The Intel® Compiler(1): Overview, Vectorization

Martyn Corden
Developer Products Division
Software & Services Group
Intel Corporation

June 2010
Agenda (day 1)

- Intel® Compilers for Intel64 Linux
  - Getting Started
  - Principal Optimization Features
  - **Vectorization**
  - 64 bit issues
  - Language specific features
  - **Parallelism**
    - Floating-point issues
      - Reproducibility, precision, exceptions
    - Debugging
    - Upcoming compiler features

- Performance Libraries
- Intel® VTune™ Performance Analyzer
Agenda

• Overview
  – Getting Started

• Principal Optimization Features
  – Scalar
  – Loops
  – Advanced (IPO, PGO)

• Vectorization
  – SSE
  – Optimization reports
  – Helping the compiler
  – AVX
**Developer Tools for 32/64 bit on Windows*, Linux*, and Mac OS* X**

Intel® Software Development Products support multiple platforms

<table>
<thead>
<tr>
<th>Compilers</th>
<th>C++</th>
<th>Fortran</th>
<th>Windows*/<em>Linux</em></th>
<th>Windows</th>
<th>Linux</th>
<th>Mac OS*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance Analyzers</td>
<td>VTune(tm) Performance Analyzer</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>Performance Libraries</td>
<td>Integrated Performance Primitives</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td></td>
<td>Math Kernel Library</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Threading Library</td>
<td>Threading Building Blocks</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Threading Analysis Tools</td>
<td>Thread Checker</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Thread Profiler</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>Cluster Tools</td>
<td>MPI Library</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Trace Analyzer and Collector</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Math Kernel Library</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cluster Toolkit</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>All-in-one parallelism toolkit for C/C++</td>
<td>Intel Parallel Studio</td>
<td></td>
<td></td>
<td></td>
<td>✔️</td>
<td></td>
</tr>
</tbody>
</table>
Some Generic Features

- Supports standards (ANSI C, ISO C++, ANSI C99, Fortran95, most Fortran2003, some C++0x)
- C/C++ compatible with leading open-source tools (especially gcc)
  - Makes use of GNU libc, libstdc++
  - Intel optimized libimf pre-empts GNU libm
  - Intel Fortran is not binary compatible with g77 or gfortran

- Supports all instruction set extensions via vectorization
- OpenMP* 3.0 support and Automatic Parallelization
- Sophisticated optimizations
  - Multi-file inter-procedural optimization
  - Profile-guided optimization
- Detailed optimization reports
- Intel Parallel Debugger (idb)
Getting Started

• Set the environment, e.g. from bin directory:
  – source iccvars.sh ia32 (or iccvars.csh)
  – source ifortvars.sh intel64

• Drivers:
  – icpc for C++
  – icc for C
    but treats .c as C and .cpp as C++ by default
  – ifort
    treats .f90 as free format, .f as fixed
    .F90, .F, invoke preprocessor by default

• Linking:
  – Simplest: use compiler driver
    – links Intel-specific libraries, such as optimized math functions
  – else use xild, which invokes ld
## A few General Switches

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Linux*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable optimization</td>
<td>-O0</td>
</tr>
<tr>
<td>Optimize (limit code size increase), no vectorization</td>
<td>-O1</td>
</tr>
<tr>
<td>Optimize for speed (default), includes vectorization</td>
<td>-O2</td>
</tr>
<tr>
<td>High-level optimizer (aggressive loop optimizations)</td>
<td>-O3</td>
</tr>
<tr>
<td>Processor-specific optimizations (-msse2 is default)</td>
<td>-x..., -m..., (many)</td>
</tr>
<tr>
<td>Create symbols for debugging</td>
<td>-g, -debug</td>
</tr>
<tr>
<td>Generate assembly files</td>
<td>-S</td>
</tr>
<tr>
<td>Optimization report generation</td>
<td>-opt-report</td>
</tr>
<tr>
<td>OpenMP support</td>
<td>-openmp</td>
</tr>
<tr>
<td>Automatic parallelization for OpenMP* threading</td>
<td>-parallel</td>
</tr>
</tbody>
</table>
Intel® Compiler Architecture

C++ Front End

FORTRAN Front End

Profiler

Interprocedural analysis and optimizations: inlining, constant prop, whole program detect, mod/ref, points-to

Loop optimizations: data deps, prefetch, vectorizer, unroll/interchange/fusion/dist, auto-parallel/OpenMP

Global scalar optimizations: partial redundancy elim, dead store elim, strength reduction, dead code elim

Code generation: vectorization, software pipelining, global scheduling, register allocation, code generation

Disambiguation: types, array, pointer, structure, directives
InterProcedural Optimization
Extends optimizations across file boundaries

**Without IPO**

- Compile & Optimize → file1.c
- Compile & Optimize → file2.c
- Compile & Optimize → file3.c
- Compile & Optimize → file4.c

**With IPO**

- Compile & Optimize → file1.c
- Compile & Optimize → file3.c
- Compile & Optimize → file4.c
- Compile & Optimize → file2.c

---

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ip</td>
<td>Only between modules of one source file</td>
</tr>
<tr>
<td>-ipo</td>
<td>Modules of multiple files/whole application</td>
</tr>
</tbody>
</table>
IPO – A Multi-pass Optimization
A Two-Step Process

Pass 1

Compiling
icc -c -ipo main.c func1.c func2.c

virtual .o

Pass 2

Linking
icc -ipo main.o func1.o func2.o

executable

contains intermediate language
What you should know about IPO

• -O2 and -O3 activate most of file-local IPO (-ip)
  – Only a very few, time-consuming IP-optimizations are omitted
  – Switch -ip-no-inlining disables in-lining

• IPO extends compilation time and memory usage
  – See compiler manual when running into limitations

• In-lining of functions is most important feature of IPO but there is much more:
  – Inter-procedural constant propagation
  – MOD/REF analysis (for dependence analysis)
  – Routine attribute propagation
  – Dead code elimination
  – Induction variable recognition
  – …many, many more

• -opt-report-phase ipo_inl to get inlining report
IPO in parallel reduce time for IPO builds

• `-ipo<m>` asks the compiler to generate `<m>` separate object files as input to final IPO-linkage phase
  - By default (`-ipo`), compiler generates multiple object files automatically when the application is very large
  - `-ipo-separate` asks for one object for each source file

• `-ipo-jobs<n>` specifies the number `<n>` of commands (jobs) to be executed simultaneously during the link phase of Interprocedural Optimization (IPO).
  - Must have `n ≤ m`
Intel® Compiler: Some Default Scalar Optimizations

**icc** (or icl or ifort) \(-O2\)

- inlining intrinsic functions, some user functions
- constant propagation
- copy propagation
- dead-code elimination
- global register allocation
- global instruction scheduling
- control speculation
- partial redundancy elimination
- strength reduction
- induction variable simplification
- exception handling optimizations
- tail recursions
- peephole optimizations
- structure assignment lowering optimizations
- dead store elimination.

---

Put repeated sub-expressions first:

\[
\begin{align*}
  x &= a + b + d \\
  y &= a + b + e \\
  \text{not} \\
  x &= d + a + b \\
  y &= e + a + b
\end{align*}
\]
Intel® Compilers: Loop Optimizations

**icc** (or icl or ifort) **–O3**

- Loop optimizations:
  - Automatic vectorization‡ (use of packed SIMD instructions)
  - Loop interchange‡ (for more efficient memory access)
  - Loop unrolling‡ (more instruction level parallelism)
  - Prefetching (for patterns not recognized by h/w prefetcher)
  - Cache blocking (for more reuse of data in cache)
  - Loop peeling (allow for misalignment)
  - Loop versioning ‡ (for loop count; data alignment; runtime dependency tests)
  - Memcpy recognition ‡ (call Intel’s fast memcpy, memset)
  - Loop splitting ‡ (facilitate vectorization)
  - Loop fusion (more efficient vectorization)
  - Scalar replacement‡ (reduce array accesses by scalar temps)
  - Loop rerolling (enable vectorization)
  - Loop reversal (handle dependencies)
  - etc.

‡ all or partly enabled at –O2
**SIMD: Single Instruction, Multiple Data**

- **Scalar processing**
  - traditional mode
  - one instruction produces one result

- **SIMD processing**
  - with SSE instructions
  - one instruction can produce multiple results

\[
\begin{align*}
X \quad &\quad X \\
+ \quad &\quad + \\
Y \quad &\quad Y \\
= \quad &\quad = \\
X + Y \quad &\quad X + Y \\
\end{align*}
\]

\[
\begin{align*}
x_3 \quad &\quad x_3 + y_3 \\
x_2 \quad &\quad x_2 + y_2 \\
x_1 \quad &\quad x_1 + y_1 \\
x_0 \quad &\quad x_0 + y_0 \\
\end{align*}
\]
SSE Data Types

SSE
- 4x floats

SSE-2
- 2x doubles
- 16x bytes
- 8x 16-bit shorts
- 4x 32-bit integers
- 2x 64-bit integers
- 1x 128-bit(!) integer

*Other brands and names are the property of their respective owners.*
**Evolution of SSE**

<table>
<thead>
<tr>
<th>Year</th>
<th>Instructions</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>1999</td>
<td>70 instr</td>
<td>Single-Precision Vectors, Streaming operations</td>
</tr>
<tr>
<td>2000</td>
<td>144 instr</td>
<td>Double-precision Vectors, 8/16/32 64/128-bit vector integer</td>
</tr>
<tr>
<td>2004</td>
<td>13 instr</td>
<td>Complex Data</td>
</tr>
<tr>
<td>2006</td>
<td>32 instr</td>
<td>Video Graphics building blocks, Advanced vector instr</td>
</tr>
<tr>
<td>2007</td>
<td>47 instr</td>
<td>String/XML processing POP-Count CRC</td>
</tr>
<tr>
<td>2008</td>
<td>8 instr</td>
<td></td>
</tr>
</tbody>
</table>

**continued by**

- **Intel AES-NI (Cryptography, Westmere‡ processor)**
- **Intel® AVX (256 bit SIMD, Sandy Bridge‡ processor)**

*processor codenames*
Automatic Vectorization by Compiler
Translates Loops into SIMD Parallelism (SSE)
loop is stripmined (unrolled), strip length of 4 for floats

```c
for (i=0; i<=MAX; i++)
    c[i] = a[i] + b[i];
```
Did my loop vectorize?

icc -vec-report1 -c Multiply.c
Multiply.c(31): (col. 3) remark: LOOP WAS VECTORIZED.

- Vectorization and reports are enabled only at `-O2` and above
- You can also see SIMD registers and instructions in the assembly code:

  icc -S Multiply.c; grep xmm Multiply.s

  ...
  movaps (%rdx,%r14,8), %xmm2
  mulpd (%r8,%r14,8), %xmm2
  addpd %xmm2, %xmm0
  ...

  "pd" = packed double, ss = "scalar single", etc. (packed is good)
  movaps ⇔ movapd is an aligned 16 byte load/store (good)
  - movupd is unaligned; movhpd, movsd load/store only 8 bytes
Requirements for Vectorization

• Must be an inner loop.
• Straight-line code (masked assignments OK)
• Avoid:
  – Function/subroutine calls (unless inlined)
  – Non-mathematical operators (sqrt, sin, exp,... OK)
  – Data-dependent loop exit conditions
    – Iteration count must be known at entry to loop
  – Loop carried data dependencies
    – Reduction loops are OK
  – Non-contiguous data (indirect addressing; non-unit stride)
    – inefficient
• Directives/pragmas can help:
  – !DIR$ IVDEP ...... ignore potential dependencies
  – !DIR$ VECTOR ALWAYS ignore efficiency heuristics
  – ALIGNED assume data aligned
  – Compiler can generate runtime alignment and dependency tests for simple loops (but less efficient)
• See http://software.intel.com/en-us/articles/requirements-for-vectorizable-loops/
### Vectorizable math functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos</td>
<td>ceil</td>
</tr>
<tr>
<td>acosh</td>
<td>cos</td>
</tr>
<tr>
<td>asin</td>
<td>cosh</td>
</tr>
<tr>
<td>asinh</td>
<td>erf</td>
</tr>
<tr>
<td>atan</td>
<td>erfc</td>
</tr>
<tr>
<td>atan2</td>
<td>erfinv</td>
</tr>
<tr>
<td>atanh</td>
<td>exp</td>
</tr>
<tr>
<td>cbrt</td>
<td>exp2</td>
</tr>
<tr>
<td>fabs</td>
<td>floor</td>
</tr>
<tr>
<td>fmax</td>
<td>fmin</td>
</tr>
<tr>
<td>fmin</td>
<td>sqrt</td>
</tr>
<tr>
<td>log</td>
<td>log10</td>
</tr>
<tr>
<td>log2</td>
<td>trunc</td>
</tr>
<tr>
<td>round</td>
<td>sin</td>
</tr>
<tr>
<td>sinh</td>
<td>sqrt</td>
</tr>
<tr>
<td>sqrt</td>
<td>tan</td>
</tr>
<tr>
<td>tanh</td>
<td>10</td>
</tr>
</tbody>
</table>

Also float versions, such as sinf()

Uses short vector math library, libsvml
Why Didn’t My Loop Vectorize?

Reporting Switch:  `-vec-report<n>`

Set diagnostic level dumped to stdout

n=0: *(Default)* No diagnostic information
n=1: 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

`icc -vec-report2 -c Multiply.c` gives additionally
`Multiply.c(22): (col. 2) remark: loop was not vectorized: not inner loop`
`Multiply.c(31): (col. 3) remark: loop skipped: multiversioned.`

**Note:**
- `-opt-report-phase hpo` is similar to `-vec-report2 -par-report2`
- `-opt-report-phase hlo` reports on other loop optimizations
Vectorization Report

“Loop was not vectorized” because:

- “Low trip count”
- “Not Inner Loop”
- “Existence of vector dependence”
- "vectorization possible but seems inefficient"
- “Condition may protect exception”
- “data type unsupported..”
- Statement cannot be vectorized
- “Subscript too complex”
- ‘Unsupported Loop Structure’
- “Top test could not be found”
- “Operator unsuited for vectorization”
- ... ( some more )
Will it vectorize?

for (j=1; j<MAX; j++) a[j]=a[j-n]+b[j];

for (int i=0; i<SIZE; i+=2) b[i] += a[i] * x[i];

for (int j=0; j<SIZE; j++) {
    for (int i=0; i<SIZE; i++) b[i] += a[i][j] * x[j];
}

for (int i=0; i<SIZE; i+=2) b[i] += a[i] * x[index[i]];

for (j=1; j<MAX; j++) sum = sum + a[j]*b[j]

for (int i=0; i<length; i++)
    if ( s >= 0 ) x2[i] = (-b[i]+sqrt(s))/(2.*a[i]);
Problems with Pointers

- Hard for compiler to know whether arrays or pointers might be aliased (point to the same memory location)
  - Aliases may hide dependencies that make vectorization unsafe
- In simple cases, compiler may generate vectorized and unvectorized loop versions, and test for aliasing at runtime
- Otherwise, compiler may need help:
  - `-fargument-noalias` (`/Qalias-args-`) & similar switches
  - “restrict” keyword with `-restrict` or `-std=c99` (`/Qrestrict` or `/Qstd=c99`)
  - or by inlining
- Less of a problem for Fortran due to stricter language rules

```c
void saxpy (float *x, float *y, float*restrict z, float *a, int n) {
    int i;
    #pragma ivdep
    for (i=0; i<n; i++) z[i] = *a*x[i] + y[i];
}
```
Guidelines for Writing Vectorizable Code

• Prefer simple “for” loops
• Write straight line code. Avoid:
  – most function calls
  – branches that can’t be treated as masked assignments.

• Avoid dependencies between loop iterations
  – Or at least, avoid read-after-write dependencies

• Prefer array notation to the use of pointers
  – Without help, the compiler often cannot tell whether it is safe to vectorize code containing pointers.
  – Try to use the loop index directly in array subscripts, instead of incrementing a separate counter for use as an array address.

• Use efficient memory accesses
  – Favor inner loops with unit stride
  – Minimize indirect addressing
  – Align your data to 16 byte boundaries (for SSE instructions)
How to Align your Data

__declspec(align(base, [offset]))
Instructs the compiler to create the variable so that it is aligned on an “base”-byte boundary, with an “offset” (Default=0) in bytes from that boundary.

void* _mm_malloc (int size, int n)
Instructs the compiler to create a pointer to memory such that the pointer is aligned on an n-byte boundary.

and tell the compiler...

#pragma vector aligned | unaligned
Use aligned or unaligned loads and stores for vector accesses.

__assume_aligned(a,n)
Instructs the compiler to assume that array a is aligned on an n-byte boundary.
Intel® Compilers:
some useful loop optimization pragmas/directives

- IVDEP  
  ignore vector dependency
- LOOP COUNT  
  advise typical iteration count(s)
- UNROLL  
  suggest loop unroll factor
- DISTRIBUTED POINT  
  advise where to split loop
- PREFETCH  
  hint to prefetch data
- VECTOR  
  vectorization hints
  - Aligned  
    assume data is aligned
  - Always  
    override cost model
  - Nontemporal  
    advise use of streaming stores
- NOVECTOR  
  do not vectorize
- PARALLEL  
  override efficiency heuristics for auto-parallelization

Use where needed to help the compiler, guided by optimization reports
Switches that may help vectorization

- `-O3` performs other loop transformations first
- `-ipo` may inline, or get dependency, loop count or alignment information from calling functions
- `-xsse4.2` use all available instructions
- `-fno-alias` assume that pointers are not aliased (dangerous!)
- `-fargument-noalias`
- `-fansi-alias`
HLO Optimization Report Example

ifort -O3 -opt_report_phase hlo -opt-report-phase hpo matmul.f90

HPO VECTORIZER REPORT (matmul_)
...
matmul.f90(9:1-9:1):VEC:matmul_: PERMUTED LOOP WAS VECTORIZED
...
High Level Optimizer Report (matmul_)
#of Array Refs Scalar Replaced in matmul_ at line 9=36
...

<matmul.f90;9:9;hlo_linear_trans;matmul_;0>
LOOP INTERCHANGE in loops at line: 9 8 7
Loopnest permutation ( 1 2 3 ) --> ( 2 3 1 )
...
<matmul.f90;9:9;hlo_unroll;matmul_;0>
Loop at line 9 blocked by 128
...
Loop at line 7 blocked by 128
Loop at line 8 blocked by 128
Loop at line 8 unrolled and jammed by 4
Loop at line 7 unrolled and jammed by 4

subroutine matmul(a,b,c,n)
real(8) a(n,n),b(n,n),c(n,n)
do j=1,n
  do i=1,n
    do k=1,n
      c(j,i)=c(j,i)+a(k,i)*b(j,k)
    enddo
  enddo
enddo
enddo
end

Optimization Report Options

- **opt_report**
  - generate an optimization report to stderr (or file)
- **opt_report_file <file>**
  - specify the filename for the generated report
- **opt_report_phase <phase_name>**
  - specify the phase that reports are generated against
  - hlo, hpo, ipo, ipo_inl, pgo, etc
- **opt_report_routine <name>**
  - reports on routines containing the given name
- **opt_report_help**
  - display the optimization phases available for reporting
- **vec-report<level>**
  - Generate vectorization report
### Key Intel® Advanced Vector Extensions (Intel® AVX) Features

<table>
<thead>
<tr>
<th><strong>KEY FEATURES</strong></th>
<th><strong>BENEFITS</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Wider Vectors</td>
<td>• Up to 2x peak FLOPs (floating point operations per second) output with good power efficiency</td>
</tr>
<tr>
<td>– Increased from 128 bit to 256 bit</td>
<td></td>
</tr>
<tr>
<td>• Enhanced Data Rearrangement</td>
<td>• Organize, access and pull only necessary data more quickly and efficiently</td>
</tr>
<tr>
<td>– Use the new 256 bit primitives to broadcast, mask loads and permute data</td>
<td></td>
</tr>
<tr>
<td>• Three and four Operands, Non Destructive Syntax</td>
<td>• Fewer register copies, better register use for both vector and scalar code</td>
</tr>
<tr>
<td>– Designed for efficiency and future extensibility</td>
<td></td>
</tr>
<tr>
<td>• Flexible unaligned memory access support</td>
<td>• More opportunities to fuse load and compute operations</td>
</tr>
</tbody>
</table>

Intel® AVX is a general purpose architecture, expected to supplant SSE in all applications used today
Intel® Advanced Vector Extensions (Intel® AVX) 2X Vector Width

A 256-bit vector extension to SSE

• Intel® AVX extends all 16 XMM registers to 256 bits

- Intel AVX works on either
  - The whole 256-bits – for FP instructions
  - The lower 128-bits (like existing SSE instructions)
    - A drop-in replacement for all existing scalar/128-bit SSE instructions
    - The upper part of the register is zeroed out

• Intel AVX targets a high-performance first implementation
  - 256-bit Multiply, Add and Shuffle engines (2X today)
  - 2nd load port
Compiling for Intel® AVX

• Compile with `-xavx`
  - Vectorization works just as for SSE
  - Main speedups are for floating point
    - No integer 256 bit instructions in first generation
    - Up to ~1.8x performance for Linpack
  - More loops can be vectorized than with SSE
    - Individually masked data elements
    - More powerful data rearrangement instructions

• `-xavx` will give both SSE and AVX code paths
  - use `-x` switches to modify the default SSE code path
    - Eg `-xavx -xsse4.2` to target Nehalem and AVX

• Math libraries will target AVX automatically at runtime
## Processor-specific Compiler Switches for Intel 64

<table>
<thead>
<tr>
<th>Intel only</th>
<th>Intel and non-Intel</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-xsse2</code></td>
<td><code>-msse2</code> (default!)</td>
</tr>
<tr>
<td><code>-xsse3</code></td>
<td><code>-msse3</code></td>
</tr>
<tr>
<td><code>-xssse3</code></td>
<td><code>-mssse3</code></td>
</tr>
<tr>
<td><code>-xsse4.1</code></td>
<td><code>-msse4.1</code></td>
</tr>
<tr>
<td><code>-xsse4.2</code></td>
<td></td>
</tr>
<tr>
<td><code>-xavx</code></td>
<td></td>
</tr>
<tr>
<td><code>-xHost</code></td>
<td><code>-xHost</code></td>
</tr>
<tr>
<td>Intel cpuid check</td>
<td>No cpu id check</td>
</tr>
<tr>
<td>Runtime message if run on unsupported processor</td>
<td>Illegal instruction error if run on unsupported processor</td>
</tr>
</tbody>
</table>
Processor Dispatch (fat binaries)

- Compiler can generate multiple code paths
  - optimized for different processors
  - only when likely to help performance
  - One default code path, one or more optimized paths
  - Optimized paths are for Intel processors only
  - Default code path can be modified using switches from preceding slide

- Examples:
  - /Qaxsse4.2 (-axsse4.2)
    - default path optimized for SSE2 (Intel or non-Intel)
    - Second path optimized for SSE4.2 (Nehalem & Westmere)
  - /QaxAVX,SSE4.2 /arch:SSE3 (-axavx,sse4.2 -msse3)
    - Default path optimized for SSE3 (Intel or non-Intel)
    - Second path optimized for SSE4.2 (Nehalem, Westmere)
    - Third path optimized for AVX (Sandy Bridge)
Compiler Performance Comparison on Intel processor-based systems

Estimated Relative Compiler SPEC* CPU2006 Benchmark Performance (Higher is Better)
Intel® Xeon® Processor on Linux* 64-bit mode

<table>
<thead>
<tr>
<th>Compiler Configuration</th>
<th>Estimated SPECint@base2006</th>
<th>Estimated SPECfp@base2006</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® C++ &amp; Fortran Compiler 11.1 for Linux*</td>
<td>1.45</td>
<td>1.94</td>
</tr>
<tr>
<td>Intel® C++ &amp; Fortran Compiler 11.0 for Linux*</td>
<td>1.35</td>
<td>1.80</td>
</tr>
<tr>
<td>GCC 4.5</td>
<td>1.00</td>
<td>1.62</td>
</tr>
<tr>
<td>PGI C++ Compiler 9.0, PGI Fortran Compiler 9.0</td>
<td>0.90</td>
<td>1.27</td>
</tr>
<tr>
<td>Pathscale C++ &amp; Fortran 3.2</td>
<td>0.96</td>
<td>0.93</td>
</tr>
<tr>
<td>x86_Open64 4.2.2.2</td>
<td>0.70</td>
<td>0.83</td>
</tr>
</tbody>
</table>

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, refer to www.intel.com/performance/resources/benchmark_limitations.htm. *Other brands and names are the property of their respective owners.

Estimates Assume:
- SPEC, SPECrate, SPECfp, SPECint, SPEComp are trademarks of Standard Performance Evaluation Corporation (http://www.spec.org)
- Source: All data is based on estimates from Intel Corporation as of October 1, 2008. Performance normalized to the lowest performing compiler
- Hardware & OS: Intel(R) Xeon(R) CPU X5570 @ 2.93GHz, 2x2.93GHz, 24GB, 8192KB,

Operating System: Red Hat Enterprise Linux Server release 5.3 (Tikanga), kernel 2.6.18-128.el5 #1, glibc-2.5-34

Copyright © 2010, Intel Corporation. All rights reserved.
*Other brands and names are the property of their respective owners.
Questions?
Further Information


  http://software.intel.com/en-us/articles/requirements-for-vectorizable-loops/

- The Intel® C++ and Fortran Compiler User and Reference Guides,

- And the User Forums and Knowledge Base,
  http://software.intel.com/en-us/articles/tools
Summary

• Comprehensive set of tools for multi-core and cluster parallelism from Intel for x86 architecture
  – Best performance on Intel architecture, and competitive performance on AMD systems
  – Intel tools can be used to standardize x86 development C++/Fortran development

• Our focus is on
  – Best Performance
  – Comprehensive coverage of parallelism
  – Ease of use
  – Compatibility and software investment protection

Visit http://intel.com/software/products
Legal Disclaimer

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 PROPERTIES RIGHT.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.

Intel, the Intel logo, Itanium, Pentium, Intel Xeon, Intel Core, Intel Centrino and VTune are trademarks or registered trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2010. Intel Corporation.

http://intel.com/software/products
Sample: Align The Data

declspec(align(16)) float B[MAX];

void funca(float * B)
{
    __assume_aligned(B,16);
    A=_mm_malloc(sizeof(float)*MAX,16);
    for (I=0;I<MAX;I++)
        A[I]+=B[I];
    _mm_free(A);
}
Sample: Align The Data – 2nd Way

declspec(align(16)) float B[MAX];

void funca(float * B)
{
    A=_mm_malloc(sizeof(float)*MAX,16);

    #pragma vector aligned

    for (I=0;I<MAX;I++)
        A[I]+=B[I];

    _mm_free(A);
}

Sample: Loop was not vectorized because "Non-unit stride used"

```c
for (I=0; I<=MAX; I++)
    for (J=0; J<=MAX; J++) {
        c[I][J] += 1;  // Unit Stride
        c[J][I] += 1;  // Non-Unit
        A[J*J] += 1;   // Non-unit
        A[B[J]] += 1;  // Non-Unit
        if (A[MAX-J] == 1) last1 = J; // Non-Unit
    }
```

End Result: Loading Vector may take more cycles than executing operation sequentially
Sample: Loop was not vectorized because “Unsupported Loop Structure”

```c
struct _xx {
    int data;
    int bound;
};

doit1(int *a, struct _xx *x)
{
    for (int i=0; i<x->bound; i++)
        a[i] = 0;
}
```

• Unsupported loop structure means that the loop is not countable, or at least the compiler can’t construct a runtime expression for the tripcount
Sample: Loop was not vectorized because “Low Trip Count”

- Setting up loops for SSE instructions incurs an overhead
- Loops with a small # of iterations – probably shouldn’t be vectorized
- Tell the compiler how large the loop is

```c
#pragma loop count(5)
for (i = 0; i < MAX1; ++i) {
    a1[i] = b1[i] * c1[i] + d1[i];
}
```

It is possible too, to inform the compiler about multiple, expected number of iterations which e.g. can help to optimize loop unrolling and/or to create best set of multiple loop versions:

```c
#pragma loop count (1, 5, 60)
for (i = 0; i < MAX2; ++i) {
    a2[i] = b2[i] * c2[i] + d2[i];
}
```
Loop was not vectorized because
"Condition may protect exception"

- Conditions get turned into bit masks that execute both sides
- If the compiler calculates that one side could generate an exception – it won’t vectorize the loop

```cpp
int A[MAX-50];
for (I=0; I<MAX; i++)
{
    if (I<MAX-50) A[i]=0;
    ...
}
```
Loop was not vectorized because
“Vectorization possible but seems inefficient”

- Could be the loop is so large, it needs too many registers
  - Possible Solution: Loop Fission (frequently also called Loop Distribution)
    - Split the loop by hand or
    - Use pragma distribute point to give a hint to the compiler where to “split” the loop. Sample:
      ```c
      for (I=0;I<N;I++)
      {
        A[I]=B[i]*max(C[i],D[i])+F[i]-E[I];
        #pragma distribute point
        G[I]=H[i]*sin(I[I])+J[I]/K[I];
      }
      ```
- Other problems can create this warning
A Solution for some Vector Report Problems

• Control vectorization of the subsequent loop
  – #pragma novector – Specifies that loop should never be vectorized, even if it is legal to do so
  – #pragma vector always – Overrides heuristic decision about profitability and exception detection

```c
#pragma vector always
for (i = 0; i < MAX; ++i)
    a[i] = b[i] * c[i] + d[i];
```