2020 Computer Architecture Final Project Part3

Processing-in-Memory simulation in gem5 simulator

User Guide

1. Nouns abbreviation:

PIM: Processing-in-Memory

API: Application Programming Interface

2. Introduction:

Part1 shows how PIM works and its behavior, which puts logic computation near the memory to reduce the latency of data transmission between memory and processor. And PIM can also exploit high parallelism characteristics of memory itself to attain speedup.

Let's assume if we have a PIM-supported memory in our system, and we want to exploit PIM behavior to conduct speedup of whole program. TA will provide you a PIM function supported gem5 simulator, FM-Index C code which has introduced in part2 and a suite of PIM API as a communication interface between software and hardware.

Please understand the C code and the usage of PIM API. With the knowledge you have learned in Computer Architecture course, use PIM API to modify C code to exploit PIM characteristic. By considering the trade-off between transfer latency and computation time, make the program execution time as short as possible.

3. Files:

These files are all under *gem5* directory

(1) *FMIndex_golden_generate.cpp*: Generating golden answer to compare the answer correctness.

(2) *FMIndex.cpp*: You only need to modify this file in this FP, to make the execution way to PIM.

(3) *FMIndex.h*: header file of (2)

(4) *COsmall.txt*: input file of (1), (2)

(5) *golden.txt*: output file of (1)

(6) *pim_result.txt*: output file of (2)

(7) *CA_FMIndex_diff.py*: Compare if (4), (5) are the same or not, to verify the answer correctness.

(8) *T5_260.txt* : input file of (1), (2), the difference between this file and (3) is that TA will test for total_time using this file because it's a larger case.

Since it's a larger case, it will take pretty long time to run on gem5, so we recommend you to get familiar with this system from (3) first, then come to this case.

4. Compile command and Execution Order:

(1) Compile *FMIndex_golden_generate.cpp*: g++ FMIndex_golden_generate.cpp –o FMIndex_golden_generate.o

(2) Compile *FMIndex.cpp*: g++ -std=c++11 FMIndex.cpp -O2 -o FMIndex.o

(3) Execute *FMIndex_golden_generate.o*: ./ FMIndex_golden_generate.o

(4) Execute *FMIndex.o* on gem5: build/X86/gem5.opt configs/example/se.py -c FMIndex.o --cpu-type='TimingSimpleCPU' --cpu-clock='4GHz' --caches --l1d_size='128B'

(5) python3 CA_FMIndex_diff.py

Note: the default input file is "T5_260.txt", if you want to try smaller case like "COsmall.txt", you have to modify .cpp file yourself.

5. Execution result and Execution time calculation:

After executing *FMIndex.o* on gem5,

if the answer is correct, it will show "*compare with golden correct !!!!!!*" ,

if the answer is wrong, then it will show "*error!!!*".

If the answer is correct, you can find *stat.txt* file under *m5out* directory, where you can find final_tick and pim_cycle

Under *stat.txt* you can find the line like below

```
final_tick                                123348533000
```

```
system.cpu.workload.pim_cycles                        0
```

Which represents final_tick and pim_cycles respectively

Equation: total time = final_ticks + pim_cycles * 2,500

total time will be used to count your FP point.

6. PIM API Introduction:

This gem5 simulator can simulate system call, therefore we borrow the **pim** system call to help us trigger PIM operation.

pim definition is: **pim**(void* arg1, size_t arg2, size_t arg3, int flag)

Here, we put target data address into arg1, arg2, arg3.

And flag is used to choose the PIM operation.

7. PIM API operation:

pim format: **pim**(&arg1, &arg2, &arg3, flag)

According to the red colored arguments and fill with following table, you can conduct PIM operation.

flag	Operation	arg1	arg1	arg2	arg2	arg3	arg3	example	pim

			type		type		type		cycles
100	initialize	i[0]	int	k	int	x	UN	i[0] = k;	1
105	initialize	i[1]	char	x	UN	x	UN	i[1] = '$'	1

110	compare	out	int	ch	Char	x	UN	If(ch != '\n')
								out = 1;
								else out = 0;
									2

130	copy / query (size=1)	j[0]	char	i[0]	char	x	UN	j[0] = i[0]	1
131	copy / query (size=4)	j[0]	char	i[0]	char	x	UN	for(k=0;k<4;k++) j[i] = i[k];	3
132	copy / query (size=16)	j[0]	char	i[0]	char	x	UN	for(k=0;k<16;k++) j[i] = i[k];	9
133	copy / query (size=64)	j[0]	char	i[0]	char	x	UN	for(k=0;k<64;k++) j[i] = i[k];	27
133	copy / query (size=256)	j[0]	char	i[0]	char	x	UN	for(k=0;k<256;k++) j[i] = i[k];	81
140	multiply	k	int	i	int	j	int	k = i * j;	2
141	add	k	int	i	int	j	int	k = i + j;	1
150	double switch	i[0]	char	j[0]	char	x	UN	switch(i[0], j[0]); switch(i[1], j[1]);	4
151	switch	i[0]	char	j[0]	char	x	UN	switch(i[0], j[0]);	2

160	select and add	out[0]	int	ch	char	x	UN	if(ch == 'A')
								out[0]++;
								else if(ch=='C')
								out[1]++;
								else if(ch=='G')
								out[2]++;
								else if(ch=='T')
								out[3]++;
									10

x: random number (TA recommend you put in 0)

UN: Un-Need

Notice: each arg has it's corresponding data type. Program will run error if you put in wrong data type data

8. PIM API example:

Assume there is a line of C code:

i[0] = k;

And i[0] is an integer, k is an integer

If I want to transfer this code into PIM operation, I can rewrite this line to:

```
pim(&i[0], &k, 0, 100);
```

9. hint

Most flag can replace some code in the FMIndex.cpp, but the performance might not be better after you replace it.

Some cout code is not recommended to delete, in case that it affect compiler. But you can add cout yourself to do program profiling.

You can also optimize C code to make performance better, as long as program gets correct result.

Compiler may cause wrong logic sometime, please retry it and verify.

10.requirement

FMIndex.cpp must get correct answer

You must use at least 2 PIM operation

11.contact

If you have any issue, you can use new e3 mailing function or e-mail to following TA e-mail address.

Po-Shen Kuo (kuoposhen@gmail.com)