підготував асистент каф. ЕОМ Козак Назар Богданович

Практичне заняття № 5
(за темою лабораторної роботи №3)

Використання потокового графу алгоритму при проектуванні паралельних обчислень.

Паралельні обчислення.

Паралельні обчислення — це форма обчислень, в яких кілька дій проводяться одночасно. Є кілька різних рівнів паралельних обчислень: ***бітовий***, ***інструкцій***, ***даних*** та ***паралелізм задач***. Паралельні обчислення застосовуються вже протягом багатьох років, в основному в високопродуктивних обчисленнях, але зацікавлення ними зросло тільки недавно, через фізичні обмеження зростання частоти. Оскільки споживана потужність (і відповідно виділення тепла) комп'ютерами стало проблемою в останні роки, паралельне програмування стає домінуючою парадигмою в комп'ютерній архітектурі, в основному в формі багатоядерних процесорів.

Паралельні комп'ютери системи можуть бути грубо класифіковані згідно з рівнем, на якому апаратне забезпечення підтримує паралелізм: багатоядерні процесори; багатопроцесорні комп'ютери — комп'ютери, що мають багато обчислювальних елементів в межах одної машини, також сюди відносимо кластери; MPP та GRID — системи що використовують багато комп'ютерів для роботи над одним завданням. Також Іноді, поряд з цими традиційними комп'ютерними системами, використовуються спеціалізовані паралельні архітектури для прискорення особливих задач.

Програми для паралельних комп'ютерів писати значно складніше, ніж для послідовних, бо паралелізм додає кілька нових класів потенційних помилок, серед яких найпоширенішою є стан гонитви(конкуренції обчислювальних процесів). Комунікація, та синхронізація процесів зазвичай одна з найбільших перешкод для досягнення хорошої продуктивності паралельних програм.

Паралельні обчислення рівня машинних інструкцій.

Комп'ютерна програма, по суті, є потоком інструкцій, які виконуються процесором. Іноді ці інструкції можна перевпорядкувати, та об'єднати в групи, які потім виконувати паралельно, без зміни результату роботи програми, що відомо як паралелізм на рівні інструкцій. Такий підхід до збільшення продуктивності обчислень переважав з середини 80-тих до середини 90-тих. Сучасні процесори мають багатоетапні конвеєри команд. Кожен етап конвеєра відповідає іншій дії, що виконує процесор. Процесор що має конвеєр з N-ступенями, може одночасно обробляти N інструкцій, кожну на іншій стадії обробки. Класичним прикладом процесора з конвеєром є процесор архітектури RISC, що має п'ять етапів: завантаження інструкції, декодування, виконання, доступ до пам'яті, та запис результату. Процесор Pentium 4 має конвеєр з 35 етапами. На додачу до паралелізму на рівні інструкцій деякі процесори можуть виконувати більш ніж одну інструкцію за раз. Вони відомі як суперскалярні процесори. Інструкції групуються разом, якщо між ними не існує залежності даних. Щоб реалізувати паралелізм на рівні інструкцій використовують алгоритми Scoreboarding та Tomasulo algorithm (який аналогічний до попереднього, проте використовує перейменування регістрів).

Використання потокового графу алгоритму для паралельного програмування.

Потоковий граф алгоритму описує паралельні обчислення, в яких на заданому фрагменті виконання відсутні залежності керування за даними. Кожна вершина графу на одному ярусі виконується одночасно. Приміром на *рис.1.1* показано виконання швидкого перетворення Фур'є для 8-ми відліків.

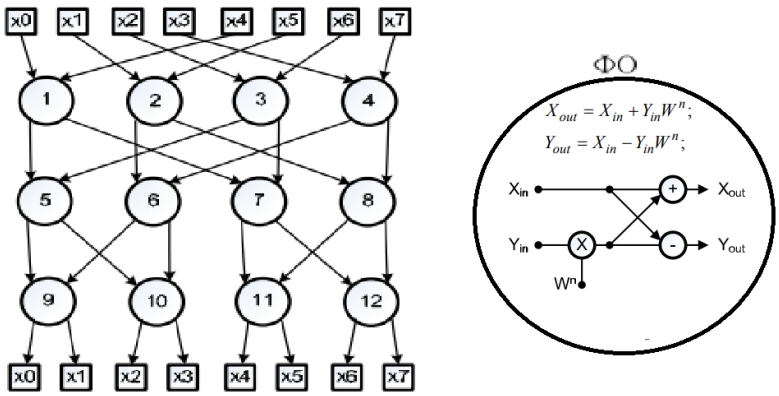
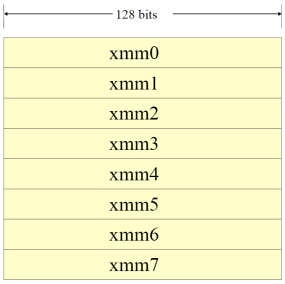


Рис.1.1. Приклад потокового графу алгоритму

Набір інструкцій SSE2.

SSE (англ. Streaming SIMD Extensions, потокове SIMD-розширення процесора) — це SIMD (англ. Single Instruction, Multiple Data, Одна інструкція — багато даних) набір інструкцій, розроблених Intel, і вперше представлених у процесорах серії Pentium III як відповідь на аналогічний набір інструкцій 3DNow! від AMD, який був представлений роком раніше. Цікаво, що початкова назва цих інструкцій була KIN, що розшифровувалася як Katmai New Instructions (Katmai — назва першої версії ядра процесора Pentium III).



SSE2 використовує вісім 128-бітних регістри (xmm0 до xmm7), що увійшли до архітектури x86 з вводом розширення SSE, кожний з яких трактується як послідовність 2 значень з плаваючою комою подвійної точності. SSE2 включає в себе набір інструкцій, який виконує дії зі скалярними і упакованими типами даних. Також SSE2 містить інструкції для потокової обробки даних з фіксованою комою у тих же 128-бітних xmm регістрах.

Лістинг 1.1

```
float a[4] = { 300.0, 4.0, 4.0, 12.0 };
float b[4] = {   1.5, 2.5, 3.5,  4.5 };
_asm {
movups xmm0, a   ; // помістити 4 змінні з рухомою крапкою із a в регістр xmm0
movups xmm1, b   ; // помістити 4 змінні з рухомою крапкою із b в регістр xmm1
mulps xmm1, xmm0 ; // перемножити пакети рухомих крапок: xmm1=xmm1*xmm0
                 ; // xmm10 = xmm10*xmm00
                 ; // xmm11 = xmm11*xmm01
                 ; // xmm12 = xmm12*xmm02
                 ; // xmm13 = xmm13*xmm03
movups a, xmm1   ; // вивантажити результати із регістра xmm1 по адресам a
```

```
};
```

Робата з SSE2 на рівні мов програмування високого рівня.

Для роботи з SSE, SSE2, SSE3, SSE4 та AVX в C/C++ зручно використовувати обгортки над цими інструкціями процесора. Їх реалізовують ***intrinsics-функції***, вичерпну інформацію по яких можна отримати за лінком: https://software.intel.com/sites/landingpage/IntrinsicsGuide/ .

Приклад.

Математичні вирази зручно подавати у формі потокового графу алгоритму, оскільки в них обо відсутні залежності керування за даними, або їх легко трансформувати в додаткові обчислення для вибору результату. Розглянемо потоковий граф алгоритму розв'язку квадратного рівняння *(рис. 2.1.)* для двократного розпаралелення при одночасному виконанні тільки однотипних інструкцій(відповідно до концепції SSE2).

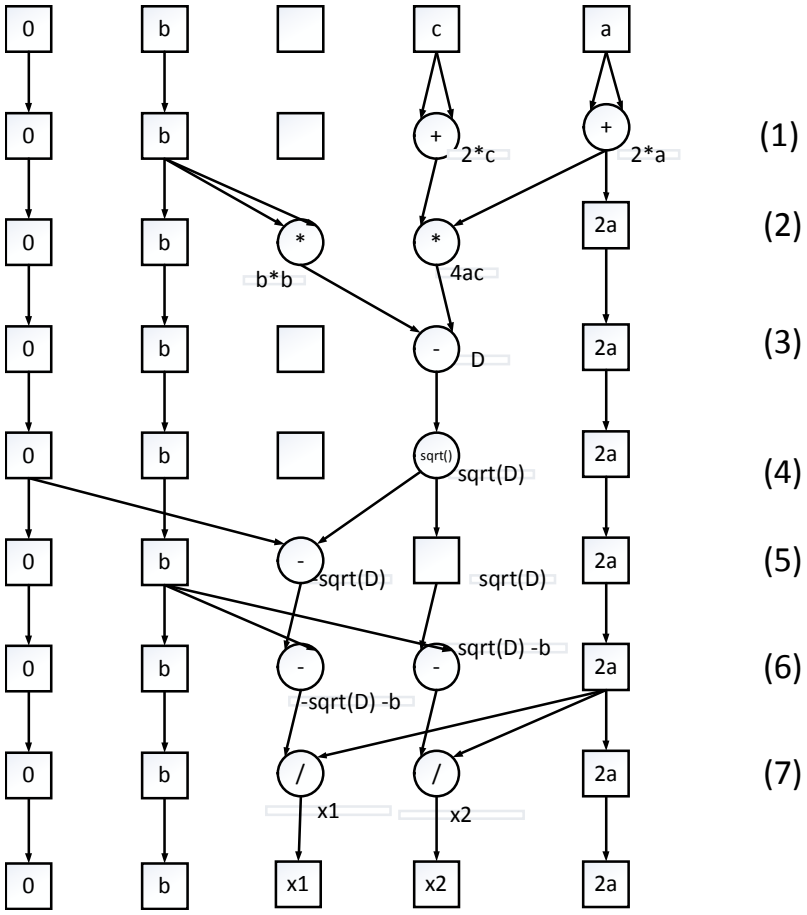


Рис. 2.1. ПГА для квадратного рівняння

ЛАБОРАТОРНА РОБОТА №3

Назва роботи: використання потокового графу алгоритму при проектуванні паралельних обчислень.

Мета роботи: ознайомитися з використанням потокового графу алгоритму при паралельному програмуванні.

*(детальніша інформація про **паралельні обчислення**, **паралельне програмування** та **потоковий граф алгоритму** в матеріалах лекційних занять)*

1.1. Паралельні обчислення.

Паралельні обчислення — це форма обчислень, в яких кілька дій проводяться одночасно. Є кілька різних рівнів паралельних обчислень: ***бітовий***, ***інструкцій***, ***даних*** та ***паралелізм задач***. Паралельні обчислення застосовуються вже протягом багатьох років, в основному в високопродуктивних обчисленнях, але зацікавлення ними зросло тільки недавно, через фізичні обмеження зростання частоти. Оскільки споживана потужність (і відповідно виділення тепла) комп'ютерами стало проблемою в останні роки, паралельне програмування стає домінуючою парадигмою в комп'ютерній архітектурі, в основному в формі багатоядерних процесорів.

Паралельні комп'ютери системи можуть бути грубо класифіковані згідно з рівнем, на якому апаратне забезпечення підтримує паралелізм: багатоядерні процесори; багатопроцесорні комп'ютери — комп'ютери, що мають багато обчислювальних елементів в межах одної машини, також сюди відносимо кластери; MPP та GRID — системи що використовують багато комп'ютерів для роботи над одним завданням. Також Іноді, поряд з цими традиційними комп’ютерними системами, використовуються спеціалізовані паралельні архітектури для прискорення особливих задач.

Програми для паралельних комп'ютерів писати значно складніше, ніж для послідовних, бо паралелізм додає кілька нових класів потенційних помилок, серед яких найпоширенішою є стан гонитви(конкуренції обчислювальних процесів). Комунікація, та синхронізація процесів зазвичай одна з найбільших перешкод для досягнення хорошої продуктивності паралельних програм.

1.2. Паралельні обчислення рівня машинних інструкцій.

Комп'ютерна програма, по суті, є потоком інструкцій, які виконуються процесором. Іноді ці інструкції можна перевпорядкувати, та об'єднати в групи, які потім виконувати паралельно, без зміни результату роботи програми, що відомо як паралелізм на рівні інструкцій. Такий підхід до збільшення продуктивності обчислень переважав з середини 80-тих до середини 90-тих. Сучасні процесори мають багатоетапні конвеєри команд. Кожен етап конвеєра відповідає іншій дії, що виконує процесор. Процесор що має конвеєр з N-ступенями, може одночасно обробляти N інструкцій, кожну на іншій стадії обробки. Класичним прикладом процесора з конвеєром є процесор архітектури RISC, що має п'ять етапів: завантаження інструкції, декодування, виконання, доступ до пам'яті, та запис результату. Процесор Pentium 4 має конвеєр з 35 етапами. На додачу до паралелізму на рівні інструкцій деякі процесори можуть виконувати більш ніж одну інструкцію за раз. Вони відомі як суперскалярні процесори. Інструкції групуються разом, якщо між ними не існує залежності даних. Щоб реалізувати паралелізм на рівні інструкцій використовують алгоритми Scoreboarding та Tomasulo algorithm (який аналогічний до попереднього, проте використовує перейменування регістрів).

1.3. Використання потокового графу алгоритму для паралельного програмування.

Потоковий граф алгоритму описує паралельні обчислення, в яких на заданому фрагменті виконання відсутні залежності керування за даними. Кожна вершина графу на одному ярусі виконується одночасно. Приміром на *рис.1.1* показано виконання швидкого перетворення Фур'є для 8-ми відліків.

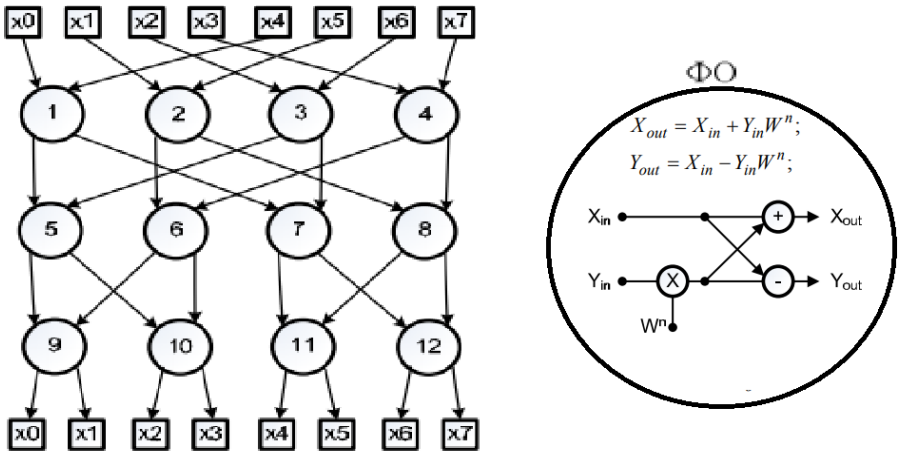
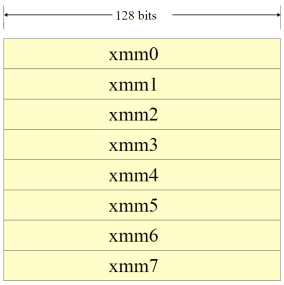


Рис.1.1. Приклад потокового графу алгоритму

1.4. Набір інструкцій SSE2.

SSE (англ. Streaming SIMD Extensions, потокове SIMD-розширення процесора) — це SIMD (англ. Single Instruction, Multiple Data, Одна інструкція — багато даних) набір інструкцій, розроблених Intel, і вперше представлених у процесорах серії Pentium III як відповідь на аналогічний набір інструкцій 3DNow! від AMD, який був представлений роком раніше. Цікаво, що початкова назва цих інструкцій була KIN, що розшифровувалася як Katmai New Instructions (Katmai — назва першої версії ядра процесора Pentium III).



SSE2 використовує вісім 128-бітних регістри (xmm0 до xmm7), що увійшли до архітектури x86 з вводом розширення SSE, кожний з яких трактується як послідовність 2 значень з плаваючою комою подвійної точності. SSE2 включає в себе набір інструкцій, який виконує дії зі скалярними і упакованими типами даних. Також SSE2 містить інструкції для потокової обробки даних з фіксованою комою у тих же 128-бітних xmm регістрах.

Лістинг 1.1

```
float a[4] = { 300.0, 4.0, 4.0, 12.0 };
float b[4] = {   1.5, 2.5, 3.5,  4.5 };
_asm {
movups xmm0, a   ; // помістити 4 змінні з рухомою комою із a в регістр xmm0
movups xmm1, b   ; // помістити 4 змінні з рухомою комою із b в регістр xmm1
mulps xmm1, xmm0 ; // перемножити пакети: xmm1=xmm1*xmm0
                 ; // xmm10 = xmm10*xmm00
                 ; // xmm11 = xmm11*xmm01
                 ; // xmm12 = xmm12*xmm02
                 ; // xmm13 = xmm13*xmm03
movups a, xmm1   ; // вивантажити результати із регістра xmm1 в a
};
```

1.5. Робата з SSE2 на рівні мов програмування високого рівня.

Для роботи з SSE, SSE2, SSE3, SSE4 та AVX в C/C++ зручно використовувати обгортки над цими інструкціями процесора. Їх реалізовують ***intrinsics-функції***, вичерпну інформацію по яких можна отримати за лінком: https://software.intel.com/sites/landingpage/IntrinsicsGuide/ .

2. Приклад.

Математичні вирази зручно подавати у формі потокового графу алгоритму, оскільки в них обо відсутні залежності керування за даними, або їх легко трансформувати в додаткові обчислення для вибору результату. Розглянемо потоковий граф алгоритму розв'язання квадратного рівняння *(рис. 2.1.)* для двократного розпаралелення при одночасному виконанні тільки однотипних інструкцій(відповідно до концепції SSE2).

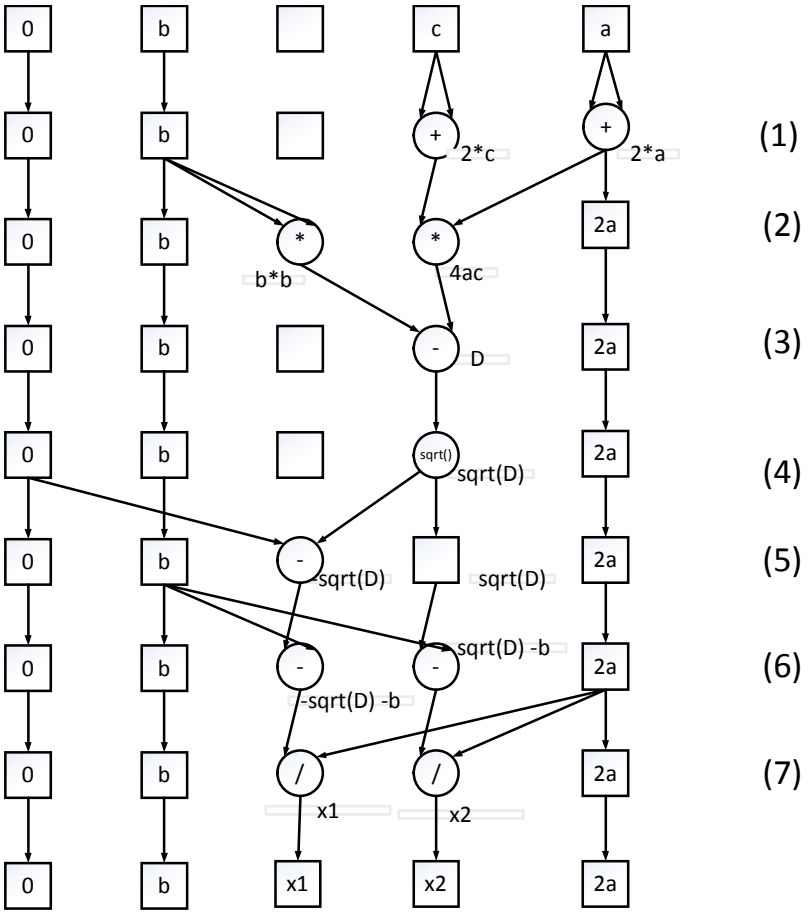


Рис. 2.1. ПГА для розв'язання квадратного рівняння

3. Приклад програми.

Лістинг 3.1

```
// don't forget to use compilation key for Linux: -lm

//-fno-tree-vectorize
//gcc -O3 -no-tree-vectorize
//gcc -O3 -ftree-vectorizer-verbose=6 -msse4.1 -ffast-math
//__attribute__((optimize("no-tree-vectorize")))
//extern "C" __attribute__ ((optimize("no-tree-vectorize")))


#include <stdio.h>
#include <stdlib.h>
//#include <x86intrin.h> // Linux
#include <intrin.h> // Windows
#include <math.h>
#include <time.h>

#define A  0.33333333
#define B  0.
#define C -3.

#pragma GCC push_options
#pragma GCC optimize ("no-unroll-loops")

#define REPEAT_COUNT 1000000
#define REPEATOR(count, code) \
for (unsigned int indexIteration = (count); indexIteration--;){ code; }
#define TWO_VALUES_SELECTOR(variable, firstValue, secondValue) \
	(variable) = indexIteration % 2 ? (firstValue) : (secondValue);

double getCurrentTime(){
	clock_t time = clock();
	if (time != (clock_t)-1) {
		return ((double)time / (double)CLOCKS_PER_SEC);
	}
	return 0.; // else
}

#pragma GCC push_options
#pragma GCC target ("no-sse2")
//__attribute__((__target__("no-sse2")))
void run_native(double * const dArr){
	double * const dAC = dArr;
	double * const dA = &dAC[0];
	double * const dC = &dAC[1];
	double * const dB = &dArr[2];
	double * const dResult = &dArr[4];
	double * const dX1 = &dResult[1];
	double * const dX2 = &dResult[0];

	REPEATOR(REPEAT_COUNT,
		TWO_VALUES_SELECTOR(*dA, 4., A);
		TWO_VALUES_SELECTOR(*dB, 3., B);
		TWO_VALUES_SELECTOR(*dC, 1., C);
		double vD = sqrt((*dB)*(*dB) - 4.*(*dA)*(*dC));
		(*dX1) = (-(*dB) + vD) / (2.*(*dA));
		(*dX2) = (-(*dB) - vD) / (2.*(*dA));
	)
}
#pragma GCC pop_options

void run_SSE2(double * const dArr){
	double * const dAC = dArr;
```

```
	double * const dA = &dAC[0];
	double * const dC = &dAC[1];
	double * const dB = &dArr[2];
	double * const dResult = &dArr[4];
	double * const dX1 = &dResult[1];
	double * const dX2 = &dResult[0];

	__m128d r__zero_zero, r__c_a, r__uORb_b, r__2cORbOR2a_2a,
		r__zero_bb, r__sqrtDiscriminant_zero, r_result;

	r__zero_zero = _mm_set_pd(0., 0.); // init

	REPEATOR(REPEAT_COUNT,
		TWO_VALUES_SELECTOR(*dA, 4., A);
		TWO_VALUES_SELECTOR(*dB, 3., B);
		TWO_VALUES_SELECTOR(*dC, 1., C);
		r__c_a = _mm_load_pd(dAC);
		// r__uORb_b = _mm_load_pd1(dB);
		r__uORb_b = _mm_load1_pd(dB);
		// b b
		r__uORb_b = _mm_unpacklo_pd(r__uORb_b, r__uORb_b);
		// (etap 1)
		r__2cORbOR2a_2a = _mm_add_pd(r__c_a, r__c_a);
		// b 2c
		r_result = _mm_unpackhi_pd(r__2cORbOR2a_2a, r__uORb_b);
		// b 2a
		r__2cORbOR2a_2a = _mm_unpacklo_pd(r__2cORbOR2a_2a, r__uORb_b);
		// bb 4ac (etap 2)
		r_result = _mm_mul_pd(r_result, r__2cORbOR2a_2a);
		r__zero_bb = _mm_unpackhi_pd(r_result, r__zero_zero);
		// zero Discriminant (etap 3)
		r_result = _mm_sub_sd(r__zero_bb, r_result);
		// zero sqrtDiscriminant (etap 4)
		r_result = _mm_sqrt_sd(r_result, r_result);
		r__sqrtDiscriminant_zero = _mm_shuffle_pd(r_result, r_result, 1);
		// sqrtDiscriminant -sqrtDiscriminant (etap 5)
		r_result = _mm_sub_sd(r__sqrtDiscriminant_zero, r_result);
		// (etap 6)
		r_result = _mm_sub_pd(r_result, r__uORb_b);
		// 2a 2a
		r__2cORbOR2a_2a = _mm_unpacklo_pd(r__2cORbOR2a_2a, r__2cORbOR2a_2a);
		// (etap 7)
		r_result = _mm_div_pd(r_result, r__2cORbOR2a_2a);
		_mm_store_pd(dResult, r_result);
	)
}

void printResult(char * const title, double * const dArr, unsigned int runTime){
	double * const dAC = dArr;
	double * const dA = &dAC[0];
	double * const dC = &dAC[1];
	double * const dB = &dArr[2];
	double * const dResult = &dArr[4];
	double * const dX1 = &dResult[1];
	double * const dX2 = &dResult[0];

	printf("%s:\r\n", title);
	printf("%fx^2 + %fx + %f = 0;\r\n", *dA, *dB, *dC);
	printf("x1 = %1.0f; x2 = %1.0f;\r\n", *dX1, *dX2);
	printf("run time: %dns\r\n\r\n", runTime);
}

int main() {
	double * const dArr = (double *)_mm_malloc(6 * sizeof(double), 16);
```

```
        double * const dAC = dArr;
        double * const dA = &dAC[0];
        double * const dC = &dAC[1];
        double * const dB = &dArr[2];
        double * const dResult = &dArr[4];
        double * const dX1 = &dResult[1];
        double * const dX2 = &dResult[0];

        double startTime, endTime;

        // native (only x86, if auto vectorization by compiler is off)
        startTime = getCurrentTime();
        run_native(dArr);
        endTime = getCurrentTime();
        printResult("x86",
               dArr,
               (unsigned int)((endTime - startTime) * (1000000000 / REPEAT_COUNT)));

        // SSE2
        startTime = getCurrentTime();
        run_SSE2(dArr);
        endTime = getCurrentTime();
        printResult("SSE2",
               dArr,
               (unsigned int)((endTime - startTime) * (1000000000 / REPEAT_COUNT)));

        _mm_free(dArr);

        printf("Press any key to continue . . .");
        getchar();
        return 0;
}

#pragma GCC pop_options
```

4. Спрощене завдання.

Скласти програму (C/C++), яка дозволяє провести порівняння двох реалізацій розв'язання квадратного рівняння(звичайний послідовний код та код на основі інструкцій SSE2, що відображає потоковий граф алгоритму) за характеристикою часової складності для таких вхідних даних:

n	**Вхідні дані**		
	c	**b**	**a**
1	111	112	107
2	145	146	141
3	165	166	161
4	185	186	181
5	205	206	201
6	225	226	221
7	245	246	241
8	265	266	261
9	285	286	281
10	305	306	301
11	325	326	321
12	345	346	341
13	365	366	361
14	385	386	381
15	405	406	401
16	425	426	421

17	445	446	441
18	465	466	461
19	485	486	481
20	505	506	501
21	525	526	521
22	545	546	541
23	565	566	561
24	585	586	581
25	605	606	601
26	625	626	621
27	645	646	641
28	665	666	661
29	685	686	681
30	705	706,	701
n – порядковий номер у журналі			

**** Для отримання 50% балів за лабораторну роботу можна використати наявний програмний код з лістингу 3.1. Для отримання 100% балів за лабораторну роботу потрібно написати власний код.***

5. Завдання базового рівня.

Скласти програму (C/C++), яка дозволяє провести порівняння двох реалізацій обчислення виразу(звичайний послідовний код та код на основі інструкцій SSE2, що відображає потоковий граф алгоритму) за характеристикою часової складності для вхідних даних, що вводяться під час роботи програми.

Варіант	**Вираз**
1	y = (a + b) + (c + d) + (e + f) + (g + h)
2	y = (a + b) - (c + d) + (e + f) - (g + h)
3	y = (a + b) * (c + d) + (e + f) * (g + h)
4	y = (a + b) / (c + d) + (e + f) / (g + h)
5	y = (a + b) + (c + d) + (e + f) + (g + h)
6	y = (a + b) - (c + d) + (e + f) - (g + h)
7	y = (a + b) * (c + d) + (e + f) * (g + h)
8	y = (a + b) / (c + d) + (e + f) / (g + h)
9	y = (a + b) + (c + d) + (e - f) + (g - h)
10	y = (a + b) - (c + d) + (e - f) - (g - h)
11	y = (a + b) * (c + d) + (e - f) * (g - h)
12	y = (a + b) / (c + d) + (e - f) / (g - h)
13	y = (a + b) + (c + d) + (e - f) + (g - h)
14	y = (a + b) - (c + d) + (e - f) - (g - h)
15	y = (a + b) * (c + d) + (e - f) * (g - h)
16	y = (a + b) / (c + d) + (e - f) / (g - h)
17	y = (a - b) + (c - d) + (e + f) + (g + h)
18	y = (a - b) - (c - d) + (e + f) - (g + h)
19	y = (a - b) * (c - d) + (e + f) * (g + h)
20	y = (a - b) / (c - d) + (e + f) / (g + h)
21	y = (a - b) + (c - d) + (e + f) + (g + h)
22	y = (a - b) - (c - d) + (e + f) - (g + h)
23	y = (a - b) * (c - d) + (e + f) * (g + h)
24	y = (a - b) / (c - d) + (e + f) / (g + h)
25	y = (a - b) + (c - d) + (e - f) + (g - h)
26	y = (a - b) - (c - d) + (e - f) - (g - h)
27	y = (a - b) * (c - d) + (e - f) * (g - h)
28	y = (a - b) / (c - d) + (e - f) / (g - h)
29	y = (a - b) + (c - d) + (e - f) + (g - h)
30	y = (a - b) - (c - d) + (e - f) - (g + h)
n – порядковий номер у журналі	

6. Зміст звіту

- Титульний лист;
- Завдання;
- Алгоритм рішення завдання;
- Код програми;
- Екранна форма з результатами роботи програми;
- Висновки.