实验1

班级：191211

姓名：孙鹤轩

学号：20211000156

实验内容：

1 课程实验平台环境的安装，基本实验工具的使用；

2 从高级语言的角度展示和解释位运算、浮点数运算的精度、cache对程序性能的影响。

实验目标：

1 完成课程实验平台环境的搭建与设置；掌握常用实验工具的基本使用方法；

2 掌握C语言中位操作语句的使用；了解浮点数表示精度在浮点数运算中的影响；了解cache、数据存储与访问模式对程序性能的影响，掌握编写cache友好代码的基本原则。

实验任务：

1 学习MOOC内容

https://www.icourse163.org/learn/NJU-1449521162

第一周 实验与开发环境的安装和使用

第2讲 虚拟机、Linux及其上实验环境的安装

第3讲 基本实验工具的使用

第二周 C语言编程实践

第1讲 数据的位运算操作

第2讲 浮点数的精度问题

第3讲 Cache友好代码

2 在自己的电脑上安装实验环境

安装虚拟机软件：VirtualBox（开源软件）或VMware（商业软件）

安装Linux系统：Linux 32位版本Debian或Ubuntu

（注：Ubuntu 16.04.6及之前版本支持32位）

熟悉软件工具：gcc，gdb，objdump

3 完成作业

**3.1 编写C语言程序，不使用中间变量，交换变量a和b的值，已知变量的初始值为a=2021，b=191，分析程序的反汇编代码，说明算法的基本原理。**

**3.1.1 程序代码和注释说明**

#include <stdio.h>

#include <stdlib.h>

int main()

{

int a = 2021,b=191;//初始化交换变量

printf("before swapping:a=%d,b=%d\n",a,b);//输出交换前的值

a=a^b;

b=b^a;//b=b^(a^b)=a;

a=a^b;//a=(a^b)^a=b;

printf("after swapping:a=%d,b=%d\n",a,b);

return 0;

}

**3.1.2 实验结果记录**

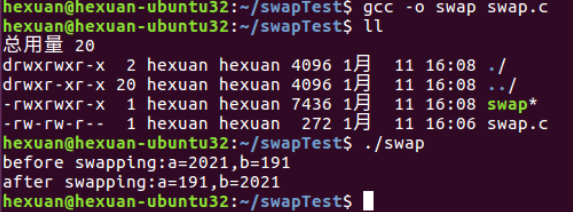


图 1交换实验结果

**3.1.3 结果分析与讨论**

从实验结果看我们成功完成了变量a与变量b的值交换。该算法通过c语言的异或操作完成，利用的是一个值与他本身异或为0的特性来完成；

程序汇编代码如下：

0804848b <main>:

804848b: 8d 4c 24 04 lea 0x4(%esp),%ecx

804848f: 83 e4 f0 and $0xfffffff0,%esp

8048492: ff 71 fc pushl -0x4(%ecx)

8048495: 55 push %ebp

8048496: 89 e5 mov %esp,%ebp

8048498: 51 push %ecx

8048499: 83 ec 14 sub $0x14,%esp

804849c: c7 45 f0 e5 07 00 00 movl $0x7e5,-0x10(%ebp)

80484a3: c7 45 f4 bf 00 00 00 movl $0xbf,-0xc(%ebp)

80484aa: 83 ec 04 sub $0x4,%esp

80484ad: ff 75 f4 pushl -0xc(%ebp)

80484b0: ff 75 f0 pushl -0x10(%ebp)

80484b3: 68 9c 85 04 08 push $0x804859c

80484b8: e8 93 fe ff ff call 8048350 <printf@plt>

80484bd: 83 c4 10 add $0x10,%esp

80484c0: 8b 45 f4 mov -0xc(%ebp),%eax

80484c3: 31 45 f0 xor %eax,-0x10(%ebp)

80484c6: 8b 45 f0 mov -0x10(%ebp),%eax

80484c9: 31 45 f4 xor %eax,-0xc(%ebp)

80484cc: 8b 45 f4 mov -0xc(%ebp),%eax

80484cf: 31 45 f0 xor %eax,-0x10(%ebp)

80484d2: 83 ec 04 sub $0x4,%esp

80484d5: ff 75 f4 pushl -0xc(%ebp)

80484d8: ff 75 f0 pushl -0x10(%ebp)

80484db: 68 b7 85 04 08 push $0x80485b7

80484e0: e8 6b fe ff ff call 8048350 <printf@plt>

80484e5: 83 c4 10 add $0x10,%esp

80484e8: b8 00 00 00 00 mov $0x0,%eax

80484ed: 8b 4d fc mov -0x4(%ebp),%ecx

80484f0: c9 leave

80484f1: 8d 61 fc lea -0x4(%ecx),%esp

80484f4: c3 ret

80484f5: 66 90 xchg %ax,%ax

80484f7: 66 90 xchg %ax,%ax

80484f9: 66 90 xchg %ax,%ax

80484fb: 66 90 xchg %ax,%ax

80484fd: 66 90 xchg %ax,%ax

80484ff: 90 nop

从汇编代码中我们可以确定变量a与变量b的存储地址为-0x10(%ebp)和-0xc(%ebp)

，我们重点关注两条调用printf函数之间的交换逻辑即：  


首先将b的值存储到eax中，然后将a与b进行异或操作，将结果保存到了a变量的地址处，则-0x10(%ebp)的内容现在是a^b，然后将a^b的值移动到eax中，对eax中的值与-0xc(%ebp)的值，即执行a^b^b，将结果a存储到-0xc(%ebp)即原先存储b的地址处，现在b的值成功变为了a，接着将-0xc(%ebp)处的值a存储到eax中，将eax与-0x10(%ebp)处的值即执行a^a^b，将结果存储到-0x10(%ebp)即原先a变量的存储地址处，则现在a变量的值变为了b，交换完成

**3.1.4 其他**

在这个实验中我们通过c语言的位运算符完成了两个数的交换，没有借助第三个变量，节省了内存消耗。

**3.2 编写C语言程序，举一个例子，说明浮点数运算误差问题，并给出解决方案。**

**3.1.1 程序代码和注释说明**

#include <stdio.h>

int main()

{

float sum=0;//存储使用kahan算法的累加和

float sum1=0;//存储没有使用kahan算法的累加和

float c=0;//累加产生的误差

float y,t;//y为经过误差修正后的加数，t为经过本次累加后的和

int i;

for(i=0;i<4000000;i++)

sum1+=0.1;

for(i=0;i<4000000;i++)

{

y=0.1-c;//计算修正后的加数

t=sum + y;//计算本次累加的和

c=(t-sum)-y;//计算本次累加产生的误差

sum=t;

}

printf("generate sum1=%f\n",sum1);

printf("kahan sum=%f\n",sum);

return 0;

}

**3.1.2 实验结果记录**

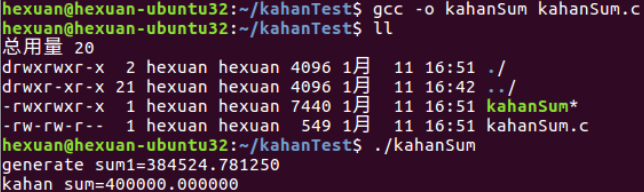


图 2kahan累加算法实验结果

**3.1.3 结果分析与讨论**

从实验结果看使用kahan算法计算4000000次0.1相加的结果是准确的，这得益于我们每次对加数进行修正。

Kahan算法的思想就是在每次累加前计算出累加误差，通过累加误差对加数进行修正，这样可以在很大程度上提高浮点数计算的精度。

**3.1.4 其他**

通过这次实验我们实现了浮点数精准计算算法kahan算法，认识到到浮点数运算误差的问题

**3.3 编写C语言程序，实现两个1024\*1024的浮点数矩阵相乘，采用不同的循环顺序，比较运行效果，并分析导致差异的原因。**

**3.3.1 程序代码和注释说明**

#include <stdio.h>

#include <stdlib.h>

#include <sys/time.h>

#include <time.h>

/\*\*\*执行矩阵相乘的函数\*\*\*/

void multMat1(int n,float \*A,float \*B,float \*C){

int i,j,k;

for(i = 0; i < n;i++)

for(j = 0; j < n;j++)

for(k = 0; k < n;k++)

C[i+j\*n] += A[i+k\*n]\*B[k+j\*n];

}

void multMat2(int n,float \*A,float \*B,float \*C){

int i,j,k;

for(i = 0; i < n;i++)

for(k = 0; k < n;k++)

for(j = 0; j < n;j++)

C[i+j\*n] += A[i+k\*n]\*B[k+j\*n];

}

void multMat3(int n,float \*A,float \*B,float \*C){

int i,j,k;

for(j = 0; j < n;j++)

for(i = 0; i < n;i++)

for(k = 0; k < n;k++)

C[i+j\*n] += A[i+k\*n]\*B[k+j\*n];

}

void multMat4(int n,float \*A,float \*B,float \*C){

int i,j,k;

for(j = 0; j < n;j++)

for(k = 0; k < n;k++)

for(i = 0; i < n;i++)

C[i+j\*n] += A[i+k\*n]\*B[k+j\*n];

}

void multMat5(int n,float \*A,float \*B,float \*C){

int i,j,k;

for(k = 0; k < n;k++)

for(i = 0; i < n;i++)

for(j = 0; j < n;j++)

C[i+j\*n] += A[i+k\*n]\*B[k+j\*n];

}

void multMat6(int n,float \*A,float \*B,float \*C){

int i,j,k;

for(k = 0; k < n;k++)

for(j = 0; j < n;j++)

for(i = 0; i < n;i++)

C[i+j\*n] += A[i+k\*n]\*B[k+j\*n];

}

int main()

{

int nmax=1024,i,n;

//创建函数指针数组

void (\*orderings[]) (int,float \*,float \*,float \*)

={&multMat1,&multMat2,&multMat3,

&multMat4,&multMat5,&multMat6};

//创建6种遍历方式

char \*names[] = {"ijk","ikj","jik","jki","kij","kji"};

//为三个矩阵申请存储空间

float \*A=(float \*)malloc(nmax\*nmax\*sizeof(float));

float \*B=(float \*)malloc(nmax\*nmax\*sizeof(float));

float \*C=(float \*)malloc(nmax\*nmax\*sizeof(float));

//声明计算运行时间的变量

struct timeval start,end;

//为三个矩阵随机赋值

for(i=0;i<nmax\*nmax;i++) A[i] = drand48()\*2-1;

for(i=0;i<nmax\*nmax;i++) B[i] = drand48()\*2-1;

for(i=0;i<nmax\*nmax;i++) C[i] = 0;

for(i=0;i<6;i++)

{

gettimeofday(&start,NULL);

(\*orderings[i])(nmax,A,B,C);//执行乘法函数

gettimeofday(&end,NULL);

//计算运行时间

double seconds = (end.tv\_sec-start.tv\_sec)

+1.0e-6\*(end.tv\_usec-start.tv\_usec);

printf("%s:\tn = %d,%.3f s \n",names[i],nmax,seconds);

}

return 0;

}

**3.3.2 实验结果记录**

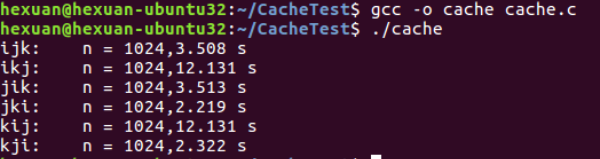


图 3不同遍历方式的矩阵相乘实验结果

**3.3.3 结果分析与讨论**

从实验结果我们可以看出不同遍历顺序时间消耗截然不同，最慢的顺序ikj是最快的顺序jki的6倍，这体现出cache对算法的效率的影响是非常大的。在计算机中，矩阵中的数据是按顺序存储的，访问相邻数据时，应当尽可能地遵从空间局部性和时间局部性。

慢顺序ijk是将B的第j行和A的第i列相乘存储到C的ji处，我们可以发现A的访问是非顺序的，导致其另一列的值不在cache中，所以计算会变慢；最慢的顺序ikj是将B的第k列与A的第ki处的元素相乘存储到C的第i列，我们可以发现矩阵B和矩阵C的访问都是非顺序的，因而其最慢。