Compilation essentials

Programming languages for HPC (EPCC11019)

Ludovic Capelli

EPCC

Contributors

In this material:

- The slides are created using LaTeX *Beamer* available at `https://ctan.org/pkg/beamer`.

License - Creative Commons BY-NC-SA-4.0[1]

Non-Commercial you may not use the material for commercial purposes.

Shared-Alike if you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

Attribution you must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

[1] You can find the full documentation about this license at:
`https://creativecommons.org/licenses/by-nc-sa/4.0/`

Nota bene

Format

This set of slides is designed to be covered as a **live presentation**, and thus contains little text and explanations. It is **less suitable** to be studied as a stand-alone document.

Structure

In this session, you will learn:

- What compilers are
- Why we need them
- How they work
- How to leverage their potential

Table of Contents

What is it?

- Your machine only understands binary code.
- C/C++ codes make no sense to your machine in that form.
- Must transform it to binary code: *compile* it.

I did not need one in Python!

- Certain programming languages such as Python are *interpreted*.
- When you execute a Python script, behind the scenes, an additional program called *an interpreter* is launched.
- The interpreter reads your program source code and issues the corresponding binary instructions on the fly.

Interpreted vs compiled

- Interpreted languages are convenient because they **do not need** to be compiled. They can be written once and run on any machine with an interpreter installed.
- The extra work of the interpreter comes at the expense of performance... an obstacle in **High-Performance** Computing.
- During compilation, the tool used has an opportunity to apply advanced optimisation techniques on the source code to compile, resulting in performance gains that can be of **orders of magnitude**.
- Compiled programming languages typically give a better control to the programmer about data placement, memory usage, etc... which are important features in HPC.

Time to get ready

- To compile, you need a tool called a **compiler**.
- The one we will use in this coding refresher is free and popular, called GCC (stands for **G**NU **C**ompiler **C**ollection).
- To activate it on Cirrus, simply load the `gcc` module:

```
1 module load gcc
```

Variety

Intel compiler: `icc`

```
module load intel-20.4/compilers
icc -o binary_name source1.c source2.c
```

GNU compiler: `gcc`

```
module load gcc
gcc -o binary_name source1.c source2.c
```

Clang compiler: `clang`

```
<not available on Cirrus>
clang -o binary_name source1.c source2.c
```

Variety

Tip

Put the compiler command in an environment variable, so that you can easily change all compiler invocation commands simply by changing a single line. Useful for benchmarks...

Flags remain mostly identical, in particular optimisation levels and warning-related flags such as `-Wall`.

From source code to executable

- In C, we typically call the main source file `main.c`
- To compile it:

```
1 gcc main.c
```

- By default, the executable produced is named `a.out`
- To execute it:

```
1 ./a.out
```

From source code to execution

- Compilers have many options, type `gcc --help` to have a list.
- An option (also called *flag*) we always use is "`-o`"[2]: it allows to change the output produced (both the location and the filename).
- To change the name of the executable to `main` and put it in the directory `my_dir`:

```
1 gcc -o my_dir/main main.c
```

- To execute `main` from the directory `my_dir`:

```
1 ./my_dir/main
```

[2] It is the **lowercase** letter 'o', not the digit 0.

Multi-file compilation

You now have headers with function declarations, source files with function definitions and your main file. However, you must still compile it all. To do so:

- You must list all sources files.
- You must **not** list header files.
- Assuming you have two source files: `main.c` and `my_functions.c`, and one header `my_functions.h`:

```
1 gcc -o main main.c my_functions.c
```

Table of Contents

Compilation pipeline

The compilation process comprises of multiple phases. The output of each phase is fed into the next phase, making a pipeline:

Lexical analysis (lexer) Reads your code and break it down into **tokens**. ("abc" is a string, '=' is a symbol, "if" is a keyword etc...)

Syntax analysis (parser) Check that these tokens are arranged in a valid way, and builds an **Abstract Syntax Tree (AST)** from it. (variable + '=' + number + ';' is an assignment)

Compilation pipeline

Compilers typically rely on a pipeline that has four stages:

Semantic analysis Check that what is written makes sense. (Not calling an undefined variable)

Optimisation analysis Applies techniques to rewrite / rearrange instructions to improve a given metric (performance, size, memory usage etc...)

Example: lexical analysis

Input:

```
1 int c = 123;
2 c = 456;
```

Work:

- Look at each space-separated block.
- Compare them against a table of regular expressions.
- Indicate the purpose of each block (i.e.: token).

Output:

```
1 keyword space identifier space symbol space number semi-
      colon
2 identifier space symbol space number semi-colon
```

Example: syntax analysis

Input:

```
keyword space identifier space symbol space number semi-colon
identifier space symbol space number semi-colon
```

Work:

- Compares each series of tokens against a table listing possible combinations.
- Maps each to corresponding statement.

Output:

```
declaration & assignment combined (c is int, c <- 123)
assignment (c <- 456)
```

Example: semantic analysis

Input:

```
1 declaration & assignment combined (c is int, c <- 123)
2 assignment (c <- 456)
3
```

Work:

- Declaring an int and assigning 123 to it: OKAY.
- Assigning 456 to it: OKAY.

Output:

```
1 declaration & assignment combined (c is int, c <- 123)
2 assignment (c <- 456)
```

Example: optimisation analysis

Input:

```
1 declaration & assignment combined (c is int, c <- 123)
2 assignment (c <- 456)
```

Work:

- Analyses for potential optimisations
- Applies optimisations deemed suitable

Output:

```
1 declaration & assignement combined (c is int, c <- 456)
```

Loops optimisations

- unrolling
- fusion
- fission /splitting
- inversion
- interchange
- peeling
- tiling / blocking
- unswitching
- vectorisation
- parallelisation
- strength reduction
- predication
- guard optimisation
- reversal

General optimisations

- dead code analysis
- function inlining
- out-of-order execution
- constant folding
- constant propogation
- strength reduction
- jump threading
- global value numbering
- branch prediction hints
- alias analysis
- code motion
- tail code optimisation
- common subexpression elimination

Optimisation needs

- Compilation is the moment you give to the compiler to go through your code and apply as many optimisations as it can while producing the binary.
- Interpreted languages only see instructions when the program executes, so do not have this phase and therefore benefit from only a subset of these optimisations.

Table of Contents

Levels of optimisations

Compilers typically offer multiple levels of optimisations. Generally, there are three levels:

- `-O0` Disable all optimisations
- `-O1` Safe optimisations
- `-O2` More aggressive safe optimisations
- `-O3` Unsafe optimisations
- `-Os` Optimise for size
- `-Og` Optimise for debugging

A few questions...

Question

Why would we ever want to use `-O0`?

Definition

It minimises the amount of information lost for debugging, as well as providing the fastest compilation time. It is also useful when checking if incorrectness is due to an optimisation technique, or when seeking a baseline for optimisation benchmarking.

A few questions...

Question

Why would we want to use something else than `-O3`?

Definition

The `-O3` optimisation level enables what is referred to as *unsafe* compiler optimisations, which impose more preconditions in order to leverage more room for optimisation. In addition, `-O3` can significantly increase the compilation time.

A few questions...

Question

Why ever using `-O1` instead of `-O2`?

Definition

Both are safe but `-O2` may increase the amount of time necessary for compilation time. In addition, it can also result in a greater loss of information for debugging.

A few questions...

Compilers are complex creatures, but now rather well documented. If you would like to learn more about compilation, please refer to the corresponding compiler documentation. For instance, for GCC:

- GCC webpage
- `gcc --help`
- `man gcc`

Table of Contents

Context

- When compiling the way we did previously, we turn all source files into a single executable binary in a unique step.
- If there is a single modification in one file, **all** source files will be **entirely** recompiled.
- In large projects, a full recompilation can last hours.
- Can make the compilation more modular to treat separate components individually.
- Changing one component will not result in a recompilation of other components.
- Instead of spending hours to recompile everything, one spends only a few seconds compiling only the changed file(s).

Terminology

What we typically refer to as *compilation* is actually two distinct processes: *compilation* and *linking*.

- Compilation
  - Generates object files from source files.
  - Object files are in binary format, and traditionally use a "`.o`" file extension.
  - The compiler makes sure that every function called matches a function declaration it can find.

Terminology

- Linking
  - It "connects" the different object files into the final executable file.
  - Applies *link resolution*, where function declarations and definitions are mapped.

Example

In `gcc`, you must pass the "`-c`" flag to ask for the production of object files.

- Compile the source file `main.c` to an object file

```
1 gcc -o main.o -c main.c
```

- Compile the source file `my_functions.c` to an object file

```
1 gcc -o my_functions.o -c my_functions.c
```

- Produce the executable by linking object files

```
1 gcc -o main main.o my_functions.o
```

Separating compilation and linking

This two-phase approach allows you to develop libraries and plug-ins.

- You write an interface, with function declarations, which act as the Application Programming Interface.
- You implement their definitions.
- You compile them

It also allows you to respect certain licenses such as GPL while keeping your code private, where you code must allow the user to freely change the libraries.

Type of errors

Classic errors during compilation

- Call to undefined function
- Use of undeclared variable
- Non-included header

Classic errors in linking

- Mismatch between function declarations and definitions
- Lack of header include guards

Table of Contents

Adding an include path

- When using an `#include` statement, the compiler must find the corresponding file.
- Locations listed as "include paths" will be visited to find the file specified.
- Adding a given location can allow you to use customised locations.
- Use the flag `-I`, followed with location to add.
- Certain locations are present by default (e.g.: `/usr/includes`.

Adding an include path

Example:

- Your program needs `my_func.h` from a separate program, located at `/home/user123/my_includes`
- To add this path, add the following flag to the compilation command: `-I/home/user123/my_includes`

Careful

There is no space, or symbol '=', between `-I` and the location to add.

Adding a library path

- Similarly to include paths, compilers also have library paths.
- These tell the compiler where to look for libraries.
- Notation is `-L` instead of `-I`, the rest is identical.

Adding a library path

Example:

- Your program needs the library `libhpc.so`, located at `/home/user123/my_libs`
- To add this path, add the following flag to the compilation command: `-L/home/user123/my_libs`

Careful

There is no space, or symbol '=', between `-L` and the location to add.

Adding a library

Once the compiler knows where to look for libraries, you can specify the libraries to fetch by using the `-l` flag. For example, to load the library `libhpc.so` mentioned previously:

```
1 -lhpc
```

Careful

You only pass the root name of the library ("hpc") and you prefix it with `-l`, not the extension of the "lib" prefix.

Adding a library

Careful

Do not confuse the uppercase `-L` to add a library path, and the lowercase `-l` to add a library.

Defines

- Passing `-DMY_CONST=123` to the compilation command will define a constant named `MY_CONST` and assign it value 123.
- Defines can simply declare a constant, without assigning a value to it.
- Defines can be handled at compiler level, with the use of the `-D` flag.
- You can use it to:
  - Trigger a debug mode in your code
  - Change a datatype
  - Control parallelism options
  - ...

Optional value

With a value:

```
-DGRID_SIZE=512
```

Without a value:

```
-DUSE_DEBUG_MODE
```

Careful

There is no space, or symbol '=', between `-D` and the constant name.

Table of Contents

Extra guidance

Compilers can also provide more guidance by being more restrictive about the checks it performs.

- Undefined variables
- Unused variables
- Unused return values
- `NULL` pointers
- Statements after return
- Branches never taken
- Returned value not used

Asking for that extra guidance

To allow the compiler to be more picky, use the following flags:

- `-Wall`
- `-Wextra`

To turn all warnings into errors right away, to prevent compilation, use `-Werrors`. To turn errors into fatal errors, to have compilation stop after the first error, use `-Wfatal-errors`

Tip

Keep `-Wall` on all the time.

Standard versions

Programming languages go through major iterations. For instance, the C programming language has the following:

- 1989 / 1990
- 1999
- 2011
- 2017
- 2023

Standard versions

Different versions provide different features. To use a specific version, use `-std=X`, with `X` having one of the values below:

Year	C	C++
1989	`c89`	-
1998	-	`c++98`
1999	`c99`	`c++99`
2003	-	`c++03`
2011	`c11`	`c++11`
2014	-	`c++14`
2017	`c17`	`c++17`
2020	-	`c++20`
2023	`c23`	`c++23`

Table: Value of compilation flags for C and C++ versions.

Table of Contents

Valuable output

- When something goes wrong, compilers typically tells you exactly what went wrong and where.
- Certain errors cascade and get the compiler to display many error messages.
  - Go to the first error.
  - Fix it. It might fix cascading errors.
  - Repeat.
- Familiarise yourselves with common error messages.

Table of Contents

Perform early actions on the code

Preprocessor directives are handled before your actual code is even parsed. For example, in C / C++ source codes, you can find:

- `#include`: copy / paste the content of the file specified at the location of the `#include`.
- `#define`: specifies that a given constant exists, and optionally assigns it a value. (e.g.: useful to turn on/off debug-specific code)
- `#ifdef / #ifndef / #else / #endif`: executes portions of the code only if a constant is defined (`#ifdef`) or not defined (`#ifndef`). (e.g.: useful for inclusion guards or hardware-specific code)

Perform early actions on the code

When doing shared memory parallelism in OpenMP you will see directives such as:

- `#pragma omp parallel`
- `#pragma omp for`
- `#pragma omp barrier`

All these are **preprocessor directives**!

Table of Contents

Limitations to assumptions and predictability

Anything that hinders compilers vision at compile-time makes their life harder to optimise code. Typically:

- Pointer jumping for memory locations (i.e.: pointer on pointer on pointer...)
- Pointer jumping for routines (i.e.: function pointers)
- Runtime-determined information
- Pointer aliasing (i.e.: can two pointers point to the same memory location?)
- Parallelism / non-determinism

Table of Contents

Summary

In this session, you have seen:

- What compilers are
- Why we need them
- How they work
- How to leverage their potential

To go further

- Compiler optimisation is an wide area
  - superoptimisers
  - synthesis-based approaches
  - stochastic superoptimisation
  - ML-aided superoptimisation
- Compilers have *front end(s)* and *back end(s)*.
- Compilers rely on an *intermediate representation* (IR).
- Programming language grammar syntax (CFG, BNF)