Producing Optimized Code
Using the Intel Compiler

Stephen Blair-Chappell
Intel Compiler Labs
Often we are happy with out-of-the-box experience.

When was the last time you looked at some documentation?
The Sample Application

- Initialises two matrices with a numeric sequence
- Does a Matrix Multiplication
The Seven Steps

**Step 1**
Build with optimization disabled

**Step 2**
Use General Optimizations

**Step 3**
Use Processor-Specific Options

**Step 4**
Add Inter-procedural

**Step 5**
Use Profile Guided Optimization

**Step 6**
Tune automatic vectorization

**Step 7**
Implement Parallelism or use Automatic Parallelism

---

**Example options**

*Windows*  
/Od (-O0)  
/01,02,03 (-O1, -O2, -O3)  
/QxSSE4.2 (-xsse4.2)  
/QxHOST (-xhost)  
/Qipo (-ipo)  
/Qprof-gen (-prof-gen)  
/Qprof-use (-prof-use)  
/Qguide (-guide)

*Linux*

Use Intel Family of Parallel Models  
/Qparallel (-parallel)
Step 1
Building with Optimisation Disabled
STEP 1 – Building with Optimisation Disabled

• Prove that the application works!
• Good option if debugging an application
• Occasionally optimisation can break an application
• NOTE: Some programmers disable
  – Turbo Boost!
  – Hyper-threading
Specification of Machines Used

• **Core 2 Laptop.** Lenovo T66. Intel Core 2 Duo CPU. T7300 @ 2.00 GHz. 2GB ram.

• **Sandy Bridge Laptop.** Lenovo W520. Intel Core i7-2820QM @ 2.30 GHz. 8GB ram.

• **Xeon Workstation.** OEM. Intel Xeon CPU, X5680 @ 3.33 GHz (2 processors).
## Results of /Od and /O2 build

<table>
<thead>
<tr>
<th>Build Machine</th>
<th>Od</th>
<th>O2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 2 Laptop</td>
<td>3.041</td>
<td>0.474</td>
</tr>
<tr>
<td>Sandy Bridge</td>
<td>2.164</td>
<td>0.293</td>
</tr>
<tr>
<td>Sandy Bridge (TurboMode)</td>
<td>1.588</td>
<td>0.211</td>
</tr>
<tr>
<td>Xeon</td>
<td>1.325</td>
<td>0.238</td>
</tr>
</tbody>
</table>
Step 2

Using the General Optimisations
/O1 (-O1) Optimize for speed and size

• This option is very similar to /O2 except that it omits optimizations that tend to increase object code size, such as the in-lining of functions. Generally useful where memory paging due to large code size is a problem, such as server and database applications.

• Auto-vectorization is not turned on, even if it is invoked individually by its fine grained switch /Qvec. However, at /O1 the vectorization associated with array notations is enabled.
/O2 (-O2) Optimize for maximum speed.

- This option will create faster code in most cases.
- Optimizations include
  - scalar optimizations
  - inlining and some other
  - Inter-procedural optimizations between functions/subroutines in the same source file
  - vectorization
  - limited versions of a few other loop optimizations such as loop versioning and unrolling that facilitate vectorization.
/O3 (-O3) OPTIMIZES FOR FURTHER SPEED INCREASES.

• This includes all the /O2 optimizations, together with other High Level Optimizations.

• These high level optimizations include more aggressive strategies such as:
  – scalar replacement,
  – data pre-fetching,
  – loop optimization,
  – among others.
Loop Transformations

- Frequently (optimal) vectorization is possible only after adapting the loops before
- The compiler component responsible for these loop transformations is phase HLO – High Level Optimization
  - While HLO is active for optimization level O2 and O3, only O3 activates the full set of transformations and applies the transformations more aggressively
- Intel compilers provide detailed report on HLO activity:
  \{L&M\}: –opt-report –opt-report-phasehlo
  \{W\}: /Qopt-report /Qopt-report-phasehlo

  ... LOOP INTERCHANGE in loops at line: 7 8 9
  Loopnest permutation (1 2 3) --> (2 3 1)

  ... Loop at line 7 unrolled and jammed by 4
  Loop at line 8 unrolled and jammed by 4
  ...
Sample for Loop Transformations

14: for (i=0; i<100; i++)
15: {
16:   a[i] = 0;
17:   for (j=0; j<100; j++)
18:      a[i] += b[j][i];
19: }

Report from vectorizer:
file.c(16) : (col. 8) remark: PARTIAL LOOP WAS VECTORIZED.
file.c(14) : (col. 8) remark: loop was not vectorized: not inner loop.
file.c(18) : (col. 10) remark: PERMUTED LOOP WAS VECTORIZED.

Transformations done by compiler:
1) i-loop is distributed into 2 loops: a single loop and a nested loop
2) Nested loop is interchanged to exploit spatial locality on b[j][i]
3) A single loop is vectorized. (1st VECTORIZED message)
4) Inner loop of the interchanged nested loop is vectorized (2nd VECTORIZED message)
Some HLO (Loop) Transformations
Enabled for –O3

- Loop interchange (for more efficient memory access)
- Loop unrolling (more instruction level parallelism)
- Cache blocking (for more reuse of data in cache)
- Loop peeling (allow for misalignment)
- Loop versioning (for loop count, data alignment, ...)
- Memcpy recognition (call Intel’s fast memcpy, memset)
- Loop splitting (facilitate vectorization)
- Loop fusion (more efficient vectorization)
- Scalar expansion (remove dependency)
- Loop rerolling (enable vectorization)
- Loop reversal (handle dependencies)

*Blue color: Applied too for optimization level O2
The Results of using the General Options

- Core 2 Laptop
- Xeon Server
- SNB No TurboBoost
- SNB with TurboBoost

Time (Seconds) vs. General Options: O1, O2, O3, O4

- Core 2 Laptop shows a decrease in time from O1 to O4.
- Xeon Server has a consistent time across O1 to O4.
- SNB No TurboBoost shows a decrease from O1 to O3, then plateaus.
- SNB with TurboBoost has a decrease from O1 to O3, then a sharp increase at O4.
**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
Compiler Reports – Optimization Report

Compiler switch:
-`opt-report-phase[phase]` (Linux)
  
  `phase` can be:

  - `ipo` – Interprocedural Optimization
  - `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:

`/Qopt-report [0|1|2|3]` (Windows)

`-opt-report [0|1|2|3]` (Linux, MacOS X)
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
...

icc -vec-report2 (icl /Qvec-report2) for just the vectorization report
There are lots of Phases!

**icl /Qopt-report-help**
Intel® C++ Intel® 64 Compiler XE for applications running on Intel® 64, Version 12.0.3.175 Build 20110309
Copyright (C) 1985-2011 Intel Corporation. All rights reserved.

**Intel® Compiler Optimization Report Phases**

usage: -Qopt_report_phase <phase>

ipo, ipo_inl, ipo_cp, ipo_align, ipo_modref, ipo_lpt, ipo_subst, ipo_ratt, ipo_vaddr, ipo_pdce, ipo_dp, ipo_gprel, ipo_pmerge, ipo_dstat, ipo_fps, ipo_ppi, ipo_unref, ipo_wp, ipo_dl, ipo_pspliit, ilo, ilo_arg_prefetching, ilo_lowering, ilo_strength_reduction, ilo_reassociation, ilo_copy_propagation, ilo_convert_insertion, ilo_convert_removal, ilo_tail_recursion, hlo, hlo_fusion, hlo_distribution, hlo_scalar_replacement, hlo_unroll, hlo_prefetch, hlo_loadpair, hlo_linear_trans, hlo_opt_pred, hlo_data_trans, hlo_string_shift_replace, hlo_ftae, hlo_reroll, hlo_array_contraction, hlo_scalar_expansion, hlo_gen_matmul, hlo_loop_collapsing, hpo, hpo_analysis, hpo_openmp, hpo_threadization, hpo_vectorization, pgo, tcollect, offload, all
Step 3

Using Processor Specific Options
SIMD Instruction Enhancements

- **1999**: SSE
  - 70 instr
  - Single-Precision Vectors
  - Streaming operations

- **2000**: SSE2
  - 144 instr
  - Double-precision Vectors
  - 8/16/32
  - 64/128-bit vector integer

- **2004**: SSE3
  - 13 instr
  - Complex Data

- **2006**: SSSE3
  - 32 instr
  - Decode

- **2007**: SSE4.1
  - 47 instr
  - Video
  - Graphics building blocks
  - Advanced vector instr

- **2008**: SSE4.2
  - 8 instr
  - String/XML processing
  - POP-Count CRC

- **2009**: AES-NI
  - 7 instr
  - Encryption and Decryption
  - Key Generation

- **2010\11**: AVX
  - ~100 new instr.
  - ~300 legacy sse instr updated
  - 256-bit vector
  - 3 and 4-operand instructions
**addSS**  Scalar Single-FP Add

Single precision FP data

Scalar execution mode

**addps**  Packed Single-FP Add

Single precision FP data

Packed execution mode

\[
\begin{array}{cccc}
  x_4 & x_3 & x_2 & x_1 \\
  y_4 & y_3 & y_2 & y_1 \\
\end{array}
\]

\[
\begin{array}{cccc}
  x_4 + x_3 + x_2 + x_1 & \\
  y_4 + y_3 + y_2 + y_1 & \\
\end{array}
\]
Auto-Vectorization

Transforming sequential code to exploit the vector (SIMD, SSE) processing capabilities

```c
for (i=0;i<MAX;i++)
    c[i]=a[i]+b[i];
```
Many Ways to Introduce SSE Vectorization

- Compiler: Fully automatic vectorization
- Cilk Plus Array Notation
- User Mandated Vectorization (SIMD Directive)
- Compiler: Auto vectorization hints (#pragma ivdep, ...)
- Manual CPU Dispatch (__declspec(cpu_dispatch ...))
- SIMD intrinsic class (F32vec4 add)
- Vector intrinsic (mm_add_ps())
- Assembler code (addps)
How do I know if a loop is vectorised?

- -vec-report

```bash
> icl /Qvec-report MultArray.c
MultArray.c(92): (col. 5) remark: LOOP WAS VECTORIZED.
```
static double A[1000], B[1000], C[1000];
void add() {
    int i;
    for (i=0; i<1000; i++)
        if (A[i]>0)
            A[i] += B[i];
        else
            A[i] += C[i];
}

Examples of Code Generation

.S1.2::
movaps xmm2, A[rdx*8]
xorps xmm0, xmm0
cmpLtD xmm0, xmm2
movaps xmm1, B[rdx*8]
andps xmm1, xmm0
andnps xmm0, C[rdx*8]
orps xmm1, xmm0
addpd xmm2, xmm1
movaps A[rdx*8], xmm2
add rdx, 2
cmp rdx, 1000
jl .S1.2

.S1.2::
vmovaps ymm3, A[rdx*8]
vmovaps ymm1, C[rdx*8]
vcmpeqD ymm2, ymm3, ymm0
vblendvpD ymm4, ymm1, B[rdx*8], ymm2
vaddD ymm5, ymm3, ymm4
vmovaps A[rdx*8], ymm5
add rdx, 4
cmp rdx, 1000
jl .S1.2

.S1.2::
vmovaps xmm2, A[rdx*8]
xorps xmm0, xmm0
cmpLtD xmm0, xmm2
movaps xmm1, C[rdx*8]
blendvpD xmm3, xmm1, B[rdx*8], xmm0
addpd xmm2, xmm1
movaps A[rdx*8], xmm2
add rdx, 2
cmp rdx, 1000
jl .S1.2

Software & Services Group, Developer Products Division
Copyright© 2011, Intel Corporation. All rights reserved. *Other brands and names are the property of their respective owners.
## Results of Enhancing Auto-Vectorisation

<table>
<thead>
<tr>
<th>Setting</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE2</td>
<td>0.293</td>
<td>1</td>
</tr>
<tr>
<td>AVX</td>
<td>0.270</td>
<td>1.09</td>
</tr>
</tbody>
</table>
Step 4

Using Inter Procedural Optimisation
Interprocedural Optimizations
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
  - file2.c
  - file3.c
  - file4.c

<table>
<thead>
<tr>
<th>/Qip, -ip</th>
<th>/Qipo, -ipo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only between modules of one source file</td>
<td>Modules of multiple files/whole application</td>
</tr>
</tbody>
</table>
Interprocedural Optimizations (IPO)
Usage: Two-Step Process

<table>
<thead>
<tr>
<th>Compiling</th>
<th>Linux*</th>
<th>Windows*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>icc -c -ipo main.c func1.c</code></td>
<td><code>icl -c /Qipo main.c func1.c</code></td>
</tr>
<tr>
<td></td>
<td><code>func2.c</code></td>
<td><code>func2.c</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Linking</th>
<th>Linux*</th>
<th>Windows*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>icc -ipo main.o func1.o</code></td>
<td><code>icl /Qipo main.o func1.o</code></td>
</tr>
<tr>
<td></td>
<td><code>func2.o</code></td>
<td><code>func2.obj</code></td>
</tr>
</tbody>
</table>

**Intermediate language (mock) object**

**Pass 1**

**Pass 2**

**Executable**
What you should know about IPO

- O2 and O3 activate “almost” file-local IPO (-ip)
  - Only a very few, time-consuming IP-optimizations are not done but for most codes, -ip is not adding anything
  - 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
- IPO works for libraries too
  - Not trivial topic – see documentation
# Results of IPO

<table>
<thead>
<tr>
<th>Platform</th>
<th>O2</th>
<th>IPO</th>
<th>QXHOST</th>
<th>Speedup O2 to IPO</th>
<th>Speedup O2 to QXHOST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 2 Laptop</td>
<td>0.474</td>
<td>0.272</td>
<td>0.266</td>
<td>1.74</td>
<td>1.78</td>
</tr>
<tr>
<td>SNB</td>
<td>0.293</td>
<td>0.181</td>
<td>0.171</td>
<td>1.62</td>
<td>1.71</td>
</tr>
<tr>
<td>SNB Turbo</td>
<td>0.211</td>
<td>0.132</td>
<td>0.124</td>
<td>1.60</td>
<td>1.70</td>
</tr>
<tr>
<td>Xeon workstation</td>
<td>0.239</td>
<td>0.211</td>
<td>0.209</td>
<td>1.13</td>
<td>1.14</td>
</tr>
</tbody>
</table>
Impact of IPO on Auto-Vectorisation

- IPO improves auto-vectorization results of the sample application.
- IPO brings some new ‘tricky-to-find’ auto-vectorization opportunities.
Multiple messages on one line

chapter4.c(51): (col. 11) remark: LOOP WAS VECTORIZED.

...chapter4.c(51): (col. 11) remark: loop was not vectorized: not inner loop.
chapter4.c(51): (col. 11) remark: loop was not vectorized: not inner loop.
chapter4.c(51): (col. 11) remark: loop was not vectorized: existence of vector dependence.
Modified code results in more improvements

**series.c**

double Series2(int j) {
    int k;
    double sumy = 0.0;
    for( k=j; k>0; k--)
    {
        // sumy++;
        sumy = AddY(sumy, k);
    }
    return sumy;
}

**addy.c**

double AddY( double sumy, int k )
{
    // sumy--;
    sumy = sumy + (double)k;
    return sumy;
}

This modification results in a 20% boost of performance
Step 5

Using Profile Guided Optimisation
PGO Usage: Three Step Process

Step 1
Compile + link to add instrumentation
icc -prof_gen prog.c

Instrumented executable:
foo.exe

Step 2
Execute instrumented program
prog.exe (on a typical dataset)

Dynamic profile:
12345678.dyn

profmerge

Merged .dyn files:
pgopti.dpi

Step 3
Compile + link using feedback
icc -prof_use prog.c

Optimized executable:
foo.exe
# PGO Compiler Options

<table>
<thead>
<tr>
<th>Linux</th>
<th>Windows</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-prof_gen</td>
<td>/Qprof-gen</td>
<td>Add PGO instrumentation</td>
</tr>
<tr>
<td>-prof_use</td>
<td>/Qprof-use</td>
<td>Use collected feedback from to create final optimized application</td>
</tr>
<tr>
<td>-prof_genx</td>
<td>/Qprof-genx</td>
<td>Creates extra info for use with code coverage tool</td>
</tr>
<tr>
<td>-opt-report</td>
<td></td>
<td>Create pgo report</td>
</tr>
<tr>
<td>-opt-report-phase=pgO</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Results of Using PGO

<table>
<thead>
<tr>
<th>PLATFORM</th>
<th>IPO</th>
<th>PGO</th>
<th>SPEEDUP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 2 Laptop</td>
<td>0.370</td>
<td>0.261</td>
<td>1.42</td>
</tr>
<tr>
<td>SNB</td>
<td>0.264</td>
<td>0.198</td>
<td>1.33</td>
</tr>
<tr>
<td>SNB Turbo</td>
<td>0.189</td>
<td>0.141</td>
<td>1.34</td>
</tr>
<tr>
<td>Xeon workstation</td>
<td>0.211</td>
<td>0.131</td>
<td>1.61</td>
</tr>
</tbody>
</table>
Results so far ...

Lower is better

Speedup: $0.211/0.064 = 3.2$
Step 6

Tuning Automatic Vectorization
GAP – Guided Automatic Parallelization

Key design ideas:
• Use compiler infrastructure to help developer to detect what is blocking certain optimizations – in particular vectorization, parallelization and data transformations – and to change code correspondingly
  • Extend diagnostic message for failed vectorization and parallelization by specific hints to fix problem
• Not a separate tool but additional feature of C/C++ and Fortran compiler
• Exploit multi-year experience brought into the compiler development
  • Performance tuning knowledge based on dealing with numerous applications, benchmarks and compute kernels

It is not:
• Automatic vectorizer or parallelizer
  • in fact, no code is generated to accelerate analysis
• GAP does not ask the programmer to change algorithms, transformation ordering or internal heuristics of compiler
  • It is restricted to changes applied to the program to be compiled
Workflow with Compiler as a Tool

Application Source C/C++/Fortran → Compiler → Application Binary + Opt Reports → Performance Tools → Identify hotspots, problems

Application Source + Hotspots → Compiler in advice-mode → Advice messages → GAP

Modified Application Source → Compiler (extra options) → Improved Application Binary

Simplifies programmer effort in application tuning
GAP – How it Works
Compiler Switches for GAP [1]

- Activate GAP and optionally define guidance level
  \{L&M\}:  -guide[=level] \{W\}:  /Qguide[:level]

- Activate GAP individually for auto-vectorization, auto-parallelization or data transformations
  \{L&M\}:
  -guide-vec[=level]
  -guide-par[=level]
  -guide-data-trans[=level]
  \{W\}:
  -guide-vec[=level]
  -guide-par[=level]
  -guide-data-trans[=level]

Optional argument level=1,2,3,4 controls extend of analysis:
‘4’ is most advanced / most detailed and is default
You must also specify option -parallel (Linux* OS and Mac OS* X) or /Qparallel (Windows* OS) to receive auto-parallelization guidance
GAP – How it Works
Compiler Switches for GAP [2]

• Control the source code part for which analysis is done

  \{L&M\}: \texttt{-guide-opts=<string> }  \{W\}: \texttt{/Qguide-opts::<string>}

  Samples for <string>:
  - "init.c, 1-50,100-150"
    Restrict analysis to file init.c, lines 1-50 and 100-150
  - "bar.f90,'m1::func_solve`"
    Restrict analysis to file bar.f90, Fortran module “m1”,
    function ‘func_solve’

• Control where the message are going – into a new file or append messages to existing file

  \{L&M\}:
  - \texttt{-guide-file=<file_name> }
  - \texttt{-guide-file-append=<file_name>}

  \{W\}:
  - \texttt{/Qguide-file::<file_name>}
  - \texttt{/Qguide-file-append::<file_name>}

Vectorization Example [1]

```c
void f(int n, float *x, float *y, float *z, float *d1, float *d2)
{
    for (int i = 0; i < n; i++)
        z[i] = x[i] + y[i] - (d1[i]*d2[i]);
}
```

GAP Message:
g.c(6): remark #30536: (LOOP) Add -Qno-alias-args option for better type-based disambiguation analysis by the compiler, if appropriate (the option will apply for the entire compilation). This will improve optimizations such as vectorization for the loop at line 6. [VERIFY] Make sure that the semantics of this option is obeyed for the entire compilation. [ALTERNATIVE] Another way to get the same effect is to add the "restrict" keyword to each pointer-typed formal parameter of the routine "f". This allows optimizations such as vectorization to be applied to the loop at line 6. [VERIFY] Make sure that semantics of the "restrict" pointer qualifier is satisfied: in the routine, all data accessed through the pointer must not be accessed through any other

The compiler guides the user on source-change and on what pragma to insert and on how to determine whether that pragma is correct for this case.
Vectorization Example [2]

```c
void mul(NetEnv* ne, Vector* rslt
         Vector* den, Vector* flux1,
         Vector* flux2, Vector* num
    {
    float *r, *d, *n, *s1, *s2;
    int i;
    r=rslt->data;  d=den->data; n=num->data; s1=flux1->data; s2=flux2->data;

    for (i = 0; i < ne->len; ++i)
    r[i] = s1[i]*s2[i] + n[i]*d[i];
}
```

GAP Messages (simplified):
1. “Use a local variable to hoist the upper-bound of loop at line 29 (variable: ne->len) if the upper-bound does not change during execution of the loop”
2. “Use "#pragma ivdep" to help vectorize the loop at line 29, if these arrays in the loop do not have cross-iteration dependencies: r, s1, s2, n, d”

-> Upon recompilation, the loop will be vectorized
Data Transformation Example

```
struct S3 {
    int a;
    int b; // hot
    double c[100];
    struct S2 *s2_ptr;
    int d;   int e;
    struct S1 *s1_ptr;
    char *c_p;
    int f;   // hot
}
```

```
for (ii = 0; ii < N; ii++){
    sp->b = ii;
    sp->f = ii + 1;
    sp++;
}
```

peel.c(22): remark #30756: (DTRANS) Splitting the structure 'S3' into two parts will improve data locality and is highly recommended. Frequently accessed fields are 'b, f'; performance may improve by putting these fields into one structure and the remaining fields into another structure. Alternatively, performance may also improve by reordering the fields of the structure. Suggested field order:'b, f, s2_ptr, s1_ptr, a, c, d, e, c_p'. [VERIFY] The suggestion is based on the field references in current compilation...
Sometimes the **High-Level-Optimiser** sorts out many of the problems ...

Especially **Loop transformations**

HLO is applied **before** vectorisation.
Step 7

Implementing Parallelism
Guided Automatic Parallelization (GAP)

• Feature of the C++/Fortran Compiler that offers advice and, when correctly applied, results in auto-vectorization or auto-parallelization of serial code.
  – Compiler switch /Qguide in parallel with /O2 or higher gives advise for auto-vectorization
  – Compiler switch /Qguide in parallel with /Qparallel or higher gives advise for auto-parallelization

• Compiler runs in advisor mode
  – Compiler does not generate ANY code with /Qguide, – guide – vectorized or not.
Auto-parallelization

• Key requirements
  – A compiler must not alter the program semantics
  – If the compiler cannot determine all dependencies, it has to forego parallelization

• Compilers sometimes need to act very conservatively
  – Pointers make it hard for the compiler to deduce memory layout
  – Codes may produce overlapping arrays through pointer arithmetics
  – If the compiler can’t tell, it does not parallelize

• Past 30 years have shown that auto-parallelization
  – is a tough problem in general
  – is only applicable to very regular loops
  – cannot take care of manual parallelization tasks
Data Dependencies

• Suppose two statements S1 and S2
• S2 depends on S1, if S1 must be executed before S2
  – Control-flow dependence
  – Data dependence
  – Dependencies can be carried over between loop iterations

• Flavors of data dependencies

- **FLOW**
  - \( s1: a = 40 \)
  - \( b = 21 \)
  - \( s2: c = a + 2 \)

- **ANTI**
  - \( s1: b = 40 \)
  - \( s1: a = b + 1 \)
  - \( s2: b = 21 \)
Compiler Options for Parallelization

• Parallelization is not automatically enabled
• Can be controlled through command line switches:
  -parallel enable parallelization
  -par-threshold\(n\) parallelization threshold
    \(n = 0\) parallelize always
    \(n = 100\) parallelize only if gain has probability of 100%
    \(n = 50\) parallelize if probability of performance gain is 50%

• Parallelization reports:
  -par-report\(n\) enable vectorization diagnostics
    0 report no diagnostic information.
    1 report on successfully parallelized loops (default)
    2 report on successfully and unsuccessfully parallelized loops
    3 like 2, but also give information about proven and assumed data dependencies
Compiler Options for Parallelization

- Controlling scheduling: \texttt{-par-schedule-\textit{keyword}}
  - auto: Lets the compiler or run-time system determine the scheduling algorithm.
  - static: Divides iterations into contiguous pieces.
  - static-balanced: Divides iterations into even-sized chunks.
  - static-steal: Divides iterations into even-sized chunks, but allows threads to steal parts of chunks from neighboring threads.
  - dynamic: Gets a set of iterations dynamically.
  - guided: Specifies a minimum number of iterations.
  - guided-analytical: Divides iterations by using exponential distribution or dynamic distribution.
  - runtime: Defers the scheduling decision until run time.
Parallelization Example

```c
#define N 10000
double A[N], B[N];
int bar(int);
void foo(){
    int i;
    for (i=0;i<N;i++)
        A[i] = B[i] * bar(i);
}
```

funcall.c(6): remark #30528: (PAR) To parallelize the loop at line 6, annotate the routine bar with __attribute__((const)). [VERIFY] Make sure the routine satisfies the semantics of such annotation. [ALTERNATIVE] A weaker annotation that can achieve similar effect is __attribute__((concurrency_safe(profitable))). [VERIFY] Make sure the routine satisfies the semantics of this annotation. [ALTERNATIVE] Yet another way to help the loop being parallelized is to inline said routine with "#pragma forceinline recursive" (this method does not guarantee parallelization)

Compiler invocation:

```
icc -c funcall.c
   -parallel
   -guide
   -par-threshold0
```
The Seven Steps

Step 1
Build with optimization disabled

Step 2
Use General Optimizations

Step 3
Use Processor-Specific Options

Step 4
Add Inter-procedural

Step 5
Use Profile Guided Optimization

Step 6
Tune automatic vectorization

Step 7
Implement Parallelism or use Automatic Parallelism

Out-of-the-box performance

Enhanced performance
Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the “Intel® Compiler User and Reference Guides” under “Compiler Options.” Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools 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 Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® 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.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101
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 PROPERTY 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.


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

Copyright © 2010. Intel Corporation.

http://intel.com/software/products