Кафедра Параллельных вычислительных технологий

Лабораторная работа №2
по дисциплине «Архитектура ЭВМ и ВС»

Цель работы

Исследовать возможности оптимизации программы при компилировании и узнать, как влияет оптимизация исходного кода на скорость выполнения программы.

План работы

1. Написать на Си или Си++ программу умножения двух квадратных матриц с использованием SIMD расширений архитектуры x86 (SSE, SSE2, SSE3, AVX) двумя различными способами:
    - Используя SIMD intrinsics
    - Используя возможности компилятора по генерации SIMD команд (необходимо приложить фрагмент аннотированного ассемблерного листинга с сгенерированными компилятором SIMD командами)

2. Проверить правильность работы программы на нескольких тестовых наборах входных данных.

3. Измерить время работы подпрограммы умножения матриц для матриц размеров, выбранных в первой работе.

4. Построить графики зависимости времени работы подпрограммы умножения в зависимости от размера матриц для самого быстрого варианта из первой работы и для двух реализованных вариантов этой (второй) работы.
В качестве самого быстрого варианта из первой работы выбран уровень оптимизации O3, а также O2.

5. Оформить отчет о проделанной работе.

Исходный код

Умножение транспонированной матрицы из первой работы.

```
#include <iostream>
#include <ctime>
#include <stdlib.h>

void translation_multiply(int n);
void translation(int n);

const int M_SIZE = 10000;

int matrix1[M_SIZE][M_SIZE];
int matrix2[M_SIZE][M_SIZE];
int matrix3[M_SIZE][M_SIZE];

int main()
{
    struct timespec cl_start, cl_end;
    long int runtime_sec;
    long int runtime_ms;

    int n = 0;
    std::cin >> n;

    for(int row = 0; row < n; row++)
        for(int column = 0; column < n; column++)
        {
            matrix1[row][column] = rand() % 100;
            matrix2[row][column] = rand() % 100;
        }

    translation(n);
    clock_gettime(CLOCK_REALTIME, &cl_start);
    translation_multiply(n);
    clock_gettime(CLOCK_REALTIME, &cl_end);

    runtime_sec = cl_end.tv_sec - cl_start.tv_sec;
    runtime_ms = (cl_end.tv_nsec - cl_start.tv_nsec) / 1000000;
    if (runtime_ms < 0)
    {
        runtime_sec--;
        runtime_ms += 1000;
    }
    printf("Time transl.: %ld,%0.3ld sec.\n", runtime_sec, runtime_ms);

    return 0;
}


void translation_multiply(int n)
{
    for(int row = 0; row < n; row++)
        for(int column = 0; column < n; column++)
```

```
            for(int cnt = 0; cnt < n; cnt++)
                matrix3[row][column] += matrix1[row][cnt] *
matrix2[column][cnt];
}


void translation(int n)
{
    int temp;

    for(int cnt = 0; cnt < n; cnt++)
        for(int row = cnt+1, column = cnt+1; row < n; row++, column++)
        {
            temp = matrix2[row][cnt];
            matrix2[row][cnt] = matrix2[cnt][column];
            matrix2[cnt][column] = temp;
        }
}
```

Комментарий: Самое быстрое с -O2 и с -O3.

Компиляция:

```
> g++ -O2 transl_mult.cpp
> g++ -O3 transl_mult.cpp
```

Умножение с использованием SIMD-intrinsics (AVX, AVX2).

```
#include <iostream>
#include <ctime>
#include <stdlib.h>
#include <immintrin.h>

void avx_multiply(int n);

const int M_SIZE = 10000;

int matrix1[M_SIZE][M_SIZE];
int matrix2[M_SIZE][M_SIZE];
int matrix3[M_SIZE][M_SIZE];

int main()
{
    struct timespec cl_start, cl_end;
    long int runtime_sec;
    long int runtime_ms;

    int n = 0; // Divide of 16
    std::cin >> n;

    for(int row = 0; row < n; row++)
        for(int column = 0; column < n; column++)
        {
            matrix1[row][column] = rand() % 100;
            matrix2[row][column] = rand() % 100;
        }
```

```
    clock_gettime(CLOCK_REALTIME, &cl_start);
    avx_multiply(n);
    clock_gettime(CLOCK_REALTIME, &cl_end);

    runtime_sec = cl_end.tv_sec - cl_start.tv_sec;
    runtime_ms = (cl_end.tv_nsec - cl_start.tv_nsec) / 1000000;
    if (runtime_ms < 0)
    {
        runtime_sec--;
        runtime_ms += 1000;
    }
    printf("Time avx: %ld,%0.3ld sec.\n", runtime_sec, runtime_ms);

    return 0;
}


void avx_multiply(int n)
{
    for(int row = 0; row < n; row++)
    {
        for(int column = 0; column < n; column += 16)
        {
            __m256i sum1 = _mm256_setzero_si256();
            __m256i sum2 = _mm256_setzero_si256();
            for(int cnt = 0; cnt < n; cnt++)
            {
                __m256i vec1 = _mm256_set1_epi32(matrix1[row][cnt]);
                __m256i vec2a =
_mm256_loadu_si256((__m256i*)&matrix2[cnt][column]);
                __m256i vec2b =
_mm256_loadu_si256((__m256i*)&matrix2[cnt][column+8]);
                __m256i mul1 = _mm256_mullo_epi32(vec1, vec2a);
                __m256i mul2 = _mm256_mullo_epi32(vec1, vec2b);
                sum1 = _mm256_add_epi32(sum1, mul1);
                sum2 = _mm256_add_epi32(sum2, mul2);
            }
            _mm256_storeu_si256((__m256i*)&matrix3[row][column], sum1);
            _mm256_storeu_si256((__m256i*)&matrix3[row][column+8], sum2);
        }
    }
}
```

Комментарий: Чем больше векторов используется во время одной итерации (первый вложенный цикл avx_multiply), тем больше скорость перемножения матриц большИх размеров, и тем меньше она на матрицах с маленькой размерностью. В программе используется два вектора - такая реализация дает преимущество уже на матрицах размерности меньше тысячи. Если использовать 4 вектора, преимущество будет получено на размерности матрицы только более 4000.

Компиляция:

```
> g++ -march=native -O2 avx_mult.cpp
```

Простое умножение матриц с O3 компиляцией.

```
#include <iostream>
#include <ctime>
#include <stdlib.h>

void default_multiply(int n);

const int M_SIZE = 10000;

int matrix1[M_SIZE][M_SIZE];
int matrix2[M_SIZE][M_SIZE];
int matrix3[M_SIZE][M_SIZE];

int main()
{
    struct timespec cl_start, cl_end;
    long int runtime_sec;
    long int runtime_ms;

    int n = 0;
    std::cin >> n;

    for(int row = 0; row < n; row++)
        for(int col = 0; column < n; column++)
        {
            matrix1[row][column] = rand() % 100;
            matrix2[row][column] = rand() % 100;
        }

    clock_gettime(CLOCK_REALTIME, &cl_start);
    default_multiply(n);
    clock_gettime(CLOCK_REALTIME, &cl_end);

    runtime_sec = cl_end.tv_sec - cl_start.tv_sec;
    runtime_ms = (cl_end.tv_nsec - cl_start.tv_nsec) / 1000000;
    if (runtime_ms < 0)
    {
        runtime_sec--;
        runtime_ms += 1000;
    }
    printf("Time def.: %ld,%0.3ld sec.\n", runtime_sec, runtime_ms);

    return 0;
}


void default_multiply(int n)
{
    for(int row = 0; row < n; row++)
        for(int column = 0; column < n; column++)
            for(int cnt = 0; cnt < n; cnt++)
                matrix3[row][column] += matrix1[row][cnt] *
matrix2[cnt][column];
```

```
}
```

Часть ассемблерного листинга предыдущей программы:

```
.L58:
        vbroadcastss    (%r11), %xmm1
        movq    -40(%rsp), %rax
        xorl    %ecx, %ecx
        .p2align 4,,10
        .p2align 3
.L38:
        vpmulld (%rdx,%rcx), %xmm1, %xmm0
        vpaddd  (%rax,%rcx), %xmm0, %xmm0
        vmovaps %xmm0, (%rax,%rcx)
        addq    $16, %rcx
        cmpq    %rdi, %rcx
        jne     .L38
        movl    -28(%rsp), %ecx
        movq    %rax, -40(%rsp)
        cmpl    %ecx, %r14d
        je      .L39
        movl    -24(%rsp), %r12d
```

Комментарий:

Компиляция:

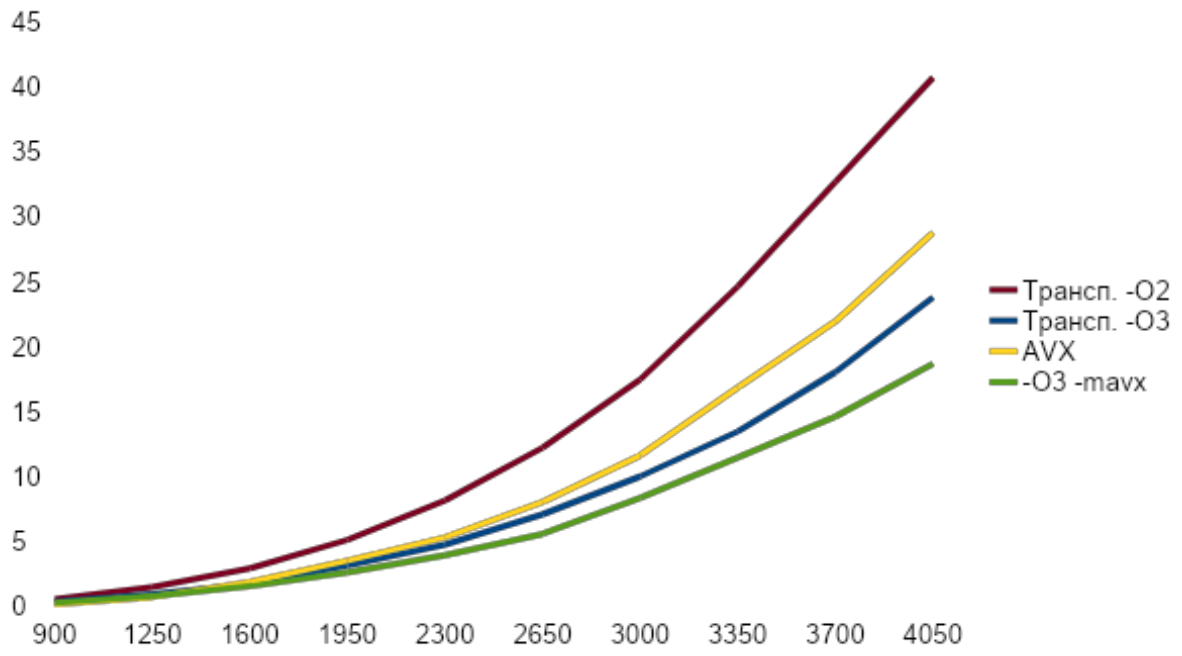
```
> g++ -O3 -mavx default_mult.cpp
```

Получение листинга:

```
> g++ -O3 -S -mavx default_mult.cpp -o assembler_listing.txt
```

Графики зависимости времени счета от размеров матриц разных программ:

X - размерность матриц
Y - время/сек.



Вывод: SIMD intrinsics позволяют сильно оптимизировать программы, но оптимизирующий компилятор делает это лучше даже без их использования. И всё же, использование SIMD intrinsics позволило немного приблизиться к скорости с оптимизацией -O3 и получить сильный выигрыш в скорости по сравнению с умножением на транспонированную матрицу с -O2 из первой работы.