

# Optimization Branches

Luca Tornatore - I.N.A.F.



DATA SCIENCE &  
ARTIFICIAL INTELLIGENCE



SCIENTIFIC &  
DATA-INTENSIVE COMPUTING

2024-2025 @ Università di Trieste

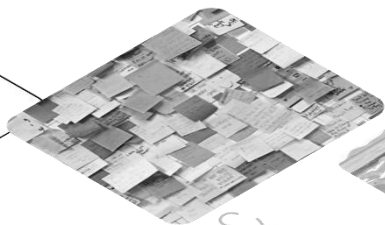


Optimization

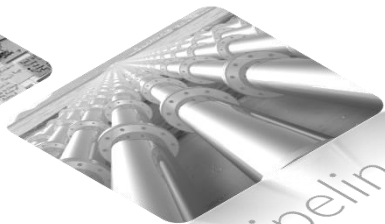
# Outline



First  
things  
first



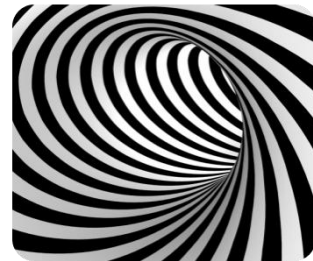
Cache &  
Memory



Pipelines



Branches



Loops



# Outline

- a) Definition of conditional branches
- b) Data-dependent execution flow
- c) Data-dependent data flow
- d) impact of conditional branches on the code efficiency
- e) 4 examples about how to clean/restructure a code
  - 1. conditional branches inside loops
  - 2. unpredictable data streams
  - 3. sorting two arrays
  - 4. filling a matrix



# Don't loose control

Whenever either (i) the sequence of operations that must be executed or (ii) the sequence of data to be processed depends on some condition, i.e. on the outcome of a test performed on some data or result, we have a *conditional execution*.

Modern architecture offer 2 distinct low-level instructions to implement a conditional execution upon a test:

- modifying the *control flow* → **data-dependent execution-flow**
- modifying the *data flow* → **data-dependent data-flow**



Let`s see the conditional execution flow as first.

At machine level, the way to alter the execution flow is through a **jump instruction**, that causes the control to be passed to a different code section.

The jump instruction can be *conditional*, when its execution depends on the outcome of some operation (a test), or *unconditional* if it is not.

*A function call is a jump instruction of particular type, in which we are not interested here.*





**jmp** is the only *unconditional* instruction; it accepts either a *direct* destination (specified by a label) or an *indirect* destination (specified through an address in a register or in memory).

**je**, **jne** and the others, are *conditional* instructions: i.e. their execution depends on a condition.

These instructions access the values stored in the bits of the flag register, a special register where the CPU inscribes some characteristics outcomes of the last arithmetic or logical operation

CF	CARRY FLAG; a carry out of the msb has been generated, signaling an overflow in unsigned op
ZF	ZERO FLAG; the most recent op resulted in a zero
SF	SIGN FLAG; the most recent operation ended in a negative result
DF	OVERFLOW FLAG; a two's-complement overflow, either negative or positive.

This table shows some of the low-level jump instructions routinely available on modern CPUs.

<b>jmp</b>	<i>Label</i> <i>*Operand</i>	direct jump indirect jump
<b>je</b>	<i>Label</i>	jump if equal / zero
<b>jne</b>	<i>Label</i>	jump if not equal / zero
<b>js</b>	<i>Label</i>	jump if negative
<b>jns</b>	<i>Label</i>	jump if not negative
<b>jg</b>	<i>Label</i>	jump if greater
<b>jge</b>	<i>Label</i>	jump if greater or equal
<b>j1</b>	<i>Label</i>	jump if less
...	...	...



# Low-level example: for loop

Let's inspect how simple for cycle translates in assembler:

```
for ( int i = 0; i < 10; i++ )  
    array[i] = 0;
```

Direct memory access at  $i$ -th element of array

AX contains the address of array

.L2:

**mov**  
**add**  
**cmp**  
**jne**

```
DWORD PTR [rax], 0  
rax, 4  
rax, rbp  
.L2
```

The address to be referenced is increased (equivalent to increase the counter  $i$ )

The comparison operator; it amounts to  $rax-rbp$  and sets the flag  $ZF$  (note:  $rbp$  contains the termination value)

jump if not equal, i.e. jumps to .L2 if the  $ZF$  flag is not set



# Low-level example: for loop

This is the linear flow of execution in the cycle body

.L2:

**mov**  
**add**  
**cmp**  
**jne**

DWORD PTR [rax],  
rax, 4  
rax, rbp  
.L2

Until  $rax \leq rbp$  (i.e.  $i < 10$ ) there is a *backward jump* and the body is executed again.

When the jump is *not* executed and the control flow continues afterwards.





# Low-level example: if statement

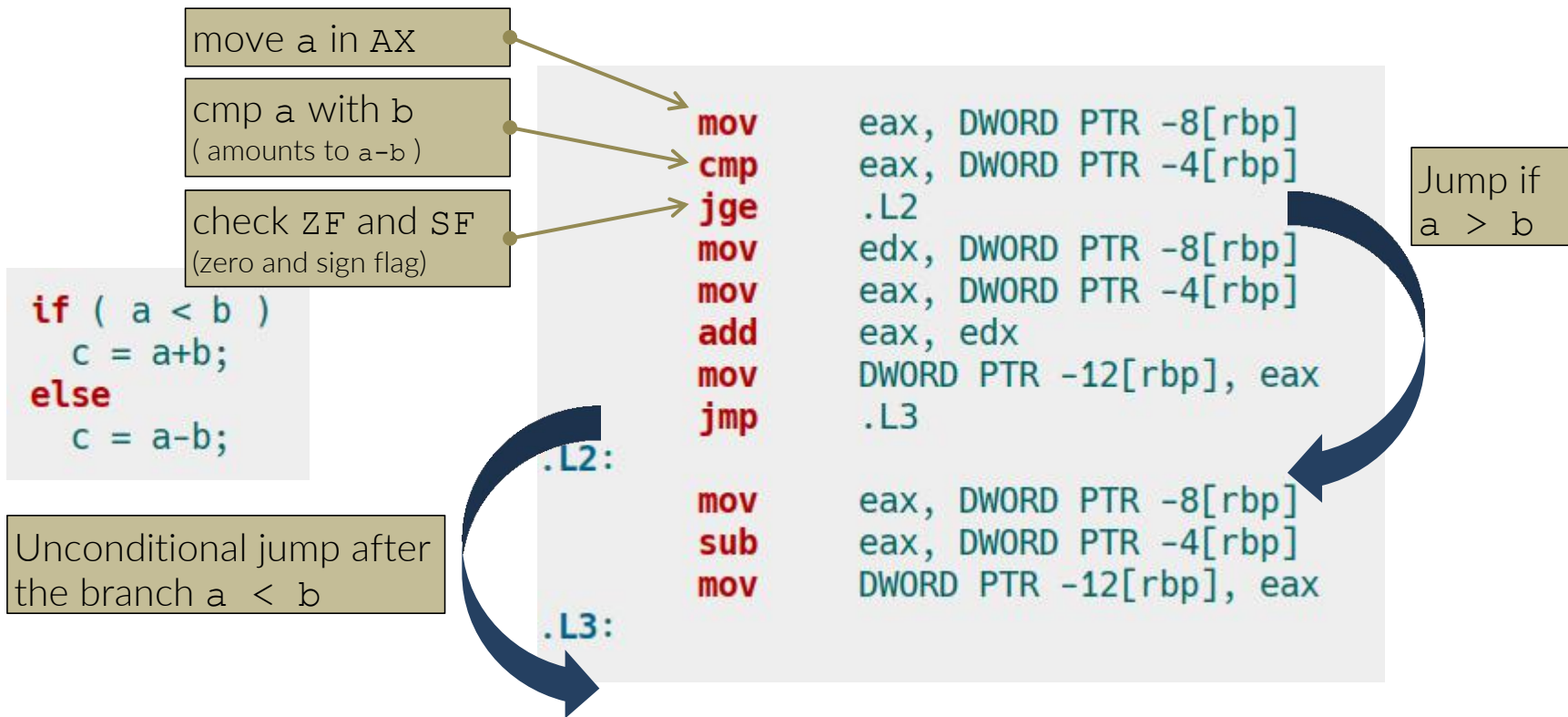
```
if ( a < b )  
    c = a+b;  
else  
    c = a-b;
```

```
    mov     eax, DWORD PTR -8[rbp]  
    cmp     eax, DWORD PTR -4[rbp]  
    jge     .L2  
    mov     edx, DWORD PTR -8[rbp]  
    mov     eax, DWORD PTR -4[rbp]  
    add     eax, edx  
    mov     DWORD PTR -12[rbp], eax  
    jmp     .L3  
.L2:  
    mov     eax, DWORD PTR -8[rbp]  
    sub     eax, DWORD PTR -4[rbp]  
    mov     DWORD PTR -12[rbp], eax  
.L3:
```

*Note: compiled without optimization, though*



# Low-level example: if statement





# Low-level example: if statement

Note:

The **true** branch is the closest to the test condition, while the **false** branch is reached upon a jump.

→ when coding, if possible pay attention to what is most likely to be true, to preserve the **code locality**.

(it is possible to suggest to compiler which branch will most probably be true)

```
c = a+b;  
else  
  c = a-b;
```

```
mov    eax, DWORD PTR -8[rbp]  
cmp    eax, DWORD PTR -4[rbp]  
jge    .L2  
mov    edx, DWORD PTR -8[rbp]  
mov    eax, DWORD PTR -4[rbp]  
add    eax, edx  
mov    DWORD PTR -12[rbp], eax  
jmp    .L3  
  
.L2:  
mov    eax, DWORD PTR -8[rbp]  
sub    eax, DWORD PTR -4[rbp]  
mov    DWORD PTR -12[rbp], eax  
  
.L3:
```



# Conditional data flow

We have seen some details about the conditional transfer of the *control flow* through the simple jump mechanism.

However, that could be quite inefficient in modern CPUs (we'll see more details on that when dealing with the pipelines).

A different mechanism is to **conditionally change the *data flow***, which is the second mechanism for conditional execution that we mentioned.



# Conditional data flow

The conditional transfer of data flow yields a very high performance but is possible only on a small subset of cases;

basically those are when simple values are involved.

A typical example is, for instance, the absolute value of a result, or something alike where a single-valued outcome is expected.

```
if ( a < b )  
    c = a+b;  
else  
    c = a-b;
```

Here  $c$  holds the result of a very simple arithmetic operation between  $a$  and  $b$ .



# Conditional data flow

```
if ( a < b )  
    c = a+b;  
else  
    c = a-b;
```

Compiled with  
`gcc -O3 -march=native`



note:  
ebx = a  
eax = b

```
mov     edx, ebx  
lea     ecx, [rbx+rax]  
sub     edx, eax  
cmp     eax, ebx  
cmovg   edx, ecx
```

A **much** shorter and efficient code!





# Conditional data flow

	eax	ebx	ecx	edx
eax = b, ebx = a	b	a	-	-
<b>mov</b> edx, ebx	b	a	-	a
<b>lea</b> ecx, [rbx+rax]	b	a	a+b	a
<b>sub</b> edx, eax	b	a	a+b	a-b
<b>cmp</b> eax, ebx	compares a and b; sets ZF and CF (zero and carry flag)			
<b>cmovg</b> edx, ecx	move ecx's value to edx if the condition (greater than) is satisfied			

The strategy is as follows:

- (i) reg ecx contains a+b while reg edx contains a-b
- (ii) the conditional move checks the result of cmp a, b ( i.e. the value of a-b )
- (iii) the content of edx is changed into ecx's just in case.

**No jump instructions have been issued. We'll be back to this in few slides**



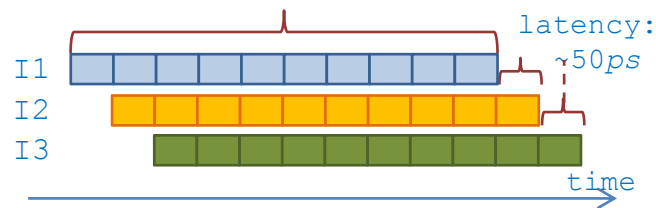
# Outline

- a) Definition of conditional branches
- b) Data-dependent execution flow
- c) Data-dependent data flow
- d) Impact of conditional branches on the code efficiency**
- e) 4 examples about how to clean/restructure a code
  - 1. conditional branches inside loops
  - 2. unpredictable data streams
  - 3. sorting two arrays
  - 4. filling a matrix



# The perils of conditional branches

As we have seen in the lecture about modern architecture, **modern processors achieve great performance thanks to the *pipelines* and *out-of-order execution***, i.e. by decomposing complex instructions in simpler steps and mixing the execution of those sub-steps for different instructions (up to tens of instructions at the same time may be “on-the-fly” in modern CPUs).



*We'll see more about pipelines in forthcoming lectures*

That, however, **requires the pipelines to be always full**; if not so, the toll of great penalties in terms of wasted cpu cycles is to be paid.

To achieve this goal, it is in turn mandatory for the scheduler to be able to **predict** in advance what will be the sequence of instructions to be executed.

**How can that be in a world full of possibilities and branches ?**



# The perils of conditional branches

In order to predict what the execution flow will be, modern cpus feature a **branch predictor**, that is an internal unit of highly sophisticated logic that guesses whether a jump instruction will succeed or not.

Best branch predictors are as good as 95% of accuracy; nonetheless, the branch mis-prediction, or branch miss, determines a huge performance loss. Typical figures for penalty are 10-30 cycles!

That is because the longer the pipeline, the further in the future you have to scrutinize the flow, the more difficult it is and the larger will be the mis-prediction penalty.

*Example*

$$P(n=20, p=0.95) = 0.95^{20}$$

*Let's say we have 140 instructions in flight, and 1 every 7 is a branching instructions. What is the probability that the pipelines shall **not** be flushed with 95% correct branch predictor? And with a 90% one?*

*answer: ~36% and ~12%*



# The perils of conditional branches

In order to predict what the execution flow will be, modern CPUs feature a **branch predictor**, that is an internal hardware structure that predicts if a jump instruction will succeed or not.

Best branch predictors are able to achieve a 90-95% prediction, or branch miss, and the associated penalty are 10-30 cycles!

That is because the longer the execution flow, the more difficult it is to predict. If the flow is linear and perfectly predictable, it generates no jump instructions and the execution flow is linear and perfectly predictable.

## Example

Let's say we have 140 instructions in a pipeline. What is the probability that the pipeline shall not be flushed?

That is why the 2<sup>nd</sup> strategy we have seen, the **conditional change of data flow** is preferable whenever possible and the compiler will try to use it as much as possible:

However, as we said, it applies only on a limited sub-set of cases.



# The perils of conditional branches

Conditional branches should be avoided as much as possible inside loops:

- moving them outside the loops and writing more specialized loops
- performing variables/quantities set-up pre-emptively outside the loops
- using pointers to functions instead of selecting functions inside the loop
- substituting conditional branches with different operations





# Outline

- a) Definition of conditional branches
- b) Data-dependent execution flow
- c) Data-dependent data flow
- d) Impact of conditional branches on the code efficiency
- e) 4 examples about how to clean/restructure a code**
  - 1. conditional branches inside loops
  - 2. unpredictable data streams
  - 3. sorting two arrays
  - 4. filling a matrix



There are basically two possible cases

**(A)** variables tested in the conditions **do not** change during the loop

the easiest case; with little restructuring, or even nothing, you're ok

**(B)** variables tested in the conditions **change** during the loop

the hardest case; possibly, significant work needed



# Clean your loops from branches

## ex 1: Taking decisions before and outside the loop

```
for(i = 1; i < top; i++)  
{  
    if(case1 == 0) {  
        if(case2 == 0) {  
            if(case3 == 0)  
                result += i;  
            else  
                result -= i;  
        }  
        else {  
            if(case3 == 0)  
                result *= i;  
            else  
                result /= i;  
        }  
    }  
}
```

```
else {  
    if(case2 == 0) {  
        if(case3 == 0)  
            result += log10((double)i);  
        else  
            result -= log10((double)i);  
    }  
    else {  
        if(case3 == 0)  
            result *= sin((double)i);  
        else  
            result /= (sin((double)i) +  
                        cos((double)i));  
    }  
}
```



SC0/examples\_on\_branching/  
if\_forest\_inside\_loop



# (Automatic) loop hoisting

When conditional branches are  
*inside* for loop and the conditions  
*do not* depend on the loop iteration

Then the compiler will issue separate  
versions of the for loop

```
int value = get_value_from_the_Universe();  
for(i = 1; i < top; i++) {  
    if ( value > 1 )  
        do_something();  
    else  
        do_something_else(); }
```

```
if ( value > 1 )  
    for(i = 1; i < top; i++)  
        do_something();  
else  
    for(i = 1; i < top; i++)  
        do_something_else(); }
```



# Clean your loops from branches

If you do not trust your compiler to perform the *loop hoisting* for you, you can do it by hands or

- define a specialized function for each case
- **before** and **outside** the loop set a function pointer to the right function

```
void (*func)(double *, int);
```

*<here make func pointing to the right place>*

```
double temp    = 0;  
double result = 0;  
for(i = 1; i < top; i++)  
{  
    func( & temp, i);  
    result = temp;  
}
```



# Clean your loops from branches

However:

- **Using function pointers you may incur in severe overhead due to function calls.**

If the code snippets in different if-branches (or at least the most executed ones) are large/expensive, it might well be irrelevant (in modern CPUs).

- “Unrolling” the if-tree outside the for – then having multiple for loops may be highly unpractical if the branches are big piece of code.

There's no a Swiss-knife recipe.

```
if (case1 == 0) {  
    if (case2 == 0) {  
        if (case3 == 0) {  
            for(i = 1; i < top; i++)  
                result += i;  
        }  
        else  
            for(i = 1; i < top; i++)  
                result -= i;  
    }  
}
```





# Clean your loops from branches

In such cases it is likely much better to use the `switch` construct instead of an `if-forest`, if it is possible to translate the different tests in a test about the values of a single value:

```
switch( case )  
{  
  case A: ...  
    break;  
  case B: ...  
    Break;  
  ...  
  default: ...  
}
```



In fact, the `switch` construct is translated in a static table of code pointers that can be addressed directly:

```
table_of_addr[ #cases ] = { addr_A, addr_B,... , addr_N, addr_default };  
  
if ( case > N )  
  jump to addr_default;  
  
jump to table_of_addr[ case ];
```



## ex 2: code restructuring for an unpredictable datastream

Consider the following code snippet

```
// generate random numbers
for (cc = 0; cc < SIZE; cc++)
    data[cc] = rand() % TOP;

// take action depending on their value
for (ii = 0; ii < SIZE; ii++)
{
    if (data[ii] > PIVOT)
        sum += data[ii];
}
```



SC0/examples\_on\_branching/  
unpredictable\_datastream



# Revise your code

Consider the following code snippet<sup>(\*)</sup>

```
// generate random numbers
for (cc = 0; cc < SIZE; cc++)
    data[cc] = rand() % TOP;

qsort(data, SIZE, sizeof(int), compare);

// take action depending on their value
for (ii = 0; ii < SIZE; ii++)
{
    if (data[ii] > PIVOT)
        sum += data[ii];
}
```

<sup>(\*)</sup>of course, you are adding an overhead due to the sorting routine, so the total running time may be even larger. Moreover, you should have all the values available so that does not work for real-time streamings. However, the point here is to focus on how – in general – it is better to avoid conditionals inside loop, with any possible trick or change in workflow



# Revise your code

You can do even better, without adding operations:

```
// generate random numbers
for (cc = 0; cc < SIZE; cc++)
    data[cc] = rand() % TOP;

// take action depending on their value
for (ii = 0; ii < SIZE; ii++)
{
    t = (data[ii] - PIVOT - 1) >> 31;
    sum += ~t & data[ii];
}
```



# Revise your code

```
luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred
sum is 983597794767, elapsed seconds: 5.40445
luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.wow
sum is 983597794767, elapsed seconds: 2.23186
(in total: 2.44473 seconds)
luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.smart
sum is 983597794767, elapsed seconds: 2.8878
luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% █

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.03
sum is 983597794767, elapsed seconds: 0.660148
luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.wow.03
sum is 983597794767, elapsed seconds: 0.650005
(in total: 0.795181 seconds)
luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.smart.03
sum is 983597794767, elapsed seconds: 0.679286
luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% █
```

-00

-03



# Revise your code

-03

```
luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.03
sum is 983597794767, elapsed seconds: 0.660148
luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.wow.03
sum is 983597794767, elapsed seconds: 0.650005
(in total: 0.795181 seconds)
luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.smart.03
sum is 983597794767, elapsed seconds: 0.679286
luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% █
```

-03

-march=native

```
luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.03n
sum is 983597794767, elapsed seconds: 0.217864
luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.wow.03n
sum is 983597794767, elapsed seconds: 0.215645
(in total: 0.355377 seconds)
luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.smart.03n
sum is 983597794767, elapsed seconds: 0.224288
```





# Revise your code

## What changes in the base version with -O3 ? → conditional move

Modern CPUs have the capability of performing *conditional move*, i.e to execute **concurrently** both branches of a conditional – **if they are “simple enough”** – and to select the right result upon the evaluation of the conditional

perform op1 → res in AX

perform op2 → res in BX

compare

if flag → mov BX in AX

**HOWEVER: loops with conditionals are harder for vectorization !!**  
(also the evaluations can be vectorized but everything may be more complicated)



# Revise your code

## Why the difference between -O3 and -O3 -march=native ?

```
.L8:
movdqu xmm0, XMMWORD PTR [rax]
movdqu xmm6, XMMWORD PTR [rax]
movdqa xmm2, xmm4
add rax, 16
pcmpgtd xmm0, xmm5
pand xmm0, xmm6
pcmpgtd xmm2, xmm0
movdqa xmm3, xmm0
punpckldq xmm3, xmm2
punpckhdq xmm0, xmm2
paddq xmm1, xmm3
paddq xmm1, xmm0
cmp rax, rcx
jne .L8
```

compare **4 integers** at a time using xmmX registers, that are common to x86\_64 architectures.

bytes reshuffling to add one int at a time. These are SSE 128-bits instructions

increase the counter by **4 int**

**12 instructions to process 4 int**

```
.L8:
vmovdqu ymm2, YMMWORD PTR [rax]
add rax, 32
vpcmpgtd ymm0, ymm2, ymm3
vpand ymm0, ymm0, ymm2
vpmovsxdq ymm2, xmm0
vextracti128 xmm0, ymm0, 0x1
vpaddq ymm1, ymm2, ymm1
vpmovsxdq ymm0, xmm0
vpaddq ymm1, ymm0, ymm1
cmp rax, rcx
jne .L8
```

bytes reshuffling inside regs to add one int at a time  
The v prefix tells you these are AVX2 256-bits instr.

compare **8 integers** at a time using ymmX registers. This requires AVX2 that is set on by -march=native for this CPU

increase the counter by **8 int**

**9 instructions to process 8 int**



# Revise your code

A simpler way to transform this code

```
for (ii = 0; ii < SIZE; ii++)  
{  
    if (data[ii] > PIVOT)  
        sum += data[ii];  
}
```

is the following:

```
for (ii = 0; ii < SIZE; ii++)  
    sum += ( data[ii]>PIVOT ) * data[ii];
```



## ex 3: code restructuring for sorting two arrays

You have 2 arrays, A and B, and you want to swap their elements so that

$$A[i] \geq B[i]$$

for all  $i$ .

A straightforward implementation would be:

```
for (i = 0; i < SIZE; i++)  
{  
    if ( A[i] < B[i] )  
    {  
        t = B[i];  
        B[i] = A[i];  
        A[i] = t; }  
}
```



[SCO/examples\\_on\\_branching/  
sort\\_2\\_arrays](#)



# Revise your code

However, that implementation suffers exactly of the same problem we have just discussed.

An alternative way to write the same code, but in a more effective style is:

```
for (i = 0; i < SIZE; i++)  
{  
    int min = A[i] > B[i] ? B[i] : A[i];  
    int max = A[i] >= B[i] ? A[i] : B[i];  
  
    A[i] = max;  
    B[i] = min;  
}
```



# Revise your code

```
for (uint ii = 0; ii < SIZE; ii++)  
{  
    if ( B[ii] > A[ii] )  
    {  
        int t = A[ii];  
        A[ii] = B[ii];  
        B[ii] = t;  
    }  
}
```

**standard**

```
for (uint ii = 0; ii < SIZE; ii++)  
{  
    int register t = -(A[ii]<B[ii]);  
    int register x = A[ii]^B[ii];  
    A[ii] = A[ii]^(x & t);  
    B[ii] = B[ii]^(x & t);  
}
```

**smart2**

```
for (uint ii = 0; ii < SIZE; ii++)  
{  
    int max = (A[ii]>B[ii])? A[ii]:B[ii];  
    int min = (A[ii]>B[ii])? B[ii]:A[ii];  
    A[ii] = max;  
    B[ii] = min;  
}
```

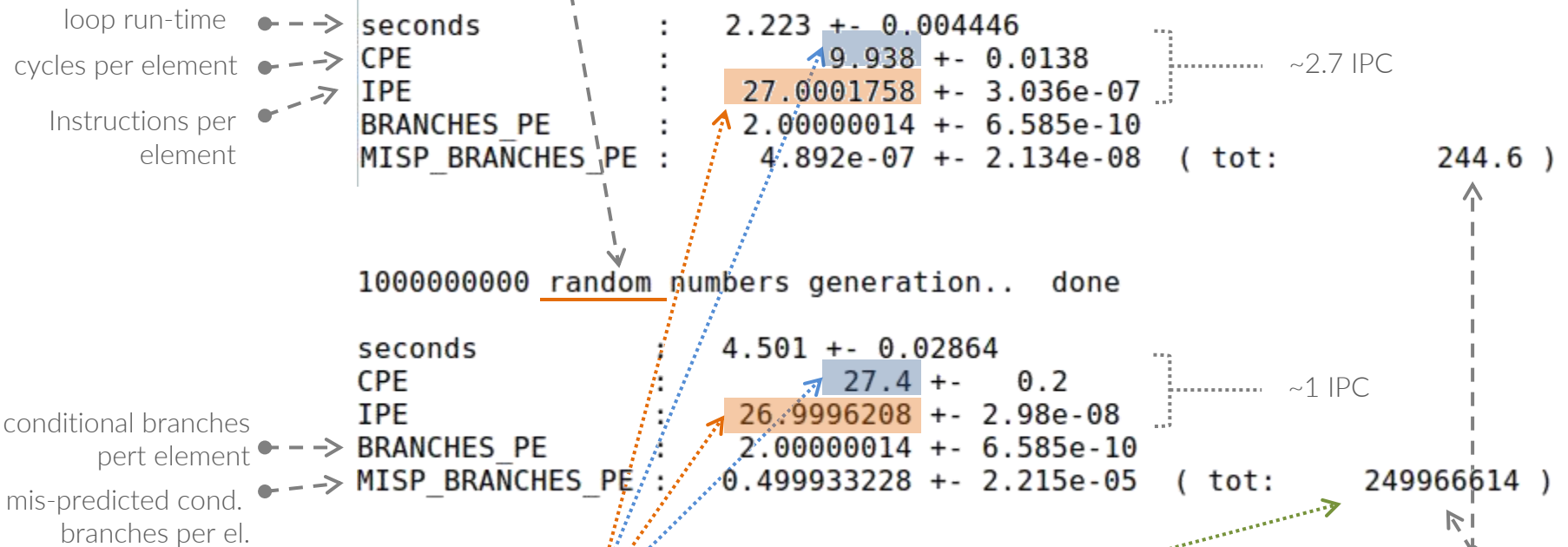
**smart**

```
for (uint ii = 0; ii < SIZE; ii++)  
{  
    int d = A[ii]-B[ii];  
    d &= (d >> 31);  
    A[ii] = A[ii] - d;  
    B[ii] = B[ii] + d;  
}
```

**smart3**

predictable data has a regular pattern easily spotted by the CPU's branch predictor

Standard, -O0



same number of IPE but almost 3 times as many CPE when data pattern is not predictable

in fact, there is 1 mis-predicted branch every 2 (arrays are 500,000,000 long)

total # of mis-predicted conditional branches





# Revise your code

```
seconds      : 1.824 +- 0.05734  
CPE          : 11.12 +- 0.341  
IPE          : 34.00000018 +- 4.344e-08  
BRANCHES_PE : 1.00000014 +- 3.293e-10  
MISP_BRANCHES_PE : 9.94e-07 +- 3.207e-08 ( tot: 497 )
```

smart, -O0

predictable

number of IPE is larger than for standard case, but the CPE is stable !

mis-predicted branches are very few and comparable in both cases

```
seconds      : 1.857 +- 0.001762  
CPE          : 11.32 +- 0.00192  
IPE          : 34.00000018 +- 2.581e-08  
BRANCHES_PE : 1.00000014 +- 3.293e-10  
MISP_BRANCHES_PE : 9.512e-07 +- 4.987e-08 ( tot: 475.6 )
```

~3 IPC

random



# Revise your code

```
seconds      : 1.628 +- 0.01015  
CPE          : 9.993 +- 0.0464  
IPE          : 32.00000017 +- 2.356e-08  
BRANCHES_PE : 1.00000014 +- 3.293e-10  
MISP_BRANCHES_PE : 3.728e-07 +- 8.898e-08 ( tot: 186.4 )
```

smart2, -O0

**predictable**

number of IPE is larger than for  
standard case, but the CPE is stable !

mis-predicted branches are very few  
and comparable in both cases

```
seconds      : 1.688 +- 0.03554  
CPE          : 10.36 +- 0.211  
IPE          : 32.00000017 +- 2.98e-08  
BRANCHES_PE : 1.00000014 +- 3.293e-10  
MISP_BRANCHES_PE : 3.1e-07 +- 5.183e-08 ( tot: 155 )
```

**random**



The standard implementation relies on the ability of your CPU's branch predictor to guess the correct data pattern.

When it is successful, it is **really** so (it exhibits the lowest CPE and IPE).

However, whenever the data pattern is not guessable by the branch predictor things quickly become really weird.

Writing the code differently may make you loosing something in terms of CPE/IPE (with respect to the best possible standard case) but not really in terms of time-to-solution.

And, above all, the code behaviour is **stable** with both predictable and unpredictable patterns.

.L20:

```

mov     eax, DWORD PTR -88[rbp] # ii
lea     rdx, 0[0+rax*4]
mov     rax, QWORD PTR -56[rbp] # B
add     rax, rdx
mov     edx, DWORD PTR [rax]
mov     eax, DWORD PTR -88[rbp] # ii
lea     rcx, 0[0+rax*4]
mov     rax, QWORD PTR -64[rbp] # A
add     rax, rcx
mov     eax, DWORD PTR [rax]

```

```

# branchpred2.c:116:      if ( B[ii] > A[ii] )
cmp     edx, eax
jle     .L19

```

```

# branchpred2.c:118:      int t = A[ii];
mov     eax, DWORD PTR -88[rbp] # ii
lea     rdx, 0[0+rax*4]
mov     rax, QWORD PTR -64[rbp] # A
add     rax, rdx

```

```

# branchpred2.c:118:      int t = A[ii];
mov     eax, DWORD PTR [rax]
mov     DWORD PTR -84[rbp], eax # t

```

```

# branchpred2.c:119:      A[ii] = B[ii];
mov     eax, DWORD PTR -88[rbp] # ii
lea     rdx, 0[0+rax*4]
mov     rax, QWORD PTR -56[rbp] # B
add     rax, rdx

```

```

# branchpred2.c:119:      A[ii] = B[ii];
mov     edx, DWORD PTR -88[rbp] # ii
lea     rcx, 0[0+rdx*4]
mov     rdx, QWORD PTR -64[rbp] # A
add     rdx, rcx

```

```

# branchpred2.c:119:      A[ii] = B[ii];
mov     eax, DWORD PTR [rax]

```

```

# branchpred2.c:119:      A[ii] = B[ii];
mov     DWORD PTR [rdx], eax

```

```

# branchpred2.c:120:      B[ii] = t;
mov     eax, DWORD PTR -88[rbp] # ii
lea     rdx, 0[0+rax*4]
mov     rax, QWORD PTR -56[rbp] # B
add     rdx, rax

```

```

# branchpred2.c:120:      B[ii] = t;
mov     eax, DWORD PTR -84[rbp] # t
mov     DWORD PTR [rdx], eax

```

```

.L19:
# branchpred2.c:112:      for (uint ii = 0; ii < SIZE; ii++)
add     DWORD PTR -88[rbp], 1 # ii,

```

```

.L18:
# branchpred2.c:112:      for (uint ii = 0; ii < SIZE; ii++)
mov     eax, DWORD PTR -88[rbp] # ii
cmp     eax, DWORD PTR -120[rbp] # SIZE
jb      .L20

```

std implementation, -O0

1 cmp instr. with a following jump

2 cmov instr. can use 2 pipelines

.L19:

```

# branchpred2.c:122:      max = A[ii] > B[ii] ? A[ii] : B[ii];
mov     eax, DWORD PTR -92[rbp] # tmp229, ii

```

```

cdq     rdx, 0[0+rax*4]
mov     rax, QWORD PTR -56[rbp] # B
mov     rax, rdx
mov     edx, DWORD PTR [rax]
mov     eax, DWORD PTR -92[rbp] # ii
cdq     rcx, 0[0+rax*4] # _51
mov     rax, QWORD PTR -64[rbp] # A
add     rax, rcx
mov     eax, DWORD PTR [rax]

```

```

# branchpred2.c:122:      max = A[ii] > B[ii] ? A[ii] : B[ii];
cmp     edx, eax
cmovge  eax, edx
mov     DWORD PTR -88[rbp], eax # max

```

```

# branchpred2.c:123:      min = A[ii] < B[ii] ? A[ii] : B[ii];
mov     eax, DWORD PTR -92[rbp] # ii

```

```

cdq     rdx, 0[0+rax*4] # _55
mov     rax, QWORD PTR -56[rbp] # B
add     rax, rdx # _56, _55
mov     edx, DWORD PTR [rax]
# branchpred2.c:123:      min = A[ii] < B[ii] ? A[ii] : B[ii];
mov     eax, DWORD PTR -92[rbp] # ii
cdq     rcx, 0[0+rax*4] # _59
mov     rax, QWORD PTR -64[rbp] # A
add     rax, rcx
mov     eax, DWORD PTR [rax]

```

```

# branchpred2.c:123:      min = A[ii] < B[ii] ? A[ii] : B[ii];
cmp     edx, eax
cmovle  eax, edx

```

```

mov     DWORD PTR -84[rbp], eax # min
# branchpred2.c:131:      A[ii] = max;
mov     eax, DWORD PTR -92[rbp] # ii

```

```

cdq     rdx, 0[0+rax*4]
mov     rax, QWORD PTR -64[rbp] # A
add     rdx, rax

```

```

# branchpred2.c:131:      A[ii] = max;
mov     eax, DWORD PTR -88[rbp] # max
mov     DWORD PTR [rdx], eax

```

```

# branchpred2.c:132:      B[ii] = min;
mov     eax, DWORD PTR -92[rbp] # ii
cdq     rdx, 0[0+rax*4] # _66
mov     rax, QWORD PTR -56[rbp] # B
add     rdx, rax

```

```

# branchpred2.c:132:      B[ii] = min;
mov     eax, DWORD PTR -84[rbp] # min
mov     DWORD PTR [rdx], eax

```

```

# branchpred2.c:107:      for (ii = 0; ii < SIZE; ii++)
add     DWORD PTR -92[rbp], 1 # ii,

```

```

.L18:
# branchpred2.c:107:      for (ii = 0; ii < SIZE; ii++)
mov     eax, DWORD PTR -92[rbp] # ii
cmp     eax, DWORD PTR -104[rbp] # SIZE
jl      .L19

```

smart implementation, -O0





# Performance of different versions with predictable data (gcc)

opt	smart	data	time +- err	CPE +- err	INS +- err	BRE
00	N	P	1.697e+00 +- 2.876e-03	1.041e+01 +- 6.740e-04	2.700e+01 +- 3.650e-08	2.000e+00
00	smart	P	1.859e+00 +- 5.434e-02	1.118e+01 +- 3.320e-01	3.400e+01 +- 3.406e-07	1.000e+00
00	smart2	P	1.815e+00 +- 2.680e-03	1.114e+01 +- 9.030e-03	3.700e+01 +- 3.942e-08	1.000e+00
00	smart3	P	1.628e+00 +- 1.015e-02	9.993e+00 +- 4.640e-02	3.200e+01 +- 2.356e-08	1.000e+00
03	N	P	6.448e-01 +- 3.248e-03	3.578e+00 +- 2.550e-03	8.000e+00 +- 2.849e-08	2.000e+00
03	smart	P	4.548e-01 +- 7.833e-03	2.064e+00 +- 2.040e-02	4.250e+00 +- 3.953e-08	2.500e-01
03	smart2	P	4.506e-01 +- 1.419e-03	2.109e+00 +- 1.650e-02	4.500e+00 +- 3.723e-08	2.500e-01
03	smart3	P	4.332e-01 +- 2.937e-03	1.905e+00 +- 6.180e-03	3.500e+00 +- 2.542e-08	2.500e-01
03n	N	P	3.518e-01 +- 1.096e-02	1.357e+00 +- 1.410e-02	1.125e+00 +- 4.850e-09	2.500e-01
03n	smart	P	3.905e-01 +- 2.258e-03	1.367e+00 +- 1.030e-02	1.125e+00 +- 2.643e-08	1.250e-01
03n	smart2	P	4.114e-01 +- 1.941e-02	1.671e+00 +- 5.650e-02	1.750e+00 +- 3.118e-08	1.250e-01
03n	smart3	P	4.119e-01 +- 3.449e-03	1.523e+00 +- 1.500e-02	1.500e+00 +- 1.613e-09	1.250e-01

The standard implementation (label “N” in the table) exhibits a better behaviour at -O0 considering CPE and above all IPE (label “INS” in the table), whereas run times are comparable among different variants (std, smart, smart2 and smart3).

However, at -O3 its behaviour is the worst one, with CPE larger by ~75% and IPE larger by a factor of ~2. Only with very aggressive optimization the compiler can generate a code comparable with the smartX ones



Branches

## Performance of different versions with non-predictable data (gcc)

opt	smart	data	time +- err	CPE +- err	INS +- err	BRE
00	N	R	4.426e+00 +- 2.966e-02	2.725e+01 +- 1.830e-01	2.700e+01 +- 2.788e-08	2.000e+00
00	smart	R	1.855e+00 +- 1.822e-03	1.139e+01 +- 1.860e-03	3.400e+01 +- 3.161e-08	1.000e+00
00	smart2	R	1.718e+00 +- 5.001e-02	1.055e+01 +- 2.940e-01	3.700e+01 +- 3.942e-08	1.000e+00
00	smart3	R	1.688e+00 +- 3.554e-02	1.036e+01 +- 2.110e-01	3.200e+01 +- 2.980e-08	1.000e+00
03	N	R	2.306e+00 +- 1.549e-02	1.418e+01 +- 8.650e-02	8.000e+00 +- 4.292e-08	2.000e+00
03	smart	R	4.178e-01 +- 3.808e-02	2.138e+00 +- 7.510e-02	4.250e+00 +- 3.838e-08	2.500e-01
03	smart2	R	4.517e-01 +- 1.640e-03	2.098e+00 +- 1.450e-02	4.500e+00 +- 2.644e-08	2.500e-01
03	smart3	R	4.321e-01 +- 2.602e-03	1.910e+00 +- 1.260e-02	3.500e+00 +- 2.710e-08	2.500e-01
03n	N	R	4.178e-01 +- 3.130e-03	1.600e+00 +- 6.140e-03	1.249e+00 +- 2.661e-08	2.500e-01
03n	smart	R	3.918e-01 +- 1.255e-03	1.363e+00 +- 1.260e-02	1.125e+00 +- 2.602e-08	1.250e-01
03n	smart2	R	4.107e-01 +- 1.860e-02	1.653e+00 +- 3.830e-02	1.750e+00 +- 9.429e-09	1.250e-01
03n	smart3	R	4.131e-01 +- 1.929e-03	1.516e+00 +- 9.290e-03	1.500e+00 +- 1.397e-09	1.250e-01

When dealing with random data, the difference between std implementation and the other ones is even more obvious up to -O3, with the CPE being larger by a factor of ~3 and ~4 than in predictable data case at -O0 and -O3 respectively.

With very aggressive optimization the compiler can generate a code comparable with the smartX ones (for this very simple code snippet).



Branches

Performance of different versions  
with non-predictable data

Let's now try with a different compiler...





Branches

## Conditional branches with pgi compiler (v 18.10.0)

opt	smart	data	time +- err	CPE +- err	INS +- err	BRE
00	N	P	9.380e-01 +- 5.053e-03	5.443e+00 +- 4.480e-03	1.700e+01 +- 3.373e-08	2.000e+00
00	N	R	3.263e+00 +- 1.724e-02	1.982e+01 +- 2.920e-02	1.700e+01 +- 7.580e-08	2.000e+00
00	smart	P	1.671e+00 +- 5.070e-02	1.016e+01 +- 3.010e-01	3.300e+01 +- 2.788e-08	3.000e+00
00	smart	R	4.132e+00 +- 4.019e-02	2.525e+01 +- 2.350e-01	3.300e+01 +- 4.344e-08	3.000e+00
00	smart2	P	1.848e+00 +- 3.021e-03	1.126e+01 +- 1.530e-02	3.000e+01 +- 2.471e-08	1.000e+00
00	smart2	R	1.775e+00 +- 6.319e-02	1.082e+01 +- 3.790e-01	3.000e+01 +- 6.909e-08	1.000e+00
00	smart3	P	1.362e+00 +- 6.870e-02	8.283e+00 +- 4.140e-01	2.300e+01 +- 3.573e-08	1.000e+00
00	smart3	R	1.462e+00 +- 2.043e-03	8.873e+00 +- 8.040e-03	2.300e+01 +- 4.012e-08	1.000e+00
03	N	P	5.530e-01 +- 1.608e-03	3.010e+00 +- 6.270e-03	7.500e+00 +- 5.952e-08	2.000e+00
03	N	R	2.343e+00 +- 1.076e-02	1.428e+01 +- 6.810e-02	7.500e+00 +- 2.998e-08	2.000e+00
03	smart	P	3.788e-01 +- 2.156e-03	1.772e+00 +- 4.020e-02	1.875e+00 +- 1.867e-08	2.500e-01
03	smart	R	3.780e-01 +- 1.939e-03	1.769e+00 +- 2.440e-02	1.875e+00 +- 1.073e-08	2.500e-01
03	smart2	P	4.013e-01 +- 1.917e-03	2.210e+00 +- 4.540e-03	3.125e+00 +- 4.165e-09	2.500e-01
03	smart2	R	4.011e-01 +- 1.736e-03	2.212e+00 +- 6.520e-03	3.125e+00 +- 1.863e-09	2.500e-01
03	smart3	P	3.862e-01 +- 4.548e-03	2.080e+00 +- 2.630e-02	2.375e+00 +- 9.701e-09	2.500e-01
03	smart3	R	3.873e-01 +- 2.408e-03	2.067e+00 +- 9.120e-03	2.375e+00 +- 2.734e-08	2.500e-01
03n	N	P	5.538e-01 +- 2.179e-03	3.012e+00 +- 5.910e-03	7.500e+00 +- 1.070e-08	2.000e+00
03n	N	R	2.403e+00 +- 1.996e-02	1.464e+01 +- 1.010e-01	7.500e+00 +- 3.822e-08	2.000e+00
03n	smart	P	3.759e-01 +- 2.942e-03	1.776e+00 +- 2.770e-02	1.875e+00 +- 3.132e-08	2.500e-01
03n	smart	R	3.831e-01 +- 2.355e-03	1.746e+00 +- 1.320e-02	1.875e+00 +- 7.552e-09	2.500e-01
03n	smart2	P	4.047e-01 +- 4.673e-03	2.225e+00 +- 9.760e-03	3.125e+00 +- 1.050e-08	2.500e-01
03n	smart2	R	4.019e-01 +- 1.069e-03	2.214e+00 +- 7.920e-03	3.125e+00 +- 2.281e-09	2.500e-01
03n	smart3	P	3.848e-01 +- 3.987e-03	2.054e+00 +- 1.290e-02	2.375e+00 +- 3.029e-08	2.500e-01
03n	smart3	R	3.836e-01 +- 1.914e-03	2.048e+00 +- 3.770e-03	2.375e+00 +- 2.684e-08	2.500e-01



As we have seen, the `gcc` compiler can generate a code comparable to what it does with `smartX` variants only with very aggressive optimization and using AVX 256-bits instructions.

When random data are used, the standard implementation exhibits a behaviour that is much worse than with data with a predictable pattern, whereas trying not to use conditional branches generates a more stable code.

In this case, `pgi` compiler proves to be less able than `gcc` to generate optimal code, with `-O3n` level (\*) still having a pronounced spike in CPE for random data.

Bottom-line is: **do not take it for granted that you lit up the compiler's optimization and everything will go seamlessly towards a triumph.**

(\*) actually it is `-O4 -fast -tp haswell -Mvect=simd:256`



# Change the point of view

## ex. 4: about the fact that design and simplicity are the best move

Just changing point of view sometimes may help:

```
for ( j = 0; j < N; j++ )  
    for ( i = 0; i < M; i++ )  
    {  
        if ( i > j )  
            matrix[j][i] = 1.0;  
        else if ( i < j )  
            matrix[j][i] = -1.0;  
        else  
            matrix[j][i] = 0.0;  
    }
```

This code initializes a  $N \times M$  matrix so that the elements in top-right triangle are set to **1.0**, the entries in the diagonal  $i=j$  are set to **0.0** and the bottom-left part is set to **-1.0**



# Change the point of view

Can easily be re-written with no conditional evaluations at all:

```
for ( int j = 0; j < N; j++ )
{
    int i;
    for ( i = 0; i < j; i++ )
        matrix[j][i] = -1.0;

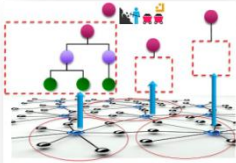
    matrix[j][i] = 0.0;

    for ( i = j+1; i < M; i++ )
        matrix[j][i] = 1.0;
}
```

# A word of caution

It may be really easy to get lost in “optimization”, in hunting every single line wondering why some incredible trick that – you’re convinced – should work, actually does not.

“Optimization” includes also optimizing your effort and your time, so always **remember that the most important ingredients** are **by far**:



The algorithms that you choose



The data model you design



The overall quality, cleanliness and robustness of your code



that's all, have fun



“So long  
and thanks  
for all the fish”