

# **Bloom Filter**

## **Lab B**

劉彥甫

R09921132

Application Acceleration with High-Level Synthesis (EEE 5060)

INSTRUCTOR: Professor Jiin Lai

2021年10月26日

Bloom Filter	1
Lab B	1
What is a Bloom filter?	3
Code analysis	6
Hash function	6
For Loop for Hash functionality	7
Profile computing score	8
Overall design	9
The performance evaluation	10
The issue on porting from DDR to HBM memory interface	10
Does the function actually speed up?	11
How does application timeline look like?	13
GitHub	14
Reference	14

# What is a Bloom filter?

Document filtering is the process in which a system monitors a stream of input documents, classifies them according to their content, and selects documents relevant to a specific target. Document filtering is used extensively in the everyday database querying, retrieval, and analysis of information that helps to identify the relevant document.

The user's interest is represented by a search array, which contains words of interest and has a weight associated with it, indicating the relevance of the word. Naive implementation access the database for every word in the documents to check if a word is in the document or not, if the word is present, retrieve the weight of the word. A more optimized approach uses a space-efficient bloom filter in your cache that can report whether a word is present in the database, which reduces the number of expensive database queries.

The bloom filter uses a hash table-based data structure that determines when an element is present in the document. False-positive matches are possible for this approach, but false-negative matches are not; in other words, a query returns either "possibly in the set" or "definitely not in the set". The advantage of using Bloom filter is that is space-efficient and reduced the number of expensive database queries for data that is not in the set. In most use, bloom filter is used as first-pass document filtering. A bloom filter is also useful in applications to implement search engines and database management systems, such as Cassandra, where it can reduce the number of disk queries and increase performance.

The following figure shows a bloom filter example representing the set  $\{x, y, z\}$ .

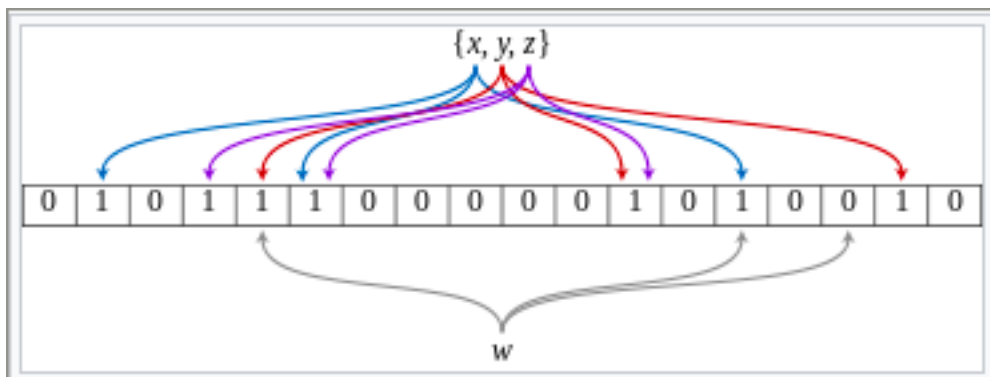


Fig. 1 Bloom filter hashed array

The colored arrows show the position in the bit array that each element in the set is mapped.

The element  $w$ , the element user queries, is not in the set  $\{x, y, z\}$  because it hashes to a one bit-array position containing 0.

The number of elements is 18 and the number of hash functions computed for each element is 3.

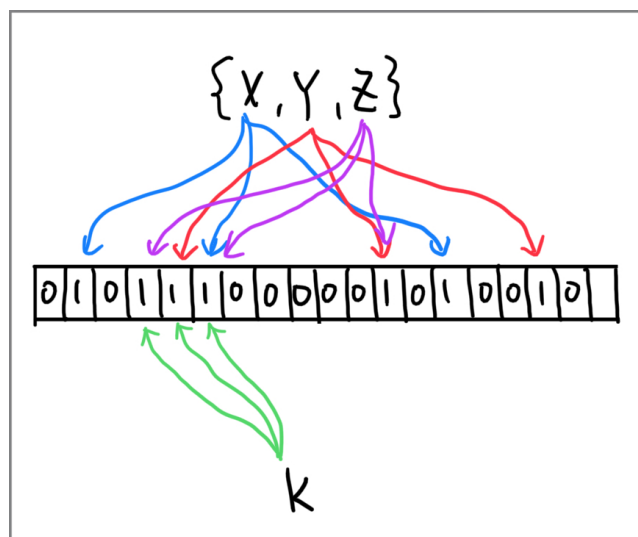


Fig. 1-1 False-positive example

In figure 1-1, the element  $k$  is not in the set  $\{x,y,z\}$  but it hashes to one bit-array position that does not contain 0, which makes element  $k$  a false-positive case.

# Code analysis

## Hash function

The first important function is the hash function, here is the "MurmurHash2" function:

```
unsigned int MurmurHash2 ( const void * key, int len, unsigned int seed )
{
    const unsigned int m = 0x5bd1e995;

    // Initialize the hash to a 'random' value
    unsigned int h = seed ^ len;

    // Mix 4 bytes at a time into the hash
    const unsigned char * data = (const unsigned char *)key;

    switch(len)
    {
        case 3: h ^= data[2] << 16;
        case 2: h ^= data[1] << 8;
        case 1: h ^= data[0];
        h *= m;
    };

    // Do a few final mixes of the hash to ensure the last few
    // bytes are well-incorporated.
    h ^= h >> 13;
    h *= m;
    h ^= h >> 15;

    return h;
}
```

The computational complexity is as following part.

The compute of the hash for a single word ID consists of 4 XORs, 3 arithmetic shifts, and 2 multiplication operations. A shift of 1-bit in an arithmetic shift operation takes one clock cycle on the CPU. The 3 arithmetic operations

shift a total of 44-bit to compute which requires 44 clock cycles to complete. On the other hand, the arbitrary number of bits can be shifted in a single cycle if you want.

Even though CPU runs at 8 times frequency faster than FPGA, the arithmetic shift and multiplication operations can perform faster on FPGA with customizable computing architecture.

## For Loop for Hash functionality

```
// Compute output flags based on hash function output for the words in all documents
for(unsigned int doc=0;doc<total_num_docs;doc++)
{
    profile_score[doc] = 0.0;
    unsigned int size = doc_sizes[doc];

    for (unsigned i = 0; i < size ; i++)
    {
        unsigned curr_entry = input_doc_words[size_offset+i];
        unsigned word_id = curr_entry >> 8;
        unsigned hash_pu = MurmurHash2( &word_id , 3,1);
        unsigned hash_lu = MurmurHash2( &word_id , 3,5);
        bool doc_end = (word_id==docTag);
        unsigned hash1 = hash_pu&hash_bloom;
        bool inh1 = (!doc_end) && (bloom_filter[ hash1 >> 5 ] & ( 1 << (hash1 & 0x1f)));
        unsigned hash2 = (hash_pu+hash_lu)&hash_bloom;
        bool inh2 = (!doc_end) && (bloom_filter[ hash2 >> 5 ] & ( 1 << (hash2 & 0x1f)));

        if (inh1 && inh2){
            inh_flags[size_offset+i]=1;
        } else {
            inh_flags[size_offset+i]=0;
        }
    }

    size_offset+=size;
}
```

First of all, if this code section is written directly into the FPGA OpenCL kernel, it will be SWI(Single-Work-item) execution. From this code, you observed the following things:

1. You are computing two hash outputs for each word in all the documents and creating output flags accordingly.
2. You already determined that the hash function is a good candidate for acceleration on FPGA.
3. The algorithm sequentially accesses the input\_doc\_words array. This is a bonus property because when implemented in the FPGA, it allows for very efficient access to the HBM. If access the memory concurrently, it will need to implement an "Atomic operation" interface that increases access latency.

## Profile computing score

```
for(unsigned int doc=0, n=0; doc<total_num_docs;doc++)
{
    profile_score[doc] = 0.0;
    unsigned int size = doc_sizes[doc];

    for (unsigned i = 0; i < size ; i++,n++)
    {
        if (inh_flags[n])
        {
            unsigned curr_entry = input_doc_words[n];
            unsigned frequency = curr_entry & 0x00ff;
            unsigned word_id = curr_entry >> 8;
            profile_score[doc]+= profile_weights[word_id] * (unsigned
long)frequency;
        }
    }
}
```

The above function is for score computing, it requires one memory access to "profile\_weights", and one MAC. The physical size of the "profile\_weight"



array is 128MB and must be stored in the HBM memory on the FPGA. The code originally is implemented on u200 which uses DDR memory instead of HBM. The issue of the difference between DDR and HBM will be further discussed later.

This code only takes about 11% of the entire run-time, you can keep it on CPU if you like.

## Overall design

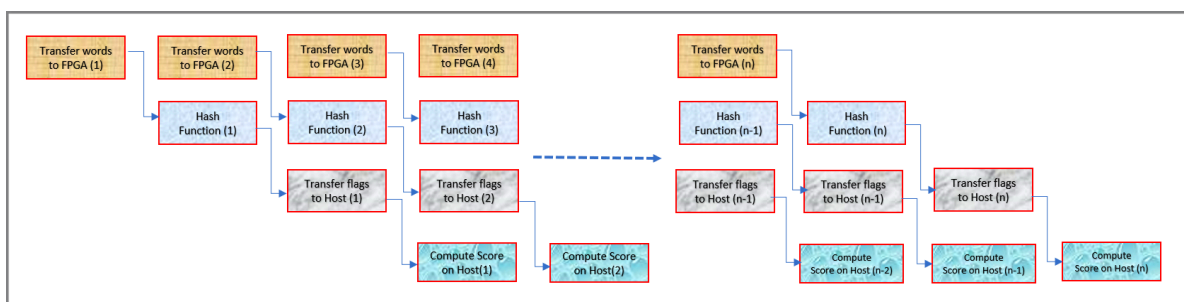
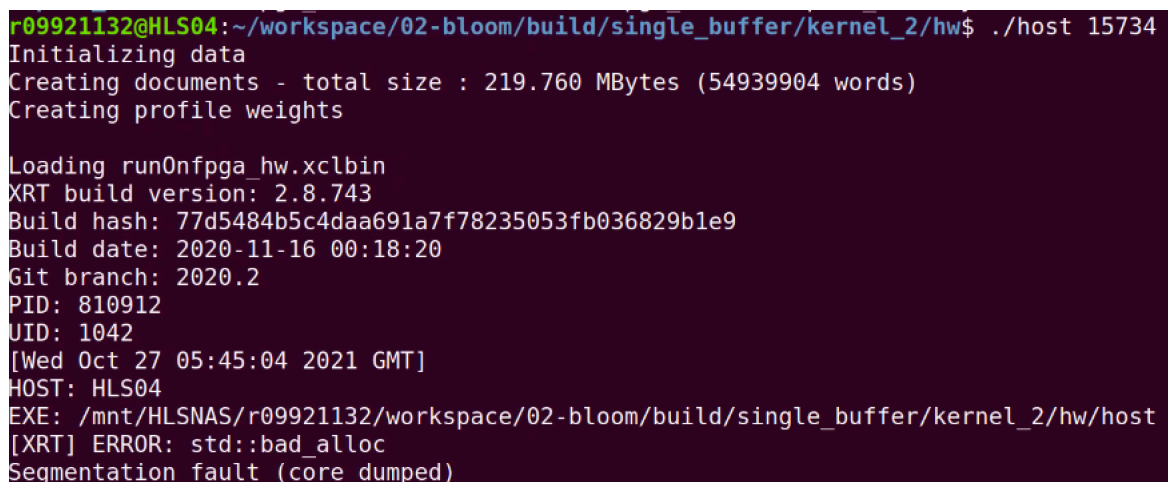


Fig. 2 Overall design

# The performance evaluation

## The issue on porting from DDR to HBM memory interface

The original code allocates the memory bank [0:1] and one to store profile weight, 128MB, and one to store the documents (one document contains around 3491 words which size is roughly 13.6 KB). The one bank of DDR on u200 FPGA is 16GB, however, HBM 1 Pseudo channel is 256MB. If you allocate memory space more than 256MB for documents, the allocation will fail and cause segmentation fault, in figure 3.

A terminal window with a dark background and light-colored text. The text shows the execution of a program on a host. It starts with 'Initializing data', 'Creating documents - total size : 219.760 MBytes (54939904 words)', and 'Creating profile weights'. Then it shows 'Loading run0nfpga\_hw.xclbin', 'XRT build version: 2.8.743', 'Build hash: 77d5484b5c4daa691a7f78235053fb036829b1e9', 'Build date: 2020-11-16 00:18:20', 'Git branch: 2020.2', 'PID: 810912', 'UID: 1042', '[Wed Oct 27 05:45:04 2021 GMT]', 'HOST: HLS04', 'EXE: /mnt/HLSNAS/r09921132/workspace/02-bloom/build/single\_buffer/kernel\_2/hw/host', '[XRT] ERROR: std::bad\_alloc', and finally 'Segmentation fault (core dumped)'.

```
r09921132@HLS04:~/workspace/02-bloom/build/single_buffer/kernel_2/hw$ ./host 15734
Initializing data
Creating documents - total size : 219.760 MBytes (54939904 words)
Creating profile weights

Loading run0nfpga_hw.xclbin
XRT build version: 2.8.743
Build hash: 77d5484b5c4daa691a7f78235053fb036829b1e9
Build date: 2020-11-16 00:18:20
Git branch: 2020.2
PID: 810912
UID: 1042
[Wed Oct 27 05:45:04 2021 GMT]
HOST: HLS04
EXE: /mnt/HLSNAS/r09921132/workspace/02-bloom/build/single_buffer/kernel_2/hw/host
[XRT] ERROR: std::bad_alloc
Segmentation fault (core dumped)
```

Fig. 3 Segmentation fault

Beside this question, there is some file need to be changed from DDR to HBM, the detailed can be found in my GitHub repository.

## Does the function actually speed up?

```
r09921132@HLS04:~/workspace/02-bloom/cpu_src$ make run NUM_DOCS=15374
rm -rf temp_dir log_dir report_dir *log host runOnfpga* *.csv *summary .run .Xil vitis* *jou xilinx* gpofresult.tx
t gmon.out
g++ -D__USE_XOPEN2K8 -D__USE_XOPEN2K8 \
-I. \
-O3 -Wall -fmessage-length=0 -std=c++11\
./compute_score_host.cpp \
./MurmurHash2.c \
./main.cpp \
-o ./host
./host 15374
Initializing data
Creating documents - total size : 214.683 MBytes (53670656 words)
Creating profile weights

Total execution time of CPU      | 266.7210 ms
Compute Hash processing time     | 223.6950 ms
Compute Score processing time    | 43.0260 ms
-----
Execution COMPLETE
```

Fig. 4 CPU execution time

In figure 4, on CPU only version, the throughput is  $214.683 \text{ MB} / 0.2667210 \text{ s} = 804.8972 \text{ MB/s}$ .

```
r09921132@HLS04:~/workspace/02-bloom/build/single_buffer/kernel_1/hw$ ./host 15374
Initializing data
Creating documents - total size : 214.683 MBytes (53670656 words)
Creating profile weights

Loading runOnfpga_hw.xclbin
Error: profiling will not be available. Reason: Could not open device file.
Error: profiling will not be available. Reason: Could not open device file.
Error: profiling will not be available. Reason: Could not open device file.
Error: profiling will not be available. Reason: Could not open device file.
Error: profiling will not be available. Reason: Could not open device file.
Processing 214.683 MBytes of data
Single_Buffer: Running with a single buffer of 214.683 MBytes for FPGA processing
-----
Executed FPGA accelerated version | 263.8967 ms ( FPGA 210.985 ms )
Executed Software-Only version   | 290.3512 ms
-----
Verification: PASS
```

Fig. 4-1 FPGA one parallel function configuration

In figure 4-1, if we config only one hardware function in FPGA and the throughput is  $214.683 \text{ MB} / 0.2638967 \text{ s} = 813.5115 \text{ MB/s}$ , which is already a speed up version with little difference. Also, if we treat CPU as one kernel function, the execution time will be larger than the execution time in figure 4.

```

r09921132@HLS04:~/workspace/02-bloom/build/single_buffer/kernel_2/hw$ ./host 15374
Initializing data
Creating documents - total size : 214.683 MBytes (53670656 words)
Creating profile weights

Loading runOnfpga_hw.xclbin
Processing 214.683 MBytes of data
Single_Buffer: Running with a single buffer of 214.683 MBytes for FPGA processing
-----
Executed FPGA accelerated version | 185.0109 ms ( FPGA 121.446 ms )
Executed Software-Only version   | 284.4367 ms
-----
Verification: PASS

```

Fig. 4-2 FPGA two parallel functions configuration

In figure 4-2, I config the FPGA to have two concurrent run functions and the ideal scalable throughput will be  $813.5115 \text{ MB/s} \times 2 = 1627 \text{ MB/s}$ . That is, the actual throughput is  $214.683 \text{ MB} / 0.1850109 \text{ s} = 1160.4 \text{ MB /s}$ .

```

r09921132@HLS04:~/workspace/02-bloom/build/single_buffer/kernel_4/hw$ ./host 15374 100
Initializing data
Creating documents - total size : 214.691 MBytes (53672704 words)
Creating profile weights

Loading runOnfpga_hw.xclbin
Processing 214.691 MBytes of data
Single_Buffer: Running with a single buffer of 214.691 MBytes for FPGA processing
-----
Executed FPGA accelerated version | 136.6819 ms ( FPGA 76.389 ms )
Executed Software-Only version   | 284.9723 ms
-----
Verification: PASS

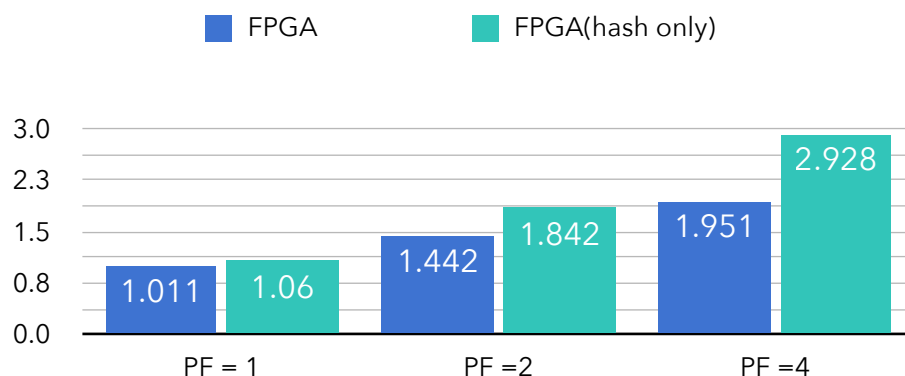
```

Fig. 4-3 FPGA four parallel functions configuration

In figure 4-3, I config the FPGA to have four concurrent run functions and the ideal scalable throughput will be  $813.5115 \text{ MB/s} \times 4 = 3254 \text{ MB /s}$ . But the actual throughput is  $214.683 \text{ MB} / 0.1366819 \text{ s} = 1570 \text{ MB /s}$ .

You should be noticed that we keep the computing score part in CPU, If we only consider the part on the hash function. The throughputs is following:

1. CPU:  $214.683 \text{ MB} / 0.223695 = 959.7 \text{ MB/s}$
2. FPGA with one parallel function:  $214.683 \text{ MB} / 0.210985 \text{ s} = 1017.5 \text{ MB/s}$
3. FPGA with one parallel function:  $214.683 \text{ MB} / 0.121446 \text{ s} = 1767.7 \text{ MB/s}$
4. FPGA with one parallel function:  $214.683 \text{ MB} / 0.076389 \text{ s} = 2810.4 \text{ MB/s}$



The speed up normalized to CPU

# How does application timeline look like?

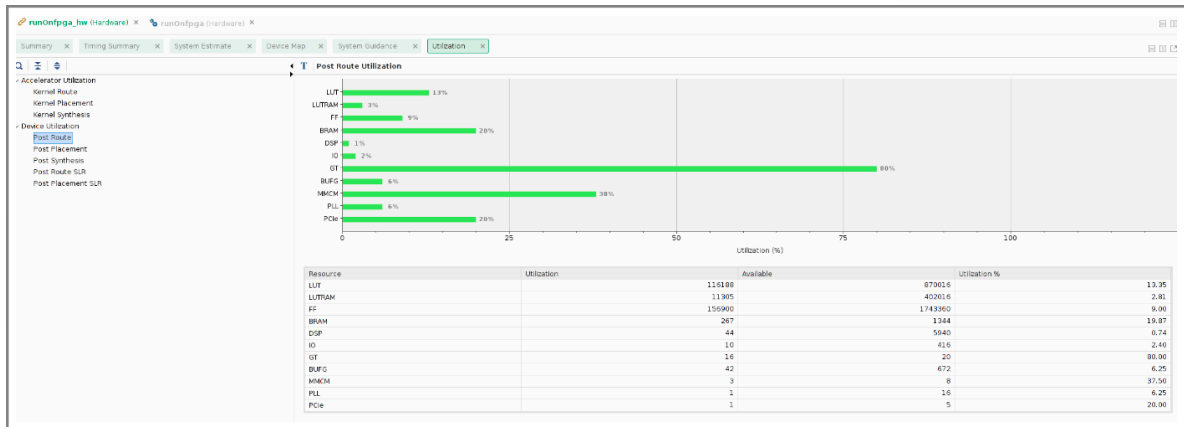


Fig. 5 PF=4 resource utilization

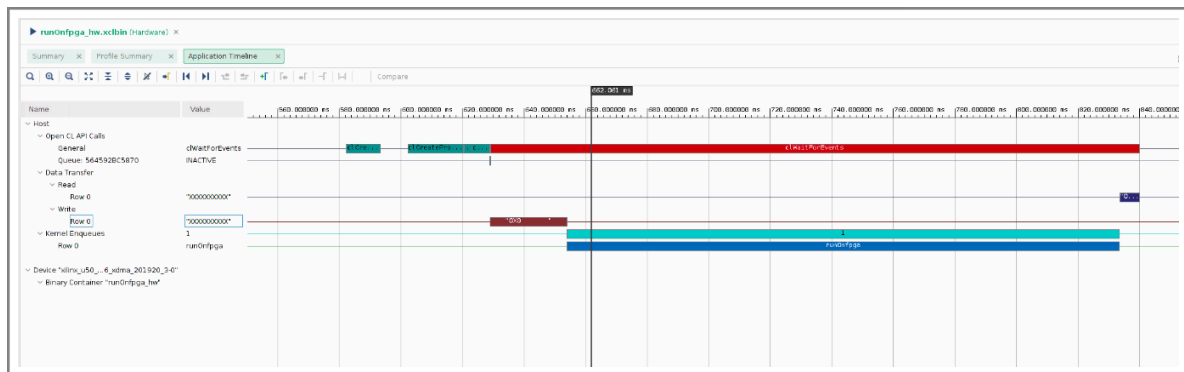


Fig. 5-1 PF=1 application timeline

# GitHub

[https://github.com/Waxpple/HLS\\_2021\\_FALL\\_LABB/tree/master/02-bloom](https://github.com/Waxpple/HLS_2021_FALL_LABB/tree/master/02-bloom)

```
|— 02-bloom
|   |— 1_overview.md
|   |— 2_experience-acceleration.md
|   |— 3_architect-the-application.md
|   |— 4_implement-kernel.md
|   |— 5_data-movement.md
|   |— 6_using-multiple-ddr.md
|   |— README.md
|   |— build
|   |— cpu_src
|   |— images
|   |— makefile
|   |— reference_files
|— report_and_slides
|   |— Lab_3_R09921132_劉彥甫.pptx
|   |— R09921132_report.pdf
7 directories, 9 files
```

## Reference

[1] <https://github.com/Xilinx/Vitis-Tutorials/blob/2021.1/>

Hardware\_Acceleration/Design\_Tutorials/02-bloom/4\_implement-kernel.md

[2] <https://github.com/Xilinx/Vitis-Tutorials/issues/95>