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 sčítání mnoha malých čísel

  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.

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

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

- Zkrácený zápis

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

- 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 předepsat.
- 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

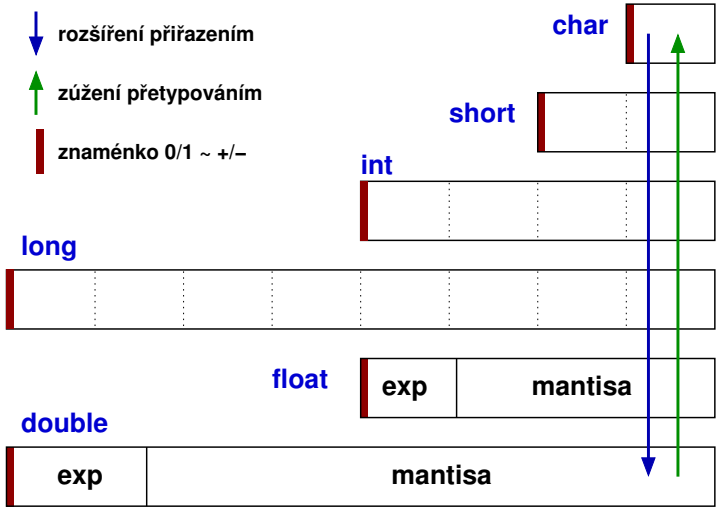
```
1 double d = 1e30;
2 int i = (int)d;
4 // i je -2147483648
5 // to je asi -2e9 misto 1e30
```

```
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 sčítání mnoha malých čísel

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 sčítání mnoha malých čísel

Příklad použití knihovny gmp

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. 1×10^{-20}) může dojít vlivem zaokrouhlování k významné chybě.

lec12/addition.c

```
// small value to be sum
float v = 1e-20f; //float literal
// 1073741824 is 2^30 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;
}
float sum3 = sum_alter(n, values, power) * k;
```

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

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. 1×10^{-20}) může dojít vlivem zaokrouhlování k významné chybě.

lec12/addition.c

```
// small value to be sum
float v = 1e-20f; //float literal
// 1073741824 is 2^30 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;
}
float sum3 = sum_alter(n, values, power) * k;
```

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

Sčítání mnoha malých necelých čísel - 2/2

```c
#include <stdio.h>
#include <stdlib.h>
float* init_values(size_t n, float v);
float sum_naive(size_t n, float *v);
float sum_alter(size_t n, float *v, size_t power);
int main(void)
{
   float v = 1e-20f; // small value to be sum
   const size_t power = 30; // try 3 vs.  30
   size_t n = 1l<<power ; // 1073741824 is 2^30 values
   const double k = 1e11;
   float *values = init_values(n, v);
   double sum1 = v * n  * k;
   double sum2 = sum_naive(n, values)  * k;
   float sum3 = sum_alter(n, values, power)  * k;
   printf("Sum of %lu numbers of the value %.22lf\n", n, v);
   printf("Sum1 (multiplication): %.22lf\n", sum1);
   printf("Sum2 (naive)         : %.22lf\n", sum2);
   printf("Sum3 (alter)         : %.22lf\n", sum3);
   free(values);
   return EXIT_SUCCESS;
}
```

```c
float* init_values(size_t n, float v)
{
   float *r = malloc(n * sizeof(v));
   if (!r) {
      fprintf(stderr, "ERROR: MEM_ALLOC\n");
      exit(-1);
   }
   for (size_t i = 0; i < n; ++i) {
      r[i] = v;
   }
   return r;
}
```

```
$ clang addition.c  -o addition && ./addition
Sum of 1073741824 numbers of the value
     0.00000000000000000000
Sum1 (multiplication): 1.073741789925364287228 1
Sum2 (naive)         : 0.022737367544323205947 9
Sum3 (alter)         : 1.073741793632507324218 8
$ calc "1e-20 * 2^30 * 1e11"
  1.073741824
```

lec12/addition.c

Implementujte s využitím knihovny gmp. Co je gmp?

Obsah

Přesnost výpočtů a numerická stability

Příklad sčítání mnoha malých čísel

Příklad použití knihovny gmp

Součin dvou velkých čísel knihovnou gmp - 1/2

- Nechť $(995663 \cdot 995669)^8$ je prvočíselný rozklad čísla

  93286507371999205962977351361478938826658030508392059192574037139225431706458485578508891574576 1.

  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

```c
#include <stdio.h>
#include <stdlib.h>
#include <gmp.h>
const char* resultSrc =
"93286507371999205962977351361478938826658030508392059"
"92059192574037139225431706458485578508891574576l";
int main(int argc, char *argv[])
{
    int ret = EXIT_SUCCESS;
    mpz_t n1, n2, result;
    mpz_init_set_str(n1, "995663", 10);
    mpz_init_set_str(n2, "995669", 10);
    mpz_init(result);
    gmp_printf("n1: %Zd\n", n1);
    gmp_printf("n2: %Zd\n", n2);
    gmp_printf("%Zd^%d x %Zd^%d\n\n", n1, 8, n2, 8);
    mpz_pow_ui(n1, n1, 8);
    mpz_pow_ui(n2, n2, 8);
    gmp_printf("%Zd x %Zd\n\n", n1, n2);
    mpz_mul(result, n1, n2);
    gmp_printf("%Zd\n\n", result);
    printf("Result from HW04\n%s\n", resultSrc);
    mpz_clears(n1, n2, result, NULL);
    return ret;
}
```

```
$ ./demo-gmp-mpz
n1: 995663
n2: 995669
995663^8 x 995669^8
9658261242946078679826999262556952968634003091 21 x
    9658726868682611515370370822602315664810477758 41
9328650737199920596297735136147893882665803050839 2059
1925740371392254317064584855785088915745761

Result from HW04
9328650737199920596297735136147893882665803050839 2059
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 necelé 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

```c
#include <stdio.h>
#include <stdlib.h>
#include <gmp.h>
double mpq2d(const mpq_t *op);
int main(int argc, char *argv[])
{
    int ret = EXIT_SUCCESS;
    unsigned long xl = 75111641767681;
    unsigned long yl = 3468686699521;
    unsigned long denl = 37395671041;
    const unsigned int digits = 21;
    double xd = 1. * xl;
    double yd = 1. * yl;
    double dend = 1. * denl;
    printf("unsigned long: %lu %lu %lu\n", xl, yl, denl);
    printf("double:        %.0lf %.0lf %.0lf\n", xd, yd, dend);
    printf("double x,y (.2): %.2lf %.2lf\n", xd/dend, yd/dend);
    printf("double x,y (.46): %.46lf %.46lf\n\n", xd/dend, yd/
     dend);
    mpq_t x, y;
    mpq_inits(x, y, NULL);
    mpq_set_ui(x, xl, denl);
    mpq_set_ui(y, yl, denl);
    mpq_canonicalize(x);
    mpq_canonicalize(y);
    mpf_t xmpf, ympf;
    mpf_inits(xmpf, ympf, NULL);
    mpf_set_q(xmpf, x);
    mpf_set_q(ympf, y);
    gmp_printf("mpq x,y (canonical form): %Qd %Qd\n", x,
     y);
    gmp_printf("mpf x,y (to %d decimal digits):  %.*Ff %.
     *Ff\n", digits, digits, xmpf, digits, ympf);
    gmp_printf("mpq x,y (double .46):  %.46lf %.46lf\n",
      mpq2d(&x), mpq2d(&y));
    mpq_clears(x, y, NULL);
    mpf_clears(xmpf, ympf, NULL);
    return ret;
}
```

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): 2008.56515416817615005129482597112655639648437500000
     92.75637000362274875442381016910076141357421875000
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):  2008.56515416817592267761938273906707763671875000000
     92.75637000362274875442381016910076141357421875
```

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

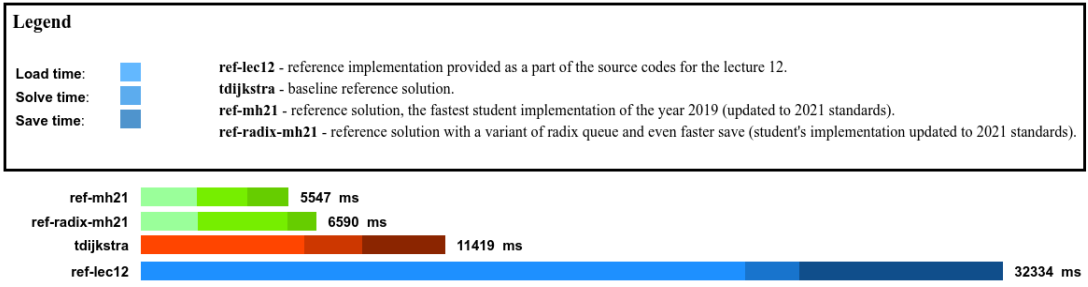
Část 3 – Implementace bonusového úlohy HW10B

Obsah

Domácí úkol HW10B

Zrychlení domácího úkolu HW10B

- V případě správného použití prioritní fronty haldou, je nejvíce času programu stráveno
  - načítáním grafu z textového soubor a
  - ukládáním výsledku do textového souboru.
- V obou případech se jedná tří 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.



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

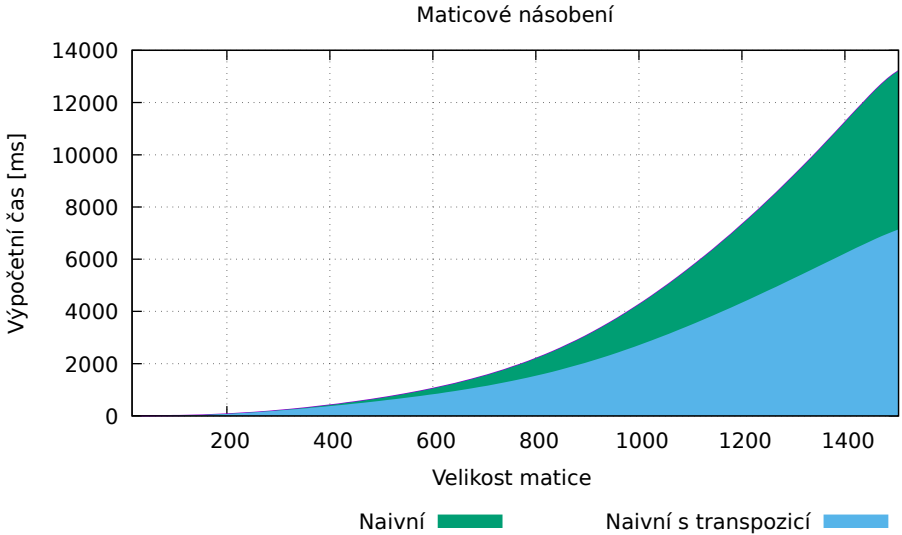
```
1  void simple_multiply(const int n, const double *a, const double *b, double *c)
2  {
3     for (int i = 0; i < n; ++i) {
4        for (int j = 0; j < n; ++j) {
5           double prod = 0;
6           for (int k = 0; k < n; ++k) {
7              prod += a[i * n + k] * b[k * n + j];
8          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



Obsah

Maticové násobení

Rychlost výpočtu

Comparing C to Machine Code

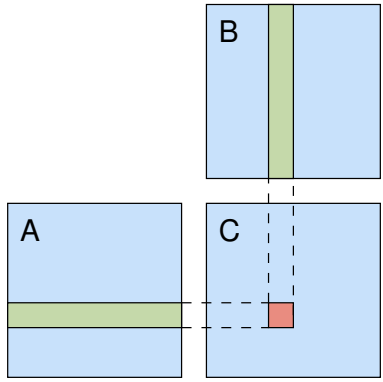
Paralelní výpočet

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 (B0B35APO)`, 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 překladů, je možné optimalizovat kódu za běhu nebo již existujících binární (přeložené) soubory.
  - **BOLT** - Binary Optimization and Layout Tool, zrychlení o až 20 %–50 %.

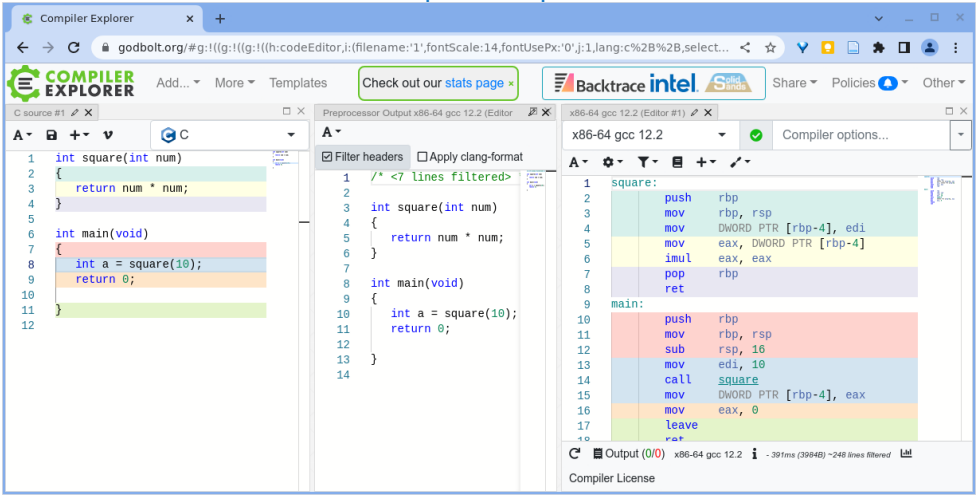
    https://arxiv.org/abs/1807.06735

    https://dl.acm.org/doi/10.5555/3314872.3314876
- Využití speciálních instrukcí v základních funkcí 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

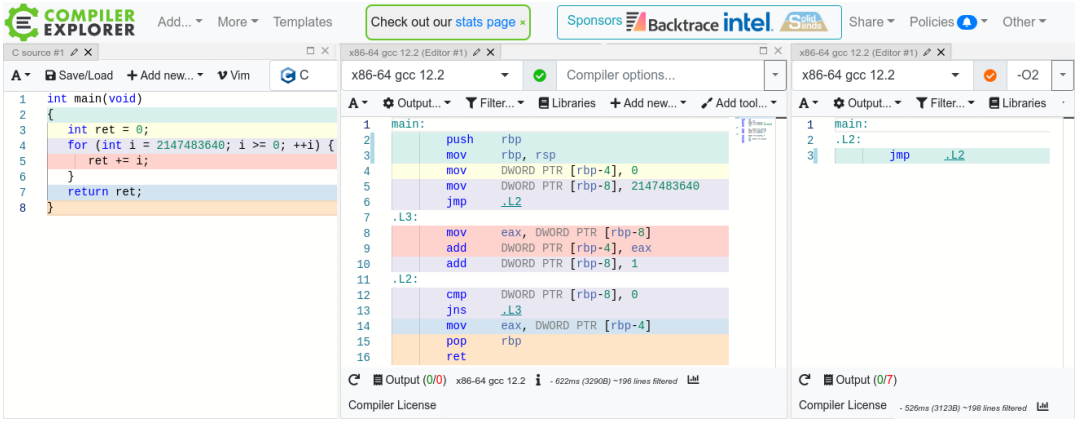


https://godbolt.org/z/K9r1eWqcd

Compiler Explorer – Analýza optimalizovaného kódu

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

Příloha 3. přednášky.



https://godbolt.org/z/G3GEz4vbv

Obsah

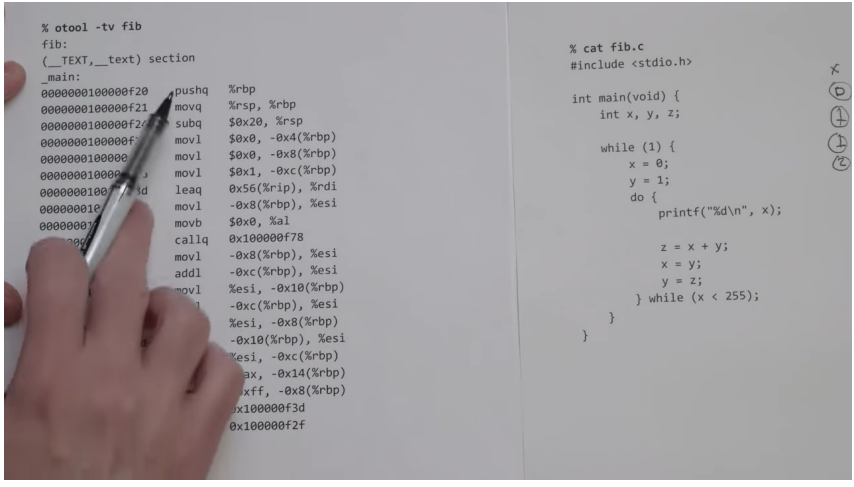
Maticové násobení

Rychlost výpočtu

Comparing C to Machine Code

Paralelní výpočet

Comparing C to Machine Code



https://www.youtube.com/watch?v=yOyaJXpAYZQ

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.

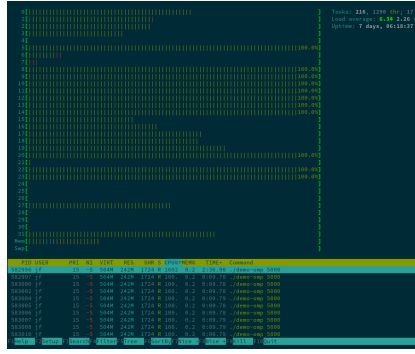
Příklad použití OpenMP - Maticové násobení 2/2

- Příklad násobení matic 1000 × 1000 s využitím OpenMP na iCore5 (2 jádra s HT ∼ 4× 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× vlákno 0.80 s;
  - 2× vlákna 0.39 s;
  - 4× vlákna 0.24 s.

- Násobení matic 5000 × 5000 (Ryzen 9 5950X).



```
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

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.*