COMPILER OPTIMIZATIONS

What's New for Intel compilers 19.1?

Advance Support for Intel® Architecture – Use Intel compiler to generate optimized code for Intel Atom® processor through Intel® Xeon® Scalable processor and Intel® Xeon Phi™ processor families

Achieve Superior Parallel Performance – Vectorize & thread your code (using OpenMP*) to take full advantage of the latest SIMD-enabled hardware, including Intel® Advanced Vector Extensions 512 (Intel® AVX-512)

What's New in C++

Initial C++20, and full C++ 17 enabled

- Enjoy advanced lambda and constant expression support
- Standards-driven parallelization for C++ developers

Initial OpenMP* 5.0, and full OpenMP* 4.5 support

- Modernize your code by using the latest parallelization specifications

What's New in Fortran

Substantial Fortran 2018 support

- Enjoy enhanced C-interoperability features for effective mixed language development
- Use advanced coarray features to parallelize your modern Fortran code

Initial OpenMP* 5.0, and substantial OpenMP* 4.5 support

- Customize your reduction operations by user-defined reductions

Common optimization options

	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 ("prototype switch") ***fast* options definitions changes over time!**	-fast same as: -ipo –O3 -no-prec-div –static –fp-model fast=2 -xHost)
OpenMP support	-qopenmp
Automatic parallelization	-parallel

High-Level Optimizations

Basic Optimizations with icc -O…

-O0 no optimization; sets -g for debugging

-O1 scalar optimizations
excludes optimizations tending to increase code size

-O2 **default** for icc/icpc (except with -g)
includes **auto-vectorization**; some loop transformations, e.g. unrolling, loop interchange;
inlining within source file;
start with this (after initial debugging at -O0)

-O3 more aggressive loop optimizations
including cache blocking, loop fusion, prefetching, …
suited to applications with loops that do many floating-point calculations or process large data sets

InterProcedural Optimizations (IPO)

Multi-pass Optimization

`icc -ipo`

Analysis and optimization across function and/or source file boundaries, e.g.

- Function inlining; constant propagation; dependency analysis; data & code layout; etc.

2-step process:

- Compile phase – objects contain intermediate representation
- “Link” phase – compile and optimize over all such objects
- Seamless: linker automatically detects objects built with -ipo and their compile options
- May increase build-time and binary size
- But build can be parallelized with `-ipo=n`
- Entire program need not be built with IPO, just hot modules

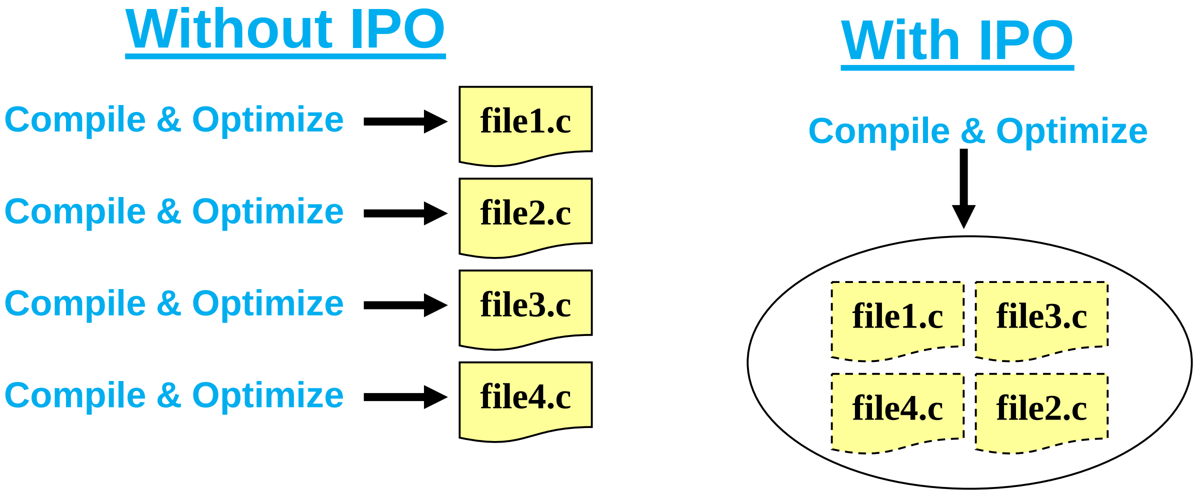
Particularly effective for applications with many smaller functions

Get report on inlined functions with `-qopt-report-phase=ipo`

InterProcedural Optimizations

Extends optimizations across file boundaries

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



Profile-Guided Optimizations (PGO)

Static analysis leaves many questions open for the optimizer like:

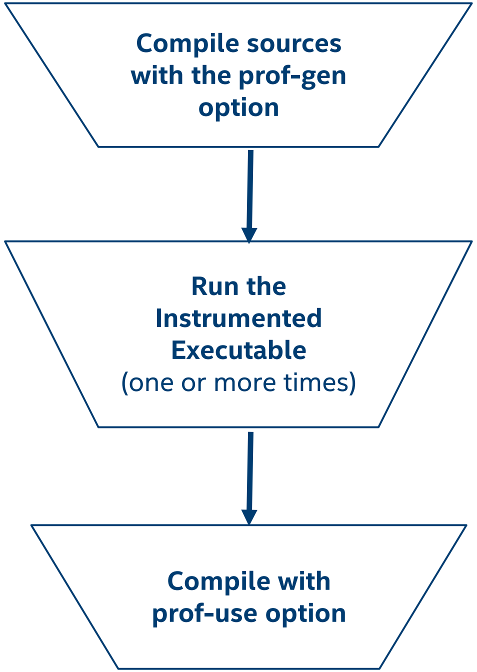
- How often is x > y
- What is the size of count
- Which code is touched how often

```
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 -o prog`

Instrumented executable:
`prog`

Step 2

Execute instrumented program
`./prog` (on a typical dataset)

Dynamic profile:
`12345678.dyn`

Step 3

Compile + link using feedback
`icc -prof-use prog.c -o prog`

Merged .dyn files:
`pgopti.dpi`

Optimized executable:
`prog`

Math Libraries

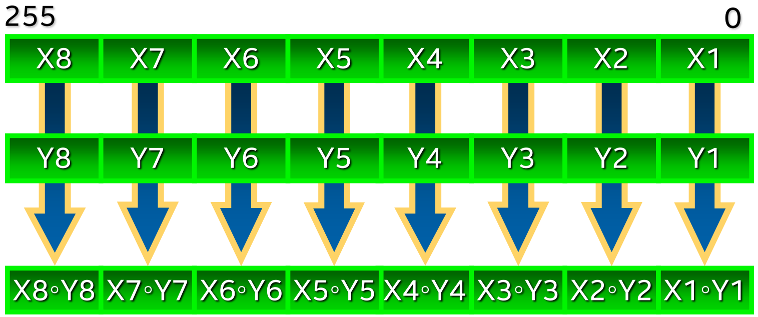
icc comes with Intel's optimized math libraries

- libimf (scalar) and libsvml (scalar & vector)
- Faster than GNU* libm
- Driver links libimf automatically, ahead of libm
- Additional functions (replace math.h by mathimf.h)

Don't link to libm explicitly! -lm

- May give you the slower libm functions instead
- Though the Intel driver may try to prevent this
- gcc needs -lm, so it is often found in old makefiles

SIMD Types for Intel® Architecture



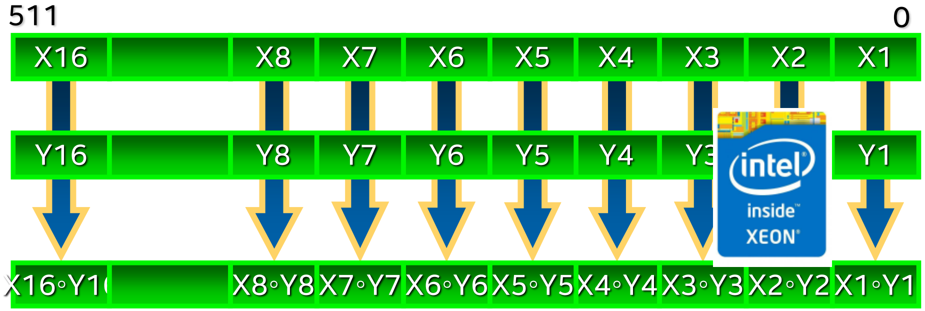
AVX

Vector size: **256 bit**

Data types:

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

VL: 4, 8, 16, 32



Intel® AVX-512

Vector size: **512 bit**

Data types:

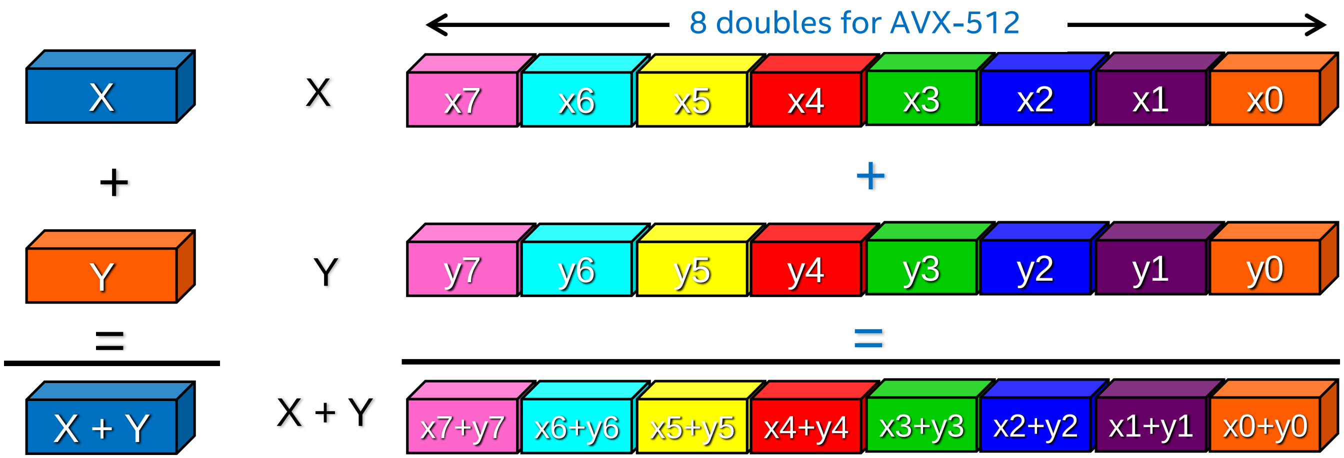
- 8, 16, 32, 64 bit integer
- 32 and 64 bit float

VL: 8, 16, 32, 64

SIMD: Single Instruction, Multiple Data

```
for (i=0; i<n; i++) z[i] = x[i] + y[i];
```

- Scalar mode
  - one instruction produces one result
  - E.g. vaddss, (vaddsd)
- Vector (SIMD) mode
  - one instruction can produce multiple results
  - E.g. vaddps, (vaddpd)



Many ways to vectorize

Compiler:
Auto-vectorization (no change of code)

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

Compiler:
Explicit vectorization with OpenMP* 4.0 & later

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

Basic Vectorization Switches I

- Requirement: -O2 or higher
- -x<feature>
  - Might enable Intel processor specific optimizations
  - Processor-check added to “main” routine:
    Application errors in case SIMD feature missing or non-Intel processor with appropriate/informative message
- -ax<features>
  - Multiple code paths: baseline and optimized/processor-specific
  - Optimized code paths for Intel processors defined by <features>
  - Multiple SIMD features/paths possible, e.g.: -axSSE2,AVX,AVX2
  - Baseline code path defaults to –msse2
  - The baseline code path can be modified by –m<feature> or –x<feature>

Basic Vectorization Switches II

- -m<feature>
  - Neither check nor specific optimizations for Intel processors:
    Application optimized for both Intel and non-Intel processors for selected SIMD feature
- Default for Linux: -msse2
  - Activated implicitly
  - Implies the need for a target processor with at least Intel® SSE2
- Special switch: -xHost
  - Compiler checks SIMD features of current host processor (where built on) and makes use of latest SIMD feature available
  - Code only executes on processors with same SIMD feature or later as on build host
  - As for -x<feature> , if "main" routine is built with –xHost or the final executable only runs on Intel processors

Compiler Reports – Optimization Report

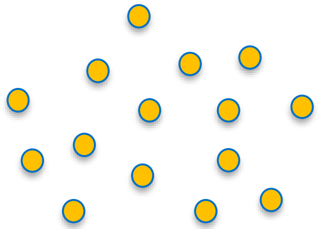
- -qopt-report[=n]: tells the compiler to generate an optimization report
  - n: (Optional) Indicates the level of detail in the report. You can specify values 0 through 5. If you specify zero, no report is generated. For levels n=1 through n=5, each level includes all the information of the previous level, as well as potentially some additional information. Level 5 produces the greatest level of detail. If you do not specify n, the default is level 2, which produces a medium level of detail.
- -qopt-report-phase[=list]: specifies one or more optimizer phases for which optimization reports are generated.
  - loop: the phase for loop nest optimization
  - vec: the phase for vectorization
  - par: the phase for auto-parallelization
  - all: all optimizer phases
- -qopt-report-filter=string: specified the indicated parts of your application, and generate optimization reports for those parts of your application.

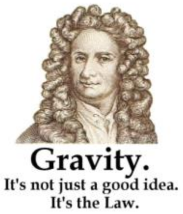
Lab: Nbody gravity simulation

Let's consider a distribution of point masses located at `r_1,…,r_n` and have masses `m_1,…,m_n`

We want to calculate the position of the particles after a certain time interval using the Newton law of gravity

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





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

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

Lab: Nbody kernel implementation

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

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

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

Exercise 1 – nbody-demo/ver0

- module load intel64/19.1up01
- Go to the folder nbody-demo/ver0
  - Take a look on the code to learn data structures, main loops, Flops calculation, etc.
- Type *make* to compile code

```
icpc -g -std=c++11 -O2  -o nbody.x GSimulation.o main.o
```

- Type *make run* to run the test and measure the timing.

```
./nbody.x
```

- If everything works fine, you should see the following results:

Exercise 1 – nbody-demo/ver0

```
Run the default test case on CPU:
./nbody.x
===============================
 Initialize Gravity Simulation
 nPart = 16000; nSteps = 10; dt = 0.1
------------------------------------------------
 s        dt       kenergy     time (s)    GFlops
------------------------------------------------
 1        0.1      26.405      5.1986      1.4281
 2        0.2      313.77      5.2053      1.4263
 3        0.3      926.56      5.3105      1.398
 4        0.4      1866.4      5.2772      1.4069
 5        0.5      3135.6      5.1825      1.4326
 6        0.6      4737.6      5.1845      1.432
 7        0.7      6676.6      5.1921      1.4299
 8        0.8      8957.7      5.2037      1.4267
 9        0.9      11587       5.2009      1.4275
 10       1        14572       5.1825      1.4326

# Number Threads         : 1
# Total Time (s)         : 52.139
# Average Performance : 1.4233 +- 0.012399
===============================
```

- Observe how the performance changes according to the number of particles: `./nbody.x 8000`
- Is the perfomance saturating?
- What is the maximum perfomance?
- Has the code performance issues?

Exercise 1 – nbody-demo/ver0

```
Run the default test case on CPU:
./nbody.x
===============================
 Initialize Gravity Simulation
 nPart = 16000; nSteps = 10; dt = 0.1
------------------------------------------------
 s         dt        kenergy      time (s)     GFlops
------------------------------------------------
 1         0.1       26.405       5.1986       1.4281
 2         0.2       313.77       5.2053       1.4263
 3         0.3       926.56       5.3105       1.398
 4         0.4       1866.4       5.2772       1.4069
 5         0.5       3135.6       5.1825       1.4326
 6         0.6       4737.6       5.1845       1.432
 7         0.7       6676.6       5.1921       1.4299
 8         0.8       8957.7       5.2037       1.4267
 9         0.9       11587        5.2009       1.4275
 10        1         14572        5.1825       1.4326

# Number Threads      : 1
# Total Time (s)      : 52.139
# Average Performance : 1.4233 +- 0.012399
===============================
```

- Observe how the performance changes according to the number of particles: `./nbody.x 8000`
- Is the perfomance saturating?
- What is the maximum perfomance?
- Has the code performance issues?
- Generate the compiler report:

  `make REPORT=yes`

  or

  `icpc -g -std=c++11 -O2` **`-qopt-report=5`** `-o nbody.x GSimulation.cpp main.cpp`

Exercise 1 – nbody-demo/ver0

```
LOOP BEGIN at GSimulation.cpp(127,20)
   remark #15542: loop was not vectorized: inner loop
was already vectorized

   LOOP BEGIN at GSimulation.cpp(130,5)
      remark #15542: loop was not vectorized: inner
loop was already vectorized
.....

remark #15417: vectorization support: number of FP up
converts: single precision to double precision 1   [
GSimulation.cpp(143,4) ]

....
remark #15300: LOOP WAS VECTORIZED
         remark #15452: unmasked strided loads: 6
         remark #15475: --- begin vector cost summary
---
         remark #15476: scalar cost: 145
         remark #15477: vector cost: 79.500
         remark #15478: estimated potential speedup:
1.820
         remark #15487: type converts: 23
```

- Observe how verbose is the report looking at the file: `GSimulation.optrpt`
  - `2974 number of lines`
- You can filter the report by running: `make REPORT=yes FILTER=yes`

  ```
  -qopt-report-phase=vec
  -qopt-report-
  filter="GSimulation.cpp,125-175"
  ```

- Look at the `Makefile` for more details on the compiler options used
- Open the new report file and read the results

Exercise 1 – nbody-demo/ver0

- How can we improve the performance using only the compiler?
  - Use different compiler options and try to target the underlying architecture:
  - -xCORE-AVX2 or -xHost
  - -O3
  - -ipo
  - -prof-gen and -prof-use
  - -parallel
- Explain why some options don't bring additional speed-up. Try some more tests here:

  git clone https://github.com/ivorobts/compiler-optimization.git

Legal Disclaimers and Optimization Notices

Legal Disclaimer & Optimization Notice

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

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

Optimization Notice

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

Notice revision #20110804

Software