VECTORIZATION FOR INTEL® C++ & FORTRAN COMPILER

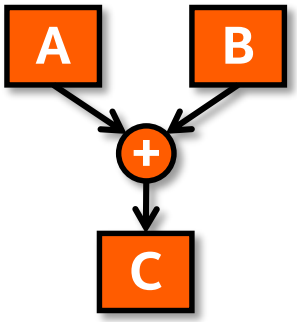
Agenda

- **Introduction to SIMD for Intel® Architecture**
- Vector Code Generation
- Compiler & Vectorization
- Validating Vectorization Success
- Reasons for Vectorization Fails
- Summary

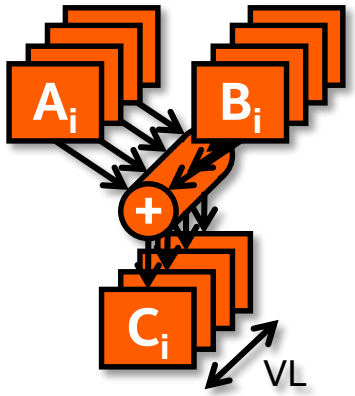
Vectorization

- **S**ingle **I**nstruction **M**ultiple **D**ata (SIMD):
  - Processing vector with a single operation
  - Provides data level parallelism (DLP)
  - Because of DLP more efficient than scalar processing
- **Vector:**
  - Consists of more than one element
  - Elements are of same scalar data types (e.g. floats, integers, …)
- **Vector length (VL):** Elements of the vector

Scalar Processing



Vector Processing

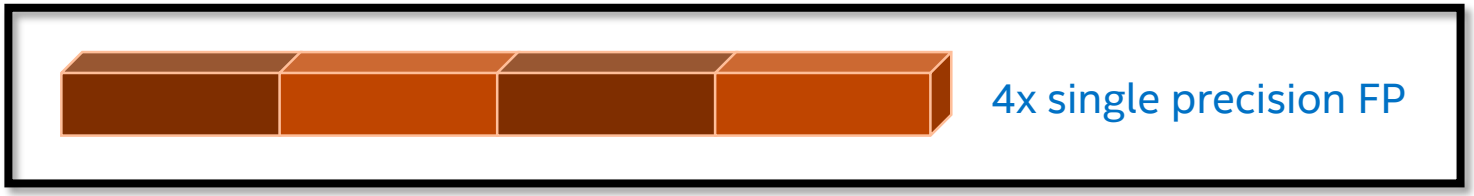


SIMD & Intel® Architecture

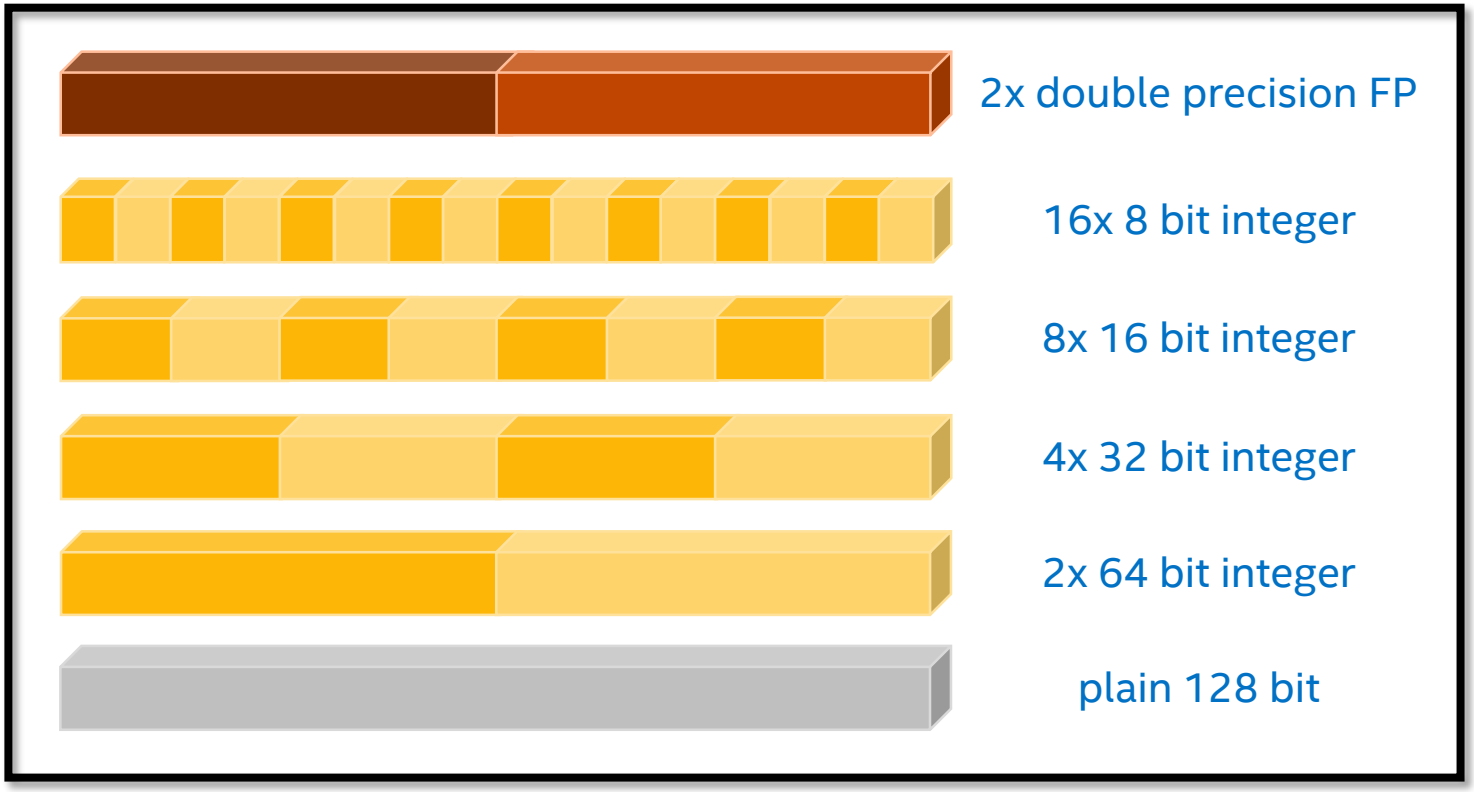
- SIMD instructions:
  - One single machine instruction for vector processing
  - Vector lengths are fixed (2, 4, 8, 16)
  - Synchronous execution on elements of vector(s)
    ⇨ Results are available at the same time
  - Masking possible to omit operations on selected elements
- SIMD is key for data level parallelism for years:
  - **64 bit** Multi-Media Extension (MMX™)
  - **128 bit** Intel® Streaming SIMD Extensions (Intel® SSE, SSE2, SSE3, SSE4.1, SSE4.2) and Supplemental Streaming SIMD Extensions (SSSE3)
  - **256 bit** Intel® Advanced Vector Extensions (Intel® AVX)
  - **512 bit** vector instruction set extension of Intel® Many Integrated Core Architecture (Intel® MIC Architecture) and Intel® Advanced Vector Extensions 512 (Intel® AVX-512)

SSE Vector Types

Intel® SSE



Intel® SSE2



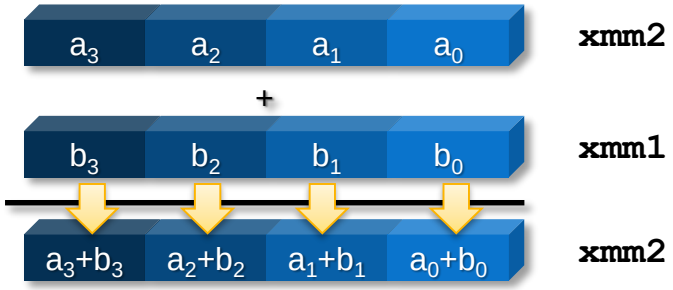
SSE Packed vs. Scalar

- **Packed** SSE instructions operate on all elements per vector
- Most of these instructions have **scalar** versions operating only on one element of vector
- Avoid scalar versions and only **use packed instructions** to exploit SIMD capabilities!

Packed single-precision FP Addition:

```
addps xmm2, xmm1
```

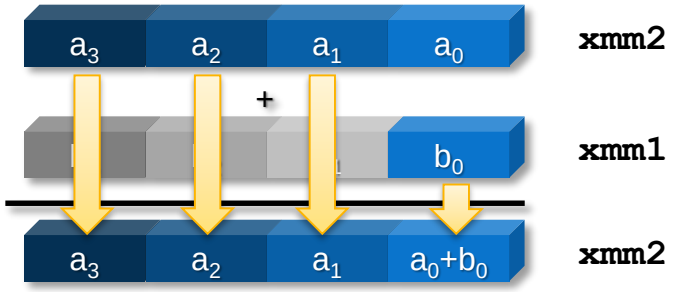
single-precision FP data type
packed execution mode



Scalar single-precision FP Addition:

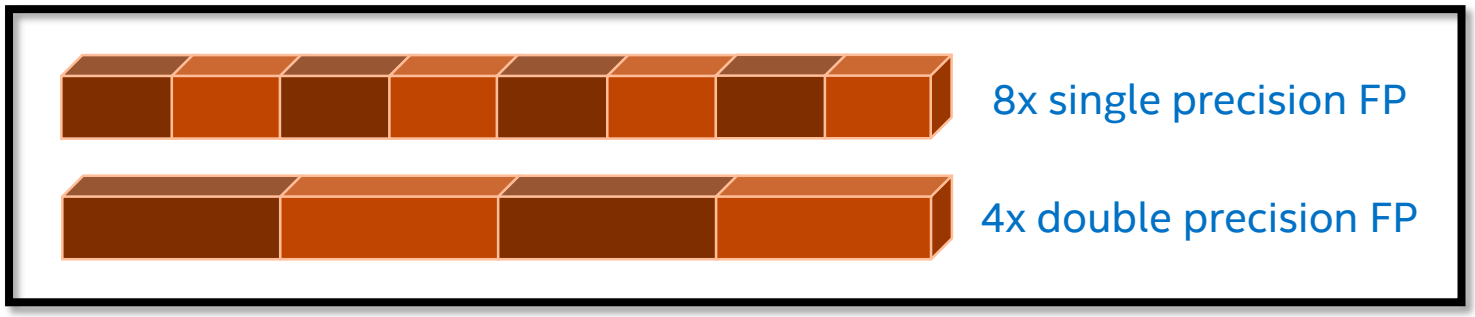
```
addss xmm2, xmm1
```

single-precision FP data type
scalar execution mode

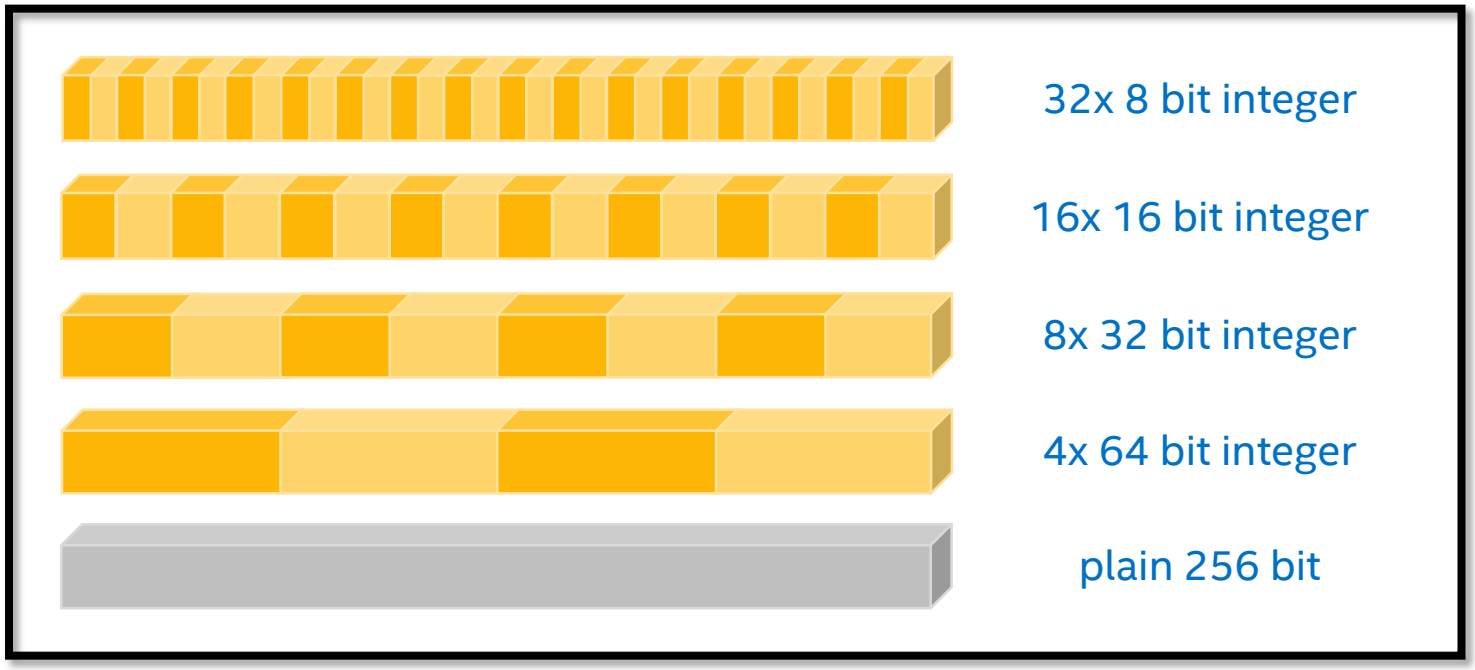


AVX Vector Types

Intel® AVX

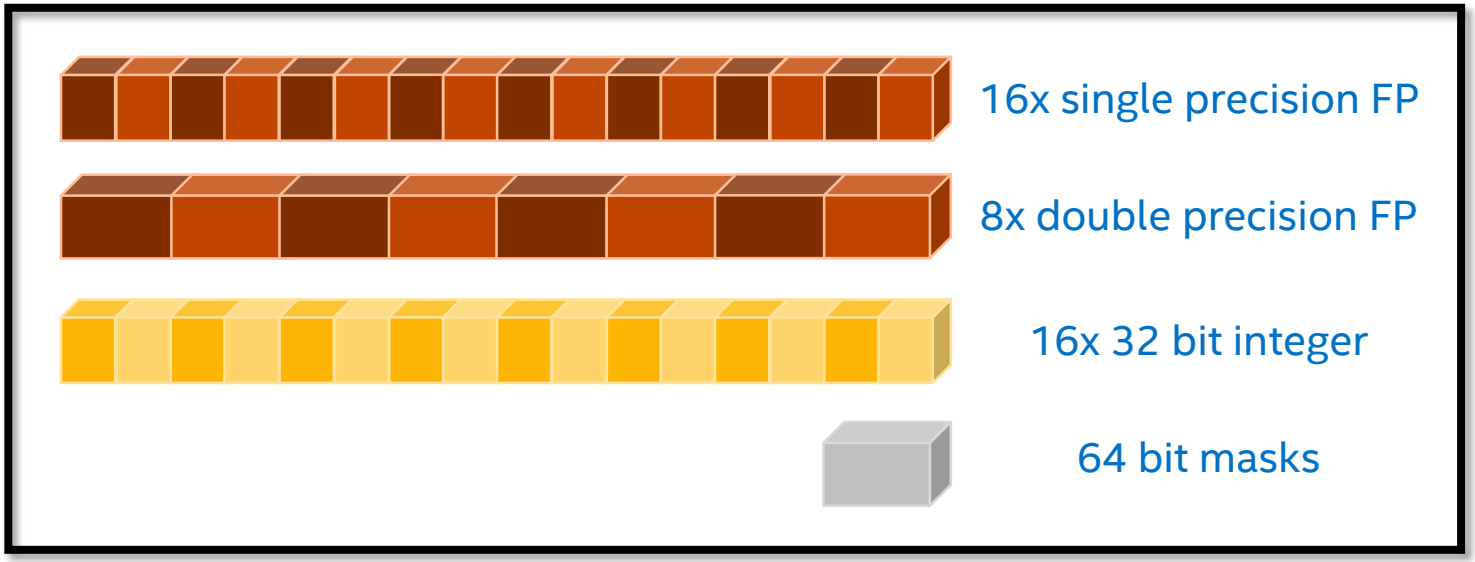


Intel® AVX2



Intel® MIC Architecture Vector Types

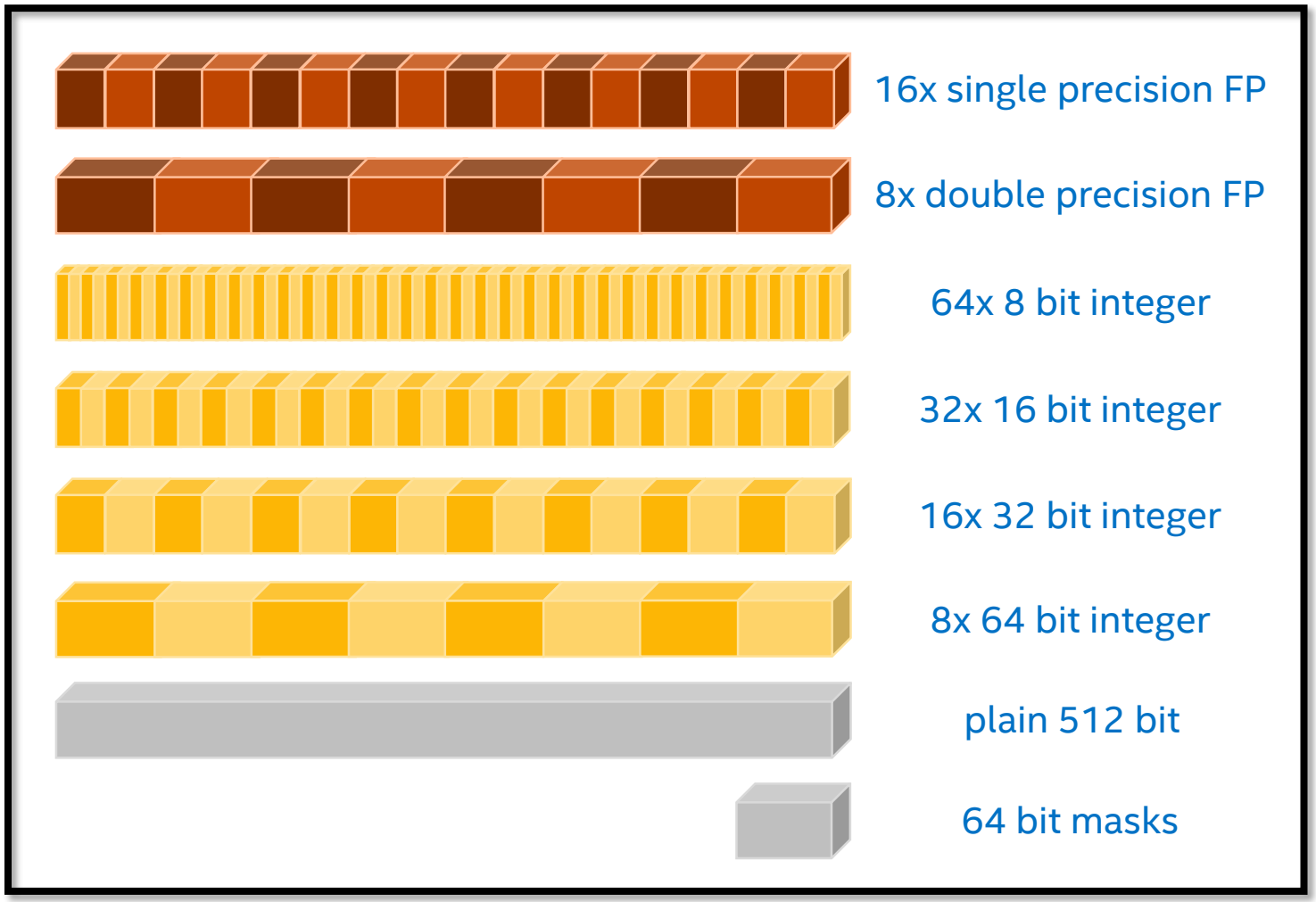
First Generation



- High level language *complex* types can also be used, compiler cares about details (halves the potential vector length)
- Use 32 bit integers where possible, avoid 64 bit integers (*short* & *char* types will be converted implicitly, though)
- Masking supported via dedicated registers (K0-7)
  ⇨ No need for bit vectors or additional compute cycles

Intel® AVX-512 Vector Types

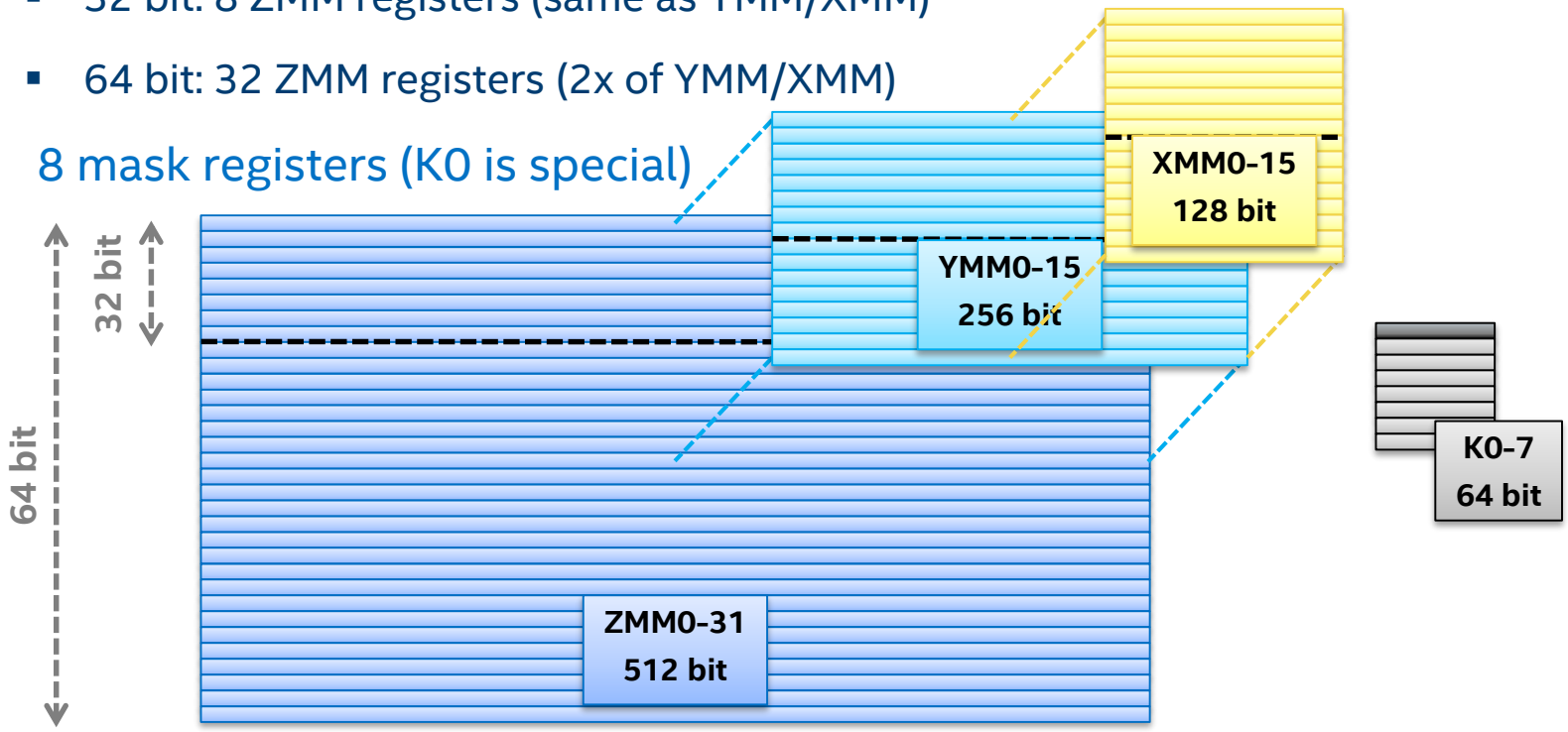
Intel® AVX-512



⇨ **Combines AVX and Intel® MIC Architecture!**

Intel® AVX-512 Registers

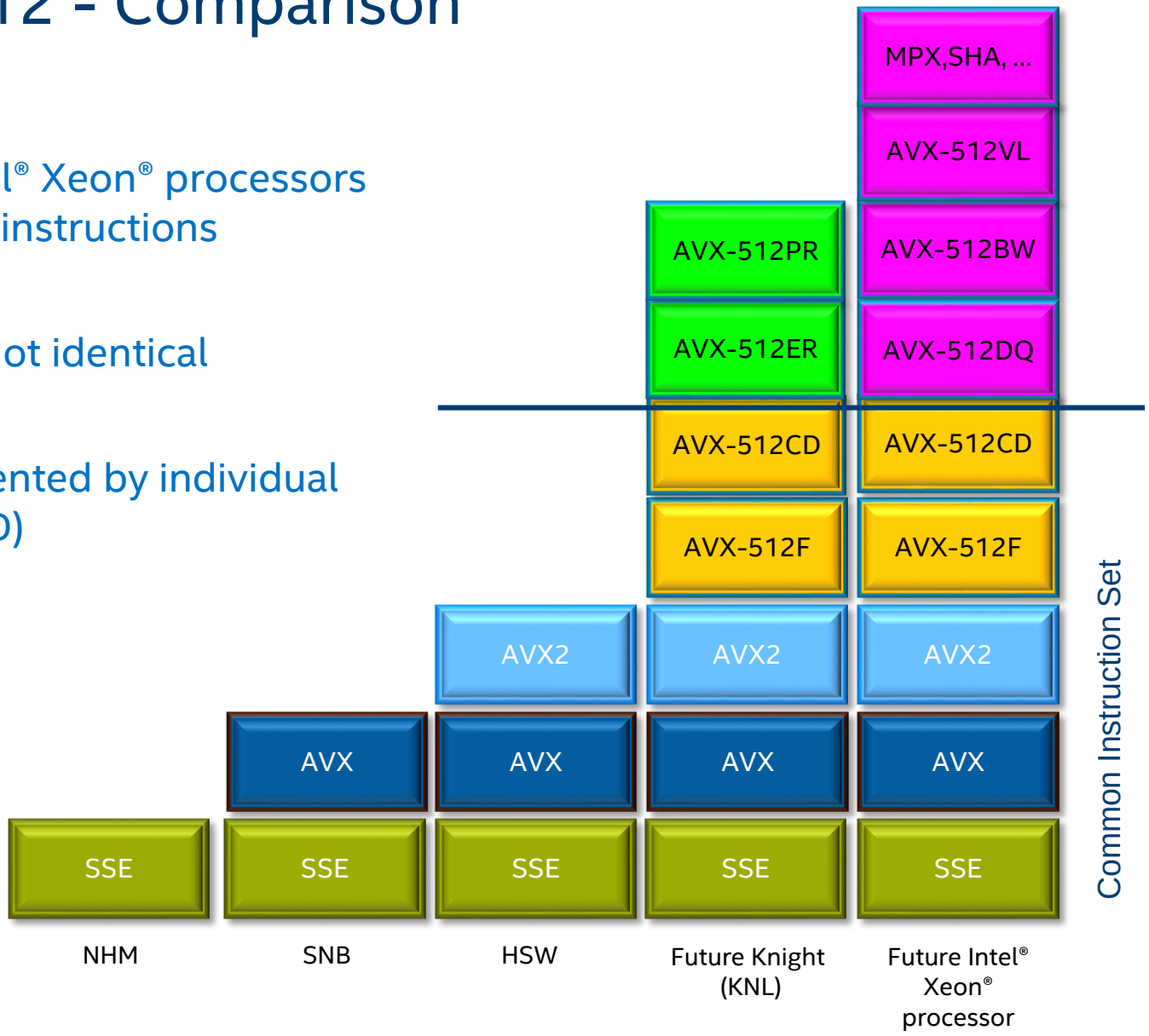
- Extended VEX encoding (EVEX) to introduce another prefix
- Extends previous AVX and SSE registers to 512 bit:
  - 32 bit: 8 ZMM registers (same as YMM/XMM)
  - 64 bit: 32 ZMM registers (2x of YMM/XMM)
- 8 mask registers (K0 is special)



⇨ **No penalty when switching between XMM, YMM and ZMM!**

Intel® AVX-512 - Comparison

- KNL and future Intel® Xeon® processors share a large set of instructions
- But some sets are not identical
- Subsets are represented by individual feature flags (CPUID)



Operating Systems & Intel® AVX-512

- OS support is required due to the new (extended) register state
- At least the following OSes are needed to get Intel® AVX:
  - Linux* kernel 3.15 or latest
  - Microsoft Windows* 8 and later
  - OS X*: unknown

Without OS support Intel® AVX-512 cannot be used even though the underlying processor supports it!

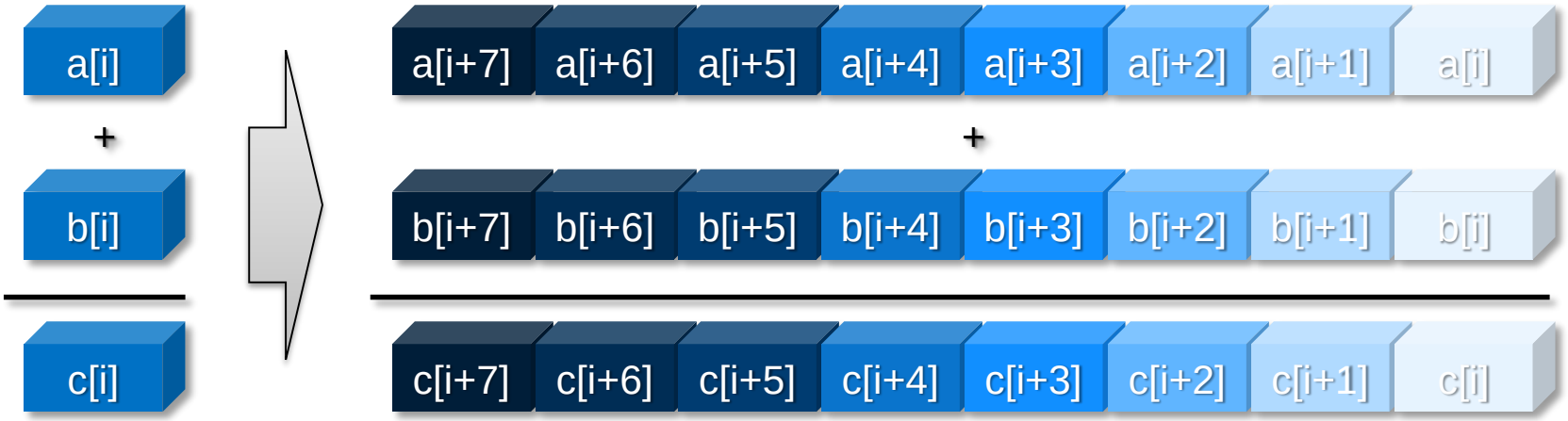
Agenda

- Introduction to SIMD for Intel® Architecture
- **Vector Code Generation**
- Compiler & Vectorization
- Validating Vectorization Success
- Reasons for Vectorization Fails
- Summary

Vectorization of Code

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

```
for(i = 0; i <= MAX;i++)
    c[i] = a[i] + b[i];
```



Use Vectorization

- How to express vectorization?
  - Fortran and C/C++ have limited ways to express it
  - But, **Intel compilers use heuristics** to vectorize
  - There are **extensions** that allow expression of vectorization explicitly
  - There are other, less portable ways…
- Select SIMD type:
  - A specific SSE/AVX version also includes all previous versions
  - Prefer AVX to SSE if available and possible; AVX also includes SSE
  - Avoid mixing SSE and AVX when using intrinsics or direct assembly
  - If target platform is not fixed/known Intel compiler can help producing multiple versions for different SIMD types:
    ⇨ **Runtime processor dispatching**

Many Ways to Vectorize

Compiler:
Auto-vectorization (no change of code)

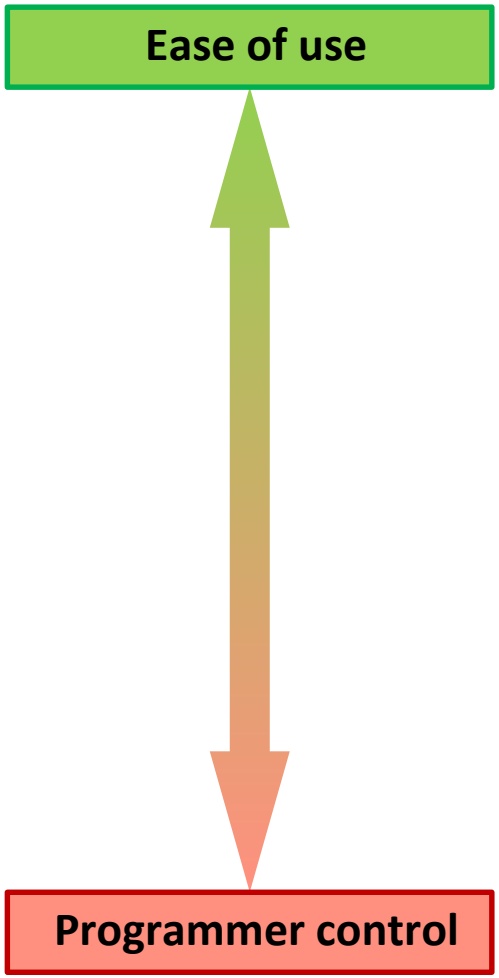
Compiler:
Auto-vectorization hints (`#pragma vector`, …)

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(…)`, …)

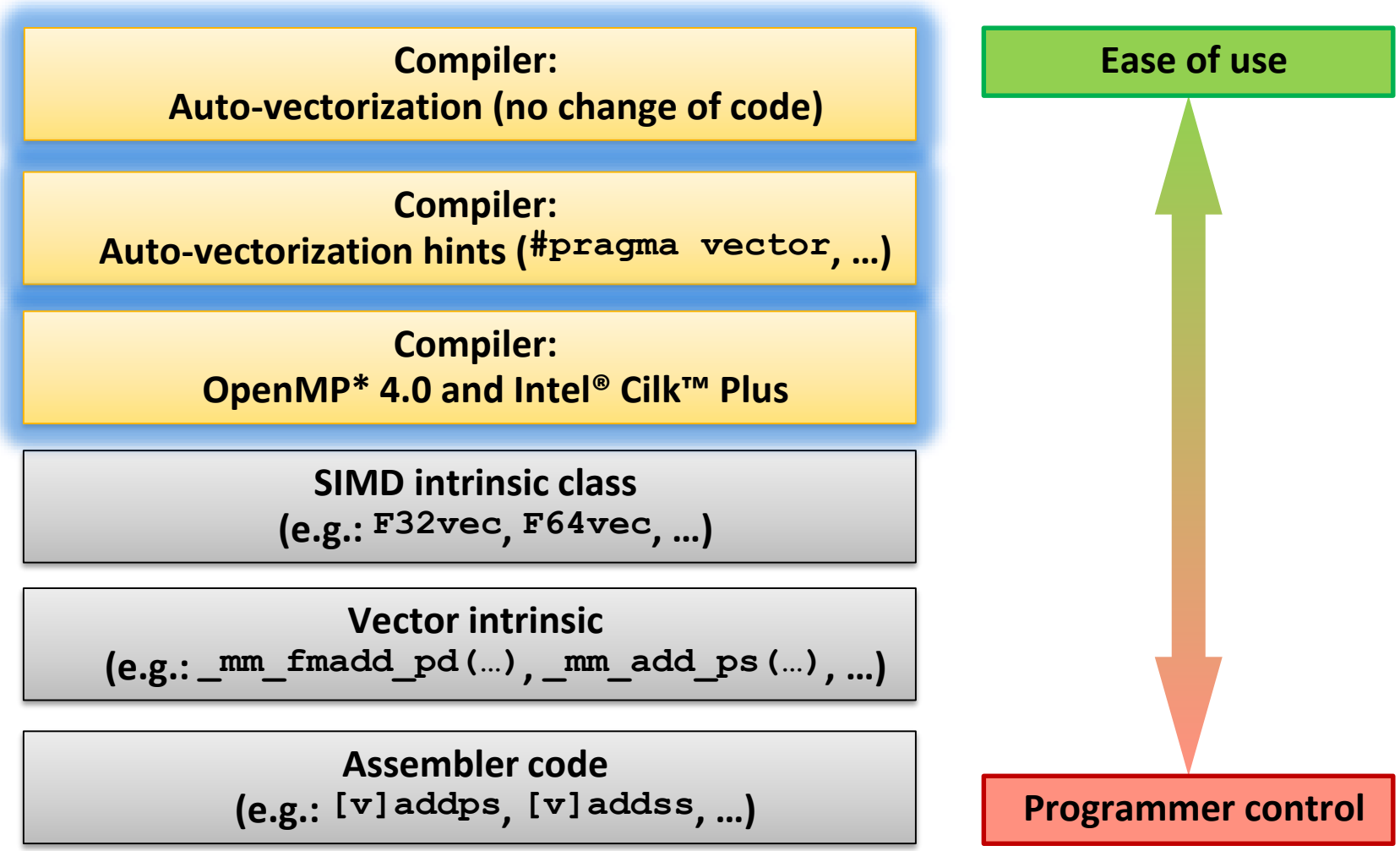
Assembler code
(e.g.: `[v]addps`, `[v]addss`, …)



Agenda

- Introduction to SIMD for Intel® Architecture
- Vector Code Generation
- **Compiler & Vectorization**
- Validating Vectorization Success
- Reasons for Vectorization Fails
- Summary

Many Ways to Vectorize



Auto-vectorization of Intel Compilers


```
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
  ...
```

Vectorization with Language Extensions

- Using vector intrinsic or SIMD intrinsic class inherently provides a guarantee of using SIMD instructions
  However, this is highly platform dependent and complex
- Auto-vectorization mostly works out of the box
  However, there are cases auto-vectorization does not work
- **Intel® Cilk™ Plus Array Notation Extensions** can alleviate both problems (C/C++ only):
  - Deterministically making use of vectorization
  - Easy to use with only minimal code changes

```
double A[1000], B[1000], C[1000], D[1000], E[1000];
for (int i = 0; i < 1000; i++)
  E[i] = (A[i] < B[i]) ? C[i] : D[i];
```

```
double A[1000], B[1000], C[1000], D[1000], E[1000];

E[:] = (A[:] < B[:]) ? C[:] : D[:];
```

- **OpenMP* 4.0** also provides extensions for vectorization (C/C++ & Fortran)

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*: **-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)

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-repot`, 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` or `#pragma omp simd`
    Fortran: `!DIR$ SIMD` or `!$OMP SIMD`

    When used, vectorization can only be turned off with:
    Linux*, OS X*: `-no-vec -no-simd -qno-openmp-simd`
    Windows*: `/Qvec- /Qsimd- /Qopenmp-simd-`

Agenda

- Introduction to SIMD for Intel® Architecture
- Vector Code Generation
- Compiler & Vectorization
- **Validating Vectorization Success**
- Reasons for Vectorization Fails
- Summary

Validating Vectorization Success I

- **Assembler code inspection (Linux*, OS X*: `-S`, Windows*: `/Fa`):**
  - Most reliable way and gives all details of course
  - Check for scalar/packed or (E)VEX encoded instructions:
    Assembler listing contains source line numbers for easier navigation
- **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 -no-simd -qno-openmp-simd` or `/ Qvec- /Qsimd- /Qopenmp-simd-`, or on a loop by loop basis via `#pragma novector` or `!DIR$ NOVECTOR`
  2. Compile and benchmark with selected SIMD feature
  3. Compare runtime differences

Validating Vectorization Success II

- **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
…
```

Optimization Report – Advanced I

- **See which levels are available for each phase:**
  - Linux*, OS X*: **-qopt-report-help**,
    Windows*: **/Qopt-report-help**

```
$ icpc -qopt-report-help
…
          vec: Vector optimizations
            Level 1: Report the loops that were vectorized.
            Level 2: Level 1 + report the loops that were not vectorized,
                     along with reason preventing vectorization.
            Level 3: Level 2 + loop vectorization summary.
            Level 4: Level 3 + report verbose details for reasons loop
                     was/wasn't vectorized.
            Level 5: Level 4 + report information about variable/memory
                     dependencies preventing vectorization.
…
```

- **Select format:**
  - Linux*, OS X*: **-qopt-report-format=[text|vs]**,
    Windows*: **/Qopt-report-format:[text|vs]**
  - **text** as textual and **vs** for Microsoft Visual Studio* IDE integration output

Agenda

- Introduction to SIMD for Intel® Architecture
- Vector Code Generation
- Compiler & Vectorization
- Validating Vectorization Success
- **Reasons for Vectorization Fails**
- Summary

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 depencence
- Bit masking

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

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 and are not described in the following!

Factors that **prevent Vectorizing** your code

1. Loop-carried dependencies

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

1.A Pointer aliasing (compiler-specific)

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

2. Function calls (incl. indirect)

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

3. Loop structure, boundary condition

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

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

4 Outer vs. inner loops

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

5. Cost-benefit (compiler specific..)

And others……

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

1.A. Indirect memory access

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

1.B Memory sub-system Latency / Throughput

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

2. Serialized or “sub-optimal” function calls

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

3. Small trip counts not multiple of VL

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

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

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

5. MANY others: spill/fill, fp accuracy trade-offs, FMA, DIV/SQRT, Unrolling, even AVX throttling..

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**

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

- $S_1 \ \delta^A \ S_2$: S_1 reads, S_2 writes: **Anti Dependence**

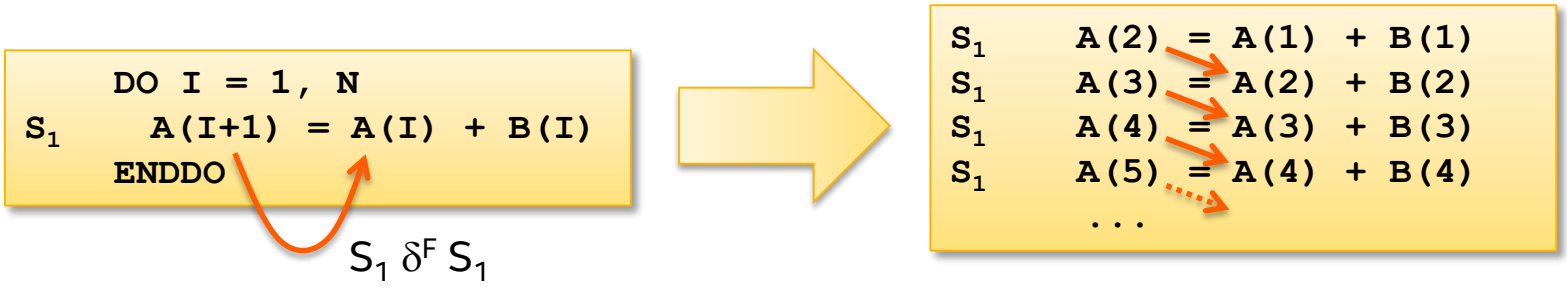
```
S1   ... = X
S2   X = ...
```

- $S_1 \ \delta^O \ S_2$: S_1 writes, S_2 writes: **Output Dependence**

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

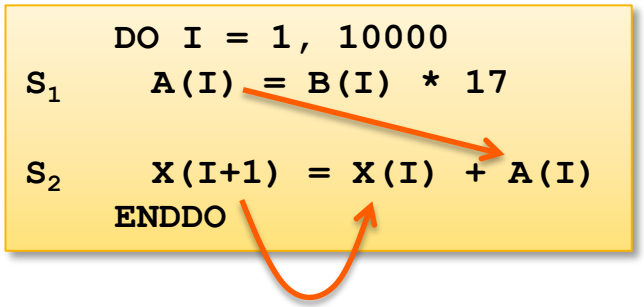
Data Dependence in Loops

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.:



$S_1\ \delta^F\ S_2$: Loop-independent dependence

$S_2\ \delta^F\ S_2$: Loop-carried dependence

Disambiguation Hints I

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

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

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

Examples:

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

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

Disambiguation Hints II

Directives:

- `#pragma ivdep` (C/C++) or `!DIR$ IVDEP` (Fortran)
- `#pragma simd` (C/C++) or `!DIR$ SIMD` (Fortran)

For C/C++:

- Assume no aliasing at all (dangerous!):
  Linux*, OS X*: `-fno-alias`, Windows*: `/Oa`
- Assume ISO C Standard aliasing rules:
  Linux*, OS X*: `-ansi-alias`, Windows*: `/Qansi-alias`
  Default with 15.0 and later but not with earlier versions!
- Turns on ANSI aliasing checker, too (thus recommended)
- No aliasing between function arguments:
  Linux*, OS X*: `-fargument-noalias`, Windows*: `/Qalias-args-`
- No aliasing between function arguments and global storage:
  Linux*, OS X*: `-fargument-noalias-global`, Windows*: N/A

Disambiguation Hints III

For Fortran:

- Assume no aliasing at all:
  Linux*, OS X*: **`-fno-alias`**, Windows*: **`/Oa`**
- Assume Fortran Standard aliasing rules:
  Linux*, OS X*: **`-ansi-alias`**, Windows*: **`/Qansi-alias`**
  Opposed to C/C++ this is default since ever!
- No aliasing of Cray* pointers:
  Linux*, OS X*: **`-safe-cray-ptr`**, Windows*: **`/Qsafe-cray-ptr`**

Inter-Procedural Dependency Analysis

- Optimization usually takes place individually for each procedure
- Dependency analysis of inter-procedural optimization (IPO) works across all procedures and thus allows **global optimization**
- Switch to turn on IPO for single file (one compilation unit)
  - Linux*, OS X*: **`-ip`**
  - Windows*: **`/Qip`**
    Subset already default for optimization levels 2 and higher
- Switch to turn on IPO for all compilation units
  - Linux*, OS X*: **`-ipo`**
  - Windows*: **`/Qipo`**
- Example:
  References of function arguments can be analyzed even if located in other compilation unit.

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 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 if aligned:

```
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

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!

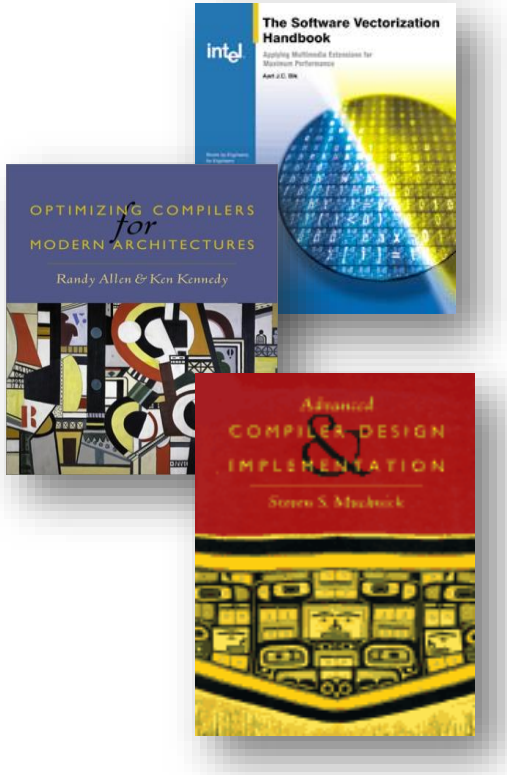
Agenda

- Introduction to SIMD for Intel® Architecture
- Vector Code Generation
- Compiler & Vectorization
- Validating Vectorization Success
- Reasons for Vectorization Fails
- **Summary**

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

References



- Aart Bik: “The Software Vectorization Handbook” http://www.intel.com/intelpress/sum_vmmx.htm
- Randy Allen, Ken Kennedy: “Optimizing Compilers for Modern Architectures: A Dependence-based Approach”
- Steven S. Muchnik, “Advanced Compiler Design and Implementation”
- Intel Software Forums, Knowledge Base, White Papers, Tools Support (see http://software.intel.com)
  Sample Articles:
  - http://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers/
  - http://software.intel.com/en-us/articles/requirements-for-vectorizable-loops/
  - http://software.intel.com/en-us/articles/performance-tools-for-software-developers-intel-compiler-options-for-sse-generation-and-processor-specific-optimizations/

THANK YOU!

Legal Disclaimer & Optimization Notice

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

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®