



VECTORIZATION CASE STUDY

INTEL[®] ADVISOR – VECTORIZATION ADVISOR

Mike Voss, Principal Engineer, Intel

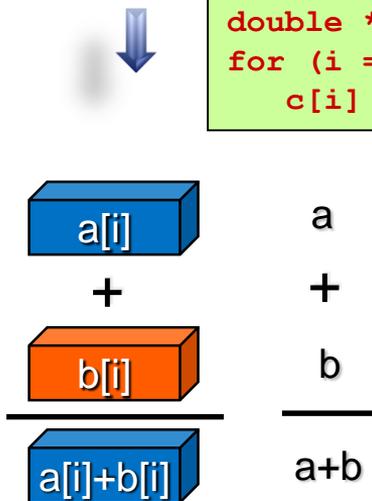
Special thanks to Alex Shinsel (Consulting Engineer, Intel)

SIMD => Single Instruction Multiple Data

VLP / Vectorization

- **Scalar**

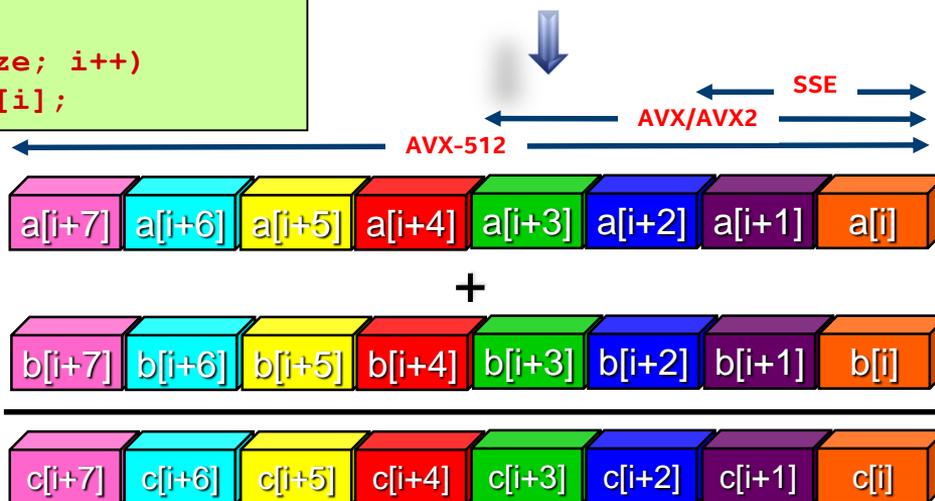
- one instruction produces one result



- **SIMD processing**

- one instruction can produce multiple results (SIMD)
- e.g. `vaddpd` / `vaddps` (p => packed)

```
double *a,*b,*c; ...  
for (i = 0; i < size; i++)  
    c[i] = a[i] + b[i];
```



Outline from Previous Lectures on Vectorization

- What is vectorization and why is it important
- The different ways we can vectorize our code
- The two main challenges in vectorization
 - Determining that vectorization is legal (the results will be the same)
 - Dependence analysis
 - Obstacles to vectorization and how to deal with them
 - Optimizing performance
 - Memory issues (alignment, layout)
 - Telling the compiler what you know (about your code & about your platform)
- Using compiler intrinsics
- Using OpenMP* simd pragmas
- A case study

In previous lectures on vectorization:

Example of Optimization Report - 1

```
$ gcc -c -qopt-report -qopt-report-phase=vec -qopt-report-file=stderr foo.cpp

LOOP BEGIN at foo.cpp(3)
<Peeled loop for vectorization, Multiversioned >
LOOP END

LOOP BEGIN at foo.cpp(5)
<Multi-versioned>
remark #15300: vectorization support: reference theta[] has aligned access [foo.cpp(5)]
remark #15301: vectorization support: reference theta[] has aligned access [foo.cpp(6)]
remark #15302: vectorization support: vector length 4
remark #15303: vectorization support: normalized vectorization overhead 0.014
remark #15304: vectorization support: normalized vectorization overhead 0.014
remark #15305: vectorization support: number of FP up converts: single precision to double precision 1 [foo.cpp(5)]
remark #15306: vectorization support: number of FP down converts: double precision to single precision 1 [foo.cpp(5)]
remark #15307: vectorization support: normalized vectorization overhead 0.014
LOOP END

LOOP BEGIN at foo.cpp(10)
<Multi-versioned>
remark #15308: vectorization support: reference theta[] has aligned access [foo.cpp(10)]
remark #15309: vectorization support: reference theta[] has aligned access [foo.cpp(11)]
remark #15310: vectorization support: vector length 4
remark #15311: vectorization support: normalized vectorization overhead 0.014
remark #15312: vectorization support: normalized vectorization overhead 0.014
remark #15313: vectorization support: number of FP up converts: single precision to double precision 1 [foo.cpp(10)]
remark #15314: vectorization support: number of FP down converts: double precision to single precision 1 [foo.cpp(10)]
remark #15315: scalar cost summary ---
remark #15316: -- begin vector cost summary ---
remark #15317: vector cost: 100
remark #15318: estimated potential speedup 2,730
remark #15319: vectorized math library calls 1
remark #15320: vector cost summary ---
LOOP END

#include <cmath>
void foo (float * theta, float * sth, int count) {
  for (int i = 0; i < count; i++)
    sth[i] = sin(theta[i])*3.1415927f;
}

• Note multiversioning
```



Example of New Optimization Report - 2

```
$ gcc -c -qopt-report -qopt-report-phase=vec -qopt-report-file=stderr foo.cpp

LOOP BEGIN at foo.cpp(3)
<Peeled loop for vectorization>
LOOP END

LOOP BEGIN at foo.cpp(5)
remark #15308: vectorization support: reference theta[] has aligned access [foo.cpp(5)]
remark #15309: vectorization support: reference theta[] has aligned access [foo.cpp(6)]
remark #15310: vectorization support: vector length 4
remark #15311: vectorization support: normalized vectorization overhead 0.014
remark #15312: vectorization support: normalized vectorization overhead 0.014
remark #15313: vectorization support: number of FP up converts: single precision to double precision 1 [foo.cpp(5)]
remark #15314: vectorization support: number of FP down converts: double precision to single precision 1 [foo.cpp(5)]
remark #15315: scalar cost summary ---
remark #15316: -- begin vector cost summary ---
remark #15317: vector cost: 100
remark #15318: estimated potential speedup 2,730
remark #15319: vectorized math library calls 1
remark #15320: vector cost summary ---
LOOP END

#include <cmath>
void foo (float * theta, float * sth, int count) {
  #pragma omp simd
  for (int i = 0; i < count; i++)
    sth[i] = sin(theta[i])*3.1415927f;
}

• OMP SIMD take care of multiversioning
• Next focus on FP converts
```

Example of New Optimization Report - 3

```
$ gcc -c -qopt-report -qopt-report-phase=vec -qopt-report-file=stderr foo.cpp

LOOP BEGIN at foo.cpp(3)
<Peeled loop for vectorization>
LOOP END

LOOP BEGIN at foo.cpp(5)
remark #15308: vectorization support: reference theta[] has aligned access [foo.cpp(5)]
remark #15309: vectorization support: reference theta[] has aligned access [foo.cpp(6)]
remark #15310: vectorization support: vector length 4
remark #15311: vectorization support: normalized vectorization overhead 0.014
remark #15312: vectorization support: normalized vectorization overhead 0.014
remark #15313: vectorization support: number of FP up converts: single precision to double precision 1 [foo.cpp(5)]
remark #15314: vectorization support: number of FP down converts: double precision to single precision 1 [foo.cpp(5)]
remark #15315: scalar cost summary ---
remark #15316: -- begin vector cost summary ---
remark #15317: vector cost: 100
remark #15318: estimated potential speedup 5,190
remark #15319: vectorized math library calls 1
remark #15320: vector cost summary ---
LOOP END

#include <cmath>
void foo (float * theta, float * sth, int count) {
  #pragma omp simd
  for (int i = 0; i < count; i++)
    sth[i] = sin(theta[i])*3.1415927f;
}

• FPI takes care of FP converts
• Next focus on vector length 4 (using SSE)
```



Example of New Optimization Report - 4

```
$ gcc -c -xCORE-AVX2 -qopt-report -qopt-report-phase=vec -qopt-report-file=stderr foo.cpp

LOOP BEGIN at foo.cpp(3)
<Peeled loop for vectorization>
LOOP END

LOOP BEGIN at foo.cpp(5)
remark #15308: vectorization support: reference theta[] has unaligned access [foo.cpp(5)]
remark #15309: vectorization support: reference theta[] has unaligned access [foo.cpp(6)]
remark #15310: vectorization support: vector length 8
remark #15311: vectorization support: normalized vectorization overhead 0.175
remark #15312: vectorization support: normalized vectorization overhead 0.175
remark #15313: OpenMP SIMD LOOP WAS VECTORIZED
remark #15314: entire loop may be executed in remainder
remark #15315: unmarked unaligned unit stride loads 1
remark #15316: unmarked unaligned unit stride stores 1
remark #15317: scalar cost: 100
remark #15318: vector cost: 5000
remark #15319: estimated potential speedup 7,780
remark #15320: vectorized math library calls 1
remark #15321: -- end vector cost summary ---
LOOP END

#include <cmath>
void foo (float * theta, float * sth, int count) {
  #pragma omp simd
  for (int i = 0; i < count; i++)
    sth[i] = sin(theta[i])*3.1415927f;
}

• CORE-AVX2 target takes vector length to 8
• Next focus on data alignment
```



Example of New Optimization Report - 5

```
$ gcc -c -xCORE-AVX2 -qopt-report -qopt-report-phase=vec -qopt-report-file=stderr foo.cpp

LOOP BEGIN at foo.cpp(3)
<Peeled loop for vectorization>
LOOP END

LOOP BEGIN at foo.cpp(5)
remark #15308: vectorization support: reference theta[] has aligned access [foo.cpp(5)]
remark #15309: vectorization support: reference theta[] has aligned access [foo.cpp(6)]
remark #15310: vectorization support: vector length 8
remark #15311: vectorization support: normalized vectorization overhead 0.013
remark #15312: OpenMP SIMD LOOP WAS VECTORIZED
remark #15313: entire loop may be executed in remainder
remark #15314: unmarked unaligned unit stride loads 1
remark #15315: unmarked unaligned unit stride stores 1
remark #15316: scalar cost: 100
remark #15317: vector cost: 3,870
remark #15318: estimated potential speedup 9,730
remark #15319: vectorized math library calls 1
remark #15320: vector cost summary ---
LOOP END

#include <cmath>
void foo (float * theta, float * sth, int count) {
  #pragma omp simd aligned(theta, sth, 64)
  for (int i = 0; i < count; i++)
    sth[i] = sin(theta[i])*3.1415927f;
}

• OMP aligned clause helps
• Overall speedup 2.73x → 9.73x
```



USING INTEL[®] ADVISOR TO ASSIST WITH VECTORIZATION

Based on a presentation by Alex Shinsel

Vectorization
Workflow

Threading
Workflow

OFF Batch mode

Run Roofline

▶ Collect  

1. Survey Target

▶ Collect  

1.1 Find Trip Counts and FL...

▶ Collect  

Trip Counts

FLOPS

Mark Loops for Deeper Anal...

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

-- There are no marked loops --

2.1 Check Dependencies

▶ Collect  

-- Nothing to analyze --

2.2 Check Memory Access P...

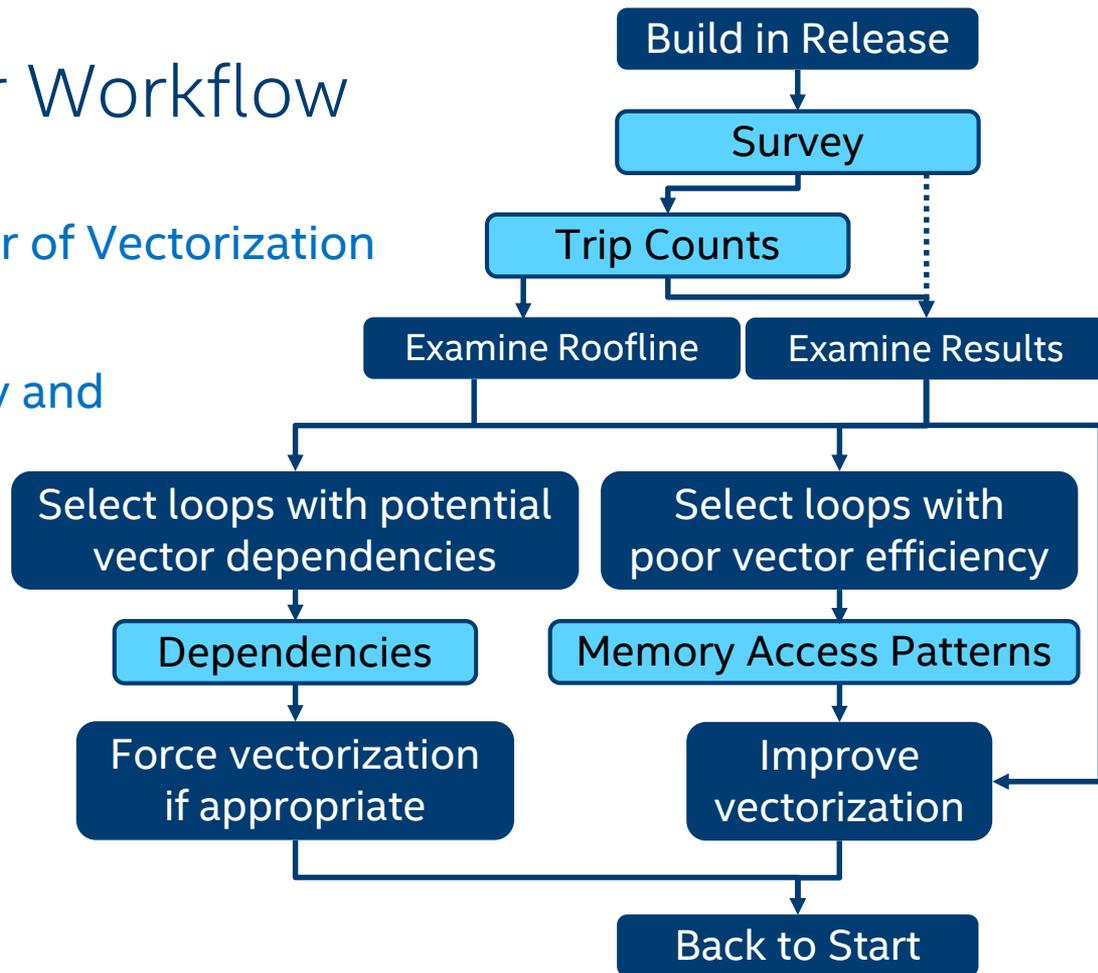
▶ Collect  

 Re-finalize Survey

VECTORIZATION ADVISOR & ROOFLINE

Vectorization Advisor Workflow

- **Survey** is the bread and butter of Vectorization Advisor! All else builds on it!
- **Trip Counts** adds onto Survey and enables the **Roofline**.
- **Dependencies** determines whether it's safe to force a scalar loop to vectorize.
- **Memory Access Patterns** diagnoses vectorization inefficiency caused by poor memory striding.



What Am I Looking At?

The screenshot shows the Intel Advisor 2019 interface with several callouts pointing to specific features:

- Workflow:** Points to the 'Vectorization Workflow' and 'Threading Workflow' tabs.
- Remove Filters:** Points to the 'Remove Filters' button in the top toolbar.
- Loop Display Toggle Buttons:** Points to the 'Vectorized' and 'Not Vectorized' buttons.
- Smart Mode (Does not work for Threading Advisor):** Points to the 'Smart Mode' dropdown menu.
- Search:** Points to the search icon in the top right.
- Filter by origin and type:** Points to the filter dropdowns: 'All Modules', 'All Sources', 'Loops And Functions', and 'All Threads'.
- Report tabs:** Points to the 'Summary', 'Survey & Roofline', 'Refinement Reports', 'Annotation Report', and 'Suitability Report' tabs.
- Primary Pane:** Points to the main table displaying function call sites and loops.
- Secondary Pane tabs:** Points to the 'Source', 'Top Down', 'Code Analytics', 'Assembly', 'Assistance', 'Recommendations', and 'Why No Vectorization?' tabs.
- Secondary Pane:** Points to the detailed view of the selected loop.

Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Type	Why No Vectoriz.
setQueen		2.202s	38.480s	Function	
[loop in setQueen at nqueens_serial.cpp:116]		0.297s	33.526s	Scalar	loop wit
setQueen		0.000s	5.250s	Function	

Function Call Sites and Loops	Total Time %	Total Time	Self Time	Type	Why No
main	100.0%	5.250s	0.000s	Function	
resolve	100.0%	5.250s	0.000s	Function	
[loop resolve at nqueens_serial.cpp:140]	100.0%	5.250s	0.000s	Scalar	loop wit
setQueen	100.0%	5.250s	0.000s	Function	

Survey

Vectorization Advisor

Tip:

For vectorization, you generally only care about loops. Set the type dropdown to "Loops".

Function/Loop Icons

-  Scalar Function
-  Vector Function
-  Scalar Loop
-  Vector Loop

Vectorizing a loop is usually best done on innermost loops. Since it effectively divides duration by vector length, you want to target loops with high self time.

Efficiency is important!

$$\text{Efficiency} = 100\% \frac{\text{Speedup}}{\text{Vec. Length}}$$

The black arrow is 1x. Gray means you got less than that. Gold means you got more. You want to get this value as high as possible!

Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops			
						Vect...	Efficiency	Gain...	VL
 [loop in main at example.cpp:38]	 1 Assumed depend...	0.391s	0.391s	Scalar	 vector dependen...				
 [loop in main at example.cpp:64]	 1 Possible inefficien...	0.297s	0.297s	Vector...		AVX2	2%	0.37x	16
 [loop in main at example.cpp:51]	 1 Possible inefficien...	0.094s	0.094s	Vector...	 1 vectorizatio...	AVX2	8%	1.23x	16
 [loop in main at example.cpp:26]		0.030s	0.030s	Vector...		AVX2	100%	7.98x	8
 [loop in main at example.cpp:14]	 3 Assumed depend...	0.000s	0.000s	Scalar	 vector dependen...				
 [loop in main at example.cpp:23]		0.000s	0.030s	Scalar	 inner loop w...				

Expand a vectorized loop to see it split into body, peel, and remainder (if applicable).

Advisor *advises* you on potential vector issues. This is often your cue to run MAP or Dependencies. Click the icon to see an explanation in the bottom pane.

The Intel Compiler embeds extra information that Advisor can report in addition to its sampled data, such as why loops failed to vectorize.

Let's look at an example...

Trip Counts

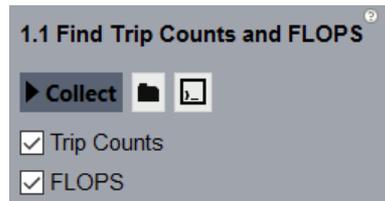
- Trip Counts extends the Survey results. It must be run separately because it has higher overhead that would interfere with timing measurements.
- Vectorization is most effective on inner loops with high iteration counts.
 - It may be beneficial to swap small inner loops and larger outer loops.
 - For maximum performance, iteration counts that are a multiple of the vector length are ideal.
- Trip Counts is useful in diagnosing data alignment and padding problems in loops that traverse multidimensional arrays.
 - In such cases, the trip counts on peel and remainder loops may change as rows/columns push each other out of alignment.

Function Call Sites and Loops	Type	Trip Counts					
		Average	Min	Max	Call Count	Iteratio...	Loop I...
[loop in main at dat...	Vectorized (Body; ...	4374; 5; 4	4374; 1; 1	4375; 7; 7	400000; 34...	< 0.001s	
[loop in main at ...	Vectorized (Body)	4374	4374	4375	400000	< 0.001s	< 0.001s
[loop in main at ...	Remainder	5	1	7	348000	< 0.001s	< 0.001s
[loop in main at ...	Peeled	4	1	7	352000	< 0.001s	< 0.001s

... and FLOPS

Part of the Trip Counts Collection

- Trip Counts and FLOPS are the same collection type, but can be toggled independently using the checkboxes in the workflow or command line flags.
- FLOPS collects information about Floating Point Operations, or FLOPs. This is used with Survey data to calculate **FLOPS, Floating Point Operations Per Second.**
- It also collects some memory data, so it can calculate Arithmetic Intensity.
- **Arithmetic Intensity** is a measurement of **FLOPs/Byte accessed**. This is a trait of the algorithm of a function/loop itself.



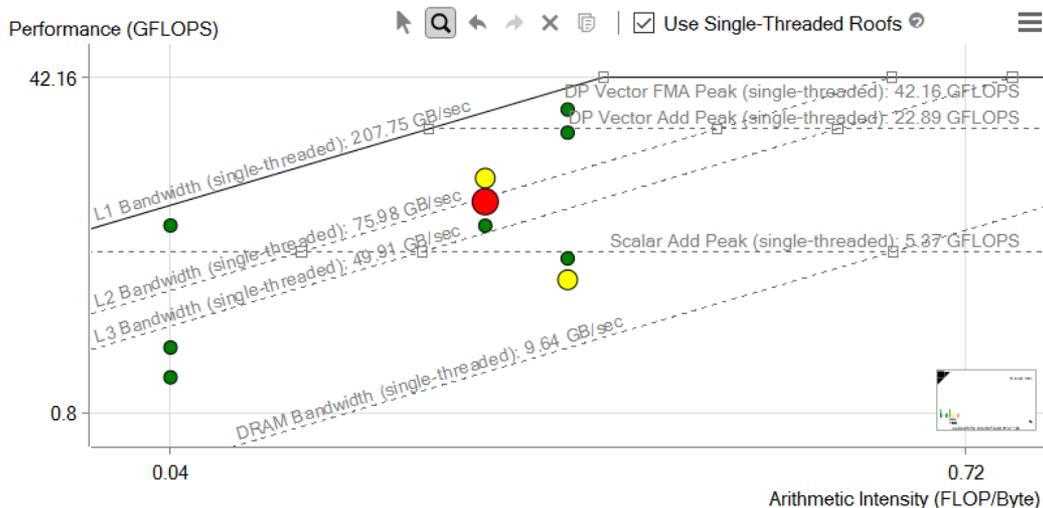
FLOPS					
GFLOPS	AI	GFLOP	Memory, GB	Elapsed Time	Total El...
3.917	0.179	53.120	296.831	13.562s	13.562s
1.756	0.045	13.280	296.831	7.563s	7.563s
7.249	0.134	53.120	395.775	7.328s	7.328s
19.999	0.179	53.120	296.831	2.656s	2.656s
7.264	0.045	13.280	296.831	1.828s	1.828s

Let's look at our example again...

What is a Roofline Chart?

A Roofline Chart plots application performance against hardware limitations.

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



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

Roofline Metrics

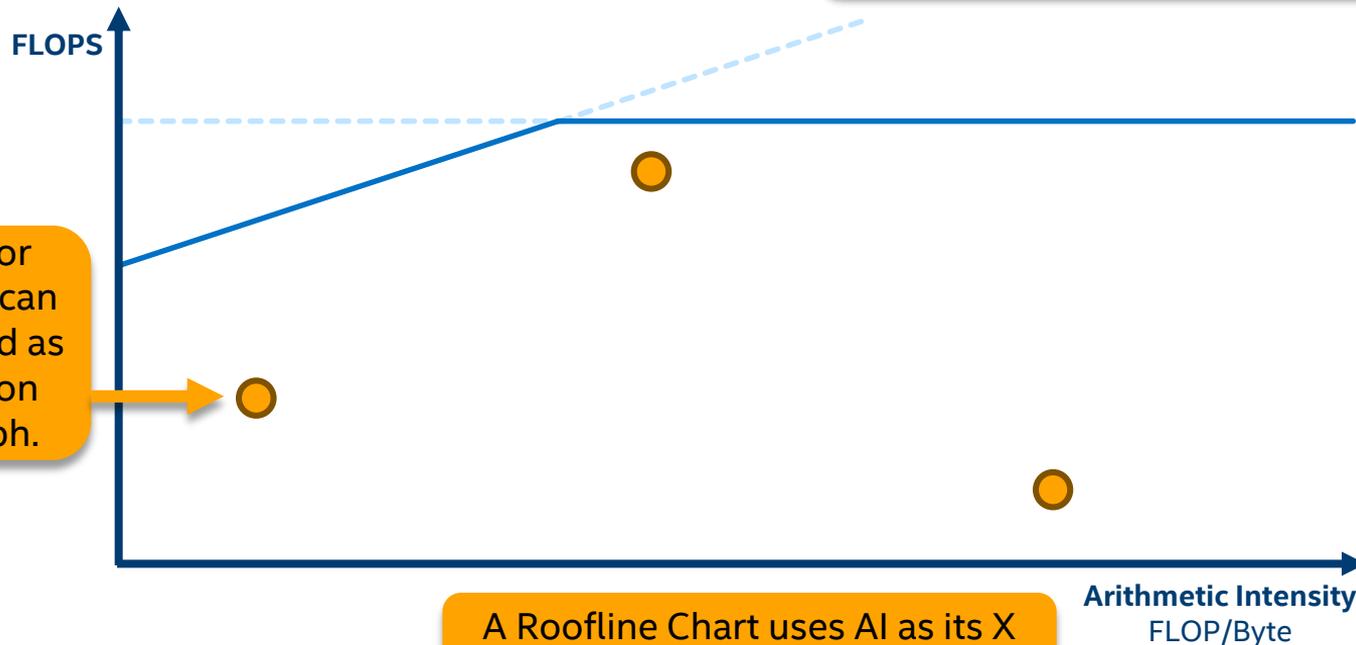
Roofline is based on Arithmetic Intensity (AI) and FLOPS.

- **Arithmetic Intensity:** FLOP / Byte Accessed
 - This is a characteristic of your algorithm



- **FLOPS:** Floating-Point Operations / Second
 - Is a measure of an implementation (it achieves a certain FLOPS)
 - **And** there is a maximum that a platform can provide

Plotting a Roofline Chart



Optimization Notice

Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.



Classic vs. Cache-Aware Roofline

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

Classical Roofline

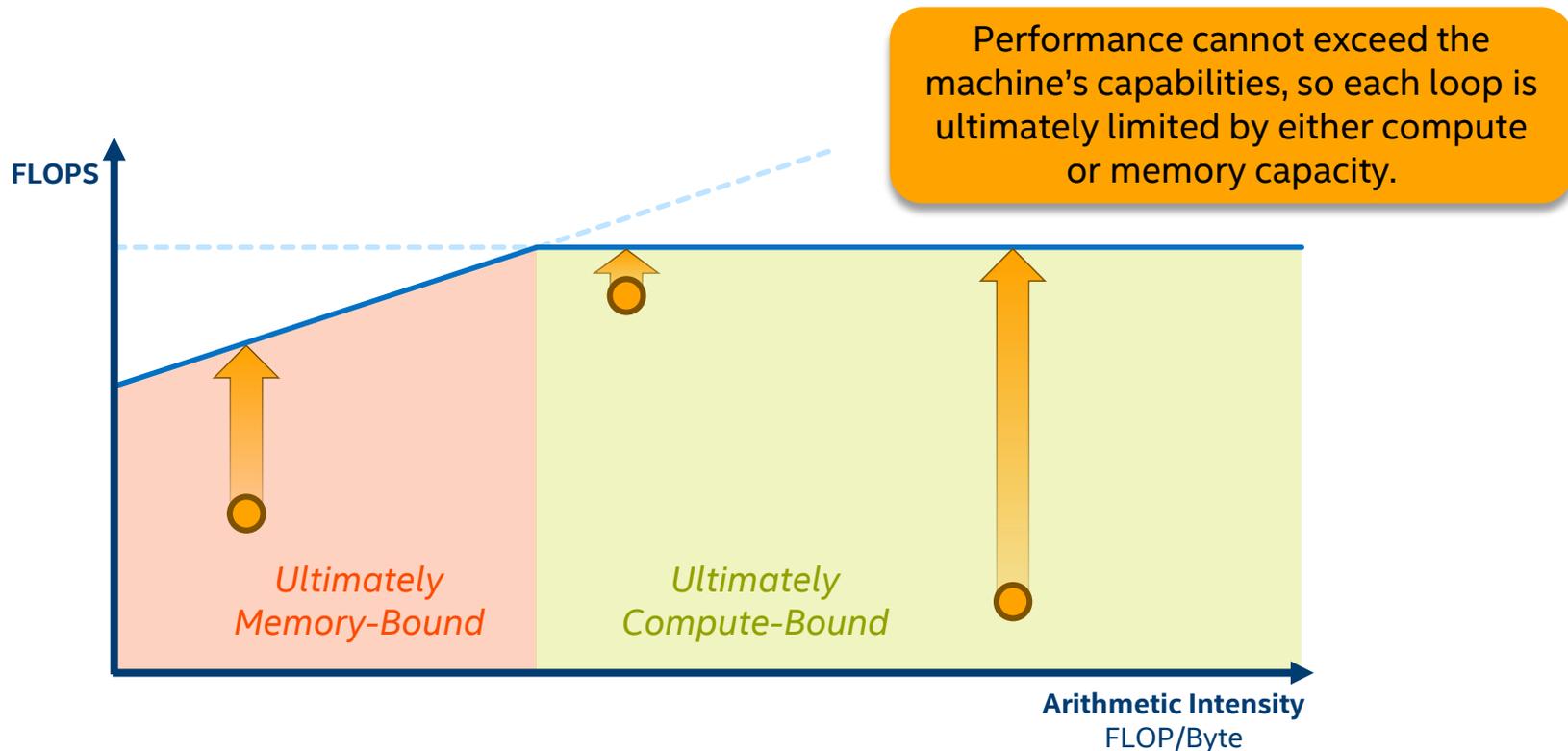
- Traffic measured from one level of memory (usually DRAM)
- AI may change with data set size
- AI changes as a result of memory optimizations

Cache-Aware Roofline

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

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

Ultimate Performance Limits

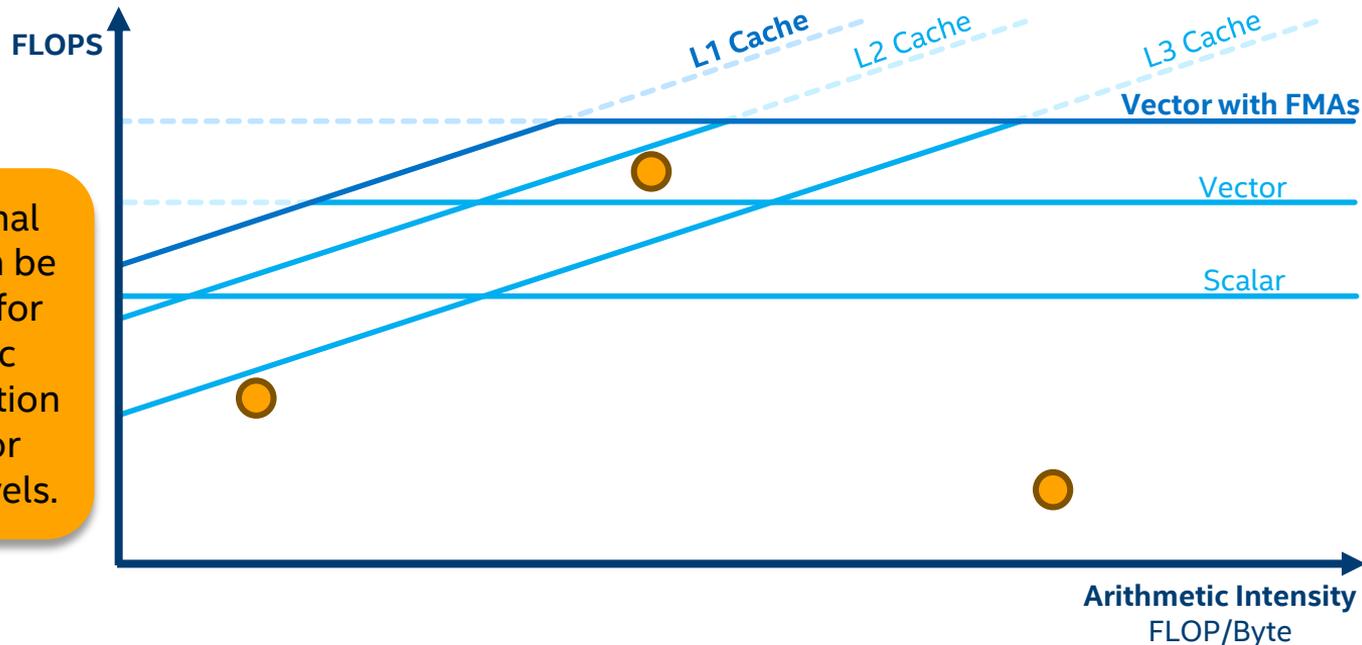


Optimization Notice

Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.



Sub-Roofs and Current Limits

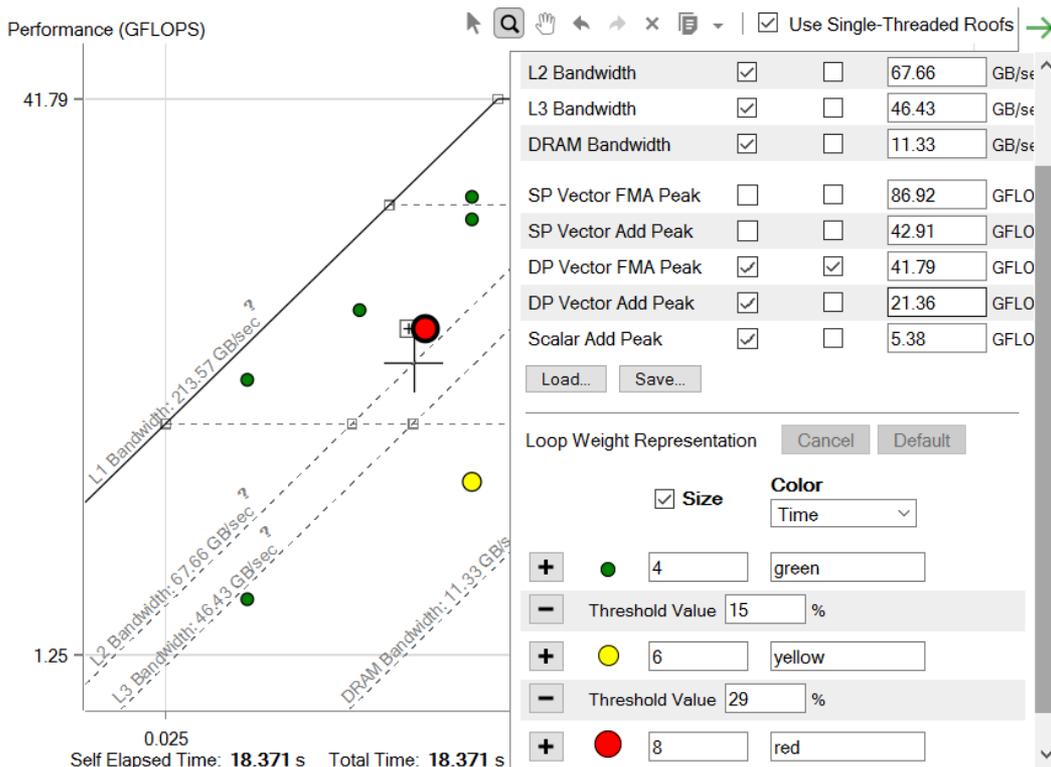


Additional roofs can be plotted for specific computation types or cache levels.

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

The Intel® Advisor Roofline Interface

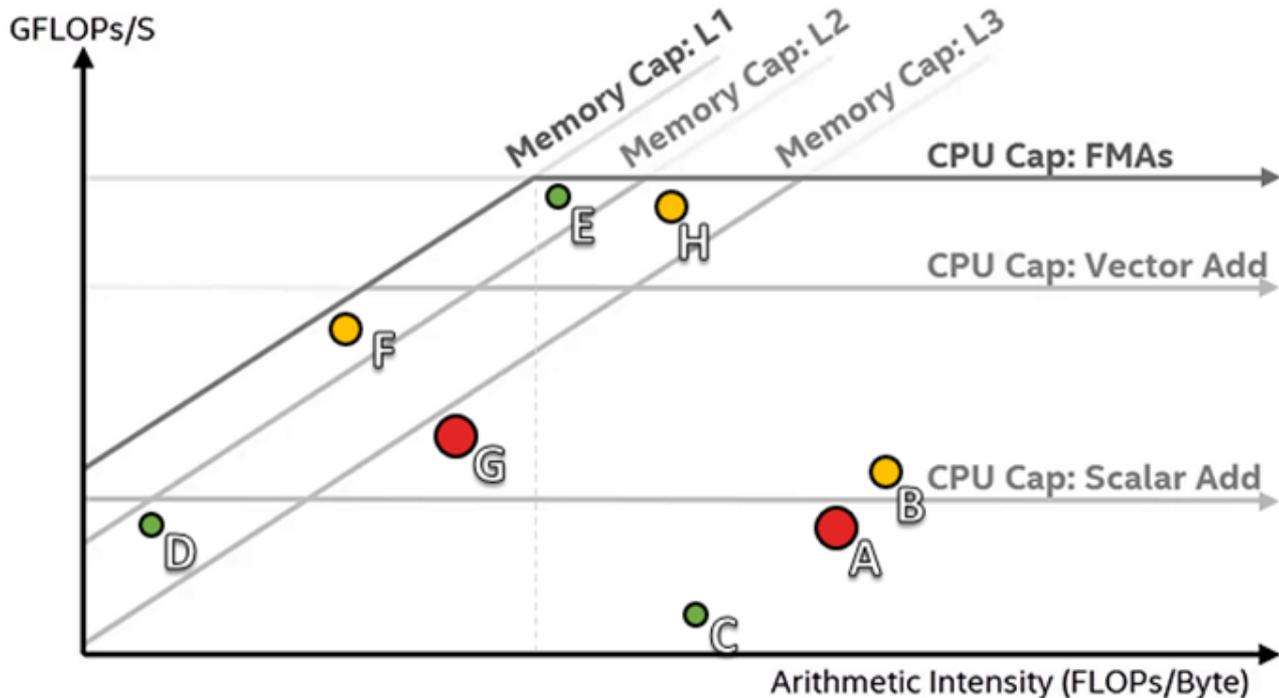
- Roofs are based on benchmarks run before the application.
- Roofs can be hidden, highlighted, or adjusted.
- Intel® Advisor has size- and color-coding for dots.
- Color code by duration or vectorization status
- Categories, cutoffs, and visual style can be modified.



Identifying Good Optimization Candidates

Focus optimization effort where it makes the most difference.

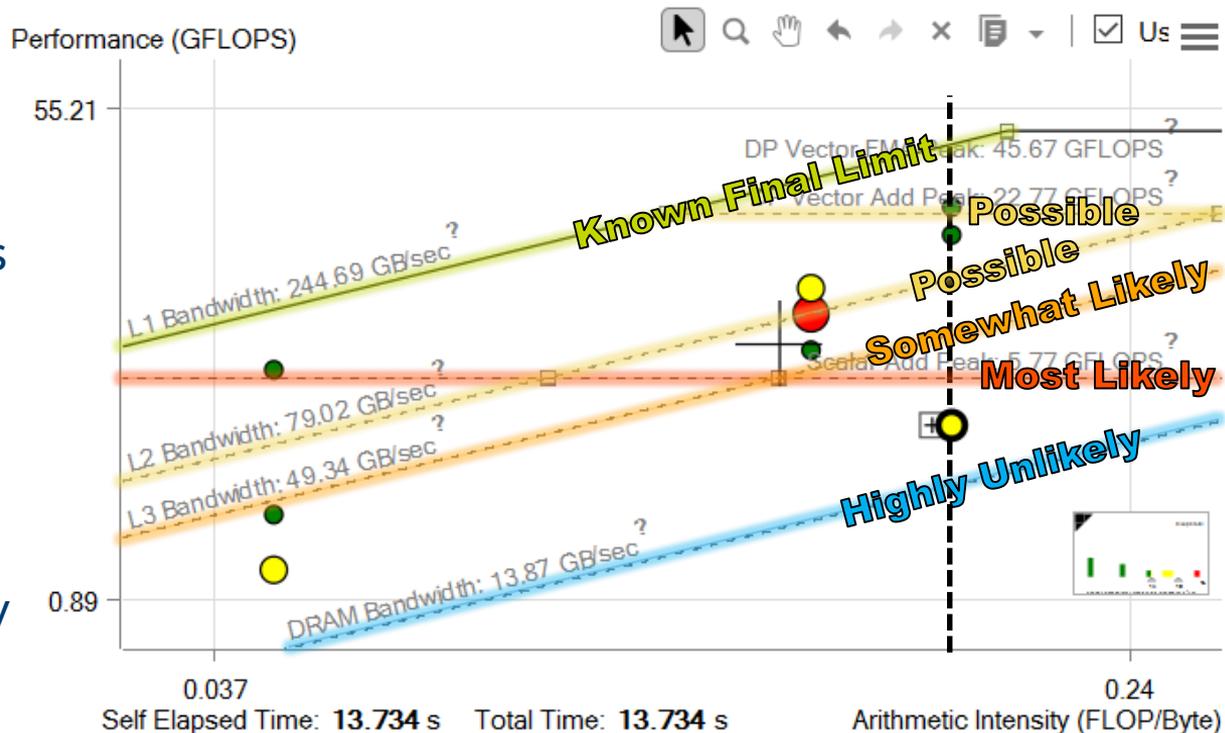
- Large, red loops have the most impact.
- Loops far from the upper roofs have more room to improve.



Identifying Potential Bottlenecks

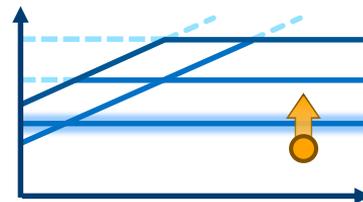
Final roofs *do* apply;
sub-roofs *may* apply.

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



Back to the example...

Overcoming the Scalar Add Peak



- Survey and Code Analytics tabs indicate vectorization status with colored icons.
🔄 = Scalar 🔄 = Vectorized
- “Why No Vectorization” tab and column in Survey explain what prevented vectorization.
- Recommendations tab may help you vectorize the loop.
- Dependencies determines if it’s safe to force vectorization.

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

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

Assumed dependency present
Confirm dependency is real

Potential underutilization of FMA instructions
Target the higher ISA

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

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

Dependencies Analysis

Vectorization Advisor

- Generally, you don't need to run Dependencies analysis unless Advisor tells you to. It produces recommendations to do so if it detects:
 - Loops that remained unvectorized because the compiler was playing it safe with autovectorization.
- Use the survey checkboxes to select which loops to analyze.
- If no dependencies are found, it's safe to force vectorization.
- Otherwise, use the reported variable read/write information to see if you can rework the code to eliminate the dependency.

☑ Recommendation: Confirm dependency is real	Confidence: 🟡 Need More Data
There is no confirmation that a real (proven) dependency is present in the loop. To confirm: Run a Dependencies analysis.	

Summary	Survey & Roofline	Refinement Reports
Site Location	Loop-Carried Dependencies	
⊕ [loop in main at example.c...	🟢 No dependencies found	
⊕ [loop in main at example.c...	🔴 RAW:1	

Back to our example...

Memory Access Patterns Analysis

Collecting a MAP

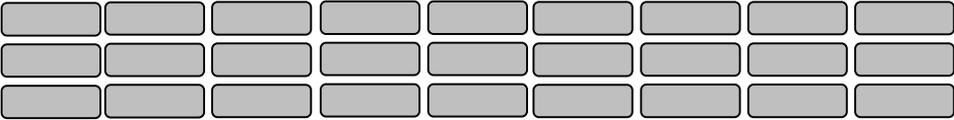
- If you have low vector efficiency, or see that a loop did not vectorize because it was deemed “possible but inefficient”, you may want to run a MAP analysis.
- Advisor will also recommend a MAP analysis if it detects a possible inefficient access pattern.
- Memory access patterns affect vectorization efficiency because they affect how data is loaded into and stored from the vector registers.
- Select the loops you want to run the MAP on using the checkboxes. It may be helpful to reduce the problem size, as MAP only needs to detect patterns, and has high overhead.
 - Note that if changing the problem size requires recompiling, you will need to re-collect the survey before running MAP.

	Vector Issues
<input checked="" type="checkbox"/>	1 Possible inefficient memory access patterns present

Memory Access Patterns Analysis

Reading a MAP

- MAP is color coded by stride type. From best to worst:
 - **Blue** is unit/uniform (stepping by 1 or 0)
 - **Yellow** is constant (stepping a set distance)
 - **Red** is variable (a changing step distance)
- Click a loop in the top pane to see a detailed report below.
 - The strides that contribute to the loop are broken down in this table.



Summary Survey & Roofline Refinement Reports

Site Location	Strides Distrib ...	Access Pa...	Max. Site Footprint	Recommendations
⊕ [loop in main ...	76% / 0% / 24 ...	Mixed stri ...	64KB	
⊕ [loop in main ...	76% / 0% / 24 ...	Mixed stri ...	64KB	
⊕ [loop in main ...	70% / 6% / 24 ...	Mixed stri ...	564MB	
⊕ [loop in main ...	100% / 0% / 0 ...	All unit str ...	70KB	
⊕ [loop in main ...	33% / 67% / 0 ...	Mixed stri ...	616MB	💡 1 Inefficient memory access pa

Memory Access Patterns Report Dependencies Report Recommendations

ID	Stride	Type	Source	Modules	Nested Func...	Variable references
⊕ P1	36000	Constant stride	stride.cpp:49	stride.exe		tableA, tableB
⊕ P2	36000	Constant stride	stride.cpp:49	stride.exe		results
⊕ P7		Parallel site info ...	stride.cpp:47	stride.exe		
P19	0	Uniform stride	stride.exe:0x...	stride.exe	_svml_atan4	
P20	0	Uniform stride	svml_dispmd...	svml_dispmd.dll	_svml_atan2	
P1.	-12; -8; ...	Variable stride	svml_dispmd...	svml_dispmd.dll	_svml_atan2...	
P1.	-12; -8; ...	Variable stride	svml_dispmd...	svml_dispmd.dll	_svml_atan2...	

Legal Disclaimer & Optimization Notice

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

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

Copyright © 2018, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

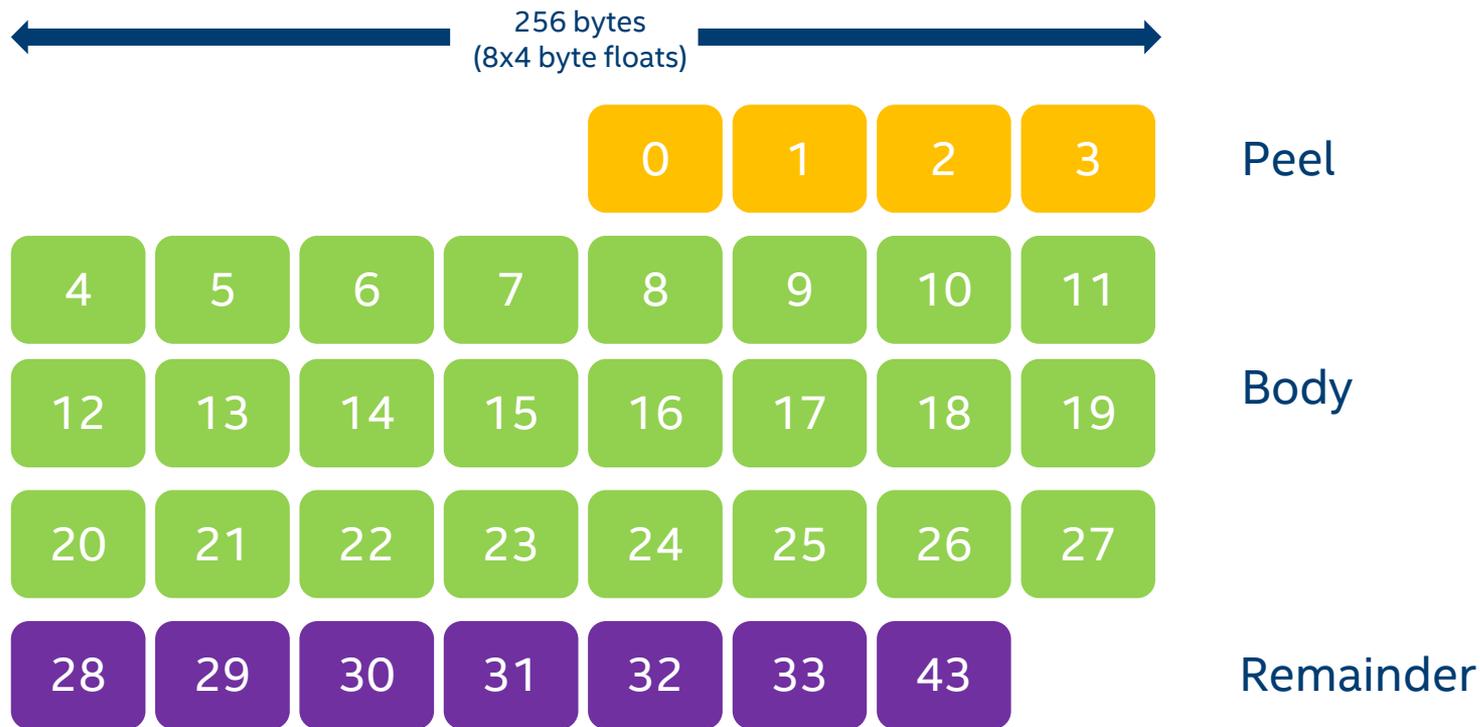
Optimization Notice

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

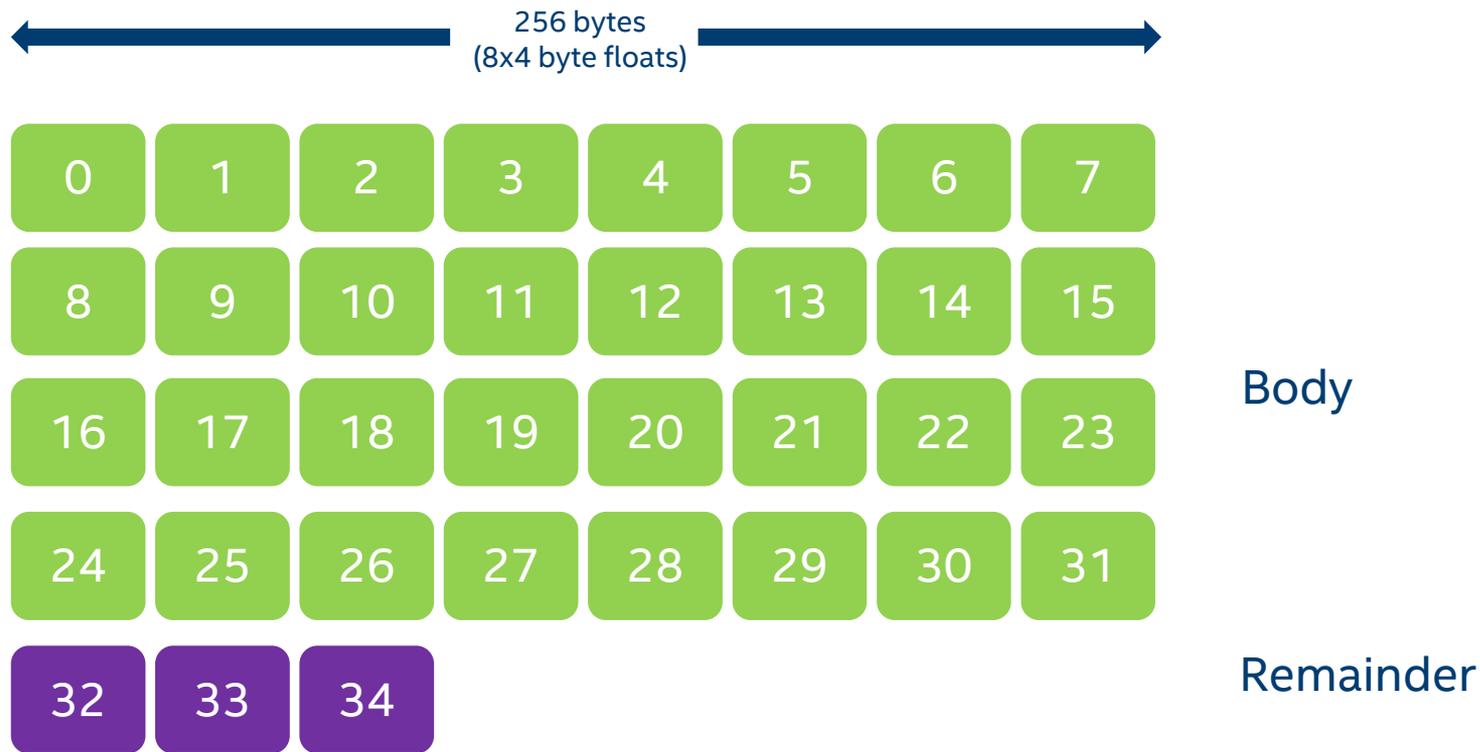
Notice revision #20110804

ALIGNMENT, PADDING AND PEEL/REMAINDER

The original 1D table in the peel example



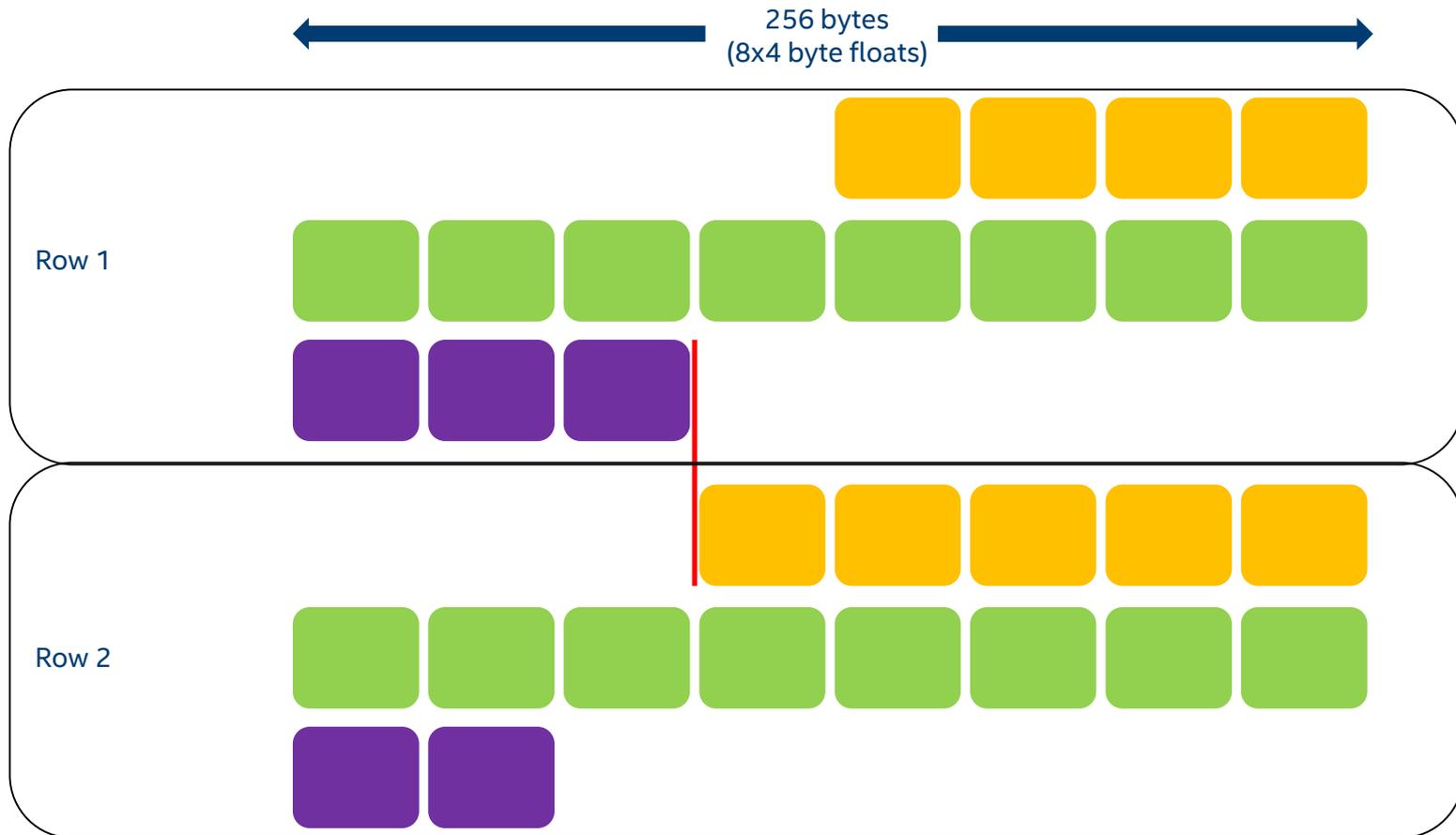
The original 1D table when aligned



The original 1D table when aligned, padded

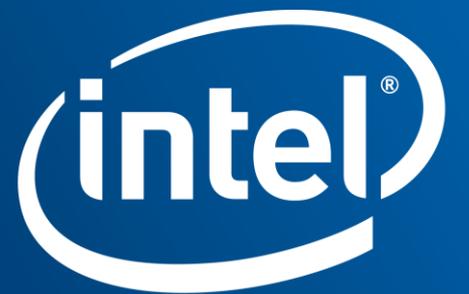


The 2D case



Optimization Notice

Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.



Software