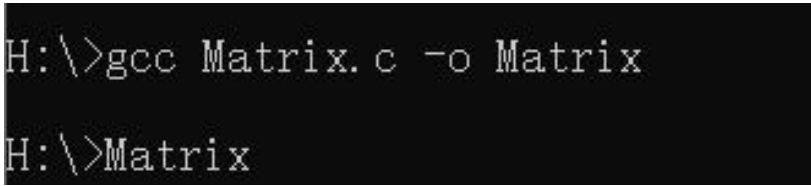


山东大学 计算机科学与技术 学院

汇编语言 课程实验报告

学号：202120130276	姓名：王云强	班级：21.2 班
实验题目：实验十三：矩阵乘法		
实验学时：4	实验日期：2023.12.26 12.29	
实验目的：掌握汇编向量化优化方法。		
实验环境：Windows10、VSCode、DOSBox-0.74、Masm64		
源程序清单： 1. Matrix.c（实验 13 源程序）		
编译及运行结果： 编译结果：  运行结果： 示例一运行结果（无误）：		

```
H:\>Matrix
Matrix 1 finish
Matrix 2 finish
Matrix 3 finish
Matrix 4 finish
Matrix 5 finish
Matrix 6 finish
Matrix 7 finish
Matrix 8 finish
Matrix 9 finish
Matrix 10 finish
Matrix 11 finish
Matrix 12 finish
Matrix 13 finish
Matrix 14 finish
Matrix 15 finish
Matrix 16 finish
naive - 279406353427184 - 102611416755 cycles
expert - 279406353427184 - 15673943422 cycles
sse - 279406353427184 - 9602729602 cycles
avx-auto - 279406353427184 - 3641701501 cycles
avx-manual - 279406353427184 - 3903720766 cycles
```

示例二运行结果（无误）：

```
H:\>Matrix
Matrix 1 finish
Matrix 2 finish
Matrix 3 finish
Matrix 4 finish
Matrix 5 finish
Matrix 6 finish
Matrix 7 finish
Matrix 8 finish
Matrix 9 finish
Matrix 10 finish
Matrix 11 finish
Matrix 12 finish
Matrix 13 finish
Matrix 14 finish
Matrix 15 finish
Matrix 16 finish
naive - 281481212671331 - 103336904287 cycles
expert - 281481212671331 - 15759536386 cycles
sse - 281481212671331 - 9649482751 cycles
avx-auto - 281481212671331 - 3674858848 cycles
avx-manual - 281481212671331 - 3915419454 cycles
```

示例三运行结果（无误）：

```
H:\>Matrix
Matrix 1 finish
Matrix 2 finish
Matrix 3 finish
Matrix 4 finish
Matrix 5 finish
Matrix 6 finish
Matrix 7 finish
Matrix 8 finish
Matrix 9 finish
Matrix 10 finish
Matrix 11 finish
Matrix 12 finish
Matrix 13 finish
Matrix 14 finish
Matrix 15 finish
Matrix 16 finish
naive - 279340940616409 - 103601243331 cycles
expert - 279340940616409 - 15622853569 cycles
sse - 279340940616409 - 9607608479 cycles
avx-auto - 279340940616409 - 3629993823 cycles
avx-manual - 279340940616409 - 3904678495 cycles
```

问题及收获：

一、本实验与 vector.c 实验相比，需要做哪些更改，哪些地方可以进行参考？

首先是对于 COMPUTE_KERNEL 函数来说，是肯定需要更改的，在 vector.c 中只是做了简单的矩阵与常数进行乘法和加法的运算，在这里我们需要改为两个矩阵相乘。而 rdtsc 函数是用来计算函数运行时间的，所以不需要更改。checksum 是需要进行修改，改为计算整个矩阵中所有元素相加的和。而手动 AVX 优化函数需要重新进行写，可以参考之前的框架。除此之外，由于实验是从文件中读入的数据，所以需要写几个读入函数来读入文件中的数据，在读的同时也将 B 矩阵进行转置。实现如下：

COMPUTE_KERNEL 函数实现：

```
#define COMPUTE_KERNEL() \
do \
{ \
    for(int i = 0; i < 1024; ++i)\
    {\
        for(int j = 0; j < 1024; ++j)\
        {\
            for(int k = 0; k < 1024; ++k)\
            {\
                dest[i][j] += src0[i][k] * src1[j][k];\
            }\
        }\
    }\
}\
while (0)
```

readB 函数实现：（该函数的作用是读入 B 矩阵同时转置）

```

void readB()
{
    char path[100] = "H:\\data1\\B.txt";
    freopen(path,"r",stdin);
    for(int i = 0; i < 1024; ++i)
    {
        for(int j = 0; j < 1024; ++j)
        {
            scanf("%d",&src1[j][i]);
        }
    }
}

```

readA 函数实现：（读入相应的 A 矩阵）

```

void readA(int num)
{
    if(num < 10)
    {
        char txt[6] = "1.txt";
        char path1[100] = "H:\\data1\\A";
        txt[0] = num;
        txt[0] += 48;
        strncat(path1,txt,5);
        freopen(path1,"r",stdin);
        for(int i=0;i<1024;++i)
        {
            for(int j=0;j<1024;++j)
            {
                scanf("%d",&src0[i][j]);
            }
        }
    }
    else
    {
        char txt1[7] = "11.txt";
        char path1[100] = "H:\\data1\\A";
        txt1[1] = num-10;
        txt1[1] += 48;
        strncat(path1,txt1,6);
        freopen(path1,"r",stdin);
        for(int i = 0; i < 1024; ++i)
        {
            for(int j = 0; j < 1024; ++j)
            {
                scanf("%d",&src0[i][j]);
            }
        }
    }
}

```

checksum 函数实现：

```
uint64_t checksum(void)
{
    uint64_t final = 0;
    for(int i=0;i<1024;++i)
    {
        for(int j=0;j<1024;++j)
        {
            final += dest[i][j];
        }
    }
    return final;
}
```

二、在 GCC 内嵌汇编中，指令与之前写的汇编语言有哪些不同，寄存器又有哪些不同？

对于基础指令（如 mul, add）的参数列表与之前写的汇编语言基本相同，不同之处在于 MASM 的目的操作数一般在前面，而在内嵌汇编中，目的操作数一般放在后面，如 ADD CX, 20H 是在汇编中的写法，在这里需要写成 ADD \$0x20, %%RCX。同样常数需要带\$符合进行标注，寄存器需要加%%符号。除此之外，在之前 8086 的汇编中，一般常用寄存器有 AX、BX、CX、DX、SI、DI，其中前四个也可以分高位低位使用（如 AL 是 8 位），其他的很少用。而在内嵌汇编中，对常用寄存器有了更详细的划分且更多（如下图）

64位	32位
rax	eax
rbx	ebx
rcx	ecx
rdx	edx
rsi	esi
rdi	edi
rbp	ebp
rsp	esp
r8-r15	r8d-r15d

三、手写 AVX 优化，整体思路是怎样的？

整体思路就是利用两个指针分别指向 A 矩阵（src0）和 B 矩阵（src1）进行遍历，根据指针的移动来取指针对应的一组向量进行运算，将结果放入 C 矩阵（dest）的对应位置。（B 矩阵提前进行了转置）每次用 A 矩阵指针指向的该行与 B 矩阵的每一行分别进行乘法和加法（即进行矩阵运算），直到 A 矩阵的该行与 B 矩阵的每一行都运算结束，说明在结果矩阵中该行的答案算完了，之后再遍历 A 矩阵的后面每一行，B 矩阵从头开始遍历，依次进行相乘，直到完成了矩阵的整个乘法运算。主要思路难点在于，如何将原本的矩阵与矩阵之间的运算、操作，转换成行与行之间的运算、操作。

四、整体代码的构成思路是怎样的？

首先我们读入 B 矩阵（在读入过程中完成了对 B 矩阵的转置，方便手写 AVX 的优化），之后进行 16 次循环，每次读入一个 A 矩阵，分别进行 5 中不同优化的运算，每次将结果和时间记录在对应的数组中，直到读完每一个 A 矩阵，完成最终的运算进行输出。（由于此处在读入时已经进行了转置，所以 `computer_kernel` 函数是对 B 的转置矩阵进行相乘的。读入矩阵的函数主要是字符串操作）

五、为什么 AVX 手写优化的最终复杂度要高于 AVX 自动优化？

本实验中一开始是想仿照 `vector.c` 一样每次处理 4 组矩阵，但是发现好多 BUG，无法实现。所以在代码中做了简单处理，一组一组的处理。同时，根据课堂上的讲解，内嵌汇编主要是用在当主要的操作是按顺序进行处理数据时，且没有复杂的嵌套和比较，会有比较好的提升与体现。但是在手动优化中，每次处理的数据不是大批量的，而且需要不断的进行判断，也有内部循环等，导致整体的性能反而不如自动 AVX 优化那么好。

收获：

一、通过本实验对于内嵌汇编有了更深的认识，如使用寄存器需要注意的情况，指令规范格式等。

二、通过自己动手实现内联汇编，对于 GCC 有了更多的体会和理解。

三、对于向量化操作有了更深的理解，明白了其内涵与优化原理。