实验三：Linux环境下调试与矩阵乘法优化

郑海刚

HITSZ 实验与创新实践教育中心
Education Center of Experiments and Innovations, HITSZ

本讲概述

- 主要内容
  - 调试工具：printf、gdb
  - 段错误（segmentation fault）
  - 浮点数误差
  - gflops
  - 代码框架
  - make&makefile

调试方法：printf

- printf：打印程序的状态
  - 输出与预期不符，可能是更早之前的代码逻辑导致
    - 二分法缩小范围
  - 每行输出注意加换行符\n

调试方法：gdb（The GNU Debugger）

- Install GDB：sudo apt-get install gdb
- 四大功能，不仅能观察，还能修改测试

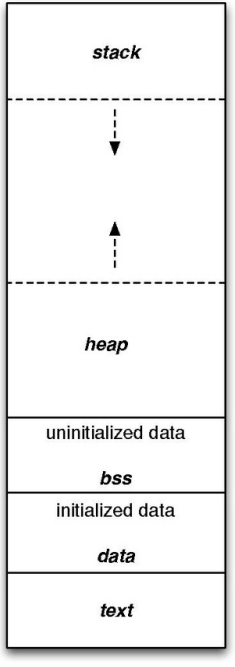
```
GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

•   Start your program, specifying anything that might affect its behavior.
•   Make your program stop on specified conditions.
•   Examine what has happened, when your program has stopped.
•   Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn
```

段错误（Segmentation fault）

- 也叫内存访问违例、非法内存访问
  - 在C语言等提供低级别内存访问接口，但无安全检查的编程中经常出现。

- 通常是以下3种非法访问
  - Attempting to access a nonexistent memory address (outside process's address space)
  - Attempting to access memory the program does not have rights to (such as kernel structure
  - Attempting to write read-only memory (such as code segment)



引起段错误的编程错误

- 通常是以下6种编程错误导致

- Dereferencing a null pointer, which usually points to an address that's not part of the process's address space
- Dereferencing or assigning to an uninitialized pointer (wild pointer, which points to a random memory address)
- Dereferencing or assigning to a freed pointer (dangling pointer, which points to memory that has been freed/deallocated/deleted)
- A buffer overflow
- A stack overflow
- Attempting to execute a program that does not compile correctly. (Some compilers[*which?*] will output an executable file despite the presence of compile-time errors.)

段错误操作系统的处理方式

- Linux下，操作系统发送SIGSEGV 信号，终止进程
- 看到的现象就只有：segmentation fault
- lab3/stackoverflow.c：无限递归，导致栈溢出

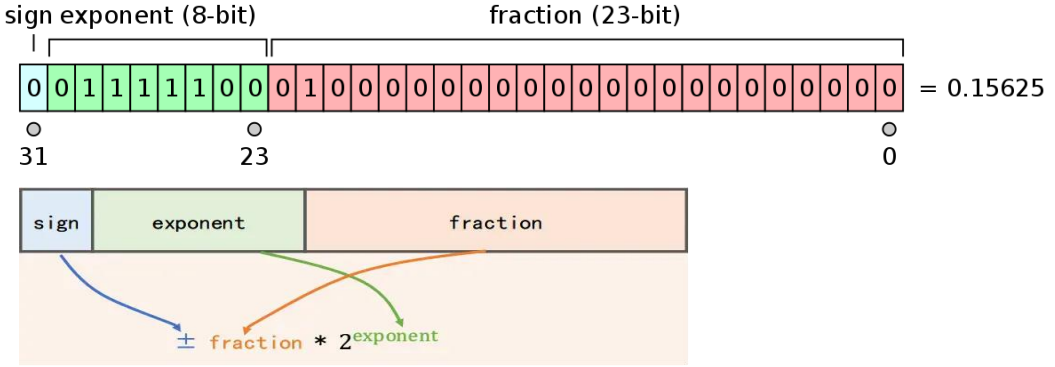
```
$ hpc/lab3 »gcc stackoverflow.c && ./a.out
[1]    14088 segmentation fault  ./a.out
$ hpc/lab3 »cat stackoverflow.c
int main(void)
{
    return main();
}
```

gdb调试段错误

- gdb调试：运行到段错误的地方，会自动停下来并显示出错的行和行号
- lab3/gemm_naive_sgfault.c
  - 编译的时候加 –g选项，生成调试信息
    - gcc -g gemm_naive_sgfault.c
  - 出错的时候backtrace命令可以查看函数调用栈，简写为bt
    - gdb a.out
  - Determine the line of code that causes a segmentation fault?

浮点数格式

- 浮点数格式标准：IEEE 754



- 浮点数在线转换

IEEE 754 Converter (JavaScript), V0.21

	Sign	Exponent	Mantissa
Value:	+1	2^{-3}	1.600000023841858
Encoded as:	0	124	5033165
Binary:	0	01111100	10011001100110011001101

You entered	0.2
Value actually stored in float:	0.20000000298023223876953125
Error due to conversion:	2.98023223876953125E-9
Binary Representation	00111110010011001100110011001101
Hexadecimal Representation	0x3e4ccccd

+1 -1

浮点数误差：代码验证

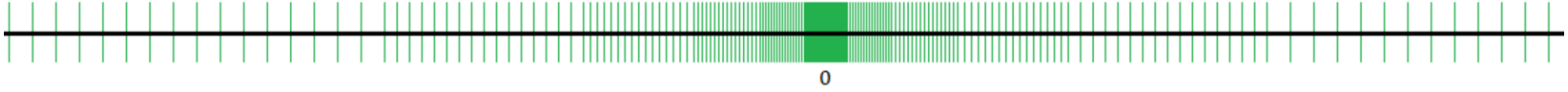
- lab3/float.c

```
$ hpc/lab3 »cat float.c
#include<stdio.h>

int main(){
    float a = 0.2;
    printf("entered: a = 0.2\n");
    printf("a actually is: %.20f\n",a);
}
$ hpc/lab3 »gcc float.c && ./a.out
entered: a = 0.2
a actually is: 0.20000000298023223877
```

浮点数密度分布：

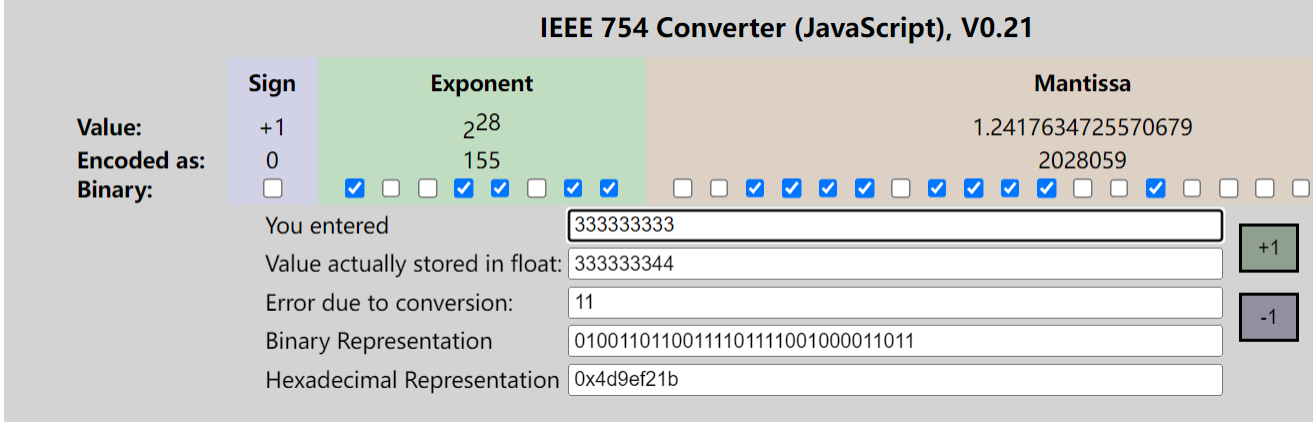
- 浮点数密度分布图
  - 靠近0的位置很密集，精度高
  - 绝对值越大的位置密度越小，误差越大



- 表达范围和精度的权衡
- 很多数学库都会频繁做归一化（-1，1）

浮点数误差：大数

- 越大的数字，距离下一个实数的距离就越大
  - 333333333、333333339等实际指都是333333344
  - 可能会带来相当的绝对误差



浮点数加法：$a+ b + c \neq a + (b + c)$

- 计算 $1+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n}$
- lab3/float_sum.c
  - 顺序不同结果不同
  - 精度不同，结果不同

```
$ hpc/lab3 »cat float_sum.c
#include<stdio.h>
// T is type, st is start, ed is end, d is distance
// 1 + 1/2 + 1/3 + ... + 1/n
#define SUM(T, st, ed, d) ({ \
    T s = 0; \
    for (int i = st; i != ed + d; i += d) \
        s += (T)1 / i; \
    s; \
})

#define n 100000

int main(){
    printf("%.16f\n", SUM(float, 1, n, 1));
    printf("%.16f\n", SUM(float, n, 1, -1));
    printf("%.16f\n", SUM(double, 1, n, 1));
    printf("%.16f\n", SUM(double, n, 1, -1));
}%
$ hpc/lab3 »gcc float_sum.c && ./a.out
12.0908508300781250
12.0901527404785156
12.0901461298633350
12.0901461298634079
```

误差的原因

- 误差的来源：用有限位宽表示无限的有理数
- a == b 需谨慎判断

```
fabs( C( i, j ) - C_ref( i, j ) ) > TOLERANCE
```

Lab2要求测量的数据

- duration是运行时间，gflops是发挥出的性能，有flosp，mflops，gflops
- naive运行的时间远多于openblas，即性能远低于openblas
- 工程上多用flops衡量代码实现的性能差距而不用运行的时间
  - 不同规模的应用运行时间差异本身就很大，需要一个统一的对比标准
  - 服务器有一个峰值性能数据，由CPU、GPU等计算芯片决定
  - 通过与峰值性能对比，衡量软件发挥硬件性能的水平

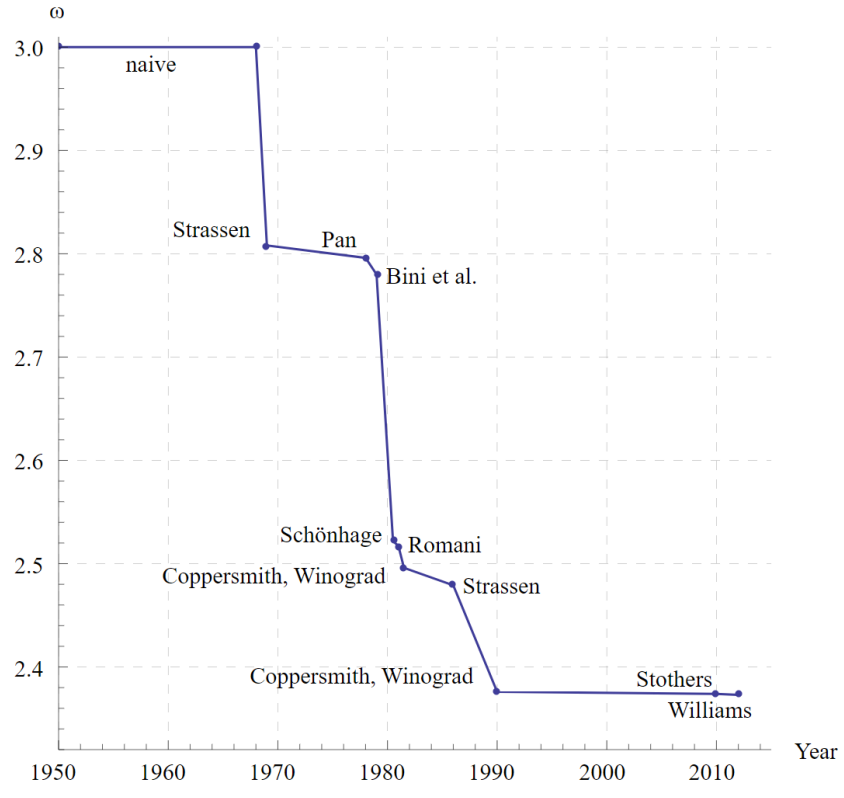
FLOPS（floating point operations per second）

- 常用于科学计算领域，浮点操作比较多
- 每秒的浮点操作数，非指令数，不区分加减乘除
  - 一条指令可以有多个浮点操作，用指令数衡量不准确
  - 向量化指令（SIMD）一次可以执行多个数的操作，比如x86的avx256
  - SIMD的FMA（融合乘加）： c=a*b+c
  - CPU的理论性能如下

$$\mathbf{FLOPS} = \mathbf{cores} \times \frac{\mathbf{cycles}}{\mathbf{second}} \times \frac{\mathbf{FLOPs}}{\mathbf{cycle}}.$$

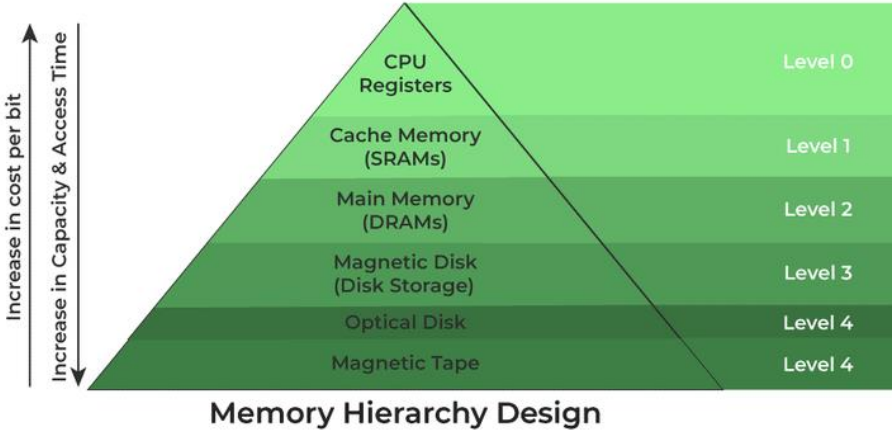
GEMM优化

- 复杂度优化
  - 最低O（n^2.371）
- 工程上是并行优化、访存优化等



计算机存储结构

- Memory Hierarchy Design
  - cache比内存快一个数量级
  - 查看寄存器：
    - gdb ls；starti; info register;
  - 查看cache：lscpu
  - 查看内存大小：free -m



Memory Location	CPU Cycles
Register	1
L1 Cache	~4
L2 Cache	~14
L3 Cache	~75
Main Memory	~200

GEMM访存次数：C=A*B+C

- 计算次数很难优化，通常优化访存，计算次数：
  - 三层循环M*N*K，每次内循环执行的浮点操作数2次，总次数2*M*N*K
- 访存次数：从内存中每读/写一个元素算一次
  - 不考虑任何缓存：2n^3+2*n^2（A、B每个元素读n次，C读写一次）
  - 全部缓存：4* n^2（A、B、C每个元素只读一次，C写一次）

矩阵乘法代码框架：lab3/how-to-optimize-gemm

- 源自开源的how-to-optimize-gemm
  - 遍历不同的矩阵规模，得到性能数据，结果保存到文件
    - 编译运行：make run
    - 画出性能数据图：python plotFlops.py
  - 随机初始化，每个元素的取值范围是(-1,1)

```
./
├── defs.h
├── makefile
├── MMult0.c
├── MMult1.c
├── plotAll.py
├── plotFlops.py
├── REF_MMult.c
├── test_MMult.c
├── test_python_dgemm.py
└── util.c
```

make&makefile

- 模块化编程
  - 大项目会拆成多个头文件，多个c文件

- 开发过程中频繁编译、运行、调试?
  - 每次gcc xx 跟一堆文件?
  - 方便、高效的实现
    - make构建工具：自动确定哪个文件重新编译，并非每次都全量编译
    - makefile告诉make怎么编译和链接

makefile语法

- 由一系列规则组成，每条规则包含3部分
  - 目标：文件名，空格分隔
  - 依赖（可选）：文件名，空格分隔，在命令执行之前要存在
  - 命令（可选）：生成跟目标同名的文件的命令，必须是tab开头

- 变量、内置变量、隐式规则等查阅手册
  - makefiletutorial
  - 跟我一起写makefile

```
targets: prerequisites
  command
  command
  command
```

框架代码的makefile

- 结合make run ” 输出日志” 阅读
- 先找到目标：run
- %.o : %.c
  - 模式规则

```
1  OLD  := MMult0
2  NEW  := MMult0
3
4
5  CC         := gcc
6  LINKER     := $(CC)
7
8  CFLAGS     := -Wall -Werror -Wno-unused-result -Wno-unused-value -Wno-unused-va
9  LDFLAGS    := -lm
10
11 DATA_DIR = _data
12 BUILD_DIR = _build
13 OBJS  := $(BUILD_DIR)/util.o $(BUILD_DIR)/REF_MMult.o $(BUILD_DIR)/test_MMult.o
14
15 $(shell mkdir -p $(BUILD_DIR) $(DATA_DIR))
16
17 $(BUILD_DIR)/%.o: %.c
18     @mkdir -p $(dir $@) && echo + CC $<
19     $(CC) -std=gnu11 $(CFLAGS) -c $< -o $@
20
21
22 all:
23     make clean;
24     make $(BUILD_DIR)/test_MMult.x
25
26 $(BUILD_DIR)/test_MMult.x: $(OBJS) defs.h
27 _   $(LINKER) $(OBJS) $(LDFLAGS) -o $@
```

makefile

- 描述了文件的依赖关系：一个有向无环图
- 目标更新的两种情况：
  - 目标不存在
  - 目标存在，依赖有更新（依赖文件的时间戳比目标文件新）
- 只有修改的文件及依赖该文件的文件才会重新编译，减少编译时间

python实现矩阵乘法

- lab3/test_python_dgemm.py
  - 运行：python test_python_dgemm.py
  - naive实现确实比C的naive版本慢
  - python也可以做到很快，numpy库

课后阅读

- 课后阅读《操作系统导论》前5章