Intel's Optimizing Compiler

Kenneth Craft

Compiler Technical Consulting Engineer

Intel® Corporation

03-06-2018

Agenda

Introduction to Intel® Compiler

Vectorization Basics

Optimization Report

Floating Point Model

Explicit Vectorization

Basic Optimizations with icc/ifort -O…

-O0 no optimization; sets -g for debugging

-O1 scalar optimizations

- Excludes optimizations tending to increase code size

-O2 **default** (except with -g)

- includes **auto-vectorization**; some loop transformations such as unrolling; inlining within source file;
- Start with this (after initial debugging at -O0)

-O3 more aggressive loop optimizations

- Including cache blocking, loop fusion, loop interchange, …
- May not help all applications; need to test

-qopt-report [=0-5]

- Generates compiler optimization reports in files *.optrpt

Common Optimization Options

	Windows*	Linux*, OS X*
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

Interprocedural Optimizations (IPO)

Multi-pass Optimization

- Interprocedural optimizations performs a static, topological analysis of your application!
- ip: Enables inter-procedural optimizations for current source file compilation
- ipo: Enables inter-procedural optimizations across files
  - Can inline functions in separate files
  - Especially many small utility functions benefit from IPO

Windows*	Linux*
/Qip	-ip
/Qipo	-ipo

Enabled optimizations:

- Procedure inlining (reduced function call overhead)
- Interprocedural dead code elimination, constant propagation and procedure reordering
- Enhances optimization when used in combination with other compiler features
- Much of ip (including inlining) is enabled by default at option O2

Profile-Guided Optimizations (PGO)

Static analysis leaves many questions open for the optimizer like:

- How often is x > y
- What is the size of count
- Which code is touched how often

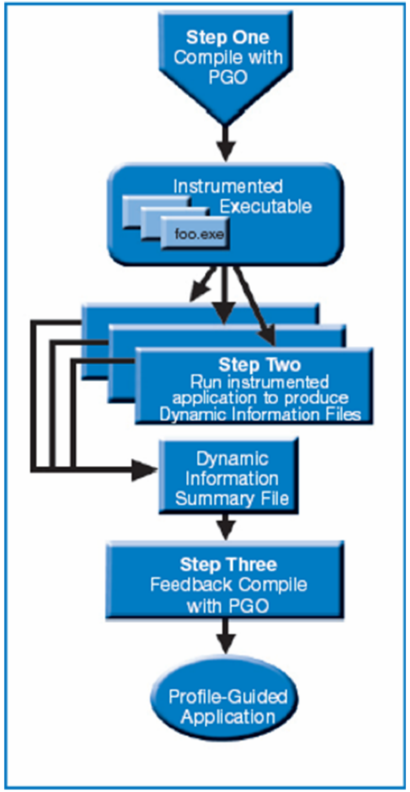
```
if (x > y)
        do_this();
 else
        do_that();
```

```
for(i=0; i<count; ++I
do_work();
```

Use execution-time feedback to guide (final) optimization

Enhancements with PGO:

- More accurate branch prediction
- Basic block movement to improve instruction cache behavior
- Better decision of functions to inline (help IPO)
- Can optimize function ordering
- Switch-statement optimization
- Better vectorization decisions



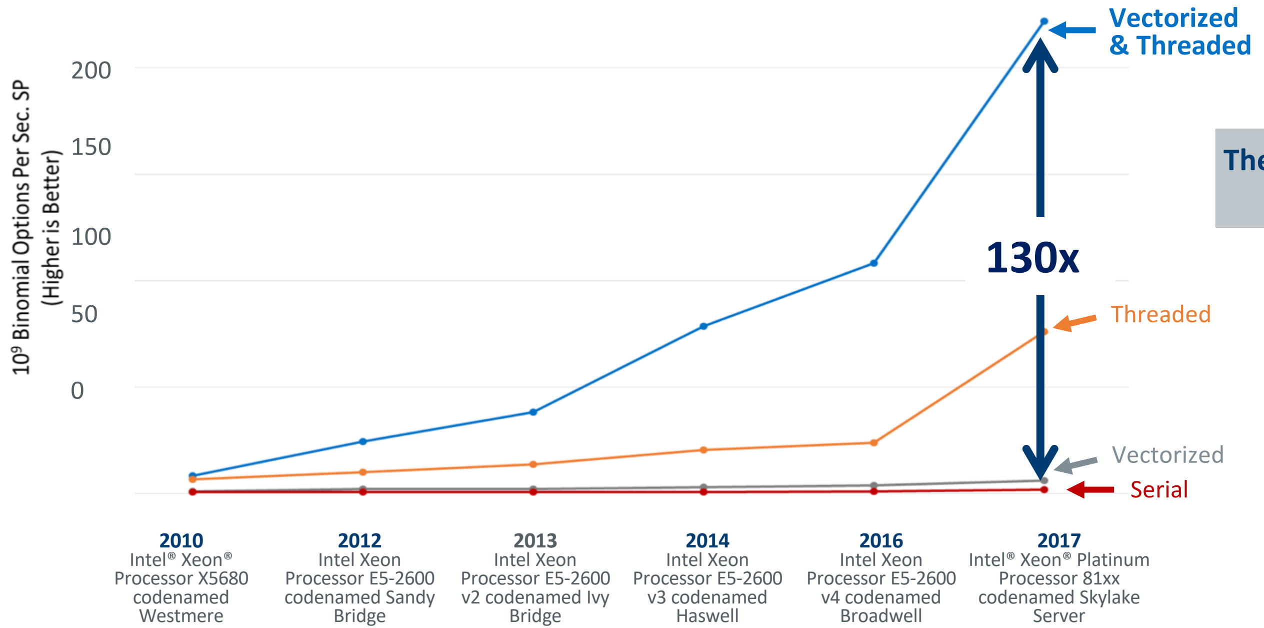
Don’t use a single Vector lane!

Un-vectorized and un-threaded software will under perform

Permission to Design for All Lanes

Threading and Vectorization needed to fully utilize modern hardware

Vectorize and Thread for Performance Boost

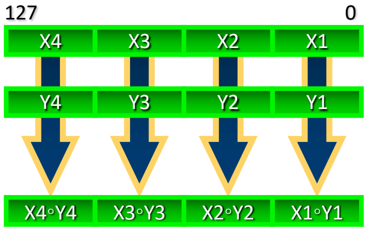


The Difference Is Growing with Each New Generation of Hardware

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 information go to http://www.intel.com/performance. Configurations for 2007-2016 Benchmarks at the end of this presentation

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, & SSSE3 instruction sets & 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

SIMD Types for Intel® Architecture



SSE

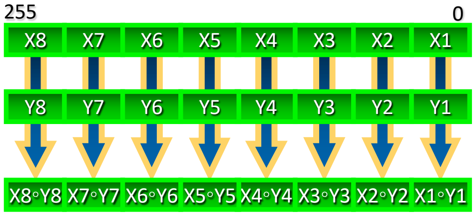
Vector size: **128 bit**

Data types:

8, 16, 32, 64 bit integer

32 and 64 bit float

VL: 2, 4, 8, 16



AVX

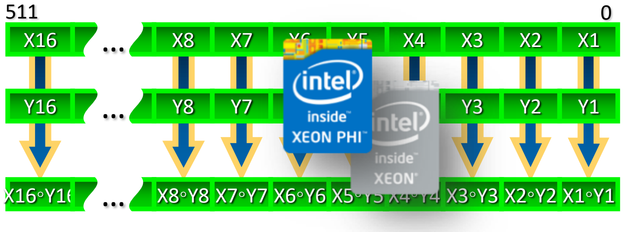
Vector size: **256 bit**

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:

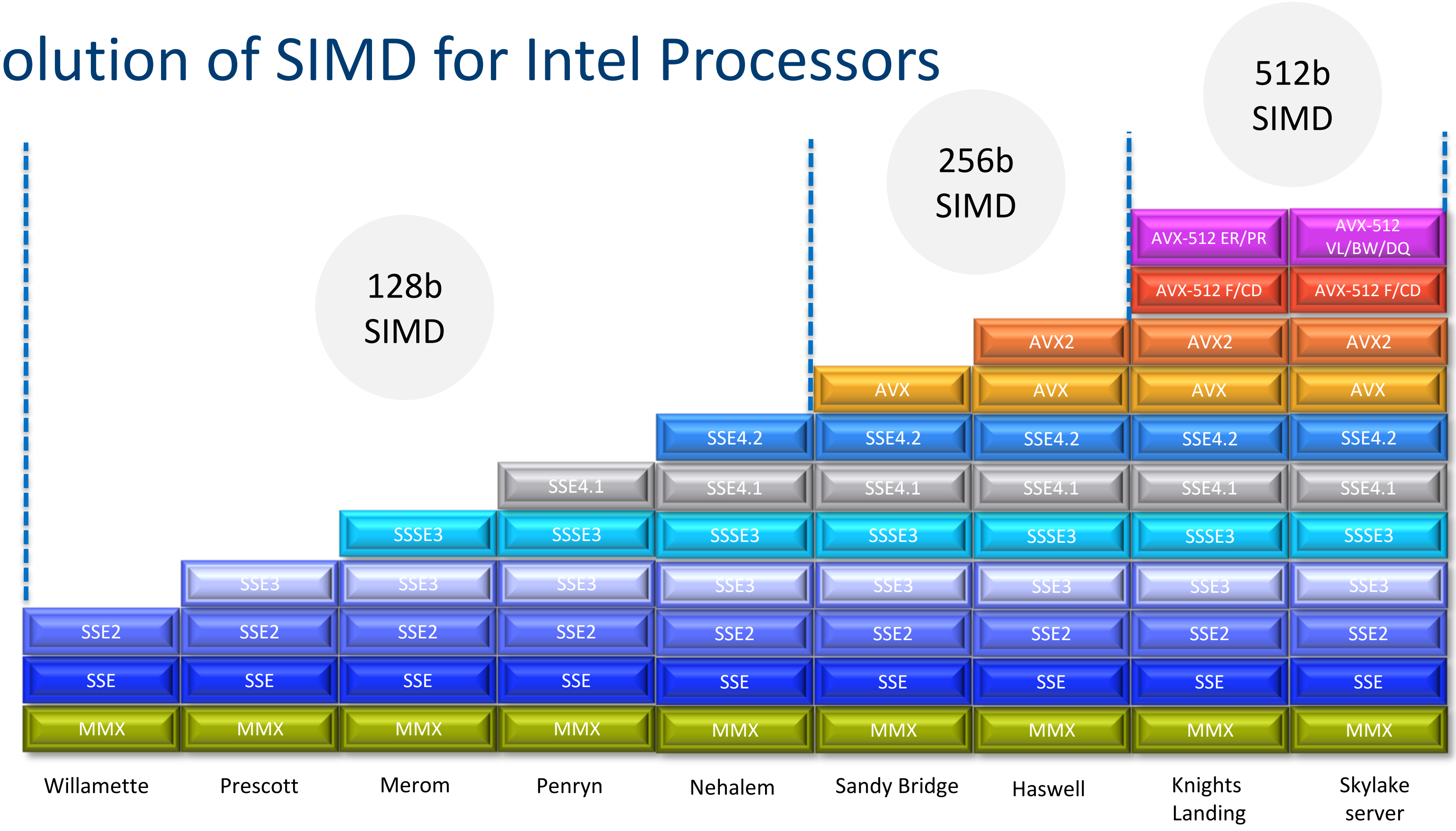
8, 16, 32, 64 bit integer

32 and 64 bit float

VL: 8, 16, 32, 64

Illustrations: Xi, Yi & results 32 bit integer

Evolution of SIMD for Intel Processors



Math Libraries

icc (ifort) comes with optimized math libraries

- libimf (scalar; faster than GNU libm)   and libsvml (vector)
- Driver links libimf automatically, ahead of libm
- More functionality   (replace math.h by mathimf.h for C)
- Optimized paths for Intel® AVX2 and Intel® AVX-512 (detected at run-time)

Don't link to libm explicitly!   -lm

- May give you the slower libm functions instead
- Though the Intel driver may try to prevent this
- GCC needs -lm, so it is often found in old makefiles

Options to control precision and "short cuts" for vectorized math library:

- -fimf-precision = < high | **medium** | low >
- -fimf-domain-exclusion = < mask >
  - Library need not check for special cases (∞, nan, singularities)

Agenda

Introduction to Intel® Compiler

Vectorization Basics

Optimization Report

Explicit Vectorization

Auto-vectorization of Intel Compilers


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

```fortran
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

```asm
.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

```asm
.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, SANDYBRIDGE, SILVERMONT, SKYLAKE, SKYLAKE-AVX512

SIMD extensions: COMMON-AVX512, MIC-AVX512, CORE-AVX512, CORE-AVX2, CORE-AVX-I, AVX, SSE4.2, etc.

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)

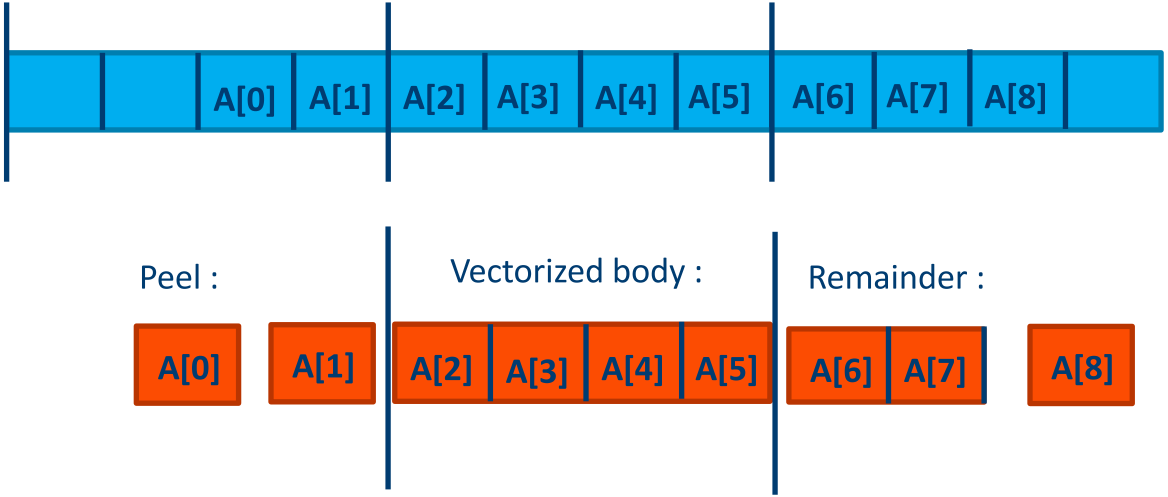
For 32 bit compilation, **`–mia32`** (**`/arch:ia32`**) can be used in case target processor does not support Intel® SSE2 (e.g. Intel® Pentium® 3 or older)

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

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

How to Align Data (Fortran)

Align array on an "n"-byte boundary (n must be a power of 2)

```
!dir$ attributes align:n :: array
```

- Works for dynamic, automatic and static arrays (not in common)

For a 2D array, choose column length to be a multiple of n,
so that consecutive columns have the same alignment (pad if necessary)

`-align array32byte` compiler tries to align all array types

And tell the compiler…

```
!dir$ vector aligned  OR
!$omp simd aligned( var [,var…]:<n>)
```

- Asks compiler to vectorize, assuming all array data accessed in loop are aligned for targeted processor
  - May cause fault if data are not aligned

```
!dir$ assume_aligned array:n   [,array2:n2, …]
```

- Compiler may assume array is aligned to n byte boundary
  - Typical use is for dummy arguments
  - Extension for allocatable arrays in next compiler version

n=16 for Intel® SSE, n=32 for Intel® AVX, n=64 for Intel® AVX-512

How to Align Data (C/C++)

Allocate memory on heap aligned to n byte boundary:

```
void* _mm_malloc(int size, int n)
int posix_memalign(void **p, size_t n, size_t size)
void* aligned_alloc(size_t alignment, size_t size)        (C11)
#include <aligned_new>                                    (C++11)
```

Alignment for variable declarations:

```
__attribute__((aligned(n)))  var_name      or
__declspec(align(n))  var_name
```

And tell the compiler…

```
#pragma vector aligned
```

- Asks compiler to vectorize, overriding cost model, and assuming all array data accessed in loop are aligned for targeted processor
- May cause fault if data are not aligned

```
__assume_aligned(array, n)
```

- Compiler may assume array is aligned to n byte boundary

n=64 for Intel® Xeon Phi™ coprocessors, n=32 for Intel® AVX, n=16 for Intel® SSE

Guidelines for Writing Vectorizable Code

Prefer simple "for" or "DO" loops

Write straight line code. Try to avoid:

- function calls (unless inlined or SIMD-enabled functions)
- branches that can't be treated as masked assignments.

Avoid dependencies between loop iterations

- Or at least, avoid read-after-write dependencies

Prefer arrays to the use of pointers

- Without help, the compiler often cannot tell whether it is safe to vectorize code containing pointers.
- Try to use the loop index directly in array subscripts, instead of incrementing a separate counter for use as an array address.
- Disambiguate function arguments, e.g. -fargument-noalias

Use efficient memory accesses

- Favor inner loops with unit stride
- Minimize indirect addressing a[i] = b[ind[i]]
- Align your data consistently where possible (to 16, 32 or 64 byte boundaries)

Agenda

Introduction to Intel® Compiler

Vectorization Basics

Optimization Report

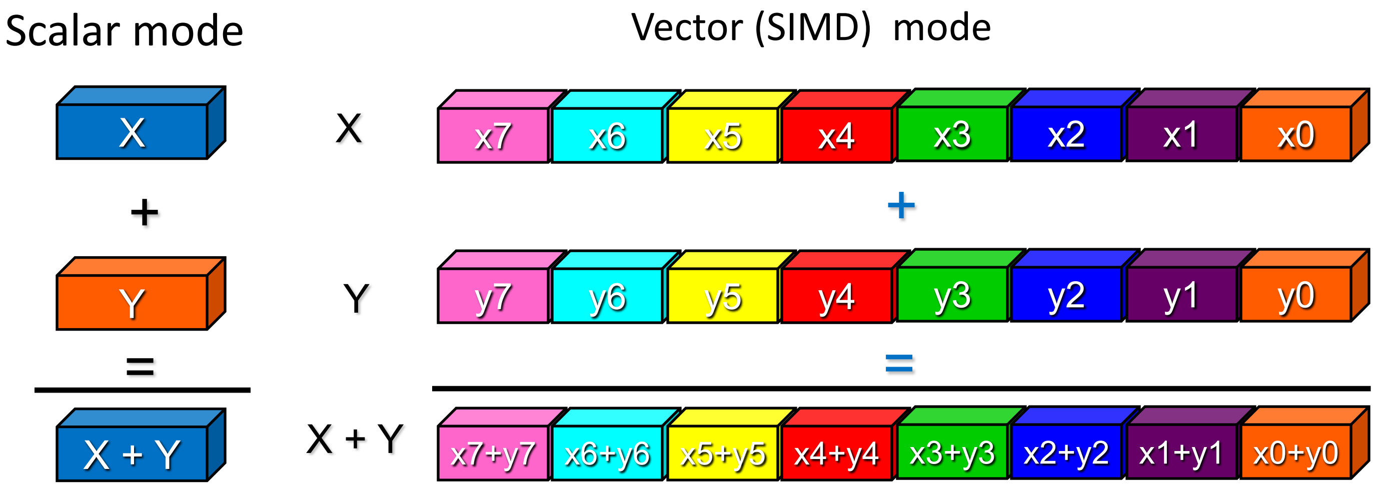
Explicit Vectorization

Limitations of auto-vectorization

Why some loops don't auto-vectorize

Auto-Vectorization Works Great…

for (i=0; i<n; i++) z[i] = x[i] + y[i];



… provided loop is not too complex – compiler must be able to:

- prove safety;
- generate corresponding SIMD code;
- envisage improved performance.

Obstacles to Auto-Vectorization

Multiple loop exits

- Or trip count unknown at loop entry

Dependencies between loop iterations

- Mostly, 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

Example of New Optimization Report

```
$ icc -c -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr foo.c

Begin optimization report for: foo

    Report from: Loop nest & Vector optimizations [loop, vec]

LOOP BEGIN at foo.c(4,3)
Multiversioned v1
   remark #25231: Loop multiversioned for Data Dependence
   remark #15135: vectorization support: reference theta has unaligned access
   remark #15135: vectorization support: reference sth has unaligned access
   remark #15127: vectorization support: unaligned access used inside loop body
   remark #15145: vectorization support: unroll factor set to 2
   remark #15164: vectorization support: number of FP up converts: single to double precision 1
   remark #15165: vectorization support: number of FP down converts: double to single precision 1
   remark #15002: LOOP WAS VECTORIZED
   remark #36066: unmasked unaligned unit stride loads: 1
   remark #36067: unmasked unaligned unit stride stores: 1
   ....  (loop cost summary)  ....
   remark #25018: Estimate of max trip count of loop=32
LOOP END

LOOP BEGIN at foo.c(4,3)
Multiversioned v2
   remark #15006: loop was not vectorized: non-vectorizable loop instance from multiversioning
LOOP END
===========================================================================
```

```c
#include <math.h>
void foo (float * theta, float * sth)  {
  int i;
  for (i = 0; i < 128; i++)
      sth[i] = sin(theta[i]+3.1415927);
}
```

Optimization Report Example

```
$ icc -c  -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr -fargument-noalias foo.c
Begin optimization report for: foo
   Report from: Loop nest & Vector optimizations [loop, vec]
```

(/Qalias-args- on Windows*)

```
LOOP BEGIN at foo.c(4,3)
  remark #15135: vectorization support: reference theta has unaligned access
  remark #15135: vectorization support: reference sth has unaligned access
  remark #15127: vectorization support: unaligned access used inside loop body
  remark #15145: vectorization support: unroll factor set to 2
  remark #15164: vectorization support: number of FP up converts: single to double precision 1
  remark #15165: vectorization support: number of FP down converts: double to single precision 1
  remark #15002: LOOP WAS VECTORIZED
  remark #36066: unmasked unaligned unit stride loads: 1
  remark #36067: unmasked unaligned unit stride stores: 1
  remark #36091: --- begin vector loop cost summary ---
  remark #36092: scalar loop cost: 114
  remark #36093: vector loop cost: 55.750
  remark #36094: estimated potential speedup: 2.040
  remark #36095: lightweight vector operations: 10
  remark #36096: medium-overhead vector operations: 1
  remark #36098: vectorized math library calls: 1
  remark #36103: type converts: 2
  remark #36104: --- end vector loop cost summary ---
  remark #25018: Estimate of max trip count of loop=32
LOOP END
```

```c
#include <math.h>
void foo (float * theta, float * sth)   {
  int i;
  for (i = 0; i < 128; i++)
      sth[i] = sin(theta[i]+3.1415927);
}
```

Optimization Report Example

```
$ icc -c -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr -fargument-noalias foo.c

Begin optimization report for: foo

    Report from: Loop nest & Vector optimizations [loop, vec]

LOOP BEGIN at foo.c(4,3)
remark #15135: vectorization support: reference theta has unaligned access
   remark #15135: vectorization support: reference sth has unaligned access
   remark #15127: vectorization support: unaligned access used inside loop body
   remark #15002: LOOP WAS VECTORIZED
   remark #36066: unmasked unaligned unit stride loads: 1
   remark #36067: unmasked unaligned unit stride stores: 1
   remark #36091: --- begin vector loop cost summary ---
   remark #36092: scalar loop cost: 111
   remark #36093: vector loop cost: 28.000
   remark #36094: estimated potential speedup: 3.950
   remark #36095: lightweight vector operations: 9
   remark #36098: vectorized math library calls: 1
   remark #36104: --- end vector loop cost summary ---
   remark #25018: Estimate of max trip count of loop=32
LOOP END
```

```c
#include <math.h>
void foo (float * theta, float * sth)  {
  int i;
  for (i = 0; i < 128; i++)
     sth[i] = sinf(theta[i]+3.1415927f);
}
```

Optimization Report Example

$ icc -c -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr -fargument-noalias **-xavx** foo.c

Begin report for: foo

Report from: Loop nest & Vector optimizations [loop, vec]

LOOP BEGIN at foo.c(4,3)
remark #15135: vectorization support: **reference theta has unaligned access**
remark #15135: vectorization support: **reference sth has unaligned access**
remark #15127: vectorization support: **unaligned access used inside loop body**
remark #15002: LOOP WAS VECTORIZED
remark #36066: **unmasked unaligned unit stride loads: 1**
remark #36067: **unmasked unaligned unit stride stores: 1**
remark #36091: --- begin vector loop cost summary ---
remark #36092: scalar loop cost: 110
remark #36093: vector loop cost: 15.370
remark #36094: estimated potential speedup: **7.120**
remark #36095: lightweight vector operations: 9
remark #36098: vectorized math library calls: 1
remark #36104: --- end vector loop cost summary ---
remark #25018: Estimate of **max trip count of loop=16**
LOOP END
===

```
#include <math.h>
void foo (float * theta, float * sth)  {
  int i;
  for (i = 0; i < 128; i++)
     sth[i] = sinf(theta[i]+3.1415927f);
}
```

Optimization Report Example

```
$ icc -c -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr -fargument-noalias -xavx foo.c

Begin optimization report for: foo

    Report from: Loop nest & Vector optimizations [loop, vec]

LOOP BEGIN at foo.c(6,3)
remark #15134: vectorization support: reference theta has aligned access
   remark #15134: vectorization support: reference sth has aligned access
   remark #15002: LOOP WAS VECTORIZED
   remark #36064: unmasked aligned unit stride loads: 1
   remark #36065: unmasked aligned unit stride stores: 1
   remark #36091: --- begin vector loop cost summary ---
   remark #36092: scalar loop cost: 110
   remark #36093: vector loop cost: 13.620
   remark #36094: estimated potential speedup: 8.060
   remark #36095: lightweight vector operations: 9
   remark #36098: vectorized math library calls: 1
   remark #36104: --- end vector loop cost summary ---
   remark #25018: Estimate of max trip count of loop=16
LOOP END
===========================================================================
```

```
#include <math.h>
void foo (float * theta, float * sth)  {
  int i;
  __assume_aligned(theta,32);
  __assume_aligned(sth,32);
  for (i = 0; i < 128; i++)
    sth[i] = sinf(theta[i]+3.1415927f);
}
```

Optimization Report Phases

- Enables the optimization report and controls the level of details
  - `/Qopt-report[:n], -qopt-report[=n]`
  - When used without parameters, full optimization report is issued on stdout with details level 2
- Control destination of optimization report
  - `/Qopt-report-file:<filename>, -qopt-report=<filename>`
  - By default, without this option, a <filename>.optrpt file is generated.
- Subset of the optimization report for specific phases only
  - `/Qopt-report-phase[:list], -qopt-report-phase[=list]`

    Phases can be:
    - all – All possible optimization reports for all phases (default)
    - loop – Loop nest and memory optimizations
    - vec – Auto-vectorization and explicit vector programming
    - par – Auto-parallelization
    - openmp – Threading using OpenMP
    - ipo – Interprocedural Optimization, including inlining
    - pgo – Profile Guided Optimization
    - cg – Code generation

Improved Optimization Report

```
subroutine test1(a, b ,c, d)
  integer, parameter        :: len=1024
  complex(8), dimension(len) :: a, b, c
  real(4),    dimension(len) :: d
  do i=1,len
     c(i) = exp(d(i)) +  a(i)/b(i)
  enddo
end
```

From assembly listing:

VECTOR LENGTH 16
MAIN VECTOR TYPE: 32-bits floating point

$ ifort -c -S -xmic-avx512 -O3 -qopt-report=4 -qopt-report-file=stderr -qopt-report-phase=loop,vec,cg -qopt-report-embed test_rpt.f90

- 1 vector iteration comprises
  - 16 floats in a single AVX-512 register   (d)
  - 16 double complex in 4 AVX-512 registers per variable   (a, b, c)
- Replace exp(d(i)) by d(i) and the compiler will choose a vector length of 4
  - More efficient to convert d immediately to double complex

Improved Optimization Report

Compiler options: -c -S -xmic-avx512 -O3 -qopt-report=4 -qopt-report-file=stderr -qopt-report-phase=loop,vec,**cg** -qopt-report-embed

```
…
   remark #15305: vectorization support: vector length 16
   remark #15309: vectorization support: normalized vectorization overhead 0.087
   remark #15417: vectorization support: number of FP up converts: single
                  precision to double precision 1   [ test_rpt.f90(7,6) ]
   remark #15300: LOOP WAS VECTORIZED
   remark #15482: vectorized math library calls: 1
   remark #15486: divides: 1
   remark #15487: type converts: 1
…
```

- New features include the code generation (CG) / register allocation report
  - Includes temporaries; stack variables; spills to/from memory

Annotated Source Listing

Get reports as annotation to source files:

- Linux*, macOS*: **-qopt-report-annotate=[text|html]**,
  Windows*: **/Qopt-report-annotate=[text|html]**

- *.annot file is generated

```
// ------- Annotated listing with optimization reports for "test.cpp" -------
//
     void add(double *A, double *B, double *C, double *D)
     {
          for (int i = 0; i < 1000; i++)
...
//LOOP BEGIN at test.cpp(3,2)
//Multiversioned v1
//test.cpp(3,2):remark #15300: LOOP WAS VECTORIZED
//LOOP END
...
               D[i] = A[i] + B[i]+C[i];
     }
```

Agenda

Introduction to Intel® Compiler

Vectorization Basics

Optimization Report

Explicit Vectorization

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

OpenMP* SIMD directive

Use !$OMP SIMD with -qopenmp-simd

```
subroutine add(A, N, X)
   integer N, X
   real    A(N)

   DO I=X+1, N
      A(I) = A(I) + A(I-X)
   ENDDO
end
```

```
subroutine add(A, N, X)
   integer N, X
   real    A(N)
 !$ OMP SIMD
   DO I=X+1, N
      A(I) = A(I) + A(I-X)
   ENDDO
end
```

loop was not vectorized:
existence of vector dependence.

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 — like OpenMP for threading
- REDUCTION
- 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

Optimization Notice

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
```

Outer Loop Vectorization - performance

Optimization Options	Speed-up	What's going on
-O1 -xavx	1.0	No vectorization
-O2 -xavx	1.5	Inner loop vectorization
-O2 -xavx -qopenmp-simd	3.5	Outer loop vectorization unknown stride
-O2 -xavx -qopenmp-simd -DKNOWN_TRIP_COUNT	6.5	Inner loop fully unrolled known outer loop stride
-O2 -xcore-avx2 -qopenmp-simd -DKNOWN_TRIP_COUNT	7.4	+ Intel® AVX2 including FMA instructions

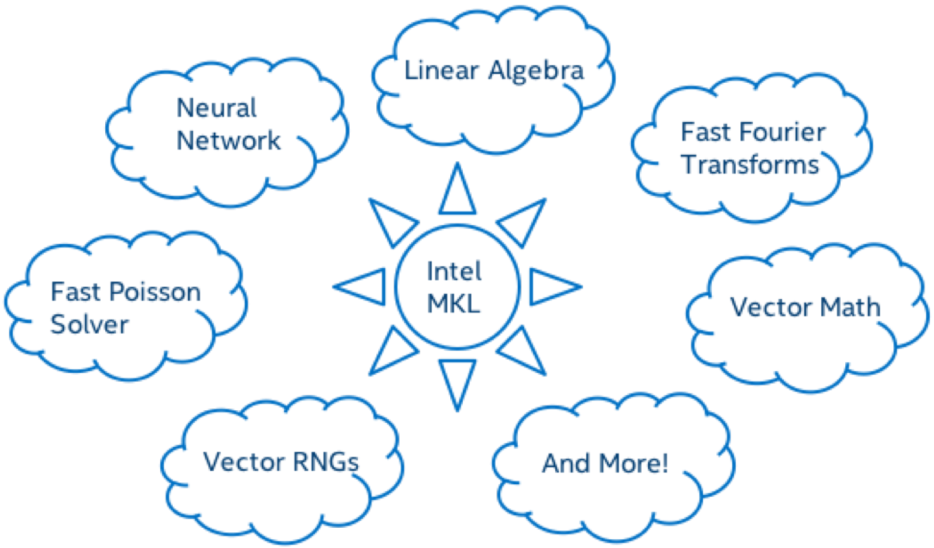
Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary.
The results above were obtained on a 4th Generation Intel® Core™ i7-4790 system, frequency 3.6 GHz, running Ubuntu* version 14.04.5 and using the Intel® C++ Compiler version 18.0.

Outer Loop Vectorization - performance

Optimization Options	Speed-up	What's going on
-O1 -xavx	1.0	No vectorization
-O2 -xavx	0.94	Inner loop vectorization
-O2 -xavx -qopenmp-simd	2.2	Outer loop vectorization unknown stride
-O2 -xavx -qopenmp-simd -DKNOWN_TRIP_COUNT	4.5	Inner loop fully unrolled known outer loop stride
-O2 -xcore-avx2 -qopenmp-simd -DKNOWN_TRIP_COUNT	4.8	+ Intel® AVX2 including FMA instructions

Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary.
The results above were obtained on a 4th Generation Intel® Core™ i7-4790 system, frequency 3.6 GHz, running Ubuntu* version 14.04.5 and using the Intel® Fortran Compiler version 18.0.

Faster, Scalable Code with Intel® Math Kernel Library



Learn More: software.intel.com/mkl

- Features highly optimized, threaded, and vectorized math functions that maximize performance on each processor family
- Utilizes industry-standard C and Fortran APIs for compatibility with popular BLAS, LAPACK, and FFTW functions—no code changes required
- Dispatches optimized code for each processor automatically without the need to branch code

What's New in Intel® MKL 2018

- Improved small matrix multiplication performance in GEMM and LAPACK
- Improved ScaLAPACK performance for distributed computation
- 24 new vector math functions
- Simplified license for easier adoption and redistribution
- Additional distributions via YUM, APT-GET, and Conda repositories

**Certain technical specifications and select processors/skus apply. See product site for more details.

Faster Machine Learning & Analytics with Intel® DAAL

- Features highly tuned functions for classical machine learning and analytics performance across spectrum of Intel® architecture devices
- Optimizes data ingestion together with algorithmic computation for highest analytics throughput
- Includes Python*, C++, and Java* APIs and connectors to popular data sources including Spark* and Hadoop*
- Free and open source community-supported versions are available, as well as paid versions that include premium support.

What's New in 2018 version

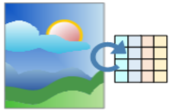
- New Algorithms:
  - Classification & Regression Decision Tree
  - Classification & Regression Decision Forest
  - k-NN
  - Ridge Regression
- Spark* MLlib-compatible API wrappers for easy substitution of faster Intel DAAL functions
- Improved APIs for ease of use
- Repository distribution via YUM, APT-GET, and Conda

Pre-processing → Transformation → Analysis → Modeling → Validation → Decision Making

Pre-processing

Decompression, Filtering, Normalization

Transformation



Aggregation, Dimension Reduction

Analysis

Summary Statistics Clustering, etc.

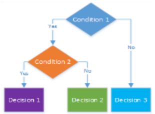
Modeling

Machine Learning (Training) Parameter Estimation Simulation

Validation

Hypothesis testing Model errors

Decision Making



Forecasting Decision Trees, etc.

Optimization Notice

Learn More: software.intel.com/daal

Intel® Integrated Performance Primitives 2018

Highly Optimized Image, Signal & Data Processing Functions

Intel® Integrated Performance Primitives provides developers with ready-to-use, processor optimized functions to accelerate ***Image, Signal, Data Processing & Cryptography computation tasks***

- Multi-core, multi-OS and multi-platform ready, computationally intensive and highly optimized functions
- Plug in and use APIs to quickly improve application performance
- Reduced cost and time-to-market on software development and maintenance
- Access Priority Support, which connects you direct to Intel engineers for technical questions (paid versions only)

What's New in 2018 version

- Added new functions to support LZ4 data compression/decompression
- You can use GraphicsMagick to access IPP optimized functions
- Platform aware APIs provide 64 bit parameters for image dimensions and vector length.

Learn More: software.intel.com

Roadmap Notice: All information provided here is subject to change without notice.
Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

**Certain technical specifications and select processors/skus apply. See product site for more details.

What's Inside Intel® Integrated Performance Primitives

High Performance , Easy-to-Use & Production Ready APIs

Image Domain
- Image Processing
- Computer Vision
- Color Conversion

Signal Domain
- Signal Processing
- Vector Math

Data Domain
- Data Compression
- Cryptography
- String Processing

Intel® Architecture Platforms

Operating System: Windows*, Linux*, Android*, MacOS[1]*

[1] Available only in Intel® Parallel Studio Composer Edition.

Bitwise Reproducibility with the Same Executable:

- Reproducibility from one run to another: -qno-opt-dynamic-align
  - Makes results independent of alignment
  - Alternative: align all data, e.g. -alignarray64byte (Fortran); _mm_malloc() etc. (C/C++)

- Reproducibility from one run-time processor to another:
  - -fimf-arch-consistency=true -qno-opt-dynamic-align
  - with 18.0, -fimf-use-svml -qno-opt-dynamic-align might suffice for Sandy Bridge and later

- Reproducibility for different domain decompositions (# threads and/or # MPI ranks)
  - -fp-model consistent (safest) with no parallel reductions (except TBB parallel_deterministic_reduce)

Bitwise Reproducibility with Different Executables:

- Reproducibility between different compile-time processor targets; different optimization levels; etc.
  - -fp-model consistent (equivalent to -fp-model precise -nofma -fimf-arch-consistency=true)
  - -fp-model consistent -fimf-use-svml re-enables vectorization of math functions in 18.0
- Reproducibility between Windows* and Linux* (being worked on) or different compiler versions ?
  - Not covered; best try is -fp-model consistent -fimf-use-svml -fimf-precision=high

Summary

- Auto Vectorize as much as possible
  - This will ensure that future architectures will vectorize
- Optimization Report is the way the compiler communicates to you
  - Tells you what didn’t vectorize and why.
- OpenMP SIMD
  - For when auto vectorization isn’t possible, or when you need vectorize outer loops.
    - Correctness is on the user

Further Information

Webinars:

https://software.intel.com/videos/getting-the-most-out-of-the-intel-compiler-with-new-optimization-reports
https://software.intel.com/videos/new-vectorization-features-of-the-intel-compiler
https://software.intel.com/videos/from-serial-to-awesome-part-2-advanced-code-vectorization-and-optimization
https://software.intel.com/videos/data-alignment-padding-and-peel-remainder-loops

Vectorization Guide (C): https://software.intel.com/articles/a-guide-to-auto-vectorization-with-intel-c-compilers/

Explicit Vector Programming in Fortran:
https://software.intel.com/articles/explicit-vector-programming-in-fortran

Initially written for Intel® Xeon Phi™ coprocessors, but also applicable elsewhere:

https://software.intel.com/articles/vectorization-essential

https://software.intel.com/articles/fortran-array-data-and-arguments-and-vectorization

The Intel® C++ and Fortran Compiler Developer Guides, https://software.intel.com/en-us/cpp-compiler-18.0-developer-guide-and-reference https://software.intel.com/en-us/fortran-compiler-18.0-developer-guide-and-reference

Legal Disclaimer & Optimization Notice

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®

BackUp

Compiler must recognize to handle apparent dependencies

OpenMP* SIMD-Enabled functions

A way to vectorize loops containing calls to functions that can't be inlined

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

SIMD-Enabled Function

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

```
    real function func(x, y, z, xp, yp, zp)
!$omp declare simd (func) uniform( y, z, xp, yp, zp )
      real, intent(in) :: x, y, z, xp, yp, zp
      denom = (x-xp)**2 + (y-yp)**2 + (z-zp)**2
      func = 1./sqrt(denom)
    end
…
!$omp simd  private(x)  reduction(+:sumx)
      do i = 1, nx-1
        x = x0 + i * h
        sumx = sumx + func(x, y, z, xp, yp, zp)
      enddo
```

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.

SIMD-enabled function must have explicit interface

!$omp simd may not be needed in simpler cases

Clauses for SIMD-Enabled Functions

#pragma omp declare simd (C/C++)

!$OMP DECLARE SIMD (fn_name) (Fortran)

- UNIFORM — argument is never vector
- LINEAR (REF|VAL|UVAL) — additional induction variables use REF(X) when vector argument is passed by reference (Fortran default)
- INBRANCH / NOTINBRANCH — specify whether function will be called conditionally
- SIMDLEN — vector length
- ALIGNED — asserts that listed variables are aligned
- PROCESSOR(cpu) — Intel extension, tells compiler which processor to target, e.g. core_2nd_gen_avx, haswell, knl, skylake_avx512
  NOT controlled by -x… switch, may default to SSE

  Simpler is to target processor specified by -x switch using **-vecabi=cmdtarget**

SIMD-Enabled Fortran Subroutine

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

```
subroutine test_linear(x, y)
!$omp declare simd (test_linear) linear(ref(x, y))
   real(8),intent(in)  :: x
   real(8),intent(out) :: y
   y = 1. + sin(x)**3
end subroutine test_linear
…
Interface
…
do j = 1,n
    call test_linear(a(j), b(j))
enddo
```

Important because arguments passed by reference in Fortran

remark #15301: FUNCTION WAS VECTORIZED.

remark #15300: LOOP WAS VECTORIZED.

SIMD-enabled routine must have explicit interface

!$omp simd   not needed in simple cases like this

SIMD-Enabled Fortran Subroutine

The LINEAR(REF) clause is very important

- In C, compiler places consecutive argument values in a vector register
- But Fortran passes arguments by reference
  - By default compiler places consecutive addresses in a vector register
  - Leads to a gather of the 4 addresses (slow)
  - LINEAR(REF(X)) tells the compiler that the addresses are consecutive; only need to dereference once and copy consecutive values to vector register
- Same method could be used for C arguments passed by reference

Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. The results above were obtained on a 4th Generation Intel® Core™ i7-4790 system, frequency 3.6 GHz, running Ubuntu* version 14.04.5 and using the Intel® Fortran Compiler version 18.0

Approx speed-up for double precision array of 1M elements built with -xcore-avx2:

SIMD options	Speed-up	Memory access	Vector length
No DECLARE SIMD	1.0	scalar	1
DECLARE SIMD but no LINEAR(REF)	1.7	Non-unit stride	2
DECLARE SIMD with LINEAR(REF)	4.3	Unit stride	4
plus -vecabi=cmdtarget	4.6	Unit stride	8

Special Idioms

Compiler must recognize to handle apparent dependencies

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

Compress – simple example

```
int compress(float *a, float *restrict b, int na) {
    int nb = 0;
    for (int ia=0; ia <na; ia++)  {
      if (a[ia] > 0.)  b[nb++]  = a[ia];   // nb causes cross-iteration dependency
    }
    return nb;
}
```

With Intel® AVX2, does not auto-vectorize

- icc -c -std=c99 -xcore-avx2 -qopt-report-file=stderr -qopt-report-phase=vec compress.c

  …

  LOOP BEGIN at compress.c(17,2)

    remark #15344: loop was not vectorized: vector dependence prevents vectorization.

    remark #15346: vector dependence: assumed ANTI dependence between nb (4:19) and nb (4:21)

  LOOP END

Compress with Intel® AVX-512

```
icc -c -std=c99 -xcommon-avx512 -qopt-report-file=stderr -qopt-report=3 compress.c
…
LOOP BEGIN at compress.c(3,2)
  remark #15300: LOOP WAS VECTORIZED
  remark #15450: unmasked unaligned unit stride loads: 1
  remark #15457: masked unaligned unit stride stores: 1
…
  remark #15478: estimated potential speedup: 14.010
  remark #15497: vector compress: 1
LOOP END
```

- Compile with -S to see new instructions in assembly code:

```
grep vcompress compress.s
vcompressps %zmm5, %zmm2{%k1}        #4.19
vcompressps %zmm2, %zmm1{%k1}        #4.19
vcompressps %zmm2, %zmm1{%k1}        #4.19
vcompressps %zmm5, %zmm2{%k1}        #4.19
```

Compress – simple example

```
subroutine compress(a, b, na, nb )
  implicit none
  real, dimension(na), intent(in ) :: a
  real, dimension(* ), intent(out) :: b
  integer,          intent(in)  :: na
  integer,          intent(out) :: nb
  integer                       :: ia
```

```
nb = 0
do ia=1, na
   if(a(ia) > 0.) then
     nb = nb + 1        ! dependency
     b(nb) = a(ia)      ! compress
   endif
 enddo
end
```

With Intel® AVX2, does not auto-vectorize

- ifort -c -xcore-avx2 -qopt-report-file=stderr -qopt-report-phase=vec -qopt-report=3 compress.f90

```
…
LOOP BEGIN at compress.f90(10,3)
   remark #15344: loop was not vectorized: vector dependence prevents vectorization.
   remark #15346: vector dependence: assumed ANTI dependence between nb (12:7) and nb (12:7)
LOOP END
```

Optimization Notice

Compress with Intel® AVX-512

```
ifort -c -xcommon-avx512 -qopt-report-file=stderr -qopt-report=3 compress.f90
…
LOOP BEGIN at compress.c(3,2)
  remark #15300: LOOP WAS VECTORIZED
  remark #15450: unmasked unaligned unit stride loads: 1
  remark #15457: masked unaligned unit stride stores: 1
…
  remark #15478: estimated potential speedup: 13.080
  remark #15497: vector compress: 1
LOOP END
```

- Compile with -S to see new instructions in assembly code:

```
grep vcompress compress.s
vcompressps %zmm5, %zmm2{%k1}          #13.7
vcompressps %zmm2, %zmm1{%k1}          #13.7
vcompressps %zmm2, %zmm1{%k1}          #13.7
vcompressps %zmm5, %zmm2{%k1}          #13.7
```

Compress – more complex example

```
int compress(int n1, int n2, float a[][n2], float b[restrict]) {
            int nb   = 0;
            for (int i1=0; i1 <n1; i1++)  {
              float sc = 0.f;
              for (int i2=0; i2<n2; i2++)  sc += a[i1][i2];
                 if (sc > 0.f) b[nb++] = sc;
            }
            return nb;
}
```

By default, the inner reduction loop over i2 is vectorized

```
icc -std=c99 -xcommon-avx512 -qopt-report-file=stderr -qopt-report-phase=vec compress3.c
    LOOP BEGIN at compress3.c(5,3)
      remark #15300: LOOP WAS VECTORIZED
    LOOP END
```

More efficient to vectorize the outer loop over i2, especially if n1 >> n2

Compress – Explicit Vectorization with OpenMP*

```
int compress(int n1, int n2, float a[][n2], float b[restrict]) {
        int nb   = 0;
#pragma omp simd
        for (int i1=0; i1 <n1; i1++)  {
          float sc = 0.f;
          for (int i2=0; i2<n2; i2++)  sc += a[i1][i2];
#pragma omp ordered simd monotonic(nb:1)
            { if (sc > 0.f) b[nb++] = sc; }
        }
        return nb;
 }
```

```
icc -std=c99 -xcommon-avx512 -qopenmp-simd …

LOOP BEGIN at compress3.c(4,2)
  remark #15301:
                OpenMP SIMD LOOP WAS VECTORIZED
  remark #15452: unmasked strided loads: 1
  remark #15457: masked unaligned unit stride stores: 1
  …
  remark #15497: vector compress: 1
  …
  LOOP BEGIN at compress3.c(6,3)
    remark #15548: loop was vectorized along with the
              outer loop
  LOOP END
LOOP END
```

omp simd — tells compiler to vectorize outer loop

omp ordered — takes care of the nb dependency

if omitted, results may be incorrect

monotonic(nb:1) — enables (much) more efficient code generation

Compress – more complex example

```
subroutine compress(a, b, na1, na2, nb )
 implicit none
 real(8), intent(in ), dimension(na1,na2)) :: a
 real(8), intent(out), dimension(*)        :: b
 integer, intent(in ) :: na1, na2
 integer, intent(out) :: nb
 integer              :: ia1, ia2, ib
 real(8)              :: sum

 nb = 0
```

```
  do ia2=1, na2
    sum = 0.
    do ia1=1, na1
      sum = sum + a(ia1,ia2)
    enddo
    if(sum.gt.0.) then
      nb = nb + 1
      b(nb) = sum
    endif
  enddo
end
```

By default, the inner reduction loop over ia1 is vectorized

ifort -xcommon-avx512 -qopt-report-file=stderr -qopt-report-phase=vec compress6.f90

LOOP BEGIN at compress6.f90(27,5)

remark #15300: LOOP WAS VECTORIZED

LOOP END

More efficient to vectorize the outer loop over ia2, especially if na2 >> na1

Compress – Explicit Vectorization with OpenMP*

ifort -xcommon-avx512 -qopt-report-file=stderr
-qopt-report-phase=vec compress6.f90

LOOP BEGIN at compress6.f90(25,3)
…
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
remark #15452: unmasked strided loads: 1
remark #15457: masked unaligned unit stride stores: 1
…
remark #15497: vector compress: 1
…
LOOP END

- Use local variable ib for compress counter, not dummy argument nb

omp simd — tells compiler to vectorize outer loop
omp ordered — takes care of the nb dependency. If omitted, results may be incorrect.
monotonic(ib) — enables (much) more efficient code generation.

```
subroutine compress(a, b, na1, na2, nb )
...
ib = 0   ! Don't use dummy argument nb here!
!$omp simd private(sum)
do ia2=1, na2
    sum = 0.
    do ia1=1, na1
      sum = sum + a(ia1,ia2)
    enddo
!$omp ordered simd monotonic(ib:1)
    if(sum.gt.0.) then
      ib = ib + 1
      b(nb) = sum
    endif
!$omp end ordered
  enddo
  nb = ib
end
```

Compress – Explicit Vectorization with OpenMP*

```
int compress(int n1, int n2, float a[][n2], float b[restrict]) {
                int nb   = 0;
#pragma omp simd
                for (int i1=0; i1 <n1; i1++)  {
                  float sc = 0.f;
                  for (int i2=0; i2<n2; i2++)  sc += a[i1][i2];
#pragma omp ordered simd monotonic(nb:1)
                    { if (sc > 0.f) b[nb++] = sc; }
                }
                return nb;
 }
```

```
icc -std=c99 -xcommon-avx512 -qopenmp-simd …

LOOP BEGIN at compress3.c(4,2)
  remark #15301:
                 OpenMP SIMD LOOP WAS VECTORIZED
  remark #15452: unmasked strided loads: 1
  remark #15457: masked unaligned unit stride stores: 1
  …
  remark #15497: vector compress: 1
  …
  LOOP BEGIN at compress3.c(6,3)
    remark #15548: loop was vectorized along with the
                outer loop
  LOOP END
LOOP END
```

omp simd tells compiler to vectorize outer loop

omp ordered takes care of the nb dependency

if omitted, results may be incorrect

monotonic(nb:1) enables (much) more efficient code generation

Compress loops - performance

Optimization Options		Speed-up (C)	Speed-up (Fortran)
Simple loop	-O2 -xcore-avx2	1.0	1.0
	-O2 -xcommon-avx512	15.4	14.6
Nested loop	-O2 xcommon-avx512	1.0	1.0
ordered	-O2 xcommon-avx512 -qopenmp-simd	2.4	1.8
monotonic	-O2 xcommon-avx512 -qopenmp-simd	8.2	5.2

Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary.
The results above were obtained on an Intel® Xeon® Platinum 8180M system, frequency 2.5 GHz, running Fedora 25 and using the Intel® Fortran Compiler version 18.0 update 1.

Search Loops

Normally, a vectorizable loop must have a single exit

- And iteration count must be known at start of execution
  - Else a later iteration may have started before an earlier iteration decides the loop should be terminated

Simple "search" loops are an exception

- Compiler recognizes
  - executes special code if an exit occurs during a SIMD iteration
  - only works if no stores back to memory

Search Loop – very simple

```c
int search(int * array, int target, int na) {
  int i;

  for(i=0; i<na; i++) {
    if(array[i] == target) break;
  }

  return i;
}
```

```fortran
integer  function search  (na, target, array)
   integer, intent(in) :: na, target, array(na)

   do i=1,na
      if(array(i) == target) exit
   enddo

   search = i
end
```

icc -c -qopt-report-file=stderr search1.c

```
…
LOOP BEGIN at search1.c(4,3)
   remark #15300: LOOP WAS VECTORIZED
LOOP END
```

ifort -c -qopt-report-file=stderr search1.f90

Search Loop – more complex

If loop contains vector store, compiler can't handle

- Do calculation until first negative value of **a** is encountered

```
int search(float* a,float *b, float*c, int n)
{
  int i;

  for(i=0; i<n; i++)  {
    if(a[i] <  0.) break;
    c[i] = sqrtf(a[i]) * b[i];
  }

  return i-1;
}
```

```
integer function search(a,b,c,n)
  real, dimension(n) :: a, b, c
  integer            :: n, i

  do i=1,n
     if(a(i) .lt. 0.) exit
     c(i) = sqrt(a(i)) * b(i)
  enddo

  search = i-1
end
```

icc -c -qopt-report-file=stderr search3.c

…

ifort -c -qopt-report-file=stderr search3.f90

…

remark #15520: loop was not vectorized: loop with multiple exits cannot be vectorized unless it meets search loop idiom criteria [search3.c(5,18)] [search3.f90(8,3)]

OpenMP* SIMD to the Rescue

```c
int search(float* a,float *b, float*c, int n)
{
  int i;
#pragma omp simd early_exit
  for(i=0; i<n; i++)  {
    if(a[i] <  0.) break;
    c[i] = sqrtf(a[i]) * b[i];
  }

  return i-1;
}
```

```fortran
integer function search(a,b,c,n)
  real, dimension(n) :: a, b, c
  integer           :: n, i
!$omp simd early_exit
  do i=1,n
     if(a(i) .lt. 0.) exit
     c(i) = sqrt(a(i)) * b(i)
  enddo

  search = i-1
end
```

icc -c -qopenmp-simd … search5.c

…

remark #15301: OpenMP SIMD LOOP WAS VECTORIZED

ifort -c -qopenmp-simd … search5.f90

…

#pragma omp simd without “early_exit” clause is not sufficient:

search5.c(6): error: break cannot be used to exit simd region

Search Loop – old way to fix

Split loop into simple search followed by a computation loop

```c
int search(float* a,float *b, float*c, int n)
{
  int i, j;

  for(i=0; i<n; i++)  {
    if(a[i] <  0.) break;
  }

  for(j=0; j<i; j++)  {
    c[j] = sqrtf(a[j]) * b[j];
  }
  return i-1;
}
```

```fortran
integer function search(a,b,c,n)
  real, dimension(n) :: a, b, c
  integer            :: n, i, j

  do i=1,n
     if(a(i).lt.0.) exit
  enddo
  search = i-1

  do j=1,search
     c(j) = sqrt(a(j)) * b(j)
  enddo

end function search
```

- Both loops then vectorize
- Generated code is simpler but need to reload **a**
- Good if SIMD not needed for other reasons

Search loops - performance

Optimization Options	Speed-up (C)	Speed-up (Fortran)
1st example -O2 -xcore-avx2 -no-vec	1.0	1.0
-O2 -xcore-avx2	2.6	1.9
2nd example -O2 xcore-avx2	1.0	1.0
-O2 xcore-avx2 -qopenmp-simd	3.5	4.0
split loops -O2 xcore-avx2	5.2	5.0

Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary.
The results above were obtained on a 4th Generation Intel® Core™ i7-4790 system, frequency 3.6 GHz, running Red Hat* Enterprise Linux* version 7.2 and using the Intel® Fortran Compiler version 18.0 update 1.

Histogramming with Intel® AVX2

```
!  Accumulate histogram of sin(x) in  h
   do i=1,n
      y     = sin(x(i)*twopi)
      ih    = ceiling((y-bot)*invbinw)
      ih    = min(nbin,max(1,ih))
      h(ih) = h(ih) + 1
   enddo
```

```
for (i=0; i<n; i++)  {
     y     = sinf(x[i]*twopi);
     ih    = floor((y-bot)*invbinw);
     ih    = ih > 0     ? ih : 0;
     ih    = ih < nbin ? ih : nbin;
     h[ih] = h[ih] + 1;
  }
```

With Intel® AVX2, this does not vectorize

- Store to **h** is a scatter (indirect addressing)
- **ih** can have the same value for different values of **i**
- Vectorization with a SIMD directive would cause incorrect results

ifort -c -xcore-avx2 histo2.f90 -qopt-report-file=stderr -qopt-report-phase=vec

LOOP BEGIN at histo2.f90(11,4)

remark #15344: loop was not vectorized: vector dependence prevents vectorization… remark #15346: vector dependence: assumed FLOW dependence between line 15 and line 15

LOOP END

INTEL® AVX-512 Conflict Detection Instructions

The **VPCONFLICT** instruction detects elements with previous conflicts in a vector of indexes

- Allows to generate a mask with a subset of elements that are guaranteed to be conflict free
- The computation loop can be re-executed with the remaining elements until all the indexes have been operated upon. In pseudo-code:

VPCONFLICT instruction
VPCONFLICT{D,Q} zmm2/mem, zmm1{k1}
VPTESTNM{D,Q} zmm2, zmm3/mem, zmm2, k2{k1}
VPBROADCASTM{W2D,B2Q} k2, zmm1
VPLZCNT{D,Q} zmm2/mem, zmm1 {k1}

```
index = vload &B[i]                                      // Load 16 B[i]
pending_elem = 0xFFFF;                                   // all still remaining
do {
   curr_elem = get_conflict_free_subset(index, pending_elem)
   old_val = vgather {curr_elem} A, index                // Grab A[B[i]]
   new_val = vadd old_val, +1.0                          // Compute new values
   vscatter A {curr_elem}, index, new_val                // Update A[B[i]]
   pending_elem = pending_elem ^ curr_elem               // remove done idx
} while (pending_elem)
```

Histogramming with Intel® AVX-512 CD

E.g. compile for Intel® Xeon Phi™ processor x200 family:

```
ifort -c -xmic-avx512 histo2.f90 -qopt-report-file=stderr -qopt-report=3 -S
…
LOOP BEGIN at histo2.f90(11,4)
   remark #15300: LOOP WAS VECTORIZED
   remark #15458: masked indexed (or gather) loads: 1
   remark #15459: masked indexed (or scatter) stores: 1
   remark #15478: estimated potential speedup: 13.930
   remark #15499: histogram: 2
LOOP END

  vpminsd   %zmm20, %zmm5, %zmm3                        #24.7 c19
  vpconflictd %zmm3, %zmm1                              #25.7 c21
  vpgatherdd (%r13,%zmm3,4), %zmm6{%k1}                 #25.15 c21
  vptestmd  .L_2il0floatpacket.5(%rip), %zmm1, %k0      #25.7 c23
  vpaddd    %zmm21, %zmm6, %zmm2                        #25.7 c27
…
  vpbroadcastmw2d %k1, %zmm4                            #25.7 c3
  vpaddd    %zmm21, %zmm2, %zmm2{%k1}                   #25.7 c5
  vptestmd  %zmm1, %zmm4, %k0{%k1}                      #25.7 c9 stall 1
  vpscatterdd %zmm2, (%r13,%zmm3,4){%k1}                #25.7 c3
```

Some remarks omitted

Optimization Notice

Histogramming with Intel® AVX-512 CD

Compile for Intel® Xeon Phi™ processor x200 family:

```
ifort -c -xmic-avx512 histo2.f90 -qopt-report-file=stderr -qopt-report=3 -S
…
LOOP BEGIN at histo2.f90(11,4)
  remark #15300: LOOP WAS VECTORIZED
  remark #15458: masked indexed (or gather) loads: 1
  remark #15459: masked indexed (or scatter) stores: 1
  remark #15478: estimated potential speedup: 13.930
  remark #15499: histogram: 2
LOOP END
```

Some remarks omitted

```
vpminsd   %zmm20, %zmm5, %zmm3
vpconflictd %zmm3, %zmm1
#                work on simd lanes without conflicts
vpgatherdd (%r13,%zmm3,4), %zmm6{%k1}      # load h
vptestmd  .L_2il0floatpacket.5(%rip), %zmm1, %k0
vpaddd    %zmm21, %zmm6, %zmm2        #increment h
…
vpbroadcastmw2d %k1, %zmm4
vplzcntd  %zmm1, %zmm4
vptestmd  %zmm1, %zmm5, %k0
```

```
..B1.18                      # loop over simd lanes with conflicts
kmovw     %r10d, %k1
vpbroadcastmw2d %k1, %zmm4
vpermd    %zmm2, %zmm0, %zmm2{%k1}
vpaddd    %zmm21, %zmm2, %zmm2{%k1} #increment histo
vptestmd  %zmm1, %zmm4, %k0{%k1}
kmovw     %k0, %r10d
testl     %r10d, %r10d
jne       ..B1.18
…
vpscatterdd %zmm2, (%r13,%zmm3,4){%k1}    #  final store
```

Histogramming – more complex

```
for (int i=0; i<n; i++) {
     float y = myfun(x[i]);
     int ih  = floor( (y-bot)*invbinw );
     ih      = ih >= 0      ? ih : 0;
     ih      = ih <= nbin-1 ? ih : nbin-1;
     ++contents[ih];
}
```

```
!  Accumulate histogram of myfun(x) in  h
   do i=1,n
       y     = myfun(x(i))
       ih    = ceiling((y-bot)*invbinw)
       ih    = min(nbin,max(1,ih))
       h(ih) = h(ih) + 1
   enddo
```

This does not auto-vectorize, even with Intel® AVX-512, due to the function call

- Can be vectorized with OpenMP* by:
  - Making myfun() a SIMD function
  - Using the OMP ORDERED SIMD pragma/directive
  - Add the OVERLAP hint to help compiler vectorize more efficiently

icc -c -std=c99 -xcommon-avx512 -qopt-report-file=stderr -qopt-report-phase=vec test3.c
…
remark #15543: loop was not vectorized: loop with function call not considered an optimization candidate.

Histogramming – SIMD function

```
#include <math.h>
#pragma omp declare simd
float myfun(float x) {
  float twopi=2.f*acosf(-1.f);
  float y = sinf(x*twopi);
  return y*y*y;
}
```

```
real function myfun(x) result(y)
!$omp declare simd linear(ref(x))
   real, intent(in) :: x
   real, parameter  :: twopi=2.*acos(-1.)

   y     = sin(x*twopi)**3
 end function myfun
```

Compiler creates both vector and scalar versions

- Use -vecabi=cmdtarget to target instruction set specified by -x… (/Qx…) switch
  - Else ABI requires arguments to be passed using xmm registers (Intel® SSE)
  - Linear(ref) clause avoids “gather” of vector of addresses
    - Needed because Fortran default is pass by reference, not value

```
icc -c -std=c99 -xcommon-avx512 -qopenmp-simd -vecabi=cmdtarget -qopt-report-file=stderr myfun.c
…
remark #15347: FUNCTION WAS VECTORIZED with zmm, simdlen=16, unmasked, formal parameter types: (vector)
remark #15347: FUNCTION WAS VECTORIZED with zmm, simdlen=16, masked, formal parameter types: (vector)
```

Histogramming – SIMD vectorization

```
type (histogram), allocatable :: hist(:)
…
subroutine hist_fill(x, nh)
  integer, intent(in)              :: nh
  real, contiguous, intent(in) :: x(:)
…
  interface myfun
    real function myfun(x) result(y)
    !$omp declare simd linear(ref(x))
…
  n = size(x)
```

```
!$omp simd private(y, ih)
  do i=1,n
    y  = myfun(x(i))
    ih = ceiling((y-hist(nh)%bot)*hist(nh)%invbinw)
    ih = min(hist(nh)%nbin,max(1,ih))
!$omp ordered simd overlap(ih)
    hist(nh)%contents(ih) = hist(nh)%contents(ih) + 1
!$omp end ordered
  enddo
end subroutine hist_fill
```

Need explicit interface to SIMD function

- omp ordered simd is sufficient for safe vectorization
  - overlap(ih) may help compiler generate more efficient code
  - ifort -c -xcommon-avx512 -qopenmp-simd -vecabi=cmdtarget histo_mod.f90 myfun.f90

```
…
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
remark #15347: FUNCTION WAS VECTORIZED with zmm, simdlen=16, unmasked, formal parameter types: (linear_ref:4)
```

Histogramming – SIMD vectorization

```
#include <math.h>
#pragma omp declare simd
float myfun(float);

void hist_fill(float *x, int *restrict
contents, int n, int nbin) {
   float bot=-1.f, top=1.f
   float invbinw = (float)nbin / (top-bot);
```

```
#pragma omp simd
   for (int i=0; i<n; i++) {
     float y = myfun(x[i]);
     int ih   = floor( (y-bot)*invbinw );
     ih       = ih >= 0       ? ih : 0;
     ih       = ih <= nbin-1 ? ih : nbin-1;
#pragma omp ordered simd overlap (ih)
        ++contents[ih];
} }
```

Function prototype must be declared as SIMD

- So that caller knows SIMD version is available
- omp ordered simd is sufficient for safe vectorization
  - overlap(ih) may help compiler generate more efficient code
  - icc -c -std=c99 -xcommon-avx512 -qopenmp-simd -vecabi=cmdtarget -qopt-report=3 test3.c

…
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
…
remark #15347: FUNCTION WAS VECTORIZED with zmm, simdlen=16, unmasked, formal parameter types: (vector)

Histogramming with Intel® AVX-512 - Performance

Optimization Options		Speed-up (C)	Speed-up (Fortran)
Simple histogram loop	-xcore-avx2	1.0	1.0
	-xcommon-avx512	3.3	3.2
loop with function call	-xcommon-avx512	1.0	1.0
ordered simd	-xcommon-avx512 -qopenmp-simd	2.3	2.6
overlap	-xcommon-avx512 -qopenmp-simd	2.5	2.9
" -vecabi=cmdtarget	-xcommon-avx512 -qopenmp-simd	3.2	3.7

Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary.
The results above were obtained on an Intel® Xeon® Platinum 8180M system, frequency 2.5 GHz, running Fedora 25 and using the Intel® Fortran Compiler version 18.0 update 1.

Histogramming with Intel® AVX-512: speed-up

Speed-up depends on problem details

- Comes mostly from vectorization of other heavy computation in the loop
  - Not from the scatter itself
- Speed-up may be (much) less if there are many conflicts
  - E.g. histograms with a singularity or narrow spike
- Similar behavior for C and Fortran versions
- Speed-up due to vectorization would be considerably higher on Intel® Xeon Phi™ x200 processors because scalar processor is slower.

Other problems map to this

- E.g. energy deposition in cells in particle transport Monte Carlo simulation

FUTURE

OpenMP 5.0 is coming

Forthcoming features

User-defined reductions

Inclusive and Exclusive Scans

Either may be used to implement MINLOC and MAXLOC reductions

- Determine minimum (maximum) value of an array and also its location

Vectorization Summary

The importance of SIMD parallelism is increasing

- Moore's law leads to wider vectors as well as more cores
- Don't leave performance "on the table"
- Be ready to help the compiler to vectorize, if necessary
  - With compiler directives and hints
  - Using information from vectorization and optimization reports
  - With explicit vector programming
  - Use Intel® Advisor and/or Intel® VTune™ Amplifier XE to find the best places (hotspots) to focus your efforts

No need to re-optimize vectorizable code for new processors

  - Typically a simple recompilation

Further Information

Webinars:

https://software.intel.com/videos/getting-the-most-out-of-the-intel-compiler-with-new-optimization-reports
https://software.intel.com/videos/new-vectorization-features-of-the-intel-compiler
https://software.intel.com/videos/from-serial-to-awesome-part-2-advanced-code-vectorization-and-optimization
https://software.intel.com/videos/data-alignment-padding-and-peel-remainder-loops

Vectorization Guide (C): https://software.intel.com/articles/a-guide-to-auto-vectorization-with-intel-c-compilers/

Explicit Vector Programming in Fortran:
https://software.intel.com/articles/explicit-vector-programming-in-fortran

Initially written for Intel® Xeon Phi™ coprocessors, but also applicable elsewhere:

https://software.intel.com/articles/vectorization-essential

https://software.intel.com/articles/fortran-array-data-and-arguments-and-vectorization

The Intel® C++ and Fortran Compiler Developer Guides, https://software.intel.com/en-us/cpp-compiler-18.0-developer-guide-and-reference https://software.intel.com/en-us/fortran-compiler-18.0-developer-guide-and-reference

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®

Prefetching for KNL

Hardware prefetcher is more effective than for KNC

Software (compiler-generated) prefetching is off by default

- Like for Intel® Xeon® processors
- Enable by -qopt-prefetch=[1-5]

KNL has gather/scatter prefetch

- Enable auto-generation to L2 with -qopt-prefetch=5
  - Along with all other types of prefetch, in addition to h/w prefetcher – careful.
- Or hint for specific prefetches
  - !DIR$ PREFETCH var_name [: type : distance]
  - Needs at least -qopt-prefetch=2
- Or call intrinsic
  - _mm_prefetch((char *) &a[i], *hint*); C
  - MM_PREFETCH(A, *hint*) Fortran

Optimization Notice