Přesnost a rychlost výpočtu

Jan Faigl

Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

Lecture 10

PRG – Programming in C

Přehled témat

- Část 1 – Přesnost výpočtu

- Část 2 – Rychlost výpočtu (programu)

- Část 3 – Implementace domácích úkolů

Part I

Část 1 – Přesnost výpočtu

Outline

Přesnost výpočtu - Příklad součtu dvou čísel

```
#include <stdio.h>

int main(void)
{
   double a = 1e+10;
   double b = 1e-10;

   printf("a   : %24.12lf\n", a);
   printf("b   : %24.12lf\n", b);
   printf("a+b: %24.12lf\n", a + b);

   return 0;
}

clang sum.c && ./a.out
a   : 10000000000.000000000000
b   :             0.000000000100
a+b: 10000000000.000000000000
```

lec10/sum.c

Přesnost výpočtu - Příklad dělení dvou čísel

```c
#include <stdio.h>

int main(void)
{
   const int number = 100;
   double dV = 0.0;
   float fV = 0.0f;

   for (int i = 0; i < number; ++i) {
      dV += 1.0 / 10.0;
      fV += 1.0 / 10.0;
   }

   printf("double value: %lf ", dV);
   printf(" float value: %lf ", fV);

   return 0;
}
```

```
clang division.c && ./a.out
double value: 10.000000  float value: 10.000002
```

lec10/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.

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

```c
#include <stdio.h>
#include <stdlib.h>

#include <gmp.h>

const char* resultSrc =
"932865073719992059629773513614789388266580305083"
"920591925740371392254317064584855785088915745761";

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

965826124294607867982699926255695296863400309121 x
     965872686868261151537037082260231566481047775841
93286507371999205962977351361478938826658030508392059
1925740371392254317064584855785088915745761

Result from HW04
93286507371999205962977351361478938826658030508392059
1925740371392254317064584855785088915745761
```

lec10/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 neceléjo čísla `mpf_t`, který využijeme pro převod `mpq_t` na necelé číslo typu `double`.

```c
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;
}
```

lec10/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 = 7511164176768l;
   unsigned long yl = 346868669952l;
   unsigned long denl = 3739567104l;
   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;
}
```

lec10/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):  2008.56515416817614620000 92.7563700036227508755000
mpq x,y (double .46):  2008.5651541681759226776193827390670776367187500000
    92.756370003622748754423810169100761413574218750
```

lec10/gmp/demo-gmp-mpq.c

Makefile s pkg-config a gmp

```
CFLAGS+=$(shell pkg-config --cflags gmp)
LDFLAGS+=$(shell pkg-config --libs gmp)

CFLAGS+=-g

DEMO_MPQ=demo-gmp-mpq
DEMO_MPZ=demo-gmp-mpz

TARGETS+=$(DEMO_MPQ) $(DEMO_MPZ)

bin: $(TARGETS)

info:
	@echo $(CFLAGS)
	@echo $(LDFLAGS)

$(DEMO_MPQ): $(DEMO_MPQ).o
	$(CC) $< $(LDFLAGS) -o $@

$(DEMO_MPZ): $(DEMO_MPZ).o
	$(CC) $< $(LDFLAGS) -o $@

%.o : %.c
	$(CC) -c $(CPPFLAGS) $(CFLAGS) $< -o $@

clean:
	$(RM) $(DEMO_MPQ) $(DEMO_MPZ) *.o
```

lec10/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
```

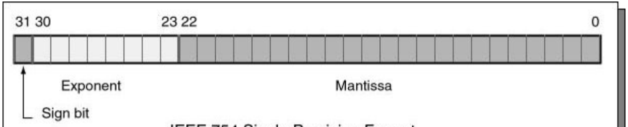
Reprezentace necelých čísel – IEEE 754

- Reálné číslo x se zobrazuje ve tvaru Základ 2.

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

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.



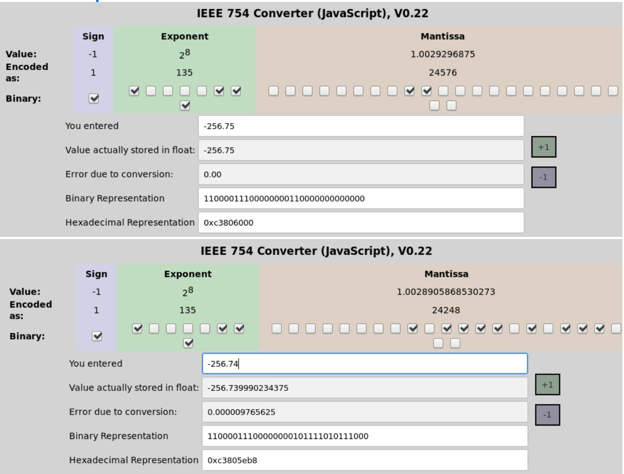
IEEE 754 Single Precision Format

- **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. 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



- Chyba reprezentace -256.75 vs -256.74.
- `Infinity` (0x7f800000), `-Infinity` (0xff800000), a `NaN` (0x7fffffff).

https://www.h-schmidt.net/FloatConverter/IEEE754.html

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 ~~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.

`lec10/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. 1×10^{-20}) může dojít vlivem zaokrouhlování k významné chybě.

lec10/addition.c

```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.

```c
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.

```c
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
     0.0000000000000000000100
Sum1 (multiplication): 1.0737417899253642872281
Sum2 (naive)         : 0.0227373675443232059479
Sum3 (alter)         : 1.0737417936325073242188

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

lec10/addition.c

- Implementujte s využitím knihovny gmp.

Part II

Část 2 – Rychlost výpočtu (programu)

Outline

Maticové násobení - Naivně

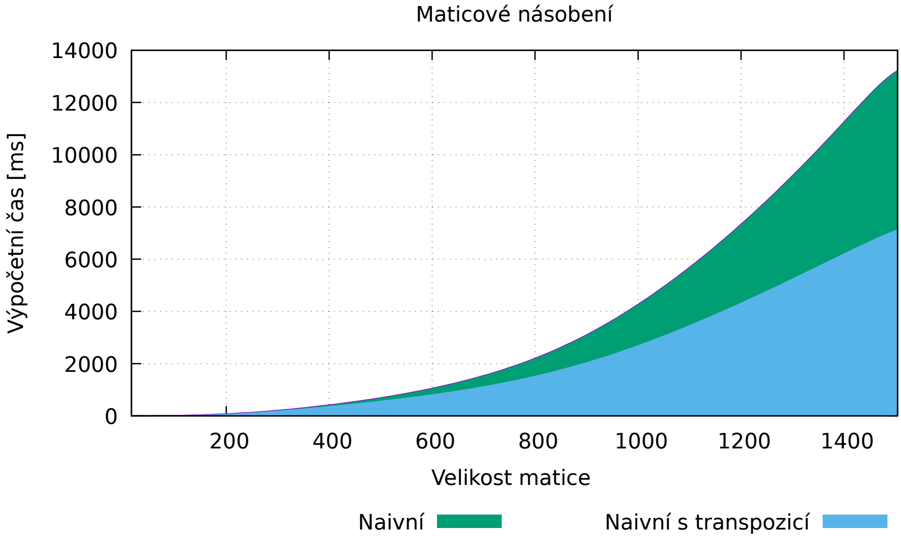
```c
void simple_multiply(const int n, const double *a, const double *b, double *c)
{
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            double prod = 0;
            for (int k = 0; k < n; ++k) {
                prod += a[i * n + k] * b[k * n + j];
            }
            c[i * n + j] = prod;
        }
    }
}
```

- Pro přehlednost předpokládáme kompatibilní rozměry matic a správně alokované.

Maticové násobení - Naivně s transpozicí

```
void simple_multiply_trans(const int n, const double *a, const double *b, double *c)
{
   double * bT = create_matrix(n); // allocate memory for transposed matrix
   for (int i = 0; i < n; ++i) {
      bT[i*n + i] = b[i*n + i];
      for (int j = i + 1; j < n; ++j) {
         bT[i*n + j] = b[j*n + i];
         bT[j*n + i] = b[i*n + j];
      }
   }
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         double tmp = 0;
         for (int k = 0; k < n; ++k) {
            tmp += a[i*n + k] * bT[j*n + k];
         }
         c[i*n + j] = tmp;
      }
   }
   free(bT);
}
```

Porovnání rychlosti násobení matic



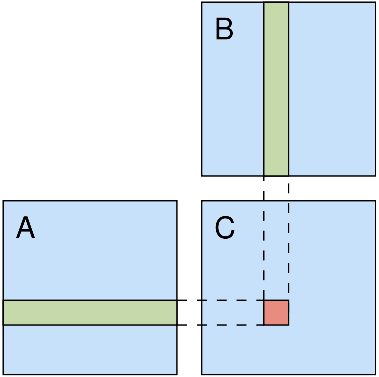
Outline

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 %.

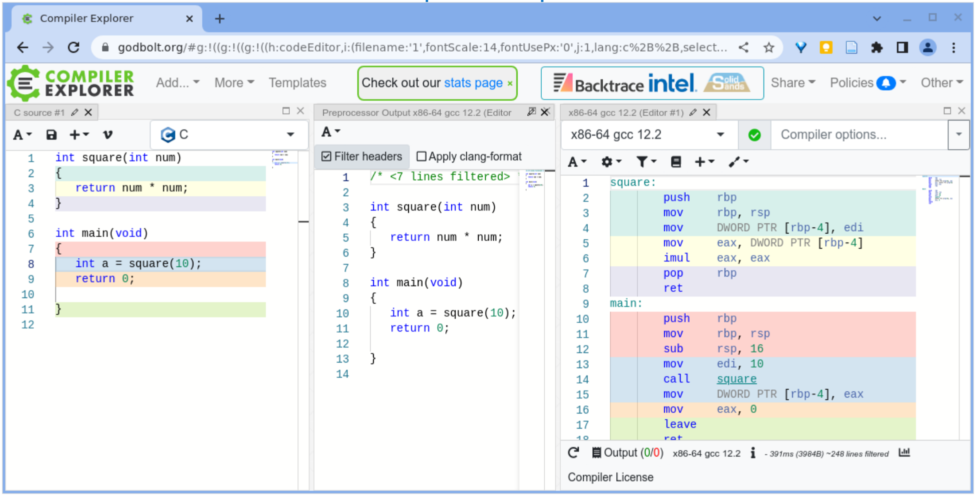
    https://arxiv.org/abs/1807.06735

    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



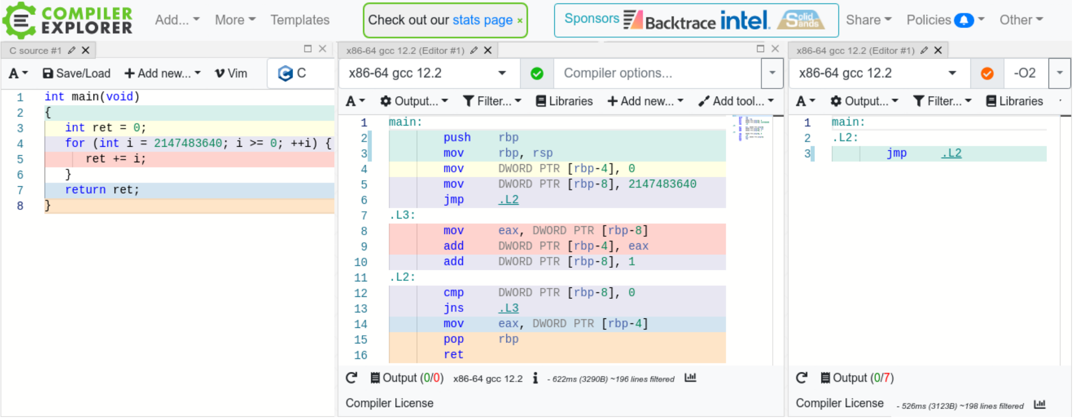
https://godbolt.org/z/K9r1eWqcd

Informativní

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

Outline

Maticové násobení

Rychlost výpočtu

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.

```c
void multiply(int n, int a[n][n], int b[n][n], int c[n][n])
{
   int i;
#pragma omp parallel private(i)
#pragma omp for schedule (dynamic, 1)
   for (i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         c[i][j] = 0;
         for (int k = 0; k < n; ++k) {
            c[i][j] += a[i][k] * b[k][j];
         }
      }
   }
}
```

lec10/demo-omp-matrix.c

Pro přehlednost uvažujeme čtvercové matice stejných rozměrů.

Příklad použití OpenMP - Maticové násobení 2/2

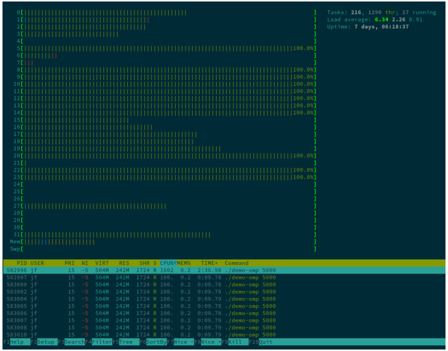
- Příklad násobení matic 1000 × 1000 s využitím OpenMP na iCore5 (2 jádra s HT ∼ 4× výpočetní jednotky).

```
gcc -std=c99 -O2 -o demo-omp demo-omp-matrix.c -fopenmp
./demo-omp 1000
Size of matrices 1000 x 1000 naive
        multiplication with O(n^3)
c1 == c2: 1
Multiplication single core 9.33 sec
Multiplication  multi-core 4.73 sec

OMP_NUM_THREADS=2 ./demo-omp 1000
Size of matrices 1000 x 1000 naive
        multiplication with O(n^3)
c1 == c2: 1
Multiplication single core 9.48 sec
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).



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

lec10/demo-omp-matrix.c

Shrnutí přednášky

Diskutovaná témata

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