ГУАП

КАФЕДРА №14

ОТЧЕТ

ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

Коренева Е.А

Должность, уч. степень, звание подпись, дата инициалы, фамилия

**ОТЧЕТ О ПРАКТИЧЕСКОЙ РАБОТЕ**

тема: SIMD инструкции

по курсу: ПРОГРАММИРОВАНИЕ НА ЯЗЫКАХ АССЕМБЛЕРА

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. 1441 Лубинец М.И

подпись, дата инициалы, фамилия

Санкт-Петербург

2016

Оглавление

1. Постановка задачи..........................................................................3

2. Формализация задачи......................................................................3

3. Алгоритм........................................................................................3

4. Листинг..........................................................................................6

5. Тестовые примеры..........................................................................7

6. Источники......................................................................................8

1. Постановка задачи

Реализовать умножение матриц, используя SIMD векторные инструкции ассемблера x86

2. Формализация задачи

Рассматривая задачу в контексте операций над матрицами в трехмерном пространстве, примем размерность матриц 4х4.

Так как в трехмерном пространстве необходимы вычисления над комплексными числами, матрицы вещественные. [1]

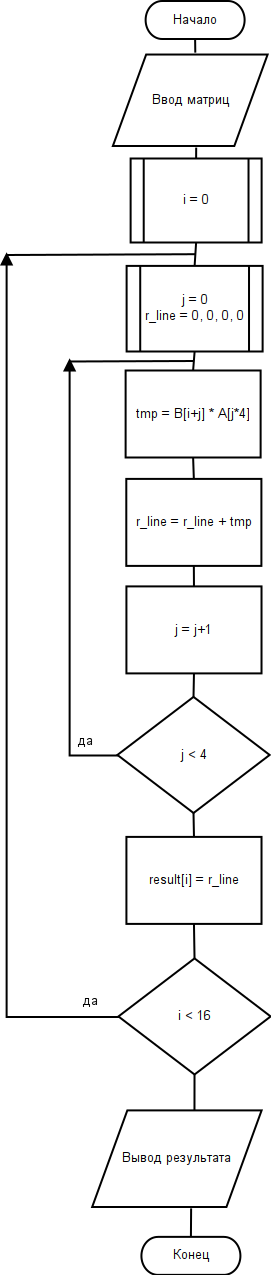
Результат умножения двух матриц размером можно разложить на получение 4х результатов умножения строк-векторов исходной матрицы.

Строка итоговой матрицы опеределена формулой r\_line = E(B[i+j] \* A[j\*4]) [2]

Где i является индексом цикла от 0 до 16, а j от 0 до 4.

Задание необходимо реализовать на языке ассемблера x86 с использованием векторных расширений SSE. [3]

3. Алгоритм [4]



Cхема алгоритма

4. Листинг

**void**

multiply\_sse\_asm(**const** **float**\* A, **const** **float**\* B, **float**\* result) {

/\* Registers map

**\*** **rcx:** **i-loop** **counter**

**\*** **rax:** **j-loop** **counter**

**\*** **r8:** **A-matrix** **current** **row** **address**

**\*** **r9:** **B-matrix** **current** **scalar** **address**

**\*** **xmm0:** **Accumulator** **line**

**\*** **xmm1:** **Result** **line**

**\*/**

**asm** **\_\_volatile\_\_** (

" **xor rcx, rcx;**" // set i-loop counter to 0

" *ILoop*:" // push i-loop counter

" **xor rax, rax;**" // set j-loop counter to 0

" **pxor xmm1, xmm1;**" // set result line to 0

" **mov r8, %0;**" // copy A addr to r8

" **lea r9, [%1 + 4 \* rcx];**" // copy B addr to r9

" *JLoop*:"

// Load B[i+j] to XMM0 (scalar)

" **movss xmm0, [r9];**"

" **add r9, 4;**" // B addr + 1 (i+j)

" **shufps xmm0, xmm0, 0;**"

// Multiply XMM0 by A[4\*j], write to XMM0

" **mulps xmm0, [r8];**"

" **add r8, 16;**" // A addr + 16 (j\*4)

// Add XMM0 to XMM1, write to XMM1

"  **addps xmm1, xmm0;**"

" **inc rax;**" // increment j-loop counter

" **cmp rax, 4;**" // jump to JLoop if rcx != 4

" **jne *JLoop*;**"

// Save XMM3 to result

" **movaps [%2 + 4\*rcx], xmm1;**"

" **add rcx, 4;**" // increment j-loop counter by 4

" **cmp rcx, 16;**" // jump to ILoop if rcx != 16

" **jne *ILoop*;**"

:

: "r" (A), "r" (B), "r" (result)

: "rcx", "rax", "r8", "r9", "xmm0", "xmm1"

);

}

**void**

multiply\_naive(**const** **float**\* A, **const** **float**\* B, **float**\* result) {

**for** (**int** i = 0; i < 16; i += 4) {

**for** (**int** j = 0; j < 4; j++) {

result[i+j] = B[i] \* A[j] +

B[i+1] \* A[j+4] +

B[i+2] \* A[j+8] +

B[i+3] \* A[j+12];

}

}

}

**void**

multiply\_sse(**const** **float**\* A, **const** **float**\* B, **float**\* result) {

**const** **float**\* a = A;

**const** **float**\* b = B;

**float**\* r = result;

\_\_m128 a\_line, b\_line, r\_line;

**for**(int i = 0; i < 16; i += 4) {

memset(&r\_line, '\0', **sizeof**(\_\_m128));

**for**(int j = 0; j < 4; j++) {

a\_line = \_mm\_load\_ps(&a[j\*4]);

b\_line = \_mm\_set1\_ps(b[i+j]);

r\_line = \_mm\_add\_ps(

\_mm\_mul\_ps(a\_line, b\_line),

r\_line

);

}

\_mm\_store\_ps(&r[i], r\_line);

}

}

5. Тестовые примеры

**Матрица A**

**1.000000 2.000000 3.000000 4.000000**

**2.000000 3.000000 4.000000 5.000000**

**3.000000 4.000000 5.000000 6.000000**

**4.000000 5.000000 6.000000 7.000000**

**Матрица B**

**1.000000 2.000000 3.000000 4.000000**

**2.000000 3.000000 4.000000 5.000000**

**3.000000 4.000000 5.000000 6.000000**

**4.000000 5.000000 6.000000 7.000000**

Результат умножения:

90.000000 100.000000 110.000000 120.000000

100.000000 110.000000 120.000000 202.000000

110.000000 120.000000 202.000000 228.000000

120.000000 202.000000 228.000000 254.000000

Замеры времени вычисления (10000000 итераций):

Оптимальная реализация умножения на Си: 323 мс

Реализация умножения на sse-интристиках: 198 мс [5]

**Реализация умножения на ассемблере x86: 199 мс**

Вывод:

В ходе исследования, проведенного в рамках практической работы, установлено, что реализация на ассемблере на 50% быстрее оптимальной реализации на Си.

Тем не менее, код, генерируемый компилятором из интристик не уступает по эффективности вручную написанному ассемблерному коду, требуя меньших затрат на отладку.

Из данной практической работы я вывел что использование SIMD инструкций может дыть существенный прирост производительности при незначительных затратах времени на отладку кода при использовании специализированных расширений компилятора.

6. Источники

1. Матрицы трансформации: <https://en.wikipedia.org/wiki/Transformation_matrix>
2. Лекции по линейной алгебре: <http://kpfu.ru/docs/F974037543/A_G_Ne_.pdf>
3. x86 Assembly: <https://en.wikibooks.org/wiki/X86_Assembly>
4. ГОСТ 19.701-90
5. http://fhtr.blogspot.ru/2010/02/4x4-float-matrix-multiplication-using.html