Optimization
Preliminaries

Luca Tornatore - I.N.A.F.

DATA SCIENCE & ARTIFICIAL INTELLIGENCE

SCIENTIFIC & DATA-INTENSIVE COMPUTING

2024-2025 @ Università di Trieste

Aim of this Lecture

We start a series of lectures about “optimizing” the code on single-core.

This one is about general concepts and preliminaries on what “optimizing” means, while in the next lectures we will go in some details about the most important concepts and traits.

The next slide presents the general outline of the lectures about Optimization.

Outline of this lecture

Introduction & general concepts

Some preliminary stuff on compiler & memory

High Performance Computing requires, by the name itself, to squeeze the maximum effectiveness from your code on the machine you run it.

“Optimizing” is, obviously, a key step in this process.

However, both “*optimizing*” and “*effectiveness*” have to be defined.

There may be no such thing as an “optimal” code *tout court*

Just try to define “optimal”..

don’t lurk to the next slide..

MEMORY CONSTRAINTS
memory imprint (data & code)

ENERGY CONSTRAINTS

I/O CONSTRAINTS
carefully crafting I/O
(disk, memory, network,..)

"OPTIMIZATION"

TIME CONSTRAINTS
time-to-solution
(worst case? optimal case? "average case"?)

ROBUSTNESS/RESILIENCE
mission-critical

Premature optimization is the root of all evil

D. Knuth

Which means that even if some of the stuff you'll learn may sound cool, your first focus must be in

(i) the **correctness** of the code,
(ii) the **data model**,
(ii) the **algorithm** you choose

A wonderfully optimized code which adopts the wrong algorithm is a non-sense because a better algorithm even if poorly implemented will (almost)always beat the former code for a large enough case.

You'd better start thinking in terms of "***improved***" code.

- You don't want to hurt your code that honestly does its job: a faster code that gives incorrect results is not optimal in *any* way
- You most probably will have to re-use that code after some time. And you will need it to be *readable, maintainable* and *well-designed*.
- *Others* most probably will have to read, understand and re-use that code: a clean, understandable, non-obscure, non-redundant code may not be "improved" in some way, but actually improves the quality of your life.

First steps to consider

To have a **clean code**

READABLE
SIMPLE
COMMENTED
FUNCTIONAL

To **re-design** its fundamental architecture

To **improve** the workflow, the memory imprint, the resource usage, the time-to-solution, …

Dryness

Do neither add **unnecessary** code nor **duplicate** code

DRY: Don't Repeat Yourself

Un-necessary code increases the amount of needed work

- to maintain the code, either debugging or updating it
- to extend its functionalities

Duplicated code increases your *bad technical debt*, that already has a large enough number of sources:

- copy-and-paste programming style
- too much *agile* approach
- hard coding, quick-and-dirty fixes, cargo-cult, sloppiness

Readability

Readability is a pillar of **maintainability.**
Maintainability means that software can be extended, upgraded, debugged, fixed.

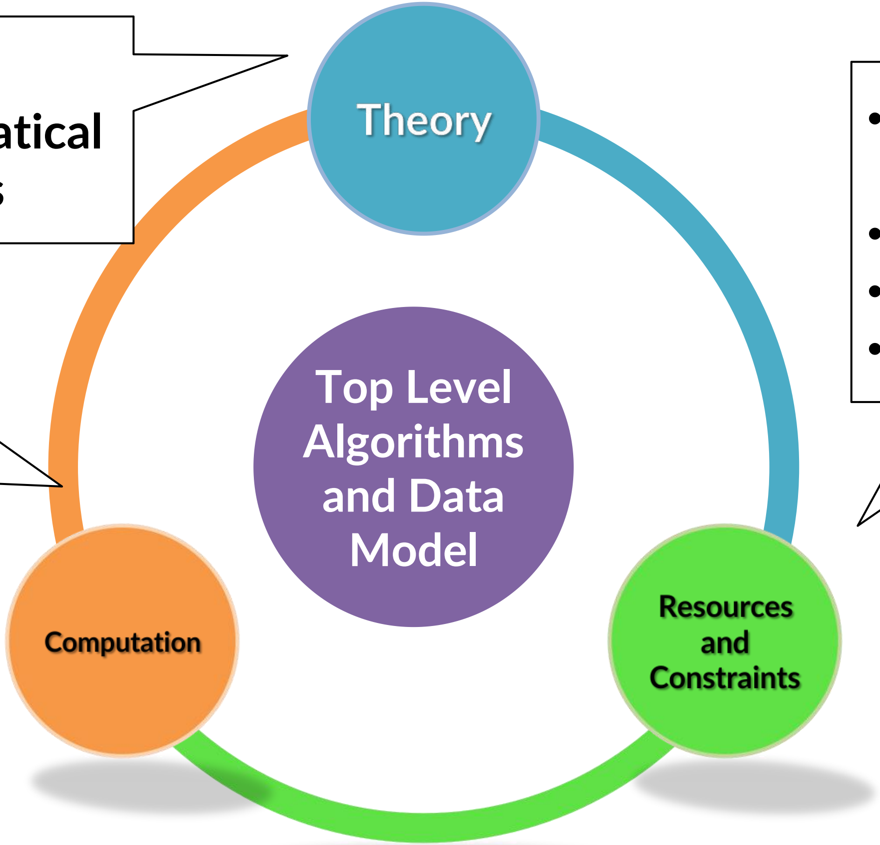
- **Baseline:** *write comments*
- **Advanced:** *WRITE COMMENTS !* (possibly, use doxygen-like tools)
- **X-advanced:** avoid stupid, obvious, useless, confusing comments

Keep in mind: "*the code is the ultimate man page*"

Design

- **Consult literature**
- **Understand mathematical and scientific aspects**

You’re not at the blackboard, but on a digital discrete system. Equations have to *translated*.



- **Concurrency & parallelism**
- **Memory constrs.**
- **I/O constrs.**
- **Energy constrs.**

Design

ARCHITECTURAL DESIGN	Highest abstract picture: - Interacting multi-components - Top-level DATA MODEL			
HIGH-LEVEL DESIGN	Sub-systems and modules Communication strategies		Modules as producers of complex tasks	
DETAILS DESIGN	Implementation details - many algorithms - concurrency	Logical structure of modules	interactions	Very low-level coding and details

ABSTRACTION

CODING DETAIL

TESTING IS part of the design

- Unit test
  separately stress each unit of the code
- Integration test
  stress the integrated behavior of all the units together
- System test

Design

VALIDATION and **VERIFICATION** **ARE** part of the design

- Validation ensures that the code does what it was meant to do
  (all modules are designed accordingly to design specifications)
- Verification ensures that the codes does what it does *correctly*
  (black-box testing against test-cases, …)

Improving the workflow, the memory imprint, the resource usage, the time-to-solution, …

Well, that is the focus of the next lectures

Outline

Some
preliminary stuff
on compiler & memory

| First things first

“Optimization” may be a tag for several different concepts, as we have seen at the beginning of this lecture.

Many concurrent facts and factors must be kept together, and it is quite difficult to give general statements about which ones are more fundamental.

Top-level design plays a key role, as well as the choice of algorithms and eventually the implementation details.

Sometimes, but only sometimes, even a single line of code – for instance a prefetching – may have brilliant (or disastrous) consequences on some limited part of the code.

First things first

1. The first goal is to have a program that delivers the **correct answers** and behave correclty under all conditions.
The code must be as much clear, clean, concise and documented as possible.

2. The first step towards optimization is to adopt the **best-suited algorithms** and data structures. What "best-suited" is must be related to the convolution of the constraints that frame your activity (time-to-solution, energy-to-solution, memory, ...).

3. The second step is that the **source code be optimizable by the compiler**.
Then, you must have a firm understanding of the compilers capabilities and limitations, as well as for your target the architecture.
Understand the best trade-off between portability and "performance" (account for the human effort into the latter).

4. The third step is to get a **data-driven, task-based workflow** which possibly, almost certainly, will be parallel in either distributed- or shared-memory paradigms, or both.

First things first

FIRST THINGS FIRST

5. Profile the code under different conditions (workload / problem size, parallelism, platforms, …) and **spot bottlenecks and inefficiencies.**

6. Apply the techniques for optimization modifying hot-spots in your code to **remove *optimization blockers*** and/or to better expose intrinsic instruction/data parallelisms.

7. IF (needed) GOTO point **1.**

That is not a simple and linear process. Optimizing a code could require several trial-and-error steps and the modern architectures are so complex and their evolution so fast that quite often may it be difficult to perfectly model a code’s performance, and even promising techniques may sometimes fail.

FIRST THINGS FIRST | 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, Techniques & tools", Pearson-Add.Wesley, 2008, 2nd Ed. Chap. 1, Introduction

| Let the compiler do his job

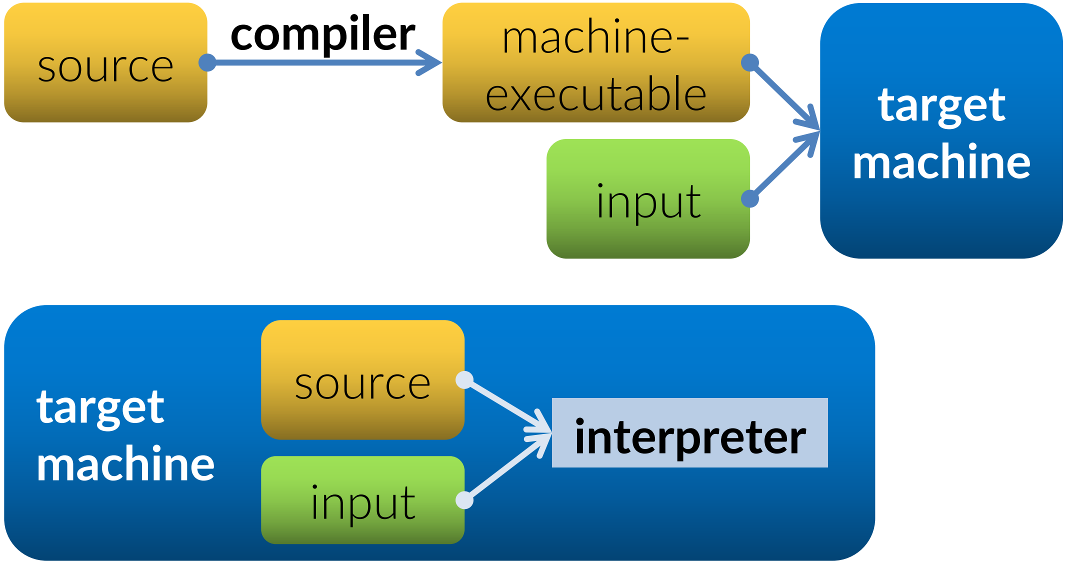
In simple words, a compiler is **a program that translates a program from a source-language into an *equivalent* program in a target-language**, while also signaling possible errors in it (mostly semantic errors and a sub-set of other types of errors).

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

FIRST THINGS FIRST

| 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

Almost always, a compiler is only a stage in 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.

FIRST THINGS FIRST

| Let the compiler do his job

example:

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

C source code

compiler

```
...
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

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

| 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

| 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

FIRST THINGS FIRST

- **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

| 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

FIRST THINGS FIRST

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

| 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

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

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 `gcc` **`-march=native -mtune=native`**, in `icc` **`-xHost`**), 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

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 have seen this in detail in the lecture 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

| 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.

| 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

| 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

`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++.

FIRST THINGS FIRST

| Memory aliasing

```
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.

Focus on the *restrict* qualifier

→ code snippet ::

`memory_aliasing_1/`
`memory_aliasing_2/`

It's time to play a bit

| 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)

Focus on the memory aliasing

What happens on your laptops ?

Memory aliasing not excluded

Memory aliasing excluded explicitly

Memory aliasing excluded explicitly and memory aligned explicitly

```
[ltornatore@hp11 aliasing]$ cat /proc/cpuinfo | grep -m1 "model name"
model name      : Intel(R) Xeon(R) CPU E5-4627 v3 @ 2.60GHz
[ltornatore@hp11 aliasing]$
[ltornatore@hp11 aliasing]$
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_a
aliasing won't be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0295128 (sigma^2: 1.27277e-10)
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_b
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0294134 (sigma^2: 3.97265e-12)
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_c
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.031259 (sigma^2: 1.79355e-10)
[ltornatore@hp11 aliasing]$
[ltornatore@hp11 aliasing]$
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_a.03
aliasing won't be explicitly excluded

 - averaging best 10 of 30 iterations - 0.00923895 (sigma^2: 6.87404e-11)
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_b.03
aliasing will be explicitly excluded
ù
 - averaging best 10 of 30 iterations - 0.00932974 (sigma^2: 3.42719e-11)
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_c.03
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.00629554 (sigma^2: 3.2011e-09)
[ltornatore@hp11 aliasing]$
```

```
[ltornatore@gen10-01 aliasing]$ cat /proc/cpuinfo | grep -m1 "model name"
model name      : Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz
[ltornatore@gen10-01 aliasing]$
[ltornatore@gen10-01 aliasing]$
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_a
aliasing won't be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0294641 (sigma^2: 3.63278e-07)
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_b
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0292034 (sigma^2: 7.0978e-07)
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_c
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0323442 (sigma^2: 9.72394e-08)
[ltornatore@gen10-01 aliasing]$
[ltornatore@gen10-01 aliasing]$
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_a.03
aliasing won't be explicitly excluded

 - averaging best 10 of 30 iterations - 0.00825699 (sigma^2: 5.46725e-10)
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_b.03
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.00830632 (sigma^2: 2.81407e-11)
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_c.03
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.005908 (sigma^2: 1.04317e-09)
[ltornatore@gen10-01 aliasing]$
```

FIRST THINGS FIRST

| Focus on the memory aliasing

What happens on your laptops ?

Without compiler opt. *With compiler opt.*

```
[ltornatore@hp11 aliasing]$ cat /proc/cpuinfo | grep -m1 "model name"
model name      : Intel(R) Xeon(R) CPU E5-4627 v3 @ 2.60GHz
[ltornatore@hp11 aliasing]$
[ltornatore@hp11 aliasing]$
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_a
aliasing won't be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0295128 (sigma^2: 1.27277e-10)
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_b
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0294134 (sigma^2: 3.97265e-12)
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_c
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.031259 (sigma^2: 1.79355e-10)
[ltornatore@hp11 aliasing]$
[ltornatore@hp11 aliasing]$
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_a.03
aliasing won't be explicitly excluded

 - averaging best 10 of 30 iterations - 0.00923895 (sigma^2: 6.87404e-11)
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_b.03
aliasing will be explicitly excluded
ù
 - averaging best 10 of 30 iterations - 0.00932974 (sigma^2: 3.42719e-11)
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_c.03
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.00629554 (sigma^2: 3.2011e-09)
[ltornatore@hp11 aliasing]$
```

```
[ltornatore@gen10-01 aliasing]$ cat /proc/cpuinfo | grep -m1 "model name"
model name      : Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz
[ltornatore@gen10-01 aliasing]$
[ltornatore@gen10-01 aliasing]$
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_a
aliasing won't be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0294641 (sigma^2: 3.63278e-07)
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_b
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0292034 (sigma^2: 7.0978e-07)
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_c
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0323442 (sigma^2: 9.72394e-08)
[ltornatore@gen10-01 aliasing]$
[ltornatore@gen10-01 aliasing]$
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_a.03
aliasing won't be explicitly excluded

 - averaging best 10 of 30 iterations - 0.00825699 (sigma^2: 5.46725e-10)
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_b.03
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.00830632 (sigma^2: 2.81407e-11)
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_c.03
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.005908 (sigma^2: 1.04317e-09)
[ltornatore@gen10-01 aliasing]$
```

| Focus on the memory aliasing

What happens on your laptops ? While the first example goes smoothly, all the other versions, or at least the first two, perform equally.

```
[ltornatore@hp11 aliasing]$ cat /proc/cpuinfo | grep -m1 "model name"
model name      : Intel(R) Xeon(R) CPU E5-4627 v3 @ 2.60GHz
[ltornatore@hp11 aliasing]$
[ltornatore@hp11 aliasing]$
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_a
aliasing won't be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0295128 (sigma^2: 1.27277e-10)
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_b
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0294134 (sigma^2: 3.97265e-12)
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_c
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.031259 (sigma^2: 1.79355e-10)
[ltornatore@hp11 aliasing]$
```

```
[ltornatore@gen10-01 aliasing]$ cat /proc/cpuinfo | grep -m1 "model name"
model name      : Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz
[ltornatore@gen10-01 aliasing]$
[ltornatore@gen10-01 aliasing]$
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_a
aliasing won't be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0294641 (sigma^2: 3.63278e-07)
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_b
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0292034 (sigma^2: 7.0978e-07)
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_c
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0323442 (sigma^2: 9.72394e-08)
[ltornatore@gen10-01 aliasing]$
```

Let’s check the generated assembly to understand this behaviour

```
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.00629554 (sigma^2: 3.2011e-09)
[ltornatore@hp11 aliasing]$
```

```
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.005908 (sigma^2: 1.04317e-09)
[ltornatore@gen10-01 aliasing]$
```

FIRST THINGS FIRST

| Focus on the memory aliasing

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

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

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.

that's all, have fun