Application Acceleration with High-Level Synthesis (11020EE521800)

Lab B – Bloom Filter

110061639 林致佑

Algorithm introduction:

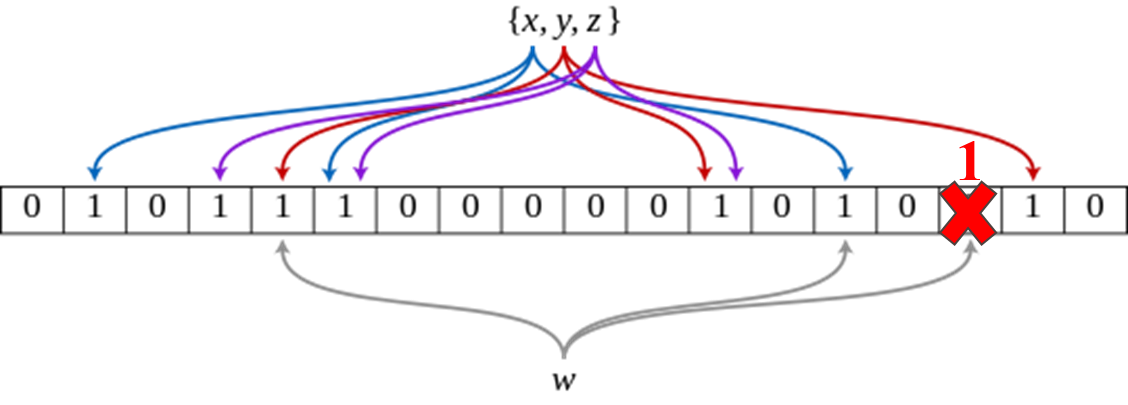
A Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. For example, checking availability of username is set membership problem, where the set is the list of all registered username. The price we pay for efficiency is that it is probabilistic in nature that means, there might be some False Positive results. False positive means, it might tell that given username is already taken but actually it’s not.

The base data structure of a Bloom filter is a Bit Vector. Here is a small one use to demonstrate:

A picture containing light

Description automatically generated

An example of a Bloom filter, representing the set {x, y, z}. The colored arrows show the positions in the bit array that each set element is mapped to. The element w is not in the set {x, y, z} , because it hashes to one bit-array position containing 0.



Now, we change the one bit array contains all 1s after element w hashed and mapped to. We can conclude that element w is a false-postive match.

In general, a Bloom filter application has use cases in data analytics, such as browsing through unstructured email and text file data to identify the documents that are closely associated with a specific user and send notifications accordingly.

Here, I listed some common applications/Usages which adopt Bloom Filter.

* Weak password detection
* Internet Cache Protocol
* Safe browsing in Google Chrome
* Tinder suggestion
* Yahoo email contact checking

Code analysis:

1. Hash function

Graphical user interface, text, application, email

Description automatically generated

The computational complexity is the number of basic computing operations required to execute the function. The compute of the hash for a single word ID consists of four XORs, three arithmetic shifts, and two multiplication operations. A shift of 1-bit in an arithmetic shift operation takes one clock cycle on the CPU. The three arithmetic operations shift a total of 44-bits (when len=3 in the above code) to compute the hash that requires 44 clock cycles just to shift the bits on the host CPU.

To implement hardware acceleration on the FPGA, we can create an accelerator that will shift the data by an arbitrary number of bits in a single clock cycle. Furthermore, FPGA also has dedicated DSP units that perform multiplications faster than the CPU. Therefore, this function is a suitable candidate for FPGA acceleration.

1. For loop for hash functionality

Graphical user interface, text

Description automatically generated

From the code structure, we can see nested for loop (2), one is looping over the number of documents, and another is looping over the number of words in the document. Here, we compute two hash outputs for each word in all the documents and creating output flags accordingly.

We can notice a few things from above code. First, we already determined that the hash function(MurmurHash2()) is a good candidate for acceleration on the FPGA. Second, the hash (MurmurHash2()) function with one word is independent of other words and can be done in parallel which improves the execution time. Lastly, the algorithm sequentially accesses to the input\_doc\_words array. This is an important property because when implemented in the FPGA, it allows for very efficient accesses to the HBM. Therefore, the code section is suitable candidate for FPGA acceleration.

1. Profile computing score

Text

Description automatically generated

From the code structure, we can see nested for loop (2) similar to the previous code block “for loop for hash functionality”. In here, we can see the compute score requires one memory access to profile\_weights, one accumulation, and one multiplication operation. Also, the memory accesses are random because they depend on the word\_id.

However, the non-sequential accesses are big performance bottlenecks. Because accesses to the profile\_weights array are random, implementing this function on the FPGA would not provide much performance benefit.

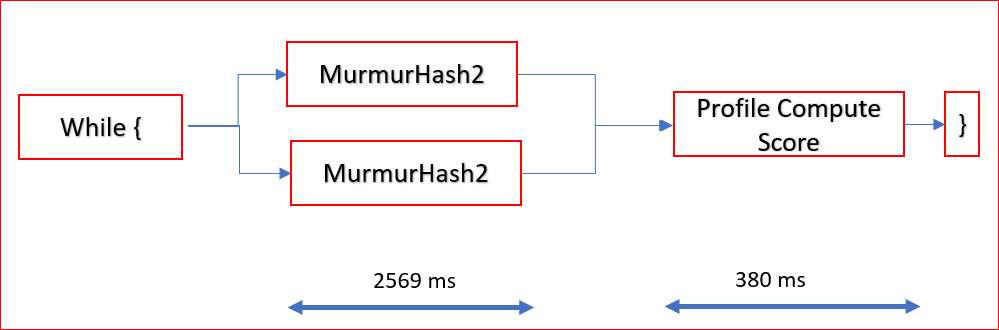
1. Timing analysis:

When doing software profiling (running application on CPU), we can see the execution time is mainly on computing hash (>85%) and the score processing time only takes around 10% of overall execution time.

Graphical user interface, text, application

Description automatically generated

Furthermore, with the code analysis, we can establish the realistic goal for the overall application. That is, optimize the hash function with FPGA acceleration (match profile compute score latency) so that the latency of hash function can be hided and establish a pipelined conceptual application.



1. Conceptual Application

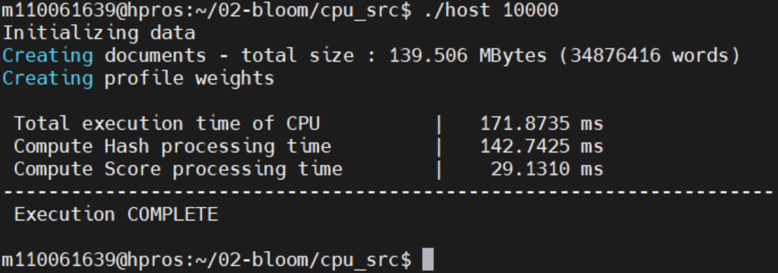
Transferring words from the host to CPU, Compute on FPGA, transferring words from the FPGA to CPU all can be executed in parallel. The CPU starts to calculate the profile score as soon as the flags are received; essentially, the profile score on the CPU can also start calculations in parallel. With the pipelining as shown above, the latency of the Compute on FPGA will become invisible.

Diagram

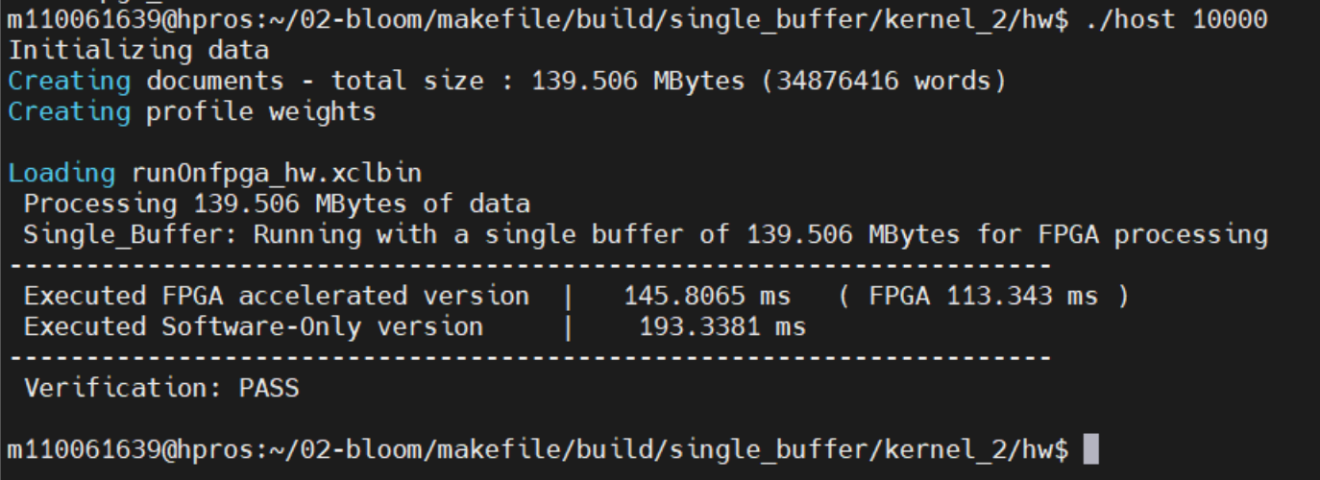
Description automatically generated

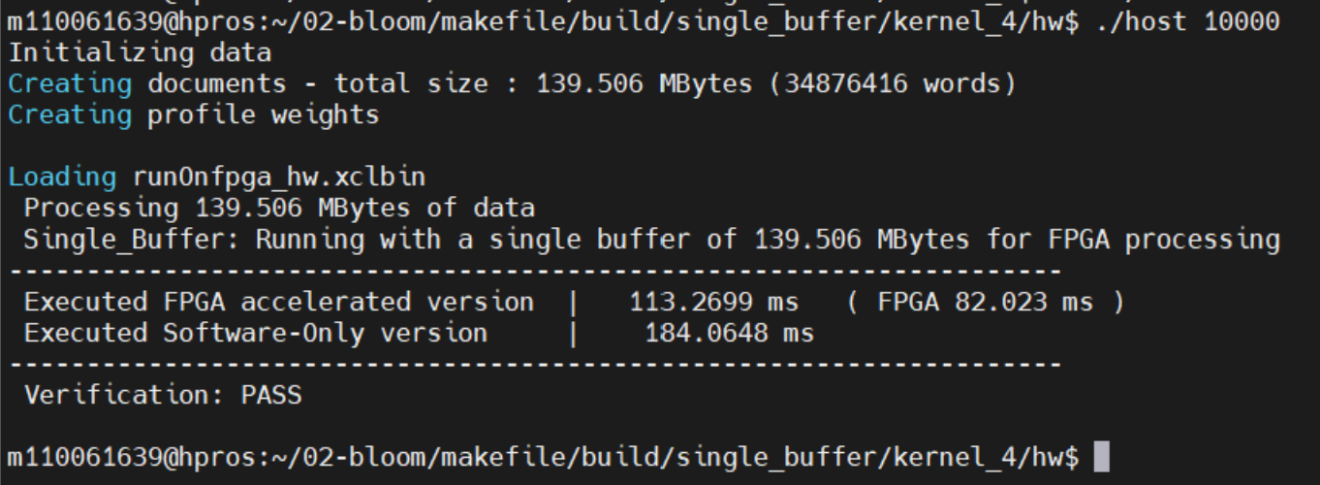
1. Performance analysis:

I run the application on actual hardware with different number of kernel (PF=2, 4, 8, 16). Below is the baseline which means running application on CPU (SW version).



On CPU only version, the throughput is 139.506 MB / 171.8735 ms = 811.678356 MBps.



Configure the FPGA to have two concurrent run functions (kernel/PF=2) and the ideal scalable throughput will be 811.678356 MBps x2 = 1623.4 MBps. However, the actual throughput is 139.506 MB / 145.8065 ms = 956.788621 MBps. 

Configure the FPGA to have two concurrent run functions (kernel/PF=4) and the ideal scalable throughput will be 811.678356 MBps x4 = 3246.7 MBps. However, the actual throughput is 139.506 MB / 113.2699 ms = 1231.6 MBps. Text

Description automatically generated

Configure the FPGA to have two concurrent run functions (kernel/PF=8) and the ideal scalable throughput will be 811.678356 MBps x8 = 3246.7 MBps. However, the actual throughput is 139.506 MB / 99.2931 ms = 1404.99 MBps.

Text

Description automatically generated

Configure the FPGA to have two concurrent run functions (kernel/PF=16) and the ideal scalable throughput will be 811.678356 MBps x16 = 12986.8MBps. However, the actual throughput is 139.506 MB / 98.8838 ms = 1410.8 MBps.

We can now use the actual hardware accelerated throughput then normalize to CPU throughput generate the speed up comparison in below graph.

We can notice that the speed up is growing from kernel/PF 2-8. However, it saturated after kernel/PF 8. That is, we may not need to use kernel/PF 16 to accelerate our hash function performance due to the hardware resource overhead when transferring data.

1. Utilization analysis:Table

   Description automatically generated

Kernel / PF = 8 utilization

Table

Description automatically generated

Kernel / PF = 16 utilization

Let’s now compare kernel / PF = 8 & 16 since they have similar performance / throughput. For the DSP and BRAM used resource, kernel / PF = 16 almost used doubled resource. So we may not need to use kernel/PF 16 to accelerate our hash function performance and save some hardware resource.

1. GitHub Repo:

<https://github.com/ChihyuLin0211/Lab_B_Vitis_Tutorials_02_Bloom>

1. Reference:

<https://github.com/Xilinx/Vitis-Tutorials/tree/2021.2/Hardware_Acceleration/Design_Tutorials/02-bloom>

<https://github.com/bol-edu/2021-fall-ntu/tree/main/Lab_B_Vitis_Tutorials/Bloom_Filter/HLS_2021_FALL_LABB-master>

<https://support.xilinx.com/s/question/0D52E00006yn3ZpSAI/i-get-an-stdbadalloc-error-when-allocating-a-buffer-in-global-memory-for-the-example-in-01rtlkernelworkflow?language=en_US>

<https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/HBM-Configuration-and-Use>

U50 HBM

Graphical user interface, text, application

Description automatically generated