C++ and Fortran Compilers for Intel & AMD architectures

Georg Zitzlsberger georg.zitzlsberger@vsb.cz
Radim Vavřík radim.vavrik@vsb.cz
Jakub Beránek jakub.beranek@vsb.cz

IT4Innovations
national supercomputing center

14th of January 2019

Agenda

Exercise 1 - Numerical Stability

Exercise 2 - Processor Dispatching

Summary

Exercise 1 - Numerical Stability

Exercise 1 - Numerical Stability

Numerical stability as a problem:

- Run-to-run variations:

```
> ./my_app
Result: 1.0109888
> ./my_app
Result: 1.0109889
```

- Run between different systems:

System A

```
> ./my_app
Result: 1.0109888
```

System B

```
> ./my_app
Result: 1.0109889
```

- Root causes:
  - FP numbers have finite resolution and
  - Rounding is done for each (intermediate) result

Exercise 1 - Numerical Stability

What has effect on numerical stability?

- Selection of algorithm:
  Conditional numerical computation for different systems and/or input data can have unexpected results.
- Non-deterministic task/thread scheduler:
  Asynchronous task/thread scheduling has best performance but reruns use different threads.
- Alignment (heap & stack):
  If alignment is not guaranteed and changes between reruns the data sets could be computed differently (e.g. vector loop prologue & epilogue of unaligned data).

$\Rightarrow$ User controls those (direct or indirect)

Exercise 1 - Numerical Stability

Order of FP operations has impact on rounded result, e.g.:
$(a + b) + c \neq a + (b + c)$

- $2 - 63 + 1 + -1 = 2 - 63$ (mathematical result)
- $(2 - 63 + 1) + -1 \approx 0$ (correct IEEE result)
- $2 - 63 + (1 + -1) \approx 2 - 63$ (correct IEEE result)

Effects of optimizations:

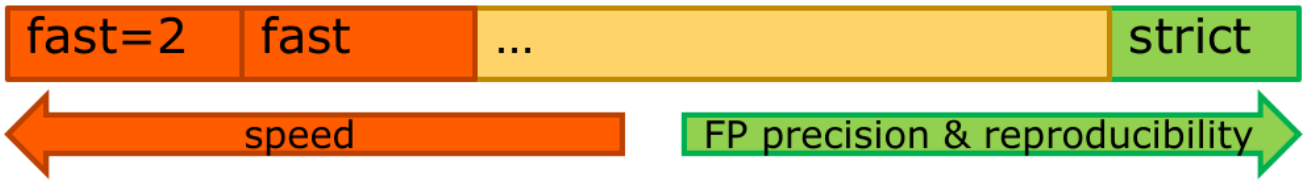
- Constant folding: $X + 0 \Rightarrow X$ or $X * 1 \Rightarrow X$
- Multiply by reciprocal: $A/B \Rightarrow A * (1/B)$
- Approximated transcendental functions (e.g. `sqrt`, `sin`)
- For SIMD instructions: flush-to-zero or contractions (FMA)
- Different code paths (e.g. SIMD vs. non-SIMD)
- ...

$\Rightarrow$ Subject of Optimizations by Compiler & Libraries

Exercise 1 - Numerical Stability

Intel C++/Fortran Compiler:

- Control with `-fp-model`:



(Image: Intel)

- Changes:
  - Value safety (associativity)
  - Expression evaluation (interim rounding)
  - Exceptions on or off
  - FP environment access (e.g. C++ `fenv_t`)
  - FP contractions (FMA)
- Default is `fast`, `strict` might fall back to x87/no vectorization

Exercise 1 - Numerical Stability

Example:

```
#include <iostream>
#define N 100
int main()
{
  float a[N], b[N];
  float c = -1., tiny = 1.e-20F;
  for (int i=0; i<N; i++) a[i]=1.0;
  for (int i=0; i<N; i++) {
    a[i] = a[i] + c + tiny;
    b[i] = 1/a[i];
  }
  std::cout << "a␣=␣" << a[0]
            << "␣b␣=␣" << b[0]
            << "\n";
}
```

- `-fp-model fast` (default):
  `a = 0, b = -nan`
- `-fp-model precise` or `-fp-model strict`:
  `a = 1e-20, b = 1e+20`

⇒ Disabled reassociation which impacts also performance

Exercise 1 - Numerical Stability

- Options for either code generation or math libraries used
- Code generation:
  - `-[no-]prec-div` and `-[no-]prec-sqrt`: Improve precision (default off)
  - `-[no-]fma`: Turn on/off FMA (default on)
  - `-[no-]fprotect-parens`: Honor parentheses (default off)
  - ...
- Math libraries:
  - `-fimf-precision=[high|medium|low]`: Accuracy of math library functions
  - `-fimf-arch-consistency=true`: Consistent results on different processor types of same architecture
  - `-fimf-use-svml`: Use SVML also for scalar math (new in 18.0)
  - More `-fimf-*` options available ...

More information can be found here

Exercise 1 - Numerical Stability

- Various options between default (favoring performance) and strict model (best reproducibility)
- Best real life trade-off:
  `-fp-model precise -fp-model source` (or for 17.0+: `-fp-model consistent`)
  Approx. 12-15% slower performance for SPECCPU2006fp
- Don't mix math libraries from different compiler versions!
- Using different processor types (of same architecture), specify: `-fimf-arch-consistency=true`
- **Use `-fimf-use-svml -fp-model consistent` for 18.0 and later**

Very good article about numerical stability can be found here

Exercise 1 - Numerical Stability

GCC:

- Fast math `-ffast-math` (off by default):

Category	Flags	Comments
Trap handlers and exceptions	`-fno-trapping-math`, `-fno-signaling-nans`	IEEE standard recommends that implementations allow trap handlers to handle exceptions like divide by zero and overflow. This flags assumes that no use-visible trap will happen.
Rounding	`-fno-rounding-math`	IEEE has four rounding modes. This flag assumes that the rounding mode is round to nearest.
Languages and compilers	`-funsafe-math-optimizations` (incl. `-fassociative-math`)	Due to roundoff errors the associative law of algebra do not necessary hold for floating point numbers and thus expressions like `(x + y) + z` are not necessary equal to `x + (y + z)`.
Special quantities (Nans, signed zeros and infinity)	`-ffinite-math-only`, `-fno-signed-zeros`	
Flush to zero		

  Equivalent to Intel's `-fp-model=consistent`
- To get equivalent to Intel's `-fp-model=strict` use: `-frounding-math -fsignaling-nans`
- Vectorize with `-ftree-vectorize` (only on with `-O3`)

See ▸ GCC Wiki for more information

Exercise 1 - Numerical Stability

LLVM/Clang:

- Same as for GCC, fast math `-ffast-math` (off by default)
- Vectorization on by default; turn off with:
  - Loop vectorization: `-fno-vectorize`
  - Superword-level parallelism: `-fno-slp-vectorize`

See ▸ Auto-Vectorization in LLVM for more information

A word on AMD compilers:

- ▸ AMD x86 Open64 Compiler Suite deprecated 2013
- Successor is ▸ AMD Optimizing C/C++ Compiler (AOCC):
  - Latest version is 1.3.0 with "Zen" support
  - Based on LLVM/Clang 7.0 with DragonEgg (Fortran) and Flang (alpha) support
  - Optimized libraries including ▸ AMD LibM (v3.2.2)

Exercise 1 - Numerical Stability

Vectorized math libraries:

- High level libraries like FFTW, OpenBLAS, ATLAS, Intel Math Kernel Library, etc.
- Low level extensions of `libm`
  - Intel Short Vector Math Library (SVML):
    - Default by Intel C++/Fortran Compiler
    - Has its own ABI (proprietary)
    - Calls to symbols like `__svml_sin8_z0` (`sin(...)` with 8 elements of AVX512 vector register)
  - Glibc `libmvec`:
    - Default by GCC and LLVM/Clang with Glibc 2.22+
    - Used automatically - only add `-lmvec` for static builds
    - Can be used with OpenMP (GCC 4.9.0+) or automatically (Glibc 2.23+ and GCC 6.1+)
    - ABI follows ▸ Vector Function ABI Specification for OpenMP SIMD with an extension

Exercise 1 - Numerical Stability

- cont'd low level extensions of `libm`:
  - AMD Core Math Library (ACML)
    - Last version ACML 6.0.6 (2014); no development anymore(?)
    - Only up to AVX (and FMA4)
    - Better alternative: `libmvec`
  - OpenPower Mathematical Acceleration Subsystem (MASS) (more on that later)
- GCC option `-mveclibabi=[svml|acml]`
  - Requires also: `-ftree-vectorize` and `-funsafe-math-optimizations` (or `-ffast-math`)
  - Default: `libmvec`
- For GFortran, `libmvec` is work in progress
- LLVM/Clang[1] option `-fmvec=[Accelerate|SVML|none]`

[1] No support for `libmvec` yet

Exercise 1 - Numerical Stability

Example (`sin.c`):

```
#include <math.h>

int N = 3200;
double b[3200];
double a[3200];

int main (void)
{
    int i;
    for (i = 0; i < N; i += 1)
        b[i] = sin (a[i]);
    return (0);
}
```

Compile (GCC 6.1+ and Glibc 2.23+):

```
$ gcc -O2 -ftree-loop-vectorize -ffast-math
      -lm -mavx2 sin.c -S
$ cat sin.s
...
.L4:
vmovupd         (%r15,%rbx), %xmm0
addl            $1, %r14d
vinsertf128     $0x1, 16(%r15,%rbx), %ymm0, %ymm0
call            _ZGVdN4v_sin@PLT
vmovups         %xmm0, (%r12,%rbx)
vextractf128    $0x1, %ymm0, 16(%r12,%rbx)
addq            $32, %rbx
cmpl            %r14d, -52(%rbp)
ja              .L4
...
```

Attention:

GCC w/ SVML: `vmldExp2`, `vmldLn2`, `vmldLog102`, `vmldPow2`, `vmldTanh2`, `vmldTan2`, `vmldAtan2`, `vmldAtanh2`, `vmldCbrt2`, `vmldSinh2`, `vmldSin2`, `vmldAsinh2`, `vmldAsin2`, `vmldCosh2`, `vmldCos2`, `vmldAcosh2`, `vmldAcos2`, `vmlsExp4`, `vmlsLn4`, `vmlsLog104`, `vmlsPow4`, `vmlsTanh4`, `vmlsTan4`, `vmlsAtan4`, `vmlsAtanh4`, `vmlsCbrt4`, `vmlsSinh4`, `vmlsSin4`, `vmlsAsinh4`, `vmlsAsin4`, `vmlsCosh4`, `vmlsCos4`, `vmlsAcosh4`, `vmlsAcos4`

GCC w/ ACML: `__vrd2_sin`, `__vrd2_cos`, `__vrd2_exp`, `__vrd2_log`, `__vrd2_log2`, `__vrd2_log10`, `__vrs4_sinf`, `__vrs4_cosf`, `__vrs4_expf`, `__vrs4_logf`, `__vrs4_log2f`, `__vrs4_log10f`, `__vrs4_powf`

⇒ Only SSE!

Exercise 1 - Numerical Stability

Exercises:

1. Compile the provided code `fp_stability.[cpp|f90]` with Intel C++/Fortran, GCC/GFortran & LLVM/Clang:
   - Compile one version optimized for speed and one with "correct" results; what are the differences?
   - Change the code and compiler options to retain correct results with optimizations (with all compilers).
2. Using vectorized math libraries:
   - Compile the provided code (`vec.[cpp|f90]`) with the Intel C++/Fortran Compiler and verify the use of SVML
   - Compile the same code with GCC and confirm use of `libmvec`.
   - For GCC, change from `libmvec` to `svml` or `acml`, for LLVM/Clang, change to `SVML`. What is the difference?

Exercise 2 - Processor Dispatching

Exercise 2 - Processor Dispatching

What we want:

- Manage different SIMD extension sets in one executable
- Run exeutable on wide range of processor generations (incl. control of numerical stability)
- Support Intel and AMD processors with the same executable

What it does not:

- Cannot mix different instruction set architectures (ISAs) in one executable (e.g. Power9 with AMD64/x86-64)
- No emulation of newer/future ISAs (use the tool Intel Software Development Emulator for this)
- Support by compilers others than Intel C++/Fortran Compiler

Exercise 2 - Processor Dispatching

Three different approaches:

- Automatic CPU Dispatching
  - Pro:
    Easy to use (only compiler options needed)
  - Con:
    Coarse grained as compiler does the differntiation
- Manual CPU Dispatching
  - Pro:
    Fine grained control down to function level
  - Con:
    Only supports Intel processors
- Vendor-neutral CPU Dispatching
  - Pro:
    Works for both Intel and AMD processors
  - Con:
    Requires new API

Exercise 2 - Processor Dispatching

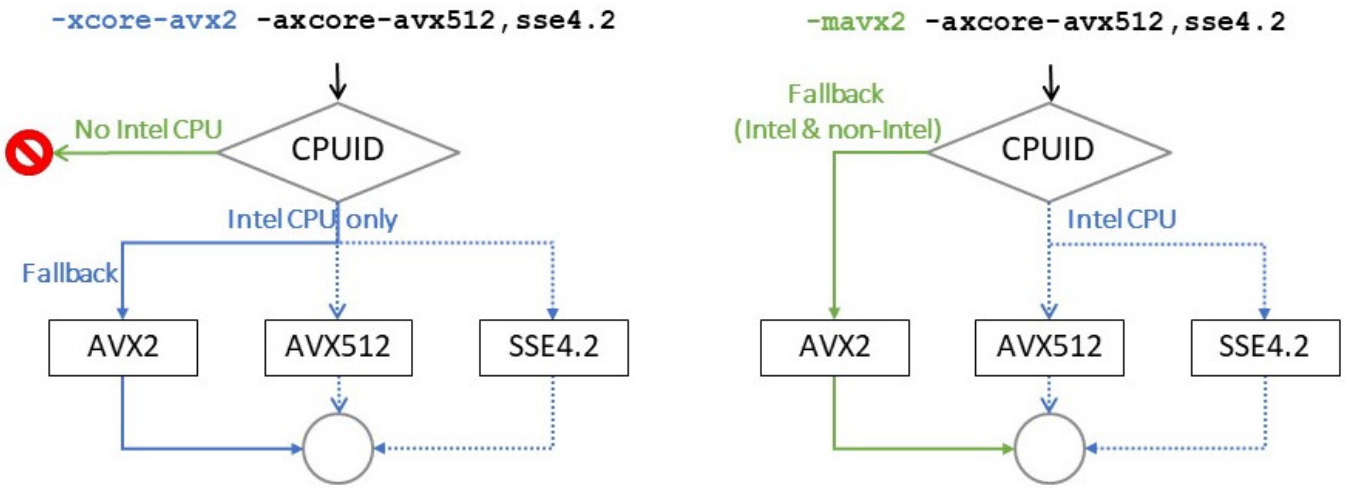
Using automatic CPU dispatching:

1. Select baseline:
   - Intel only `-x<Feature>`:
     Adds CPU test during link stage which only works for Intel processors
     Example:
     `icc app.c -xcore-avx2`
   - Compatible `-m<Feature>`:
     No CPU test is added and user is responsible to run on compatible processors
     Example:
     `icc app.c -mavx2`
2. Extend with Intel only features `-ax<Features>`:
   Add Intel only processor SIMD features
   Example:
   `icc app.c -mavx2 -axsse4.2,core-avx512`

For list of features see Intel® C++ Compiler Developer Guide and Reference

Exercise 2 - Processor Dispatching

How automatic CPU dispatching works:



Note:

For `-mavx2` options, there is no test for support of AVX2!

Example for Skylake's AVX512[2]:

```
$ icc -xcore-avx512 app.c -o app && ./app

Please verify that both the operating system and the processor support
Intel(R) AVX512DQ, AVX512F, AVX512CD, AVX512BW, AVX512VL and CLWB instructions.

$ icc -march=skylake-avx512 app.c -o app && ./app
Illegal instruction
```

[2] `-m` options are a family of options including `-march=`

Exercise 2 - Processor Dispatching

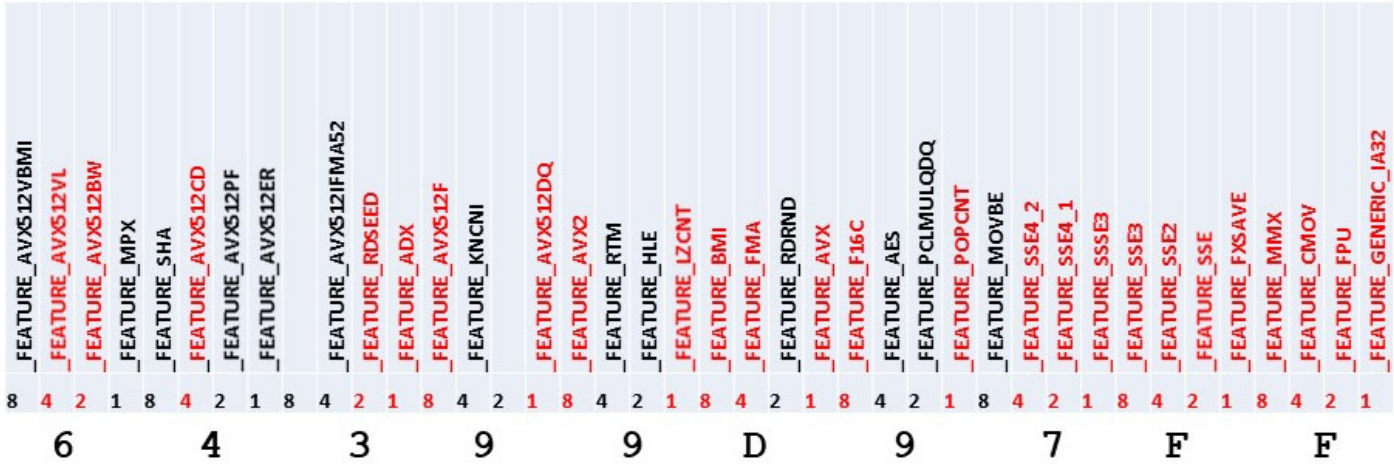
Behind the curtain:

```
$ icc -xsse4.2 -axcore-avx512 app.c -S
app.c(2): (col. 1) remark: main has been targeted for automatic cpu dispatch

$ cat app.s
...
# main
..B1.1:
        pushq     %rsi
        movq      $0x20064199d97ff , %rdx
..B1.3:
        movl      __intel_cpu_feature_indicator(%rip), %rax
        andq      %rdx, %rax
        cmpq      %rdx, %rax
        jne       ..B1.6
..B1.4:
        addq      $8, %rsp
        jmp main.V # AVX512
..B1.6:
        testb $1, __intel_cpu_feature_indicator(%rip)
        je        ..B1.8
..B1.7:
        addq      $8, %rsp
        jmp main.A # SSE4.2
..B1.8:
        call      *__intel_cpu_features_init@GOTPCREL(%rip)
        jmp       ..B1.3
...
```

Exercise 2 - Processor Dispatching

CPUID masks:



- On non-Intel processor: just the generic bit is set (`_FEATURE_GENERIC_IA32`)
- Better optimization only with `-m` options (for automatic CPU dispatching)
- Full list can be found in `immintrin.h` of Intel C++ Compiler

Exercise 2 - Processor Dispatching

Manual CPU dispatching:

- Possible to provide specific high level code for different processors
- Different processors might also require different algorithms
- Usage is simple:
  - Declare a function to be multi-versioned[3]:

```
__attribute__((cpu_dispatch(generic, ...)))
void my_func() {};
```

  - Define every version explicitly:

```
__attribute__((cpu_specific(generic)))
void my_func() {
...
}
```

[3] `__declspec` can be also used instead of `__attribute__`

Exercise 2 - Processor Dispatching

Manual CPU dispatching example:

```
#include <stdio.h>

// need to create specific function versions
__attribute__((cpu_dispatch(generic, core_5th_gen_avx)))
void dispatch_func() {};

__attribute__((cpu_specific(generic)))
void dispatch_func() {
  printf("Code for non-Intel processors and generic Intel\n");
}

__attribute__((cpu_specific(core_5th_gen_avx)))
void dispatch_func() {
  printf("Code for 5th generation Intel Core processors goes here\n");
}
int main() {
  dispatch_func();
  printf("Return from dispatch_func\n");
  return 0;
}
```

Exercise 2 - Processor Dispatching

Manual CPU dispatching CPUID arguments:

Argument	Description
`generic`	Other Intel processors for IA-32 or Intel® 64 architecture or compatible processors not provided by Intel Corporation
`core_4th_gen_avx`	4th generation Intel® Core™ processor family with support for Intel® Advanced Vector Extensions 2 (Intel® AVX2) including the RDRND instruction
`haswell`	This is a synonym for `core_4th_gen_avx`
`skylake`	Intel® microarchitecture code name Skylake. This keyword targets the Client CPU without support for Intel® AVX-512 instructions
`skylake_avx512`	Intel® microarchitecture code name Skylake. This keyword targets the Server CPU with support for Intel® AVX-512 instructions
`mic_avx512`	2nd generation Intel® Xeon Phi™ processor family with support for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions, Intel® AVX-512 Conflict Detection instructions, Intel® AVX-512 Exponential and Reciprocal instructions, Intel® AVX-512 Prefetch instructions . . .

For full list see Intel® C++ Compiler Developer Guide and Reference

Exercise 2 - Processor Dispatching

Manual CPU dispatching is typically used with:

- Inline assembler
- Compiler intrinsic functions (see Intel® Intrinsics Guide)
- Manual specialization for vector lengths (e.g. if dimensions are known)
- Fortran users can move dispatching to C functions and call back to Fortran code

$\Rightarrow$ More control over generated object code!

However:
What to do if specialization should also work for non-Intel processors?

Exercise 2 - Processor Dispatching

Vendor-neutral CPU dispatching:

- Support of Intel and non-Intel processors
- Get same benefits from manual processor dispatching
- Usage is simple:
  - Test whether CPU feature is available on executing processor:
    `int _may_i_use_cpu_feature(__int64)`
  - Tell compiler that a code section should use specific features:
    `void _allow_cpu_features(__int64)`

Feature	Description
_FEATURE_GENERIC_IA32	Fallback
_FEATURE_SSE4_2	SSE4.2
_FEATURE_AVX	AVX
_FEATURE_AVX2	AVX2
_FEATURE_AVX512F	AVX512 Foundation

For list of features, see Intel® C++ Compiler Developer Guide and Reference

Exercise 2 - Processor Dispatching

Vendor-neutral CPU dispatching example:

```
#include <stdio.h>
#include <immintrin.h>

void run_avx2() {
    _allow_cpu_features(_FEATURE_AVX2);
    printf("AVX2\n");
}
void run_sse4_2() {
    _allow_cpu_features(_FEATURE_SSE4_2);
    printf("SSE4.2\n");
}
void run_fallback() {
    _allow_cpu_features(_FEATURE_GENERIC_IA32);
    printf("Fallback\n");
}
void my_dispatch() {
    if( _may_i_use_cpu_feature(_FEATURE_AVX2) )
        run_avx2();
    else if( _may_i_use_cpu_feature(_FEATURE_SSE4_2) )
        run_sse4_2();
    else
        run_fallback();
}
int main(int argc, char **argv) {
    my_dispatch();
    return 0;
}
```

Exercise 2 - Processor Dispatching

Exercises:

1. Multi-version the given example for AVX2 (baseline) and AVX512 (Skylake-SP) using:
   - Automatic CPU dispatching
   - Manual CPU dispatching
   - Vendor-neutral CPU dispatching

   Use Intel Software Development Emulator to execute the AVX512 code paths (option `-skx`).
2. For the vendor-neutral case, compile the AVX2 and AVX512 implementations with a different compiler than Intel C++/Fortran Compiler. What is important for building and linking?

Summary

We have seen examples on how...

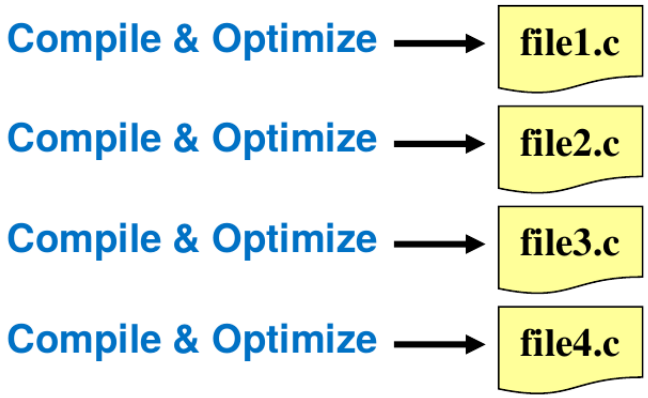
- FP numerical stability is influenced similarly for Intel and AMD architectures
- Vectorization of trancendentials is needed for performance but requires additional caution
- SW developers can influence which code paths to use and generate for different target architectures
- Intel and AMD processors can be supported with the same executable

Backup

Interprocedural Optimizations (IPO)

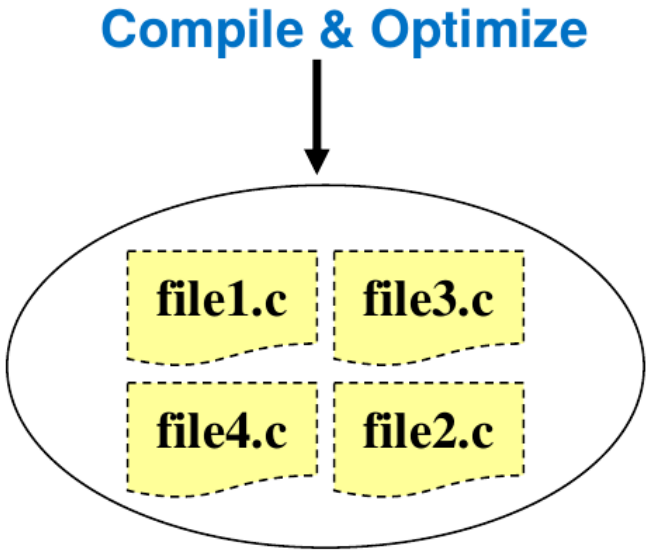
There are two IPO modes:

- Single-file IPO `-ip`:



Optimizations are limited within a single compilation unit.
Subset is default but not full!

- Multi-file IPO `-ipo`:



Optimizations are working accross multiple compilation units.

(Images: Intel)

IPO - Steps

Multi-file IPO (`-ipo`) has two phases:

1. Compiling:
   `icc -c -ipo main.c func1.c func2.c`
   Only creates intermediate fat object files (unoptimized & containing IL data!)
   Such fat object files are also called link-time optimization (LTO) object.

2. Linking:
   `icc -ipo main.o func1.o func2.o`
   Loads intermediate object files, applies optimizations(!) and links

$\Rightarrow$ Beware that all optimiztion actually happens at link time!

More Control over IPO

IPO is very memory and time complex - you can control:

- Split object files with `-ipo[n]` or `-ipo-separate`:
  - By default one single `ipo_out.o` is generated depending on compiler heuristic
    (all IPO objects are merged)
  - `-ipo[n]` creates n `ipo_out[n].o` files plus the first `ipo_out.o`
    ($m : n+1$, $m > n+1$, m number of source files);
  - **Recommended:**
    `-ipo-separate` keeps source file name for objects ($m : m$)
- Specify number of parallel IPO jobs with `-ipo-jobs[n]`
  - Runs multiple IPO builds/optimizations in parallel (useful for multi-core systems)
  - Default is one job only!
  - Requires independent IPO objects (e.g. via `-ipo-separate`)
- Stop after object creation with `-ipo-c`
- Just create assembly file into `ipo_out.s` with `-ipo-s`

IPO - xi Tools

- `xild`: Linker
  - Use to link IPO fat objects (or use the compiler drivers `icc`, `icpc` or `ifort`)
  - Calls compiler first, then `ld`
  - A normal linker would discard the IL data
  - Example: `xild -omyapp a.o b.o`
  - Help: `xild -qhelp`

- `xiar`: Archiver
  - Use to create static libraries
  - A normal archiver would discard the IL data
  - Example: `xiar cru libmine.a a.o b.o`
  - Help: `xiar -qhelp`

Link Time Optimization (LTO)

GCC and LLVM/Clang:

- IPO is Link Time Optimization (LTO) `-flto=[n]`
- Argument `n` parallelizes code generation (default 1)
- Also for the Fortran front-ends (GFortran & Flang)
- **Attention:**
  IPO/LTO enable object codes from different compilers are not compatible!

IPO - Libraries

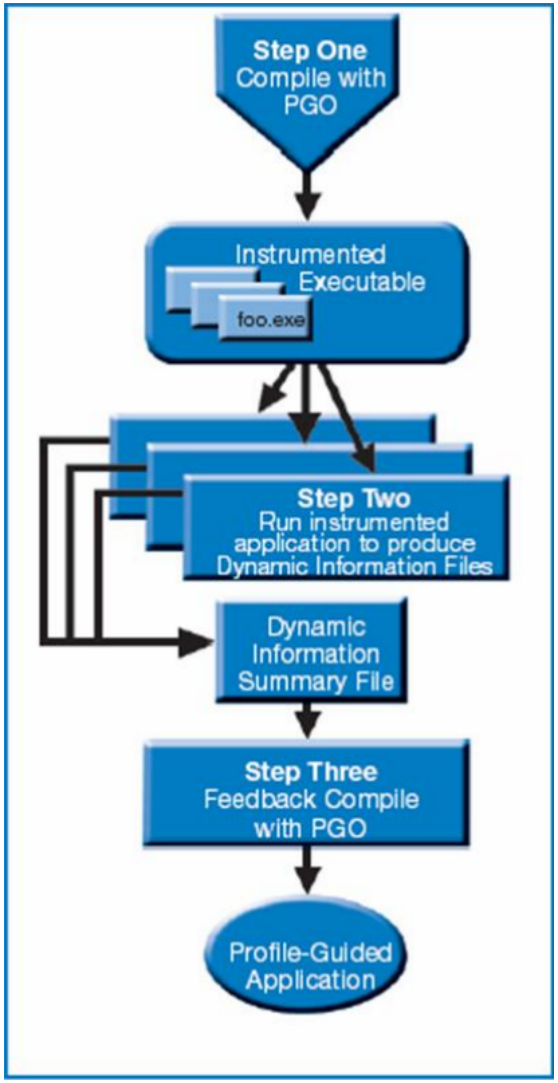
Caution when using IPO with libraries:

- IPO objects contain IL information!
  $\Rightarrow$ **Never ship/distribute libraries with IPO objects!**
- For Linux*, IPO is only reasonable for static libraries
  - Dynamic/shared libraries are only subject to runtime linkage
  - Static libraries (or their subset) can be linked to executable or other libraries
- IPO makes very well sense for static libraries:
  - Linker will only use needed functions/symbols from a static library
  - Same static library can be used in different contexts
  - Static libraries can save space - IPO can detect dead functions
- Use `-qnoipo` for xi tools to disable IPO
  (e.g. for shared libraries)

Profile Guided Optimizations (PGO)

PGO has three phases:

1. Instrumentation:
   `icc -prof-gen prog.c`
2. Profiling:
   Execution of instrumented executable creates `*.dyn/*.gcda` files
3. Optimization:
   `icc -prof-use prog.c`



(Image: Intel)

For GCC and LLVM/Clang:
Use `-fprofile-generate` and `-fprofile-use`, respectively

Why PGO?

- PGO extends compiler optimizations beyond static optimizations
- More precise weights (e.g. inlining, vectorizations, branches taken, ...), e.g.:

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

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

- PGO can improve:
  - 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 - Handling Profile Information

For Intel C++/Fortran Compiler:

- Instrumented executables leave *.dyn files behind (one per invokation & proper termination)
- Compiler is using a single *.dpi file with cumulated *.dyn information when building with `-prof_use`
- If *.dyn files are in working directory during `-prof_use`, all will be merged to a *.dpi file
- More control about which *.dyn files to select, use `profmerge` tool, e.g.:
  ```
  $ profmerge -prof_dpi my_results.dpi
  -exclude_funcs foo,bar -prof_dir ../dyns/
  ```
- Alternatively, compile with `$PROF_DIR` and/or `$PROF_DPI` set
- Also *.dpi files can be merged (hierarchical profiling)

Note: Newer compiler versions provide individual *.dyn file names even accross nodes.

More information can be found here

PGO - Handling Profile Information

For GCC and LLVM/Clang:

- Instrumented executables leave `*.gcda` (GCC) or `*.profraw` (LLVM/Clang) files behind
  (one per compilation unit)
- Different to Intel C++/Fortran Compiler, the same profile file is extended with re-runs
- For optimized build, the `*.gcda` files are used with `-profile-use`
- Different output directories can be used, too with `-fprofile-generate=<path>` and `-fprofile-use=<path>`
- For LLVM/Clang, use `llvm-profdata` tool for merging or showing information of `*.profraw` files

More information can be found...

- GCC: here
- LLVM/Clang: here

Exercise 3 - PGO & IPO

Exercises:

1. IPO:
   - Compile the provided example (`vec[1|2].[cpp|f90]`) with and without IPO (Intel) or LTO (GCC and LLVM/Clang)
   - Measure the speed difference
   - Where is the performance boost coming from?
2. PGO:
   - Instrument the provided code (`pgo.[cpp|f90]`) with PGO (Intel, GCC and LLVM/Clang)
   - Collect runtime profiles and build and optimized version of the executable
   - Where is the performance boost coming from?