Змістовний модуль:

СУЧАСНІ ПРОЦЕСОРИ ТА ПАРАЛЕЛІЗАЦІЯ ОБЧИСЛЕНЬ

Розділ: Використання SIMD команд для паралельних обчислень

ЛЕКЦІЯ 5. SIMD КОМАНДИ. ДАНІ З ПЛАВАЮЧОЮ ТОЧКОЮ

ПИТАННЯ ДЛЯ ВИВЧЕННЯ

1. Класифікація SIMD операцій.
2. SSE та AVX команди. Типи даних і класифікація функцій для чисел з плаваючою точкою.
3. Арифметичні функції.
4. Функції порівняння даних.
5. Функції для роботи з бітами.
6. Функції округлення.
7. Load та Store функції.
8. Функції перемішування.

ЗАГАЛЬНА ХАРАКТЕРИСТИКА ФУНКЦІЙ

SSE (Streaming SIMD Extensions)

SSE (Pentium 3 і вище) працює з даними з плаваючою точкою, регістри довжиною 128 біт, тобто одночасно можна опрацьовувати чотири або два числа залежно від їхньої точності, але не працюють з цілими числами.

SSE2, SSE3 (Pentium 4 і вище) працюють і з цілими, і з числами із плаваючою точкою.

У сучасних процесорах додатково реалізовані команди **SSSE3** – розширення команд від INTEL, часто в літературі називається **SSE4, SSE4.1, SSE4.2, SSE4A.** Кожна нова версія додає команди для роботи з 128 бітними даними.

AVX (Advanced Vector EXtensions (**128 →256 біт →*512* →*1024***)

Особливості функцій:

- використовують свої регістри, а не регістри із плаваючою точкою; (XMM – 128 біт, YMM – 256 біт, ZMM – 512 бітів)
- команди увесь час удосконалюються.
- працюють як для 32-бітних, так і для 64 бітних процесорів INTEL і AMD.
- використовують свій конвеєр, тому можуть виконуватися паралельно з іншими командами.

intrin.h

ПРИНЦИПИ РОБОТИ SIMD КОМАНД

4 байта (int)	4 байта (int)	4 байта (int)	4 байта (int)

+

4 байта (int)	4 байта (int)	4 байта (int)	4 байта (int)

=

4 байта (int)	4 байта (int)	4 байта (int)	4 байта (int)

4 байта	4 байта	4 байта	4 байта	4 байта	4 байта	4 байта	4 байта

+

4 байта	4 байта	4 байта	4 байта	4 байта	4 байта	4 байта	4 байта

=

4 байта	4 байта	4 байта	4 байта	4 байта	4 байта	4 байта	4 байта

ОСОБЛИВОСТІ ВИКОРИСТАННЯ ЧИСЕЛ З ПЛАВАЮЧОЮ ТОЧКОЮ. ТОЧНІСТЬ

Як використовуються паралельні програми?

1 Складається послідовна функція

2 Складається паралельна функція

3 порівнюються результати. Якщо вони співпадають – порівнюється час виконання.

Чим відрізняються способи:

1 Порядок обчислення

2 Правила обчислення

Як порівняти числа с плаваючою крапкою?

if (a == b)?

Послідовний (a + b)/3

Паралельний a/3 + b/3

Якщо обчислення з точністю до 1 цифри після точки – різні результати.

Як порівняти?

ОПЕРАЦІЇ ДЛЯ ЧИСЕЛ З ПЛАВАЮЧОЮ ТОЧКОЮ І ОКРУГЛЕННЯ

Звичайна арифметика – округлення.

Приклад. Хай необхідно задані числа округлити до 0.01:

- 3.472 = (3.472 + 0.005) = 3.47
- 3.475 = (3.475 + 0.005) = 3.48
- -3.472=(-3.472 – 0.005) =-3.47
- -3.475 = (-3.475 - 0.005) = -3.48

Правила:

x+y = y + x; x * (y + z) = x*y + x * z; (x –y)/z = x/z – y/z;

$x^2-y^2 = (x-y)(x+y)$

ОСОБЛИВОСТІ ВИКОРИСТАННЯ ЧИСЕЛ З ПЛАВАЮЧОЮ ТОЧКОЮ. ТОЧНІСТЬ

Чому точність обмежена?

Перед записом число записується в формі:

0

Нормалізоване число:1.mmmmmm * 2^p

Записується тільки цифри дробної частини

Float (32 – 1 знак, 8 біт визначають порядок числа 127 +p, решта – цифри після точки – 23 біта)

Похибка дорівнює $2^{-23}*2^p$.

Double (64 – 1 знак, 11 біт – х-ка, 52 біта – цифри після точки)

Числа завжди задаються не більше, ніж вони є.

Число 5.1

101.00011000110001100011 0 (5 + 1/16 + 1/32 + 1/512 + 1/1024+…)= 5.09999990

0.1 = 0.2=0.4=0.8=1.6=1.2

1024.2 =10000000000.0011001100110 = 1024.19995

0.2 = 0.4 = 0.8 = 1.6 = 1.2

ОСОБЛИВОСТІ ВИКОРИСТАННЯ ЧИСЕЛ З ПЛАВАЮЧОЮ ТОЧКОЮ. ТОЧНІСТЬ

Приклад 1.

Визначити результат:

```
float a=123456789.; // 0x75bcd15;
float b=123456788.; // 0x75bcd14;
f=a-b;
printf("Result: %f\n", f); // ??????
```

Приклад 2.

```
float  a = 123456789.12345678; // 0x75bcd15.1F9ADD2
double doublea = 123456789.12345678; // 0x75bcd15.1F9ADD2
printf ("%lg\n", doublea - a);
```

Результат равен -2.87654

Найближче четне

ОСОБЛИВОСТІ ВИКОРИСТАННЯ ЧИСЕЛ З ПЛАВАЮЧОЮ ТОЧКОЮ. ТОЧНІСТЬ

Приклад 1.

Визначити результат:

a=123456789.;0x75bcd15// $\Delta_a=(2^{-23}*2^{26})/2$

b=123456788.; 0x75bcd14//$\Delta_b=(2^{-23}*2^{26})/2$

f=a-b;

printf("Result: %f\n", f); // $\Delta_a - \Delta_b=8$

Висновок: Для операції віднімання близьких чисел похибка може бути великою!

Приклад 2.

float a = 123456789.12345678; $\Delta_a=(2^{-23}*2^{26})/2 = 4$

double doublea = 123456789.12345678; = $(2^{-52}*2*^{26})/2$

printf ("%lg\n", doublea - a);

Результат равен -2.87654

Висновок: Для операції використовувати дані одного типу!!

Відносна похибка $\delta = \Delta/\max(a,b)$

Дивись: Округление чисел с плавающей точкой

Функції для порівняння даних з плаваючою точкою

Відносна похибка $\delta = |\Delta|/\max(a,b)$

Функції для порівняння даних з плаваючою точкою

Відносна похибка δ = |Δ|/max (a,b)

```
template <typename T>
T Max(T x, T y){
return x < y ? y : x;
}
```

```
template <typename T>
T Abs(T x){
        return x < 0 ? -x : x;
}
```

```
bool Compare(T *a, T *b, T Eps, size_t n){
    bool bRes = true;
    for (size_t i = 0; i < n; i++){
        if (a[i] == 0 && b[i] == 0) continue;
        if (Abs(a[i] - b[i]) / Max(a[i], b[i]) > Eps){ bRes = false; break;  }
    }
    return bRes;
}
```

ТИПИ ДАНИХ

Типи даних для даних з плаваючою точкою.

__m128 для чисел із плаваючою точкою звичайної точності (4 байти);

__m128d для чисел із плаваючою точкою подвійної точності (8 байтів).

```
typedef union __declspec(intrin_type) _CRT_ALIGN(16) __m128 {
 float m128_f32[4];
 unsigned __int64 m128_u64[2];
 __int8 m128_i8[16];
 __int16 m128_i16[8];
 __int32 m128_i32[4];
 __int64 m128_i64[2];
 unsigned __int8 m128_u8[16];
 unsigned __int16 m128_u16[8];
 unsigned __int32 m128_u32[4];
 } __m128; // float
typedef struct __declspec(align(16)) __m128d {
    double              m128d_f64[2];
} __m128d;//double
```

__m256 (m256_f32[0].. m256_f32[7]) __m256d(m256d_f64[0].. m256d_f64[3])

ЗАГАЛЬНИЙ ВИД ІМЕНІ ФУНКЦІЙ

<Prefix>_<Code> {p|s}{s|d},
де:

Prefix *– _mm для SSE і _mm256 для AVX*

Code - код операції, що виконується, наприклад, add, sub;

p|s показує, чи виконується функція над одним елементом (s - Single) або над всіма компонентами блока (p - Pack);

s|d задає тип даних, що обробляються (s - число звичайної точності; d - подвійної точності).

КЛАСИФІКАЦІЯ ФУНКЦІЙ

1. Арифметичні функції.
2. Функції для порівняння даних.
3. Функції для виконання операцій побітової обробки.
4. Функції для перетворення даних.
5. Функції для округлення чисел (починаючи з SSE4).
6. Функції для керування кешем.
7. Інші функції

АРИФМЕТИЧНІ ФУНКЦІЇ (FLOAT)

Код функції	Параметри	Значення, що повертається	Призначення
add (sub, mul, div)	a , b	r	ss: r[0] = a[0] Q b[0], r[i]= a[i], i = 1.. ps: r[i]= a[i], i = 0, 1, ... (ps) Q = {+, -, *, /}
_hadd, hsub	a , b	r	ps: r[2i] = a [i] + a[i+1];; r[2i +1] = b [i] + b[i+1]; (I = 0, 1, …) AVX: окремо для кожного SSE
addsub	a , b	r	ps: r[2i] = a [i] - a[i+1]; (ps); r[2i +1] = b [i] - b[i+1]; i=0, 1, … . AVX: окремо для кожного SSE
sqrt	a	r	ss: r[0] = √a[0]; r[i]= a[i], (i = 1, …) ps: r[i]= a[i], (i = 0..)
rcp rsqrt	a	r	ss: r[0] = 1/a[0], r[i]= a[i], (i = 1, …) ps: r[i]= a[i], i = 0..
Min max	a , b	R	ss: r[0] = min (a[0],b[0])r[i]= a[i], i = 1..7 ps: r[i] = min (a[i],b[i])

АРИФМЕТИЧНІ ОПЕРАЦІЇ . ПРИКЛАД 1

Скласти код для для покомпонентного додавання елементів масивів.
Z[i] = x [i] + y [i] (i = 0..n-1)

АРИФМЕТИЧНІ ОПЕРАЦІЇ . ПРИКЛАД 1

```
#include <intrin.h>
#include <stdlib.h>
#include <omp.h>
#include <float.h>
#include <math.h>
void Add(const float *x, const float *y, float *z, size_t n){
	for (int i = 0; i < n; i++)  z[i] = x[i] + y[i];
}
void SSEAdd(const float *x, const float *y, float *z, size_t n){
	__m128 *px = (__m 128 *)x, *py = (__m 128 *)y, *pz = (__m 128 *)z;
	for (int i = 0; i < n / 8; i++)
		pz[i] = _mm_add_ps (px[i], py[i]);
}
void AVXAdd(const float *x, const float *y, float *z, size_t n){
	__m256 *px = (__m256*)x, *py = (__m256*)y, *pz = (__m256*)z;
	for (int i = 0; i < n / 8; i++)
		pz[i] = _mm256_add_ps (px[i], py[i]);}
```

АРИФМЕТИЧНІ ОПЕРАЦІЇ . ПРИКЛАД 1

```
#define N (1024 * 1024)
__declspec (align (32))
float x[N], y[N], z1[N], z2[N];
int _tmain(int argc, _TCHAR* argv[]){
for (size_t i = 0; i < N; i++){ x[i] = rand() + 0.; y[i] = rand() + 0.;}
double start, finish, min = DBL_MAX, dif;
for (int i = 0; i < 10; i++){
        start = omp_get_wtime();
        Add(x, y, z1, N);
        finish = omp_get_wtime();
        dif = finish - start;
        if (dif < min) min = dif;
}
printf("Add: %lg\n", min);
min = DBL_MAX;
 for (int i = 0; i < 10; i++)
         start = omp_get_wtime();
         AVXAdd(x, y, z2, N);
         finish = omp_get_wtime();
         dif = finish – start;
        if (dif < min) min = dif; }
```

```
bool bRes = true;
for (int i = 0; i < N; i++){
                if (fabs(z1[i] - z2[i]) >= 1e-6){
                                bRes = false; break;
                }
}
printf("AVXAdd: %lg %s\n", min, bRes? "Yes" : "No");
```

0.0018 0.0018 0.00177

АРИФМЕТИЧНІ ОПЕРАЦІЇ . ПРИКЛАД 1

Дізасемблювання коду
Enable Enhanced Instruction Set: Advanced Vector Extensions:AVX2
debug→Windows →Disassembly

```
z[i] = x[i] + y[i];
010F1010  vmovups     ymm0,ymmword ptr [eax+18F3380h]
010F1018  vaddps      ymm0,ymm0,ymmword ptr [eax+1CF3380h]
010F1020  vmovups     ymmword ptr [eax+14F3380h],ymm0
010F1028  vmovups     ymm0,ymmword ptr [eax+18F33A0h]
010F1030  vaddps      ymm0,ymm0,ymmword ptr [eax+1CF33A0h]
010F1038  vmovups     ymmword ptr [eax+14F33A0h],ymm0
010F1040  add         eax,40h
```

AVXAdd(x, y, z2, N);

```
010F1130  vmovups     ymm0,ymmword ptr [eax+18F3380h]
010F1138  vaddps      ymm0,ymm0,ymmword ptr [eax+1CF3380h]
010F1140  vmovups     ymmword ptr [eax+10F3380h],ymm0
010F1148  add         eax,20h
```

Висновок: Компілятор може. Але чи все?

АРИФМЕТИЧНІ ОПЕРАЦІЇ. ПРИКЛАД 2

Скласти функцію для обчислення відстаней між заданою точкою й точками з масиву

Структури точки:

```
typedef struct{
        float x, y;
 }MYPOINT, *PMYPOINT;
```

АРИФМЕТИЧНІ ОПЕРАЦІЇ/ ПРИКЛАД 2

Функція для послідовного режиму і фрагмент головної програми для її використання

```
void SecDims(MYPOINT ps[], PMYPOINT p0, float Dims[], size_t n){
        const float x0 = p0->x, y0 = p0->y;
        for (size_t i = 0; i < n; i++){
                float r1 = ps[i].x - x0; float r2 = ps[i].y - y0;
                Dims[i] = sqrt(r1 * r1 + r2 * r2);
        }
}

double start, finish;
double min = DBL_MAX, diff;
for (int i = 0; i < M; i++)
                start = omp_get_wtime();
                SecDims(p, &p0, Dims1, N);
                finish = omp_get_wtime();
                diff = finish - start; if (diff < min) min = diff;
}
printf("SecDims          time = %lg\n", min);
```

АРИФМЕТИЧНІ ОПЕРАЦІЇ/ ПРИКЛАД 2

Функція для послідовного режиму і фрагмент головної програми для її використання

	X[i]	Y[i]	X[i+1]	Y[i+1]	X[i+2]	Y[i+2]	X[i+3]	Y[i+3]
	X0	Y0	X0	Y0	X0	Y0	X0	Y0
sub	X[i] – x0	Y[i] – y0	X[i+1]-x0	Y[i+1]-y0	X[i+2] – x0	Y[i+2] – y0	X[i+3]-x0	Y[i+3]-y0
mul	$(X[i] – x0)^2$	$(Y[i] – y0)^2$	$(X[i+1]-x0)^2$	$(Y[i+1]-y0)^2$	$(X[i+2] – x0)^2$	$(Y[i+2] – y0)^2$	$(X[i+3]-x0)^2$	$(Y[i+3]-y0)^2$
hadd	$(X[i] – x0)^2+(Y[i] – y0)^2$		$(X[i+1]-x0)^2+(Y[i+1]-y0)^2$		$(X[i+2] – x0)^2+(Y[i+2] – y0)^2$		$(X[i+3]-x0)^2+(Y[i+3]-y0)^2$	
sqrt	D[0]		D[1]		D[2]		D[3]	

АРИФМЕТИЧНІ ОПЕРАЦІЇ. ПРИКЛАД 2

АРИФМЕТИЧНІ ОПЕРАЦІЇ, ПРИКЛАД 2

***// float.* Функції з SSE. float**

```
void SSEDims(MYPOINT A[], PMYPOINT C, float Dims[], size_t n){
	const float x0 = C->x, y0 = C->y;
	__m128 m128_c = { x0, y0, x0, y0 };
	__m128 *pm128_A = (__m128 *)A;
	__m128 *pDims = (__m128 *)Dims;
	for (size_t i = 0, j = 0; i < n/2; i+= 2){
		__m128 r1 = _mm_sub_ps(pm128_A[i], m128_c);
		__m128 r2 = _mm_sub_ps(pm128_A[i + 1], m128_c);
		r1 = _mm_mul_ps(r1, r1);
		r2 = _mm_mul_ps(r2, r2);
		r1 = _mm_hadd_ps(r1, r2);
		pDims [j++] = _mm_sqrt_ps(r1);
	}
}
```

Для AVX – проблема горизонтального складання – дивись нижче!!!

АРИФМЕТИЧНІ ОПЕРАЦІЇ. ПРИКЛАД 2

Функція	Час виконання N = 1024 * 1024	Прискорення
SecDims	0.0066	1
SSEDims	0.0017	3.88
DoubleDims	?	?
SSEDoubleDims	?	?

LOAD І STORE ФУНКЦІЇ

Призначення функцій – завантаження з пам'яті в регістр (Load) і навпаки (Store).

__m128 _mm_load_ps(float * p);
__m128 _mm_loadu_ps(float * p);
__m128 _mm_load_pd(double * p);
__m128 _mm_loadu_pd(double * p);
__m256 _mm256 _load_ps(float * p);
__m 256 _mm256_loadu_ps(float * p);
__m 256 _mm256_load_pd(double * p);
__m 256 _mm256_loadu_pd(double * p);

void _mm_store_ps(float*, _m128);

Приклад. Реалізувати попередню задачу без вимоги вирівнювання та кратності кількості точек 4

ВИРІШЕННЯ ЗАДАЧИ ДЛЯ ЗАГАЛЬНОГО ВИПАДКУ

```
void SSEDimsU (MYPOINT A[], PMYPOINT C, float Dims[], size_t n){
const float x0 = C->x, y0 = C->y;
__m128 m128_c = { x0, y0, x0, y0 };
size_t n_ = n / 4 * 4;
for (size_t i = 0, j = 0; i < n_; i += 4){
      __m128 t1 = _mm_loadu_ps((const float*)&A[i]);
      __m128 t2 = _mm_loadu_ps((const float*)&A[i + 2]);
      __m128 r1 = _mm_sub_ps(t1, m128_c);
      __m128 r2 = _mm_sub_ps(t2, m128_c);
      r1 = _mm_mul_ps(r1, r1); r2 = _mm_mul_ps(r2, r2);
      r1 = _mm_hadd_ps(r1, r2); r1 = _mm_sqrt_ps(r1);
      _mm_storeu_ps(&Dims[j], r1); j += 4;
}
for (size_t i = n_; i < n; i++){
      float r1 = A[i].x - x0; float r2 = A[i].y - y0;
      Dims[i] = sqrt(r1 * r1 + r2 * r2);
}} Час виконання Було 0.0017 стало 0.0018                    + 5%
```

ПОРІВНЯННЯ ДАНИХ

Для функцій можна задавати умову порівняння:

Іменем функції (для SSE)

Спеціальною маскою (AVX)

Загальний вигляд імені функції з завданням умови в імені:

<Префікс>_{cmp}[<Умова>]_ {s|p}{s|d},

де:

cmp - повертають 1 у всіх бітах, якщо умова, задана в команді порівняння, виконується. Повертає 0, якщо умова не виконується;

s відповідає порівнянню одного даного, а *p* – всіх компонентів блоку; для команд типу *comi, ucomi* використовується тільки *s;*

s відповідає використанню даних зі звичайної, а *d* – з подвійною точністю.

Умови порівнювання: {eq, ne, lt, le, gt, ge, neq, nlt, nle, ngt, ge}.

Приклади функцій: *_mm_cmpeq_ss, _mm_cmpeq_ps, _mm_cmpeq_pd*

ПОРІВНЯННЯ ДАНИХ

Використання спеціального параметру (AVX)

__m256 _mm256_cmp_ **{s|p}{s|d}**(__m256 a, __m256 b, const int maska);

де:

cmp - повертають 1 у всіх бітах, якщо умова, задана в команді порівняння, виконується. Повертає 0, якщо умова не виконується;

s відповідає порівнянню одного даного, а *p* – всіх компонентів блоку; для команд типу *comi, ucomi* використовується тільки *s;*

s відповідає використанню даних зі звичайної, а *d* – з подвійною точністю.

Приклади маски:

Умови: EQ (==), NE (!=), LT (<), NLT (!<), LE (<=),NLE (<=), GT (>), NGT (!>), GE (>=), NGE (>=)

Приклади масок:

_CMP_LT_OS, _CMP_LE_OS, _CMP_NEQ_OS,…

Решта властивостей для інших функцій.

ФУНКЦІЇ ДЛЯ РОБОТИ З БІТАМИ

Код функції	Параметри	Результат	Призначення
and	a, b	r	r = a & b
or	a, b	r	r = a | b
xor	a, b	r	r = a ^ b
andnot	a, b	r	r = ~a &b

ФУНКЦІЇ ПОРІВНЯННЯ ТА РОБОТИ З БІТАМИ. Приклад 3

Вычислить значение x[i] = y [i] / z [i], если x[i]<=0, в противном случае оставить значение x[i] без изменения

Послідовний код:

```
void IfDiv(float *x, float *y, float *z, size_t n)
{
	for (size_t i = 0; i < n; ++i)
	{
		if (x[i] <=0)
		{
			x[i] = y[i] / z[i];
		}
	}
}
```

ФУНКЦІЇ ПОРІВНЯННЯ ТА РОБОТИ З БІТАМИ. Приклад 3

Функція для SSE

```
void SSEIfDiv(float *x, float *y, float *z, size_t n)
{
	__m128 *x128 = (__m128 *)x, *y128 = (__m128 *)y, *z128 = (__m128 *)z;
	__m128 zero = _mm_setzero_ps();
	for (size_t i = 0; i < n / 4; ++i)
	{
		const __m128 r1 = _mm_div_ps(y128[i], z128[i]);
		const __m128 r2 = _mm_cmple_ps(x128[i], zero);
		const __m128 r3 = _mm_andnot_ps(r2, x128[i]);
		const __m128 r4 = _mm_and_ps(r2, r1);
			x128[i] =	_mm_or_ps(r3, r4);
	}
}
```

ФУНКЦІЇ ПОРІВНЯННЯ ТА РОБОТИ З БІТАМИ. Приклад 3

Функція для AVX

```
void AVXIfDiv(float *x, float *y, float *z, size_t n)
{
	__m256 *x256 = (__m256 *)x, *y256 = (__m256 *)y, *z256 = (__m256 *)z;
	const __m256 zero = _mm256_setzero_ps();
	for (size_t i = 0; i < n/ (sizeof(__m256)/ sizeof (float)); ++i)
	{
		const __m256 r1 = _mm256_div_ps(y256[i], z256[i]);
		const __m256 r2 = _mm256_cmp_ps(x256[i], zero, _CMP_LE_OS);
		const __m256 r3 = _mm256_andnot_ps(r2, x256[i]);
		const __m256 r4 = _mm256_and_ps(r2, r1);
		x256[i] = _mm256_or_ps(r3, r4);
	}
}
```

Результати (N = 1024 * 1024)

Функція	Час(сек)	Прискорення
IfDiv	0.0059	1
SSEIfDiv	0.0019	2.4
AVXIfDiv	0.0019	2.4

ОКРУГЛЕННЯ ЧИСЕЛ

Режим	Призначення
_MM_FROUND_TO_NEAREST_INT _MM_FROUND_NINT	Округлення до найближчого цілого. 3.7 ≈ 4; 3.4≈3; -3.7≈-4; -3.4≈-3
_MM_FROUND_TO_NEG_INF	Округлення до найближчого меншого 3.7 ≈ 3; 3.4≈3; -3.7≈-4; -3.4≈-4
_MM_FROUND_TO_POS_INF	Округлення до найближчого більшого 3.7 ≈ 4; 3.4≈4; -3.7≈-3; -3.4≈-3
_MM_FROUND_TO_ZERO	Усічення числа 3.7 ≈ 3; 3.4≈3; -3.7≈-3; -3.4≈-3
_MM_FROUND_RAISE_EXC	Виключення, якщо результат не відповідає коректному значенню
_MM_FROUND_NO_EXC	Немає виключення навіть в разі, якщо результат не відповідає коректному значенню

ОКРУГЛЕННЯ ЧИСЕЛ

Основні функції

```
__m128   _mm_round_ss(__m128 src, int mode);
__m128   _mm_round_ps(__m128 src, int mode);
__m256d  _mm256_round_pd(__m128d src, int mode);

__m256   _mm256_round_ss(__m256 src, int mode);
__m256   _mm256_round_ps(__m256 src, int mode);
__m256d  _mm256_round_pd(__m256d src, int mode);
```

ОКРУГЛЕННЯ ЧИСЕЛ (SSE4)

Скласти функції для округлення чисел із плаваючою точкою в режимі ***_mm256_FROUND_TO_NEAREST_INT*** за допомогою звичайних операцій та SSE функцій.

```
void round(float *x, float *y, size_t n) {
        for (size_t i = 0; i < n; ++i)  {
        if (x[i] < 0) y[i] = float((int)(x[i] - 0.5)); else y[i] = float((int)(x[i] + 0.5));
        }
}
 void SSEround(float *x, float *y, size_t n) {
         __m128 *px = (__m128 *)x, *py = (__m128 *)y;
       for (size_t i = 0; i < n / 4; ++i)          {
        py[i] = _mm256_round_ps(px[i], _mm256_ROUND_NEAREST);
}       // 0.12 vs 0.012
```

Висновок. Прискорення в 10 разів!!!

ФУНКЦІЇ ПЕРЕМІШУВАННЯ

- Функції дозволяють вибрати компоненти із заданими номерами з двох блоків відповідно до заданої маски. Результат формується з окремих компонентів першого та другого блоків
- #define _mm256_SHUFFLE(z, y, x, w) \ ((z<<6) | (y<<4) | (x<<2) | w)

r[0] = a[w]; r [1] = a [x]; r[2] = b[y]; r[3] = b [z]

#define _mm256_SHUFFLE2(x, y) ((x<<1) | y)

r[0] = a[y]; r [1] = b [x]

_mm256_shuffle_ps

_mm256_shuffle_pd

ФУНКЦІЇ ПЕРЕМІШУВАННЯ

Обчислити додаток для комплексних чисел

```
COMPLEX a [N], b[N], c1 [N], c2 [N];
void cSSEMul (const PCOMPLEX a, const PCOMPLEX b, PCOMPLEX
c, size_t n){
__m128 *pa = (__m128 *)a, *pb = (__m128 *)b, *pc = (__m128 *)c;
for (size_t i = 0; i < n/2; i ++){
pc [i] = _mm256_addsub_ps (
_mm256_mul_ps (_mm256_shuffle_ps (pa [i], pa
[i],_mm256_SHUFFLE (2, 2, 0, 0)), pb [i]),
_mm256_mul_ps (_mm256_shuffle_ps (pa [i], pa
[i],_mm256_SHUFFLE (3, 3, 1, 1)),
_mm256_shuffle_ps (pb [i], pb [i], _mm256_SHUFFLE (2, 3, 0, 1))));
}} 44107 vs 109628
```

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.