2.3. Homework Submission

Your writeup should follow the writeup guidelines. Your writeup should include the following:

1. **Identify**
   a. For the application you downloaded in hw2/assignment, describe the operation performed on the input data by each function and why you might want to perform the operation (3 lines for each of Scale, Filter, Differentiate, Compress).
2. **Measure**
   Build a table similar to the following, and populate it when answering the following questions:

Functions	**Average Latency (** $T_{measured\_avg}$ **ns)**	**% of total latency**	**Average Latency (** $T_{measured\_avg}$ **cycles)**
Scale			
Filter_horizontal			
Filter_vertical			
Differentiate			
Compress			

Table 2.1 Example Profile

   a. Report the average latencies (i.e. time to execute one call of the function) of `Scale`, `Filter_horizontal`, `Filter_vertical`, `Differentiate`, `Compress` in nanoseconds. For this, you will need to instrument the code (refer to Instrumentation-based Profiling with Timers in the profiling tutorial).
   Write a Makefile (refer to the profiling tutorial) and use `-O2` optimization (we will explore optimization levels more in HW4).
   b. Report the percentage of time each function (`Scale`, `Filter_horizontal`, `Filter_vertical`, `Differentiate`, `Compress`) takes in your program. For this, you will need to use `gprof` (refer to Profiling using GNU gprof in the profiling tutorial).
   c. Report the latencies of 2a and 2b in cycles. Assume that each operation takes one clock cycle at 4.7 GHz.
3. **Analyze**
   a. Which function from Table 2.1 has the highest latency? (1 line)
   b. Assuming that `LOOP2` of `Filter_horizontal` is unrolled completely, draw a Data Flow Graph (DFG) of the body of the loop over `X`. You may ignore index computations (i.e. only include the compute operations (multiply, accumulate and shift) that work on `Input`).
   Index computations are operations used to calculate the index to be used with a pointer to get an element. For e.g. `4*i` in `x[4*i]` is an index computation.
   c. Assuming that the operations in the DFG execute sequentially, count the total number of compute operations. Using this number, estimate the average latency in cycles of `Filter_horizontal`. Assume that each operation takes one clock cycle at 4.7 GHz.

> **Hint**
>
> This should be a simple calculation, and it won't necessarily match what you found in Table 2.1; we'll be working on that in subsequent questions.
> Just like 3b, we want you to focus on parts of the function that execute more times than others (i.e. the multiply, shift and accumulate). Hence, you are asked to not estimate the impact of the parts that contribute little to runtime. We'll get a better picture of that when we look at the assembly code.

d. If you would apply a $2\times$ speedup to one of the stages (`Scale`, `Filter_horizontal`, `Filter_vertical`, `Differentiate`, `Compress`) which one would you choose to obtain the best overall performance? (1 line)
e. Use Amdahl's Law to determine the highest overall application speedup that one could achieve assuming you accelerate the one stage that you identified above. You don't have to restrict yourself to this platform. (1 line)
f. Assuming a platform that has unlimited resources and you are free to exploit associativity for mathematical operations, draw the DFG with the lowest critical path delay for the same unrolled loop body (Part 3b) as before.
g. Determine the critical path length (i.e. assume instructions can execute in the same cycle) of the same unrolled loop (Part 3f) in terms of compute operations.
h. Assuming a platform that has 4 multipliers, 2 adders, and a shifter, report the resource capacity lower bound for the same loop body (Part 3b) as before. (4 lines)

4. **Refine**
   As you hopefully noticed, our model of using a DFG and counting compute operations did not estimate `Filter_horizontal` very accurately in Part 3c. Let's see whether we can improve it.
   First, we'll build a table similar to the following, which records the total number of cycles for each instruction:

Assembly Instructions	Annotation	Number of function calls (N)	Number of cycles per call (T)	Number of instructions executed in parallel (N_{par})	Total number of cycles ($\frac{N}{N_{par}} \times T$)
`add w0, w0, 1`	addition(s) for array indexing	100	1	13	8
...	...	...	...	...	...

Table 2.2 Example Table

We will then group these instructions into three bins—*fast mem*, *slow mem* and *non-memory*, and develop the following model for the runtime of this filter computation:

$$T_{filter\_h\_measured\_avg} = \frac{N_{non\_memory}}{N_{par}} \times T_{cycle\_non\_memory} + \frac{N_{fast\_mem}}{N_{par}} \times T_{cycle\_fast\_mem} + N_{slow\_mem} \times T_{cycle\_slow\_mem} \quad (2.1)$$

The real model is still more complicated than this, but this is a first-order model that can start helping us reason about the performance of the computation including memory access.

a. Make a table like Table 2.2 and add all the instructions in the loop body (not the instructions that setup the loop, but the instruction that are executed on each trip through the loop) of the innermost loop. (Do not assume that it is unrolled this time.) You can see the instructions by:
   - Compiling your code with: `g++ -Wall -S -O2 -c Filter.c -o Filter.s`, and
   - Opening the generated `Filter.s` assembly file.

   Which of these instructions are the compute operations you identified in 3c?

b. Annotate each instruction with one of the descriptions below as appropriate, and add to Table 2.2. You will not be able to annotate some instructions. Don't worry, that is part of what the question is setting up.
    i. multiplication for array indexing
    ii. addition(s) for array indexing
    iii. multiplication of coefficient with input
    iv. addition to Sum
    v. reads from arrays
    vi. increment of loop variable
    vii. comparison of loop variable to loop limit
    viii. branch to top of loop

    You can use your C code to infer the annotations for the instructions.

c. After identifying the called-out instructions above, there are additional assembly instructions. What type of instructions are these? What do these assembly instructions do? Provide a 1 line (or less) description for each type of instruction identified, and use them to annotate the additional instructions in your table.

d. Fill in the rest of the table by determining the number of function calls for each instruction. Assume that one instruction completes per cycle. Also assume that 13 instructions are executed at the same time. Using your table, estimate the latency ($T_{filter\_h\_analytical}$) of the function in cycles. (1 line)

> **Note**
>
> The model here is:
>
> $$T_{filter\_h\_analytical} = \frac{N_{instr}}{N_{par}} \times T_{cycle}$$
>
> where $T_{cycle} = 1$ and $N_{par} = 13$.

e. Now assume that only the non-memory instructions identified in Table 2.2 complete in one cycle, and also assume that the multiple execution of instructions ($N_{par} = 13$) only applies to non-memory instructions, estimate the average latency (T_{cycle_memory}) in cycles of the memory operations. (3 lines)

> **Hint**
>
> Use the measured latency of `Filter_horizontal` from Table 2.1 and solve for T_{cycle_memory}.

> **Note**
>
> Refining from 4d, the model here is:
>
> $$N_{instr} = N_{non\_memory} + N_{memory}$$
>
> $$T_{filter\_h\_measured\_avg} = \frac{N_{non\_memory}}{N_{par}} \times T_{cycle} + N_{memory} \times T_{cycle\_memory}$$
>
> where $T_{cycle} = 1$ and $N_{par} = 13$.

f. For the identified memory operations, how many of these loads and store cycles are to memory locations ***not*** loaded during this invocation of `Filter_horizontal`)?

> **Hint**
>
> You are finding out N_{slow_mem} of equation (2.1)

Add a column to your instruction table and identify the fraction of time the specified instruction is to a new memory location not previously encountered during a call to the function.

g. Assuming memory locations that have already been loaded during a call to this function (Part 4f) also take a single cycle and multiple execution of instructions ($N_{par} = 13$) also applies to them, what is the average number of cycles ($T_{cycle\_slow\_mem}$) for the remaining loads? This will require you to use the fraction you added to the table in the previous question. (3 lines)

> **Note**
>
> Refining from 4e, this gives us the model for the runtime of this filter computation:
>
> $$N_{memory} = N_{fast\_mem} + N_{slow\_mem}$$
>
> $$T_{filter\_h\_measured\_avg} = \frac{N_{non\_memory}}{N_{par}} \times T_{cycle} + \frac{N_{fast\_mem}}{N_{par}} \times T_{cycle} + N_{slow\_mem} \times T_{cycle\_slow\_mem}$$
>
> where $T_{cycle} = 1$ and $N_{par} = 13$.

5. **Coding**

a. Implement the `hash_func` and `cdc` functions from the following Python code in C/C++. You can find the starter code at `hw2/cdc/cdc.cpp`. You are free to use C/C++ standard library data structures as you see fit.

```
win_size = 16
prime = 3
modulus = 256
target = 0


def hash_func(input, pos):
    hash = 0
    for i in range(0, win_size):
        hash += ord(input[pos+win_size-1-i])*(pow(prime, i+1))
    return hash


def cdc(buff, buff_size):
    for i in range(win_size, buff_size-win_size):
        if((hash_func(buff, i) % modulus)) == target:
            print(i)


def test_cdc(filename):
    with open(filename) as f:
        buff = f.read()
        cdc(buff, len(buff))


test_cdc("prince.txt")
```

Verify that your program outputs the following:

> **Output**
>
> 299 366 367 838 1001 1263 2141 2283 2660 2708 2748 2820 3143 3210 3211 3682 3845 4107 4985 5127 5504 5552 5592 5664 5987 6054 6055 6526 6689 6951 7829 7971 8348 8396 8436 8508 8831 8898 8899 9370 9533 9795 10673 10815 11192 11240 11280 11352 11675 11742 11743 12214 12377 12639 13517 13659 14036 14084 14124 14196

> **Tip**
>
> - To verify, you can first create a text file with the output, e.g. `golden.txt`
> - You can then write the output of your program to another text file buy redirecting the output in the command line, e.g. `./cdc > out.txt`
> - You can then use `diff` which will show differences in values if any, e.g. `diff out.txt golden.txt`

b. It is more efficient to not recompute the whole hash at every window. Convince yourself that the next hash computation can be expressed as:

```
hash_func(input, pos+1) = hash_func(input, pos)*prime - input[pos]*pow(prime, win_size+1)
+ input[pos+win_size]*prime
```

Develop a second, revised `cdc` function that uses this observation to reduce the work. Verify that your program is producing the same outputs with the changes.

c. Time the two `cdc` implementations and compare.

> **Tip**
>
> - Work together with your partner!
> - Read the following resources to gain more context about the code:
>
> Content-Defined Chunking (Rabin Fingerprint)
>
> - https://moinakg.wordpress.com/tag/rabin-fingerprint/
> - https://en.wikipedia.org/wiki/Rabin_fingerprint

2.3.1. Deliverables

In summary, upload the following in their respective links in canvas:

- a tarball containing your instrumented code and Makefile with targets for compilation and running `gprof`.

> **Quick linux commands for tar files**

- a tarball containing your two `cdc` implementations code and Makefile to compile it.
- writeup in pdf.