HLRN User Workshop 3-6 Nov 2020

Intel Compiler

Overview

Klaus-Dieter Oertel

intel.

Key Updates

- New Underlying Back End Compilation Technology based on LLVM
- LLVM technology available in oneAPI Betas for DPC++, C++ and Fortran
- IL0/"Classic" Back End Compilation Technology Continues to be Available
- OpenMP Offload supported only on LLVM versions
- F2018 Full coverage in IL0 (oneAPI HPC Toolkit)

Intel® Compilers – DPC++, C/C++ & Fortran

Intel Compiler (Technology)	Compiler driver	Target	OpenMP Offload*	Current Status
C/C++ (IL0)	icc	CPU	No	Production
C/C++ (LLVM)	icx / icc -qnextgen	CPU, GPU*	Yes	Production/Beta
DPC++ (LLVM)	dpcpp	CPU, GPU, FPGA	NA	Beta
Fortran (IL0)	ifort	CPU	No	Production
Fortran (LLVM)	ifx	CPU, GPU*	Yes	Beta

Cross Compiler Binary Compatible and Linkable

What's New for Intel compilers 19.1?

"classic" icc/ifort

Advance Support for Intel® Architecture – Use Intel compiler to generate optimized code for Intel Atom® processor through Intel® Xeon® Scalable processor families

Achieve Superior Parallel Performance – Vectorize & thread your code (using OpenMP*) to take full advantage of the latest SIMD-enabled hardware, including Intel® Advanced Vector Extensions 512 (Intel® AVX-512)

What's New in C++

Initial C++20, and full C++ 17 enabled

- Enjoy advanced lambda and constant expression support
- Standards-driven parallelization for C++ developers

Initial OpenMP* 5.0, and full OpenMP* 4.5 support

- Modernize your code by using the latest parallelization specifications

What's New in Fortran

Substantial Fortran 2018 support

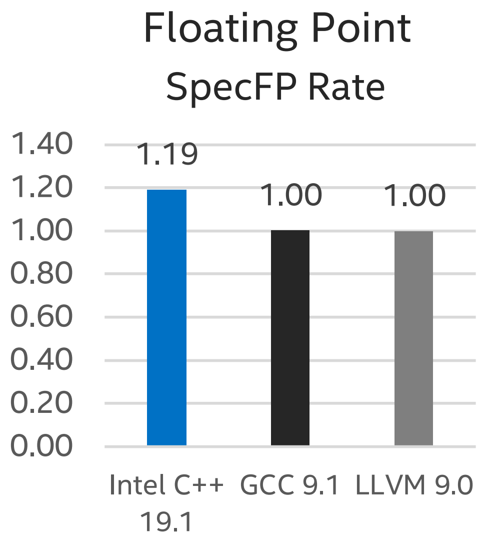
- Enjoy enhanced C-interoperability features for effective mixed language development
- Use advanced coarray features to parallelize your modern Fortran code

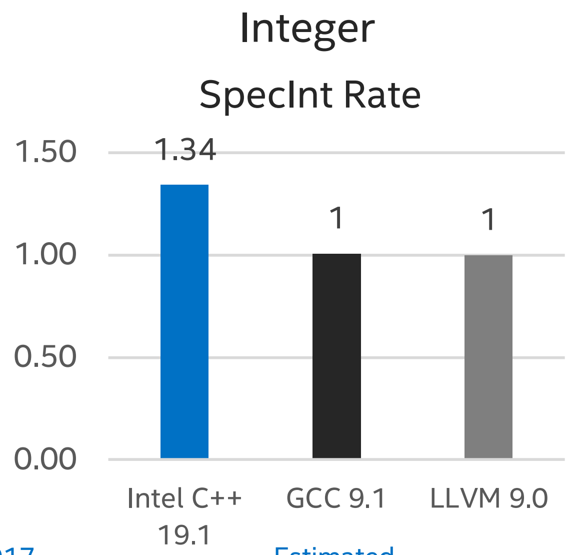
Initial OpenMP* 5.0, and substantial OpenMP* 4.5 support

- Customize your reduction operations by user-defined reductions

Intel® C++ Compiler Boosts Application Performance on Linux*

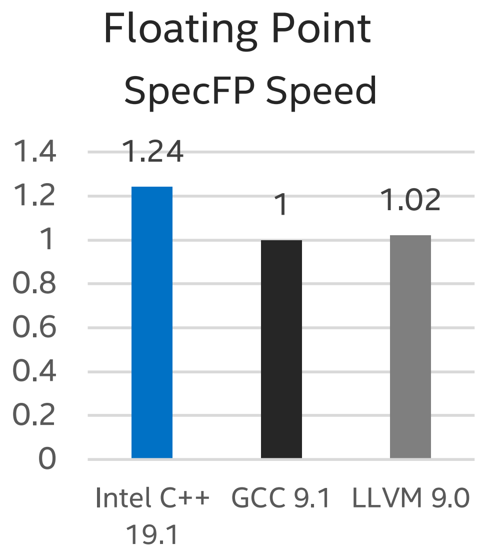
Relative geomean performance (FP Rate Base and FP Speed Base; higher is better)

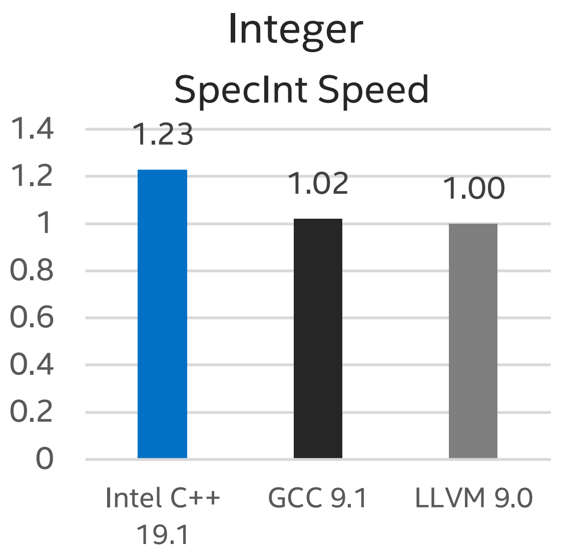




Estimated geometric mean of SPEC* CPU2017 SPECint® **_rate_base2017** Floating Point **RATE BASE** C/C++ benchmarks

Estimated





Estimated geometric mean of SPEC* CPU2017 SPECint® **_speed_base2017** Floating Point **SPEED BASE** C/C++ benchmarks

Estimated

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer. Performance results are based on testing as of Aug. 26, 2019 and may not reflect all publicly available security updates. See configuration disclosure for details. No product can be absolutely secure. Software and workloads used in performance tests may have been optimized. for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information about performance and benchmark results, visit www.intel.com/benchmarks.

Configuration: Testing by Intel as of Aug. 26, 2019. Linux hardware: Intel® Xeon® Platinum 8180 CPU @ 2.50GHz, 384 GB RAM, HyperThreading is on. Software: Intel® C++ Compiler 19.1, GCC 9.1.0. Clang/LLVM 9.0. Linux OS: Red Hat* Enterprise Linux Server release 7.4 (Maipo), 3.10.0-693.el7.x86_64. SPEC* Benchmark (www.spec.org). SPECint®_rate_base_2017 compiler switches: qkmalloc was used for Intel C++ Compiler 19.1 SPECint rate test, jemalloc 5.0.1 was used for GCC and Clang/LLVM SPECint rate test. Intel® C Compiler / Intel C++ Compiler 19.1: -xCORE-AVX512 -ipo -O3 -no-prec-div -qopt-mem-layout-trans=4. GCC 9.1.0 -march=skylake-avx512 -mfpmath=sse -Ofast -funroll-loops -flto. Clang 9.0: -march=skylake-avx512 -mfpmath=sse -Ofast -funroll-loops –flto. SPECfp®_rate_base_2017 compiler switches: jemalloc 5.0.1 was used for Intel C++ Compiler 19.1, GCC and Clang/LLVM SPECfp rate test. Intel C/C++ compiler 19.1: -xCORE-AVX512 -ipo -O3 -no-prec-div -qopt-prefetch -ffinite-math-only -qopt-mem-layout-trans=4. GCC 9.1.0: -march=skylake-avx512 -mfpmath=sse -Ofast -fno-associative-math -funroll-loops -flto. Clang 9.0: -march=skylake-avx512 -mfpmath=sse -Ofast -funroll-loops –flto. SPECint®_speed_base_2017 compiler switches: Intel C Compiler / Intel C++ Compiler 19.1: -xCORE-AVX512 -ipo -O3 -no-prec-div -qopt-mem-layout-trans=4 -qopenmp. GCC 9.1.0: -march=skylake-avx512 -mfpmath=sse -Ofast -funroll-loops -flto -fopenmp. Clang 9.0: -march=skylake-avx512 -mfpmath=sse -Ofast -funroll-loops -flto -fopenmp=libomp. SPECfp®_speed_base_2017 compiler switches: Intel C Compiler / Intel C++ Compiler 19.1: -xCORE-AVX512 -ipo -O3 -no-prec-div -qopt-prefetch -ffinite-math-only -qopenmp. GCC 9.1.0: -march=skylake-avx512 -mfpmath=sse -Ofast -fno-associative-math -funroll-loops -flto -fopenmp. Clang 9.0: -march=skylake-avx512 -mfpmath=sse -Ofast -funroll-loops -flto -fopenmp=libomp. compiler switches: jemalloc 5.0.1 was used for Intel C++ Compiler 19.0 update 4, GCC and Clang/LLVM SPECfp rate test. Intel C/C++ compiler 19.1:-.

Common optimization options

	Linux*
Disable optimization	-O0
Optimize for speed (no code size increase)	-O1
Optimize for speed (default)	-O2
High-level loop optimization	-O3
Create symbols for debugging	-g
Multi-file inter-procedural optimization	-ipo
Profile guided optimization (multi-step build)	-prof-gen -prof-use
Optimize for speed across the entire program ("prototype switch") ***fast* options definitions changes over time!**	-fast same as: -ipo –O3 -no-prec-div –static –fp-model fast=2 -xHost)
OpenMP support	-qopenmp
Automatic parallelization	-parallel

https://tinyurl.com/icc-user-guide

High-Level Optimizations

Basic Optimizations with icc -O…

-O0 no optimization; sets -g for debugging

-O1 scalar optimizations
excludes optimizations tending to increase code size

-O2 **default** for icc/icpc (except with -g)
includes **auto-vectorization**; some loop transformations, e.g. unrolling, loop interchange;
inlining within source file;
start with this (after initial debugging at -O0)

-O3 more aggressive loop optimizations
including cache blocking, loop fusion, prefetching, …
suited to applications with loops that do many floating-point calculations or process large data sets

InterProcedural Optimizations (IPO)

Multi-pass Optimization

`icc -ipo`

Analysis and optimization across function and/or source file boundaries, e.g.

- Function inlining; constant propagation; dependency analysis; data & code layout; etc.

2-step process:

- Compile phase – objects contain intermediate representation
- "Link" phase – compile and optimize over all such objects
- Seamless: linker automatically detects objects built with -ipo and their compile options
- May increase build-time and binary size
- But build can be parallelized with `-ipo=n`
- Entire program need not be built with IPO, just hot modules

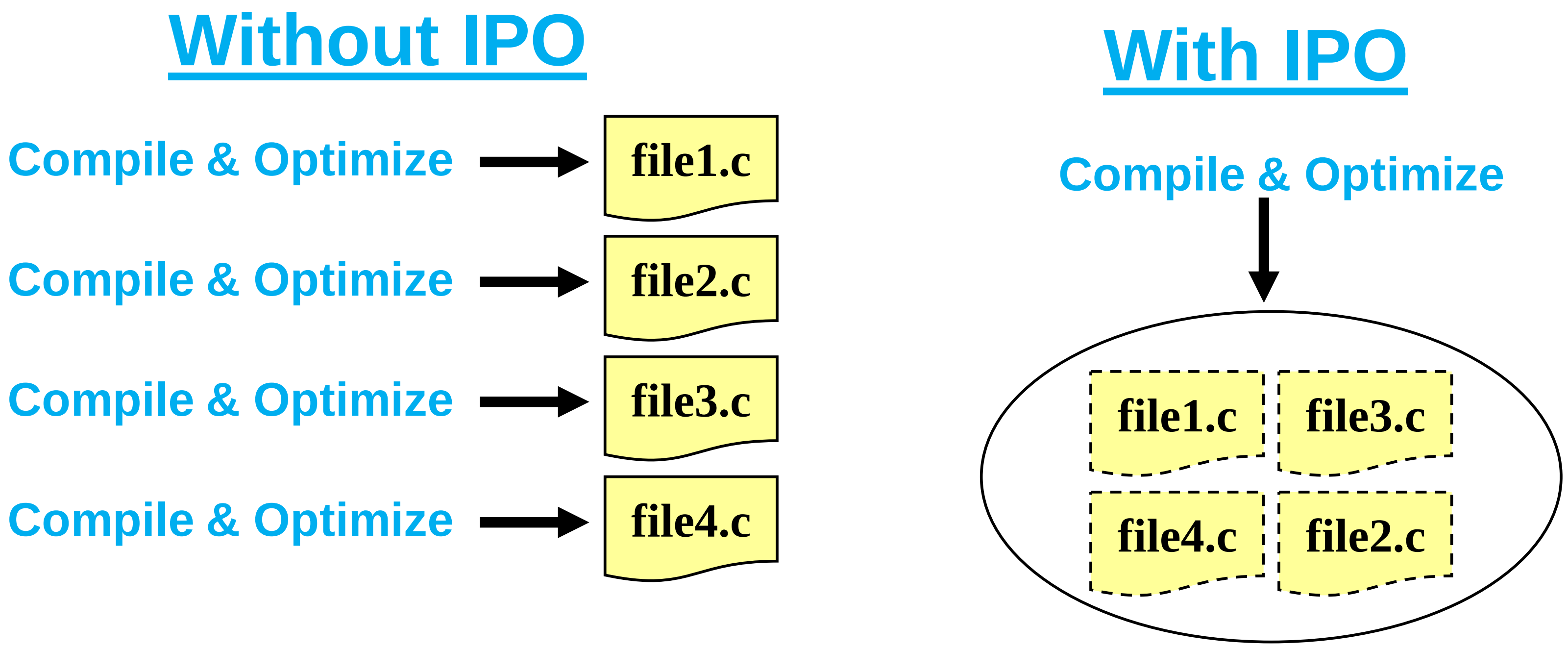
Particularly effective for applications with many smaller functions

Get report on inlined functions with `-qopt-report-phase=ipo`

InterProcedural Optimizations

Extends optimizations across file boundaries

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



Profile-Guided Optimizations (PGO)

Static analysis leaves many questions open for the optimizer like:

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

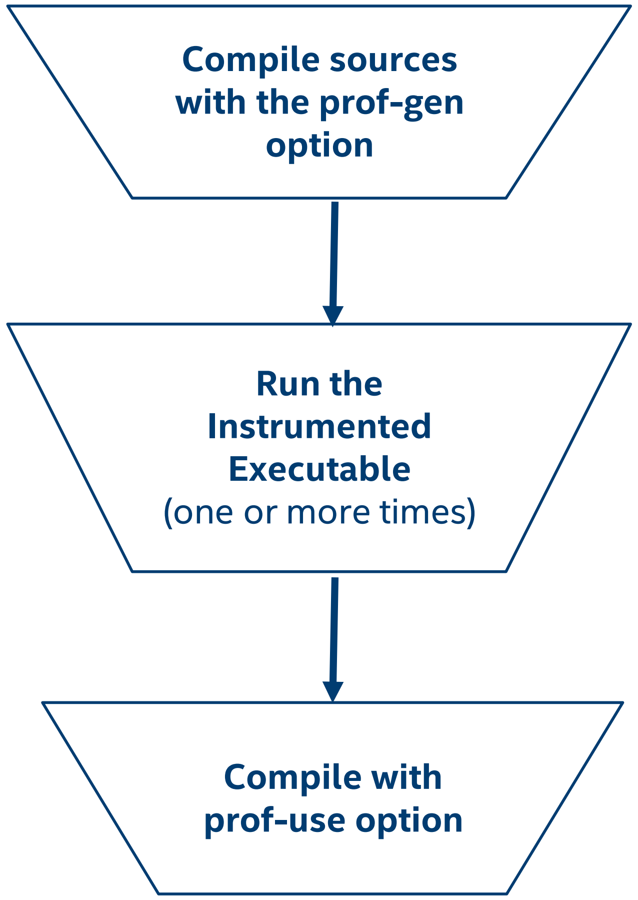
```
if (x > y)
     do_this();
 else
     do that();
```

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

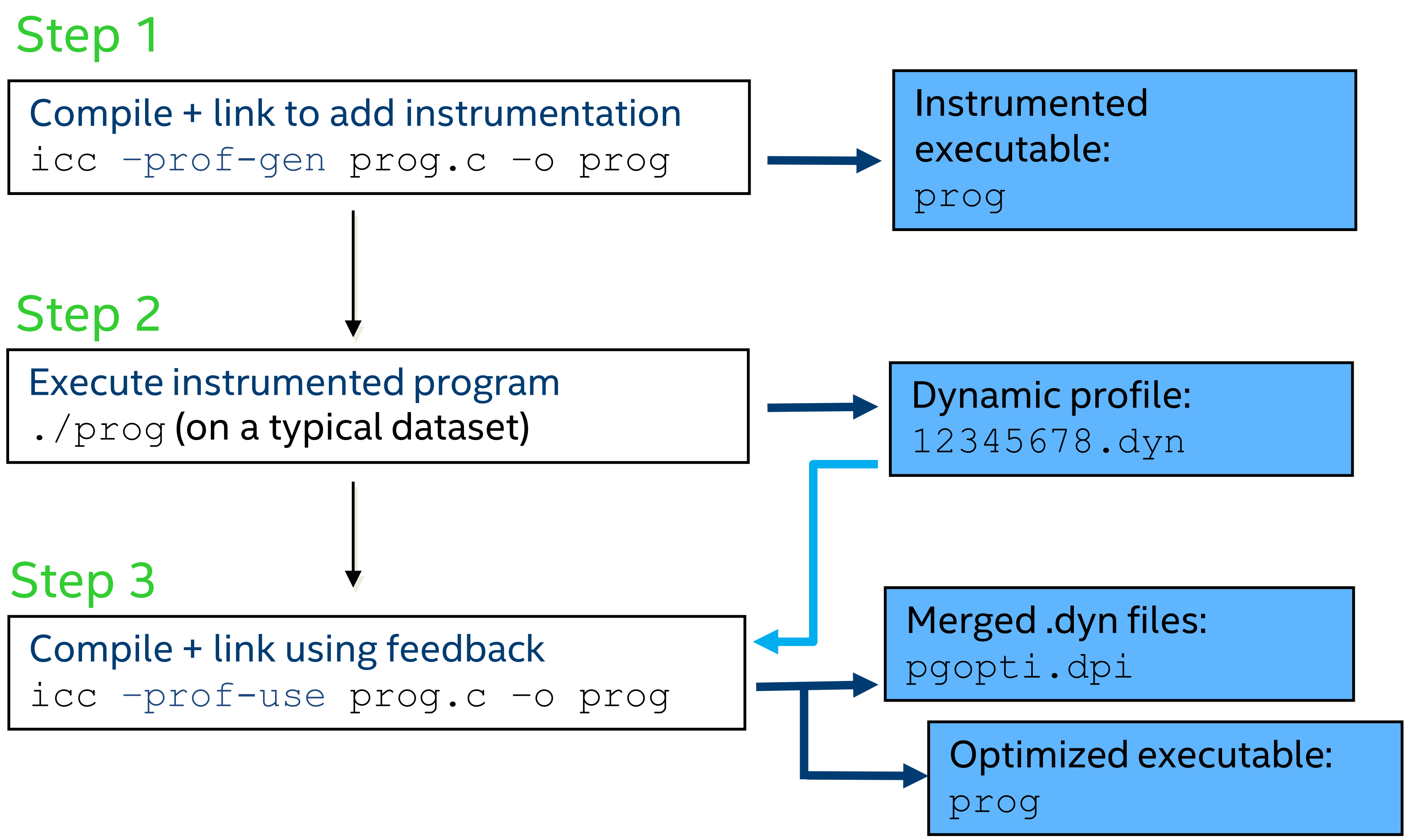
Use execution-time feedback to guide (final) optimization

Enhancements with PGO:

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


Compile sources with the prof-gen option
Run the Instrumented Executable (one or more times)
Compile with prof-use option

PGO Usage: Three-Step Process



Step 1

Compile + link to add instrumentation

```
icc -prof-gen prog.c -o prog
```

Instrumented executable:

```
prog
```

Step 2

Execute instrumented program

`./prog` (on a typical dataset)

Dynamic profile:

```
12345678.dyn
```

Step 3

Compile + link using feedback

```
icc -prof-use prog.c -o prog
```

Merged .dyn files:

```
pgopti.dpi
```

Optimized executable:

```
prog
```

Math Libraries

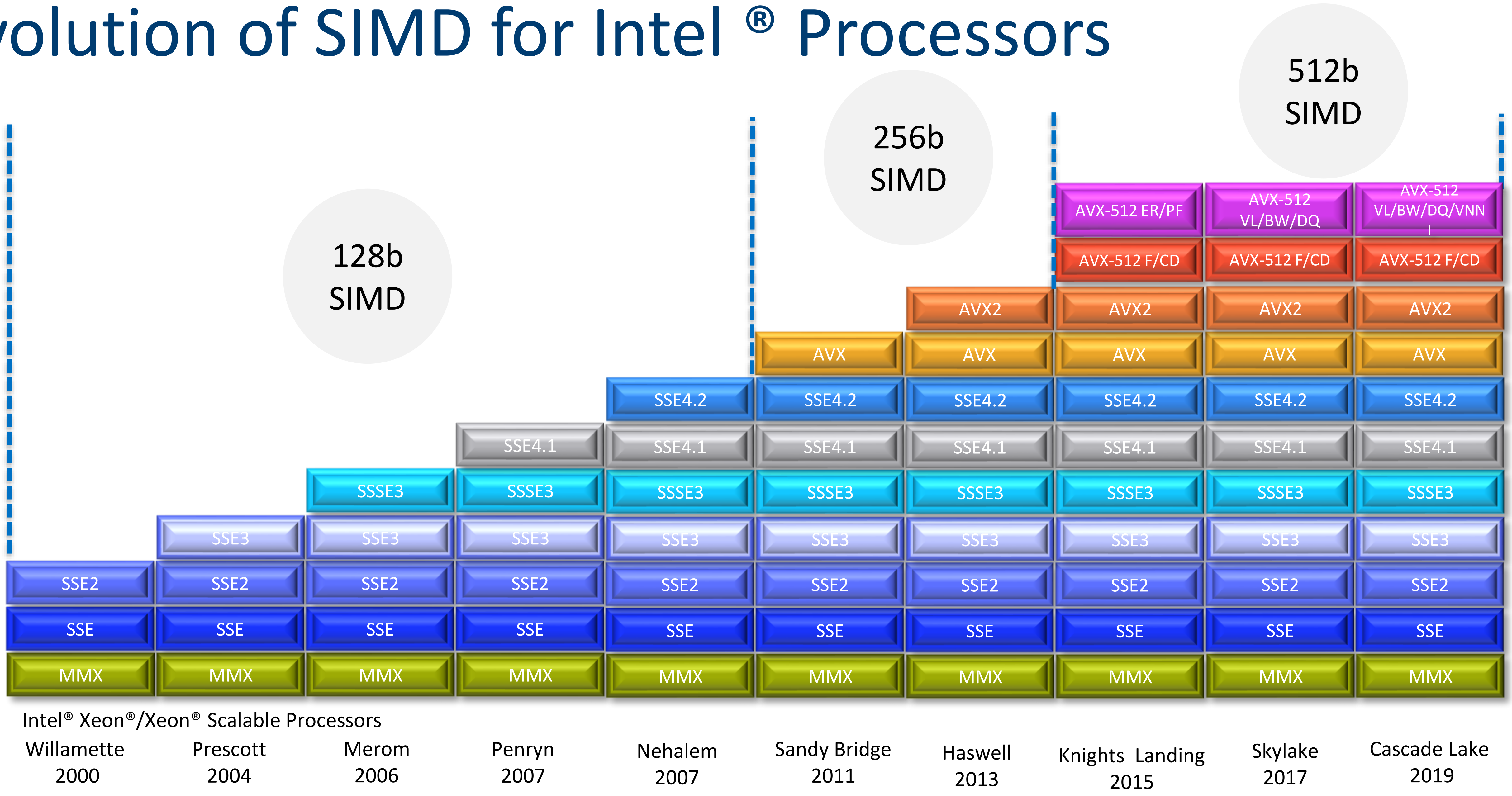
icc comes with Intel's optimized math libraries

- libimf (scalar) and libsvml (scalar & vector)
- Faster than GNU* libm
- Driver links libimf automatically, ahead of libm
- Additional functions (replace math.h by mathimf.h)

Don't link to libm explicitly!　　🚫 -lm 🚫

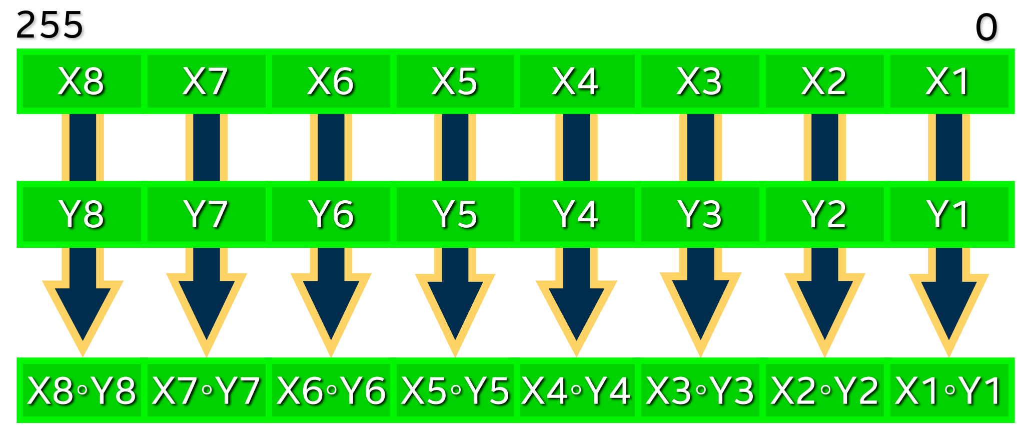
- May give you the slower libm functions instead
- Though the Intel driver may try to prevent this
- gcc needs -lm, so it is often found in old makefiles

Evolution of SIMD for Intel ® Processors



Willamette 2000	Prescott 2004	Merom 2006	Penryn 2007	Nehalem 2007	Sandy Bridge 2011	Haswell 2013	Knights Landing 2015	Skylake 2017	Cascade Lake 2019
							AVX-512 ER/PF	AVX-512 VL/BW/DQ	AVX-512 VL/BW/DQ/VNN I
							AVX-512 F/CD	AVX-512 F/CD	AVX-512 F/CD
						AVX2	AVX2	AVX2	AVX2
					AVX	AVX	AVX	AVX	AVX
				SSE4.2	SSE4.2	SSE4.2	SSE4.2	SSE4.2	SSE4.2
			SSE4.1	SSE4.1	SSE4.1	SSE4.1	SSE4.1	SSE4.1	SSE4.1
		SSSE3	SSSE3	SSSE3	SSSE3	SSSE3	SSSE3	SSSE3	SSSE3
	SSE3	SSE3	SSE3	SSE3	SSE3	SSE3	SSE3	SSE3	SSE3
SSE2	SSE2	SSE2	SSE2	SSE2	SSE2	SSE2	SSE2	SSE2	SSE2
SSE	SSE	SSE	SSE	SSE	SSE	SSE	SSE	SSE	SSE
MMX	MMX	MMX	MMX	MMX	MMX	MMX	MMX	MMX	MMX

SIMD Types for Intel® Architecture



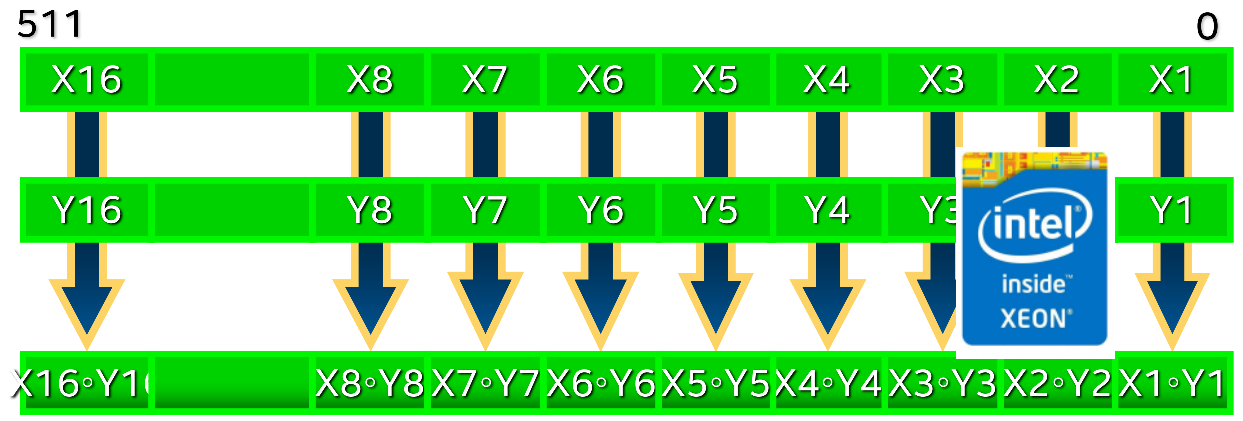
AVX

Vector size: **256 bit**

Data types:

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

VL: 4, 8, 16, 32



511
0
X16 X8 X7 X6 X5 X4 X3 X2 X1
Y16 Y8 Y7 Y6 Y5 Y4 Y3 Y1
X16◦Y16 X8◦Y8 X7◦Y7 X6◦Y6 X5◦Y5 X4◦Y4 X3◦Y3 X2◦Y2 X1◦Y1
intel inside XEON

Intel® AVX-512

Vector size: **512 bit**

Data types:

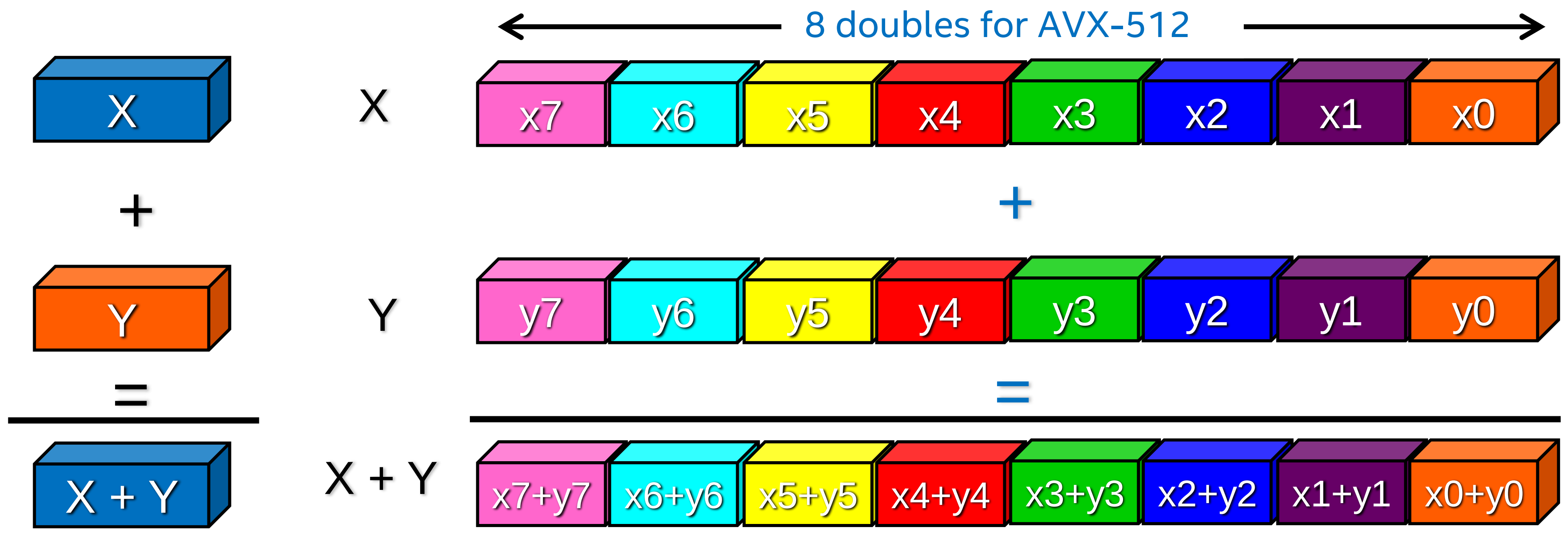
- 8, 16, 32, 64 bit integer
- 32 and 64 bit float

VL: 8, 16, 32, 64

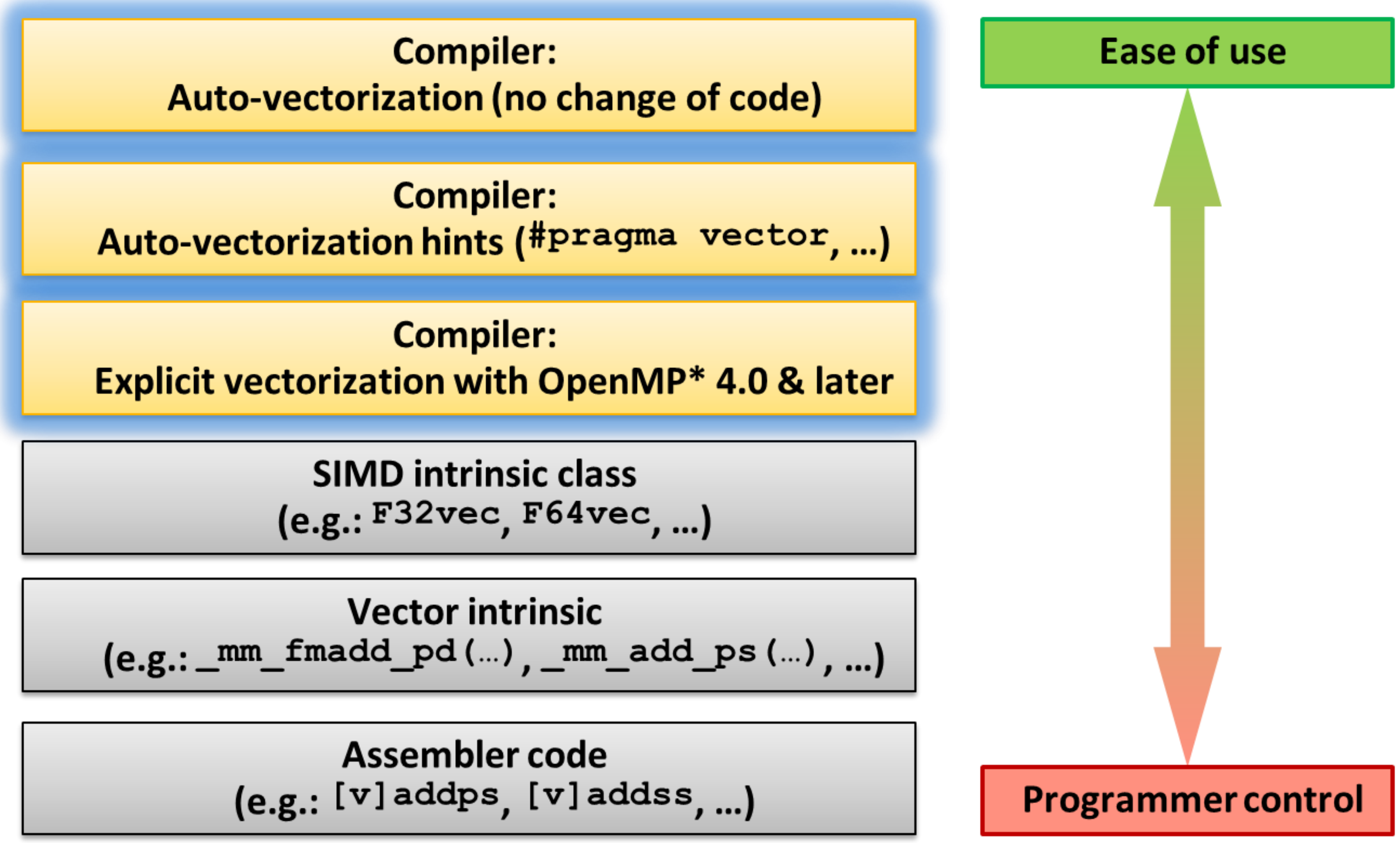
SIMD: Single Instruction, Multiple Data

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

- Scalar mode
  - one instruction produces one result
  - E.g. vadd**ss**, (vadd**sd**)
- Vector (SIMD) mode
  - one instruction can produce multiple results
  - E.g. vadd**ps**, (vadd**pd**)



Many ways to vectorize



Compiler:
Auto-vectorization (no change of code)

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

Compiler:
Explicit vectorization with OpenMP* 4.0 & later

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`, …)

Basic Vectorization Switches I

`-x<code>`

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

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

Microarchitecture code names: BROADWELL, HASWELL, IVYBRIDGE, KNL, KNM, SANDYBRIDGE, SILVERMONT, SKYLAKE, SKYLAKE-AVX512

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

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

Basic Vectorization Switches II

`-ax<code>`

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

```
icc -axCORE-AVX512 -xAVX test.c
icc -axCORE-AVX2,CORE-AVX512 test.c
```

`-m<code>`

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

`-xHost`

Tuning for Skylake/Cascade Lake - Compiler options

- Skylake/Cascade Lake and Knights Landing processors have support for Intel® AVX-512 instructions. There are three ISA options in the Intel® Compiler:
- **`-xCORE-AVX512`** : Targets Skylake/Cascade Lake, contains instructions not supported by KNL
- **`-xCOMMON-AVX512`** : Targets all Skylake/Cascade Lake and Knights Landing
- **`-xMIC-AVX512`** : Targets Knights Landing, includes instructions not supported by Skylake

- Intel® Compiler is conservative in its use of ZMM (512bit) registers so to enable their use with Skylake/Cascade Lake the additional flag **`-qopt-zmm-usage=high`** must be set.

Compiler Reports – Optimization Report

- -qopt-report[=n]: tells the compiler to generate an optimization report
  - n: (Optional) Indicates the level of detail in the report. You can specify values 0 through 5. If you specify zero, no report is generated. For levels n=1 through n=5, each level includes all the information of the previous level, as well as potentially some additional information. Level 5 produces the greatest level of detail. If you do not specify n, the default is level 2, which produces a medium level of detail.
- -qopt-report-phase[=list]: specifies one or more optimizer phases for which optimization reports are generated.
  - loop: the phase for loop nest optimization
  - vec: the phase for vectorization
  - par: the phase for auto-parallelization
  - all: all optimizer phases
- -qopt-report-filter=string: specified the indicated parts of your application, and generate optimization reports for those parts of your application.

Example of Optimization report 1/3

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

Begin optimization report for: foo

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

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

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

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

Example of Optimization report 2/3

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

Begin optimization report for: foo

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


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

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

Example of Optimization report 3/3

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

Begin optimization report for: foo

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

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

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

Auto-Parallelization

Based on OpenMP* runtime

Compiler automatically translates loops into equivalent multithreaded code with using this option:

```
-parallel
```

The auto-parallelizer detects simply structured loops that may be safely executed in parallel, and automatically generates multi-threaded code for these loops.

The auto-parallelizer report can provide information about program sections that were parallelized by the compiler. Compiler switch:

```
-qopt-report-phase=par
```

The -fp-model switch

-fp-model

▪ fast [=1]	allows value-unsafe optimizations (default)
▪ fast=2	allows a few additional approximations
▪ precise	value-safe optimizations only
▪ source | double | extended	imply “precise” unless overridden see “FP Expression Evaluation” for more detail
▪ except	enable floating-point exception semantics
▪ strict	precise + except + disable fma + don’t assume default floating-point environment
▪ consistent	most reproducible results between different processor types and optimization options

-fp-model precise -fp-model source

- recommended for best reproducibility
- also for ANSI/ IEEE standards compliance, C++ & Fortran
- “source” is default with “precise” on Intel 64

Looking for best compiler options?

It depends!

- workload, hw, OS, compiler version, memory allocation, etc.
- take a look on benchmark results and options for reference:

SPECint®_rate_base_2017

-xCORE-AVX512 -ipo -O3 -no-prec-div -qopt-mem-layout-trans=4

SPECfp®_rate_base_2017

-xCORE-AVX512 -ipo -O3 -no-prec-div -qopt-prefetch -ffinite-math-only -qopt-mem-layout-trans=4

SPECint®_speed_base_2017

-xCORE-AVX512 -ipo -O3 -no-prec-div -qopt-mem-layout-trans=4 -qopenmp

SPECfp®_speed_base_2017

-xCORE-AVX512 -ipo -O3 -no-prec-div -qopt-prefetch -ffinite-math-only -qopenmp

Notices & Disclaimers