

# Intel® Advisor Vectorization

Dmitry Tarakanov

Software Technical Consulting Engineer



intel®

# Notices & Disclaimers

Performance varies by use, configuration, and other factors. Learn more at [www.Intel.com/PerformanceIndex](http://www.Intel.com/PerformanceIndex).

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details.

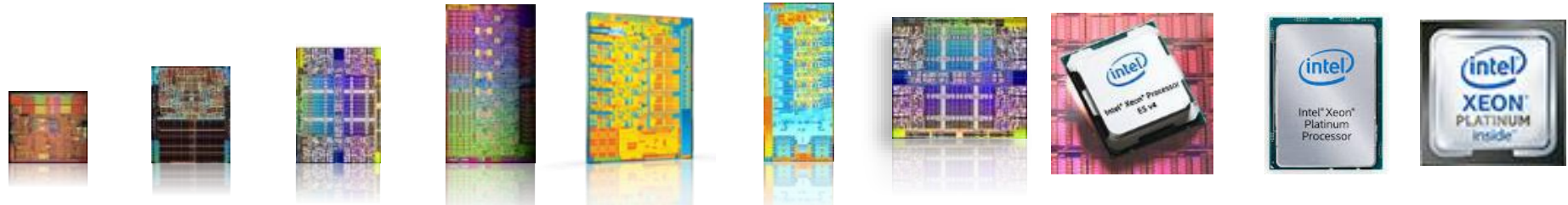
Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

# World is changing: HW and SW change, too!

More Cores → More Threads → Wider Vectors



	Intel® Xeon® Processor								Intel® Xeon® Scalable Processor	
	64-bit	5100 series	5500 series	5600 series	E5-2600	E5-2600 V2	E5-2600 V3	E5-2600 V4	Platinum 8180	Platinum 9282
Core(s)	1	2	4	6	8	12	18	22	28	56
Threads	2	2	8	12	16	24	36	44	56	112
SIMD Width	128	128	128	128	256	256	256	256	512	512

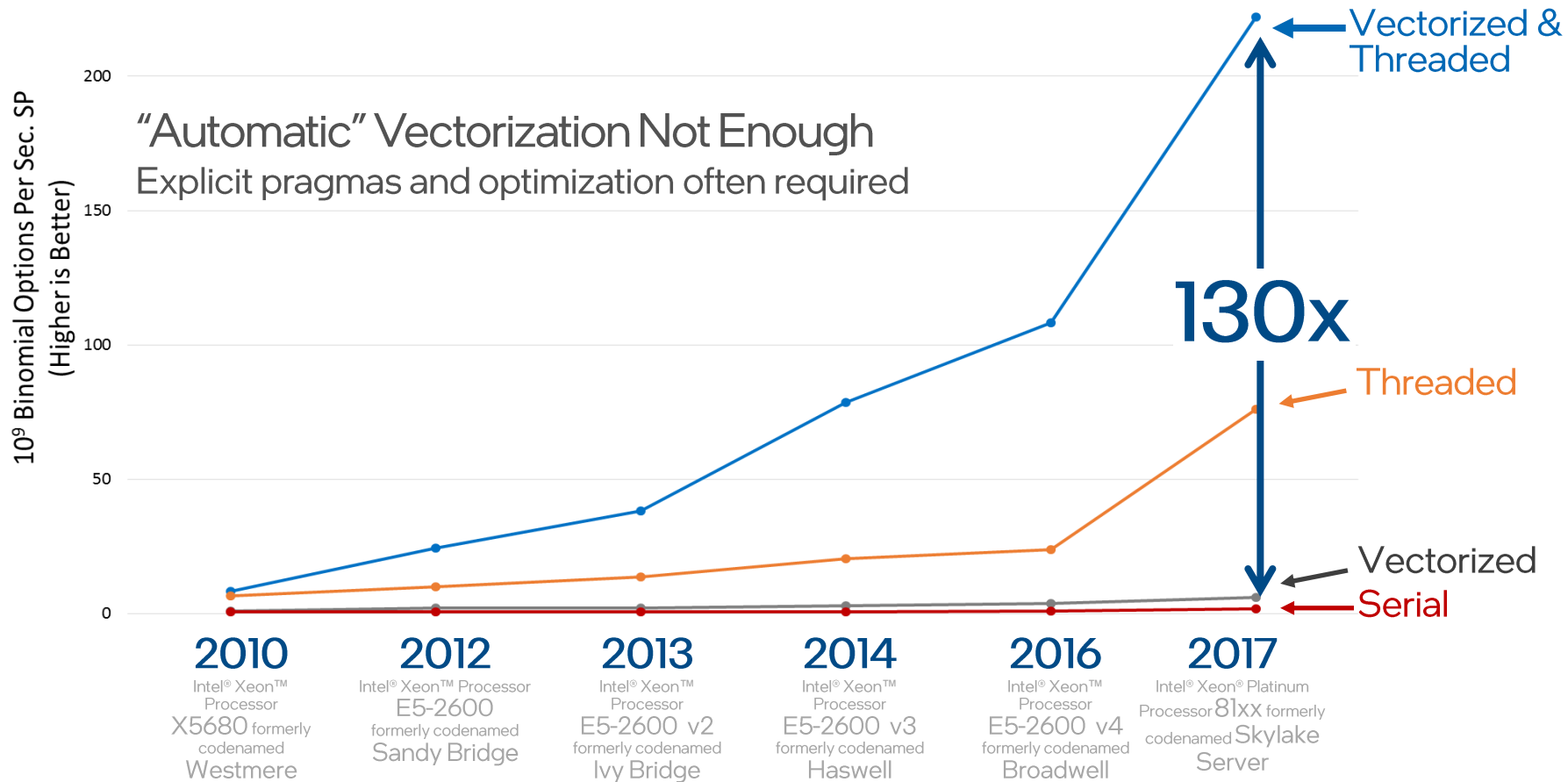
High performance software must be both

- Parallel (multi-thread, multi-process)
- Vectorized

\*Product specification for launched and shipped products available on [ark.intel.com](http://ark.intel.com).

# Intel® Advisor: Vectorize & Thread or Performance Dies

Threaded + Vectorized Can Be Much Faster than Either One Alone



**Testing Date:** Performance results are based on testing by Intel employees as of 2017 and may not reflect all publicly available security updates.

**Configuration Details and Workload Setup:** See [Vectorize & Thread or Performance Dies Configurations for 2010-2016 Benchmarks](#) in Backup.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.

Performance varies by use, configuration, and other factors. Learn more at [www.intel.com/PerformanceIndex](http://www.intel.com/PerformanceIndex). Your costs and results may vary.

# Rich Set of Capabilities for High Performance Code Design

## Intel® Advisor



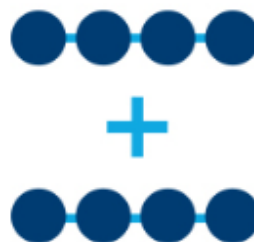
### Offload Modelling

Design offload strategy and model performance on GPU.



### Roofline Analysis

Optimize your application for memory and compute.



### Vectorization Optimization

Enable more vector parallelism and improve its efficiency.



### Thread Prototyping

Model, tune, and test multiple threading designs.



### Build Heterogeneous Algorithms

Create and analyze data flow and dependency computation graphs.

# Get Faster Code Faster! Intel® Advisor Vectorization Optimization

## ■ Have you:

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

## ■ Data Driven Vectorization:

- What vectorization will pay off most?
- What's blocking vectorization? Why?
- Are my loops vector friendly?
- Will reorganizing data increase performance?
- Is it safe to just use pragma simd?

The screenshot shows the Intel Advisor 2019 Vectorization Advisor interface. The top bar indicates 'Elapsed time: 125.72s' and 'Vectorized' status. The 'FILTER' section shows 'All Modules', 'All Sources', 'Loops And Functions', and 'All Threads'. The 'Summary' tab is active, displaying a table of loop performance and vectorization status.

Function Call Sites and Loops	Perfor... Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops				Instruction Set	
						Vect...	Efficiency	Gain...	VL ..		
[loop in main at roofline.cpp:295]		18.538s	18.538s	Vectorized (B...		AVX	~100%	5.34x	4	5.34x	Flo
[loop in main at roofline.cpp:310]		18.394s	18.394s	Vectorized (Bo...		AVX	~100%	5.34x	4	5.34x	Flo
[loop in main at roofline.cpp:221]		14.741s	14.741s	Scalar	novector dire...						Flo
[loop in main at roofline.cpp:234]		11.117s	11.117s	Scalar	inner loop w...						Flo
[loop in main at roofline.cpp:247]		6.967s	6.967s	Vectorized (Bo...		AVX	~31%	1.22x	4	1.22x	Inserts; U...
[loop in main at roofline.cpp:138]		6.949s	6.949s	Scalar	novector dire...						Flo
[loop in main at roofline.cpp:260]		3.285s	3.285s	Vectorized (Bo...		AVX	~100%	5.09x	4	5.09x	Flo
[loop in main at roofline.cpp:199]		2.454s	2.454s	Vectorized (Bo...		AVX	~100%	5.14x	4	5.14x	Flo
[loop in main at roofline.cpp:273]		2.258s	2.258s	Vectorized (Bo...		AVX2	~100%	4.73x	4	4.73x	FMA
[loop in main at roofline.cpp:151]		1.899s	1.899s	Vectorized (Bo...		AVX	~100%	4.80x	4	4.80x	Flo
[loop in main at roofline.cpp:256]	1 Oppo..	0.042s	3.327s	Scalar	inner loop w...						Flo
[loop in main at roofline.cpp:304]		0.040s	18.434s	Scalar	inner loop w...						Flo

"Intel® Advisor's Vectorization Advisor permitted me to focus my work where it really mattered. When you have only a limited amount of time to spend on optimization, it is invaluable."

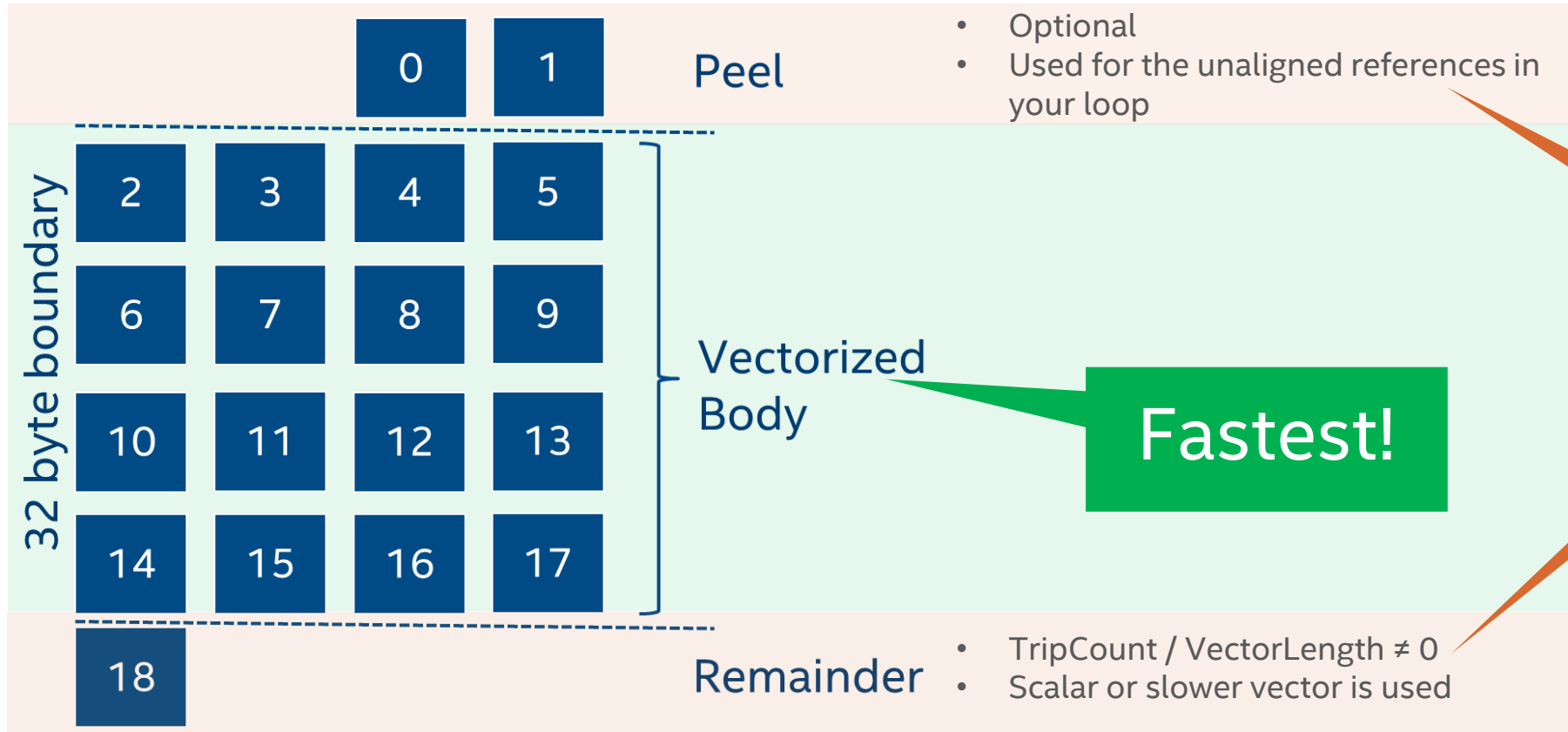
*Gilles Civario*

*Senior Software Architect*

*Irish Centre for High-End Computing*

# Spend your time in the most efficient place!

## A typical vectorized loop consists of...



Less Fast

Fastest!

# The Right Data At Your Fingertips

Get all the data you need for high impact vectorization

Filter vectorized loops

What prevents vectorization?

Function Call Sites and Loops	Performance Issues	CPU Time		Type	Why No Vectorization?	Vectorized Loops				
		Total Time	Self Time			Vector...	Efficiency	Gain E...	VL	
[loop in serial_mandelbrot at mandelbrot.cpp:70]		0.202s	35.3%	0.202s	27.9%	Scalar				
[loop in main\$omp\$parallel@164 at mandelbrot.cpp:181]		0.152s		0.152s		Scalar				
[loop in main\$omp\$parallel@219 at mandelbrot.cpp:237]		0.108s		0.108s		Inside vectorized				
[loop in serial_mandelbrot at mandelbrot.cpp:126]		0.088s		0.088s		Inside vectorized				
[loop in serial_mandelbrot at mandelbrot.cpp:114]	2 Possible ineffi...	0.100s		0.012s	l	Vectorized (Body)	AVX2	67%	2.69x	4
[loop in main\$omp\$parallel@164 at mandelbrot.cpp:169]	1 Data type conv...	0.162s	28.3%	0.010s	l	Scalar				
[loop in serial_mandelbrot at mandelbrot.cpp:58]	1 Data type conv...	0.202s	35.3%	0.000s	l	Scalar				
[loop in serial_mandelbrot at mandelbrot.cpp:57]	1 Data type conv...	0.202s	35.3%	0.000s	l	Scalar				
[loop in serial_mandelbrot at mandelbrot.cpp:112]	1 Data type conv...	0.100s		0.000s	l	Scalar				
[loop in main\$omp\$parallel@164 at mandelbrot.cpp:164]	2 Assumed depe...	0.162s	28.3%	0.000s	l	Threaded (OpenMP)				
[loop in main\$omp\$parallel@164 at mandelbrot.cpp:164]		0.162s	28.3%	0.000s	l	Scalar				
[loop in main\$omp\$parallel@219 at mandelbrot.cpp:225]	1 Data type conv...	0.108s		0.000s	l	Threaded (OpenMP)				
[loop in main\$omp\$parallel@219 at mandelbrot.cpp:225]		0.108s		0.000s	l	Scalar				
[loop in main\$omp\$parallel@219 at mandelbrot.cpp:225]	2 Possible ineffi...	0.108s		0.000s	l	Vectorized (Body)	AVX2	4	1.79x	4

Focus on hot loops

What vectorization issues do I have?

Which Vector instructions are used?

How efficient is the code?

Get Faster Code Faster!



# 5 Steps to Efficient Vectorization

### 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)

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

**Issue: Peeled/Remainder loop(s) present**

All or some 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

### 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	2	1	9	< 0,0001s	1173619
0.010s	3	1	5	< 0,0001s	1312315

### 4. Loop-Carried Dependency Analysis

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.cox1063	RAW:1	No information available	No information available
loop_site_139	runCRawLoops	runCRawLoops.cox622	No information available	39% / 36% / 25%	Mixed strides
loop_site_160	runCRawLoops	runCRawLoops.cox925	No information available	100% / 0% / 0%	All unit strides

ID	Stride	Type	Source	Modules	Alignment
P22	0; 0; 1	Unit stride	runCRawLoops.cox637	lcal.exe	
P23	0; 0	Unit stride	runCRawLoops.cox638	lcal.exe	
P30	-1575; -63; -26; -25; -1; 0; 1; 25; 26; 63; 2164801	Variable stride	runCRawLoops.cox628	lcal.exe	

# 1. Compiler diagnostics + Performance Data + SIMD efficiency information

## Efficiently Vectorize your code

Elapsed time: 0.50s | Vectorized | Not Vectorized | Filter: All Modules | mandelbrot.cpp | Loops | All Threads

Summary | Survey & Roofline | Refinement Reports

Function Call Sites and Loops	Performance Issues	CPU Time		Type	Why No Vectorization?
		Total Time	Self Time		
[loop in simd_mandelbrot at mandelbrot.cpp:126]		0.088s	0.088s	Inside vectorized	
[loop in simd_mandelbrot at mandelbrot.cpp:114]	2 Possible ineffici...	0.100s	0.012s	Vectorized (Body)	
[loop in main\$omp\$parallel@164 at mandelbrot.cpp:169]	1 Data type conv...	0.162s	28.3%	Scalar	outer loop was not auto-vectorized: consider using SIMD directive
[loop in serial_mandelbrot at mandelbrot.cpp:58]	1 Data type conv...	0.202s	35.3%	Scalar	outer loop was not auto-vectorized: consider using SIMD directive
[loop in serial_mandelbrot at mandelbrot.cpp:57]	1 Data type con ...	0.202s	0.000s	Scalar	outer loop was not auto-vectorized: consider using SIMD directive
[loop in simd_mandelbrot at mandelbrot.cpp:112]	1 Data type conv...	0.100s	0.000s	Scalar	inner loop was already vectorized

Source | Top Down | Code Analytics | Assembly | Recommendations | Why No Vectorization?

File: mandelbrot.cpp:57 serial\_mandelbrot

Line	Source	Total Time	%	Loop/Function Time	%
54	// Traverse the sample space in equally spaced steps with width + height				
55	// samples				
56	//#pragma omp simd // vectorize code				
57	for (int j = 0; j < height; ++j) { <div style="border: 1px solid black; padding: 5px; margin-top: 5px;">                     [loop in serial_mandelbrot at mandelbrot.cpp:57]                      Scalar loop. Outer loop was not auto-vectorized: consider using SIMD directive                      No loop transformations applied                 </div>			202,006.000usec	

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

# Get Specific Advice For Improving Vectorization

The screenshot displays the Intel Advisor interface for a C++ application named 'mandelbrot.cpp'. The top navigation bar shows 'Elapsed time: 0.50s', 'Vectorized' (checked), 'Not Vectorized' (unchecked), and filters for 'All Modules', 'mandelbrot.cpp', 'Loops', and 'All Threads'. The main table lists performance issues with columns for CPU Time (Total and Self), Type, and Why No Vectorization?.

Function Call Sites and Loops	Performance Issues	CPU Time		Type	Why No Vectorization?
		Total Time	Self Time		
[loop in serial_mandelbrot at mandelbrot.cpp:70]		0.202s 35.3%	0.202s 27.9%	Scalar	loop control variable was not identified
[loop in main\$omp\$parallel@164 at mandelbrot.cpp:169]		0.152s	0.152s	Scalar	loop control variable was not identified
[loop in main\$omp\$parallel@164 at mandelbrot.cpp:169]		0.108s	0.108s	Inside vectorized	
[loop in simd_mandelbrot at mandelbrot.cpp:126]		0.088s	0.088s	Inside vectorized	
[loop in simd_mandelbrot at mandelbrot.cpp:114]	2 Possible inefficient memory access patterns present	0.100s	0.012s	Vectorized (Body)	
[loop in main\$omp\$parallel@164 at mandelbrot.cpp:169]	1 Data type conversions present	0.162s 28.3%	0.010s	Scalar	outer loop was not auto-vectorized: compiler could not find a common loop control variable

A blue callout box points to the '2 Possible inefficient memory access patterns present' issue with the text 'Click to see recommendation'. Another blue callout box points to the 'Recommendations' tab with the text 'Advisor shows hints on issue fix'.

**Recommendations:**

- 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.  
**Confirm inefficient memory access patterns**  
There is no confirmation inefficient memory access patterns are present. To fix: Run a [Memory Access Patterns analysis](#).
- Data type conversions present**  
There are multiple data types within loops. Utilize hardware vectorization support more effectively by avoiding data type conversion.  
**Use the smallest data type**  
The source loop contains data types of different widths. To fix: Use the smallest data type that gives the needed precision to use the entire vector register width.

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

Identify how many times the loop executes & collect loop trip counts data

Function Call Sites and Loops	Performance Issues	CPU Time				Trip Counts	
		Total Time	Self Time	Total Elapsed...	Self Elapsed ...	Average	Call Count
[loop in main\$omp\$parallel@219 at mandelbrot.cpp:237]		0.108s	0.108s	0.020s	0.020s	30	524288
[loop in simd_mandelbrot at mandelbrot.cpp:126]		0.088s	0.088s	0.088s	0.088s	30	524288
[loop in simd_mandelbrot at mandelbrot.cpp:114]	2 Possible ineffi...	0.100s	0.012s	0.100s	0.012s	512	1024
[loop in main\$omp\$parallel@164 at mandelbrot.cpp:169]	1 Data type conv ...	0.162s	0.010s	0.022s	0.010s	2048	1024
[loop in serial_mandelbrot at mandelbrot.cpp:58]	1 Data type conv ...	0.202s	0.000s	0.202s	0s	2048	1024
[loop in serial_mandelbrot at mandelbrot.cpp:57]	1 Data type conv ...	0.202s	0.000s	0.202s	0s	1024	1

Check actual trip counts

Not enough to know the time spent in a loop

Need to know the number of iterations, too

## 4. Loop-Carried Dependency Analysis

# Factors that **prevent** Vectorizing your code

### 1. Loop-carried dependencies

```
DO I = 1, N
  A(I + M) = A(I) + B(I)
ENDDO
```

**M >= SIMDlength?**

#### 1a. Pointer aliasing (compiler-specific)

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

### 2. Function calls (incl. indirect)

```
for (i = 1; i < nx; i++) {
  x = x0 + i * h;
  sumx = sumx + func(x, y, xp);
}
```

### 3. Loop structure, boundary condition

```
struct _x { int d; int bound; };

void doit(int *a, struct _x *x)
{
  for(int i = 0; i < x->bound; i++)
    a[i] = 0;
}
```

### 4. Outer vs. inner loops

```
for(i = 0; i <= MAX; i++) {
  for(j = 0; j <= MAX; j++) {
    D[j][i] += 1;
  }
}
```

### 5. Cost-benefit (compiler specific..)

## 4. Loop-Carried Dependency Analysis

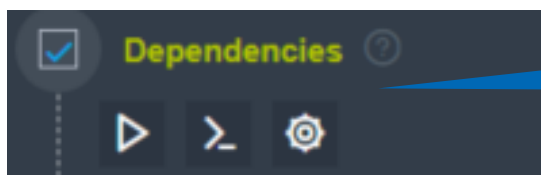
# Is It Safe to Vectorize?

Dependencies Analysis to identify and explore loop-carried dependencies

Function Call Sites and Loops	Performance Issues	CPU Time		Type	Why No Vectorization?
		Total Time	Self Time		
[loop in serial_mandelbrot at mandelbrot.cpp:58]	1 Data type conv...	0.202s 35.3%	0.000s	Scalar	outer loop was not auto-vectorized: consider u
[loop in serial_mandelbrot at mandelbrot.cpp:57]	1 Data type conv...	0.202s 35.3%	0.000s	Scalar	outer loop was not auto-vectorized: consider u
[loop in simd_mandelbrot at mandelbrot.cpp:112]	1 Data type conv...	0.100s	0.000s	Scalar	inner loop was already vectorized
[loop in main\$omp\$parallel@164 at mandelbrot.cpp:164]	2 Assumed dep ...	0.162s	0.000s	Threaded (Open ...	vector dependence prevents vectorization
[loop in main\$omp\$parallel@164 at mandelbrot.cpp:164]		0.162s 28.3%	0.000s	Scalar	loop control variable was not identified. Explic
[loop in main\$omp\$parallel@219 at mandelbrot.cpp:225]	Data type conv...	0.108s	0.000s	Threaded (Open ...	inner loop was already vectoriz

Select loop and run Dependency Analysis

Vector Dependence prevents Vectorization



## 5. Memory Access Patterns Analysis

# Factors that **slow-down** your Vectorized code

### 1a. Indirect memory access

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

### 1b. Memory sub-system Latency / Throughput

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

### 2. Serialized or "sub-optimal" function calls

```
for (i = 1; i < nx; i++) {  
    sumx = sumx +  
    serialized_func_call(x, y, xp);  
}
```

2.19x

Vectorization Gain

~55%

Vectorization Efficiency

Why 45%  
lost?

Run MAP analysis

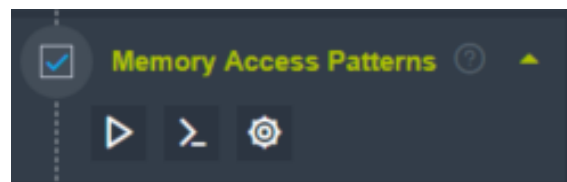
### 3. Small trip counts not multiple of VL

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

### 4. Branchy codes, outer vs. inner loops

```
for(i = 0; i <= MAX; i++) {  
    if ( D[i] < N)  
        do_this(D);  
    else if (D[i] > M)  
        do_that();  
    //...  
}
```

### 5. MANY others: spill/fill, floating-point accuracy trade-offs, FMA, DIV/SQRT, Unrolling

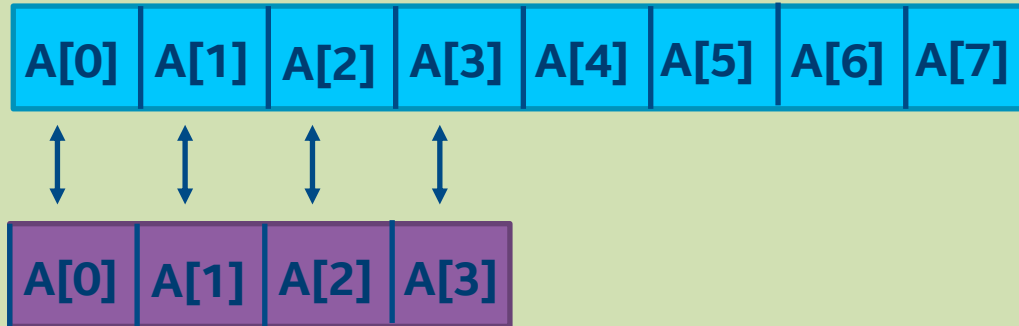


## 5. Memory Access Patterns Analysis

Memory  
access  
patterns

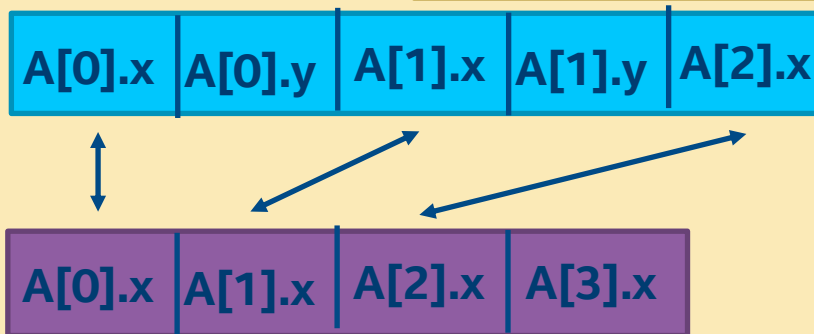
Unit strided (contiguous):

Efficient



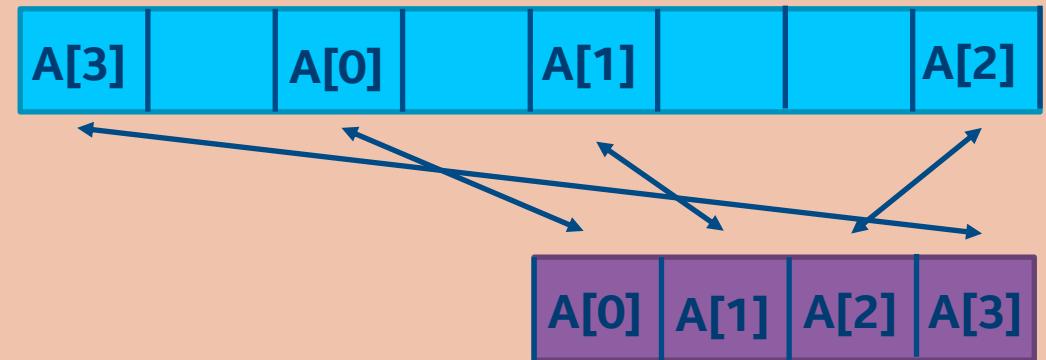
Constant strided:

Less efficient



Arbitrary access:

Inefficient





# Vectorization Accuracy Levels

Comparison / Accuracy Level	Low	Medium	High
<b>Overhead</b>	1.1x	5 - 8x	10 - 40x
<b>Goal</b>	Get basic insights about how well your application is vectorized and how you can improve vectorization efficiency	Get more insights about how well your application is vectorized and the number of iterations in loops/functions	Get detailed insights about your application performance, including performance issues and detailed optimization recommendations
<b>Analyses</b>	Survey	Survey + Characterization (Trip Counts)	Survey + Characterization (Trip Counts, FLOP, Call Stacks) + Memory Access Patterns
<b>Result</b>	Basic Survey report	Survey report extended with trip count data	Extended Survey report with trip counts and floating-point and integer operations (FLOP and INTOP)  Memory Access Patterns with memory traffic data and memory usage issues

# Vectorization Lab – Prepare Data

## 1. Build C++ application

```
cd ./base && make
```

## 2. Run Survey analysis to find hotspots and get performance data for your application

```
advisor --collect=survey --project-dir=./advisor_results -- ./release/Mandelbrot
```

## 3. Collect more detailed data

### i. Determine the number of loop iterations and collect data about floating-point and integer operations

```
advisor --collect=tripcounts --flop --project-dir=./advisor_results  
-- ./release/Mandelbrot
```

### ii. Get IDs and locations of loops

```
advisor --report=survey --project-dir=./advisor_results  
-- ./release/Mandelbrot
```

### iii. Mark up loops for deeper analysis (e.g. 2 scalar loops)

```
advisor --mark-up-loops --select=mandelbrot.cpp:57,mandelbrot.cpp:69  
--project-dir=./advisor_results -- ./release/Mandelbrot
```

### iv. Check for possible dependencies

```
advisor --collect=dependencies --project-dir=./advisor_results  
--search-dir src:r=./src -- ./release/Mandelbrot
```

### v. Check memory access patterns

```
advisor --collect=map --project-dir=./advisor_results  
--search-dir src:r=./src -- ./release/Mandelbrot
```

# Vectorization Lab – Analyze Results (Serial)

Check details on the loops of interest

- Dependencies? – No. Can vectorize!

  - Refinement Analysis Data

These loops were analyzed for memory access patterns and dependencies:

Site Location

loop in [serial\\_mandelbrot](#) at [mandelbrot.cpp:60](#)

loop in [serial\\_mandelbrot](#) at [mandelbrot.cpp:71](#)

Dependencies

No dependencies found

No dependencies found

- Vectorized? – No. Try to vectorize!

Function Call Sites and Loops	Performance Issues	CPU Time		Type	Why No Vectorization?
		Total Time	Self Time		
<input checked="" type="checkbox"/> [loop in serial_mandelbrot at mandelbrot.cpp:69]	<input checked="" type="checkbox"/>	0.180s	0.180s 85.7%	Scalar	<input checked="" type="checkbox"/> loop control variable was not identified. Explicitly compute the iteration count
<input checked="" type="checkbox"/> serial_mandelbrot	<input type="checkbox"/>	0.180s	0.000s	Inlined Function	
<input checked="" type="checkbox"/> [loop in serial_mandelbrot at mandelbrot.cpp:57]	<input checked="" type="checkbox"/> 1 Data type con ...	0.180s	0.000s	Scalar	<input checked="" type="checkbox"/> outer loop was not auto-vectorized: consider using SIMD directive
<input checked="" type="checkbox"/> [loop in serial_mandelbrot at mandelbrot.cpp:56]	<input type="checkbox"/> 1 Data type conv ...	0.180s	0.000s	Scalar	<input checked="" type="checkbox"/> outer loop was not auto-vectorized: consider using SIMD directive

# Vectorization Lab – Vectorize

Run Advisor for SIMD implementation (with `#pragma omp simd` used) of application

Function Call Sites and Loops	Performance Issues	CPU Time		Type	Why No Vectorization?	Vectorized Loops				
		Total Time	Self Time			Vector...	Efficiency	Gain E...	VL (	
[loop in simd_mandelbrot at mandelbrot.cpp:125]		0.082s	91.2%	0.082s	63.1%	Inside vectorized				
[loop in simd_mandelbrot at mandelbrot.cpp:113]	1 Data type con...	0.090s		0.008s		Vectorized (Body)	AVX2	67%	2.69x	4
f simd_mandelbrot	1 Data type conv...	0.090s	100.0%	0.000s		Inlined Function				
[loop in simd_mandelbrot at mandelbrot.cpp:111]	1 Data type conv...	0.090s	100.0%	0.000s		Scalar				

`mandelbrot.cpp:113` loop is vectorized

Total Time is 2 times less than in scalar case

intel®