

Vektorové instrukce

Cvičení 7

B4B36PDV – Paralelní a distribuované výpočty
FEL ČVUT

- Opakování z minulého cvičení
- Autovektorizace
- Ruční vektorizace pomocí intrinsics

Opakování z minulého cvičení

<http://goo.gl/a6BEMb>

Který způsob je efektivnější?

```
bool mat[M][N];

// A:
#pragma omp parallel
#pragma omp for
for(int i = 0; i < M; i++) {
    for(int j = 0; j < N; j++){
        if (mat[i][j]){ /* report solution and terminate all; */ }
    }
}

// B:
for(int i = 0; i < M; i++) {
    #pragma omp parallel
    #pragma omp for
    for(int j = 0; j < N; j++){
        if (mat[i][j]){ /* report solution and terminate all; */ }
    }
}
```

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;
```

Možné odpovědi:

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

Paralelizace:

Pipelining (procesor)

Vektorizace (kompilátor)

Vlákná (Vy 😁)

Paralelizace:

Pipelining (procesor)

Vektorizace (kompilátor)

(Vy 😊)

Vlákná (Vy 😊)

`float x = 0.5f`

`float y = 1.2f`

`(float) x + y = 1.7f`

$$_m256\ x = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0.5f & 0.2f & 0.6f & 0.0f & 1.5f & 1.3f & 2.5f & 0.3f \\ \hline \end{array}$$

$$_m256\ y = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1.2f & 1.8f & 0.2f & 0.0f & 1.2f & 0.3f & 2.4f & 0.3f \\ \hline \end{array}$$

$$(_m256)\ _mm256_add_ps(x, y) = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1.7f & 2.0f & 0.8f & 0.0f & 2.7f & 1.6f & 4.9f & 0.6f \\ \hline \end{array}$$

$$_m256\ x = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0.5f & 0.2f & 0.6f & 0.0f & 1.5f & 1.3f & 2.5f & 0.3f \\ \hline \end{array}$$

$$_m256\ y = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1.2f & 1.8f & 0.2f & 0.0f & 1.2f & 0.3f & 2.4f & 0.3f \\ \hline \end{array}$$

$$(_m256)\ _mm256_add_ps(x, y) = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1.7f & 2.0f & 0.8f & 0.0f & 2.7f & 1.6f & 4.9f & 0.6f \\ \hline \end{array}$$

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

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

```
__m256
```

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

```
#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).

```
#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

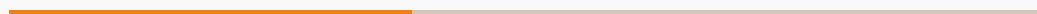
single-precision number aka float

Otestujte, jaká vektorová rozšíření podporuje váš procesor.

Použijte program `07_0test_simd_support` (`0test_simd_support.cpp`) a zjistěte jakou velikost integerových a floatových vektorů váš procesor podporuje.

Level 1: Autovektorizace

?



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])  
        ));  
}
```

Parametry kompilátoru GCC

`-march=native`

zapne kompilaci přímo na konkrétní hw, včetně zpřístupnění vektorových instrukcí.

`-ftree-vectorize`

zapne autovektorizaci.

`-fopt-info-vec-all`

informace o autovektorizaci.

`-O2`

musíme snížit level optimalizace, abychom mohli kontrolovat autovektorizaci.

Vyzkoušejte si autovektorizaci

Spusťte `07_1autovectorization` (`1autovectorization.cpp`) s autovektorizací a následně zkuste autovektorizaci vypnout:

GCC: zakomentujte v souboru `CMakeLists.txt` řádek s `"-ftree-vectorize"`.

Clang: zakomentujte v souboru `CMakeLists.txt` řádek s `"-fvectorize"` `"-fslp-vectorize"`.

Jak se program zpomalí, pokud vypnete autovektorizaci?

Také se podívejte do logu ze sestavování programu na zprávy o proběhlé autovektorizaci.

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

Výhody a nevýhody autovektorizace

- + 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...

Co limituje autovektorizaci?

- 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 * x;  
float y2 = x * x;  
y2 = y2 * y2;  
y2 = y2 * y2;  
assert(y1 == y2);
```

Level 2: Intel SPMD Compiler



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.

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.

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

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.

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

Bohužel nemáme čas se ISPC na PDV věnovat :-)

Level 3: Intrinsic



Intrinsics

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

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

Instrukční sada: AVX / AVX2

Intrinsics

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

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

Instrukční sada: AVX / AVX2

⚠ V GCC je třeba všechny kódy kompilovat s `-march=native` !

Intrinsics

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

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

Instrukční sada: AVX / AVX2

⚠ V GCC je třeba všechny kódy kompilovat s `-march=native` !

Intel Intrinsics Guide: <https://intel.ly/2G0Hp7r>

Výborná reference! Využívejte, když si nebudete jistí!

Intrinsics test

Vyzkoušejte vektory pomocí intrinsics v programu `07_2intrinsics_test` (`2intrinsics_test.cpp`).

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 `3normal_distribution.cpp` doimplementujte načtení a zpětný zápis `__m256` vektoru z adresy `&data[i]`.

Výpočet hustoty normálního rozdělení

Načítat a ukládat stejná data je nuda...

Doimplementujte výpočet hustoty normálního rozdělení

Pro každý prvek načteného vektoru spočtete 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)`

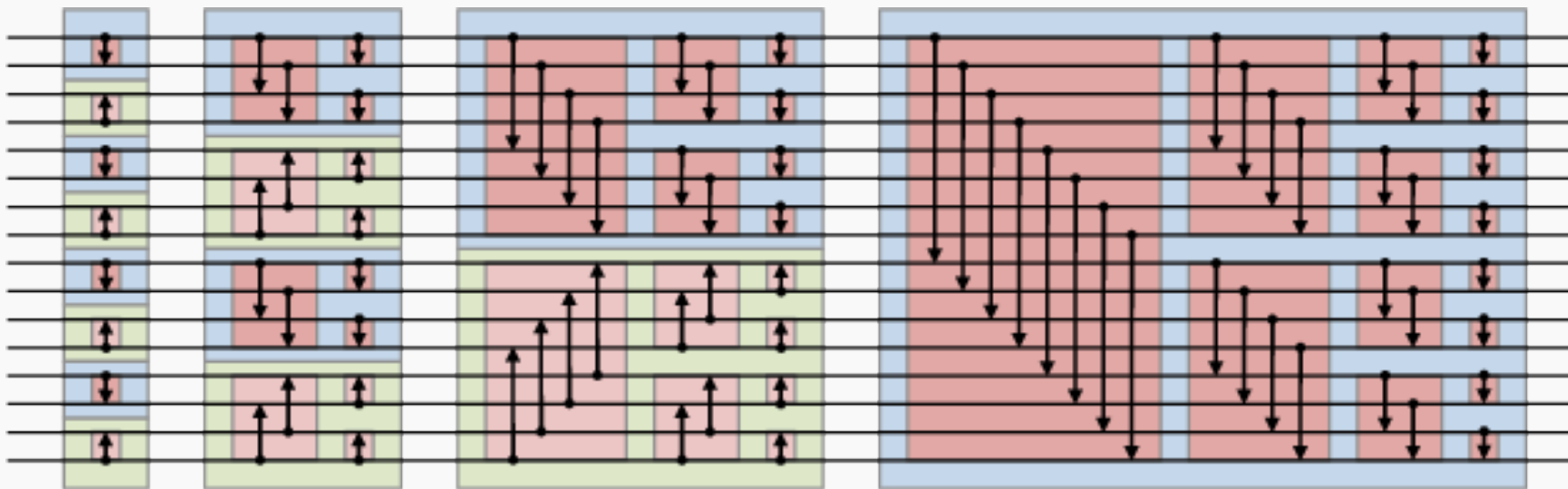
`__m256 __mm256_mul_ps(x, y), __m256 __mm256_div_ps(x, y)`

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

Paralelní řazení bitonic sort



Součástí řazení je i podmíněné prohazování prvků v poli: zjednodušená verze na dalších slidech.

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]);
}
```


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):
```

`__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 mask =`

1	0	0	1	1	1	0	0
---	---	---	---	---	---	---	---

`_mm256_blendv_ps(x, y, mask) =`

1.2f	0.2f	0.6f	0.0f	1.2f	0.3f	2.5f	0.3f
------	------	------	------	------	------	------	------

Doimplementujte tělo metody `condswap_vec(...)`

Doimplementujte tělo metody `conditional_swap_vec(...)` v souboru `4conditional_swap.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)`

S primitivními vektorovými instrukcemi jste se setkali už dříve!

například $x \& y$ nebo $x \wedge y$

S primitivními vektorovými instrukcemi jste se setkali už dříve!

například $x \& y$ nebo $x \wedge y$

My se podíváme na něco zajímavějšího...

```
_mm_popcnt_u64(uint64_t x)
```

Počet jedničkových bitů v čísle *x*

Doimplementujte tělo metody `popcnt_intrinsic(...)`

Doimplementujte tělo metody `popcnt_intrinsic(...)` v souboru `popcnt.cpp`.

$$y = Ax$$

Typická operace, pro kterou je vektorizace velmi výhodná...

Násobení matice vektorem

Prostudujte `5matrix.cpp`.

Semestrální úloha

Prohledávání stavového prostoru

