Vektorové instrukce

3. dubna 2019

B4B36PDV – Paralelní a distribuované výpočty

Osnova

- Opakování z minulého cvičení

- Autovektorizace
- Ruční vektorizace pomocí intrinsics

Opakování z minulého cvičení

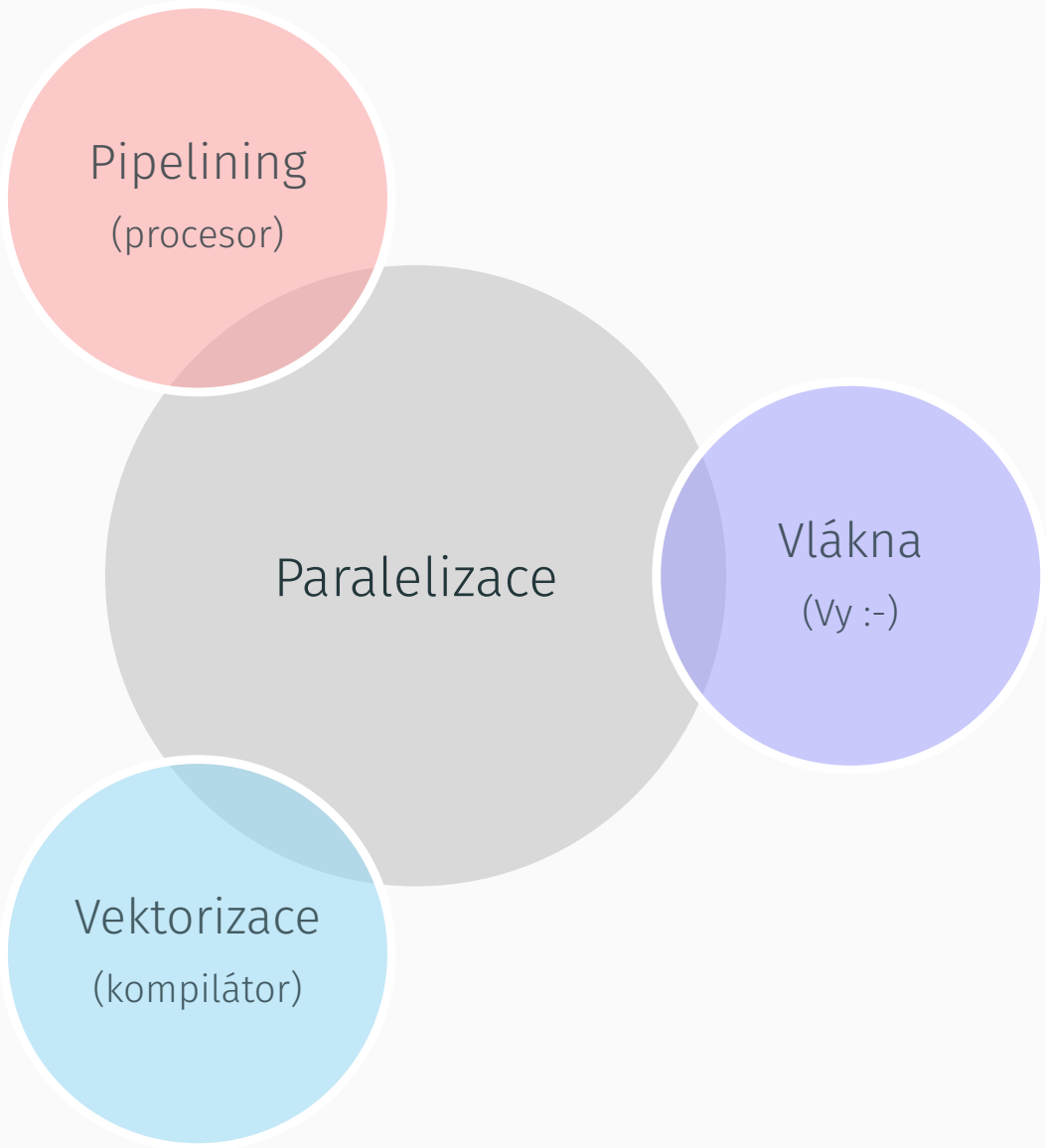
http://goo.gl/a6BEMb

Jakým způsobem bude následující kód proveden?

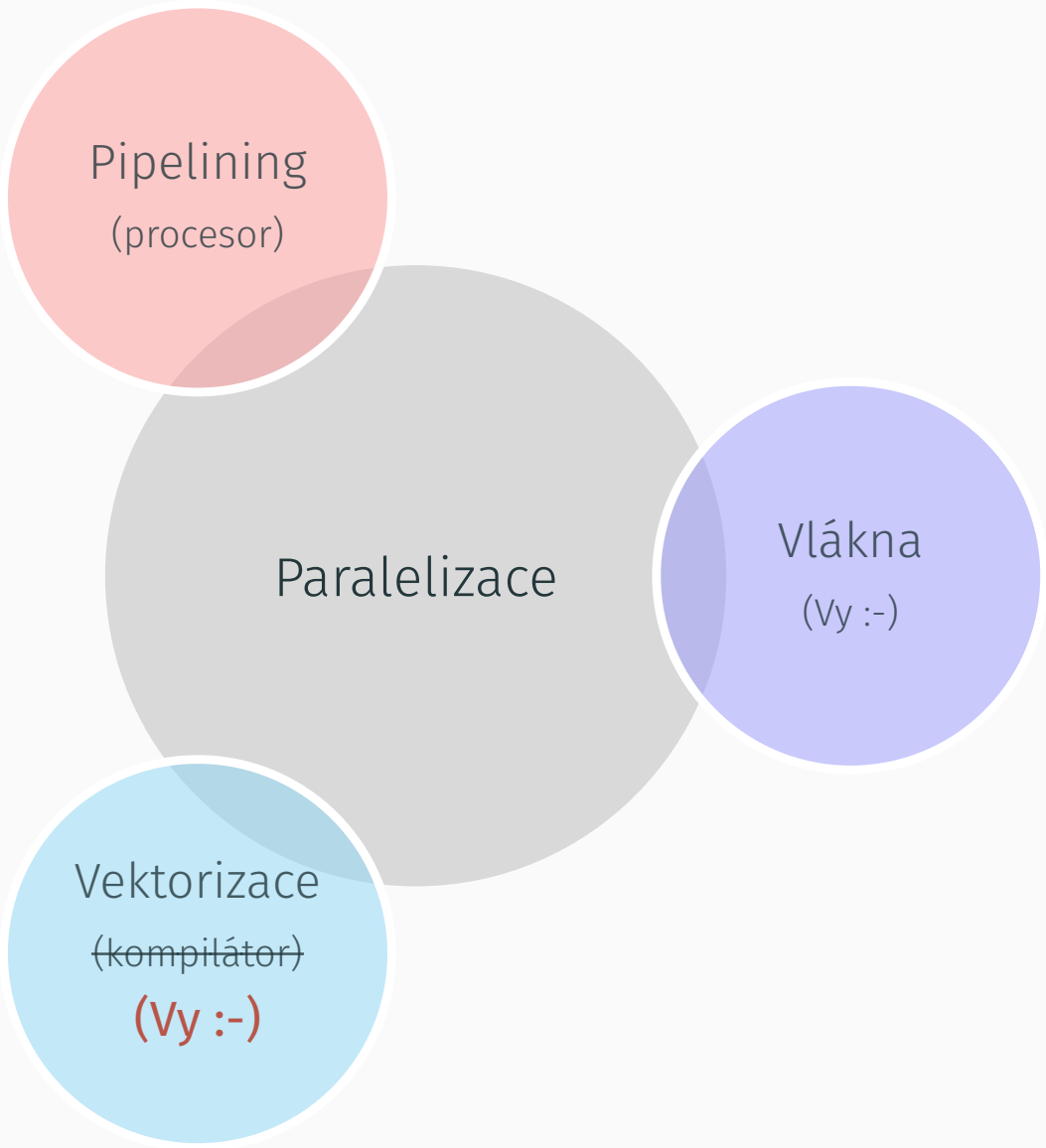
```
bool mat[M][N];
for(int i = 0; i < M; i++) {
    #pragma omp parallel
    #pragma omp for
    for(int j = 0; j < N; j++){
        #pragma omp cancellation point for
        if (mat[i][j]){
            #pragma omp cancel for
        }
    }
}
std::cout << "Finished!" << std::endl;
```

- Výpočet končí okamžitě po nalezení prvního řešení.
- Po nalezení prvního řešení výpočet skončí, až všechna vlákna narazí na 'cancellation point'.
- Ani jedna z předchozích odpovědí není správná.

Moderní procesor



Moderní procesor



Skalární zpracování dat

$$\texttt{float}\ x = \boxed{0.5\text{f}}$$

$$\texttt{float}\ y = \boxed{1.2\text{f}}$$

$$(\texttt{float})\ x + y = \boxed{1.7\text{f}}$$

Vektorové zpracování dat

`__m256` $x =$	0.5f	0.2f	0.6f	0.0f	1.5f	1.3f	2.5f	0.3f
`__m256` $y =$	1.2f	1.8f	0.2f	0.0f	1.2f	0.3f	2.4f	0.3f

(`__m256`) `_mm256_add_ps(`x`,`y`)=`	1.7f	2.0f	0.8f	0.0f	2.7f	1.6f	4.9f	0.6f

Vektorové zpracování dat

`__m256` $x =$	0.5f	0.2f	0.6f	0.0f	1.5f	1.3f	2.5f	0.3f
`__m256` $y =$	1.2f	1.8f	0.2f	0.0f	1.2f	0.3f	2.4f	0.3f

(`__m256`) `_mm256_add_ps`$(x,y) =$	1.7f	2.0f	0.8f	0.0f	2.7f	1.6f	4.9f	0.6f

Není to až taková magie, jak to vypadá :-)

Vektorové zpracování dat (pomocí AVX / AVX2)

```
#include <immintrin.h>
```

- `__m256` - datový typ „vektor délky 256 bitů“
  (`float` má 32 bitů, a proto se do takového vektoru vejde 8x)

Vektorové zpracování dat (pomocí AVX / AVX2)

```
#include <immintrin.h>
```

- `__m256` - datový typ „vektor délky 256 bitů“
  (`float` má 32 bitů, a proto se do takového vektoru vejde 8x)
- `_mm256_add_ps(`*x*`,`*y*`)`
  - Nad dvěma 256-bitovými vektory *x* a *y*... (`_mm256_`)
  - ...provádím operaci sčítání... (`add`)
  - ...při čemž vektory obsahují elementy typu *packed-single* (`_ps`).

Vektorové zpracování dat (pomocí AVX / AVX2)

```
#include <immintrin.h>
```

- `__m256` - datový typ „vektor délky 256 bitů“
  (`float` má 32 bitů, a proto se do takového vektoru vejde 8x)
- `_mm256_add_ps(`*x*`,`*y*`)`
  - Nad dvěma 256-bitovými vektory *x* a *y*... (`_mm256_`)
  - ...provádím operaci sčítání... (`add`)
  - ...při čemž vektory obsahují elementy typu *packed-single* (`_ps`).

- *packed* – vektor „zabaluje“ více prvků stejného typu
- *single – single-precision number* aka `float`

Level 1: Autovektorizace GCC 

Moderní kompilátor se snaží zdetekovat `for` smyčky, které lze vektorizovat...

Například:

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

lze převést na

```
for(int i = 0 ; i < 1024 ; i += 8) {
  _mm256_storeu_ps(&a[i],
    _mm256_add_ps(
      _mm256_loadu_ps(&b[i]),
      _mm256_loadu_ps(&c[i])
  ));
}
```

Vyzkoušejte si autovektorizaci GCC

Zkompilujte soubor `autovec.cpp` s parametry kompilace:

1. `-std=c++11 -march=native -O2` a
2. `-std=c++11 -march=native -O2 -ftree-vectorize`

Jak se program zrychlí, pokud použijete autovektorizaci (`-ftree-vectorize`)?

Můžete si i vyzkoušet použití parametru `-fopt-info-vec-all` (detailní výstup o proběhlé vektorizaci).

+ Je to „zadarmo“ (kompilátor se pokusí vektorizaci provést za Vás)

+ Je to „zadarmo“ (kompilátor se pokusí vektorizaci provést za Vás)

- Ne vždy se to kompilátoru musí povést...

+ Je to „zadarmo“ (kompilátor se pokusí vektorizaci provést za Vás)

- Ne vždy se to kompilátoru musí povést…
  - Kompilátor vám nemusí „rozumět“
    (často dokáže vektorizovat jenom smyčky v určitém tvaru)

+ Je to „zadarmo“ (kompilátor se pokusí vektorizaci provést za Vás)

- Ne vždy se to kompilátoru musí povést…
  - Kompilátor vám nemusí „rozumět“
    (často dokáže vektorizovat jenom smyčky v určitém tvaru)
  - Kompilátor musí zajistit, že výsledek programu bude identický, jako kdyby nevektorizoval **i za těch nejhorších možných podmínek**
    - Musí uvažovat, že může dojít k datovým závislostem
    - Musí zajistit, že dojde ke stejnému zaokrouhlení při floating-point operacích

+ Je to „zadarmo“ (kompilátor se pokusí vektorizaci provést za Vás)

- Ne vždy se to kompilátoru musí povést…
  - Kompilátor vám nemusí „rozumět“
    (často dokáže vektorizovat jenom smyčky v určitém tvaru)
  - Kompilátor musí zajistit, že výsledek programu bude identický, jako kdyby nevektorizoval **i za těch nejhorších možných podmínek**
    - Musí uvažovat, že může dojít k datovým závislostem
    - Musí zajistit, že dojde ke stejnému zaokrouhlení při floating-point operacích

---

```
float x;
float y1 = x * x * x * x * x * x * x * x;

float y2 = x * x;
y2 = y2 * y2;
y2 = y2 * y2;

assert(y1 == y2);
```

Level 2: Intel SPMD Compiler (a jiné) 

Intel SPMD Compiler (ISPC)

Tušíte co znamená zkratka SPMD?

Intel SPMD Compiler (ISPC)

Tušíte co znamená zkratka SPMD?

SPMD = *single-program multiple-data*

Napíšete jeden program, který ale pomocí vektorizace poběží na více daty současně. Kompilátor za vás rozhodne, jak má vektorizace proběhnout.

Intel SPMD Compiler (ISPC)

- Nadstavba jazyka C
- Od základu uvažuje o programu jako o paralelním!

Intel SPMD Compiler (ISPC)

- Nadstavba jazyka C
- Od základu uvažuje o programu jako o paralelním!

Bohužel nemáme čas se ISPC na PDV věnovat :-(

Level 3: Intrinsics 

Intrinsics

Intrinsics – Funkce a datové typy, které zpřístupňují nativní instrukce procesoru **bez nutnosti programovat v assembleru**

Instrukční sada: AVX / AVX2

Intrinsics

Intrinsics – Funkce a datové typy, které zpřístupňují nativní instrukce procesoru **bez nutnosti programovat v assembleru**

Instrukční sada: AVX / AVX2

`https://intel.ly/2GOHp7r` (Intel Intrinsics Guide)

Výborná reference! Využívejte, když si nebudete jistí!

```
#include <immintrin.h>
```

⚠ Všechny kódy kompilujte s `-march=native` !

AVX / AVX2 intrinsics

Datový typ vektor: `__m256...`

- `__m256` – vektor obsahující 8 x 32bit `float`
- `__m256d` – vektor obsahující 4 x 64bit `double`
- `__m256i` – vektor obsahující celočíselné typy

Načtení a zápis 256 bitů (8 x 32bit `float`) z/do adresy `float * x`:

```
__m256 data = _mm256_loadu_ps(x);
_mm256_storeu_ps(x, data);
```

AVX / AVX2 intrinsics

Datový typ vektor: `__m256...`

- `__m256` – vektor obsahující 8 x 32bit `float`
- `__m256d` – vektor obsahující 4 x 64bit `double`
- `__m256i` – vektor obsahující celočíselné typy

Načtení a zápis 256 bitů (8 x 32bit `float`) z/do adresy `float * x`:

```
__m256 data = _mm256_loadu_ps(x);
_mm256_storeu_ps(x, data);
```

Doimplementujte načtení a zápis dat do metody `normaldist_vec(...)`

Do těla `for` smyčky v metodě `normaldist_vec(...)` v souboru `prob.cpp` doimplementujte načtení a zpětný zápis `__m256` vektoru z adresy `&data[i]`.

Načítat a ukládat stejná data je nuda...

Doimplementujte výpočet hustoty normálního rozdělení

Pro každý prvek načteného vektoru spočtěte hodnotu funkce

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

`__m256 _mm256_set1_ps(x)`

Nastaví všechny prvky vektoru na `x`

`__m256 _mm256_add_ps(x, y)`, `__m256 _mm256_sub_ps(x, y)`
`__m256 _mm256_mul_ps(x, y)`, `__m256 _mm256_div_ps(x, y)`

Vypočte součet, rozdíl, součin a podíl vektorů `x` a `y`

Pro aproximaci $\exp(x)$ (vektorově) použijte `__m256 exp_vec(x)`

$$\exp(x) \approx \frac{(x+3)^2 + 3}{(x-3)^2 + 3}$$ (2, 2)-Padé aproximátor

Podmíněné zpracování

Občas chceme zpracovat různé prvky různým způsobem...

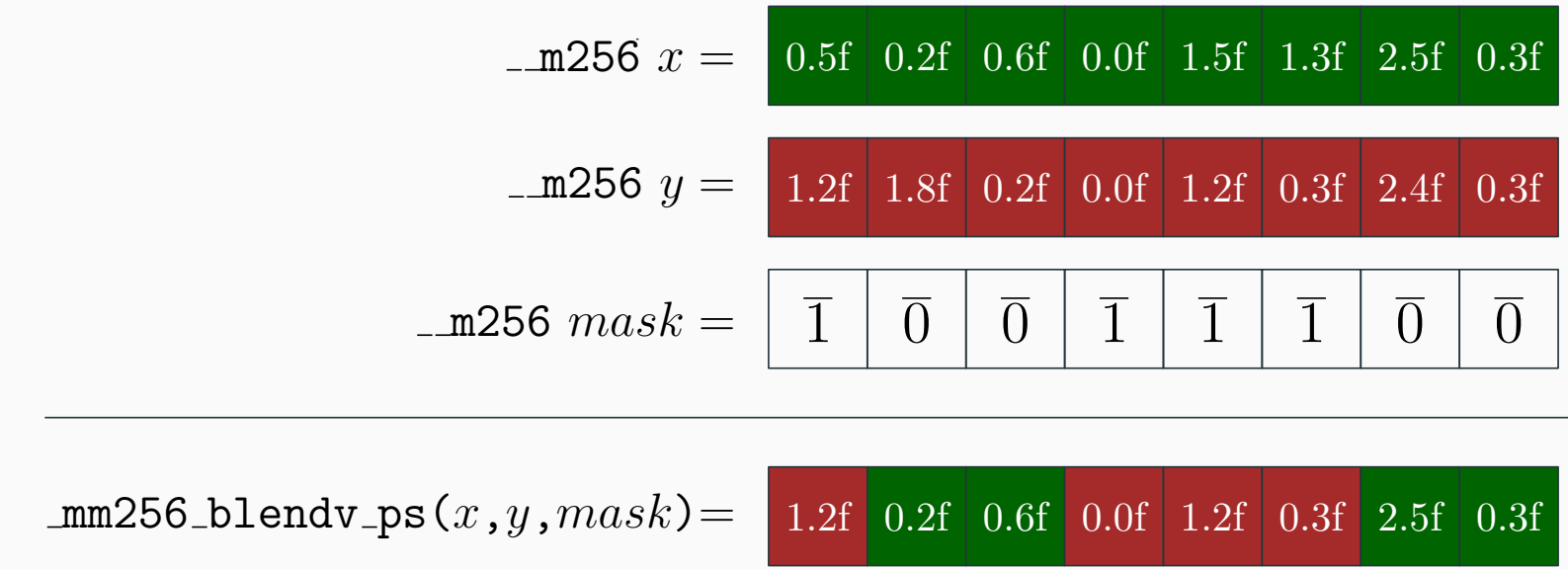
```
size_t half = N / 2;
for(unsigned int i = 0 ; i < half ; i++) {
    if(data[i] > data[i+half])
      std::swap(data[i], data[i+half]);
}
```

Podmíněné zpracování

Občas chceme zpracovat různé prvky různým způsobem...

```
size_t half = N / 2;
for(unsigned int i = 0 ; i < half ; i++) {
    if(data[i] > data[i+half])
      std::swap(data[i], data[i+half]);
}
```

`__m256 _mm256_blendv_ps(x, y, mask)`:



Podmíněné zpracování

Doimplementujte tělo metody `condswap_vec(...)`

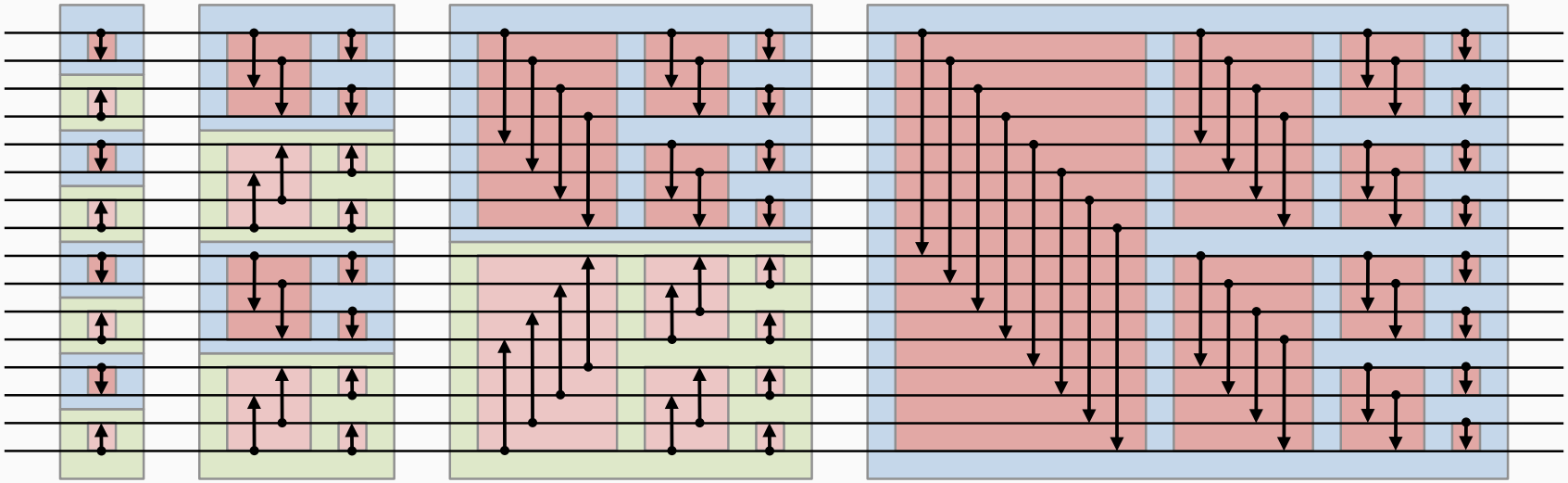
Doimplementujte tělo metody `condswap_vec(...)` v souboru `cond.cpp`, která bude vektorově vykonávat následující kód

```
size_t half = N / 2;
for(unsigned int i = 0 ; i < half ; i++) {
    if(data[i] > data[i+half])
      std::swap(data[i], data[i+half]);
}
```

Pro implementaci podmínky využijte `_mm256_blendv_ps(x,y,mask)`.

Vektorová instrukce pro porovnání vektorů $x < y$ typu *packed-single*:
`__m256 _mm256_cmp_ps(x, y, _CMP_LT_OQ)`

Bitonic sort



Bitové operace

S primitivními vektorovými instrukcemi jste se setkali už dříve!

například `x & y` nebo `x ^ y`

Bitové operace

S primitivními vektorovými instrukcemi jste se setkali už dříve!

například `x & y` nebo `x ^ y`

My se podíváme na něco zajímavějšího...

Bitové operace

```
_lzcnt_u64(uint64_t x)
_tzcnt_u64(uint64_t x)
```

Počet *leading*, resp. *trailing zeros* v čísle `x`

Například:

```
_lzcnt_u64(0b00001000 ... 00011100) = 4
_tzcnt_u64(0b00001000 ... 00011100) = 2
```

Bitové operace

`_lzcnt_u64(uint64_t x)`
`_tzcnt_u64(uint64_t x)`

Počet *leading*, resp. *trailing zeros* v čísle `x`

Například:

```
_lzcnt_u64(0b00001000 ... 00011100) = 4
_tzcnt_u64(0b00001000 ... 00011100) = 2
```

Doimplementujte tělo metody `log2_lzcnt(...)`

Doimplementujte tělo metody `log2_lzcnt(...)` v souboru `lzcnt.cpp`. Pro `x > 0` má tato metoda provést výpočet ekvivalentní `(int)log2(x)`.

Tip: Jaký vztah má pozice nejvyššího jedničkového bitu k hodnotě logaritmu o základu 2?

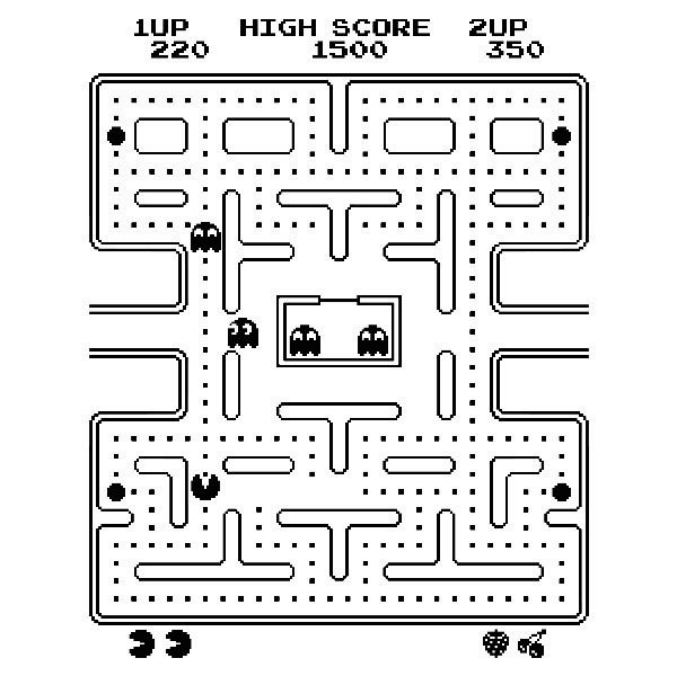
Bitové operace

```
_mm_popcnt_u64(uint64_t x)
```

Počet jedničkových bitů v čísle `x`

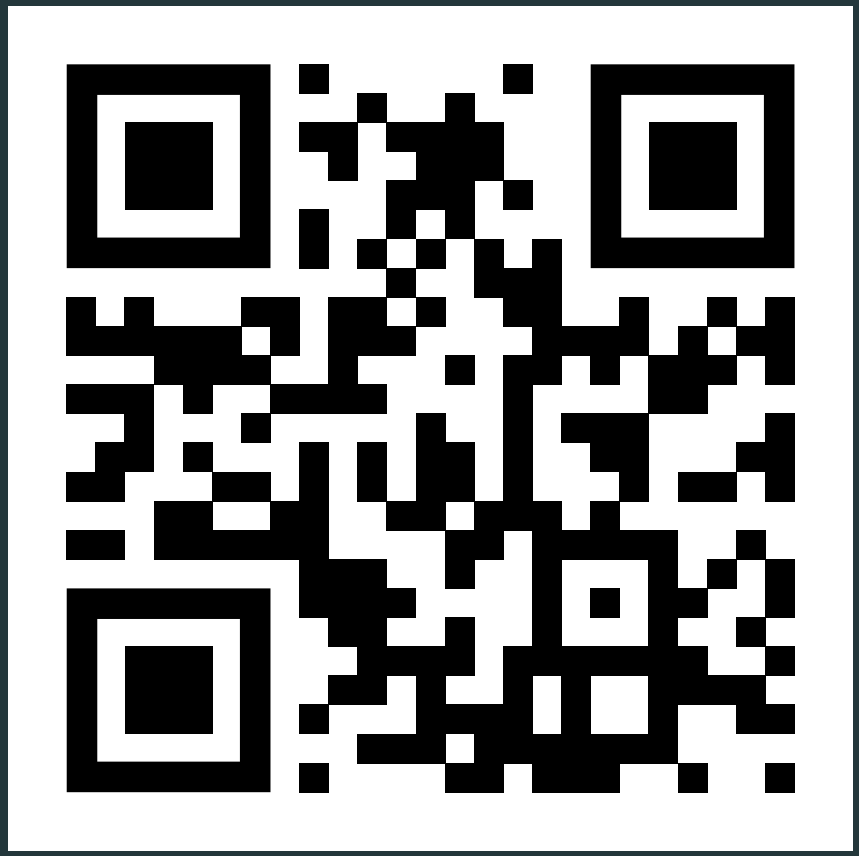
Semestrální úloha

Prohledávání stavového prostoru



Díky za pozornost!

Budeme rádi za Vaši zpětnou vazbu! →



http://bit.ly/2FFZsdm