Application Acceleration with High-Level Synthesis (11020EE521800)

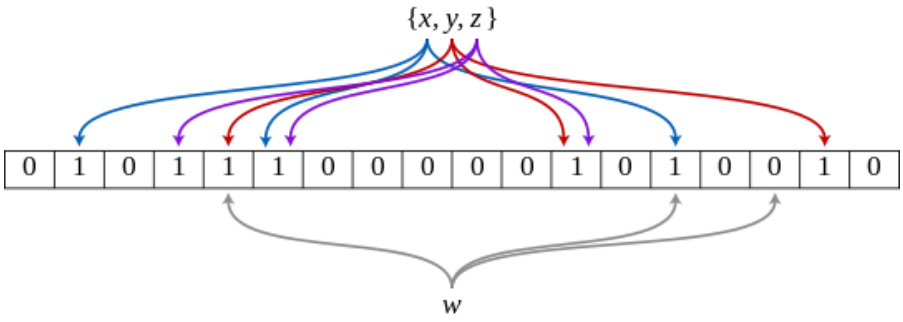
Lab B – Bloom Filter

110061639 林致佑

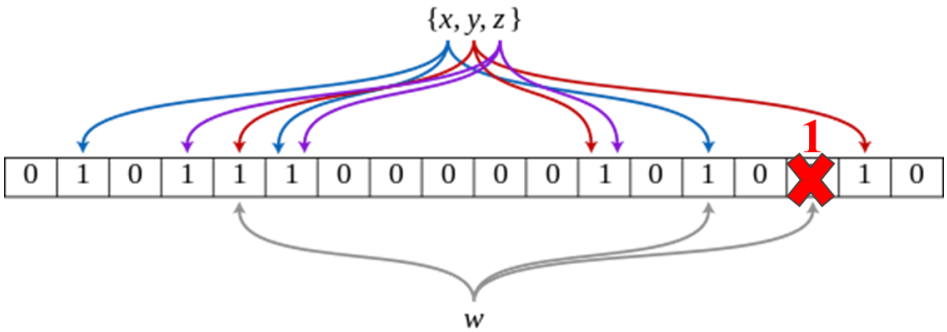
Algorithm introduction:

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:



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

```
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 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.

2. 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;
}
```

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.

3. 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;
    }
  }
}
```

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.

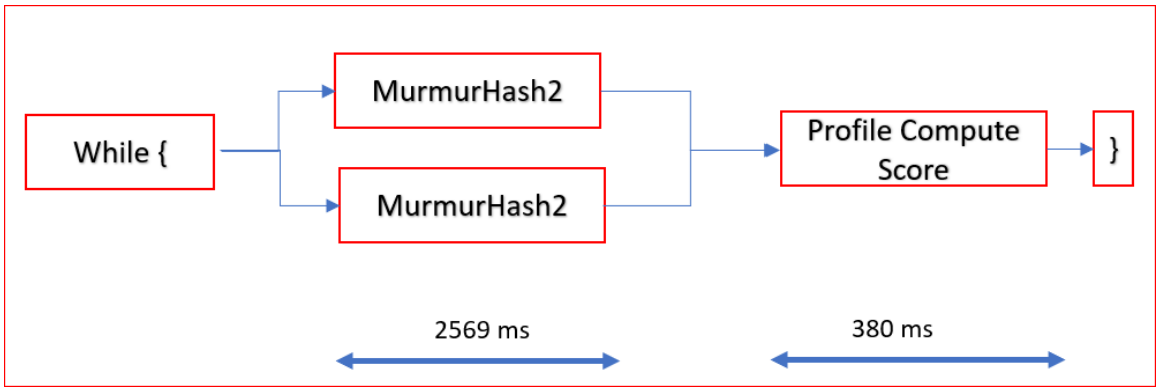
However, the non-sequential accesses are big performance bottlenecks. Because accesses to the profile_weights array are random, implementing this function on the FPGA would not provide much performance benefit.

4. 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.

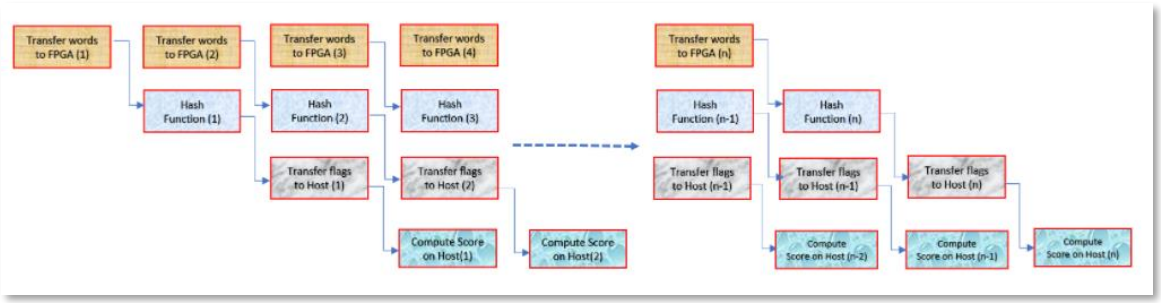
```
./host 100000
Initializing data
Creating documents - total size : 1398.903 MBytes (349725824 words)
Creating profile weights

Total execution time of CPU          |  2949.3867 ms
Compute Hash processing time         |  2569.3266 ms
Compute Score processing time        |   380.0601 ms
----------------------------------------------------------------------
Execution COMPLETE
```

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.



5. 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.



6. 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).

```
m110061639@hpros:~/02-bloom/cpu_src$ ./host 10000
Initializing data
Creating documents - total size : 139.506 MBytes (34876416 words)
Creating profile weights

 Total execution time of CPU          |   171.8735 ms
 Compute Hash processing time         |   142.7425 ms
 Compute Score processing time        |    29.1310 ms
------------------------------------------------------------------------
 Execution COMPLETE

m110061639@hpros:~/02-bloom/cpu_src$
```

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

```
m110061639@hpros:~/02-bloom/makefile/build/single_buffer/kernel_2/hw$ ./host 10000
Initializing data
Creating documents - total size : 139.506 MBytes (34876416 words)
Creating profile weights

Loading runOnfpga_hw.xclbin
 Processing 139.506 MBytes of data
 Single_Buffer: Running with a single buffer of 139.506 MBytes for FPGA processing
-----------------------------------------------------------------
 Executed FPGA accelerated version  |   145.8065 ms   ( FPGA 113.343 ms )
 Executed Software-Only version     |    193.3381 ms
-----------------------------------------------------------------
 Verification: PASS

m110061639@hpros:~/02-bloom/makefile/build/single_buffer/kernel_2/hw$
```

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.

```
m110061639@hpros:~/02-bloom/makefile/build/single_buffer/kernel_4/hw$ ./host 10000
Initializing data
Creating documents - total size : 139.506 MBytes (34876416 words)
Creating profile weights

Loading runOnfpga_hw.xclbin
 Processing 139.506 MBytes of data
 Single_Buffer: Running with a single buffer of 139.506 MBytes for FPGA processing
-----------------------------------------------------------------
 Executed FPGA accelerated version  |   113.2699 ms   ( FPGA 82.023 ms )
 Executed Software-Only version     |    184.0648 ms
-----------------------------------------------------------------
 Verification: PASS

m110061639@hpros:~/02-bloom/makefile/build/single_buffer/kernel_4/hw$
```

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.

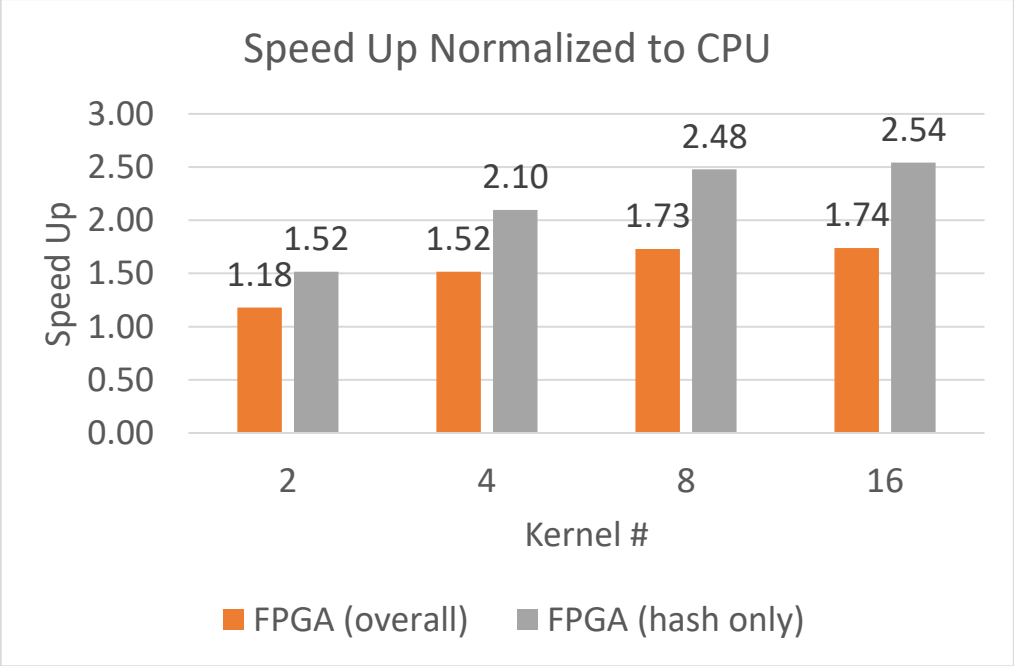
```
m110061639@hpros:~/02-bloom/makefile/build/single_buffer/kernel_8/hw$ ./host 10000
Initializing data
Creating documents - total size : 139.506 MBytes (34876416 words)
Creating profile weights

Loading runOnfpga_hw.xclbin
 Processing 139.506 MBytes of data
 Single_Buffer: Running with a single buffer of 139.506 MBytes for FPGA processing
-----------------------------------------------------------------
 Executed FPGA accelerated version  |    99.2931 ms   ( FPGA 69.350 ms )
 Executed Software-Only version     |    179.2956 ms
-----------------------------------------------------------------
 Verification: PASS

m110061639@hpros:~/02-bloom/makefile/build/single_buffer/kernel_8/hw$
```

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.

```
m110061639@hpros:~/02-bloom/makefile/build/single_buffer/kernel_16/hw$ ./host 10000
Initializing data
Creating documents - total size : 139.506 MBytes (34876416 words)
Creating profile weights

Loading runOnfpga_hw.xclbin
 Processing 139.506 MBytes of data
 Single_Buffer: Running with a single buffer of 139.506 MBytes for FPGA processing
-----------------------------------------------------------------------
 Executed FPGA accelerated version  |    98.8838 ms   ( FPGA 67.574 ms )
 Executed Software-Only version     |    190.4421 ms
-----------------------------------------------------------------------
 Verification: PASS

m110061639@hpros:~/02-bloom/makefile/build/single_buffer/kernel_16/hw$
```

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.

7. Utilization analysis:

Kernel Route Utilization

Name	LUT	LUTAsMem	REG	BRAM	URAM	DSP
Platform	109080	10218	142009	180	0	4
User Budget	760936	391798	1601351	1164	640	5936
Used Resources	7608	1087	15614	151	0	80
Unused Resources	753328	390711	1585737	1013	640	5856
runOnfpga (1)	7608	1087	15614	151	0	80
runOnfpga_1	7608	1087	15614	151	0	80

Kernel / PF = 8 utilization

Kernel Route Utilization

Name	LUT	LUTAsMem	REG	BRAM	URAM	DSP
Platform	109079	10218	142015	180	0	4
User Budget	760937	391798	1601345	1164	640	5936
Used Resources	8963	1087	17550	279	0	160
Unused Resources	751974	390711	1583795	885	640	5776
runOnfpga (1)	8963	1087	17550	279	0	160
runOnfpga_1	8963	1087	17550	279	0	160

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.

8. GitHub Repo:

https://github.com/ChihyuLin0211/Lab_B_Vitis_Tutorials_02_Bloom

9. 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

This implementation provides:

- 16 GB HBM
- 256 MB for Alveo U50 HBM segments, called pseudo channels (PCs)
- 512 MB for Alveo U55C PCs
- An independent AXI channel for communication with the FPGA through a segmented crossbar switch per pseudo channel
- A two-channel memory controller per two PCs
- 14.375 GB/s max theoretical bandwidth per PC
- 460 GB/s (32 * 14.375 GB/s) max theoretical bandwidth for the HBM subsystem