

Přesnost výpočtů

Jan Faigl

Katedra počítačů

Fakulta elektrotechnická

České vysoké učení technické v Praze

Přednáška 12

B0B36PRP – Procedurální programování



Přehled témat

- Část 1 – Typové konverze

Typové konverze

S. G. Kochan: kapitola 14 (typové konverze)

- Část 2 – Přesnost výpočtů

Přesnost výpočtů a numerická stability

Příklad použití knihovny gmp

- Část 3 – Implementace bonusového úlohy HW10B

Domácí úkol HW10B

- Část 4 – Rychlost výpočtů (*volitelné*)

Maticové násobení

Rychlost výpočtu

Comparing C to Machine Code

Paralelní výpočet



Část I

Část 1 – Typové konverze



Obsah

Typové konverze



Přiřazovací operátor a příkaz

- Slouží pro nastavení hodnoty proměnné.

Uložení číselné hodnoty do paměti, kterou proměnná reprezentuje.

- Tvar přiřazovacího operátoru.

$\langle \text{proměnná} \rangle = \langle \text{výraz} \rangle$

Výraz je literál, proměnná, volání funkce, ...

- Zkrácený zápis

$\langle \text{proměnná} \rangle \langle \text{operátor} \rangle = \langle \text{výraz} \rangle$

- Přiřazení je výraz **asociativní zprava**.
- Přiřazovací příkaz – výraz zakončený středníkem ;

```
1  int x; //definice promenne x
2  int y; //definice promenne y
4  x = 6;
5  y = x = x + 6;
```

```
1  int x, y; //definice promennych x a y
3  x = 10;
4  y = 7;
6  y += x + 10;
```



Typové konverze

- Typová konverze je operace převedení hodnoty nějakého typu na hodnotu typu jiného.
- Typová konverze může být
 - **implicitní** – vyvolá se automaticky;
 - **explicitní** – je nutné v programu explicitně uvést.
- Konverze typu **int** na **double** je implicitní.

*Hodnota typu **int** může být použita ve výrazu, kde se očekává hodnota typu **double**, dojde k automatickému převodu na hodnotu typu **double**.*

Příklad

```
1 double x;  
2 int i = 1;  
  
4 x = i; // hodnota 1 typu int se automaticky převede  
5       // na hodnotu 1.0 typu double
```

- Implicitní konverze je bezpečná.



Explicitní typové konverze

- Převod hodnoty typu **double** na **int** je třeba **explicitně** předeepsat.
- Dojde k „odseknutí“ necelé části hodnoty **int**.

Příklad

```
1 double x = 1.2; // definice proměnné typu double
2 int i;          // definice proměnné typu int
3 int i = (int)x; // hodnota 1.2 typu double se převede
4                // na hodnotu 1 typu int
```

- Explicitní konverze je potenciálně nebezpečná.

Příklady

```
1 double d = 1e30;
2 int i = (int)d;
4 // i je -2147483648
5 // to je asi -2e9 místo 1e30
```

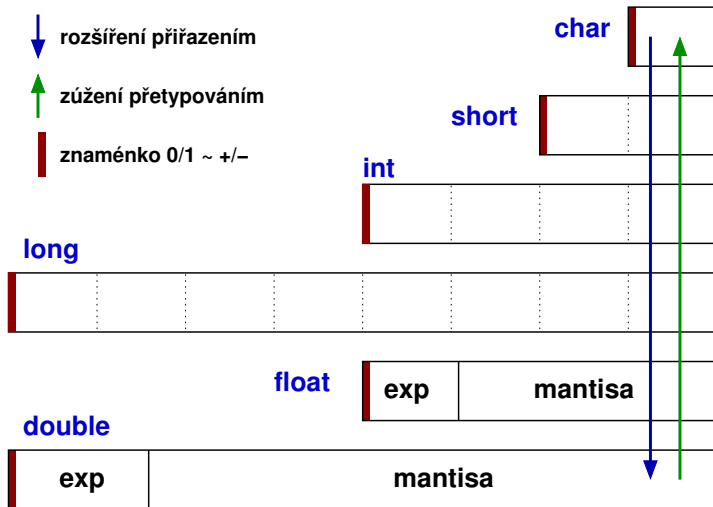
```
1 long l = 5000000000L;
2 int i = (int)l;
4 // i je 705032704
5 // (oříznuté 4 bajty)
```

lec12/demo-type_conversion.c



Konverze primitivních číselných typů

- Primitivní datové typy jsou vzájemně nekompatibilní, ale jejich hodnoty lze převádět.



Matematické funkce

- `<math.h>` – základní funkce pro práci s „reálnými“ čísly.

- Výpočet odmocniny necelého čísla x .

```
double sqrt(double x);, float sqrtf(float x);
```

V C funkce nepřetěžujeme, proto jsou jména odlišena.

- `double pow(double x, double y);` – výpočet obecné mocniny.
- `double atan2(double y, double x);` – výpočet $\arctan y/x$ s určením kvadrantu.
- Symbolické konstanty – `M_PI`, `M_PI_2`, `M_PI_4`, atd.
 - `#define M_PI 3.14159265358979323846`
 - `#define M_PI_2 1.57079632679489661923`
 - `#define M_PI_4 0.78539816339744830962`
- `isfinite()`, `isnan()`, `isless()`, ... – makra pro porovnání reálných čísel.
- `round()`, `ceil()`, `floor()` – zaokrouhlování, převod na celá čísla.

- `<complex.h>` – funkce pro počítání s komplexními čísly.

ISO C99

- `<fenv.h>` – funkce pro řízení zaokrouhlování a reprezentaci dle IEEE 754.

man math



Část II

Část 2 – Přesnost výpočtu



Obsah

Přesnost výpočtů a numerická stability

Příklad použití knihovny gmp



Přesnost výpočtu - Příklad dělení dvou čísel

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

lec12/division.c



Přesnost výpočtu - strojová přesnost

- Strojová přesnost ϵ_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 ϵ_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.



Obsah

Přesnost výpočtů a numerická stability

Příklad použití knihovny gmp



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>
4  #include <gmp.h>
6  const char* resultSrc =
7  "932865073719992059629773513614789388266580305083"
8  "920591925740371392254317064584855785088915745761"
    ;
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);
18     gmp_printf("n1: %Zd\n", n1);
19     gmp_printf("n2: %Zd\n", n2);
21     gmp_printf("%Zd^%d x %Zd^%d\n\n", n1, 8, n2, 8)
        ;
23     mpz_pow_ui(n1, n1, 8);
24     mpz_pow_ui(n2, n2, 8);

```

```

26     gmp_printf("%Zd x %Zd\n\n", n1, n2);
28     mpz_mul(result, n1, n2);
29     gmp_printf("%Zd\n\n", result);
31     printf("Result from HW04\n%s\n", resultSrc);
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

```

lec12/gmp/demo-gmp-mpz.c



Racionální čísla knihovny gmp - 1/3

- „Libovolné přesnosti“ reprezentace, např. souřadnic v rovině jako výsledek operací výpočetní geometrie, můžeme realizovat podílem 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`.

```
49 double mpq2d(const mpq_t *op)
50 {
51     double ret;
52     mpf_t v;
53     mpf_init(v);
54     mpf_set_q(v, *op);
55     ret = mpf_get_d(v);
56     mpf_clear(v);
57     return ret;
58 }
```

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



Racionální čísla knihovny gmp - 2/3

```
1  #include <stdio.h>
2  #include <stdlib.h>
4  #include <gmp.h>
6  double mpq2d(const mpq_t *op);
8  int main(int argc, char *argv[])
9  {
10     int ret = EXIT_SUCCESS;
12     unsigned long x1 = 75111641767681;
13     unsigned long y1 = 3468686699521;
14     unsigned long den1 = 37395671041;
15     const unsigned int digits = 21;
17     double xd = 1. * x1;
18     double yd = 1. * y1;
19     double dend = 1. * den1;
21     printf("unsigned long: %lu %lu %lu\n", x1, y1, den1);
22     printf("double:          %.01f %.01f %.01f\n", xd, yd, dend);
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);
```

```
27     mpq_t x, y;
28     mpq_inits(x, y, NULL);
29     mpq_set_ui(x, x1, den1);
30     mpq_set_ui(y, y1, den1);
32     mpq_canonicalize(x);
33     mpq_canonicalize(y);
35     mpf_t xmpf, ympf;
36     mpf_inits(xmpf, ympf, NULL);
37     mpf_set_q(xmpf, x);
38     mpf_set_q(ympf, y);
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", digits, digits, xmpf, digits, ympf);
42     gmp_printf("mpq x,y (double .46): %.461f %.461f\n",
        mpq2d(&x), mpq2d(&y));
44     mpq_clears(x, y, NULL);
45     mpf_clears(xmpf, ympf, NULL);
46     return ret;
47 }
```

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)
4  CFLAGS+=-g
6  DEMO_MPQ=demo-gmp-mpq
7  DEMO_MPZ=demo-gmp-mpz
9  TARGETS+=$(DEMO_MPQ) $(DEMO_MPZ)
11 bin: $(TARGETS)
13 info:
14     @echo $(CFLAGS)
15     @echo $(LDFLAGS)
```

```
17 $(DEMO_MPQ): $(DEMO_MPQ).o
18     $(CC) $< $(LDFLAGS) -o $@
20 $(DEMO_MPZ): $(DEMO_MPZ).o
21     $(CC) $< $(LDFLAGS) -o $@
23 %.o : %.c
24     $(CC) -c $(CPPFLAGS) $(CFLAGS) $< -o $@
26 clean:
27     $(RM) $(DEMO_MPQ) $(DEMO_MPZ) *.o
```

lec12/gmp/Makefile

```
$ make info
-I/usr/local/include -g
-L/usr/local/lib -lgmp
```

```
$ gmake
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
```



Část III

Část 3 – Implementace bonusového úlohy HW10B



Obsah

Domácí úkol 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-lec11`) využívá funkce `fscanf()` a `fprintf()`, které jsou relativně komplexní a pomalé.

Legend

Load time: 
Solve time: 
Save time: 

ref-lec12 - reference implementation provided as a part of the source codes for the lecture 12.

tdijkstra - baseline reference solution.

ref-mh21 - reference solution, the fastest student implementation of the year 2019 (updated to 2021 standards).

ref-radix-mh21 - 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.*

```
1  int parse_int(const char *str, int len)
2  {
3      int num = 0;
4      int i = 0;
5      while (i < len && str[i] >= '0' && str[i] <= '9') {
6          num = num * 10 + (str[i++] - '0');
7      }
8      return str[i] == ' ' || str[i] == '\n' ? num : -1;
9  }
```

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.*

```
1 char* int_to_string(int v, char *buf, size_t capacity)
2 {
3     char *cur = buf + capacity - 1; //last char of the buffer buf
4     *cur = '\0';
5     do {
6         int least = v % 10;
7         v /= 10;
8         *(--cur) = least + '0';
9     } while (v != 0);
10    return cur;
11 }
```

lec12/int_string.c



Část IV

Část 4 – Rychlost výpočtu (programu)



Obsah

Maticové násobení

Rychlost výpočtu

Comparing C to Machine Code

Paralelní výpočet



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.



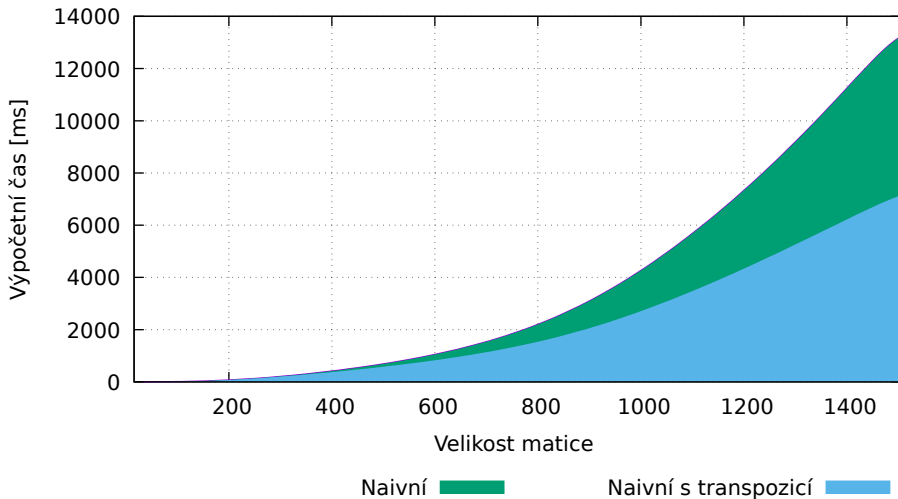
Maticové násobení - Naivně s transpozicí

```
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

Maticové násobení



Obsah

Maticové násobení

Rychlost výpočtu

Comparing C to Machine Code

Paralelní výpočet



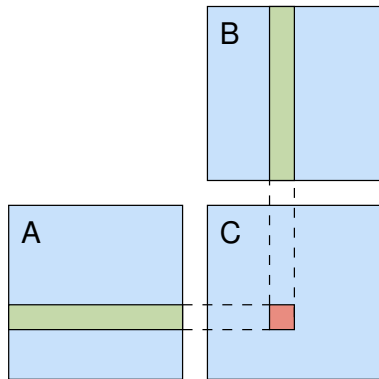
Architektura procesoru a způsob výpočtu

- 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 (B0B35AP0)**, 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 three main panels:

- C source #1:** Contains the following C code:

```
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 (Editor):** Shows the output after preprocessing, with a note that 7 lines were filtered. The visible code is:

```
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 }
```
- x86-64 gcc 12.2 (Editor #1):** Displays the assembly code generated by the compiler. The assembly for the `square` function is:

```
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
```

At the bottom, the output section shows: `Output (0/0) x86-64 gcc 12.2 i -391ms (3984B) ~248 lines filtered`. The Compiler License is also visible at the bottom.

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



Compiler Explorer – Analýza optimalizovaného kódu

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

Příloha 3. přednášky.

The screenshot displays the Compiler Explorer interface with three panels. The left panel shows the C source code for a program that increments a variable `i` until it overflows. The middle panel shows the assembly output for `gcc 12.2` with default optimization, which includes a loop body with multiple instructions. The right panel shows the assembly output for `gcc 12.2` with `-O2` optimization, where the loop body is replaced by a single `jmp` instruction, indicating an infinite loop due to undefined behavior. The status bars at the bottom of the middle and right panels show compilation times and line counts.

Compiler Explorer Add... More Templates Check out our stats page Sponsors Backtrace intel Solid

C source #1 x86-64 gcc 12.2 (Editor #1) x86-64 gcc 12.2 (Editor #1) x86-64 gcc 12.2 (Editor #1)

Save/Load + Add new... Vim C

x86-64 gcc 12.2 Compiler options... **x86-64 gcc 12.2** **-O2**

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 `}`

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`

1 `main:`
2 `.L2:`
3 `jmp .L2`

Output (0/0) x86-64 gcc 12.2 - 622ms (3290B) ~196 lines filtered
Compiler License

Output (0/7)
Compiler License - 526ms (3123B) ~196 lines filtered

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



Obsah

Maticové násobení

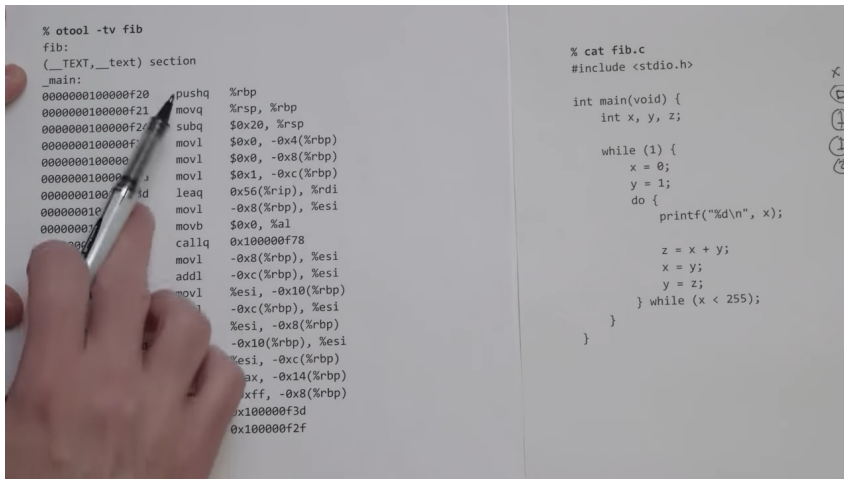
Rychlost výpočtu

Comparing C to Machine Code

Paralelní výpočet



Comparing C to Machine Code



<https://www.youtube.com/watch?v=y0yaJXpAYZQ>



Obsah

Maticové násobení

Rychlost výpočtu

Comparing C to Machine Code

Paralelní výpočet



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

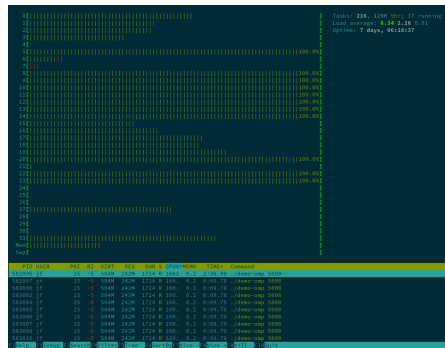
- Příklad násobení matic 1000×1000 s využitím OpenMP na iCore5 (2 jádra s HT $\sim 4\times$ výpočetní jednotky).

```
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
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 5000×5000 (Ryzen 9 5950X).



- 1 `OMP_NUM_THREADS=16 ./demo-omp 5000`
- 2 Size of matrices 5000×5000 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



Shrnutí přednášky



Diskutovaná témata

- Typové konverze.
 - Numerická přesnost.
 - Knihovna gmp.
 - Bonusová úloha HW10B.
-
- *Maticové násobení a organizace paměti.*
 - *Rychlost výpočtu a architektura procesoru.*
 - *Paralení výpočty OpenMP.*

