Code Optimization

Unleash the Compiler

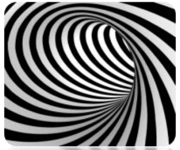
INAF HPC School 2025
Catania, Sep. 22nd - 26th

Luca Tornatore
O.A.T.S

Outline

Avoid the avoidable inefficiencies

Cache & Memory



Loops

Branches

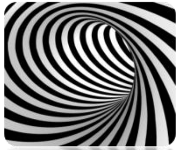
Pipelines

Unleash the Compiler

Outline

Avoid the avoidable inefficiencies

Cache & Memory



Loops

Branches

Pipelines

Unleash the Compiler

Let the compiler do his job

Programming languages are notations for describing computations to people and to machines.

[...] all the software running on all the computers was written in some programming language.

But, before a program can be run, it first must be translated into a form in which it can be executed by a computer.

The software systems that do this translation are called **compilers.**

Taken from "Compilers. Principles, Technqieus & tools", Pearson-Add.Wesley, 2008, 2nd Ed.
Chap. 1, Introduction

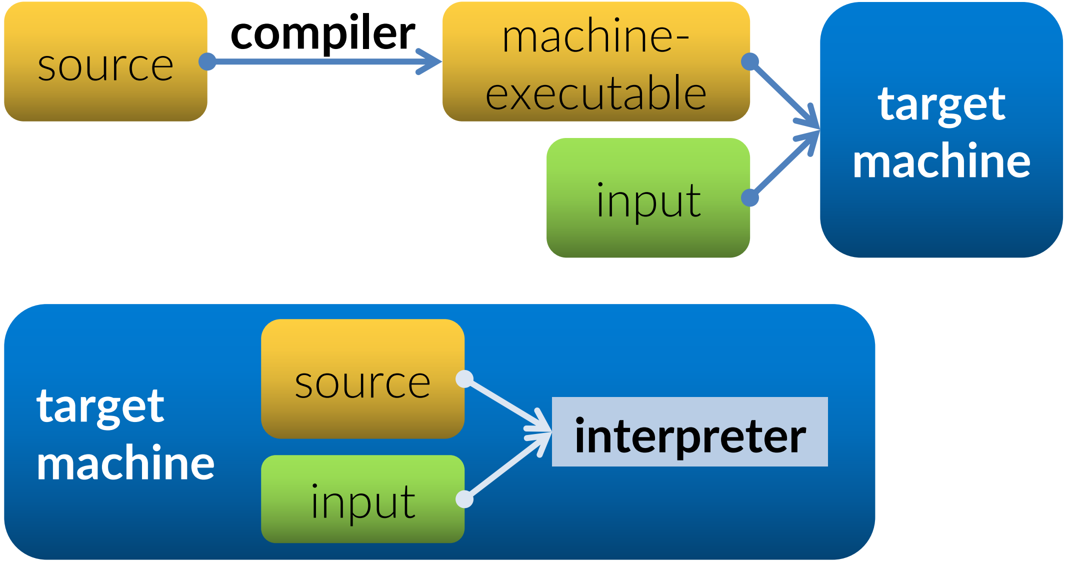
Let the compiler do his job

In simple words, a compiler is **a program that translates a program from a source-language into an *equivalent* program in a target-language**, while also signaling possible errors in it (mostly semantic errors and a sub-set of other types of errors).

If the target-language is executable by a machine, it can then be called directly from the machine to process inputs and produce outputs.

Let the compiler do his job

An **interpreter** is a different language-processing program that executes itself a program in a given source-language.



Usually **compiled languages** execute much faster, while **interpreted languages** offer enhanced error analysis and portability.

FIRST THINGS FIRST

Let the compiler do his job

What we call “the compiler” is a long pipeline

- A ***preprocessor*** may get through the source including headers, expanding macros etc.
- A ***front-end*** specific for some language (C, C++, Fortran,…) may translate the source in a high abstraction-level language.
- An ***assembler*** can actually process the *assembly code* produced by the compiler and output a relocatable machine code (or *object* code) for every compilation unit.
- A ***linker*** resolves memory addressed among different sections of the code and potential references to libraries.

Let the compiler do his job

example:

```c
int main ( void )
{
   int a = 1;
   int b = 2;
   return a + b;
}
```

C source code

compiler →

```asm
...
mov     DWORD PTR -8[rbp], 1
mov     DWORD PTR -4[rbp], 2
mov     edx, DWORD PTR -8[rbp]
mov     eax, DWORD PTR -4[rbp]
add     eax, edx
ret
...
```

x86_64 asm source code

FIRST THINGS FIRST

Let the compiler do his job

```
...
mov     DWORD PTR -8[rbp], 1
mov     DWORD PTR -4[rbp], 2
mov     edx, DWORD PTR -8[rbp]
mov     eax, DWORD PTR -4[rbp]
add     eax, edx
ret
...
```

assembler →

```
...
c7 45 f8 01 00 00 00
c7 45 fc 02 00 00 00
8b 55 f8
8b 45 fc
01 d0
c3
...
```

x86_64 asm source code

disassembly of the object code using `objdump`

FIRST THINGS FIRST

Let the compiler do his job

All the previous passes are as simple as:

```
cc -o example example.c
```

where example.c reads as

```
int main( void ) {
  int a = 1;
  int b = 2;
  return a + b; }
```

Try to lurk at the results of

```
objdump -d example
```

Let the compiler do his job

Compilers are also able to perform **sophisticated analysis** of the source code so that to produce a target code (usually an assembly code) which is **highly optimized for a given target architecture**.

FIRST THINGS FIRST

How to call a compiler

Compilers are plenty of options, so the first good move is to read the manual.

However, standards are in place so that you can immediately deliver basic expected results with every decent compiler.

compile a source	`cc source_name -o executable_name`
compile a source with debugging info	`cc source_name -g -o executable_name`
compile a source with optimizations	`cc -On source_name -o executable_name` where *n* tipically is 1, 2, 3

widely used, high-quality C/C++ compilers:
`gnu (gcc), clang, pgi, intel`

Have a look at the amazing project godbolt:
`https://godbolt.org`

Compiler's optimization

FIRST THINGS FIRST

Optimization level: O*n*

It is not granted that `-O3`, although often generating a faster code, is what you really need.
For instance, sometimes expensive optimizations may generate more code that on some architecture (*e.g. with **smaller caches***) run slower, and using `-Os` may bring surprising results.

Take into accounts that modern compilers allow for local specific optimizations or compilation flags.

In gcc for instance:

```
__attribute__ ((__option__ ("…")))
__attribute__ ((optimize(n)))
```

Compile for specific CPU model

FIRST THINGS FIRST

Optimization level: native

The compiler knows the architecture it is compiling on, of course. However, it will generate a *portable* code , i.e. a code that can run on *any* cpu belonging to that class of architecture.
Example: x86_64, x86_32, ARM, POWER9, are all classes of architecture.

Besides a general set of instructions that all the cpus of a given class can understand, specific models have specific different ISA that are not compatible with others (normally you have back-compatibility).
Using appropriate switch (in `icx` **`-xHost,`** in `gcc` **`-march=native -mtune=native`**,), the compiler will optimize for exactly the specific cpu it's running on, much probably producing a more performant code for it.

Use automatic profiling

FIRST THINGS FIRST

Profile-guided optimization

Compilers (**`gcc`** , **`icc`** and **`clang`**) are able to instrument the code so to generate run-time information to be used in a subsequent compilations.
Knowing the typical execution patterns enables the compiler to perform more focused optimizations, especially if several branches are present.

For **`gcc`**:

```
gcc -fprofile-arcs
< … run … >
gcc -fbranch-probabilities
```

Specific for branch prediction

```
gcc -fprofile-generate
< … run … >
gcc -fprofile-use
```

More general; enables also
`-fprofile-values`
`-freorder-functions`

A remind on memory

Memory allocation

We’ll see some detail about memory allocation.

Try to allocate **contiguous memory** and to **re-use it efficiently** avoiding fragmentation

Some C-specific hint

Storage classes

- `extern`
  Global variables, they exist forever
- `auto`
  Local variables, allocated on the stack for a limited scope, and then destroyed. They must be initialized
- `register`
  Suggests that the compiler puts this variable directly in a CPU register

FIRST THINGS FIRST

Some C-specific hints

Variable qualifiers

- `const`
  Indicates that this variable won't be changed in the current variable's scope.
- `volatile`
  Indicates that this variable can be accessed, and modified, from outside the program.
- `restrict`
  A memory address is accessed only via the specified pointer.

FIRST THINGS FIRST

Memory aliasing

One among the major optimization blockers, probably the primary one, is a poor usage of memory references.

Consider the two functions below : (*)

```
void func1 ( int *a, int *b ) {
    *a += *b;
    *a += *b; }

void func2 ( int *a, int *b ) {
    *a += 2 * *b; }
```

(*) example taken from “Compuer Systems. A Programmer’s Perspective”, Pearson

FIRST THINGS FIRST

Memory aliasing

An incautious analysis may conclude that a compiler, or even a programmer, should immediately transform `func1()` into `func2()` because, having three less memory references, it should yield to a better assembly code.

However, is it really true that the two functions behave exactly the same way in all possible conditions?

What if `a = b`, i.e. if `a` and `b` points to the same memory location?

Memory aliasing

if `a` and `b` points to the same memory location, and let's say that `*a = 1`:

```
void func1 ( int *a, int *b ) {
    *a += *b;  -> *a and *b now contains 2
    *a += *b;  -> *a and *b now contains 4
}

void func2 ( int *a, int *b ) {
    *a += 2 * *b; -> *a and *b now contains 3
}
```

This condition, i.e. when 2 pointer variables reference the same memory address is called ***memory aliasing*** and is a major performance blocker in those languages that allows pointer arithmetic like C and C++.

Memory aliasing

Focus on the *restrict* qualifier

```
void my_function( double *a, double *b, int n)
{
    for( int i = ; i < n; i++ )
      a[ i ] = s * b[ i - 1 ];
}
```

The compiler can not optimize the access to **a** and **b** because it can not assume that **a** and **b** are pointing to the same memory locations or, in general, that the references will never overlap.
That is called *aliasing*, formally forbidden in FORTRAN: which is the reason why in some cases fortran may compile in faster executables without you paying any attention.

Help your C compiler in doing the best effort, either writing a clean code or using restrict or using `-fstrict-aliasing` `-Wstrict-aliasing` options.

Memory aliasing

Focus on the *restrict* qualifier

```
void my_function( double *restrict a,
                  double *restrict b,
                  int n )
{
        for( int i = ; i < n; i++ )
           a[ i ] = s * b[ i - 1 ];
}
```

Now you’re telling the compiler that the memory regions referenced by a and b will never overlap.
So, it will feel confident in optimizing the memory accesses as much as it can (basically avoiding to re-read locations)

FIRST THINGS FIRST

Focus on the memory aliasing

```
937                             .globl  add_float_array
939                     add_float_array:
947                     # pointers_aliasing_a.c:129:   for ( int i = 0; i < N; i++ )
129:pointers_aliasing_a.c ****       C[ i ] += A[ i ] + B[ i ];
949 0060 85FF                   test    edi, edi        # N
950 0062 0F8E1801               jle     .L36    #,
951 0068 4C8D4110               lea     r8, 16[rcx]     # tmp156,
952 006c 4C8D5610               lea     r10, 16[rsi]    # _31,
953 0070 4C39C6                 cmp     rsi, r8 # C, tmp156
954 0073 8D47FF                 lea     eax, -1[rdi]    # _33,
955 0076 410F93C1               setnb   r9b     #, tmp158
956 007a 4C39D1                 cmp     rcx, r10        # B, _31
957 007d 410F93C0               setnb   r8b     #, tmp160
958 0081 4509C1                 or      r9d, r8d        # tmp161, tmp160
959 0084 4C8D4210               lea     r8, 16[rdx]     # tmp162,
960 0088 4C39C6                 cmp     rsi, r8 # C, tmp162
961 008b 410F93C0               setnb   r8b     #, tmp164
962 008f 4C39D2                 cmp     rdx, r10        # A, _31
963 0092 410F93C2               setnb   r10b    #, tmp166
964 0096 4509D0                 or      r8d, r10d       # tmp167, tmp166
965 0099 4584C1                 test    r9b, r8b        # tmp161, tmp167
966 009c 0F84AE00               je      .L38    #,
967 00a2 83F802                 cmp     eax, 2  # _33,
968 00a5 0F86A500               jbe     .L38    #,
969 00ab 4189F8                 mov     r8d, edi        # bnd.78, N
970 00ae 31C0                   xor     eax, eax        # ivtmp.105
971 00b0 41C1E802               shr     r8d, 2  #,
972 00b4 49C1E004               sal     r8, 4   # _110,

976                     .L39:
980 00c0 0F100402               movups  xmm0, XMMWORD PTR [rdx+rax]     # MEM[base: A_16(D)]
981 00c4 0F100C01               movups  xmm1, XMMWORD PTR [rcx+rax]     # MEM[base: B_17(D)]
984 00c8 0F101406               movups  xmm2, XMMWORD PTR [rsi+rax]     # MEM[base: C_15(D)]
```

Often the compiler is good enough to understand that it could generate 2 different loops:
one for the case in which there is memory overlap and a different one for the case in which there is not.
The second loop is very similar to what it generates if you tell him so through the *restrict* keyword.

Let the compiler do his job

As a general guideline just keep in mind that "optimization" reads

"let the compiler squeeze the maximum from your code"

Compilers are quite good indeed, and have a deep insight on the hardware they are running on.

So, as first, just learn how to :

- write non-obfuscated code
- design a good data structure layout
- design a "good" workflow
- take advantage of the modern out-of-order, super-scalar, multi-core architectures

Write non-obfuscated code

- **write non-obfuscated code**
  → -avoid memory aliasing
  → -make it clear what a variable is used for and when
  → -take care of your loops
  → -keep your conditional branches under control

- design a good data structure layout
- design a “good” workflow
- take advantage of the modern out-of-order, super-scalar, multi-core architectures

FIRST THINGS FIRST

Your data are the red pill

- write non-obfuscated code
- **design a good data structure layout**
  → -be cache-friendly (but oblivious)
  → -what is used together, stays together
  → -be NUMA-conscious
  → -avoid false-sharing in multi-threaded cores
- design a "good" workflow
- take advantage of the modern out-of-order, super-scalar, multi-core architectures

Let it flow

- write non-obfuscated code
- design a good data structure layout
- **design a “good” workflow**
  → -compiler will be able to optimize branches and memory access patterns
  → -prefetching will work better
  → -make it easier to use multi-threading
- take advantage of the modern out-of-order, super-scalar, multi-core architectures

May the force be with you

- write non-obfuscated code
- design a good data structure layout
- design a “good” workflow
- **take advantage of the modern out-of-order, super-scalar, multi-core architectures**
  → -let the compiler exploit pipelining through operation ordering and unloop
  → -let the compiler exploit the vectorization capabilities of CPUs
  → -think task-based, data-driven

that’s all, have fun