计算机科学与技术学院

《计算机系统结构》课程实验

学　　号：XXXXXXXXXX

班　　级：XXXXXXXX

专　　业：计算机科学与技术

学生姓名：XXX

20xx 年 x 月 x 日

实　验　报　告 1

学生姓名：	学　号：	时间： 地点：

实验课程名称：计算机系统结构

一、实验名称：流水线中的相关—判断一个四位数是否是回文数

二、实验原理：

1、WinDLX 平台与流水线

WinDLX 模拟器是一个图形化、交互式的 DLX 流水线模拟器，它采取伪汇编形式编码，模拟流水线的工作方式，能够演示 DLX 流水线是如何工作的。流水线的指令执行分为 5 个阶段：取指、译码、执行、访存、写回。

WinDLX 模拟器还提供了对流水线操作的统计功能，便于对流水线进行性能分析。

2、流水线中的相关及解决办法

（1）结构相关：当某一条机器指令需要访问物理器件时，该器件可能正在被占用，例如连续的两条加法指令都需要用到浮点加法器，就产生结构相关，可以通过增加加法器的方式解决结构相关；

（2）数据相关：当某一条指令需要访问某个寄存器时，此时这个寄存器正被另一条指令所使用，从而产生数据相关，可以通过重定向技术解决数据相关；

（3）控制相关：当程序执行到某个循环语句时，顺序执行的下一条语句将被跳继续执行循环体的内容，从而产生控制相关，可以通过循环展开解决控制相关。

三、实验目的：

1、加深对流水线理论知识的理解；

2、掌握对流水线性能分析的方法，了解影响流水线效率的因素；

3、熟悉在 WinDLX 体系结构下的汇编代码编写和优化；

4、了解相关的类型及各类相关的解决办法；

5、培养运用所学知识解决实际问题的能力。

四、实验内容：

1、根据 WinDLX 模拟器伪汇编指令规则编写判断一个四位数是否是回文数的程序 p.s 和 input.s；

2、分别按照不同顺序将 p.s 和 input.s 装入主存，分析输入顺序不同对运行结果产生的影响；

3、观察程序中出现的数据、控制、结构相关，指出程序中出现上述现象的指令组合，并提出解决相关的办法；

4、分别考察各类解决的相关办法，分析解决相关后性能的变化。

注意： 除解决结构相关，其他情况下加、乘、除运算器都只有一个。

本问题中所有浮点延迟部件设置为：加法：2 个延迟周期；乘法：5 个延迟周期；除法：19 个延迟周期。

五、实验器材（设备、元器件）：

电脑一台
VMware Workstation
虚拟机（Windowsxp 32 位操作系统）
WinDLX 模拟器

六、实验步骤及操作：

1、初始化 WinDLX 模拟器

（1）为 WinDLX 创建目录，C:\WinDLX。将 WinDLX 和 p.s、ps.s、input.s 放在这个目录中。

（2）初始化 WinDLX 模拟器：点击 File 菜单中的 Reset all 菜单项，弹出一个“Reset DLX”对话框，点击窗口中的“确认”按钮即可。如图 1 所示。

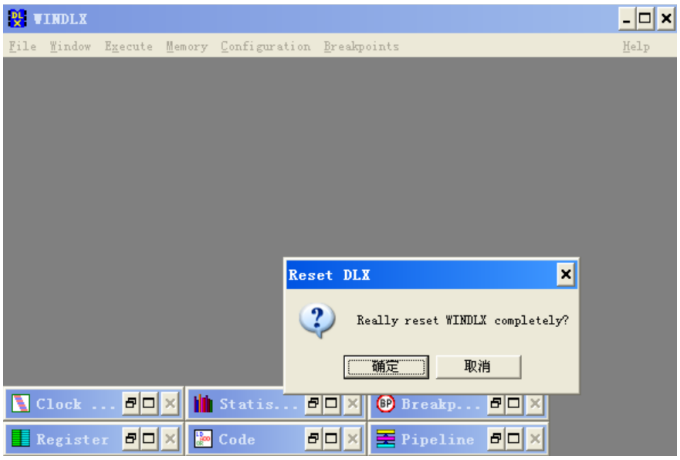


图 1 初始化模拟器界面

2、将程序装入 WinDLX 平台

点击 File 菜单中的 Load Code or Data 项，依次双击 p.s 和 input.s。点击 load，将两个程序装入。如图 2 所示。

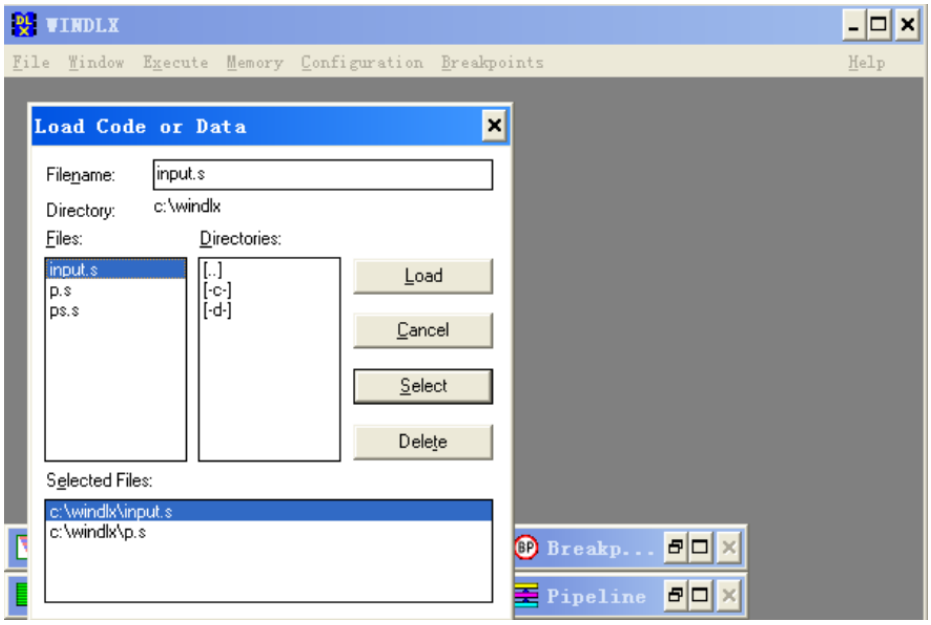


图 2 程序装入界面

3、运行程序并观察

进行单步调试，在 WinDLX 模拟器的 6 个子窗口观察程序的执行情况。观察程序运行的总时钟周期，产生的相关种类以及每种相关的数量。

4、解决数据相关

勾选 Enable Forwading，采用重定向技术添加专用数据通路减少数据相关，观察数据相关的数量变化。

5、解决结构相关

将 Multiplication Units 的数目由 1 到 2，观察结构相关的数量变化。

6、解决控制相关

将 p.s 的循环体展开形成新文件 ps.s，采用循环展开的方法减少控制相关，观察控制相关的数量变化。

七、实验数据及结果分析：

1、程序装入顺序对运行结果的影响

先装入 p.s 再装入 input.s 时，程序能够正确执行；当先装入 input.s 再装入 n 原.s 时，因为 input.s 的地址高，而程序顺序执行到 input.s 时无法正确的输出，不会出现结果。

2、主要代码及说明

```
addi r1,r0,Prompt          ;将 Prompt 字符串首地址放入 r1 寄存器中
jal     InputUnsigned      ;调用 input 子函数读取一个四位数
add     r2,r0,r1           ;将 input 函数读取的数放入寄存器中
add     r6,r0,r1
addi    r3,r0,0                ;r3 寄存器中数清 0
addi    r7,r0,0                ;r7 寄存器中数清 0
addi    r8,r0,10               ;立即数 10 写入 r8

;求 r6 中的四位数逆序对应的新四位数，存在 r7 中，如：1234->4321
Loop:
        ;循环内  使 r6=原 r6/10,r7=原 r7*10+原 r6%10
        ;r6 中所存数为 0 则跳转向 check 所标识的指令地址
        beqz    r6,check
        ;使 r7=原 r7*10,r9=原 r6/10,r10=原 r6%10
        divu r9,r6,r8   ;r6 寄存器中的数除以 r8 寄存器中的数放到 r9 寄存器中
        mult r10,r9,r8      ;r9 寄存器中的数乘以 r8 寄存器中的数放到 r10 中
        mult r7,r7,r8   ;r7 寄存器中的数乘以 r8 寄存器中的数放到 r7 中
        sub     r10,r2,r10      ;将 r2 和 r10 的差送入 r10
        ;使 r6=原 r6/10,r7=原 r7*10+原 r6%10
        add     r6,r0,r9
        add     r7,r7,r10
        j       Loop

;判断是不是回文数，即 r2 和 r7 是否相等
check:
        sub  r2,r2,r7   ;将 r2 和 r7 的差送入 r2
        beqz    r2,output1      ;r2 中所存数为 0 则跳转向 output1 所标识的指令地址
        j       output2

;输出结果
output1:
        addi r1,r0,PrintfFormat1
        sw      PrintfPar,r1
        addi r14,r0,PrintfPar
        trap 5              ;调用中断，格式化为标准输出
        j       over
output2:
```

```
        addi r1,r0,PrintfFormat2
        sw      PrintfPar,r1
        addi r14,r0,PrintfPar
        trap 5              ;调用中断，格式化为标准输出
        j                   over

over:
        trap   0        ;调用系统中断，0 表示程序执行结束
```

3、程序分析及运行结果

（1）根据提示输入一个四位数。

测试用例 1：1221

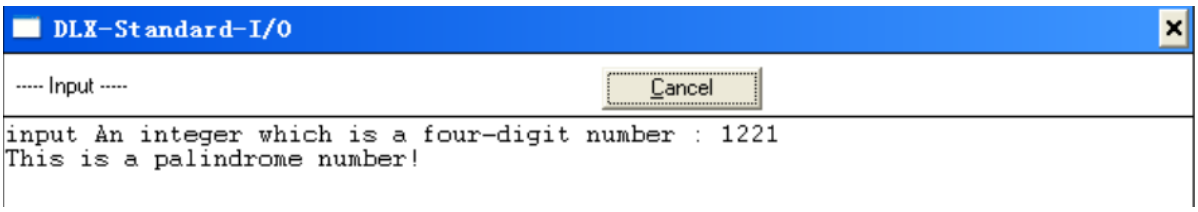


图 3 运行结果截图 1

测试用例 2：1234

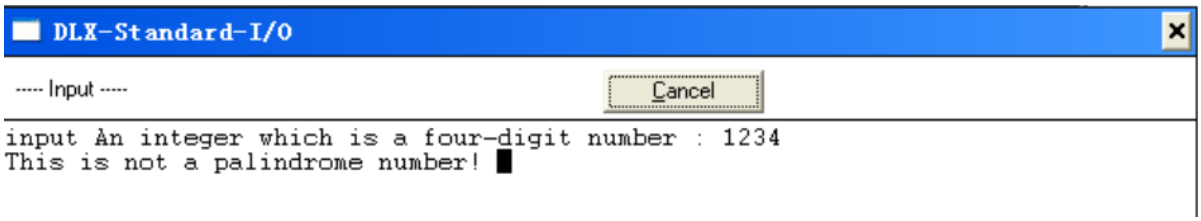


图 4 运行结果截图 2

由图 3、图 4，1221 是回文数，1234 不是回文数，计算结果正确。在下文使用测试样例 1 来分析各种相关。

（2）点击 Statistics 窗口，查看程序执行的时钟周期以及数据相关、结构相关、控制相关的发生次数。

程序执行共用 288 个时钟周期，数据相关发生 161 次，结构相关发生 16 次，控制相关发生 14 次，如图 5 所示。

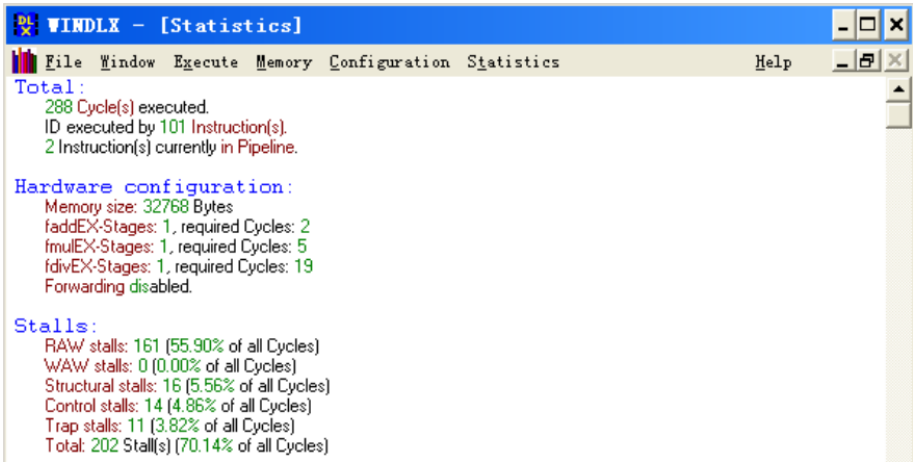


图 5 时钟周期和相关数据截图

4、数据相关及解决

（1）数据相关产生的原因

```
sub     r2,r2,r7   ;将 r2 和 r7 的差送入 r2
beqz    r2,output1     ;r2 中所存数为 0 则跳转向 output1 所标识的指令地址
```

beqz 指令要使用 r2 寄存器的数据，但是上一条指令刚刚执行完数据还没有更新，产生数据相关，如图 6 所示。

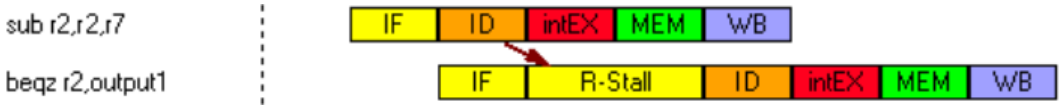


图 6 数据相关截图

（2）数据相关的解决

采用重定向技术，勾选 Configuration 的 Enable Forwading 选项。在第一条指令结束后直接将寄存器 r2 的内容更新，消除数据相关，如图 7 所示。

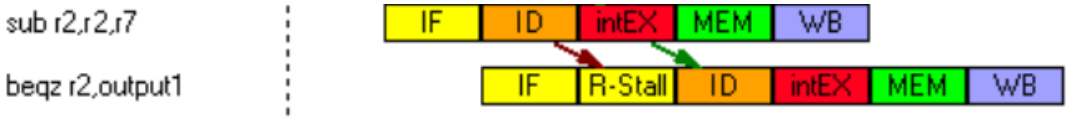


图 7 解决数据相关截图

查看运行结果，数据相关数量降低，数据相关个数为 116，如图 8 所示。

图 8 解决数据相关的数据截图

5、控制相关及解决

（1）控制相关的产生原因

```
Loop:
        ;循环内  使 r6=原 r6/10,r7=原 r7*10+原 r6%10
        ;r6 中所存数为 0 则跳转向 check 所标识的指令地址
        beqz    r6,check
        ;使 r7=原 r7*10,r9=原 r6/10,r10=原 r6%10
        divu r9,r6,r8   ;r6 寄存器中的数除以 r8 寄存器中的数放到 r9 寄存器中
        multr10,r9,r8       ;r9 寄存器中的数乘以 r8 寄存器中的数放到 r10 中
        multr7,r7,r8   ;r7 寄存器中的数乘以 r8 寄存器中的数放到 r7 中
        sub     r10,r2,r10      ;将 r2 和 r10 的差送入 r10
        ;使 r6=原 r6/10,r7=原 r7*10+原 r6%10
        add     r6,r0,r9
        add     r7,r7,r10
        j       Loop
```

在这段程序中，循环体的出现造成了控制相关。

（2）控制相关的解决

由于输入为四位数，循环次数固定为四次，将循环体的内容展开。可以降低控制相关的个数。如下：

```
divur9,r6,r8   ;r6 寄存器中的数除以 r8 寄存器中的数放到 r9 寄存器中
mult    r10,r9,r8       ;r9 寄存器中的数乘以 r8 寄存器中的数放到 r10 中
mult    r7,r7,r8   ;r7 寄存器中的数乘以 r8 寄存器中的数放到 r7 中
sub     r10,r2,r10      ;将 r2 和 r10 的差送入 r10
add     r6,r0,r9
add     r7,r7,r10

divur9,r6,r8   ;r6 寄存器中的数除以 r8 寄存器中的数放到 r9 寄存器中
mult    r10,r9,r8       ;r9 寄存器中的数乘以 r8 寄存器中的数放到 r10 中
mult    r7,r7,r8   ;r7 寄存器中的数乘以 r8 寄存器中的数放到 r7 中
sub     r10,r2,r10      ;将 r2 和 r10 的差送入 r10
add     r6,r0,r9
add     r7,r7,r10

divur9,r6,r8   ;r6 寄存器中的数除以 r8 寄存器中的数放到 r9 寄存器中
mult    r10,r9,r8       ;r9 寄存器中的数乘以 r8 寄存器中的数放到 r10 中
mult    r7,r7,r8   ;r7 寄存器中的数乘以 r8 寄存器中的数放到 r7 中
sub     r10,r2,r10      ;将 r2 和 r10 的差送入 r10
add     r6,r0,r9
add     r7,r7,r10

divur9,r6,r8   ;r6 寄存器中的数除以 r8 寄存器中的数放到 r9 寄存器中
mult    r10,r9,r8       ;r9 寄存器中的数乘以 r8 寄存器中的数放到 r10 中
```

```
mult    r7,r7,r8   ;r7 寄存器中的数乘以 r8 寄存器中的数放到 r7 中
sub     r10,r2,r10     ;将 r2 和 r10 的差送入 r10
add     r6,r0,r9
add     r7,r7,r10
```

重新运行后控制相关数量减少为 9，如图 9 所示。

图 9 解决控制相关的数据截图

6、结构相关及解决

（1）结构相关产生的原因

```
mult    r10,r9,r8      ;r9 寄存器中的数乘以 r8 寄存器中的数放到 r10 中
mult    r7,r7,r8   ;r7 寄存器中的数乘以 r8 寄存器中的数放到 r7 中
```

在这段语句运行时需要连续进行乘法操作，由于乘法器只有一个，产生结构相关。

（2）结构相关的解决

添加乘法器的个数，如图 10 所示。

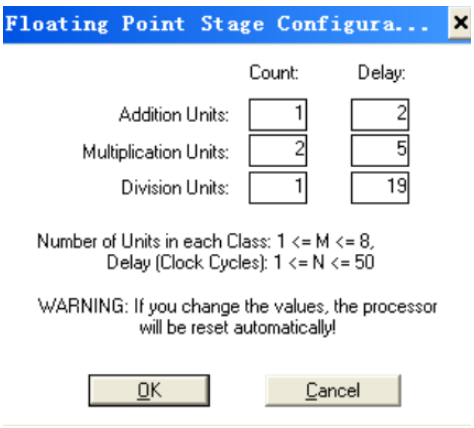


图 10 增加乘法器界面的截图

再次运行程序可以发现结构相关数量降低，降低到 0 个，如图 11 所示。

图 11 解决结构相关的数据截图

7、程序流程图

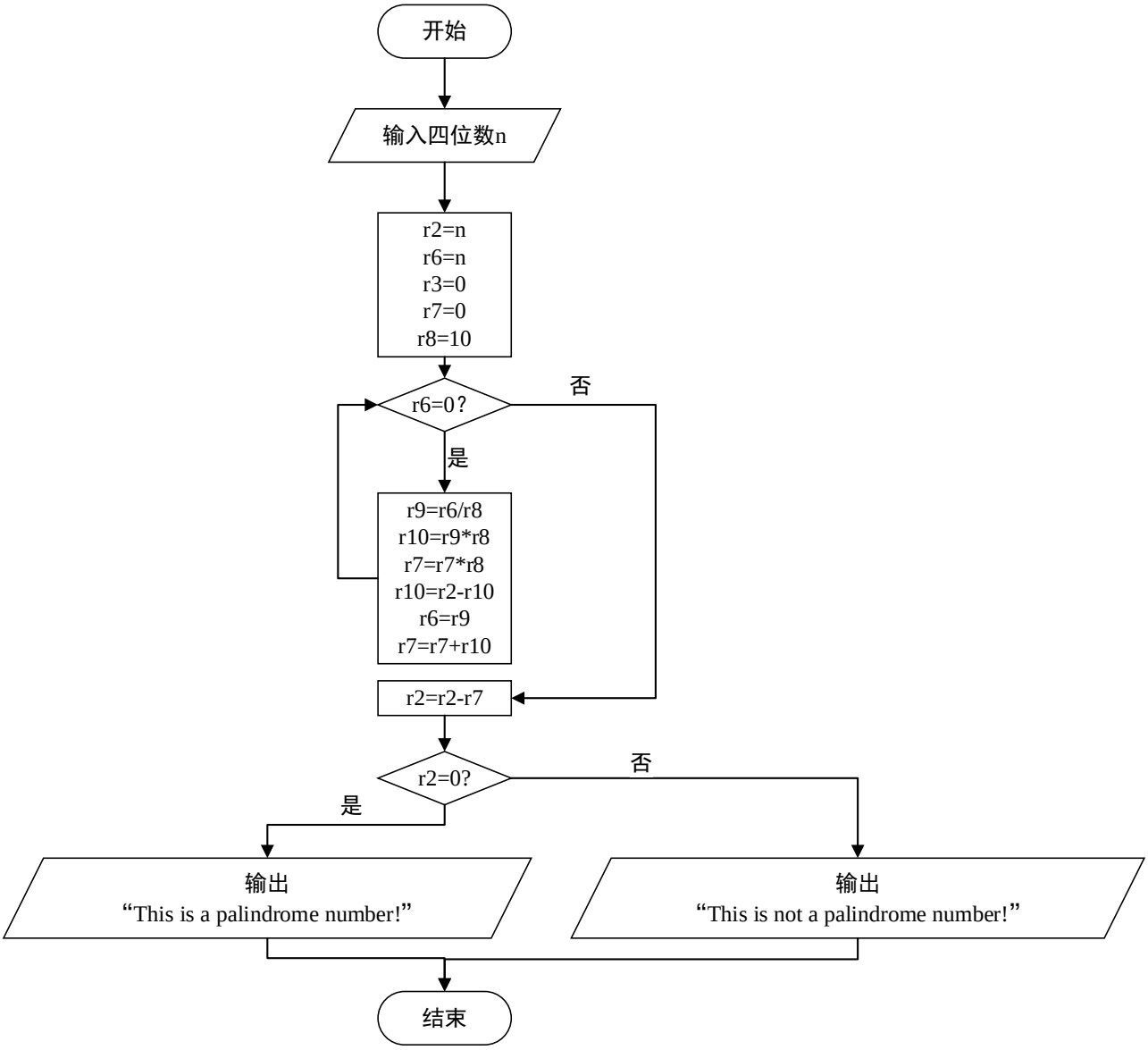


图 12 程序流程图

八、实验结论：

1、通过采用重定向技术减少了数据相关；

2、通过展开循环来减少控制相关；

3、通过增加硬件的数目来减少结构相关；

4、执行程序的顺序会影响程序执行是否正确，必须先执行源程序，再执行 input.s；修改后的程序必须清空之前所有的操作之后再重新运行。

九、总结及心得体会：

在实验过程中，通过编写实验代码，熟悉了一些基本 DLX 汇编指令的使用，对 DLX 汇编语言的 trap 机制有了一定了解；在减少程序运行过程中出现的三种相关的过程中，加深了对于数据相关、控制相关、结构相关的理解，对指令流水有了更加深刻的认识。通过帮助部分同学，提高了对 DLX 汇编语言的阅读能力，对 DLX 汇编中输入、输出、整形与浮点型数据之间的转换等功能更加熟练。

报告评分：

指导教师签字：

实　验　报　告 2

学生姓名：	学　号：	时间： 地点：
实验课程名称：计算机系统结构		

一、实验名称：鲲鹏 Hyper Tuner 性能分析工具实验

二、实验原理：

1、鲲鹏性能分析工具 Hyper Tuner

Hyper Tuner 即性能分析工具，支持鲲鹏平台上的系统性能分析、Java 性能分析和系统诊断，提供系统全景及常见应用场景下的性能采集和分析功能，并基于调优专家系统给出优化建议。同时提供调优助手，指导用户快速调优系统性能。鲲鹏性能分析工具支持 IDE 插件(vs Code、IntelliJ)和浏览器两种工作模式，分别同性能分析 Server 一起完成性能分析和优化等任务。

2、矩阵乘法优化方法

矩阵乘法可以拆分并行计算，且并行计算分支相对独立，可以使用鲲鹏的 NEON 指令来提升执行效率。NEON 指令通过将对单个数据的操作扩展为对寄存器，也即同一类型元素矢量的操作，从而大大减少了操作次数，以此来提升执行效率。

三、实验目的：

本实验基于鲲鹏云服务器部署并熟悉性能分析工具 Hyper Tuner。通过此次实验，能够掌握使用鲲鹏性能分析工具 Hyper Tuner 创建系统性能分析以及函数分析任务　2、使用鲲鹏的 NEON 指令来提升矩阵乘法执行效率

四、实验内容：

1、准备实验环境，可以使用华为云官方提供的沙箱实验室，也可以使用学校提供的华为云弹性服务器，根据实验教程安装依赖工具和 Hyper Tuner；

2、修改程序，因内存空间不足，修改数据规模 N 的定义，修改为 130000000；

3、登录 Hyper Tuner，编译并运行源代码，创建工程和任务进行系统性能全景分析。

4、创建进程/线程性能分析任务；

5、创建 C/C++性能分析任务；

6、编译程序并查看 multiply.c 程序中乘法函数 multiply 消耗时间和热点函数的占用率；编译 NEON 指令优化后的代码并查看消耗时间和占用率，进行对比；

7、阅读 multiply_simd.c 源代码，通过互联网搜索 NEON 指令的文档，自主编写两个 N*N 矩阵相乘的代码并进行优化；

8、将自主编写的代码上传至华为云服务器，编译执行，统计执行的时间进行对比

五、实验器材（设备、元器件）：

电脑一台
FinalShell（用于远程连接云服务器）
华为云服务器
鲲鹏 Hyper Tuner 性能分析工具

六、实验步骤及操作：

1、准备实验环境

（1）配置安全组，开放 8086 端口，如图 1 所示。

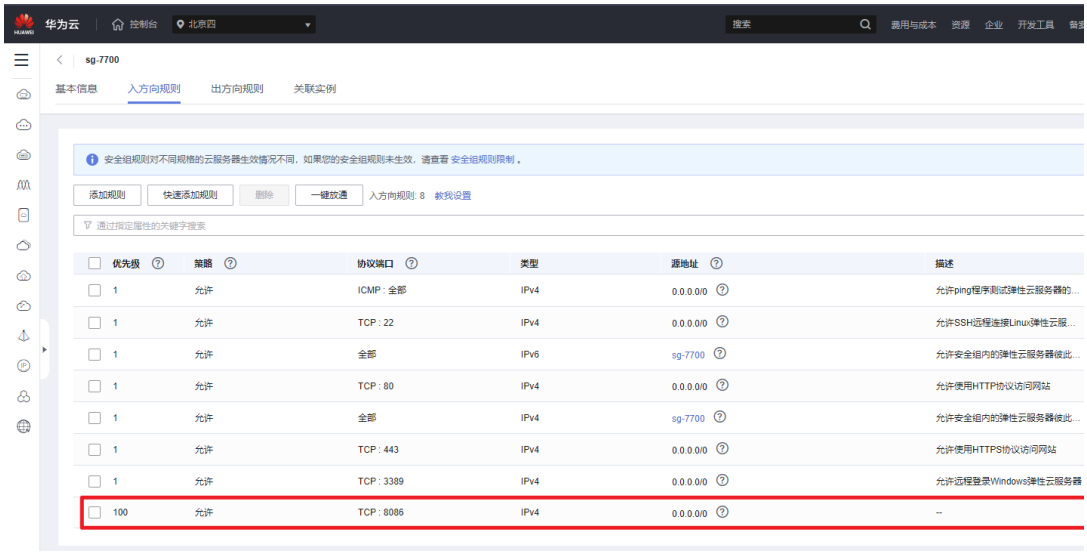


图 1 安全组配置

（2）购买云服务器，云服务器配置如图 2

图 2 云服务器配置

（3）用 Finalshell 登录已购买的云服务器，如图 3。

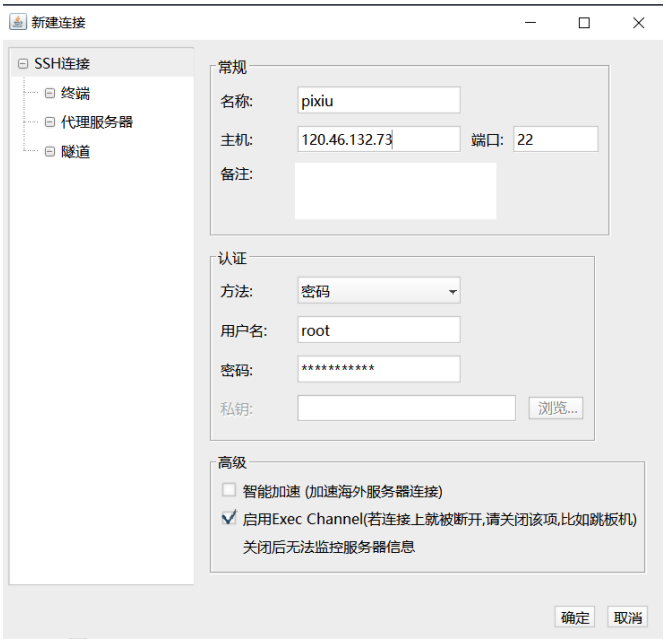


图 3 登录云服务器

（2）依次使用下列命令，安装依赖工具并激活

① 设置 SSH 超时断开时间，防止服务器断连

sed -i '112a ClientAliveInterval 600\nClientAliveCountMax 10' /etc/ssh/sshd_config && systemctl restart sshd

② 安装 centos-release-scl，安装完成后如图 4 所示。

yum install centos-release-scl -y

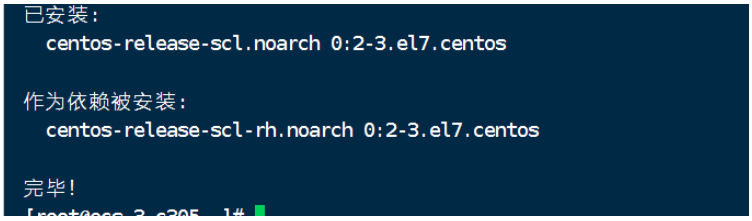


图 4 centos-release-scl 安装完成

③ 安装 devtoolset，安装完成后如图 5 所示。

yum install devtoolset-7-gcc* -y

图 5 devtoolset 安装完成

④ 激活对应的 devtoolset

scl enable devtoolset-7 bash

⑤ 安装 jdk 11 版本并在/home 目录下重命名为 jdk 文件夹

cd /home && wget https://mirrors.huaweicloud.com/kunpeng/archive/compiler/bisheng_jdk/bisheng-jdk-11.0.9-linux-aarch64.tar.gz && tar -zxvf bisheng-jdk-11.0.9-linux-aarch64.tar.gz && mv bisheng-jdk-11.0.9 jdk

（3）安装 Hyper Tuner

① 由于 C/C++性能分析任务需要最新版本的 Hyper Tuner 工具，因此用云服务器在鲲鹏社区下载软件包“Hyper-Tuner_2.5.0.1_linux.tar.gz”安装在“/home”的根目录下，并解压，命令如下，下载和解压过程如图 6 所示。

cd /home && wget https://kunpeng-repo.obs.cn-north-4.myhuaweicloud.com/Hyper%20Tuner/Hyper%20Tuner%202.5.0.1/Hyper-Tuner_2.5.0.1_linux.tar.gz && tar -zxvf Hyper-Tuner_2.5.0.1_linux.tar.gz

```
[root@ecs-3-c305 ~]# cd /home && wget https://kunpeng-repo.obs.cn-north-4.myhuaweicloud.com/Hyper%20Tuner/Hyper%20Tuner%202.5.0.1/Hyper-Tuner_2.5.0.1_linux.tar.gz
 && tar -zxvf Hyper-Tuner_2.5.0.1_linux.tar.gz
--2023-04-24 14:23:58--  https://kunpeng-repo.obs.cn-north-4.myhuaweicloud.com/Hyper%20Tuner/Hyper%20Tuner%202.5.0.1/Hyper-Tuner_2.5.0.1_linux.tar.gz
正在解析主机 kunpeng-repo.obs.cn-north-4.myhuaweicloud.com (kunpeng-repo.obs.cn-north-4.myhuaweicloud.com)... 100.125.80.29
正在连接 kunpeng-repo.obs.cn-north-4.myhuaweicloud.com (kunpeng-repo.obs.cn-north-4.myhuaweicloud.com)|100.125.80.29|:443... 已连接。
已发出 HTTP 请求，正在等待回应... 200 OK
长度: 1344967343 (1.3G) [application/gzip]
正在保存至: “Hyper-Tuner_2.5.0.1_linux.tar.gz”

100%[=============================================================================================================>] 1,344,967,343  219MB/s 用时 6.2s

2023-04-24 14:24:04 (207 MB/s) - 已保存 “Hyper-Tuner_2.5.0.1_linux.tar.gz” [1344967343/1344967343])

Hyper_tuner/
Hyper_tuner/Hyper-Tuner_2.5.0.1_linux.tar.gz
Hyper_tuner/Hyper-Tuner_2.5.0.1_linux_HPC-Collector-cmd.tar.gz
Hyper_tuner/Open_Source_Software_Notice.txt
Hyper_tuner/crldata.crl
Hyper_tuner/file_list.txt
Hyper_tuner/file_list.txt.cms
Hyper_tuner/install.sh
[root@ecs-3-c305 home]#
```

图 6 下载并解压软件包

② 实验如下命令，安装系统性能优化工具，其中 192.168.1.30 为服务器私有 ip，安装完成后如图 7 所示

cd /home/Hyper_tuner && ./install.sh -a -i -ip=192.168.1.30 -jh=/home/jdk

```
Hyper_tuner install Success
  ==========================================================
  The login URL of Hyper_Tuner is https://192.168.1.30:8086/user-management/#/login
  ==========================================================
  If 192.168.1.30:8086 has mapping IP, please use the mapping IP.
[root@ecs-3-c305 Hyper_tuner]#
```

图 7 系统性能优化工具安装完成

③ 登录 Hyper Tuner

用浏览器访问链接 https://120.46.132.73:8086/user-management/#/login，其中 120.46.132.73 为服务器弹性公网 ip，首次登录情况如图 8，设置管理员密码，选择系统性能分析功能。

图 8 首次登录 Hyper Tuner

2、编译并运行源代码，创建工程和任务进行系统性能全景分析

（1）执行如下命令，新建文件夹/opt/testdemo：

mkdir /opt/testdemo

（2）使用 Finalshell 上传源代码到文件夹/opt/testdemo 如图 9 所示，修改数据规模 N 的定义如图 10 所示。

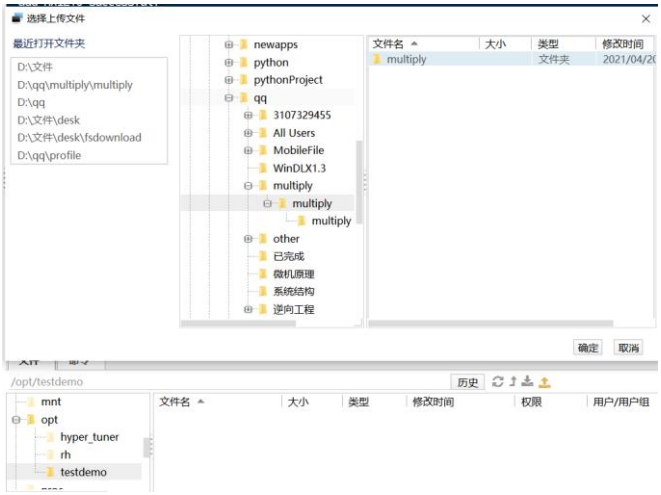
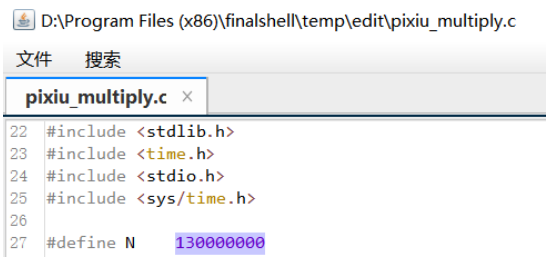


图 9 上传源代码



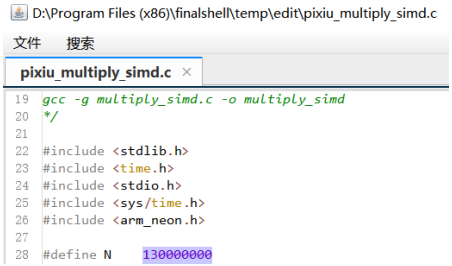


图 10 修改数据规模 N 的定义

（3）执行如下命令，进入 multiply 文件夹，编译 multiply.c 并赋予执行文件所有用户只读、只写、可执行权限：

cd /opt/testdemo/multiply && gcc -g multiply.c -o multiply && chmod -R 777 /opt/testdemo/multiply

（4）执行如下命令，将 multiply 测试程序绑定 CPU 核启动（当前程序绑定到 CPU 核 1，循环运行 multiply 程序 200 次），使用后台启动脚本，程序运行的输出（标准输出（1））将会保存到 multiply.out 文件，错误信息（2）会重定向到 multiply.out 文件，执行后会显示进程的 PID，如图 11 所示。

cd /opt/testdemo/multiply && nohup bash multiply_start.sh >>multiply.out 2>&1 &

```
[root@ecs-3-c305 multiply]# cd /opt/testdemo/multiply && nohup bash multiply_start.sh >>multiply.out 2>&1 &
[1] 20160
```

图 11 启动测试程序

（5）创建工程，如图 12 所示

图 12 创建工程

（6）创建全景分析任务，如图 13 所示

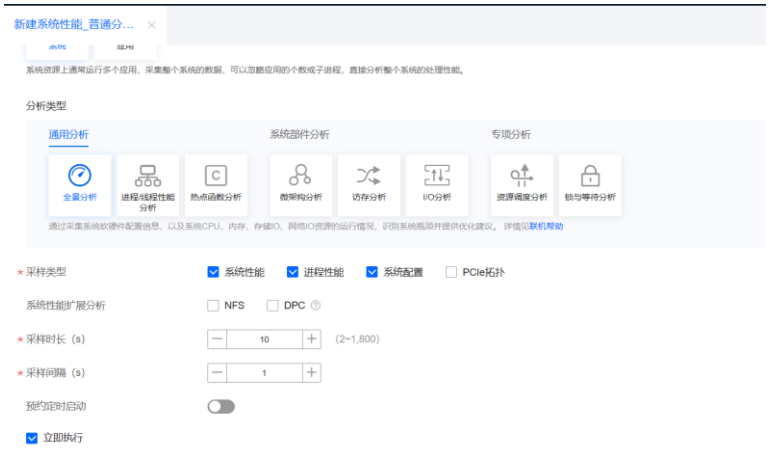


图 13 创建全景分析任务

（7）查看采集分析结果

① 系统配置，点击“检测到 CPU 利用率高”显示优化建议，如图 14 所示；

图 14 优化建议

② 系统性能，在图 15 中，可以看到 top5 的 CPU 核在采集时间内的利用率变化，在图 16 中，可以看到在采集时间内的各项数据的平均值。

由此可见，当前 CPU 核 1 的使用率（“性能”页签下%CPU 的数值）接近 100%，并且绝大部分消耗在用户态。说明该程序全部消耗在用户态计算，没有其他 IO 或中断操作。

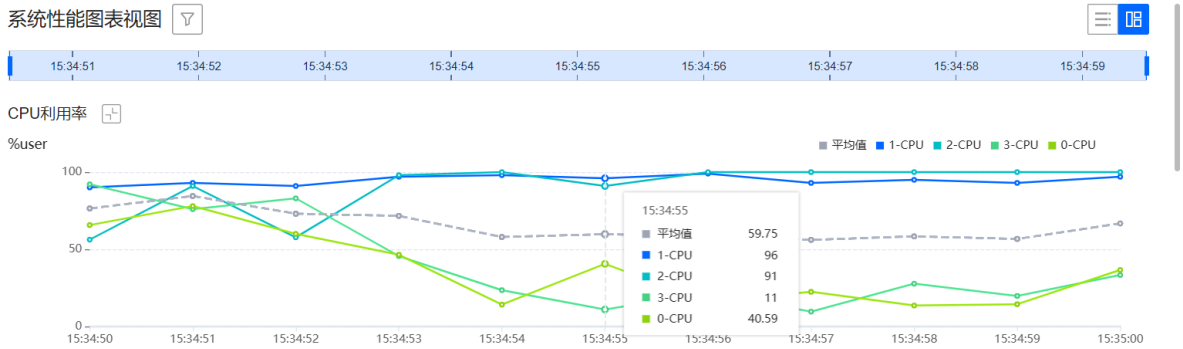


图 15 CPU 利用率

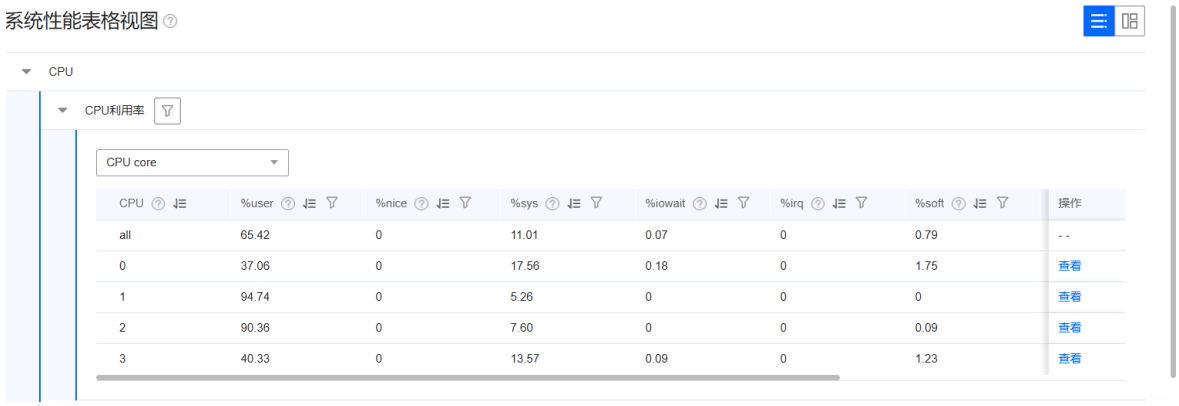


图 16 在采集时间内各 CPU 核各项数据的平均值

3、进程/线程性能分析

（1）创建进程/线程性能分析任务，如图 17 所示

图 17 创建进程/线程性能分析任务

（2）查看采集分析结果

在图 18 中，可以看到 multiply 程序在消耗大量的 CPU，同时全部消耗在用户态中，由此我们可以推测很可能是自身代码实现算法差的问题。

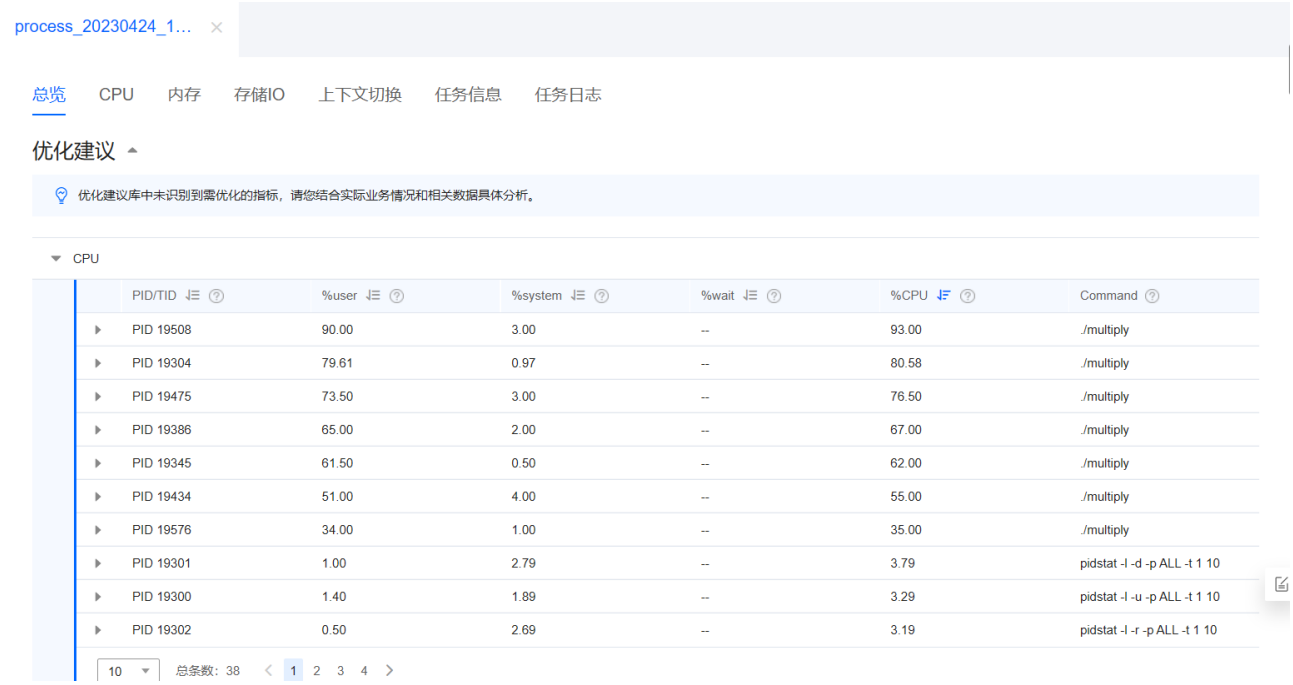


图 18 各任务按%CPU（占用 CPU 百分比）降序排列

4、C/C++性能分析

（1）创建热点函数分析任务，如图 19。

图 19 创建热点函数分析任务

（2）查看采集分析结果

① 点击“检测到 C/C++程序的 CPU 利用率高”显示优化建议，如图 20 所示；

图 20 优化建议

② 在图 21 中，可以看到 multiply 程序消耗大量的 CPU 时钟周期，由此我们可以推测很可能是自身代码实现算法差的问题。

图 21 各热点模块信息

③ 从图 22 可以看出，在 multiply 程序中乘法函数 multiply 占用大量 CPU 时钟周期，首先考虑优化 multiply 函数。

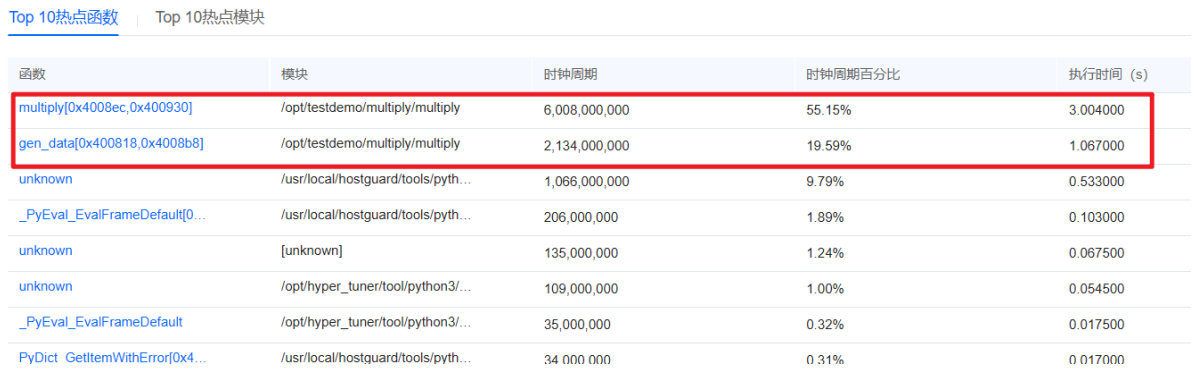


图 22 各热点函数信息

5、对比分析优化前后乘法函数 multiply 消耗时间和热点函数的占用率

（1）使用如下命令，编译程序 multiply.c：

cd /opt/testdemo/multiply && gcc -g -O2 -o multiply multiply.c && chmod -R 777

/opt/testdemo/multiply

（2）使用如下命令运行 multiply.c 程序，查看乘法函数 multiply 消耗时间，实验结果如图 23 所示。

cd /opt/testdemo/multiply/ && ./multiply

```
[root@ecs-3-c305 ~]# cd /opt/testdemo/multiply/ && ./multiply
217156.000000, 217156.000000, 217156.000000, 217156.000000
Execution time = 322.667 ms
```

图 23 multiply.c 程序中乘法函数 multiply 消耗时间

（3）依次使用如下命令采集热点函数占用率，结果如图 24 所示，可以看到 multiply 函数占用 67.78%的 CPU，首先考虑优化 multiply 函数。

cd /opt/testdemo/multiply/ && sudo perf record --call-graph dwarf ./multiply -d 1 -b

sudo perf report -i perf.data > perf_multiply.txt

less perf_multiply.txt

```
# Total Lost Samples: 0
#
# Samples: 1K of event 'cpu-clock:pppH'
# Event count (approx.): 492000000
#
# Children      Self  Command   Shared Object      Symbol
# ........  ........  ........  .................  ...............................
#
    99.95%     0.00%  multiply  multiply           [.] _start
            |
            ---_start
               _start
               __libc_start_main
               main
               |
               |--67.78%--multiply
               |          |
               |           --5.03%--el0_da
```

图 24 部分热点函数占用率

（4）使用如下命令编译运行 multiply_simd.c 程序，查看乘法函数 multiply 消耗时间，实验结果如图 25 所示，相比于未优化的 322.667ms，优化后的程序中乘法函数 multiply 消耗时间为 151.767ms，得到了大幅降低。

cd /opt/testdemo/multiply && gcc -g -O2 -o multiply_simd multiply_simd.c && chmod -R 777 /opt/testdemo/multiply

cd /opt/testdemo/multiply/ && ./multiply_simd

```
[root@ecs-3-c305 multiply]# cd /opt/testdemo/multiply && gcc -g -O2 -o multiply_simd multiply_simd.c && chmod -R 777 /opt/testdemo/multiply
[root@ecs-3-c305 multiply]# cd /opt/testdemo/multiply/ && ./multiply_simd
217156.000000, 217156.000000, 217156.000000, 217156.000000
Execution time = 151.767 ms
```

图 25 multiply_simd.c 程序中乘法函数 multiply 消耗时间

（5）依次使用如下命令采集热点函数占用率，结果如图 26 所示，可以看到 multiply 函数占用 51.42%的 CPU，相比未优化前的 67.78%有所降低。

cd /opt/testdemo/multiply/ && sudo perf record --call-graph dwarf ./multiply_simd -d 1 -b

sudo perf report -i perf.data > perf_multiply_simd.txt

less perf_multiply_simd.txt

```
# Total Lost Samples: 0
#
# Samples: 1K of event 'cpu-clock:pppH'
# Event count (approx.): 325250000
#
# Children      Self  Command       Shared Object      Symbol
# ........  ........  ............  .................  ...............................
#
    99.85%     0.00%  multiply_simd  multiply_simd      [.] _start
            |
            ---_start
               _start
               __libc_start_main
               main
               |
               |--51.42%--multiply_neon
               |          |
               |          |--12.68%--vmulq_f32 (inlined)
               |          |
:
```

图 26 部分热点函数占用率

7、对比分析优化前后两个 N*N 矩阵相乘的消耗时间

（1）将编写的两个矩阵乘法函数上传到云服务器的文件夹/opt/testdemo

（2）分别使用如下命令编译运行程序 multiply_2.c 和 multiply_2_simd.c，查看乘法函数 multiply 消耗时间，实验结果如图 27 所示，相比于未优化的 1997.762ms，优化后的程序中乘法函数 multiply 消耗时间为 600.730ms，得到了大幅降低。

cd /opt/testdemo/multiply_2 && gcc -g -O2 -o multiply_2 multiply_2.c && chmod -R 777 /opt/testdemo/multiply_2

cd /opt/testdemo/multiply_2/ && ./multiply_2

cd /opt/testdemo/multiply_2 && gcc -g -O2 -o multiply_2_simd multiply_2_simd.c && chmod -R 777 /opt/testdemo/multiply_2

cd /opt/testdemo/multiply_2/ && ./multiply_2_simd

```
[root@ecs-3-c305 ~]# cd /opt/testdemo/multiply_2 && gcc -g -O2 -o multiply_2 multiply_2.c && chmod -R 777 /opt/testdemo/multiply_2
[root@ecs-3-c305 multiply_2]# cd /opt/testdemo/multiply_2/ && ./multiply_2
217156000.000000, 217156000.000000, 217156000.000000, 217156000.000000
Execution time = 1997.762 ms
[root@ecs-3-c305 multiply_2]# cd /opt/testdemo/multiply_2 && gcc -g -O2 -o multiply_2_simd multiply_2_simd.c && chmod -R 777 /opt/testdemo/multiply_2
[root@ecs-3-c305 multiply_2]# cd /opt/testdemo/multiply_2/ && ./multiply_2_simd
217156000.000000, 217156000.000000, 217156000.000000, 217156000.000000
Execution time = 600.730 ms
[root@ecs-3-c305 multiply_2]#
```

图 27 程序 multiply_2.c(上)和 multiply_2_simd.c(下)中乘法函数 multiply 消耗时间

七、实验数据及结果分析：

1、一维矩阵主要代码及说明

① 未优化前的程序 multiply.c 的主要代码如下

```
// 为矩阵分配内存空间，并用伪随机数填充一维矩阵
void gen_data(void){
    unsigned i;
    g_a = (float*)malloc(N * sizeof(float));
    g_b = (float*)malloc(N * sizeof(float));
    g_c = (float*)malloc(N * sizeof(float));
    if (g_a == NULL || g_b == NULL || g_c == NULL) {
        perror("Memory allocation through malloc failed");
        exit(EXIT_FAILURE);
    }
    for (i = 0; i < N; i++) {
        g_a[i] = (float)(SEED * 0.1);
        g_b[i] = (float)(SEED * 0.1);
    }
}
// 释放为矩阵分配的空间
void free_data(void){
    free(g_a);
    free(g_b);
    free(g_c);
}
// 将矩阵的对应位置相乘
void multiply(void){
    unsigned i;
    for (i = 0; i < N; i++) {
        g_c[i] = g_a[i] * g_b[i];
    }
}
// 输出一维矩阵 c 的最前两个和最后两个元素
void print_data(void){
    printf("%f, %f, %f, %f\n", g_c[0], g_c[1], g_c[N - 2], g_c[N - 1]);
}
int main(void){
    double msecs;
    struct timeval before, after;
    gen_data();
    // 记录 multiply 函数执行之前的时间
    gettimeofday(&before, NULL);
    multiply();
```

```
        // 记录 multiply 函数执行之后的时间
        gettimeofday(&after, NULL);
        // 转化时间进制
        msecs = (after.tv_sec - before.tv_sec) * 1000.0 + (after.tv_usec - before.tv_usec) / 1000.0;
        print_data();
        printf("Execution time = %2.3lf ms\n", msecs);
        free_data();
        return 0;
    }
```

② 优化后程序 multiply_simd.c 的主要代码，除乘法函数外其余均与 multiply.c 一致，乘法函数主要代码如下：

```
    void multiply_neon(void)
    {
        int i;
        float32x4_t src1, src2, dst;
        for (i = 0; i < (N & ((~(unsigned)0x3))); i += 4) {
            src1 = vld1q_f32(g_a + i);      // 将地址从 g_a+i 开始的 4 个浮点数加载到向量 src1
            src2 = vld1q_f32(g_b + i);
            dst = vmulq_f32(src1, src2);   // 将向量 src1 和 src2 对应元素相乘，存到向量 dst
            vst1q_f32(g_c + i, dst);        // 将向量 dst，存到 g_c 的对应位置
        }
        // 处理剩余部分的数据
        for (; i < N; i++) {
            g_c[i] = g_a[i] * g_b[i];
        }
    }
```

2、二维矩阵相乘主要代码及说明

① 未优化前的程序 multiply_2.c 的主要代码

```
    #define N       1000                    // 设置矩阵大小 1000*1000
    #define SEED      0x1234
    double *g_a, *g_b, *g_c;                // 由于数据精度原因，三个矩阵数据类型设置为 double
    void gen_data(void)
    {
        unsigned i;
        g_a = (double*)malloc(N * N * sizeof(double));
        g_b = (double*)malloc(N * N * sizeof(double));
        g_c = (double*)malloc(N * N * sizeof(double));
        if (g_a == NULL || g_b == NULL || g_c == NULL) {
            perror("Memory allocation through malloc failed");
            exit(EXIT_FAILURE);
        }
        for (i = 0; i < N*N; i++) {
```

```
            g_a[i] = (double)(SEED * 0.1);
            g_b[i] = (double)(SEED * 0.1);
        }
    }
    // 释放为矩阵分配的空间
    void free_data(void){
        free(g_a);
        free(g_b);
        free(g_c);
    }
    // 进行矩阵乘法计算
    void multiply(void)
    {
        unsigned i,j,k;
        // 为了便于用 NEON 指令进行优化，矩阵 a 按行存储，矩阵 b 按列存储
        for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++) {
                for (k = 0; k < N; k++) {
                    g_c[i*N+j] += g_a[i*N+k] * g_b[j*N+k];
                }
            }
        }
    }
    // 输出矩阵 c 的四个顶点位置的数据
    void print_data(void){
        printf("%f, %f, %f, %f\n", g_c[0], g_c[N-1], g_c[(N-1)*N], g_c[N*N-1]);
    }
```

② 优化后程序 multiply_2_simd.c 的主要代码，除乘法函数外其余均与 multiply.c 一致，乘法函数主要代码如下：

```
    void multiply(void)
    {
        unsigned i,j,k;
        float64x2_t srca1,srca2,srcb1,srcb2,tmp1,tmp2,dst1,dst2;   // 用两个 float64x2_t 的向量存储四个数据
        for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++) {
                // 清零向量 dst1 和 dst2，以重用两向量
                dst1 = vdupq_n_f64(0.0f);
                dst2 = vdupq_n_f64(0.0f);
                for (k = 0; k < (N & ((~(unsigned)0x3))); k+=4) {
                    srca1 = vld1q_f64(g_a + i*N+k);
                    srca2 = vld1q_f64(g_a + i*N+k+2);
                    srcb1 = vld1q_f64(g_b + j*N+k);
                    srcb2 = vld1q_f64(g_b + j*N+k+2);
```

```
                        // 对应位置相乘
                        tmp1 = vmulq_f64(srca1, srcb1);
                        tmp2 = vmulq_f64(srca2, srcb2);
                        // 累加
                        dst1 = vaddq_f64(tmp1,dst1);
                        dst2 = vaddq_f64(tmp2,dst2);
                    }
                    g_c[i*N+j] = vgetq_lane_f64(dst1,0) + vgetq_lane_f64(dst1,1) + vgetq_lane_f64(dst2,0) + vgetq_lane_f64(dst2,0); // 从向量 dst1 和 dst2 取数，相加得到乘积总和
                    // 加上未被计算的剩余部分求和
                    for (; k < N; k++){
                        g_c[i*N+j] += g_a[i*N+k] * g_b[j*N+k];
                    }
                }
            }
        }
```

八、实验结论：

1、使用 NEON 指令优化矩阵乘法，一次性加载四个数据到寄存器，用一条指令处理多个数据，可以减少程序执行过程中的访存次数，提高程序并行程度，从而提高程序运行效率。

2、通过使用 NEON 指令来优化计算，可以大幅减少函数指令数目，缩短程序运行时间，提升程序执行效率。

十、总结及心得体会：

通过本次实验，熟练掌握了云服务器的购买及其使用，熟悉了用鲲鹏性能分析工具 Hyper Tuner 创建系统性能分析任务、进程/线程分析任务和热点函数分析任务对程序进行分析的方法。通过自行编写矩阵乘法并用 NEON 指令对程序进行优化，使用 SIMD 技术，向量化处理数据，用一条指令处理多个数据，以提高程序效率，我熟悉了 NEON 指令的使用，并且了解了 NEON 指令提高程序运行速度的原理。

报告评分：

指导教师签字：