



Intel® Compilers Version 15.0

Part of Intel® Parallel Studio XE Composer Edition
2015

February 2015



Why Use Intel® Compilers?

Compatibility

- OS Support: Windows*, Linux*, OS X* (for OS/X , both traditional Intel C++ front end and CLANG version)
- IDE support: Visual Studio* in Windows*, Eclipse* in Linux*, Xcode* in OS X*
- Compatible with GNU* Compiler collection (gcc) – adapts to specific version up to 4.9
- Source and binary compatibility with Microsoft Visual C++* Compilers
- ISO Standard for C : almost full C99 compatibility, a few new features of C11
- ISO C++ Standard: **all of C++11**
- **Full Fortran 2003**, many features from Fortran 2008 including DO CONCURRENT, COARRAYS, BLOCK

Parallelism

- Language Extension (Intel Cilk® Plus™ for C/C++) for task parallelism
- Explicit Vector Programming (OpenMP* / Cilk SIMD, Array Notation)
- Support for OpenMP* 4.0 (except user-defined reductions)

Why Use Intel® Compilers?

Performance

- Code generation tuned for latest microarchitecture
- Full – and very early - support of Intel processor instruction sets (SSE, AVX, AVX2, AVX-512)

Optimization

- Sophisticated optimizations like interprocedural and profile-guided optimization
- Automatic vectorization
- Automatic parallelization
- Optimization reports
- Highly optimized version of libm (Intel® Math Library libimf) and vector math library libsvml

PGO Usage: Three Step Process

Step 1

Compile + link to add instrumentation

```
icc -prof_gen foo.c -o foo
```

Instrumented executable:

foo.exe

Step 2

Execute instrumented program

foo.exe (on a typical dataset)

Dynamic profile:

12345678.dyn

profmerge

Step 3

Compile + link using feedback

```
icc -prof_use prog.c -o foo
```

Merged .dyn files:

pgopti.dpi

Optimized executable:

foo.exe

Simple PGO Example: Code Re-Order

```
for (i=0; i < NUM_BLOCKS; i++)
{
    switch (check3(i))
    {
        case 3:                /* 25% */
            x[i] = 3; break;
        case 10:               /* 75% */
            x[i] = i+10; break;
        default:               /* 0% */
            x[i] = 99; break
    }
}
```

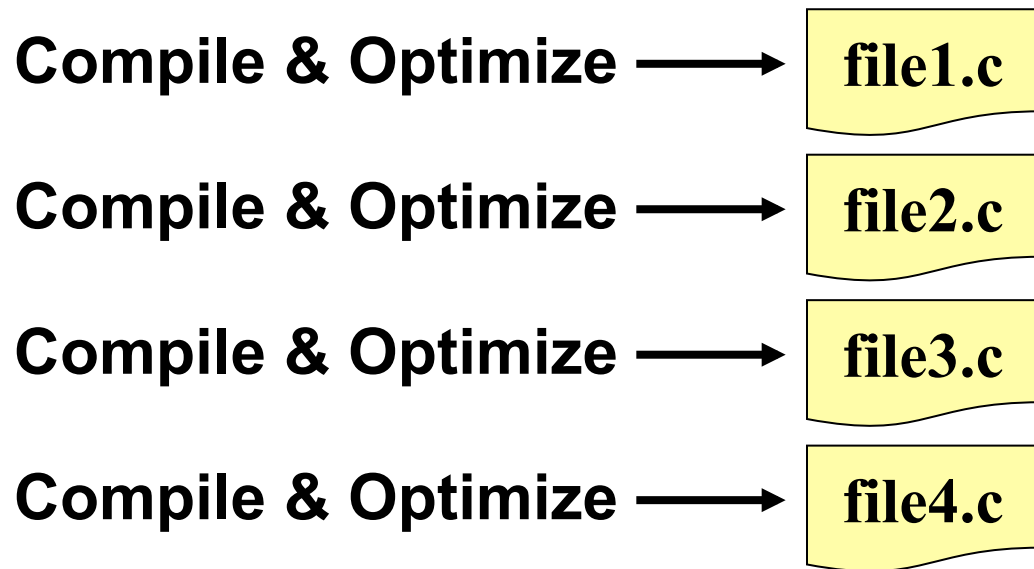
“Case 10” is moved to the beginning

- PGO can eliminate most tests&jumps for the common case - less branch mispredicts

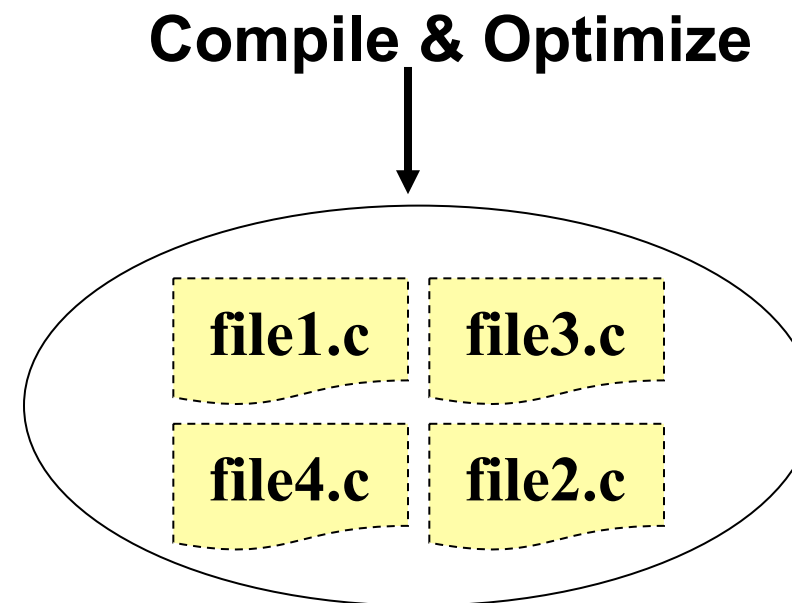
Interprocedural Optimizations

Extends optimizations across file boundaries

Without IPO



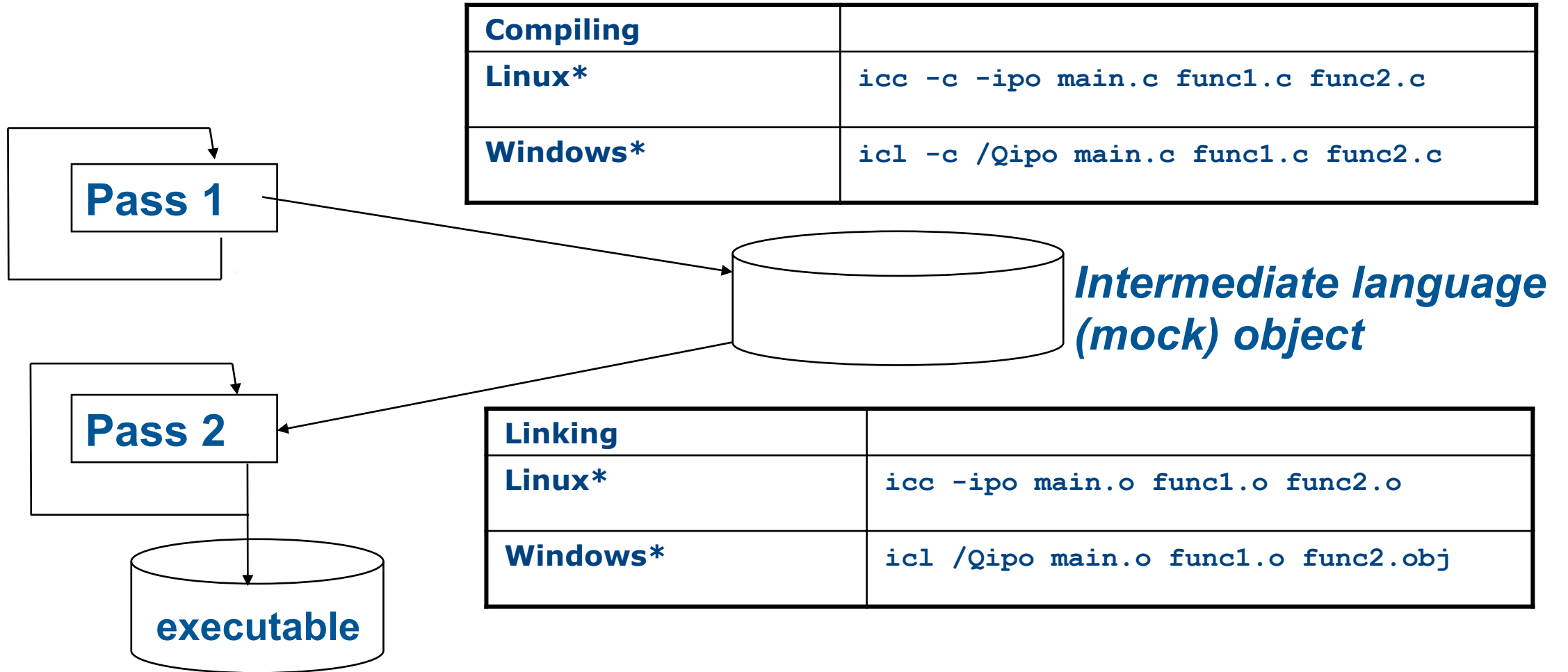
With IPO



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

Interprocedural Optimizations (IPO)

Usage: Two-Step Process



Optimization Reports

Optimization Report Redesign

- Old functionality implemented under `-opt-report`, `-vec-report`, `-openmp-report`, `-par-report` replaced by unified `-opt-report` compiler options
 - `[vec, openmp, par]-report` options deprecated and map to equivalent `opt-report-phase`
- Can still select phase with `-opt-report-phase` option. For example, to only get vectorization reports, use `-opt-report-phase=vec`
- Output now defaults to a `<name>.optrpt` file where `<name>` corresponds to the output object name. This can be changed with `-opt-report-file=[<name>|stdout|stderr]`
- Windows*: `/Qopt-report`, `/Qopt-report-phase=<phase>` etc
 - Optimization report integrated into Microsoft Visual Studio*

Let's take a Look at an Example

```
1  double a[1000][1000],b[1000][1000],c[1000][1000];
2
3  void foo() {
4      int i,j,k;
5
6      for( i=0; i<1000; i++) {
7          for( j=0; j< 1000; j++) {
8              c[j][i] = 0.0;
9              for( k=0; k<1000; k++) {
10                 c[j][i] = c[j][i] + a[k][i] * b[j][k];
11             }
12         }
13     }
14 }
```

15.0 Loop Optimization Report

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

LOOP BEGIN at d:\iusers\hsaito\ics\15_0\dev\test.c(7,5)

Distributed chunk1

remark #25430: LOOP DISTRIBUTION (2 way)

remark #25448: Loopnest Interchanged : (1 2) --> (2 1)

remark #25424: Collapsed with loop at line 6

remark #25412: memset generated

remark #15144: loop was not vectorized: loop was transformed to memset or memcpy

LOOP END

LOOP BEGIN at d:\iusers\hsaito\ics\15_0\dev\test.c(7,5)

Distributed chunk2

remark #25448: Loopnest Interchanged : (1 2 3) --> (2 3 1)

remark #15018: loop was not vectorized: not inner loop

LOOP BEGIN at d:\iusers\hsaito\ics\15_0\dev\test.c(9,7)

Distributed chunk2

remark #15018: loop was not vectorized: not inner loop

LOOP BEGIN at d:\iusers\hsaito\ics\15_0\dev\test.c(6,3)

remark #15145: vectorization support: unroll factor set to 4

remark #15003: PERMUTED LOOP WAS VECTORIZED

LOOP END

LOOP BEGIN at d:\iusers\hsaito\ics\15_0\dev\test.c(6,3)

remark #15003: REMAINDER LOOP WAS VECTORIZED

LOOP END

LOOP END

LOOP END

Clearer view of what happens
where and in what order

Optimization Report Phases

The compiler reports optimizations from 9 phases:

LOOP: Loop Nest Optimizations

PAR: Auto-Parallelization

VEC: Vectorization

OPENMP: OpenMP

OFFLOAD: Offload

IPO: Interprocedural Optimizations

PGO: Profile Guided Optimizations

CG: Code Generation Optimizations

TCOLLECT: Trace Analyzer Collection

LOOP/PAR/VEC share a unified loop structure, a hierarchical output, to seamlessly display optimizations in an integrated format; list of phases significantly simplified from 14.0

Selecting phases for compiler optimization reporting is highly customizable to satisfy customers' specific requirements.

- Single Phase Reporting:
 - Compiler Option: `-[Q]opt-report-phase=VEC`
- Multiple Phase Reporting (use a comma separated list):
 - Compiler Option: `-[Q]opt-report-phase=VEC, OPENMP, IPO, LOOP`
- Default is “ALL” phases and default reporting verbosity level is 2
 - Want to encourage use of integrated HPO report instead of just `vec-report[n]`
 - Lot of changes from 14.0 to remove extraneous information

Optimization Report Levels

The compiler's optimization report have 5 verbosity levels.

- Specifying report verbosity level:
 - Compiler Option: `-opt-report=N` where N = level of desired verbosity
When option omitted, default N=2.
 - For each optimization phase, higher verbosity level indicates higher level of detail reported.
 - Each verbosity level is inclusive of lower levels.
- Example, VEC Phase Levels:
 - Level 1: Reports when vectorization has occurred.
 - Level 2: Adds diagnostics why vectorization did not occur.
 - Level 3: Adds vectorization loop summary diagnostics.
 - Level 4: Adds additional available vectorization support information.
 - Level 5: Adds detailed data dependency information diagnostics.
- **Each** phase can support up to 5 levels

Example Code for IPO Opt Report

```
1  #include <stdio.h>
2
3  static void __attribute__((noinline)) bar(float
4      a[100][100], float b[100][100]) {
5      int i, j;
6      for (i = 0; i < 100; i++) {
7          for (j = 0; j < 100; j++) {
8              a[i][j] = a[i][j] + 2 * i;
9              b[i][j] = b[i][j] + 4 * j;
10         }
11     }
12
13     static void foo(float a[100][100], float b[100][100])
14     {
15         int i, j;
16         for (i = 0; i < 100; i++) {
17             for (j = 0; j < 100; j++) {
18                 a[i][j] = 2 * i;
19                 b[i][j] = 4 * j;
20             }
21         }
22     }
23     bar(a, b);
24 }
```

```
24     extern int main() {
25         int i, j;
26         float a[100][100];
27         float b[100][100];
28
29         for (i = 0; i < 100; i++) {
30             for (j = 0; j < 100; j++) {
31                 a[i][j] = i + j;
32                 b[i][j] = i - j;
33             }
34         }
35         foo(a, b);
36         foo(a, b);
37         fprintf(stderr, "%d %d\n", a[99][9], b[99][99]);
38     }
```

Compiled with:

```
icc -opt-report=L -opt-report-phase=ipo sm.c
```

with L = 1, 2, 3, 4, 5

Level 5 for Vectorization Report

```
VECRPT (col. 3) LOOP WAS VECTORIZED.  
VECRPT (col. 3) entire loop may be executed in scalar remainder  
VECRPT (col. 3) estimated potential speedup: 3.540000.  
VECRPT (col. 3) lightweight vector operations: 26.  
VECRPT (col. 3) loop inside vectorized loop at nesting level: 1.  
VECRPT (col. 3) loop was vectorized (with peel/with remainder)  
VECRPT (col. 3) medium-overhead vector operations: 10.  
VECRPT (col. 3) scalar loop cost: 14.  
VECRPT (col. 3) unmasked aligned unit stride loads: 4.  
VECRPT (col. 3) unmasked aligned unit stride stores: 4.  
VECRPT (col. 3) unmasked unaligned unit stride loads: 8.  
VECRPT (col. 3) unmasked unaligned unit stride stores: 2.  
VECRPT (col. 3) unroll factor set to 2.  
VECRPT (col. 3) vector loop cost: 7.500000.
```

```
6:   do i=1,n  
7:       a(i)= a(i)-b(i)*d(i)  
8:       c(i)= a(i)+c(i)  
9:   enddo
```

Annotated Assembly Listings

```
.L11:          # optimization report
              # LOOP WAS INTERCHANGED
              # loop was not vectorized: not inner loop

xorl         %edi, %edi                #38.3
movsd       b.279.0.2(%rax,%rsi,8), %xmm0 #41.32
unpcklpd    %xmm0, %xmm0              #41.32
              # LOE rax rcx rbx rsi rdi r12 r13 r14 r15 edx xmm0

..B1.11:      # Preds ..B1.11 ..B1.10

..L12:          # optimization report
              # LOOP WAS INTERCHANGED
              # LOOP WAS VECTORIZED
              # VECTORIZATION HAS UNALIGNED MEMORY REFERENCES
              # VECTORIZATION SPEEDUP COEFFECIENT 2.250000

movaps      a.279.0.2(%rcx,%rdi,8), %xmm1 #41.22
movaps      16+a.279.0.2(%rcx,%rdi,8), %xmm2 #41.22
movaps      32+a.279.0.2(%rcx,%rdi,8), %xmm3 #41.22
movaps      48+a.279.0.2(%rcx,%rdi,8), %xmm4 #41.22
mulpd       %xmm0, %xmm1              #41.32
mulpd       %xmm0, %xmm2              #41.32
```

<...>

```
L4::          ; optimization report
              ; PEELED LOOP FOR VECTORIZATION

$LN36:
$LN37:
vaddss      xmm1, xmm0, DWORD PTR [r8+r10*4] ;4.5

snip snip snip

L5::          ; optimization report
              ; LOOP WAS VECTORIZED
              ; VECTORIZATION HAS UNALIGNED MEMORY REFERENCES
              ; VECTORIZATION SPEEDUP COEFFECIENT 8.398438

$LN46:
vaddps      ymm1, ymm0, YMMWORD PTR [r8+r9*4] ;4.5

snip snip snip

L6::          ; optimization report
              ; LOOP WAS VECTORIZED
              ; REMAINDER LOOP FOR VECTORIATION
              ; VECTORIZATION HAS UNALIGNED MEMORY REFERENCES
              ; VECTORIZATION SPEEDUP COEFFECIENT 2.449219

$LN78:
add         r10, 4                    ;3.3

snip snip snip

L7::          ; optimization report
              ; REMAINDER LOOP FOR VECTORIATION

$LN93:
inc         rax                       ;3.3
```


Control of Floating Point Operations

FP Programming Objectives Differ


Accuracy	Produce results that are 'close' to the correct value	Measured in relative error, possibly ulps
Consistence	Produce consistent / reproducible results	From one run to the next From one set of build options to another From one compiler to another From one platform to another
Conformanc e	Produce results a standard or agreed on convention defines	IEEE-754/758, C/C++, FORTRAN
Performance	Produce most efficient code	Default, primary goal of Intel compilers

These objectives usually conflict! Wise design of program code and use of compiler options lets you control the tradeoffs

Reassociation Example: A Reduction

```
float sum(const float A[], int n )
{
    float sum=0;
    for (int i=0; i<n; i++)
        sum = sum + A[i];
    return sum;
}
```

- Scalar reduction gives 7-8X perf gain for SSE – AVX even more !
- Invalid in SAFE modes
- Even in SAFE mode, OpenMP, MPI, TBB might do 'unsafe' reductions



```
float sum( const float A[], int n )
{
    int n4 = n-n%4; // or n4=n4&(~3)
    int i;
    float sum1=0, sum2=0, sum3=0, sum4=0;
    for (i=0; i<n4; i+=4) {
        sum = sum + A[i];
        sum1 = sum1 + A[i+1];
        sum2 = sum2 + A[i+2];
        sum3 = sum3 + A[i+3];
    }
    sum += sum1 + sum2 + sum3;
    for (; i<n; i++)
        sum = sum + A[i];
    return sum;
}
```

Controlling FP Computation by fp-model Switch

The `-fp-model` switch lets you specify the compiler rules for

- Value safety
- FP expression evaluation
- FPU environment access
- Precise FP exceptions
- FP contractions

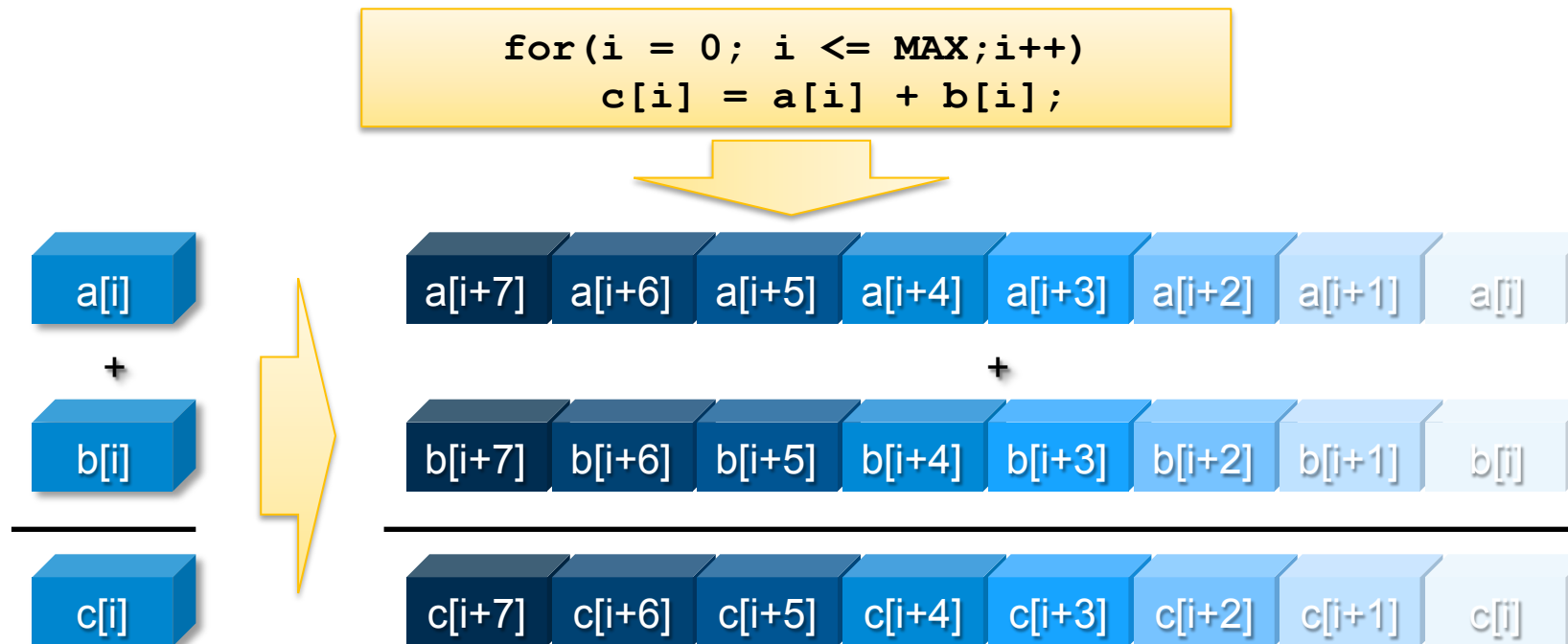
In the past (still available), a mix of many, many switches (`-mp`, `-mp1`, `-pc64`, `-pc80` etc) had to be used

- Most marked 'deprecated' now; not very structured, not well documented, partially inconsistent

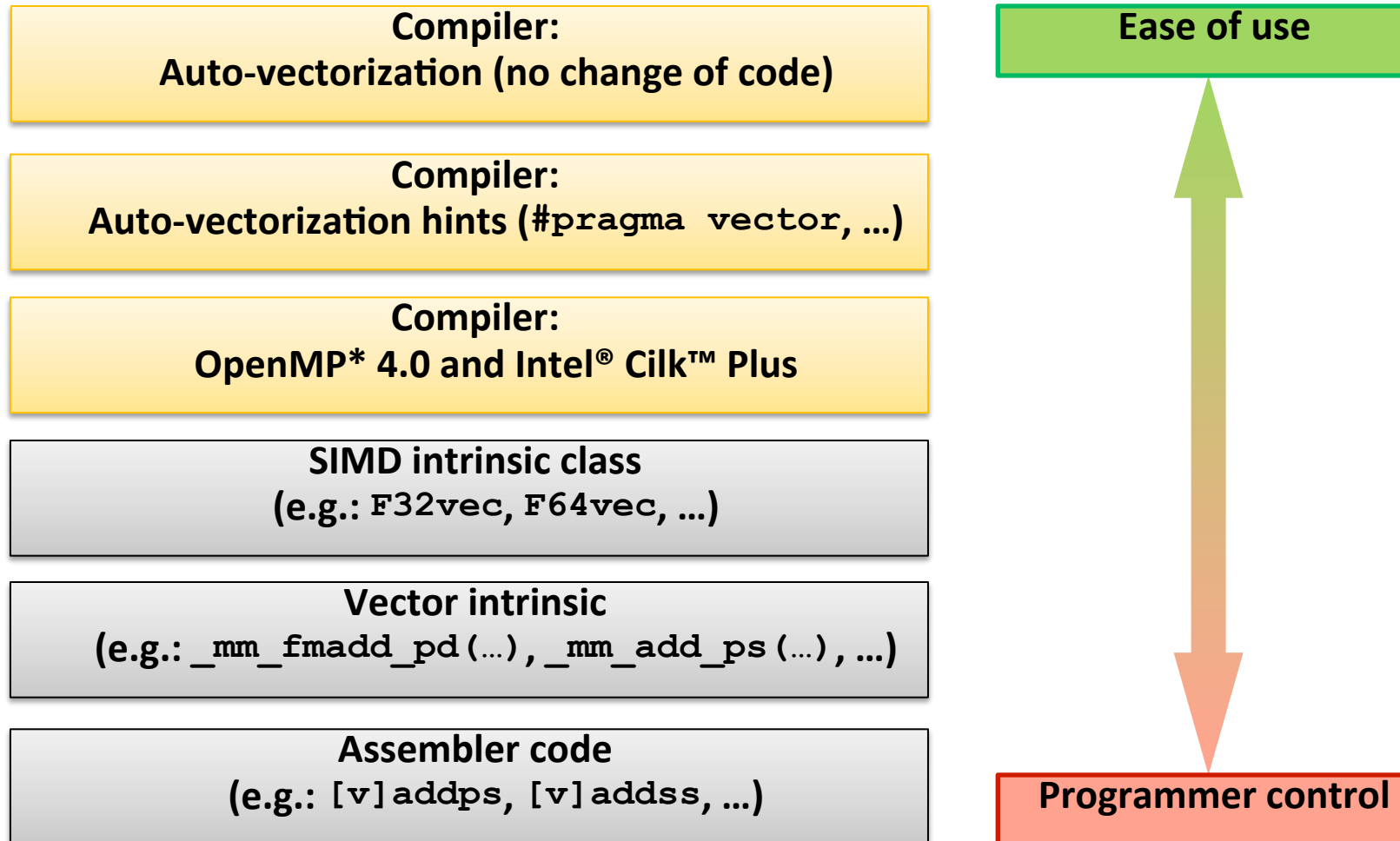
Still there are a couple of FP related switches outside of `-fp-model` which remain useful

What is Vectorization ?

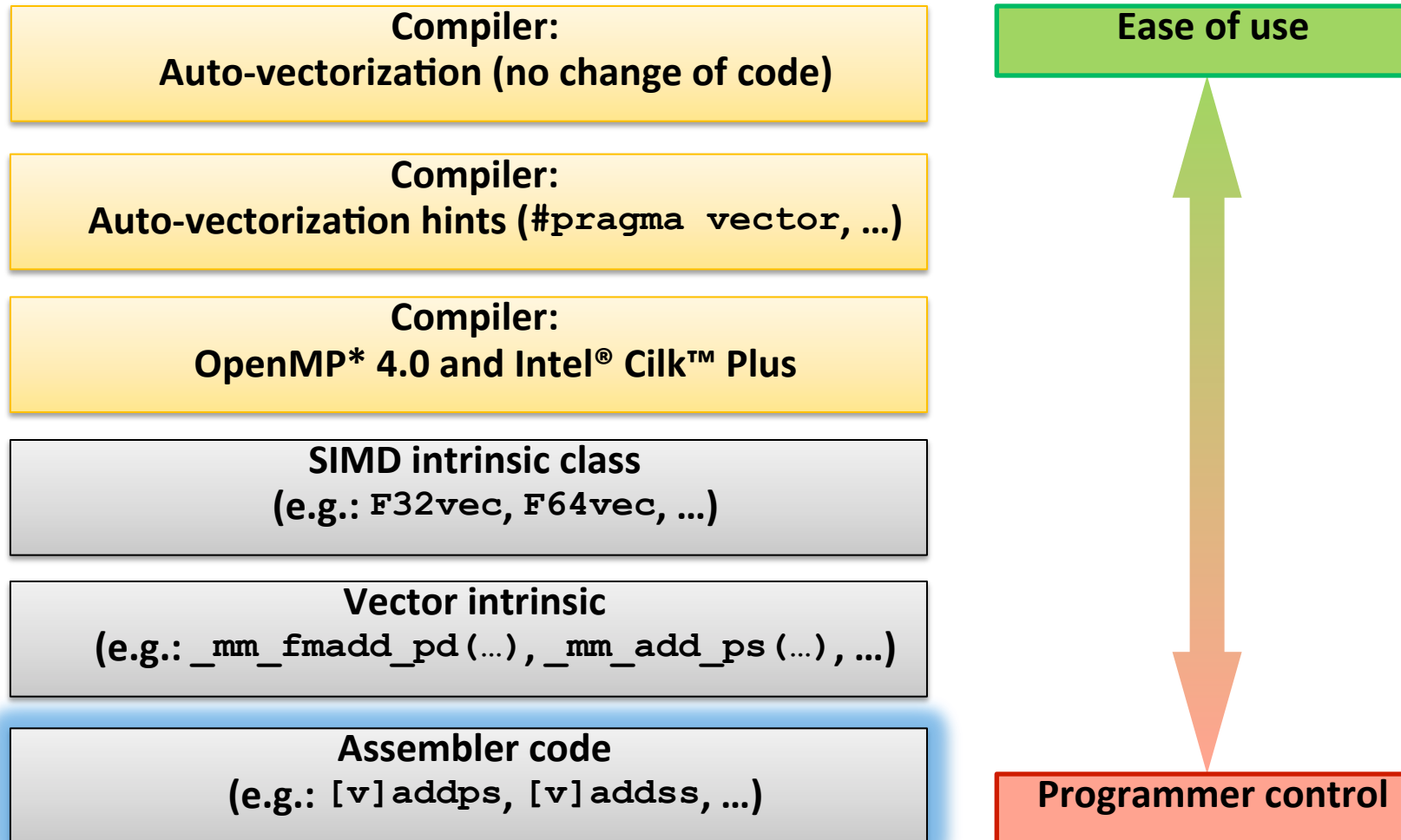
- Transform sequential code to exploit vector processing capabilities (SIMD) of Intel processors
 - Manually by explicit syntax
 - Automatically by tools like a compiler



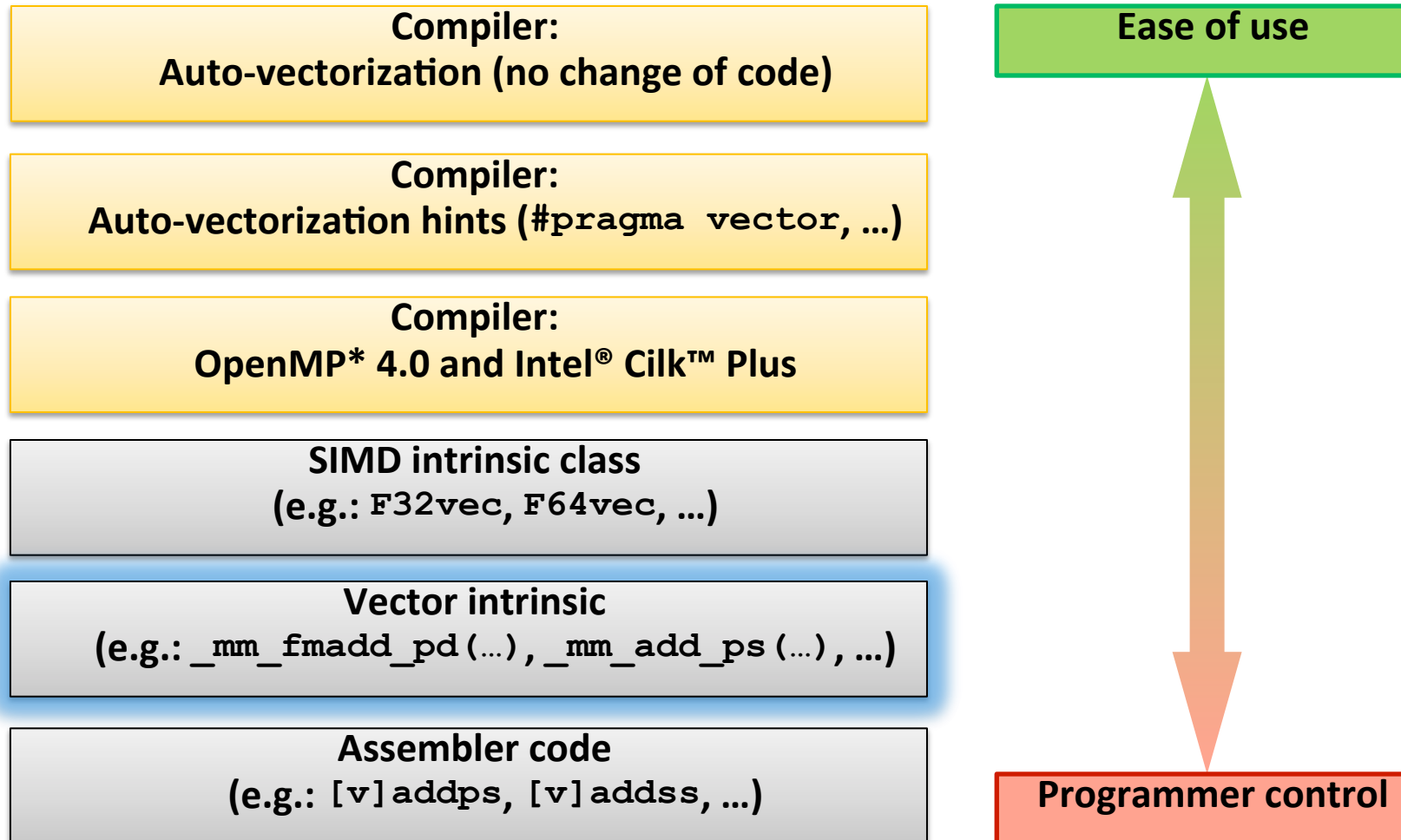
Many Ways to Vectorize



Many Ways to Vectorize



Many Ways to Vectorize



Intrinsics – Sample

- Example using AVX intrinsics:

```
#include <immintrin.h>

double A[40], B[40], C[40];
for (int i = 0; i < 40; i += 4) {
    __m256d a = _mm256_load_pd(&A[i]);
    __m256d b = _mm256_load_pd(&B[i]);
    __m256d c = _mm256_add_pd(a, b);
    _mm256_store_pd(&C[i], c);
}
```

- Example using Intel® MIC Architecture/Intel® AVX-512 intrinsics:

```
#include <immintrin.h>

double A[40], B[40], C[40];
for (int i = 0; i < 40; i += 8) {
    __m512d a = _mm512_load_pd(&A[i]);
    __m512d b = _mm512_load_pd(&B[i]);
    __m512d c = _mm512_add_pd(a, b);
    _mm512_store_pd(&C[i], c);
}
```

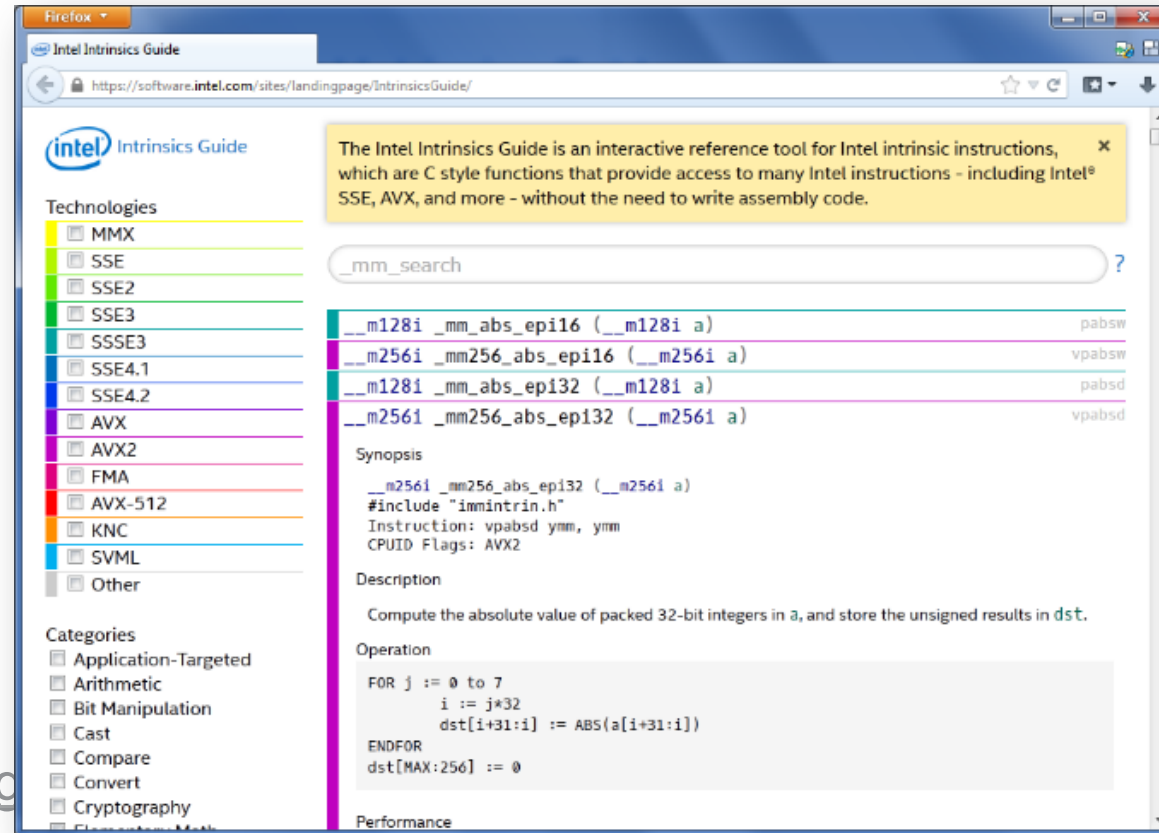
Intel® Intrinsic Guide

Intel provides an interactive intrinsics guide:

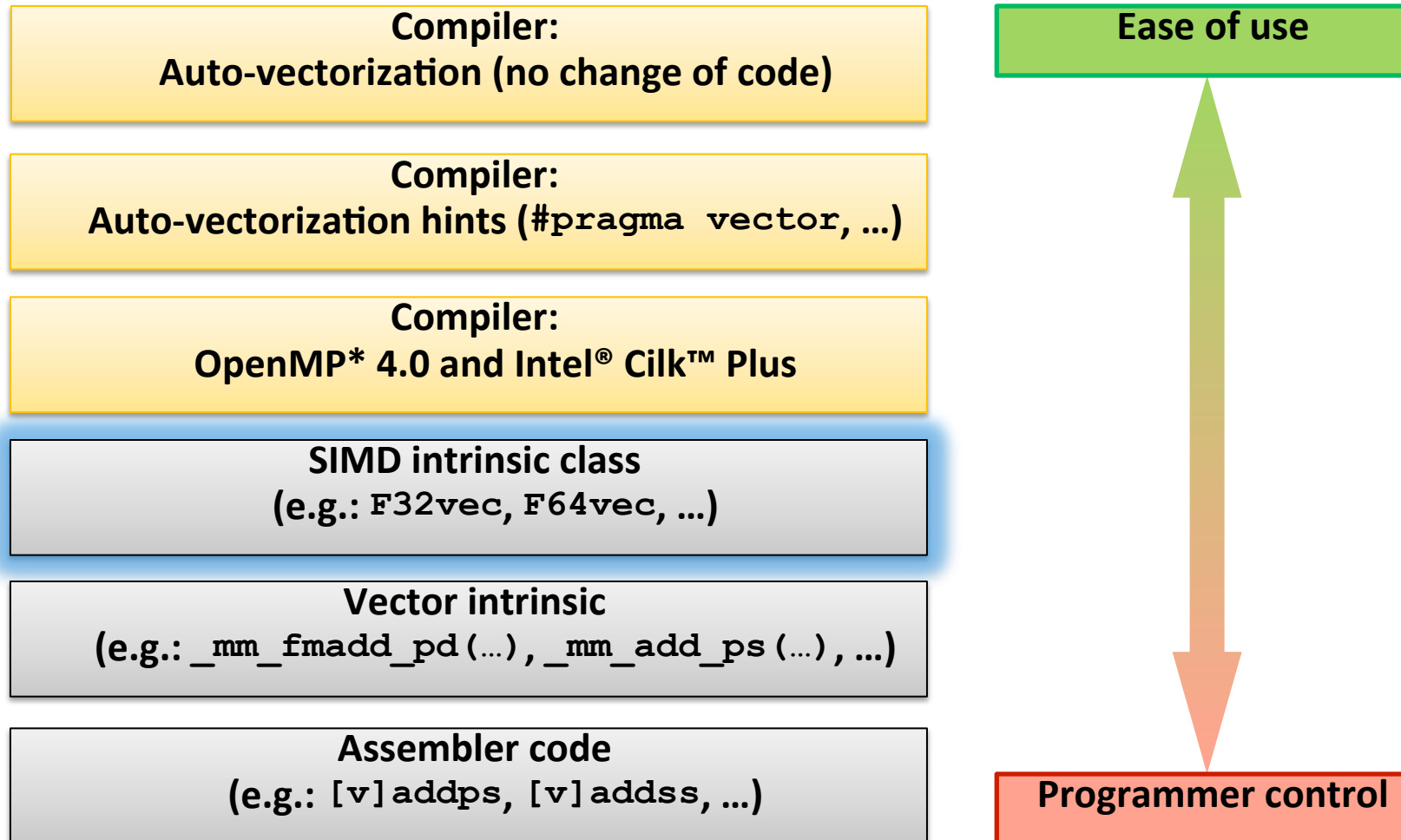
- Lists all supported intrinsics
- Sorted by SIMD feature version and generation
- Quickly find the intrinsic via instant search
- Rich documentation of each intrinsic
- Filters for technologies, types & categories

Access it here:

<https://software.intel.com/sites/landingpage/IntrinsicGuide/>



Many Ways to Vectorize



SIMD Intrinsic Class

- For a full list, please refer to the header files!
- Example for AVX:

```
#include <dvec.h>

// 4 elements per vector * 25 = 100 elements
F64vec4 A[25], B[25], C[25];

for(int i = 0; i < 25; i++)
    C[i] = A[i] + B[i];
```

Many Ways to Vectorize

Compiler:
Auto-vectorization (no change of code)

Compiler:
Auto-vectorization hints (`#pragma vector, ...`)

Compiler:
OpenMP* 4.0 and Intel® Cilk™ Plus

SIMD intrinsic class
(e.g.: `F32vec`, `F64vec`, ...)

Vector intrinsic
(e.g.: `_mm_fmadd_pd(...)`, `_mm_add_ps(...)`, ...)

Assembler code
(e.g.: `[v]addps`, `[v]addss`, ...)

Ease of use



Programmer control

Why is (automatic) Vectorization not so

easy?

```
float *A;
void vectorize()
{
    int i;
    for (i=0; i<102400; i++)
        A[i] *= 2.0f;
}
```

Using default compilation:

```
icc -c vec.c
```

Compiler 14.0 says
dependence from A to A – no
vectorization.

Compiler 15.0 vectorizes loop

Using pre-15.0 Intel Compiler

```
icc -c -opt-report test1.c
```

test1.c(4): (col. 3) remark: loop was not
vectorized: existence of vector
dependence.

test1.c(5): (col. 7) remark: vector
dependence: assumed FLOW
dependence between A line 5 and A line
5.

test1.c(5): (col. 7) remark: vector
dependence: assumed ANTI dependence
between A line 5 and A line 5.

Compiler has to assume **the Worst** ...

```
float *A;  
void vectorize()  
{  
    int i;  
    for (i=0; i<102400; i++)  
        A[i] *= 2.0f;  
}
```

Loop Body:

- Load of A
- Load of A[i]
- Multiply with 2.0f

Q: Will the store modify A?

A: Maybe

Recompile with `-ansi-alias` (14.0 – default for 15.0 now):

- `icc -opt-report -ansi-alias test1.c`
 - `test1.c(4): (col. 3) remark: LOOP WAS VECTORIZED.`

Add “`#pragma ivdep`” to the loop.

- `icc -opt-report test1b.c`
 - `test1b.c(5): (col. 3) remark: LOOP WAS VECTORIZED.`

The C/C++ standards don't allow this kind of aliasing while sometimes older application code rely on this ! The Intel compilers before 14.0 by default accepted the violation: Switch `-ansi-alias` had to be used to enforce standard conformance - since 15.0, `-ansi-alias` is the default

What about this ?

```
float A[102400];  
void vectorize()  
{  
    int i;  
    for (i=0; i<102400; i++)  
        A[i] *= 2.0f;  
}
```

```
$ gcc -c test1c.c -opt-report
```

```
test1c.c(10): (col. 5) remark: LOOP WAS VECTORIZED
```

Since no pointer is involved anymore, there can't be any aliasing and so the loop is always vectorized

Auto-vectorization of Intel Compilers: Target Architecture makes a Difference !!



```
void add(A, B, C)
double A[1000]; double B[1000]; double C[1000];
{
  int i;
  for (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® AVX

```
..B1.2:
vmovupd    (%rsp,%rax,8), %ymm0
vmovupd    32(%rsp,%rax,8), %ymm2
vmovupd    64(%rsp,%rax,8), %ymm4
vmovupd    96(%rsp,%rax,8), %ymm6
vaddpd     8032(%rsp,%rax,8), %ymm2, %ymm3
vaddpd     8000(%rsp,%rax,8), %ymm0, %ymm1
vaddpd     8064(%rsp,%rax,8), %ymm4, %ymm5
vaddpd     8096(%rsp,%rax,8), %ymm6, %ymm7
vmovupd    %ymm1, 16000(%rsp,%rax,8)
vmovupd    %ymm3, 16032(%rsp,%rax,8)
vmovupd    %ymm5, 16064(%rsp,%rax,8)
vmovupd    %ymm7, 16096(%rsp,%rax,8)
addq       $16, %rax
cmpq       $992, %rax
jb         ..B1.2
...
```

Intel® SSE4.2

```
..B1.2:
movaps     (%rsp,%rax,8), %xmm0
movaps     16(%rsp,%rax,8), %xmm1
movaps     32(%rsp,%rax,8), %xmm2
movaps     48(%rsp,%rax,8), %xmm3
addpd     8000(%rsp,%rax,8), %xmm0
addpd     8016(%rsp,%rax,8), %xmm1
addpd     8032(%rsp,%rax,8), %xmm2
addpd     8048(%rsp,%rax,8), %xmm3
movaps     %xmm0, 16000(%rsp,%rax,8)
movaps     %xmm1, 16016(%rsp,%rax,8)
movaps     %xmm2, 16032(%rsp,%rax,8)
movaps     %xmm3, 16048(%rsp,%rax,8)
addq       $8, %rax
cmpq       $1000, %rax
jb         ..B1.2
...
```

Basic Vectorization Switches I

- Linux*, OS X*: **-x<feature>**, Windows*: **/Qx<feature>**
 - 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
- Linux*, OS X*: **-ax<features>**, Windows*: **/Qax<features>**
 - Multiple code paths: baseline and optimized/processor-specific
 - Optimized code paths for Intel processors defined by **<features>**
 - 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<feature>** or **-x<feature>** (**/arch:<feature>** or **/Qx<feature>**)

Basic Vectorization Switches II

- Linux*, OS X*: **-m<feature>**, Windows*: **/arch:<feature>**
 - Neither check nor 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
- Default for Linux*: **-msse2**, Windows*: **/arch:sse2**:
 - Activated implicitly
 - Implies the need for a target processor with at least Intel® SSE2
- Default for OS X*: **-xsse3** (IA-32), **-xssse3** (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)

Basic Vectorization Switches III

- Special switch for Linux*, OS X*: **-xHost**, Windows*: **/QxHost**
 - Compiler checks SIMD features of current host processor (where built on) and makes use of latest SIMD feature available
 - Code only executes on processors with same SIMD feature or later as on build host
 - As for **-x<feature>** or **/Qx<feature>**, if “main” routine is built with **-xHost** or **/QxHost** the final executable only runs on Intel processors

Vectorization Pragma/Directive

- SIMD features can also be set on a function/subroutine level via pragmas/directives:

- C/C++:

```
#pragma intel optimization_parameter target_arch=<CPU>
```

- Fortran:

```
!DIR$ ATTRIBUTES OPTIMIZATION_PARAMETER:TARGET_ARCH= <CPU>
```

- Examples:

- C/C++:

```
#pragma intel optimization_parameter target_arch=AVX  
void optimized_for_AVX()  
{  
  ...  
}
```

- Fortran:

```
function optimized_for_AVX()  
!DIR$ ATTRIBUTES OPTIMIZATION_PARAMETER:TARGET_ARCH=AVX  
  ...  
end function
```

Control Vectorization I

- Disable vectorization:
 - Globally via switch:
Linux*, OS X*: **-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*, OS X*: **-vec**, Windows*: **/Qvec**
 - Enforce for a single loop (override compiler efficiency heuristic) if semantically correct:
C/C++: **#pragma vector always**, Fortran: **!DIR\$ VECTOR ALWAYS**
 - Influence efficiency heuristics threshold:
Linux*, OS X*: **-vec-threshold[n]**
Windows*: **/Qvec-threshold[[:]n]**
n: 100 (default; only if profitable) ... **0** (always)

Control Vectorization II

- Verify vectorization:
 - Globally:
Linux*, OS X*: **-opt-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:
C/C++: **#pragma simd**
Fortran: **!DIR\$ SIMD**
- We’ll address those later in more detail

Validating Vectorization Success I

- **Assembler code inspection (Linux*, OS X*: `-S`, `-Fa`, 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
- **Using Intel® VTune™ Amplifier:**
 - Different events can be selected to measure use of vector units, e.g. `FP_COMP_OPS_EXE.SSE_PACKED_[SINGLE|DOUBLE]`
 - For Intel® MIC Architecture: Use metric **Vectorization Intensity**
- **Difference method:**
 1. Compile and benchmark with `-no-vec//Qvec-` or on a loop by loop basis via `#pragma novector/!DIR$NOVECTOR`
 2. Compile and benchmark with selected SIMD feature
 3. Compare runtime differences

Validating Vectorization Success II

- **Intel® Software Development Emulator:**

- Emulate (future) Intel® Architecture Instruction Set Extensions (e.g. Intel® AVX-512, Intel® MPX, ...)
- Use the “mix histogramming tool” to check for instructions using vectors
- Also possible to debug the application while emulated

- Source:

<https://software.intel.com/en-us/articles/intel-software-development-emulator>



- **Intel® Architecture Code Analyzer:**

- Statically analyze the data dependency, throughput and latency of code snippets (aka. kernels)
- Considers ideal front-end, out-of-order engine and memory hierarchy conditions
- Identifies binding of the kernel instructions to the processor ports & critical path

- Source:

<https://software.intel.com/en-us/articles/intel-architecture-code-analyzer/>



Validating Vectorization Success III

- **Optimization report:**
 - Linux*, OS X*: `-opt-report=<n>`, Windows*: `/Qopt-report:<n>`
`n: 0, ..., 5` specifies level of detail; `2` is default (more later)
 - Prints optimization report with vectorization analysis
 - Also known as vectorization report for Intel® C++/Fortran Compiler before 15.0:
Linux*, OS X*: `-vec-report=<n>`, Windows*: `/Qvec-report:<n>`
Deprecated, don't use anymore – use optimization report instead!
- **Optimization report phase:**
 - Linux*, OS X*: `-opt-report-phase=<p>`,
Windows*: `/Qopt-report-phase:<p>`
 - `<p>` is `all` by default; use `vec` for just the vectorization report
- **Optimization report file:**
 - Linux*, OS X*: `-opt-report-file=<f>`, Windows*: `/Qopt-report-file:<f>`
 - `<f>` can be `stderr`, `stdout` or a file (default: `*.optrpt`)

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 -opt-report=5
ifort: remark #10397: optimization reports are generated in *.optrpt
files in the output location

$ cat novec.optrpt
...
LOOP BEGIN at novec.f90(4,5)
  remark #15344: loop was not vectorized: vector dependence prevents
vectorization
  remark #15346: vector dependence: assumed FLOW dependence between y
line 5 and y line 5
  remark #25436: completely unrolled by 9
LOOP END
...
```

Reasons for Vectorization Fails I

Most frequent reasons:

- Data dependence
- Alignment
- Unsupported loop structure
- Non-unit stride access
- Function calls/in-lining
- Non-vectorizable Mathematical functions
- Data types
- Control dependence
- Bit masking

All those are common!

Reasons for Vectorization Fails II

Other reasons:

- Outer loop of loop nesting cannot be vectorized
- Loop body too complex (register pressure)
- Vectorization seems inefficient (low trip count)
- Many more

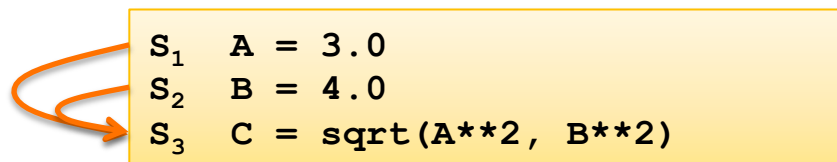
Those are less likely!

Data/Control Dependence

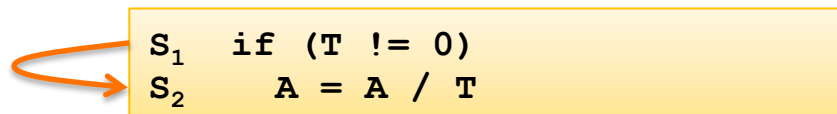
Dependence is a key term for vectorization:

- Vectorization is a transformation changing the execution order of statements
- The execution order of statements as defined by the program source code can be changed as long as the dependencies between all statements are preserved

A dependence either is a data or control dependence :



Data dependence from
S₁ to S₃ and from S₂ to S₃



Control dependence from
S₁ to S₂

Data Dependence

Definition of data dependence:

There is a data dependence from statement S_1 to statement S_2 (written as $S_1 \delta S_2$) if and only if:

- There is a potential execution flow from S_1 to S_2
- S_1 and S_2 reference a common memory location S_1 or S_2 write to

Note: S_1 and S_2 can be the very same statement

Data dependence classification:

- $S_1 \delta^F S_2$: S_1 writes, S_2 reads: **Flow Dependence**
- $S_1 \delta^A S_2$: S_1 reads, S_2 writes: **Anti Dependence**
- $S_1 \delta^O S_2$: S_1 writes, S_2 writes: **Output Dependence**

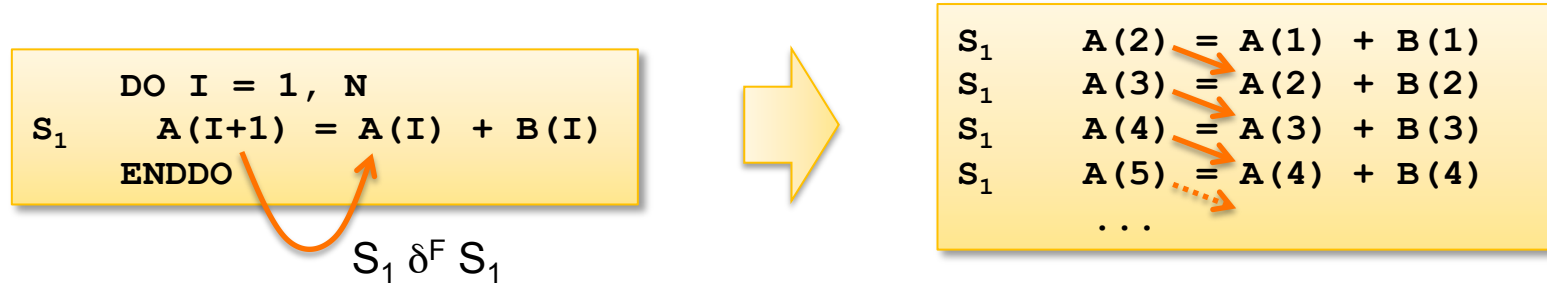
```
S1  X = ...  
S2  ... = X
```

```
S1  ... = X  
S2  X = ...
```

```
S1  X = ...  
S2  X = ...
```

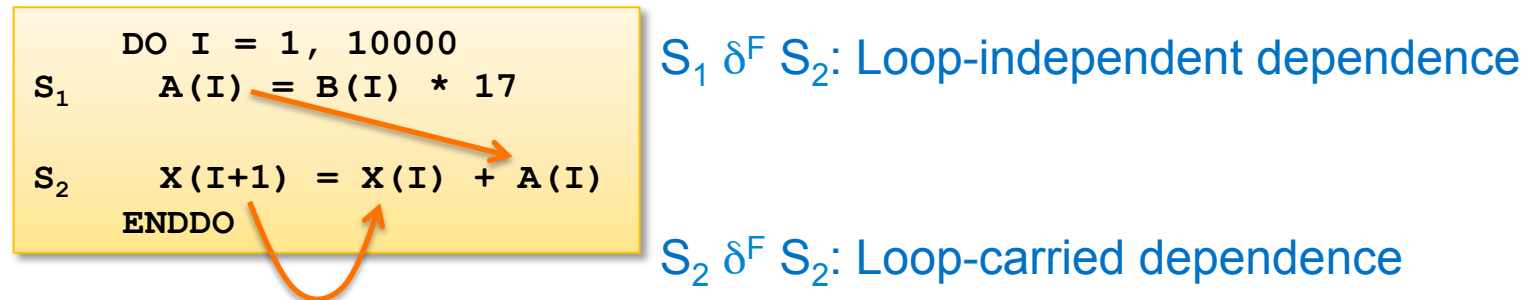
Data Dependence in Loops

Dependencies in loops become more obvious by virtually unrolling the loop:



In case the dependency requires execution of any previous loop iteration, we call it **loop-carried dependence**. Otherwise, **loop-independent dependence**.

E.g.:



Dependence and Vectorization

Vectorization of a loop is similar to parallelization (loop iterations executed in parallel), however not identical:

- **Parallelization** requires all iterations to be independent to be executed in any order: Loop-carried dependencies are not permitted; loop-independent dependencies are OK
- **Vectorization** is applied to single operations of the loop body: The same operations can be applied for multiple iterations at once if they follow serial order; both loop-carried & loop-independent dependencies need to be taken into account!

Example: Loop cannot be parallelized but vectorization possible:

```
DO I = 1, N
  A(I + 1) = B(I) + C
  D(I) = A(I) + E
END DO
```



```
A(2:N + 1) = B(1:N) + C
D(1:N) = A(1:N) + E
```

Key Theorem for Vectorization

A loop can be vectorized if and only if there is no cyclic dependency chain between the statements of the loop body!

Example:

Although we have a cyclic dependency chain, 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
```

Alignment

Caveat with using unaligned memory access:

- Unaligned loads and stores can be **very slow** due to higher I/O because two cache-lines need to be loaded/stored (not always, though)
- Compiler can mitigate expensive unaligned memory operations by using two partial loads/stores – **still slow**
(e.g. two 64 bit loads instead of one 128 bit unaligned load)
- The compiler can use “versioning” in case alignment is unclear:
Run time checks for alignment to use fast aligned operations if possible, the slower operations otherwise – **better but limited**

Best performance: User defined aligned memory

- 16 byte for SSE
- 32 byte for AVX
- 64 byte for Intel® MIC Architecture & Intel® AVX-512

Alignment Hints for C/C++ I

- Aligned heap memory allocation by intrinsic/library call:
 - `void* _mm_malloc(int size, int base)`
 - Linux*, OS X* only:
`int posix_memaligned(void **p, size_t base, size_t size)`
- `#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!

Alignment Hints for C/C++ II

- Align attribute for variable declarations:
 - Linux*, OS X*, Windows*: `__declspec(align(base)) <var>`
 - Linux*, OS X*: `<var> __attribute__((aligned(base)))`
 - **Portability caveat:**
`__declspec` is not known for GCC and `__attribute__` not for Microsoft Visual Studio*!
- Hint that start address of an array is aligned (Intel Compiler only):
`__assume_aligned(<array>, base)`

Alignment Hints for Fortran

- **!DIR\$ VECTOR [ALIGNED|UNALIGNED]**
 - 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!
- Hint that an entity in memory is aligned:
!DIR\$ ASSUME_ALIGNED address1:base [, address2:base] ...
- Align variables:
!DIR\$ ATTRIBUTES ALIGN: base :: variable
- Align data items globally:
Linux*, OS X*: **-align <a>**, Windows*: **/align:<a>**
 - **<a>** can be **array<n>byte** with **<n>** defining the alignment for arrays
 - Other values for **<a>** are also possible, e.g.: **[no]commons**, **[no]records**, ...

All are Intel® Fortran Compiler only directives and options!

Alignment & Processor Architecture

- Instructions with unaligned access are very slow except for SSE vector memory operations (128 bit) on 2nd and 3rd generation Intel® Core™ processors (as fast as aligned access)
- For AVX vectors (256 bit) unaligned accesses are slower compared to their aligned accesses, even on 3rd generation Intel® Core™ processors
- Independent on processor generation and instruction set features, one unaligned instructions can replace a sequence of multiple instructions:
 - Fewer instructions result in less cycles, better use of instruction-cache and less power consumption
 - To benefit make sure to at least use latest SSE/AVX feature set (default for Intel® C++/Fortran Compiler is Intel® SSE2)
- **Attention:**
When using SSE instructions directly (e.g. intrinsics) any aligned move on unaligned data still fails!

Alignment Impact: Example

Compiled both cases using **-xAVX**:

```
void mult(double* a, double* b, double* c)
{
    int i;
    #pragma vector unaligned
    for (i = 0; i < N; i++)
        c[i] = a[i] * b[i];
}
```

```
..B2.2:
vmovupd    (%rdi,%rax,8), %xmm0
vmovupd    (%rsi,%rax,8), %xmm1
vinsertf128 $1, 16(%rsi,%rax,8), %ymm1, %ymm3
vinsertf128 $1, 16(%rdi,%rax,8), %ymm0, %ymm2
vmulpd     %ymm3, %ymm2, %ymm4
vmovupd   %xmm4, (%rdx,%rax,8)
vextractf128 $1, %ymm4, 16(%rdx,%rax,8)
addq       $4, %rax
cmpq       $1000000, %rax
jb         ..B2.2
```

More efficient

```
void mult(double* a, double* b, double* c)
{
    int i;
    #pragma vector aligned
    for (i = 0; i < N; i++)
        c[i] = a[i] * b[i];
}
```

```
..B2.2:
vmovupd    (%rdi,%rax,8), %ymm0
vmulpd     (%rsi,%rax,8), %ymm0, %ymm1
vmovntpd  %ymm1, (%rdx,%rax,8)
addq       $4, %rax
cmpq       $1000000, %rax
jb         ..B2.2
```


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;  
  
}
```

Non-Unit Stride Access

- Non-consecutive memory locations are being accessed in the loop
- Vectorization works best with contiguous memory accesses
- Vectorization still be possible for non-contiguous memory access, but...
 - Data arrangement operations might be too expensive (e.g. access pattern linear/regular)
 - Vectorization report issued when too expensive:
Loop was not vectorized: vectorization possible but seems inefficient
- Examples:

```
for(i = 0; i <= MAX; i++) {  
  for(j = 0; j <= MAX; j++) {  
    D[i][j] += 1;           // Unit stride  
    D[j][i] += 1;           // Non-unit stride but linear  
    A[j * j] += 1;          // Non-unit stride  
    A[B[j]] += 1;           // Non-unit stride (scatter)  
    if(A[MAX - j] == 1) last = j; // Non-unit stride  
  }  
}
```

Function Calls/In-lining I

- Function calls prevent vectorization in general
- Exceptions:
 - Call of intrinsic routines such as mathematical functions: Implementation is known to compiler
 - Successful in-lining of called routine: IPO enables in-lining of routines across source files

```
for (i = 1; i < nx; i++) {  
    x = x0 + i * h;  
    sumx = sumx + func(x, y, xp, yp);  
}  
  
// Defined in different compilation unit!  
float func(float x, float y, float xp, float yp)  
{  
    float denom;  
    denom = (x - xp) * (x - xp) + (y - yp) * (y - yp);  
    denom = 1. / sqrt(denom);  
    return denom;  
}
```

Function Calls/In-lining II

- Success of in-lining can be verified using the optimization report:
Linux*, OS X*: **-opt-report=<n> -opt-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”:
 - Short Vector Math Library (**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 which is optimized for throughput
- Routines in **libsvml** can also be called explicitly, using intrinsics (see manual)
- 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	

How to Succeed in Vectorization?

- Most frequent reason of failing vectorization is **Dependence**:
Minimize dependencies among iterations by design!
- **Alignment**: Align your arrays/data structures
- **Function calls in loop body**: Use aggressive in-lining (IPO)
- **Complex control flow/conditional branches**:
Avoid them in loops by creating multiple versions of loops
- **Unsupported loop structure**: Use loop invariant expressions
- **Not inner loop**:
Manual loop interchange possible? Intel Compilers 12.1 and higher can do outer loop vectorization now as well!
- **Mixed data types**:
Avoid type conversions in rare cases Intel Compiler cannot do automatically

How to Succeed in Vectorization? II

- **Non-unit stride between elements:**
Possible to change algorithm to allow linear/consecutive access?
- **Loop body too complex reports:** Try splitting up the loops!
- **Vectorization seems inefficient reports:**
Enforce vectorization, benchmark and verify results!

!DIR\$ SIMD Clauses for Fortran

- **VECTORLENGTH(n1 [,n2] ...)**
n1, n2, ... must be **2, 4, 8, ...**: The compiler can assume a safe vectorization for a vector length of **n1, n2, ...**
- **PRIVATE(v1, v2, ...)**
Variables private to each iteration; supersets (extensions):
 - **FIRSTPRIVATE(...)**: initial value is broadcast to all private instances
 - **LASTPRIVATE(...)**: last value is copied out from the last iteration instance
- **LINEAR(v1:step1, v2:step2, ...)**
For every iteration of original scalar loop **v1** is incremented by **step1, ...** etc. Therefore it is incremented by **step1 * VL** for the vectorized loop.
- **REDUCTION(operator:v1, v2, ...)**
Variables **v1, v2, ...** etc. are reduction variables for operation **operator**
- **[NO]ASSERT**
Warning (default: **NOASSERT**) or error with failed vectorization

!DIR\$ SIMD Example for Fortran

Problem:

“Enforced” vectorization still fails
with the following message:

loop was not vectorized: conditional assignment to a scalar

loop was not vectorized with "simd"

Solution:

Clarify that scalar is a reduction with
operator **+**.

Attention:

Same as for OpenMP* reduction variables can only be associated with the operator **+**.

```
!DIR$ SIMD
do i = 1,n
  if (a(i) .GT. 0) then
    sum2 = sum2 + a(i) * b(i)
  else
    sum2 = sum2 + a(i)
  endif
enddo
```

```
!DIR$ SIMD REDUCTION(+:sum2)
do i = 1,n
  if (a(i) .GT. 0) then
    sum2 = sum2 + a(i) * b(i)
  else
    sum2 = sum2 + a(i)
  endif
enddo
```

IVDEP vs. SIMD Pragma/Directives

Differences between IVDEP & SIMD pragmas/directives:

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

- Ignore vector dependencies (IVDEP): Ignore assumed but not proven dependencies for a loop

- Example:

```
void foo(int *a, int k, int c, int m)
{
    #pragma ivdep
    for (int i = 0; i < m; i++)
        a[i] = a[i + k] * c;
}
```

- **#pragma simd** (C/C++) or **!DIR\$ SIMD** (Fortran):

- Aggressive version of IVDEP: Ignores **all** dependencies inside a loop and ignore efficiency heuristic
- It's an imperative that forces the compiler try everything to vectorize
- **Attention:** This can break semantically correct code!
However, it can **vectorize** code legally in some cases that wouldn't be possible otherwise!

OpenMP* 4.0

- OpenMP* 4.0 ratified July 2013
- Specifications:
<http://openmp.org/wp/openmp-specifications/>
- Well established in HPC – parallelism is critical there
- Extension to C/C++ & Fortran
- New features with 4.0:
 - Target Constructs: Accelerator support
 - Distribute Constructs/Teams: Better hierarchical assignment of workers
 - **SIMD (Data Level Parallelism!)**
 - Task Groups/Dependencies: Runtime task dependencies & synchronization
 - Affinity: Pinning workers to cores/HW threads
 - Cancellation Points/Cancel: Defined abort locations for workers
 - User Defined Reductions: Create own reductions

Pragma SIMD

- Pragma SIMD:

The simd construct can be applied to a loop to indicate that the loop can be transformed into a SIMD loop (that is, multiple iterations of the loop can be executed concurrently using SIMD instructions).

[OpenMP* 4.0 API: 2.8.1]

- Syntax:

```
#pragma omp simd [clause [,clause]...]  
    for-loop
```

- For-loop has to be in “canonical loop form” (see OpenMP 4.0 API:2.6)

- Random access iterators required for induction variable (integer types or pointers for C++)
- Limited test and in-/decrement for induction variable
- Iteration count known before execution of loop
- ...

Pragma SIMD Clauses

- **safelen**(**n1** [, **n2**] ...)
n1, **n2**, ... must be power of 2: The compiler can assume a vectorization for a vector length of **n1**, **n2**, ... to be safe
- **private**(**v1**, **v2**, ...): Variables private to each iteration
 - **lastprivate**(...): last value is copied out from the last iteration instance
- **linear**(**v1:step1**, **v2:step2**, ...)
For every iteration of original scalar loop **v1** is incremented by **step1**, ... etc. Therefore it is incremented by **step1 * vector length** for the vectorized loop.
- **reduction**(**operator:v1**, **v2**, ...)
Variables **v1**, **v2**, ... etc. are reduction variables for operation **operator**
- **collapse**(**n**): Combine nested loops – collapse them
- **aligned**(**v1:base**, **v2:base**, ...): Tell variables **v1**, **v2**, ... are aligned; (default is architecture specific alignment)

Pragma SIMD Example

Ignore data dependencies, indirectly mitigate control flow dependence & assert alignment:

```
void vec1(float *a, float *b, int off, int len)
{
  #pragma omp simd safelen(32) aligned(a:64, b:64)
  for(int i = 0; i < len; i++)
  {
    a[i] = (a[i] > 1.0) ?
           a[i] * b[i] :
           a[i + off] * b[i];
  }
}
```

```
LOOP BEGIN at simd.cpp(4,5)
  remark #15388: vectorization support: reference a has aligned access [ simd.cpp(6,9) ]
  remark #15388: vectorization support: reference b has aligned access [ simd.cpp(6,9) ]
  ...
  remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
  ...
LOOP END
```

SIMD-Enabled Functions

- **SIMD-Enabled Function (aka. declare simd construct):**
The declare simd construct can be applied to a function [...] to enable the creation of one or more versions that can process multiple arguments using SIMD instructions from a single invocation from a SIMD loop.
[OpenMP* 4.0 API: 2.8.2]
- **Syntax:**
`#pragma omp declare simd [clause [,clause]...]`
function definition or declaration
- **Intent:**
Express work as scalar operations (kernel) and let compiler create a vector version of it. The size of vectors can be specified at compile time (SSE, AVX, ...) which makes it portable!
- **Remember:**
Both the function definition as well as the function declaration (header file) need to be specified like this!

SIMD-Enabled Function Clauses

- **simdlen(len)**
len must be power of 2: Allow as many elements per argument (default is implementation specific)
- **linear(v1:step1, v2:step2, ...)**
Defines **v1, v2, ...** to be private to SIMD lane and to have linear (**step1, step2, ...**) relationship when used in context of a loop
- **uniform(a1, a2, ...)**
Arguments **a1, a2, ...** etc. are not treated as vectors (constant values across SIMD lanes)
- **inbranch, notinbranch**: SIMD-enabled function called only inside branches or never
- **aligned(a1:base, a2:base, ...)**: Tell arguments **a1, a2, ...** are aligned; (default is architecture specific alignment)

SIMD-Enabled Function Example

Ignore data dependencies, indirectly mitigate control flow dependence & assert alignment:

```
#pragma omp declare simd simdlen(16) notinbranch uniform(a, b, off)
float work(float *a, float *b, int i, int off)
{
    return (a[i] > 1.0) ? a[i] * b[i] : a[i + off] * b[i];
}

void vec2(float *a, float *b, int off, int len)
{
    #pragma omp simd safelen(64) aligned(a:64, b:64)
    for(int i = 0; i < len; i++)
```

```
INLINE REPORT: (vec2(float *, float *, int, int)) [4/9=44.4%] simd.cpp(8,1)
-> INLINE: (12,16) work(float *, float *, int, int) (isz = 18) (sz = 31)

LOOP BEGIN at simd.cpp(10,5)
    remark #15388: vectorization support: reference a has aligned access [ simd.cpp(4,20) ]
    remark #15388: vectorization support: reference b has aligned access [ simd.cpp(4,20) ]
    ...
    remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
    ...
LOOP END
```

Target Constructs (Fortran)

!\$omp target [*clause*[[,] *clause*],...] *new-line*
structured-block

!\$omp end target

Clauses: *device*(*scalar-integer-expression*)
 map(*alloc* | *to* | *from* | *tofrom*: *list*)
 if(*scalar-expr*)

!\$omp target data [*clause*[[,] *clause*],...] *new-line*
structured-block

!\$omp end target data

Clauses: *device*(*scalar-integer-expression*)
 map(*alloc* | *to* | *from* | *tofrom*: *list*)
 if(*scalar-expr*)

!\$omp target update [*clause*[[,] *clause*],...] *new-line*

Clauses: *to*(*list*)
 from(*list*)
 device(*integer-expression*)
 if(*scalar-expression*)

SIMD loops: syntax

#pragma omp simd [*clauses*]

for-loop

!\$omp simd [*clauses*]

do-loops

!\$omp end simd]

Loop has to be in “Canonical loop form”

- as do/for worksharing

SIMD loop clauses

safelen (length)

- Maximum number of iterations that can run concurrently without breaking a dependence
 - in practice, maximum vector length

linear (list[:linear-step])

- The variable value is in relationship with the iteration number
 - $x_i = x_{\text{orig}} + i * \text{linear-step}$

aligned (list[:alignment])

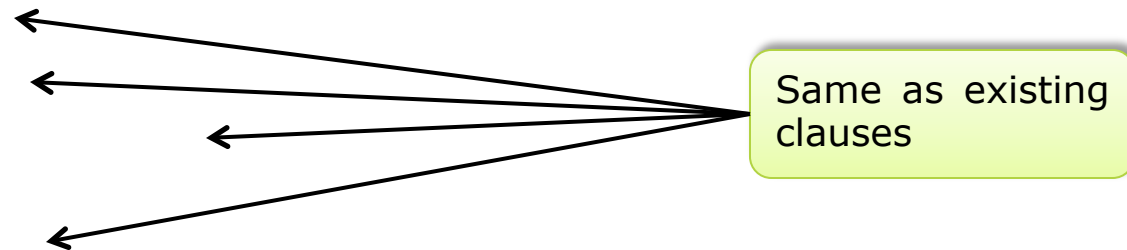
- Specifies that the list items have a given alignment
- Default is alignment for the architecture

private (list)

lastprivate (list)

reduction (operator:list)

collapse (n)



SIMD loop example

```
double pi()  
{  
    double pi = 0.0;  
    double t;  
  
    #pragma omp simd private(t) reduction(+:pi)  
    for (i=0; i<count; i++) {  
        t = (double)((i+0.5)/count);  
        pi += 4.0/(1.0+t*t);  
    }  
    pi /= count  
    return pi;  
}
```

OpenMP* 4.0: Compilers

The following compilers support OpenMP* 4.0:

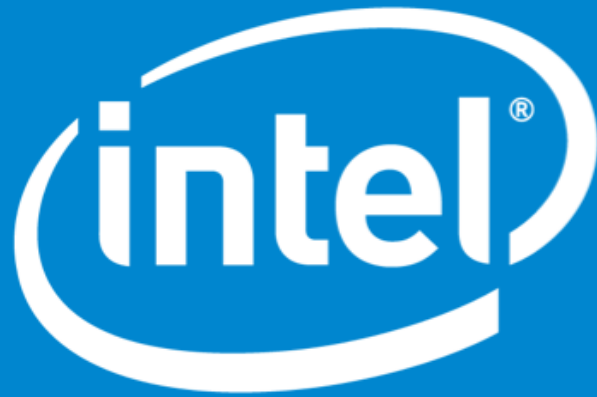
- GNU* GCC 4.9:
4.9 for C/C++ (4.9.1 for Fortran); no accelerator support (yet)
- clang/LLVM 3.5:
Not official yet but development branch exists: <http://clang-omp.github.io/>
- Intel® C++/Fortran Compiler:
Beginning with 14.0; full 15.0 (except user defined reductions)

SIMD extensions require at least **-fopenmp-simd** (or **-fopenmp**)!

OpenMP* runtime is not needed, though.

Summary

- Intel® C++ Compiler and Intel® Fortran Compiler provide sophisticated and flexible support for vectorization
- They also provide a rich set of reporting features that help verifying vectorization and optimization in general
- Directives and compiler switches permit fine-tuning for vectorization
- Vectorization can even be enforced for certain cases where language standards are too restrictive
- Understanding of concepts like dependency and alignment is required to take advantage from SIMD features
- Intel® C++/Fortran Compiler can create multi-version code to address a broad range of processor generations, Intel and non-Intel processors and individually exploiting their feature set



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.

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

Optimization Notice

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

Notice revision #20110804

