

# PRK: Optimalizace

LenkaKT, NTI TUL, LS 2024

# Optimalizace

Optimalizace je úprava kódu s cílem vylepšit nějaké jeho parametry.

Cílem optimalizace je typicky:

1. Rychlejší výkon kódu za běhu
2. Menší nároky na paměť za běhu
3. Menší stopa v paměti pro celý kód
4. Menší spotřeba energie za běhu

Vždy vybíráme hlavní cíl, některé cíle si protiřečí.

Dobrý anglický popis: [https://en.wikipedia.org/wiki/Optimizing\\_compiler](https://en.wikipedia.org/wiki/Optimizing_compiler)

# Optimalizační postupy

- Lokální vs. celkové
- Optimalizace smyček:
  - Bude dále, minimalizujeme “zbytečné”, zrychlujeme průchod
- Optimalizace práce s úložištěm:
  - Analýza toku, minimalizace přenosu dat mezi pamětí a úložištěm
- Vkládání funkcí
  - Zrychlení - minimalizace skoku a ukládání kontextu
- “Trimovací” - v závěru podrobně zkoumáme kód a nahrazujeme více instrukcí jednou (násobení 2 - posun vpravo).

# Zrychlení kódu za běhu

- Vyndáme vše, co zdržuje:
  - Skoky (vyhodnocení podmínek)
  - Volání funkcí
  - Cykly
  - Nepotřebné instrukce a duplicity
  - Invariant cyklu
- Kontrolujeme tok dat:
  - Minimalizace přesunu mezi pamětí a registry
  - Úprava pořadí operací

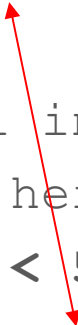
# Pravdivé podmínky

*Čím začínáme*

```
int i=1;
```

```
some i independent  
code here;
```

```
if (i < 5) then  
{  
    i = 0;  
}
```



*Čím končíme*

```
some i independent  
code here;
```

```
i = 0;
```

# Omezení skoků - Inlining

U malých cyklů ( $n < 20$ ) je výhodnější vložit přímo  $n$  opakování:

```
a=0;
```

```
for (i<=2;i=0;i++) {
```

```
    a++;}
```

```
a=0;
```

```
a++;
```

```
a++;
```

```
a++;
```

```
int sum (int a,int b)
```

```
{return a+b;}
```

```
x = sum(a,b)
```

```
x = a+b;
```

# Invariant cyklu

```
const int a=10;
```

```
int i,j,q;
```

```
q=0;
```

```
for (i=1,i<10,i++) {
```

```
    q = i+q;
```

```
    j = a;
```

```
}
```

```
const int a=10;
```

```
int i,j,q;
```

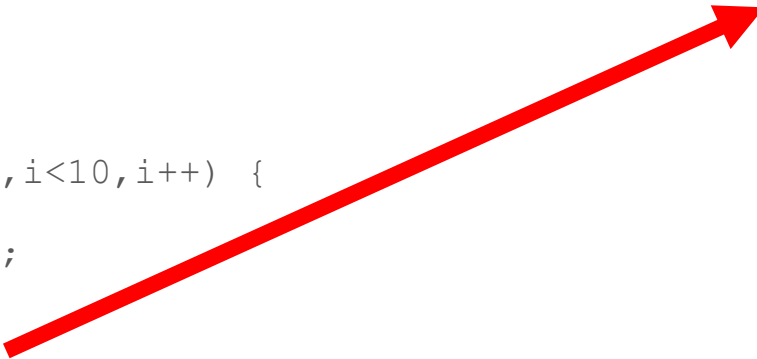
```
j = a;
```

```
q=0;
```

```
for (i=1,i<10,i++) {
```

```
    q = i+q;
```

```
}
```



# Duplicity

## *Čím začínáme*

```
int i=1;

/* spousta kódu č. 1, nemění i */

i = 1;

/* spousta kódu č. 2, nemění i */

i = 1;
```

## *Čím končíme*

```
i = 1;

/* spousta kódu č. 1, nemění i */

/* spousta kódu č. 2, nemění i */
```



# Optimalizace pořadí

```
int i, j, k;
```

```
k = (i + j) / (3 + (i + j));
```

Výraz  $i+j$  se zbytečně počítá 2x.



```
int i, j, k;
```

```
q = i + j;
```

```
k = q / (3 + q);
```

# Spočítat, co jde spočítat

```
const float pi = 3.14;
```

```
const float earth_radius = 6378;
```

```
earth_length=2*pi*earth_radius;
```

```
earth_length = 20026,92;
```

# Vektorizace kódu

```
for (i=0; i<1024; i++) {  
    c[i] = a[i] + b[i]  
}
```

```
for (i = 0; i < 1024; i+=4) {  
    C[i:i+3] = A[i:i+3]+B[i:i+3];  
}
```

Proměnné a, b, c jsou v paměti realizovány jako vektory a sčítány najednou (speciální instrukce)

Nutná podpora HW i překladače

Nutná analýza smyček a závislosti

# Menší nároky na paměť za běhu

- Závisí i na HW: souvisí s cache, predikcí kódu, využitím registrů
- Opakovaný výpočet hodnot (neukládám mezivýsledky apod.)
- Optimalizace uložení polí

```
for i = 0 to N-1
```

```
    for j = 0 to N-1
```

```
        A[j][i] = i*j;
```

```
for j = 0 to N-1 /* Zde indexace umožní vektorizaci, čte se po vektorech */
```

```
    for i = 0 to N-1
```

```
        A[j][i] = i*j;
```

[https://en.wikipedia.org/wiki/Loop\\_interchange](https://en.wikipedia.org/wiki/Loop_interchange)

# Malá paměťová stopa

1. Odstranění všeho nepotřebného
2. Odstranění všech duplicit
3. Výpočet všech dílčích výrazů

Pozor: Malá paměťová stopa často implikuje dlouhé trvání kódu:

Nutí nás vynechat inlining.

Nutí nás nechat funkce a skoky.

# Spotřeba energie

- Hodně architekturně vázané
- **Minimalizace přenosu dat mezi pamětí a procesorem**
- Tlak na využití cache
- Upravuje se pipeline
- Každá instrukce “něco stojí”
- Dá se ladit pořadí instrukcí
- Dá se pracovat s využitím (napájením) sběrnic - například sběrnic k paměťovým čipům apod.

# Optimalizace a matematika

Výpočty typicky zahrnují násobení a sčítání nad vektory

Lze využít standardní optimalizační volby

Existují specializované frameworky pro tyto typy optimalizací

Existují speciální verze knihoven, psané optimálně pro dané užití.

Stále méně se optimalizuje „ručně“

# Hardwarově závislé optimalizace

Typicky práce s registry CPU (Intel A,B,C vs. ARM R0-R7).

Vyhodnocení “životnosti” - live range - hodnoty, podle toho rozhodujeme o přemísťování z a do paměti.

Posuzování pomocí grafů.

Algoritmicky složité.

[https://en.wikipedia.org/wiki/Register\\_allocation](https://en.wikipedia.org/wiki/Register_allocation)

Architektura	32b	64b
Intel	15	31
ARM	8	16
RISC-V	16 (32)	32

Srovnání architektur CPU vzhledem k počtům registrů (zdroj: Wikipedia.org)



# Co si musím pamatovat

- Optimalizace **velmi silně** mění výsledný kód
- Silná optimalizace => těžké ladění (nelze krokovat)
- Typicky pro ladění překlad bez optimalizací
- Optimalizovaný kód se musí znovu testovat!

# Optimalizace - GCC

Rychlý přehled:

<https://www.rapidtables.com/code/linux/gcc/gcc-o.html>

Přesný přehled:

<https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options>

<https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/compiler-options/compiler-option-details/optimization-options/o.html>