Використання SIMD команд при розробці програм

Мета заняття

Сучасні обчислювальні системи мають у складі команд SIMD команди, які передбачають виконання однієї команди для масиву даних заданої довжини. Як за правило, SIMD команди реалізуються за допомогою окремого конвеєру, тому вони виконуються паралельно з основним потоком команд. Деякі процесори мають декілька контейнерів для виконання таких команд, це дозволяє паралельно обробляти декілька масивів, або їх частин. Метою даного заняття є придбання практичних навиків використання SIMD команд при створенні програм.

Методичні вказівки з організації самостійної роботи студентів

Вивчить тему 3 [1, 87-176]

SSE команди

Блок даних 128-біт. Команди цієї групи можуть використовуватися для цілих даних та даних з плаваючою крапкою.
Особливості цих команд:

— використовують власні регістри, тому не треба переключати режим виконання;

— використовують окремий контейнер для роботи. Виконується одночасно дві команди додавання і множення для чисел з плаваючою крапкою;

— реалізовані для 32-бітних, та 64 бітних процесорів.

Усі функції визначені в файлі **intrin.h**.
При розгляді функцій цього заголовного файлу треба мати на увазі, що функції, які позначені як __MACHINEX64, реалізовані тільки для 64 бітних процесорів, та можуть використовуватись тільки для 64-бітних програм.
В залежності від заповнення блоку даних використовуються наступні типи даних:
__m128 для чисел з плаваючою крапкою звичайної точності (4 байта);
__m128d для чисел плаваючою крапкою подвійної точності (8 байтів).
__m128i для цілих чисел довжиною 1, 2, 4, 8 байтів. Для доступу до окремих елементів 128 бітного даного типу __m128 використовується об'єднання з полями:
m128_f32 - для чисел з плаваючою крапкою звичайної точності (4 байта);
m128_i8 - для цілих чисел довжиною 8 біт
m128_i16 - для цілих чисел довжиною 16 біт
m128_i32 - для цілих чисел довжиною 32 біт
m128_i64 - для цілих чисел довжиною 64 біт
m128_u8 - для цілих чисел без знаку довжиною 8 біт
m128_u16 - для цілих чисел без знаку довжиною 16 біт
m128_u32 - для цілих чисел без знаку довжиною 32 біт

m128_u64 - для цілих чисел без знаку довжиною 64 біт
Для доступу до окремих елементів 128 бітного даного типу __m128d використовується поле m128d_f64.
Для доступу до окремих елементів 128 бітного даного типу __m128i використовується об'єднання з полями:
m128i_i8 - цілі дані довжиною 8 біт;
m128i_i16 - цілі дані довжиною 16 біт;
m128i_i32 - цілі дані довжиною 32 біт;
m128i_i64 - цілі дані довжиною 32 біт;
m128i_u8 - цілі дані довжиною 8 біт;
m128i_u16 - цілі дані довжиною 16 біт;
m128i_u32 - цілі дані довжиною 32 біт;
m128i_u64 - цілі дані довжиною 64 біт;

Загальний вид імені функції для даних з плаваючою крапкою:

_mm_<Код><Масив / Одне число> <Тип даного>,

де:
Код - код операції, наприклад, add, sub;
Масив / Одне число показує, чи виконується операція над одним даним (літера s) або над упакованим масивом (літера p);
Тип даного - число зі звичайною точністю (літера s) або подвійною точністю (літера d).

Загальний вид імені функції для цілих даних:

_mm_<Код>[s|r]_{ep|s}<Тип ><Довжина >,
де:
Код - код операції, наприклад, add, sub;
[s|r] - s задається для команд із насищенням і не задається для звичайних команд;
r (reverse) - обробка елементів масиву в зворотному порядку;
{ep|s}- ep для роботи з масивом і s для роботи з одним даним;
Тип - визначає тип даного, літера i означає тип int, літера u - unsigned;
Довжина - довжина елементу в бітах (8, 16, 32, 64, 128).

Довідник основних SSE функцій наведено в Додатках А та Б. Нижче наведено загальну характеристику функцій.

Функції для роботи з даними із плаваючою крапкою

Усі функції можна розділити на класи:

- функції для виконання операцій (арифметичних, логічних, …);
- функції для округлення чисел (SSE4);
- функції для керування кешем;
- функції ініціалізації;
- функції для завантаження SSE регістрів;
- функції для запису даних з SSE регістрів пам'ять;

— інші функції.

Функції для виконання операцій в свою чергу діляться на:

— арифметичні;

— функції для побітової обробки;

— функції порівняння;

— функції перетворення.

Арифметичні функції для даних зі звичайною точністю наведені в Додатку В, табл. ДВ.1, а для подвійної точності - в Додатку А, табл. ДА.2.
Функції побітової обробки (Додаток А, табл. ДА.3) розглядають блок даних, як блок 128 бітів, над якими виконуються потрібні операції.
Функції порівняння (Додаток А, табл. ДА.4). Порівняння виконується покомпонентно. В результаті записуються у всіх бітах відповідного поля 1, якщо дані для цього поля співпадають, і 0, в противному разі. В якості умови для порівняння використовуються наступні операції:

lt	<	nlt	!<
le	<=	nle	!<=
eq	==	neq	!=
gt	>	ngt	!>
ge	>=	nge	!>=

Функції для перетворення даних (Додаток А, табл. ДА.5). Використовуються для перетворення компонентів даних з формату з плаваючою крапкою в формат цілий і навпаки.
Функції для округлення, SSE4 (Додаток А, табл. ДА.6). Дозволяють визначити режим округлення за допомогою констант:
_MM_FROUND_TO_NEAREST_INT - округлення до найближчого;
_MM_FROUND_TO_NEG_INF - округлення до найближчого меншого;
_MM_FROUND_TO_POS_INF - округлення до найближчого більшого;
_MM_FROUND_TO_ZERO - усікати дрібну частину;
_MM_FROUND_CUR_DIRECTION – використовувати поточні установки для режиму округлення.
Функції для керування кешем (Додаток А, табл. ДА.7) призначені для упередженого завантаження даних в Кеш з метою зменшення витрат, пов’язаних з промахом Кешу, а також керування режимами використання або не використання Кешів різного рівня.
Функції для ініціалізації та завантаження блоку даних з масиву (Додаток А, табл. ДА.8) дозволяють задати початковий стан даних. Ці функції можна використовувати також замість виразів для завдання окремих компонентів числа. Вони також використовуються, якщо необхідно мати два масиви: для роботи з SSE функціями та для обробки звичайних масивів. Замість використання різних даних можна використовувати один регіон пам’яті та 2 показника, наприклад:

```
// Пам’ять для масиву, яку попередньо вирівнюємо на потрібну границю
__declspec (align (16))
        float fa [4] = {1, 2, 3, 4};
// Показник для використання в форматі __m128:
__m128 *pfa = (__m128 *) fa;
```

Якщо необхідно змінити порядок задавання даних, або завантажити частини числа, використовуються функції цієї групи.

Функції копіювання даних, на відміну від попередніх функцій, виконують копіювання даних з формату __m128 (__m128d) в формат звичайного масиву даних відповідного типу. Якщо дані при копіюванні не змінюються, можна використовувати показники для опрацювання даних різних форматів, тобто, якщо для попередніх даних записати float *pfb = (float*) pfa; то далі можна використовувати масив pfb, як масив даних типу float. При необхідності зміни порядку елементів в масиві, або копіювання частини числа використовують спеціальні функції, які визначені в Додатку В, табл. ДА.9.
В якості інших функцій розглянемо функції для обрання компонентів числа відповідно заданої маски спочатку для даних типу float, а потім для даних типу double.
Для даних типу float номер поля приймає значення 0, 1, 2, 3, тобто для його завдання достатньо 2 біта. Для завдання усіх полів достатньо одного байту.
Для завдання цього байту використовується макрос: _MM_SHUFFLE(z, y, x, w), який визначено так:

```
#define        _MM_SHUFFLE     ((z<<6) | (y<<4) | (x<<2) | w)
```

Заголовок функції:

```
__m128 _mm_shuffle_ps (__m128 a , __m128 b , int i );
```

де:
a, b - числа, з яких обираються дані;
i - маска.
x, w задають номера 32-бітових компонент першого числа;
z, y задають номера 32- бітових компонент другого числа.
Результат записується в такому порядку: w, x, y, z.

Приклад.

```
__m128 aa, bb, cc;
aa.m128_u32[0] = 0x11111111;
aa.m128_u32 [1] = 0x22222222;
aa.m128_u32 [2] = 0x33333333;
aa.m128_u32[3] = 0x44444444;
bb.m128_u32[0] = 0x55555555;
bb.m128_u32 [1] = 0x66666666;
bb.m128_u32 [2] = 0x77777777;
bb.m128_u32[3] = 0x88888888;
cc = _mm_shuffle_ps (aa, bb, _MM_SHUFFLE(1, 0, 2, 3));
```

Очікуваний результат:

```
cc.m128_u32[0] = 0x44444444;
cc.m128_u32[1] = 0x33333333;
cc.m128_u32[2] = 0x55555555;
cc.m128_u32[3] = 0x66666666;
```

Для полів типу double в якості макросу для обчислення маски використовується макрос _MM_SHUFFLE2, який визначено так:

```
#define        _MM_SHUFFLE2(x, y)  ((x<<1) | y)
```

Використовується функція:

```
__m128d _mm_shuffle_pd (__m128d a, __m128d b, int i);
```

a, b - дані,. з з яких обираються поля;
i - маска.

Приклад.

```
__declspec(align(16)) double a[2] = { 1.0, 1.1 };
        __declspec(align(16)) double b[2] = { 2.0, 2.1 };
        __declspec(align(16)) double c[2] ;
        __m128d *pa = (__m128d *)a;
        __m128d *pb = (__m128d *)b;
        __m128d *pc = (__m128d *)c;
        *pc = _mm_shuffle_pd (*pa, *pb, _MM_SHUFFLE2(0, 1));
        printf_s ("%lg %lg\n", c [0], c [1]);
```

Функції для цілих чисел

Групи функцій:

- для завантаження;
- арифметичні;
- для роботи з бітами;
- для порівняння даних;
- для упакування та розпакування;
- для ініціалізації;
- для перемішування;
- для перетворення даних;
- інші функції.

Функції завантаження використовуються для завантаження заданих значень в SSE змінну (Додаток Б, табл. ДБ.1). Замість цих функцій можна використовувати показники, наприклад:

```
__declspec (align (16))
                int ia [] = {1, 2, 3, 4};
        __m128i *pia = (__m128i*) ia;
        for (i = 0; i < 4; i++)
                printf ("%d ", pia->m128i_i32 [i]);
        printf ("\n");
```

Арифметичні функції (Додаток Б, табл. ДБ.2) дозволяють виконати арифметичні операції (+, -, *) над окремими елементами даних. Деякі функції дозволяють обробляти дані в горизонтальному напрямі. Для операції множення є можливість окремо отримати молодшу та старшу частину результату для уникнення переповнення. Серед операцій є операції обчислення максимального та мінімального елементів масивів, модуля, а також спеціальний набір операцій, які використовуються при обробці візуальних даних.

Функції для роботи з бітами (Додаток Б, табл. ДБ.3) розглядають дані, як рядок бітів довжиною 128 біт. Команди зсуву можна виконувати як покомпонентно, так і для усіх 128 бітів разом. В першому випадку, при зсуві компонента розряди зліва чи справа в залежності від напряму зсуву, пропадають.

Функції порівняння даних (Додаток Б, табл. ДБ.4) дозволяють порівняти покомпонентно елементи блоку даних. Як і для команд порівняння для даних з плаваючою крапкою, функції повертають значення -1, якщо відповідний компонент співпадає, і 0, якщо не співпадають.

Функції для упакування та розпакування (Додаток Б, табл. ДБ.5) дозволяють виконати упакування 2-х чисел в одне, використовуючи різні частини числа і різні правила їх перетворення.

Функції для ініціалізації даних (Додаток Б, табл. ДБ.6). Використовуються, якщо необхідно встановити значення компонентів (функції виду _mm_set…), або усіх компонентів однаковими значеннями (функції виду _mm_set1). Функції типу _mm_setr... встановлюють задані значення в зворотному порядку, тобто, починаючи зі старшого елементу. В функціях типу _mm_store адреса результату передається через параметри.
Функції для перемішування даних (Додаток Б, табл. ДБ.7). Виконуються, якщо необхідно створити 128 бітне число за допомогою 2-х інших чисел, перемішуя їх компоненти по заданому закону.
Функції для перетворення даних (Додаток Б, табл. ДБ.8). Використовуються, якщо необхідно змінити формат даних. Функції для перетворення цілих даних в дані з плаваючою крапкою та навпаки, задані при розгляді даних з плаваючою крапкою (Додаток А).
Інші функції (Додаток Б, табл. ДБ.9) дозволяють отримати окремий компонент з блоку даних, або переставити їх відповідно заданому порядку, отримати інформацію про знаки компонент числа, або новий блок даних з окремих частин двох блоків даних.
Уважний перегляд усіх розглянутих функцій показує, що вони можуть використовуватись тільки за угоди, коли процесор може виконувати відповідні команди. У разі застосування функції, що відповідає команді, яку процесор не підтримує, буде виключення. Для того, щоб додаток працював нормально незалежно від процесора, який використовується, необхідно вміти визначати властивості процесору під час виконання інсталяції програми, а іноді і під час виконання програми.

Особливості AVX команд

Загальна характеристика

AVX команди – це SIMD операції для блоків розміром 256 бітів для даних з плаваючою точкою (AVX - Advanced Vector Extensions), та цілих даних (AVX2). Розгляд функцій для роботи з 512 бітними блоками не передбачено. Це пов’язано з тим, що процесори, які підтримують такі блоки, тільки з’являються в 2016 році і автори не мають змоги перевірити їх застосування.
Більшість операцій, які підтримуються для блоків довжиною 128 бітів (SSE операції), є і для блоків довжиною 256 бітів. Крім того є додаткові операції, а також спеціальні режими використання (наприклад, команди порівняння).
Сучасні компілятори використовують цей клас операцій при побудові коду. Але можливості компілятору обмежені тільки тривіальними випадками, наприклад, виконання однієї і тієї ж операції над елементами масиву.
Для використання функцій цього класу необхідно переконатися, що процесор та операційна система підтримує цей клас операцій. Процедура перевірки можливості використання команд AVX2 розглянута нижче.
Для використання цього класу операцій необхідно в командному рядку для компілятору задати додаткову опцію /arch:AVX2. Для цього треба для властивостей проекту обрати Configuration Properties→C/C++→Command Line та в полі Additional Options записати /arch:AVX2.
Типи даних та заголовки функцій визначені в файлі заголовків immintrin.h, який автоматично підключається при підключенні файлу заголовків intrin.h.

Типи даних

Усі типи даних вирівняні на границю довжини блоку (32 байта), що задається визначенням: _CRT_ALIGN(32), тобто:
#define _CRT_ALIGN(x) __declspec(align(x)).
Блок даних типу float складається з 8 даних цього типу

```
typedef union __declspec(intrin_type) _CRT_ALIGN(32) __m256 {
   float m256_f32[8];
} __m256;
```
Блок даних типу double складається з 4 даних:
```
typedef struct __declspec(intrin_type) _CRT_ALIGN(32) __m256d {
   double m256d_f64[4];
} __m256d;
```
Блок даних типу int, unsigned і кількість елементів в блоці залежить від довжини компоненту блоку, ось чому для визначення використовується union замість struct:
```
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;
```

Характеристика функцій

Загальний вид більшості функції:
_mm256_<Код операції>_Суфікс
Код операції задає операцію, яка виконується, наприклад, add – операція додавання;
суфікс залежить від типу даних що обробляються, точності для даних з плаваючою точкою, довжини компоненту для цілих даних.
Загальний вид суфіксу для даних з плаваючою крапкою:
$\begin{Bmatrix} p \\ s \end{Bmatrix} \begin{Bmatrix} s \\ d \end{Bmatrix}$

Перша дужка визначає, обробляється один компонент (s), або усі компоненти блоку (p). Друга дужка визначає тип. Буква s відповідає типу float, буква d – типу double.

Загальний вид суфіксу для цілих даних $\begin{Bmatrix} s \\ ep \end{Bmatrix} \begin{Bmatrix} i \\ u \end{Bmatrix} < length >$

Де:
s – обробка цілого блоку;
ep – обробка окремих компонентів;
i – з урахуванням знаку числа;
u – без урахування знаку числа;
length – довжина елементу, що обробляється. Може мати значення 8, 16, 32, 64, 128. 256. Не усі команди можуть обробляти компоненти усіх наведених довжин. В разі обробки цілого блоку довжина завжди 256.
Більшість команд для AVX (AVX2) відрізняються від відповідних команд (SSE) тільки префіксом. В Додатку В наведені тільки ті команди, які додатково додані в AVX (AVX2).

Інформація про процесор

По документації до Visual Studio вивчить функції __cpuid (заголовний файл intrin.h) та __cpuidex:
```
void __cpuid(int a[4], int b);
void __cpuidex(int a[4], int b, int c);
```

де:
b – номер функції, яка виконується. Визначає, що робить ця функція;
c – додатковий номер функції, яка виконується. Задається, якщо команді cpuid треба задавати додаткову інформацію в регістрі ECX. Визначає, що робить ця функція. Функція __cpuid фактично реалізована як __cpuid(int a[4], int b, 0);
a – масив даних. Визначає вхідні та вихідні дані для функції, яка виконується.
Дані функції є реалізацією команди CPUID процесорів фірм AMD, INTEL. Масив *a* задає значення в регістрах EAX, EBX, ECX, EDX відповідно.
Усі команди, які виконуються функціями __cpuid, __cpuidex діляться на 2 класи: *звичайні* та *розширені*. Щоб узнати, які команди підтримуються процесором, необхідно визначити максимальне значення номеру функції для звичайного та розширеного варіанту. Для визначення максимального значення номеру звичайної функції необхідно використовувати функцію номер 0. Функція повертає максимальне значення звичайної функції в регістрі EAX (значення a [0]). Для визначення максимального номеру розширеної функції необхідно задати номер функції 0x80000000. Функція повертає максимальне значення розширеної функції в регістрі EAX (значення a [0]).
Отримані значення можна використовувати для визначення типу процесора. Відповідність максимальних номерів функції і типу процесора наведено в табл. 1 для процесору типу INTEL, а основні команди для отримання інформації про процесор – в табл.. 3.2
Таблиця 3.1 – Відповідність номерів звичайних та розширених функцій і типу процесора

Тип процесора	Звичайна функція	Розширена функція
Pentium 3 Processor	3	Не підтримуються
Pentium 4 Processor с Hyper threading	5	0x80000008
2-х ядерні Intel	10	0x80000008

Перевірка можливості використання AVX2 команд

Операції AVX2 можливо використовувати, якщо:

— процесор підтримує такі операції;

— операційна система підтримує роботу з 256 бітними регістрами, які використовуються цими операціями

Підтримка AVX2 включена до деяких процесорів, які вироблені, починаючи з 2013 року.

Таблиця 3.2 – Функції для отримання інформації про процесор та пам'ять комп'ютера

Функція (b)	Результат
0	a [0] – максимальний номер звичайної функції; a [1], a [3], a [2] – тип процесора. Для Intel повинні отримати рядок GenuineIntel – оригінальний Intel, Для AMD повинні отримати рядок AuthenticAMD.
1	a [0] – додаткова інформація про тип процесору (тип, країна, модель,…) a [1] – Біти 8-15 – розмір рядка кешу в 8–байтових елементах; Біти 16-23 – максимальна кількість ядер процесору; a [2] – Біт 0 – підтримка SSE3; a [2] – Біт 9 – підтримка SSSE3; a [2] – Біт 19 – підтримка SSE4.1; a [2] – Біт 20 – підтримка SSE4.2;

Функція (b)	Результат
	a [3] – Біт 32 – підтримка MMX; a [3] – Біт 25 – підтримка SSE; a [3] – Біт 26 – підтримка SSE2; a [3] – Біт 28 – підтримка Super Threading.
2	Визначення геометричних розмірів Кешу a [0]**:** Молодший байт – кількість викликів команди для отримання повної інформації про Кеш (1) Решта байтів a [0] і усі байти a [1], a [2], a [3] – коди інформації про кеш.
0x80000000	a [3] – Біт 31 – підтримка 3DNow! (AMD) a [3] – Біт 30 – підтримка 3DNow!2 (AMD)
0x80000001	a [2] – Біт 6 – підтримка SSE4A a [2] – Біт 11 – підтримка SSE5
0x80000002 - 0x80000004	a [0], a [1], a [2], a [3] – повна назва процесора та його тактова частота (INTEL, AMD)
0x80000005	(AMD) ECX – інформація про кеш даних EDX – інформація про кеш команд Структура інформації: 0-7 біти – розмір рядка кешу; 8-15 біти – кількість рядків кешу; 16-23 біти – асоціативність; 24-31 біти – розмір кешу в байтах

Приклади процесорів

Нижче наведені приклади процесорів, які підтримують операції AVX2 фірм INTEL, та AMD[2]
Фірма INTEL:

- Haswell processor, Q2 2013
- Haswell E processor, Q3 2014
- Broadwell processor, Q4 2014
- Broadwell E processor, Q2 2016
- Skylake processor, Q3 2015
- Kaby Lake processor, 2016
- Cannonlake processor, очикуються в 2017 році

Фірма AMD:

- Steamroller-based processor, Q1 2014
- Excavator-based processor, 2015
- Jaguar-based processor, 2016
- Puma-based processor, 2016

Використання списку процесорів для визначення можливості застосування команд AVX2 незручно, тому далі розглянемо програмне визначення такої можливості.

[2] https://en.wikipedia.org/wiki/Advanced_Vector_Extensions

Програмна перевірка процесору

Для програмної перевірки процесору на підтримку AVX2 команд використовується команда процесору CPUID. Перед виконанням цієї команди необхідно в регістр EAX записати номер функції, яку треба виконати. Деякі команди потребують задавати вхідні дані не тільки в регістрі EAX, а і в регістрі ECX (додатковий номер функції). Команда повертає результат в регістрах EAX, EBX, ECX, EDX.
Необхідні функції. а також вхідні та вихідні дані наведені в табл. 3.3
Таблиця 3.4 Перевірка процесору на підтримку AVX2 команд

Призначення	EAX	ECX	Результат
Визначення максимального номера функції	0	0	EAX
Підтримка FMA	1	0	ECX, біт 12 =1
Підтримка AVX2	7	0	EBX, біт 5 =1

Функція для перевірки.

```
bool AVX2ProcessorSupport(){
	bool b = false; int reg[4];
	__cpuidex(reg, 0, 0);
	if (reg[0] >= 7){
		__cpuidex(reg, 1, 0); b = ((reg[2] >> 12) & 1) != 0;
		if (b){
			__cpuidex(reg, 7, 0);  b = ((reg[1] >> 5) & 1) != 0;
		}
	}
	return b;
}
```

Перевірка операційної системи

Перевірка операційної системи має сенс тільки в випадку, якщо процесор підтримує команди AVX.
При виконанні AVX команд процесор застосовує регістри довжиною 256 бітів, які позначаються YMM<номер>, далі YMM регістри. Для успішної роботи з масивом цих регістрів, операційна система повинна мати змогу зберігати ці регістри, а також оновлювати їх стан. Приклади операційних систем наведені нижче

Приклади операційних систем

Windows: підтримка почалася з Windows 7 з Service Pack 1 (липень 2010), та передбачена для усіх подальших операційних систем;
Linux: підтримка почалася з ядра версії 2.6.30 (червень 2009 року), та передбачена для усіх подальших ядер;
Apple OS X: підтримка почалася з ядра версії 10.6.8 (червень 2011 року) , та передбачена для усіх подальших ядер;

Програмна перевірка операційної системи

Як було сказано вище, щоб операційна система могла опрацьовувати AVX команди, вона повинна використовувати операції зберігання та оновлення 256 бітних регістрів (YMM регістрів, при переключенні задач (переключенні контексту). Для цього ОС повинна використовувати для зберігання – оновлення контексту команди процесору XSAVE/XRSTOR, які повинні підтримуватися процесором, а також необхідні режими переключення. Якщо процесор підтримує AVX2 команди, то він гарантовано підтримує команди XSAVE/XRSTOR, тому цей факт можна не перевіряти, а ось режими переключення треба перевірити.

Режими переключення, тобто завдання того, що треба зберігати, операційна система задає в регістрі XCR0. Для цього регістру використовується символічне ім'я _XCR_XFEATURE_ENABLED_MASK. Склад інформації в регістрі XCR0:
Біт 0 Необхідність зберігання регістрів з плаваючою точкою (FPU регістрів). Встановлено в 1 для усіх сучасних операційних систем;
Біт 1 Необхідність зберігання SSE регістрів (XMM регістрів), які фактично є молодшою частиною регістрів YMM для процесорів, які підтримують роботу з AVX командами;
Біт 2 Необхідність зберігання AVX регістрів (YMM регістрів), якщо включено, то включено і біт 1.
Таким чином, достатньо перевірити тільки Біт 2 регістру XCR0.
Для визначення вмісту регістрів керування, у тому числі регістру XCR0 достатньо використовувати команду XGETBV. Номер регістру, для якого визначається вміст, треба записати в регістр ECX перед виконанням команди XGETBV. Для регістру XCR0 цей номер дорівнює 0. Результат виконання команди – 64 бітне значення, яке записується в регістри: <EAX, EDX>, молодше значення в регістр EAX. Таким чином, ОС підтримує використання AVX команд, якщо біт2 регістру EAX встановлено в 1

```
bool AVX2OSSupport(){
        return  (_xgetbv(_XCR_XFEATURE_ENABLED_MASK) & 4) != 0;
}
```

Загальна функція для перевірки можливості використання AVX2

```
bool AVX2Support(){
        bool b = AVX2ProcessorSupport();
        if (b) b = AVX2OSSupport();
        return b;
}
```

Контрольні запитання і завдання

1 Що таке SIMD команди. Які типи команд цього класу Ви знаєте?
2. Дайте характеристику SSE команд.
3 Порівняйте SSE, AVX, AVX2 команди
4 Яка команда процесора забезпечує отримання повної інформації про нього?
5 Яким чином можна узнати тип процесору (AMD, Intel)
6 Які необхідні умови, щоб підтримувались команди AVX?
6 Які необхідні умови, щоб підтримувались команди AVX2?

Приклади аудиторних задач

1. Скласти функцію для визначення типу процесора (Intel, AMD, інші).

```
typedef enum {
        INTEL, AMD, OTHER
} ProcessorType;
ProcessorType DefineProcessorType (){
                ProcessorType Type = OTHER;
                CHAR INTELName [13] = "GenuineIntel";
                CHAR AMDName [13] = "AuthenticAMD";
                CHAR ProcessorName [13];
```

```
                ProcessorName [12] = 0;
                INT Regs [4];
                __cpuid (Regs, 0);
                memcpy (ProcessorName, &Regs [1], 4);
                memcpy (ProcessorName + 4, &Regs [3], 4);
                memcpy (ProcessorName + 8, &Regs [2], 4);
                if (memcmp (INTELName, ProcessorName, 12) == 0)
                        Type = INTEL;
                else{
                        if (memcmp (AMDName, ProcessorName, 12) == 0)
                                Type = AMD;
                }
                return Type;
}
```

Код головної програми:

```
ProcessorType Type;
Type = DefineProcessorType ();
switch (Type){
case INTEL:
                _tprintf (TEXT("ProcessorType: INTEL\n")); break;
                case AMD:
                _tprintf (TEXT("ProcessorType: AMD\n"));          break;
                case OTHER:
                _tprintf (TEXT("ProcessorType: OTHER\n")); break;
        }
```

2. Визначити розширену інформацію про процесор.

```
VOID DefineExtendProcessorType (CHAR BrandString []){
        int Regs [4];
CHAR *pBrandString = BrandString;
for (unsigned uCommand = 0x80000002; uCommand < 0x80000005;  uCommand++)    {
                __cpuid(Regs, uCommand);
                memcpy (pBrandString, &Regs [0], 4); pBrandString       +=4;
                memcpy (pBrandString, &Regs [1], 4); pBrandString       +=4;
                memcpy (pBrandString, &Regs [2], 4); pBrandString       +=4;
                memcpy (pBrandString, &Regs [3], 4); pBrandString       +=4;
        }
}
CHAR BrandString [64];
        DefineExtendProcessorType (BrandString );
        printf ("BrandString = %s\n", BrandString);
```

Зверніть увагу на те, що отримуємо ANSI інформацію, а не універсальну!

3. Скласти функції для перевірки підтримки MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2.

Визначення файлу заголовків (SIMD.h)

```
#include "stdafx.h"
#include <intrin.h>
#include <string.h>
#include <windows.h>
#include <omp.h>
typedef enum {
```

```
                UNKNOWN,
                INTEL,
                AMD
} PROCERRORTYPE;
typedef enum {
                SSESUPPORT, SSE2SUPPORT, SSE3SUPPORT,          SSSE3SUPPORT,
                SSE41SUPPORT, SSE42SUPPORT, AVXSUPPORT, AVX2SUPPORT
};

int MaxFunctionNumber (int *ExtMaxFunctionNumber);
PROCERRORTYPE GetProcessorType ();
int GetSIMDSupport ();
```

Визначення властивостей процесору

```
#include "SIMD.h"
// Визначення максимальної та додаткової функції для команди CPUID
int MaxFunctionNumber (int *ExtMaxFunctionNumber){
                int res;
                int b = 0x80000000;
                int a [4];
                // Если расширенная информация нужна
                if (ExtMaxFunctionNumber){
                        __cpuid (a, b);
                        *ExtMaxFunctionNumber = a[0];
                }
                b = 0;
                __cpuid (a, b);
                return a [0];
}

// Функція для перевірки, чи підтримує процесор команди AVX2
bool AVX2ProcessorSupport(){
        bool b = false; int reg[4];
        __cpuidex(reg, 0, 0);
        if (int MaxFunctionNumber (NULL) >= 7){
                __cpuidex(reg, 1, 0);
                b = ((reg[2] >> 12) & 1) != 0;
                if (b){
                        __cpuidex(reg, 7, 0);
                        b = ((reg[1] >> 5) & 1) != 0;
                }
        }
        return b;
}
// Функція для перевірки, чи підтримує OS роботу з командами AVX2
bool AVX2OSSupport(){
                return  (_xgetbv(_XCR_XFEATURE_ENABLED_MASK) & 4) != 0;
}
// Функція для перевірки усіх режимів.
// Спочатку перевіряється AVX2. Якщо цей режим підтримується, то решта режимів
// підтримується, потім AVX, і т.д.
int GetSIMDSupport (){
```

```
BOOL bRes;
        BOOL bOSAVXSupport = isAvxSupportedByWindows ();
if  (bOSAVXSupport){
                bRes = bool AVX2ProcessorSupport();
}
if (bRes) return ((1 << (1+AVX2SUPPORT)) -1;
int Regs[4];
__cpuid (Regs, 1);
bRes  = (Regs [2] >> 11) & 1 ? 1 : 0;
if (bRes) return (1 << (AVXSUPPORT + 1)) -1;
bRes  = (Regs [2] >> 20) & 1 ? 1:0;
if (bRes) return (1 <<(1 +  SSE42SUPPORT)) -1;
        bRes = (Regs [2] >> 19) & 1 ? 1 : 0;
if (bRes) return (1 << (1 + SSE41SUPPORT)) -1;
        bRes = (Regs [2] >> 9) & 1 ? 1 : 0;
if (bRes) return (1 << (1 +SSSE3 SUPPORT)) -1;
        bRes = (Regs [2] >>0) & 1 ? 1 : 0;
if (bRes) return (1 <<(1 +  SSE3 SUPPORT)) -1;
        bRes = (Regs [3] >>26) & 1 ? 1 : 0;
if (bRes) return (1 << (1 + SSE2 SUPPORT)) -1;
        bRes = (Regs [3] >>25) & 1 ? 1 : 0;
if (bRes) return (1 << (1 + SSE SUPPORT)) -1;
return 0;
        }
```

4. Скласти універсальну функцію для обчислення: b =∑ |a[i]| без використання SIMD команд, а потім з використанням для усіх можливих типів даних

Універсальна функція

```
template <typename T>
T Sum (T* a, size_t n){
                T r = 0;
                for (size_t i = 0; i < n; ++i)    {
                        r+= a [i] >0 ? a [i] : - a[i];
                }
                return r;
}
```

Функція з використанням SSE для даних типу float. Звертаємо Вашу увагу, що для цього типу даних стандартної функції для обчислення модулю немає

```
float SSESumF (float *a, size_t n){
        __m128 *pa =  (__m128 *)a;
        __m128 zero = _mm_setzero_ps(), r1, r2, r3, res = _mm_setzero_ps();
        float * fres = (float *)&res;
        for (size_t i = 0; i < n / (sizeof (__m128) / sizeof (a [0])); ++i){
                r1 = _mm_cmple_ps (zero, pa [i]);
                r2 = _mm_and_ps (r1, pa [i]);
                r3 = _mm_andnot_ps (r1, pa [i]);
                r3 = _mm_sub_ps (zero, r3);
                res = _mm_add_ps (res, r2);
                res = _mm_add_ps (res, r3);
        }
        return fres [0] + fres [1] + fres [2] + fres [3];
}
```

Функція з використанням AVX для даних типу float. Звертаємо Вашу увагу, що для цього типу даних стандартної функції для обчислення модулю немає

```
float AVXSumF (float *a, size_t n){
	__m256 *pa =  (__m256 *)a;
	__m256 zero = _mm256_setzero_ps(), r1, r2, r3, res = _mm256_setzero_ps();
	float * fres = (float *)&res;
	for (size_t i = 0; i < n / (sizeof (__m256) / sizeof (a [0])); ++i){
		r1 = _mm256_cmp_ps (zero, pa [i], _CMP_LE_OS);
		r2 = _mm256_and_ps (r1, pa [i]);
		r3 = _mm256_andnot_ps (r1, pa [i]);
		r3 = _mm256_sub_ps (zero, r3);
		res = _mm256_add_ps (res, r2);
		res = _mm256_add_ps (res, r3);
	}
	return fres [0] + fres [1] + fres [2] + fres [3];
}
```

Функція з використанням SSE для даних типу int. Стандартна функція для обчислення абсолютного значення є

```
int SSESumI(int *a, size_t n){
	__m128i *pa = (__m128i *)a;
	__m128i r1, res = _mm_setzero_si128();
	int * ires = (int* )&res;
	for (size_t i = 0; i < n / (sizeof(__m128) / sizeof(a[0])); ++i){
		r1 = _mm_abs_epi32(pa[i]);
		res = _mm_add_epi32(res, r1);
	}
	return ires[0] + ires[1] + ires[2] + ires[3];
}
```

Функція з використанням AVX для даних типу int. Стандартна функція для обчислення абсолютного значення є

```
int AVXSumI(int *a, size_t n){
	__m256i *pa = (__m256i *)a;
	__m256i r1, res = _mm256_setzero_si256();
	int * ires = (int*)&res;
	for (size_t i = 0; i < n / (sizeof(__m256) / sizeof(a[0])); ++i){
		r1 = _mm256_abs_epi32(pa[i]);
		res = _mm256_add_epi32(res, r1);
	}
	return ires[0] + ires[1] + ires[2] + ires[3] +
		ires[4] + ires[5] + ires[6] + ires[7];
}
```

Обов’язково визначте час виконання для кожного варіанта використання функцій та зробіть висновки по ефективності використання SIMD операцій!!!

5 Скласти функцію для визначення максимального елементу без та з використання SIMD команд

Універсальна функція

```
template <typename T>
T MaxItem (T *matr, size_t n){
	T Max = matr [0];
	for (size_t i = 1; i < n * n; ++i){
		if (Max < matr [i])     Max = matr [i];
	}
```

```
	return Max;
}
```

Функція з використання SSE для даних типу float

```
float SSEMaxItemF (float *matr, size_t n){
	__m128 *pm = (__m128 *)matr;
	__m128 Max = pm [0];
	float *max = (float *)&Max;
	for (size_t i = 1; i < n * n / (sizeof (__m128) / sizeof (matr [0])); ++i){
		Max = _mm_max_ps (Max, pm [i]);
	}
	float r1 = max [0] > max [1]? max [0] : max [1];
	if (r1 < max [2]) r1 = max [2];
	if (r1 < max [3]) r1 = max [3];
	return r1;
}

float AVXMaxItemF (float *matr, size_t n){
	__m256 *pm = (__m256 *)matr;
	__m256 Max = pm [0];
	float *max = (float *)&Max;
	for (size_t i = 1; i < n * n / (sizeof (__m256) / sizeof (matr [0])); ++i){
		Max = _mm256_max_ps (Max, pm [i]);
	}
	float r1 = max [0] > max [1]? max [0] : max [1];
	if (r1 < max [2]) r1 = max [2];
	if (r1 < max [3]) r1 = max [3];
	return r1;
}
```

Обов’язково визначте час виконання для кожного варіанта використання функцій та зробіть висновки по ефективності використання SIMD операций!!!

Приклади домашніх задач

1 Для завдань 4 та 5 реалізуйте функції для решти типів даних. дослідіть їх ефективність.