Intel® C++ Compiler for applications running on Embedded OS Linux*

Key Files Supplied with Compiler

Linux*

Intel compiler

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

Linker driver

- `xild`: Invokes `ld`

Intel include files, libraries

Compatibility to Standards

The Intel C++ Compiler provides the following language conformances:

ANSI/ISO standard for C language compilation (ISO/IEC9899:1990)

ANSI/ISO standard (ISO/IEC 14882:1998) for the C++ language

Common Optimization Switches

	Linux*
Disable optimization	-O0
Optimize for speed (no code size increase)	-O1
Optimize for speed (default)	-O2
High-level loop optimization	-O3
Create symbols for debugging	-g
Multi-file inter-procedural optimization	-ipo
Profile guided optimization (multi-step build)	-prof-gen -prof-use
Optimize for speed across the entire program **warning: -fast def’n changes over time	-fast (same as: -ipo –O3 -no-prec-div -static -xHost)

Compiler Reports – Optimization Report

Compiler switch:
`-opt-report-phase[=phase]` (Linux*)

`phase` can be:
`ipo_inl` - Interprocedural Optimization Inlining Report
`ilo` – Intermediate Language Scalar Optimization
`hpo` – High Performance Optimization
`hlo` – High-level Optimization
`all` – All optimizations (not recommended, output too verbose)

Control the level of detail in the report:
`-opt-report[0|1|2|3]` (Linux*)

- If you do not specify the level (i.e. /Qopt-report, -opt-report) level 2 is being used.

Save report output to file:
`-opt-report-file=[file]` (Linux*)

Vectorization subset report:
`/Qvec-report2, -vec-report2`

Optimization Report Example

```
icc -O3 -opt-report-phase=hlo -opt-report-phase=hpo
```

```
…
LOOP INTERCHANGE in loops at line: 7 8 9
Loopnest permutation ( 1 2 3 ) --> ( 2 3 1 )
…
Loop at line 8 blocked by 128
Loop at line 9 blocked by 128
Loop at line 10 blocked by 128
…
Loop at line 10 unrolled and jammed by 4
Loop at line 8 unrolled and jammed by 4
…
…(10)…  loop was not vectorized: not inner loop.
…(8)…   loop was not vectorized: not inner loop.
…(9)…   PERMUTED LOOP WAS VECTORIZED
…
```

High-Level Optimizer (HLO)

Compiler switches:
`-O2, -O3` (Linux*)

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

Interprocedural Optimizations (IPO)
Multi-pass Optimization

- Interprocedural optimizations performs a static, topological analysis of your application!
- ip: Enables inter-procedural optimizations for current source file compilation
- ipo: Enables inter-procedural optimizations across files
  - Can inline functions in separate files
  - Especially many small utility functions benefit from IPO

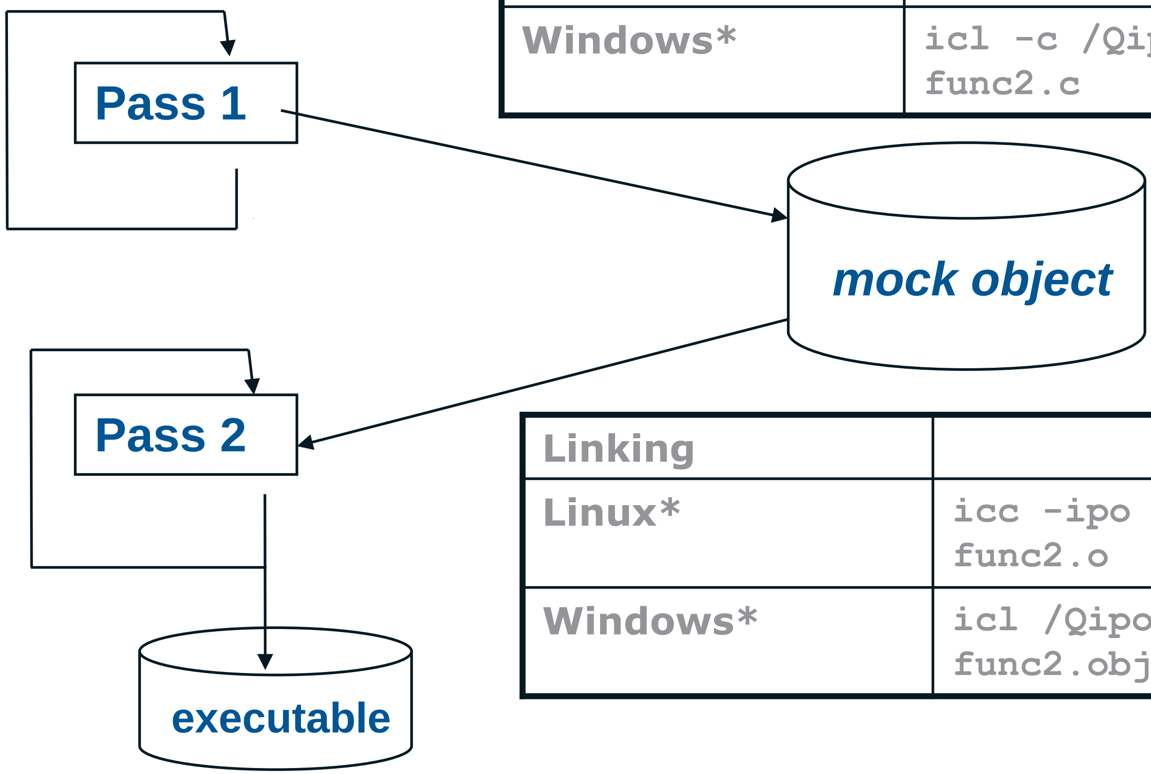
Linux*
-ip
-ipo

Enabled optimizations:
- Procedure inlining (reduced function call overhead)
- Interprocedural dead code elimination, constant propagation and procedure reordering
- Enhances optimization when used in combination with other compiler features

Interprocedural Optimizations (IPO)

Usage: Two-Step Process

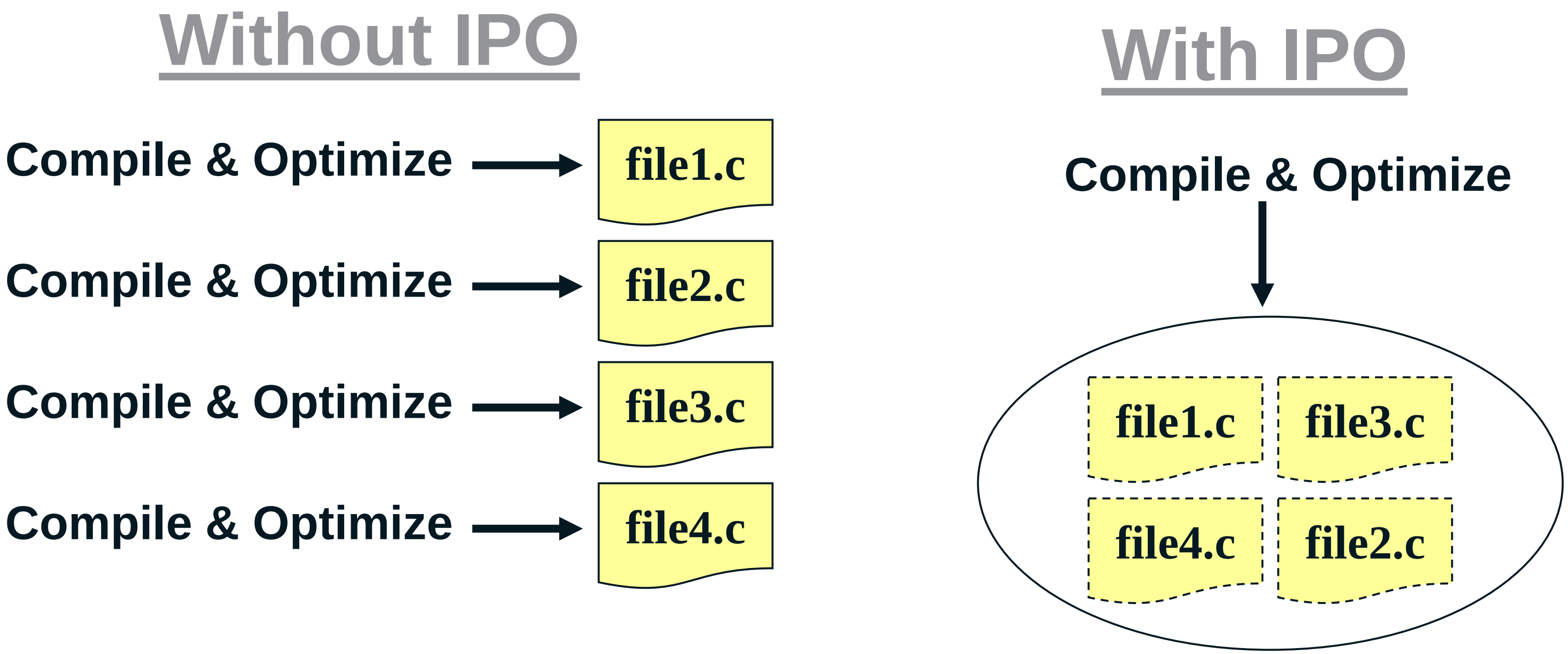
Compiling	
Linux*	`icc -c -ipo main.c func1.c func2.c`
Windows*	`icl -c /Qipo main.c func1.c func2.c`



Linking	
Linux*	`icc -ipo main.o func1.o func2.o`
Windows*	`icl /Qipo main.o func1.o func2.obj`

Interprocedural Optimizations

Extends optimizations across file boundaries



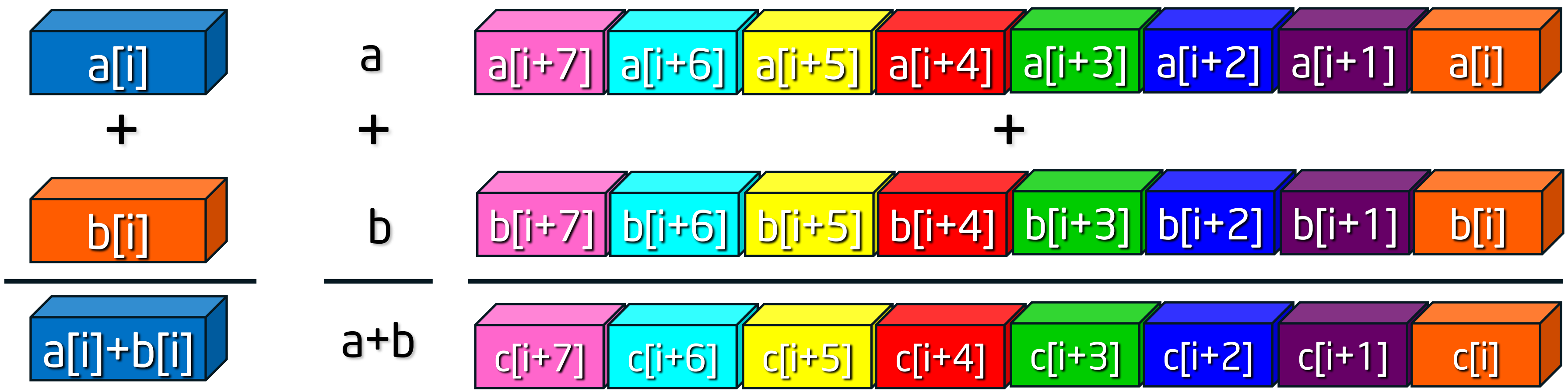
`/Qip, -ip`	Only between modules of one source file
`/Qipo, -ipo`	Modules of multiple files/whole application

Auto-Vectorization

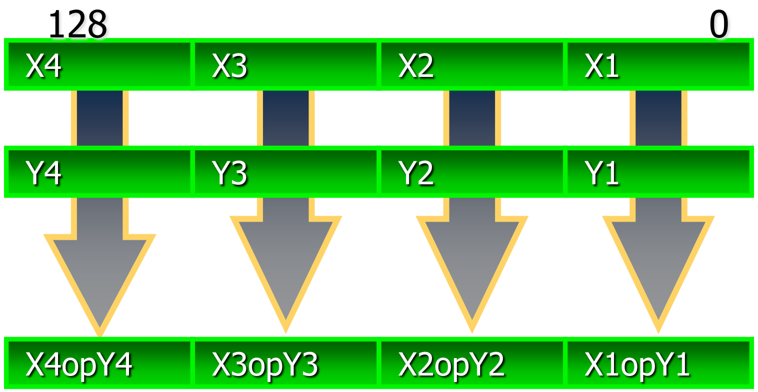
SIMD – Single Instruction Multiple Data

- Scalar mode
  - one instruction produces one result
- SIMD processing
  - with SSE or AVX instructions
  - one instruction can produce multiple results

```
for (i=0;i<=MAX;i++)
    c[i]=a[i]+b[i];
```

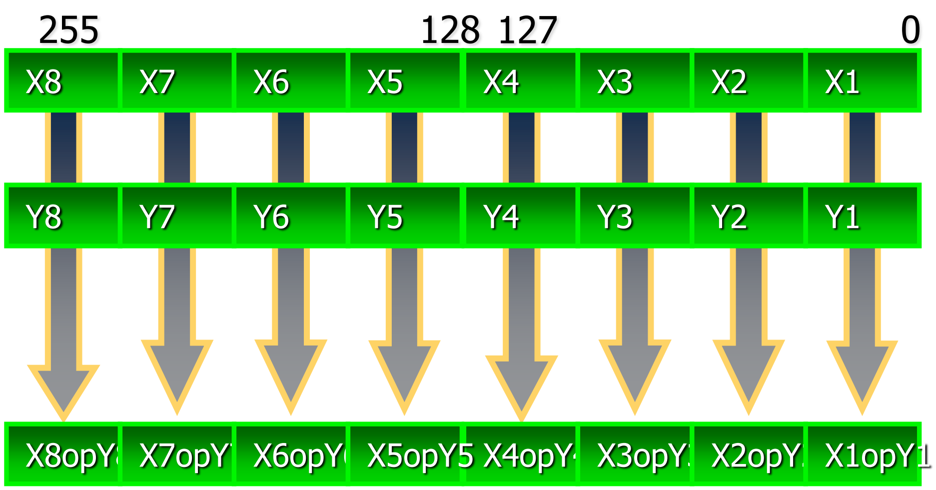


Vectorization is Achieved through SIMD Instructions & Hardware



Intel® SSE

Vector size: 128bit
Data types:
8,16,32,64 bit integers
32 and 64bit floats
VL: 2,4,8,16
Sample: Xi, Yi bit 32 int / float



Intel® AVX

Vector size: 256bit
Data types: 32 and 64 bit floats
VL: 4, 8, 16
Sample: Xi, Yi 32 bit int or float
First introduced in 2011

Comparison of Ways Applications can Take Advantage of Vectorization

	Effort Required	Code Maintain -ability	Performance Potential	Scale Forward
Assembly/Intrinsics	Most	Least	Best	No
Existing libraries such as Intel® IPP, Intel® MKL	Least	Most	Best	Yes
Intel Compiler Auto-Vectorization	Least	Most	Good	Yes
High-level Constructs	Moderate	Most	Best	Yes

Compiling for Intel® AVX and SSSE3 using Intel® C++ Compiler

Compile with –xavx (/Qxavx on Windows*)

- Main speedups are for floating point
  - Integer 256 bit arithmetic instructions coming for AVX2
  - Best if 32 byte aligned

-axavx (/Qaxavx) gives both SSE and AVX code paths

- use –x (/Qx) switches to modify the default SSE code path
  - e.g. –axavx –xsse3_atom target Intel Core i7 and Intel Atom™ Processor simultaneously (/Qaxavx /Qxssse3_atom on Windows)

software.intel.com/en-us/articles/how-to-compile-for-intel-avx/

software.intel.com/en-us/articles/atom-optimized-compiler/

Optimization Notice

Compiler Based Vectorization

Extension Specification

Feature	SIMD Extension
Intel® Streaming SIMD Extensions 2 (Intel® SSE2) as available in initial Pentium® 4 or compatible non-Intel processors	sse2
Intel® Streaming SIMD Extensions 3 (Intel® SSE3) as available in Pentium® 4 or compatible non-Intel processors	sse3
Supplemental Streaming SIMD Extensions 3 (SSSE3) as available in Intel® Core™2 Duo processors	ssse3
Intel® SSE4.1 as first introduced in Intel® 45nm Hi-K next generation Intel Core™ micro-architecture	sse4.1
Intel® SSE4.2 Accelerated String and Text Processing instructions supported first by Intel® Core™ i7 processors	sse4.2
Like ssse3 but also generates the MOVBE instruction that is available for the Intel® Atom™ processor and Intel® Centrino® Atom™ Processor Technology	ssse3_atom
Intel® Advanced Vector Extensions (Intel® AVX) as available in 2nd generation Intel® Core™ processor family	avx
Intel® Advanced Vector Extension (Intel® AVX) including instructions offered by the 3rd generation Intel® Core processor	core-avx-i
Intel® Advanced Vector Extension 2 (Intel® AVX2) as provided by a future Intel processor	core-avx2

15

Compiler Reports – Vectorization Report

Compiler switch:

`-vec-report<n>` (Linux)

Set diagnostic level dumped to stdout

n=*0*: No diagnostic information
n=1: (Default) Loops successfully vectorized
n=*2*: Loops not vectorized – and the reason why not
n=3: Adds dependency Information
n=4: Reports only non-vectorized loops
n=5: Reports only non-vectorized loops and adds dependency info

Automatic Vectorization by Compiler

Intel Compiler will auto vectorize the source code for you if it can

Pros:

- Minimal effort required
- Maintainable – source code is not changed
- Portable across Intel SIMD architectures
- Optimal performance is possible in best cases
- Scales forward!

Cons:

- Compiler is conservative; will not generate unsafe code

=> Advanced optimization techniques help to improve Data Level Parallelization using Vectorization

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.

  `-check-pointers=[none|write|rw]`
- Enable checking for dangling pointer references:

  `-check-pointers-dangling=[none|heap|stack|all]`
- Enable checking of bounds for arrays without dimensions:

  `-[no]check-pointers-undimensioned`
- Intrinsics 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)`

Inlining Functions

When the compiler inlines a function call, the function's code gets inserted into the caller's instruction stream

Benefits:

Reducing overhead of calling a function

- writing the registers and parameters to/from stack
- restore the registers when the function returns.

Improving performance because the optimizer can procedurally integrate the called function and can do better optimizations

- sub-expression elimination
- copy propagation

Drawbacks:

Overuse of inlining can actually make programs slower. Depending on a function's size, inlining it can cause the code size to increase, resulting in more cache misses and more pressure on the instruction cache

The speed benefits of inline functions tend to diminish as the function grows in size. At some point the overhead of the function call becomes small compared to the execution of the function body, and the benefit is lost.

Compiler Floating Point Model

The Floating Point options allow to control the optimizations of floating-point instructions. These options can be used to tune the performance, level of accuracy or result consistency.

Accuracy

Produce results that are “close” to the correct value

- Measured in relative error, possibly ulps (units in the last place)

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

Performance

Produce the most efficient code possible

- Default, primary goal of Intel® Compilers

These objectives usually conflict! Wise use of compiler options lets you control the tradeoffs.

Optimization Notice

Compiler Floating-Point Model

The Floating-Point Compiler Switch

`-fp-model` *`keyword`* (Linux*)

Lets you choose the FP semantics at a coarse granularity and specify the compiler rules for

- Value safety
- FP expression evaluation
- FPU environment access
- Precise FP exceptions
- FP contractions
- Abrupt underflow (flush to zero)
  - Denormals are set to zero
  - May improve performance, esp. if HW doesn‵t support denormals

Floating-Point Keywords

Controls consistency of floating point results by restricting certain optimizations. Values for *keywords* are

- `fast[=1|2]`; default is `fast=1`
  - Allows „value-unsafe" optimizations (=default)
  - Allows aggressive optimizations at a slight cost in accuracy or consistency.
  - Some additional approximations allowed with `fast=2`
- `precise`
  - Enables only value-safe optimizations on floating point code.
- `source`
  - Implies precise and enables intermediates to be computed in source precision.
  - Source is the recommended form for the majority of situations on processors supporting Intel® 64 and IA-32 platforms when SSE are enabled with /QxSSE2 or higher.

Floating-Point Keywords (2)

- `double`
  - Implies precise and enables intermediates to be computed in double or extended precision.
  - Not avaliable in Intel® Fortran Compilers
- `extended`
  - Rounds intermediate results to 64-bit (extended) precision
  - Enables value safe optimization
- `except`
  - Enables floating point exception semantics
- `strict`
  - Strictest mode of operation, enables both the precise and except options and disables contractions (i.e., precise + strict + disable fma)

The *–fp-model<key>* Switch

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 except-	*/** *	* *	* *	Yes 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.

Optimization Notice

New Parallelism Method: Intel® Cilk™ Plus

An extension to C and C++ for expressing fine-grained task parallelism

- Shared-memory multiprocessing (like OpenMP)

Very simple syntax of 3 keywords only: ***_Cilk_spawn*** and ***_Cilk_sync, _Cilk_for***

- #include <cilk/cilk.h> in order to get **cilk_spawn**, **cilk_sync**, and **cilk_for**

Every Cilk program preserves the ***serial semantic***

Cilk provides ***performance guarantees*** since it is based on theoretically efficient ***work-stealing*** scheduler

Preventing races using ***reducer hyperobjects***

Array Notations to provide data parallelism for sections of arrays or whole arrays

Elemental Functions to enable data parallelism of whole functions or operations

#pragma SIMD to express vector parallelism using SIMD hardware registers

Optimization Notice

Summary

Intel® C++ Compiler for applications running on Embedded OS Linux*

- 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

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