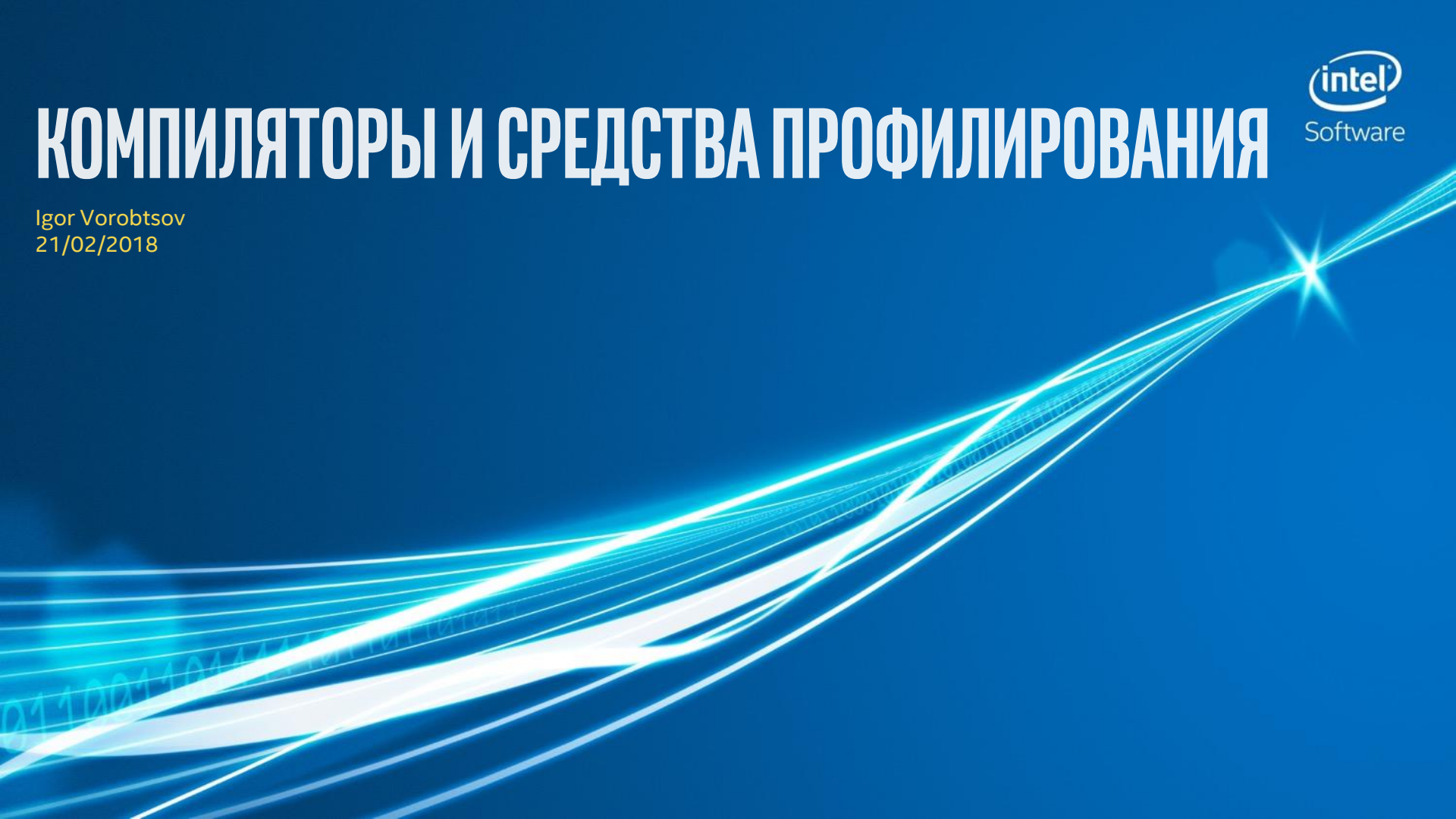
КОМПИЛЯТОРЫ И СРЕДСТВА ПРОФИЛИРОВАНИЯ

Igor Vorobtsov
21/02/2018



AGENDA

- Compiler Basic Optimizations
- Vectorization Essentials
  - Introduction
  - Auto-vectorization of Intel® Compilers
  - Reasons for Vectorization Failures and Inefficiency
- Profilers
  - Intel® Advisor
  - Intel® VTune Amplifier

COMMON OPTIMIZATION OPTIONS

	Windows*	Linux*, macOS*
Disable optimization	/Od	-O0
Optimize for speed (no code size increase)	/O1	-O1
Optimize for speed (default)	/O2	-O2
High-level loop optimization	/O3	-O3
Create symbols for debugging	/Zi	-g
Multi-file inter-procedural optimization	/Qipo	-ipo
Profile guided optimization (multi-step build)	/Qprof-gen /Qprof-use	-prof-gen -prof-use
Optimize for speed across the entire program (“prototype switch”) ***fast* options definitions changes over time!**	/fast same as: /O3 /Qipo /Qprec-div-, /fp:fast=2 /QxHost)	-fast same as: Linux: -ipo –O3 -no-prec-div –static –fp-model fast=2 -xHost) OS X: -ipo -mdynamic-no-pic -O3 -no-prec-div -fp-model fast=2 -xHost
OpenMP support	/Qopenmp	-qopenmp
Automatic parallelization	/Qparallel	-parallel

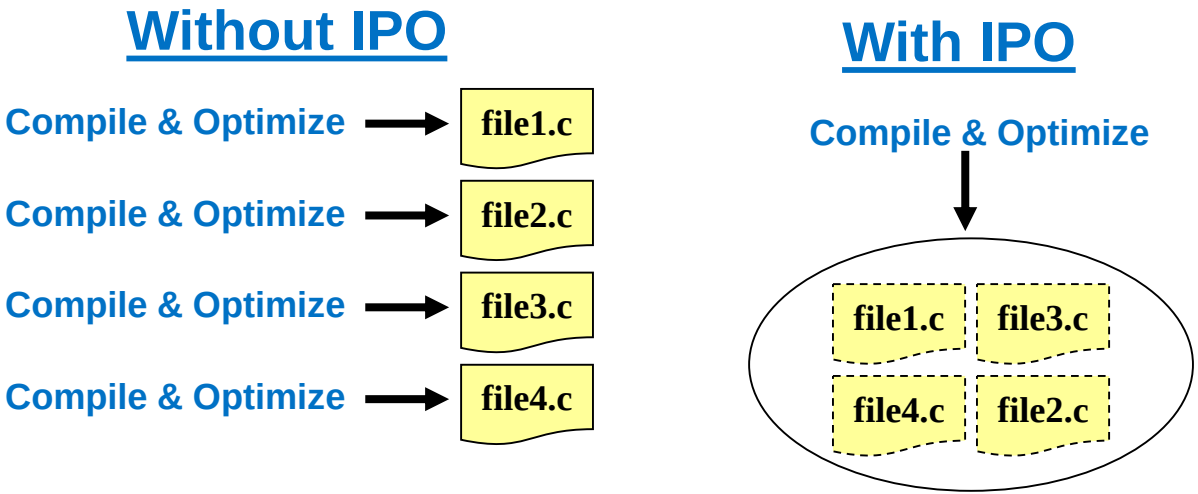
HIGH-LEVEL OPTIMIZATION (HLO)

- Compiler switches:
  `/O2, -O2 (default), /O3, -O3`
  - O3 is suited to applications that have loops that do many floating-point calculations or process large data sets.
  - Some of the optimizations are the same as at O2, but are carried out more aggressively. Some poorly suited applications might run slower at O3 than O2
- Loop level optimizations
  - loop unrolling, cache blocking, prefetching
- More aggressive dependency analysis
  - Determines whether or not it‘s safe to reorder or parallelize statements
- Scalar replacement
  - Goal is to reduce memory by replacing with register references

INTERPROCEDURAL OPTIMIZATIONS

EXTENDS OPTIMIZATIONS ACROSS FILE BOUNDARIES

`/Qip, -ip`	Only between modules of one source file
`/Qipo, -ipo`	Modules of multiple files/whole application



AGENDA

- Compiler Basic Optimizations
- Vectorization Essentials
  - Introduction
    - Why Is Vectorization Important?
    - Basic Vectorization Terms
    - Evolution of SIMD for Intel® Processors
  - Auto-vectorization of Intel® Compilers
  - Reasons for Vectorization Failures and Inefficiency
- Profilers
  - Intel® Advisor
  - Intel® VTune Amplifier

SINGLE INSTRUCTION MULTIPLE DATA (SIMD)

SIMD from Intel has been key for data level parallelism for years:

- 128 bit Intel® Streaming SIMD Extensions (Intel® SSE, SSE2, SSE3, SSE4.1, SSE4.2) and Supplemental Streaming SIMD Extensions (SSSE3)
- 256 bit Intel® Advanced Vector Extensions (Intel® AVX)
- 512 bit Intel® Advanced Vector Extensions 512 (Intel® AVX-512)

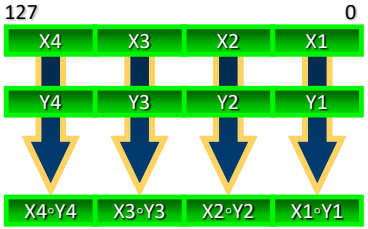
DON'T USE A SINGLE VECTOR LANE!

UN-VECTORIZED AND UN-THREADED SOFTWARE WILL UNDER PERFORM

PERMISSION TO DESIGN FOR ALL LANES

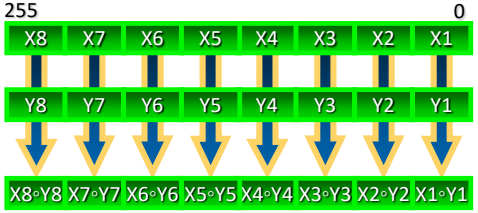
THREADING AND VECTORIZATION NEEDED TO FULLY UTILIZE MODERN HARDWARE

SIMD TYPES FOR INTEL® ARCHITECTURE



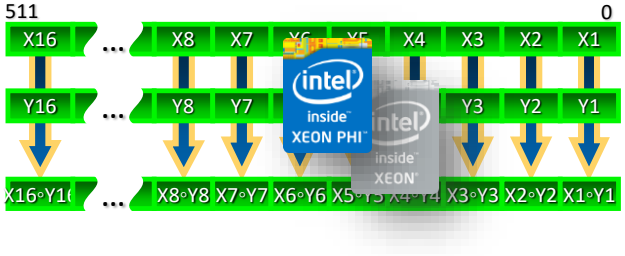
SSE

Vector size: **128 bit**
Data types:
8, 16, 32, 64 bit integer
32 and 64 bit float
VL: 2, 4, 8, 16



AVX

Vector size: **256 bit**
Data types:
8, 16, 32, 64 bit integer
32 and 64 bit float
VL: 4, 8, 16, 32

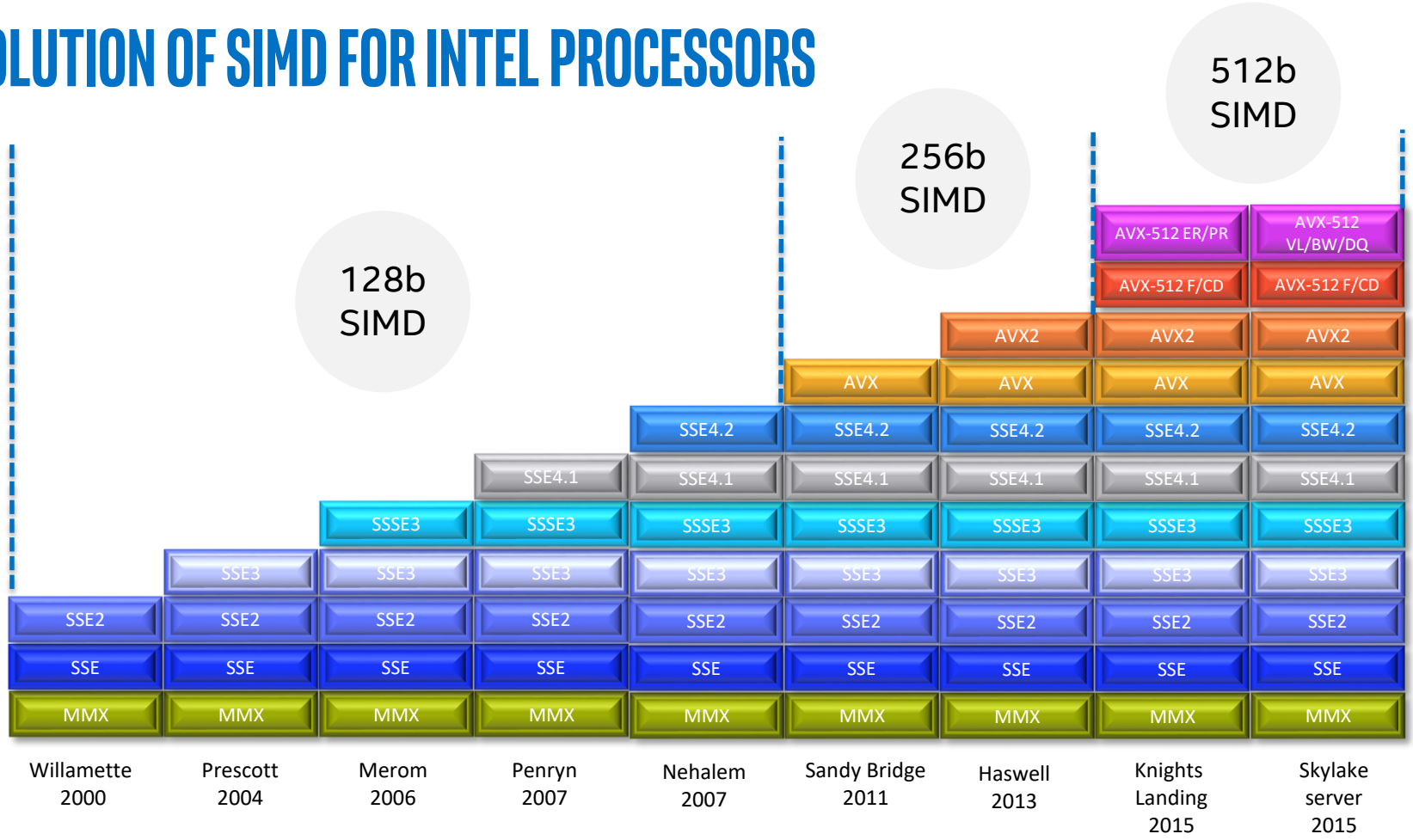


Intel® AVX-512

Vector size: **512 bit**
Data types:
8, 16, 32, 64 bit integer
32 and 64 bit float
VL: 8, 16, 32, 64

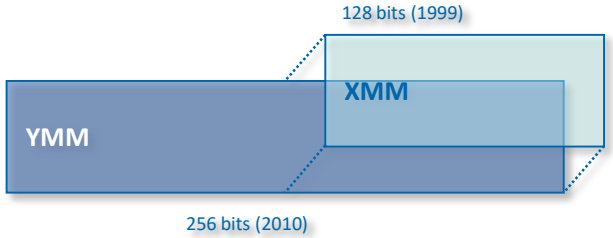
Illustrations: Xi, Yi & results 32 bit integer

EVOLUTION OF SIMD FOR INTEL PROCESSORS

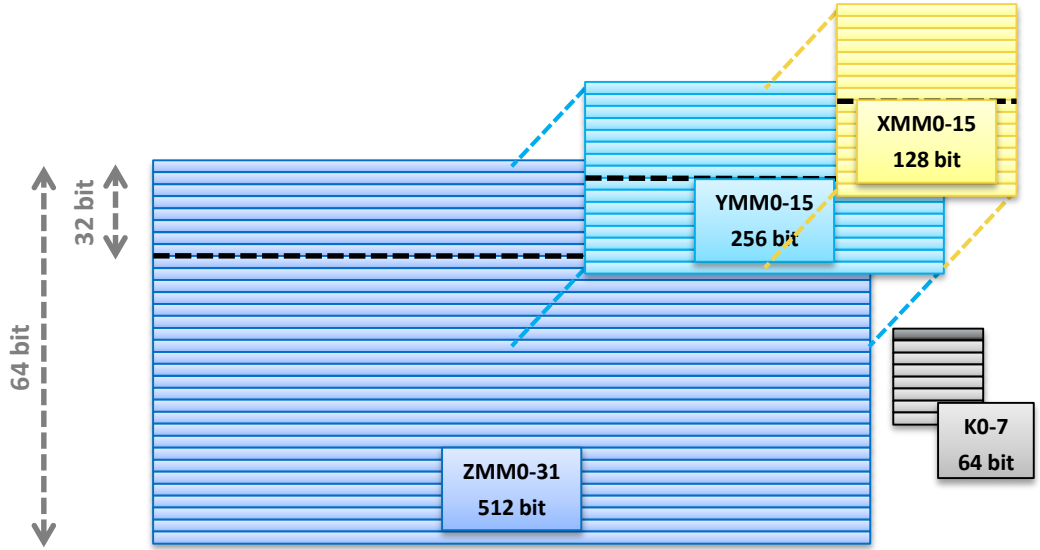


INTEL® AVX AND AVX-512 REGISTERS

AVX is a 256 bit vector extension to SSE:



AVX-512 extends previous AVX and SSE registers to 512 bit:

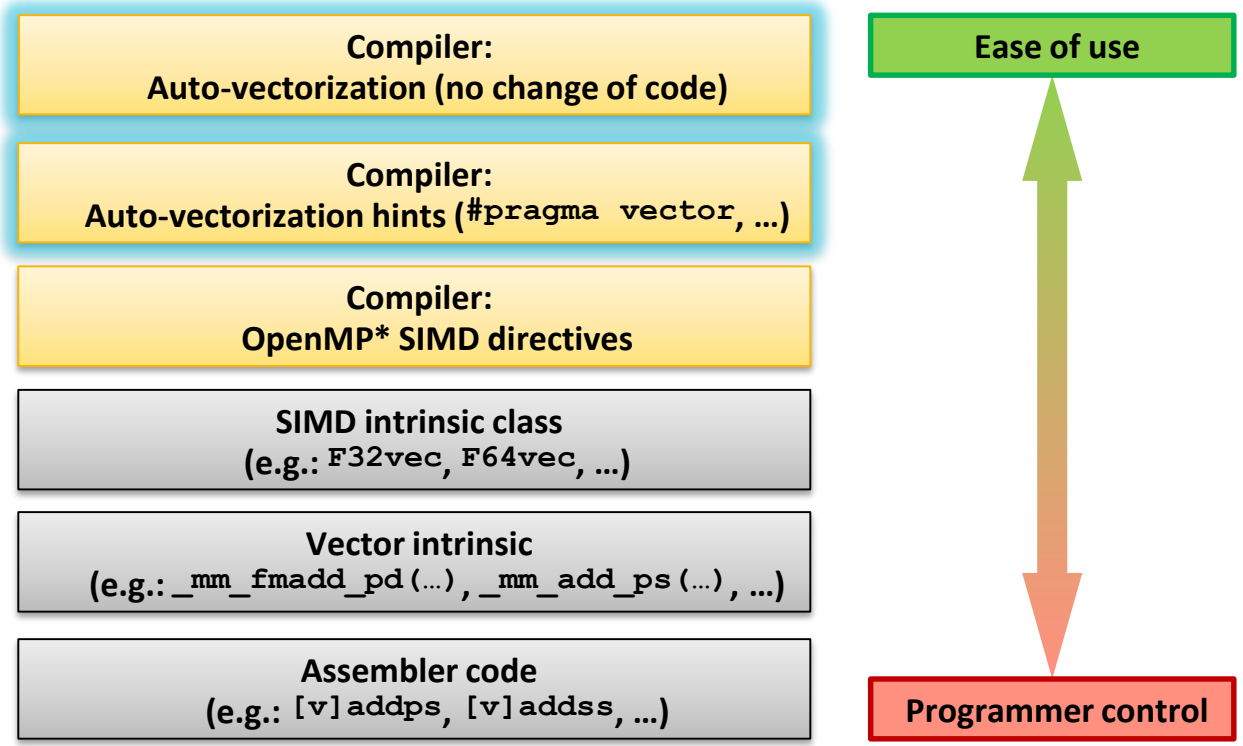


OS support is required

AGENDA

- Compiler Basic Optimizations
- Vectorization Essentials
  - Introduction
  - Auto-vectorization of Intel Compilers
    - Basic Vectorization Switches
    - Vectorization Hints
    - Validating Vectorization Success
    - Optimization Report
  - Reasons for Vectorization Failures and Inefficiency
- Profilers
  - Intel® Advisor
  - Intel® VTune Amplifier

MANY WAYS TO VECTORIZE



AUTO-VECTORIZATION OF INTEL COMPILERS


```
void add(double *A, double *B, double *C)
{
  for (int i = 0; i < 1000; i++)
  C[i] = A[i] + B[i];
}
```

```
subroutine add(A, B, C)
  real*8 A(1000), B(1000), C(1000)
  do i = 1, 1000
      C(i) = A(i) + B(i)
  end do
end
```

Intel® SSE4.2

```
.B2.14:
movups    xmm1, XMMWORD PTR [edx+ebx*8]
movups    xmm3, XMMWORD PTR [16+edx+ebx*8]
movups    xmm5, XMMWORD PTR [32+edx+ebx*8]
movups    xmm7, XMMWORD PTR [48+edx+ebx*8]
movups    xmm0, XMMWORD PTR [ecx+ebx*8]
movups    xmm2, XMMWORD PTR [16+ecx+ebx*8]
movups    xmm4, XMMWORD PTR [32+ecx+ebx*8]
movups    xmm6, XMMWORD PTR [48+ecx+ebx*8]
addpd     xmm1, xmm0
addpd     xmm3, xmm2
addpd     xmm5, xmm4
addpd     xmm7, xmm6
movups    XMMWORD PTR [eax+ebx*8], xmm1
movups    XMMWORD PTR [16+eax+ebx*8], xmm3
movups    XMMWORD PTR [32+eax+ebx*8], xmm5
movups    XMMWORD PTR [48+eax+ebx*8], xmm7
add       ebx, 8
cmp       ebx, esi
jb        .B2.14
...
```

Intel® AVX

```
.B2.15
vmovupd   ymm0, YMMWORD PTR [ebx+eax*8]
vmovupd   ymm2, YMMWORD PTR [32+ebx+eax*8]
vmovupd   ymm4, YMMWORD PTR [64+ebx+eax*8]
vmovupd   ymm6, YMMWORD PTR [96+ebx+eax*8]
vaddpd    ymm1, ymm0, YMMWORD PTR [edx+eax*8]
vaddpd    ymm3, ymm2, YMMWORD PTR [32+edx+eax*8]
vaddpd    ymm5, ymm4, YMMWORD PTR [64+edx+eax*8]
vaddpd    ymm7, ymm6, YMMWORD PTR [96+edx+eax*8]
vmovupd   YMMWORD PTR [esi+eax*8], ymm1
vmovupd   YMMWORD PTR [32+esi+eax*8], ymm3
vmovupd   YMMWORD PTR [64+esi+eax*8], ymm5
vmovupd   YMMWORD PTR [96+esi+eax*8], ymm7
add       eax, 16
cmp       eax, ecx
jb        .B2.15
```

BASIC VECTORIZATION SWITCHES I

Linux*, macOS*: **-x<code>**, Windows*: **/Qx<code>**

- Might enable Intel processor specific optimizations
- Processor-check added to “main” routine:
  Application errors in case SIMD feature missing or non-Intel processor with appropriate/informative message

<code> indicates a feature set that compiler may target (including instruction sets and optimizations)

- Microarchitecture code names: BROADWELL, HASWELL, IVYBRIDGE, KNL, KNM, SANDYBRIDGE, SILVERMONT, SKYLAKE, SKYLAKE-AVX512
- SIMD extensions: COMMON-AVX512, MIC-AVX512, CORE-AVX512, CORE-AVX2, CORE-AVX-I, AVX, SSE4.2, etc.

- Example:

```
icc -xCORE-AVX2 test.c
ifort -xSKYLAKE test.f90
```

BASIC VECTORIZATION SWITCHES II

Linux*, macOS*: **-ax<code>**, Windows*: **/Qax<code>**

- Multiple code paths: baseline and optimized/processor-specific
- Optimized code paths for Intel processors defined by **<code>**
- Multiple SIMD features/paths possible, e.g.: **-axSSE2,AVX**
- Baseline code path defaults to **-msse2** (**/arch:sse2**)
- The baseline code path can be modified by **-m<code>** or **-x<code>** (**/arch:<code>** or **/Qx<code>**)
- Example: `icc -axCORE-AVX512 -xAVX test.c`

Linux*, macOS*: **-m<code>**, Windows*: **/arch:<code>**

- No check and no specific optimizations for Intel processors:
  Application optimized for both Intel and non-Intel processors for selected SIMD feature
- Missing check can cause application to fail in case extension not available

BASIC VECTORIZATION SWITCHES III

Default for Linux*: **`-msse2`**, Windows*: **`/arch:sse2`**:

- Activated implicitly
- Implies the need for a target processor with at least Intel® SSE2

Default for macOS*: **`-msse3`** (IA-32), **`-mssse3`** (Intel® 64)

For 32 bit compilation, **`-mia32`** (**`/arch:ia32`**) can be used in case target processor does not support Intel® SSE2 (e.g. Intel® Pentium® 3 or older)

Special switch for Linux*, macOS*: **`-xHost`**, Windows*: **`/QxHost`**

- Compiler checks SIMD features of current compilation host processor and makes use of latest SIMD feature available
- Works with non-Intel processors as well
- Code only executes on processors with same SIMD feature or later as on build host

CONTROL VECTORIZATION I

Disable vectorization:

- Globally via switch:
  Linux*, macOS*: **`-no-vec`**, Windows*: **`/Qvec-`**
- For a single loop:
  C/C++: **`#pragma novector`**, Fortran: **`!DIR$ NOVECTOR`**
- Compiler still can use some SIMD features

Using vectorization:

- Globally via switch (default for optimization level 2 and higher):
  Linux*, macOS*: **`-vec`**, Windows*: **`/Qvec`**
- Vectorize even if compiler doesn't expect a performance benefit:
  C/C++: **`#pragma vector always`**, Fortran: **`!DIR$ VECTOR ALWAYS`**
- Influence efficiency heuristics threshold:
  Linux*, macOS*: **`-vec-threshold[n]`**
  Windows*: **`/Qvec-threshold[[:]n]`**
  `n`: **`100`** (default; only if profitable) … **`0`** (always)

CONTROL VECTORIZATION II

Verify vectorization:

- Globally:
  Linux*, macOS*: **`-qopt-report`**, Windows*: **`/Qopt-report`**
- Abort compilation if loop cannot be vectorized:
  C/C++: **`#pragma vector always assert`**
  Fortran: **`!DIR$ VECTOR ALWAYS ASSERT`**

Advanced:

- Ignore Vector DEPendencies (IVDEP):
  C/C++: **`#pragma ivdep`**
  Fortran: **`!DIR$ IVDEP`**
- "Enforce" vectorization: - Refer to Explicit Vectorization training for more details
  C/C++: **`#pragma omp simd ...`**
  Fortran:**`!$OMP SIMD ...`**

  Developer is responsible to verify the correctness of the code
  Requires option:
  Linux*, macOS*: **`-qopenmp-simd`**
  Windows*: **`/Qopenmp-simd`**

VALIDATING VECTORIZATION SUCCESS I

Optimization report:

- Linux*, macOS*: **`-qopt-report=<n>`**, Windows*: **`/Qopt-report:<n>`**
  `n`: **`0`**, ..., **`5`** specifies level of detail; **`2`** is default (more later)
- Prints optimization report with vectorization analysis

Optimization report phase:

- Linux*, macOS*: **`-qopt-report-phase=<p>`**,
  Windows*: **`/Qopt-report-phase:<p>`**
- **`<p>`** is **`all`** by default; use **`vec`** for just the vectorization report

Optimization report file:

- Linux*, macOS*: **`-opt-report-file=<f>`**, Windows*: **`/Qopt-report-file:<f>`**
- **`<f>`** can be **`stderr`**, **`stdout`** or a file (default: *.optrpt)

VALIDATING VECTORIZATION SUCCESS II

- Assembler code inspection (Linux*, macOS*: **`-S`**, Windows*: **`/Fa`**):
  - Most reliable way and gives all details of course
  - Check for scalar/packed or (E)VEX encoded instructions:
    Assembler listing contains source line numbers for easier navigation
  - Compiling with **`-qopt-report-embed`** (Linux*, macOS*) or **`/Qopt-report-embed`** (Windows*) helps interpret assembly code
- Performance validation:
  - Compile and benchmark with **`-no-vec -qno-openmp-simd`** or **`/Qvec- /Qopenmp-simd-`**, or on a loop by loop basis via
    `#pragma novector` or **`!DIR$ NOVECTOR`**
  - Compile and benchmark with selected SIMD feature
  - Compare runtime differences
- Intel® Advisor

OPTIMIZATION REPORT EXAMPLE

Example **novec.f90**:

```
1: subroutine fd(y)
2:   integer :: i
3:   real, dimension(10), intent(inout) :: y
4:   do i=2,10
5:     y(i) = y(i-1) + 1
6:   end do
7: end subroutine fd
```

```
$ ifort novec.f90 -c -qopt-report=5 -qopt-report-phase=vec
ifort: remark #10397: optimization reports are generated in *.optrpt files in the output location

$ cat novec.optrpt
…
Begin optimization report for: FD

    Report from: Vector optimizations [vec]


LOOP BEGIN at novec.f90(4,3)
   remark #15344: loop was not vectorized: vector dependence prevents vectorization
   remark #15346: vector dependence: assumed FLOW dependence between y(i)  (5:5) and y(i-1)  (5:5)
LOOP END
…
```

AGENDA

- Compiler Basic Optimizations
- Vectorization Essentials
  - Introduction
  - Auto-vectorization of Intel® Compilers
  - Reasons for Vectorization Failures and Inefficiency
    - Data Dependence
    - Alignment
    - Unsupported Loop Structure
    - Non-Unit Stride Access
    - Mathematical Functions
- Profilers
  - Intel® Advisor
  - Intel® VTune Amplifier

REASONS FOR VECTORIZATION FAILURES AND INEFFICIENCY

Most frequent reasons:

- Data dependence
- Alignment
- Unsupported loop structure
- Non-unit stride access
- Function calls
- Non-vectorizable mathematical functions

All those are common and will be explained in detail next!

DATA DEPENDENCY AND VECTORIZATION

Flow Dependency

```
X = …
… = X
```

read-after-write
RAW

Anti Dependency

```
… = X
X = …
```

write-after-read
WAR

Output Dependency

```
X = …
X = …
```

write-after-write
WAW

```
DO I = 1, 10000
  A(I) = B(I) * 17

  X(I+1) = X(I) + A(I)
ENDDO
```

Loop-independent dependence

Loop-carried dependence

Example:
Despite cyclic dependency, the loop can be vectorized for SSE or AVX in case of VL being max. 3 times the data type size of array **A**.

```
DO I = 1, N
  A(I + 3) = A(I) + C
END DO
```

FAILING DISAMBIGUATION

- Many potential dependencies detected by the compiler result from unresolved memory disambiguation:

  The compiler has to be conservative and has to assume the worst case regarding “aliasing”!

  Example:

```
void scale(int *a, int *b)
{
    for (int i = 0; i < 10000; i++) b[i] = z * a[i];
}
```

- Without additional information (like inter-procedural knowledge) the compiler has to assume a and b to be aliased!
- Use directives, switches and attributes to aid disambiguation!
- This is programming language and operating system specific
- Use with care as the compiler might generate incorrect code in case the hints are not fulfilled!

DISAMBIGUATION HINTS I

- Disambiguating memory locations of pointers in C99:
  Linux*, macOS*: **`-std=c99`**, Windows*: **`/Qstd=c99`**

- Intel® C++ Compiler also allows this for other modes
  (e.g. **`-std=c89`**, **`-std=c++0x`**, ...), too - **not standardized**, though:
  Linux*, macOS*: **`-restrict`**, Windows*: **`/Qrestrict`**

- Declaring pointers with keyword **`restrict`** asserts compiler that they only reference individually assigned, non-overlapping memory areas
- Also true for any result of pointer arithmetic (e.g. **`ptr + 1`** or **`ptr[1]`**)
- Examples:

```
void scale(int *a, int *restrict b)
{
    for (int i = 0; i < 10000; i++) b[i] = z * a[i];
}

void mult(int a[][NUM], int b[restrict][NUM])
{ ... }
```

DISAMBIGUATION HINTS II

- Directive:

  `#pragma ivdep` (C/C++) or **`!DIR$ IVDEP`** (Fortran)

- For C/C++:
  - Assume no aliasing at all (dangerous!):
    Linux*, macOS*: **`-fno-alias`**, Windows*: **`/Oa`**
  - Assume ISO C Standard aliasing rules:
    Linux*, macOS*: **`-ansi-alias`**, Windows*: **`/Qansi-alias`**
    Default on Linux, not on Windows
    - Turns on ANSI aliasing checker
  - No aliasing between function arguments:
    Linux*, macOS*: **`-fargument-noalias`**, Windows*: **`/Qalias-args-`**
  - No aliasing between function arguments and global storage:
    Linux*, macOS*: **`-fargument-noalias-global`**, Windows*: N/A

MULTIVERSIONING FOR DATA DEPENDENCE

Example `test.cpp`:

```
1: void add(double *A, double *B, double *C)
2: {
3:    for (int i = 0; i < 1000; i++)
4:          C[i] = A[i] + B[i];
5: }
```

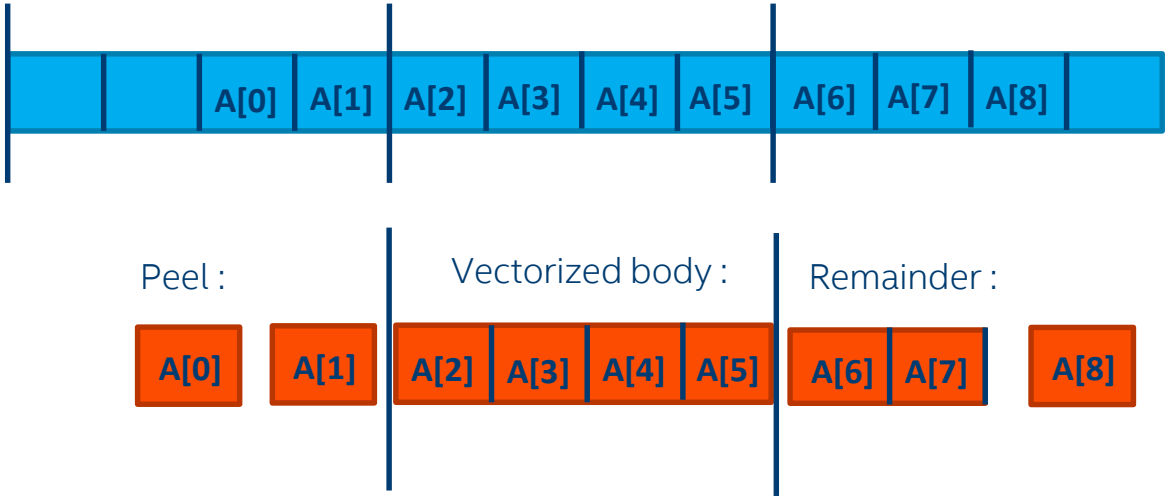
```
$ icpc test.cpp -c -qopt-report=5 -qopt-report-phase=vec
icpc: remark #10397: optimization reports are generated in *.optrpt files in the output location
$ cat test.optrpt
...
```

MULTIVERSIONING FOR DATA DEPENDENCE

```
LOOP BEGIN at  test.cpp(3,2)
Multiversioned v1
test.cpp(4,3):remark #15388: vectorization support: reference C[i] has aligned access
test.cpp(4,3):remark #15389: vectorization support: reference A[i] has unaligned access
test.cpp(4,3):remark #15388: vectorization support: reference B[i] has aligned access
test.cpp(3,2):remark #15381: vectorization support: unaligned access used inside loop body
test.cpp(3,2):remark #15305: vectorization support: vector length 2
test.cpp(3,2):remark #15399: vectorization support: unroll factor set to 4
test.cpp(3,2):remark #15309: vectorization support: normalized vectorization overhead 0.607
test.cpp(3,2):remark #15300: LOOP WAS VECTORIZED
test.cpp(3,2):remark #15442: entire loop may be executed in remainder
test.cpp(3,2):remark #15448: unmasked aligned unit stride loads: 1
test.cpp(3,2):remark #15449: unmasked aligned unit stride stores: 1
test.cpp(3,2):remark #15450: unmasked unaligned unit stride loads: 1
test.cpp(3,2):remark #15475: --- begin vector cost summary ---
test.cpp(3,2):remark #15476: scalar cost: 8
test.cpp(3,2):remark #15477: vector cost: 3.500
test.cpp(3,2):remark #15478: estimated potential speedup: 2.250
test.cpp(3,2):remark #15488: --- end vector cost summary ---
LOOP END
...
LOOP BEGIN at test.cpp(3,2)
Multiversioned v2
test.cpp(3,2):remark #15304: loop was not vectorized: non-vectorizable loop instance from
multiversioning
LOOP END
```

COMPILER HELPS WITH ALIGNMENT

SSE:	16 bytes
AVX:	32 bytes
AVX512	64 bytes



Compiler can split loop in 3 parts to have aligned access in the loop body

DATA ALIGNMENT FOR C/C++

- Aligned heap memory allocation by intrinsic/library call:

```
void* aligned_alloc( std::size_t alignment, std::size_t size );
```
(since C++17)

```
void* _mm_malloc(int size, int base)
```

Linux*, macOS* only:

```
int posix_memaligned(void **p, size_t base, size_t size)
```

- Automatically allocate memory with the alignment of that type using new operator:

```
#include <aligned_new>
```

- Align attribute for variable declarations:

alignas specifier (since C++11):

alignas(64) char line[128];

Linux*, macOS*, Windows*: **__declspec(align(base)) <var>**

Linux*, macOS*: **<var> __attribute__((aligned(base)))**

Portability caveat:

__declspec is not known for GCC and **__attribute__** not for Microsoft Visual Studio*!

COMPILER ALIGNMENT HINTS FOR C/C++

- Hint that start address of an array is aligned (Intel Compiler only):
  `__assume_aligned(<array>, base)`

- `#pragma vector [aligned|unaligned]`
  - Only for Intel Compiler
  - Asserts compiler that aligned memory operations can be used for all data accesses in loop following directive
  - Use with care:
    The assertion must be satisfied for all(!) data accesses in the loop!

PROBLEMS DEFINING ALIGNMENT

```
void matvec(double a[][ROWWIDTH], double b[], double c[])
{
  int i, j;
  for(i = 0; i < size1; i++) {
    b[i] = 0;
#pragma vector aligned
    for(j = 0; j < size2; j++)
      b[i] += a[i][j] * c[j];
  }
}
```

- Let's assume **a**, **b** and **c** are be declared 16 byte aligned in calling routine
- Question: Would this be correct when compiled for Intel® SSE2?
- Answer: It depends on **ROWWIDTH**!
  - **ROWWIDTH** is even: Yes
  - **ROWWIDTH** is odd: No, vectorized code fails with alignment error after first row!
- Solution:
  Instead of pragma, use **__assume_aligned(<array>, base)**. This refers to the start address only.
  Vectorization is still limited, though!

UNSUPPORTED LOOP STRUCTURE

Loops where compiler does not know the iteration count:

- Upper/lower bound of a loop are not loop-invariant
- Loop stride is not constant
- Early bail-out during iterations (e.g. **break**, exceptions, etc.)
- Too complex loop body conditions for which no SIMD feature instruction exists
- Loop dependent parameters are globally modifiable during iteration
  (language standards require load and test for each iteration)

Transform is possible, e.g.:

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

void doit(int *a, struct _x *x)
{

  for(int i = 0; i < x->bound; i++)
    a[i] = 0;
}
```

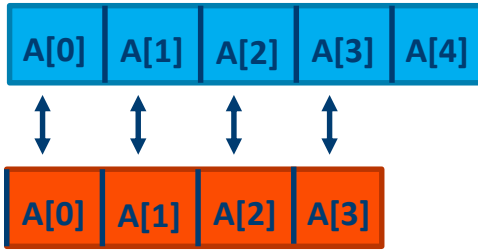
```
struct _x { int d; int bound; };

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

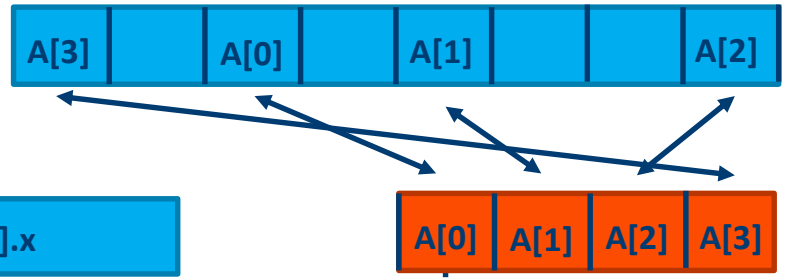
```
loop was not vectorized: loop control variable i was found, but
loop iteration count cannot be computed before executing the loop
```

MEMORY ACCESS PATTERNS

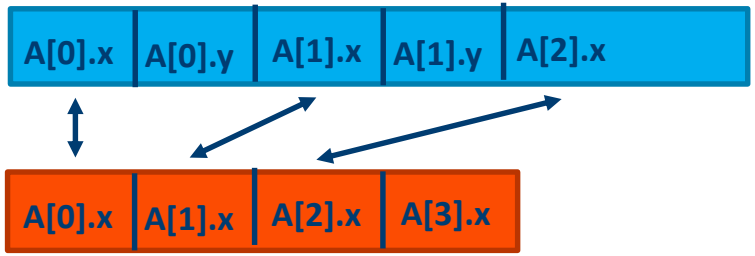
Unit strided (contiguous):



Arbitrary access:

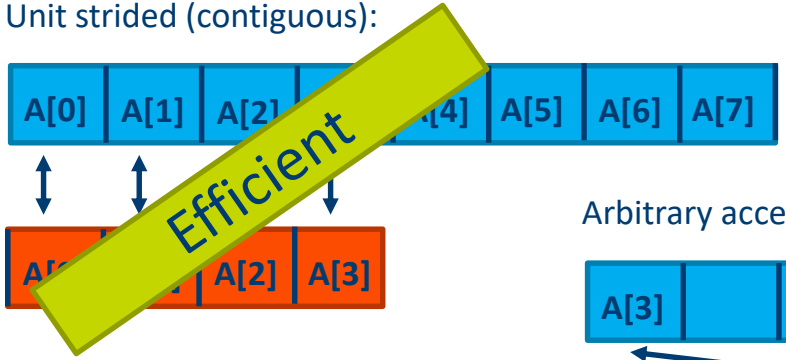


Constant strided:

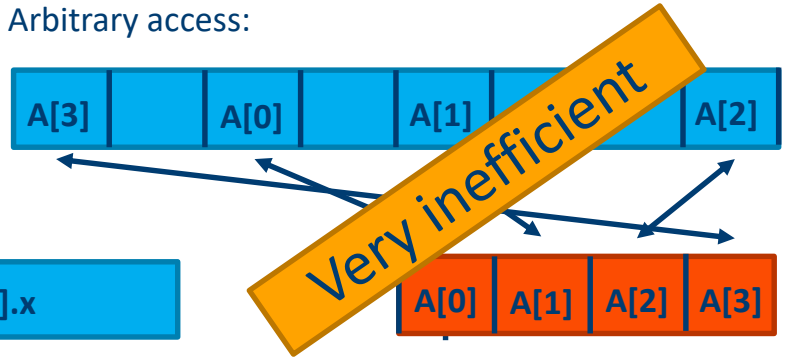


MEMORY ACCESS PATTERNS

Unit strided (contiguous):



Arbitrary access:



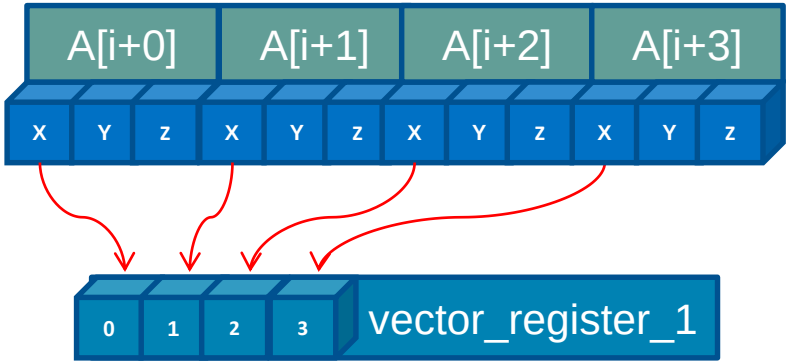
Constant strided:

Extract/insert, shuffle, gather/scatter instructions are used

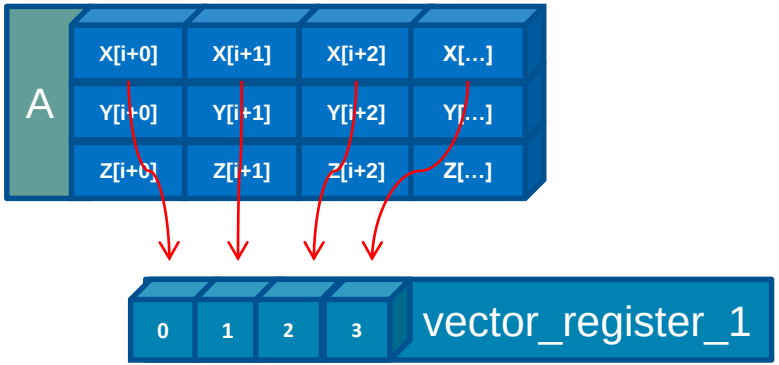
WHAT IS INTEL® SDLT?

- The SIMD Data Layout Template library is a C++11 template library to quick convert *Array of Structures* to *Structure of Arrays* representation
- SDLT vectorizes your code by making memory access contiguous, which can lead to more efficient code and better performance

AOS



SOA



- https://software.intel.com/en-us/videos/improving-vectorization-efficiency-using-intel-simd-data-layout-templates-intel-sdlt

USE FUNCTION CALLS INSIDE LOOP

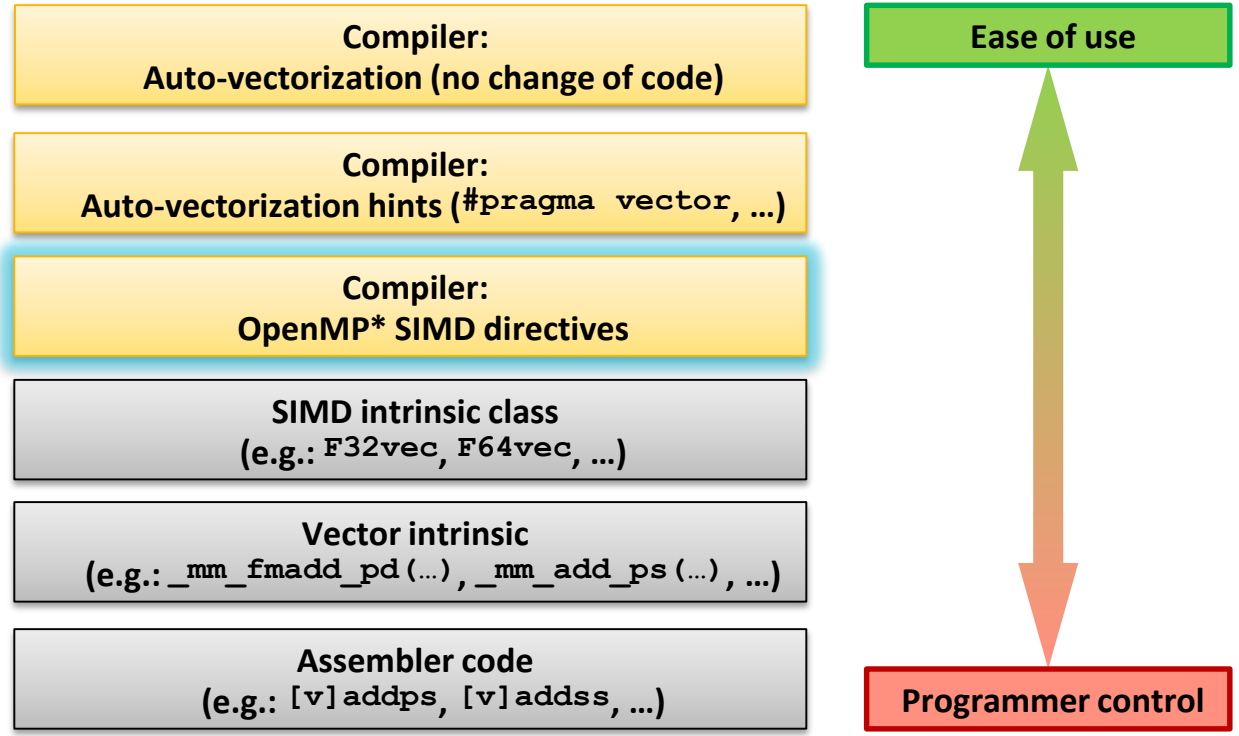
- Success of in-lining can be verified using the optimization report:
  Linux*, macOS*: **`-qopt-report=<n> -qopt-report-phase=ipo`**
  Windows*: **`/Qopt-report:<n> /Qopt-report-phase:ipo`**
- Intel compilers offer a large set of switches, directives and language extensions to control in-lining globally or locally, e.g.:
  - **`#pragma [no]inline`** (C/C++), **`!DIR$ [NO]iNLINE`** (Fortran):
    Instructs compiler that all calls in the following statement can be in-lined or may never be in-lined
  - **`#pragma forceinline`** (C/C++), **`!DIR$ FORCEINLINE`** (Fortran):
    Instructs compiler to ignore the heuristic for in-lining and to inline all calls in the following statement
  - See section “Inlining Options” in compiler manual for full list of options
- IPO offers additional advantages to vectorization
  - Inter-procedural alignment analysis
  - Improved (more precise) dependency analysis

VECTORIZABLE MATHEMATICAL FUNCTIONS

- Calls to most mathematical functions in a loop body can be vectorized using "Short Vector Math Library" (SVML):
  - SVML (**libsvml**) provides vectorized implementations of different mathematical functions
  - Optimized for latency compared to the VML library component of Intel® MKL which realizes same functionality but optimized for throughput
- Routines in **libsvml** can also be called explicitly, using intrinsics (C/C++)
- These mathematical functions are currently supported:

acos	acosh	asin	asinh	atan	atan2	atanh	cbrt
ceil	cos	cosh	erf	erfc	erfinv	exp	exp2
fabs	floor	fmax	fmin	log	log10	log2	pow
round	sin	sinh	sqrt	tan	tanh	trunc	

MANY WAYS TO VECTORIZE



OBSTACLES TO AUTO-VECTORIZATION

Multiple loop exits
- Or trip count unknown at loop entry

Dependencies between loop iterations
- Mostly, avoid read-after-write “flow” dependencies

Function or subroutine calls
- Except where inlined

Nested (Outer) loops
- Unless inner loop fully unrolled

Complexity
- Too many branches
- Too hard or time-consuming for compiler to analyze

https://software.intel.com/articles/requirements-for-vectorizable-loops

OPENMP* SIMD PROGRAMMING

Vectorization is so important

➔ consider explicit vector programming

Modeled on OpenMP* for threading (explicit parallel programming)

Enables reliable vectorization of complex loops the compiler can't auto-vectorize

E.g. outer loops

Directives are commands to the compiler, not hints

E.g. #pragma omp simd or !$OMP SIMD

Compiler does no dependency and cost-benefit analysis !!

Programmer is responsible for correctness (like OpenMP threading)

E.g. PRIVATE, REDUCTION or ORDERED clauses

Incorporated in OpenMP since version 4.0 ⇒ portable

-qopenmp or -qopenmp-simd to enable

OPENMP* SIMD PRAGMA

- Use #pragma omp simd with -qopenmp-simd

```
void addit(double* a, double* b, int
m, int n, int x)
{
  for (int i = m; i < m+n; i++)    {
        a[i] = b[i] + a[i-x];
  }
}
```

loop was not vectorized:
existence of vector dependence.

```
void addit(double* a, double * b, int m,
int n, int x)
{
#pragma omp simd  // I know x<0
  for (int i = m; i < m+n; i++)    {
        a[i] = b[i] + a[i-x];
  }
}
```

SIMD LOOP WAS VECTORIZED.

- Use when you **KNOW** that a given loop is safe to vectorize

The Intel® Compiler will vectorize if at all possible

  - (ignoring dependency or efficiency concerns)
  - Minimizes source code changes needed to enforce vectorization

CLAUSES FOR OMP SIMD DIRECTIVES

- The programmer (i.e. you!) is responsible for correctness
  - Just like for race conditions in loops with OpenMP* threading
- Available clauses:
  - PRIVATE
  - LASTPRIVATE
  - REDUCTION

  } like OpenMP for threading
  - COLLAPSE (for nested loops)
  - LINEAR (additional induction variables)
  - SIMDLEN (preferred number of iterations to execute concurrently)
  - SAFELEN (max iterations that can be executed concurrently)
  - ALIGNED (tells compiler about data alignment)

EXAMPLE: OUTER LOOP VECTORIZATION

```
#ifdef  KNOWN_TRIP_COUNT
#define MYDIM 3
#else                              // pt     input  vector of points
#define MYDIM nd                   // ptref  input  reference point
#endif                             // dis    output vector of distances
#include <math.h>

void dist( int n, int nd, float pt[][MYDIM], float dis[], float ptref[]) {
/* calculate distance from data points to reference point */

#pragma omp simd
      for (int ipt=0; ipt<n; ipt++) {
            float d = 0.;

            for (int j=0; j<MYDIM; j++) {
                  float t = pt[ipt][j] - ptref[j];
                  d+= t*t;
            }

            dis[ipt] = sqrtf(d);
      }
}
```

Outer loop with high trip count

Inner loop with low trip count

OUTER LOOP VECTORIZATION

```
icc -std=c99 -xavx -qopt-report-phase=loop,vec -qopt-report-file=stderr -c dist.c
...
LOOP BEGIN at dist.c(26,2)
  remark #15542: loop was not vectorized: inner loop was already vectorized
...
  LOOP BEGIN at dist.c(29,3)
    remark #15300: LOOP WAS VECTORIZED
```

We can vectorize the outer loop by activating the pragma using -qopenmp-simd

```
#pragma omp simd
```

Would need private clause for d and t if declared outside SIMD scope

```
icc -std=c99 -xavx -qopenmp-simd -qopt-report-phase=loop,vec -qopt-report-file=stderr -qopt-report=4 -c dist.c
...
LOOP BEGIN at dist.c(26,2)
  remark #15328: ... non-unit strided load was emulated for the variable <pt[ipt][j]>, stride is unknown to compiler
  remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
  LOOP BEGIN at dist.c(29,3)
    remark #25460: No loop optimizations reported
```

UNROLLING THE INNER LOOP

There is still an inner loop.

If the trip count is fixed and the compiler knows it, the inner loop can be fully unrolled. Outer loop vectorization is more efficient also because stride is now known

```
icc -std=c99 -xavx -qopenmp-simd -DKNOWN_TRIP_COUNT -qopt-report-phase=loop,vec -
qopt-report-file=stderr –qopt-report=4 -c dist.c

...
LOOP BEGIN at dist.c(26,2)
  remark #15328: vectorization support: non-unit strided load was emulated for the variable <pt[ipt][j]>,
                 stride is 3   [ dist.c(30,14) ]
  remark #15301: OpenMP SIMD LOOP WAS VECTORIZED

  LOOP BEGIN at dist.c(29,3)
    remark #25436: completely unrolled by 3   (pre-vector)
  LOOP END
LOOP END
```

LOOPS CONTAINING FUNCTION CALLS

- Function calls can have side effects that introduce a loop-carried dependency, preventing vectorization
- Possible remedies:
  - Inlining
    - best for small functions
    - Must be in same source file, or else use -ipo
  - OMP SIMD pragma or directive to vectorize rest of loop, while preserving scalar calls to function (last resort)
  - SIMD-enabled functions
    - Good for large, complex functions and in contexts where inlining is difficult
    - Call from regular “for” or “DO” loop
    - In Fortran, adding “ELEMENTAL” keyword allows SIMD-enabled function to be called with array section argument

SIMD-ENABLED FUNCTION

Compiler generates SIMD-enabled (vector) version of a scalar function that can be called from a vectorized loop:

```
#pragma omp declare simd uniform(y,z,xp,yp,zp)
float func(float x, float y, float z, float xp, float yp, float zp)
{
float denom = (x-xp)*(x-xp) + (y-yp)*(y-yp) + (z-zp)*(z-zp);
  denom = 1./sqrtf(denom);
  return denom;
}
...
#pragma omp simd  private(x)  reduction(+:sumx)
for (i=1; i<nx; i++) {
   x = x0 + (float) i * h;
   sumx = sumx + func(x, y, z, xp, yp, zp);
  }
```

y, z, xp, yp and zp are constant, x can be a vector

FUNCTION WAS VECTORIZED with ...

These clauses are required for correctness, just like for OpenMP*

SIMD LOOP WAS VECTORIZED.

#pragma omp simd may not be needed in simpler cases

SPECIAL IDIOMS

- Dependency on an earlier iteration usually makes vectorization unsafe
  - Some special patterns can still be handled by the compiler
    - Provided the compiler recognizes them (auto-vectorization)
      - Often works only for simple, 'clean' examples
    - Or the programmer tells the compiler (explicit vector programming)
      - May work for more complex cases
    - Examples: reduction, compress/expand, search, histogram/scatter, minloc
  - Sometimes, the main speed-up comes from vectorizing the rest of a large loop, more than from vectorization of the idiom itself

REDUCTION – SIMPLE EXAMPLE

```
double reduce(double a[], int na) {
/*   sum all positive elements of a  */
     double sum = 0.;
     for (int ia=0; ia <na; ia++)  {
        if (a[ia] > 0.)  sum += a[ia];    // sum causes cross-iteration dependency
     }
     return sum;
}
```

- Auto-vectorizes with any instruction set:

icc -std=c99 -O2 -qopt-report-phase=loop,vec -qopt-report-file=stderr reduce.c;

…

LOOP BEGIN at reduce.c(17,6))

remark #15300: LOOP WAS VECTORIZED

REDUCTION – WHEN AUTO-VECTORIZATION DOESN'T WORK

icc -std=c99 -O2 -fp-model precise -qopt-report-phase=loop,vec -qopt-report-file=stderr reduce.c;
…

LOOP BEGIN at reduce.c(17,6))
remark #15331: loop was not vectorized: precise FP model implied by the command line or a directive prevents vectorization. Consider using fast FP model [reduce.c(18,26)

- Vectorization would change order of operations, and hence the result
  - Can use a SIMD pragma to override and vectorize:

```
#pragma omp simd reduction(+:sum)
      for (int ia=0; ia <na; ia++)
{
            sum += …
```

Without the reduction clause, results would be incorrect because of the flow dependency. See "SIMD-Enabled Function" section for another example.

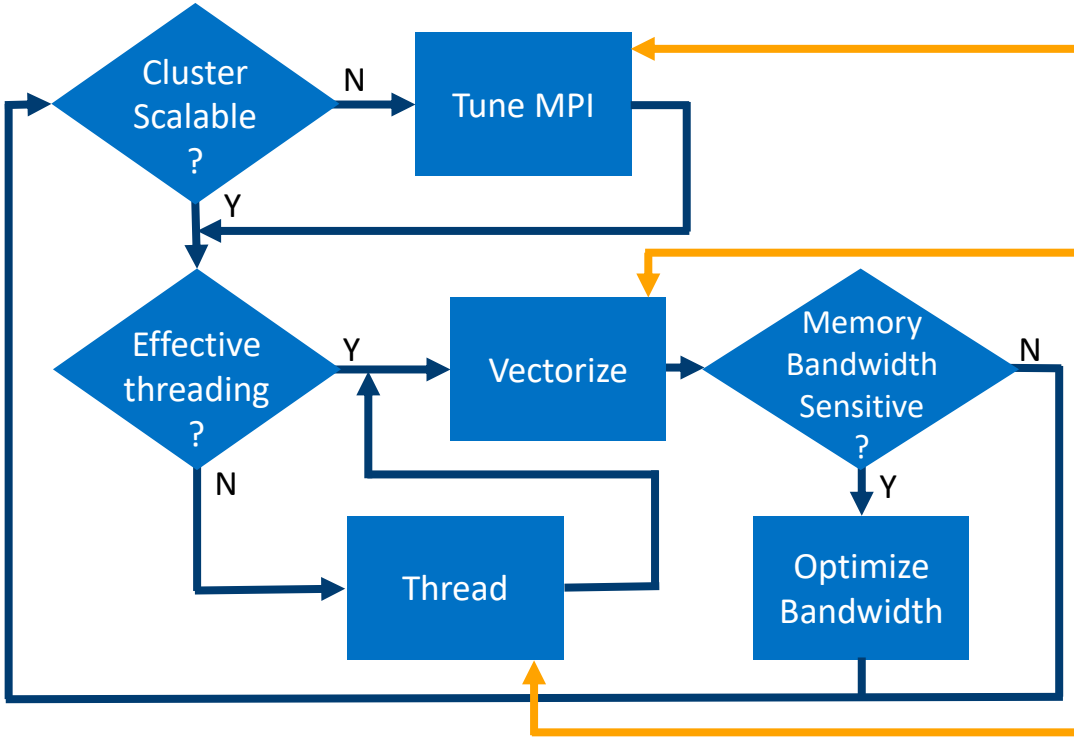
icc -std=c99 -O2 -fp-model precise –qopenmp-simd -qopt-report-file=stderr reduce.c;
LOOP BEGIN at reduce.c(18,6)
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED

AGENDA

- Compiler Basic Optimizations
- Vectorization Essentials
  - Introduction
  - Auto-vectorization of Intel Compilers
  - Reasons for Vectorization Failures and Inefficiency
    - Data Dependence
    - Alignment
    - Unsupported Loop Structure
    - Non-Unit Stride Access
    - Mathematical Functions
- Profilers
  - Intel® Advisor
  - Intel® VTune Amplifier

TOOLS FOR HIGH PERFORMANCE IMPLEMENTATION

INTEL® PARALLEL STUDIO XE



Intel® MPI Library
Intel® MPI Benchmarks

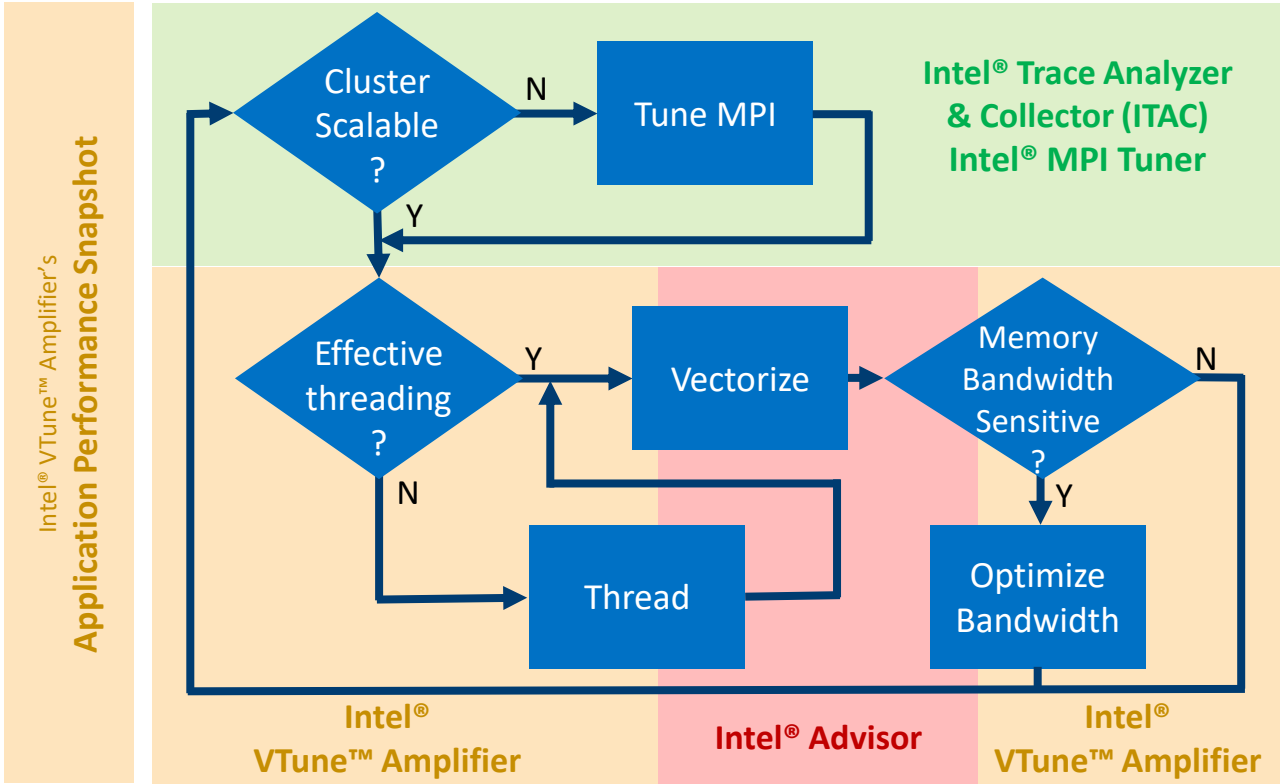
Intel® Compiler

Intel® Math Kernel Library
Intel® IPP – Media & Data Library
Intel® Data Analytics Library
Intel® OpenMP*

Intel® TBB – Threading Library

PERFORMANCE ANALYSIS TOOLS FOR DIAGNOSIS

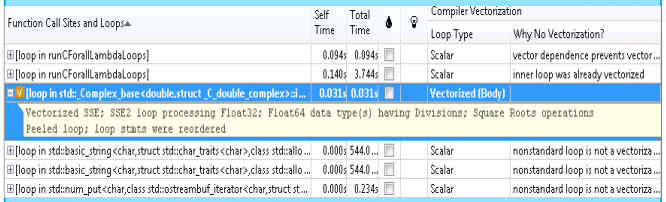
INTEL® PARALLEL STUDIO XE



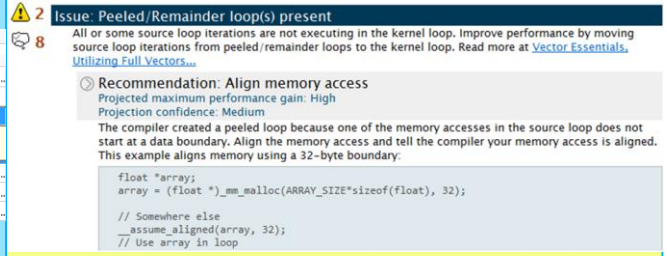
4 STEPS TO EFFICIENT VECTORIZATION

INTEL® ADVISOR – VECTORIZATION ADVISOR

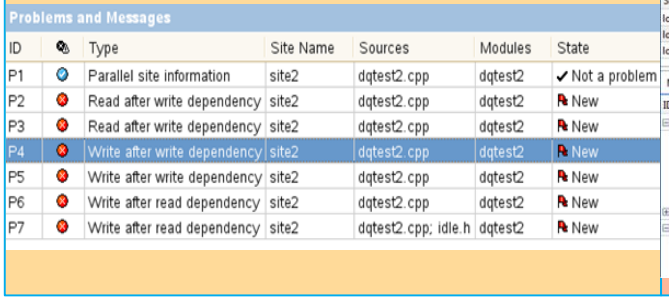
1. Compiler diagnostics + Performance Data + SIMD efficiency information



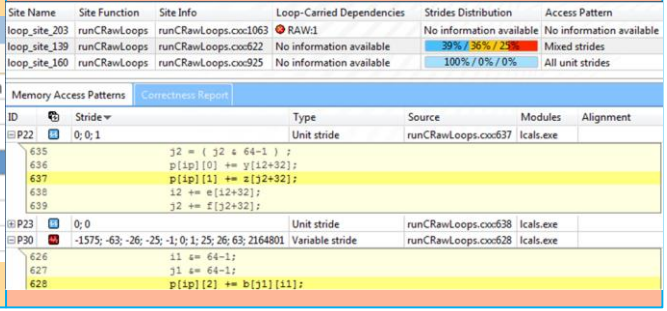
2. Guidance: detect problem and recommend how to fix it



3. Loop-Carried Dependency Analysis



4. Memory Access Patterns Analysis



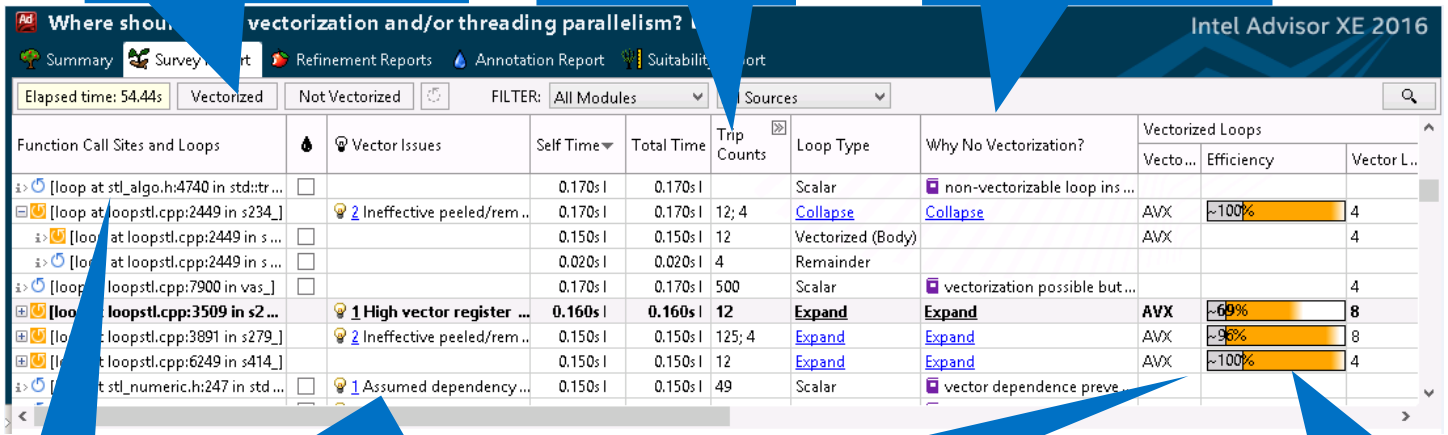
THE RIGHT DATA AT YOUR FINGERTIPS

GET ALL THE DATA YOU NEED FOR HIGH IMPACT VECTORIZATION

Filter by which loops are vectorized!

Trip Counts

What prevents vectorization?



Focus on hot loops

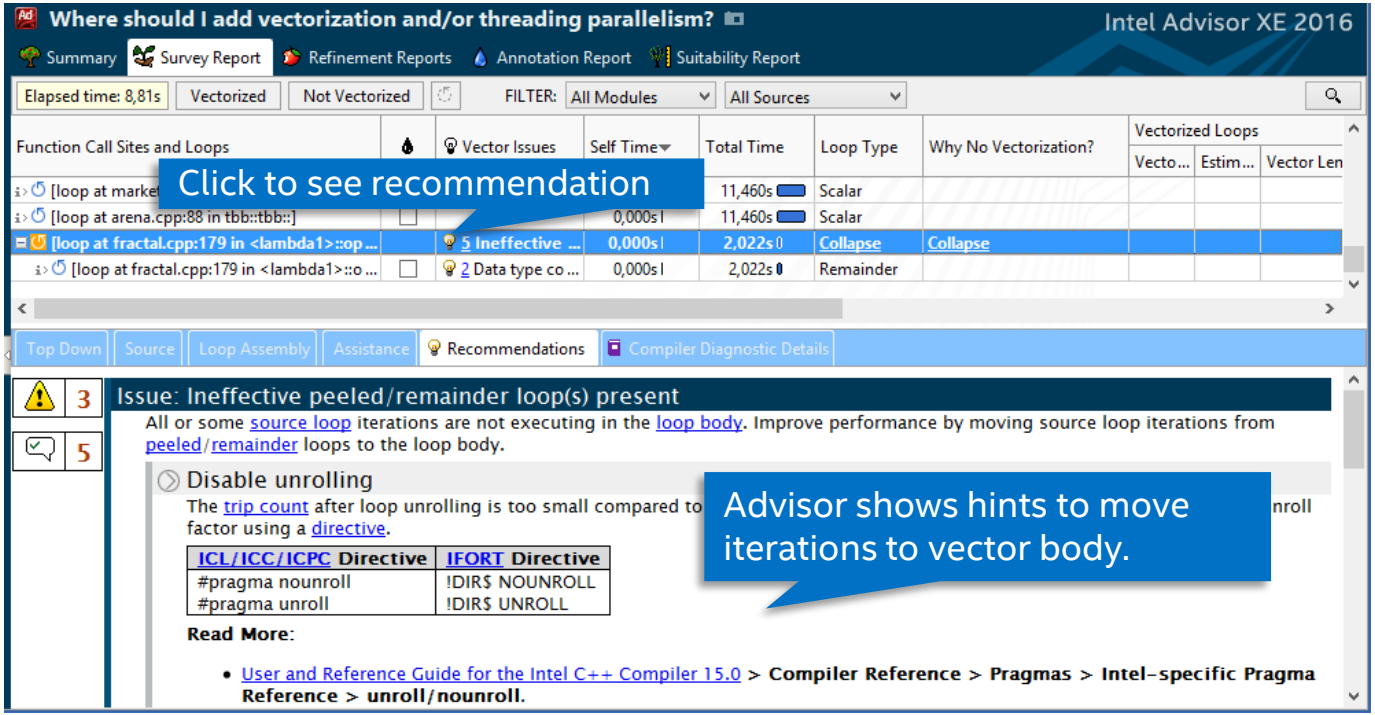
What vectorization issues do I have?

Which Vector instructions are being used?

How efficient is the code?

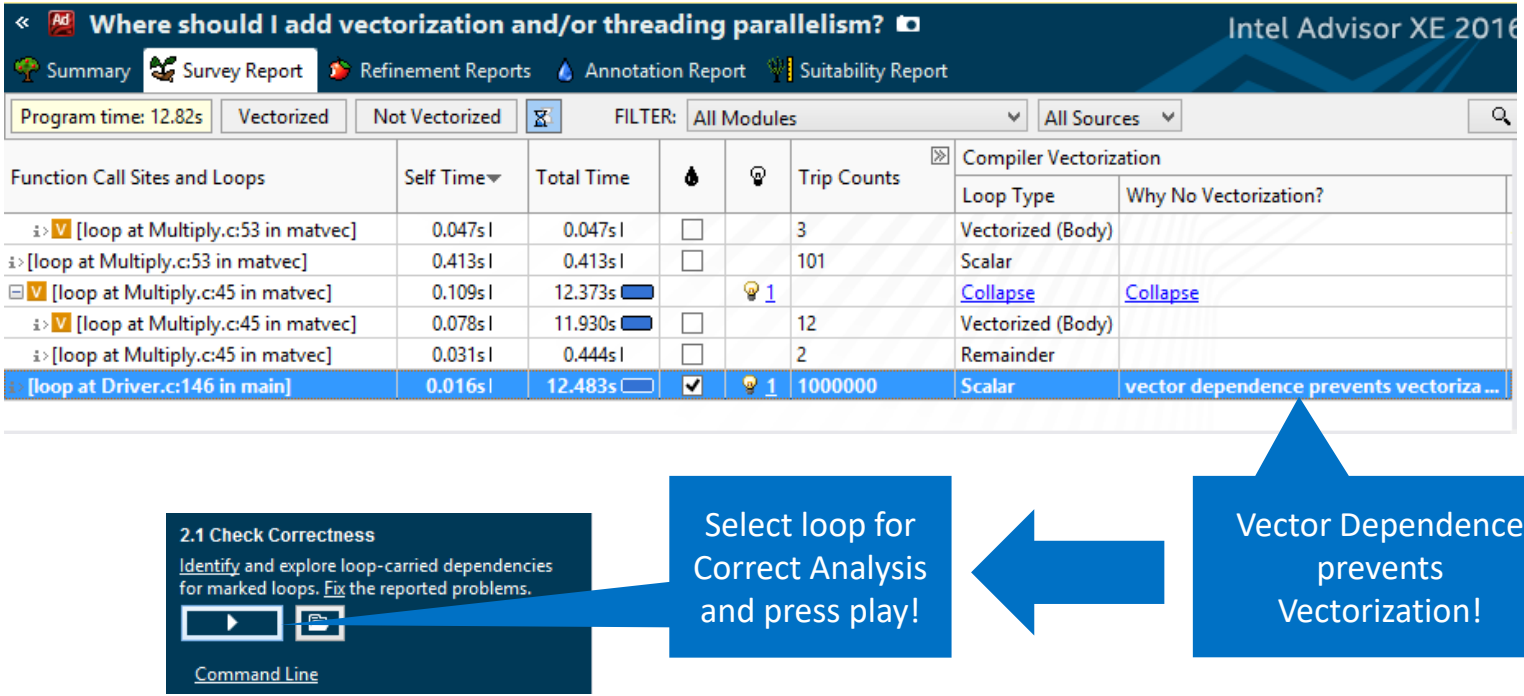
Get Faster Code Faster!

GET SPECIFIC ADVICE FOR IMPROVING VECTORIZATION



IS IT SAFE TO VECTORIZE?

LOOP-CARRIED DEPENDENCIES ANALYSIS VERIFIES CORRECTNESS



DATA DEPENDENCIES – TOUGH PROBLEM #1

IS IT SAFE TO FORCE THE COMPILER TO VECTORIZE?

```
for (i=0;i<N;i++)          // Loop carried dependencies!
    A[i] = A[i-1]*C[i]; // Need to check if it is safe to force
                           // the compiler to vectorize!
```

Issue: Assumed dependency present

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

Enable vectorization

Potential performance gain: Information not available until Beta Update release

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

The Correctness analysis shows there is no real dependency in the loop for the given workload. Tell the compiler it is safe to vectorize using the `restrict` keyword or a directive.

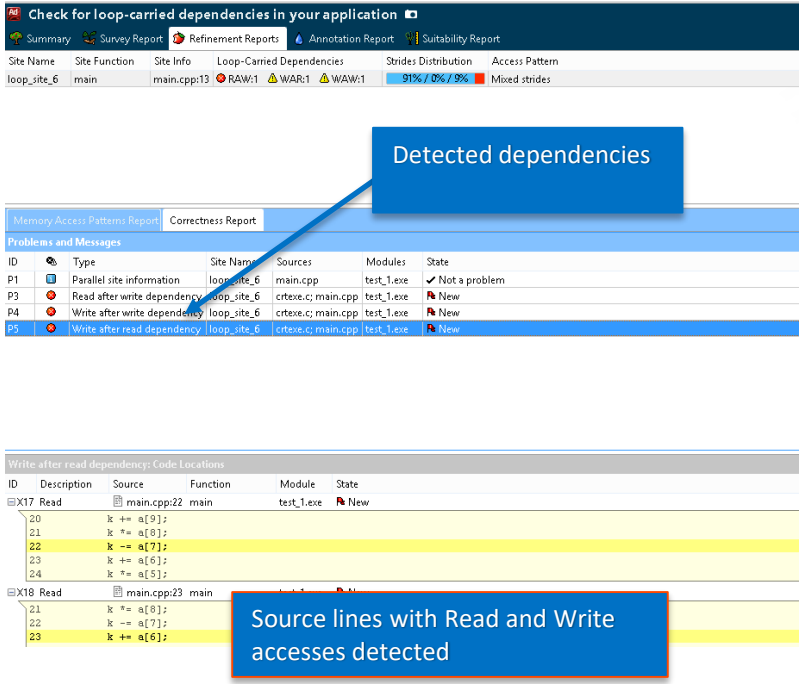
ICL/ICC/ICPC Directive	IFORT Directive	Outcome
#pragma simd or #pragma omp simd	!DIR$ SIMD or !$OMP SIMD	Ignores all dependencies in the loop
#pragma ivdep	!DIR$ IVDEP	Ignores only vector dependencies (which is safest)

Read More:

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

CORRECTNESS – IS IT SAFE TO VECTORIZE?

LOOP-CARRIED DEPENDENCIES ANALYSIS



Received recommendations to force vectorization of a loop:

1. Mark-up loop and check for REAL dependencies
2. Explore dependencies with code snippets

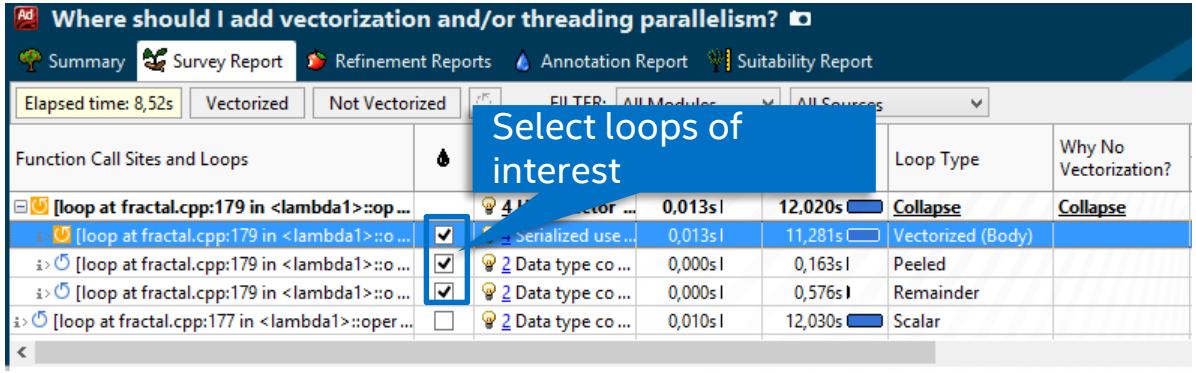
In this example 3 dependencies were detected:

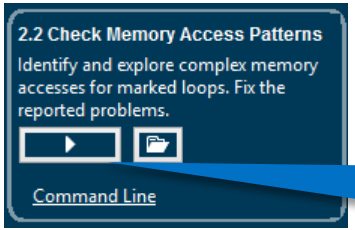
- RAW – Read After Write
- WAR – Write After Read
- WAW – Write After Write

This is NOT a good candidate to force vectorization!

IMPROVE VECTORIZATION

MEMORY ACCESS PATTERN ANALYSIS



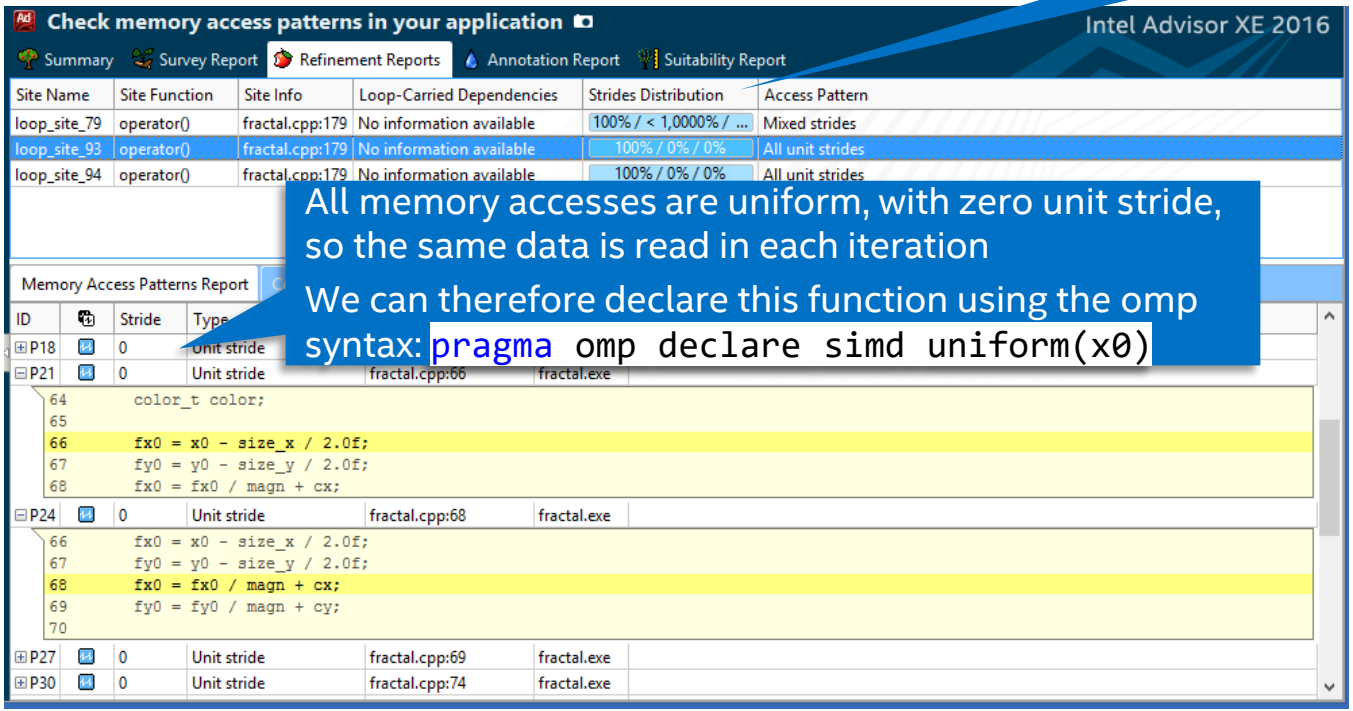


Run Memory Access Patterns analysis, just to check how memory is used in the loop and the called function

FIND VECTOR OPTIMIZATION OPPORTUNITIES

MEMORY ACCESS PATTERN ANALYSIS

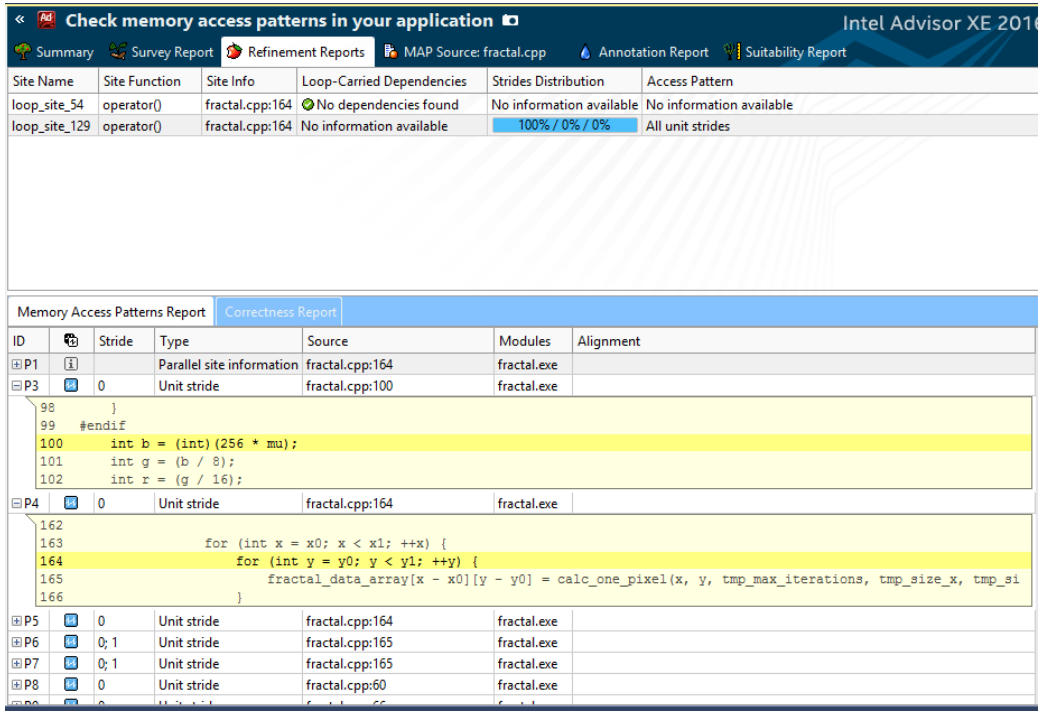
Stride distribution



QUICKLY FIND LOOPS WITH NON-OPTIMAL STRIDE

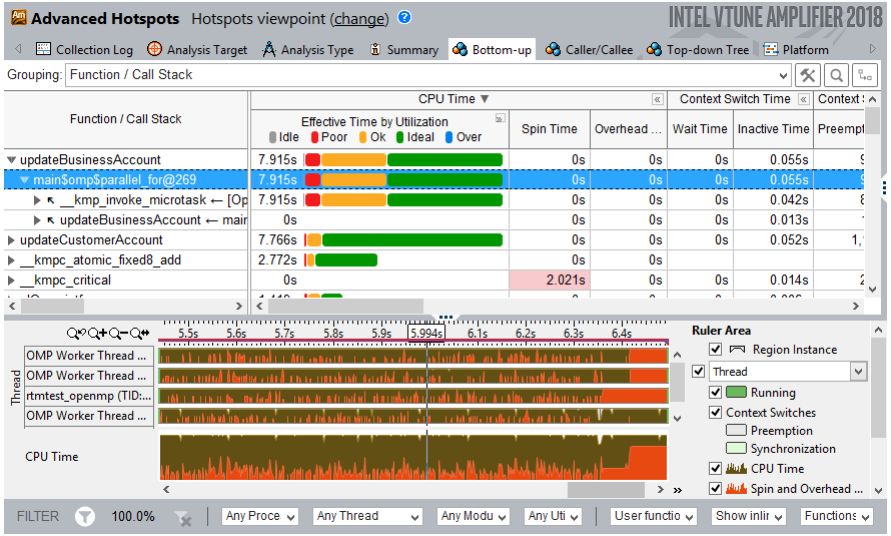
MEMORY ACCESS PATTERN ANALYSIS

- Quickly identify loops that are good, bad or mixed.
- Unit stride memory accesses are preferable.
- Find unaligned data



INTEL® VTUNE™ AMPLIFIER - PERFORMANCE PROFILER

ANALYZE & TUNE APPLICATION PERFORMANCE & SCALABILITY



- Faster, Scalable Code, Faster
  - Accurately profile C, C++, Fortran*, Python*, Go*, Java*, or any mix
  - Optimize CPU/GPU, threading, memory, cache, MPI, storage & more
  - Save time: rich analysis leads to insight
- New for 2018! (Partial List)
  - Easier Profiling of Remote Linux Systems
  - More embedded & real-time OSs (see release notes)
  - OpenCL™ kernel hotspot analysis

RICH SET OF PROFILING FEATURES FOR MULTIPLE MARKETS

INTEL® VTUNE™ AMPLIFIER PERFORMANCE PROFILER

Basic Profiling
- Hotspots

Threading Analysis
- Concurrency, Locks & Waits
- OpenMP, Intel® TBB

Micro Architecture Analysis
- Cache, branch prediction, …

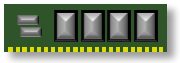
Vectorization (+ Intel® Advisor)
- FLOPS estimates

MPI (+ Intel® Trace Analyzer & Collector)
- Scalability, imbalance, overhead





- Use Memory Efficiently
  - Tune data structures & NUMA
- Optimize for High Speed Storage
  - I/O and compute imbalance
- Easy OpenCL* & GPU Analysis
  - Summary + extended counters
- Intel Media SDK Integration
  - Meaningful media stack metrics
- Low Overhead Java*, Python, Go
  - Managed + native code
- Containers
  - Docker*, Mesos*, LXC*




A RICH SET OF PERFORMANCE DATA
INTEL® VTUNE™ AMPLIFIER

Software Collector	Hardware Collector
Basic Hotspots Which functions use the most time?	**Advanced Hotspots** Which functions use the most time? Where to inline? – Statistical call counts
Concurrency Tune parallelism. Colors show number of cores used.	**General Exploration** Where is the biggest opportunity? Cache misses? Branch mispredictions?
Locks and Waits Tune the #1 cause of slow threaded performance: – waiting with idle cores.	**Advanced Analysis** Dig deep to tune access contention, etc.
Any IA86 processor, any VM, no driver	Higher res., lower overhead, system wide

No special recompiles

FIND ANSWERS FAST

INTEL® VTUNE™ AMPLIFIER

Adjust Data Grouping

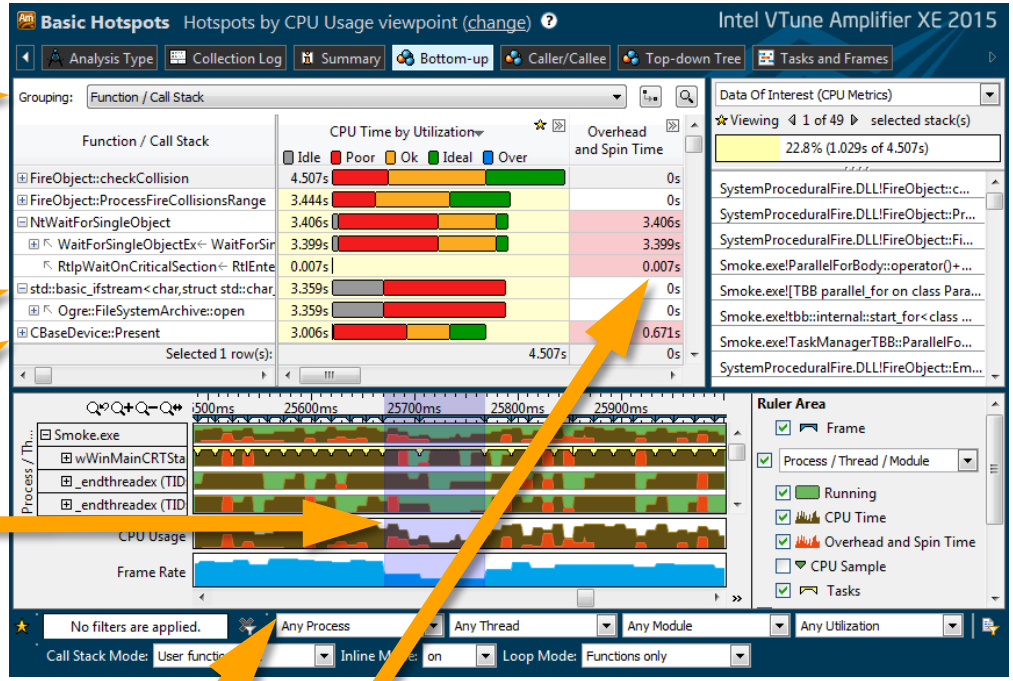
Function - Call Stack
Module - Function - Call Stack
Source File - Function - Call Stack
Thread - Function - Call Stack
... (Partial list shown)

Double Click Function to View Source

Click [+] for Call Stack

Filter by Timeline Selection (or by Grid Selection)

Zoom In And Filter On Selection
Filter In by Selection
Remove All Filters



Filter by Process & Other Controls

Tuning Opportunities Shown in Pink. Hover for Tips

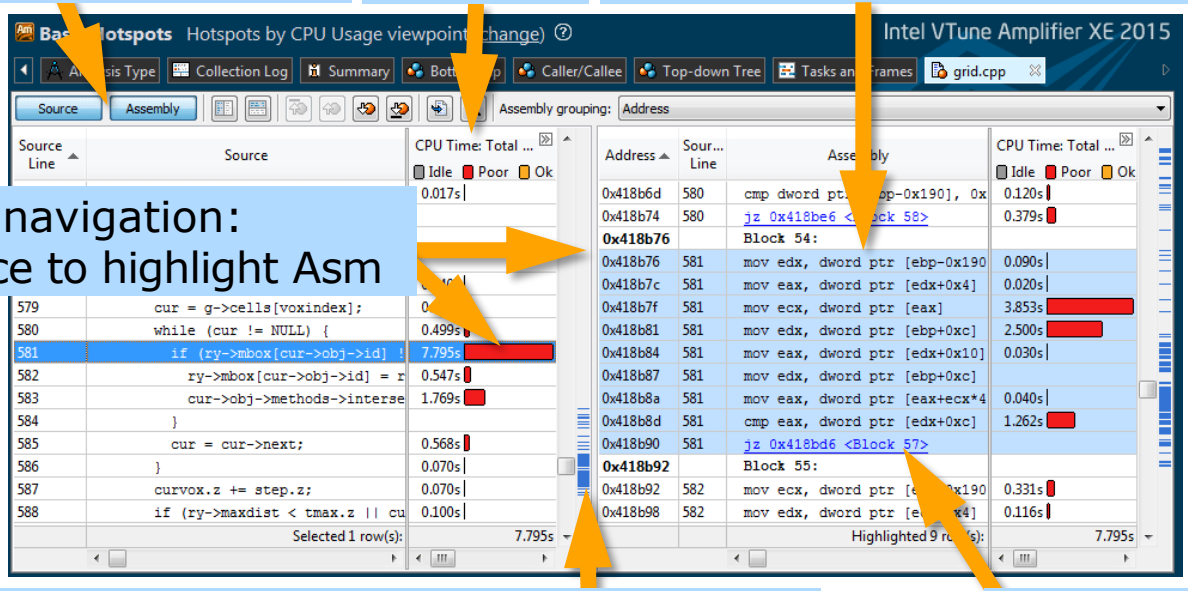
SEE PROFILE DATA ON SOURCE / ASM

DOUBLE CLICK FROM GRID OR TIMELINE

View Source / Asm or both

CPU Time

Right click for instruction reference manual



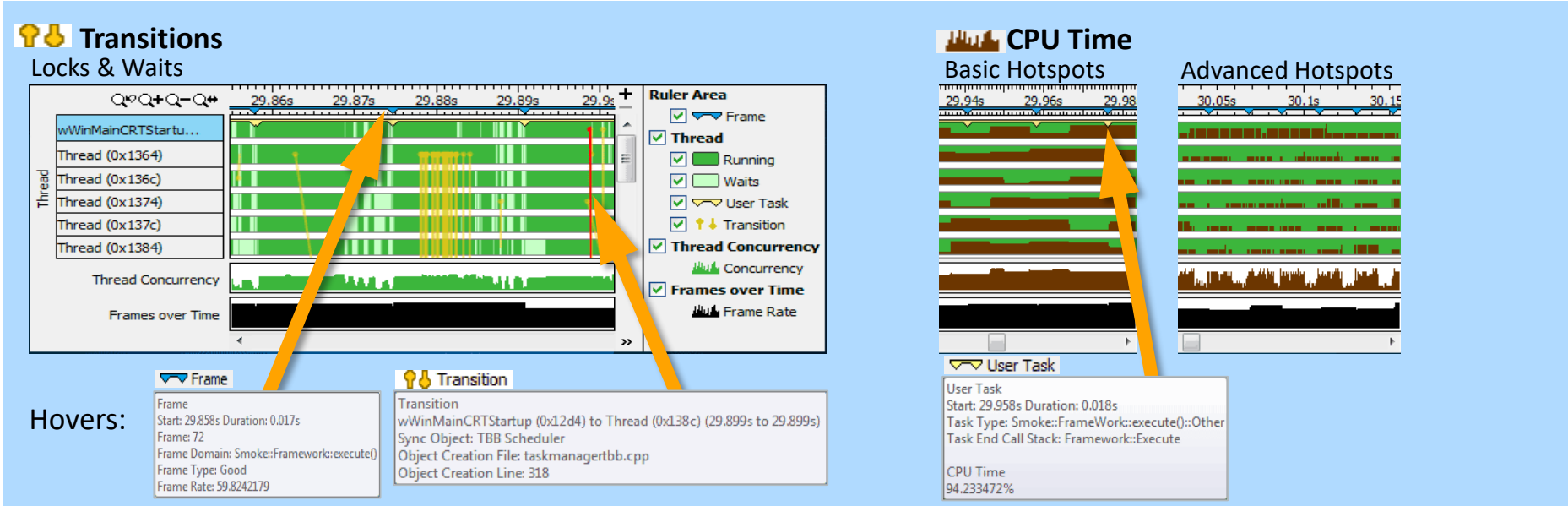
Quick Asm navigation:
Select source to highlight Asm

Scroll Bar “Heat Map” is an overview of hot spots

Click jump to scroll Asm

TIMELINE VISUALIZES THREAD BEHAVIOR

INTEL® VTUNE™ AMPLIFIER



- Optional: Use API to mark frames and user tasks 

- Optional: Add a mark during collection 

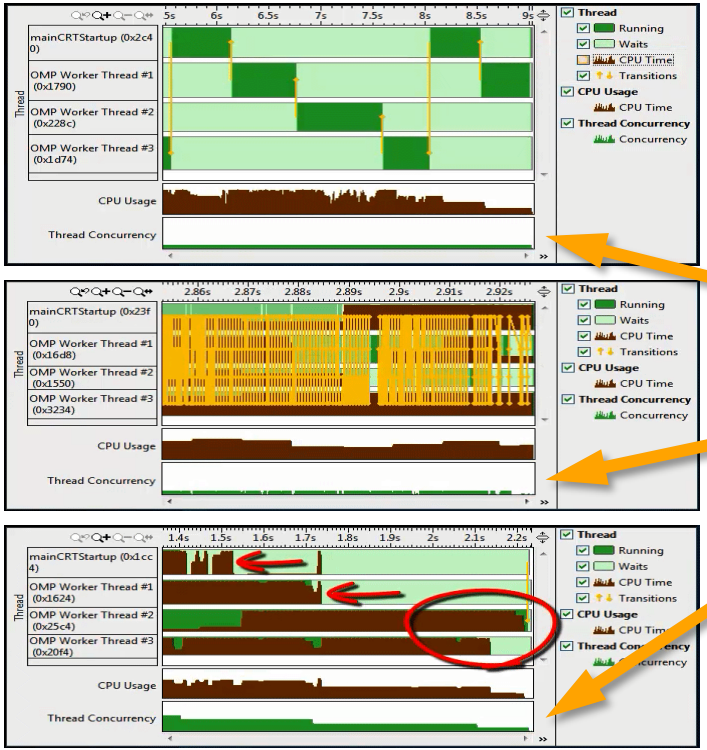
VISUALIZE PARALLEL PERFORMANCE ISSUES

LOOK FOR COMMON PATTERNS

Coarse Grain Locks

High Lock Contention

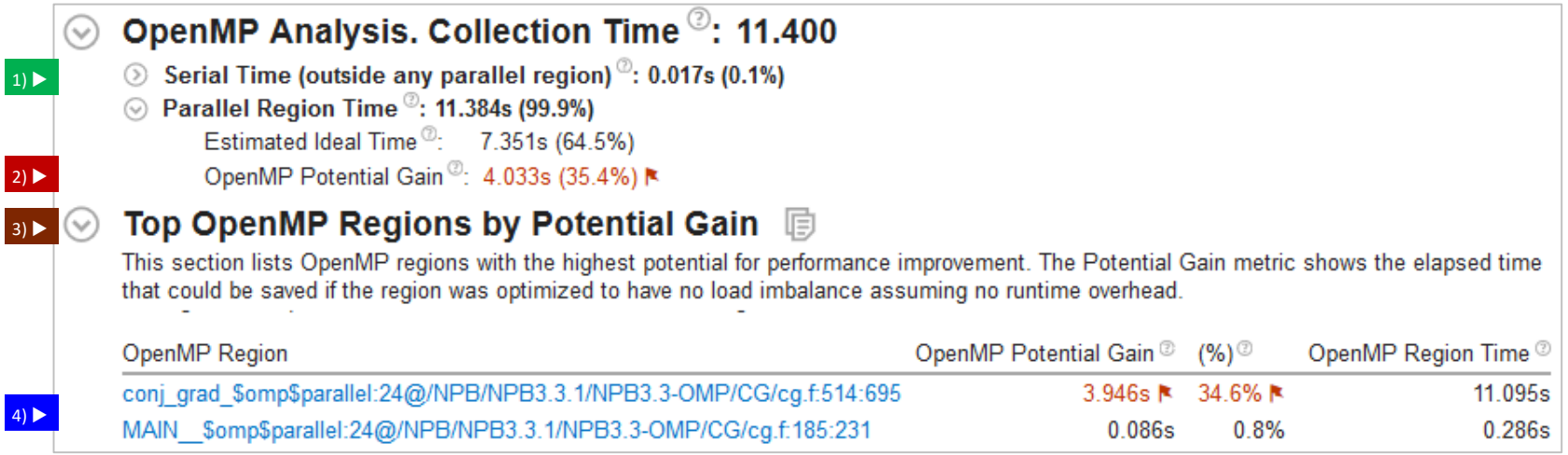
Load Imbalance



Low Concurrency

TUNE OPENMP FOR EFFICIENCY AND SCALABILITY

FAST ANSWERS: IS MY OPENMP SCALABLE? HOW MUCH FASTER COULD IT BE?



OpenMP Analysis. Collection Time : 11.400

1) ▶ **Serial Time (outside any parallel region) : 0.017s (0.1%)**

Parallel Region Time : 11.384s (99.9%)

Estimated Ideal Time : 7.351s (64.5%)

2) ▶ OpenMP Potential Gain : 4.033s (35.4%)

3) ▶ **Top OpenMP Regions by Potential Gain**

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

OpenMP Region	OpenMP Potential Gain	(%)	OpenMP Region Time
4) ▶ conj_grad_ompparallel:24@/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:514:695	3.946s	34.6%	11.095s
MAIN__ompparallel:24@/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:185:231	0.086s	0.8%	0.286s

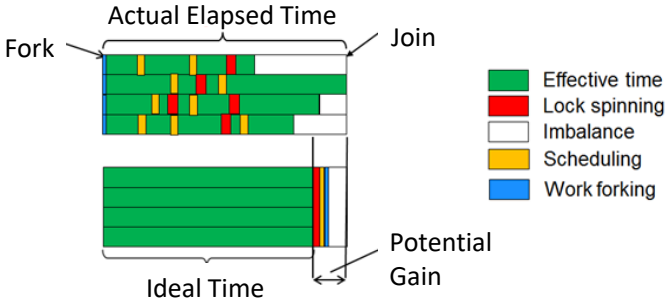
The summary view shown above gives fast answers to four important OpenMP tuning questions:

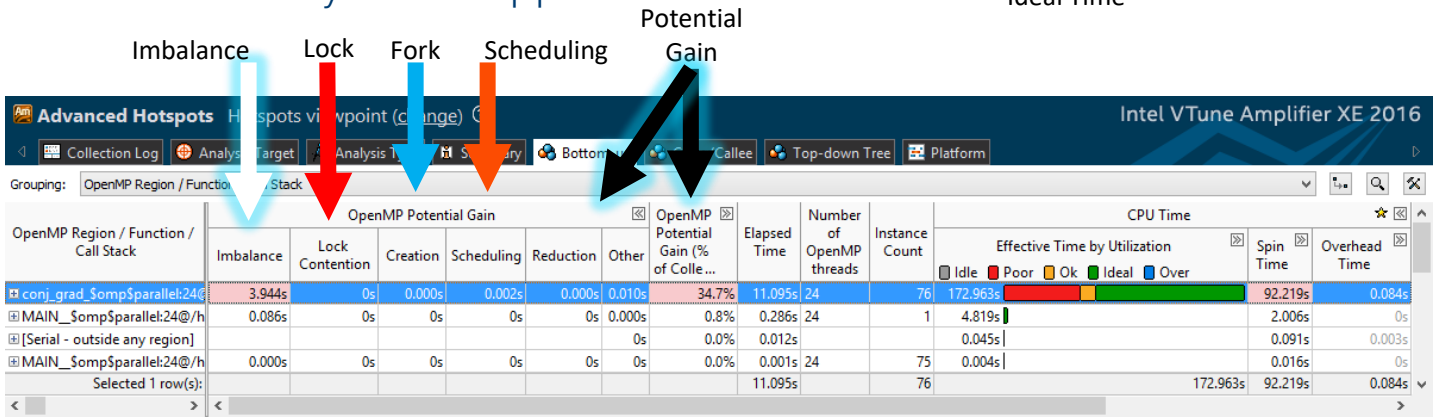
1) Is the serial time of my application significant enough to prevent scaling?
2) How much performance can be gained by tuning OpenMP?
3) Which OpenMP regions / loops / barriers will benefit most from tuning?
4) What are the inefficiencies with each region? (click the link to see details)

Tune OpenMP for Efficiency and Scalability

See the wall clock impact of inefficiencies, identify their cause

- Focus On What's Important
  - What region is inefficient?
  - Is the potential gain worth it?
  - Why is it inefficient? Imbalance? Scheduling? Lock spinning?
  - Intel® Xeon Phi systems supported





QUESTIONS?

Q&A

LEGAL DISCLAIMER & OPTIMIZATION NOTICE

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

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

Optimization Notice

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

Notice revision #20110804


intel
Software