Intel® Compiler introduction

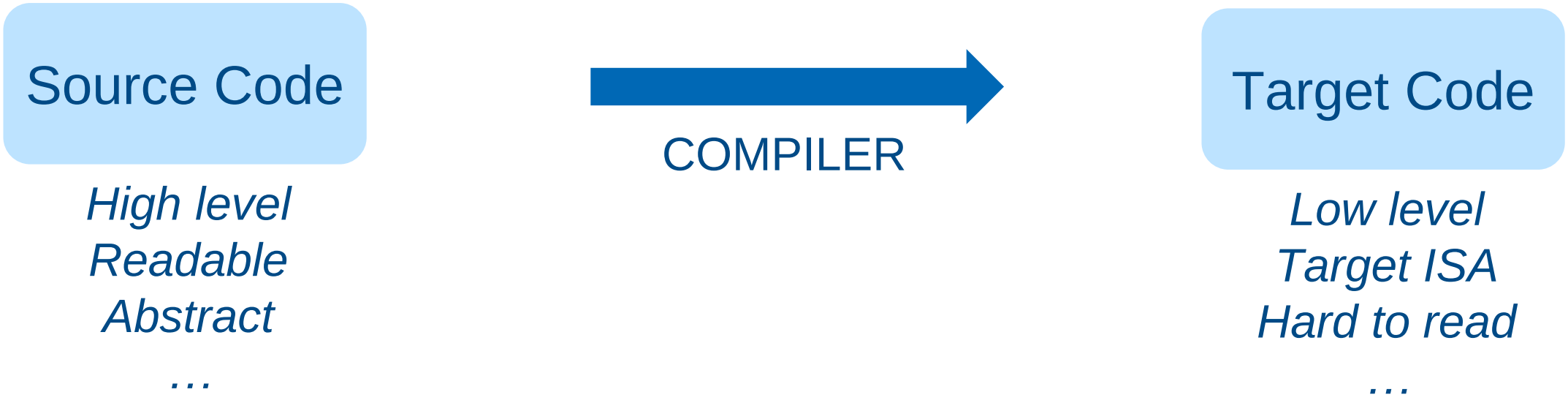
November 2021

Agenda

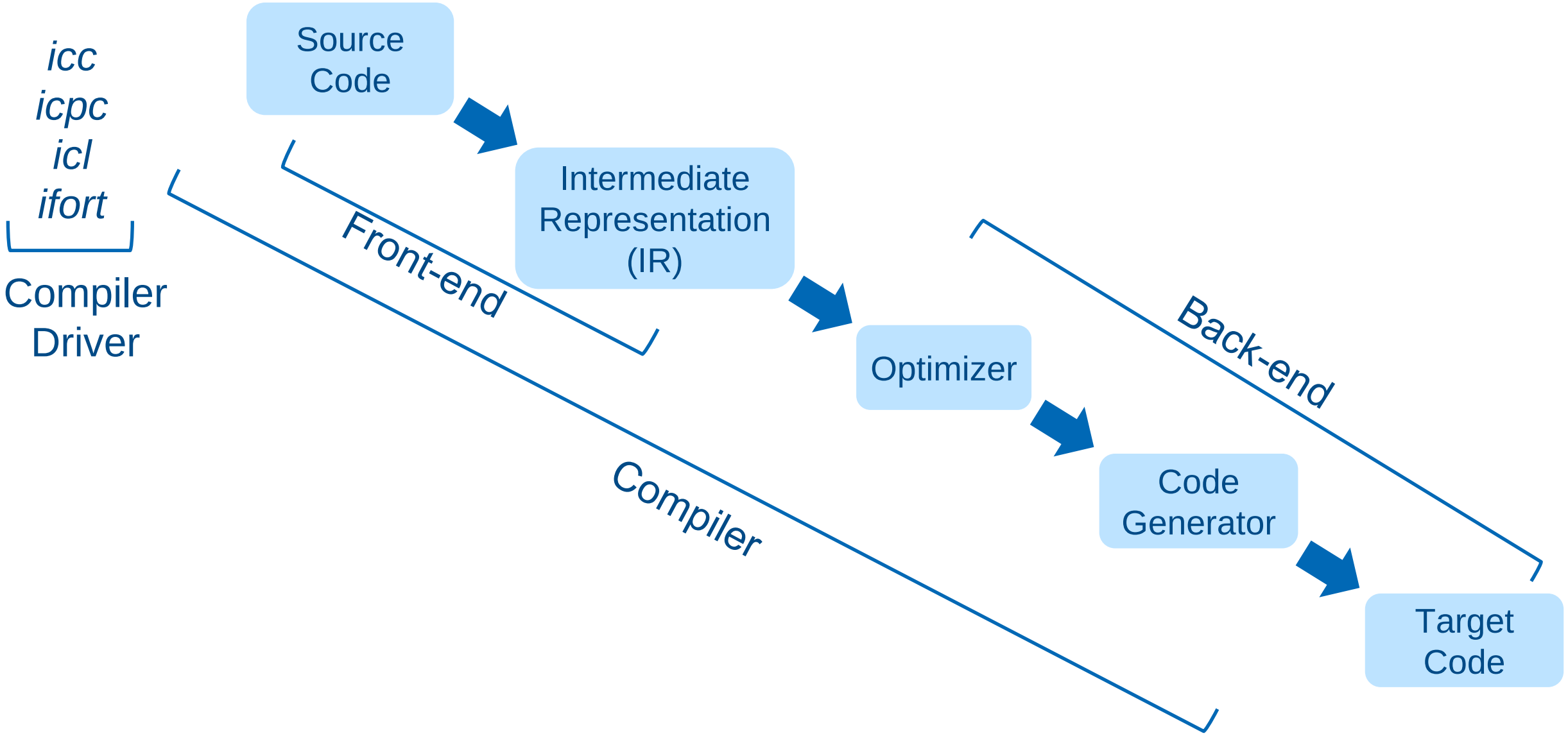
- Introduction
- Compiler Optimizations
  - High-Level Optimizations
  - InterProcedural Optimizations
  - Profile-Guided Optimizations
  - Auto-Parallelization
  - FP model
- Vectorization
- Lab & Demo
- LLVM-based Intel Compilers

Introduction

Introduction to Compilers



Compiler Architecture – simplified model



Compiler Optimizations

intel®

Common optimization options

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

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

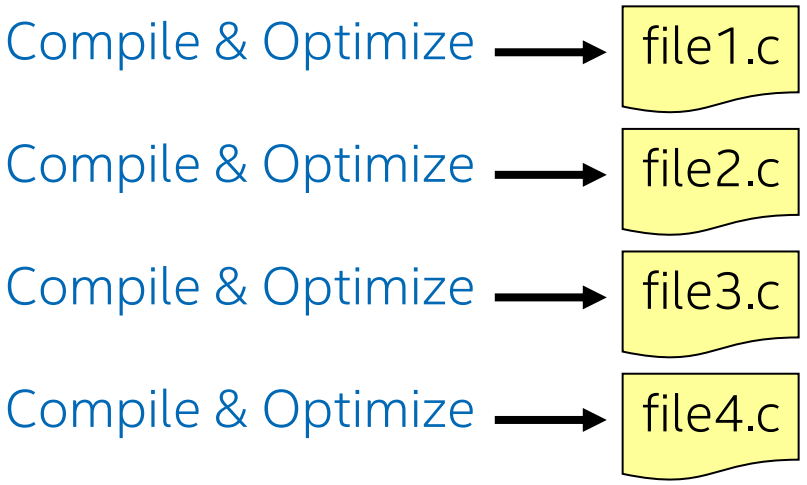
InterProcedural Optimizations

Extends optimizations across file boundaries

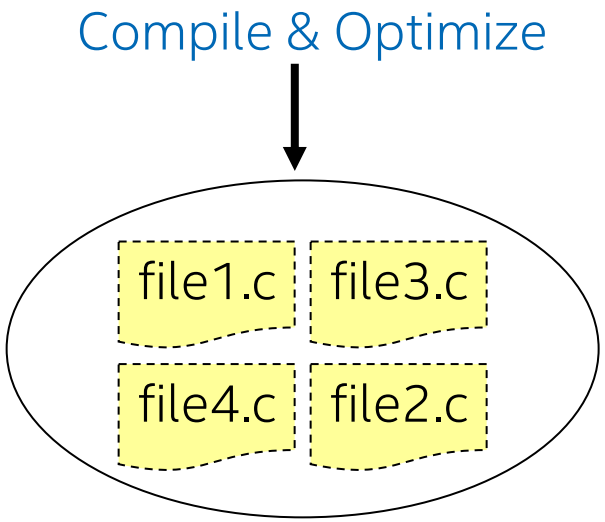
icc/ifort

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

Without IPO



With IPO



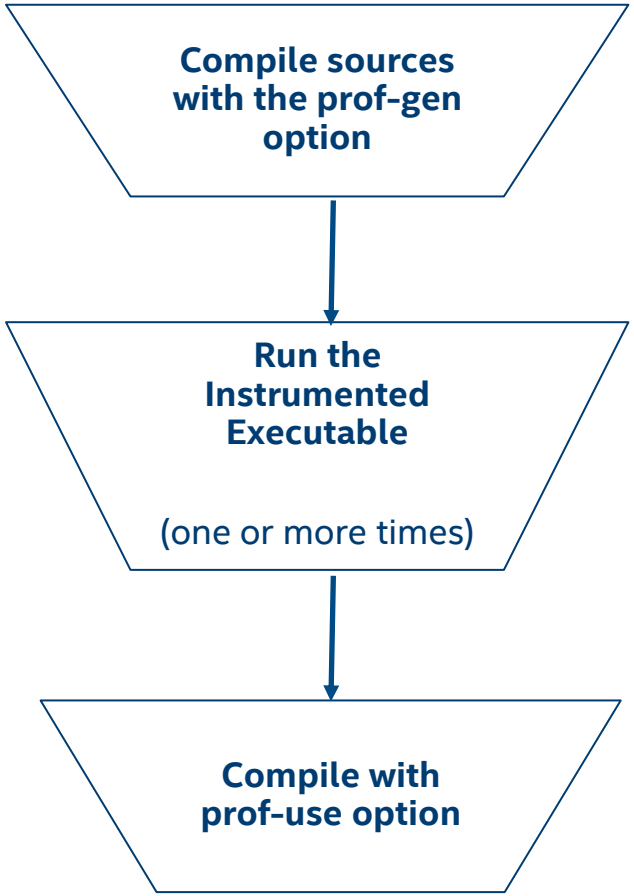
Profile-Guided Optimizations (PGO)

- 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

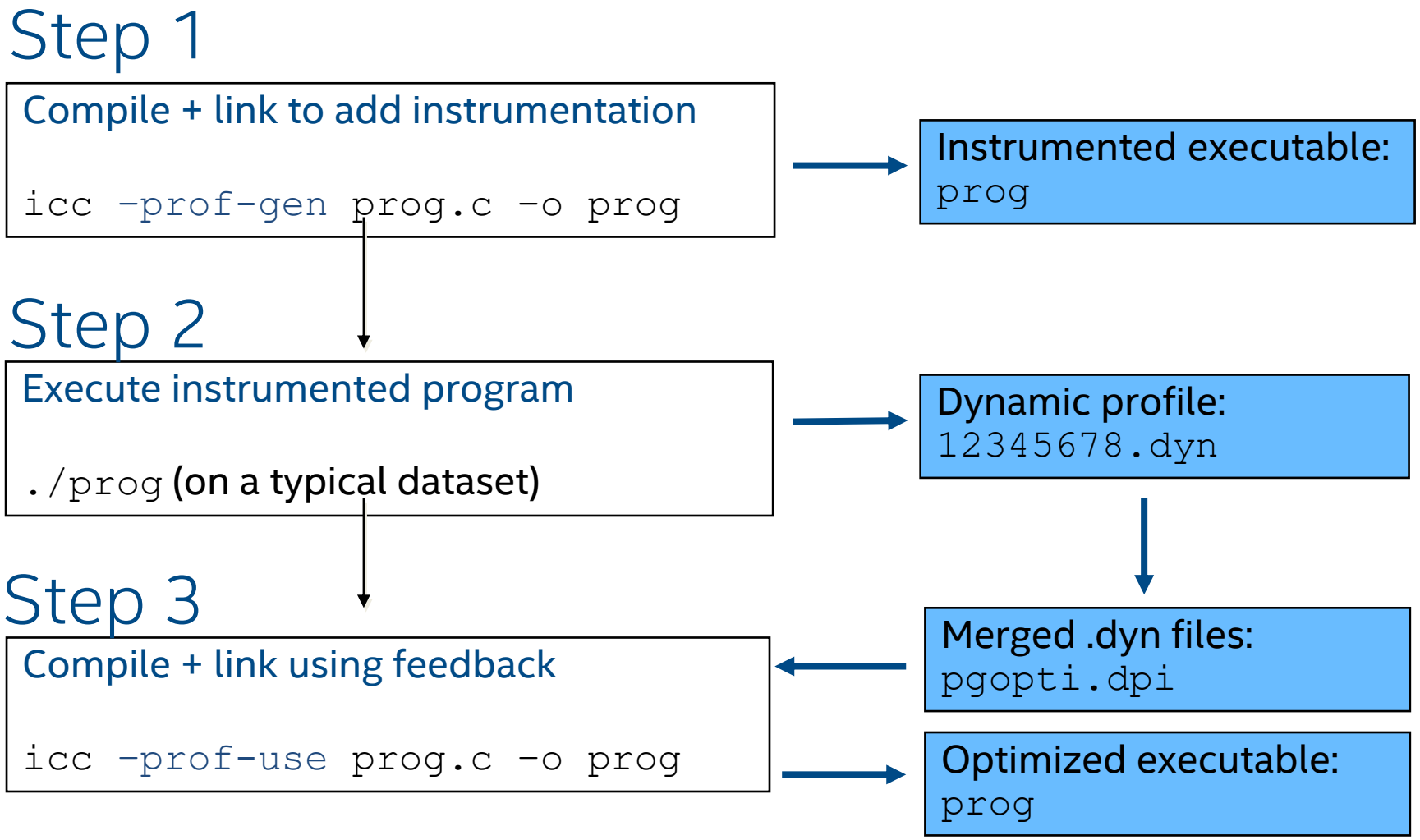
```
if (x > y)
     do_this();
 else
     do that();
```

```
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



PGO Usage: Three-Step Process



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

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

Floating-Point (FP) Programming Objectives

- Accuracy
  - Produce results that are “close” to the correct value
    - Measured in relative error, possibly in ulp
- Performance
  - Produce the most efficient code possible
- Reproducibility
  - Produce consistent results
    - From one run to the next
    - From one set of build options to another
    - From one compiler to another
    - From one platform to another

These objectives usually conflict!
Use of compiler options lets you control the tradeoffs.
Different compilers have different defaults.

Floating Point Reproducibility Controls

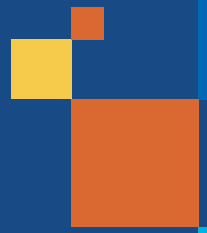
-fp-model

- 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 overridde
- 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

Vectorization

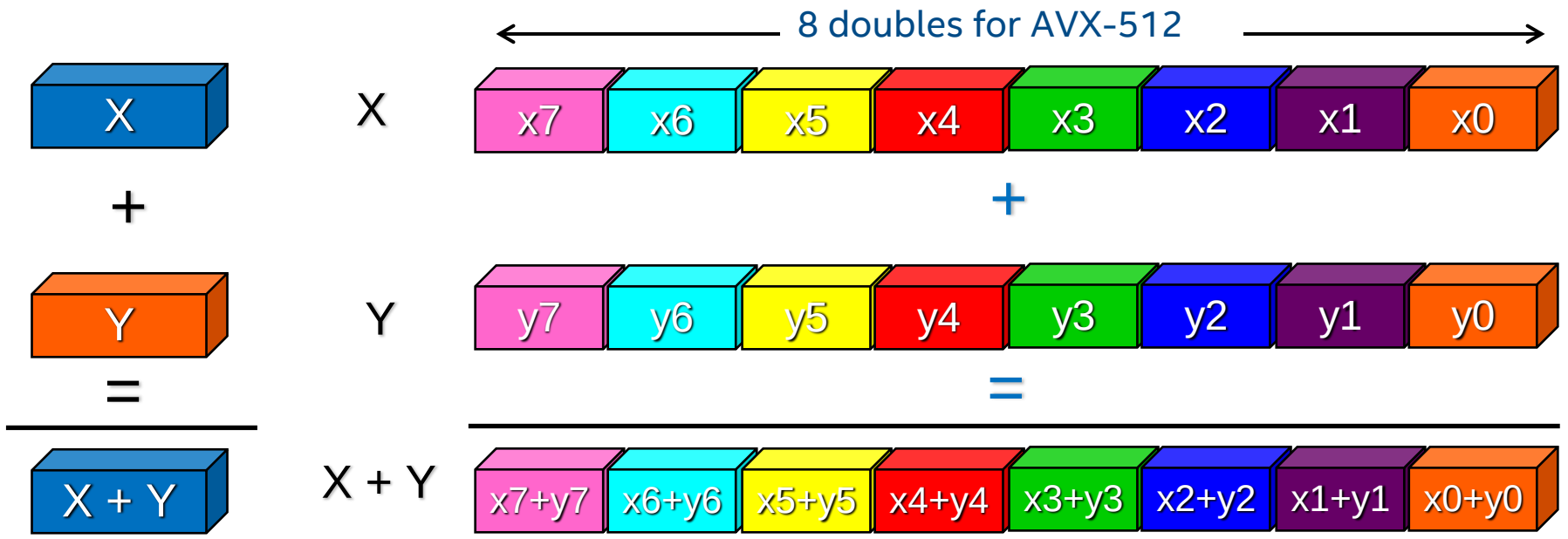


intel®

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:
OpenMP* SIMD directives

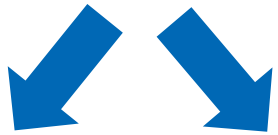
SIMD intrinsic class
(e.g.: F32vec, F64vec, ...)

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

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

Auto-vectorization of Intel Compilers

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

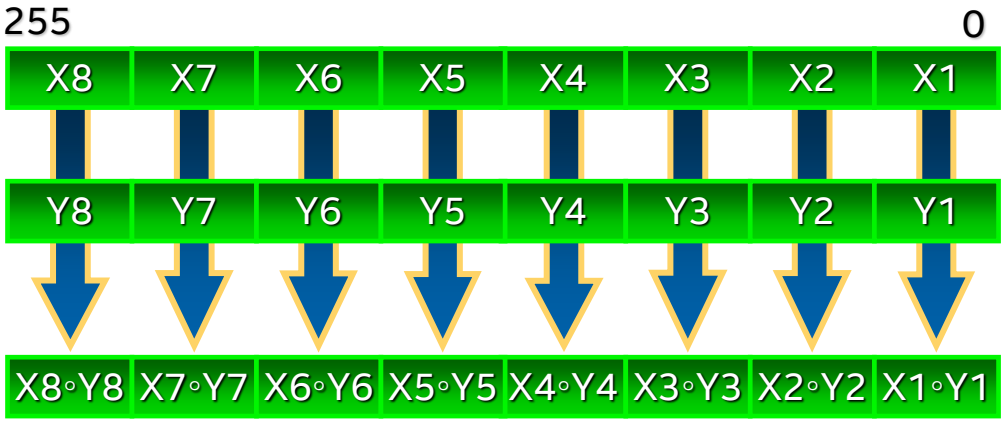


```
vmovups    zmm1, ZMMWORD PTR [r12+rdi*4]                  #5.21
vextractf64x4 ymm2, zmm1, 1                               #5.21
vcvtps2pd zmm3, ymm1                                      #5.21
vaddpd     zmm0, zmm16, zmm3                              #5.30
vcvtps2pd zmm4, ymm2                                      #5.21
vaddpd     zmm17, zmm16, zmm4                             #5.30
call       QWORD PTR [__svml_sin8@GOTPCREL+rip]           #5.17
vmovaps    zmm18, zmm0                                    #5.17
vmovaps    zmm0, zmm17                                    #5.17
call       QWORD PTR [__svml_sin8@GOTPCREL+rip]           #5.17
vcvtpd2ps ymm2, zmm18                                     #5.17
vcvtpd2ps ymm1, zmm0                                      #5.17
vinsertf64x4 zmm3, zmm2, ymm1, 1                          #5.17
vmovups    ZMMWORD PTR [rsi+rdi*4], zmm3                  #5.8
```

```
vcvtss2sd xmm16, xmm16, DWORD PTR [r12+r14*4]             #5.17
vaddsd    xmm0, xmm16, QWORD PTR .L_2il0floatpacket.1[rip] #5.17
call      sin                                             #5.17
vcvtsd2ss xmm0, xmm0, xmm0                                #5.8
vmovss    DWORD PTR [r13+r14*4], xmm0
```

godbolt.org/z/dWvEh7GGc

SIMD Types for Intel® Architecture



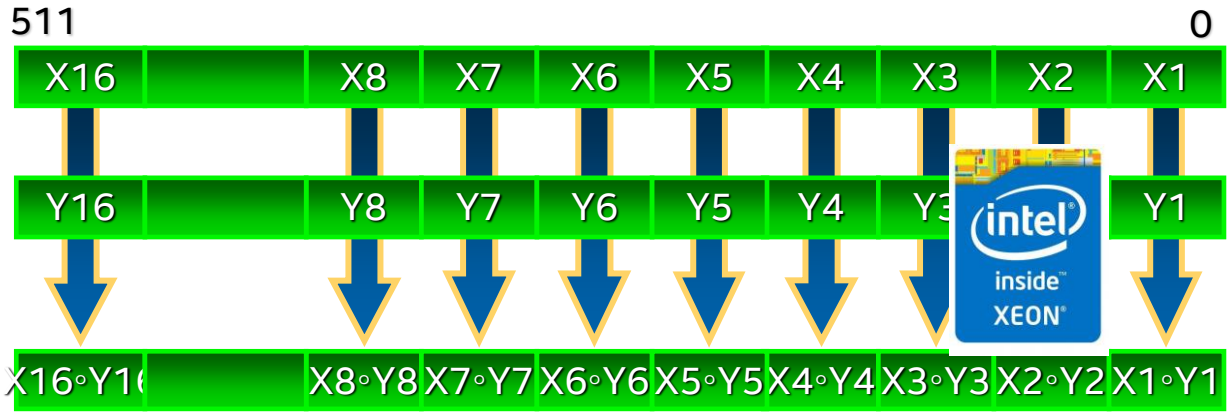
AVX

Vector size: **256 bit**

Data types:

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

VL: 4, 8, 16, 32



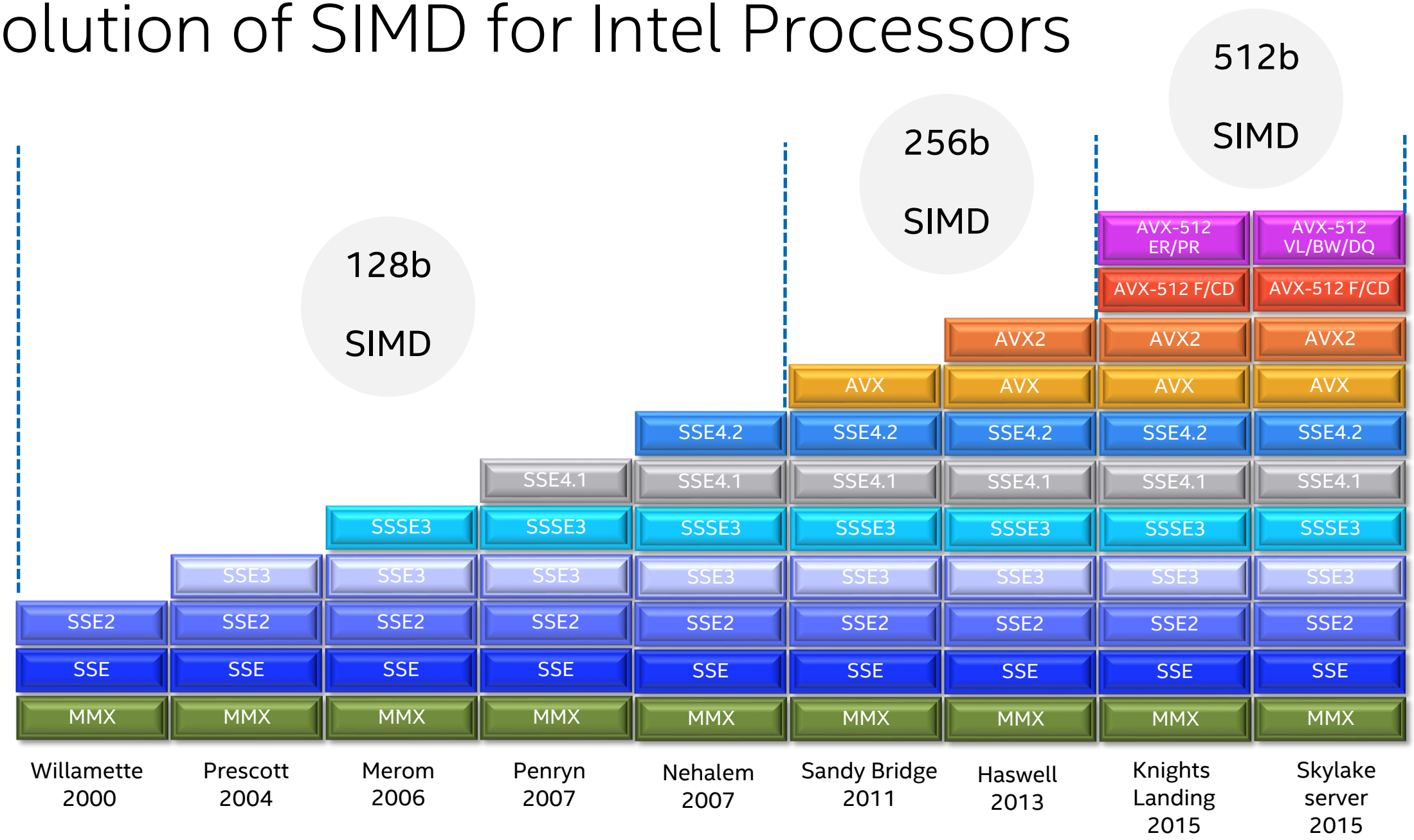
Intel® AVX-512

Vector size: **512 bit**

Data types:

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

Evolution of SIMD for Intel Processors



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

Intel® AVX-512 generation for SKX/CLX

Compile with processor-specific option -xCORE-AVX512

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

A new compiler option -qopt-zmm-usage=low|high is added to enable a smooth transition from AVX2 to AVX-512

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

```
icpc -c -xCORE-AVX512 -qopenmp -qopt-report:5 foo.cpp
```

```
remark #15305: vectorization support: vector length 4
  …
remark #15321: Compiler has chosen to target XMM/YMM
vector. Try using -qopt-zmm-usage=high to override
  …
remark #15478: estimated potential speedup: 5.260
```

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*

SPEC HPC2021: -qopt-zmm-usage=high -Ofast -xCORE-AVX512 -qopenmp -ipo -qopt-multiple-gather-scatter-by-shuffles -fimf-precision=low:sin,sqrt [for IFORT: -align array64byte -nostandard-realloc-lhs]

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.

Reasons for Vectorization Failures and Inefficiency

Most frequent reasons:

- Data dependence
- Alignment
- Unsupported loop structure
- Non-unit stride access
- Function calls
- Non-vectorizable mathematical functions

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

Data Dependency and vectorization

Flow Dependency

```
X = …
… = X
```

read-after-write

Anti Dependency

```
… = X
X = …
```

write-after-read

Output Dependency

```
X = …
X = …
```

write-after-write

```
DO I = 1, 10000
  A(I) = B(I) * 17

  X(I+1) = X(I) + A(I)
ENDDO
```

Loop-independent dependence

Loop-carried dependence

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

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

Failing Disambiguation

- Many potential dependencies detected by the compiler result from unresolved memory disambiguation:

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

  Example:

```
void test(int *a, int *b, int *c, int* d, int* e)
{
    for (int i = 0; i < 10000; i++) b[i] = a[i]+c[i]*d[i]+e[i];
}
```

- Without additional information (like inter-procedural knowledge) the compiler has to assume a and b to be aliased!
- **Use directives, switches and attributes to aid disambiguation!**
- This is programming language and operating system specific
- Use with care as the compiler might generate incorrect code in case the hints are not fulfilled!

godbolt.org/z/qf4TKdEca

Disambiguation Hints I

- Disambiguating memory locations of pointers in C99: -std=c99
- Intel® C++ Compiler also allows this for other modes (e.g. -std=c89, -std=c++11, …), too - **not standardized**, though: -restrict
- Declaring pointers with keyword **restrict** asserts compiler that they only reference individually assigned, non-overlapping memory areas
- Also true for any result of pointer arithmetic (e.g. ptr + 1 or ptr[1])
- Example:

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


godbolt.org/z/r34hPaz3j

Disambiguation Hints II

- Directive:

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

- For C/C++:
  - Assume no aliasing at all (dangerous!): -fno-alias
  - No aliasing between function arguments: -fargument-noalias
  - No aliasing between function arguments and global storage: -fargument-noalias-global

Optimization Report – An Example

$ icc -c -xcommon-avx512 **-qopt-report=3** -qopt-report-phase=loop,vec foo.c

Creates foo.optrpt summarizing which optimizations the compiler performed or tried to perform.
Level of detail from 0 (no report) to 5 (maximum).
-qopt-report-phase=loop,vec asks for a report on vectorization and loop optimizations only
Extracts:

```
LOOP BEGIN at foo.c(4,3)
Multiversioned v1
   remark #25228: Loop multiversioned for Data Dependence...
   remark #15300: LOOP WAS VECTORIZED
   remark #15450: unmasked unaligned unit stride loads: 1
   remark #15451: unmasked unaligned unit stride stores: 1
   ....  (loop cost summary)  ....
LOOP END

LOOP BEGIN at foo.c(4,3)
<Multiversioned v2>
   remark #15304: 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 < 512; i++)
       sth[i] = sin(theta[i]+3.1415927);
}
```

godbolt.org/z/dWvEh7GGc

Optimization Report – An Example

$ icc -c -xcommon-avx512 -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr **-fargument-noalias** foo.c

report to stderr instead of foo.optrpt

...

LOOP BEGIN at foo.c(4,3)

...

remark #15417: vectorization support: number of FP up converts: single precision to double precision 1
remark #15418: vectorization support: number of FP down converts: double precision to single precision 1
remark #15300: LOOP WAS VECTORIZED
remark #15450: unmasked unaligned unit stride loads: 1
remark #15451: unmasked unaligned unit stride stores: 1
remark #15475: --- begin **vector cost summary** ---
remark #15476: **scalar cost: 111**
remark #15477: **vector cost: 10.310**
remark #15478: **estimated potential speedup: 10.740**
remark #15482: vectorized math library calls: 1
remark #15487: **type converts: 2**
remark #15488: --- end vector cost summary ---
remark #25015: 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 < 512; i++)
       sth[i] = sin(theta[i]+3.1415927);
}
```

godbolt.org/z/dWvEh7GGc

Optimization Report – An Example

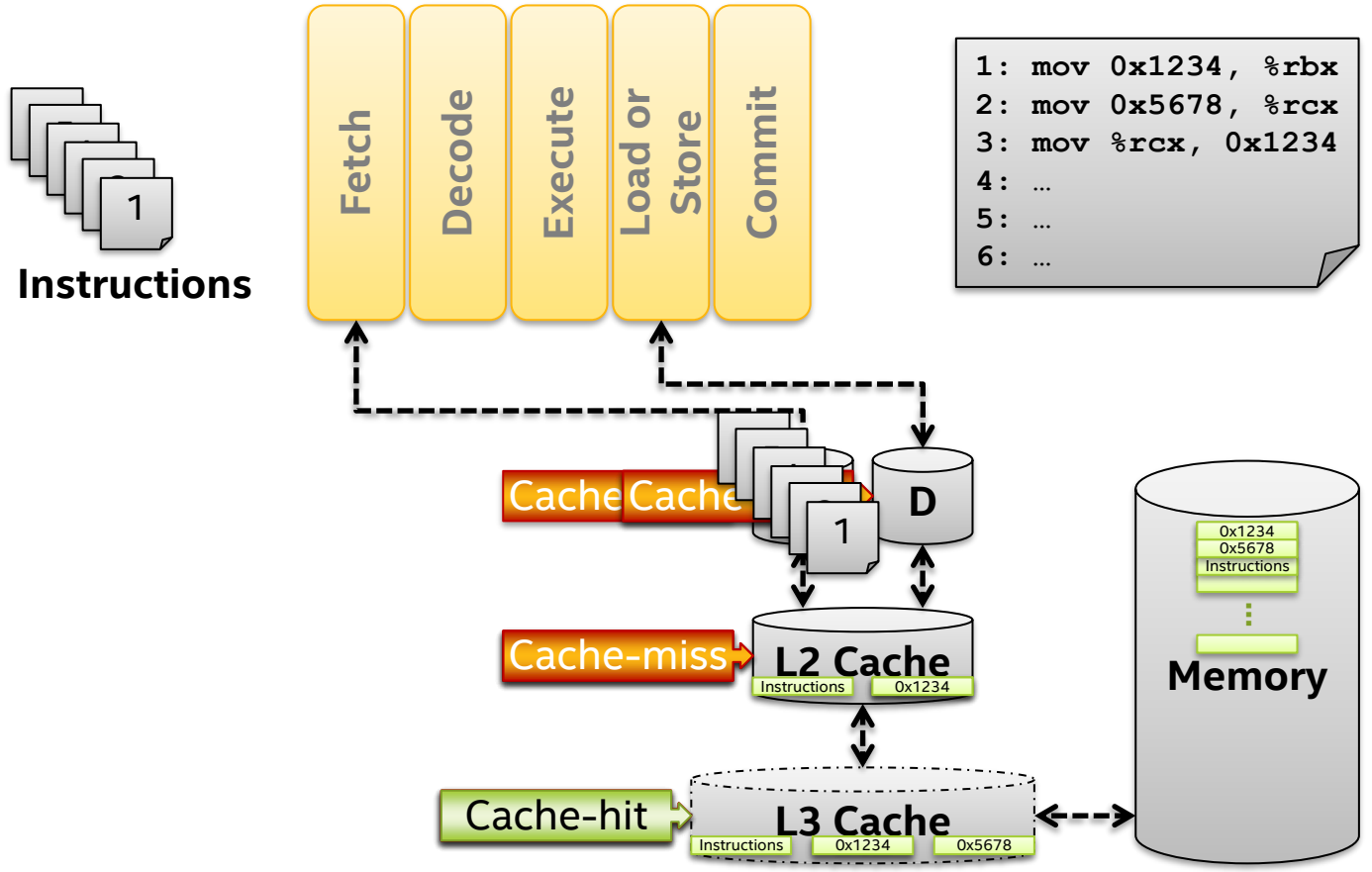
```
$ icc -S -xcommon-avx512 -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr -fargument-noalias foo.c
LOOP BEGIN at foo2.c(4,3)
...
remark #15305: vectorization support: vector length 32
remark #15300: LOOP WAS VECTORIZED
   remark #15450: unmasked unaligned unit stride loads: 1
   remark #15451: unmasked unaligned unit stride stores: 1
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 109
   remark #15477: vector cost: 5.250
   remark #15478: estimated potential speedup: 20.700
   remark #15482: vectorized math library calls: 1
   remark #15488: --- end vector cost summary ---
   remark #25015: Estimate of max trip count of loop=32
LOOP END

$ grep sin foo.s
        call      __svml_sinf16_b3
```

```
#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/dWvEh7GGc

Cache-Hierarchy & Cache-Line



Cache-hierarchy:

- For data and instructions
- Usually inclusive caches
- Races for resources
- Can improve access speed
- Cache-misses & cache-hits

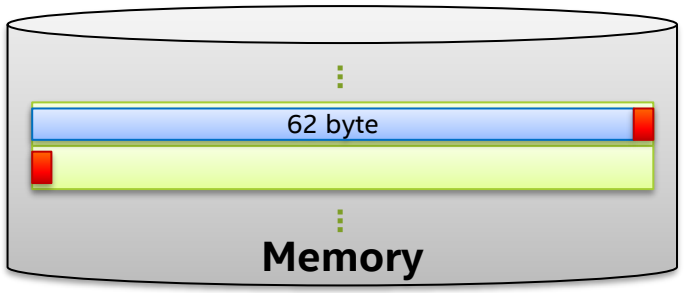
Cache-line:

- Always full 64 byte block
- Minimal granularity of every load/store
- Modifications invalidate entire cache-line (dirty bit)

Alignment

What happens if data spawns across cache-lines?

- Loads/stores require all cache-lines of a datum to be loaded into cache.
- In worst case this doubles load/store times:



Cache-line n
Cache-line n+1

```
#pragma pack(2)
struct {
… // 62 byte
int x; // 4 byte
} data;
```

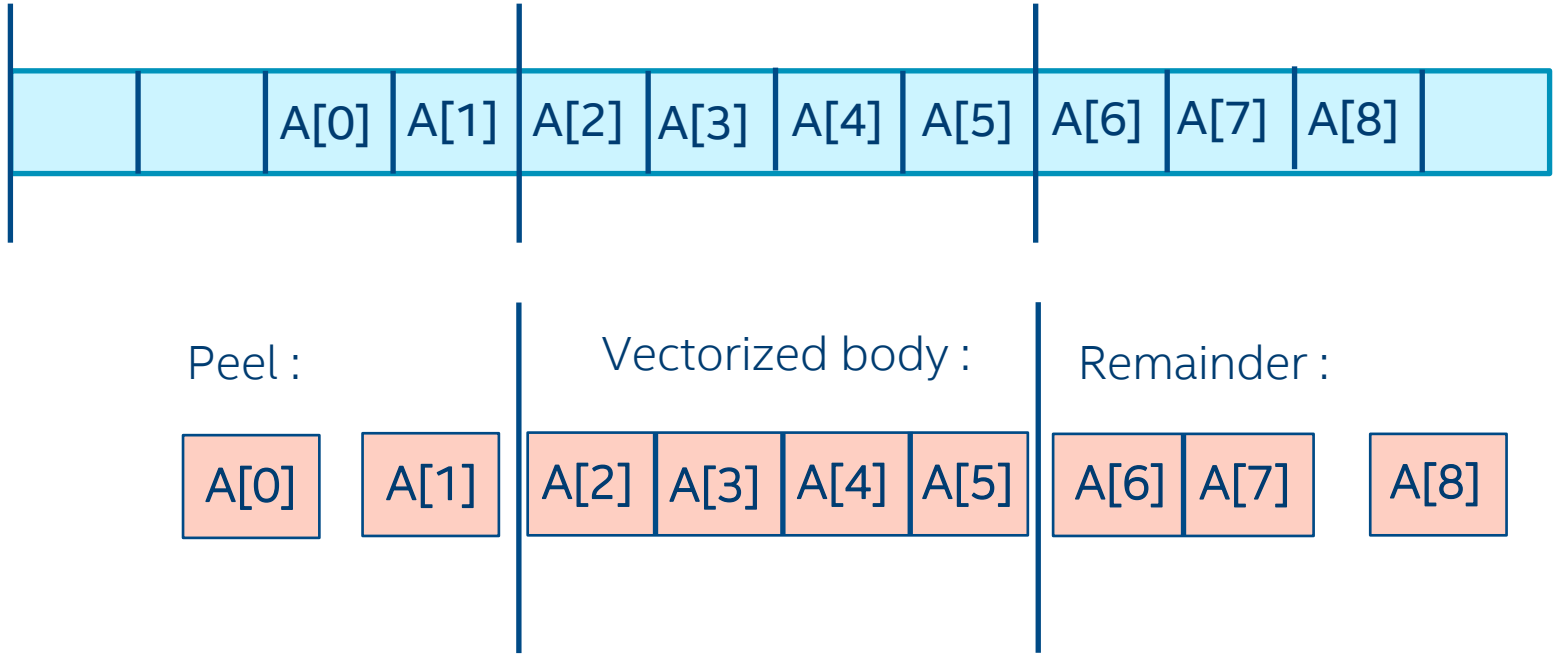
Both cache-lines **n** and **n+1** need to be loaded up to L1 cache to access **x**!

Align data by padding or explicitly by compiler attributes to fit into exactly one cache-line, e.g.:

```
#pragma pack(2)
struct {
… // 62 byte
… // 2 byte
int x; // 4 byte
} data;
```

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

Data Alignment for C/C++

- Aligned heap memory allocation by intrinsic/library call:

```
void* aligned_alloc( std::size_t alignment, std::size_t size ); (since C++17)

void* _mm_malloc(int size, int base)

int posix_memaligned(void **p, size_t base, size_t size)
```

- Automatically allocate memory with the alignment of that type using new operator:

```
#include <aligned_new>
```

- Align attribute for variable declarations:

  alignas specifier (since C++11):

```
alignas(64) char line[128];

<var> __attribute__((aligned(base)))
```

Compiler Alignment Hints for C/C++

- Hint that start address of an array is aligned (Intel Compiler only):
  `__assume_aligned(<array>, base)`

- **`#pragma vector [aligned|unaligned]`**
  - Only for Intel Compiler
  - Asserts compiler that aligned memory operations can be used for all data accesses in loop following directive
  - Use with care:
    The assertion must be satisfied for all(!) data accesses in the loop!

Alignment Hints for Fortran

- Align variables:
  `!DIR$ ATTRIBUTES ALIGN: base :: variable`
- Align data items globally:

  `-align array<n>byte`

  Aligns the start of arrays on an n-byte boundary

- **`!DIR$ VECTOR [ALIGNED|UNALIGNED]`**
  - Asserts compiler that aligned memory operations can be used for all data accesses in loop following directive
  - Use with care:
    The assertion must be satisfied for all(!) data accesses in the loop!
- Hint that an entity in memory is aligned:
  `!DIR$ ASSUME_ALIGNED address1:base [, address2:base] ...`

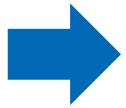
Unsupported Loop Structure

Loops where compiler does not know the iteration count:

- Upper/lower bound of a loop are not loop-invariant
- Loop stride is not constant
- Early bail-out during iterations (e.g. **break**, exceptions, etc.)
- Too complex loop body conditions for which no SIMD feature instruction exists
- Loop dependent parameters are globally modifiable during iteration
  (language standards require load and test for each iteration)

Transform is possible, e.g.:

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



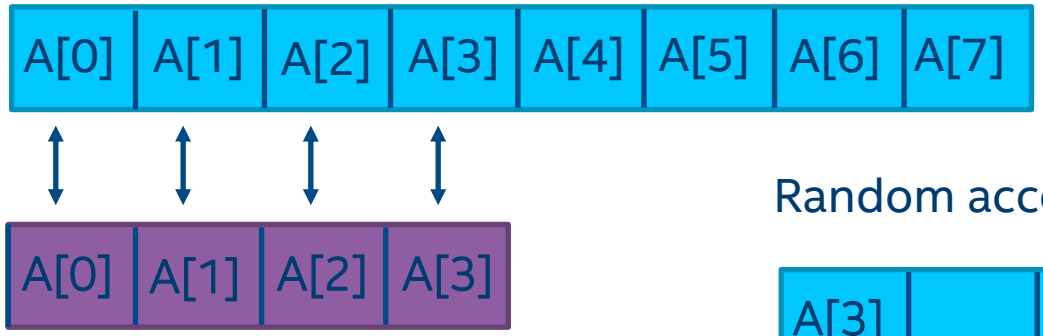
```
struct _x { int d; int bound; };
void doit(int *a, struct _x *x)
{
  int local_ub = x->bound;
  for(int i = 0; i < local_ub; i++)
    a[i] = 0;
}
```

loop was not vectorized: loop control variable i was found, but loop iteration count cannot be computed before executing the loop

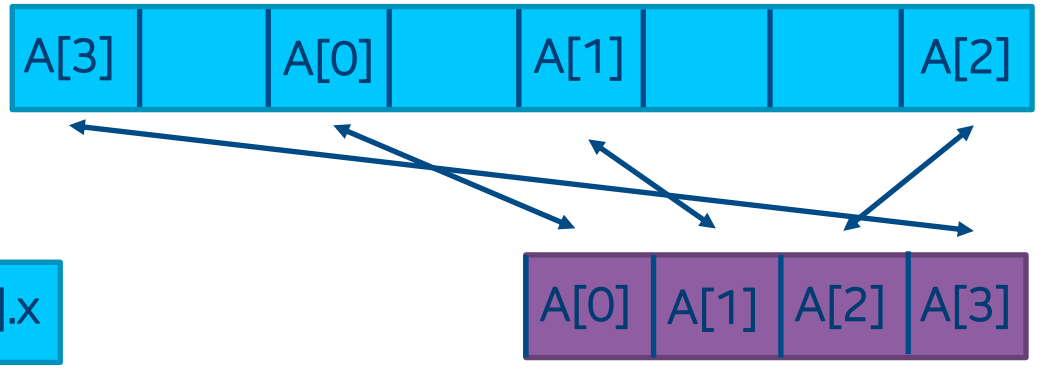
godbolt.org/z/qPed69Tsh

Memory access patterns

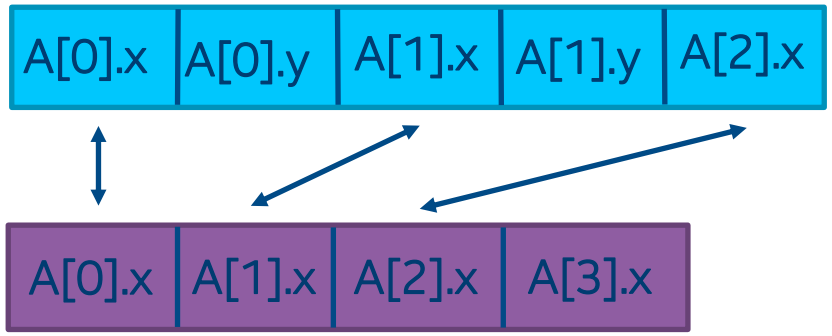
Unit stride (contiguous):



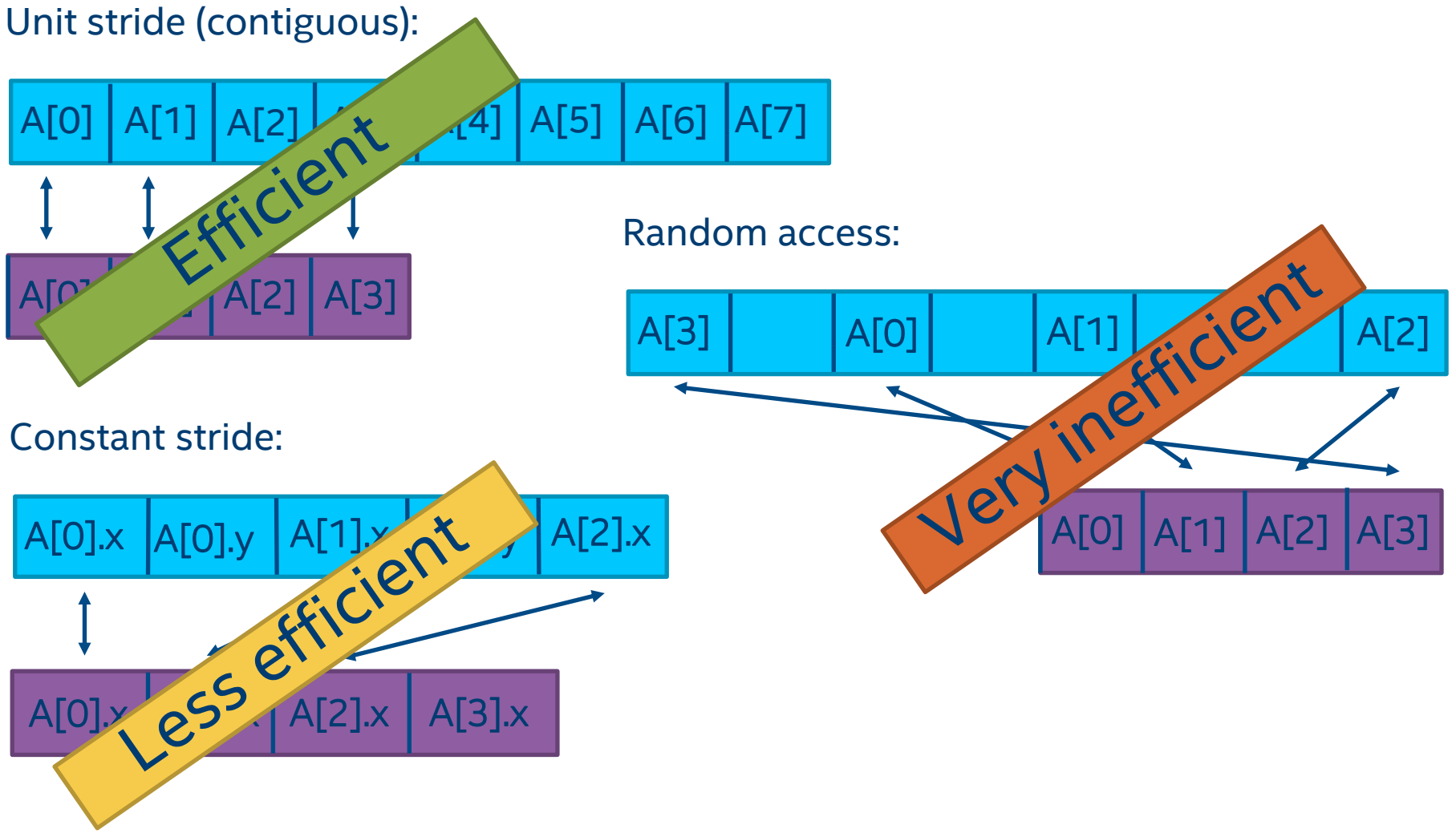
Random access:



Constant stride:

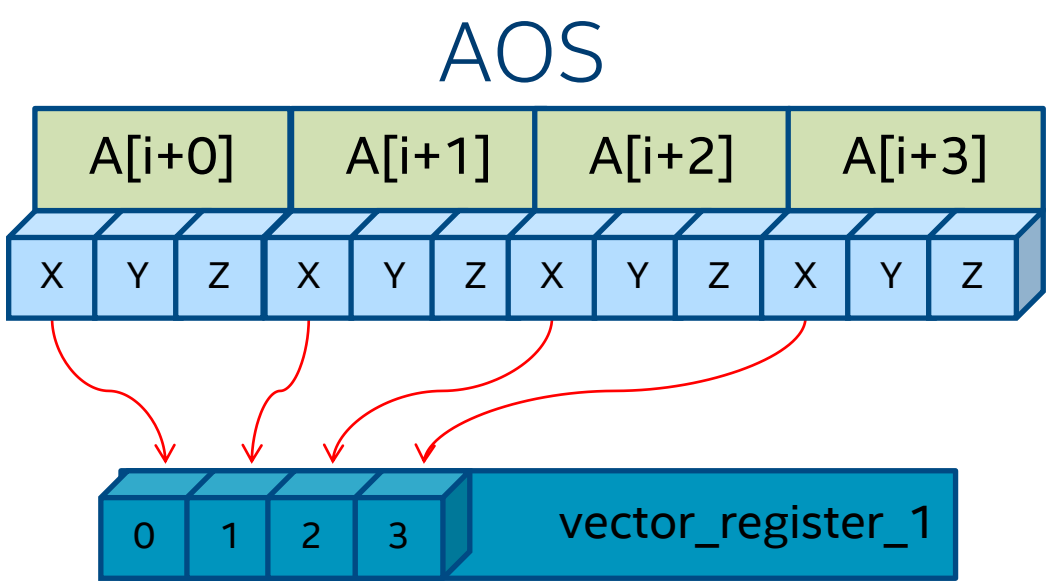


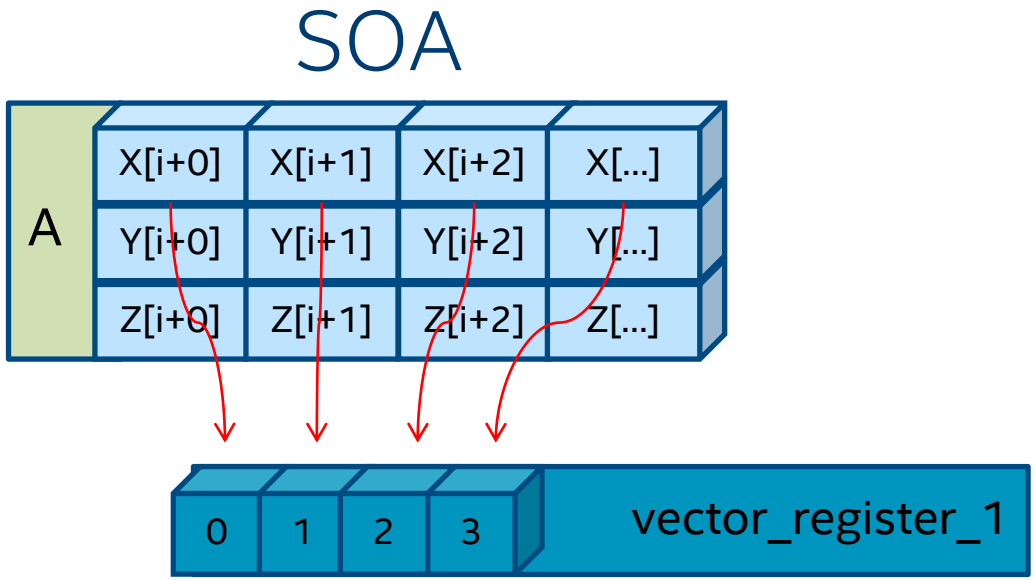
Memory access patterns



What is Intel® SDLT?

- The SIMD Data Layout Template library is a C++11 template library to quick convert *Array of Structures* to *Structure of Arrays* representation
- SDLT vectorizes your code by making memory access contiguous, which can lead to more efficient code and better performance





Intel® SDLT Example

```
#include <iostream>
#define N 1024

typedef struct RGBs {
    float r;
    float g;
    float b;
} RGBTy;

void main()
{
    RGBTy a[N];
    #pragma omp simd
    for (int k = 0; k<N; ++k) {
        a[k].r = k*1.5;  // non-unit stride access
        a[k].g = k*2.5;  // non-unit stride access
        a[k].b = k*3.5;  // non-unit stride access
    }
    std::cout << "k =" << 10 <<
        ", a[k].r =" << a[10].r <<
        ", a[k].g =" << a[10].g <<
        ", a[k].b =" << a[10].b << std::endl;
}
```

AVX-512

```
vscatterdps DWORD PTR [rcx+ymm0*4]{k1}, ymm7          #15.9
vscatterdps DWORD PTR [96+rcx+ymm0*4]{k2}, ymm8       #15.9
vpaddd      ymm5, ymm5, ymm6                          #14.5
vpaddd      ymm4, ymm4, ymm6                          #14.5
add         edx, 16                                   #14.5
lea         rsi, QWORD PTR [4+rsp+rax]                #15.9
vscatterdps DWORD PTR [rsi+ymm0*4]{k3}, ymm9          #16.9
lea         rdi, QWORD PTR [8+rsp+rax]                #15.9
vscatterdps DWORD PTR [96+rsi+ymm0*4]{k4}, ymm10      #16.9
vscatterdps DWORD PTR [rdi+ymm0*4]{k5}, ymm13         #17.9
vscatterdps DWORD PTR [96+rdi+ymm0*4]{k6}, ymm14      #17.9
```

AVX2

```
vextractf128 xmm9, ymm8, 1                            #15.9
vmovss      DWORD PTR [rsp+rcx], xmm8                 #15.9
vmovss      DWORD PTR [48+rsp+rcx], xmm9              #15.9
vextractps DWORD PTR [12+rsp+rcx], xmm8, 1            #15.9
vextractps DWORD PTR [24+rsp+rcx], xmm8, 2            #15.9
vextractps DWORD PTR [36+rsp+rcx], xmm8, 3            #15.9
vmulps      ymm8, ymm0, ymm2                          #17.20
vextractps DWORD PTR [60+rsp+rcx], xmm9, 1            #15.9
vextractps DWORD PTR [72+rsp+rcx], xmm9, 2            #15.9
vextractps DWORD PTR [84+rsp+rcx], xmm9, 3            #15.9
vextractf128 xmm13, ymm12, 1                          #16.9
vextractf128 xmm15, ymm14, 1                          #16.9
vextractf128 xmm2, ymm1, 1                            #17.9
vextractf128 xmm9, ymm8, 1                            #17.9
vmovss      DWORD PTR [4+rsp+rax], xmm12              #16.9
vmovss      DWORD PTR [52+rsp+rax], xmm13             #16.9
vextractps DWORD PTR [16+rsp+rax], xmm12, 1           #16.9
vextractps DWORD PTR [28+rsp+rax], xmm12, 2           #16.9
vextractps DWORD PTR [40+rsp+rax], xmm12, 3           #16.9
```


godbolt.org/z/GTfWWY4P4

Intel® SDLT Example

```
#include <iostream>
#include <sdlt/sdlt.h>
#define N 1024

typedef struct RGBs {
    float r;
    float g;
    float b;
} RGBTy;

SDLT_PRIMITIVE(RGBTy, r, g, b)

void main()
{
    // Use SDLT to get SOA data layout
    sdlt::soa1d_container<RGBTy> aContainer(N);
    auto a = aContainer.access();

    // use SDLT Data Member Interface to access struct members r, g, and b.
    // achieve unit-stride access after vectorization
    #pragma omp simd
    for (int k = 0; k<N; ++k) {
        a[k].r() = k*1.5;  // non-unit stride access
        a[k].g() = k*2.5;  // non-unit stride access
        a[k].b() = k*3.5;  // non-unit stride access
    }
    std::cout << "k =" << 10 <<
        ", a[k].r =" << a[10].r() <<
        ", a[k].g =" << a[10].g() <<
        ", a[k].b =" << a[10].b() << std::endl;
}
```

AVX-512

```
vmulps    ymm5, ymm2, ymm7                    #23.22
vmulps    ymm6, ymm1, ymm7                    #24.22
vmulps    ymm8, ymm0, ymm7                    #25.22
vmulps    ymm3, ymm2, ymm10                   #23.22
vmulps    ymm9, ymm1, ymm10                   #24.22
vmulps    ymm11, ymm0, ymm10                  #25.22
vmovups   YMMWORD PTR [rsi+rcx*4], ymm5       #116.9
vmovups   YMMWORD PTR [rdx+rcx*4], ymm6       #116.9
vmovups   YMMWORD PTR [rax+rcx*4], ymm8       #116.9
vmovups   YMMWORD PTR [32+rsi+rcx*4], ymm3    #116.9
vmovups   YMMWORD PTR [32+rdx+rcx*4], ymm9    #116.9
vmovups   YMMWORD PTR [32+rax+rcx*4], ymm11   #116.9
```


godbolt.org/z/z8dGooGbv

Use function calls inside loop

- Success of in-lining can be verified using the optimization report:
  `-qopt-report=<n> -qopt-report-phase=ipo`
  Intel compilers offer a large set of switches, directives and language extensions to control in-lining globally or locally, e.g.:
  - **`#pragma [no]inline`** (C/C++), **`!DIR$ [NO]INLINE`** (Fortran):
    Instructs compiler that all calls in the following statement can be in-lined or may never be in-lined
  - **`#pragma forceinline`** (C/C++), **`!DIR$ FORCEINLINE`** (Fortran):
    Instructs compiler to ignore the heuristic for in-lining and to inline all calls in the following statement
  - See section “Inlining Options” in compiler manual for full list of options
- IPO offers additional advantages to vectorization
  - Inter-procedural alignment analysis
  - Improved (more precise) dependency analysis

Vectorizable Mathematical Functions

- Calls to most mathematical functions in a loop body can be vectorized using “Short Vector Math Library” (SVML):
  - SVML (**libsvml**) provides vectorized implementations of different mathematical functions
  - From 18.0 version also has scalar implementation
    - Enable scalar math library functions using the Short Vector Math Library (SVML) via

    -fimf-use-svml=true
  - Optimized for latency compared to the VML library component of Intel® MKL which realizes same functionality but optimized for throughput
- Routines in **libsvml** can also be called explicitly, using intrinsics (C/C++)
- These mathematical functions are currently supported:

acos	acosh	asin	asinh	atan	atan2	atanh	cbrt
ceil	cos	cosh	erf	erfc	erfinv	exp	exp2
fabs	floor	fmax	fmin	log	log10	log2	pow
round	sin	sinh	sqrt	tan	tanh	trunc	

Lab

Hands-on exercises

- git clone https://github.com/ivorobts/compiler-optimization.git
- Use Vectorization_Lab.pdf for instructions
  - C++/Fortran – choose what you prefer

- Build – on login node
- Run – on compute node:

```
sbatch -n1 -p cm2_inter -w "i22r07c05s01" -o vec_res.txt job.sh
```

source /opt/intel/oneapi/setvars.sh

Exercise – vectorization/matvector/c

- Go to the folder vectorization/matvector/c
- Build without vectorization:

```
icc -O2 -xAVX -no-vec multiply.c driver.c -o matvector
```

- Run: `./matvector`

Exercise – vectorization/matvector/c

- Go to the folder vectorization/matvector/c
- Build with vectorization:

```
icc -O2 -xAVX multiply.c driver.c -o matvector
```

- Run: `./matvector`
- Please have a look at the optimization report:

```
icc -O2 -xAVX multiply.c driver.c -o matvector -qopt-report=3
```

- *No improvement. Why?*

Exercise – Unit Strides

- Please have a look at the multiply.c
  - What is inc_i and inc_j? Does compiler know their value in compile-time?
- Build with vectorization and check optimization report:

```
icc -O2 -xAVX multiply.c driver.c -o matvector -qopt-report=3
```

- Run: `./matvector`
- *Are loops vectorized now?*

Solution: go to the folder vectorization/matvector/c/solutions/unit-stride

Exercise – Multiversioning

- Check opt report again
  - *Please pay attention at 2 block. What does it mean?*

```
LOOP BEGIN at multiply.c(32,9)
   remark #15344: loop was not vectorized: vector dependence
   remark #15346: vector dependence: assumed FLOW dependence
   remark #25439: unrolled with remainder by 2
LOOP END

LOOP BEGIN at multiply.c(32,9)
<Remainder>
LOOP END
```

- Build with vectorization and check optimization report:

```
icc -O2 -xAVX multiply.c driver.c -o matvector -qopt-report=3
```

- Run: **`./matvector`**
- *Are loops vectorized now?*

Solution: go to the folder vectorization/matvector/c/solutions/multiversioning_off

Exercise – Vectorization of Inner Loop

- Please have a look at the compiler report for multiply.c
  - *The compiler needs additional information to recognize the assumed dependencies*

```
LOOP BEGIN at multiply.c(32,9)
   remark #15344: loop was not vectorized: vector dependence prevents vectorization. First dependence is shown below.
   remark #15346: vector dependence: assumed FLOW dependence between b[i] (33:13) and b[i] (33:13)
   remark #25439: unrolled with remainder by 2
LOOP END
```

- Can we provide information without modifying the code?

`icc -O2 -xAVX multiply.c driver.c -o matvector -fargument-noalias`

- Run: **`./matvector`**
- *Are loops vectorized now?*

Solution: -fargument-noalias option

Exercise – Vectorization of Inner Loop

- Please have a look at the compiler report for multiply.c
  - *The compiler needs additional information to recognize the assumed dependencies*

```
LOOP BEGIN at multiply.c(32,9)
   remark #15344: loop was not vectorized: vector dependence prevents vectorization. First dependence is shown below.
   remark #15346: vector dependence: assumed FLOW dependence between b[i] (33:13) and b[i] (33:13)
   remark #25439: unrolled with remainder by 2
LOOP END
```

- Can we provide information for a specific **loop**?
  - Please, add **`#pragma ivdep`** to the appropriate place

```
icc -O2 -xAVX multiply.c driver.c -o matvector
```

- Run: **`./matvector`**
- *Are loops vectorized now?*

Solution: go to the folder vectorization/matvector/c/solutions/ivdep

Exercise – Vectorization of Inner Loop

- Please have a look at the compiler report for multiply.c
  - *The compiler needs additional information to recognize the assumed dependencies*

```
LOOP BEGIN at multiply.c(32,9)
   remark #15344: loop was not vectorized: vector dependence prevents vectorization. First dependence is shown below.
   remark #15346: vector dependence: assumed FLOW dependence between b[i] (33:13) and b[i] (33:13)
   remark #25439: unrolled with remainder by 2
LOOP END
```

- Can we provide information for a specific **pointer**?
  - Please, add **`restrict`** keyword to the appropriate place and add **`-restrict`** option

```
icc -O2 -xAVX multiply.c driver.c -o matvector -restrict
```

- Run: **`./matvector`**
- *Are loops vectorized now?*

Solution: go to the folder vectorization/matvector/c/solutions/restrict

Exercise – Alignment Improvements

- Please have a look at driver.c and check how **`a, b`** and **`c`** are allocated
  - Can we guarantee alignment of these arrays? How?
- *Are loops vectorized now?*

Solution: go to the folder vectorization/matvector/c/solutions/align

Exercise – Alignment Improvements

- Please have a look at driver.c and check how **`a, b`** and **`c`** are allocated
  - Can we guarantee alignment of these arrays? How?
    - Attributes?
    - Compiler directives?
- *Are loops vectorized now?*

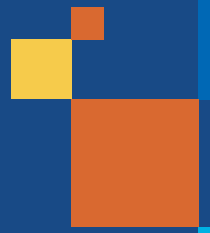
Solution #1: go to the folder vectorization/matvector/c/solutions/align
Solution #2: go to the folder vectorization/matvector/c/solutions/assume_aligned

Exercise – Alignment Improvements

- Advanced:
  - Please look closer to each row of a matrix
  - Can we apply the knowledge about SIDM vector size?
    - *What happens with the remainder elements?*
- *Are loops vectorized now?*

Solution: go to the folder vectorization/matvector/c/solutions/align

NBody Demo

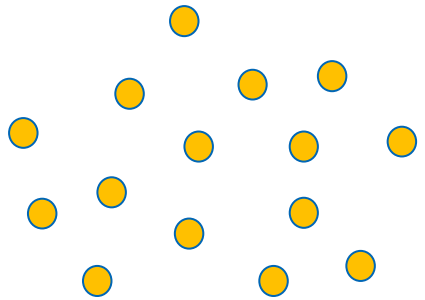


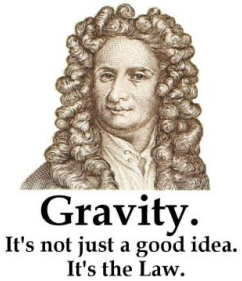
intel®

Nbody gravity simulation

- Let's consider a distribution of point masses located at `r_1,…,r_n` and have masses `m_1,…,m_n`
- We want to calculate the position of the particles after a certain time interval using the Newton law of gravity

```
struct Particle
{
  public:
    Particle() { init();}
    void init()
    {
      pos[0] = 0.; pos[1] = 0.; pos[2] = 0.;
      vel[0] = 0.; vel[1] = 0.; vel[2] = 0.;
      acc[0] = 0.; acc[1] = 0.; acc[2] = 0.;
      mass   = 0.;
    }
    real_type pos[3];
    real_type vel[3];
    real_type acc[3];
    real_type mass;
};
```





$$\vec{F}_{ij} = \frac{G m_i m_j}{\left|\vec{r}_j - \vec{r}_i\right|^3}\left(\vec{r}_j - \vec{r}_i\right)$$

$$\vec{F} = m\vec{a} = m\frac{d\vec{v}}{dt} = m\frac{d^2\vec{x}}{dt^2}$$

Nbody kernel implementation

GSimulation.cpp:

```
...
for (i = 0; i < n; i++) {                                    // update acceleration
     for (j = 0; j < n; j++) {
          real_type distance, dx, dy, dz;
          real_type distanceSqr = 0.0;
          real_type distanceInv = 0.0;

          dx = particles[j].pos[0] - particles[i].pos[0];
          dy = particles[j].pos[1] - particles[i].pos[1];
          dz = particles[j].pos[2] - particles[i].pos[2];

          distSqr = dx*dx + dy*dy + dz*dz + softeningSquared;
          distInv = 1.0 / sqrt(distanceSqr);
          particles[i].acc[0] += dx * G * particles[j].mass * distInv * distInv * distInv;
          particles[i].acc[1] += …
          particles[i].acc[2] += …
     }
}
...
```

nbody-demo/ver0

- Use different compiler options and try to target the underlying architecture:
  - -xCORE-AVX2 or -xHost
  - -O3
  - -ipo
  - -prof-gen and -prof-use
  - -parallel
- Explain why some options don‘t bring additional speed-up. Try some more tests here:

git clone https://github.com/fbaru-dev/nbody-demo.git

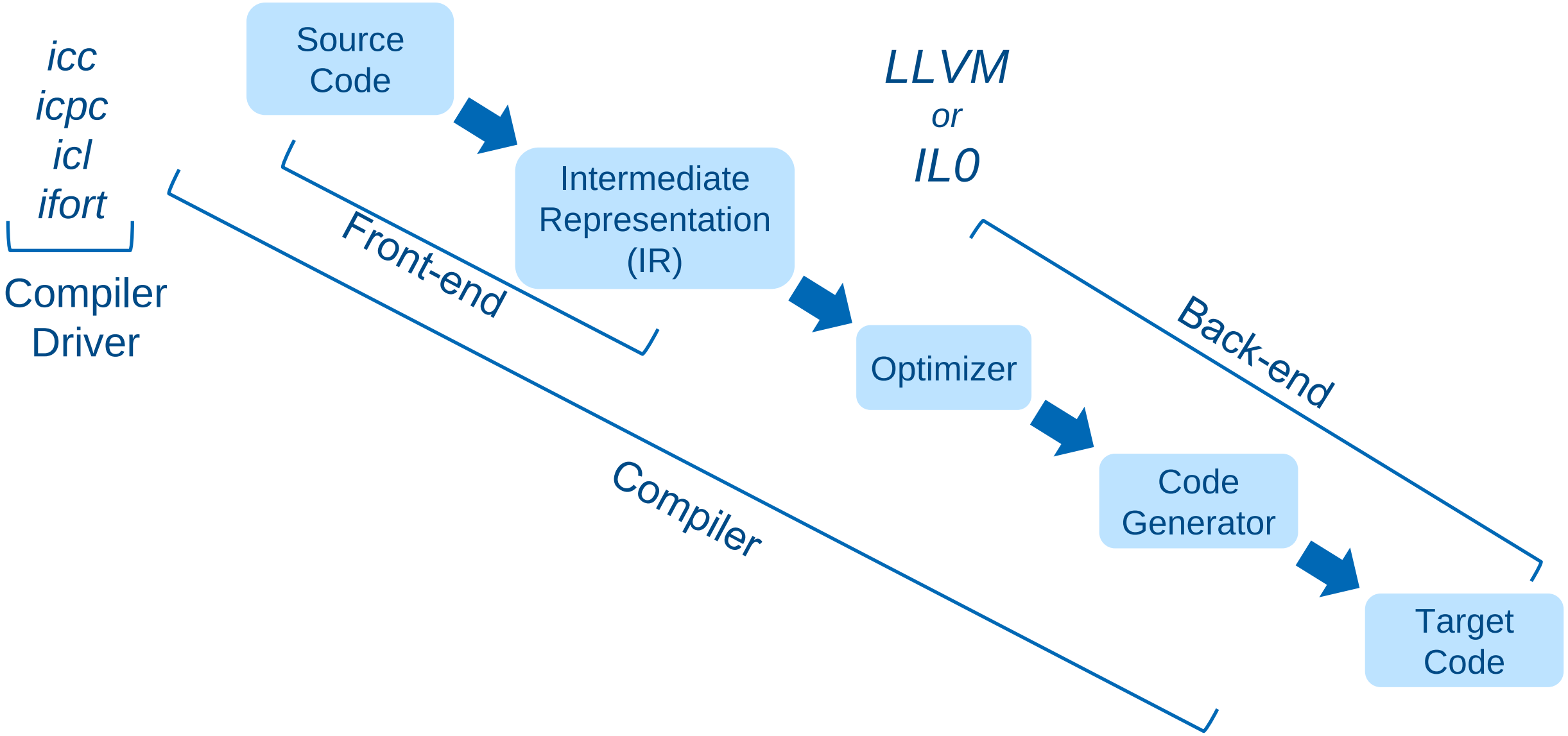
nbody-demo/ver4

- Go to the folder nbody-demo/ver4
- Type *make* to compile code.
- Type *make run* to run the test and measure the timing.
- Please have a look at the compiler report.

LLVM-based Intel Compilers

intel®

Compiler Architecture – simplified model



Intel Compiler Transition: Classic to LLVM

Start Your Migration Now

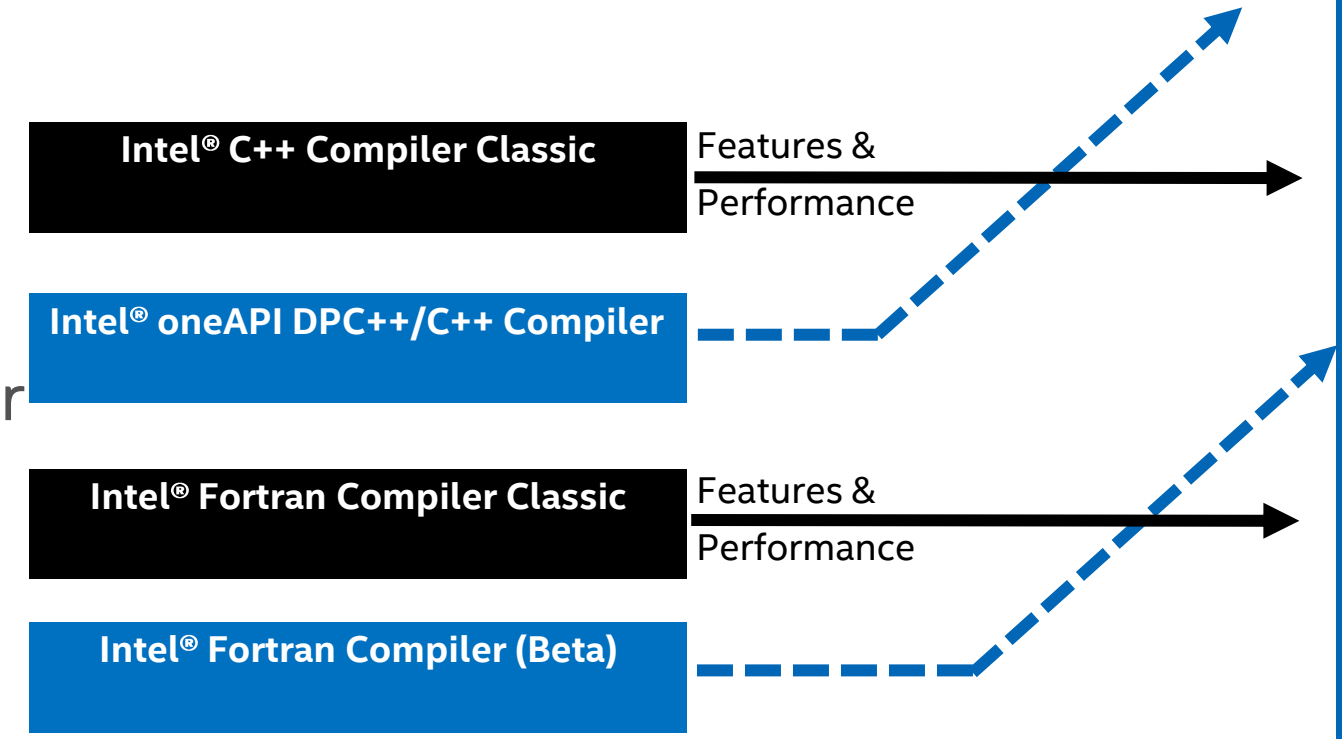
- Expand to XPUs
- Modern LLVM Infrastructure

Intel® oneAPI DPC++/C++ Compiler

- Use for all new projects
- Migrate legacy projects

Intel® Fortran Compiler (Beta)

- Test drive now & Provide feedback
- Prepare for migration



Performance, Quality, and Support Continues

Intel® Compilers

Intel Compiler	Target	OpenMP Support	OpenMP Offload Support	Included in oneAPI Toolkit
Intel® C++ Compiler Classic, IL0 ***icc/icpc/icl***	CPU	Yes	No	HPC
Intel® Fortran Compiler Classic, IL0 ***ifort***	CPU	Yes	No	HPC
Intel® Fortran Compiler(Beta), LLVM ***ifx***	CPU, GPU	Yes	Yes	HPC
Intel® oneAPI DPC++/C++ Compiler, LLVM ***dpcpp***	CPU, GPU, FPGA*	Yes	Yes	Base
Intel® oneAPI DPC++/C++ Compiler, LLVM ***icx/icpx***	CPU GPU*	Yes	Yes	Base

Cross Compiler Binary Compatible and Linkable!

tinyurl.com/oneapi-standalone-components

What is ICX?

- Close collaboration with Clang*/LLVM* community
- ICX is Clang front-end (FE), LLVM infrastructure
  - PLUS Intel proprietary optimizations and code generation
- Clang FE pulled down frequently from open source, kept current
  - Always up to date in ICX
  - We contribute! Pushing enhancements to both Clang and LLVM
- Enhancements working with community – better vectorization, opt-report, for example

tinyurl.com/blog-on-icx

Major Changes Overview

tinyurl.com/icc-to-icx-migration-guide

- Written for ICC to ICX transition
- LLVM is a different compilation technology. EXPECT differences
- Options: undocumented IL0 options all ignored
  - icx -qnextgen-diag option to get a list of supported and unsupported options
- IPO/PGO - new LLVM LTO/PGO instead
- FP Model is not the same
- Intrinsics are handled very differently
- C/C++ Pragmas – a lot of Intel proprietary ones not supported
  - enable -Wunknown-pragmas to warn on unsupported pragmas
- __INTEL_LLVM_COMPILER is defined instead of __INTEL_COMPILER

Major Changes Overview

- Not supported features:
  - Auto-parallelization (-parallel option)
  - Intel Cilk™ Plus (pragma simd) replaced by OpenMP pragmas
  - -ax not implemented yet
- Optimization reports
- Analyzers may be affected
  - Advisor needs opt reports for Vectorization Advisor
  - VTune can't get code lines/clear symbols for OpenMP regions
- no macOS support

Fortran Essentials and Specifics

tinyurl.com/ifort-to-ifx-migration-guide

Intel® Fortran Compiler Classic

- ifort - Intel Fortran front end + IL0 Intel proprietary backend
- v2021.x in oneAPI HPC Toolkit
- CPU only, traditional compiler. NO OFFLOAD TO GPU
- ***Recommended Fortran Compiler for 2021 for Production Use***

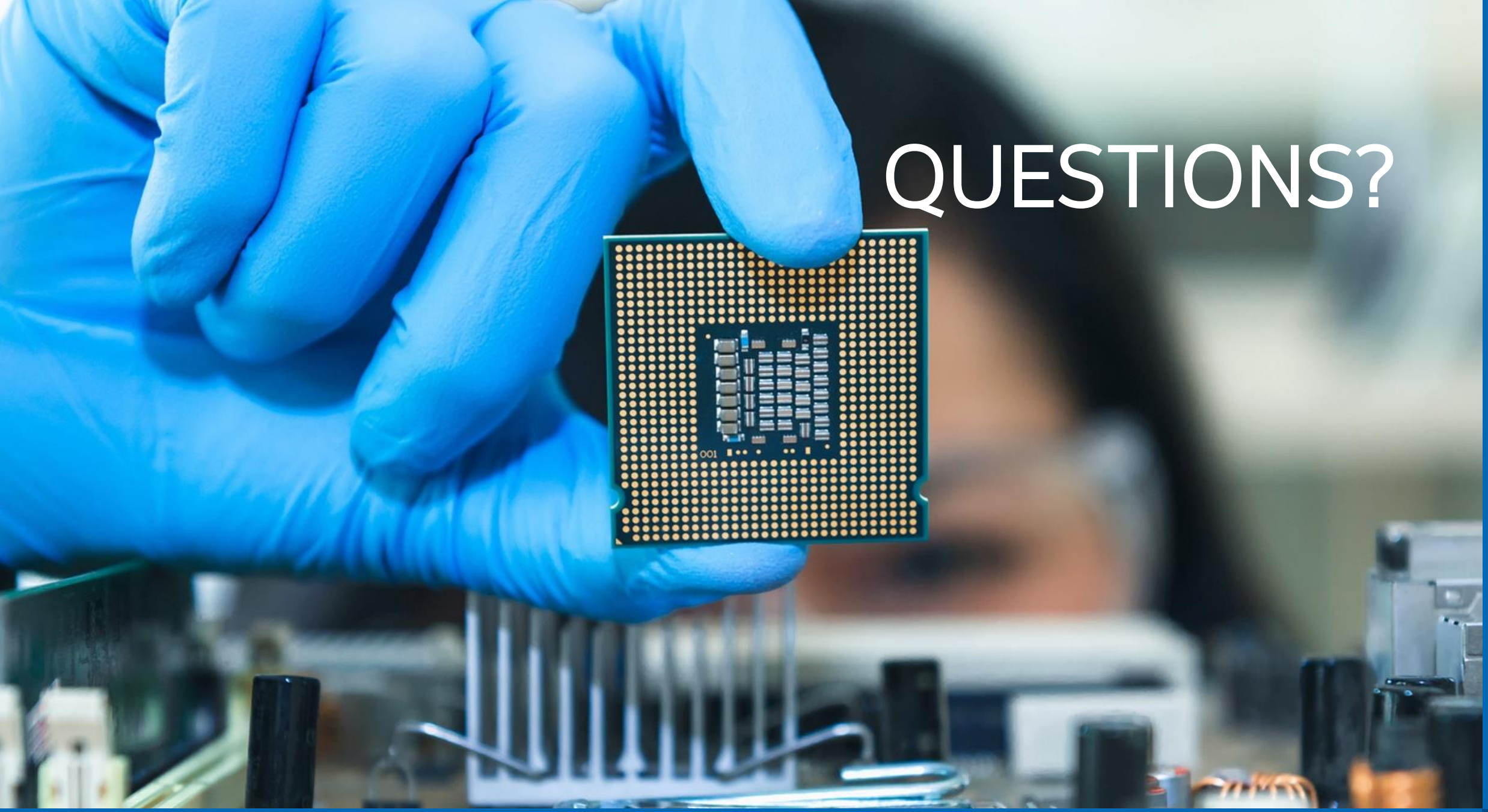
Intel® Fortran Compiler (Beta)

- ifx – Intel Fortran front end + LLVM backend
- Same “ifort” front-end v2021.x
- Offload to Intel GPUs via OpenMP
- Binary compatible with DPCPP, ICX, ICC, IFORT

IFX Status, Setting Expectations

- BETA (Nov 2021)
  - IFX CORE Fortran LANGUAGE
    - F77, F90/95, majority of F03 and F08
    - Use **`-stand f03`** if you want warnings for features not in F2003
    - Use **`-stand f08 –warn errors`** options to abort if any F08 or above detected.
  - IFX OpenMP Support
    - All OpenMP 4.5 features supported (2021.4 version)
    - Substantial part of OpenMP5.0/5.1

software.intel.com/content/www/us/en/develop/articles/fortran-language-and-openmp-features-in-ifx.html


QUESTIONS?

Notices & Disclaimers

- This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice.
- 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.
- 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 visit www.intel.com/benchmarks.
- INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.
-

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