



VECTORIZATION FOR INTEL® COMPILER

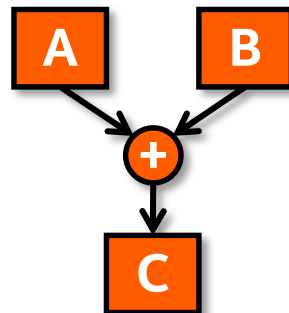
Agenda

- **Introduction to SIMD for Intel® Architecture**
- Vector Code Generation
- Compiler & Vectorization
- Validating Vectorization Success
- Reasons for Vectorization Fails
- Intel® Cilk™ Plus
- OpenMP* 4.0
- Summary

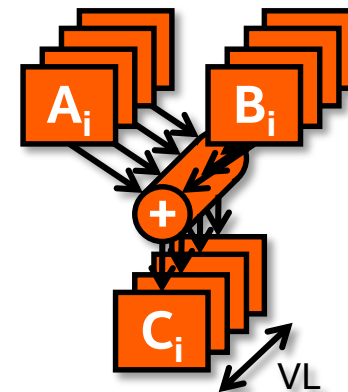
Vectorization

- **Single Instruction Multiple Data (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**



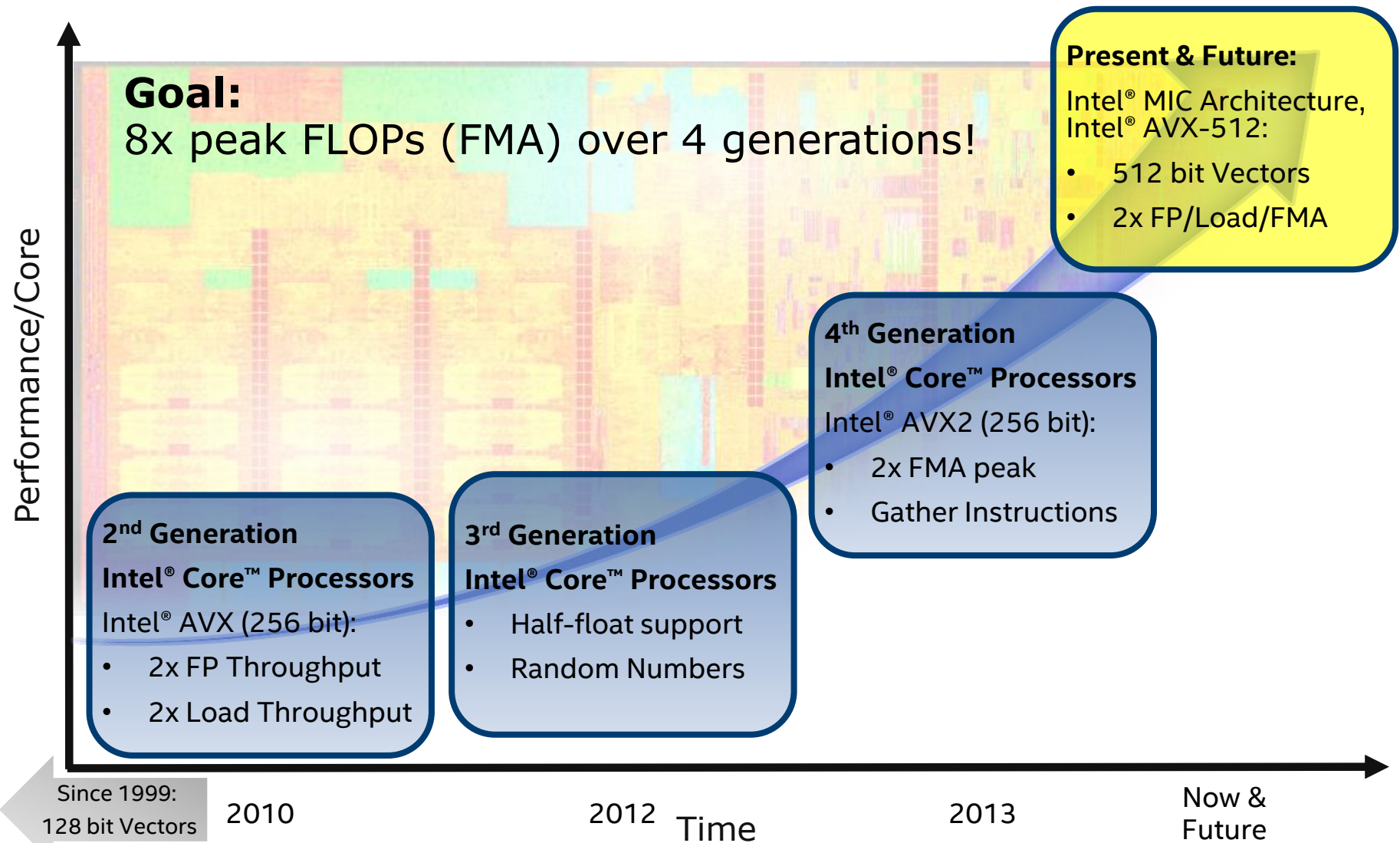
**Vector
Processing**



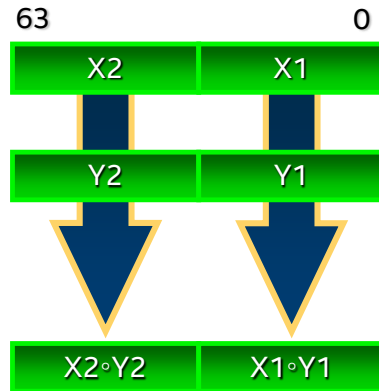
SIMD & Intel® Architecture

- 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)

Evolution of SIMD for Intel Processors



SIMD Types for Intel® Architecture I



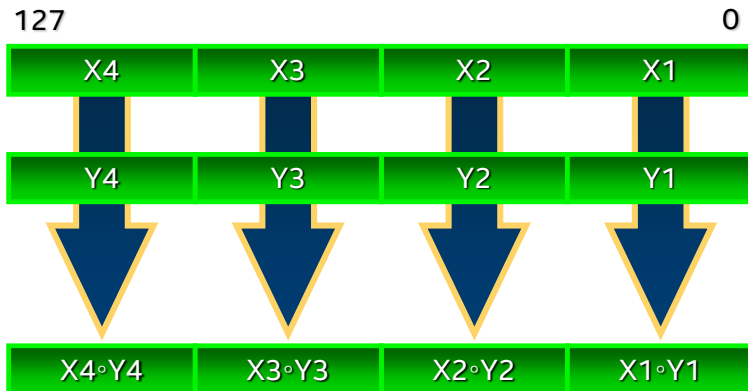
MMX™

Vector size: **64 bit**

Data types:

- 8, 16 and 32 bit integer

VL: 2, 4, 8



SSE

Vector size: **128 bit**

Data types:

- 8, 16, 32, 64 bit integer
- 32 and 64 bit float

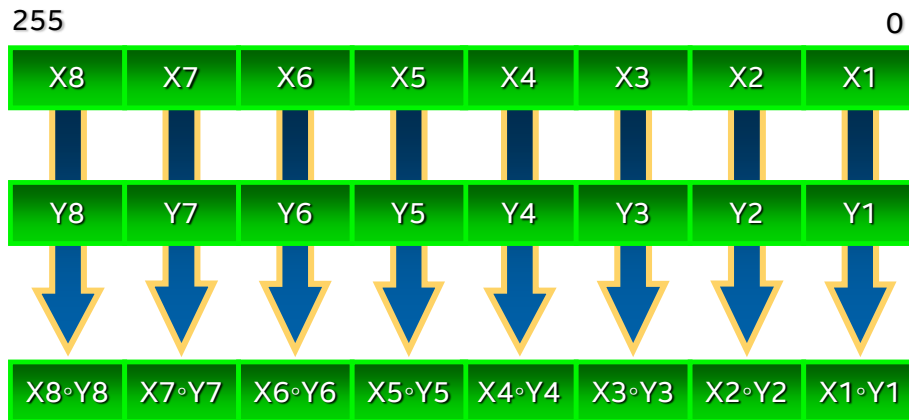
VL: 2, 4, 8, 16

Illustrations: Xi, Yi & results 32 bit integer

Optimization Notice

Copyright © 2016, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

SIMD Types for Intel® Architecture II



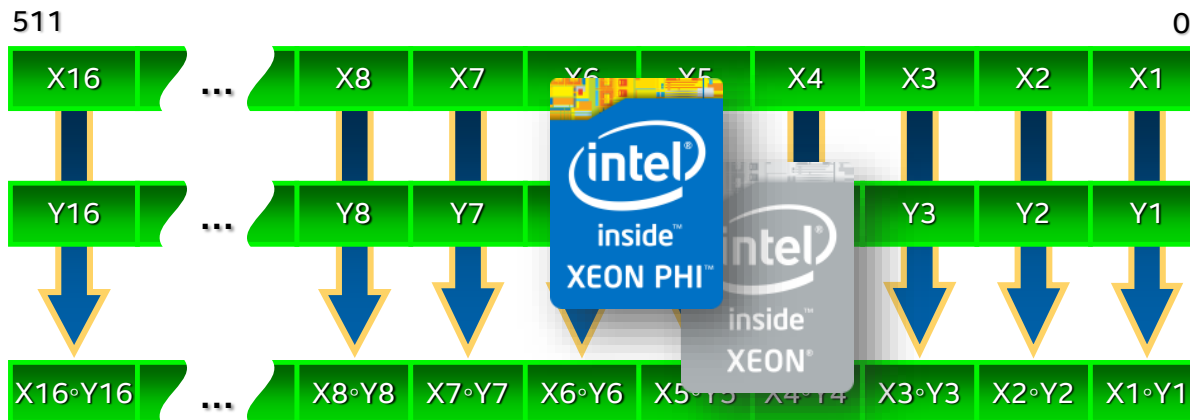
AVX

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 & Intel® MIC Architecture

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

Optimization Notice

Copyright © 2016, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.



AVX Features

KEY FEATURES

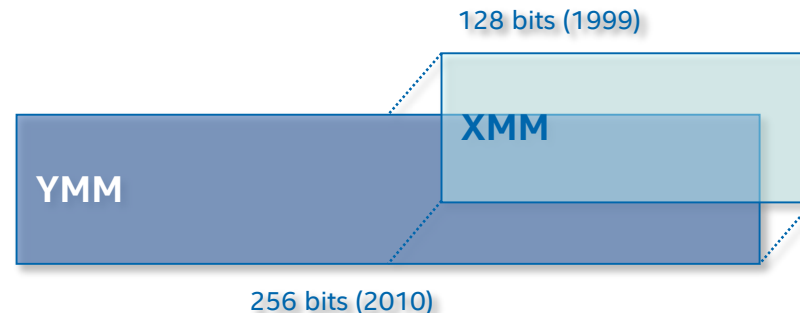
- Wider Vectors
 - Increased from 128 to 256 bit
 - Two 128-bit load/store ports
- Enhanced data rearrangement:
Use the new 256 bit primitives to broadcast, mask loads and permute data
- Three and four Operands:
Non-destructive syntax for both VEX.128 and VEX.256
- Flexible unaligned memory access support
- Extensible new opcode encoding (VEX)

BENEFITS

- Up to 2x peak FLOPs (floating point operations per second) output with good power efficiency
- Organize, access and pull only necessary data more quickly and efficiently
- Fewer register copies, better register use for both vector and scalar code
- More opportunities to fused load and compute operations
- Code size reduction

AVX is SSE Extension

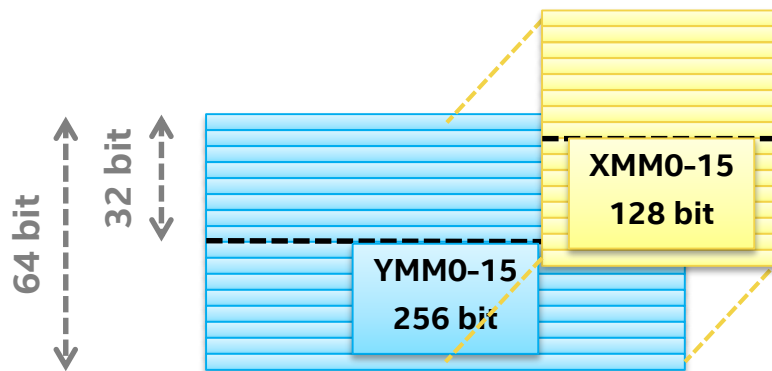
- A 256 bit vector extension to SSE:
 - SSE uses dedicated 128 bit registers called XMM (8 for IA-32 & 16 for Intel® 64)
 - Extends all XMM registers to 256 bit called YMM
 - Lower 128 bit of YMM register are mapped/shared with XMM
- AVX works on either
 - The whole 256 bit
 - The lower 128 bit; zeros the higher 128 bit
 - AVX counterparts for almost all existing SSE instructions:
For initial generation (Intel® AVX) full 256 bit vectors for FP only; integers will follow!



AVX Registers

Amount of registers depends on architecture:


- 32 bit: 8 XMM/YMM registers
- 64 bit: 16 XMM/YMM registers
- Lower half of YMM register is shared with SSE registers (XMM)
⇒ **Penalty with context switch when mixing SSE & AVX instructions!**
- No penalty with **vzeroupper** instruction when switching from AVX to SSE



Intel® AVX2 I

- Basically same as Intel® AVX with following additions:
 - Doubles** width of **integer vector** instructions to 256 bits
 - Floating point fused multiply add (**FMA**)

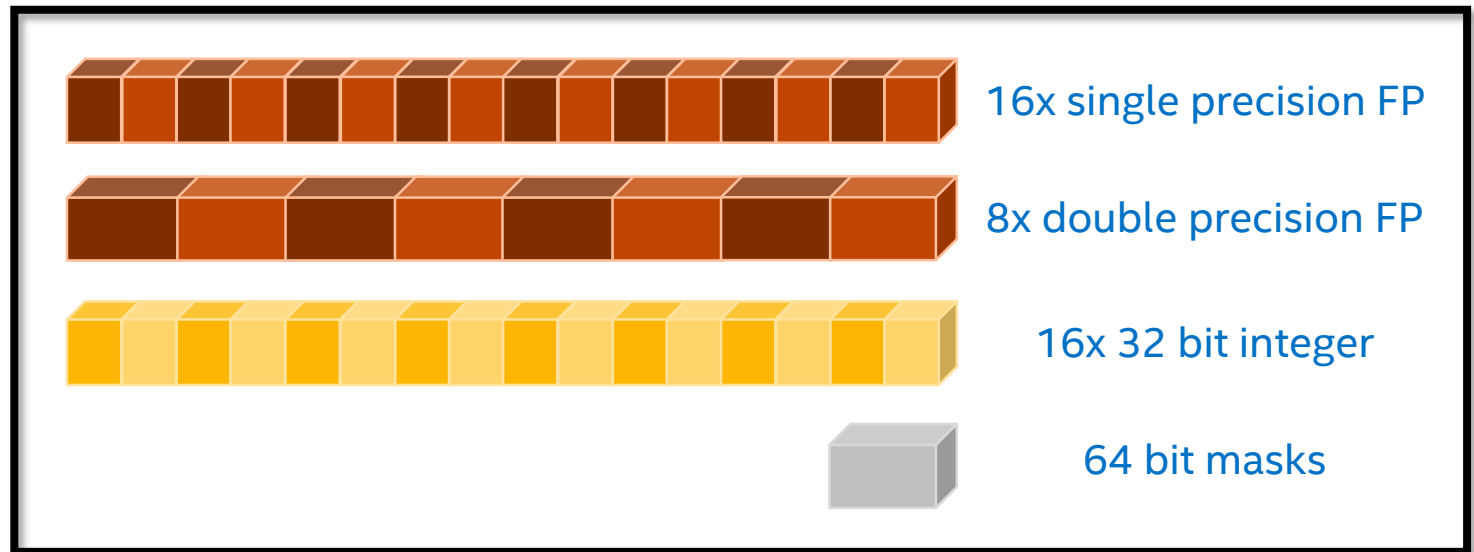
Processor Family	Instruction Set	Single Precision FLOPs Per Clock	Double Precision FLOPs Per Clock
Pre 2 nd generation Intel® Core™ Processors	SSE 4.2	8	4
2 nd and 3 rd generation Intel® Core™ Processors	AVX	16	8
4th generation Intel® Core™ Processors	AVX2	32	16



- Bit Manipulation Instructions (BMI)
- Gather instructions (scatter for the future)
- Any-to-any permutes
- Vector-vector shifts

Intel® MIC Architecture Vector Types

First
Generation



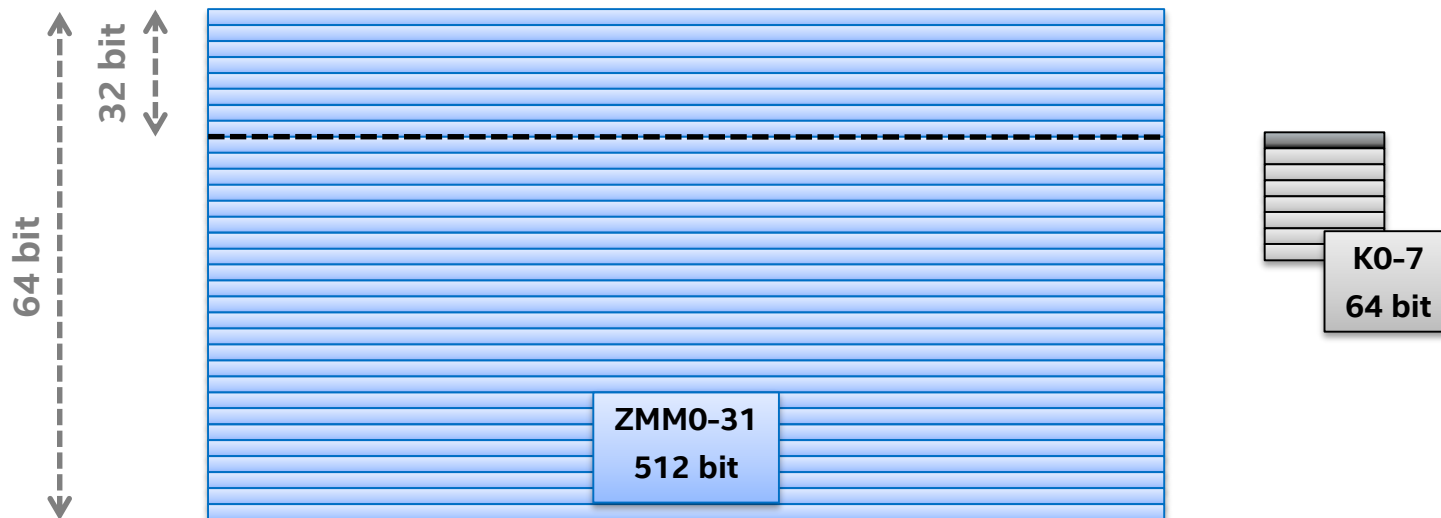
- High level language *complex* types can also be used, compiler cares about details (halves the potential vector length)
- Use 32 bit integers where possible, avoid 64 bit integers (*short* & *char* types will be converted implicitly, though)
- Masking supported via dedicated registers (K0-7)
 - ⇒ No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

Intel® MIC Architecture Features

- First generation Intel® Xeon Phi™ coprocessor only
 - Vector extensions not backward compatible to SSE/AVX
 - Introduced 512 bit vectors:
 - Single & double precision FP
 - Little bit of integer (32 bit only)
 - Introduced mask registers:
 - Operation + masking in one instruction
 - Does not require additional instruction to mask out elements in a vector
- ⇒ **Will be merged into Intel® AVX-512 which is backward compatible**

Intel® MIC Architecture Registers

- Extends previous AVX and SSE registers to 512 bit
 - ~~32 bit: 8 ZMM registers~~ (no real need for 32 bit)
 - 64 bit: 32 ZMM registers
- 8 mask registers (K0 is special)
- **Not compatible to AVX or SSE – no YMM/XMM registers!**

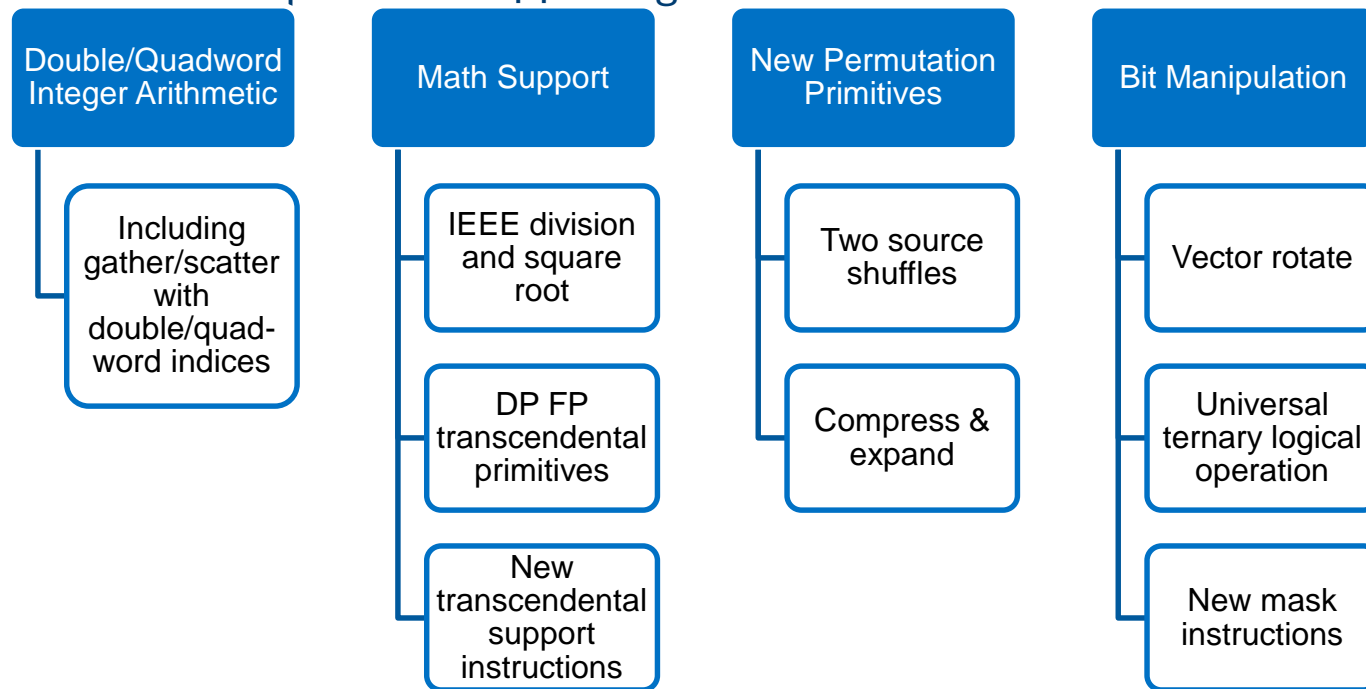


Intel® AVX-512 Features I

Different versions of Intel® AVX-512:

- Intel® AVX-512 **Foundation**:

- Extension of AVX known instruction sets including mask registers
- Available in all products supporting Intel® AVX-512



Intel® AVX-512 Features II

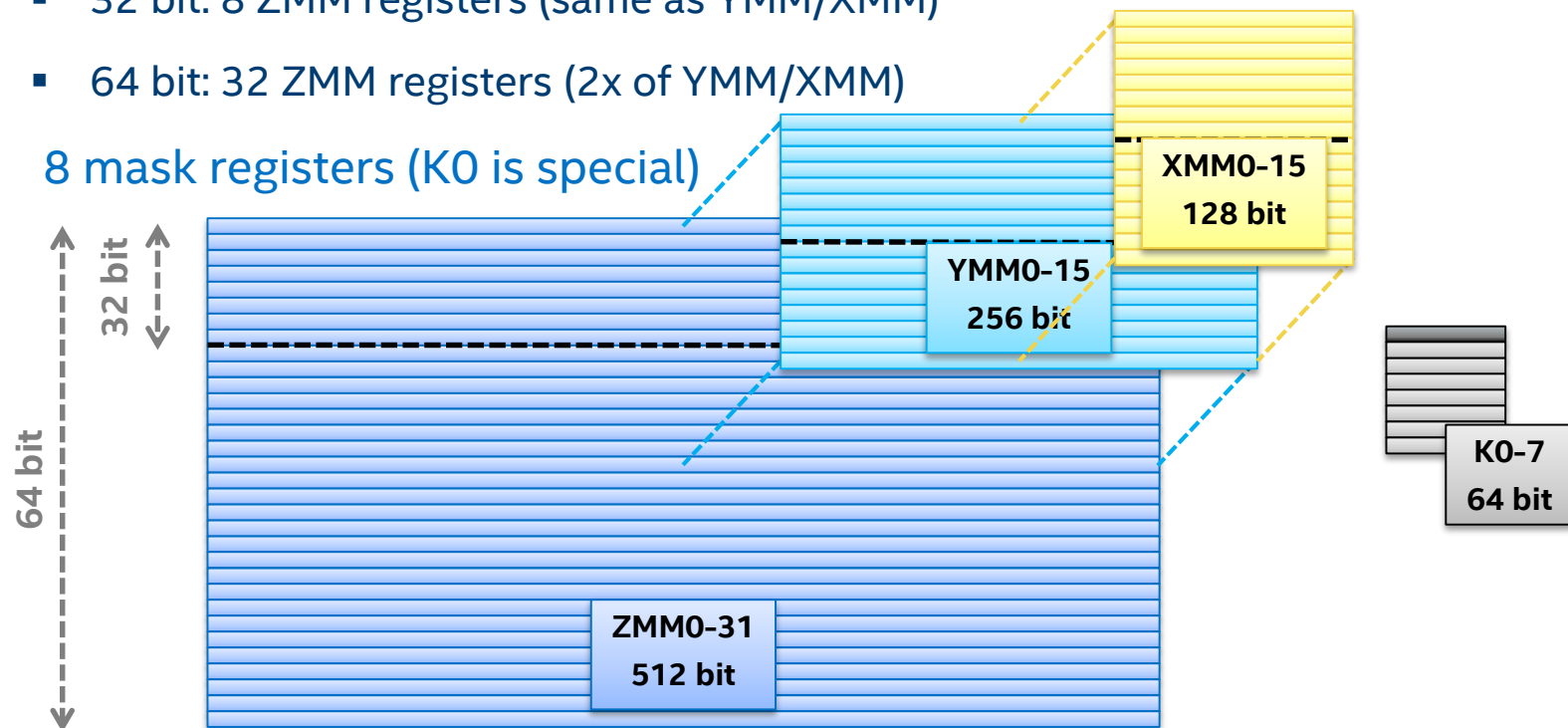
- **Intel® AVX-512 Vector Length Extension:**
 - Freely select the vector length (512 bit, 256 bit and 128 bit)
 - Orthogonal extension but planned for future Intel® Xeon® processors only
- **Intel® AVX-512 Byte/Word and Doubleword/Quadword:**
 - Two groups, planned for future Intel® Xeon® processors:
 - 8 and 16 bit integers
 - 32 and 64 bit integers & FP
- **Intel® AVX-512 Conflict Detection:**
 - Check identical values inside a vector (for 32 or 64 bit integers)
 - Used for finding colliding indexes (32 or 64 bit) before a gather-operation-scatter sequence
 - Likely to be available in future for both Intel® Xeon Phi™ coprocessors and Intel® Xeon® processors

Intel® AVX-512 Features III

- Intel® AVX-512 **Exponential & Reciprocal Instructions:**
 - Higher accuracy (28 bit) with HW based sqrt, reciprocal and exp function
 - Likely only for future Intel® Xeon Phi™ coprocessors
- Intel® AVX-512 **Prefetch Instructions:**
 - Manage data streams for higher throughput (incl. gather & scatter)
 - Likely only for future Intel® Xeon Phi™ coprocessors
- More here:
<https://software.intel.com/en-us/blogs/additional-avx-512-instructions>

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 Features – More Detail I

AVX-512 F

512-bit Foundation instructions

- ❑ Available in all products supporting Intel® AVX-512
- ❑ Comprehensive vector extension for HPC and enterprise
- ❑ All the key Intel® AVX-512 features: masking, broadcast, ...
- ❑ 32-bit and 64-bit integer and floating-point instructions
- ❑ Promotion of many Intel® AVX and Intel® AVX2 instructions to Intel® AVX-512
- ❑ Many new instructions added to accelerate HPC workloads

AVX-512CD

Conflict Detection Instructions

- ❑ Allow vectorization of loops with possible address conflict
- ❑ Will show up on Intel® Xeon® processors

AVX-512ER

Exponential, Reciprocal and Prefetch Operations

AVX-512PR

- ❑ Fast (28 bit) instructions for exponential and reciprocal and transcendentals (as well as RSQRT)
- ❑ New prefetch instructions: gather/scatter prefetches and PREFETCHWT1

Intel® AVX-512 Features – More Detail II

AVX-512DQ

Double and Quad Word Instructions

- ☐ All of (packed) 32bit/64 bit operations AVX-512F doesn't provide
- ☐ Close 64bit gaps like VPMULLQ : packed 64x64 → 64
- ☐ Extend mask architecture to word and byte (to handle vectors)
- ☐ Packed/Scalar converts of signed/unsigned to SP/DP

AVX-512BW

Byte and Word Instructions

- ☐ Extend packed (vector) instructions to byte and word (16 and 8 bit) data type
- ☐ MMX™/Intel® SSE2/Intel® AVX2 re-promoted to Intel® AVX512 semantics
- ☐ Mask operations extended to 32/64 bits to adapt to number of objects in 512bit
- ☐ Permute architecture extended to words (VPERMW, VPERMI2W, ...)

AVX-512VL

Vector Length Extensions

- ☐ Vector length orthogonality
- ☐ Support for 128 and 256 bits instead of full 512 bit
- ☐ Not a new instruction set but an attribute of existing 512bit instructions

Other New Instructions Imminent

MPX

Intel® Memory Protection Extension

- ❑ Set of instructions to implement checking a pointer against its bounds
- ❑ Pointer Checker support in HW (today a SW only solution of e.g. Intel Compilers)
- ❑ Debug and security features

SHA

Intel® Secure Hash Algorithm

- ❑ Fast implementation of cryptographic hashing algorithm as defined by NIST FIPS PUB 180

CLFLUSHOPT

Single Instruction – Flush a cache line

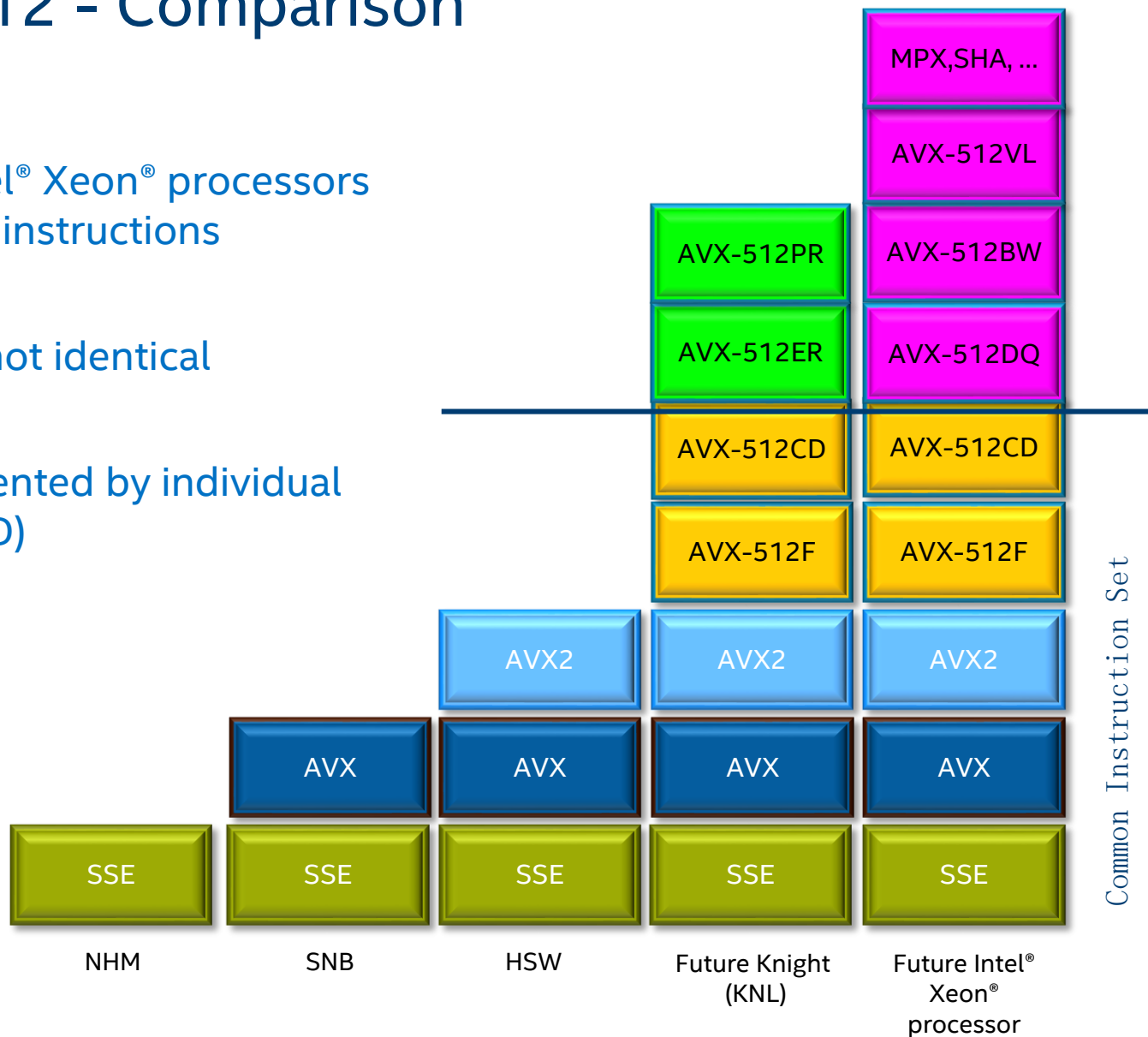
- ❑ needed for future memory technologies

XSAVE{S,C}

Save and Restore Extended Processor State

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)

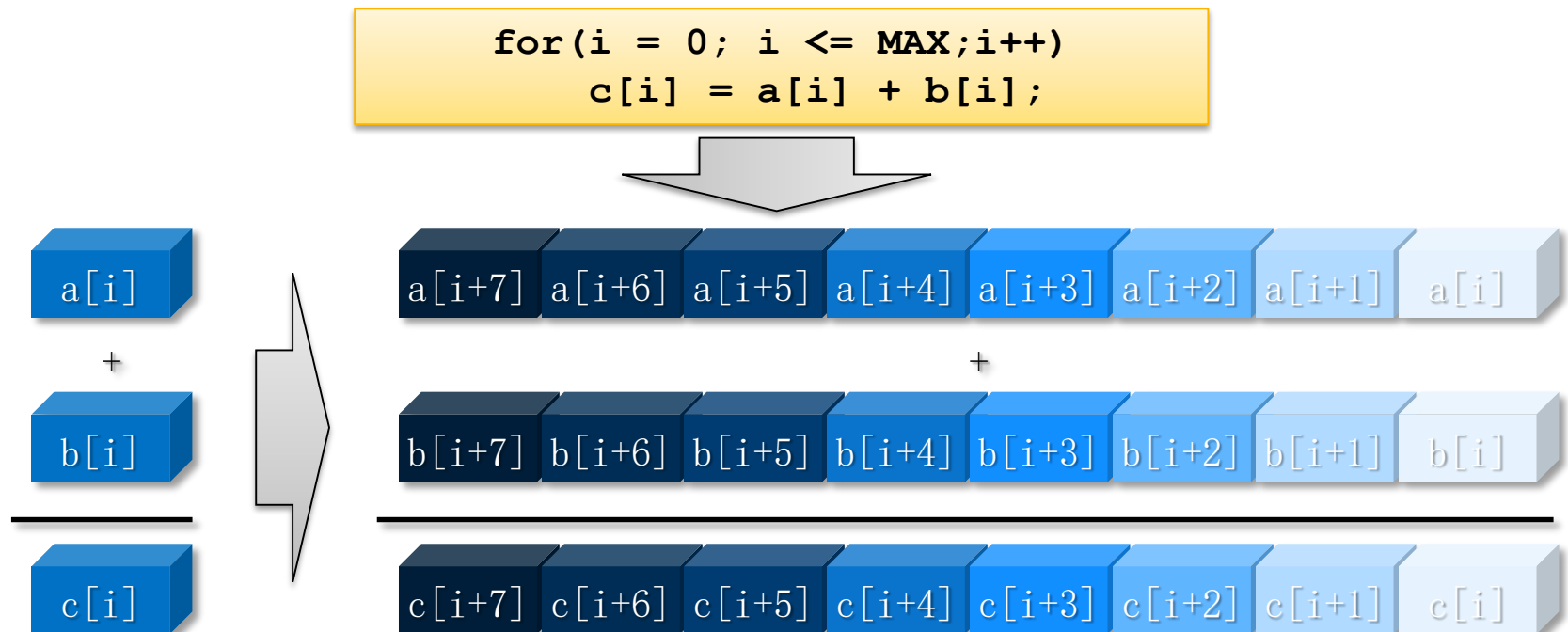


Agenda

- Introduction to SIMD for Intel® Architecture
- **Vector Code Generation**
- Compiler & Vectorization
- Validating Vectorization Success
- Reasons for Vectorization Fails
- Intel® Cilk™ Plus
- OpenMP* 4.0
- Summary

Vectorization of Code

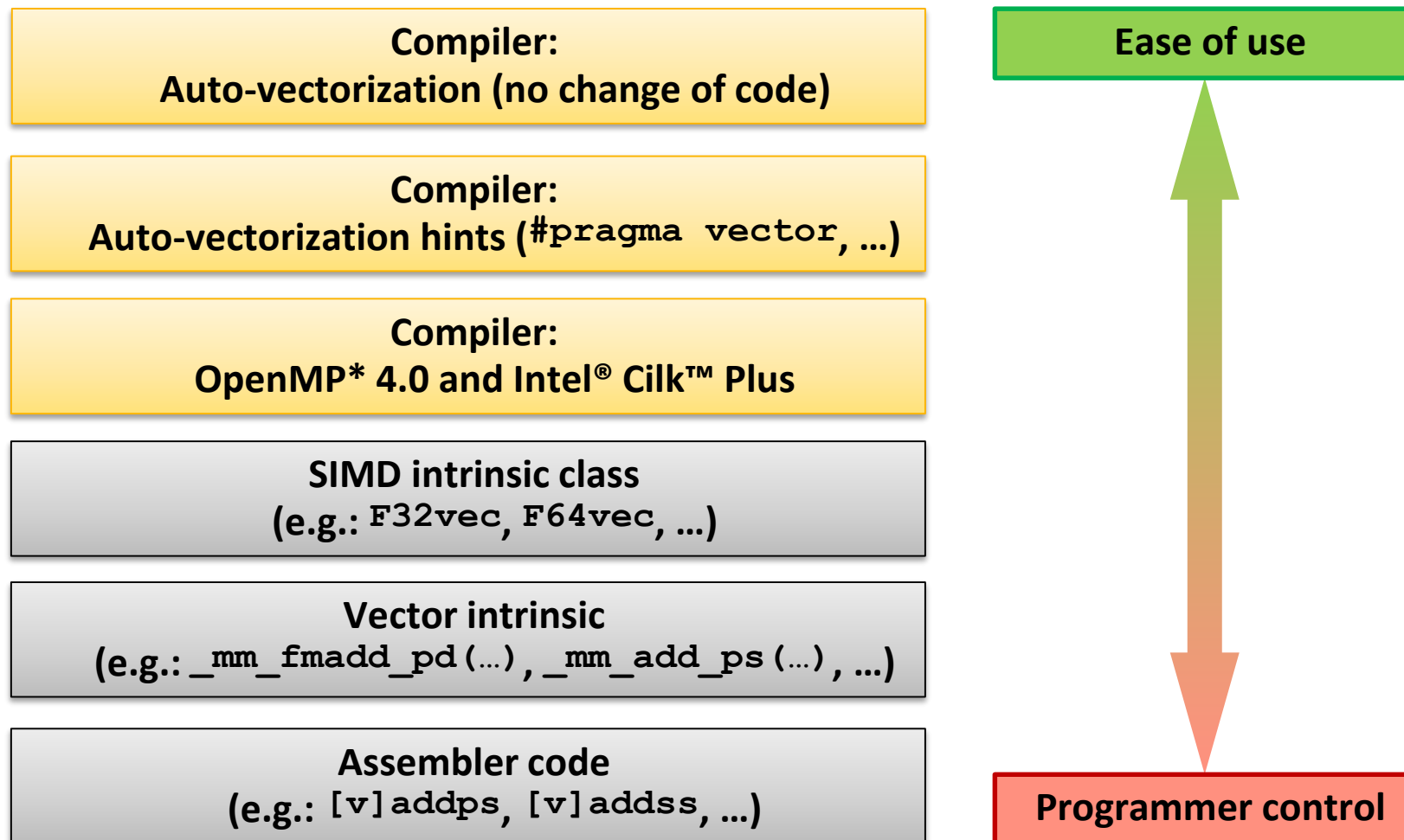
- Transform sequential code to exploit vector processing capabilities (SIMD) of Intel processors
 - Manually by explicit syntax
 - Automatically by tools like a compiler



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



Intrinsics III

- SSE or AVX VEX.128 intrinsics have `_mm_` prefix, e.g.:
 - `_mm_exp2_ps(__m128)`
 - `_mm_add_pd(__m128d, __m128d)`
Depending on the selected SIMD feature (e.g. `-xSSE4.2`, `-xAVX`, ...) the compiler can create SSE or AVX VEX.128 encoded instructions!
- AVX VEX.256 intrinsics have `_mm256_` prefix, e.g.:
 - `_mm256_exp2_ps(__m256)`
 - `_mm256_add_pd(__m256d, __m256d)`
- Intel® MIC Architecture/Intel® AVX-512 intrinsics with `_mm512_` prefix, e.g.:
 - `_mm512_exp2_ps(__m512)`
 - `_mm512_add_pd(__m512d, __m512d)`
- Use of SSE and AVX intrinsics can be mixed but be aware of penalty in changing architectural state (except for Intel® AVX-512)!

Intrinsics IV

- 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);
}
```

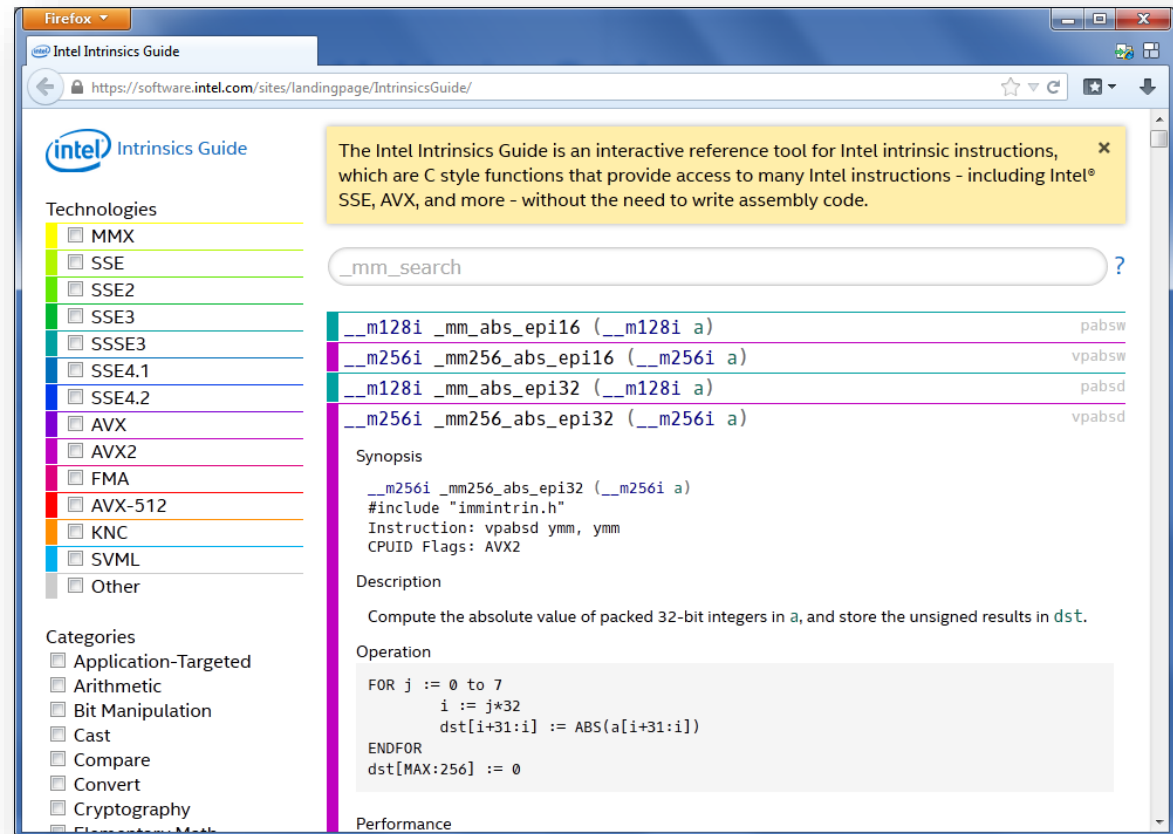
Intel® Intrinsic Guide

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/IntrinsicsGuide/>



Agenda

- Introduction to SIMD for Intel® Architecture
- Vector Code Generation
- **Compiler & Vectorization**
- Validating Vectorization Success
- Reasons for Vectorization Fails
- Intel® Cilk™ Plus
- OpenMP* 4.0
- Summary

Many Ways to Vectorize

Compiler:
Auto-vectorization (no change of code)

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

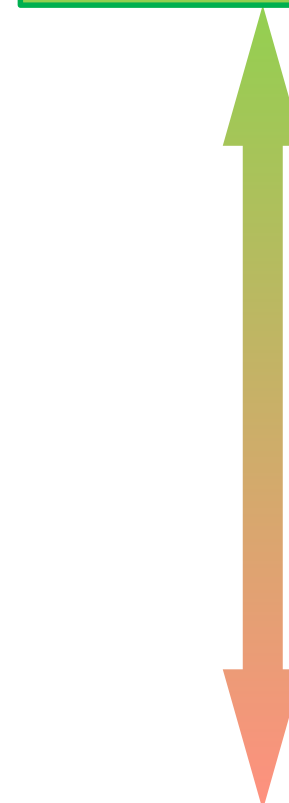
Compiler:
OpenMP* 4.0 and Intel® Cilk™ Plus

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

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
...
```

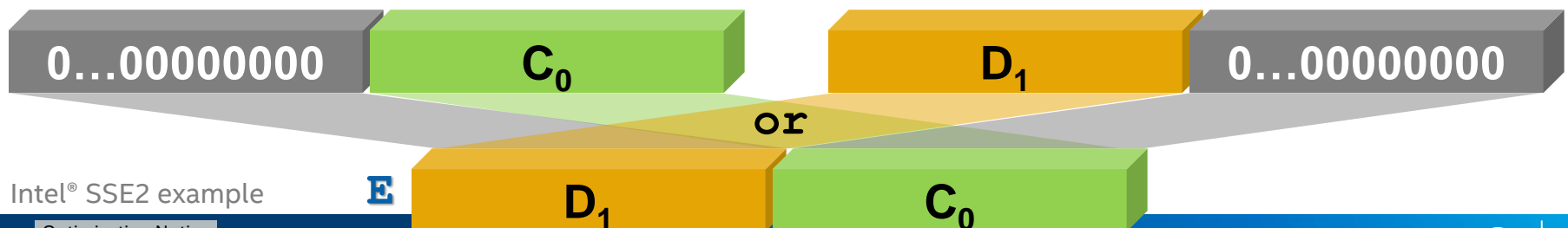
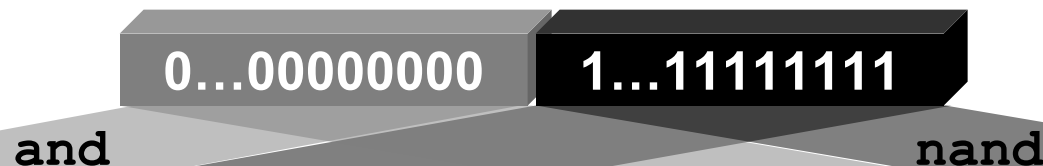
Intel® SSE4.2

..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
...
```


Advanced Auto-vectorization Example

```
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];
```



Intel® SSE2 example

Optimization Notice

Copyright © 2016, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.



Vectorization using Intrinsics

- Same Example as before but implemented with vector intrinsics:

```
double A[1000], B[1000], C[1000], D[1000], E[1000];
for (int i = 0; i < 1000; i += 2) {
    __m128d a = _mm_load_pd(&A[i]);
    __m128d b = _mm_load_pd(&B[i]);
    __m128d c = _mm_load_pd(&C[i]);
    __m128d d = _mm_load_pd(&D[i]);
    __m128d e;
    __m128d mask = _mm_cmplt_pd(a, b);
    e = _mm_or_pd(
        _mm_and_pd (mask, c),
        _mm_andnot_pd(mask, d));
    _mm_store_pd(&E[i], e);
}
```

- Vector intrinsic (or SIMD intrinsic class) can be easier than assembler:
 - Similar performance, very close to best manually written assembler code
 - Hides many details like register allocation and scheduling
 - Intrinsics more portable and supported by all popular compilers!
- **Using intrinsics is not as portable compared to auto-vectorization!**

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

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** 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
- Intel® Cilk™ Plus
- OpenMP* 4.0
- Summary

Validating Vectorization Success

- **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

Optimization Report – Advanced II

- **For use with Intel® VTune™ Advisor XE and Intel® Amplifier XE:**
 - Embed optimization reports in the object/executable to visualize it with Intel® VTune™ Advisor XE or Intel® Amplifier XE directly.
 - Linux*, OS X*: **-qopt-report-help**,
Windows*: **/Qopt-report-help**
 - Default already if optimization report are enabled and **-g** or **/zi** are set!
- **Select sections in the code:**
 - Limit optimization reports to code section(s)
 - Linux*, OS X*: **-qopt-report-filter=<string>**,
Windows*: **/Qopt-report-filter:<string>**

<string>	Description
filename	Source file
filename, routine	Source file + routine name
filename, range [, range] ...	Source file + lines
filename, routine, range [, range] ...	Source file + routine name + lines

Optimization Report – Advanced III

- **Filter for specific routines:**
 - A substring can be specified to filter for routines
 - Linux*, OS X*: **-qopt-report-routine=<substring>**,
Windows*: **/Qopt-report-routine:<substring>**
- **Enable/disable C++ name mangling:**
 - C++ symbol names are mangled (encode function name, namespace(s), parameter types, ...)
 - Linux*, OS X*: **-qopt-report-names=[mangled|unmangled]**,
Windows*: **/Qopt-report-names:[mangled|unmangled]**

Agenda

- Introduction to SIMD for Intel® Architecture
- Vector Code Generation
- Compiler & Vectorization
- Validating Vectorization Success
- **Reasons for Vectorization Fails**
- Vectorization of Special Program Constructs & Loops
- Intel® Cilk™ Plus
- OpenMP* 4.0
- 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 dependence
- 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

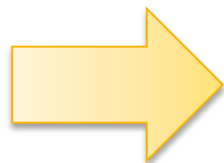
Those are less likely and are not described in the following!

Data Dependence in Loops

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

```
DO I = 1, N
S1   A(I+1) = A(I) + B(I)
ENDDO
```

$S_1 \delta^F S_1$



```
S1   A(2) = A(1) + B(1)
S1   A(3) = A(2) + B(2)
S1   A(4) = A(3) + B(3)
S1   A(5) = A(4) + B(4)
...

```

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

E.g.:

```
DO I = 1, 10000
S1   A(I) = B(I) * 17
S2   X(I+1) = X(I) + A(I)
ENDDO
```

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

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

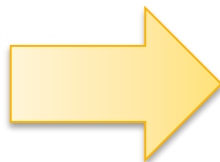
Dependence and Vectorization

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!

- For the formal proof we refer to the literature [3]
- The theorem takes into account that certain semantic-preserving reordering transformations can be applied (e.g. loop distribution, loop fusion, etc.)
- The theorem assumes an “unlimited” vector length (VL).
In cases where VL is limited, loop carried dependencies might be ignored if more than “VL” iterations are required to exist.
Thus in some cases vectorization for SSE or AVX might be still valid, opposed to the theorem!

Example:

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

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

Failing Disambiguation

Many dependencies assumed by compiler are false dependencies caused by unresolved memory disambiguation:

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

Example:

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

Without additional information (like inter-procedural knowledge) the compiler has to assume a and b to alias!

Use directives, switches and attributes to aid disambiguation!

- This is programming language and operating system specific
- Use with care as the compiler might generate incorrect code in case the hints are not fulfilled!

Disambiguation Hints I

- Disambiguating memory locations of pointers in C99:
Linux*, 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])
{ ... }
```

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 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  
    }  
}
```

Avoiding Non-Unit Stride Access

- Code transformations like loop interchange can avoid non-unit access frequently in case access is linear
- Compiler can do this automatically via loop interchange in most cases, e.g. matrix multiplication loop:

```
for(i = 0; i < N; i++)  
  for(j = 0; j < N; j++)  
    for(k = 0; k < N; k++)  
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
```



- But in other cases the exchange has to be done manually, e.g.:

```
// Non-unit stride  
for (j = 0; j < N; j++)  
  for (i = 0; i < j; i++)  
    c[i][j] = a[i][j] + b[i][j];
```



```
// Unit stride  
for (i = 0; i < N; i++)  
  for (j = i + 1; i < N; j++)  
    c[i][j] = a[i][j] + b[i][j];
```

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;  
}
```

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

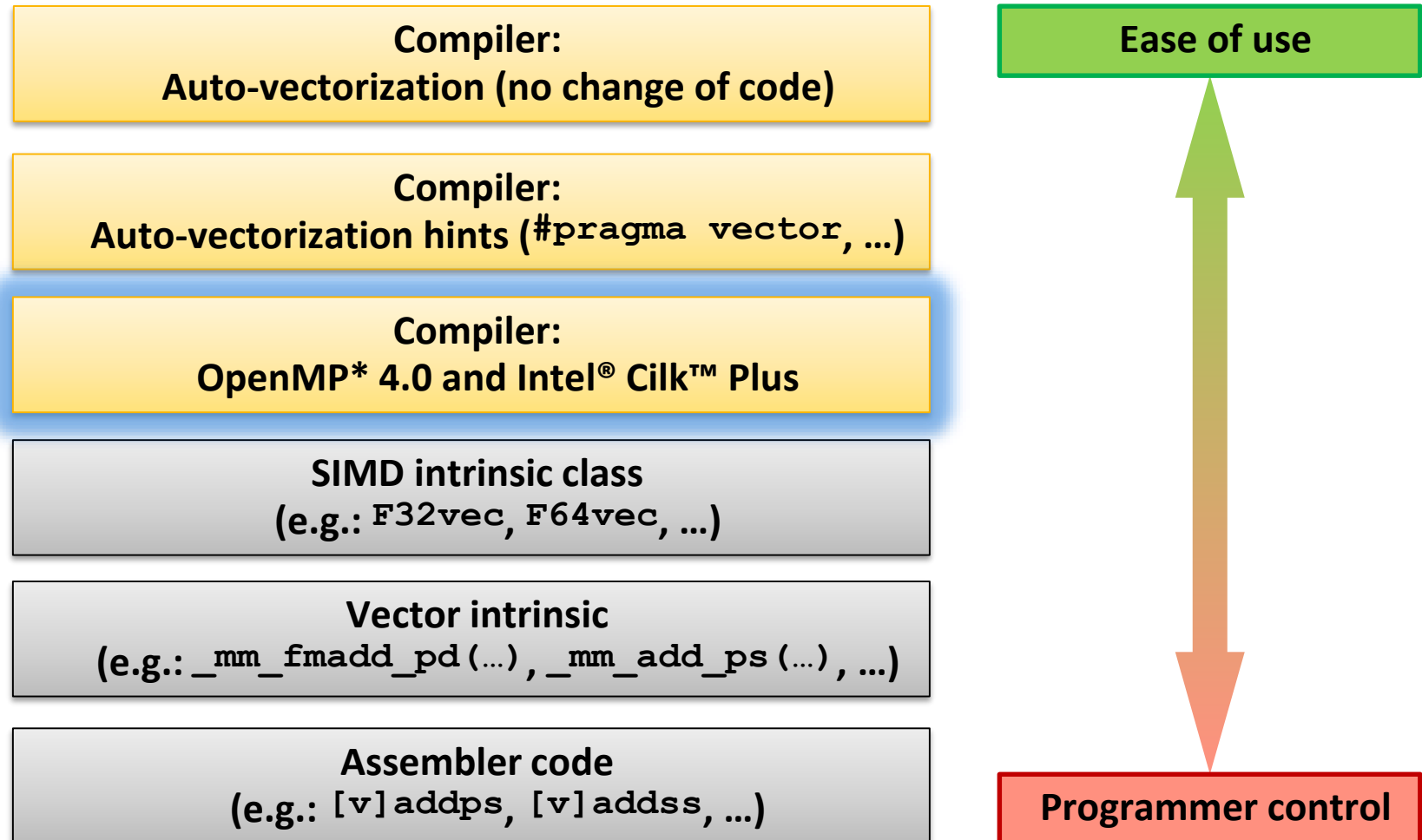
How to Succeed in Vectorization? II

- **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!

Agenda

- Introduction to SIMD for Intel® Architecture
- Vector Code Generation
- Compiler & Vectorization
- Validating Vectorization Success
- Reasons for Vectorization Fails
- **Intel® Cilk™ Plus**
- OpenMP* 4.0
- Summary

Intel® Cilk™ Plus



Intel® Cilk™ Plus

Task Level Parallelism

Simple Keywords

Set of keywords, for expression of task parallelism:

```
cilk_spawn  
cilk_sync  
cilk_for
```

Reducers

(Hyper-objects)

Reliable access to nonlocal variables without races

```
cilk::reducer_opadd<int> sum(3);
```

Data Level Parallelism

Array Notation

Provide data parallelism for sections of arrays or whole arrays

```
mask[:] = a[:] < b[:] ? -1 : 1;
```

SIMD-enabled Functions

Define actions that can be applied to whole or parts of arrays or scalars

Execution Parameters

Runtime system APIs, Environment variables, pragmas

Intel® Cilk™ Plus Pragma/Directive I

C/C++: `#pragma simd [clause [,clause]...]`

Fortran: `!DIR$ SIMD [clause [,clause]...]`

Without any clause, the directive “enforces” vectorization of the loop, ignoring all dependencies (even if they are proved!)

Example:

```
void addfl(float *a, float *b, float *c, float *d, float *e, int n)
{
    #pragma simd
    for(int i = 0; i < n; i++)
        a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
}
```

Without SIMD directive, vectorization likely fails since there are too many pointer references to do a run-time check for overlapping (compiler heuristic). The compiler won't create multiple versions here.

Using the directive asserts the compiler that none of the pointers are overlapping.

SIMD-Enabled Functions Syntax

Windows*:

```
__declspec(vector([clause [,clause]...]))  
function definition or declaration
```

Linux*/OS* X:

```
__attribute__((vector([clause [,clause]...])))  
function definition or declaration
```

- C/C++ only
- 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 Functions Clauses

- **processor(cpuid)**
cpuId for which (Intel) processor to create a vector version
- **vectorlength(len)**
len must be power of 2: Allow as many elements per argument
- **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)
- **[no]mask**: SIMD-enabled function called only inside branches (masked) or never (not masked)

Intrinsic also available: **__intel_simd_lane()**:
Return the SIMD lane with range: **[0:vector length - 1]**

SIMD-Enabled Functions

Write a function for one element and add `__declspec(vector):`

```
__declspec(vector)  
float foo(float a, float b, float c, float d)  
{  
    return a * b + c * d;  
}
```

Call the scalar version:

```
e = foo(a, b, c, d);
```

Call scalar version via SIMD loop:

```
#pragma simd  
for(i = 0; i < n; i++) {  
    A[i] = foo(B[i], C[i], D[i], E[i]);  
}
```

Call it with array notations:

```
A[:] = foo(B[:], C[:], D[:], E[:]);
```

SIMD-Enabled Functions: Invocation

```
__declspec(vector) float my_simdf (float b) { ... }
```

Construct	Example	Semantics
Standard for loop	<pre>for (j = 0; j < N; j++) { a[j] = my_simdf(b[j]); }</pre>	Single thread, maybe auto- vectorizable
#pragma simd	<pre>#pragma simd for (j = 0; j < N; j++) { a[j] = my_simdf(b[j]); }</pre>	Single thread, vectorized; use the appropriate vector version
Array notation	<pre>a[:] = my_simdf(b[:]);</pre>	Single thread, vectorized
OpenMP* 4.0	<pre>#pragma omp parallel for simd for (j = 0; j < N; j++) { a[j] = my_simdf(b[j]); }</pre>	Multi-threaded, vectorized

Intel® Cilk™ Plus

Task Level Parallelism

Simple Keywords

Set of keywords, for expression of task parallelism:

```
cilk_spawn  
cilk_sync  
cilk_for
```

Reducers

(Hyper-objects)

Reliable access to nonlocal variables without races

```
cilk::reducer_opadd<int> sum(3);
```

Data Level Parallelism

Array Notation

Provide data parallelism for sections of arrays or whole arrays

```
mask[:] = a[:] < b[:] ? -1 : 1;
```

SIMD-enabled Functions

Define actions that can be applied to whole or parts of arrays or scalars

Execution Parameters

Runtime system APIs, Environment variables, pragmas

Array Notation Extension: Syntax I

- An extension to C/C++ only
- Perform operations on sections of arrays in parallel
- Example:

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



```
A[:] = B[:] + C[:];
```

Not exactly the same: Aliasing is ignored by Array Notations!

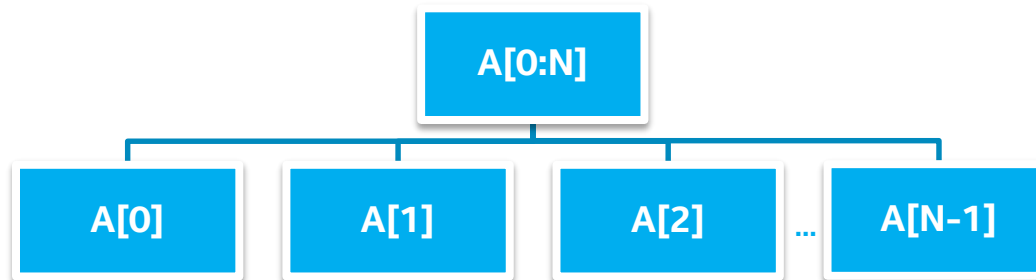
- Well suited for code that:
 - Performs per-element operations on arrays
 - Without an implied order between them (aliasing is ignored)
 - With an intent to execute in vector instructions

Array Notation Extension: Syntax II

- Syntax:

```
A[:]  
A[start_index : length]  
A[start_index : length : stride]
```

- Use a “:” for all elements (if size is known)
- “length” specifies number of elements of subset
- “stride”: distance between elements for subset

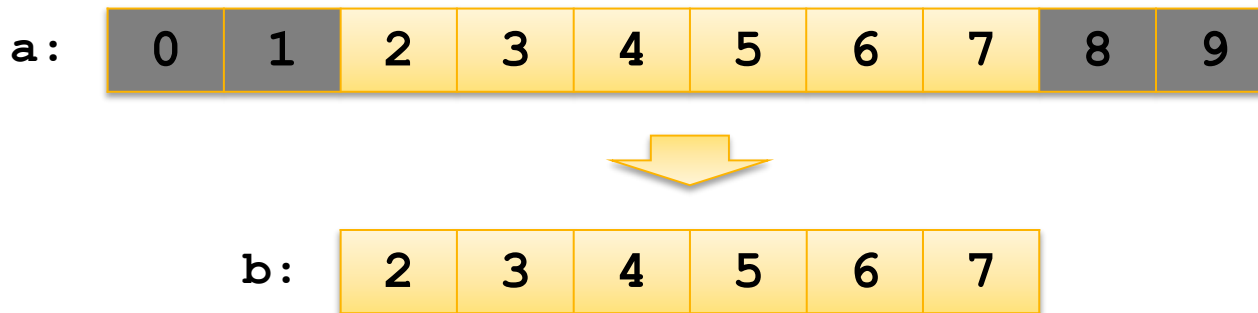


Explicit Data Parallelism Based on C/C++ Arrays

Array Notation Extension: Example I

Accessing a section of an array:

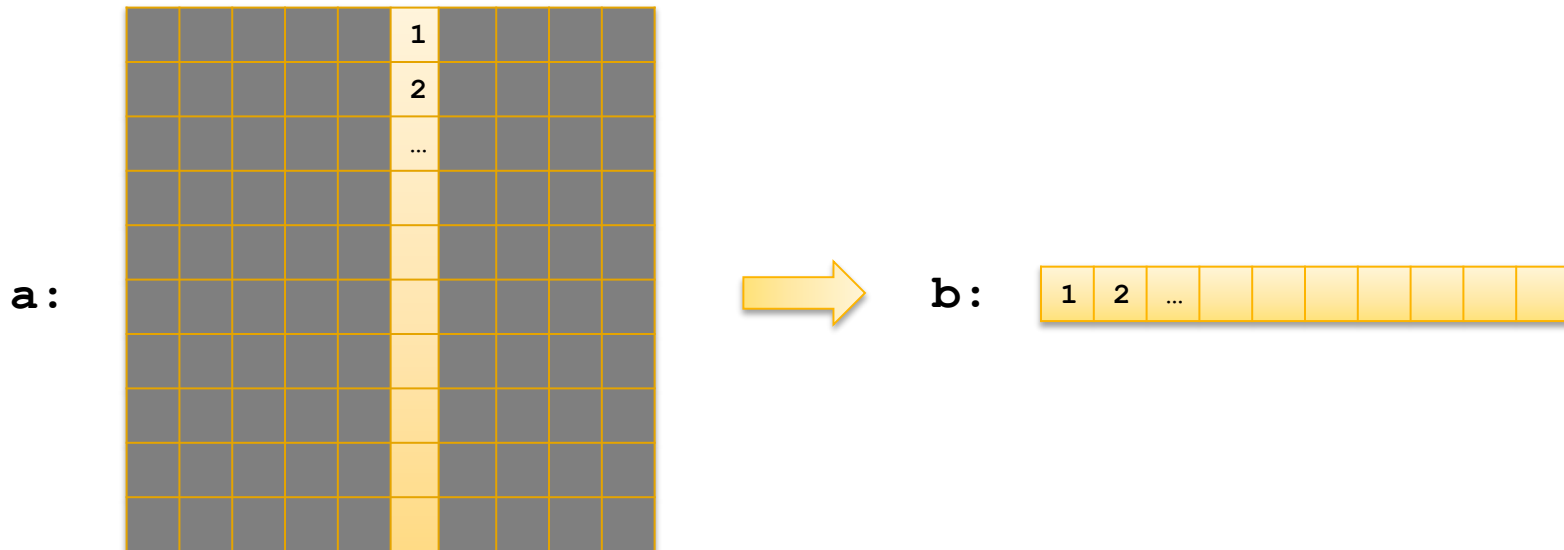
```
float a[10], b[6];  
...  
// allocate *b  
...  
b[:] = a[2:6];  
...
```



Array Notation Extension: Example II

Section of 2D array:

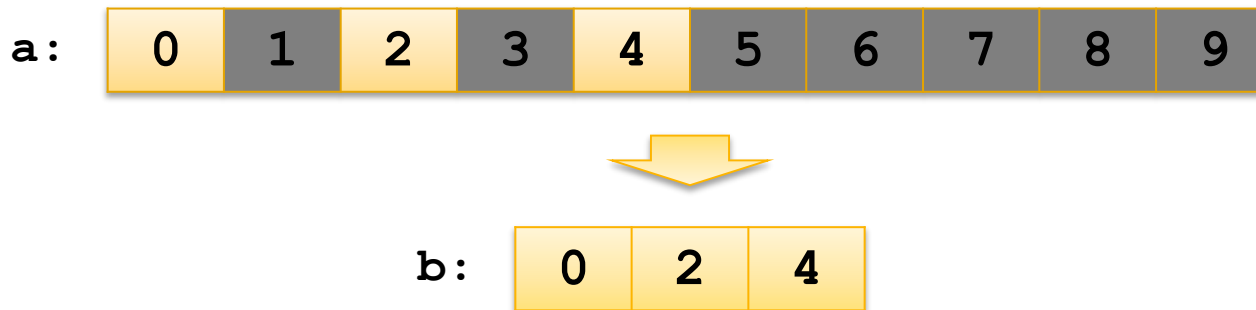
```
float a[10][10], *b;  
...  
// allocate *b  
...  
b[0:10] = a[:][5];  
...
```



Array Notation Extension: Example III

Strided section of an array:

```
float a[10], *b;  
...  
// allocate *b  
...  
b[0:3] = a[0:3:2];  
...
```



Array Notation Extension: Operators

Most C/C++ operators are available for array sections:

`+`, `-`, `*`, `/`, `%`, `<`, `==`, `!=`, `>`, `|`, `&`, `^`, `&&`, `||`, `!`, `-` (unary), `+` (unary), `++`, `--`, `+=`, `-=`, `*=`, `/=`, `*` (pointer de-referencing)

Examples:

```
a[:] * b[:]           // element-wise multiplication
a[3:2][3:2] + b[5:2][5:2] // matrix addition
a[0:4][1:2] + b[1:2][0:4] // error, different rank sizes
a[0:4][1:2] + c        // adds scalar c to array section
```

- Operators are implicitly mapped to all elements of the array section operands.
- Operations on different elements can be executed in parallel without any ordering constraints.
- Array operands must have the same **rank** and **size**.
- Scalar operands are automatically expanded.

Array Notation Extension: Reductions

Combine array section elements using a predefined operator, or a user function:

```
int a[] = {1,2,3,4};
sum = __sec_reduce_add(a[:]); // sum is 10
res = __sec_reduce(0, a[:], func);
    // apply function func to all
    // elements in a[], initial value is 0
int func(int arg1, int arg2)
{
    return arg1 + arg2;
}
```

Other reductions (list not exhaustive):

```
__sec_reduce_mul, __sec_reduce_all_zero,
__sec_reduce_all_nonzero, __sec_reduce_any_nonzero,
__sec_reduce_max, __sec_reduce_min,
__sec_reduce_max_ind, __sec_reduce_min_ind
```

Much more! Take a look at the specification:

https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm

Array Notation Extension: Example I

Serial version:

```
float dot_product(unsigned int size, float A[size], float B[size])
{
    int i;
    float dp = 0.0f;
    for (i=0; i<size; i++) {
        dp += A[i] * B[i];
    }
    return dp;
}
```

Array Notation version:

```
float dot_product(unsigned int size, float A[size], float B[size])
{
    // A[:] can also be written as A[0:size]
    return __sec_reduce_add(A[:] * B[:]);
}
```

Array Notation Extension: Example II

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

```
void vec3(float *a, float *b, int off, int len)
{
    __assume_aligned(a, 64);
    __assume_aligned(b, 64);
    a[0:len] = (a[0:len] > 1.0) ?
                a[0:len] * b[0:len] :
                a[off:len] * b[0:len];
}
```

LOOP BEGIN at simd.cpp(5,9)

remark #15388: vectorization support: reference a has aligned access [simd.cpp(5,28)]

remark #15388: vectorization support: reference b has aligned access [simd.cpp(5,28)]

...

remark #15300: LOOP WAS VECTORIZED

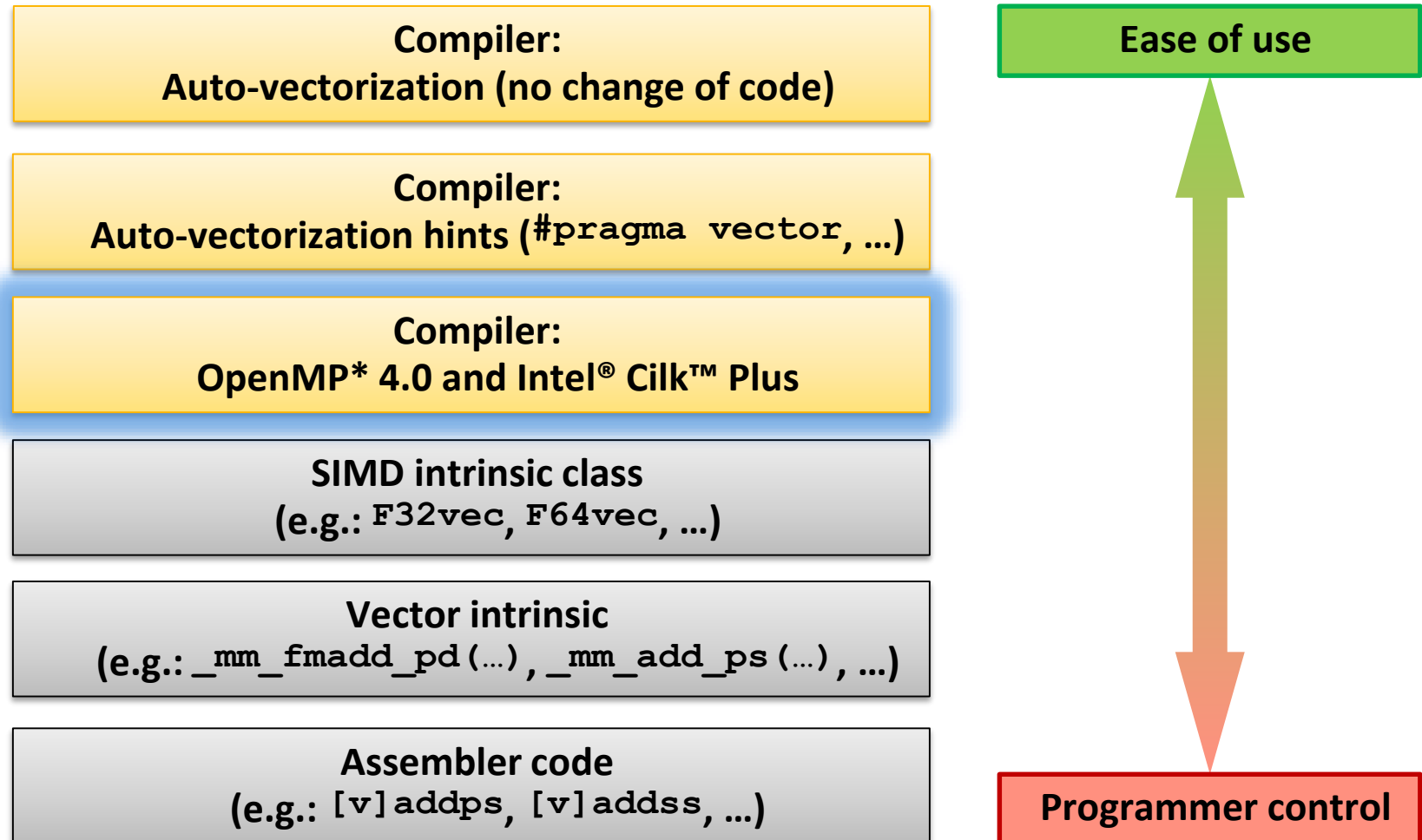
...

LOOP END

Agenda

- Introduction to SIMD for Intel® Architecture
- Vector Code Generation
- Compiler & Vectorization
- Validating Vectorization Success
- Reasons for Vectorization Fails
- Intel® Cilk™ Plus
- **OpenMP* 4.0**
- Summary

Intel® Cilk™ Plus



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++)
    {
        a[i] = work(a, b, i, off);
    }
}
```

```
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
```

Agenda

- Introduction to SIMD for Intel® Architecture
- Vector Code Generation
- Compiler & Vectorization
- Validating Vectorization Success
- Reasons for Vectorization Fails
- Vectorization of Special Program Constructs & Loops
- Intel® Cilk™ Plus
- OpenMP* 4.0
- **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/>



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 © 2016, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

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

Notice revision #20110804

