Intel® Compilers

April 2026

All information provided in this deck is subject to change without notice.
Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

Agenda

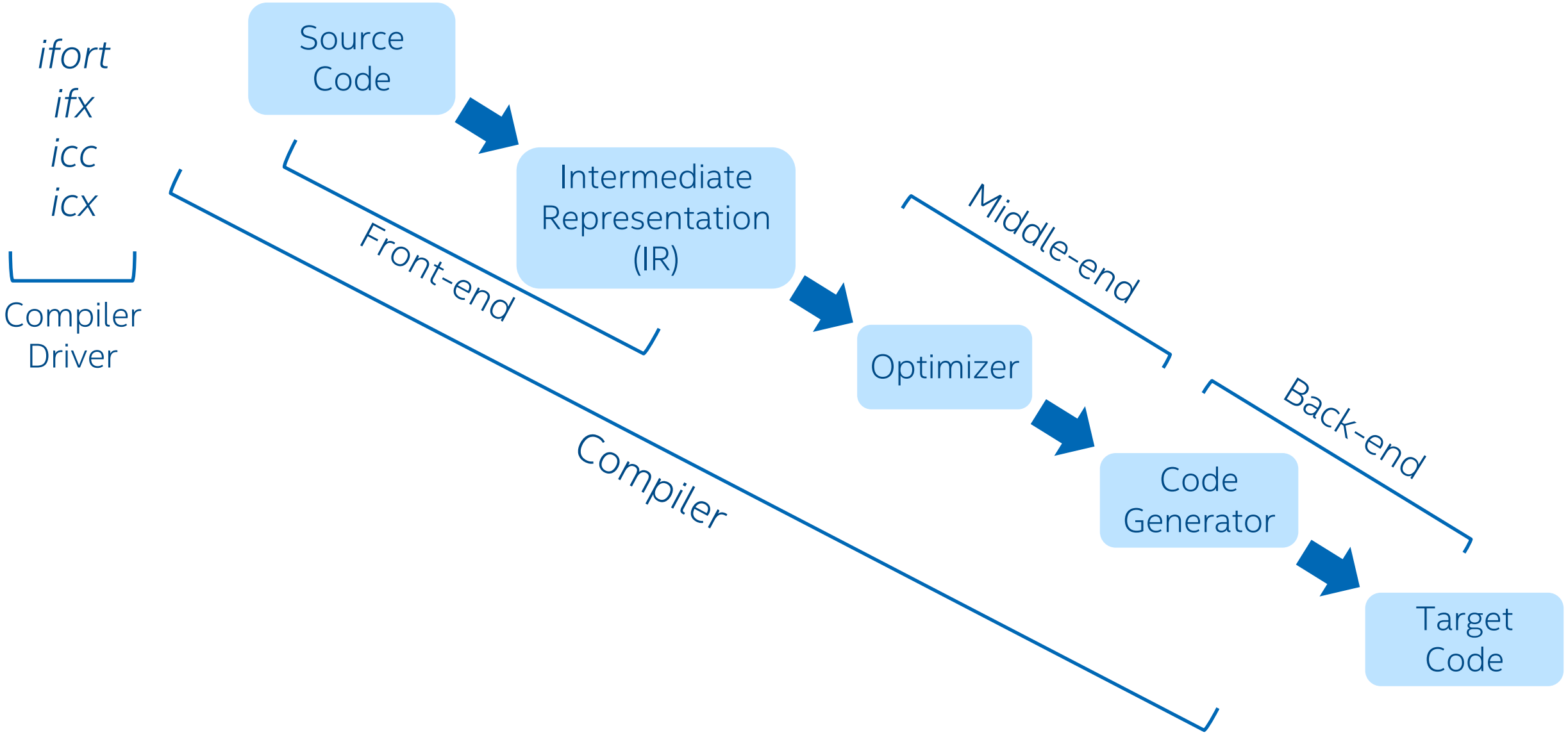
- Part I – Functional Migration:
  - Intel Compilers Evolution
  - Migration from Intel Compilers Classic (ICC/IFORT) to LLVM-based (ICX/IFX)
    - Key Differences
    - C++ Specifics: Changes in Diagnostics
      - Lab 1
  - Compiler Options for Debugging
    - Lab2
- Part II - Optimizations
  - Vectorization
    - Lab 3
  - Floating-point Model
  - Interprocedural Optimizations
  - Intrinsic Usage Model

tinyurl.com/Intel-Compiler-Workshop-Slides

Part I – Functional Migration

Intel Compiler Evolution

Compiler Architecture – Simplified View



Intel Compiler Evolution

- icc/icpc - Intel C++ Compiler Classic
  - IL0-based compiler (EDG Front-End, proprietary IR and Optimizer)
  - Deprecated July 2023; last public build: 2021.10 (included in oneAPI 2023.2)
  - Removed from oneAPI 2024.0 packages (late 2023)

- icx/icpx - Intel® oneAPI DPC++/C++ Compiler
  - Next-Generation LLVM-Based Compiler
  - Clang Front-End for parsing and semantic analysis
    - Improved Diagnostics
    - Strong support for latest standards, e.g. C++17/C++20/C++23
  - Modern LLVM-based backend: LLVM IR + LLVM optimizations + Intel proprietary optimization passes
  - GPU offload via SYCL/OpenMP to Intel GPUs
  - Microsoft Visual Studio integration* for Windows* development

Intel Compiler Evolution
Open-source Components

- The following parts are publicly available
  - Clang Front-End
  - LLVM Middle-End
  - SYCL / DPC++ compiler infrastructure
  - Repository: https://github.com/intel/llvm
  - Intel actively contributes to upstream LLVM and keeps this fork closely aligned with it
- Proprietary pieces are only shipped as binaries
  - Certain vectorization, IPO, and performance heuristics
  - Microarchitecture-specific scheduling and modeling
  - Intel® oneAPI DPC++/C++ Compiler, Intel® Fortran Compiler, Intel® oneAPI Toolkits

Intel Compiler Evolution

- ifort - Intel Fortran Compiler Classic
  - IL0-based compiler (proprietary Front-End, IR, and Optimizer)
  - Deprecated Nov 2023; last public build: 2021.13.1 (included in oneAPI 2024.2.1)
  - Removed from oneAPI 2025.0 packages (late 2024)

- ifx - Intel Fortran Compiler
  - Next-Generation LLVM-Based Compiler
  - Same Fortran Front-End (parser/analyzer) used by ifort
    - semantics and diagnostics feel familiar
    - Full Fortran 2018 support + majority of ifort directives & options
  - Modern LLVM-based backend: LLVM IR + LLVM optimizations + Intel proprietary optimization passes
  - GPU offload via OpenMP to Intel GPUs
  - Microsoft Visual Studio integration* for Windows* development

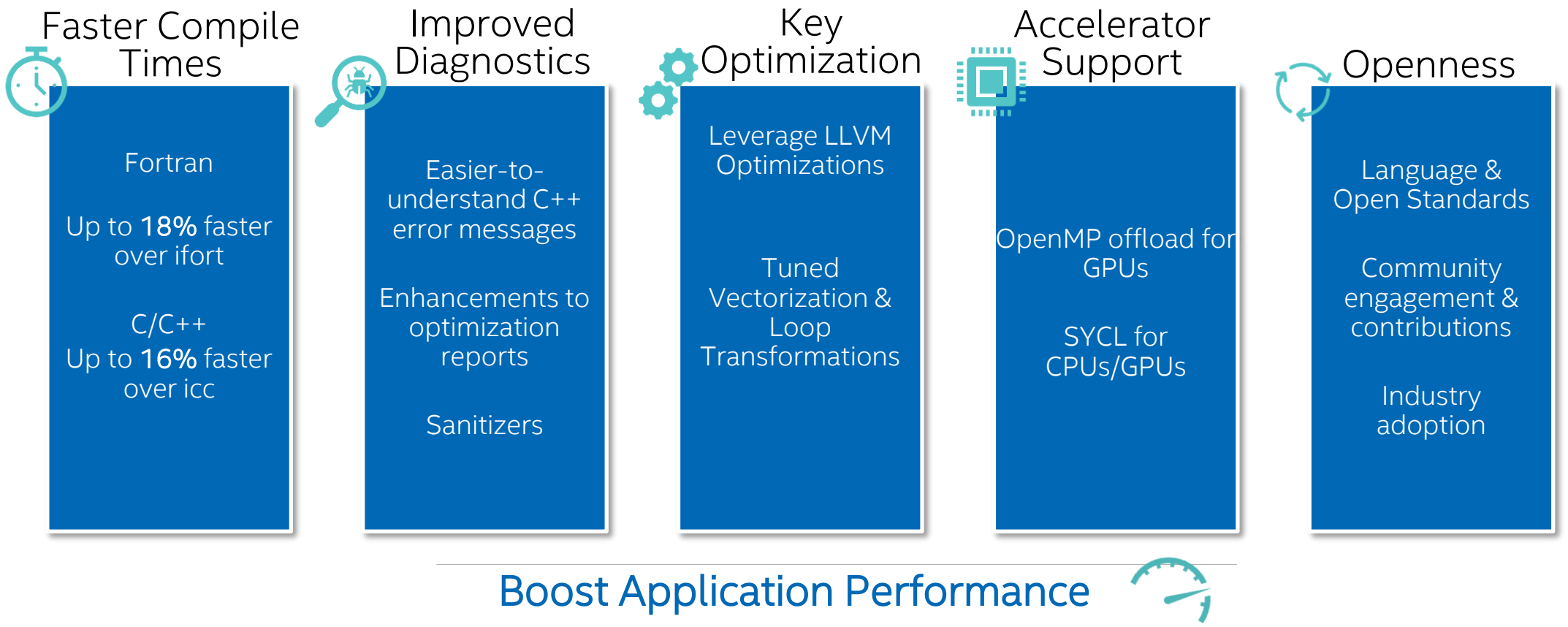
Why Migration to Intel LLVM-based Compilers Matters?

- The Technology Landscape Has Shifted
  - LLVM is the modern standard used by major tech companies
  - Unified ecosystem - Consistent toolchain across platforms and vendors
  - Active development - Continuous innovation and community contributions
  - Modular design enables rapid optimization improvements

- Intel® C++ Compiler Classic (ICC) and Fortran Compiler Classic (IFORT) End-of-Life Reality
  - ICC and IFORT compilers, based on proprietary architecture, are EOL - No future bug fixes
  - Limited Architecture Support - Last supported: Sapphire Rapids (SPR)
  - Performance Risk - No optimization for newer Intel processors beyond SPR

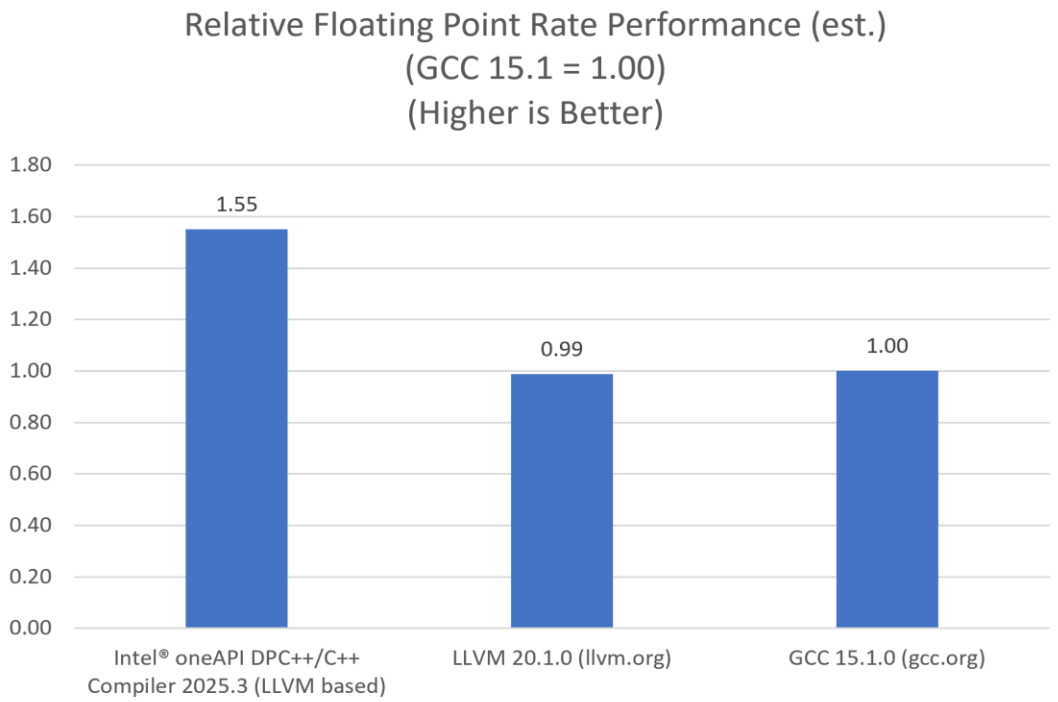
Migrate now to unlock performance potential on modern Intel architectures

Modernizing Intel Compilers with LLVM

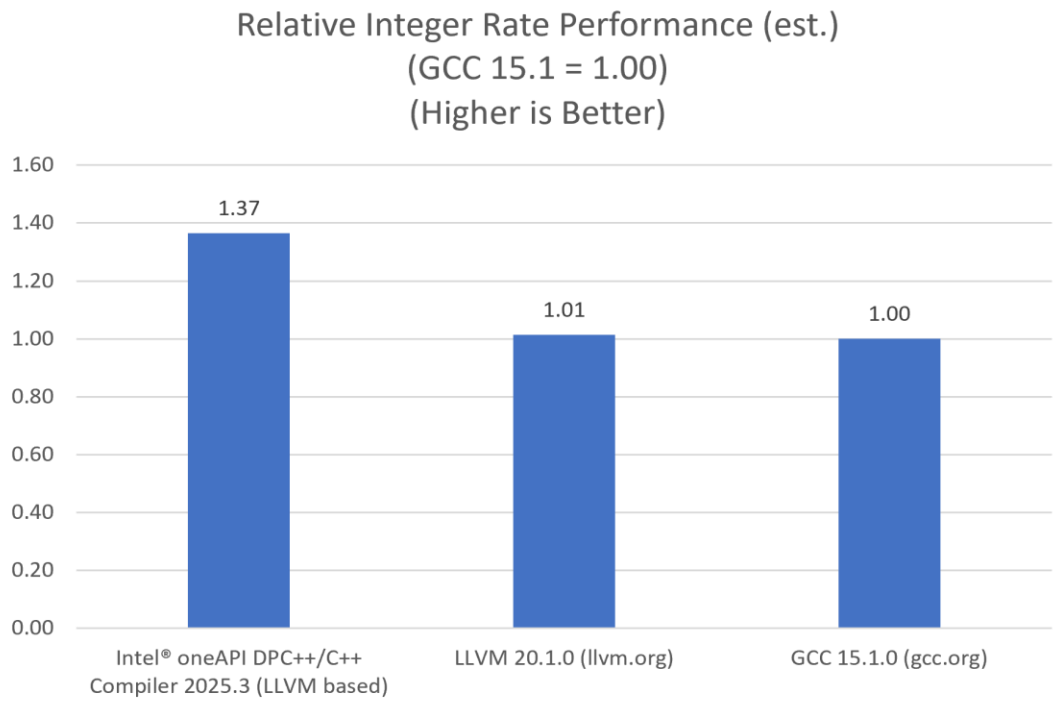


Intel® Compilers Boost C++ Application Performance on Linux*

Performance advantage relative to other compilers on Intel® Xeon® 6980P Processor



Estimated: internal measurement of the geometric mean of the C/C++ workloads from the SPECrate* 2017 Floating Point suite (base tune)



Estimated: internal measurement of the geometric mean of the C/C++ workloads from the SPECrate* 2017 Integer suite (base tune)

Testing Date: Performance results are **based on testing by Intel as of October 10, 2025** and may not reflect all publicly available security updates.

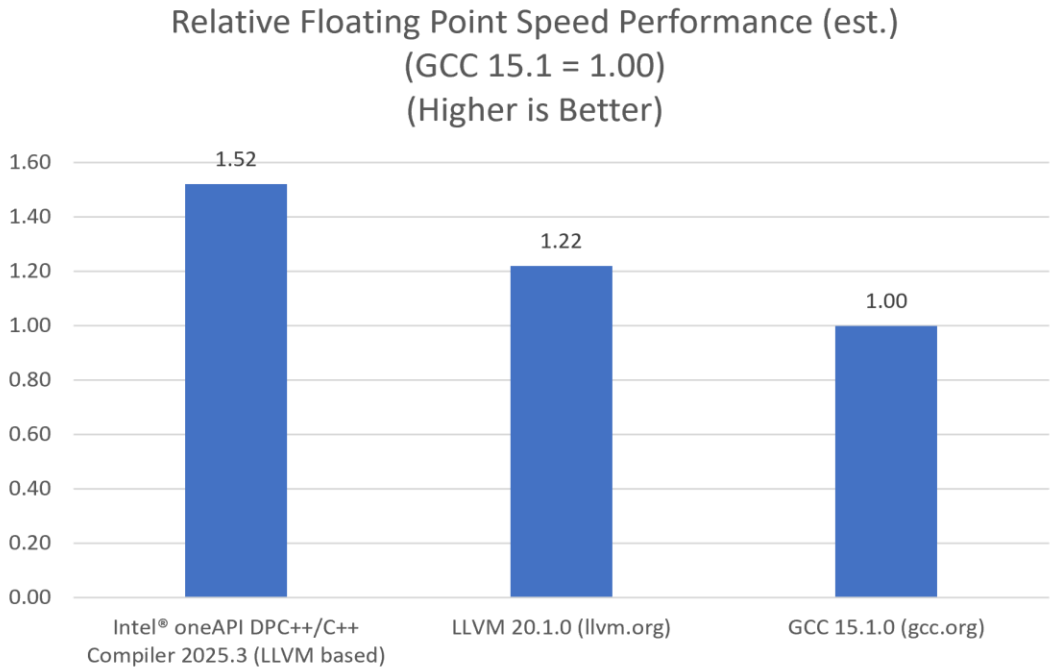
Configuration Details and Workload Setup: Intel® Xeon® 6980P CPU @ 2.0GHz, 2 sockets, Hyper Thread on, Turbo on, 64G x24 DDR5 8800 (1DPC). Software: Intel® oneAPI DPC++/C++ Compiler for applications running on Intel® 64, Version 2025.3.0 Build 20251010; GCC 15.1.0; LLVM 20.1.0. Ubuntu 24.04 LTS, 6.8.0-55-generic. SPECint®_rate_base_2017 compiler switches: Intel® oneAPI DPC++/C++ Compiler: -xgraniterapids -O3 -ffast-math -flto -mfpmath=sse -funroll-loops -qopt-mem-layout-trans=4. GCC: -march=native -mfpmath=sse -Ofast -funroll-loops -flto. LLVM: -march=native -mfpmath=sse -Ofast -funroll-loops –flto. qkmalloc used for Intel compiler. jemalloc 5.0.1 used for gcc and llvm. SPECfp®_rate_base_2017 compiler switches: Intel® oneAPI DPC++/C++ Compiler: -xgraniterapids -mprefer-vector-width=512 –O3 -ffast-math -flto -mfpmath=sse -funroll-loops -qopt-mem-layout-trans=4. GCC: -march=native -mfpmath=sse -Ofast -funroll-loops -flto. LLVM: -march=native -mfpmath=sse -Ofast -funroll-loops -flto. jemalloc 5.0.1 used for Intel compilers, GCC and LLVM.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.

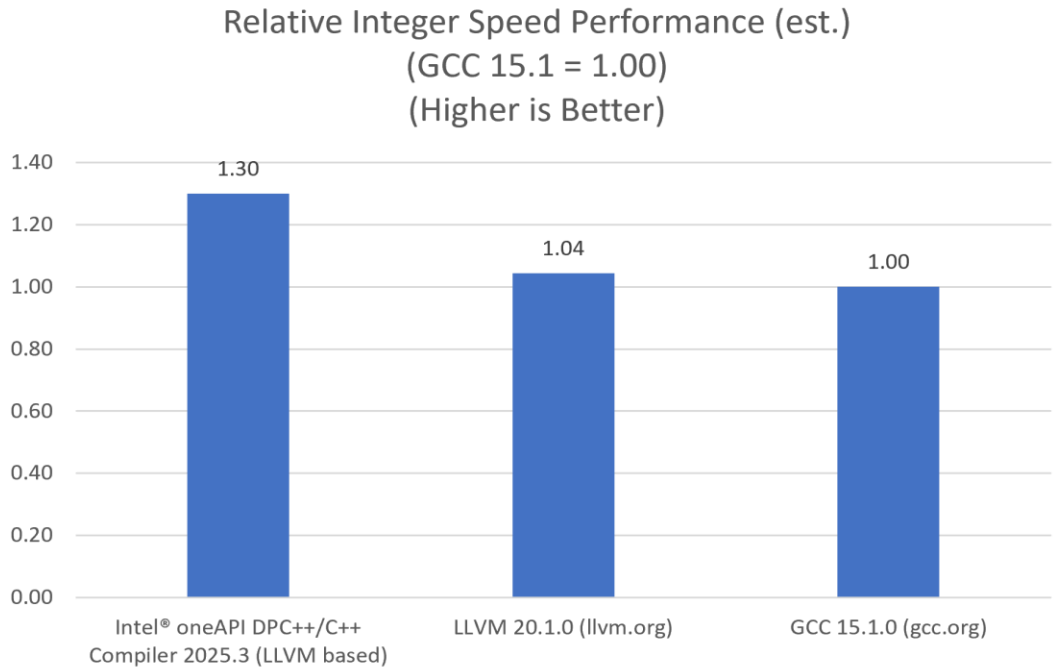
Performance varies by use, configuration, and other factors. Learn more at www.intel.com/PerformanceIndex. More information on the SPEC benchmarks can be found at: http://www.spec.org

Intel® Compilers Boost C++ Application Performance on Linux*

Performance advantage relative to other compilers on Intel® Xeon® 6980P Processor



Estimated: internal measurement of the geometric mean of the C/C++ workloads from the SPECspeed* 2017 Floating Point suite (base tune)



Estimated: internal measurement of the geometric mean of the C/C++ workloads from the SPECspeed* 2017 Integer suite (base tune)

Testing Date: Performance results are **based on testing by Intel as of October 10, 2025** and may not reflect all publicly available security updates.

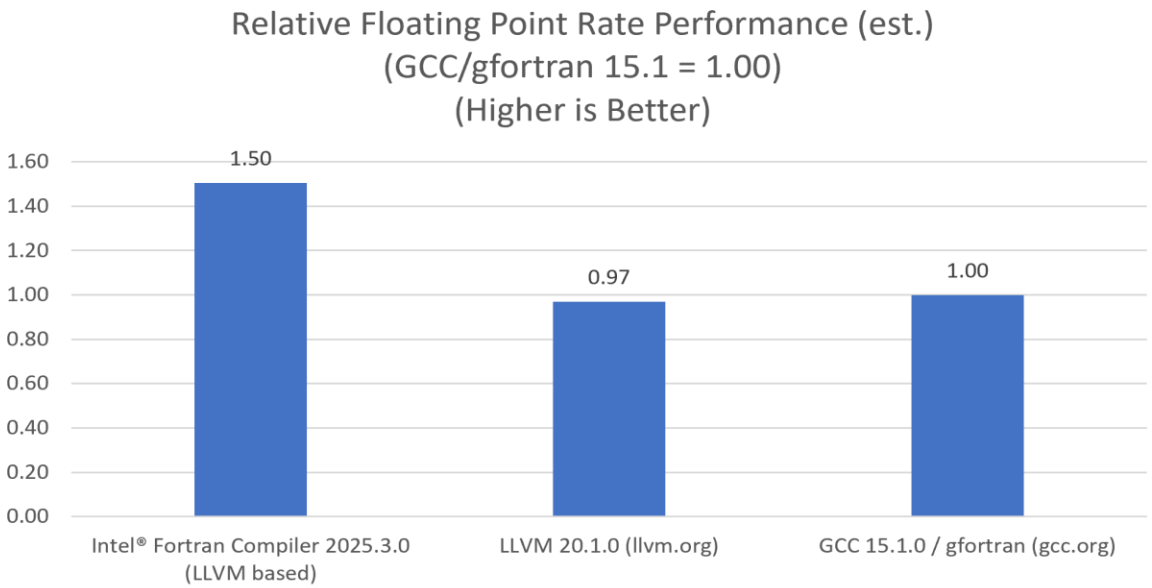
Configuration Details and Workload Setup: Intel® Xeon® 6980P CPU @ 2.0GHz, 2 sockets, Hyper Thread on, Turbo on, 64G x24 DDR5 8800 (1DPC). Software: Intel® oneAPI DPC++/C++ Compiler for applications running on Intel® 64, Version 2025.3.0 Build 20251010; GCC 15.1.0; LLVM 20.1.0. Ubuntu 24.04 LTS, 6.8.0-55-generic. SPECint®_speed_base_2017 compiler switches: Intel® oneAPI DPC++/C++ Compiler: -xgraniterapids -O3 -ffast-math -flto -mfpmath=sse -funroll-loops -qopt-mem-layout-trans=4 -fiopenmp -ljemalloc. GCC: -march=native -mfpmath=sse -Ofast -funroll-loops -flto -fopenmp -ljemalloc. LLVM: -march=native -mfpmath=sse -Ofast -funroll-loops -flto -fopenmp -ljemalloc. jemalloc 5.0.1 used for Intel compilers, gcc and llvm. SPECfp®_speed_base_2017 compiler switches: Intel® oneAPI DPC++/C++ Compiler: -xgraniterapids -O3 -ffast-math -flto -mfpmath=sse -funroll-loops -qopt-mem-layout-trans=4 -fiopenmp -ljemalloc. GCC: -march=native -mfpmath=sse -Ofast -funroll-loops -flto -fopenmp -ljemalloc. LLVM: -march=native -mfpmath=sse -Ofast -funroll-loops -flto -fopenmp -ljemalloc. jemalloc 5.0.1 used for Intel compilers, GCC and LLVM.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.

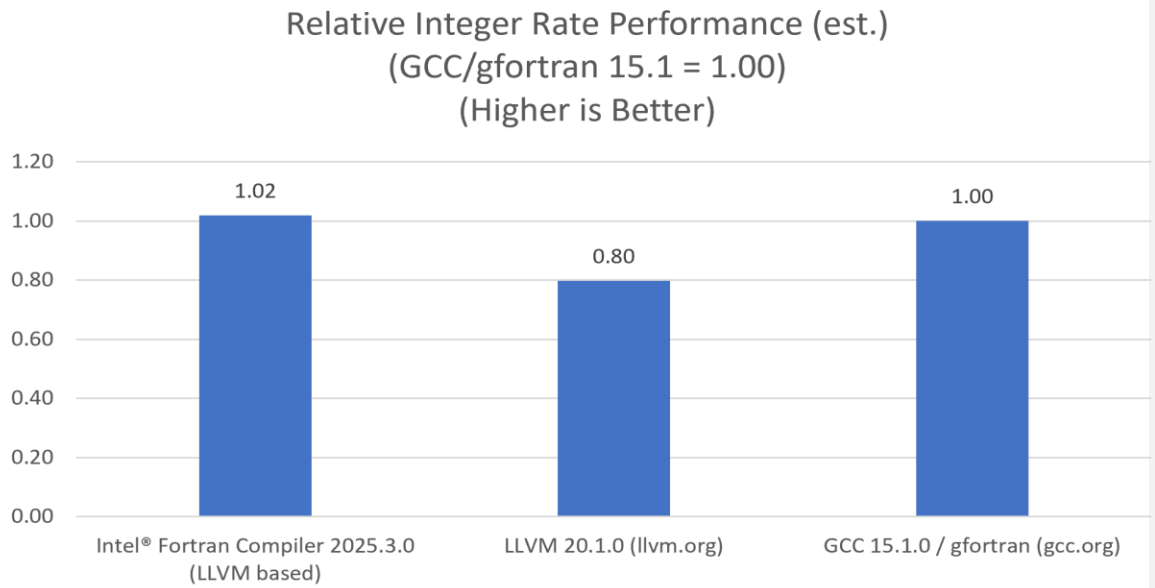
Performance varies by use, configuration, and other factors. Learn more at www.intel.com/PerformanceIndex. More information on the SPEC benchmarks can be found at: http://www.spec.org

Intel® Compilers Boost Fortran Application Performance on Linux*

Performance advantage relative to other compilers on Intel® Xeon® 6980P Processor



Estimated: internal measurement of the geometric mean of the Fortran workloads from the SPECrate* 2017 Floating Point suite (base tune)



Estimated: internal measurement of the geometric mean of the Fortran workloads from the SPECrate* 2017 Integer suite (base tune)

Performance varies by use, configuration, and other factors. Learn more at www.intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.

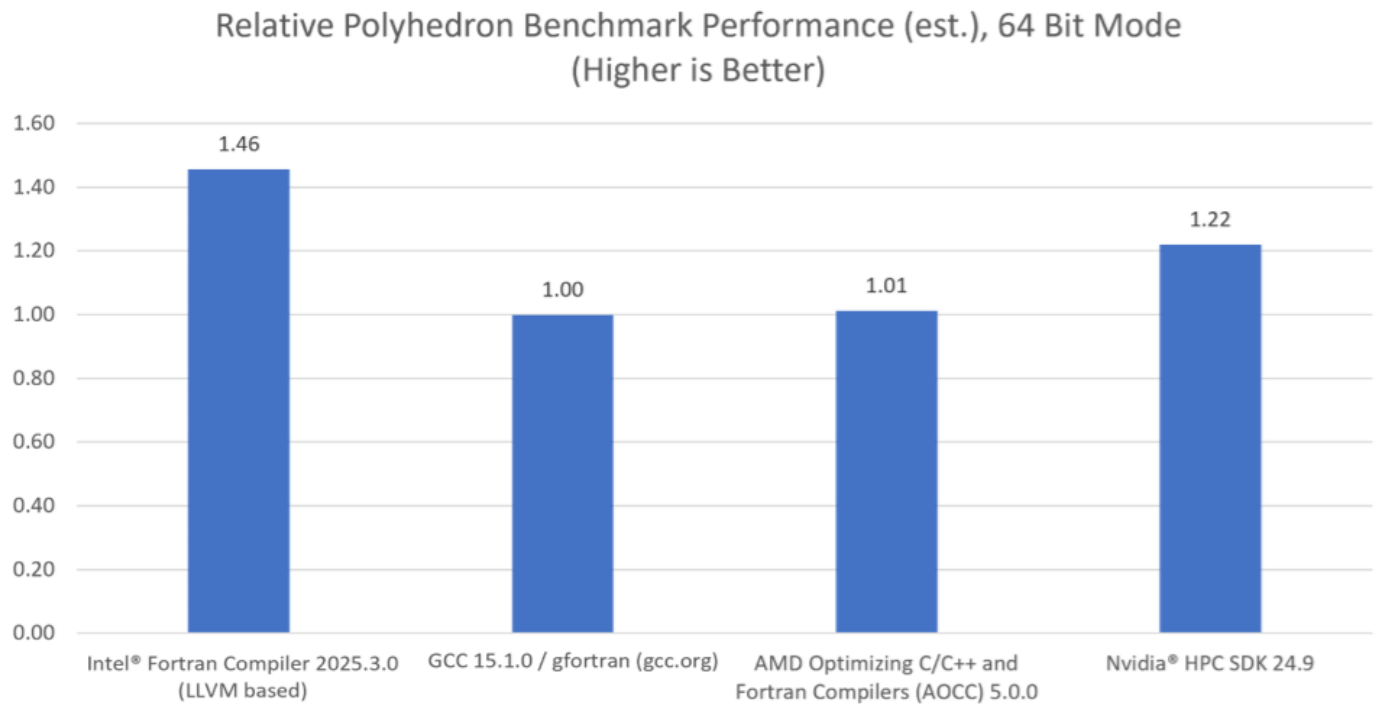
Your costs and results may vary. Intel technologies may require enabled hardware, software, or service activation.

More information on the SPEC benchmarks can be found at: http://www.spec.org

Configuration: Testing by Intel as of Oct 10, 2025. Intel(R) Xeon(R) 6980P CPU @ 2.0GHz, 2 sockets, Hyper Thread on, Turbo on, 64G x24 DDR5 8800 (1DPC). Software: Intel(R) Fortran Compiler for applications running on Intel(R) 64, Version 2025.3.0 Build 20251010; Intel(R) Fortran Intel(R) 64 Compiler Classic for applications running on Intel(R) 64, Version 2021.10.0 Build 20230609_000000; GCC 15.1; LLVM 20.1.0; AMD* Optimizing C/C++ and Fortran Compiler 5.0.0. Ubuntu 24.04 LTS, 6.8.0-55-generic. SPECint®_rate_base_2017 compiler switches: Intel(R) Fortran Compiler: -xgraniterapids -O3 -ffast-math -flto -mfpmath=sse -funroll-loops -qopt-mem-layout-trans=4. Intel(R) Fortran Intel(R) 64 Compiler Classic: -xCORE-AVX512 -ipo -O3 -no-prec-div -qopt-mem-layout-trans=4 -qopt-multiple-gather-scatter-by-shuffles. GCC: -march=native -mfpmath=sse -Ofast -funroll-loops -flto. LLVM: -march=native -mfpmath=sse -Ofast -funroll-loops –flto. AOCC: -O3 -ffast-math -march=znver5 -flto -fstruct-layout=7. qkmalloc used for Intel compiler. jemalloc 5.0.1 used for gcc, amdmalloc used for AOCC. SPECfp®_rate_base_2017 compiler switches: Intel(R) Fortran Compiler: -xgraniterapids -mprefer-vector-width=512 –O3 -ffast-math -flto -mfpmath=sse -funroll-loops -qopt-mem-layout-trans=4. Intel(R) Fortran Intel(R) 64 Compiler Classic: -xCORE-AVX512 -ipo -O3 -no-prec-div -qopt-prefetch -ffinite-math-only -qopt-multiple-gather-scatter-by-shuffles -qopt-mem-layout-trans=4. GCC: -march=native -mfpmath=sse -Ofast -funroll-loops -flto. LLVM: -march=native -mfpmath=sse -Ofast -funroll-loops -flto. AOCC: -O3 -march=znver5 -fveclib=AMDLIBM -ffast-math -flto -fstruct-layout=7. jemalloc 5.0.1 used for Intel compilers and GCC, amdmalloc used for AOCC.

Intel® Fortran Compiler Boosts Application Performance on Linux*

Performance Advantage Measured by Polyhedron* Fortran Benchmark on Intel® Xeon® 6980P Processor



Testing Date: Performance results are **based on testing by Intel as of October 10, 2025** and may not reflect all publicly available security updates.

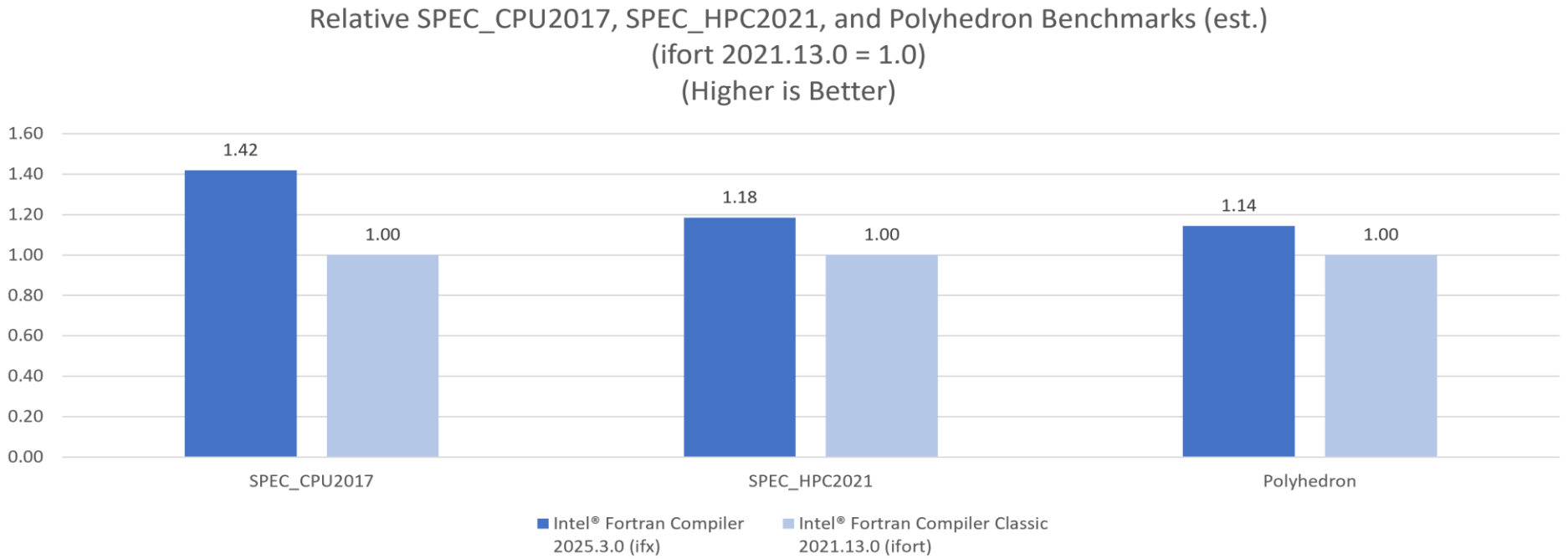
Configuration Details and Workload Setup: Intel® Xeon® 6980P CPU @ 2.0GHz, 2 sockets, Hyper Thread on, Turbo on, 64G x24 DDR5 8800 (1DPC). Software: Intel® Fortran Compiler for applications running on Intel® 64, Version 2025.3.0 Build 20251010; GCC 15.1.0 / gfortran; AMD* Optimizing C/C++ Compiler 5.0.0 / flang - AMD clang version 17.0.6 (CLANG: AOCC_5.0.0-Build#1377 2024_09_24); NVIDIA* HPC SDK 24.9 / nvfortran; Ubuntu 24.04 LTS, 6.8.0-55-generic. Compiler switches: Intel® Fortran Compiler: -Ofast -xCORE-AVX512 -flto -nostandard-realloc-lhs. GCC / gfortran: -Ofast -mfpmath=sse -flto -march=skylake-avx512 -funroll-loops. AMD* Optimizing C/C++ Compiler / flang: -O3 -ffast-math -march=znver5 -fveclib=AMDLIBM -flto -mllvm -unroll-aggressive -mllvm -unroll-threshold=500. NVIDIA* HPC SDK / nvfortran: -fast -Mipa=fast,inline -Mallocatable=03 -Mfprelaxed -Mstack_arrays.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.

Performance varies by use, configuration, and other factors. Learn more at www.intel.com/PerformanceIndex. Your costs and results may vary.

Intel® Fortran Compiler (ifx) & Intel® Fortran Compiler Classic (ifort) Performance on Linux*

Performance Advantage of ifx to ifort on Intel® Xeon® 6980P Processor



Estimated: internal measurement of the geometric mean of each of the SPEC_CPU2017, SPEC_HPC2021, and Polyhedron Benchmarks suites (base tune)

Testing Date: Performance results are **based on testing by Intel as of October 10, 2025** and may not reflect all publicly available security updates.

Configuration Details and Workload Setup: Intel® Xeon® 6980P CPU @ 2.0GHz, 2 sockets, Hyper Thread on, Turbo on, 24x 64G MRDIMM DDR5-8800 (1DPC). Software: Intel® Fortran Compiler for applications running on Intel® 64, Version 2025.3.0 Build 20251009; Intel® Fortran Compiler Classic for applications running on Intel® 64, Version 2021.13.0 Build 20240602_000000. Ubuntu 24.04 LTS, 6.8.0-55-generic. Compiler switches: Intel® Fortran Compiler for SPEC_CPU2017: -xgraniterapids -mprefer-vector-width=512 -Ofast -ffast-math -flto -mfpmath=sse -funroll-loops -qopt-mem-layout-trans=4 -ljemalloc. Intel® Fortran Compiler Classic for SPEC_CPU2017: -xCORE-AVX512 -ipo -O3 -no-prec-div -qopt-prefetch -ffinite-math-only -qopt-multiple-gather-scatter-by-shuffles -qopt-mem-layout-trans=4 -ljemalloc. jemalloc 5.0.1 used for Intel compilers. Intel® Fortran Compiler for SPEC_HPC2021: -Ofast -xgraniterapids -mprefer-vector-width=512 -fiopenmp -flto -qopt-multiple-gather-scatter-by-shuffles -funroll-loops -qopt-streaming-stores=always -nostandard-realloc-lhs -align array64byte. Intel® Fortran Compiler Classic for SPEC_HPC2021: -Ofast -xCORE-AVX512 -qopt-zmm-usage=high -qopt-multiple-gather-scatter-by-shuffles -ansi-alias -qopenmp -ipo. Intel® Fortran Compiler for Polyhedron: -Ofast -xCORE-AVX512 -flto -nostandard-realloc-lhs -qopt-dynamic-align -Qoption,f,"-target-feature -fast-vector-fsqrt" -auto. Intel® Fortran Compiler Classic for Polyhedron: -O3 -xCORE-AVX512 -ipo -nostandard-realloc-lhs -no-prec-div.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.

Performance varies by use, configuration, and other factors. Learn more at www.intel.com/PerformanceIndex. More information on the SPEC benchmarks can be found at: http://www.spec.org

Key Differences

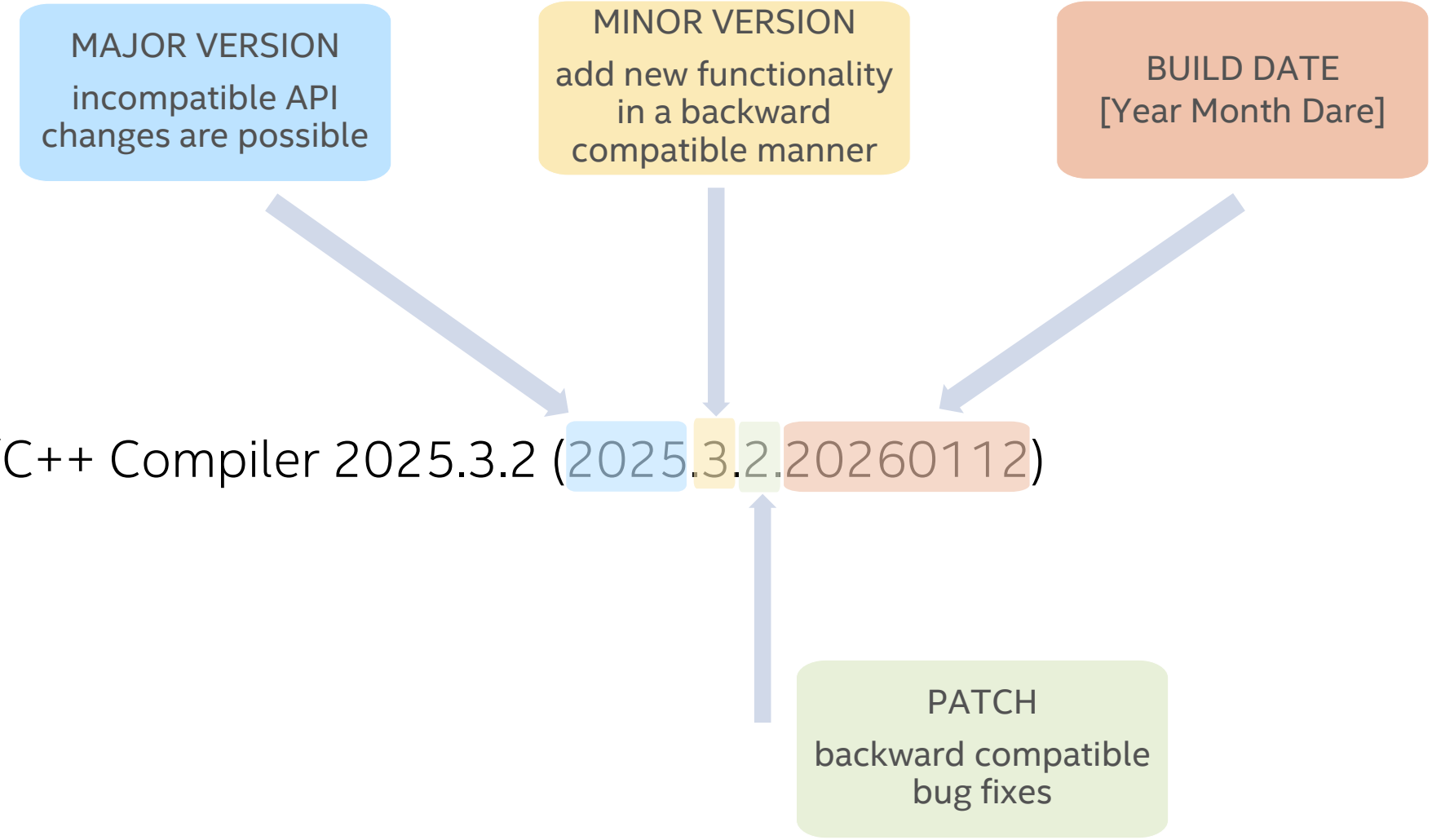
Key Differences Overview

- Different compiler with different optimizations and behavior
  - Optimizer is different
    - IPO - LLVM uses Link Time Optimization (LTO) technology
    - PGO - Profiling is available using the Clang **`-fprofile`** compiler options
    - FP model
    - Loop optimization logic, e.g. loop unrolling, Vecorization
  - Linker behavior
- Not supported features:
  - IA-32 support: **`-m32`**
  - Code Coverage
  - Undocumented options
  - Auto-parallelization (**`-parallel`** option)
  - Intel® Cilk™ Plus (aka simd directive)

Key Differences Overview

- A new versioning macro is defined
  - __INTEL_LLVM_COMPILER
- Binary compatible with the classic compiler
  - Pay attention to the compiler options in use: **`-ipo`**, **`-vecabi`**
- Accept many classic compiler options
  - not all
  - **`-qnextgen-diag`** option emits a list of ifort options that are NOT accepted

Versioning Convention



icx --version

Intel(R) oneAPI DPC++/C++ Compiler 2025.3.2 (2025.3.2.20260112)

icx for C code

icpx for C++ code

Changes in Diagnostics

Changes in Diagnostics

- icc uses **-diag** options to control the display of diagnostic information during compilation.

```
#include <iostream>

int main() {
    int X;
    std::cout << "Hello World!";
    return 0;
}
```

```
$ icpc -diag-error 177 hello.cpp
hello.cpp(4): error #177: variable "X" was declared but never referenced
    int X;
        ^
```

godbolt.org/z/38q8jcan9

Changes in Diagnostics

- ICPX classifies diagnostic messages using descriptive phrases.

```
$ icpx -Wall hello.cpp
hello.cpp:4:9: warning: unused variable 'X' [-Wunused-variable]
    int X;
        ^
1 warning generated.
$ icpx -Werror=unused-variable hello.cpp
hello.cpp:4:9: error: unused variable 'X' [-Werror,-Wunused-variable]
    int X;
        ^
1 error generated.
```

- The clang diagnostics are continuously improving

https://clang.llvm.org/docs/DiagnosticsReference.html https://clang.llvm.org/diagnostics.html

Pragmas support

Do NOT assume ICC or GCC pragmas are supported by ICX!

- Use **`-Wunknown-pragmas`** to check for the unsupported pragmas

```
$ cat unknown-pragmas.c
int main(void) {
   float arr[1000];

   #pragma totallybogus           // pragma; does NOT exist in ICC Classic, GCC, or ICX => causes warning
   #pragma simd                   // ICC Classic pragma; NOT supported by ICX => causes warning
   #pragma vector                 // is recognized and implemented by ICX
   for (int k=0; k<1000; k++) {
      arr[k] = 42.0;
   }
}

$ icx -Wunknown-pragmas unknown-pragmas.c
unknown-pragmas.c:4:9: warning: unknown pragma ignored [-Wunknown-pragmas]
#pragma totallybogus
        ^
unknown-pragmas.c:5:9: warning: unknown pragma ignored [-Wunknown-pragmas]
#pragma simd
        ^
2 warnings generated.
```

godbolt.org/z/4PfYs57dM

Lab 1 - ICC vs ICX Compiler Diagnostics

- **Objective:** Understand the differences between Intel's legacy ICC and modern ICX/ICPX compilers in handling diagnostics and warnings.
- **What You'll Learn:**
  - ICX/ICPX (Clang-based): Uses descriptive flags like -Wunused-variable
  - ICC (Legacy): Uses Intel's -diag options for diagnostic control
  - How to enable, suppress, and control warning levels in both compilers
  - Practical comparison of compiler behavior with real code examples
- **Key Takeaway:** Modern Intel compilers (ICX/ICPX) adopt Clang's more intuitive warning system, making it easier to control diagnostics and ensure compatibility with other LLVM-based toolchains.
- **Time:** ~15-20 minutes

tinyurl.com/workshop-compiler-lab1

Compiler Options for Debugging

How to Debug?

Basic techniques

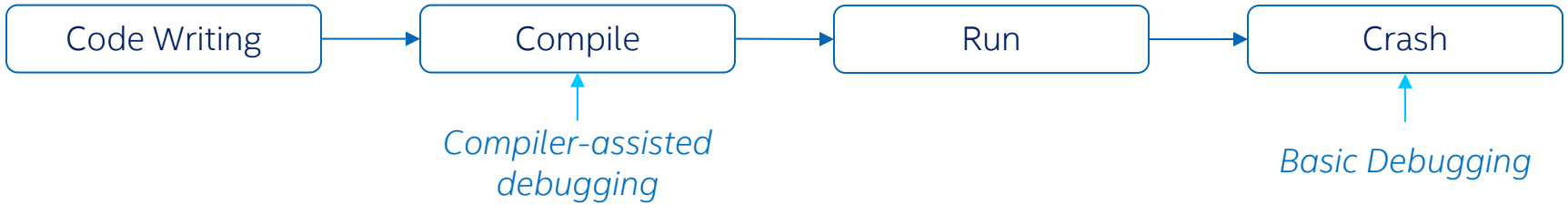
*Reactive - **after** bugs appear*

- Using **print** and carefully checking the code 👀
- Interactive **debugger** (e.g., breakpoints, stepping)
- **Valgrind*** for memory issues
- Require significant manual efforts

Compiler-assisted debugging

*Proactive - **before** and during execution*

- Compile-time diagnostics **(warnings, errors)**
- **Runtime** instrumentation
- **Sanitizers** (memory, UB, address)
- **Debug-friendly builds** (symbols, no/low optimization)
- Often automatic



Compilers don't just build programs - they help find bugs early

Debugging C++ and Fortran Applications
Preparing debug-friendly builds

- **`-g`** (Linux*) - generates debugging information
  - Automatically disables optimizations
    - This behavior may differ in other compilers!
  - Debugging an optimized application (`-O1` and higher) is **not recommended**
    - Optimizations can reorder, inline, or remove code
  - Debugging vs Profiling
    - Learn more: Intel® VTune™ Profiler User Guide : View Data on Inline Functions

Goal	Recommended Flags
Debugging	`-g -O0`
Profiling	`-g -O2` (or chosen level)

Debugging C++ and Fortran Applications
Preparing a Reproducer, Debugging Build Pipeline

- **`-### \ -dryrun`** - shows the driver tool commands but does not execute them
  - Useful when the issue happens between compiler versions – good idea to check the defaults (options, macro definitions, include paths, etc)

```
$ ifx test.f90 -dryrun
   /opt/intel/oneapi/compiler/latest/bin/compiler/xfortcom \
       -triple \
       x86_64-unknown-linux-gnu \
       …
       hello_math.f90
   ld  \
   …
```

- **`-save-temps`** - Tells the compiler to save intermediate files created during compilation
  - Especially useful for SYCL and OpenMP offload cases where the compiler flow is more complex
  - Where to find the files?
    - On Linux: /tmp/ifx*** for Fortran or /tmp/icx***for C++
    - On Windows: C:\Users\<username>\AppData\Local\Temp

Debugging C++ and Fortran Applications
Warnings Levels & Warnings Suppression

- Warnings are proactive diagnostics that can help catch unused variables, type mismatches, suspicious control flow, non-portable or fragile code

Warning Levels:

- **Default:** minimal level of warnings is enabled
- **`-Wall`** - Enables most commonly useful warnings
- **`-Wextra`** - Enables additional, stricter warnings
- **`-Werror`** - Converts warnings into compilation errors. Prevents ignoring warnings, but can be too strict for legacy code

Best Practices

- Enable warnings **early and often**
- Use:
  - Development builds: **`-Wall -Wextra`**
  - Code cleanup, refactoring: **`-Wextra`**
  - CI / release builds: **`-Werror`**
- **Suppress** warnings only when **justified**
- Review **warning changes** across **compiler versions**

Warnings Suppression:

- Disable a specific warning:
  - `warning: using the result of an assignment as a condition without parentheses [-Wparentheses]`
  - Can be disabled with `-Wno-parentheses` ()
- Disable ALL warnings: **`-w`**
- Legacy ICC option **`-diag-disable`** is not supported

Compiler warnings are the first and cheapest debugging tool

Debugging Fortran Applications
Fortran-specific Warning Messages

- **-warn** - Specifies diagnostic messages to be issued by the compiler.
  - **Default**: alignments, general, nodeclarations, noerrors, noexternals, noignore_bounds, nointerfaces, noshape, nostderrors, notruncated_source, nounused, uncalled, usage

```
$ cat warn.f90
 program warn
    implicit none
    real :: afunc, b
    integer :: abc   !<- not used variable - we will display this warning with -warn unused
    afunc(b) = 123*b !<- not used function - we will suppress this warning with -warn nouncalled

 end program warn

$ ifx warn.f90
warn.f90(3): remark #7713: This statement function has not been used.   [AFUNC]
    real :: afunc, b
------------^

$ ifx  warn.f90 -warn unused -warn nouncalled
warn.f90(4): remark #7712: This variable has not been used.   [ABC]
  integer :: abc      !<- variable not used - show warning #4
-------------^
```

godbolt.org/z/TYxfYeaez

Debugging Fortran Applications
Fortran Language Options

- **-stand** - Tells the compiler to issue compile-time messages for nonstandard language elements.
  - **Default**: nostand - The compiler issues no messages for nonstandard language elements

```
$ cat standard-violation.f90
module m
  implicit none
  type :: t
  end type
contains
  pure subroutine sub(x)
    class(t), allocatable, intent(out) :: x
    allocate(x)
  end subroutine
end module
```

```
$ ifx standard-violation.f90  -c
standard-violation.f90(21): warning #9000: An
INTENT(OUT) dummy argument of a pure subroutine must
not be polymorphic or have a polymorphic allocatable
ultimate component.   [X]
  pure subroutine sub(x)
----------------------^

$ ifx standard-violation.f90  -c  -stand f18
standard-violation.f90(21): error #9001: Fortran 2018
requires that an INTENT(OUT) dummy argument of a pure
subroutine must not be polymorphic or have a
polymorphic allocatable ultimate component.   [X]
  pure subroutine sub(x)
----------------------^
```

-stand does not change program behavior - it changes how strictly the compiler applies the language rules.

godbolt.org/z/Ee5xo98qo

Debugging Fortran Applications
Fortran Language Options

- **`-standard-semantics`**
  - determines whether the current Fortran Standard behavior of the compiler is fully implemented
  - enables all of the options that implement the current Fortran 2018 Standard if the standard level is not specified with -stand option
- **`-assume [keyword]`**
  - Controls semantic assumptions the compiler makes about Fortran program behavior. The assumptions affect language semantics, correctness diagnostics, and the safety of optimizations.
  - Default: a list of options, check documentation – performance is the priority, and historical compatibility
  - E.g., **`-assume [no]realloc_lhs`** determines whether the compiler uses the current Fortran Standard rules or the old Fortran 2003 rules when interpreting assignment statements

Debugging Fortran Applications
Fortran Language Options

- Additional *assume* options are set by option *standard-semantics* when a *stand* option specifies a standard level (from IFX 2025.0):

Option assume setting	Option standard-semantics	Options stand:f90 and stand:f95	Option stand:f03	Option stand:f08	Option stand:f18	Option stand:f23
byterecl	Y	Y	Y	Y	Y	Y
failed_images	Y				Y	Y
fpe_summary	Y	Y	Y	Y	Y	Y
ieee_compares	Y				Y	Y
ieee_mode_restore	Y		Y	Y	Y	Y
minus0	Y		Y	Y	Y	Y
noold_e0g0_format	Y				Y	Y
noold_inquire_recl	Y				Y	Y
noold_ldout_format	Y	Y	Y	Y	Y	Y
noold_ldout_zero	Y		Y	Y	Y	Y
noold_maxminloc	Y		Y	Y	Y	Y
noold_unit_star	Y	Y	Y	Y	Y	Y
noold_xor	Y	Y	Y	Y	Y	Y
protect_parens	Y	Y	Y	Y	Y	Y
realloc_lhs	Y		Y	Y	Y	Y
recursion	Y				Y	Y
std_expon_output	Y					Y
std_intent_in	Y		Y	Y	Y	Y
std_minus0_rounding	Y		Y	Y	Y	Y
std_mod_proc_name	Y	Y	Y	Y	Y	Y
std_value	Y			Y	Y	Y

Debugging Fortran Applications
Runtime Checking - Example

Commonly used **-assume** options:

- **-assume protect_parens** - Preserves parentheses exactly as written for safer floating-point evaluation
- **-assume nostd_intent_in / std_intent_in** – controls enforcement of INTENT(IN) semantics
- **-assume noold_unit_star** - Disables legacy READ(*,...) and WRITE(*,...) behavior
- **-assume byterecl** - Interprets record length units as bytes (legacy compatibility)

```
$ cat assume_realloc_lhs.f90
program assume_realloc_lhs
   implicit none
   integer, allocatable :: x(:)
   allocate( x(2) )
   print *, "Before assignment x(2): shape(x) = ", shape(x)
   x = [ 1, 2, 3 ]
   print *, "After assignment [1,2,3]: shape(x) = ", shape(x)
end program assume_realloc_lhs

$ ifx -assume norealloc_lhs assume_realloc_lhs.f90 && ./a.out
 Before assignment: shape(x) =           2
 After assignment: shape(x)  =           2
$ ifx -assume realloc_lhs assume_realloc_lhs.f90 && ./a.out
 Before assignment: shape(x) =           2
 After assignment: shape(x)  =           3
```

Debugging Fortran Applications
Getting Source-Level Call Stack

- **-traceback** - Generates extra information in the object file to provide source file traceback information.
  - **Default:** notraceback. The compiler does not generate extra information in the object file

```
$ cat fpe.f90
     (1) program fpe
     (2)   implicit none
     (3)   real :: a, b     <-- b is not initialized
     (4)   a = b / 0.0      <-- "-traceback -g" will give you the line number
     (5)   print *, a
     (6) end program fpe

$ ifx fpe.f90 && ./a.out

            NaN

$ ifx fpe.f90 -ftrapuv -traceback -g -O0 && ./a.out
forrtl: error (182): floating invalid - possible uninitialized real/complex variable.
Image              PC                Routine            Line        Source
libc.so.6          0000130398245320  Unknown            Unknown     Unknown
a.out              0000000000405272  initsnan           4           fpe.f90
```

godbolt.org/z/PTcv1c5YK

Debugging Fortran Applications
Floating Point Exceptions

- **-fpe** - Allows some control over floating-point exception handling for the main program at runtime.
  - **Default:** -fpe3 - All floating-point exceptions are disabled!

```
$ cat fpe.f90
     (1) program fpe
     (2)   implicit none
     (3)    real :: a, b
     (4)    b = 3.0
     (5)    a = b / 0.0
     (6)    print *, a
     (7) end program fpe

$ ifx fpe.f90 && ./a.out

            Infinity

$ ifx fpe.f90 -fpe0 -traceback -g -O0 && ./a.out
forrtl: error (73): floating divide by zero
Image              PC                Routine            Line        Source
libc.so.6          0000057EDB645320  Unknown               Unknown  Unknown
a.out              000000000040526B  fpe                       5  fpe.f90
a.out              000000000040522D  Unknown               Unknown  Unknown
..
```

godbolt.org/z/o16fnxT3j

Debugging Fortran Applications
Catching Uninitialized Variables

- **`-ftrapuv`** - Initializes stack local variables to an unusual value to aid error detection. This is a **runtime** check.
- **`-init=snan`** - Determines whether the compiler initializes to signaling NaN all uninitialized variables of intrinsic type REAL or COMPLEX that are saved, local, automatic, or allocated variables.
  - Disables **`-fpe3`**
  - **Default**: disabled, the application will not crash if you are using an uninitialized variable

Debugging Fortran Applications
Catching Uninitialized Variables

```
$ cat fpe.f90
     (1) program fpe
     (2)   implicit none
     (3)   real :: a, b
     (4)   a = b / 0.0
     (5)   print *, a
     (6) end program fpe

$ ifx fpe.f90 -fpe0 && ./a.out

            0.0000000E+00

$ ifx fpe.f90 -init=snan -traceback -g -O0 && ./a.out
forrtl: error (182): floating invalid - possible uninitialized real/complex variable.
Image              PC                Routine            Line        Source
libc.so.6          00000CC7A6245320  Unknown               Unknown  Unknown
a.out              0000000000405272  fpe                       4  fpe.f90
..
```

godbolt.org/z/v6jqhqz8K

Debugging Fortran Applications
Heap

- **-heap-arrays** - Puts automatic arrays and arrays created for temporary computations on the heap instead of the stack.

```
$ cat heap_arrays.f90
    program heap_arrays
     implicit none
     call myfoo(5000000)
     contains
       subroutine myfoo(n)
          integer, intent(in) :: n
          integer, dimension(n) :: a, b
          a = 123
          b = 456
          a = 2 * a + b
          print *, sum(a)
       end subroutine myfoo
    end program heap_arrays
$ ifx heap_arrays.f90
$ ./a.out
forrtl: severe (174): SIGSEGV, segmentation fault occurred
Image              PC                Routine            Line        Source
libpthread-2.31.s  00007F65079AF3C0  Unknown               Unknown  Unknown
a.out              00000000004051F0  Unknown               Unknown  Unknown
...
```

```
$ ifx heap_arrays.f90 -heap-arrays
$ ./a.out
 -1486967296
```

godbolt.org/z/GnbE7cb7P

Debugging Fortran Applications
Runtime Checking

- **`-check[keyword]`** - Enables runtime correctness checks in the Intel® Fortran compiler and turns undefined behavior into actionable runtime errors.
  - **Default**: nocheck
- When do these checks trigger?
  - Array index violations - **`bounds`**
  - Uninitialized pointers - **`pointers`**
  - Uninitialized stack- or heap-allocated variables – **`uninit`** (Linux only! Host only! Must be used in compilation AND linking!)
  - And more
  - Note: **`stack`** is not supported by IFX. Use address sanitizer instead.
  - Use **`all`** to enable all checks. This option **disables optimization** and **overrides** any **optimization level** set by option O
- Errors are **reported at the point of failure**, not later
- For **meaningful diagnostics**, combine with: **`-traceback -g`**
- **Performance note: `-check`** options add runtime overhead

Full list of available keywords and description

Debugging Fortran Applications
Runtime Checking - Example

```
 $ cat check_demo.f90
(1)  program check_demo
(2)      implicit none
(3)
(4)      integer :: i
(5)      integer :: j
(6)      integer :: a(5)
(7)      integer, pointer :: p
(8)
(9)      do i = 1, size(a) + 1
(10)        a(i) = 1
(11)     end do
(12)
(13)     print *, "Value of j:", j
(14)
(15)     p = 10
(16)
(17) end program check_demo
```

```
$ ifx check_demo.f90 && ./a.out

Value of j:           0
forrtl: severe (174): SIGSEGV, segmentation fault occurred
Image              PC                Routine            Line        Source
libc.so.6          00000CD05B245320  Unknown               Unknown  Unknown
a.out              0000000000405301  Unknown               Unknown  Unknown

$ ifx check_ demo.f90 -traceback -g -check bounds && ./a.out
forrtl: severe (408): fort: (2): Subscript #1 of the array A has value 6 which is
greater than the upper bound of 5

Image              PC                Routine            Line        Source
a.out              0000000000405342  check_demo                10  check_demo.f90
a.out              000000000040522D  Unknown               Unknown  Unknown
libc.so.6          000014BBEE22A1CA  Unknown               Unknown  Unknown
libc.so.6          000014BBEE22A28B  __libc_start_main     Unknown  Unknown
a.out              0000000000405115  Unknown               Unknown  Unknown
```

There are more issues, but even with **`-check all`**, they will pop up one by one
This example is equivalent to **`-check bounds`**

godbolt.org/z/WWY71xoWj

Debugging Fortran Applications
Runtime Checking - Example

```
 $ cat check_demo.f90
(1)  program check_demo
(2)      implicit none
(3)
(4)      integer :: i
(5)      integer :: j
(6)      integer :: a(5)
(7)      integer, pointer :: p
(8)
(9)      do i = 1, size(a) ! + 1
(10)        a(i) = 1
(11)     end do
(12)
(13)     print *, "Value of j:", j
(14)
(15)     p = 10
(16)
(17)  end program check_demo
```

```
$ ifx check_demo.f90 && ./a.out

 Value of j:           0
forrtl: severe (174): SIGSEGV, segmentation fault occurred
Image              PC                Routine            Line        Source
libc.so.6          00000568C4E45320  Unknown               Unknown  Unknown
a.out              00000000004052F1  Unknown               Unknown  Unknown

$ ifx check_ demo.f90 -traceback -g -check all && ./a.out
 Value of j:           0
forrtl: severe (408): fort: (7): Attempt to use pointer P when it is not associated
with a target

Image              PC                Routine            Line        Source
a.out              00000000004053F0  check_demo                15  check_demo.f90
a.out              000000000040522D  Unknown               Unknown  Unknown
libc.so.6          000011ED80A2A1CA  Unknown               Unknown  Unknown
libc.so.6          000011ED80A2A28B  __libc_start_main     Unknown  Unknown
a.out              0000000000405115  Unknown               Unknown  Unknown
```

There are more issues, but even with **`-check all`**, they will pop up one by one
This example is equivalent to **`-check pointer`**

godbolt.org/z/WWY71xoWj

Debugging Fortran Applications
Runtime Checking - Example

```
  $ cat check_demo.f90
(1)   program check_demo
(2)       implicit none
(3)
(4)       integer :: i
(5)       integer :: j
(6)       integer :: a(5)
(7)       integer, pointer :: p
(8)
(9)       do i = 1, size(a) ! + 1
(10)         a(i) = 1
(11)       end do
(12)
(13)       print *, "Value of j:", j
(14)
(15)      ! p = 10
(16)
(17)  end program check_demo
```

```
$ ifx check_demo.f90 && ./a.out

 Value of i:            6
forrtl: severe (174): SIGSEGV, segmentation fault occurred
Image              PC                Routine            Line        Source
libc.so.6          00000568C4E45320  Unknown               Unknown  Unknown
a.out              00000000004052F1  Unknown               Unknown  Unknown

$ ifx check_ demo.f90 -traceback -g -check all && ./a.out

Value of j:            0

$ ifx check_ demo.f90 -traceback -g -check all,uninit && ./a.out
 ==1668947==WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x000000537c6c in for__format_value (/home/user/a.out+0x537c6c) (BuildId: ..
    #1 0x0000004fa8f0 in for_write_seq_lis_xmit (/home/user/a.out+0x4fa8f0) ..
    #2 0x0000004dc0a8 in MAIN /home/user/check_demo.f90:13:3
…
  Uninitialized value was stored to memory at
    #0 0x0000004dc04e in MAIN /home/user/check_demo.f90:13:3
…
Uninitialized value was created by an allocation of 'check_demo_$J' in the stack frame
    #0 0x0000004dbbc5 in MAIN /home/user/check_demo.f90:1
```

This example is equivalent to **-fsanitize=memory**

LLVM Sanitizers Unleashed: Empowering Developers to Build Robust Applications

Address Sanitizer (ASan)

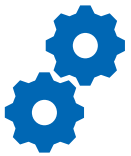
What it does – ASan detects memory errors like buffer overflows and use-after-free issues by monitoring memory access

Developer benefit – ASan helps developers find and fix memory-related bugs early, preventing crashes and vulnerabilities

Memory Sanitizer (MSan)

What it does – MSan finds uses of uninitialized memory, which can lead to unpredicted behavior.

Developer benefit – MSan helps identify uninitialized memory, reducing bugs and potential security vulnerabilities.



Thread Sanitizer (TSan) + Archer for OpenMP

What it does -TSan identifies data races and synchronization issues in multi-threaded code. Archer on top of TSan detects OpenMP races.

Developer benefit – TSan + Archer helps developers write thread-safe code, reducing hard-to-debug concurrency problems.

Undefined Behavior (UBSan)

What it does – UBSan detects and reports undefined behavior in code during program execution.

Developer benefit – UBSan helps developers improve code reliability, stability and reducing the risk of unexpected runtime issues.

Leak Sanitizer (LSan)

What it does – LSan detects runtime memory leaks, improving resource management.

Developer Benefit – LSan helps developers find and fix memory leaks early, improving code stability and preventing memory-related issues.

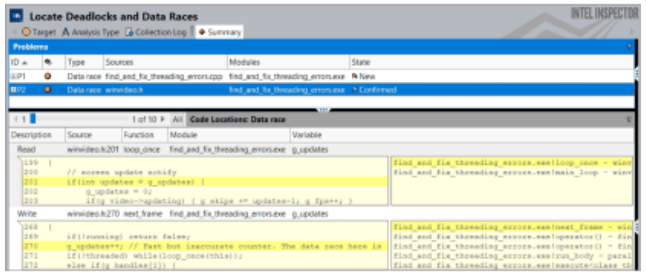
Numerical Stability Sanitizer (NSAN)

What it does – NSan detects floating point accuracy issues

Developer Benefit – NSan helps developers find and fix floating point issues in real-life applications, improving reliability and reducing developers' efforts on debugging

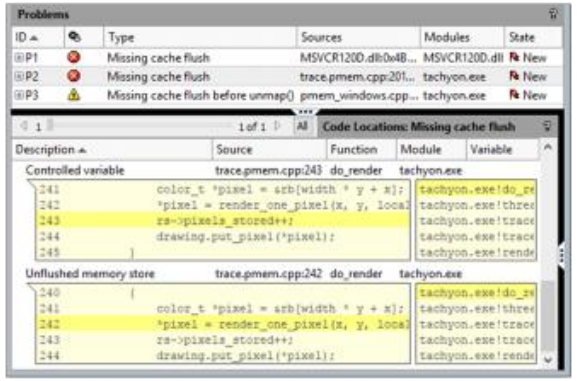
Each of these sanitizers is like a helpful assistant for developers, catching specific types of bugs and issues early in the development process. They make code more reliable, secure, and easier to maintain by preventing crashes, vulnerabilities, and hard-to-debug problems.

Transition from Intel Inspector to Sanitizers



Locate Nondeterministic Threading Errors: deadlocks and data races

Thread Sanitizer (TSan) + Archer for OpenMP



Detect Hard-to-Find Memory Errors: memory leaks, corruption, mismatched allocation and deallocation API, inconsistent use of memory API, illegal memory access, and uninitialized memory read

Address Sanitizer (ASan), Memory Sanitizer (MSan), Undefined Behavior (UBSan) , Leak Sanitizer (LSan)

Intel Compilers Support for LLVM Sanitizers

Enhancing Code Quality and Reliability using LLVM Sanitizers in Intel® oneAPI DPC++/C++ Compiler (ICX) and Intel® Fortran Compiler (IFX)

	Support	Memory (MSan)	Leak (LSan)	Thread (TSan) + Archer for OpenMP	Address (ASan)	Undefined Behavior	Numerical Stability
Linux	ICX	√	√	√	√	√	√
	Device Code: SYCL, C++ OpenMP Offload	√ (device USM only)		√	√		
	IFX	√		√	√ + leak detection		
	Device Code: Fortran OpenMP Offload				√		
Windows	ICX				√	√	
	IFX				√		

More Reliable and Secure Applications

Intel Compilers Support for LLVM Sanitizers

- Supported by Intel® oneAPI DPC++/C++ Compiler and intel/llvm open-source compiler
- Supported by Intel® Fortran Compiler
- enable sanitizers with **`-fsanitize=<sanitizer_name>`**
- **`-g`** and **`-fno-omit-frame-pointer`** for meaningful output
- Compiler and link in one command:
  - icpx -g -fsanitize=address test.cpp
- Two-step solution:
  - Compile
    - icpx -g -fsanitize=address -fno-omit-frame-pointer -c test.cpp
  - Link
    - icpx -g -fsanitize=address example_test.o

AddressSanitizer – Use After Free

```
$ icpx -g -fsanitize=address -fno-omit-frame-pointer example_UseAfterFree.cpp
$ ./a.out
```

```
int main(int argc, char **argv) {
 int *array = new int[100];
 delete [] array;
 return array[argc];
}
```

```
==710593==ERROR: AddressSanitizer: heap-use-after-free on address 0x514000000044 at pc
0x00000050c926 bp 0x7ffe2dc12310 sp 0x7ffe2dc12308
READ of size 4 at 0x514000000044 thread T0
    #0 0x50c925 in main /home/example_UseAfterFree.cpp:4:9
    #1 0x7f88f2bd2d8f in __libc_start_call_main csu/../sysdeps/nptl/libc_start_call_main.h:58:16
    #2 0x7f88f2bd2e3f in __libc_start_main csu/../csu/libc-start.c:392:3
    #3 0x4294ad in _start /home/a.out+0x4294ad)

0x514000000044 is located 4 bytes inside of 400-byte region [0x514000000040,0x5140000001d0)
freed by thread T0 here:
    #0 0x50aed1 in operator delete[](void*) (/home/a.out+0x50aed1)
    #1 0x50c8f0 in main /home/example_UseAfterFree.cpp:3:2
    #2 0x7f88f2bd2d8f in __libc_start_call_main csu/../sysdeps/nptl/libc_start_call_main.h:58:16

previously allocated by thread T0 here:
    #0 0x50a671 in operator new[](unsigned long) (/home/a.out+0x50a671)
    #1 0x50c8e5 in main /home/example_UseAfterFree.cpp:2:15
    #2 0x7f88f2bd2d8f in __libc_start_call_main csu/../sysdeps/nptl/libc_start_call_main.h:58:16

SUMMARY: AddressSanitizer: heap-use-after-free /home/example_UseAfterFree.cpp:4:9 in main
```

clang.llvm.org/docs/AddressSanitizer.html#introduction

godbolt.org/z/zMezccxTj

Lab 2 - Debugging with Intel Fortran Compiler (IFX)

- **Objective:** Master IFX debugging options to catch common Fortran programming errors at compile-time and runtime.
- **What You'll Learn:**
  - Warning control (-warn options) - Catch unused variables and dead code
  - Standard conformance (-stand options) - Ensure portable, standards-compliant code
  - Automatic reallocation (-assume options) - F2003+ vs F95 behavior
  - FP exception handling - Catch division by zero, overflow, invalid operations
  - Uninitialized variables (-ftrapuv, -init=snan) - Detect use-before-initialization bugs
  - Runtime checks (-check bounds, -check all) - Array bounds and comprehensive validation
- **Key Takeaway:** IFX provides powerful compile-time and runtime checks that catch bugs early in development.
- **Time:** ~30-45 minutes

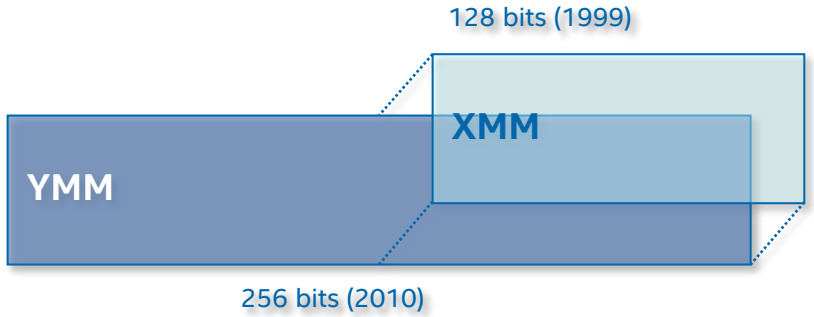
tinyurl.com/workshop-compiler-lab2

Part II - Optimizations

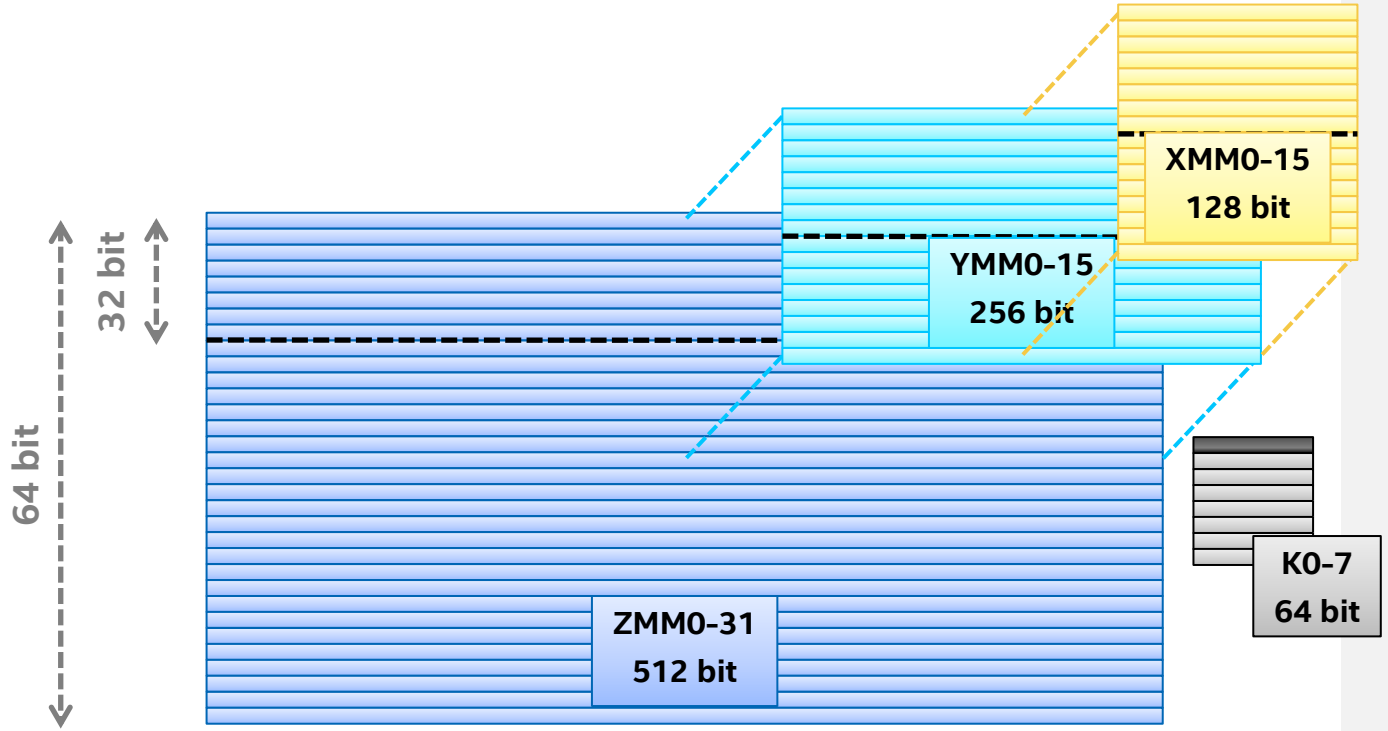
Vectorization

Intel® AVX and AVX-512 Registers

AVX is a 256 bit vector extension to SSE:



AVX-512 extends previous AVX and SSE registers to 512 bit:



OS support is required

Auto-vectorization

```
#include <math.h>
void foo (float * theta, float * sth)  {
  int i;
  for (i = 0; i < 512; i++)
       sth[i] = sin(theta[i]+3.1415927);
}
```

```
vcvtps2pd       ymm0, xmmword ptr [r14 + 4*rsi + 16]
vaddpd  ymm0, ymm8, ymm0
call    rdi
vcvtpd2ps       xmm0, ymm0
vmovupd xmmword ptr [rbx + 4*rsi + 16], xmm0
```

```
vmovss  xmm0, dword ptr [r14 + 4*r15]
vcvtss2sd       xmm0, xmm0, xmm0
vaddsd  xmm0, xmm0, qword ptr [rip + .LCPI0_0]
call    sin@PLT
vcvtsd2ss       xmm0, xmm0, xmm0
vmovss  dword ptr [rbx + 4*r15], xmm0
```

godbolt.org/z/6nc7Eqcda

intel.com/content/www/us/en/developer/articles/technical/vectorization-llvm-gcc-cpus-gpus.html

Many Ways to Vectorize

Compiler:
Auto-vectorization (no change of code)

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

Compiler:
OpenMP* SIMD directives

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 Options for Intel CPUs

-x<code>

- Might enable Intel processor specific optimizations
- Processor-check added to “main” routine
  Application errors in case the SIMD feature is missing or a non-Intel processor with an appropriate/informative message
- <code> indicates a feature set that the compiler may target (including instruction sets and optimizations)
  - Microarchitecture code names: ICELAKE, SAPPHIRERAPIDS, GRANITERAPIDS
  - SIMD extensions: CORE-AVX512, CORE-AVX2, CORE-AVX-I, AVX, SSE4.2, etc.
  - Example: *icx -xCORE-AVX512 test.c*

    icx -xGRANITERAPIDS test.c

-ax<code>

- Multiple code paths: baseline and optimized/processor-specific
- Optimized code paths for Intel processors defined by <code>

Basic Vectorization Options

`-m<code>`

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

`-march=processor`

- Generate code using the CPU feature set of a specific processor as the baseline, e.g. sapphirerapids, graniterapids
- Non-Intel CPUs are also supported, e.g. **`-march=znver5`** for AMD® Turin

`-xHost`

- Generate instructions for the highest instruction set available on the compilation host processor

Optimization Report

- **`-qopt-report[=n]`** tells the compiler to generate an optimization report

  n: (Optional) Indicates the level of detail in the report. Level 5 produced the greatest level of detail with icc.

  icx has n=3 as a max level and includes Loop Optimizations, OpenMP parallelization and Register Allocation messages

  Generates a file called *.optrpt containing the optimization report messages.

- **`-qopt-report-phase[=list]`** specifies one or more optimizer phases for which optimization reports are generated.

  cg (Code Generation), ipo (Interprocedural Optimization), loop (Loop Nest Optimization), openmp (OpenMP*,) pgo (Profile Guided Optimization), vec (Vectorization), all (All optimizer phases)

Intel® AVX-512 Generation for GNR

Compile with processor-specific option to target instruction set:

-xCORE-AVX512

Or optimizes for the specified Intel® processor microarchitecture code name:

-xGRANITERAPIDS

```
void foo(double *a, double *b, int size) {
  #pragma omp simd
  for(int i=0; i<size; i++) {
     b[i]=exp(a[i]);
  }
}
```

```
icpx -c -xGRANITERAPIDS -qopenmp -qopt-report=3 foo.cpp
```

```
 remark #15569: Compiler has chosen to target XMM/YMM
vector. Try using -mprefer-vector-width=512 to
override.
```

By default, it will not optimize for more restrained ZMM register usage which works best for certain applications

-qopt-zmm-usage=low|high or **-mprefer-vector-width** controls ZMM registers usage

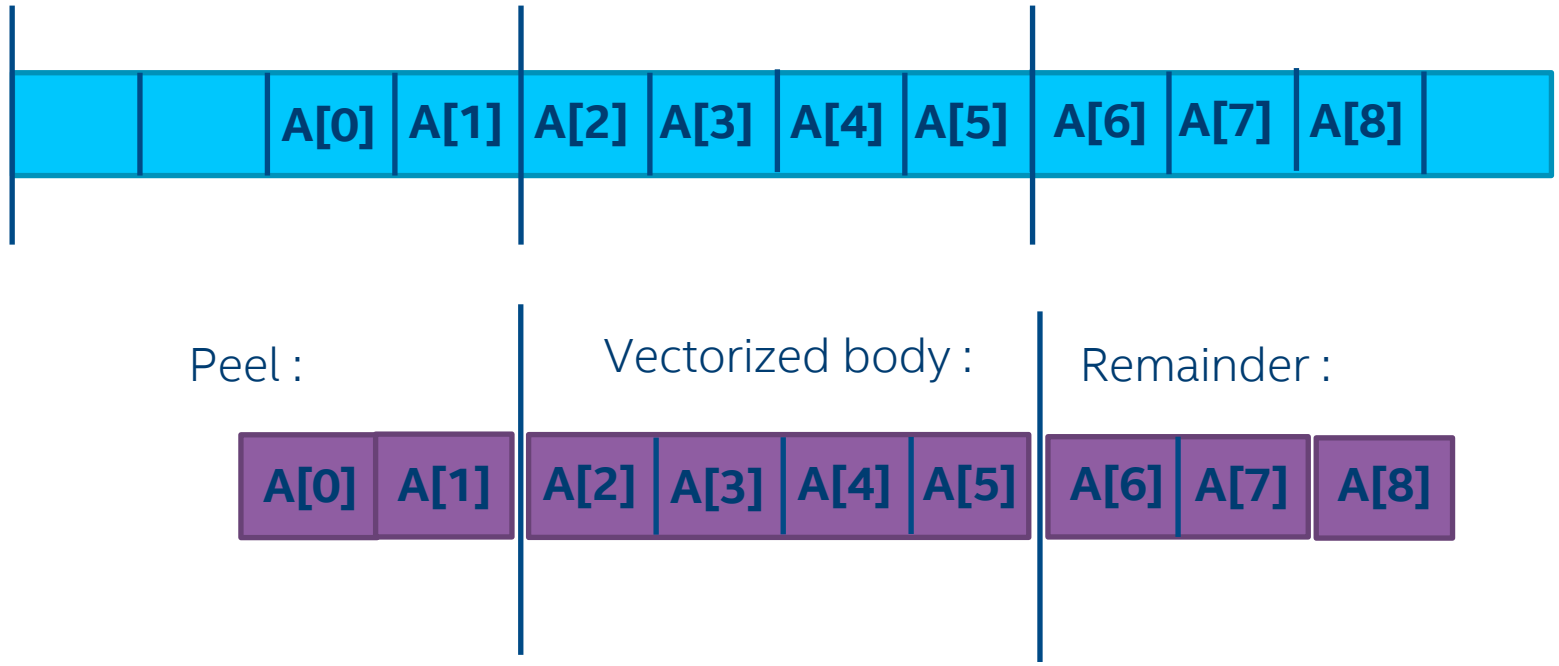
AVX10: Advanced Vector Extensions

The converged vector ISA for all Intel CPUs

- It is a unified SIMD ISA definition that:
  - Covers 128, 256, and 512-bit vector widths
  - Uses the same EVEX encoding and opmask model
  - Works across CPUs that may or may not physically support 512-bit execution
  - AVX10 decouples vector semantics from maximum hardware width.
- Standardize one SIMD ISA. Width becomes an implementation choice.
- AVX 10.1 - Baseline AVX10 instruction set (support in GNR)
- AVX10.2 - Superset of AVX10.1 with new compute features (DMR)
- Enabled via **`-mavx10.1-256`**, **`-mavx10.1-512`**

Compiler Helps with Alignment

SSE:	16 bytes
AVX:	32 bytes
AVX512	64 bytes



Compiler can split loop in 3 parts to have aligned access in the loop body

Optimization Report

Optimization Report – An Example

$ icpx -O2 -xcore-avx512 -qopt-report=3 example.cpp

Creates example.optrpt summarizing which optimizations the compiler performed or tried to perform.
Level of detail from 0 (no report) to 3 (maximum).
Extracts:

```
LOOP BEGIN at example.cpp (4, 3)
<Multiversioned v1>
   remark #25228: Loop multiversioned for Data Dependence
   remark #15300: LOOP WAS VECTORIZED
   remark #15305: vectorization support: vector length 4
   remark #15475: --- begin vector loop cost summary ---
  ....  (loop cost summary)  ....
LOOP END

LOOP BEGIN at <source> (4, 3)
<Multiversioned v2>
   remark #15615: Loop was not vectorized: not vectorizable due to data dependence, fall-back loop for
multiversioning
LOOP END
```

```
#include <math.h>
void foo (float * theta, float * sth)  {
  int i;
  for (i = 0; i < 512; i++)
       sth[i] = sin(theta[i]+3.1415927);
}
```

godbolt.org/z/KsjGMPsG3

Optimization Report – An Example

$ icpx -O2 -xcore-avx512 -qopt-report=3 -qopt-report-file=stdout -qopt-report-file=stderr **-fargument-noalias** example.cpp

report to stderr instead of example.optrpt

```
...
LOOP BEGIN at example.cpp (4, 3)
   remark #15300: LOOP WAS VECTORIZED
   remark #15305: vectorization support: vector length 4
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar cost: 48.000000
   remark #15477: vector cost: 12.062500
   remark #15478: estimated potential speedup: 3.968750
...
 EXTERN: __svml_sin4 <source> (5,17)
```

```
#include <math.h>
void foo (float * theta, float * sth)  {
  int i;
  for (i = 0; i < 512; i++)
       sth[i] = sin(theta[i]+3.1415927);
}
```

godbolt.org/z/esKa4bdv5

Optimization Report – An Example

$ icpx -O2 -xcore-avx512 -qopt-report=3 -qopt-report-file=stdout -qopt-report-file=stderr **-fargument-noalias** example.cpp

...

LOOP BEGIN at example.cpp (4, 3)

remark #15300: LOOP WAS VECTORIZED

remark #15305: vectorization support: vector length 8

remark #15475: --- begin vector loop cost summary ---

remark #15476: scalar cost: 31.000000

remark #15477: vector cost: 4.203125

remark #15478: estimated potential speedup: 7.343750

...

EXTERN: __svml_sinf8 <source> (5,17)

```
#include <math.h>
void foo (float * theta, float * sth)  {
  int i;
  for (i = 0; i < 512; i++)
       sth[i] = sinf(theta[i]+3.1415927f);
}
```

godbolt.org/z/nY4YcEqvM

Difference in Vectorization
ICC vs ICX

- Vectorizer behavior is different with -x option
  - E.g. always generate unaligned instructions; clang behavior with –m option
- Vectorization of loops with omp simd
- Cost model
- Loop unrolling
- vecabi

Lab 3 - Vectorization with Intel Compilers – Part I

- **Objective:** Master Intel compiler vectorization features to maximize performance on latest Intel® architectures
- **What You'll Learn:**
  - Loop multiversioning & optimization report - Analyze vectorization decisions and eliminate overhead with -fargument-noalias
  - Architecture-specific flags (-m, -x, -ax) - Target specific CPUs or create portable multi-version binaries with runtime dispatch
  - ICC vs ICX compiler strategies - use OpenMP SIMD pragmas for explicit control
- **Key Takeaway:** Intel compilers provide powerful auto-vectorization with optimization reports to identify bottlenecks. OpenMP SIMD pragmas give explicit control when needed, achieving significant performance gains
- **Time:** 45 minutes

tinyurl.com/workshop-compiler-lab3

User-Mandated Vectorization

Obstacles to Auto-Vectorization

- Multiple loop exits
  - Or trip count unknown at loop entry
- Dependencies between loop iterations
  - Mostly, avoid read-after-write “flow” dependencies
- Function or subroutine calls
  - Except where inlined
- Nested (Outer) loops
  - Unless inner loop fully unrolled
- Complexity
  - Too many branches
  - Too hard or time-consuming for compiler to analyze

OpenMP* SIMD Programming

Vectorization is so important
➔ consider explicit vector programming

Modeled on OpenMP* for threading (explicit parallel programming)

Enables reliable vectorization of complex loops the compiler can't auto-vectorize
E.g. outer loops

Directives are commands to the compiler, not hints
#pragma omp simd

Compiler does no dependency and cost-benefit analysis !!
Programmer is responsible for correctness (like OpenMP threading)
E.g. PRIVATE, REDUCTION or ORDERED clauses

Incorporated in OpenMP since version 4.0 ⇒ portable
`-qopenmp` or **`-qopenmp-simd`** to enable

OpenMP* SIMD pragma

- Use #pragma omp simd (ICX requires **-qopenmp-simd** or **-qopenmp** option)
  - ICC has `-qopenmp-simd` set by default

```
void addit(double* a, double* b, int m,
           int n, int x)
{
  for (int i = m; i < m+n; i++) {
        a[i] = b[i] + a[i-x];
  }
}
```

```
void addit(double* a, double* b, int m,
           int n, int x)
{
#pragma omp simd  // I know x<0
  for (int i = m; i < m+n; i++) {
        a[i] = b[i] + a[i-x];
  }
}
```

- Loop was multiversioned with icc; assumed dependency with icx
  - With more pointers, the compiler will not multiversion
- Use SIMD pragma when you **KNOW** that a given loop is safe to vectorize

  The Intel® Compiler will vectorize if at all possible
  - (ignoring dependency or efficiency concerns)
  - Minimizes source code changes needed to enforce vectorization

godbolt.org/z/x8TxG56Gr

OpenMP* SIMD pragma

Vectorization Example with ICC vs ICX

```
#pragma omp simd
  for (std::size_t i = 1; i < n2; ++i)
    {
      const std::size_t jd0 = std::min(ii[i], ii[i + 1] - 1);
      ys[i] = R(jd0, jd0);
    }
}
```

- ICC vectorizes this loop:
  LOOP BEGIN at <source>(22,3)
    remark #15301: SIMD LOOP WAS VECTORIZED
- ICX 2025.1 doesn't
  warning: loop not vectorized: the optimizer was unable to perform the requested transformation; the transformation might be disabled or specified as part of an unsupported transformation ordering [-Wpass-failed=transform-warning]
- ICX 2025.3.1 does
  LOOP BEGIN at <source> (21, 1)
  remark #15301: SIMD LOOP WAS VECTORIZED

Use latest ICX version

godbolt.org/z/dc4e77noY

Clauses for OMP SIMD directives

The programmer (i.e. you!) is responsible for correctness

- Just like for race conditions in loops with OpenMP* threading

Available clauses:

- PRIVATE
- LASTPRIVATE
- REDUCTION

} like OpenMP for threading

- COLLAPSE (for nested loops)
- LINEAR (additional induction variables)
- SIMDLEN (preferred number of iterations to execute concurrently)
- SAFELEN (max iterations that can be executed concurrently)
- ALIGNED (tells compiler about data alignment)

Example: Outer Loop Vectorization

```
#ifdef  KNOWN_TRIP_COUNT
#define MYDIM 3
#else                              // pt      input  vector of points
#define MYDIM nd                   // ptref   input  reference point
#endif                             // dis     output vector of distances
#include <math.h>

void dist( int n, int nd, float pt[][MYDIM], float dis[], float ptref[]) {
/* calculate distance from data points to reference point */

#pragma omp simd
           for (int ipt=0; ipt<n; ipt++) {
                      float d = 0.;

                      for (int j=0; j<MYDIM; j++) {
                                 float t = pt[ipt][j] - ptref[j];
                                 d+= t*t;
                      }

                      dis[ipt] = sqrtf(d);
           }
}
```

Outer loop with high trip count

Inner loop with low trip count

godbolt.org/z/bx3WWaWcq

Outer Loop Vectorization

-xcore-avx512 -qopt-report-phase=loop,vec -qopt-report-file=stderr

```
...
LOOP BEGIN at <source>(10,2)
   ...
  remark #15541: loop was not vectorized: outer loop is not an auto-vectorization candidate.
...
  LOOP BEGIN at <source>(13,3)
     remark #15300: LOOP WAS VECTORIZED
```

- We can vectorize the outer loop by adding the pragma

```
#pragma omp simd
```

- Would need private clause for d and t if declared outside SIMD scope

```
...
LOOP BEGIN at <source> (10, 1)
    remark #15301: SIMD LOOP WAS VECTORIZED
    remark #15305: vectorization support: vector length 16
...
   LOOP BEGIN at <source>(14,3)
      remark #15548: loop was vectorized along with the outer loop
   LOOP END
LOOP END
```

godbolt.org/z/bx3WWaWcq

Unrolling the Inner Loop

- If the trip count is fixed and the compiler knows it, the inner loop can be fully unrolled. Outer loop vectorization is also more efficient because the stride is now known

```
-xcore-avx512 -qopt-report-phase=loop,vec -qopt-report-file=stderr -DKNOWN_TRIP_COUNT

LOOP BEGIN at <source>(10,1)
    remark #15301: SIMD LOOP WAS VECTORIZED

   LOOP BEGIN at <source>(14,3)
      remark #25436: Loop completely unrolled by 3
   LOOP END
LOOP END
```

godbolt.org/z/WqjfW54zK

Loops Containing Function Calls

Function calls can have side effects that introduce a loop-carried dependency, preventing vectorization

Possible remedies:

- Inlining
  - best for small functions
  - Must be in same source file, or else use `-ipo`
- Vector versions of math functions (SVML)
- SIMD-enabled functions
  - Good for large, complex functions and in contexts where inlining is difficult
  - Call from regular “for” loop

SIMD-enabled Function

- The compiler generates **multiple ABI-compatible "SIMD variants" (function multiversions)**
  - ICX follows the OpenMP SIMD Function ABI
  - Scalar version func(float, float, ...) - non-SIMD code
  - Vector versions (no mask) _ZGVb**N4**vuuuuu_func, _ZGVc**N8**vuuuuu_func, ... - Full SIMD iteration
  - Vector versions (masked) _ZGVb**M4**vuuuuu_func, _ZGVc**M8**vuuuuu_func - Remainder loops

- #pragma omp simd may not be needed in simpler cases

```
#pragma omp declare simd uniform(y,z,xp,yp,zp)
float func(float x, float y, float z, float xp, float yp, float zp)
{
float denom = (x-xp)*(x-xp) + (y-yp)*(y-yp) + (z-zp)*(z-zp);
  denom = 1./sqrtf(denom);
  return denom;
}
…
#pragma omp simd  private(x)  reduction(+:sumx)
for (i=1; i<nx; i++) {
    x = x0 + (float) i * h;
    sumx = sumx + func(x, y, z, xp, yp, zp);
  }
```

y, z, xp, yp and zp are constant, x can be a vector

These clauses are required for correctness, just like for OpenMP*

godbolt.org/z/Y3bn9h8Wb

Clauses for SIMD-enabled Functions

#pragma omp declare simd

- UNIFORM — argument is never a vector
- LINEAR (REF|VAL|UVAL) — additional induction variables use REF(X) when the vector argument is passed by reference (Fortran default)
- INBRANCH / NOTINBRANCH — specify whether the function will be called conditionally
- SIMDLEN — vector length
- ALIGNED — asserts that the listed variables are aligned
- PROCESSOR(cpu) — Intel ICC extension, tells compiler which processor to target, e.g. haswell, skylake_avx512; ICX doesn't support it

- ICX: Target selection belongs to the compiler invocation, not the source code.
- Simpler is to target the processor specified by **`-x`** switch using **`-vecabi=cmdtarget`**

godbolt.org/z/KjhTPWqna

Special Idioms

Compress Loop Pattern
Auto-vectorization

```
int compress(float *a, float * b, int na)
{
    int nb = 0;
    for (int ia=0; ia <na; ia++)
    {
        if (a[ia] > 0.f)
            b[nb++] = a[ia];
    }

    return nb;
}
```

VCOMPRESSPD|PS|D|Q	Store sparse packed floating-point values into dense memory
VEXPANDPD|PS|D|Q	Load sparse packed floating-point values from dense memory

double/single-precision/doubleword/quadword

- Loop-carried dependency
  - The write index depends on how many previous elements passed the test.
- AVX-512 introduces compress-store instruction `vcompressps ymmword ptr [rsi + 4*rax] {k1}, ymm1`
  - load 16 floats
  - compare
  - mask
  - compress-store to b
  - count mask bits to update nb

godbolt.org/z/Txsnvefjo

Compress Loop Pattern
Auto-vectorization

Targeting Intel® AVX2

-xCORE-AVX2 -O2 -qopt-report-file=stderr -qopt-report-phase=vec -qopt-report=3 -fargument-noalias

LOOP BEGIN

remark #15344: Loop was not vectorized: vector dependence prevents vectorization

remark #15346: vector dependence: assumed FLOW dependence between nb [<source> (7, 9)] and nb [<source> (7, 5)]

remark #15346: vector dependence: assumed FLOW dependence between nb [<source> (7, 9)] and nb [<source> (7, 9)]

...

LOOP END

Targeting Intel® AVX-512

-xCORE-AVX512 -O2 -qopt-report-file=stderr -qopt-report-phase=vec-qopt-report=3 -fargument-noalias

LOOP BEGIN

remark #15300: LOOP WAS VECTORIZED

remark #15305: vectorization support: vector length 16

...

LOOP END

Compress/Expand loop pattern doesn't vectorize on architectures like Intel® AVX2 and the previous ones and does with Intel® AVX-512

godbolt.org/z/7cbGqd418

Compress Loop Pattern

Complex example

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

godbolt.org/z/KsEf54Mre

Compress Loop Pattern

Complex example

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

1. Outer loop vectorization can be achieved using OpenMP SIMD pragma.
2. The Compress/Expand loop pattern can hinted to compiler using monotonic clause.
3. The ordered clause takes into account the nb dependency (if omitted, wrong results)

godbolt.org/z/GY8rr1WGd

Compress Loop Performance

Optimization Options	Speed-up	What's going on
Simple Loops (-O2 -xCORE-AVX2)	1.0	No vectorization
(-O2 -xCORE-AVX512)	~12x	Auto-vectorized
Nested Loops (-O2 –xCORE-AVX512)	1.0	Loop was auto-vectorized as ordered
Monotonic (-O2 –xCORE-AVX512)	~4x	vcompress instruction used

1. Ordered clause will serialize the execution of the compress logic
2. Monotonic clause hints the compiler on the specific loop pattern and helps code generation in picking vcompress/vexpand vector instruction which leads to better performance.

Lab 3 - Vectorization with Intel Compilers – Part II

- **Objective:** Master Intel compiler vectorization features to maximize performance on latest Intel® architectures
- **What You'll Learn:**
  - Outer loop vectorization - Process multiple iterations simultaneously with known trip counts for optimal performance
  - Performance benchmarking - Measure real-world speedups across optimization levels
  - Special idioms (compress pattern) - Leverage AVX-512 instructions like vcompresspd for data-dependent operations
- **Key Takeaway:** Intel compilers provide powerful auto-vectorization with optimization reports to identify bottlenecks. OpenMP SIMD pragmas give explicit control when needed, achieving significant performance gains
- **Time:** 45 minutes

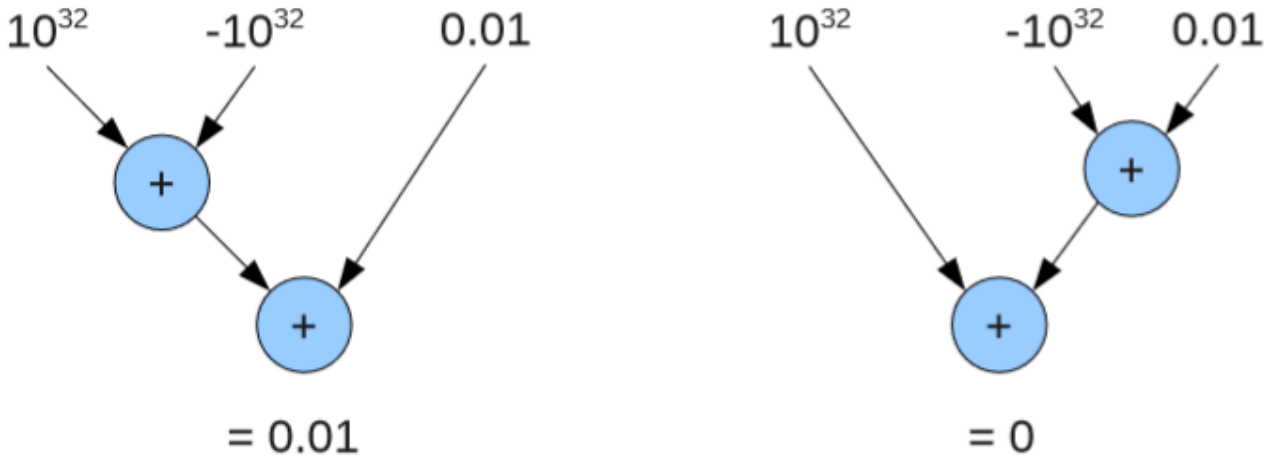
tinyurl.com/workshop-compiler-lab3

Floating Point Model

Numerical Reproducibility

Basics to Remember

- Floating point numbers are not associative:
  - (A + B) + C ≠ A + (B + C)

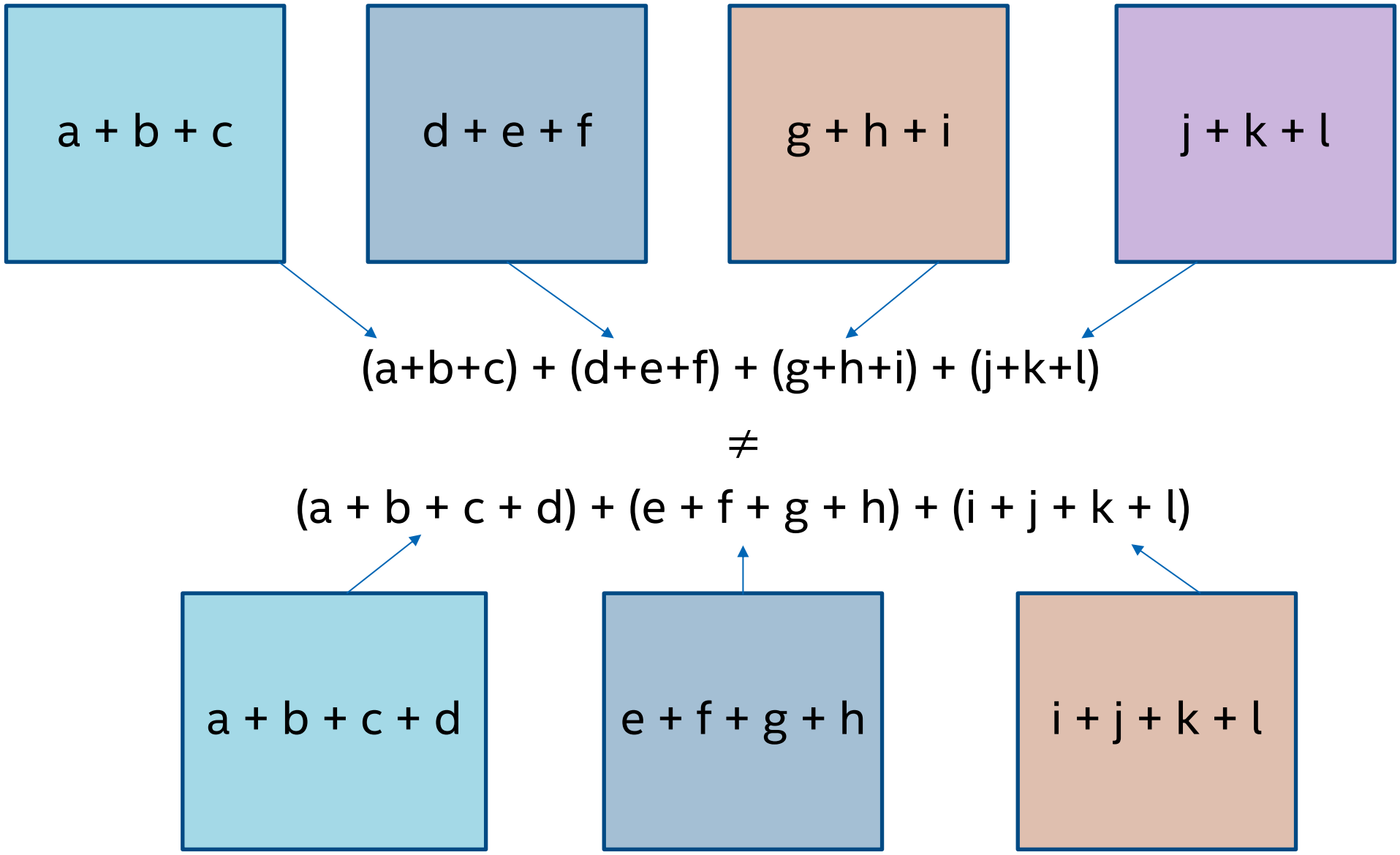


Scenarios

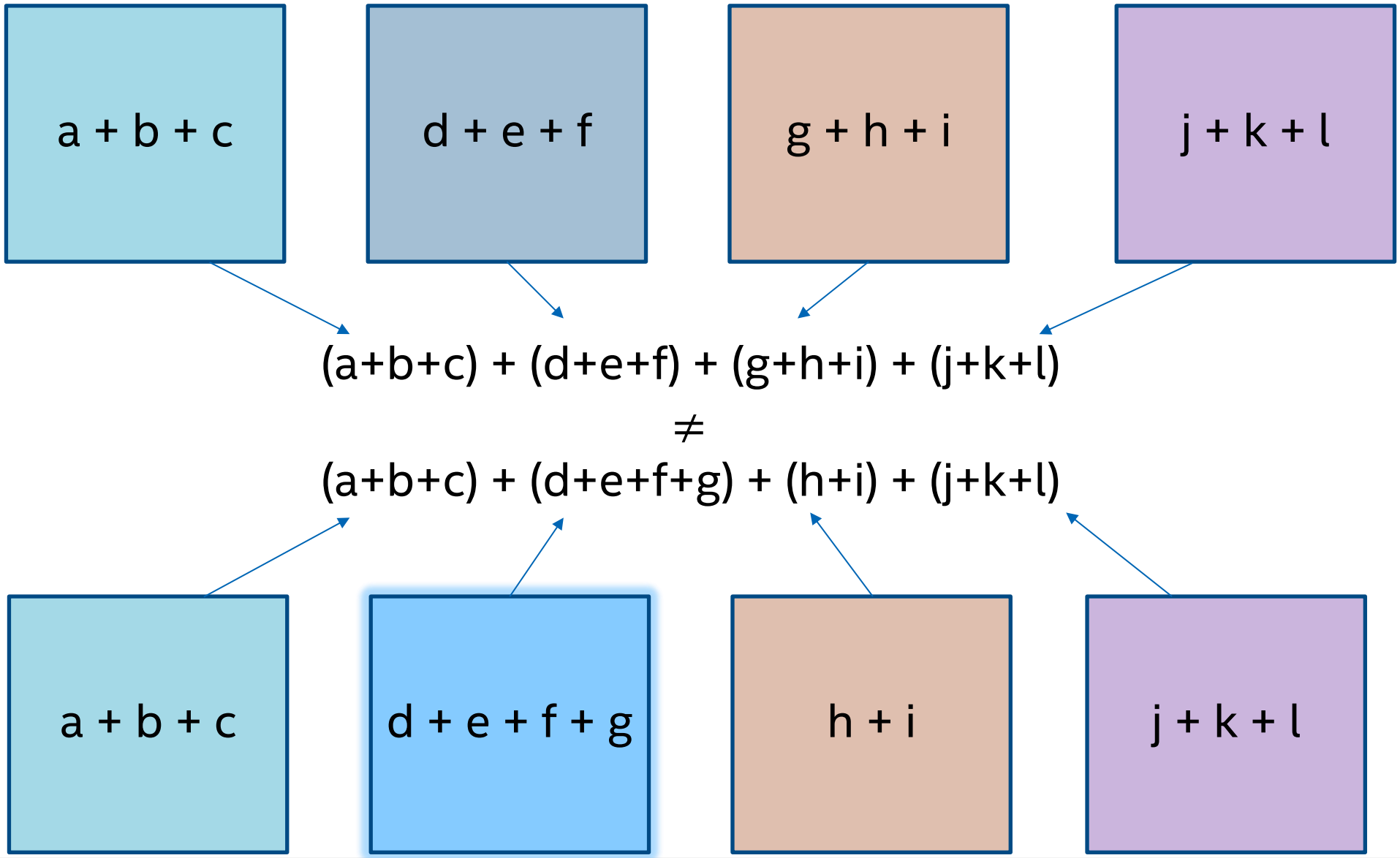
Let’s look at some common but simple scenarios and identify the root causes and costs in performance and portability. We’ll look at

- a multithreaded (or multiprocess) dot product,
- vectorization of a loop, and
- contractions and specialized instructions

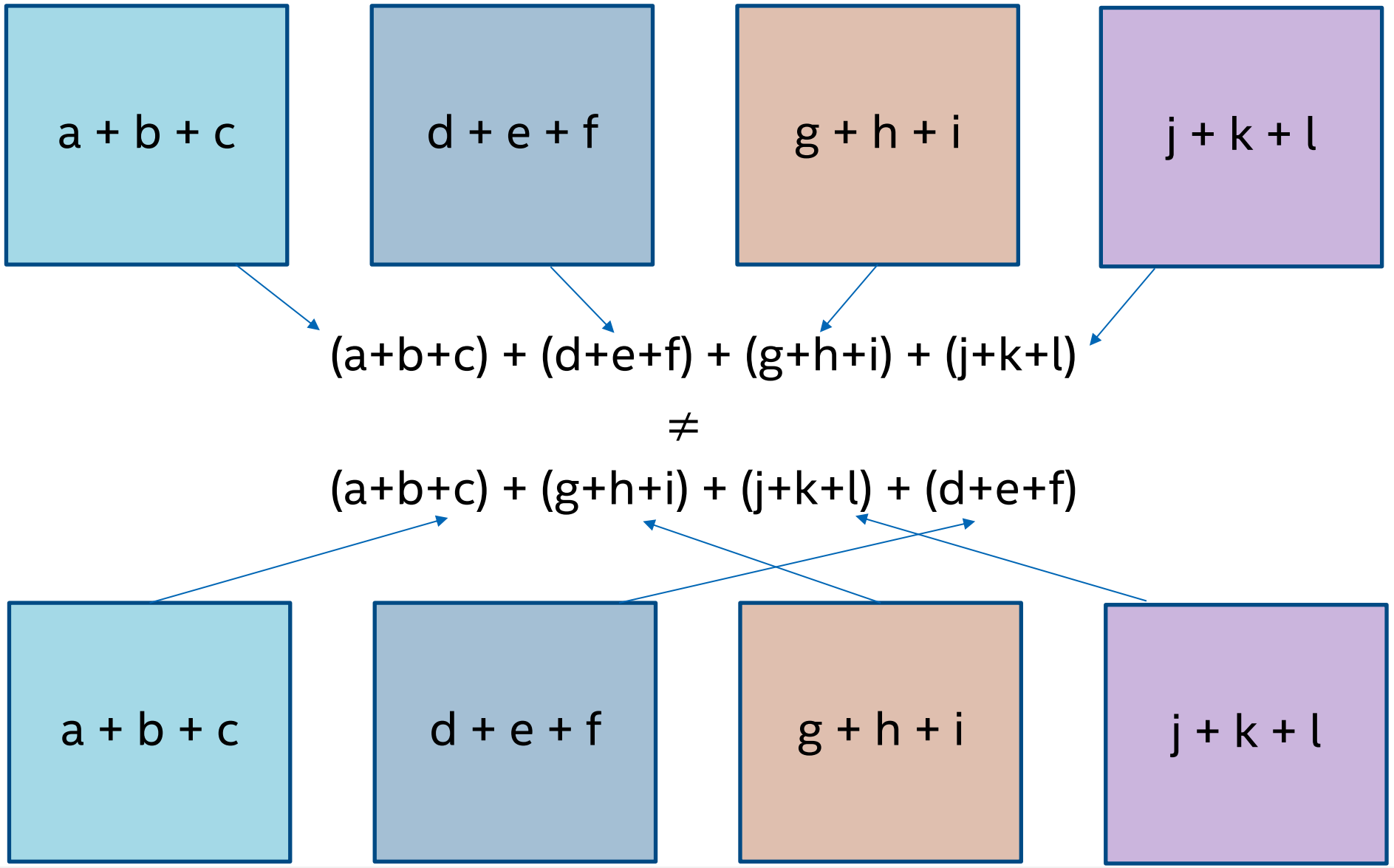
Variable number of threads



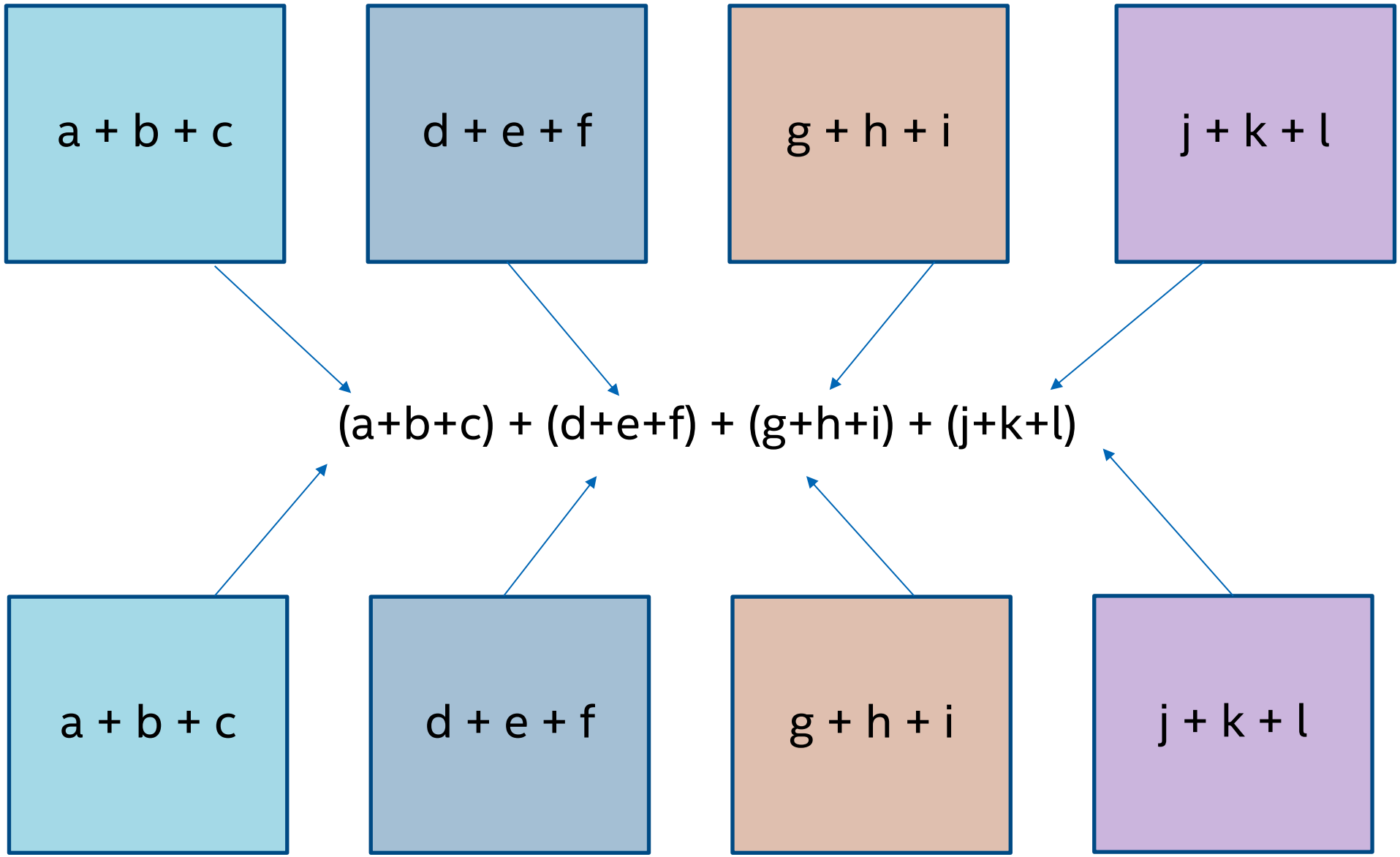
Fix number of threads – dynamic balance



Fix number of threads, static balance



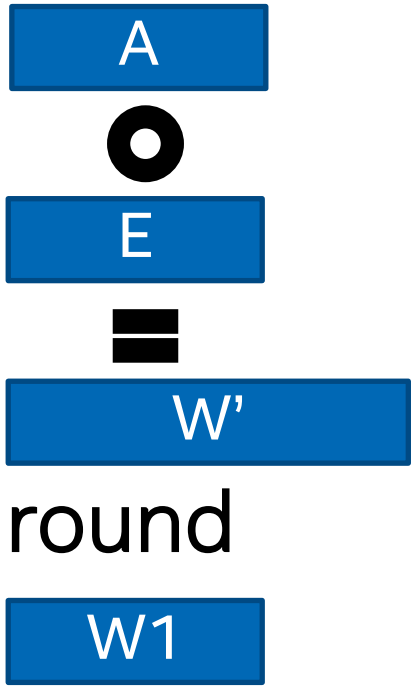
Fix number of threads, static balance, deterministic reduction order

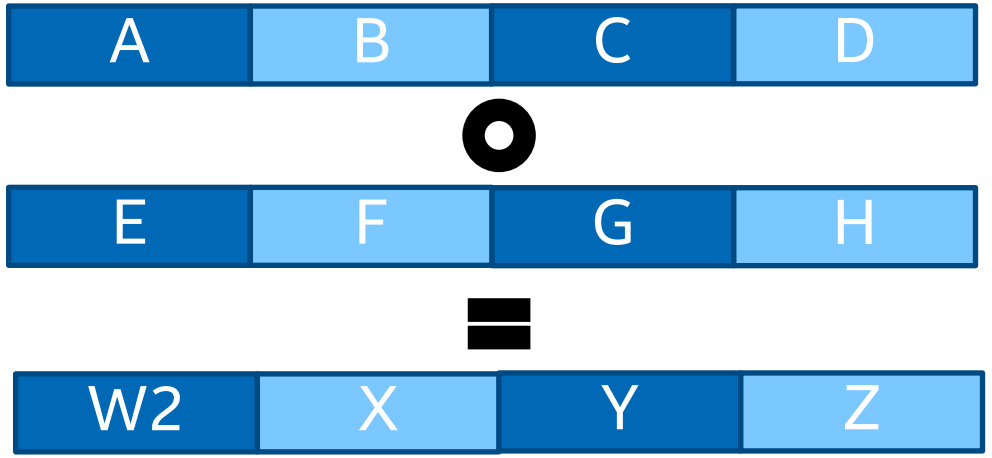


Scalar vs Vectorized Loops: Precision & Rounding Effects

Let $f(\cdot)$ be a function which rounds to the nearest number in 80 bit precision, and $\mathbf{round}(\cdot)$ be function which rounds to the nearest number in 64 bit precision:

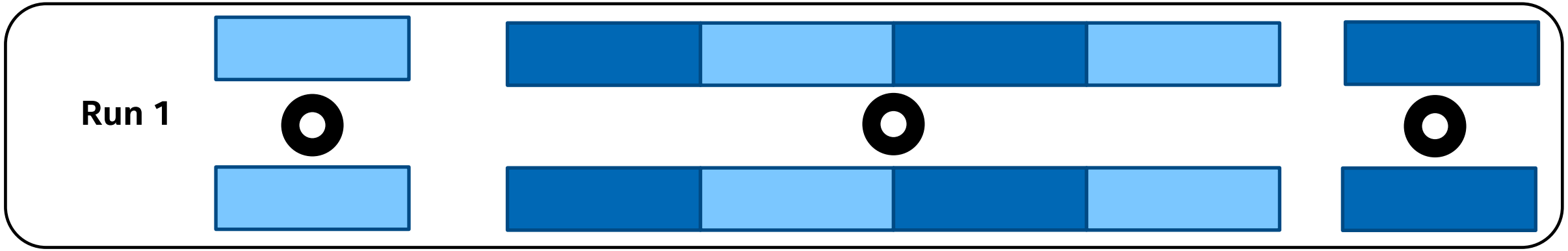
$$round\big(f(a \circ e)\big) \neq round(a \circ e)$$





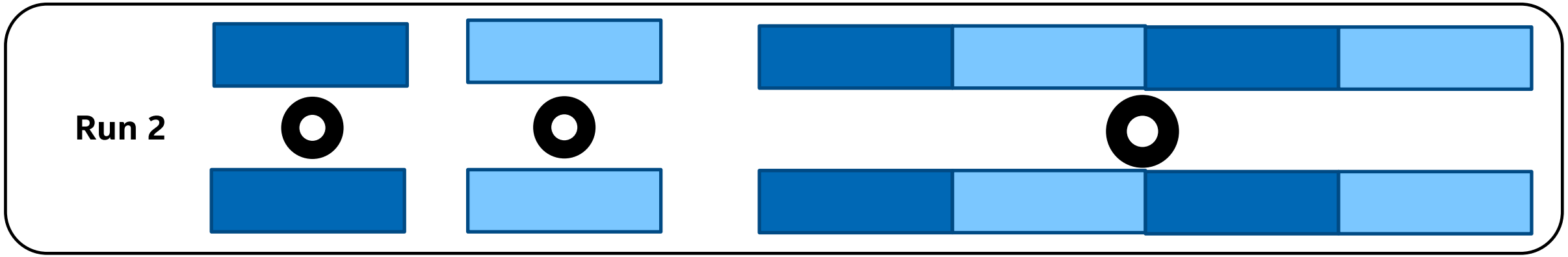
External libraries can call different functions for scalar and vector case – might not be bitwise compatible

Vectorized Loop – Run-to-Run

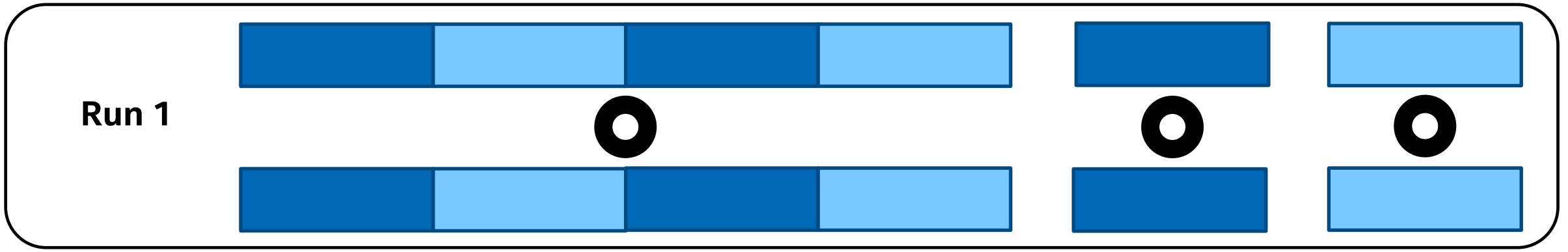


Peel and remainder are not computed as vector instructions – **may have different intermediate precision**.

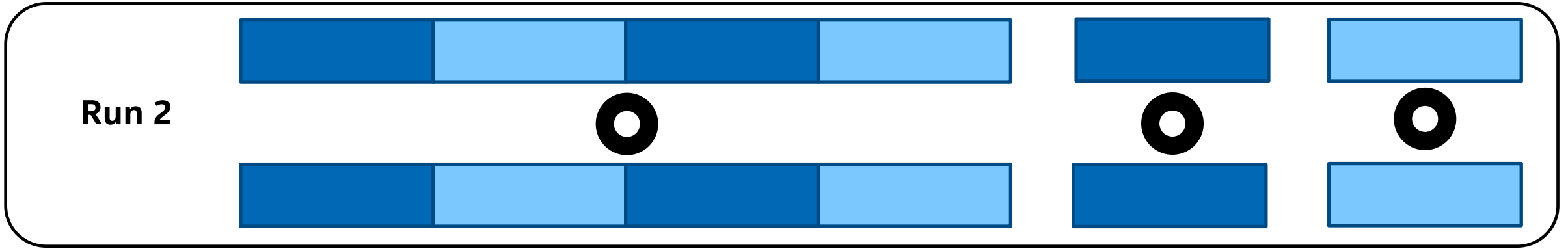
Run-to-run variation is caused because variation in the memory alignment (for example by a variable length stack assignment of a string containing the current user, or directory, or timestamp) causes the length of the peel, remainder, and vector parts to vary. This means non-reproducible results!



Vectorized Loop – Explicitly Align Memory



Now run-to-run reproducible; also portable ONLY to other machines (and compiler versions) with the same available vector instructions and registers.



Fused Multiply Add

- Finally, it's worth investigating the fused-multiply add, which in one instruction performs:

$$\pm a * b \pm c$$

- With only one rounding at the end of the complete operation, rather than first rounding $a * b$, adding c to the rounded result, and rounding that result.
- FMA can also be done on vector registers, revisiting the previous scenario.
- Still allows **run-to-run reproducibility**, but no longer machine-to-machine reproducibility.

Domain Decomposition

Heap or stack alignment

FMA

Vector Math vs
Scalar Math

Non-deterministic
scheduling

FP Environment

Rounding

Association

Key Requirements of CNR

Consistent

Order-of-operations

- Can't have dynamic load balance
  - Fixed number of threads/cores
  - Fixed distribution
- Can't perform optimizations which reassociate arithmetic

Consistent

Intermediate precision

- Can't have variable vectorization
  - Must maintain same vector register size and usage
  - Must maintain same memory alignment for loop peel and remainder
  - Compiler matters!
- Can't offload tasks to accelerators with different FP environments
- Consistent use of contractions (FMA)

Reproducibility must be balanced with Portability and Performance

Floating Point Reproducibility Controls

- Default FP model: *-fp-model fast*
  - *The default for clang is -fp-model precise*
- *-fma is* enabled by default
- *-fp-model consistent* option is supported
- No support for *#pragma fenv_access(on)*
  - *Use #pragma STDC FENV_ACCESS ON*
- Math library functions accuracy and consistency controls are supported, e.g. *-fimf-precision, -fimf-max-error, -fimf-accuracy-bits, -fimf-arch-consistency, etc.*

Interprocedural Optimizations

Interprocedural Optimizations

Extends optimizations across file boundaries

icc

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

icx

-ipo (mapped to -flto)	Link Time Optimization in ICX

Without IPO

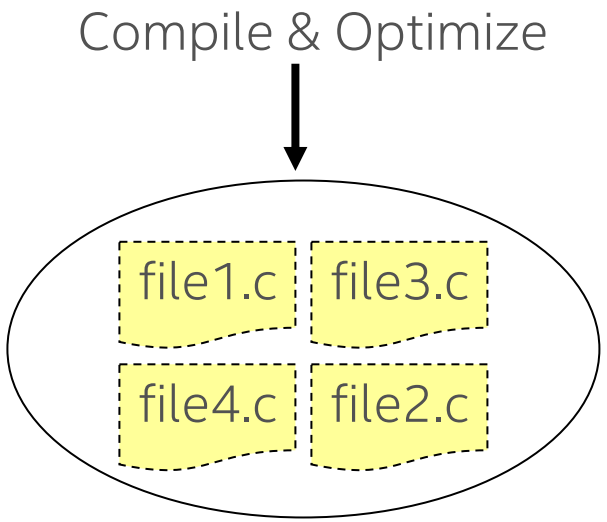
Compile & Optimize → file1.c

Compile & Optimize → file2.c

Compile & Optimize → file3.c

Compile & Optimize → file4.c

With IPO



Link Time Optimization (LTO): tinyurl.com/clang-lto

Interprocedural Optimizations

- icx uses Link Time Optimization (LTO) technology (-flto)
- -ipo should be added to both compilation and linking steps (or replace original linker with the 'lld –fuse-ld=lld')
- Intel tools 'xilink', 'xild', and 'xiar' are removed from ICX and should be replaced in projects settings, makefiles, etc. with equivalent
- Binaries compiled with icc and icx and IPO are not compatible

```
$ icpc -ipo -c hello.cpp
$ icpx -ipo hello.o -o hello
/usr/bin/ld: hello.o:(.data+0x0): undefined reference to `__must_be_linked_with_icc_or_xild'
clang-13: error: linker command failed with exit code 1 (use -v to see invocation)

$ icpx -ipo -c hello.cpp
$ icpc hello.o -o hello
hello.o: file not recognized: file format not recognized
```

- Use llvm-ar for libraries
  - Make sure tools from bin/compiler folder are used

https://llvm.org/docs/LinkTimeOptimization.html

Intrinsic Usage Model

Intrinsic Usage Model

- Target Architecture Must Be Enabled Explicitly
  - Intrinsic types (e.g. __m256d) and functions are unavailable unless the target ISA is enabled
  - Use the compiler option -march or -m, -x
  - LLVM does not implicitly enable ISAs based on intrinsic usage
- Intrinsics Require Strict Type-Correct Arguments
  - ICC was more permissive
- Calling an intrinsic without its required CPU feature results in a compile-time error

Intrinsic Usage Model

- allow_cpu_features Scope Is Function-Level Only
- Operator Overloading on Intrinsic Types Is Not Allowed
- Deprecated Intrinsics Removed
  - Legacy intrinsic names (e.g. extgather, extscatter) supported in ICC

LLVM Intrinsics Handling Differences

Target Architecture Must Be Enabled Explicitly

```
#include <immintrin.h>

double reduce_vector(__m256d input) {
  __m256d temp = _mm256_hadd_pd(input, input);
  return ((double*)&temp)[0] + ((double*)&temp)[2];
}
```

```
icx -c test.cpp
test.cpp(3,23): error: unknown type name '__m256d'
double reduce_vector1(__m256d input) {
                      ^
test.cpp(4,3): error: unknown type name '__m256d'
  __m256d temp = _mm256_hadd_pd(input, input);
  ^
2 errors generated.

icx -c -xCORE-AVX2 test.cpp
```

godbolt.org/z/WcbqzGPPs

LLVM Intrinsics Handling Differences

Intrinsics Require Strict Type-Correct Arguments

```
#include <immintrin.h>
#define CACHE_LINE_SIZE 64
__attribute__((always_inline))
inline void Prefetch_Block(const void* addr, size_t sz, int hint)
{
    char* pref_addr = (char*)addr;
    size_t pref_iters = (sz + CACHE_LINE_SIZE - 1) / CACHE_LINE_SIZE;
    for (int i = 0; i < pref_iters; i++)
    {
        _mm_prefetch(pref_addr, hint/*_MM_HINT_T1*/);
        pref_addr += CACHE_LINE_SIZE;
    }
}
```

```
$ icc -c sample_mm_prefetch.c
$ icx -c sample_mm_prefetch.c
sample_mm_prefetch.c:10:9: error: argument to '__builtin_prefetch' must be a constant integer
        _mm_prefetch(pref_addr, hint/*_MM_HINT_T1*/);
        ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

godbolt.org/z/Txh6G9hPs

LLVM Intrinsics Handling Differences
Missing Target Feature

```
#include "immintrin.h"
int main(){
    __m128i blocks = _mm_aesenc_si128(_mm_setzero_si128(), _mm_setzero_si128());
    return 0;
}
```

```
$ icc -xHASWELL test.c
$ icx -xHASWELL test.c
test.c:4:22: error: always_inline function '_mm_aesenc_si128' requires target feature 'aes', but would be
inlined into function 'main' that is compiled without support for 'aes'
    __m128i blocks = _mm_aesenc_si128(_mm_setzero_si128(), _mm_setzero_si128());
                     ^
1 error generated.

$ icx -xHASWELL -mintrinsic-promote test.c
test.c:3:5: warning: Function target features +aes added due to call to an intrinsic [-Wintrinsic-promote]
int main(){
    ^
test.c:4:22: note: Intrinsic called here
    __m128i blocks = _mm_aesenc_si128(_mm_setzero_si128(), _mm_setzero_si128());
                     ^
1 warning generated.
```

-mintrinsic-promote enables functions containing calls to intrinsics that require a specific CPU feature to have their target architecture automatically promoted to allow the required feature

godbolt.org/z/qPr9aEYxb

LLVM Intrinsics Handling Differences

Operator Overloading on Intrinsic Types Is Not Allowed

```
#include <immintrin.h>

__forceinline __m256 operator+( __m256 aValue1, __m256 aValue2)
{
        return _mm256_add_ps(aValue1,aValue2);
}

__forceinline __mmask16      operator<( __m512 aValue1, __m512 aValue2)
{
        return _mm512_cmplt_ps_mask(aValue2, aValue1);
}
```

```
$ icpc -c op_oveload.cpp
$ icpx -c op_oveload.cpp
op_oveload.cpp:3:22: error: overloaded 'operator+' must have at least one parameter of class or enumeration type
__forceinline __m256 operator+( __m256 aValue1, __m256 aValue2)
                     ^
op_oveload.cpp:8:29: error: overloaded 'operator<' must have at least one parameter of class or enumeration type
__forceinline __mmask16      operator<( __m512 aValue1, __m512 aValue2)
                             ^
2 errors generated.
```

Wrap the intrinsic types in a user-defined type (struct or class) and overload operators on that wrapper: godbolt.org/z/9McEs5Wv7

godbolt.org/z/x77j63Gz4

LLVM Intrinsics Handling Differences

Deprecated Intrinsics Removed

```
#include <immintrin.h>

int main()
{
__m512i w;
float * bv = new float[10];
//w = _mm512_i32gather_epi32(w, bv, 4);
w = _mm512_i32extgather_epi32(w, bv, _MM_UPCONV_EPI32_NONE, 4, 0);
return 0;
}
```

```
$ icpc test.cpp
$ icpx -xCORE-AVX512 test.cpp
test.cpp:9:38: error: use of undeclared identifier '_MM_UPCONV_EPI32_NONE'
w = _mm512_i32extgather_epi32(w, bv, _MM_UPCONV_EPI32_NONE, 4, 0);
^
1 error generated.

$ icpx -xCORE-AVX512 test.cpp
test.ccpp:9:5: error: use of undeclared identifier '_mm512_i32extgather_epi32'
w = _mm512_i32extgather_epi32(w, bv, 0, 4, 0);
^
1 error generated.
```

The intrinsic names like "extgather" and "extscatter" are deprecated in icc and removed in icx

godbolt.org/z/aoYPn47eP

LLVM Intrinsics Handling Differences

allow_cpu_features Scope Is Function-Level Only

```
/* ICC-only: allow specific CPU features starting here */
   _allow_cpu_features(_FEATURE_SSE2 | _FEATURE_AVX);

   for (i = 0; i < numimages; i++) {
       if (!strcmp(filename, imagelist[i])) {
           *found = imagelist[i];
           *intable = 1;
           break;
       }
   }
```

- icc supports *_allow_cpu_feature* intrinsic to use the specified processor feature at a code block level
- Use this intrinsic function to use the specified processor feature at a code block level. The function only affects the scope of the code following the function call.
- LLVM only allows specifying target-features on a per-function basis

godbolt.org/z/Gn569v7vz

LLVM Intrinsics Handling Differences

allow_cpu_features Scope Is Function-Level Only

```
#include <immintrin.h>

__attribute__((allow_cpu_features(_FEATURE_AVX2)))
void compute(float a[], float b[])
{ for (int i = 0; i < 100; i++) a[i] = a[i] + b[i] * a[i]; }
```

- Direct conceptual replacement for ICC's block-level_allow_cpu_features
- Scoped per function, in line with LLVM rules.
- https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2025-2/allow-cpu-features.html

godbolt.org/z/qT5GT74nr

Intel® oneAPI DPC++/C++ Compiler - Online Resources

ICC to ICX Porting Guide

Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference

Compiler Release Notes

Intel Open-source LLVM-based Compiler Project

Intel® oneAPI DPC++/C++ Compiler Forum

Intel® Fortran Compiler - Online Resources

Porting Guide for Intel® Fortran Compiler

Intel® Fortran Developer Guide and Reference

Compiler Release Notes

Fortran Compiler Forum

Notices & Disclaimers

Performance varies by use, configuration and other factors. Learn more on the Performance Index site.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure. Results have been estimated or simulated.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

All product plans and roadmaps are subject to change without notice.

Code names are used by Intel to identify products, technologies, or services that are in development and not publicly available. These are not "commercial" names and not intended to function as trademarks.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

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