Compiling C and C++ Programs with Intel's Compiler

Speed Compiler

Intel offers a compiler suite designed to optimize programs written in C, C++, and Fortran for its own processors. This article looks into embedding Intel tools in a Linux environment and discusses the advantages they offer.

BY STEPHAN SIEMEN

It is common knowledge that most Linux programs are written in C or C++, and that *gcc* or *g++* from the GNU Compiler Collection (GCC) [1] are typically used to compile them. Intel also provides C, C++, and Fortran compilers both for Linux and Windows.

In contrast to the GCC compilers, the Intel programs are commercial. However, the processor manufacturer has decided to provide its Linux compiler free of charge for non-commercial use. Intel has 32 and 64 bit compilers for i386 and Itanium series processors, but we will be concentrating on the 32-bit variant in this article.

Intel and GCC

One of Intel's major motivations towards developing a compiler was to demonstrate and provide access to the whole range of capabilities provided by Intel processors. Thus, you can expect current and future compilers to leverage the characteristics of Intel processors. Just like GCC, Intel also adheres to the latest standards such as C99 for the C compiler.

The Intel compiler provides optimizations that are not yet available for other compilers, such as the following features:

- Automatic vectorizing changes the structure of the code to allow machine programs to leverage SIMD extensions (Single Instruction, Multiple Data).
- The compiler uses OpenMP to distribute parallelized (multithreaded) code across multiple processors.
- For interprocedural optimization (to avoid overhead caused by function calls) icc is not restricted to file boundaries.
- Profile based optimization allows the compiler to aggregate statistics when executing a program and optimize the code to reflect these figures.

Even if you do not use Intel programs there is very little danger that your programs will not support these capabilities in future. Over the past few years, the GCC developers have demonstrated how quickly they can implement options of this type. It should not take too long for the GCC compiler to catch up and present similar features.

Intel Compilers

Includes:	Compiler for C, C++, and Fortran, debugger, profiler and various other tools in addition to several optimized libraries.
License:	Proprietary
Price:	Free of charge for non-commercial use, otherwise US $400 (C/C++) or US $700 (Fortran) per compiler license.
Features:	Support OpenMP, MMX, SSE, and SSE2, automatic creation of parallelized and SIMD programs.
Hardware:	Any x86 processor and Intel's Itanium/Itanium2 64-Bit processors.

Compiler Options

After completing the installation and configuration steps (see the "Installation" box), the C and C++ compilers should be available as *icc* and *icpc*. Version 7.1 provides similar options to those provided by the GCC compiler, particularly the parameters that control code translation.

However, the optimization options are very different in parts. Table 1 provides an overview of some of the most interesting options the Intel compiler offers. Figure 1 shows a simple compiler operation with *icc* converting loops to vector operations.

The *-axW* flag tells the compiler to rewrite the code to use SIMD vector instructions. The *icc* command will then output the code segments it was able to convert. In our case a loop has been converted to SIMD code (vectorized), and the *calculatesine()* (see Listing 1) has been designated for automatic CPU dispatching. This is the compiler's capability to automatically create multiple

versions of a function, each of which has been optimized for a specific processor, in the object code. The program can then decide which version to launch at runtime.

Compatibility to GCC

On Linux the GCC compilers are the standard tools for compiling source code. From the kernel, through the X Server to most Linux applications, GCC – and more specifically the C compiler – have always played a central role in the development of Linux. Programmers wanting to use a different compiler always need to ensure that it will cooperate with GCC, or should at least be aware of any compatibility issues.

One important point is binary compatibility between the object files created by the compiler. To be able to mix object code created by arbitrary compilers, most manufacturers tend to use a common ABI (Application Binary Interface) to describe the format of object files. Object files can only be interlinked if they are ABI-compatible. Intel respects the goal of a common interface, and GCC introduced this ABI in Version 3.2, however, there are one or two slight differences. More information on ABI is available from [3].

First the good news: The Intel C and C++ compilers are binary compatible to the GCC C compiler. Object files created by different tools can thus be linked together. This is important as many Linux libraries are compiled using *gcc*.

The bad news is that object files created using *g++* cannot be linked with similar files created by the Intel C++ compiler. Some ABI deviations on the part of *g++* are to blame, but let's hope for fully compatible versions in future.

Listing 1: main.c

```
#ifdef __INTEL_COMPILER
#include <mathimf.h>
#else
#include <math.h>
#endif

int calculatesine(double *a, ➥
double *b, int N) {

  int i;

  for(i=0; i<N; i++){
    b[i] = sin(a[i]);
  }
}
```

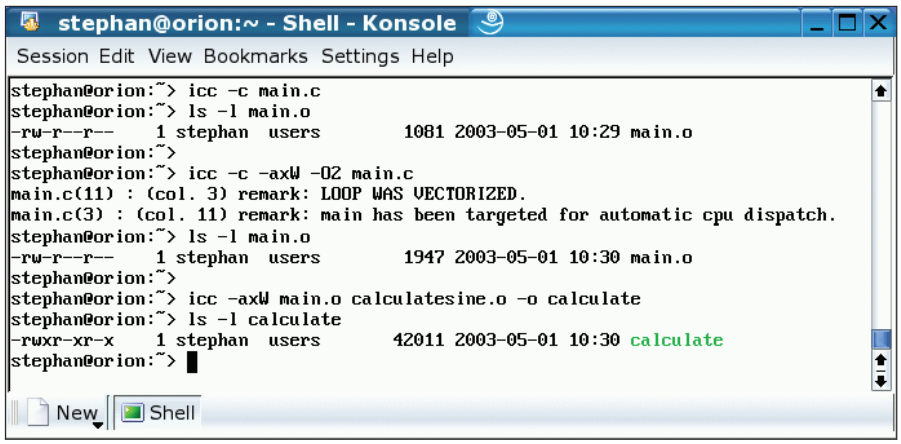

Figure 1: While compiling *main.c* the Intel compiler vectorizes a loop and designates a function for automatic CPU dispatch optimization

GCC Extensions

There are also some differences in the instruction set a compiler understands. GCC comprises a few extensions that the ANSI standard does not include. Intel's C++ compiler understands the ANSI standard, its own extensions, and most GCC extensions. A list of all GCC extensions and details on the extensions supported by the Intel C++ compiler is available from [5].

Installation

You will need a Linux kernel version 2.4 or later and about 100 Mbytes of disk space to install the Intel C++ Compiler 7.1. If you intend to use object code compiled by the GNU compiler in the Intel compiler, you will also need Glibc version 2.2.4 or later.

The Intel homepage [2] states that you may experience some issues with Red Hat 8.0, as this distribution uses Glibc 2.2.93. Incidentally, an Intel processor is not essential, although the Intel compiler will optimize the code for the manufacturer's own processors more effectively than for competitive products.

Installation

The first step is to fill out a form at [2] and agree to the terms of the license. The license only permits you to compile private code. After doing so, Intel will mail you with precise instructions at the address you provided. Downloading the package will place a massive 64 Mbyte tarball on your hard disk – it is not gzipped, unfortunately. Unpacking the tarball reveals the RPM files and the *install* script.

Before you start installing, you will first need to copy the license file from the mail message to the directory indicated by the *INTEL_LICENSE_FILE* variable. Although you can choose any directory for this variable, it makes sense to keep to the default at */opt/intel/licenses*.

The first thing *install* does is to prompt for the compiler version you require. (In Version 7.1) Intel offers two 32-bit and three 64-bit variants for various kernels and Glibc versions. Answering the prompt will provide access to the C++ compilers and a debugger. If you are uncomfortable with the Intel tools, the */opt/intel/compiler70/ia32/bin/uninstall* will de-install them cleanly. The path is still *..../compiler70/...* despite the version number having moved to 7.1; this may be designed to simplify an update from 7.0.

Configuration

After installing *icc* you will need to set a few preferences. If you installed the Intel compiler to */opt/intel/*, the preferences for the default options will be located in */opt/intel/compiler70/ia32/bin/icc.cfg*. You can edit this file to reflect your preferences.

Intel provides scripts that automatically set the *PATH* and *LD_LIBRARY_PATH* environment variables. Programmers are advised to load the script appropriate for their shells before compiling, as in the following example for Bash:

```
. /opt/intel/compiler70/ia32/➥
bin/iccvars.sh
```

The admin user on a development system should place these commands in the start files for the shell, such as */etc/bashrc*.

Table 1: Important icc options

Option	Meaning
-v	Display all steps when compiling.
-i_dynamic	Dynamically link Intel libraries.
-pg	Create output for *gprof*.
-tpp6, -tpp7	Options for Pentium II/III and Pentium 4, like the GCC *-mcpu* option.
-axK, -axW	Enable automatic vectorization for Pentium III (SSE) and Pentium 4 (SSE2).
-vec_report{0,1,2,3}	Provides details on code modified by automatic vectorization.
-opt_report	Report on *stderr* containing details on optimizations.
-ip, -ipo	Enables interprocedural optimization for single files or across file boundaries.
-prof_gen, -prof_use, -prof_file	Options for profile guided optimization (PGO).

Listing 2: calculatesine.c

```
int calculatesine(double *a, double *b, int N);

int main() {

  const int N = 100000;
  double a[N], b[N];
  int i;

  for( i=0; i<N; i++)
    a[i] = i;

  for( i=0; i<100; i++)
    calculatesine(a, b, N);
}
```

To demonstrate that its C/C++ compilers are on a par with GCC, Intel intends to use its own programs to compile the Linux kernel in future. It might take some mileage before Intel gets there, but you can follow Intel's progress at [6].

Linking Code

There are quite a few pitfalls to watch out for when linking code with the Intel compiler. Intel uses its own libraries in part. Table 2 provides an overview of the libraries installed with the Intel compiler. It quickly becomes apparent that many libraries are only available as static variants, and this typically makes sense, as it makes it easier to distribute programs compiled on Intel. Most target systems will not have these libraries installed.

The *libcxa* library is obligatory, but it is normally linked in dynamically. There are two ways of changing this:

- Statically linking *libcxa* only: *-static-libcxa*
- Statically linked all libraries: *-static*

In both cases the program you create will have a larger footprint.

Conversely, the *-i_dynamic* option ensures that the compiled program will load Intel libraries dynamically. Figure 2 shows an example of dynamically and statically linked libraries.

Dynamic linking leads to two libraries less with static linking, and reduces the footprint of the executable by about a third.

If you want to know exactly what happens during compilation, you can use the *-v* flag to instruct the compiler to output every step.

Maths à la Intel

libimf is another Intel library. This optimized version of the math library *libm* is used by *icc* by default. If you prefer, however, to still use Libm instead, you will have to link it in before the Libimf library.

To leverage the optimized functions the program will need to include the *mathimf.h* header file. Listings 1 and 2 show some code for a small sample program. *calculatesine.c* uses a loop to calculate 100 000 sine values.

Not very useful perhaps, but it does demonstrate how well a program handles a large number of operations. The file first ascertains whether it is being translated using the Intel compiler, in order to include the right header file for the sine function.

Figure 3 shows how this code compiles. The *-lm* option that GCC requires has been left out on purpose for the *icc* call. It would slow the program down by using Libm instead of the quicker Libimf.

The example also shows that the Linker cannot resolve all the references when *gcc* is used.

To allow the GNU Linker to bind the code successfully, any Intel libraries have to be included as options. This is not necessary when linking with *icc*, as it will automatically select the correct libraries. The following options are required by *gcc*:

Table 2: Intel Compiler Libraries

Library	Description
libcxa.so, libcxa.a	Intel's own library for various C++ techniques, such as RTTI (Run Time Type Identification).
libimf.a	Intel's optimized math library (similar to Libm).
libsvml.a	Short-Vector Math Library; used for vectorization to run code as SIMD.
libirc.a	The compiler uses this library for various optimization operations, such as PGO (Profile Guided Optimization).
libcprts.a, libcprts.so	Intel's standard C++ runtime library.
libguide.a, libguide.so	Library for parallel programming with OpenMP.
libunwind.a, libunwind.so	The Unwinder library analyzes the stack to trace function calls.

Table 3: Tools included in the package

Program	Description
icc	Script for launching the C compiler.
iccfilt	Helps analyze *nm* output.
icid	Outputs information on the compiler installation.
icpi	Compilation problem isolator.
icpc	Script for launching the C++ compiler.
idb	Intel Application Debugger.
Mcpcom	Compiler – should only be launched by *icc* or *icpc* calls.
profmerge	Merges PGO files; is used by the *-prof_use* option.
proforder	Organizes PGO files.
xiar	Creates static libraries, just like GNU's *ar*.
xild	Program for linking object files. Similar to *ld*.

IPO and PGO

Intel's compiler provide optimization techniques called IPO and PGO. Both need to be enabled using additional compiler flags.

IPO: Interprocedural Optimization

Compilers normally compile each code file separately. This also applies to optimization. When IPO is enabled, the compiler will view all the source code files as a group, reorganizing the whole code and automatically adding smaller subprograms to the calling code. Thus, IPO is particularly good at optimizing programs that repeatedly call small and medium-sized functions.
The resulting program is normally smaller and quicker. However, this makes the code difficult to debug and can change the way a program works at times. It makes sense to enable the *-ipo* option when you have finished developing the program to test the results.

PGO: Profile Guided Optimization

Optimization normally takes place during compilation, however, PGO accesses the finished program. The compiler aggregates statistics from multiple test runs in order to perform optimization steps. The following steps are required to use PGO:

- Enable the *-prof_gen* option when compiling your source code.
- Run the compiled program multiple times using typical options and input. The runtimes of the individual program components, referred to as profiles, are stored in temporary files.
- When re-compiling your code, now stipulate the *-prof_use* option to optimize the code so that the most commonly used subroutines are as quick as possible. This also outputs a summary of the optimization steps.

```
gcc main.o calculatesine.o -o ➥
calculate -L/opt/intel/compiler➥
70/ia32/lib -lsvml -limf -lirc
```

Debugger and other Tools

In addition to the C, C++ and Fortran compilers, the Intel package also includes the *idb* debugger. Calling the debugger with *idb -gdb* will allow the debugger to understand GNU debugger *gdb* commands. Intel provides a comprehensive manual for this tool at [9].

The package also includes a few other programs; refer to Table 3 for an overview. If you adhere to the standard installation steps, the programs will be located in */opt/intel/compiler70/ia32/bin/*. The *xiar* and *xlid* tools will create extended libraries that the GNU programs *ar* and *ld* will not support.

A Small Example

Of course, programmers will be interested in the differences between the GCC and Intel compiled code. Unfortunately, it is difficult to produce generic benchmarks. Benchmarks tend to test individual capabilities. So it stands to reason that different investigations will produce different results. An article in the next issue of Linux Magazine will discuss comparisons of this type.

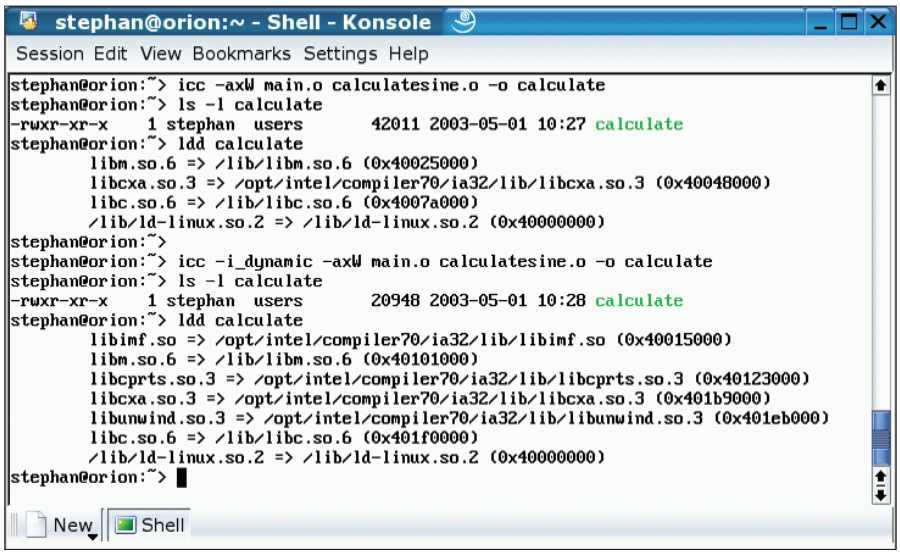

Figure 2: After compiling, a call to *ldd* shows the libraries that the program will load dynamically. Statically linked libraries are not displayed

Documentation

Intel provides a number of PDF document downloads at [2]. This site also hosts the FAQs and support pages [7]. Documentation is in English and it is also quite easy to locate various comparisons between GCC and Intel compilers on the Web – at Coyote Gulch [8], for example.

The Intel compiler package also provides a fair amount of documentation. The */opt/intel/compiler70/docs/* directory contains HTML and PDF docs on the compiler, such as a User's Guide and the Debugger Manual. The Tutorial in */opt/intel/compiler70/training/optimize/index.htm* is also quite interesting. But unfortunately, the documentation tends to concentrate on the Windows version of the compiler, although it does provide useful information on the subject of optimization.

In the following sections we will be using a mini-program from Listings 1 and 2 as an example, to demonstrate the main differences between the various compilers and options.

We can use *time* to ascertain the runtimes for the program versions. This may not be a precise scientific approach, but it is good enough for our purposes and simple to understand.

The *time programname* command outputs three values of time. When running the program, the first line (*real*) contains the total time from launching to terminating the program. The lines that follow (*user* and *sys*) show how long the user program and the operating system took to complete the process.

Table 4: Runtimes

Compiler Call	Runtime
gcc	1.05 seconds
gcc -O2 -march=pentium4	0.90 seconds
icc	0.79 seconds
icc -axW	0.40 seconds

Table 4 contains four measured values (*user* time). After running these tests we can see that the Intel results are a lot quicker than those for GCC. This is also true after enabling optimization.

This trivial ad hoc test shows the kind of improvements that an Intel compiler is capable of providing, particularly in the case of tasks that make heavy use of maths.

Speedy Fortran

Not everyone programs in C and C++. Fortran is still used quite extensively in scientific research. Intel also provides a Fortran compiler subject to the same conditions and on the same website [2] as the C/C++ counterparts. The compiler supports Standard ISO Fortran 95 and OpenMP for parallel programming. The former makes the Fortran compiler extremely interesting for programmers who need access to the new features Fortran 95 provides, but *g77* does not.

Unfortunately, object files compiled with the GNU *g77* compiler cannot be linked together with Intel Fortran object files. As in the case of the C++ compiler this again boils down to ABI incompatibilities.

Conclusion

Intel's compilers are a useful way of leveraging the extra power out of an Intel processor. Intel seems to be making an honest effort to adopt its compilers to the GCC world. Compiling the Linux kernel will provide a genuinely difficult test of Intel's progress in this area.

It remains to be seen, how the Linux Community will take to the Intel compilers. Although these programs are available free of charge for non-commercial use, they are neither Open Source nor free software. ■

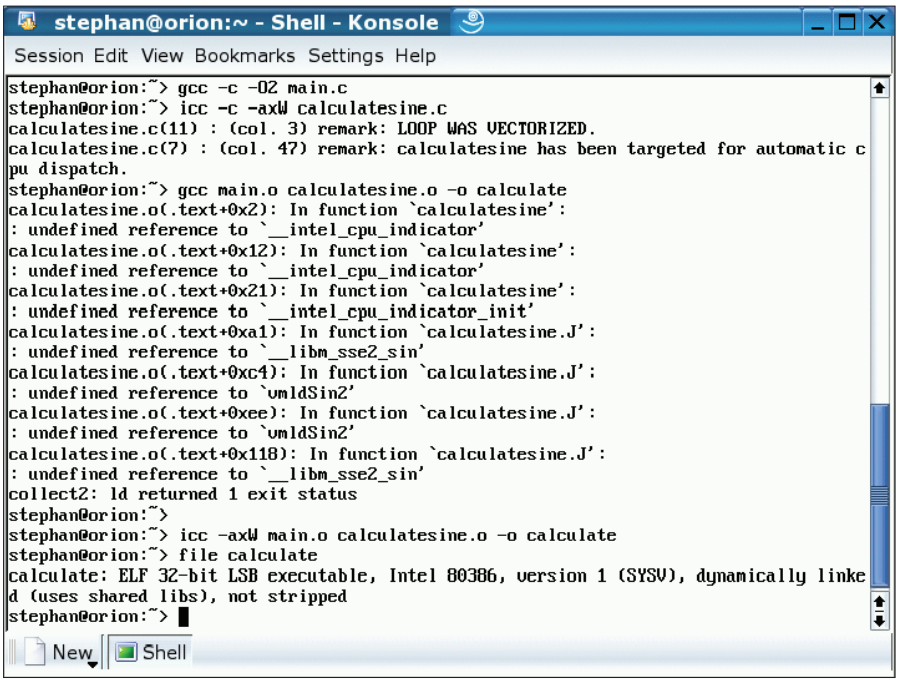

Figure 3: The Intel compiler uses the optimized Libimf library for mathematical operations. To link the results using *gcc*, the libraries need to be specifically included

INFO

[1] GCC: *http://gcc.gnu.org*

[2] Intel Compilers: *http://www.intel.com/software/products/compilers/*

[3] ABI Definition: *http://www.codesourcery.com/cxx-abi/*

[4] ABI Information for GCC: *http://gcc.gnu.org/gcc-3.2/c++-abi.html*

[5] Compatibility of Intel Compilers and GCC: *http://www.intel.com/software/products/compilers/techtopics/LinuxCompilersCompatibility702.htm*

[6] Using Intel compilers to compile the kernel: *http://www.intel.com/support/performancetools/c/linux/kernel.htm*

[7] Intel's Linux pages: *http://www.intel.com/support/performancetools/c/linux/*

[8] Benchmarks: *http://www.coyotegulch.com/reviews/*

[9] IDB Manual *http://www.intel.com/software/products/compilers/techtopics/iidb_debugger_manual.htm*

THE AUTHOR

Stephan Siemen is a research scientist at Essex University (UK), where he is involved in developing software for 3D representation of climatic systems and teaching students computer graphics and programming.