

# Přesnost a rychlost výpočtu

Jan Faigl

Katedra počítačů

Fakulta elektrotechnická

České vysoké učení technické v Praze

Přednáška 12

**B0B36PRP – Procedurální programování**

# Přehled témat

- Část 1 – Přesnost výpočtu

Přesnost výpočtu

- Část 2 – Rychlost výpočtu (programu)

Maticové násobení

Rychlost výpočtu

Paralelní výpočet

- Část 3 – Implementace domácích úkolů

Domácí úkol HW10B

# Část I

## Část 1 – Přesnost výpočtu

## Přesnost výpočtu - Příklad součtu dvou čísel

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      double a = 1e+10;
6      double b = 1e-10;
7
8      printf("a  : %24.12lf\n", a);
9      printf("b  : %24.12lf\n", b);
10     printf("a+b: %24.12lf\n", a + b);
11
12     return 0;
13 }
14
15 clang sum.c && ./a.out
16 a  : 10000000000.000000000000
17 b  :                0.000000000100
18 a+b: 10000000000.000000000000
```

lec12/sum.c

Připomínka z 6. přednášky

## Přesnost výpočtu - Příklad dělení dvou čísel

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      const int number = 100;
6      double dV = 0.0;
7      float fV = 0.0f;
8
9      for (int i = 0; i < number; ++i) {
10         dV += 1.0 / 10.0;
11         fV += 1.0 / 10.0;
12     }
13
14     printf("double value: %lf ", dV);
15     printf(" float value: %lf ", fV);
16
17     return 0;
18 }
19
20 clang division.c && ./a.out
21 double value: 10.000000 float value: 10.000002
```

lec12/division.c

Přípomínka z 6. přednášky

## Přesnost výpočtu - strojová přesnost

- Strojová přesnost  $\epsilon_m$  - nejmenší desetinné číslo, které přičtením k 1.0 dává výsledek různý od 1, pro  $|v| < \epsilon_m$ , platí

$$v + 1.0 == 1.0.$$

*Symbol == odpovídá porovnání dvou hodnot (test na ekvivalenci).*

- Zaokrouhlovací chyba - nejméně  $\epsilon_m$ .
- Přesnost výpočtu - aditivní chyba roste s počtem operací v řádu  $\sqrt{N} \cdot \epsilon_m$ .
  - Často se však kumuluje preferabilně v jedno směru v řádu  $N \cdot \epsilon_m$ .

## Zdroje a typy chyby

- Chyby matematického modelu - matematická aproximace fyzikální situace.
  - Chyby vstupních dat.
  - Chyby numerické metody.
  - Chyby zaokrouhlovací.
- 
- Absolutní chyba aproximace  
 $E(x) = \hat{x} - x$ ,  $\hat{x}$  přesná hodnota,  $x$  aproximace.
  - Relativní chyba  $RE(x) = \frac{\hat{x} - x}{x}$ .

## Podmíněnost numerických úloh

- Podmíněnost úlohy  $C_p = \frac{\text{relativní chyba výstupních údajů}}{\text{relativní chyba vstupních údajů}}$ .
- Dobře podmíněná úloha  $C_p \approx 1$ .
- Výpočet je dobře podmíněný, je-li málo citlivý na poruchy ve vstupních datech.
- Numericky stabilní výpočet - vliv zaokrouhlovacích chyb na výsledek je malý.
- Výpočet je stabilní, je-li dobře podmíněný a numericky stabilní.



## Možnosti zvýšení přesnosti

- Reprezentace racionálních čísel - podíl dvou celočíselných hodnot, např. *Homogenní souřadnice*.
- „Libovolná přesnost“ - speciální knihovny, např. `gmp` až do výše volné paměti.  
<https://gmplib.org/manual/index>
  - Souřadnice  $x,y$  - 7511164176768 346868669952 3739567104  $\sim$  2008,57; 92,76.

## Součin dvou velkých čísel knihovnou gmp - 1/2

- V HW04B je uveden příklad  $(995663 \cdot 995669)^8$  jako prvočíselný rozklad čísla  
 932865073719992059629773513614789388266580305083920591925740371392254317064584855785088915745761.  
<https://cw.fel.cvut.cz/wiki/courses/b0b36prp/hw/hw04>
- Použijme knihovnu gmp pro mocninu a součin dvou čísel, `#include<gmp.h>`.  
<https://gmplib.org/>
  - Typ celých čísel `mpz_t`, pomocné funkce `mpz_init_set_str()`, `mpz_init()`, `gmp_printf()` a `mpz_clears()` a operace `mpz_pow_ui()` a `mpz_mul()`.  
 Mocnina `unsigned integer` a násobení - multiplication.
- Knihovna nemusí být součástí operačního systému, proto může být nutné specifikovat cestu k hlavičkovému souboru a vlastní knihovně (`-lgmp`).
  - Můžeme zadat cestu ručně při kompilaci (nebo do `Makefile`).
  - Alternativně můžeme použít nástroj `pkg-config` (nebo `pkgconf`).  
<https://www.freedesktop.org/wiki/Software/pkg-config/>  
<http://pkgconf.org/>
- Argumenty pro překlad (`CFLAGS`).
 

```
$ pkgconf --cflags gmp
-I/usr/local/include
```
- Argumenty pro linkování (`LDFLAGS`).
 

```
$ pkgconf --libs gmp
-L/usr/local/lib -lgmp
```

## Součin dvou velkých čísel knihovnou gmp - 2/2

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include <gmp.h>
5
6  const char* resultSrc =
7  "932865073719992059629773513614789388266580305083"
8  "920591925740371392254317064584855785088915745761";
9
10 int main(int argc, char *argv[])
11 {
12     int ret = EXIT_SUCCESS;
13     mpz_t n1, n2, result;
14     mpz_init_set_str(n1, "995663", 10);
15     mpz_init_set_str(n2, "995669", 10);
16     mpz_init(result);
17
18     gmp_printf("n1: %Zd\n", n1);
19     gmp_printf("n2: %Zd\n", n2);
20
21     gmp_printf("%Zd~%d x %Zd~%d\n\n", n1, 8, n2, 8);
22
23     mpz_pow_ui(n1, n1, 8);
24     mpz_pow_ui(n2, n2, 8);
25
26     gmp_printf("%Zd x %Zd\n\n", n1, n2);
27
28     mpz_mul(result, n1, n2);
29     gmp_printf("%Zd\n\n", result);
30
31     printf("Result from HW04\n%s\n", resultSrc);
32
33     mpz_clears(n1, n2, result, NULL);
34     return ret;
35 }

```

```

./demo-gmp-mpz
n1: 995663
n2: 995669
995663^8 x 995669^8

```

```

965826124294607867982699926255695296863400309121 x
965872686868261151537037082260231566481047775841
93286507371999205962977351361478938826658030508392059
1925740371392254317064584855785088915745761

```

```

Result from HW04
93286507371999205962977351361478938826658030508392059
1925740371392254317064584855785088915745761

```

## Racionální čísla knihovny gmp - 1/3

- Libovolné přesnosti např. souřadnic v rovině jako výsledek operací výpočetní geometrie můžeme reprezentovat jako podíl dvou (libovolně velkých) celých čísel.
  - Souřadnice  $x, y$  - 7511164176768 346868669952 3739567104  $\sim$  2008,57; 92,76.
- Knihovna gmp k tomuto účelu poskytuje typ `mpq_t`, kromě typu necelého čísla `mpf_t`, který využijeme pro převod `mpq_t` na celé číslo typu `double`.

```
double mpq2d(const mpq_t *op)
{
    double ret;
    mpf_t v;
    mpf_init(v);
    mpf_set_q(v, *op);
    ret = mpf_get_d(v);
    mpf_clear(v);
    return ret;
}
```

lec12/gmp/demo-gmp-mpq.c

## Racionální čísla knihovny gmp - 2/3

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include <gmp.h>
5
6  double mpq2d(const mpq_t *op);
7
8  int main(int argc, char *argv[])
9  {
10     int ret = EXIT_SUCCESS;
11
12     unsigned long xl = 75111641767681;
13     unsigned long yl = 3468686699521;
14     unsigned long denl = 37395671041;
15     const unsigned int digits = 21;
16
17     double xd = 1. * xl;
18     double yd = 1. * yl;
19     double dend = 1. * denl;
20
21     printf("unsigned long: %lu %lu %lu\n", xl, yl, denl);
22     printf("double:          %.01f %.01f %.01f\n", xd, yd, dend);
23
24     printf("double x,y (.2): %.21f %.21f\n", xd/dend, yd/dend);
25     printf("double x,y (.46): %.461f %.461f\n\n", xd/dend, yd/dend);
26
27     mpq_t x, y;
28     mpq_inits(x, y, NULL);
29     mpq_set_ui(x, xl, denl);
30     mpq_set_ui(y, yl, denl);
31
32     mpq_canonicalize(x);
33     mpq_canonicalize(y);
34
35     mpf_t xmpf, ympf;
36     mpf_inits(xmpf, ympf, NULL);
37     mpf_set_q(xmpf, x);
38     mpf_set_q(ympf, y);
39
40     gmp_printf("mpq x,y (canonical form): %Qd %Qd\n", x, y);
41     gmp_printf("mpf x,y (to %d decimal digits): %. *Ff %. *Ff\n",
42               digits, digits, xmpf, digits, ympf);
43     gmp_printf("mpq x,y (double .46): %.461f %.461f\n",
44               mpq2d(&x), mpq2d(&y));
45
46     mpq_clears(x, y, NULL);
47     mpf_clears(xmpf, ympf, NULL);
48     return ret;
49 }

```

[lec12/gmp/demo-gmp-mpq.c](#)

## Racionální čísla knihovny gmp - 3/3

- Souřadnice  $x,y$  - 7511164176768 346868669952 3739567104  $\sim$  2008,57; 92,76.

```
$ make
```

```
clang -c -I/usr/local/include -g demo-gmp-mpq.c -o demo-gmp-mpq.o
clang demo-gmp-mpq.o -L/usr/local/lib -lgmp -o demo-gmp-mpq
clang -c -I/usr/local/include -g demo-gmp-mpz.c -o demo-gmp-mpz.o
clang demo-gmp-mpz.o -L/usr/local/lib -lgmp -o demo-gmp-mpz
```

```
$ ./demo-gmp-mpq
```

```
unsigned long: 7511164176768 346868669952 3739567104
double:      7511164176768 346868669952 3739567104
double x,y (.2): 2008.57 92.76
double x,y (.46): 2008.5651541681761500512948259711265563964843750000
                92.7563700036227487544238101691007614135742187500
```

```
mpq x,y (canonical form): 399190273/198744 1536231/16562
mpf x,y (to 21 decimal digits): 2008.565154168176146200000 92.756370003622750875500
mpq x,y (double .46): 2008.5651541681759226776193827390670776367187500000
                92.756370003622748754423810169100761413574218750
```

[lec12/gmp/demo-gmp-mpq.c](#)

# Makefile s pkg-config a gmp

```

1  CFLAGS+=$(shell pkg-config --cflags gmp)
2  LDFLAGS+=$(shell pkg-config --libs gmp)
3
4  CFLAGS+=-g
5
6  DEMO_MPQ=demo-gmp-mpq
7  DEMO_MPZ=demo-gmp-mpz
8
9  TARGETS+=$(DEMO_MPQ) $(DEMO_MPZ)
10
11 bin: $(TARGETS)
12
13 info:
14     @echo $(CFLAGS)
15     @echo $(LDFLAGS)
16
17 $(DEMO_MPQ): $(DEMO_MPQ).o
18     $(CC) $< $(LDFLAGS) -o $@
19
20 $(DEMO_MPZ): $(DEMO_MPZ).o
21     $(CC) $< $(LDFLAGS) -o $@
22
23 %.o : %.c
24     $(CC) -c $(CPPFLAGS) $(CFLAGS) $< -o $@
25
26 clean:
27     $(RM) $(DEMO_MPQ) $(DEMO_MPZ) *.o

```

lec12/gmp/Makefile

## Reprezentace necelých čísel – IEEE 754

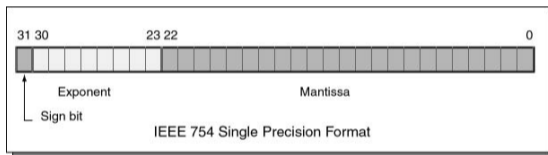
- Reálné číslo  $x$  se zobrazuje ve tvaru

$$x = (-1)^s \text{mantisa} \cdot 2^{\text{exponent} - \text{bias}}$$

Základ 2.

IEEE 754, ISO/IEC/IEEE 60559:2011

- Mantisa je **normalizována** na první jedničku vlevo (v soustavě o dvojkovém základu).
- float** – 32 bitů (4 bajty):  $s$  – 1 bit znaménko (+ nebo –), **exponent** – 8 bitů, tj. 256 možností.  
**mantisa** – 23 bitů  $\approx 16,7$  milionu možností.



- double** – 64 bitů (8 bajtů).
  - $s$  – 1 bit znaménko (+ nebo –).
  - exponent** – 11 bitů, tj. 2048 možností.
  - mantisa** – 52 bitů  $\approx 4,5$  biliardy možností (4 503 599 627 370 495).
- bias** umožňuje reprezentovat exponent vždy jako kladné číslo.

Lze zvolit, např.  $\text{bias} = 2^{eb-1} - 1$ , kde  $eb$  je počet bitů exponentu.

<http://www.root.cz/clanky/norma-ieee-754-a-pribuzni-formaty-plovouci-radove-tecky>



# Příklad reprezentace float hodnot dle IEEE 754

The image displays two screenshots of an "IEEE 754 Converter (JavaScript), V0.22" tool. Each screenshot shows the internal IEEE 754 representation of a floating-point number entered by the user.

**Top Screenshot: Input -256.75**

	Sign	Exponent	Mantissa
Value:	-1	$2^8$	1.0029296875
Encoded as:	1	135	24576
Binary:	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/>	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
You entered	-256.75		
Value actually stored in float:	-256.75		
Error due to conversion:	0.00		
Binary Representation	11000011100000000011000000000000		
Hexadecimal Representation	0xc3806000		

**Bottom Screenshot: Input -256.74**

	Sign	Exponent	Mantissa
Value:	-1	$2^8$	1.0028905868530273
Encoded as:	1	135	24248
Binary:	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/>	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/>
You entered	-256.74		
Value actually stored in float:	-256.739990234375		
Error due to conversion:	0.000009765625		
Binary Representation	1100001110000000101111010111000		
Hexadecimal Representation	0xc3805eb8		

- Chyba reprezentace -256.75 vs -256.74.
- **Infinity** (0x7f800000), **-Infinity** (0xff800000), a **NaN** (0xff7fffff).

## Příklady reprezentace hodnot typu float

### Reprezentace čísla 85,125 (float)

- 85 odpovídá  $1010101_{(2)}$ .
- 0,125 odpovídá 001
  - $0,125/2^{-1} = 0,25 \quad | \quad 0$
  - $0,125/2^{-2} = 0,50 \quad | \quad 0$
  - $0,125/2^{-3} = 1,00 \quad | \quad 1$
- 85,125 odpovídá  $1010101,001_{(2)} = 1,010101001_{(2)} \times 2^6$ ,
- Bias pro float je 127.
- Exponet je  $127 + 6 = 133$
- 133 odpovídá  $10000101_{(2)}$ .
- Normalizovaná mantisa je  $010101001_{(2)}$ , kterou doplníme nulami na 23 bitů (zprava, je to desetinné číslo).
- **0 - 1000 0101 - 0101 0100 1000 0000 0000 000.**
- **01000010 10101010 01000000 00000000.**
- V šestnáctkové soustavě **0x42 0xaa 0x40 0x00**, tedy **0x42aa4000**.

### Reprezentace čísla 0,1 (float)

- 0,1 má periodický rozvoj
  1.  $0,1 * 2 = 0,2 \quad | \quad 0$
  2.  $0,2 * 2 = 0,4 \quad | \quad 0$
  3.  $0,4 * 2 = 0,8 \quad | \quad 0$
  4.  $0,8 * 2 = 1,6 \quad | \quad 1$
  5.  $0,6 * 2 = 1,2 \quad | \quad 1$
  6.  $0,2 * 2 = 0,4 \quad | \quad 0$
- Opakuje se 0011, 23-bitů tak reprezentuje menší hodnotu.
- $0,1_{(10)} \sim 0,0001\ 1001\ 1001\ 1001\ 1001\ 1001\ 100_{(2)} = 1,1001\ 1001\ 1001\ 1001\ 1001\ 100_{(2)} \times 2^{-4}$ .
- Exponet je  $127 - 4 = 123$  odpovídá  $0111\ 1011_{(2)}$ .
- Normalizovaná mantisa  $\bar{1},100\ 1100\ 1100\ 1100\ 1100\ 1100$ .
- **0 - 0111 1011 - 100 1100 1100 1100 1100 1100.**
- **00111101 11001100 11001100 11001100.**
- V šestnáctkové soustavě to je **0x3d 0xcc 0xcc 0xcc**, tedy **0x3dcccccc**.
- **Prakticky je 0,1 převedeno na o něco větší číslo 0x3dcccccd, protože absolutní chyba je menší.**

[lec12/floats.c](#)

## Sčítání mnoha malých necelých čísel - 1/2

- Na příkladu součtu dvou velmi odlišných čísel (např.  $1 \times 10^{10} + 1 \times 10^{-10}$ ) dochází z důvodu omezené reprezentace mantisy k zaokrouhlovací chybě.
- V případě naivní implementace součtu velkého počtu (např.  $2^{30}$ ) velmi malých hodnot (např.  $1 \times 10^{-20}$ ) může dojít vlivem zaokrouhlování k významné chybě.

lec12/addition.c

```
// small value to be sum
float v = 1e-20f; //float literal

// 1073741824 is 230 values (1e9)
const size_t power = 30;
size_t n = 1l<<power ;

// multiplication factor for print
const double k = 1e11;

float *values = init_values(n, v);
```

```
double sum1 = v*n * k;
```

Přímé násobení - výsledek  
1.073 741 789 925 364 287 228 1.

```
float sum_naive(size_t n, float *v)
{
    float r = 0;
    for (size_t i = 0; i < n; ++i) {
        r += v[i];
    }
    return r;
}
```

```
double sum2 = sum_naive(n, values) * k;
```

Naivní součet - výsledek  
0.022 737 367 544 323 205 947 9.

```
float sum_alter(size_t n, float *v, size_t power)
{
    float r = 0;
    const size_t order = power - 1;
    size_t k = 2;
    for (size_t l = 1; l < order; ++l, k *= 2) {
        for (size_t i = 0; i < n; i += k) {
            v[i] = v[i] + v[i+k/2];
        }
        k /= 2;
    }
    for (size_t i = 0; i < n; i += k) {
        r += v[i];
    }
    return r;
}
```

Sčítání po dvojicích - výsledek  
1.073 741 793 632 507 324 218 8.

## Sčítání mnoha malých necelých čísel - 2/2

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  float* init_values(size_t n, float v);
5  float sum_naive(size_t n, float *v);
6  float sum_alter(size_t n, float *v, size_t power);
7
8  int main(void)
9  {
10     float v = 1e-20f; // small value to be sum
11     const size_t power = 30; // try 3 vs. 30
12     size_t n = 11<<power; // 1073741824 is 230 values
13     const double k = 1e11;
14
15     float *values = init_values(n, v);
16
17     double sum1 = v * n * k;
18     double sum2 = sum_naive(n, values) * k;
19     float sum3 = sum_alter(n, values, power) * k;
20
21     printf("Sum of %lu numbers of the value %.22lf\n", n, v);
22     printf("Sum1 (multiplication): %.22lf\n", sum1);
23     printf("Sum2 (naive)           : %.22lf\n", sum2);
24     printf("Sum3 (alter)           : %.22lf\n", sum3);
25     free(values);
26     return EXIT_SUCCESS;
27 }

```

```

29 float* init_values(size_t n, float v)
30 {
31     float *r = malloc(n * sizeof(v));
32     if (!r) {
33         fprintf(stderr, "ERROR: MEM_ALLOC\n");
34         exit(-1);
35     }
36     for (size_t i = 0; i < n; ++i) {
37         r[i] = v;
38     }
39     return r;
40 }

```

```

$ clang addition.c -o addition && ./addition
Sum of 1073741824 numbers of the value
0.00000000000000000000100

```

```

Sum1 (multiplication): 1.0737417899253642872281
Sum2 (naive)           : 0.0227373675443232059479
Sum3 (alter)           : 1.0737417936325073242188

```

```

$ calc "1e-20 * 2^30 * 1e11"
1.073741824

```

lec12/addition.c

- Implementujte s využitím knihovny gmp.

## Část II

### Část 2 – Rychlost výpočtu (programu)

## Maticové násobení - Naivně

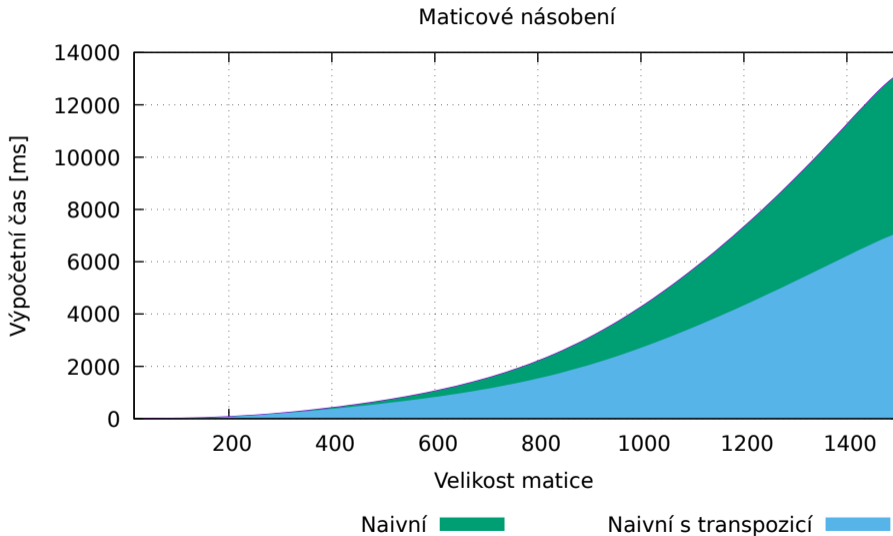
```
1 void simple_multiply(const int n, const double *a, const double *b, double *c)
2 {
3     for (int i = 0; i < n; ++i) {
4         for (int j = 0; j < n; ++j) {
5             double prod = 0;
6             for (int k = 0; k < n; ++k) {
7                 prod += a[i * n + k] * b[k * n + j];
8             }
9             c[i * n + j] = prod;
10        }
11    }
12 }
```

- Pro přehlednost předpokládáme kompatibilní rozměry matic a správně alokované.

## Maticové násobení - Naivně s transpozicí

```
1 void simple_multiply_trans(const int n, const double *a, const double *b, double *c)
2 {
3     double * bT = create_matrix(n); // allocate memory for transposed matrix
4     for (int i = 0; i < n; ++i) {
5         bT[i*n + i] = b[i*n + i];
6         for (int j = i + 1; j < n; ++j) {
7             bT[i*n + j] = b[j*n + i];
8             bT[j*n + i] = b[i*n + j];
9         }
10    }
11    for (int i = 0; i < n; ++i) {
12        for (int j = 0; j < n; ++j) {
13            double tmp = 0;
14            for (int k = 0; k < n; ++k) {
15                tmp += a[i*n + k] * bT[j*n + k];
16            }
17            c[i*n + j] = tmp;
18        }
19    }
20    free(bT);
21 }
```

# Porovnání rychlosti násobení matic





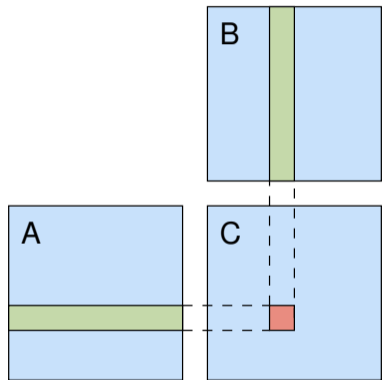
## Architektura procesoru a způsob výpočtu

- Příklad násobení matic a násobení transponované matice ukazuje, že kromě instrukcí má zásadní vliv **organizace dat a přístup do paměti**.
- V moderních procesorech hraje **cache** zásadní roli společně s řetězením instrukcí, tzv. **pipelining**, a využitím specifických instrukcí.

SIMD - Single Instruction Multiple Data

- Proniknutí do detailů fungování cache a řetězení instrukcí je náplní předmětu **Architektura počítačů (BOB35AP0)**, kde máte možnost se seznámit s přicházející architekturou RISC V.

<https://cw.felk.cvut.cz/wiki/courses/b35apo/>



## Optimalizace kódu

- Kromě optimalizace výsledného spustitelného kódu při překladu, je možné optimalizovat kódu za běhu nebo již existujících binární (přeložené) soubory.
  - **BOLT** - Binary Optimization and Layout Tool, zrychlení o až 20 %–50 %.

<https://arxiv.org/abs/1807.06735>

<https://dl.acm.org/doi/10.5555/3314872.3314876>

- Využití speciálních instrukcí v základních funkcích může výpočty (programy) výrazně urychlit, zejména pokud se funkce používají masivně.
  - AVX2 a EVEX instrukce (ze sady SSE4.2) ve funkcích porovnání řetězců `str{n}casecmp()` – až o 38 % méně potřebného času.

03/2022 - <https://www.phoronix.com/news/Glibc-strcasecmp-AVX2-EVEX>

- *V obou případech (a obecně) je vhodné rozumět principu a využít instrukce Assembleru.*

*Informativní*

# Compiler Explorer

The screenshot shows the Compiler Explorer interface with the following content:

**Source Code (C source #1):**

```

1 int square(int num)
2 {
3     return num * num;
4 }
5
6 int main(void)
7 {
8     int a = square(10);
9     return 0;
10 }
11
12

```

**Preprocessor Output (x86-64 gcc 12.2):**

```

1 /* <7 lines filtered>
2
3 int square(int num)
4 {
5     return num * num;
6 }
7
8 int main(void)
9 {
10     int a = square(10);
11     return 0;
12 }
13
14

```

**Assembly (x86-64 gcc 12.2):**

```

1 square:
2     push    rbp
3     mov     rbp, rsp
4     mov     DWORD PTR [rbp-4], edi
5     mov     eax, DWORD PTR [rbp-4]
6     imul  eax, eax
7     pop     rbp
8     ret
9
10 main:
11     push   rbp
12     mov    rbp, rsp
13     sub   rsp, 16
14     mov   edi, 10
15     call square
16     mov  DWORD PTR [rbp-4], eax
17     mov  eax, 0
18     leave
19     ret

```

**Output (0/0):** x86-64 gcc 12.2 i - 391ms (3994B) ~248 lines filtered

<https://godbolt.org/z/K9r1eWqcd>

# Compiler Explorer – Analýza optimalizovaného kódu

- Vliv optimalizace `-O2` na výsledný kód, který obsahuje nedefinované chování, přetečení celého čísla.

The screenshot shows the Compiler Explorer interface with three panes. The left pane shows the source code:

```

1 int main(void)
2 {
3     int ret = 0;
4     for (int i = 2147483640; i >= 0; ++i) {
5         ret += i;
6     }
7     return ret;
8 }

```

The middle pane shows the assembly code for the default optimization level:

```

1 main:
2     push    rbp
3     mov     rbp, rsp
4     mov     DWORD PTR [rbp-4], 0
5     mov     DWORD PTR [rbp-8], 2147483640
6     jmp     .L2
7 .L3:
8     mov     eax, DWORD PTR [rbp-8]
9     add     DWORD PTR [rbp-4], eax
10    add     DWORD PTR [rbp-8], 1
11 .L2:
12    cmp     DWORD PTR [rbp-8], 0
13    jns     .L3
14    mov     eax, DWORD PTR [rbp-4]
15    pop     rbp
16    ret

```

The right pane shows the assembly code for the `-O2` optimization level:

```

1 main:
2 .L2:
3     jmp     .L2

```

The status bar at the bottom of each pane shows compilation details for `x86-64 gcc 12.2`. The `-O2` pane shows a significantly faster compilation time (526ms) compared to the default level (622ms).

<https://godbolt.org/z/G3GEz4vbv>

## Příklad použití OpenMP - Maticové násobení 1/2

- Open Multi-Processing (OpenMP) - aplikační programové rozhraní (API) multiplatformních výpočtů se sdílenou pamětí. <http://www.openmp.org>
- Direktivou preprocesoru můžeme instruovat kompilátor k vytvoření kódu paralelního výpočtu, např. paralelizace přes vnější proměnnou `i`.

```
1 void multiply(int n, int a[n][n], int b[n][n], int c[n][n])
2 {
3     int i;
4     #pragma omp parallel private(i)
5     #pragma omp for schedule (dynamic, 1)
6     for (i = 0; i < n; ++i) {
7         for (int j = 0; j < n; ++j) {
8             c[i][j] = 0;
9             for (int k = 0; k < n; ++k) {
10                c[i][j] += a[i][k] * b[k][j];
11            }
12        }
13    }
14 }
```

[lec12/demo-omp-matrix.c](#)

*Pro přehlednost uvažujeme čtvercové matice stejných rozměrů.*

## Příklad použití OpenMP - Maticové násobení 2/2

- Příklad násobení matic  $1000 \times 1000$  s využitím OpenMP na iCore5 (2 jádra s HT  $\sim 4 \times$  výpočetní jednotky).

```
1 gcc -std=c99 -O2 -o demo-omp demo-omp-matrix.c -fopenmp
2 ./demo-omp 1000
3 Size of matrices 1000 x 1000 naive
4 multiplication with  $O(n^3)$ 
5 c1 == c2: 1
6 Multiplication single core 9.33 sec
7 Multiplication multi-core 4.73 sec
8
9 OMP_NUM_THREADS=2 ./demo-omp 1000
10 Size of matrices 1000 x 1000 naive
11 multiplication with  $O(n^3)$ 
12 c1 == c2: 1
13 Multiplication single core 9.48 sec
14 Multiplication multi-core 6.23 sec
```

- i7-6700K:

- $1 \times$  vlákno 0.80s;
- $2 \times$  vlákna 0.39s;
- $4 \times$  vlákna 0.24s.

- Násobení matic  $5000 \times 5000$  (Ryzen 9 5950X).

OMP_NUM_THREADS	Time (sec)
1	256.00
2	128.00
4	64.00
8	32.00
16	18.05

- ```
1 OMP_NUM_THREADS=16 ./demo-omp 5000
2 Size of matrices 5000 x 5000 naive
3 multiplication with  $O(n^3)$ 
4 Multiplication single core 256.00 sec
5 Multiplication multi-core 18.05 sec
```

lec12/demo-omp-matrix.c




## Část III

### Část 3 – Implementance domácích úkolů (HW10B)

## Zrychlení domácího úkolu HW10B

- V případě správného použití prioritní fronty haldou, je nejvíce času programu stráveno
  - načítáním grafu z textového souboru a
  - ukládáním výsledku do textového souboru.
- V obou případech se jedná tři celá čísla (v rozsahu `int`) na řádek.
  - Převádíme znaky na celá čísla a zpět.
  - Referenční řešení (`ref-1ec12`) využívá funkce `fscanf()` a `fprintf()`, které jsou relativně komplexní a pomalé.

### Legend

|             |                                                                                   |                                                                                                                                                           |
|-------------|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| Load time:  |  | <code>ref-1ec12</code> - reference implementation provided as a part of the source codes for the lecture 12.                                              |
| Solve time: |  | <code>tdijkstra</code> - baseline reference solution.                                                                                                     |
| Save time:  |  | <code>ref-mh21</code> - reference solution, the fastest student implementation of the year 2019 (updated to 2021 standards).                              |
|             |                                                                                   | <code>ref-radix-mh21</code> - reference solution with a variant of radix queue and even faster save (student's implementation updated to 2021 standards). |





## Zrychlení domácího úkolu HW10B - Načítání

- Při načítání grafu můžeme využít faktu, že vstupní soubor obsahuje pouze kladná čísla oddělená mezerami v rozsahu `int`.
- Načítání můžeme urychlit přímou konverzí načteného znaku na celé číslo.
- Princip načtení celého čísla z řetězce může být následující. *Vstup je posloupnost znaků.*

```
int parse_int(const char *str, int len)
{
    int num = 0;
    int i = 0;
    while (i < len && str[i] >= '0' && str[i] <= '9') {
        num = num * 10 + (str[i++] - '0');
    }
    return str[i] == ' ' || str[i] == '\n' ? num : -1;
}
```

lec12/int\_string.c

Vyžadujeme oddělení číselných hodnot mezerou nebo znakem nového řádku a předpokládáme `str[len] == '\0'`.

## Zrychlení domácího úkolu HW10B - Ukládání

- Můžeme využít maximálního počtu znaků v rozsahu typu `int` a výstupní posloupnost znaků reprezentující celé číslo vytvářet od nejnižšího řádu postupným dělením.
- Princip konverze kladného celého čísla může být následující. *Výstup je posloupnost znaků.*

```
char* int_to_string(int v, char *buf, size_t capacity)
{
    char *cur = buf + capacity - 1; //last char of the buffer buf
    *cur = '\0';
    do {
        int least = v % 10;
        v /= 10;
        *(--cur) = least + '0';
    } while (v != 0);
    return cur;
}
```

`lec12/int_string.c`

# Shrnutí přednášky

## Diskutovaná témata

- Numerická přesnost.
- Knihovna `gmp`.
- Maticové násobení a organizace paměti.
- Rychlost výpočtu a architektura procesoru.
- Paralení výpočty `OpenMP`.
- Domácí úkol HW10B.