

# Optimalizácia

kuko

27.10.2020

Vybrané partie z dátových štruktúr

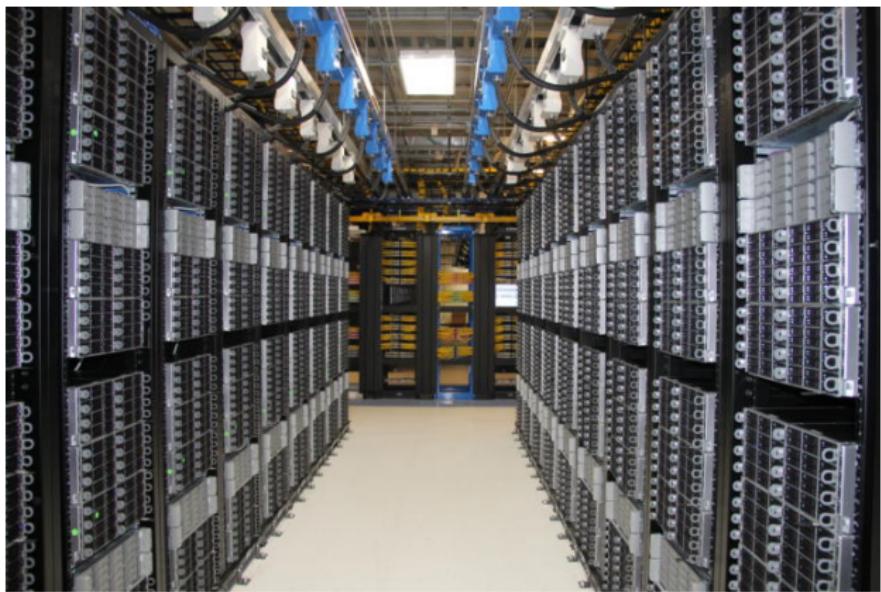
## **Načo optimalizovať?**



**Software is getting  
slower more rapidly than  
hardware becomes faster.**

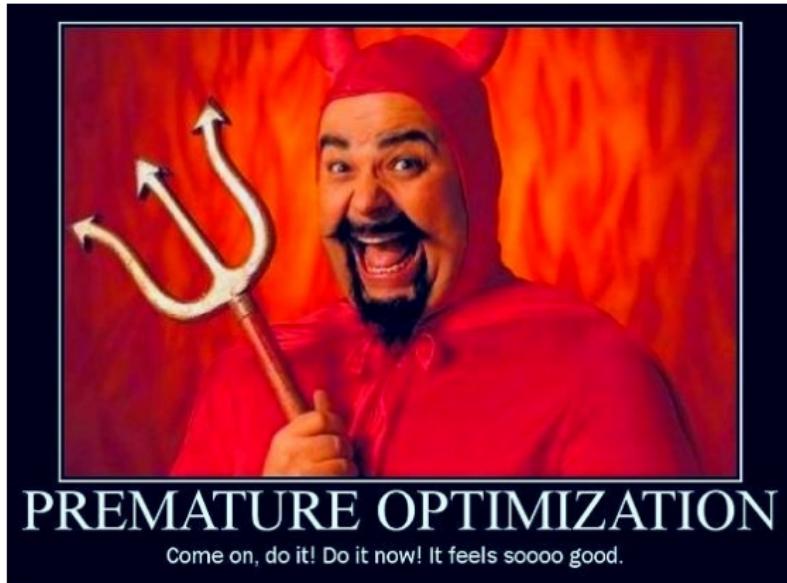
Niklaus Wirth

 [quotationly](#)





## **Tip #1: NEoptimalizujte**



Knuth: Premature optimization is the root of all evil

- korektnosť
- udržiavateľnosť
- efektívnosť

When do I optimize my code?

- 1. Don't***
- 2. Don't yet***
- 3. Profile before optimizing***

Two independent parts A B

Original process



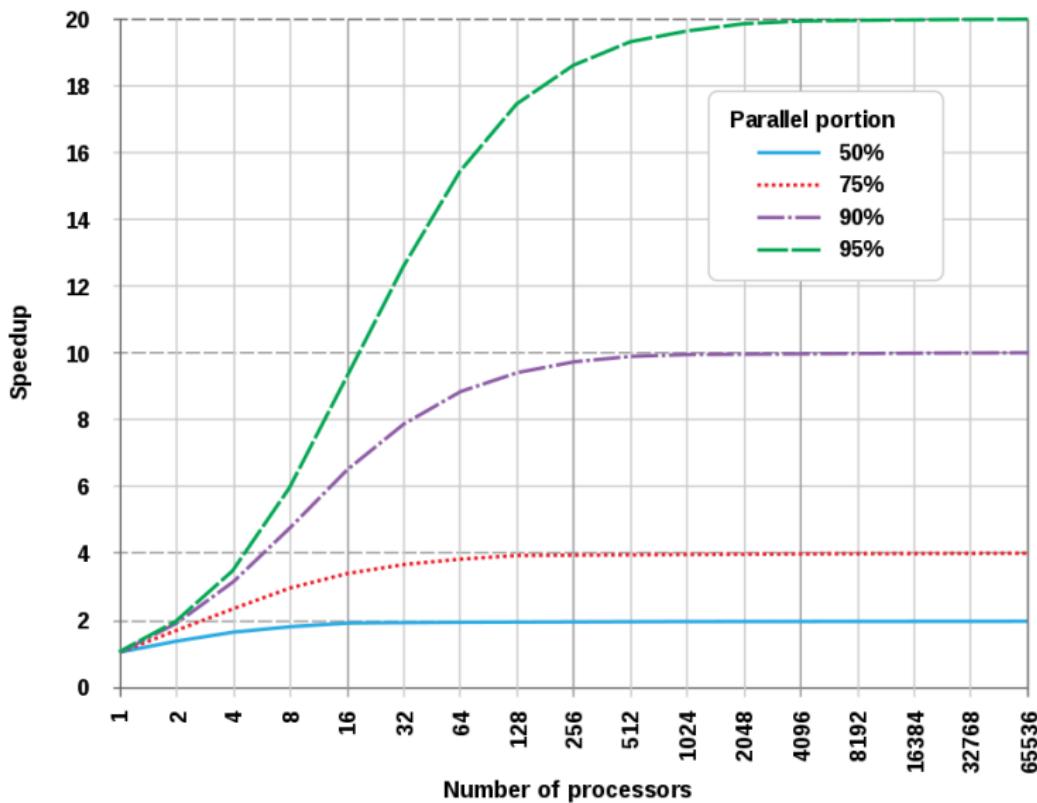
Make B 5x faster



Make A 2x faster



## Amdahl's Law



## CPUS HAVE A HIERARCHICAL CACHE SYSTEM

One cycle on a 3 GHz processor	1	ns	
L1 cache reference	0.5	ns	
Branch mispredict	5	ns	
L2 cache reference	7	ns	14x L1 cache
Mutex lock/unlock	25	ns	
Main memory reference	100	ns	20x L2, 200x L1
Compress 1K bytes with Snappy	3,000	ns	
Send 1K bytes over 1 Gbps network	10,000	ns	0.01 ms
Read 4K randomly from SSD*	150,000	ns	0.15 ms
Read 1 MB sequentially from memory	250,000	ns	0.25 ms
Round trip within same datacenter	500,000	ns	0.5 ms
Read 1 MB sequentially from SSD*	1,000,000	ns	1 ms 4X memory
Disk seek	10,000,000	ns	10 ms 20x datacenter RT
Read 1 MB sequentially from disk	20,000,000	ns	20 ms 80x memory, 20X SSD
Send packet CA->Netherlands->CA	150,000,000	ns	150 ms

<https://gist.github.com/hellerbarde/2843375>

## **Tip #1': Nerobte prácu kompilátora**

The image shows two side-by-side windows from the Compiler Explorer tool. The left window displays a C++ source code snippet for counting neighbors in a 2D vector. The right window shows the generated assembly code for x86-64 architecture using gcc 10.2.

**C++ Source Code:**

```
1 #include <vector>
2 using namespace std;
3
4 int countNeighbours(vector<vector<int>> a, int i, int j){
5     int c=0;
6     for (int k=i-1; k<=i+1; k++)
7         for (int l=j-1; l<=j+1; l++)
8             if (k!=i || l!=j) c += a[k][l];
9     return c;
10 }
```

**Assembly Output:**

```
1 countNeighbours(std::vector<std::vector<int>, std::allocator<int>>& a, int i, int j)
2     sub    esi, 1
3     mov    rdi, QWORD PTR [rdi]
4     sub    edx, 1
5     movsx  rsi, esi
6     movsx  rdx, edx
7     lea    rcx, [rsi+rsi*2]
8     sal    rcx, 3
9     mov    rsi, QWORD PTR [rdi+rcx]
10    mov    eax, DWORD PTR [rsi+rdx*4]
11    add    eax, DWORD PTR [rsi+4+rdx*4]
12    add    eax, DWORD PTR [rsi+8+rdx*4]
13    mov    rsi, QWORD PTR [rdi+24+rcx]
14    mov    rcx, QWORD PTR [rdi+48+rcx]
15    add    eax, DWORD PTR [rsi+rdx*4]
16    add    eax, DWORD PTR [rsi+8+rdx*4]
17    add    eax, DWORD PTR [rcx+rdx*4]
18    add    eax, DWORD PTR [rcx+4+rdx*4]
19    add    eax, DWORD PTR [rcx+8+rdx*4]
20    ret
```

<https://godbolt.org/z/fsnhb9>

https://godbolt.org

# COMPILER EXPLORER

C++ source #1

A ▾  Save/Load

```
1 int f(int num) {  
2     return num * 3;  
3 }
```

x86-64 clang (trunk) (Editor #1, Compiler #1) C++

x86-64 clang (trunk) -O3

A ▾  11010  .LX0:  .text  //  \s+  Int

```
1 f(int):  
2     lea     eax, [rdi + 2*rdi]  
3     ret
```

https://godbolt.org

# COMPILER EXPLORER

C++ source #1

A ▾ Save/Load + Add new... ▾

```
1 int f(int num) {
2     return (num << 1) + num;
3 }
```

x86-64 clang (trunk) (Editor #1, Compiler #1) C++ ▾

x86-64 clang (trunk) ▾ ✓ -O3

A ▾  11010  .LX0:  .text  //  \s+  Int

```
1 f(int):
2     lea      eax, [rdi + 2*rdi]
3     ret
```

https://godbolt.org

# COMPILER EXPLORER

C++ source #1

A ▾ Save/Load + Add new... ▾

```
1 int f(int num) {
2     return num * 34;
3 }
```

x86-64 clang (trunk) (Editor #1, Compiler #1) C++

x86-64 clang (trunk) -O3

A ▾ 11010 .LX0: .text // \s+ Int

```
1 f(int):
2     mov    eax, edi
3     shl    eax, 5
4     lea    eax, [rax + 2*rdi]
5     ret
```

## Ako optimalizovať

## **1. časť: Menej práce**

**Tip #2: Zvoľte správny algoritmus a DŠ**

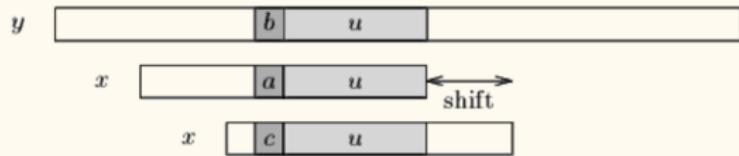
- asymptotická zložitosť
  - ignoruje konštanty
  - správanie pre veľké  $n$
  - horný odhad

- triedenie
- vyhľadávanie
- najbližší menší
- vyhľadávanie v texte
- vyhľadávanie regulárnych výrazov

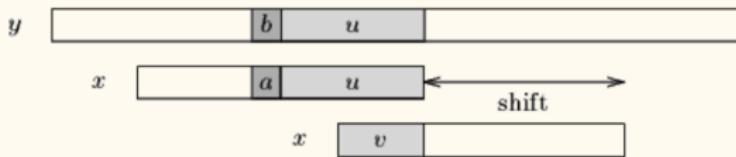
## Vyhľadávanie v texte

TABLE 3 An example of BM algorithm matching process, 4 shifts, 14 comparisons

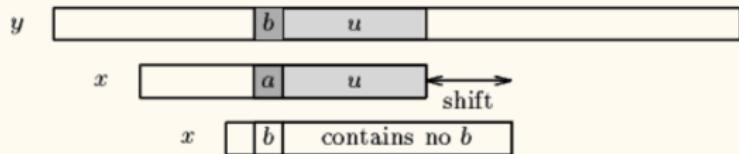
	J	U	L	I	E	T	T	H	O	T	E	L	T	A	N	G	O	F	O	X	T	R	O	T
1	F	O	X	T	R	<span style="color:red">O</span>	<span style="color:blue">T</span>																	
2				F	O	X	T	<span style="color:red">R</span>	<span style="color:red">O</span>	<span style="color:blue">T</span>														
3									<span style="color:blue">F</span>	<span style="color:blue">O</span>	<span style="color:blue">X</span>	<span style="color:blue">T</span>	<span style="color:blue">R</span>	<span style="color:blue">O</span>	<span style="color:red">T</span>									
4									<span style="color:blue">F</span>	<span style="color:blue">O</span>	<span style="color:blue">X</span>	<span style="color:blue">T</span>	<span style="color:blue">R</span>	<span style="color:blue">O</span>	<span style="color:red">T</span>									
5																<span style="color:blue">F</span>	<span style="color:blue">O</span>	<span style="color:blue">X</span>	<span style="color:blue">T</span>	<span style="color:blue">R</span>	<span style="color:blue">O</span>	<span style="color:blue">T</span>		



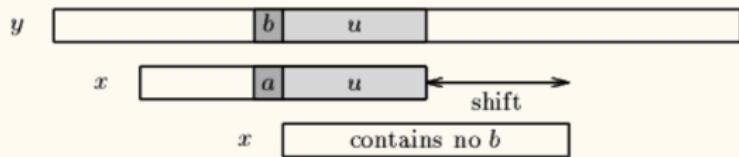
**Figure 13.1.** The good-suffix shift,  $u$  re-occurs preceded by a character  $c$  different from  $a$ .



**Figure 13.2.** The good-suffix shift, only a suffix of  $u$  re-occurs in  $x$ .

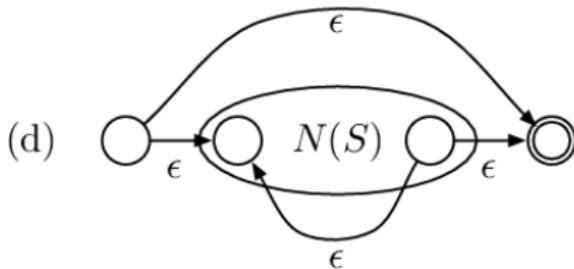
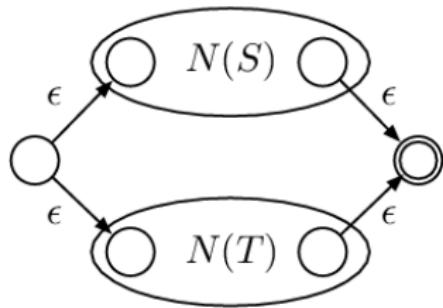
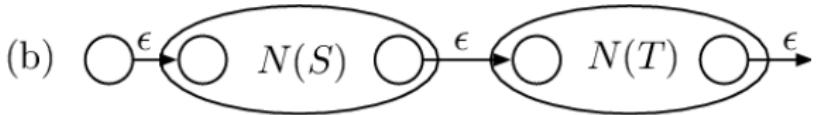
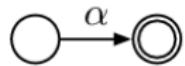


**Figure 13.3.** The bad-character shift,  $a$  occurs in  $x$ .



**Figure 13.4.** The bad-character shift,  $b$  does not occur in  $x$ .

## Vyhľadávanie regulárnych výrazov



text: aaaaa

pattern: a?a?a?a?a?aaaaa

<https://swtch.com/~rsc/regexp/regexp1.html>

```
import re
n = 20
t = n*"a"
p = n*"a?" + t
print "text: ", t
print "pattern: ", p
re.match(p, t).group(0)
```



# Hešovanie

```
string r = "";
FOREACH(it, vec) r += *it;
```

```
string r = boost::algorithm::join(vec, "")
```

## **Tip #3: Robte menej**

```
std::vector<int> v;
for (int i=0; i<N; ++i)
|   v.push_back(rand());
```

```
std::vector<int> v;
v.reserve(N);
for (int i=0; i<N; ++i)
| v.push_back(rand());
```

```
std::vector<int> v(N);
for (int i=0; i<N; ++i)
|   v[i] = rand();
```

### Quick C++ Benchmark

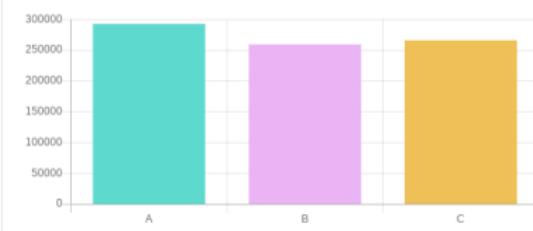
Discover Build Bench! Support Quick Bench Suite ▾ More ▾

```
1 const int N = 10000;
2 static void A(benchmark::State& state) {
3     for (auto _ : state) {
4         std::vector<int> v;
5         for (int i=0; i<N; ++i) v.push_back(rand());
6     }
7 }
8 static void B(benchmark::State& state) {
9     for (auto _ : state) {
10        std::vector<int> v(N);
11        for (int i=0; i<N; ++i) v[i] = rand();
12    }
13 }
14 static void C(benchmark::State& state) {
15     for (auto _ : state) {
16         std::vector<int> v;
17         v.reserve(N);
18         for (int i=0; i<N; ++i) v.push_back(rand());
19     }
20 }
21 BENCHMARK(A);
22 BENCHMARK(B);
23 BENCHMARK(C);
```

compiler = GCC 10.1 ▾ std = c++17 ▾ optim = O3 ▾  
STL = libstdc++(GNU) ▾

Record disassembly  Clear cached results  
 

Charts Assembly



Benchmark	ratio (CPU time / Noop time)
A	~290,000
B	~250,000
C	~260,000

ratio (CPU time / Noop time)  
Lower is faster

<https://quick-bench.com/q/wKD9W3fhG4IE0wUQDA64rbds4jc>

```
#include<iostream>
using namespace std;

int main(int argc, char** argv) {
    for (int i=0; i<10000000; ++i)
        cout << "Hello World!" << endl;
    return 0;
}
```

```
#include<iostream>
using namespace std;

int main(int argc, char** argv) {
    for (int i=0; i<10000000; ++i)
        cout << "Hello World!\n" << flush;
    return 0;
}
```

```
#include<iostream>
using namespace std;

int main(int argc, char** argv) {
    for (int i=0; i<10000000; ++i)
        cout << "Hello World!\n";
    return 0;
}
```

- lineárny find

The screenshot shows the Godbolt Compiler Explorer interface. On the left, there is a code editor window titled "C++" containing the following C++ code:

```
1 #include <vector>
2 using namespace std;
3
4 int find(vector<int> &a, int x)
5     int i=0;
6     for (; i<a.size(); ++i) {
7         if (a[i] == x) break;
8     }
9     return i;
10
```

On the right, there is a assembly output window titled "x86-64 gcc 10.2 -O3". The assembly code generated for the `find` function is:

```
1 find(std::vector<int>, std::allocator<int> &, int):
2     mov    rcx, QWORD PTR [rdi]
3     mov    rdx, QWORD PTR [rdi+8]
4     sub    rdx, rcx
5     sar    rdx, 2
6     je     .L4
7     xor    eax, eax
8     jmp    .L3
9
.L7:
10    lea    r8d, [rax+1]
11    add    rax, 1
12    cmp    rdx, rax
13    je     .L1
14
.L3:
15    mov    r8d, eax
16    cmp    DWORD PTR [rcx+rax*4], esi
17    jne    .L7
18
.L1:
19    mov    eax, r8d
20    ret
21
.L4:
22    xor    r8d, r8d
23    jmp    .L1
```

<https://godbolt.org/z/Y9o9bW>

The image shows two side-by-side code editors. The left editor is for C++ and contains the following code:

```
1 #include <vector>
2 using namespace std;
3
4 int find(vector<int> &a, int x) {
5     int last = a.size()-1;
6     int tmp = a[last];
7     a[last] = x;
8     int i=0;
9     while (a[i] != x) i++;
10    if (i == last && x != tmp) i++;
11    a[last] = tmp;
12    return i;
13 }
```

The right editor shows the generated x86-64 assembly code for the `find` function, compiled with gcc 10.2, using optimization level -O3:

```
1 find(std::vector<int>, std::allocator<int> &, int):
2     mov    rdx, QWORD PTR [rdi]
3     mov    rcx, QWORD PTR [rdi+8]
4     sub    rcx, rdx
5     sar    rcx, 2
6     sub    ecx, 1
7     movsx  rax, ecx
8     lea    rdi, [rdx+rax*4]
9     mov    r9d, DWORD PTR [rdi]
10    mov    DWORD PTR [rdi], esi
11    cmp    esi, DWORD PTR [rdx]
12    je    .L5
13    mov    eax, 1
14 .L3:
15    mov    r8d, eax
16    add    rax, 1
17    cmp    DWORD PTR [rdx-4+rax*4], esi
18    jne    .L3
19 .L2:
20    cmp    ecx, r8d
21    jne    .L4
22    cmp    r9d, esi
23    setne al
24    cmp    al, 1
25    sbb    r8d, -1
26 .L4:
27    mov    DWORD PTR [rdi], r9d
28    mov    eax, r8d
29    ret
30 .L5:
31    xor    r8d, r8d
32    jmp    .L2
```

<https://godbolt.org/z/rbh6d8>

The image shows a debugger interface with two panes. The left pane displays the C++ source code for a function named `find2`. The right pane shows the generated assembly code for the same function, produced by the x86-64 gcc 10.2 compiler with optimization level -O3.

**C++ Source Code:**

```
1 #include <algorithm>
2 #include <vector>
3 using namespace std;
4
5 int find2(vector<int> &a, int x) {
6     return std::find(a.begin(), a.end(), x) - a.begin();
7 }
```

**Assembly Output:**

```
1 find2(std::vector<int>, std::allocator<int> &, int):
2     mov    rax, rdi
3     mov    rdi, QWORD PTR [rdi]
4     mov    r8, QWORD PTR [rax+8]
5     mov    rcx, r8
6     sub    rcx, rdi
7     mov    rax, rcx
8     sar    rcx, 4
9     sar    rax, 2
10    test   rax, rcx
11    jle    .L14
12    sal    rcx, 4
13    mov    rdx, rdi
14    add    rcx, rdi
15    jmp    .L8
16 .L3:   cmp    esi, DWORD PTR [rdx+4]
17    je     .L25
18    cmp    esi, DWORD PTR [rdx+8]
19    je     .L26
20    cmp    esi, DWORD PTR [rdx+12]
21    je     .L27
22    add    rdx, 16
23    cmp    rcx, rdx
24    je     .L28
25
26 .L8:   cmp    esi, DWORD PTR [rdx]
27    jne    .L3
28
29 .L22:  sub    rdx, rdi
30    mov    rax, rdx
31    sar    rax, 2
32
33 .L4:   ret
```

## **Tip #4: Používajte zarážky**

## **Tip #5: Zrýchlite častý/priemerný prípad**

## Príklad 1: UTF8

<b>Number of bytes</b>	<b>Bits for code point</b>	<b>First code point</b>	<b>Last code point</b>	<b>Byte 1</b>	<b>Byte 2</b>	<b>Byte 3</b>	<b>Byte 4</b>
1	7	U+0000	U+007F	0xxxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

```
    while (s[i]) {
        if ((s[i] & 0xc0) != 0x80) j++;
        i++;
    }
    return j;
```

<https://godbolt.org/z/8vbssd>

```
        while (s[i] > 0) {
ascii:
    |     i++;
}
count += i - iBefore;

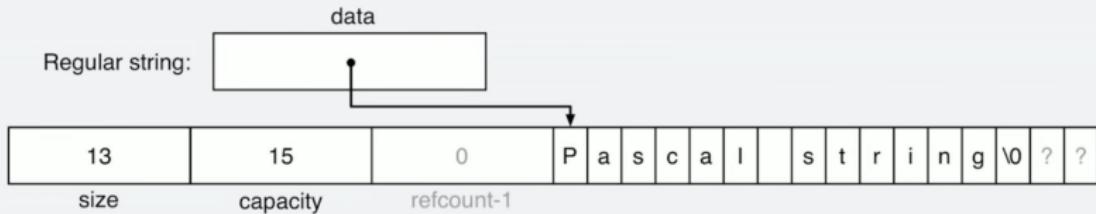
while (s[i]) {
    if (s[i] > 0) {
        iBefore = i;
        goto ascii;
    } else {
        switch (0xF0 & s[i]) {
        case 0xE0:
            i += 3;
            break;
        case 0xF0:
            i += 4;
            break;
        default:
            i += 2;
            break;
        }
    }
    count++;
}
```

<https://godbolt.org/z/1dhKq5>

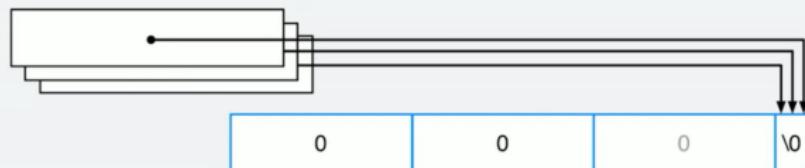
## Príklad 2: Ako funguje string?

```
struct string {  
    int size;  
    int capacity;  
    char * data;  
};
```

## gcc string (version <5)

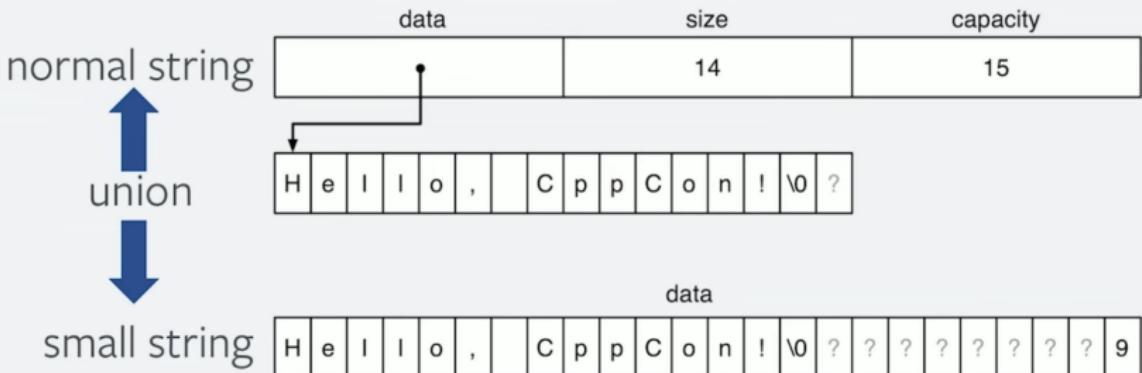


All empty strings:

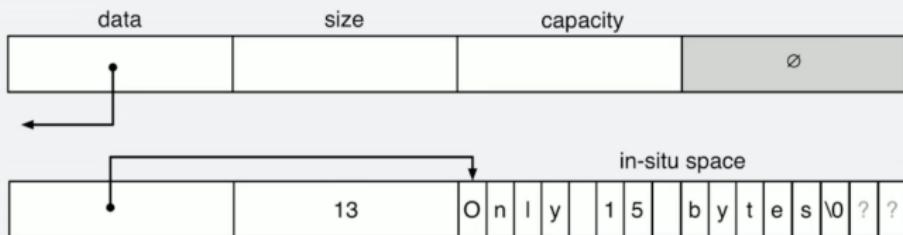


# fbstring

@author Andrei Alexandrescu



## gcc string (version >=5)



- + Has SSO
- + `data()`, `size()` very fast
- + Size, 32, is power of 2
- Only 15-byte capacity
- Move is no longer memcpy
- Size is 33% larger than `fbstring`

- ① NEoptimalizujte, nerobte prácu kompilátora
- ② Zvoľte správny algoritmus a DŠ
- ③ Robte menej
- ④ Používajte zarážky
- ⑤ Zrýchlite častý/priemerný prípad

## **2. časť: Rovnaká práca rýchlejšie**

## CPU'S HAVE A HIERARCHICAL CACHE SYSTEM

One cycle on a 3 GHz processor	1	ns	
L1 cache reference	0.5	ns	
Branch mispredict	5	ns	
L2 cache reference	7	ns	14x L1 cache
Mutex lock/unlock	25	ns	
Main memory reference	100	ns	20x L2, 200x L1
Compress 1K bytes with Snappy	3,000	ns	
Send 1K bytes over 1 Gbps network	10,000	ns	0.01 ms
Read 4K randomly from SSD*	150,000	ns	0.15 ms
Read 1 MB sequentially from memory	250,000	ns	0.25 ms
Round trip within same datacenter	500,000	ns	0.5 ms
Read 1 MB sequentially from SSD*	1,000,000	ns	1 ms 4X memory
Disk seek	10,000,000	ns	10 ms 20x datacenter RT
Read 1 MB sequentially from disk	20,000,000	ns	20 ms 80x memory, 20X SSD
Send packet CA->Netherlands->CA	150,000,000	ns	150 ms

## **Tip #6: Využívajte efektívne cache**

- prechod 2D pola po riadkoch vs. po stĺpcach vs. náhodne
- vector vs. list
- qsort vs. heapsort
- $d$ -árna vs. binárna halda
- flat\_hash\_set vs. unordered\_set
- podielové vs. bloom filtre

## **Tip #7: Bitová a vektorová mágia**

## Subsets of $0, \dots, N - 1$

a subset	{ 0, 3, 5 }
good/bad numbers	0, 1, 2, 3, 4, 5
yes/no bits	1 0 0 1 0 1 <- binary!
the number	$2^{0+} 2^{3+} 2^{5} = 41$

## Bitwise operations

union: bitwise or

intersection: bitwise and

invert mask: bitwise xor

set  $\{i\}$ : bitwise shifts:  $1 \ll i$

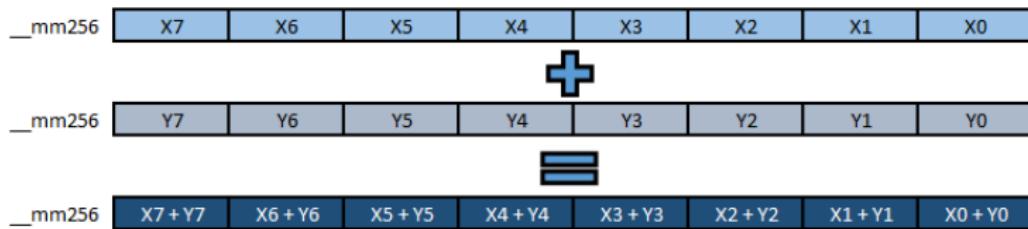
### SSE Data Types (16 XMM Registers)

<code>_m128</code>	Float	Float	Float	Float	4x 32-bit float
<code>_m128d</code>	Double		Double		2x 64-bit double
<code>_m128i</code>	B	B	B	B	16x 8-bit byte
<code>_m128i</code>	short	short	short	short	8x 16-bit short
<code>_m128i</code>	int	int	int	int	4x 32bit integer
<code>_m128i</code>	long long		long long		2x 64bit long
<code>_m128i</code>			doublequadword		1x 128-bit quad

### AVX Data Types (16 YMM Registers)

<code>_mm256</code>	Float	Float	Float	Float	Float	Float	Float	8x 32-bit float
<code>_mm256d</code>	Double		Double		Double		Double	4x 64-bit double
<code>_mm256i</code>	256-bit Integer registers. It behaves similarly to <code>_m128i</code> . Out of scope in AVX, useful on AVX2							

## AVX Operation



<b>Number of bytes</b>	<b>Bits for code point</b>	<b>First code point</b>	<b>Last code point</b>	<b>Byte 1</b>	<b>Byte 2</b>	<b>Byte 3</b>	<b>Byte 4</b>
1	7	U+0000	U+007F	0xxxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

```
while (s[i]) {
    if ((s[i] & 0xc0) != 0x80) j++;
    i++;
}
return j;
```

```
/* Handle complete blocks. */
for ( ; ; s += sizeof(size_t)) {
    /* Grab 4 or 8 bytes of UTF-8 data. */
    u = *(size_t *)(s);

    /* Exit the loop if there are any zero bytes. */
    if ((u - ONEMASK) & (~u) & HIMASK) break;

    /* Count bytes which are NOT the first byte of a character. */
    count += __builtin_popcountll(u & HIMASK & ((~u) << 1));
}
```

<https://godbolt.org/z/Pn7xTb>

<http://www.daemonology.net/blog/2008-06-05-faster-utf8-strlen.html>

0	1	1	1	1	0	1	1	1	1	1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	0	0	0	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

x

0	1	1	1	1	0	1	1	1	1	1	1	1	0	1	0	1	0	1	0	1	0	1	0	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

x

x & 0x80808080

0	1	1	1	0	1	1	1	1	0	1	0	1	0	1	0	0	0	0	0	1	0	0	0	1	1	1		
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0

x

x & 0x80808080

x & 0x80808080 & ((~x) << 1)

0	1	1	1	1	0	1	1	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

**ONEMASK** = 0x01010101

三

CINEMASK = 0x01010101

X - OMENASK

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
0	1	1	1	1	0	1	1	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0	1	1	1

(~x) & 0x80808080

x

ONEMASK = 0x01010101

x - ONEMASK

```
/* Handle complete blocks. */
for ( ; ; s += sizeof(size_t)) {
    /* Prefetch 256 bytes ahead. */
    __builtin_prefetch(&s[256], 0, 0);

    /* Grab 4 or 8 bytes of UTF-8 data. */
    u = *(size_t *)s;

    /* Exit the loop if there are any zero bytes. */
    if ((u - ONEMASK) & (~u) & HIMASK) break;

    /* Count bytes which are NOT the first byte of a character. */
    count += __builtin_popcountll(u & HIMASK & ((~u) << 1));
}
```

<https://godbolt.org/z/Pn7xTb>

```

constexpr size_t ONEMASK = ((size_t)(-1) / 0xFF); // 0x0101010101010101
constexpr size_t HIMASK = ONEMASK * 0x80;           // 0x8000000000000000

static size_t cp_strlen_utf8(const char * _s) {
    const char * s;
    size_t count = 0;
    size_t u;
    unsigned char b;

    /* Handle any initial misaligned bytes. */
    for (s = _s; (uintptr_t)(s) & (sizeof(size_t) - 1); s++) {
        b = *s;

        /* Exit if we hit a zero byte. */
        if (b == '\0') goto done;

        /* Is this byte NOT the first byte of a character? */
        count += (b >> 7) & ((~b) >> 6);
    }

    /* Handle complete blocks. */
    for (; ; s += sizeof(size_t)) {
        /* Prefetch 256 bytes ahead. */
        __builtin_prefetch(&s[256], 0, 0);

        /* Grab 4 or 8 bytes of UTF-8 data. */
        u = *(size_t *)s;

        /* Exit the loop if there are any zero bytes. */
        if ((u - ONEMASK) & (~u) & HIMASK) break;

        /* Count bytes which are NOT the first byte of a character. */
        count += __builtin_popcountl(u & HIMASK & ((~u) << 1));
    }

    /* Take care of any left-over bytes. */
    for (; ; s++) {
        b = *s;
        if (b == '\0') break;
        count += (b >> 7) & ((~b) >> 6);
    }
}

done:
    return ((s - _s) - count);
}

```

<https://godbolt.org/z/Pn7xTb>

```
__m128i ZERO = _mm_set1_epi8(0);
__m128i SND = _mm_set1_epi8(0x40);

for ( ; ; s += sizeof(__m128i)) {
    /* Prefetch 2048 bytes ahead. */
    __builtin_prefetch(&s[2048], 0, 0);

    u = *(__m128i *)s;
    if (_mm_movemask_epi8(_mm_cmpeq_epi8(u, ZERO))) break;

    __m128i snd = _mm_andnot_si128(u, SND);
    __m128i res = _mm_and_si128(_mm_add_epi8(snd, snd), u);
    count += __builtin_popcountll(_mm_movemask_epi8(res));
}
```

# Popcount

0	1	0	0	1	0	1	1	1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

x

0	1	0	0	1	0	1	1	1	0	0	0	0	0	0	0	0
0	1	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1

x

(x-1)

0	1	0	0	1	0	1	1	1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

x

0	1	0	0	1	0	1	1	0	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(x-1)

0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

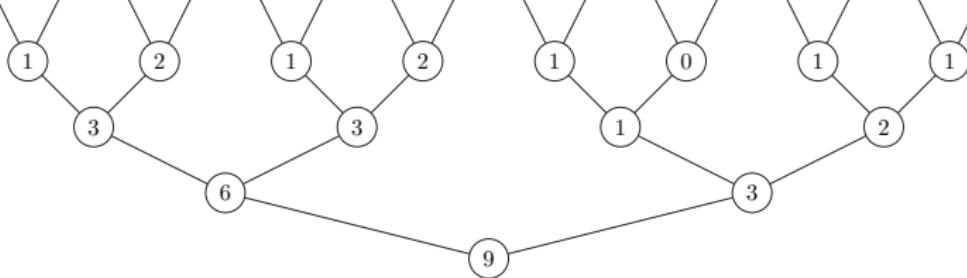
x & (x-1)

```
int count(uint64_t x) {
    int v = 0;
    while(x != 0) {
        x &= x - 1;
        v++;
    }
    return v;
}
```

**FIGURE 4.** The Wegner function in C.

0	1	1	1	1	0	1	1	1	0	0	0	1	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

x



0	1	1	1	0	1	1	1	0	0	0	1	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

x

0	1	1	1	0	1	1	1	0	0	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---

x

0	1	1	1	1	0	1	1	1	0	0	0	1	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

x

0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(x & 0x5555)

0	0	1	0	1	0	1	0	1	0	0	0	1	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(x & 0xaaaa)

0	1	1	1	1	0	1	1	1	0	0	0	1	0	1	0
<b>x</b>															
0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0
(x & 0x5555)															
0	0	0	1	0	1	0	1	0	1	0	0	0	1	0	1
(x & 0xaaaa) >> 1															

0	1	1	1	1	0	1	1	1	0	0	0	1	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

x

0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(x & 0x5555)

0	0	0	1	0	1	0	1	0	1	0	0	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(x & 0aaaa) >> 1

0	1	1	0	0	1	1	0	0	1	0	0	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(x & 0x5555) + ((x & 0aaaa) >> 1)

0	1	1	1	1	0	1	1	1	0	0	0	1	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

x

0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(x & 0x5555)

0	0	0	1	0	1	0	1	0	1	0	0	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(x & 0aaaa) >> 1

1	2	1	2	1	0	1	1
---	---	---	---	---	---	---	---

(x & 0x5555) + ((x & 0aaaa) >> 1)

1	2	1	2	1	0	1	1
---	---	---	---	---	---	---	---

x



1	2	1	2	1	0	1	1
---	---	---	---	---	---	---	---

x

0	0	1	0	0	0	1	0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(x & 0x3333)

0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(x & 0xcccc)

1	2	1	2	1	0	1	1	x
0	0	1	0	0	1	0	0	(x & 0x3333)
0	0	0	1	0	0	1	0	(x & 0xcccc) >> 2

i	2	i	2	i	0	i	1
---	---	---	---	---	---	---	---

x

0	0	1	0	0	1	0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---

(x & 0x3333)

0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(x & 0xcccc) >> 2

0	0	1	1	0	0	1	1	0	0	0	1	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(x & 0x3333) + ((x & 0xcccc) >> 2)

i	2	i	2	i	0	i	1
---	---	---	---	---	---	---	---

x

0	0	1	0	0	1	0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---

(x & 0x3333)

0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(x & 0xcccc) >> 2

3	3	1	2
---	---	---	---

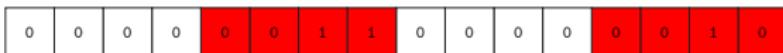
(x & 0x3333) + ((x & 0xcccc) >> 2)

3	3	1	2
---	---	---	---

x



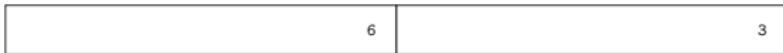
x



(x & 0x0f0f)



(x & 0xf0f0)



(x & 0x0f0f) + ((x & 0xf0f0) >> 4)

	6		3	x
0	0	0	0	(x & 0xffff)
0	0	0	0	(x & 0xff00)
	9			(x & 0xffff) + ((x & 0xff00) >> 8)

```
uint64_t c1 = UINT64_C(0x5555555555555555);
uint64_t c2 = UINT64_C(0x3333333333333333);
uint64_t c4 = UINT64_C(0xF0F0F0F0F0F0F0F0);
uint64_t c8 = UINT64_C(0x00FF00FF00FF00FF);
uint64_t c16 = UINT64_C(0x0000FFFF0000FFFF);
uint64_t c32 = UINT64_C(0x00000000FFFFFF);

uint64_t count(uint64_t x) {
    x = (x & c1) + ((x >> 1) & c1);
    x = (x & c2) + ((x >> 2) & c2);
    x = (x & c4) + ((x >> 4) & c4);
    x = (x & c8) + ((x >> 8) & c8);
    x = (x & c16) + ((x >> 16)) & c16;
    return (x & c32) + ((x >> 32) & c32);
}
```

```
uint64_t c1    = UINT64_C(0x5555555555555555);
uint64_t c2    = UINT64_C(0x3333333333333333);
uint64_t c4    = UINT64_C(0x0F0F0F0F0F0F0F0F);

uint64_t count(uint64_t x) {
    x -= (x >> 1) & c1;
    x = ((x >> 2) & c2) + (x & c2);
    x = (x + (x >> 4)) & c4;
    x *= UINT64_C(0x0101010101010101);
    return x >> 56;
}
```

**FIGURE 3.** The Wilkes-Wheeler-Gill function in C

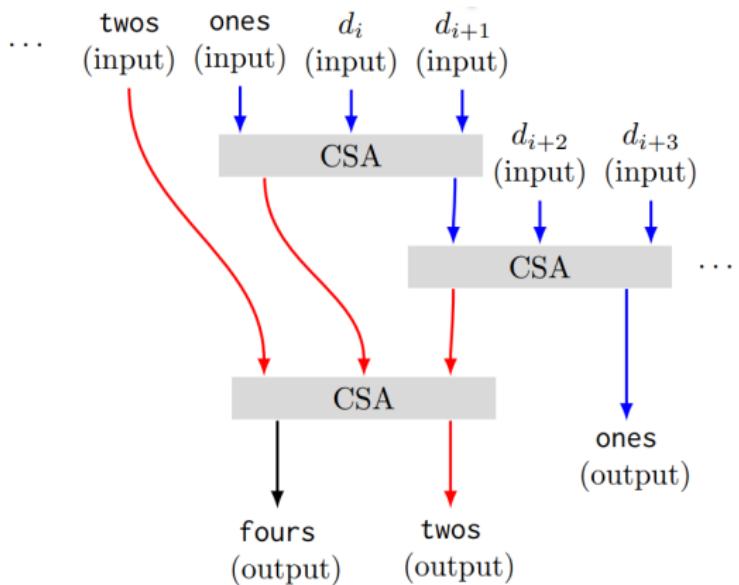
$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	$x_8$	$x_9$	$x_{10}$	$x_{11}$	$x_{12}$
$\downarrow$ po 2											
$\downarrow$ po 4											
$\downarrow$ po 8											
$\downarrow$ po 16											
$\downarrow$ po 32											
$\downarrow$ po 64 +											

$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	$x_8$	$x_9$	$x_{10}$	$x_{11}$	$x_{12}$
$\downarrow$ po 2											
$\downarrow$ po 4											
$\downarrow$ po 8											
$\downarrow$ po 16											
$\downarrow$ po 32											
$\downarrow$ po 64 +											

$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	$x_8$	$x_9$	$x_{10}$	$x_{11}$	$x_{12}$
$\downarrow$ po 2											
po 4 +											
$\downarrow$ po 8			$\downarrow$ po 8								
			+			+		+		+	
											$\downarrow$ po 16
											$\downarrow$ po 32
											$\downarrow$ po 64

**TABLE 1.** Sum of three bits  $a + b + c$ . We use  $\oplus$  for XOR,  $\wedge$  for AND and  $\vee$  for OR.

$a$	$b$	$c$	$a + b + c$	$(a \oplus b) \oplus c$	$(a \wedge b) \vee ((a \oplus b) \wedge c)$
0	0	0	0	0	0
0	0	1	1	1	0
0	1	0	1	1	0
1	0	0	1	1	0
0	1	1	2	0	1
1	0	1	2	0	1
1	1	0	2	0	1
1	1	1	3	1	1



- ① NEoptimalizujte, nerobte prácu komplátora
- ② Zvoľte správny algoritmus a DŠ
- ③ Robte menej
- ④ Používajte zarázky
- ⑤ Zrýchlite častý/priemerný prípad
- ⑥ Využívajte efektívne cache
- ⑦ Bitová a vektorová mágia