

Why Do We Need the Roofline Model?

Need a sense of absolute performance when optimizing applications

- How do I know if my performance is good?
- Why am I not getting peak performance of the platform?

Many potential optimization directions

- How do I know which one to apply?
- What is the limiting factor in my app's performance?
- How do I know when to stop?