



Intel[®] C++ & Fortran Compiler

Presenter: Georg Zitzlsberger

Date: 09-07-2015



Agenda

- **Introduction**
- How to Use
- Compiler Highlights
- Numerical Stability
- What's New (16.0)?
- Summary

Why Use Intel® C++/Fortran Compiler?

Compatibility

- Multiple OS Support: Windows*, Linux*, OS X*
- Integration into development environments: Visual Studio* in Windows*, Eclipse* in Linux*, Xcode* in OS X*
- Source and binary compatibility - can mix and match files as needed
- C99, C11 (partly), C++11& C++14 (partly):
 - <https://software.intel.com/en-us/articles/c99-support-in-intel-c-compiler>
 - <https://software.intel.com/en-us/articles/c11-support-in-intel-c-compiler>
 - <https://software.intel.com/en-us/articles/c0x-features-supported-by-intel-c-compiler>
 - <https://software.intel.com/en-us/articles/c14-features-supported-by-intel-c-compiler>

Why Use Intel® C++/Fortran Compiler?

- Fortran 2003, many features from Fortran 2008:
 - <https://software.intel.com/en-us/articles/intel-fortran-compiler-support-for-fortran-language-standards>

Parallelism

- Automatic vectorization (data level parallelization)
- Language Extension (Intel Cilk® Plus™ for C/C++) for task parallelism
- C++ Multithreading Library (Intel® TBB)
- Multithreaded Performance Libraries (MKL, IPP)
- Own runtime supporting OpenMP* 4.0 (<https://www.openmp.org/>)

Why Use Intel® C++/Fortran Compiler?

Performance

- Code generation tuned for latest micro architecture
- New instructions enable new opportunities (SSE, AVX, AVX2)
- Support for multi-core, many-core

Optimization

- Optimizing compilers
- Highly Optimized Libraries
- MKL – Math functions (BLAS, FFT, LAPACK, etc.)
- IPP – Compression, Vides Encoding, Image Processing, etc.)

Hardware Support

- Skylake support
- Knights Landing (KNL) support
- Compute on Intel Graphics

GDB Debugger now standard



Intel provides enhanced GDB as standard debug solution with Intel® Parallel Studio XE

- Increase C++ and now Fortran application reliability

Faster debug cycles for hybrid programming through full simultaneous debug support across host and Intel® Xeon Phi™ coprocessor targets – on Linux* and Windows* host

Intel® MPX and Intel® AVX-512 support for more robust and faster applications

Fast bug fixing through Intel® Processor Trace support

Fast and efficient analysis and debugging of past program execution

Agenda

- Introduction
- **How to Use**
- Compiler Highlights
- Numerical Stability
- What's New (16.0)?
- Summary

Key Files Supplied with Compilers

Linux*, OS X*

Intel Compiler

- `icc`, `icpc`, `ifort`: C/C++ compiler, Fortran compiler
- `compilervars.(c)sh`: Source scripts to setup the complete compiler/debugger/libraries environment (C/C++ and Fortran)

Linker Driver

- `xild`: Invokes `ld`

Archiver Driver

- `xiar` Invokes `ar`

Intel include files, libraries

Intel Enhanced GDB Debugger

- `gdb-ia` Command Line Debugger for IA
- `gdb-mic` Command Line Debugger for Intel® MIC (Linux ONLY)

Agenda

- Introduction
- How to Use
- **Compiler Highlights**
- Numerical Stability
- What's New (16.0)?
- Summary

Easier to Use Optimization Reports

Intel Compilers

Improved message clarity

- Reference to function names, data variables, control structure

Messages suggest actions for next steps

- For example, Try an option, pragma or clause to change behavior

```
int size();  
void foo(double *restrict a, double  
*b){  
    int i;  
    for (i=0;i<size();i++){  
        a[i] += b[i];  
    }  
}
```



```
icpc -c -O3 -restrict -opt-report x.cpp
```

14.0 compiler:

x.cpp(6) (col. 15) remark: loop was not vectorized: unsupported loop structure

15.0 compiler:

LOOP BEGIN at x.cpp(6,15)

remark #15523: loop was not vectorized: cannot compute loop iteration count before executing the loop.

LOOP END

Compiler Reports – Optimization Report

- Enables the optimization report and controls the level of details
 - `/Qopt-report[:n]` (Windows), `-qopt-report[=n]` (Linux, OS X)
 - When used without parameters, full optimization report is issued on stdout with details level 2
- Writes optimization report to file
 - `/Qopt-report-file:<filename>` (Windows), `-qopt-report-file=<filename>` (Linux, OS X)
 - By default, without this option, *.optrpt files is generated for each source file.
- Subset of the optimization report for specific phases only
 - `/Qopt-report-phase[:list]` (Windows), `-qopt-report-phase=[list]` (Linux, OS X); phases can be:
 - cg: Code generation
 - **ipo: Interprocedural optimization report**
 - **loop: Loop nest optimization report**
 - offload: Optimizations for Intel® MIC Architecture offloaded code
 - **openmp: OpenMP parallelized code report**
 - pgo: Profile Guided Optimization report
 - tcollect: Trace collection report
 - **vec: Vectorization report**
 - all: Full optimization report, same as `/Qopt-report`, `-qopt-report` used without parameter

High-Level Optimization (HLO)

Compiler switches:

/O2, /O3, /Ox (Windows*), -O2, -O3 (Linux*, Mac OS*)

Loop level optimizations

- loop unrolling, cache blocking, prefetching

More aggressive dependency analysis

- Determines whether or not it's safe to reorder or parallelize statements

Scalar replacement

- Goal is to reduce memory by replacing with register references

Intel-Specific Pragmas (some)

Pragma	Description
<code>alloc_section</code>	allocates variable in specified section
<code>cilk grainsize</code>	specifies the grain size for one <code>cilk_for</code> loop
<code>distribute_point</code>	instructs the compiler to prefer loop distribution at the location indicated
<code>inline</code>	instructs the compiler that the user prefers that the calls in question be inlined
<code>ivdep</code>	instructs the compiler to ignore assumed vector dependencies
<code>loop_count</code>	indicates the loop count is likely to be an integer
<code>nofusion</code>	prevents a loop from fusing with adjacent loops
<code>novector</code>	specifies that the loop should never be vectorized
<code>optimize</code>	enables or disables optimizations for specific functions; provides some degree of compatibility with Microsoft's implementation of optimize pragma
<code>optimization_level</code>	enables control of optimization for a specific function
<code>optimization_parameter</code>	tells the compiler to generate code specialized for a particular processor, at the function level, similar to the <code>-m (/arch)</code> option
<code>parallel/noparallel</code>	facilitates auto-parallelization of an immediately following DO loop; using keyword [always] forces the compiler to auto-parallelize; <code>noparallel</code> pragma prevents auto-parallelization
<code>simd</code>	enforces vectorization of innermost loops
<code>unroll/nounroll</code>	instructs the compiler the number of times to unroll/not to unroll a loop
<code>unroll_and_jam/ nounroll_and_jam</code>	instructs the compiler to partially unroll higher loops and jam the resulting loops back together. Specifying the <code>nounroll_and_jam</code> pragma prevents unrolling and jamming of loops.
<code>unused</code>	describes variables that are unused (warnings not generated)
<code>vector</code>	indicates to the compiler that the loop should be vectorized according to the arguments: <code>always/aligned/unaligned/nontemporal/temporal</code>

...and more (e.g. OpenMP* or offloading for Intel® MIC)

Intel-Specific Directives (some)

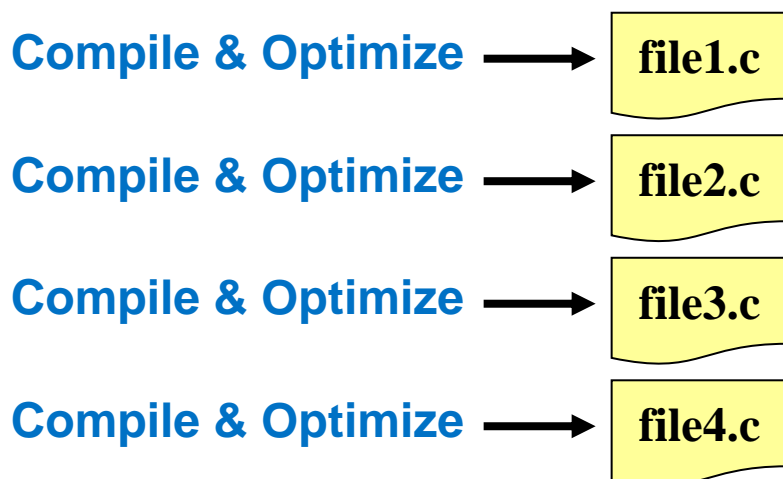
Directive	Description
ASSUME_ALIGNED	Specifies that an entity in memory is aligned.
DISTRIBUTE POINT	Suggests a location at which a DO loop may be split.
INLINE/FORCEINLINE/NOINLINE	Tell the compiler to perform the specified inlining on routines within statements or DO loops.
IVDEP	Assists the compiler's dependence analysis of iterative DO loops.
LOOP COUNT	Specifies the typical trip count for a DO loop; this assists the optimizer.
MESSAGE	
NOFUSION	Prevents a loop from fusing with adjacent loops.
OPTIMIZE and NOOPTIMIZE	Enables or disables optimizations for the program unit.
OPTIONS	Affects data alignment and warnings about data alignment.
PACK	Specifies the memory alignment of derived-type items.
PARALLEL and NOPARALLEL	Facilitates or prevents auto-parallelization by assisting the compiler's dependence analysis of the immediately following DO loop.
SIMD	Requires and controls SIMD vectorization of loops.
UNROLL/NOUNROLL	Tells the compiler's optimizer how many times to unroll a DO loop or disables the unrolling of a DO loop.
UNROLL_AND_JAM/ NOUNROLL_AND_JAM	Enables or disables loop unrolling and jamming.
VECTOR/NOVECTOR	Overrides default heuristics for vectorization of DO loops.

...and more (e.g. OpenMP* or offloading for Intel® MIC)

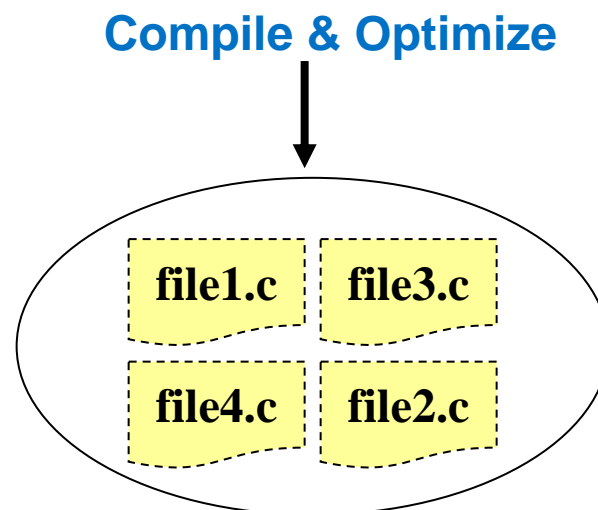
Interprocedural Optimizations

Extends optimizations across file boundaries

Without IPO



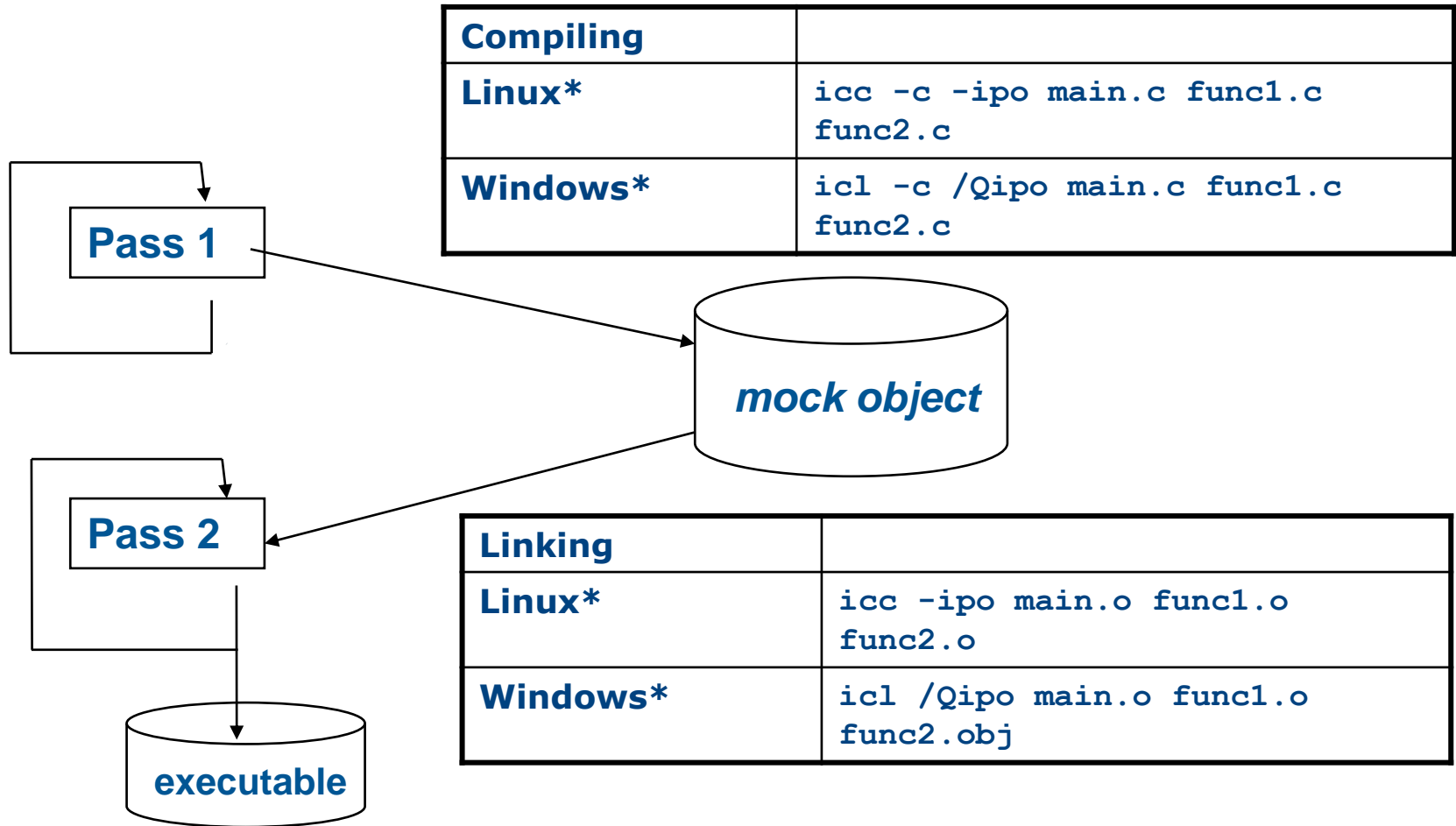
With IPO



<code>/Qip, -ip</code>	Only between modules of one source file
<code>/Qipo, -ipo</code>	Modules of multiple files/whole application

Interprocedural Optimizations (IPO)

Usage: Two-Step Process



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

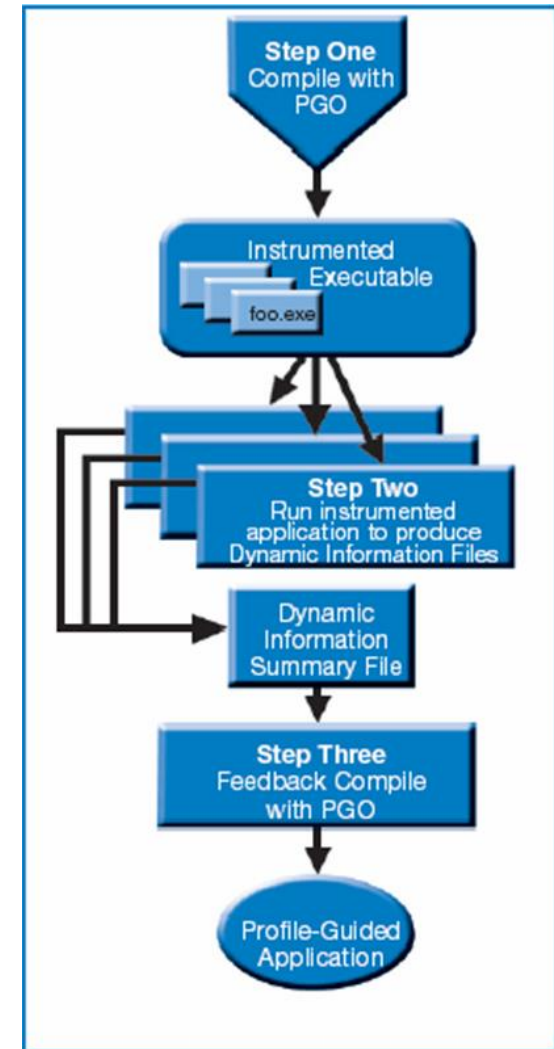
```
if (x > y)
    do_this();
else
    do_that();
```

```
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

Step 1

Compile + link to add instrumentation

```
icc -prof_gen prog.c
```

Instrumented
executable:
prog.exe

Step 2

Execute instrumented program

prog.exe (on a typical dataset)

Dynamic profile:
12345678.dyn

Step 3

Compile + link using feedback

```
icc -prof_use prog.c
```

Merged .dyn files:
pgopti.dpi

Optimized executable:
prog.exe

Code Coverage Tool

Combines static and dynamic profiling information to generate a view in HTML of how much code is exercised for particular workload(s)

Available for C++ and Fortran, on all platforms

Capable of basic block level, function level, partial or component code coverage analysis, excluding function/file, differential coverage between different test runs (.dyn).

See: <https://software.intel.com/en-us/node/522743>

Use the Code Coverage Tool for Component Testing

Loop Profiler

Identify Time Consuming Loops/Functions

Compiler switch:

/Qprofile-functions, -profile-functions

- Insert instrumentation calls on function entry and exit points to collect the cycles spent within the function.

Compiler switch:

**/Qprofile-loops=<inner|outer|all>,
-profile-loops= <inner|outer|all>**

- Insert instrumentation calls for function entry and exit points as well as the instrumentation before and after instrumentable loops of the type listed as the option's argument.

Loop Profiler switches trigger generation of text (.dump) and XML (.xml) output files

- Invocation of XML viewer on command line:

```
java -jar loopprofviewer.jar <xml datafile>
```

Loop Profiler Text Dump (.dump file)

time(abs)	time(%)	self(abs)	self(%)	call_count	exit_count	loop_ticks(%)	file:line
4378070322	99.83	1810826157	41.29	1	1	41.29	deflate.c:623
647499316	14.76	642829878	14.66	16768796	16768796	0.01	trees.c:961
1462208444	33.34	487754966	11.12	10546669	10546669	7.07	deflate.c:360
119744296	2.73	119686890	2.73	513	513	2.73	util.c:63
137053240	3.13	89563374	2.04	512	512	2.04	bits.c:185
198253943	4.52	66623288	1.52	512	512	1.46	deflate.c:478
47165828	1.08	47102278	1.07	1025	1025	0.00	util.c:153
69547871	1.59	32157819	0.73	344059	344059	0.66	trees.c:454
119928920	2.73	24484703	0.56	1536	1536	0.12	trees.c:611
132122614	3.01	12346758	0.28	513	513	0.00	zip.c:106
22904224	0.52	6738036	0.15	1537	1537	0.15	trees.c:571
6675927	0.15	6672487	0.15	1	1	0.00	gzip.c:1511
15997356	0.36	6029850	0.14	139288	139288	0.12	bits.c:151
2792164	0.06	2620132	0.06	1536	1536	0.06	trees.c:485
1290621	0.03	1175933	0.03	1024	1024	0.03	trees.c:699
495976	0.01	387220	0.01	513	513	0.01	trees.c:408
259246700	5.91	261552	0.01	512	512	0.00	trees.c:857
47392247	1.08	162869	0.00	1025	1025	0.00	util.c:121
5225406	0.12	113633	0.00	512	512	0.00	trees.c:791
4385684262	100.00	66840	0.00	1	1	0.00	gzip.c:424
630948	0.01	56083	0.00	1	1	0.00	deflate.c:289
4385610543	100.00	35086	0.00	1	1	0.00	gzip.c:704
6711753	0.15	32771	0.00	1	1	0.00	gzip.c:853
82503	0.00	23661	0.00	1	1	0.00	trees.c:335
53760	0.00	22140	0.00	513	513	0.00	bits.c:164
4378817661	99.84	19622	0.00	1	1	0.00	zip.c:35
26175	0.00	13585	0.00	1	1	0.00	gzip.c:1605
12528	0.00	12466	0.00	1	1	0.00	gzip.c:1583
30798	0.00	11268	0.00	512	512	0.00	bits.c:122
9294	0.00	9232	0.00	1	1	0.00	gzip.c:915
7575	0.00	7463	0.00	1	1	0.00	util.c:170
6237	0.00	6113	0.00	2	2	0.00	gzip.c:1421
4761	0.00	4699	0.00	1	1	0.00	util.c:283
2358	0.00	2234	0.00	2	2	0.00	util.c:183
9588	0.00	1153	0.00	1	1	0.00	gzip.c:1062
10218	0.00	862	0.00	1	1	0.00	gzip.c:989
8373	0.00	686	0.00	1	1	0.00	gzip.c:939
216	0.00	154	0.00	1	1	0.00	gzip.c:1703
87	0.00	25	0.00	1	1	0.00	bits.c:99
84	0.00	22	0.00	1	1	0.00	gzip.c:1398

Loop Profiler Data Viewer GUI (copy from sl. 46)

The screenshot displays the Loop Profiler Data Viewer GUI with two main sections: Function Profile View and Loop Profile View.

Function Profile View: This section shows a table of functions and their performance metrics. The table has the following columns: Function, Function file:line, Time, % Time, Self time, % Self time, Call Count, and % Time in Loops. A blue arrow points to the column headers, indicating that they allow selection to control sort criteria independently for function and loop table. Another blue arrow points to the menu, indicating that it allows the user to enable filtering or displaying the source code.

Loop Profile View: This section shows a table of loops and their performance metrics. The table has the following columns: Function, Function file:line, Loop file:line, Time, % Time, Self time, % Self time, Loop entries, Min iterations, Avg iterations, and Max iterations. A blue arrow points to the table, indicating that it displays the loop profile data.

Menu: The menu is located in the top right corner and contains the following options: Filter: Function total time > 2.0%, Filter: Function self time > 2.0%, Filter: Loops for selected function, and View: Function source for selected function.

Pointer Checker (C/C++)

- Out-of-bounds memory checking at runtime
 - Checks before any memory access through a pointer that the pointer address is inside the object pointed to.
 - Checks for accesses through pointers that have been freed.
- Enable pointer checker via compiler switches.
 - `/Qcheck-pointers:[none|write|rw]`
 - `-check-pointers=[none|write|rw]`
- Enable checking for dangling pointer references:
 - `/Qcheck-pointers-dangling:[none|heap|stack|all]`
 - `-check-pointers-dangling=[none|heap|stack|all]`
- Enable checking of bounds for arrays without dimensions:
 - `/Qcheck-pointers-undimensioned[-]`
 - `-[no]check-pointers-undimensioned`
- Intrinsic allow user to get lower/upper bounds associated with pointer and create / destroy bounds for a pointer.
 - `- void * __chkp_lower_bound(void **)`
 - `- void * __chkp_upper_bound(void **)`
 - `- void * __chkp_kill_bounds(void *p)`
 - `- void * __chkp_make_bounds(void *p, size_t size)`

Pointer Checker Example

```
void main()
```

```
{
```

```
    char a[5];
```

```
    foo(a);
```

```
}
```

```
void foo(char *a)
```

```
{
```

```
    int i;
```

```
    for (i=0; i<100; i++) {
```

```
        a[i] = 0;
```

```
    }
```

```
}
```

```
$ icc main.c -check-pointers=write
```

```
$ ./a.out
```

CHKP: Bounds check error

Traceback:

foo [0x4010E0]

main [0x40104D]

__tmainCRTStartup [0x4014A6]

BaseThreadInitThunk [0x760E3677]

RtlInitializeExceptionChain
[0x77869F02]

RtlInitializeExceptionChain
[0x77869ED5]

Agenda

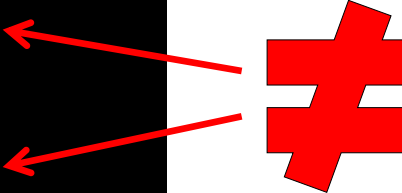
- Introduction
- How to Use
- Compiler Highlights
- **Numerical Stability**
- What's New (16.0)?
- Summary

Numerical Stability - The Problem

Numerical (FP) results change on run-to-run:

```
C:\Users\me>test.exe
4.012345678901111

C:\Users\me>test.exe
4.012345678902222
```

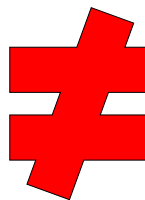


Numerical results change between different systems:

Intel® Xeon® Processor E5540

```
C:\Users\me>test.exe
4.012345678901111

C:\Users\me>test.exe
4.012345678901111
```



Intel® Xeon® Processor E3-1275

```
C:\Users\me>test.exe
4.012345678902222

C:\Users\me>test.exe
4.012345678902222
```

Why Results Vary I

Basic problem:

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

Caused by 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)

⇒ **User controls those (direct or indirect)**

Why Results Vary II

Order of FP operations has impact on rounded result, e.g.

$$(a+b)+c \neq a+(b+c)$$

$$2^{-63} + 1 + -1 = 2^{-63} \text{ (mathematical result)}$$

$$(2^{-63} + 1) + -1 \approx 0 \text{ (correct IEEE result)}$$

$$2^{-63} + (1 + -1) \approx 2^{-63} \text{ (correct IEEE result)}$$

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(...)`, ...)

Flush-to-zero (for SIMD instructions)

Contractions (e.g. FMA)

Different code paths (e.g. SIMD & non-SIMD or Intel AVX vs. SSE)

...

\Rightarrow Subject of Optimizations by Compiler & Libraries

Compiler Optimizations

Why compiler optimizations:

- Provide best performance
- Make use of processor features like SIMD (vectorization)
- In most cases performance is more important than FP precision and reproducibility
- Use faster FP operations (not legacy x87 coprocessor)



FP model of compiler limits optimizations and provides control about FP precision and reproducibility:

Default is “**fast**”

Controlled via:

Linux*, OS X*: **-fp-model**

Windows*: **/fp:**

FP Model I

FP model does more:

- Value safety ($(x + y) + z \neq x + (y + z)$)
- Floating-point expression evaluation
- Precise floating-point exceptions
- Floating-point contractions
- Floating-point unit (FPU) environment access

FP Model II

FP model settings:

- **precise**: allows value-safe optimizations only
- **source/double/extended**: intermediate precision for FP expression eval.
- **except**: enables strict floating point exception semantics
- **strict**: enables access to the FPU environment disables floating point contractions such as fused multiply-add (fma) instructions implies “**precise**” and “**except**”
- **fast[=1]** (default):
Allows value-unsafe optimizations compiler chooses precision for expression evaluation
Floating-point exception semantics not enforced
Access to the FPU environment not allowed
Floating-point contractions are allowed
- **fast=2**: some additional approximations allowed

FP Model - Comparison

Key	Value Safety	Expression Evaluation	FPU Environ. Access	Precise FP Exceptions	FP contract
precise source double extended	Safe	Varies Source Double Extended	No	No	Yes
strict	Safe	Varies	Yes	Yes	No
fast=1 (default)	Unsafe	Unknown	No	No	Yes
fast=2	Very Unsafe	Unknown	No	No	Yes
except	*/**	*	*	Yes	*
except-	*	*	*	No	*
*	These modes are unaffected. -fp-model except[-] only affects the precise FP exceptions mode.				
**	It is illegal to specify -fp-model except in an unsafe value safety mode.				

FP Model - Example

Using `-fp-model [precise|strict]:`

- Disables reassociation
- Enforces standard conformance (left-to-right)
- May carry a significant performance penalty

Disabling of reassociation also impacts vec

```
#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";
}
```

Other FP Options I

- Linux*, OS X*: **-[no-]ftz**, Windows*: **/Qftz [-]**
Flush denormal results to zero
- Linux*, OS X*: **-[no-]prec-div**, Windows*: **/Qprec-div [-]**
Improves precision of floating point divides
- Linux*, OS X*: **-[no-]prec-sqrt**, Windows*:
/Qprec-sqrt [-]
Improves precision of square root calculations
- Linux*, OS X*: **-fimf-precision=name**, Windows*:
/Qimf-precision:name
high, medium, low: Controls accuracy of math library functions
- Linux*, OS X*: **-fimf-arch-consistency=true**, Windows*:
/Qimf-arch-consistency:true
Math library functions produce consistent results on different processor types of the same architecture

Other FP Options II

- Linux*, OS X*: **-fpe0**, Windows*: **/fpe:0**
Unmask floating point exceptions (Fortran only) and disable generation of denormalized numbers
- Linux*, OS X*: **-fp-trap=common**, Windows*: **/Qfp-trap:common**
Unmask common floating point exceptions (C/C++ only)
- Linux*, OS X*: **-[no-]fast-transcendentals**, Windows*: **/Qfast-transcendentals[-]**
Enable/disable fast math functions
- Fortran only:
Linux*, OS X*: **-assume [no]protect_parens**, Windows*: **/assume:[no]protect_parens**
Default is **noprotect_parens**
- ...

Recommendation

- The **default FP model** is fast but has less precision/reproducibility (vectorization)
- The **strict FP model** has best precision/reproducibility but is slow (no vectorization; x87 legacy)
- For best trade-off between precision, reproducibility & performance use:
Linux*, OS X*: **-fp-model precise -fp-model source**
Windows*: **/fp:precise /fp:source**
Approx. 12-15% slower performance for SPECCPU2006fp
- Don't mix math libraries from different compiler versions!
- Using different processor types (of same architecture), specify:
Linux*, OS X*: **-fimf-arch-consistency=true**
Windows*: **/Qimf-arch-consistency:true**

More information:

<http://software.intel.com/en-us/articles/consistency-of-floating-point-results-using-the-intel-compiler>

Agenda

- Introduction
- How to Use
- Compiler Highlights
- Numerical Stability
- **What's New (16.0)?**
- Summary

Intel® C/C++ & Fortran Compilers 16.0

What's New?

- **BLOCK_LOOP** directive, **block_loop** pragma
- Annotated Source Listing:

```
1      int* foo(int* a, int* b, int upperbound) {
2
3          int* c = new int[upperbound];
4          #pragma omp parallel for
OpenMP DEFINED LOOP WAS PARALLELIZED
5          for (int i = 0; i < upperbound; ++i) {
LOOP BEGIN at Test/library.cpp(5,2)
<Peeled>
LOOP END
LOOP BEGIN at Test/library.cpp(5,2)
    remark #25460: No loop optimizations reported
LOOP END
...
```

[/Q|-q]opt-report-annotate=[text|html],
[/Q|-q]opt-report-annotate-position=[caller|callee|both]

New for Fortran

- Submodules from Fortran 2008
- **IMPURE ELEMENTAL** from Fortran 2008
- Further C Interoperability from Fortran 2015 (TS29113)
- Fortran 2015 **TYPE (*)**, **DIMENSION (..)**, **RANK** intrinsic, attributes for args with BIND
- **-init** enhancements
- **-fpp-name** option

Status of Fortran support:

<https://software.intel.com/en-us/articles/intel-fortran-compiler-support-for-fortran-language-standards>

New for C/C++

- Compile time improvements:
 - Intrinsic prototypes are excluded
 - Restore behavior with `-D__INTEL_COMPILER_USE_INTRINSIC_PROTOTYPES`
- Operators are now overloaded to allow SIMD intrinsic types
- Honoring parenthesis with `-f[no-]protect-parens`
- Improved C++14 (generic lambdas, member initializers, aggregates, ...)
- Improved C11 (`_Static_assert`, `_Generic`, `_Noreturn`, ...)
- Feature macros

Agenda

- Introduction
- How to Use
- Compiler Highlights
- Numerical Stability
- What's New (16.0)?
- **Summary**

Summary

Intel® C++ and Intel® Fortran Compilers of Intel® Parallel Studio provide powerful features, especially

- High level optimizations
- Auto-vectorization/-parallelization to parallelize serial code
- Sophisticated programming methods for multithreading
- Runs on GNU* environments or integrates into Eclipse (Linux*)

More information on Intel's software offerings and services at <http://software.intel.com>

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2015, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

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

Notice revision #20110804

