YOLO 卷积函数的 SMART 仿真

1、实验目的

通过将 YOLO 源码的卷积函数放在 SMART 上进行仿真和性能监测，进一步介绍嵌入式 C 裸机程序在 C910 上的开发过程和仿真原理。通过观测分支预测开关对程序性能的影响，理解分支预测在现代处理器中的重要性。通过指令分布统计，了解内存墙的概念，并进一步理解缓存在现代处理器中的重要性。

2、实验步骤（包括实验结果，数据记录、截图等）

1) 文件核对和执行仿真

a. 首先替换项目文件：

```
admin:/home/ECDesign/ecd21/qzh_21307140085>[66]cd smart9_release/
admin:/home/ECDesign/ecd21/qzh_21307140085/smart9_release>[67]cp ~/lab9/axi_slave128_copy.v ./rtl/platform/amba/axi/
cp: overwrite './rtl/platform/amba/axi/axi_slave128_copy.v'? y
admin:/home/ECDesign/ecd21/qzh_21307140085/smart9_release>[68]cp ~/lab9/soc.v ./rtl/platform/common/
cp: overwrite './rtl/platform/common/soc.v'? y
admin:/home/ECDesign/ecd21/qzh_21307140085/smart9_release>[69]cp -y ~/lab9/Makefile ./lib/
cp: invalid option -- 'y'
Try 'cp --help' for more information.
admin:/home/ECDesign/ecd21/qzh_21307140085/smart9_release>[70]cp ~/lab9/Makefile ./lib/
cp: overwrite './lib/Makefile'? y
admin:/home/ECDesign/ecd21/qzh_21307140085/smart9_release>[71]cp ~/lab9/tb.v ./tb/
cp: overwrite './tb/tb.v'? y
admin:/home/ECDesign/ecd21/qzh_21307140085/smart9_release>[72]]cp ~/lab9/crt0.s ./lib/
]cp: Command not found.
admin:/home/ECDesign/ecd21/qzh_21307140085/smart9_release>[73]cp ~/lab9/crt0.s ./lib/
cp: overwrite './lib/crt0.s'? y
admin:/home/ECDesign/ecd21/qzh_21307140085/smart9_release>[74]cp ~/lab9/pmu.h ./lib/clib/
cp: overwrite './lib/clib/pmu.h'? y
admin:/home/ECDesign/ecd21/qzh_21307140085/smart9_release>[75]cp -rf ~/lab9/conv_test/ ./case/
admin:/home/ECDesign/ecd21/qzh_21307140085/smart9_release>[76]
```

b. 然后在/workdir 目录下进行仿真，当完成仿真之后在 case/conv_test 目录下对生成的二进制文件反汇编：

```
admin:/home/ECDesign/ecd21/qzh_21307140085/smart9_release/case/conv_test>[93]../../tools/toolchain/RV64GC/bin/riscv64-unknown-elf-objdu
mp -S conv_test.elf > conv_test.s
conv_test.s: File exists.
```

c. 打开反汇编生成的 conv_test.s 文件，查看 guard_end 和 guard_start 的指令开始地址，与 tb 文件进行对比，发现其与 tb 文件中定义的开始结束位置不相等，对 tb 文件进行修改，然后将 conv_test.s 文件删除后重新跑仿真：

```
void guard_start()
{
    1b50:       1141                    addi    sp,sp,-16
    1b52:       e406                    sd      ra,8(sp)
```

```
void guard_end()
{
    1c10:       1141                    addi    sp,sp,-16
    1c12:       e406                    sd      ra,8(sp)

    num_cycle_end = get_cycle();
    1c14:       8dcff0ef                jal     ra,cf0 <get_cycle>
```

（反汇编的.s 文件中的 guard_start 和 guard_end）

```
`define GUARD_START_PC      40'h1850
`define GUARD_END_PC        40'h1910
```

(tb 文件中的始末位置)

```
iui.cp0_iu_ex3_rslt_vld
`define CP0_RSLT            `CPU_TOP.x_ct_top_0.x_ct_core.x_ct_cp0_top.x_ct_cp0_
iui.cp0_iu_ex3_rslt_data[63:0]

`define APB_BASE_ADDR       40'h4000000000

`define GUARD_START_PC      40'h1b50
`define GUARD_END_PC        40'h1c10


module tb();
reg clk;
```

(更改后的 tb 文件)

d. 跑出来后查看 run.log 文件，记录数据

```
num_cycle is 8936645

num_instret is 9449136

CPI is 0.945763

num_conditional_branch_mis is 253598

num_L1_Dcache_read_access is 4579889

num_L1_Dcache_read_miss is 12291

num_L1_Dcache_write_access is 1920568

num_L1_Dcache_write_miss is 1805

num_L2_Dcache_read_access is 481926

num_L2_Dcache_read_miss is 4183

num_L2_Dcache_write_access is 122750

num_L2_Dcache_write_miss is 400

guard_cycle_count is    8937286
```

(all_off 的 run.log)

f. 跑完上述步骤后，打开/smart9_release/lib 目录下的 ctrs0.s 文件，更改分支预测的设置，重复以上步骤：

```
# mhcr 4-RS, 5-BPE, 6-BTB, 7-IBPE, 12-L0BTB
# reserve the low 3 bits asserted, select 1 of 4 at the same time

li x3, 0x10f7     #all prediction on
#li x3,0x00b7      #BTB,L0BTB off
#li x3,0x0007      #all prediction off

csrs 0x7c1,x3  #mhcr
#csrs mhcr,x3

#mccr2
#li x3,0xe0000000
```

2) CPI、Cache 缺失率和分支预测统计

configure	all prediction off	BPE on, BTB off	all prediction on
cycle	8936645	5202663	5131578
insts	9449136	9449136	9449136
CPI	0.945763	0.550597	0.543074
conditional branch miss	253598	4439	5006
L1_Dread access	4579889	3780647	3795212
L1_Dread miss	12291	14250	14227
L1_Dread miss_rate	0.2684%	0.3770%	0.3749%
L1_Dwrite access	1920568	1897811	1898553
L1_Dwrite miss	1805	1858	1827
L1_Dwrite miss_rate	0.0940%	0.0979%	0.0962%
L2_Dread access	481926	469435	470737
L2_Dread miss	4183	4164	4164
L2_Dread miss_rate	0.8680%	0.8870%	0.8846%
L2_Dwrite access	122750	121027	121008
L2_Dwrite miss	400	399	399
L2_Dwrite miss_rate	0.3258%	0.3297%	0.3297%

3) 指令分布统计绘图

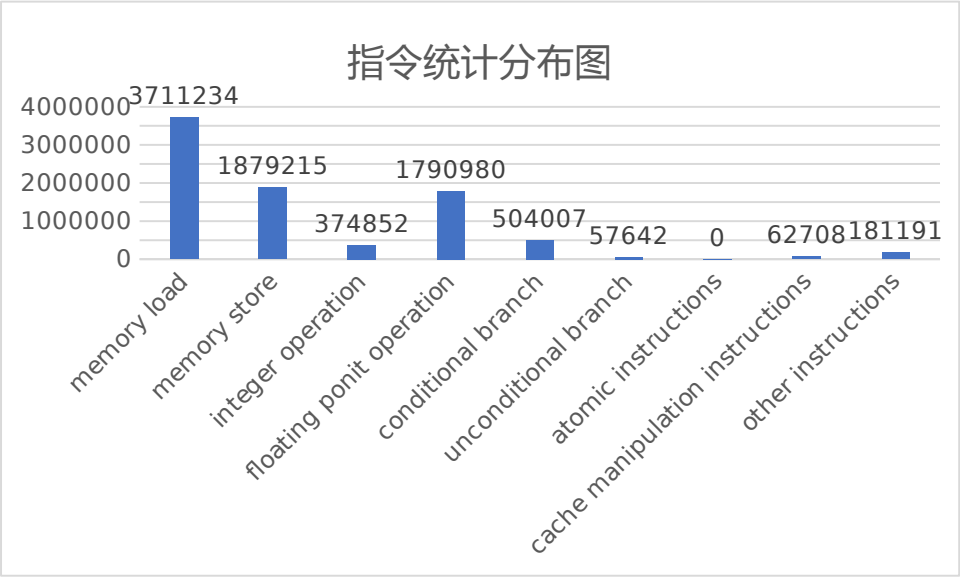
a. 在/workdir 下生成指令分布统计

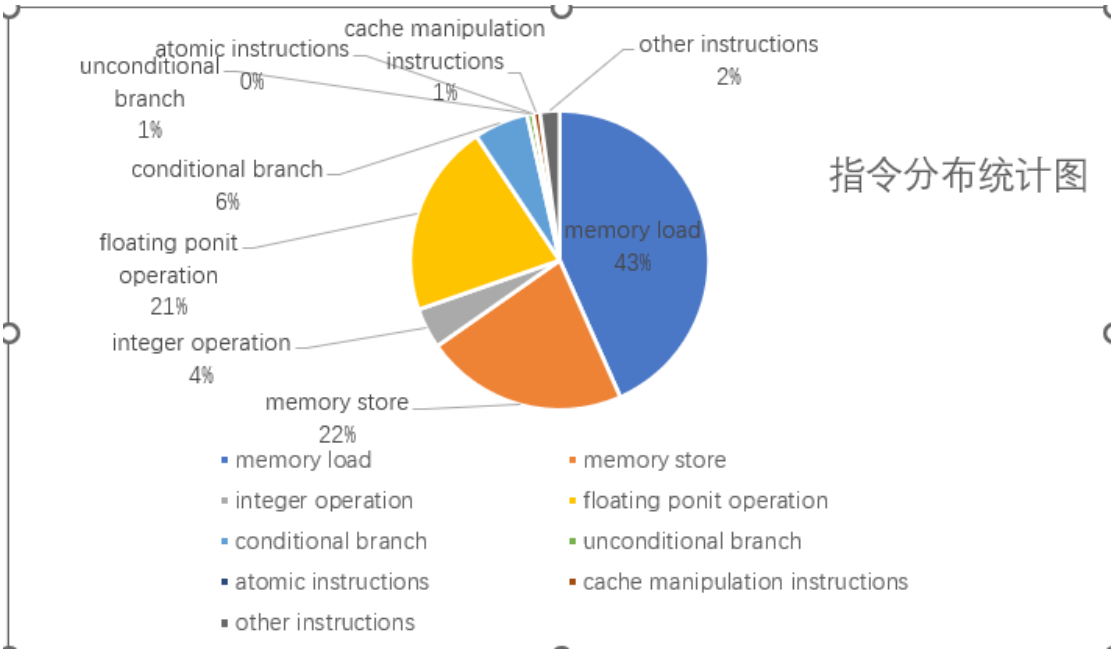
```
admin:/home/ECDesign/ecd21/qzh_21307140085/smart9_release/workdir>[44]cp -f ../case/conv_test/inst_proc ./
cp: overwrite './inst_proc'? y
admin:/home/ECDesign/ecd21/qzh_21307140085/smart9_release/workdir>[45]chmod +x inst_proc
admin:/home/ECDesign/ecd21/qzh_21307140085/smart9_release/workdir>[46]./inst_proc
```

b. 打开 inst_class 文件就可以查看指令的分布了：

```
memory access instructions:
        memory load:    3711234
        memory store:   1879215

arithmetic  instructions:
        integer operation:      374852
        floating ponit operation:       1790980

branch instructions:
        conditional branch:     504007
        unconditional branch:   57642

atomic instructions:    0
cache manipulation instructions (C910 customized extension):    62708
other instructions:     181191

total:  8561829
```

c. 根据该文件画出统计图：





3、实验分析和总结

1）C910 的开发与仿真原理：

a)首先，在/workdir 目录下执行脚本 run_case 时，run_case 脚本会解析命令行参数获得源码路径，使用./lib 下的 Makefile 文件通过工具链 RV64GC 对源码进行编译和链接。

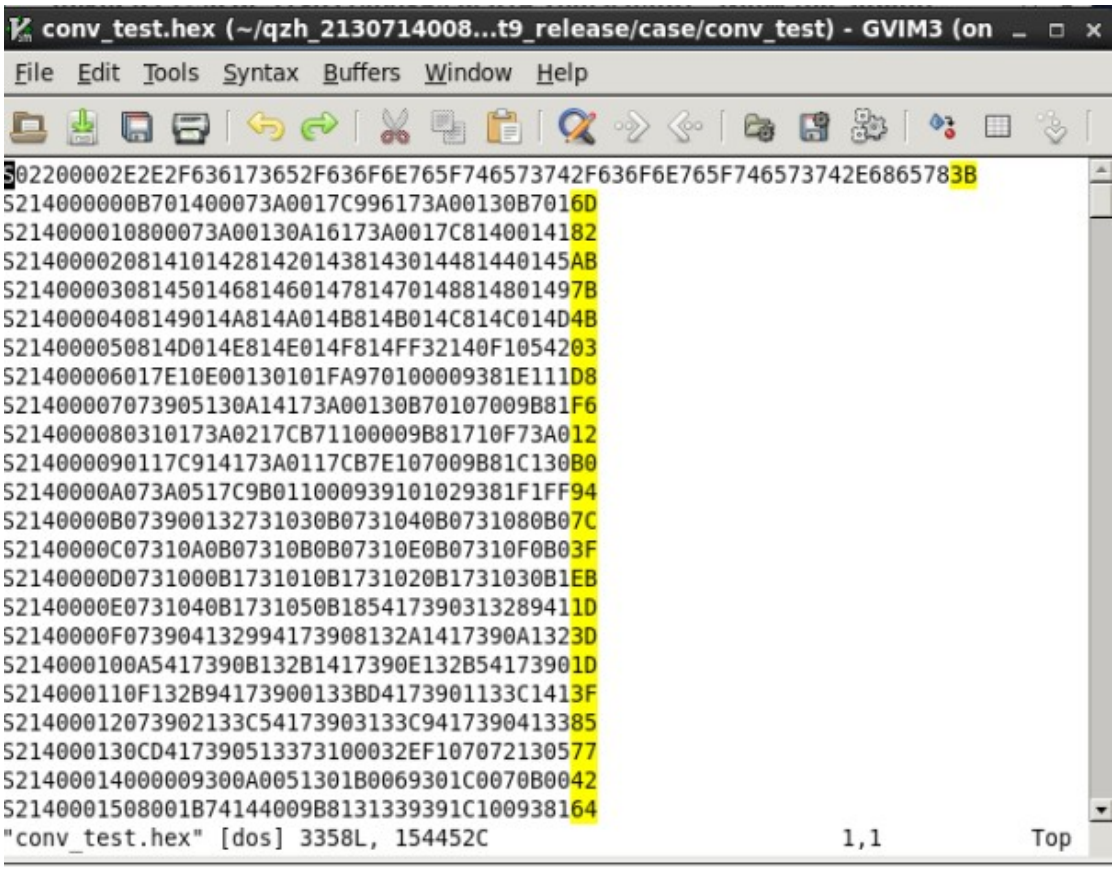
b)在./lib 文件中还有启动代码 crt0.s 和链接脚本 linker.lcf，其中链接脚本将内存划分为 MEM1 和 MEM2 两块，分别用于存储代码段于数据段。可以看见首先存入 MEM1 的代码是 crt0.s 的代码，用来完成对程序和寄存器的初始化。

```
MEMORY
{
MEM1(RWX)  : ORIGIN = 0x00000000,  LENGTH = 0x40000
MEM2(RWX)  : ORIGIN = 0x00040000,  LENGTH = 0xc0000
}
__kernel_stack = 0xee000 ;

ENTRY(__start)

SECTIONS {
    .text :
    {
        crt0.o (.text)
        *(.text*)
    } >MEM1
```

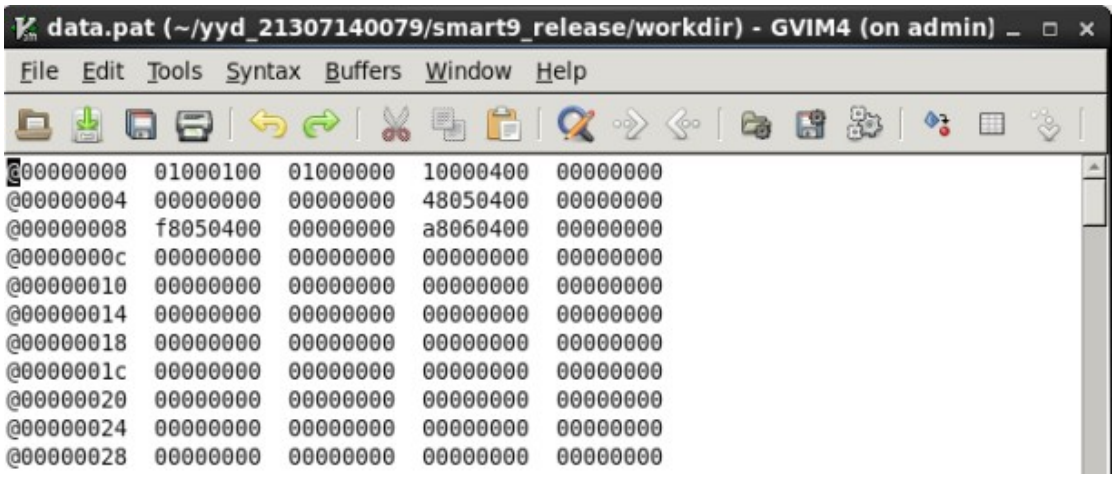
c)在完成程序的初始化之后，在 crt0.s 中将执行 jal main 语句跳转到需要执行的代码的主函数的位置。最终，GCC 工具依据链接脚本生成完整的二进制文件：

```
csrw mhpmevent21,x3

#*4 configure mcounteren and scounteren
#li x3,0xffffffff
#csrw mcounteren,x3 # enable super mode to read hpmcounter
#li x3,0xffffffff
#csrw scounteren,x3 # enable user mode to read hpmcounter

#*5 enable couters to count what you want
csrw mcountinhibit,x0

jal    main

.global __exit
```

d)生成二进制文件后，Makefile 继续编译,通过 RISC-V 工具链中的 objcopy 工具将二进制文件转换为.hex 文件，该文件以 16 进制字符的形式展现输入内存的内容。



e）为了能将二进制文件通过 Verilog 语言中的$readmenh 语法读入到内存模型中，再使用/tools/Srec2vmem 脚本将 hex 文件拆分为 inst.pat 和 data.pat 文件，分别对应链接脚本中的 MEM1 和 MEM2 的数据。该文件的格式满足$readmenh 的语法要求。



f)在 tb.v 文件中，可以看到仿真文件中将 inst.pat 和 data.pat 文件的内容通过中间数组 mem_inst_temp 读入到内存模块中。如此完成了对仿真模型的初始化输入，在运行仿真时，cpu 会自动依次读取内存模块的指令和数据，将需要输出的结果调用 write 函数输出到 run.log 中。

```
end
$readmemh("data_in",mem_datain_tmp);
```

```
RTL_MEM_DATA.ram1.mem[j][7:0]= mem_datain_tmp[i+1][7:0];
RTL_MEM_DATA.ram2.mem[j][7:0]= mem_datain_tmp[i+2][7:0];
RTL_MEM_DATA.ram3.mem[j][7:0]= mem_datain_tmp[i+3][7:0];
```

2）观测分支预测开关对程序性能的影响。

观测表格，可以得到如下结论：

ⅰ. 开启分支预测后，CPI 明显下降，且开启 BPE 的下降效果比开启 BTB 的下降效果明显，这可能是由于 BPE 的覆盖范围更大的原因。

ii. 开了分支预测后条件语句的预测错误数量显著下降，但是开启 BTB 之后预测错误数量反而上升，这是因为 BTB 开启后对一些原来 BPE 未预测的语句进行了预测。

iii. 对于 cache 而言，开启分支预测之后，cache 的访问次数显著减少，但是访问缓存的失效率均略有提升。次数减少可能是因为分支预测使用之后总的指令数减少，访存指令也减少。失效率上升可能是部分是因为总的访问数减少导致失效率上升，对于 L1_Dread 的影响较大可能是对应 C 程序的原因。

3）指令分布

a) 关于统计指令时的计数时机：因为输出函数 printf 会占用大量指令周期，如果将其执行周期算入到总的周期之中会干扰对反应卷积函数本身的指令分布统计，所以需要加入哨兵函数 guard_start 和 guard_end，从 PC 等于 guard_start 函数地址时开始计数，执行到 guard_end 函数时结束计数，因为编译器在编译时对地址分配的随机性，需要在跑完一次后进行反编译，看进入 guard_start 和 guard_end 函数时具体的 pc 值，将值同步到 tb.v 文件的宏定义中。

通过观察柱状图和饼状图可以发现：

i. memory access instructions 是占比最高的，占了总指令的 65.3%。其次是 arithmetic instructions，占比 25.3%,branch instructions 占比 6.56%，other instructions 占比 2.12%。

ii. 访存类指令数量多，占比大，执行周期长。所以访存的速度极大地影响

了 CPU 的性能，即内存墙的现象。通过引入缓存可以大大减少在访问内存时损耗的时间。

4、实验收获、存在问题、改进措施或建议等

实验收获：

1）整体了解了仿真平台的仿真流程，加深了对 soc 仿真的理解。

2）直观感受到了分支预测对减少 CPI，提高 CPU 性能的巨大作用

3）通过指令分布统计，深刻感受到程序中访存指令对 CPU 性能的影响，以及通过引入缓存来解决内存墙问题。

存在问题以及建议：

1） 对引入缓存与否对程序 CPI 的减少没有直观对比，希望可以跑一遍关闭或者无缓存情况下的仿真，来直观感受引入缓存的作用。

2）对仿真中调用的各个脚本以及具体的实现还是不太清楚，需要自己上网查阅资料。