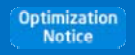
Intel® Compilers Version 15.0

Part of Intel® Parallel Studio XE Composer Edition 2015



Intel® Parallel Studio XE 2015

Old Name	Intel® Composer XE, Intel® Visual Fortran Composer XE, Intel® Fortran Composer XE, Intel® C++ Composer XE	Intel® Parallel Studio XE	Intel® Cluster Studio XE
New Name	**Intel® Parallel Studio XE Composer Edition1**	**Intel® Parallel Studio XE Professional Edition1**	**Intel® Parallel Studio XE Cluster Edition9**
Intel® C++ Compiler	√	√	√
Intel® Fortran Compiler	√	√	√
Intel® Threading Building Blocks (C++ only)	√	√	√
Intel® Integrated Performance Primitives (C++ only)	√	√	√
Intel® Math Kernel Library	√	√	√
Intel® Cilk™ Plus (C++ only)	√	√	√
Intel® OpenMP*	√	√	√
Rogue Wave IMSL* Library2 (Fortran only)	Add-on	Add-on	Bundled and Add-on
Intel® Advisor XE		√	√
Intel® Inspector XE		√	√
Intel® VTune™ Amplifier XE4		√	√
Intel® MPI Library4			√
Intel® Trace Analyzer and Collector			√
Operating System (Development Environment)	Windows* (Visual Studio*) Linux* (GNU) OS X*3 (XCode*)	Windows (Visual Studio) Linux (GNU)	Windows (Visual Studio) Linux (GNU)

Why Use Intel® Compilers?

Compatibility

- OS Support: Windows*, Linux*, OS X* (for OS/X , both traditional Intel C++ front end and CLANG version)
- IDE support: Visual Studio* in Windows*, Eclipse* in Linux*, Xcode* in OS X*
- Compatible with GNU* Compiler collection (gcc) – adapts to specific version up to 4.9
- Source and binary compatibility with Microsoft Visual C++* Compilers
- ISO Standard for C : almost full C99 compatibility, a few new features of C11
- ISO C++ Standard: all of C++11
- Full Fortran 2003, many features from Fortran 2008 including coarrays

Parallelism

- Language Extension (Intel Cilk® Plus™ for C/C++) for task parallelism
- Explicit Vector Programming (OpenMP* / Cilk SIMD, Array Notation)
- Support for OpenMP* 4.0 (except user-defined reductions)
- C++ Multithreading Library (Intel® TBB)

Why Use Intel® Compilers?

Performance

- Code generation tuned for latest microarchitecture
- Full – and very early - support of Intel processor instruction sets (SSE, AVX, AVX2, AVX-512)

Optimization

- Sophisticated optimization like interprocedural and profile-guided optimization
- Automatic vectorization
- Automatic parallelization
- Highly optimized version of libm (Intel® Math Library libimf) and vector math library libsvml

C++11 Support – Operating System Specifics

Linux

- Switch –std=c++11 (same as –std=c++0x) needed
- C++11 support very much depends on the GNU tools chain version installed (due to include files and library
  - Intel Compiler adapts to G++ version detected
  - 15.0 version supports all up to GNU 4.9 (4.8 or later is required for all of C++11)

Windows

- by default (no additional switch) we adapt to what the installed Microsoft Visual Studio* version supports
  - E.g. Intel Compiler 14.0 integrated into VS2013 supports all the Microsoft compiler supports
- Microsoft support is behind the one of Intel and GNU
- Support for all the C++11 feature of Intel requires switch /Qstd=c++11

Debugging of Lambda Functions (a C++11 Feature)

Debugging of Lambda functions has been a challenge – 15.0 closes the gap

```
int main()
{
  int cap = 0x123;

  auto lambda =
        [cap] (int par) -> int
  {
    int var = 0x456;
    return cap + par + var;
  };

  return lambda(0x789);
}
```

```
(gdb)breakpoint 1,
main::{lambda(int)#1::operator()(int) const {
   this=0x7fffffffb9e8, par=1929)
1return cap + par + var;
(gdb) p /x cap
$1 = 0x123
(gdb) p /x var
$2 = 0x456
(gdb) p /x par
$3 = 0x789
(gdb) p /x lambda
$1 = {
      captured = 0x123
}
```

Sample for Fortran 2008 Support: BLOCK Construct

- An executable construct that may contain declarations
- Variables declared within the construct are local to that scope
- No COMMON, EQUIVALENCE, NAMELIST, IMPLICIT
- SAVE allowed, local to that construct
- SAVE in outer scope does not affect BLOCK
- Labels and formats are not local
- Useful with DO CONCURRENT for threadlocals

```
IF (swaxpy) THEN
  BLOCK
    REAL(KIND(x)) tmp
    tmp = x
    x = y
    y = tmp
  END BLOCK
END IF
```

```
! without BLOCK, no way to define
! thread-local temp variable
DO CONCURRENT (I = 1:N)
  BLOCK
    REAL T
    T = A(I) + B(I)
    C(I) = T + SQRT(T)
  END BLOCK
END DO
```

Some Intel® C++ and Fortran Compiler Switches

	Windows*	Linux*
Disable optimization	/Od	**-O0**
Optimize for speed (no code size increase)	/O1	**-O1**
Optimize for speed (default)	/O2	**-O2**
More aggressive code (loop) transformations and optimizations – not necessarily better than –O2	/O3	**-O3**
Create symbols for debugging	/Zi	**-g**
Inter-procedural optimization for multiple files	/Qipo	**-ipo**
Profile guided optimization (muli-step build)	/Qprof-gen /Qprof-use	**-prof-gen** **-prof-use**
Optimize for speed across the entire program (not recommended on Linux due to static linking!)	/fast (same as: /O3 /Qipo /Qprec-div- /QxHost)	**-fast** (same as: **-ipo –O3 -no-prec-div -static -xHost**)
OpenMP support	/Qopenmp	**-openmp**
Automatic parallelization	/Qparallel	**-parallel**

Compatibility to Linux GNU Compilers

Compatibility between compilers a topic of four levels:

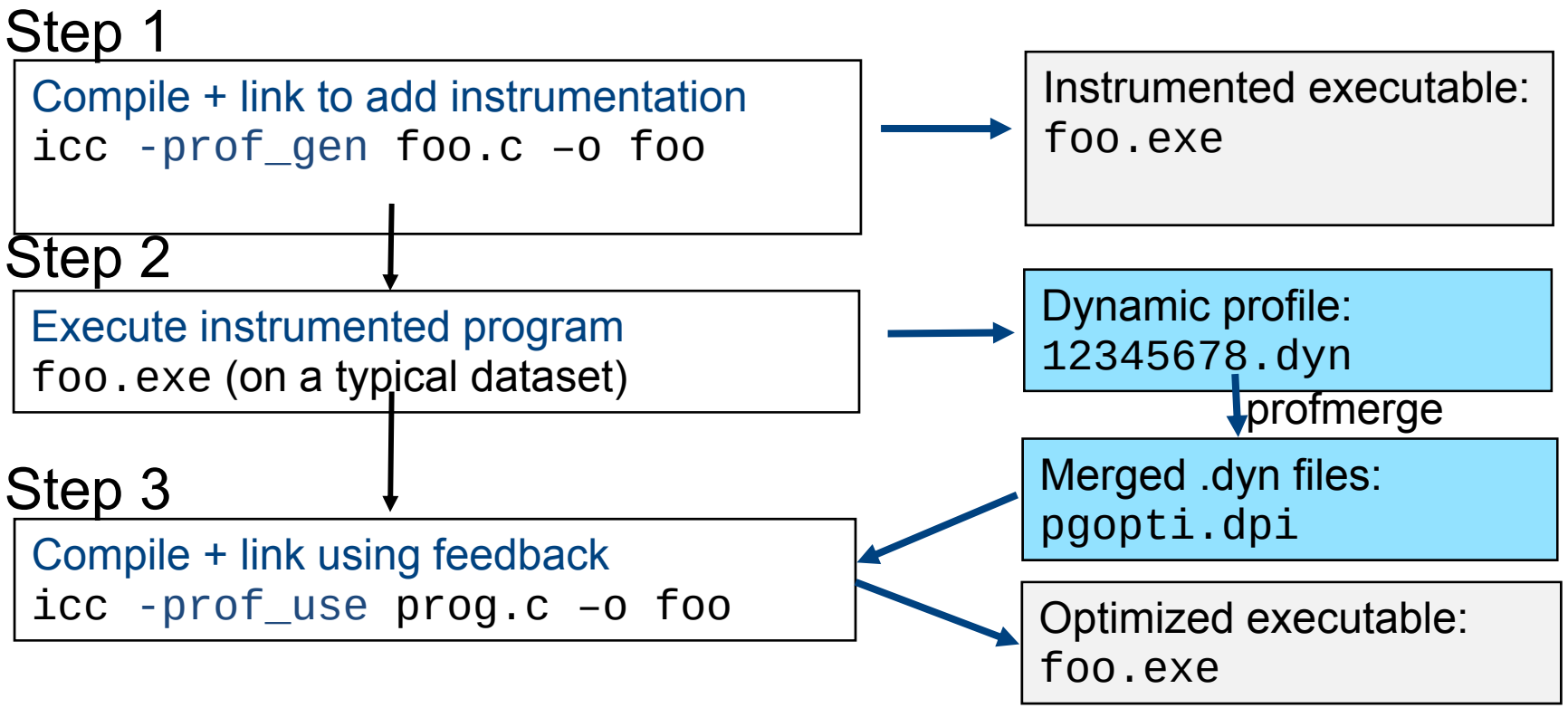
- Compatibility of object code including name mangling
  - Code compiled by ICC and GCC can be mixed without exceptions
  - Any incompatibility would be considered a very critical bug !!
- Compiler switches
  - Main switches of GCC and ICC are identical; both have specific options however
  - In most cases, gcc in a Makefile simply can be replaced by icc
- Source code language – features, syntax and semantic
  - for the relevant parts, full compatibility of ICC and GCC – in particular standard conformance (C and C++)
- OpenMP: Intel run time works for GCC too
  - No restriction to mix OpenMP of ICC- and GCC-compiled code

White paper “Intel® Compilers for Linux*: Compatibility with GNU Compilers” on software.intel.com has all details for all compiler releases

Some Intel-Specific Directives

Pragma	Description
`distribute, distribute_point`	instructs the compiler to prefer loop distribution at the location indicated
`inline`	instructs the compiler that the user prefers that the calls in question be inlined
`ivdep`	instructs the compiler to ignore assumed vector dependencies
`loop_count`	indicates the loop count is likely to be an integer
`novector`	specifies that the loop should never be vectorized
`optimization_level`	enables control of optimization for a specific function
`parallel/noparallel`	facilitates auto-parallelization of an immediately following DO loop; using keyword [always] forces the compiler to auto-parallelize; noparallel pragma prevents auto-parallelization
`simd`	enforces vectorization of innermost loops
`unroll/nounroll`	instructs the compiler the number of times to unroll/not to unroll a loop
`unroll_and_jam/ nounroll_and_jam`	instructs the compiler to partially unroll higher loops and jam the resulting loops back together. Specifying the nounroll_and_jam pragma prevents unrolling and jamming of loops.
`unused`	describes variables that are unused (warnings not generated)
`vector`	indicates to the compiler that the loop should be vectorized according to the arguments: always/aligned/unaligned/nontemporal/temporal

PGO Usage: Three Step Process



Simple PGO Example: Code Re-Order

```
for (i=0; i < NUM_BLOCKS; i++)
{
    switch (check3(i))
    {
        case 3:                         /* 25% */
            x[i] = 3;  break;
        case 10:                        /* 75% */
            x[i] = i+10;  break;
        default:                        /* 0% */
            x[i] = 99;  break
    }
}
```

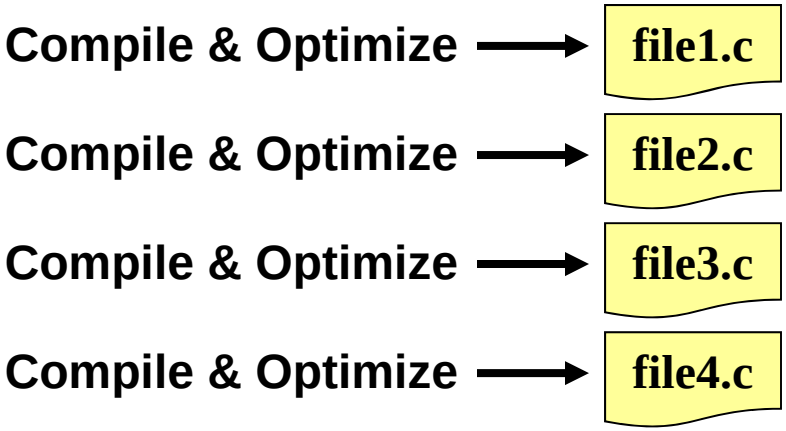
"Case 10" is moved to the beginning

- PGO can eliminate most tests&jumps for the common case – less branch mispredicts

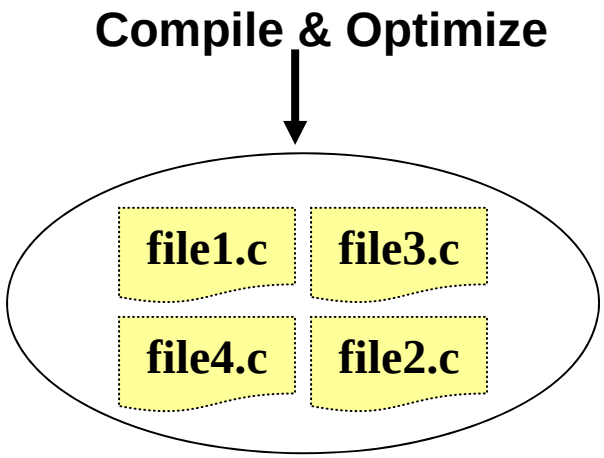
Interprocedural Optimizations

Extends optimizations across file boundaries

Without IPO



With IPO

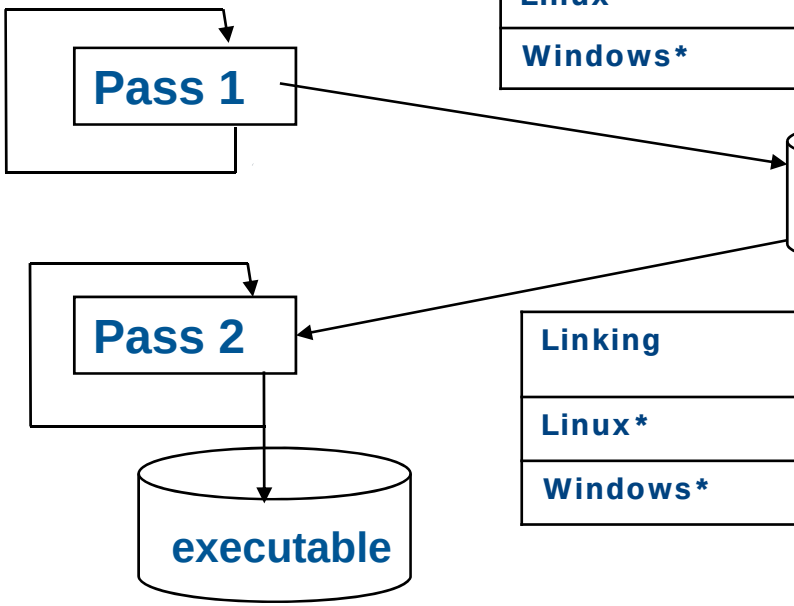


`/Qip, -ip`	Only between modules of one source file
`/Qipo, -ipo`	Modules of multiple files/whole application

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`



Intermediate language (mock) object

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

OpenMP 4.0

OpenMP* 4.0 Specification

Released July 2013

- http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
- A document of examples is expected to release soon

Changes from 3.1 to 4.0 (Appendix E.1):

- SIMD directives
- Device/Accelerator directives
- Taskgroup and dependant tasks
- Thread affinity
- Cancellation directives
- User-defined reductions
- Sequentially consistent atomics
- Extended Fortran2003 support

Support by Intel® C/C++ and Fortran Compilers

In 14.0 Update 2

- Places and thread affinity
- Main features of SIMD directives
- Main features of Device / Accelerator directives
- Sequentially consistent atomics

In 15.0

- Combined SIMD, parallel, target, teams, distribute constructs
- Taskgroup and dependent tasks
- Cancellation directives
- Extended Fortran 2003 support

TBD:

- User-defined reductions

Loop Profiler

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 instrument able loops of the type listed as the option's argument.

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

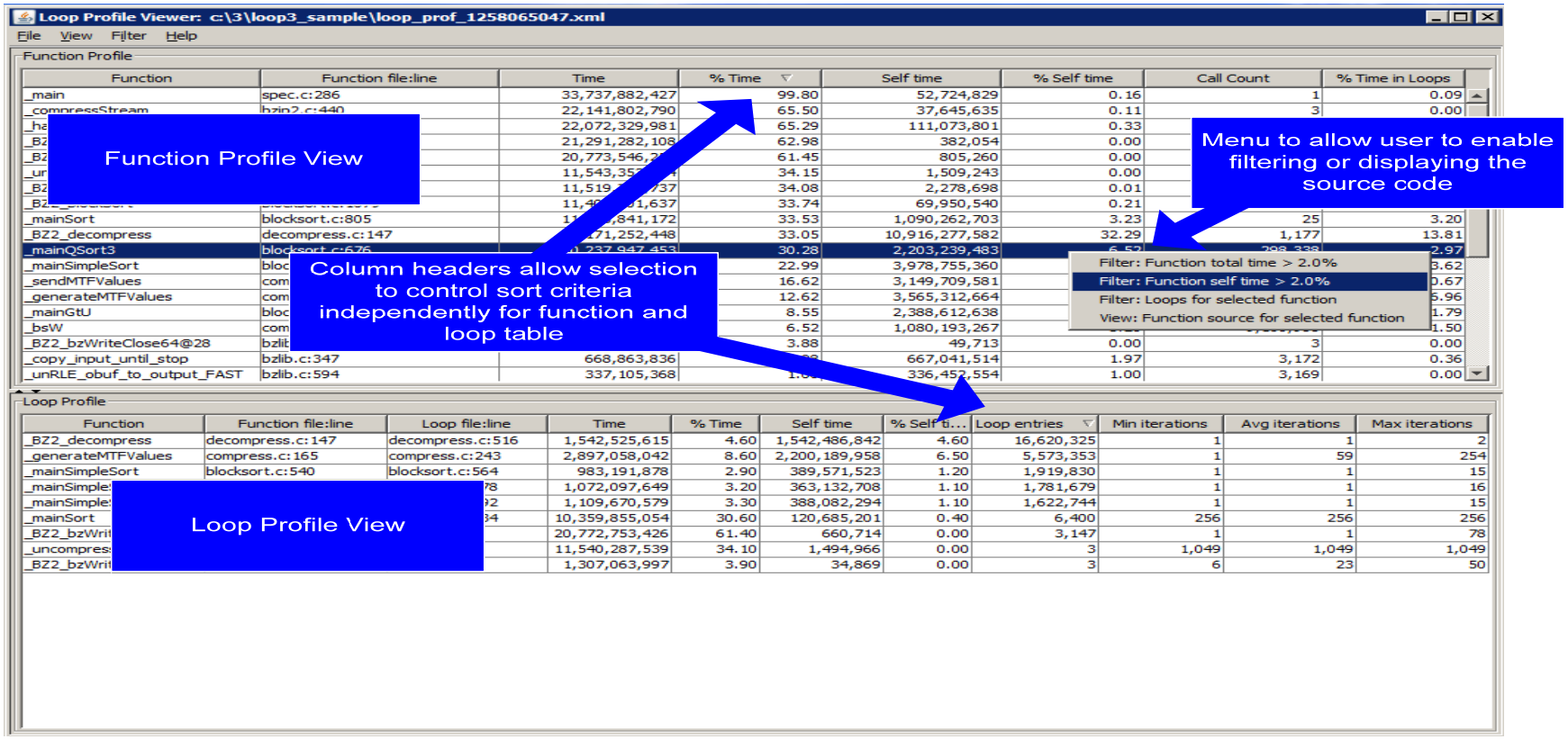
- Invocation of XML viewer on command line:
- `java -jar loopprofviewer.jar <xml datafile>`

Note: Can't be used on multi-threaded code yet

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



Optimization Reports

Optimization Report Redesign

- Old functionality implemented under **`-opt-report`**, **`-vec-report, -openmp-report, -par-report`** replaced by unified **`-opt-report`** compiler options
  - **`[vec,openmp,par]-report`** options deprecated and map to equivalent opt-report-phase
- Can still select phase with **`-opt-report-phase`** option. For example, to only get vectorization reports, use **`-opt-report-phase=vec`**
- Output now defaults to a **`<name>.optrpt`** file where **`<name>`** corresponds to the output object name. This can be changed with **`-opt-report-file=[<name>|stdout|stderr]`**
- Windows*: **`/Qopt-report, /Qopt-report-phase=<phase> etc`**
  - Optimization report integrated into Microsoft Visual Studio*

Let's take a Look at an Example

```
 1  double a[1000][1000],b[1000][1000],c[1000][1000];
 2
 3  void foo() {
 4    int i,j,k;
 5
 6    for( i=0; i<1000; i++) {
 7      for( j=0; j< 1000; j++) {
 8        c[j][i] = 0.0;
 9        for( k=0; k<1000; k++) {
10          c[j][i] = c[j][i] + a[k][i] * b[j][k];
11        }
12      }
13    }
14  }
```

15.0 Loop Optimization Report

```
Report from: Loop nest,  Vector optimizations [loop, vec, par]


LOOP BEGIN at d:\iusers\hsaito\ics\15_0\dev\test.c(7,5)
 Distributed chunk1
   remark #25430: LOOP DISTRIBUTION (2 way)
   remark #25448: Loopnest Interchanged : ( 1 2 ) --> ( 2 1 )
   remark #25424: Collapsed with loop at line 6
   remark #25412: memset generated
   remark #15144: loop was not vectorized: loop was transformed to
memset or memcpy
LOOP END
```

```
LOOP BEGIN at d:\iusers\hsaito\ics\15_0\dev\test.c(7,5)
Distributed chunk2
   remark #25448: Loopnest Interchanged : ( 1 2 3 ) --> ( 2 3 1 )
   remark #15018: loop was not vectorized: not inner loop
   LOOP BEGIN at d:\iusers\hsaito\ics\15_0\dev\test.c(9,7)
   Distributed chunk2
      remark #15018: loop was not vectorized: not inner loop

      LOOP BEGIN at d:\iusers\hsaito\ics\15_0\dev\test.c(6,3)
         remark #15145: vectorization support: unroll factor set to 4
         remark #15003: PERMUTED LOOP WAS VECTORIZED
      LOOP END

      LOOP BEGIN at d:\iusers\hsaito\ics\15_0\dev\test.c(6,3)
         remark #15003: REMAINDER LOOP WAS VECTORIZED
      LOOP END
   LOOP END
LOOP END
```

Clearer view of what happens where and in what order

Optimization Report Phases

The compiler reports optimizations from 9 phases:

LOOP: Loop Nest Optimizations
PAR: Auto-Parallelization
VEC: Vectorization
OPENMP: OpenMP
OFFLOAD: Offload

IPO: Interprocedural Optimizations
PGO: Profile Guided Optimizations
CG: Code Generation Optimizations
TCOLLECT: Trace Analyzer Collection

LOOP/PAR/VEC share a unified loop structure, a hierarchical output, to seamlessly display optimizations in an integrated format; list of phases significantly simplified from 14.0

Selecting phases for compiler optimization reporting is highly customizable to satisfy customers' specific requirements.

- Single Phase Reporting:
  - Compiler Option: -[Q]opt-report-phase=VEC
- Multiple Phase Reporting (use a comma separated list):
  - Compiler Option: -[Q]opt-report-phase=VEC, OPENMP, IPO, LOOP
- Default is "ALL" phases and default reporting verbosity level is 2
  - Want to encourage use of integrated HPO report instead of just vec-report[n]
  - Lot of changes from 14.0 to remove extraneous information

Optimization Report Levels

The compiler's optimization report have 5 verbosity levels.

- Specifying report verbosity level:
  - Compiler Option: –opt-report=N where N = level of desired verbosity

When option omitted, default N=2.

  - For each optimization phase, higher verbosity level indicates higher level of detail reported.
  - Each verbosity level is inclusive of lower levels.
- Example, VEC Phase Levels:
  - Level 1: Reports when vectorization has occurred.
  - Level 2: Adds diagnostics why vectorization did not occur.
  - Level 3: Adds vectorization loop summary diagnostics.
  - Level 4: Adds additional available vectorization support information.
  - Level 5: Adds detailed data dependency information diagnostics.
- **Each** phase can support up to 5 levels

Example Code for IPO Opt Report

```
 1  #include <stdio.h>
 2
 3  static void __attribute__((noinline)) bar(float
                    a[100][100], float b[100][100]) {
 4   int i, j;
 5   for (i = 0; i < 100; i++) {
 6        for (j = 0; j < 100; j++) {
 7           a[i][j] = a[i][j] + 2 * i;
 8            b[i][j] = b[i][j] + 4 * j;
 9       }
10    }
11 }
12
13   static void foo(float a[100][100], float b[100][100])
{
14   int i, j;
15    for (i = 0; i < 100; i++) {
16       for (j = 0; j < 100; j++) {
17         a[i][j] = 2 * i;
18           b[i][j] = 4 * j;
19       }
20    }
21    bar(a, b);
22 }
23
```

```
24   extern int main() {
25   int i, j;
26    float a[100][100];
27     float b[100][100];
28
29   for (i = 0; i < 100; i++) {
30        for (j = 0; j < 100; j++) {
31           a[i][j] = i + j;
32            b[i][j] = i - j;
33       }
34    }
35    foo(a, b);
36    foo(a, b);
37    fprintf(stderr, "%d %d\n", a[99][9], b[99][99]);
38   }
```

Compiled with:

```
icc  -opt-report=L –opt-report-phase=ipo  sm.c
```

with L = 1, 2, 3, 4, 5

Level 5 for Vectorization Report

```
VECRPT (col. 3) LOOP WAS VECTORIZED.
VECRPT (col. 3) entire loop may be executed in scalar remainder
VECRPT (col. 3) estimated potential speedup: 3.540000.
VECRPT (col. 3) lightweight vector operations: 26.
VECRPT (col. 3) loop inside vectorized loop at nesting level: 1.
VECRPT (col. 3) loop was vectorized (with peel/with remainder)
VECRPT (col. 3) medium-overhead vector operations: 10.
VECRPT (col. 3) scalar loop cost: 14.
VECRPT (col. 3) unmasked aligned unit stride loads: 4.
VECRPT (col. 3) unmasked aligned unit stride stores: 4.
VECRPT (col. 3) unmasked unaligned unit stride loads: 8.
VECRPT (col. 3) unmasked unaligned unit stride stores: 2.
VECRPT (col. 3) unroll factor set to 2.
VECRPT (col. 3) vector loop cost: 7.500000.

   6:   do i=1,n
   7:       a(i)= a(i)-b(i)*d(i)
   8:       c(i)= a(i)+c(i)
   9:   enddo
```

Optimization Notes in Text Editor

Optimization notes at callee site

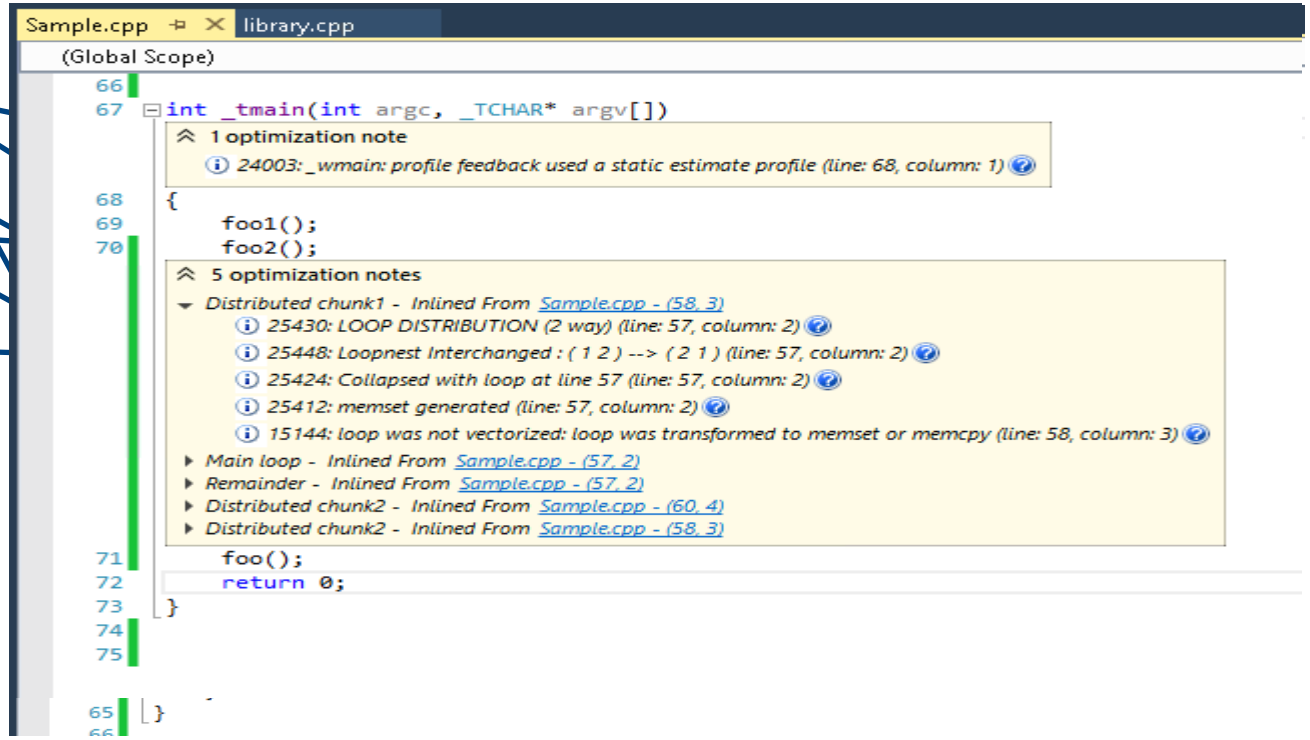
Report content

Jump to call site by click

Click '?' to get help for message

Optimization notes at call site

Jump to callee site by click

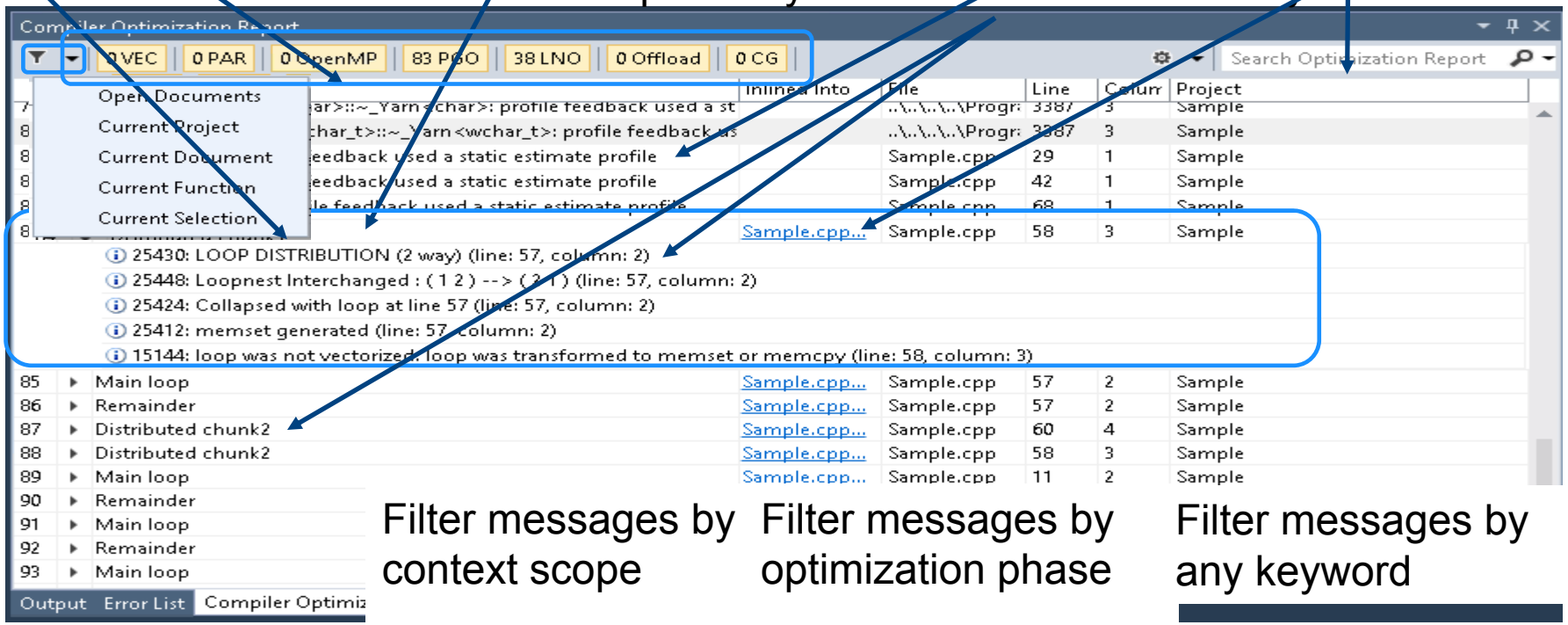


Compiler Optimization Report Tool Window

Hierarchically Presented Report

Jump to source position by double click

Jump to call site by click



Filter messages by context scope

Filter messages by optimization phase

Filter messages by any keyword

Control of Floating Point Operations

Compiler Optimizations for FP Operations

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 <model>`**
Windows*: **`/fp:<model>`**

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 except-	*/** *	* *	* *	Yes No	* *
*					

* hese modes are unaffected. -fp-

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 vectorization (e.g. partial sums)!

```
#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";
}
```

Data Parallel Extensions

(Explicit Coding for SIMD Parallelism)

Adding a new Way to Vectorize

Compiler:
Auto-vectorization (no change of code)

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

Explicit Vector Programming by Cilk Plus data-parallel extensions

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`, …)

Ease of use

Programmer control

Data Parallel Enhancements

Serial Code

```
for(i = 0; i < N; i++){
  A[i] = B[i] + C[i];
}
```

Array Notation for C/C++

```
A[:] = B[:] + C[:];
```

SIMD Pragma/Directive

```
#pragma simd

for(i = 0; i < N; i++)
  A[i] = B[i] + C[i+delta];
```

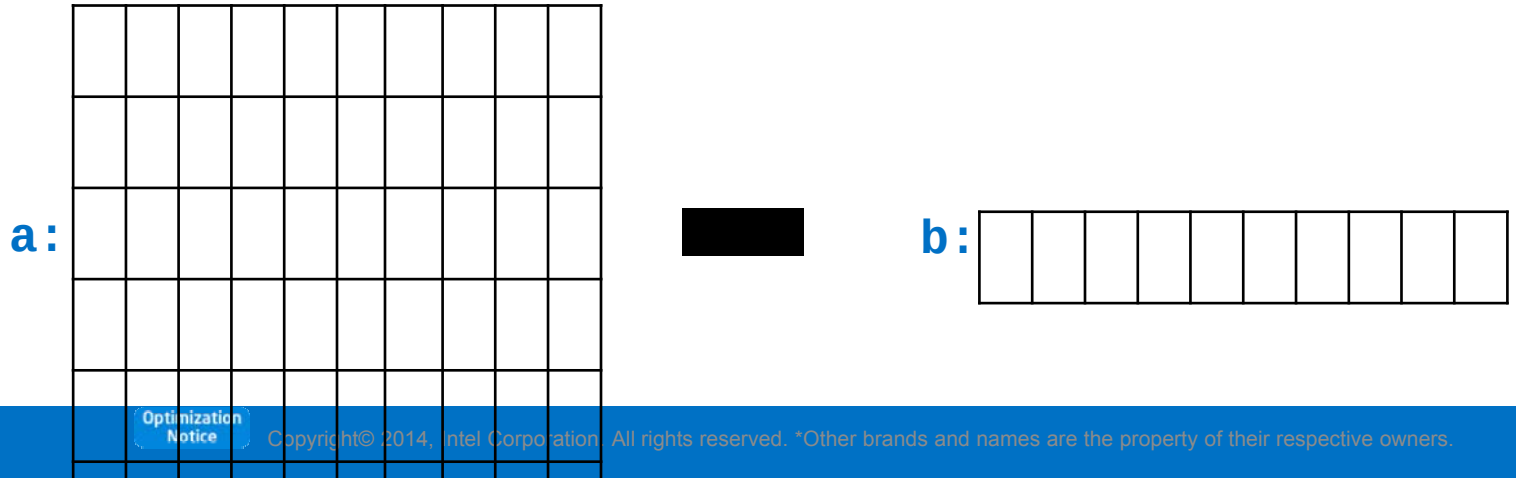
SIMD-Enabled Function

```
__declspec(vector(vectorlength(VLEN)))
float vadd(float B, float C)
{
  return B + C;
}
…
for(i = 0; i < N; i+=VLEN)
  A[i:VLEN] = vadd(B[i:VLEN], C[i:VLEN]);
```

Intel® Cilk™ Plus Array Notation Example

Section of 2D array:

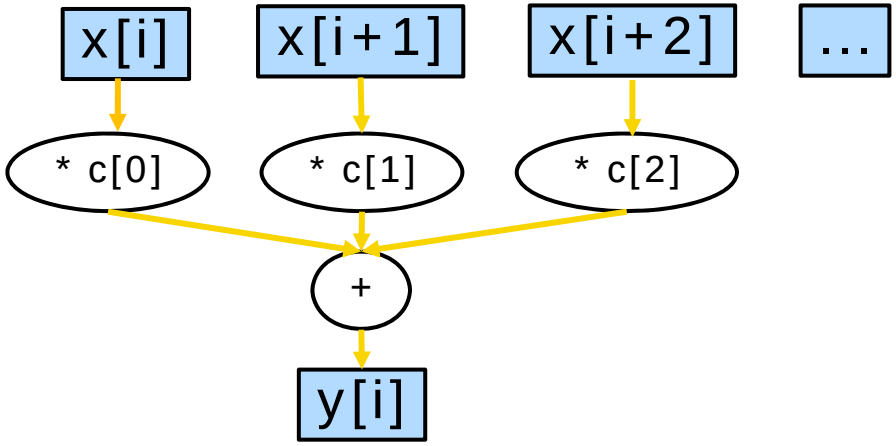
```
float a[10][10], *b;
…
// allocate space for 10 elements for *b
…
b[0:10] = a[:][5];
…
```



Sample Array Section Notation: FIR Filter

Scalar code:

```
for (i=0; i < M-K; i++) {
    s = 0;
    for (j = 0; j < K; j++)
        s += x[i+j] * c[j]
    y[i] = s;
}
```



Inner (j) loop vectorized:

```
for (i=0; i < M-K; i++)
    y[i] = __sec_reduce_add(x[i:K] * c[0:K]); // perform K multiplications in parallel
                                              // and add up the products
```

Outer (i) loop vectorized:

```
y[0:M-K] = 0;                    // calculate all the y[i] results in parallel
for (j = 0; j < K; j++)
    y[0:M-K] += x[j:M-K] * c[j]
```

Summary / Call to Action

- Intel® Composer XE 2015 introduces key new features like
  - New vetorization and optimization reporting
  - Extended platform support of GT compiler
  - Standard support extensions for C++11, FORTRAN 03/08 and OpenMP 4.0
- Please use new compiler & libraries :
  - New features check (in particular C++11 / Fortran 03/08)
  - Performance comparison
  - Optimization reports - what do you still miss ?

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.

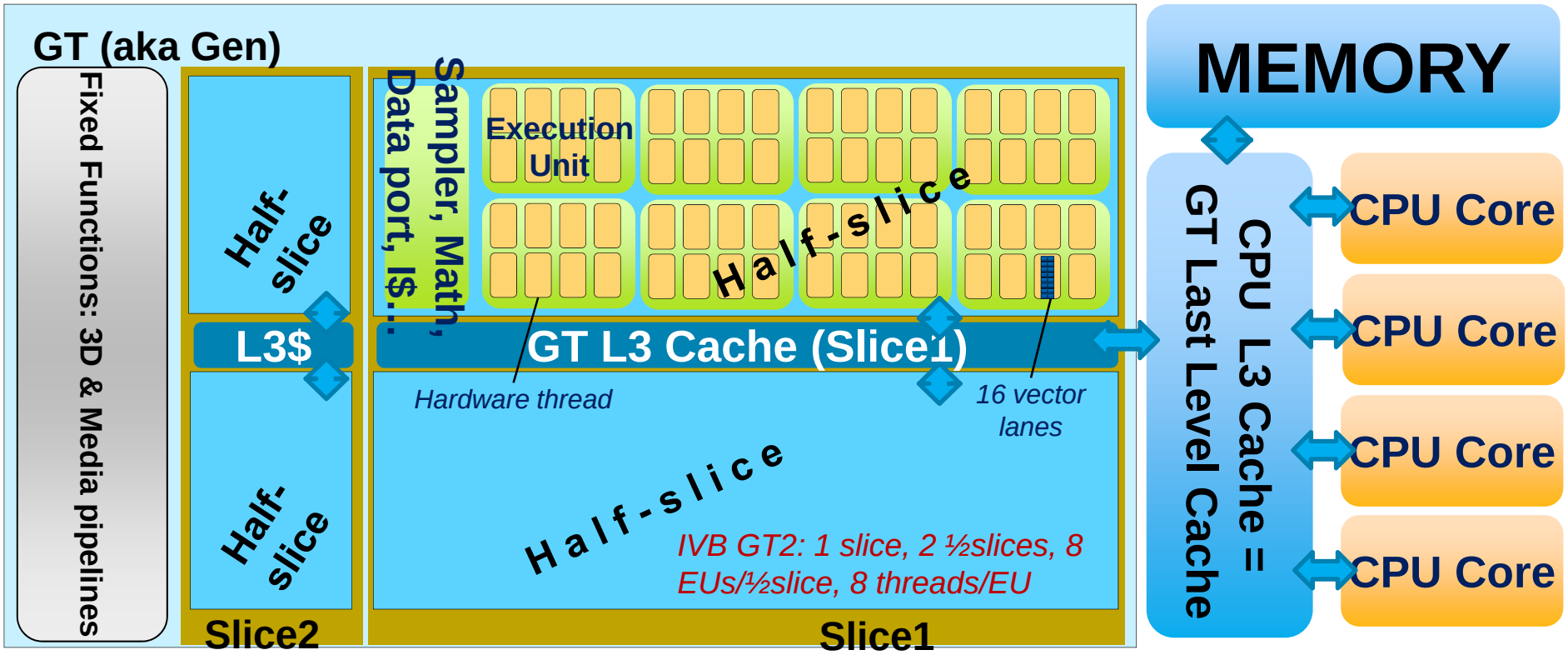
Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may 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.

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.

Selected Topic: GT Compiler

Intel® Graphics Technology Architecture

GT building block hierarchy: Slice/Half slice/EU/thread/vector lane



GT Compiler Status before Version 15.0

- Exists as a preview (but undocumented) feature in 14.0
- Windows / IVB+ only.
- 32-bit apps only
- Documentation internally available
- Requires Intel HD Graphics driver and Media Development Framework
- Test program with some 20 ISVs

Hardware and OS platforms supported in 15.0

- Hardware support

3rd/4th generation Intel® Core™ Processors	Intel® Pentium® Processors	Intel® Celeron® Processors
Intel® HD Graphics 2500 Intel® HD Graphics 4000	Intel® HD Graphics Model Numbers: 20xx	Intel® HD Graphics Model Numbers: 10xx
Intel® HD Graphics 4200 Intel® HD Graphics 4400 Intel® HD Graphics 4600 Intel® HD Graphics 5000 Intel® HD Graphics 5100 Intel® HD Graphics 5200	Intel® HD Graphics Model Numbers: 3xxx	Intel® HD Graphics Model Numbers: 29xx

- For more information on hardware, please visit: http://ark.intel.com
- Operating systems:
  - Windows* 32/64 bit. On Windows* 7, the system shouldn't be locked for GFX offload to work.
  - Linux* 64 bit (Only Ubuntu 12.04 and SLES11 initially)
  - Support for CentOS and RHEL6.x coming later
  - No OS X* and Android* support

Synchronous and Asynchronous Offload

Synchronous offload :

- Annotate data parallel code section with #pragma offload target(gfx)
- Annotate functions invoked from the offloaded sections and global data with __declspec(target(gfx))
- Host thread waits for the offloaded code to finish execution
- Constraint - #pragma offload target(gfx) statement should be followed by a cilk_for loop
- *Compiler automatically generates both, host side as well as GFX code*

Asynchronous offload :

- An API based offload solution
- By annotating functions with __declspec(target(gfx_kernel))
- Above annotation creates the named kernel functions (Kernel entry points)
- Non-blocking API calls from CPU until the first explicit wait is specified.
- GFX kernels are en-queued into an in-order GPGPU queue
- Explicit control over data transfer, data decoupled from Kernel, data persistence across multiple kernel executions.
- *Compiler just generates the GFX code.*
  - *User has to explicitly code the host version of offload section*

Ease of Porting to GFX

Original Host code:

```
void vector_add(float *a, float *b, float *c){
for(int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
return;
}
```

Parallel Host code using Intel® Cilk™ Plus:

```
void vector_add(float *a, float *b, float *c){
cilk_for(int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
return;
}
```

Offloading function body to GFX:

```
void vector_add(float *a, float *b, float *c){
#pragma offload target(gfx) pin(a, b, c:length(N))
cilk_for(int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
return;
}
```

Creating GFX kernel for asynchronous offload:

```
__declspec(target(gfx_kernel))
void vector_add(float *a, float *b, float *c){
cilk_for(int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
return;
}
```

Asynchronous offload – Vector Addition

Host Code

```
float *a = new float[TOTALSIZE];
float *b = new float[TOTALSIZE];
float *c = new float[TOTALSIZE];
float *d = new float[TOTALSIZE];

a[0:TOTALSIZE] = 1;
b[0:TOTALSIZE] = 1;
c[0:TOTALSIZE] = 0;
d[0:TOTALSIZE] = 0;

_GFX_share(a, sizeof(float)*TOTALSIZE);
_GFX_share(b, sizeof(float)*TOTALSIZE);
_GFX_share(c, sizeof(float)*TOTALSIZE);
_GFX_share(d, sizeof(float)*TOTALSIZE);

_GFX_enqueue("vec_add", c, a, b, TOTALSIZE); // Non-blocking offload
_GFX_enqueue("vec_add", d, c, a, TOTALSIZE); // Place next kernel in
                                             // in-order queue
_GFX_wait();                                 // wait for all tasks

_GFX_unshare(a);
_GFX_unshare(b);
_GFX_unshare(c);
_GFX_unshare(d);
```

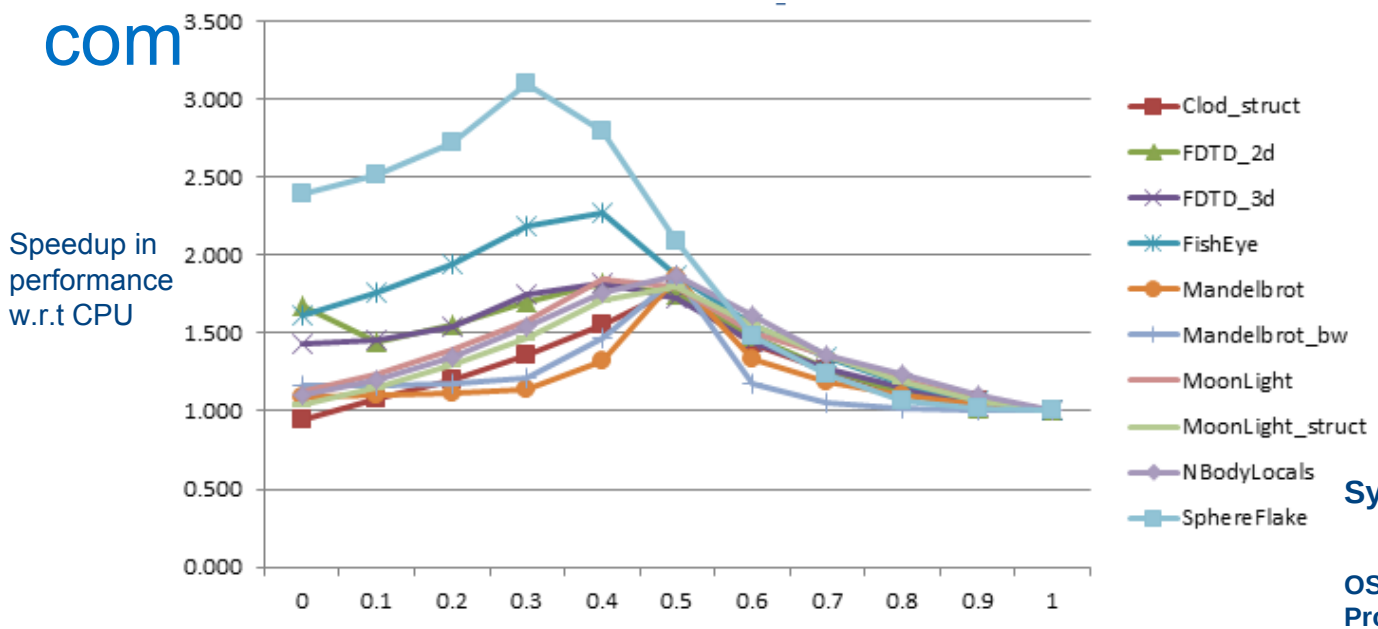
GPU Code

```
__declspec(target(gfx_kernel))
void vec_add(float *res, float *a, float *b, int size){
    cilk_for (int i = 0; i < size; i++)
    {
        res[i] = a[i] + b[i];
    }
    return;
}
```

Limitations

- Main language restrictions
  - No exceptions, RTTI, longjmp/setjmp, VLA, variable parameter list, indirect control flow (virtual functions, function pointers, indirect calls and jumps)
  - No shared virtual memory
  - No pointer or reference typed globals
  - No OpenMP* or Intel® Cilk™ Plus tasking
- Runtime limitations
  - No ANSI C runtime library except math SVML library.
  - Inefficient 64-bit float and integer (due to HW limitations)
- No debugger support for GFX as of now but being worked on (based on GDB)

Preliminary Performance gain using GFX + CPU com



Left to right -> Pure GPU execution to Pure CPU execution
0 – Pure GPU execution mode , 1- Pure CPU execution mode
0.1 – 10% workload on CPU and 90% on GPU
0.9 – 10% workload on GPU and 90% on CPU

System Specification:

OS : Ubuntu 12.04 (64 bit)
Processor: Intel® Core™ i7-3770 @ 3.5GHz
Memory : 16GB
Processor Graphics SKU : GT2
Compiler version : Intel C++ Compiler 15.0 Update 1 Beta
HD Driver : 16.3.2.21305
Compiler option : -std=c++11 -xAVX