Compiling codes on different systems

- What is compilation?
  - C code
  - Assembler instructions
  - Binary (hex)
- Why do I need to recompile my code?
  - Performance tuning
  - Debugging and error checking
- Compilation - the general case
  - Architecture specific optimizations
    - Bellatrix (Intel SandyBridge)
    - Castor (Intel IvyBridge)
    - Deneb (Intel IvyBridge and Haswell)
    - Fidis (Intel Broadwell and Skylake)
    - Building a "universal" binary
    - Libraries and Codes on the clusters
- Options for Intel and GCC
- Build Nodes
- Related articles

What is compilation?

Compilation is the process by which human readable code (Fortran, C, C++, etc) is transformed into instructions that the CPU understands. At compilation time, one can apply optimizations and thus make the executable code run faster. The "problem" is that optimization is a hard job and so, by default, the compiler will do as little as possible. This means that anything compiled without asking explicitly for optimization will be running a lot slower than it could be.

Compilation is a two-stage process: from human readable code to assembler (which is still readable), then from assembler to a binary object. The following example shows a simple function as it passes through these steps when the compiler (GCC 4.8.3) is invoked without any options via the command "gcc matest.c".

C code

```c
float matest(float a, float b, float c) {
    a = a*b + c;
    return a;
}
```

Assembler instructions

```assembly
matest(float, float, float):
    push rbp
    mov rbp, rsp
    movss DWORD PTR [rbp-0x4], xmm0
    movss DWORD PTR [rbp-0x8], xmm1
    movss DWORD PTR [rbp-0xc], xmm2
    movss xmm0, DWORD PTR [rbp-0x4]
    mulss xmm0, DWORD PTR [rbp-0x8]
    addss xmm0, DWORD PTR [rbp-0xc]
    movss DWORD PTR [rbp-0x4], xmm0
    mov eax, DWORD PTR [rbp-0x4]
    mov DWORD PTR [rbp-0x10], eax
    movss xmm0, DWORD PTR [rbp-0x10]
    pop rbp
    ret
```

Binary (hex)
The binary version is the only thing that the CPU understands but is also the least useful for humans.

Why do I need to recompile my code?

There are two main reasons why you need to recompile your code. The first one is that the compiler will, if asked, compile your code in such a way as to make it run faster and more efficiently. The second equally important case is for debugging and error checking when developing and fixing code.

Performance tuning

You can ask the compiler to try and make your code run faster with the -O flag. The variants -O1, -O2 and -O3 are described below:

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O1</td>
<td>Enables optimizations for speed and disables some optimizations that increase code size and affect speed. To limit code size, this option enables global optimization; this includes data-flow analysis, code motion, strength reduction and test replacement, split-lifetime analysis, and instruction scheduling.</td>
</tr>
<tr>
<td>O2</td>
<td>Enables optimizations for speed. This is the generally recommended optimization level. Vectorization is enabled at O2 and higher levels.</td>
</tr>
<tr>
<td>O3</td>
<td>Performs O2 optimizations and enables more aggressive loop transformations such as Fusion, Block-Unroll-and-Jam, and collapsing IF statements.</td>
</tr>
</tbody>
</table>

Returning to the example shown in the introduction if we compile with -O2 the assembler becomes

```
matest(float, float, float):
mulss  xmm0,xmm1
addss  xmm0,xmm2
ret
```

It's not difficult to see that this requires far fewer steps that the unoptimized version!

There is also a plethora of options related to memory access and other subtleties of the processor architecture that are best left alone, unless you are sure that you know what they do.

In addition to the "general" optimization, different CPUs have different instructions that can be used to make operations faster, with the Intel AVX (Advanced Vector Extensions) being a good example. The new Haswell processors introduced AVX2 which adds a Fused Multiply-Add (FMA) instruction so that a=b*a+c is performed in one step rather than requiring two instructions.

Taking the simple example and optimizing for Haswell processors (read on for how to do this) the assembler becomes

```
matest(float, float, float):
mulss  xmm0,xmm1
addss  xmm0,xmm2
ret
```

It's not difficult to see that this requires far fewer steps that the unoptimized version!
matest(float, float, float):
  vfmadd132ss xmm0,xmm2,xmm1
  ret

So rather than two steps to calculate the product, it takes now only one.

Debugging and error checking

It is often useful to compile code with no optimization and debugging enabled to find bugs in your code. There are also options for error checking that are strongly recommended while you develop your code.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Wall</td>
<td>Enables warning and error diagnostics. This is highly recommended!</td>
</tr>
<tr>
<td>-O0</td>
<td>Disables all optimizations.</td>
</tr>
<tr>
<td>-g</td>
<td>Tells the compiler to generate full debugging information which is very useful when tracking down errors.</td>
</tr>
<tr>
<td>-check bounds / -fbounds-check</td>
<td>(Intel / GCC) For Fortran code this enables compile-time and run-time checking for array subscript and character substring expressions. An error is reported if the expression is outside the dimension of the array or the length of the string. For array bounds, each individual dimension is checked. For arrays that are dummy arguments, only the lower bound is checked for a dimension whose upper bound is specified as * or where the upper and lower bounds are both 1.</td>
</tr>
<tr>
<td>-pedantic</td>
<td>(GCC only) This will issue all the warnings demanded by strict ISO C and ISO C++; reject all programs that use forbidden extensions, and some other programs that do not follow ISO C and ISO C++. For ISO C, follows the version of the ISO C standard specified by any -std option used.</td>
</tr>
<tr>
<td>-Werror</td>
<td>(GCC only) This will turn all warnings into hard errors. Source code which triggers warnings will be rejected.</td>
</tr>
</tbody>
</table>

Compilation - the general case

Here we provide examples using the syntax of the Intel Composer. A table showing the equivalent options for the GCC compilers can be found at the bottom of the page.

To compile something we need the following information:

- The name of the source code file(s)
- The libraries to link against
- Where to find these libraries
- Where to find the header files
- A nice name for the executable

Putting this together, we get something like

```
compiler -l libraries -L <path to libraries> -I <path to header files> -o <name of executable> mycode.c
```

where the compiler might be gcc, icc, ifort or something else.

To this we may add options such as:

```
-O3 -xCORE-AVX2
```

This will perform aggressive optimization and use the features for the CORE-AVX2 architecture (Haswell).

A code compiled with the above options will run optimally on Haswell CPUs, but will not run at all on older systems.

```
-O0 -Wall -g -check bounds
```

This performs no optimization, gives lots of warnings and adds full debugging information in the binary as well as bounds checking for Fortran arrays.

This code will run slowly but will point out syntax problems, tell you if you make errors when accessing arrays and provides clear information when run through a debugger such as GDB or TotalView.
Architecture specific optimizations

As already mentioned, different CPUs have different features that we may want to take advantage of and this implies recompiling applications for different families of CPUs.

The easiest option is to use `xHOST` which will optimize the code for the architecture on which it is compiled. This approach works well on Bellatrix and Castor.

If you were to call this explicitly on our clusters, then the options needed are:

**Bellatrix (Intel SandyBridge)**

`-xAVX`

Which means "May generate Intel(R) Advanced Vector Extensions (Intel(R) AVX), Intel(R) SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel(R) processors."

**Castor (Intel IvyBridge)**

`-xCORE-AVX-I`

Which means "May generate Float-16 conversion instructions and the RDRND instruction, Intel(R) Advanced Vector Extensions (Intel(R) AVX), Intel(R) SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel(R) processors."

For scientific codes there is little difference in the instructions between that and SandyBridge, so `-xAVX` works just as well.

**Deneb (Intel IvyBridge and Haswell)**

This is where things get a bit complicated as Deneb is a heterogeneous cluster with two different CPU architectures.

For the *IvyBridge* part, as already seen:

`-xCORE-AVX-I`

For the *Haswell* part

`-xCORE-AVX2`

Which means "May generate Intel(R) Advanced Vector Extensions 2 (Intel(R) AVX2), Intel(R) AVX, SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel(R) processors."

On Deneb, if you wish to run your jobs on a specific architecture, then please, pass the following option to the batch system

For *IvyBridge*

```bash
#SBATCH --constraint=E5v2
```

For *Haswell*

```bash
#SBATCH --constraint=E5v3
```

If you do not specify anything, then the code may run on either, but a multi-node job will never span both architectures - it will run either all on IvyBridge or all on Haswell. The E5v2 and E5v3 are the "official" model types for the processors.

Please note that the debug partition contains two IvyBridge nodes and two Haswell nodes.

**Fidis (Intel Broadwell and Skylake)**

Fidis, like Deneb, is a heterogeneous cluster with two different CPU architectures.

For the *Broadwell* part (`--constraint=E5v4`):

`-xCORE-AVX2`
Which is the same as for the Haswell processor

For the **Skylake** nodes (``-constraint=s6g1``):

``-xCORE-AVX512``

Which means "May generate Intel(R) Advanced Vector Extensions 512 (Intel(R) AVX-512) Foundation instructions, Intel(R) AVX-512 Conflict Detection instructions, Intel(R) AVX-512 Doubleword and Quadword instructions, Intel(R) AVX-512 Byte and Word instructions and Intel(R) AVX-512 Vector Length extensions, as well as the instructions enabled with CORE-AVX2."

Building a “universal” binary

Rather than just building for one processor, we can ask the compiler to generate feature-specific auto-dispatch code paths if there is a performance benefit.

``-xAVX -xCORE-AVX2``

where -x gives the baseline for the compilation and -ax is a list of the feature-specific code paths to build.

As the Intel compiler documentation explains:

> If the compiler finds such an opportunity, it first checks whether generating a feature-specific version of a function is likely to result in a performance gain. If this is the case, the compiler generates both a feature-specific version of a function and a baseline version of the function. At run time, one of the versions is chosen to execute, depending on the Intel(R) processor in use. In this way, the program can benefit from performance gains on more advanced Intel processors, while still working properly on older processors and non-Intel processors. A non-Intel processor always executes the baseline code path.

---

**Note for GCC compilers**

The GCC compilers do not support multiple code paths and so a "universally" optimized binary is not possible. Here -march gives the baseline and -mtune the processor to tune for whilst respecting the instruction set of the baseline

``-march=corei7-avx -mtune=core-avx2``

This means "using the features available to the SandyBridge processors tune the code so that it would run optimally on a Haswell processor". Such an optimization would not be able to make use of the FMA instructions as they are not present on the baseline.

The official documentations states

> Tune to cpu-type everything applicable about the generated code, except for the ABI and the set of available instructions. While picking a specific cpu-type schedules things appropriately for that particular chip, the compiler does not generate any code that cannot run on the default machine type unless you use a -march=cpu-type option. For example, if GCC is configured for i686-pc-linux-gnu then -mtune=pentium4 generates code that is tuned for Pentium 4 but still runs on i686 machines.

---

**Libraries and Codes on the clusters**

The software installed on the clusters is, in general, optimised for the relevant hardware. Behind the scenes, the way this works is that we build and optimise the packages provided via modules for each architecture thanks to the magic of [SPACK](https://spack.readthedocs.io) and [Jenkins](https://jenkins.io).

The exceptions are "commercial" closed source codes such as MATLAB and Mathematica where we have no control over how they are optimised.

---

**Courses and further reading**

SCITAS offers a number of courses that you may find useful with the [full list here](#)

In particular you might wish to attend

- [Compiling code and using MPI on the central HPC facilities](#)
- [Introduction to profiling and software optimisation](#)

Everything you never wanted to know about the internals of CPUs can be found in the [Intel® 64 and IA-32 Architectures Software Developer Manuals](https://software.intel.com/content/www/us/en/architecture/intel-64-and-ia-32-architectures-software-developer-manuals.html)
Options for Intel and GCC

This table shows the main differences between the options given to the Intel and GCC compilers. The majority of the basic options (-g, -O2, -Wall etc) are the same for both.

<table>
<thead>
<tr>
<th>Intel</th>
<th>GCC</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-xAVX</td>
<td>-march=corei7-avx</td>
<td>SandyBridge Optimizations (GCC 4.8.3)</td>
</tr>
<tr>
<td>-xAVX</td>
<td>-march=sandybridge</td>
<td>SandyBridge Optimizations (GCC 4.9.2 and newer)</td>
</tr>
<tr>
<td>-xCORE-AVX2</td>
<td>-march=core-avx2</td>
<td>Haswell Optimizations (GCC 4.8.3)</td>
</tr>
<tr>
<td>-xCORE-AVX2</td>
<td>-march=haswell</td>
<td>Haswell Optimizations (GCC 4.9.2 and newer)</td>
</tr>
<tr>
<td>-xCORE-AVX2</td>
<td>-march=broadwell</td>
<td>Broadwell Optimizations (GCC 4.9.2 and newer)</td>
</tr>
<tr>
<td>-xCORE-AVX512</td>
<td>-march=skylake-avx512</td>
<td>Skylake Server Optimisations (GCC 6.4 and newer)</td>
</tr>
<tr>
<td>-xHOST</td>
<td>-march=native</td>
<td>Optimize for the current machine</td>
</tr>
<tr>
<td>-check bounds</td>
<td>-fbounds-check</td>
<td>Fortran array bounds checking</td>
</tr>
</tbody>
</table>

Build Nodes

The SCITAS clusters have a dedicated partition for compiling codes on specific architectures.

To use this you should use the Sinteract tool and specify the build partition and the appropriate architecture with the "-s" flag:

```
Sinteract -p build -s E5v4 -c 2 -m 16G
```

The list of clusters and architectures is:

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Architecture</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deneb</td>
<td>E5v2</td>
<td>Ivy Bridge</td>
</tr>
<tr>
<td></td>
<td>E5v3</td>
<td>Haswell</td>
</tr>
<tr>
<td>Fidis</td>
<td>E5v4</td>
<td>Broadwell</td>
</tr>
<tr>
<td></td>
<td>s6g1</td>
<td>Skylake</td>
</tr>
<tr>
<td>Helvetios</td>
<td>s6g1</td>
<td>Skylake</td>
</tr>
</tbody>
</table>

Be aware that the build nodes are shared and you need to request the appropriate number of CPUs and amount of memory.

Related articles

- Compiling codes on different systems