



Intel[®] Advisor Vectorization Optimization and Thread Prototyping

Software Must Vectorize & Thread or Performance Dies

True today – More true tomorrow – Difference can be substantial!

Vectorization - Have you:

- Recompiled for AVX2 or AVX512 with little gain
- Wondered where to vectorize?
- Recoded intrinsics for new arch.?
- Struggled with compiler reports?

Threading - Have you:

- Threaded an app, but seen little benefit?
- Hit a “scalability barrier”?
- Delayed release due to sync. errors?

“Intel® Advisor’s Vectorization Advisor fills a gap in code performance analysis. It can guide the informed user to better exploit the vector capabilities of modern processors and coprocessors.”

Dr. Luigi Iapichino
Scientific Computing Expert
Leibniz Supercomputing Centre

“Intel® Advisor has allowed us to quickly prototype ideas for parallelism, saving developer time and effort”

Simon Hammond
Senior Technical Staff
Sandia National Laboratories

Intel Advisor - Thread Prototyping

Design Parallelism

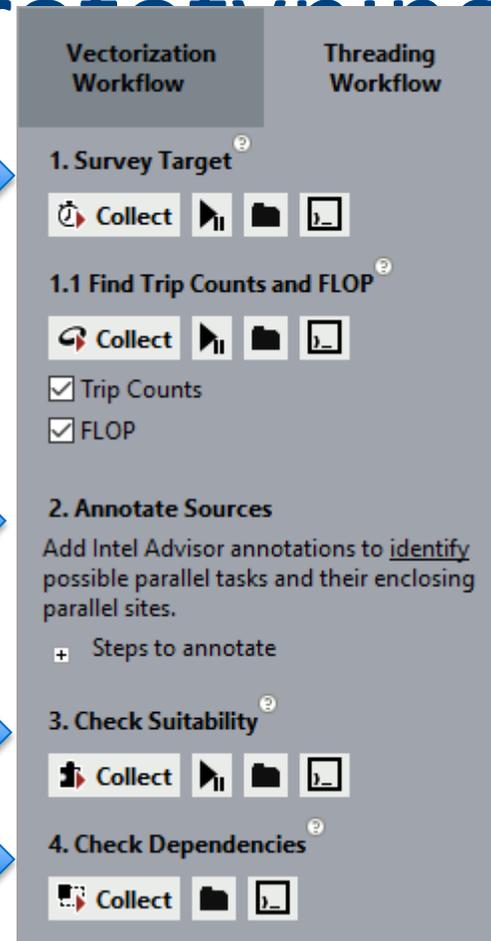
- No disruption to regular development
- All test cases continue to work
- Tune and debug the design before you implement it

1) Analyze it.

2) Design it.
(Compiler ignores these annotations.)

3) Tune it.

4) Check it.



The screenshot shows the Intel Advisor Threading Workflow interface. It is divided into two tabs: 'Vectorization Workflow' and 'Threading Workflow'. The 'Threading Workflow' tab is active and displays a series of steps:

- 1. Survey Target**: Includes a 'Collect' button and icons for play, stop, and refresh.
- 1.1 Find Trip Counts and FLOP**: Includes a 'Collect' button, checkboxes for 'Trip Counts' and 'FLOP', and icons for play, stop, and refresh.
- 2. Annotate Sources**: Includes the text 'Add Intel Advisor annotations to identify possible parallel tasks and their enclosing parallel sites.' and a '+ Steps to annotate' button.
- 3. Check Suitability**: Includes a 'Collect' button and icons for play, stop, and refresh.
- 4. Check Dependencies**: Includes a 'Collect' button and icons for play, stop, and refresh.

Intel Advisor Annotation Concepts

- Advisor uses 3 primary concepts to create a model
 - **SITE**
 - A region of code in your application you want to transform into parallel code
 - **TASK**
 - The region of code in a SITE you want to execute in parallel with the rest of the code in the SITE
 - **LOCK**
 - Mark regions of code in a TASK which must be serialized

NOTE

- All of these regions may be nested
- You may create more than one SITE
- Just macros, so work with any C/C++ compiler

```
#include "advisor-annotate.h"
...
void solve() {
int * queens = new int[size];
//array representing queens placed on a chess board...

ANNOTATE_SITE_BEGIN(solve);

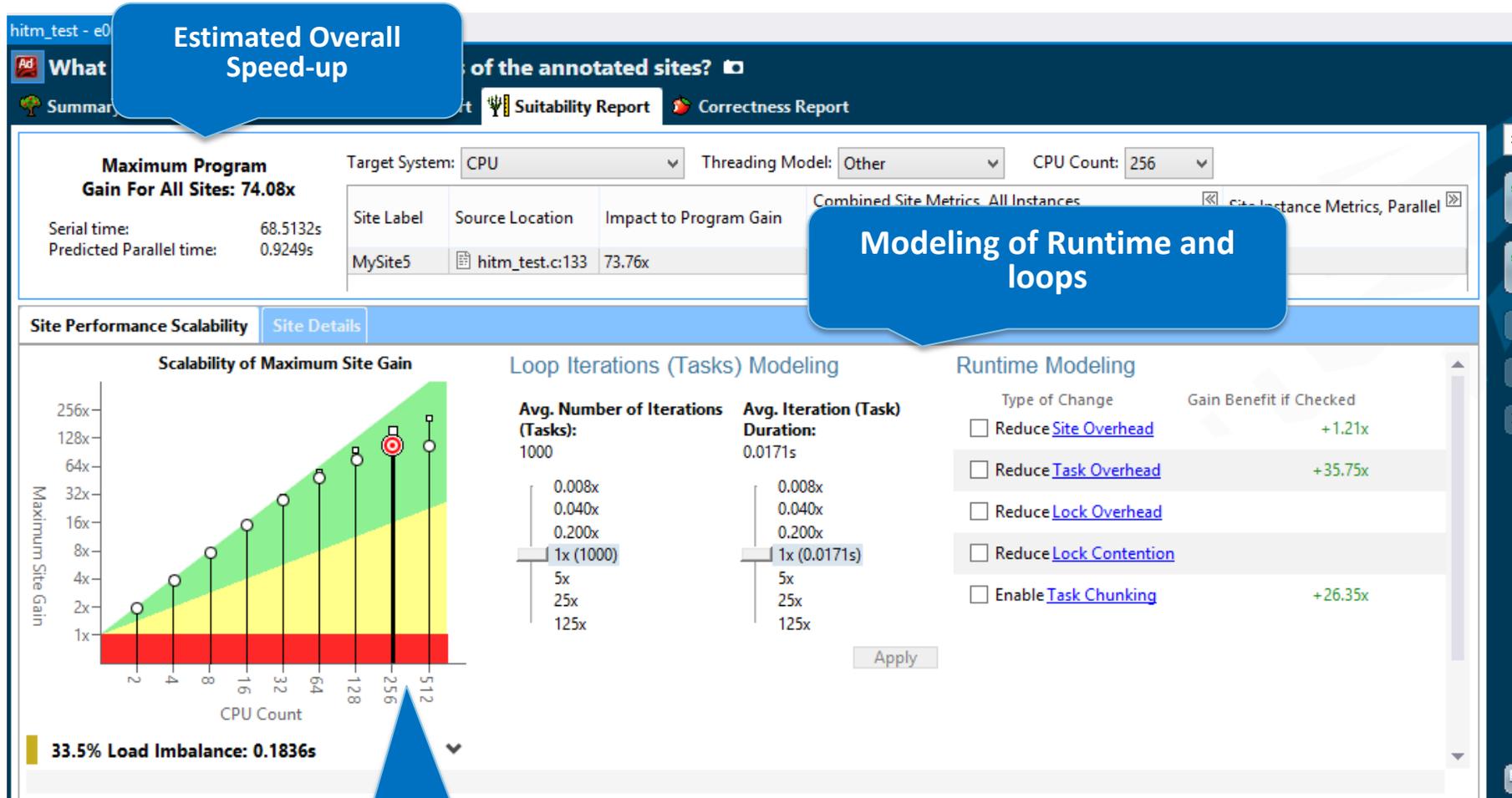
for(int i=0; i<size; i++)
{ // try all positions in first row

ANNOTATE_ITERATION_TASK(setQueen);
setQueen(queens, 0, i);
}

ANNOTATE_SITE_END();

...
}
```

Suitability report

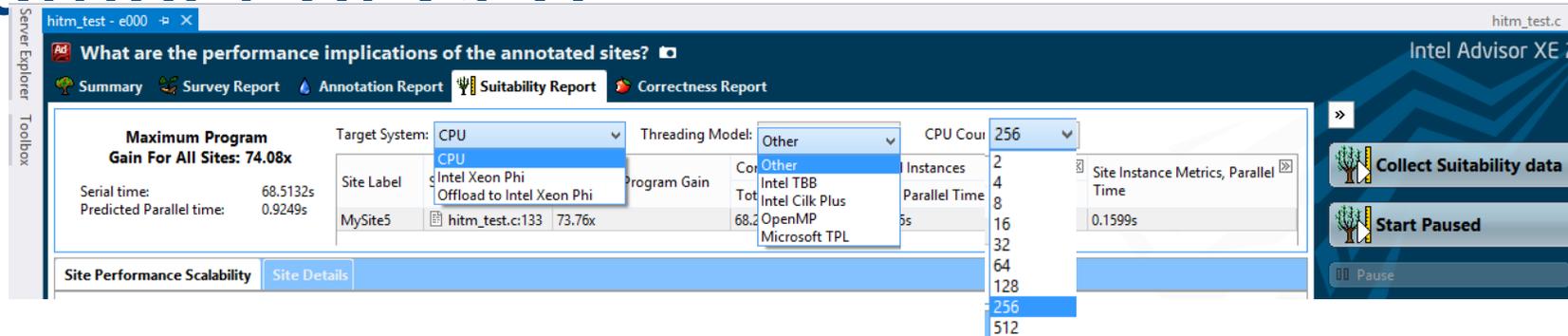


Estimated Overall Speed-up

Modeling of Runtime and loops

Scalability Graph

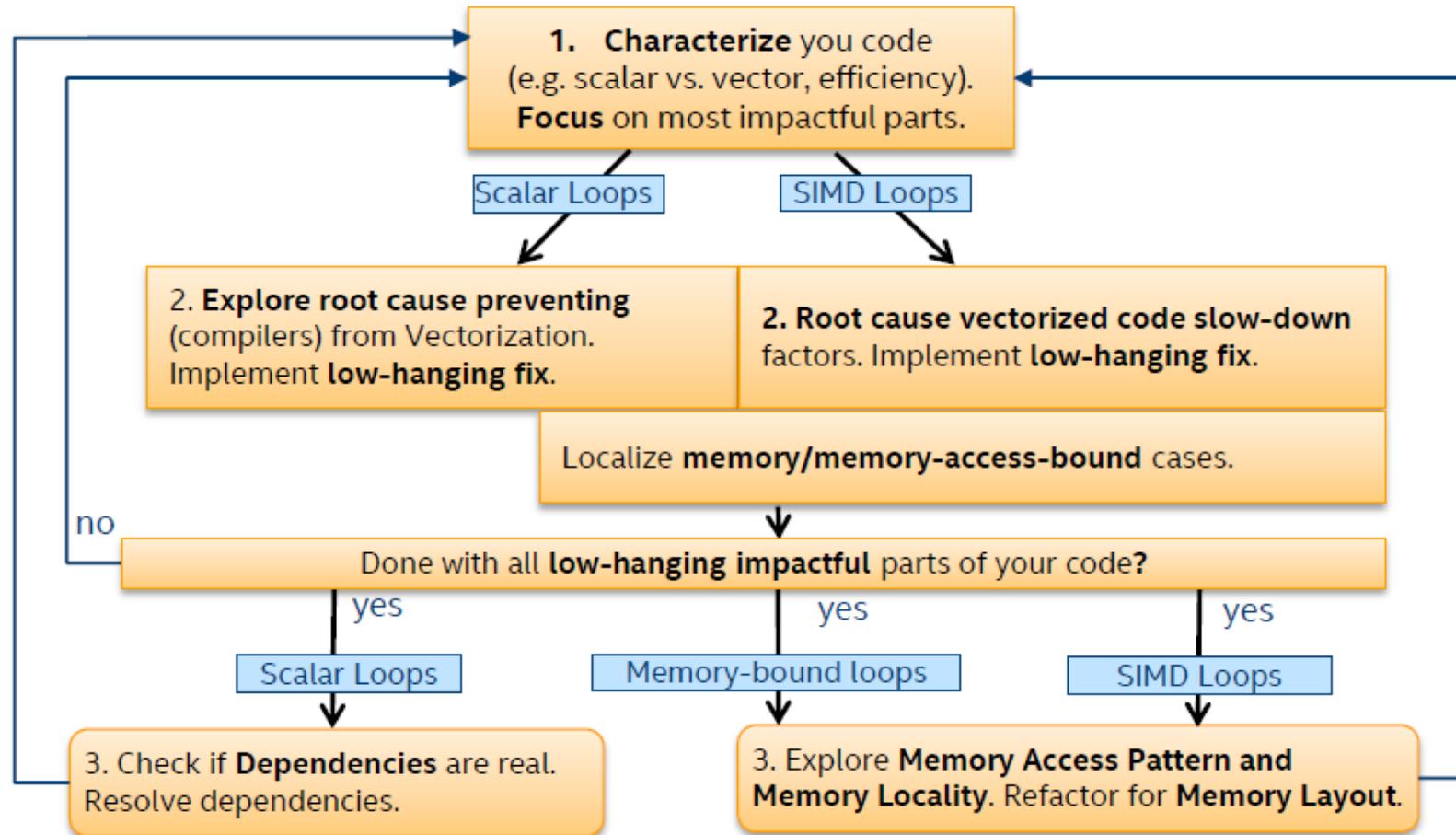
Adjustable: Target architecture, threading models and number of CPU



- To set up data collection determine the target architecture, threading model and number of CPU's
- Collect the Scalability data, and determine how it differs between the architectures and threading models.

Intel® Advisor - Vectorization

Recommended methodology



The Right Data At Your Fingertips

Get all the data you need for high impact vectorization

Filter by which loops are vectorized!

What prevents vectorization?

Focus on hot loops

What vectorization issues do I have?

Which Vector instructions are being used?

How efficient is the code?

Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops			
						Vect...	Efficiency	Gain...	VL (...)
[loop in matvec at Multiply.c:72]	1 Inefficient ...	5.281s	5.281s	Vectorized (Bo...	AVX	~55%	2.19x	4	
[loop in matvec at Multiply.c:66]	1 Ineffective ...	2.828s	2.828s	Vectorized (Bo...	AVX	~91%	3.65x	4	
[loop in matvec at Multiply.c:49]		0.531s	5.812s	Scalar					
[loop in matvec at Multiply.c:49]		0.516s	3.344s	Scalar					
[loop in matvec at Multiply.c:85]	1 Assumed d...	0.063s	0.063s	Scalar	dependence...				
[loop in main at driver.c:151]		0.016s	0.207s	Scalar	loop with function				
[loop in ...]		0s							

Get Fast Code Fast!

5 Steps to Efficient Vectorization - Vector Advisor

(part of Intel® Advisor, Parallel Studio, Cluster Studio 2016)

1. Compiler diagnostics + Performance Data + SIMD efficiency information

Function Call Sites and Loops	Self Time	Total Time	Compiler Vectorization
			Loop Type Why No Vectorization?
[loop in runCForallLambdaLoops]	0.094s	0.094s	Scalar vector dependence prevents vector...
[loop in runCForallLambdaLoops]	0.140s	3.744s	Scalar inner loop was already vectorized
[loop in std::Complex_base<double,struct_C_double_complex>::ci...]	0.031s	0.031s	Vectorized (Body)

Vectorized SSE; SSE2 loop processing Float32; Float64 data type
Peeled loop; loop stmts were reordered

Function Call Sites and Loops	Self Time	Total Time
[loop in std::basic_string<char,struct_std::char_traits<char>,class_std::allo...	0.000s	0.000s
[loop in std::basic_string<char,struct_std::char_traits<char>,class_std::allo...	0.000s	0.000s
[loop in std::num_put<char,class_std::ostreambuf_iterator<char,struct st...	0.000s	0.000s

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

Issue: Peeled/Remainder loop(s) present

All or some source loop iterations are not executing in the kernel loop. Improve performance by moving source loop iterations from peeled/remainder loops to the kernel loop. Read more at [Vector Essentials, Utilizing Full Vectors...](#)

Recommendation: Align memory access
Projected maximum performance gain: High
Projection confidence: Medium

Use one of the memory accesses in the source loop does not align with the memory access boundary. Tell the compiler your memory access is aligned to the byte boundary:

```
SIZE*sizeof(float), 32);
```

3. "Precise" Trip Counts + FLOPs & MASKS: understand utilization, parallelism granularity & overheads

Total Time	Trip Counts			Iteration Duration	Call Count
	Median	Min	Max		
3.151s	1	1	1	3.1509s	1
0.440s	1	1	1	< 0,0001s	2408000
0.010s	1	1	2	< 0,0001s	207596
0.010s	1	2	1	< 0,0001s	1173619
0.010s	1	3	1	< 0,0001s	1312315

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.coc1063	RAW:1	No information available	No information available
loop_site_139	runCRawLoops	runCRawLoops.coc622	No information available	39% / 36% / 25%	Mixed strides
loop_site_160	runCRawLoops	runCRawLoops.coc925	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.coc637	lcal.exe	
P23	0; 0	Unit stride	runCRawLoops.coc638	lcal.exe	
P30	-1575; -63; -26; -25; -1; 0; 1; 25; 26; 63; 2164801	Variable stride	runCRawLoops.coc628	lcal.exe	

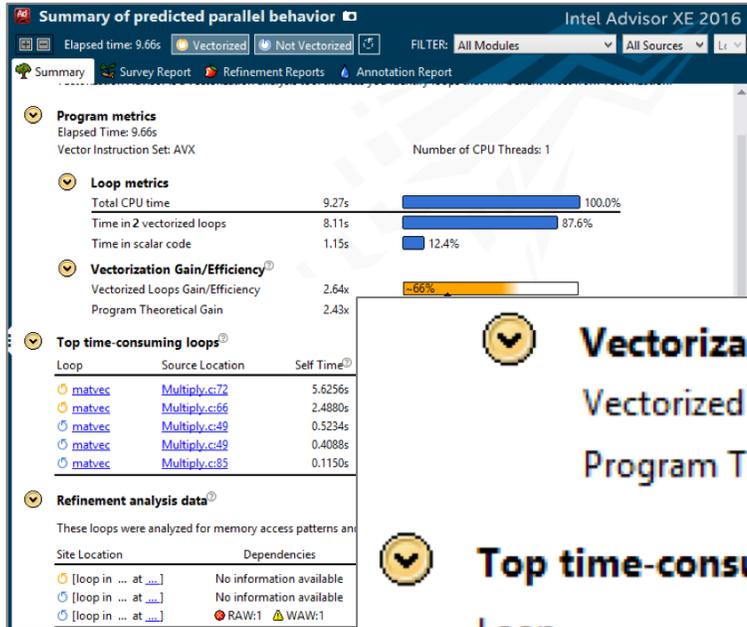
```

635     j2 = ( j2 + 64-1 ) ;
636     p[ip][0] += y[i2+32];
637     p[ip][1] += z[j2+32];
638     i2 += e[i2+32];
639     j2 += f[j2+32];
    
```

```

626     i1 = 64-1;
627     j1 = 64-1;
628     p[ip][2] += b[j1][i1];
    
```

Summary View: Plan Your Next Steps



What can I expect to gain?

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

Top time-consuming loops

Loop	Source Location	Self Time	Total Time
matvec	Multiply.c:72	5.6256s	5.6256s
matvec	Multiply.c:66	2.4880s	2.4880s
matvec	Multiply.c:49	0.5234s	6.1490s
matvec	Multiply.c:49	0.4088s	2.8968s
matvec	Multiply.c:85	0.1150s	0.1150s

Where do I start?

Vectorization Advisor runs on and optimizes for Intel® Xeon Phi™ architecture

AVX-512 ERI – specific to Intel® Xeon Phi

Loops	Vector Issues	Self Time	Loop Type	Vectorized Loops		Instruction Set Analysis							
				Vector ISA	Efficiency	Gain Esti...	VL (V...	Traits	Data Types	Vector ...	Instruction Sets		
[Loop ...]	3 Possible i...	35.226s	5.4%	Vectorized+Threaded (Body; Peeled; Re...	AVX512	-28%	2.21x	8	Divisions; FMA; Gathers	Float32; ...	256/512	AVX; AVX2; AVX512; ...	Masked L
[loc ...]	2 Possible in ...	26.025s	4.0%	Vectorized (Body)+Threaded (OpenMP)	AVX512			8	Divisions; Gathers; FMA	Float32; ...	256/512	AVX; AVX512ER_512; AVX512F...	
[loc ...]	1 High vecto...	5.876s		Vectorized (Peeled)+Threaded (OpenMP)	AVX512			8	Divisions; Gathers; FMA	Float32; ...	256/512	AVX2; AVX512ER_512; AVX512...	Masked Lc
[loc ...]	1 High vecto...	3.324s		Vectorized (Remainder)+Threaded (Open...	AVX512			8	Divisions; Gathers; FMA	Float32; ...	256/512	AVX2; AVX512ER_512; AVX512...	Masked Lc
[loop ...]		34.599s	5.3%	Vectorized (Body; Remainder)	AVX512	-70%	5.64x	8	Divisions; FMA; Square Roots	Float32; ...	256/51...	AVX2; AVX512ER_512; AVX512...	Masked Lc
[loop ...]	1 Possible in ...	33.849s	5.2%	Vectorized (Body; Peeled; Remainder)	AVX512	-28%	2.24x	8	Divisions; FMA; Gathers	Float32; ...	256/512	AVX; AVX2; AVX512ER_512; AV...	Masked Lc
[loop ...]		19.839s	3.1%	Vectorized (Body; Remainder)	AVX512	-72%	11.48x	16; 8					

Efficiency (72%), Speed-up (11.5x), Vector Length (16)

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

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

Confidence: Need More Data

Issue: Ineffective peeled/remainder loop(s) present

All or some [source loop](#) iterations are not executing in the [loop body](#). Improve performance by moving source loop iterations from [peeled/remainder](#) loops to the loop body.

Recommendation: Collect trip counts data

The Survey Report lacks [trip counts](#) data that might generate more precise recommendations. To fix: Run a [Trip Counts analysis](#).

Recommendation: Align data

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

Performance optimization problem and advice how to fix it

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

Loop metrics

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

Check if It Is Safe to Vectorize

Loop-Carried Dependencies Analysis Verifies Correctness

Function Call Sites and Loops	Self Time	Total Time	Lightbulb	Trip Counts	Compiler Vectorization
i> [loop at Multiply.c:53 in matvec]	0.047s	0.047s	<input type="checkbox"/>	3	Vectorized (Body)
i> [loop at Multiply.c:53 in matvec]	0.413s	0.413s	<input type="checkbox"/>	101	Scalar
[-] [loop at Multiply.c:45 in matvec]	0.109s	12.373s	<input type="checkbox"/>	1	Collapse
i> [loop at Multiply.c:45 in matvec]	0.078s	11.930s	<input type="checkbox"/>	12	Vectorized (Body)
i> [loop at Multiply.c:45 in matvec]	0.031s	0.444s	<input type="checkbox"/>	2	Remainder
[loop at Driver.c:146 in main]	0.016s	12.483s	<input checked="" type="checkbox"/>	1000000	Scalar vector dependence prevents vectoriza ...

2.1 Check Dependencies

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



Command Line

Select loop for
Dependency
Analysis and
press play!

Vector Dependence
prevents Vectorization!

Improve Vectorization

Memory Access Pattern Analysis

Ad Where should I add vectorization and/or threading parallelism?

Summary Survey Report Refinement Reports Annotation Report Suitability Report

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

Function Call Sites and Loops				Loop Type	Why No Vectorization?
[loop at fractal.cpp:179 in <lambda1>::op ...				Collapse	Collapse
i> [loop at fractal.cpp:179 in <lambda1>::o ...	<input checked="" type="checkbox"/>	4 Serialized use ...	0,013s 12,020s	Vectorized (Body)	
i> [loop at fractal.cpp:179 in <lambda1>::o ...	<input checked="" type="checkbox"/>	2 Data type co ...	0,000s 0,163s	Peeled	
i> [loop at fractal.cpp:179 in <lambda1>::o ...	<input checked="" type="checkbox"/>	2 Data type co ...	0,000s 0,576s	Remainder	
i> [loop at fractal.cpp:177 in <lambda1>::oper ...	<input type="checkbox"/>	2 Data type co ...	0,010s 12,030s	Scalar	

Select loops of interest

2.2 Check Memory Access Patterns

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



Command Line

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

Get specific advice for Improving Vectorization

Source Top Down Loop Assembly Recommendations Compiler Diagnostic Details

Issue: Ineffective peeled/remainder loop(s) present

All or some [source loop](#) iterations are not executing in the [loop body](#). Improve performance by moving source loop iterations from [peeled/remainder](#) loops to the loop body.

Recommendation: Collect trip counts data Confidence: Need More Data

Recommendation: Specify the expected loop trip count Confidence: Low

The compiler cannot statically detect the [trip count](#). To fix: Identify the expected number of iterations using a [directive](#): `#pragma loop_count`.

Example: Iterate through a loop a minimum of three, maximum of ten, and average of five times:

```
#include <stdio.h>
int mysum(int start, int end, int a) {
    int iret=0;
    #pragma loop_count min(3), max(10), avg(5)
    for (int i=start;i<=end;i++)
```

Read More:

- [loop_count](#)
- [Getting Started with Intel Compiler Pragmas and Directives and ...resources for Intel Advisor users](#)

Advisor XE shows hints how to decrease vectorization overhead

Recommendation: Enforce vectorized remainder Confidence: Low

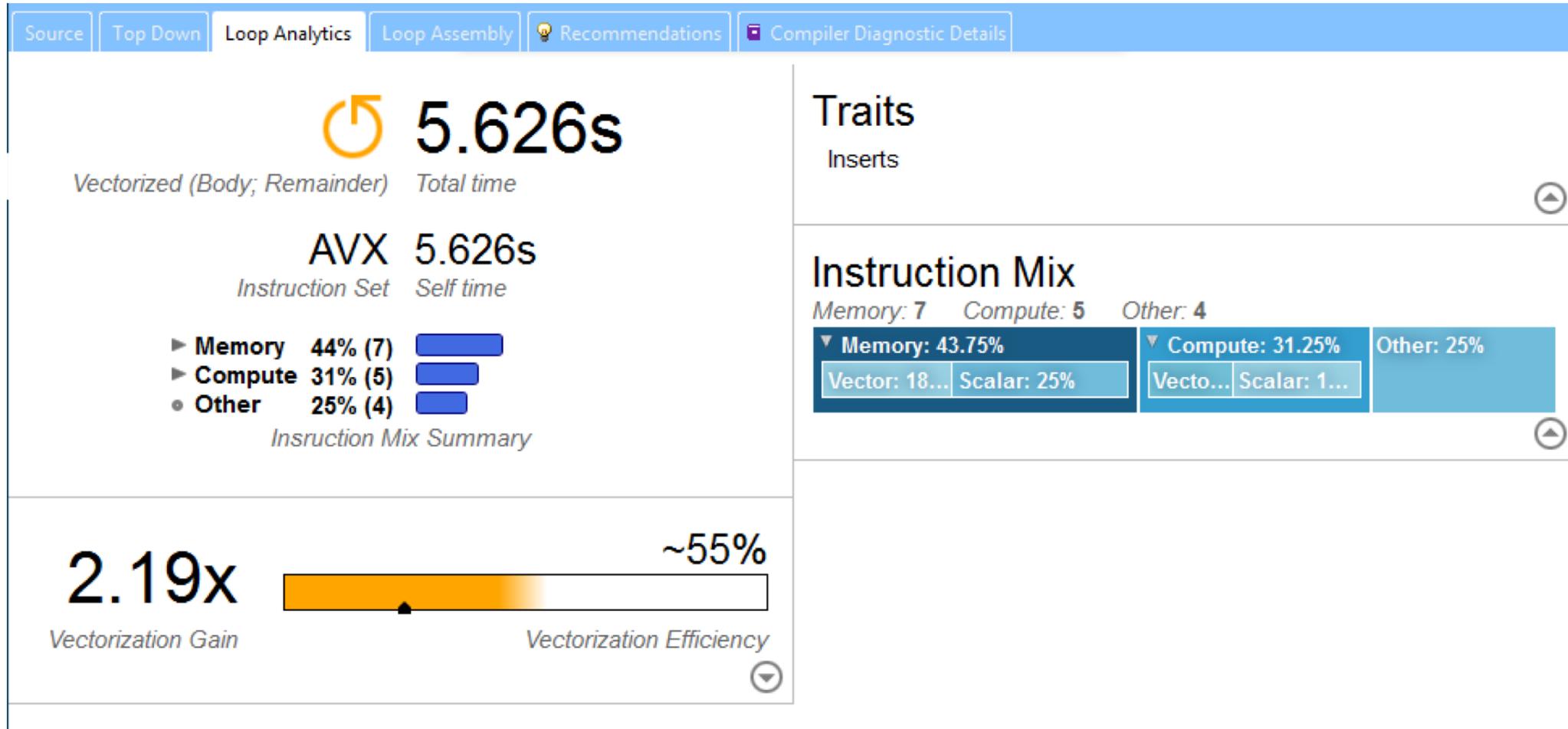
Recommendation: Use a smaller vector length Confidence: Low

Recommendation: Align data Confidence: Low

Recommendation: Add data padding Confidence: Low

Loop Analytics

Get detailed information about your loops



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; ...

Address	Line	Assembly	Total Time	%	Self Time	%	Traits
0x140054b58	6004	vfmadd231ps zmm12, k0, zmm5, zmm16					FMA
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, zmm...					

Summarized Traits in **Survey Report**.

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

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

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	}					

Gather/Scatter analysis is very important for AVX-512

- AVX512 Gather/Scatter in wider use than on previous instruction sets
- Many more applications can now be vectorized
- Gives good average performance but far from optimal
- Much greater need for Gather/Scatter profiling
- With Intel® Advisor you get both dynamic and static gather/scatter information

Irregular access patterns decreases performance!

Gather profiling

- Run Memory Access Pattern Analysis (MAP)



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

Am I bound by VPU/CPU or by Memory?: Advisor Memory Access Pattern and Footprint

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

Source	Stride	Operand Type	Operand Size ...	Aggregated footprint
m=1; m<=half; m++) {	[0]	int	32	4B
= fCpMod(i + lbv[3*m], Xmax);	[0] [3]	int	32:64	88B
= fCpMod(j + lbv[3*m+1], Ymax);	[0] [3]	int	32	88B
= fCpMod(k + lbv[3*m+2], Zmax);	[0] [3]	int	32	88B
= (nextx * Ymax + nexty) * Zmax + nextz;				
lbsitelength + 1*lbsy.nq + m + half], lbf[i]next*lbsitel...	[0] [1] [-4..	float64:int	32:64	9MB
lbsitelength + 1*lbsy.nq + m + 1], lbf[i]l*lbsitelength + 1				

Assembly	Physical Stride	Operand Info	Address range	Memory access
f290 1250 add rax, 0x3				
f294 1254 imul r14d, r11d				
f298 1254 add r14d, r12d				
f2cb 1256 mov r12, qword ptr [r9+rsi*8]	-43775, 118377...	int*, int*, i...	0x27561058 - 0x27e6cf20	9MB
f2cf 1256 vmovsd xmm0, qword ptr [r8+rbx*8]	1	float64*1	0x27561098 - 0x275610d0	64B
f2d5 1256 mov qword ptr [r8+rbx*8], r12	1	int*1	0x27561098 - 0x275610d0	64B
f2d9 1256 mov r13d, dword ptr [rip+0x1565bc]	0	int*1	0x18589c - 0x18589c	4B

AVX-512 FLOPS and Mask profiler

1. Survey Target

Collect

1.1 Find Trip Counts And FLOPS

Collect



FLOPs, Masks, Trip Counts					
Median	GFLOPs/s	Arithmetic Intensity	Mask Utiliz...	GBytes/s	GFLOP
19	2,456	0.125		19.6498	3.94488
4; 3	2,351	0.125	63,29%	18.8111	0.36693
19	2,136	0.0795455		26.8513	2.50208
19	1,910	0.0681818		28.011	1.0723
3	1,774	0.0833333		21.2898	0.1128
4	1,192	0.0666667		17.8726	0.1505
19	0,911	0.0681818		13.3635	0.0285

Low mask population -> low performance (in spite of "high SIMD efficiency")

Function Call Sites and Loops	Vector Issues	Self Time	Type	FLOPS			Vector ISA	Efficiency	Gain Esti...	VL	Instruction Set Analysis
				GFLOPS	AI	Mask Utilization					
[loop in s2711 at loops90.f:16 ...]		0,010s	Vectorized (Remainder)	0,800	0,1000	100%	AVX512	~56%	8.98x	16	FMA
[loop in s252 at loops90.f:1172]	2 Ineffective peeled/r ...	0,171s	Vectorized Versions	1,684	0,0968	100%	AVX512	~41%	13.05x	16;	Blends; Divisions; Extracts; I
[loop in s116 at loops90.f:257]	1 Ineffective peeled/r ...	0,100s	Vectorized (Body; Remainder)	4,951	0,0833	88%	AVX512	~79%	12.68x	16	Unpacks
[loop in s174 at loops90.f:765]	1 Ineffective peeled/r ...	0,080s	Vectorized (Body; Remainder)	1,251	0,0833	78%	AVX512	~41%	13.05x	16;	Unpacks
[loop in s173 at loops90.f:7 ...]	1 Ineffective peeled ..	0,090s	Vectorized (Body; Remainder)	1,111	0,0833	78%	AVX512	~41%	13.05x	16;	Unpacks
[loop in s152 at loops90.f:624]	1 Ineffective peeled/r ...	0,010s	Vectorized (Body; Remainder)	10,001	0,0833	89%	AVX512	~37%	11.70x	16;	FMA
[loop in s121 at loops90.f:324]	1 Ineffective peeled/r ...	0,020s	Vectorized (Body; Remainder)	4,950	0,0833	88%	AVX512	~36%	11.47x	16;	Unpacks
[loop in s151s at loops90.f:609]	1 Ineffective peeled/r ...	0,020s	Vectorized (Body; Remainder)	4,950	0,0833	88%	AVX512	~36%	11.47x	16;	Unpacks
[loop in s131 at loops90.f:521]		0,020s	Vectorized (Body)	4,808	0,0833	100%	AVX512	~36%	11.47x	32	
[loop in s119 at loops90.f:302]	1 Ineffective peeled/r ...	0,040s	Vectorized (Body; Remainder)	2,450	0,0833	88%	AVX512	~35%	11.25x	16;	Unpacks
[loop ...]				1,649	0,0833	89%	AVX512	~35%	11.25x	16;	Unpacks
[loop ...]				1,400	0,0833	88%	AVX512	~35%	11.25x	16;	Unpacks
[loop ...]				0,797	0,0833		AVX512	~22%	3.55x	16	FMA
[loop ...]				0,840	0,0833	78%	AVX512	~19%	2.97x	16	2-Source Permutes; Gather

Efficiency, FLOPS and Arithmetic Intensity correlation (more memory bound -> lower SIMD performance)

Why is Mask Utilization important?

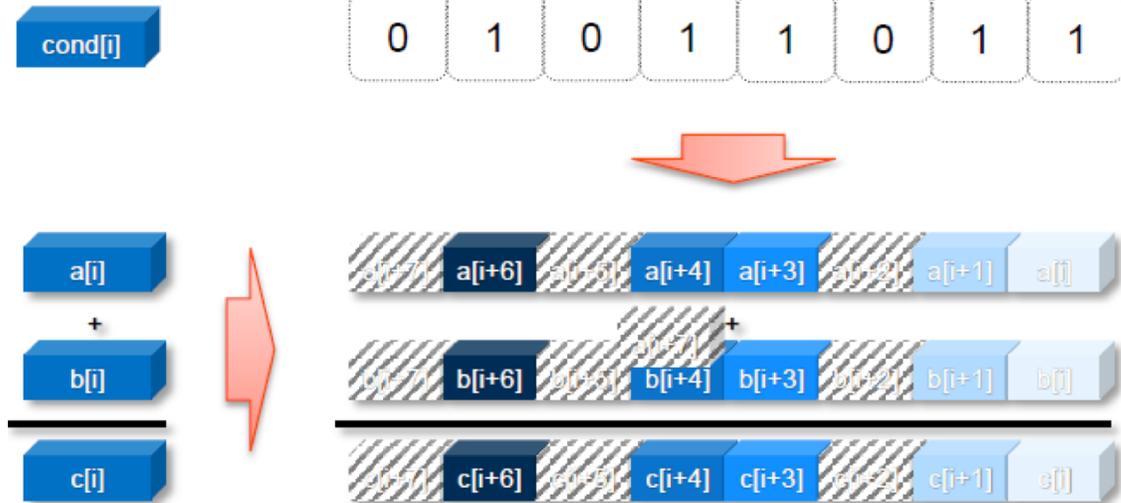
3 elements suppressed

SIMD Utilization = 5/8
(62.5%)

Not utilizing full vectors!!

Fully utilized!

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



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



Advisor Roofline: under the hood

Roofline application profile:

Axis Y: $\text{FLOP/S} = \# \text{FLOP (mask aware)} / \# \text{Seconds}$

Axis X: $\text{AI} = \# \text{FLOP} / \# \text{Bytes}$

Seconds

User-mode **sampling**

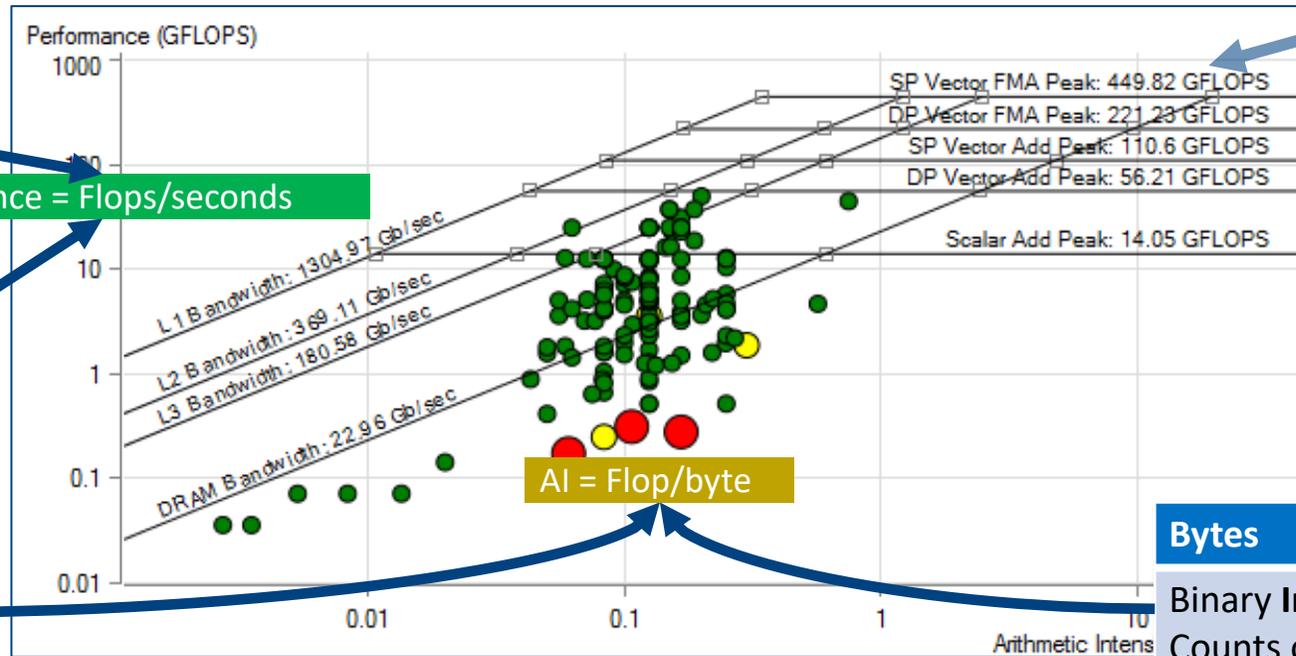
Root access not needed

Roofs

Microbenchmarks

Actual peak for the current configuration

Performance = Flops/seconds



#FLOP

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

Bytes

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

Getting Roofline in Advisor

FLOP/S = #FLOP/Seconds	Seconds	#FLOP Count - Mask Utilization - #Bytes
Step 1: Survey <ul style="list-style-type: none">- Non intrusive. <i>Representative</i>- Output: Seconds (+much more)		
Step 2: FLOPS <ul style="list-style-type: none">- Precise, instrumentation based- Physically count Num-Instructions- Output: #FLOP, #Bytes		

1. Survey Target [?]

▶ Collect ▶   

1.1 Find Trip Counts and FLOPS [?]

▶ Collect  

Cache-Aware Roofline

Next Steps

If under or near a memory roof...

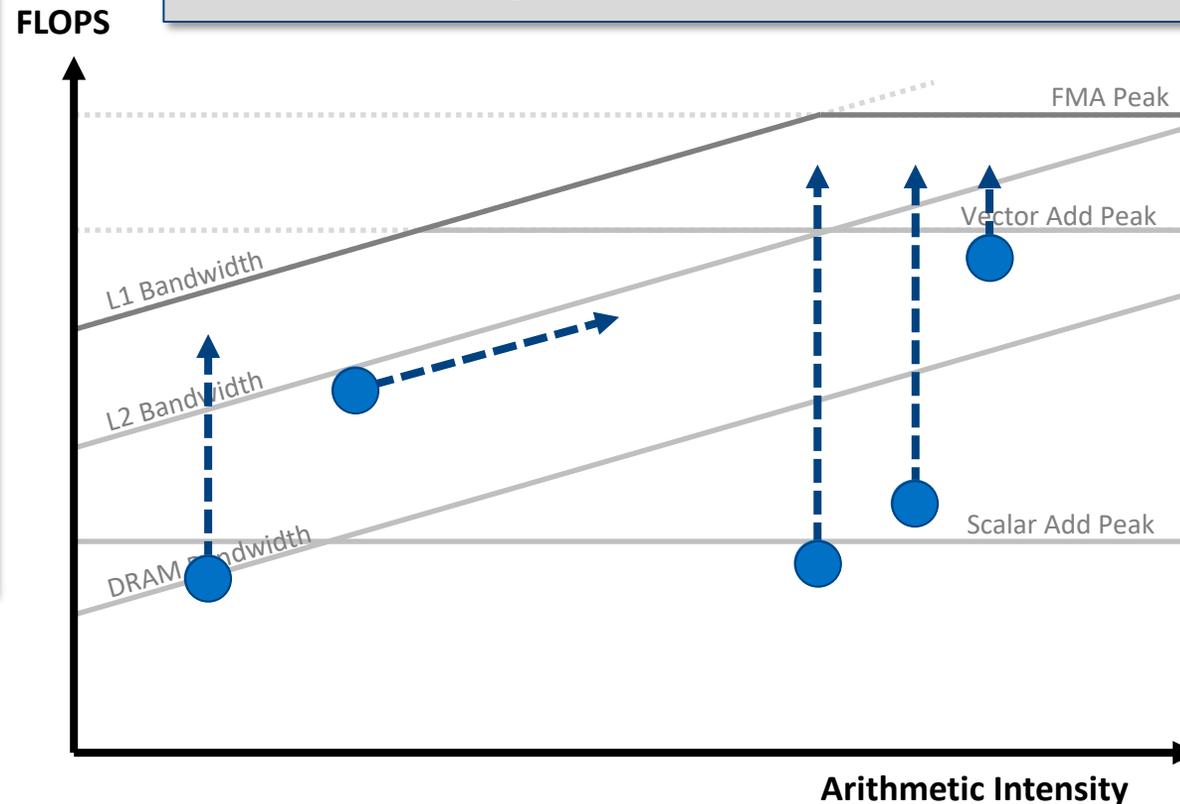
- Try a MAP analysis. Make any appropriate **cache optimizations**.
- If cache optimization is impossible, try **reworking the algorithm to have a higher AI**.

If Under the Vector Add Peak

Check “Traits” in the Survey to see if FMAs are used. If not, try altering your code or compiler flags to **induce FMA usage**.

If just above the Scalar Add Peak

Check **vectorization efficiency** in the Survey. Follow the recommendations to improve it if it's low.



If under the Scalar Add Peak...

Check the Survey Report to see if the loop vectorized. If not, try to **get it to vectorize** if possible. This may involve running Dependencies to see if it's safe to force it.

Typical Vectorization Optimization Workflow

- There is no need to recompile or relink the application, but the use of `-g` is recommended.
1. Collect survey and tripcounts data
 - Investigate application place within roofline model
 - Determine vectorization efficiency and opportunities for improvement
 2. Collect memory access pattern data
 - Determine data structure optimization needs
 3. Collect dependencies
 - Differentiate between real and assumed issues blocking vectorization

Nbody demonstration

Nbody gravity simulation

<https://github.com/fbaru-dev/nbody-demo> (Dr. Fabio Baruffa)

- Let's consider a distribution of point masses m_1, \dots, m_n located at r_1, \dots, r_n .
- We want to calculate the position of the particles after a certain time interval using the Newton law of gravity.

```
struct Particle
{
    public:
        Particle() { init();}
        void init()
        {
            pos[0] = 0.; pos[1] = 0.; pos[2] = 0.;
            vel[0] = 0.; vel[1] = 0.; vel[2] = 0.;
            acc[0] = 0.; acc[1] = 0.; acc[2] = 0.;
            mass = 0.;
        }
        real_type pos[3];
        real_type vel[3];
        real_type acc[3];
        real_type mass;
};
```

```
for (i = 0; i < n; i++){           // update acceleration
    for (j = 0; j < n; j++){
        real_type distance, dx, dy, dz;
        real_type distanceSqr = 0.0;
        real_type distanceInv = 0.0;

        dx = particles[j].pos[0] - particles[i].pos[0];
        ...

        distanceSqr = dx*dx + dy*dy + dz*dz + softeningSquared;
        distanceInv = 1.0 / sqrt(distanceSqr);

        particles[i].acc[0] += dx * G * particles[j].mass *
                               distanceInv * distanceInv * distanceInv;
        particles[i].acc[1] += ...
        particles[i].acc[2] += ...
```

Collect Roofline Data

- Starting with version 2 of the code we collect both survey and tripcounts data:
 - `mpirun -n 1 -N 1 advixe-cl --collect survey --project-dir ./adv_res --search-dir src:=./ \`
 - `--search-dir bin:=./ -- ./nbody.x`
 - `mpirun -n 1 -N 1 advixe-cl --collect tripcounts -flops-and-masks --project-dir ./adv_res \`
 - `--search-dir src:=./ --search-dir bin:=./ -- ./nbody.x`
 - `mpirun -n 1 -N 1 advixe-cl -collect roofline (available starting 2018 U1)`
- If finalization is too slow on compute add `-no-auto-finalize` to collection line.

Summary Report

Elapsed time: 10.24s | Vectorized | Not Vectorized | FILTER: All Modules | All Sources | INTEL ADVISOR 2018

Summary | Survey & Roofline | Refinement Reports

Vectorization Advisor

Vectorization Advisor is a vectorization analysis toolset that lets you identify loops that will benefit most from vector parallelism, discover performance issues preventing from effective vectorization and characterize your memory vs. vectorization bottleneck. Advisor Roofline model automation.

Program metrics

Elapsed Time	10.24s		
Vector Instruction Set	AVX512, AVX2, AVX	Number of CPU Threads	1
Total GFLOP Count	21.20	Total GFLOPS	2.07
Total Arithmetic Intensity [®]	0.35165		

Loop metrics

Metrics	Total		
Total CPU time	10.14s	<div style="width: 100.0%;"></div>	100.0%
Time in 1 vectorized loop	10.08s	<div style="width: 99.4%;"></div>	99.4%
Time in scalar code	0.06s	<div style="width: 0.6%;"></div>	
Total GFLOP Count	21.20	<div style="width: 100.0%;"></div>	100.0%
Total GFLOPS	2.07		

Vectorization Gain/Efficiency

Vectorized Loops Gain/Efficiency [®]	10.05x	<div style="width: 63%;"></div>	63%
Program Approximate Gain [®]	10.00x		

Top time-consuming loops[®]

Loop	Self Time [®]	Total Time [®]	Trip Counts [®]
[loop in GSimulation-start at GSimulation.cpp:138]	10.080s	10.080s	125
[loop in GSimulation-start at GSimulation.cpp:136]	0.060s	10.140s	2000
[loop in GSimulation-start at GSimulation.cpp:133]	0s	10.140s	500

- GUI left panel provides access to further tests
- Summary provides overall performance characteristics
- Lists instruction set(s) used
- Top time consuming loops are listed individually
- Loops are annotated as vectorized and non-vectorized
- Vectorization efficiency is based on used ISA, in this case Intel[®] Advanced Vector Extensions 512 (AVX512)

Survey Report (Source)

The screenshot displays the Intel Advisor 2018 interface. The top bar shows 'Elapsed time: 10.24s' and various filters. The 'Survey & Roofline' tab is active, showing a table of performance issues. The selected issue is '[loop in GSsimulation::start at GSsimulation.cpp:138]' with a self time of 10.080s. The table below shows the source code for this loop, which is vectorized using AVX512. The vectorization efficiency is 63%, and the vector length used is 16. The total time for the loop is 10.100s.

Function Call Sites and Loops	Performance Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops	FLOPS
						Vector ...	Self GFLOPS
[loop in GSsimulation::start at GSsimulation.cpp:138]	2 Inefficient gather/sc...	10.080s	10.080s	Vectorized (Body)		AVX5...	2.093
[loop in GSsimulation::start at GSsimulation.cpp:136]	1 Opportunity for outer l...	0.060s	10.140s	Scalar	inner loop was already v...		1.700
f_start		0.000s	10.140s	Function			
main		0.000s	10.140s	Function			
GSimulation::start		0.000s	10.140s	Function			
[loop in GSsimulation::start at GSsimulation.cpp:133]	1 Data type conversions ...	0.000s	10.140s	Scalar	inner loop was already v...		

Line	Source	Total Time	%	Loop/Function Time	%	Traits
132	const double t0 = time.start();					
133	for (int s=1; s<=get_nsteps(); ++s)					
134	{					
135	t0 += time.start();					
136	for (i = 0; i < n; i++)// update acceleration					
137	{					
138	for (j = 0; j < n; j++)	0.100s		10.080s		
	[loop in GSsimulation::start at GSsimulation.cpp:138]					
	Vectorized AVX512ER_512; AVX512P_512 loop processes Float32; Int32; UInt32 data type(s) and includes 2-Source Perm					
	No loop transformations applied					
139	{					
140	real_type dx, dy, dz;					
141	real_type distanceSqr = 0.0f;					
142	real_type distanceInv = 0.0f;					
143	}					
Selected (Total Time):		0.100s				

- Inline information regarding loop characteristics
- ISA used
- Types processed
- Compiler transformations applied
- Vector length used
- ...

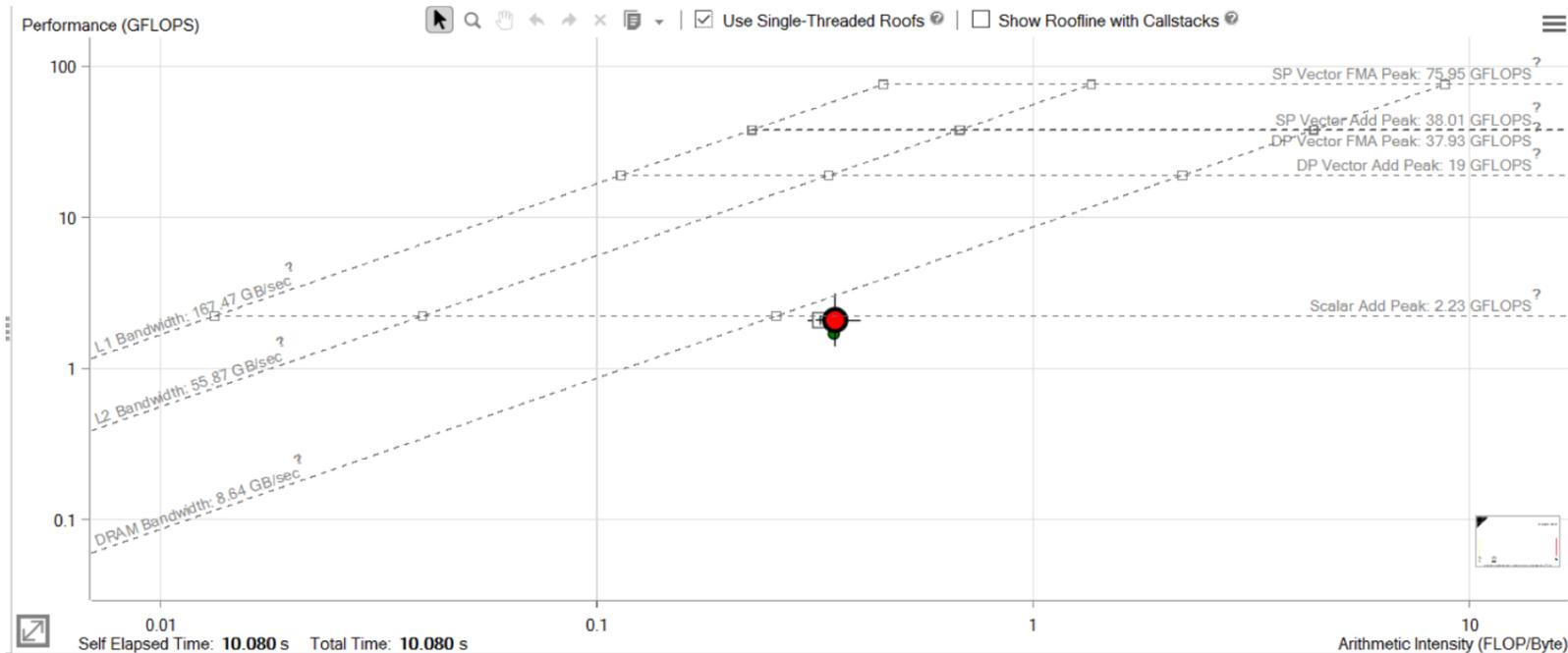
Survey Report (Code Analytics)

- Detailed loop information
- Instruction mix
- ISA used, including subgroups
- Loop traits
 - FMA
 - Square root
 - Gathers / Blends point to memory issues and vector inefficiencies

The screenshot displays a code analytics tool interface. At the top, a table lists performance metrics for various code elements. A red arrow points to the 'ROOTLINE' sidebar on the left. The main content area shows a detailed view of a loop in GSimulation.cpp, including its total time (10.080s), self time (10.080s), and vectorization efficiency (63%). Below this, there are sections for 'Static Instruction Mix Summary' and 'Dynamic Instruction Mix Summary', both showing a mix of Memory (39%), Compute (37%), Mixed (4%), and Other (21%) instructions. The 'Traits' section lists 'Square Roots' (Gathers) and 'Blends', both with a note: 'Irregular Memory Access Patterns May Decrease Performance. Suggestion: See Recommendations Tab'. The 'FMA' section lists '2-Source Permutes' and 'Mask Manipulations'. On the right, summary statistics include 'Average Trip Counts: 125', 'GFLOPs: 2.09325', 'AVX-512 Mask Usage: 37', and 'Static Instruction Mix' with 'Memory: 22', 'Compute: 21', 'Mixed: 2', and 'Other: 12'.

Function Call Sites and Loops	Performance Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops	FLOPs
						Vector... Efficiency Gain E... VL (Ve... Self C	
[loop in GSimulation::start at GSimulation.cpp:138]	2 Inefficient gat...	10.080s	10.080s	Vectorized (Body)		AVX5... 63% 10.05x 16	2.093
[loop in GSimulation::start at GSimulation.cpp:136]	1 Opportunity for...	0.060s	10.140s	Scalar	inner loop was already v...		1.700
f_start		0.000s	10.140s	Function			
f_main		0.000s	10.140s	Function			
f GSimulation::start		0.000s	10.140s	Function			
[loop in GSimulation::start at GSimulation.cpp:133]	1 Data type conv...	0.000s	10.140s	Scalar	inner loop was already v...		

CARM Analysis



- Using single threaded roof
- Code vectorized, but performance on par with scalar add peak?
- Irregular memory access patterns force gather operations.
- Overhead of setting up vector operations reduces efficiency.

Next step is clear: perform a **Memory Access Pattern** analysis

Memory Access Pattern Analysis (Refinement)

```
mpirun -n 1 -N 1 advixe-cl --collect map -mark-up-list=1 --project-dir ./adv_res --search-dir src:=./ --search-dir bin:=./ -- ./nbody.x
```

Site Location	Loop-Carried Dependencies	Strides Distribution	Access Pattern	Max. Site Footprint	Site Name	Recommendations
[loop in start at GSimulation.cpp:1...	No information available	33% / 33% / 33%	Mixed strides	5KB	loop_site_1	2 Inefficient gather/scatter instructions present

ID	Stride	Type	Source	Nested Function	Variable references	Max. Site Footprint	Modules	Site Name	Access Type
P1	10; 40	Constant stride	GSimulation.cpp:144		block 0x60a0b0 allocated at GSimulation.cpp:109	4KB	nbody.x	loop_site_1	Read
142 real_type distanceInv = 0.0f; 143 144 dx = particles[j].pos[0] - particles[i].pos[0]; //1flop 145 dy = particles[j].pos[1] - particles[i].pos[1]; //1flop 146 dz = particles[j].pos[2] - particles[i].pos[2]; //1flop									
P2		Gather stride	GSimulation.cpp:144		block 0x60a0b0 allocated at GSimulation.cpp:109	5KB	nbody.x	loop_site_1	Read
142 real_type distanceInv = 0.0f; 143 144 dx = particles[j].pos[0] - particles[i].pos[0]; //1flop 145 dy = particles[j].pos[1] - particles[i].pos[1]; //1flop 146 dz = particles[j].pos[2] - particles[i].pos[2]; //1flop									
P3		Parallel site information	GSimulation.cpp:144				nbody.x	loop_site_1	
142 real_type distanceInv = 0.0f; 143 144 dx = particles[j].pos[0] - particles[i].pos[0]; //1flop 145 dy = particles[j].pos[1] - particles[i].pos[1]; //1flop 146 dz = particles[j].pos[2] - particles[i].pos[2]; //1flop									
P5	0	Uniform stride	GSimulation.cpp:149			4B	nbody.x	loop_site_1	Read
147 148 distanceSqr = dx*dx + dy*dy + dz*dz + softeningSquared; //6flops 149 distanceInv = 1.0f / sqrtf(distanceSqr); //1div+1sqrt 150 151 particles[i].acc[0] += dx * G * particles[j].mass * distanceInv * distanceInv * distanceInv; //6flops									

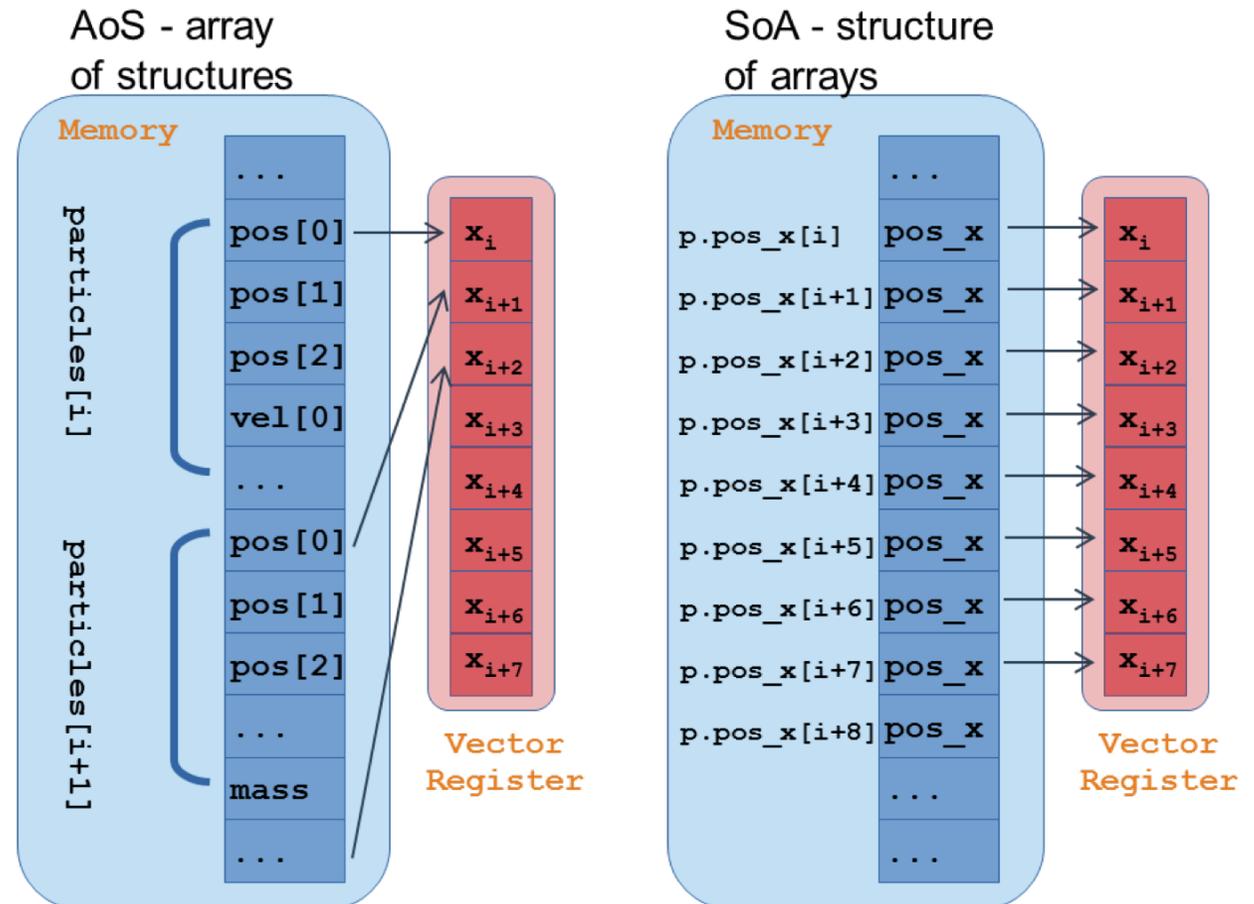
- Storage of particles is in an Array Of Structures (AOS) style
- This leads to regular, but non-unit strides in memory access
- 33% unit
- 33% uniform, non-unit
- 33% non-uniform
- Re-structuring the code into a Structure Of Arrays (SOA) may lead to unit stride access and more effective vectorization

Vectorization: gather/scatter operation

- The compiler might generate gather/scatter instructions for loops automatically vectorized where memory locations are not contiguous

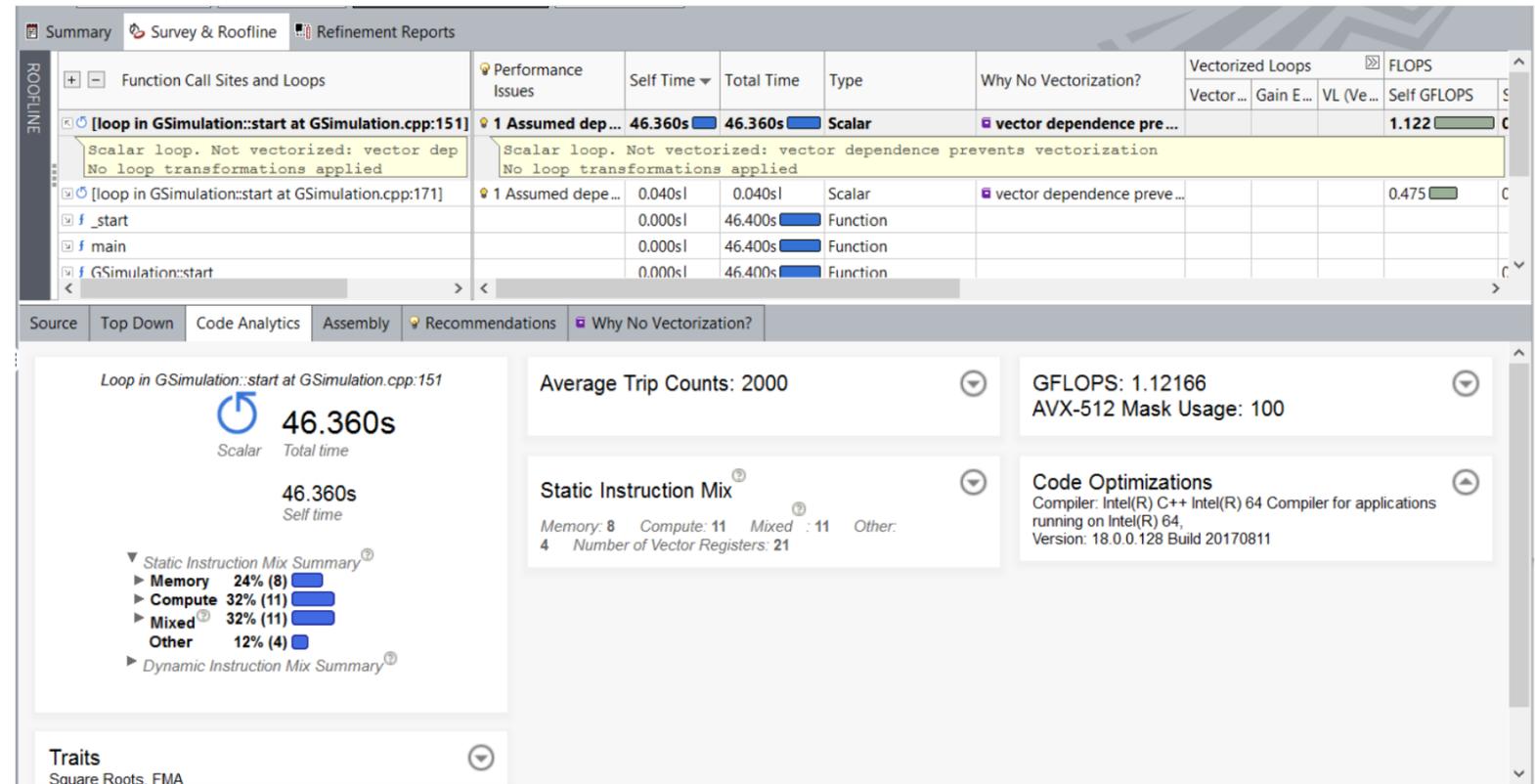
```
struct Particle
{
  public:
  ...
  real_type pos[3];
  real_type vel[3];
  real_type acc[3];
  real_type mass;
};
```

```
struct ParticleSoA
{
  public:
  ...
  real_type *pos_x,*pos_y,*pos_z;
  real_type *vel_x,*vel_y,*vel_z;
  real_type *acc_x,*acc_y,*acc_z;
  real_type *mass;
};
```



Performance After Data Structure Change

- In this new version (version 3 in github sample) we introduce the following change:
- Change particle data structures from AOS to SOA
- Note changes in report:
- Performance is lower
- Main loop is no longer vectorized
- Assumed vector dependence prevents automatic vectorization



Next step is clear: perform a **Dependencies** analysis

Dependencies Analysis (Refinement)

```
aprun -n 1 -N 1 advixe-cl --collect dependencies --project-dir ./adv_res \  
--search-dir src:=./ --search-dir bin:=./ -- ./nbody.x
```

Summary Survey & Roofline Refinement Reports Dependencies Source: GSImulation.cpp

Site Location	Loop-Carried Dependencies	Strides Distribution	Access Pattern	Max. Site Footprint	Site Name	Recommendations
[loop in start at GSImulation.cpp:157]	RAW:4	No information available	No information available	No information available	loop_site_1	1 Proven (real) dependency present

Memory Access Patterns Report Dependencies Report Recommendations

Problems and Messages

ID	Type	Site Name	Sources	Modules	State
P1	Parallel site information	loop_site_1	GSImulation.cpp	nbody.x	✓ Not a problem
P3	Read after write dependency	loop_site_1	GSImulation.cpp	nbody.x	New
P4	Read after write dependency	loop_site_1	GSImulation.cpp; main.cpp	nbody.x	New
P5	Read after write dependency	loop_site_1	GSImulation.cpp	nbody.x	New
P6	Read after write dependency	loop_site_1	GSImulation.cpp	nbody.x	New

Read after write dependency: Code Locations

ID	Instruction Address	Description	Source	Function	Variable references	Module	State
X3	0x401c85	Parallel site	GSImulation.cpp:157	start		nbody.x	New
X6	0x401cb8, 0x401d17	Read	GSImulation.cpp:164	start	register XMM1	nbody.x	New
X7	0x401d1e	Write	GSImulation.cpp:164	start		nbody.x	New

```
155 real_type distanceInv = 0.0f;  
156  
157 dx = particles->pos_x[j] - particles->pos_x[i]; //1flop  
158 dy = particles->pos_y[j] - particles->pos_y[i]; //1flop  
159 dz = particles->pos_z[j] - particles->pos_z[i]; //1flop  
162 distanceInv = 1.0f / sqrtf(distanceSqr); //1div+1sqrt  
163  
164 particles->acc_x[i] += dx * G * particles->mass[j] * distanceInv * distanceInv * distanceInv; //6flops  
165 particles->acc_y[i] += dy * G * particles->mass[j] * distanceInv * distanceInv * distanceInv; //6flops  
166 particles->acc_z[i] += dz * G * particles->mass[j] * distanceInv * distanceInv * distanceInv; //6flops  
162 distanceInv = 1.0f / sqrtf(distanceSqr); //1div+1sqrt  
163  
164 particles->acc_x[i] += dx * G * particles->mass[j] * distanceInv * distanceInv * distanceInv; //6flops
```

- Dependencies analysis has high overhead:
- Run on reduced workload
- Advisor Findings:
- RAW dependency
- Multiple reduction-type dependencies

Recommendations

Memory Access Patterns Report

Dependencies Report

💡 Recommendations

All Advisor-detectable issues: [C++](#) | [Fortran](#)

Recommendation: Resolve dependency

The Dependencies analysis shows there is a real (proven) dependency in the loop. To fix: Do one of the following:

- If there is an anti-dependency, enable vectorization using the directive `#pragma omp simd safelen(length)`, where `length` is smaller than the distance between dependent iterations in anti-dependency. For example:

```
#pragma omp simd safelen(4)
for (i = 0; i < n - 4; i += 4)
{
    a[i + 4] = a[i] * c;
}
```

- If there is a reduction pattern dependency in the loop, enable vectorization using the directive `#pragma omp simd reduction(operator:list)`. For example:

```
#pragma omp simd reduction(+:sumx)
for (k = 0; k < size2; k++)
{
    sumx += x[k]*b[k];
}
```

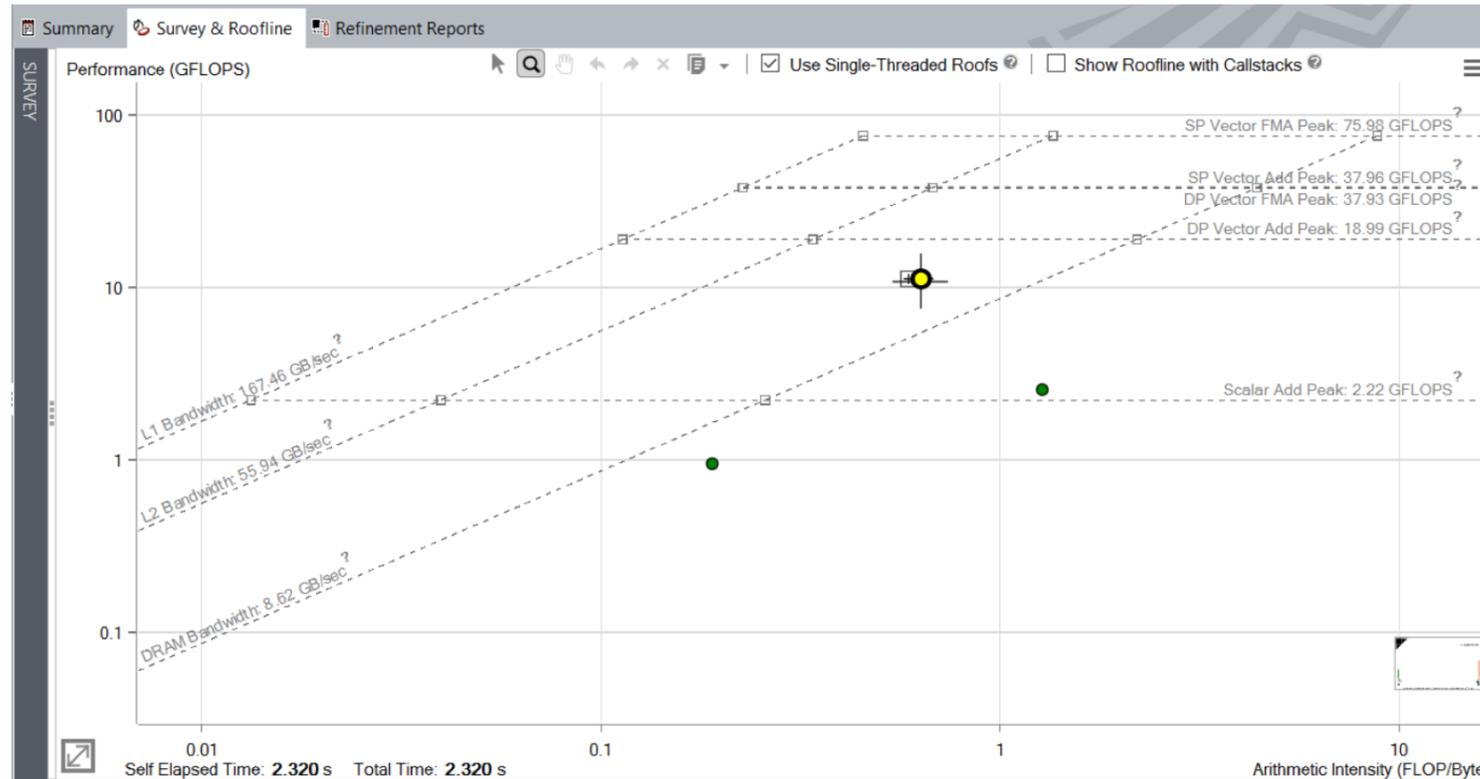
ISSUE: PROVEN (REAL) DEPENDENCY PRESENT

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.



[Resolve dependency](#)

Performance After Resolved Dependencies



New memory access pattern plus vectorization produces much improved performance!

What next?

- Let's explore threading with a suitability analysis.
- Recompile including annotation definitions
- Add headers to file
- Annotate suggested loops
- Run suitability collection

The screenshot displays the Intel Advisor Threading Workflow interface. The left sidebar shows the '2. Annotate Sources' step, which includes instructions for identifying parallel tasks and adding annotations. The main panel shows the 'Summary' report, which includes a message that no source files were found for annotations. Below this, the 'Program metrics' section provides performance data: Elapsed Time (2.43s), Vector Instruction Set (AVX512, AVX2, AVX), Total GFLOP Count (26.12), Total Arithmetic Intensity (0.63431), Number of CPU Threads (1), and Total GFLOPS (10.77). The 'Loop metrics' section is expanded to show 'Top time-consuming loops', with a table listing four loops and their performance metrics.

Loop	Self Time [®]	Total Time [®]	Trip Counts [®]
[loop in GSimulation::start at GSimulation.cpp:143]	0s	2.380s	500
[loop in GSimulation::start at GSimulation.cpp:146]	0.040s	2.360s	2000
[loop in GSimulation::start at GSimulation.cpp:154]	2.320s	2.320s	125
[loop in GSimulation::start at GSimulation.cpp:177]	0.020s	0.020s	2000

Annotating the code

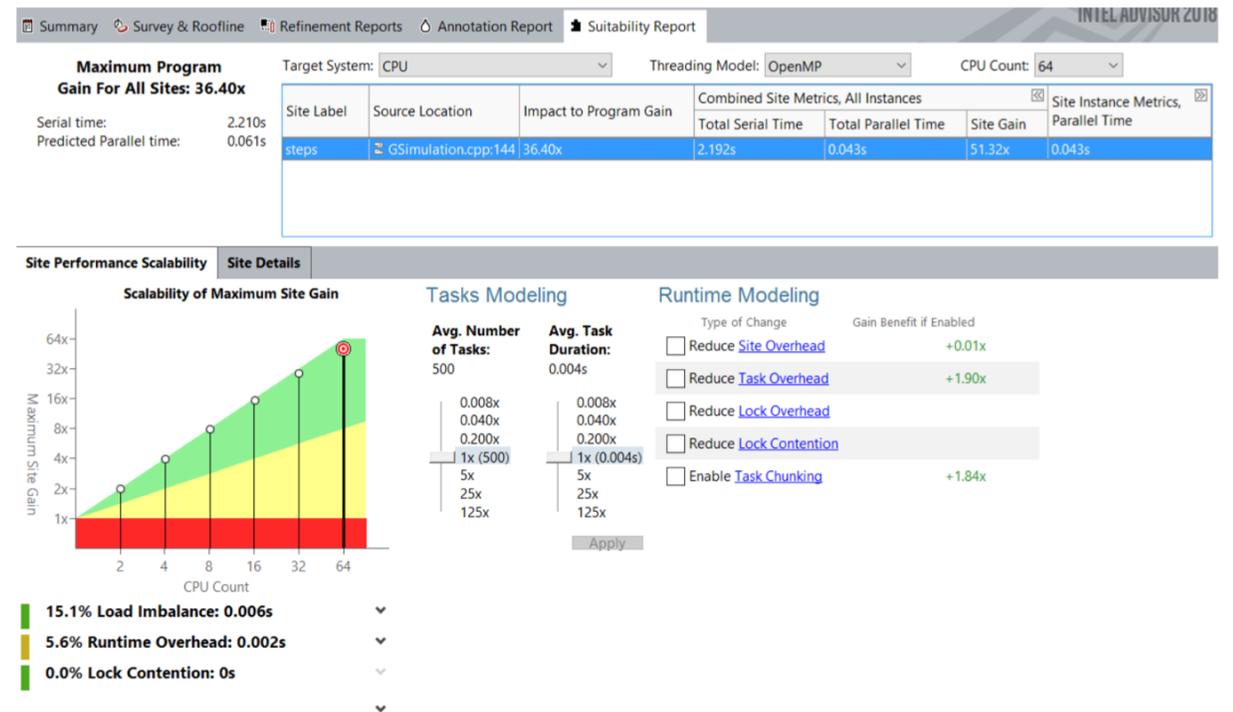
- Add annotations as shown on the left sample
- Complex sites may be analyzed in more detail using task sections if needed
- ANNOTATE_SITE_BEGIN / ANNOTATE_SITE_END
- ANNOTATE_TASK_BEGIN / ANNOTATE_TASK_END
- Recompile including annotation definitions:
 - `-I/opt/intel/advisor/include`
- Collect suitability data

```
#include "advisor-annotate.h"
...
ANNOTATE_SITE_BEGIN(steps)
for (int s=1; s<=get_nsteps(); ++s)
{
    ...
    ANNOTATE_TASK_BEGIN(particles)
    for (i = 0; i < n; i++)
    {
        ...
    }
    ANNOTATE_TASK_END(particles)
}
ANNOTATE_SITE_END(steps)
```

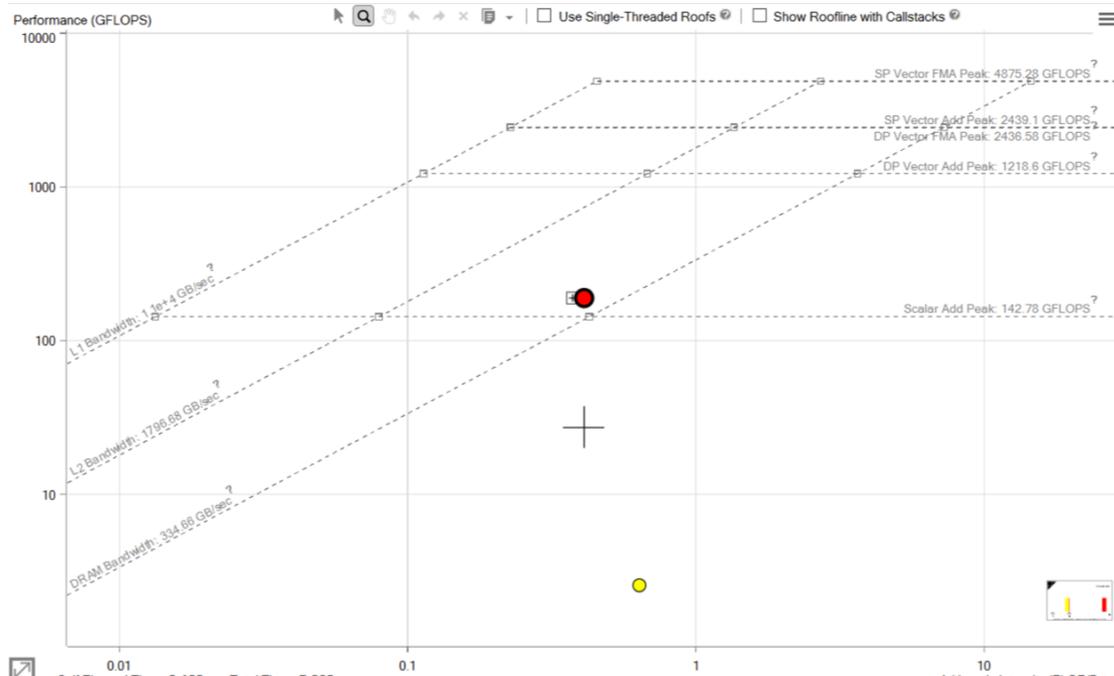
```
mpirun -n 1 -N 1 advixe-cl --collect suitability --project-dir ./adv_res \
--search-dir src:=./ --search-dir bin:=./ -- ./nbody.x
```

Suitability report

- Good speedup expected, but far from ideal (~56% efficiency).
- Modeling shows that increasing the task length would improve efficiency.
- Next step: add omp parallel region to code and re-test



Roofline for Threaded Version



Now using regular roofline, instead of single-threaded

Still room for improvement, but at this point we need additional detail regarding shared resource utilization

```
for (int s=1; s<=get_nsteps(); ++s)
{
    ts0 += time.start();

    #pragma omp parallel for
    for (i = 0; i < n; i++) // update
acceleration
    {
        ...
        real_type ax_i = particles->acc_x[i];
        real_type ay_i = particles->acc_y[i];
        real_type az_i = particles->acc_z[i];

    #pragma omp simd reduction(+:ax_i,ay_i,az_i)
    for (j = 0; j < n; j++)
    {
        real_type dx, dy, dz;
        real_type distanceSqr = 0.0f;
        real_type distanceInv = 0.0f;

        dx = particles->pos_x[j] - particles->pos_x[i];
```

