Змістовний модуль:

СУЧАСНІ ПРОЦЕСОРИ ТА ПАРАЛЕЛІЗАЦІЯ ОБЧИСЛЕНЬ

Розділ: Використання SIMD команд для паралельних обчислень

ЛЕКЦІЯ 7. SIMD КОМАНДИ. ПРОДОВЖЕННЯ

ПИТАННЯ ДЛЯ ВИВЧЕННЯ

1. Комплексні операції
2. SSE2 та AVX 2.
3. Функції порівняння даних.
4. Функції для роботи з бітами.
5. Функції округлення.
6. Функції перемішування.

КОМПЛЕКСНІ ФУНКЦІЇ

Функція для додавання добутків деяких компонентів полів:

__m128 _mm_dp_ps(__m128 a, __m128 b, const int mask);

__m256 _mm256_dp_ps(__m256 a, __m256 b, const int maska);

Обчислює суму добутків.

Які добутки обчислюються задає старші 4 біта маски: 1 – компонент блоку перемножується. Якщо треба обчислювати добутки усіх компонентів 1111

Молодші 4 біта задають поля, куди результат після обчислення суми

Решта полів результату – 0.

Для AVX довжина маски залишилася без змін, тому маска задає зразу 2 поля.

КОМПЛЕКСНІ ФУНКЦІЇ

Приклад 1

```
a128 = _mm_setr_ps(1., 2., 3., 4.); b128 = _mm_setr_ps(8., 9., 10., 11.);
c128 = _mm_dp_ps(a128, b128, 0xFA);
for (int i = 0; i < 4; i++)
printf("%g\n", c128.m128_f32[i]);
```

Результат

….

Приклад 2

```
a256 = _mm256_setr_ps(1., 1., 2., 3., 4., 5., 6., 7.);
b256 = _mm256_setr_ps(8., 9., 10., 11., 12., 13., 14., 15.);
c256 = _mm256_dp_ps(a256, b256, 0x3A);
for (int i = 0; i < 8; i++)
printf("%g\n", c256.m256_f32[i]);
```

ДОДАТКОВІ ФУНКЦІЇ

Приклад 1

a128 = _mm_setr_ps(1., 2., 3., 4.); b128 = _mm_setr_ps(8., 9., 10., 11.);

c128 = _mm_dp_ps(a128, b128, 0xFA);

for (int i = 0; i < 4; i++)

printf("%g\n", c128.m128_f32[i]);

Результат

Маска : 1111

1 * 8 + 2 * 9 + 3 * 10 + 4 * 11 = 100

C128[0] = C128[2] = 100; C128[1] = C128[3] = 0;

Приклад 2

a256 = _mm256_setr_ps(1., 1., 2., 3., 4., 5., 6., 7.);

b256 = _mm256_setr_ps(8., 9., 10., 11., 12., 13., 14., 15.);

c256 = _mm256_dp_ps(a256, b256, 0x3A);

for (int i = 0; i < 8; i++) printf("%g\n", c256.m256_f32[i]);

Маска 0011. поля 0 и 1 ; 5 и 6 (1 * 8 + 1 * 9) и 12 * 4 + 13 * 5

Результаты записывают в поля c256 [1] = c256 [3] = 17; c256 [5] = c256 [7] =113. решта - 0

ДОДАТКОВІ ФУНКЦІЇ

Приклад 3

Скласти функцію для обчислення скалярного добутку з використанням звичайних обчислень, SSE, AVX звичайних та функції _mm256_dp_ps

ДОДАТКОВІ ФУНКЦІЇ

Приклад 3

Скласти функцію для обчислення скалярного добутку з використанням звичайних обчислень, SSE, AVX звичайних та додаткових операцій

```
float smSSE(float *x, float *y, size_t n){
	__m128 *px = (__m128 *)x, *py = (__m128 *)y, s128 = _mm_setzero_ps();
	float *s = (float*)&s128;
	for (size_t i = 0; i < n / 4; i++)
		s128 = _mm_add_ps(s128, _mm_mul_ps(px[i], py[i]));
	s128 = _mm_hadd_ps(s128, s128);
	return s[0] + s[1];
}
float smAVX1(float *x, float *y, size_t n){
	__m256 *px = (__m256 *)x, *py = (__m256 *)y, s256 = _mm256_setzero_ps();
	float *s = (float*)&s256;
	for (size_t i = 0; i < n / 8; i++)
		s256 = _mm256_add_ps(s256, _mm256_mul_ps(px[i], py[i]));
	s256 = _mm256_hadd_ps(s256, s256); s256 = _mm256_hadd_ps(s256, s256);
	return s[0] + s[4];
//dest [0]=s1[0]+s1[1]; dest[1]=s1[2]+s1[3]; dest[2]=s2[0]+s2[1]; dest[3]=s2[2]+s2[3];
//dest [4]=s1[4]+s1[5]; dest[5]=s1[6]+s1[7]; dest[6]=s2[4]+s2[5]; dest[7]=s2[6]+s2[7];
}
```

ДОДАТКОВІ ФУНКЦІЇ

Приклад 3

Скласти функцію для обчислення скалярного добутку з використанням звичайних обчислень, SSE, AVX звичайних та додаткових операцій

```
// Функція з використанням комплексної AVX операції _mm256_dp_ps
float smAVX2(float *x, float *y, size_t n){
    __m256 *px = (__m256 *)x, *py = (__m256 *)y;
    __m256 s256 = _mm256_setzero_ps();
    float *s = (float*)&s256;
    for (size_t i = 0; i < n / 8; i++)
        s256 = _mm256_add_ps(s256, _mm256_dp_ps(px[i], py[i], 0xF1));
    return s[0] + s[4];
}
```

Часові характеристики

Sm 0.0028 smSSE 0.0012 smAVX1 0.00071 smAVX2 0.00071

ЗАВДАННЯ ДАНИХ ЦІЛОГО ТИПУ

#include <intrin.h> Структура даних (визначена в файлі *emmintrin.h*)

```
typedef union __declspec(intrin_type)
    __declspec(align(16)) __m128i {        // Вирівнювання
 __int8 m128i_i8[16];                      // signed char
 __int16 m128i_i16[8];                     // signed short
__int32 m128i_i32[4];                      // int
 __int64 m128i_i64[2];                     // __int64
 unsigned __int8 m128i_u8[16];             // unsigned char
 unsigned __int16 m128i_u16[8];            // unsigned short
 unsigned __int32 m128i_u32[4];            // unsigned int
 unsigned __int64 m128i_u64[2];            // unsigned __int64
} __m128i; .
```

ЗАВДАННЯ ДАНИХ ЦІЛОГО ТИПУ

#include <intrin.h> **128 →256**

```
typedef union  __declspec(intrin_type) _CRT_ALIGN(32) __m256i {
    __int8              m256i_i8[32];
    __int16             m256i_i16[16];
    __int32             m256i_i32[8];
    __int64             m256i_i64[4];
    unsigned __int8     m256i_u8[32];
    unsigned __int16    m256i_u16[16];
    unsigned __int32    m256i_u32[8];
    unsigned __int64    m256i_u64[4];
} __m256i;
```

ЗАГАЛЬНИЙ ВИД ІМЕНІ ФУНКЦІЇ

<Префікс>_<Код>[s|r]_{ep|s}<Тип ><Довжина >,

де:

Код - код операції, наприклад, *add, sub*;

[s|r] - *s* задається для команд із насиченням і не задається для звичайних команд;

r (reverse) – обробка елементів масиву у зворотному порядку;

{ep|s} - *ep* для роботи з масивом і *s* для роботи з одним даним;

Тип - визначає тип даних, буква *i* означає тип *int*, буква *u* - *unsigned*;

Довжина - довжина компонента в бітах (8, 16, 32, 64, 128).

Приклад:

Функція додавання для блоку даних типу *int* без насичення:

_mm_add_epi32

_mm256_add_epi32

АРИФМЕТИЧНІ ФУНКЦІЇ

1. **+, -** (Для усіх довжин, з насиченням та без)
2. ***Горизонтальне +, -*** (_mm_hadd_epi16, _mm_hadds_epi16, _mm_hadd_epi32)-r[i]=a[2i]+a[2i+1]; (i=0..3); r[i+4]=b[2i]+b[2i+1]; (i=0..3);
3. ***Обчислення середнього*** (_mm_avg_epu8, _mm_avg_epu16)
4. ***Комбіновані операції*** (_mm_madd_epi16 - r[i]=(a[2i] * b[2i]) +(a[2i+1] * b[2i+1])(i = 0..3))
5. ***Множення*** для молодших та старших частин (_mm_mulhi_epi16, _mm_mullo_epi16, _mm_mulhi_epi32, _mm_mullo_epi32, _mm_mul_epi32 (R[0] = low(a[0] * b[0]); r[1] = high(a[0] * b[0]); r[2] = low(a[2] * b[2]); r[3] = high (a[2] * b[2]);))
6. ***Abs*** (8, 16, 32)
7. ***Sign*** (8, 16, 32) (R[i]=b[i] < 0? –a[i]: b[i] == 0? 0 : a[i];)
8. SSE4: ***Min, max*** (8, 16, 32), _mm_minpos_epu16 (r [0] = min, r [1] = pos)

АРИФМЕТИЧНІ ФУНКЦІЇ. ПРИКЛАД

Дослідити вплив типу даних на продуктивність без використання й з використанням SSE

Цей приклад передбачає, що буде виконуватись одна й та ж операція (+) для усіх відповідних даних масиву. При цьому визначаються залежність часових характеристик від типу даних (char, unsigned, int, __int64).

Розмір масиву фіксований.

АРИФМЕТИЧНІ ФУНКЦІЇ. ПРИКЛАД

```
// Звичайний режим

template <typename TYPE>
VOID Add(CONST TYPE *a, CONST TYPE *b, TYPE *c, size_t n){

    size_t i; for (i = 0; i < n; ++i) c[i] = a[i] + b[i];

}
```

АРИФМЕТИЧНІ ФУНКЦІЇ. ПРИКЛАД

```
VOID SSECHARAdd(CONST char *a, CONST char *b, char *c, size_t n){
        size_t i; __m128i *pa = (__m128i *)a, *pb = (__m128i *)b, *pc = (__m128i *)c;
        for (i = 0; i < n / 16; ++i) pc[i] = _mm_add_epi8(pa[i], pb[i]);
}
VOID SSESHORTAdd(CONST short *a, CONST short *b, short *c, size_t n){
        size_t i; __m128i *pa = (__m128i *)a, *pb = (__m128i *)b, *pc = (__m128i *)c;
        for (i = 0; i < n / 8; ++i) pc[i] = _mm_add_epi16(pa[i], pb[i]);
}
VOID SSEINTAdd(CONST int *a, CONST int *b, int *c, size_t n){
        size_t i;
        __m128i *pa = (__m128i *)a, *pb = (__m128i *)b, *pc = (__m128i *)c;
        for (i = 0; i < n / 4; ++i) pc[i] = _mm_add_epi32(pa[i], pb[i]);
}
VOID SSELONGAdd(CONST __int64 *a, CONST __int64 *b, __int64 *c, size_t n)
{
        size_t i; __m128i *pa = (__m128i *)a, *pb = (__m128i *)b, *pc = (__m128i *)c;
        for (i = 0; i < n / 2; ++i) pc[i] = _mm_add_epi64(pa[i], pb[i]);
}
```

АРИФМЕТИЧНІ ФУНКЦІЇ. ПРИКЛАД

```
VOID AVXCHARAdd(CONST char *a, CONST char *b, char *c, size_t n){

    size_t i; __m256i *pa = (__m256i *)a, *pb = (__m256i *)b, *pc = (__m256i *)c;

    for (i = 0; i < n / 32; ++i) pc[i] = _mm256_add_epi8(pa[i], pb[i]);

}

…
VOID AVXLONGAdd(CONST __int64 *a, CONST __int64 *b, __int64 *c, size_t n)
{

    size_t i; __m256i *pa = (__m256i *)a, *pb = (__m256i *)b, *pc = (__m256i *)c;

    for (i = 0; i < n / 4; ++i) pc[i] = _mm256_add_epi64(pa[i], pb[i]);
}
```

АРИФМЕТИЧНІ ФУНКЦІЇ. РЕЗУЛЬТАТИ

Тип даних N= 10000000	Без SIMD	SSE	AVX
char	3.6	3.8	3.9
short	7.3	8.5	8.4
int	14.5	15.6	14.5
__int64	30	29	32

АРИФМЕТИЧНІ ФУНКЦІЇ. ВИСНОВКИ

1. Компілятор вбудовує відповідні команди для обчислення, тому для різних режимів використання отримуємо практично однакові результати
2. Швидкість обчислень суттєво залежить від типу даних. Чим менше розмір даних, тим швидше виконуються операції
3. **При обранні типу даних треба обирати тип з мінімальною можливою довжиною**

АРИФМЕТИЧНІ ФУНКЦІЇ. НАСИЧЕННЯ

Хай необхідно покомпонентно додати елементи масиву типу usigned char. Якщо результат більше ніж 255 – результат – 255

```
void Adds(unsigned char *src1, unsigned char *src2, int n, unsigned char *dest){
for (int i = 0; i < n; i++)
{
#if 0
unsigned short t1 = src1[i]; t1 += src2[i]; if (t1 > 255) t1 = 255;
dest[i] = (unsigned char)t1;
#else
dest[i] = src1[i] + src2[i]; if (dest[i] < src1[i]) dest[i] = 255;
#endif
}
}
```

АРИФМЕТИЧНІ ФУНКЦІЇ. НАСИЧЕННЯ

```
void SSEAdds(unsigned char *src1, unsigned char *src2, int n,
unsigned char *dest)
{
__m128i *ps1 = (__m128i *)src1, *ps2 = (__m128i *)src2, *pd =
(__m128i *)dest;
        for (int i = 0; i < n/16 ; i++)
        {
                pd[i] = _mm_adds_epu8(ps1[i], ps2[i]);
        }
}
```

Результати:

Adds – 130 мс

SSEAdds 4 мс

ФУНКЦІЇ ПОРІВНЯННЯ

_mm_cmpeq_epi{8, 16, 32, 64}
_mm256_cmpeq_epi{8, 16, 32, 64}

_mm_cmpgt_epi{8, 16, 32, 64}
_mm256_cmpgt_epi{8, 16, 32, 64}

0…. 1……

ФУНКЦІЇ ДЛЯ РОБОТИ З БІТАМИ

&, |, ^, &~

***_mm_<Code>_si128*,**

де ***Code*** – *and, or, xor, andnot*.

Приклади імен функцій: *_mm_xor_si128, _mm_and_si128.*

Наведені вище функції виконують відповідно операції додавання по модулю 2 та операцію логічного множення.

<<, >>

_mm_s{l|r}{a|l}[i]{i128|epi16|epi32|epi64}

l|r – напрям зсуву;

a|l – тип зсуву;

i – константа зсуву;

I128 – зсув цілком 128 біт;

epi – зсув окремого компонента блока.

ФУНКЦІЇ ДЛЯ РОБОТИ З БІТАМИ. ПРИКЛАД

Обчислити:

c= (b[0] & a[0]) ^ (b[1] & a[1]) … ^(b[n-1] & a[n-1]).

```
// Функція без SSE
void AndXor (unsigned a[][16], unsigned b[][16],  unsigned c[16], size_t n)
{
        int i, j; unsigned temp ;
        for (i = 0; i < 16; i++) c[i] = 0;
        for (i = 0; i < n; i++)        {
                for (j = 0; j < 16; j++){
                        temp  = a[i][j] & b [i][j]; c[j]^= temp ;
                }
        }
}
```

ФУНКЦІЇ ДЛЯ РОБОТИ З БІТАМИ. ПРИКЛАД SSE

Функція з SSE

```
void SSEAndXor (unsigned a[][16], unsigned b[][16], unsigned c[16], size_t n  )
{
      int i, j; __m128i  temp ;
      __m128i *pa = (__m128i *)a, *pb = (__m128i *)b, *pc = (__m128i *)c;
      for (i = 0; i < 4; i++)
            pc [i] = _mm_setzero_si128 ();
            for (i = 0; i < n; i++) {
                  for (j = 0; j < 4; j++) {
                        temp  = _mm_and_si128 (pa[j], pb [j]);
                        pc [j] = _mm_xor_si128 (pc [j], temp);
                  }
                  pb += 4; pa += 4;
            }
}
```

Результат:

AndXor	*130438*
SSEAndXor	331054;
Прискорення:	**S = 2.54**

ФУНКЦІЇ ДЛЯ РОБОТИ З БІТАМИ. ПРИКЛАД AVX

```
void AVXAndXor(unsigned a[][16], unsigned b[][16], unsigned c[16], size_t n){
    int i, j; __m256i temp;
    __m256i *pa = (__m256i *)a, *pb = (__m256i *)b, *pc = (__m256i *)c;
    pc[0] = _mm256_setzero_si256();
    pc[1] = _mm256_setzero_si256();
    for (i = 0; i < n; i++) {
        temp = _mm256_and_si256(pa[0], pb[0]);
        pc[0] = _mm256_xor_si256(pc[0], temp);
        temp = _mm256_and_si256(pa[1], pb[1]);
        pc[1] = _mm256_xor_si256(pc[1], temp);
        pb += 2; pa += 2;
    }
}
```

Результати:

AndXor	***112 mc***
SSEAndXor	***7.2 mc***
AVXAndXor	***7.1 mc***

ФУНКЦІЇ ПОРІВНЯННЯ ТА РОБОТИ З БІТАМИ.

Задано масив цілих чисел, які обчислені по заданому модулі q і мають значення 0.. q-1.

Перетворити цей масив таким чином, щоб значення були в інтервалі –q/2..q/2.

```
void ModConvert(int *src, int * dest, int q, size_t n){
int q_2 = q / 2;
for (size_t i = 0; i < n; i++){
    dest[i] = src[i];
    if (src[i] > q_2)
        dest[i] -= src[i] - q;
}
}
```

ФУНКЦІЇ ПОРІВНЯННЯ ТА РОБОТИ З БІТАМИ.

Задано масив цілих чисел, які обчислені по заданому модулі q і мають значення 0.. q-1.

Перетворити цей масив таким чином, щоб значення були в інтервалі –q/2..q/2.

```
void AVXModConvert(int *src, int * dest, int q, size_t n){
int q_2 = q / 2;
__m256i *psrc = (__m256i *)src, *pdest = (__m256i *)dest;
__m256i temp1 = _mm256_set1_epi32(q_2), temp2 = _mm256_set1_epi32(q), temp3,
    temp4, temp5;
for (int i = 0; i < n / 8; i++){
temp3 = _mm256_cmpgt_epi32(psrc[i], temp1);
temp4 = _mm256_andnot_si256(temp3, psrc[i]);
temp5 = _mm256_sub_epi32(psrc[i], temp2);
temp5 = _mm256_and_si256(temp5, temp3);
pdest[i] = _mm256_add_epi32(temp4, temp5);}}
```

ФУНКЦІЇ ЗСУВУ

$$\_mm\_s\begin{Bmatrix} l \\ r \end{Bmatrix}\begin{Bmatrix} l \\ a \end{Bmatrix}\begin{Bmatrix} i \\ v \end{Bmatrix}\begin{Bmatrix} \_si256 \\ \_epi\begin{Bmatrix} 16 \\ 32 \\ 64 \end{Bmatrix} \end{Bmatrix},$$

$$\_\mathrm{mm256}\_s\begin{Bmatrix} l \\ r \end{Bmatrix}\begin{Bmatrix} l \\ a \end{Bmatrix}\begin{Bmatrix} i \\ v \end{Bmatrix}\begin{Bmatrix} \_si256 \\ \_epi\begin{Bmatrix} 16 \\ 32 \\ 64 \end{Bmatrix} \end{Bmatrix},$$

$\begin{Bmatrix} l \\ r \end{Bmatrix}$ **- напрям зсуву;**

$\begin{Bmatrix} l \\ a \end{Bmatrix}$ **- тип зсуву, арифметичний або логічний;**

$\begin{Bmatrix} i \\ v \end{Bmatrix}$ **- спосіб завдання на скільки зсув**

$\begin{Bmatrix} \_si256 \\ \_epi\begin{Bmatrix} 16 \\ 32 \\ 64 \end{Bmatrix} \end{Bmatrix}$ **- що зсувається.**

ФУНКЦІЇ ЗСУВУ. ПРИКЛАД

Написати оператори для по компонентного циклічного зсуву елементів AVX блоку на 11 бітів вліво (це одна з операцій, яку використовує поширений алгоритм шифрування ГОСТ 28147)

ФУНКЦІЇ ЗСУВУ. ПРИКЛАД

Step 1. Виділити старші 11 бітів, для цього треба виконати операцію & над заданим числом та маскою 0xFFE00000;

Step 2. Виконати покомпонентний зсув результату Step 1 вправо на 21 біт (32 - 11). Зсув логічний;

Step 3.Виконати покомпонентний зсув заданого числа на 11 бітів вліво.

Step 4. Виконати операцію | над результатами Step 2 та Step 3.

ФУНКЦІЇ ЗСУВУ. ПРИКЛАД

```
const __m256i maska11 =
_mm256_set1_epi32(0xFFE00000);
DestNumber =
_mm256_or_si256(
_mm256_srli_epi32(
_mm256_and_si256(SrcNumber, maska11),
 21),
_mm256_slli_epi32(DesNumber, 11)
        );
```

ФУНКЦІЇ ПЕРЕМІШУВАННЯ

__m128i ***_mm_shuffle_epi32*** *(__m128i a, int mask);*

_MM_SHUFFLE(z, y, x, w) – задає порядок компонентів блоку в блоці-результаті.

Приклад. «Зашифрувати» масив цілих чисел, переставивши числа в кожній четвірці у зворотному порядку.

Функція без використання SSE команд.

```
VOID Swap4 (const int *x, int *y, size_t n){
        for (size_t i = 0; i < n; i+= 4){
                y [i] = x [i + 3]; y [i + 1] = x [i + 2];y [i + 2] = x [i + 1];y [i + 3] = x [i ];
        }
}
```

Функція з використанням SSE функцій

```
VOID SSESwap4 (const int *x, int *y, size_t n){
    __m128i *px = (__m128i *)x, *py = (__m128i *)y;
    for (size_t i = 0; i < n / 4; ++i)
        py[i]=_mm_shuffle_epi32(px[i],_MM_SHUFFLE(0,1,2,3));
}
```

Swap4:134351, SSESwap4:478120 . Прискорення 3.55

Float: 109139; *SSEDoubleEncrypt* 455676. Прискорення 4.17

ВИЗНАЧЕННЯ ХАРАКТЕРИСТИК ПРОЦЕСОРІВ З ПОГЛЯДУ ПІДТРИМКИ SIMD КОМАНД

- **Навіщо визначати ці властивості?**
- Як показав аналіз різних типів *SIMD* команд, деякі з процесорів підтримують всі типи команд, у списку типів яких є *__m64*, а підтримка різних версій функцій типу SSE залежить не від виробника й розрядності процесора, а від його версії. Можна звичайно використовувати тільки ті команди, які підтримуються всіма процесорами, але тоді прийдеться відмовитися від *SIMD* команд взагалі. Це нерозумно з двох точок зору.(**Enable Intrinsic Functions**)
- 1. Більшість сучасних процесорів підтримують ці команди;
- 2. Використання команд цієї групи дуже ефективно не тільки за рахунок паралельної обробки цілого блоку даних, але й за рахунок того, що для роботи із цими командами використовується окремий конвеєр, часто навіть не один.

ЯК ВРАХОВУВАТИ ВЛАСТИВОСТІ ПРОЦЕСОРІВ?

Код бібліотеки

#if defined (AVX_512)

//Визначення функцій з використанням AVX-512

#elif defined (AVX2)

//Визначення функцій з використанням AVX-512

#elif defined (__SSE4)

#else...

…

#endif

#endif

КОМАНДА *CPUID* І ФУНКЦІЇ *__cpuid,* __cpuidex

Команда *CPUID* використовується для визначення властивостей процесора. Команда без параметрів. Дії, що виконуються командою, залежать від умісту регістра *EAX*, що повинен бути сформований до команди *CPUID*. Інколи для визначення функції додатково використовується регістр ECX.

Результати роботи функцій повертаються в регістрах EAX, EBX, ECX, EDX.

Команди діляться на звичайні й розширені.

Щоб довідатися, які команди підтримуються заданим процесором, необхідно довідатися максимальне значення регістра *EAX*, яких можна задавати в команді.

Для визначення максимального номера звичайної команди, яку можна використовувати, необхідно в регістр *EAX* записати число 0. Після виконання команди *CPUID* у регістрі *EAX* одержимо максимальний номер звичайної команди.

ФУНКЦІЇ __*cpuid*, __cpuidex (INTRIN.H)

Для більш простого використання команди *CPUID* можна використовувати синтаксис мови *C++*, для цього необхідно:

- Підключити заголовний файл *intrin.h*

#include < intrin.h >

- Виділити масив даних типу *int* розміром чотири елементів для вмісту регістрів *EAX, EBX, ECX, EDX* (результати).
- Використовувати функції:

void __cpuid(int a[4], int b),

void __cpuidex(int a[4], int b, int c)

де

a – виділений раніше масив цілих для запису результатів (*a[0]* відповідає регістру *EAX, a[1] –EBX, a[2] - ECX, a [3] - EDX*);

b – номер функції (те, що заноситься в регістр *EAX* до виконання функції).

Функцію __*cpuid* можна використовувати, якщо команда процесора підтримується. Ім'я функції __*cpuid*, як і кожної функції мови *C++*, регістро - залежне. Надалі будемо використовувати функції __*cpuid,* __***cpuidex*** замість команди *CPUID*.

ВИЗНАЧЕННЯ MAX НОМЕРІВ ФУНКЦІЙ

```
int MaxFunctionNumber (int *ExtMaxFunctionNumber)
{
    int res = -1;
    if (CPUIDSupport ())
    {
        int b = 0x80000000; int a [4];
        // Якщо ррозширена інформація потрібна
        if (ExtMaxFunctionNumber) {
            __cpuid (a, b); *ExtMaxFunctionNumber = a[0];
        }
        b = 0; __cpuid (a, b); res = a [0];
    }
    return res;
}
```

I3 I5: **MaxOrinaryFunct = 13 ExtMaxFunctionNumber = 80000008.**

I7?

I9?

ТАБЛИЦЯ ОСНОВНИХ ФУНКЦІЙ

Функція (b)	Призначення
0	a [0] – максимальний номер звичайної функції; a [1], a [3], a [2] – тип процесору. Intel: “GenuineIntel” (оригінальний Intel), AMD: “AuthenticAMD” (справжній AMD)
1	a [2] –0 : SSE3; 9 :SSSE3; 19 :SSE4.1; 20 :SSE4.2 a [3] –23: MMX; 25: SSE; 26: SSE2; 28 : Super Threading
0x80000002 - 0x80000004	a [0], a [1], a [2], a [3] – повна назва процесору та його тактова частота (INTEL, AMD)

ВИЗНАЧЕННЯ ТИПУ ПРОЦЕСОРУ

```
typedef enum {
    UNKNOWN,    INTEL,    AMD
} PROCERRORTYPE;
PROCERRORTYPE GetProcessorType (){
    PROCERRORTYPE ProcessorType =  UNKNOWN;
    char Etalons [2][13] = { "GenuineIntel",          "AuthenticAMD" };
    int Regs[4]; Regs [0] = MaxFunctionNumber (0);
    if (Regs [0] != -1) {
        __cpuid( Regs, 0); int iVendor [3];
        iVendor [0] = Regs [1]; iVendor [1] = Regs [3]; iVendor [2] = Regs [2];
        char *cVendor = (char *) iVendor;
        if (strncmp (cVendor, Etalons [0], 12) == 0) ProcessorType = INTEL;
        else if (strncmp (cVendor, Etalons [1], 12) == 0) ProcessorType = AMD;
    }
    return ProcessorType;
}
```

ПОВНЕ ІМ'Я ПРОЦЕСОРУ

```
bool GetExtProcessotType (char *pExtProcessotType){
    bool b = false; int Regs [4]; int ExtMaxFunNumber;
    Regs [0] = MaxFunctionNumber (&ExtMaxFunNumber);
    if (Regs [0] != -1 && (unsigned)ExtMaxFunNumber >=0x80000004) {
        char *pCur = pExtProcessotType; __cpuid( Regs, 0x80000002);
        memcpy (pCur, Regs, sizeof (Regs)); pCur += sizeof (Regs);
        __cpuid( Regs, 0x80000003);
        memcpy (pCur, Regs, sizeof (Regs)); pCur += sizeof (Regs);
        __cpuid( Regs, 0x80000004);
        memcpy (pCur, Regs, sizeof (Regs)); pCur += sizeof (Regs);
        *pCur = 0;          b = true;
    }
    return b;
}
```

ТАБЛИЦЯ КОМАНД ДЛЯ SIMD ТА AVX ОПЕРАЦІЙ

Тип команд	Функція	Регістр	Біт
SSE	1	EDX	25
SSE2	1	EDX	26
SSE3	1	ECX	0
SSSE3	1	ECX	9
SSE4.1	1	ECX	19
SSE4.2	1	ECX	20
AVX , XSAVE	1	ECX	26
AVX , OSXSAVE	1	ECX	27
AVX	1	ECX	28
AVX2	7	EBX	5
AVX-512	?		

ОСОБЛИВОСТІ ВИЗНАЧЕННЯ ПІДТРИМКИ ДЛЯ AVX ОПЕРАЦІЙ

1. Необхідно перевірити можливості процесора та ОС:

1.1 Наявність команди _xgetbv (відповідна функція є для VS - 2013)

Windows 7 (SP Діск)

Windows 8

#if (_MSC_FULL_VER >= 160040219)

…

ФУНКЦІЯ ДЛЯ ВИЗНАЧЕННЯ ПІДТРИМКИ SIMD КОМАНД. ЧАСТИНА 1

```
typedef enum
{
SSESUPPORT,
SSE2SUPPORT,
SSE3SUPPORT,
SSSE3SUPPORT,
SSE41SUPPORT,
SSE42SUPPORT,
SSE4ASUPPORT,
AVXSUPPORT,
AVX2SUPPORT,
} SIMDSUPPORT;
```

ФУНКЦІЯ ДЛЯ ВИЗНАЧЕННЯ ПІДТРИМКИ SIMD КОМАНД. ЧАСТИНА 2

```
iint SIMDSupport (){
        // AVX
        SIMDSUPPORT res = (SIMDSUPPORT)0;
        int cpuInfo [4];
        __cpuidex(cpuInfo, 1, 0);
        if (((cpuInfo [2] >> 26) & 15) == 15)
                res = AVX2SUPPORT;
        else if (((cpuInfo [2] >> 26) & 7) == 7)
                res = AVXSUPPORT;
        if (res != (SIMDSUPPORT)0){
                unsigned XCR0Value = GetXCR0 ();
                if (((XCR0Value >> 2)&1) == 1)
                        return res;
                res = (SIMDSUPPORT)0;
        }
…
```

ФУНКЦІЯ ДЛЯ ВИЗНАЧЕННЯ ПІДТРИМКИ SIMD КОМАНД. ЧАСТИНА 2

```
iint SIMDSupport (){
        // AVX
        SIMDSUPPORT res = (SIMDSUPPORT)0;
        int cpuInfo [4];
        __cpuidex(cpuInfo, 1, 0);
        if (((cpuInfo [2] >> 26) & 15) == 15)
                res = AVX2SUPPORT;
        else if (((cpuInfo [2] >> 26) & 7) == 7)
                res = AVXSUPPORT;
        if (res != (SIMDSUPPORT)0){
                unsigned XCR0Value = GetXCR0 ();
                if (((XCR0Value >> 2)&1) == 1)
                        return res;
                res = (SIMDSUPPORT)0;
        }
…
```

РЕАЛЬНИЙ ПРИКЛАД ВИКОРИСТАННЯ SIMD КОМАНД

Перетворення масиву коефіцієнтів поліному в байтовий масив

Дано.

Поліном має цілі коефіцієнти, задані за модулем m (просте число) як дані

В інтервалі –m/2..m/2.

Функція повинна перетворити ці коефіцієнти до цілих в інтервалі 0,, m-1 а потім їх представити як масив байтів

3 варіанта рішення.

1 без використання SIMD операцій;

2 Виконувати цикл

Для наступного даного типу __m256i виконати перетворення інтервалу

Для отриманого блоку виконати упаковку

3

Виконувати цикл

Для наступного даного типу __m256i виконати перетворення інтервалу

Виконувати цикл для упаковки усіх даних

РЕАЛЬНИЙ ПРИКЛАД ВИКОРИСТАННЯ SIMD КОМАНД

Перетворення масиву коефіцієнтів поліному в байтовий масив

Дано.

Поліном має цілі коефіцієнти, задані за модулем m (просте число) як дані

В інтервалі –m/2..m/2.

Функція повинна перетворити ці коефіцієнти до цілих в інтервалі 0,, m-1 а потім їх представити як масив байтів

3 варіанта рішення.

1 без використання SIMD операцій; time = 0.0013

2 Виконувати цикл

Для наступного даного типу __m256i виконати перетворення інтервалу

Для отриманого блоку виконати упаковку time = 0.03 (S = 0.04)

3

Виконувати цикл

Для наступного даного типу __m256i виконати перетворення інтервалу

Виконувати цикл для упаковки усіх даних time = 0.00066 (S = 2)

РЕКОМЕНДАЦІЇ З ВИКОРИСТАННЯ SIMD КОМАНД

- *SIMD* команди є потужним механізмом сучасних процесорів, які можуть істотно збільшити продуктивність при роботі з масивами (в 2 і більше разів), якщо над елементами масиву необхідно виконувати однакові операції.
- *SIMD* команди виконуються одночасно зі звичайними командами, тому бажано в програмі чергувати *SIMD* команди й звичайні команди таким чином, щоб ці команди не залежали друг від друга. Якщо є залежність – навпаки, продуктивність падає!!!. Дивись приклад далі.
- Ефективність *SIMD* команд значно збільшується, якщо використовувати для роботи з ними вирівняні дані й відповідні функції.
- Використання *SIMD* функцій не приводить до накладних витрат, пов'язаним з викликом функцій, тому що всі вони є inline функціями.
- Використання *SIMD* команди не приводить до накладних витрат, пов'язаним з використанням потоків.
- Перед використанням *SIMD* команд необхідно обов'язково переконатися в тім, що процесор підтримує роботу з такими командами.
- Якщо додаток є 64 бітним додатком, то функції, які використовують тип *__m64*, не підтримуються.
- Якщо до додатка пред'являються підвищені вимоги по їхній продуктивності, краще мати динамічні бібліотеки для різних варіантів використання *SIMD* команд. Підключати ту бібліотеку, що найбільш повно використовує властивості процесора, на якому буде виконуватися даний додаток.

ВИСНОВКИ

- Найбільше розповсюдження мають SSE команди, які реалізовані для усіх сучасних процесорів загального призначення.
- В лекції досліджені деякі типи цих команд.
- Використання команд вимагає, щоб адреси початку масивів були вирівняні на границю даних типу __m128i, тобто ділилися на 16 або на 32 (__m256i).
- Прискорення суттєво залежить від типу цілих даних, які використовуються. Тому рекомендується завжди обирати мінімально достатній тип.
- Сучасні процесори мають декілька версій SSE команд. Перед використанням старших версій та AVX (AVX2) команд необхідно обов'язково перевіряти можливість їх використання. Для визначення властивостей процесору використовуються функції ***__cpuid (__cpuidex).***
- Якщо в масиві треба виконувати одну й ту ж операцію над усіма елементами масиву – завжди треба перевірити можливість використання SIMD команд для їх паралельної обробки

ПИТАННЯ ДЛЯ САМОСТІЙНОГО ВИВЧЕННЯ

1. Вивчить SIMD команди для порівняння елементів масивів. Дослідіть їх ефективність, зробіть висновки.
2. Дослідіть можливість використання SSE операцій Load та Store для виконання дій для масивів цілих чисел та перевірте їх ефективність.
3. Для Вашого процесору знайдіть документацію по команді CPUID та вивчіть функції цієї команди.
4. Знайдіть усі SSE команди відповідно Вашого типу процесору та дослідіть їх ефективність

МАТЕРІАЛИ ДЛЯ ЕКСПРЕС-КОНТРОЛЮ

1. Як вирівняти адресу масиву на задану границю?
2. Що робити, якщо довжина масиву не кратна розміру блоку?
3. Складіть програму для виконання усіх арифметичних операцій без використання та з використанням SSE та дослідіть залежність прискорення від операції, типу даних, розміру масивів (цілі дані).
4. Знайдіть серед команд команди для перетворення даних та дослідіть їх ефективність.
5. Вивчіть макроси для округлення даних та наведіть приклади їх використання.

ВИСНОВКИ

- Найбільше розповсюдження мають SSE та AVX команди, які реалізовані для усіх сучасних процесорів загального призначення.
- В лекції досліджені деякі типи цих команд.
- Використання команд вимагає, щоб адреси початку масивів були вирівняні на границю даних типу __m128 (__m256), тобто ділилися на 16 (32).
- Порівняння різних типів команд показує, що є команди, які дозволяють отримати прискорення дуже значне (наприклад, функції, пов'язані з обчисленням кореня квадратного). На жаль, є команди, які не дають прискорення. Таким чином, перед використанням має сенс досліджувати ефективність команд для конкретних масивів.
- Сучасні процесори мають декілька версій SSE команд. Перед використанням старших версій необхідно обов'язково перевіряти можливість їх використання. Як це робити – буде розглянуто нижче.
- Далі будуть розглянуті функції для роботи з цілими числами.

ПИТАННЯ ДЛЯ САМОСТІЙНОГО ВИВЧЕННЯ

1. Вивчить команди типу MMX, 3DNow!, порівняйте їх ефективність з відповідними командами SSE, зробіть висновки.
2. Дослідіть можливість використання SSE операцій для виконання дій для масивів комплексних чисел та перевірте їх ефективність.
3. Використовуючи MSDN знайдіть класи функцій для чисел з плаваючою точкою, які не розглянуті в курсі, вивчіть їх та перевірте ефективність їх використання.

МАТЕРІАЛИ ДЛЯ ЕКСПРЕС-КОНТРОЛЮ

1. Як вирівняти адресу масиву на задану границю?
2. Що робити, якщо довжина масиву не кратна розміру блоку?
3. Складіть програму для виконання усіх арифметичних операцій без використання та з використанням SSE та дослідіть залежність прискорення від операції, типу даних, розміру масивів.
4. Знайдіть серед команд команди для перетворення даних та дослідіть їх ефективність.
5. Вивчіть макроси для округлення даних та наведіть приклади їх використання.