lab_B Bloom filter 108061173 邱崇喆

Lab overview:

In this lab, we want to score all documents with a given user array and word weights, and every document will be inputted as an array of uint words, where the most significant 24 bits are word ID and the other 8 bits indicate its occurring frequency.

Only words (word ID) that occur in the given user array will count for the score of a document, namely their corresponding weights are not zero, and that is why we need a bloom filter to efficiently search if a word is inside the input array.

We roughly divide the implementation into three part:

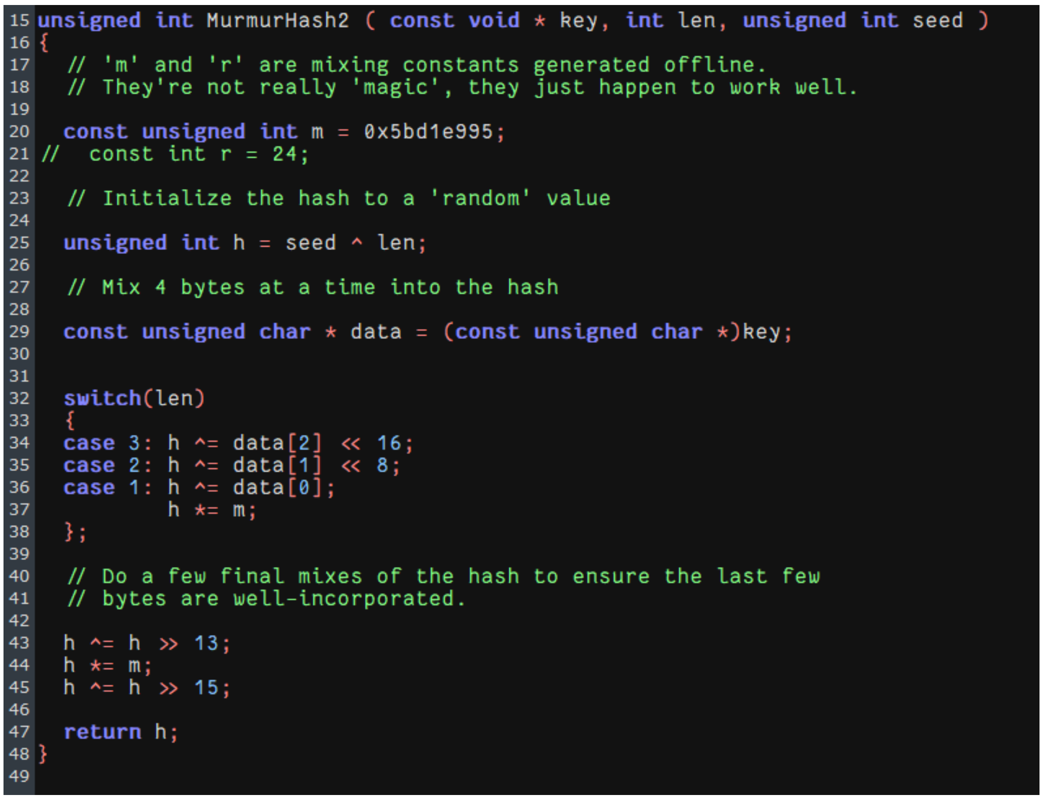
- prepare input data, including documents array and bloom filter coefficient
- check the existence of all words in the documents by bloom filter
- evaluate every document

Where do we need FPGA?

Let’s see the software-only codes first:

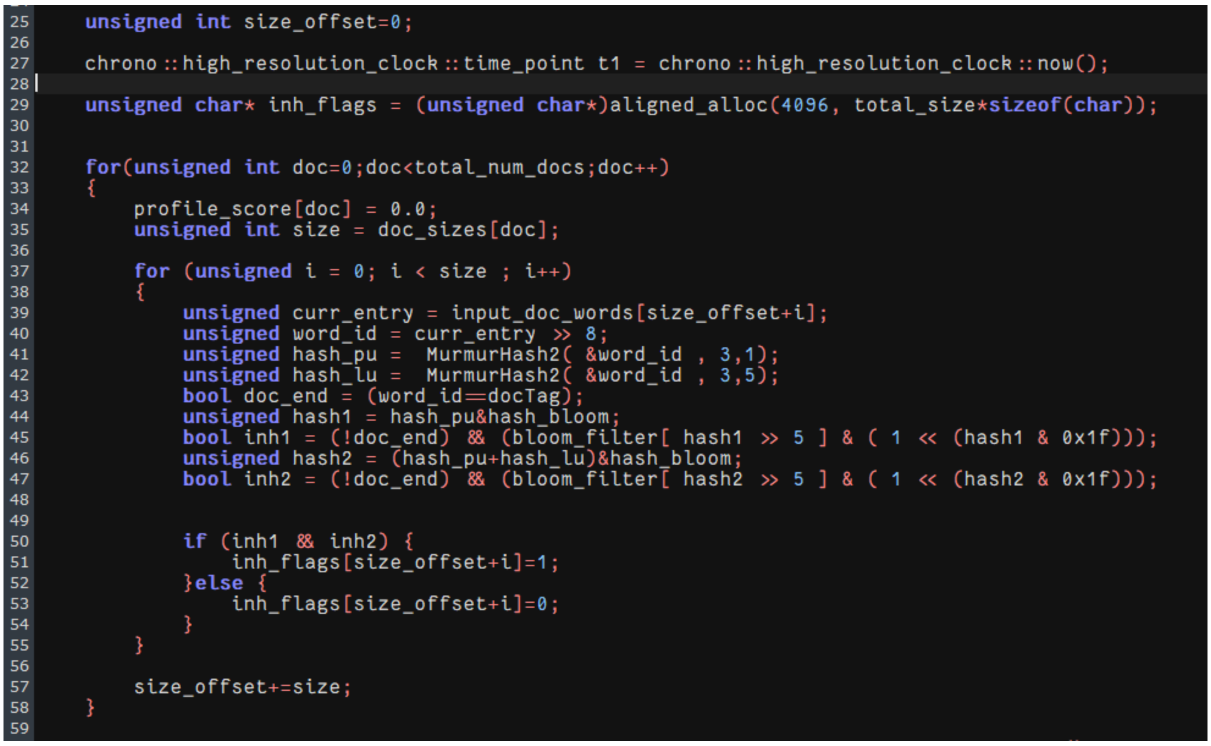
//

MurmurHash2() can be used to calculate the hash of input doc words:



```
unsigned int MurmurHash2 ( const void * key, int len, unsigned int seed )
{
  // 'm' and 'r' are mixing constants generated offline.
  // They're not really 'magic', they just happen to work well.

  const unsigned int m = 0x5bd1e995;
//  const int r = 24;

  // 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;
}
```

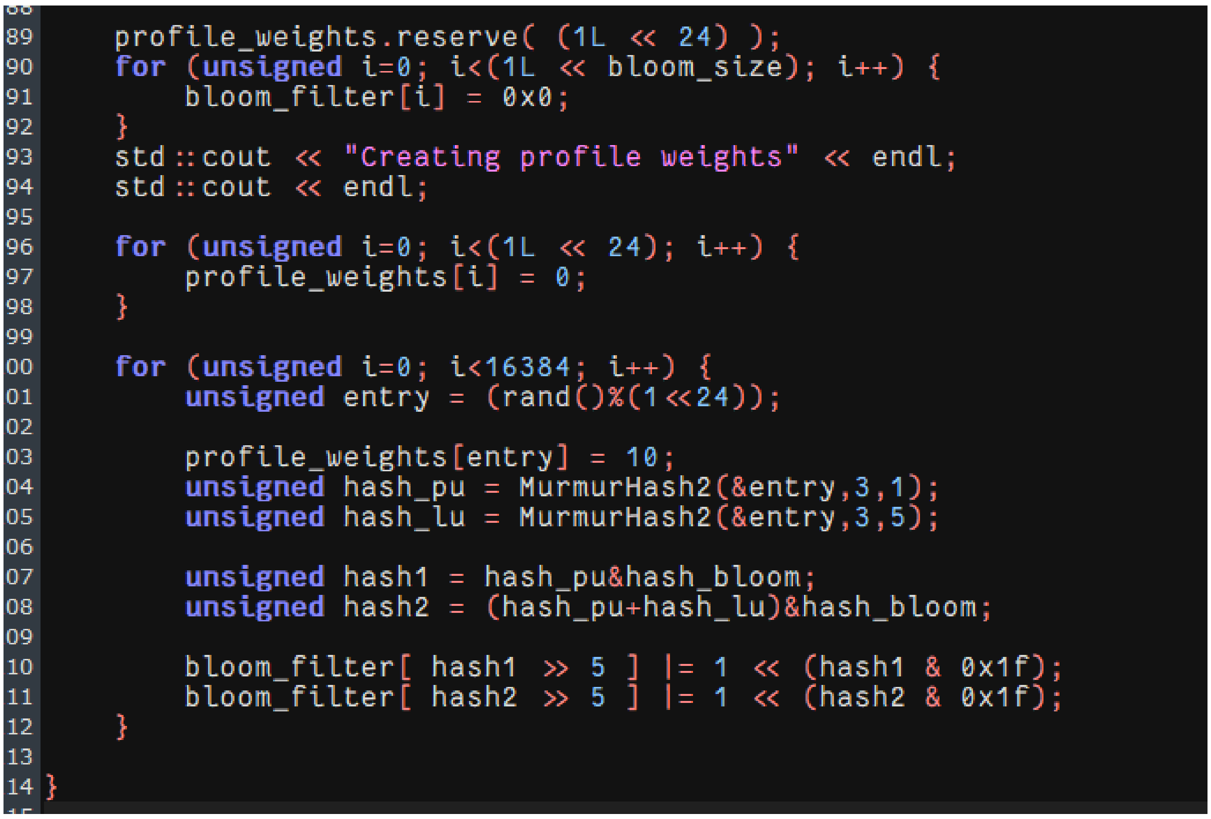
Use the results of Murmur for bloom filtering, checking the existence of the words:



```
    unsigned int size_offset=0;

    chrono::high_resolution_clock::time_point t1 = chrono::high_resolution_clock::now();

    unsigned char* inh_flags = (unsigned char*)aligned_alloc(4096, total_size*sizeof(char));


    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;
    }
```

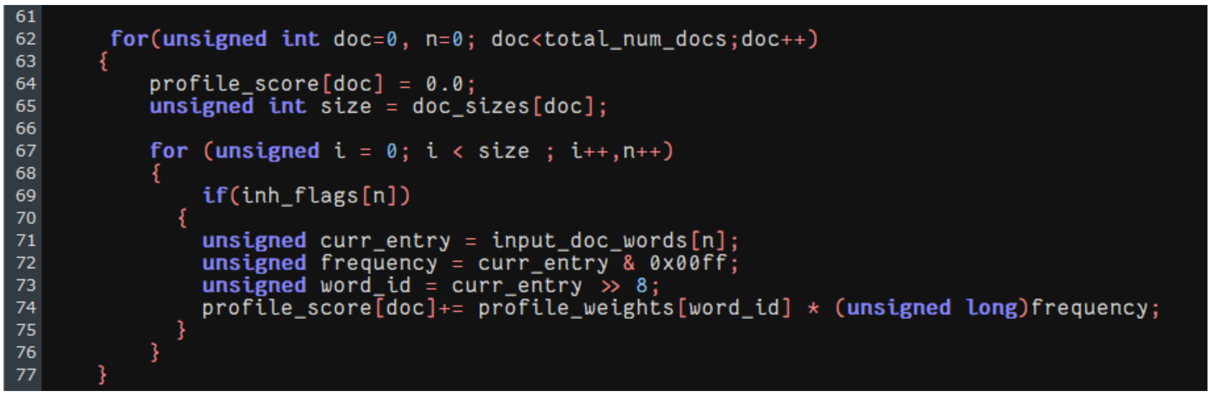
If a word is existing in the bloom filter, we'll set the corresponding bool entry in the inh_flags array as 1, which will be used for computing score.
//

a code snippet of setupData() sets up the bloom filter:



```
    profile_weights.reserve( (1L << 24) );
    for (unsigned i=0; i<(1L << bloom_size); i++) {
        bloom_filter[i] = 0x0;
    }
    std::cout << "Creating profile weights" << endl;
    std::cout << endl;

    for (unsigned i=0; i<(1L << 24); i++) {
        profile_weights[i] = 0;
    }

    for (unsigned i=0; i<16384; i++) {
        unsigned entry = (rand()%(1<<24));

        profile_weights[entry] = 10;
        unsigned hash_pu = MurmurHash2(&entry,3,1);
        unsigned hash_lu = MurmurHash2(&entry,3,5);

        unsigned hash1 = hash_pu&hash_bloom;
        unsigned hash2 = (hash_pu+hash_lu)&hash_bloom;

        bloom_filter[ hash1 >> 5 ] |= 1 << (hash1 & 0x1f);
        bloom_filter[ hash2 >> 5 ] |= 1 << (hash2 & 0x1f);
    }

}
```

The following is a code snippet of compute_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;
          }
        }
    }
```

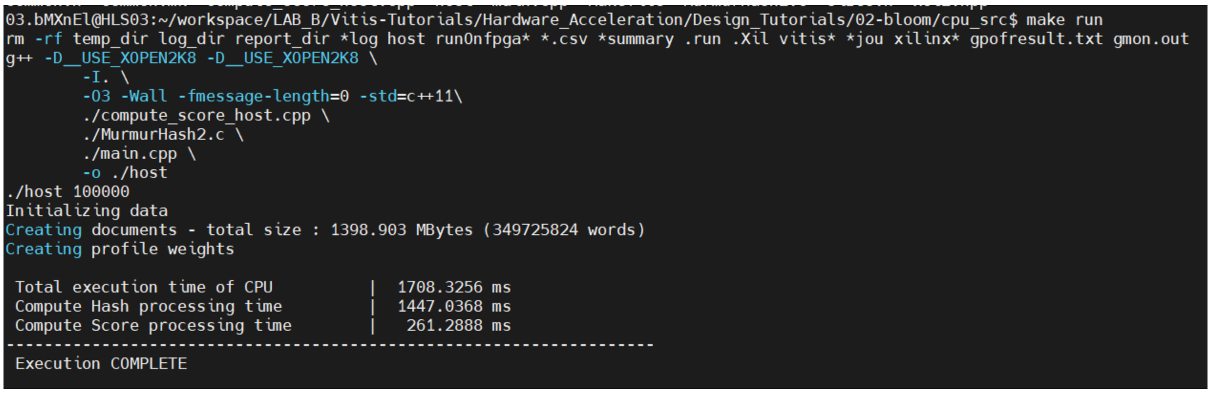
Among the aforementioned codes, the best part we can choose to implement on FPGA is the two snippets in the red segment, because:

- MurmurHash2 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, so we need at most 44 cycles.
- We can create custom architectures on FPGA, and therefore create an accelerator that will shift the data by an arbitrary number of bits in single clock cycle.
- FPGAs have DSP units that can perform faster multiplication than CPU.
- The second snippet in the red segment sequentially accesses the input_doc_array, which allows sequential accesses to DDR. It'll be efficient if FPGAs can do this.

The aforementioned compute_score snippet won't be implemented on FPGA because:

- The profile_weights array (weights) is not sequentially but randomly accessed.
  - The profile_weights array in this lab is 128MB, which means if implemented on FPGAs, it will be a big performance bottleneck.
- Another reason is that it only takes about 11% of running time, so we can keep this on the host side.

The following is the execution time of mere software:



```
03.bMXnEl@HLS03:~/workspace/LAB_B/Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/02-bloom/cpu_src$ make run
rm -rf temp_dir log_dir report_dir *log host runOnfpga* *.csv *summary .run .Xil vitis* *jou xilinx* gpofresult.txt 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 100000
Initializing data
Creating documents - total size : 1398.903 MBytes (349725824 words)
Creating profile weights

 Total execution time of CPU          |  1708.3256 ms
 Compute Hash processing time         |  1447.0368 ms
 Compute Score processing time        |   261.2888 ms
--------------------------------------------------------------------
 Execution COMPLETE
```

We can see a majority of processing time is spent on Hashing.

Task Flow:

In this lab, we'll input 100000 documents equivalent to about 350M words (1.4GB) and bloom filter coefficients to the kernel to compute hashes and flags, and then transmit the flags back to the host, and finally score every document.

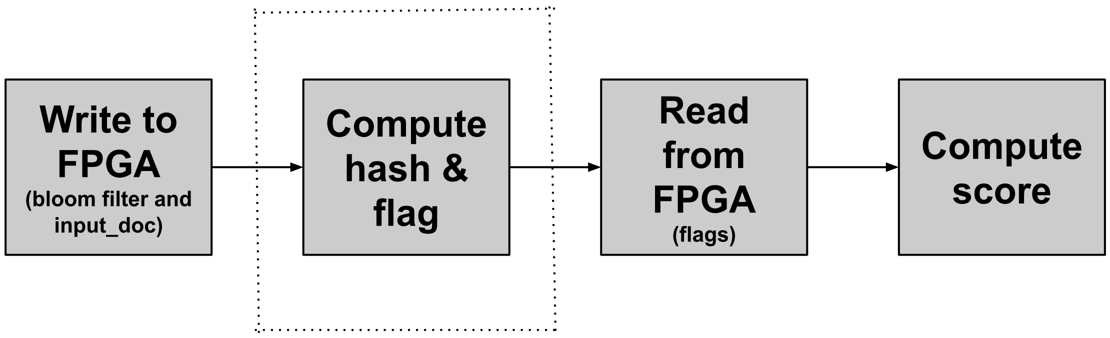


fig.1 The flow of the lab, the block surrounded by the dotted line is implemented on FPGA

It is obvious that if we implement the lab like fig 1, namely sequentially running the task blocks like running on CPU, we'll end up yielding bad performance, so we'll need to take advantage of concurrent processing and overlapping of the FPGA.

We'll start to discuss how to further optimize the above task flow in the following section.

Optimizations of the Task Flow:

Process multiple word in the kernel (Kernel Parallelization) :

In this lab, we use the following interface requirements to create kernel:

- Read multiple words stored in the DDR as a 512-bit DDR access, equivalent of reading 16 words per DDR access.
- Write multiple flags to the DDR as a 512-bit DDR access, equivalent to writing 32 flags per DDR access.
- Compute 4 words in parallel with each word requiring two MurmurHash2 functions
- Compute the hash (two MurmurHash2 functions) functions for 4 words every cycle.

The kernel has 5 arguments:

1. 512-bit "output_flags" (through buffer)
2. 512-bit "input_words"   (through buffer)
3. bloom_filter: Pointer of array with Bloom coefficients. (through buffer)
4. Total number of words to be computed
5. load_filter: Enable or disable loading bloom filter coefficients.

Following this requirements, we can expand the kernel (dotted lined block in fig.1) into fig.2

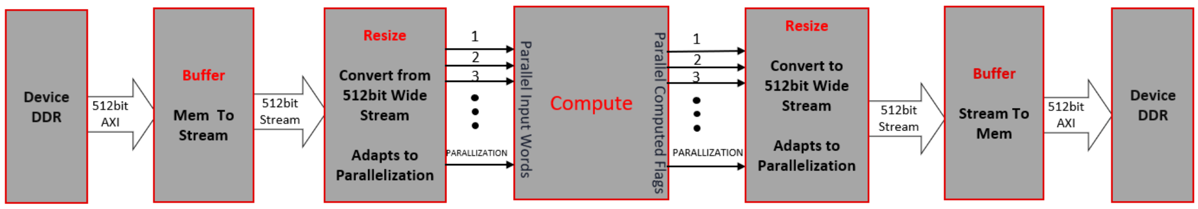


fig.2 The flow of the kernel

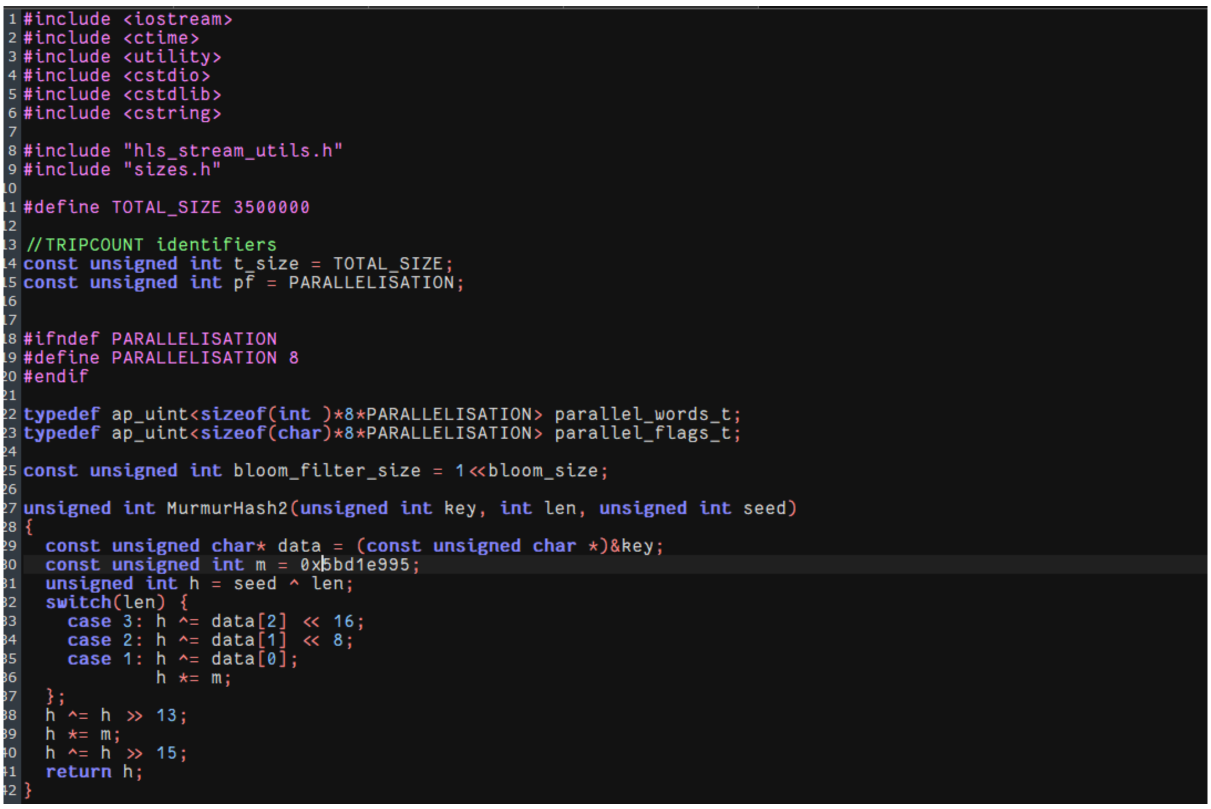
source:
https://github.com/Xilinx/Vitis-Tutorials/blob/2022.1/Hardware_Acceleration/Design_Tutorials/02-bloom/4_implement-kernel.md

The function compute_hash_flags_dataflow is responsible for this flow.

```
void compute_hash_flags_dataflow(
       ap_uint<512>*   output_flags,
       ap_uint<512>*   input_words,
       unsigned int    bloom_filter[PARALLELIZATION][bloom_filter_size],
       unsigned int    total_size)
{
#pragma HLS DATAFLOW

    hls::stream<ap_uint<512> >     data_from_gmem;
    hls::stream<parallel_words_t> word_stream;
    hls::stream<parallel_flags_t> flag_stream;
    hls::stream<ap_uint<512> >     data_to_gmem;
    . . . .
}
```

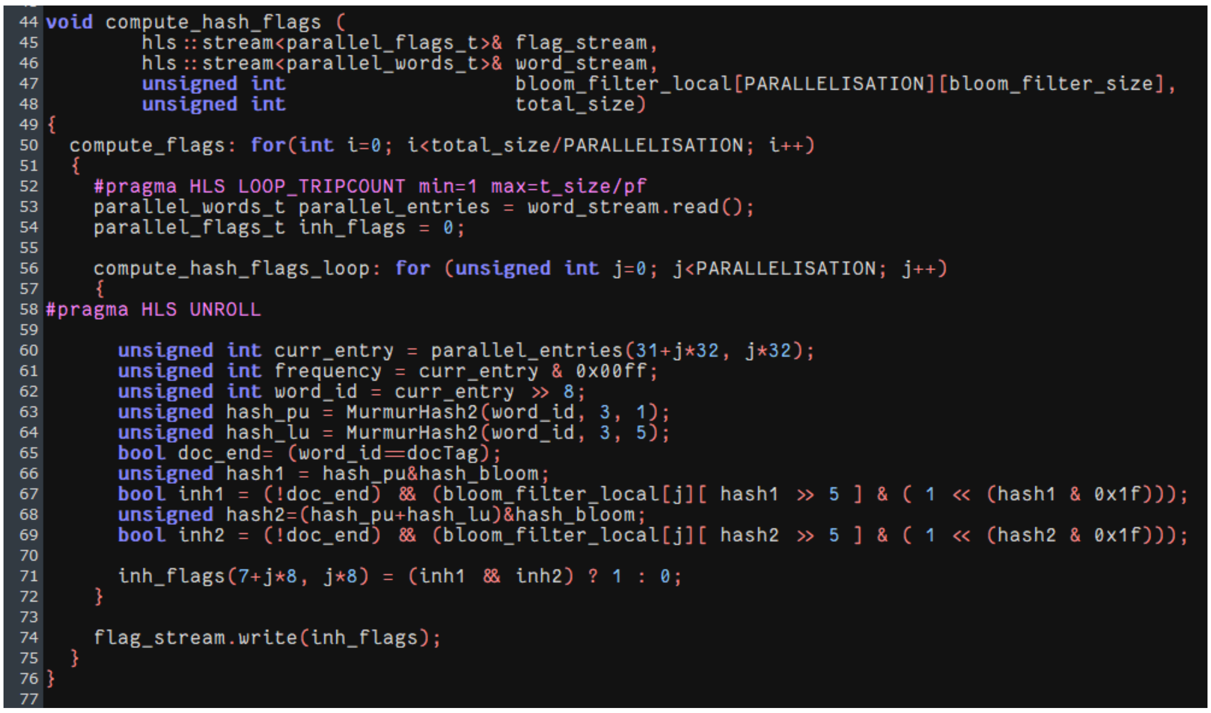
The kernel file includes:

1. Murmurhash2() — Hashing



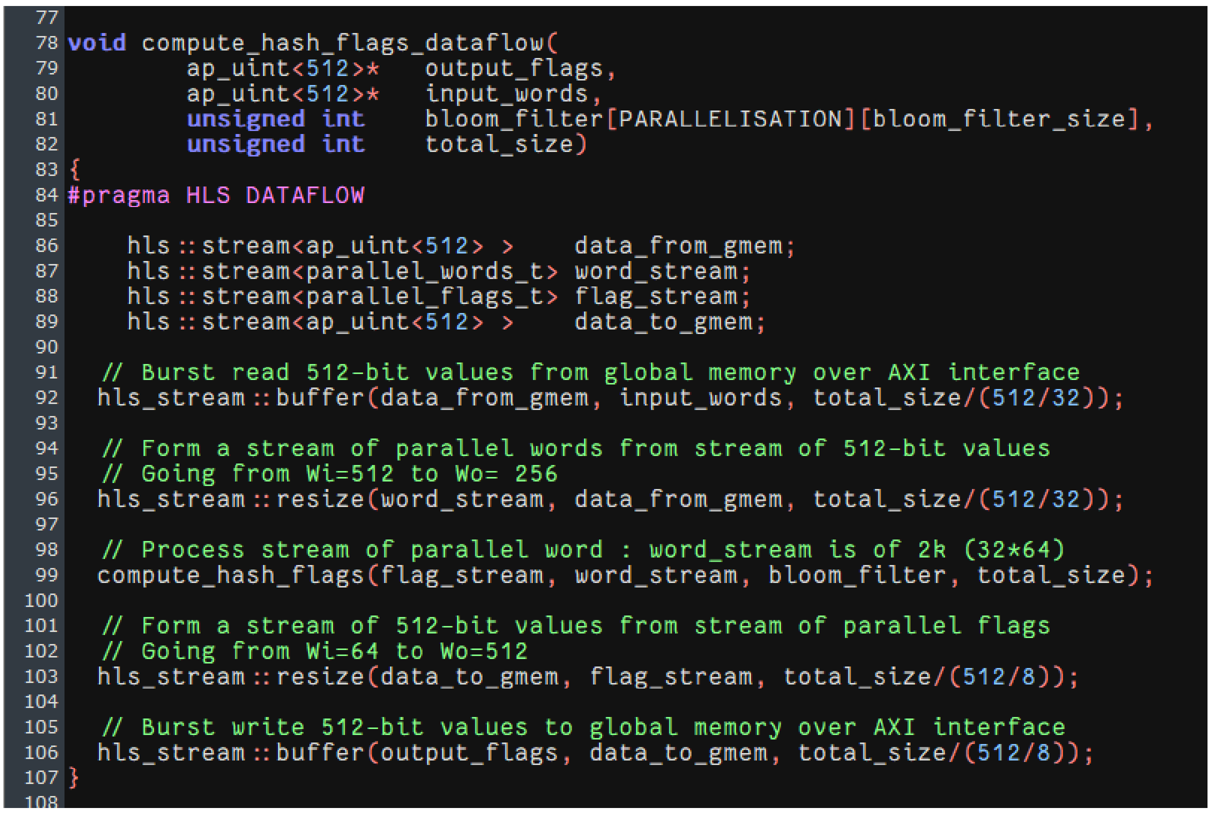
```
#include <iostream>
#include <ctime>
#include <utility>
#include <cstdio>
#include <cstdlib>
#include <cstring>

#include "hls_stream_utils.h"
#include "sizes.h"

#define TOTAL_SIZE 3500000

 //TRIPCOUNT identifiers
const unsigned int t_size = TOTAL_SIZE;
const unsigned int pf = PARALLELISATION;


#ifndef PARALLELISATION
#define PARALLELISATION 8
#endif

typedef ap_uint<sizeof(int )*8*PARALLELISATION> parallel_words_t;
typedef ap_uint<sizeof(char)*8*PARALLELISATION> parallel_flags_t;

const unsigned int bloom_filter_size = 1<<bloom_size;

unsigned int MurmurHash2(unsigned int key, int len, unsigned int seed)
{
  const unsigned char* data = (const unsigned char *)&key;
  const unsigned int m = 0x5bd1e995;
  unsigned int h = seed ^ len;
  switch(len) {
    case 3: h ^= data[2] << 16;
    case 2: h ^= data[1] << 8;
    case 1: h ^= data[0];
            h *= m;
  };
  h ^= h >> 13;
  h *= m;
  h ^= h >> 15;
  return h;
}
```

2. compute_hash_flags — Compute the flag by the generated two hashes, the corresponding task block in fig 2. is the "compute" block



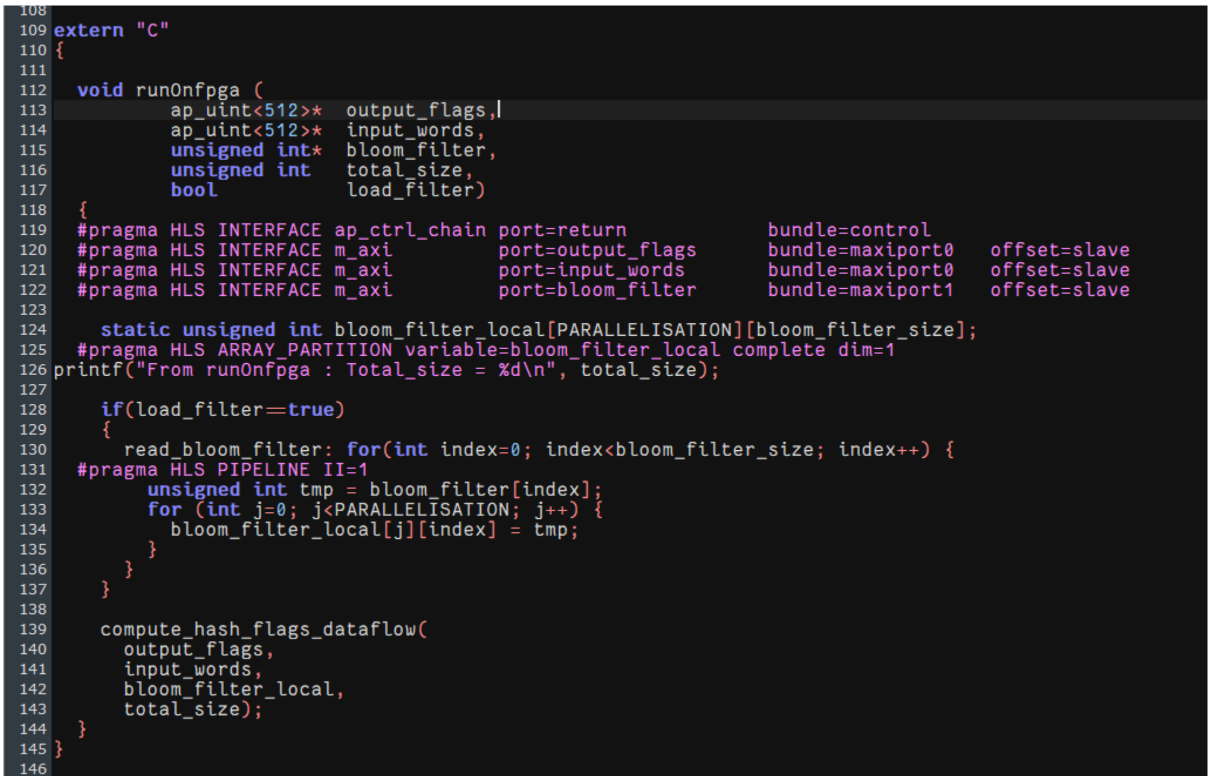
```
void compute_hash_flags (
        hls::stream<parallel_flags_t>& flag_stream,
        hls::stream<parallel_words_t>& word_stream,
        unsigned int                   bloom_filter_local[PARALLELISATION][bloom_filter_size],
        unsigned int                   total_size)
{
  compute_flags: for(int i=0; i<total_size/PARALLELISATION; i++)
  {
    #pragma HLS LOOP_TRIPCOUNT min=1 max=t_size/pf
    parallel_words_t parallel_entries = word_stream.read();
    parallel_flags_t inh_flags = 0;

    compute_hash_flags_loop: for (unsigned int j=0; j<PARALLELISATION; j++)
    {
#pragma HLS UNROLL

      unsigned int curr_entry = parallel_entries(31+j*32, j*32);
      unsigned int frequency = curr_entry & 0x00ff;
      unsigned int 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_local[j][ hash1 >> 5 ] & ( 1 << (hash1 & 0x1f)));
      unsigned hash2=(hash_pu+hash_lu)&hash_bloom;
      bool inh2 = (!doc_end) && (bloom_filter_local[j][ hash2 >> 5 ] & ( 1 << (hash2 & 0x1f)));

      inh_flags(7+j*8, j*8) = (inh1 && inh2) ? 1 : 0;
    }

    flag_stream.write(inh_flags);
  }
}
```

3. compute_hash_flags_dataflow — implementing the flow in fig 2., this function will resize the stream data to desired number of words (e.g. Parallelization=4 words) and then feed into the function “compute_hash_flags”:



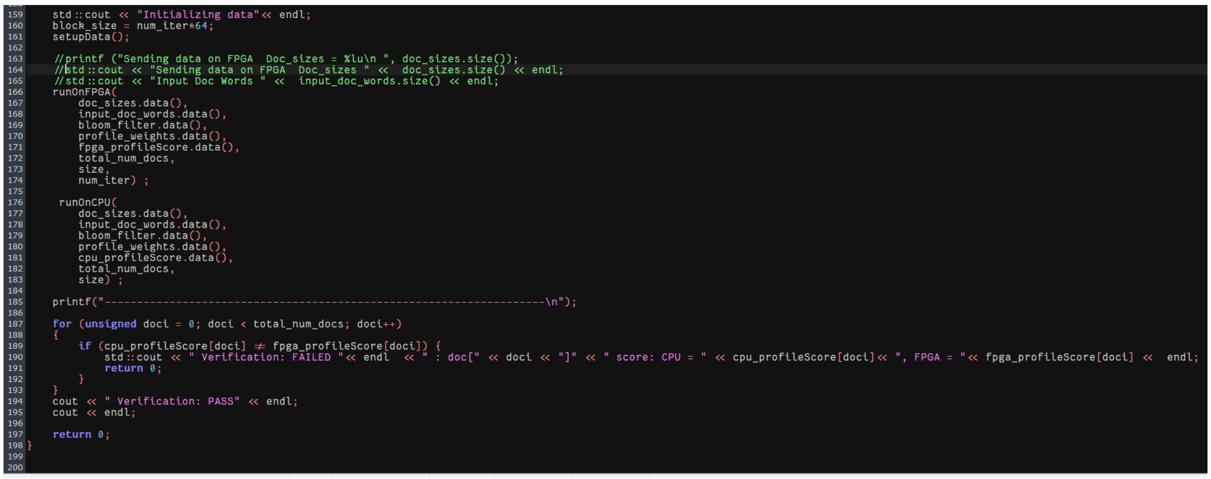
```
void compute_hash_flags_dataflow(
        ap_uint<512>*   output_flags,
        ap_uint<512>*   input_words,
        unsigned int    bloom_filter[PARALLELISATION][bloom_filter_size],
        unsigned int    total_size)
{
#pragma HLS DATAFLOW

    hls::stream<ap_uint<512> >    data_from_gmem;
    hls::stream<parallel_words_t> word_stream;
    hls::stream<parallel_flags_t> flag_stream;
    hls::stream<ap_uint<512> >    data_to_gmem;

  // Burst read 512-bit values from global memory over AXI interface
  hls_stream::buffer(data_from_gmem, input_words, total_size/(512/32));

  // Form a stream of parallel words from stream of 512-bit values
  // Going from Wi=512 to Wo= 256
  hls_stream::resize(word_stream, data_from_gmem, total_size/(512/32));

  // Process stream of parallel word : word_stream is of 2k (32*64)
  compute_hash_flags(flag_stream, word_stream, bloom_filter, total_size);

  // Form a stream of 512-bit values from stream of parallel flags
  // Going from Wi=64 to Wo=512
  hls_stream::resize(data_to_gmem, flag_stream, total_size/(512/8));

  // Burst write 512-bit values to global memory over AXI interface
  hls_stream::buffer(output_flags, data_to_gmem, total_size/(512/8));
}
```

4. runOnfpga — The kernel top function, it’ll load the bloom filter’s coefficients first, and then perform “compute_hash_flags_dataflow” where the function ”computer_hash_flags” is called:



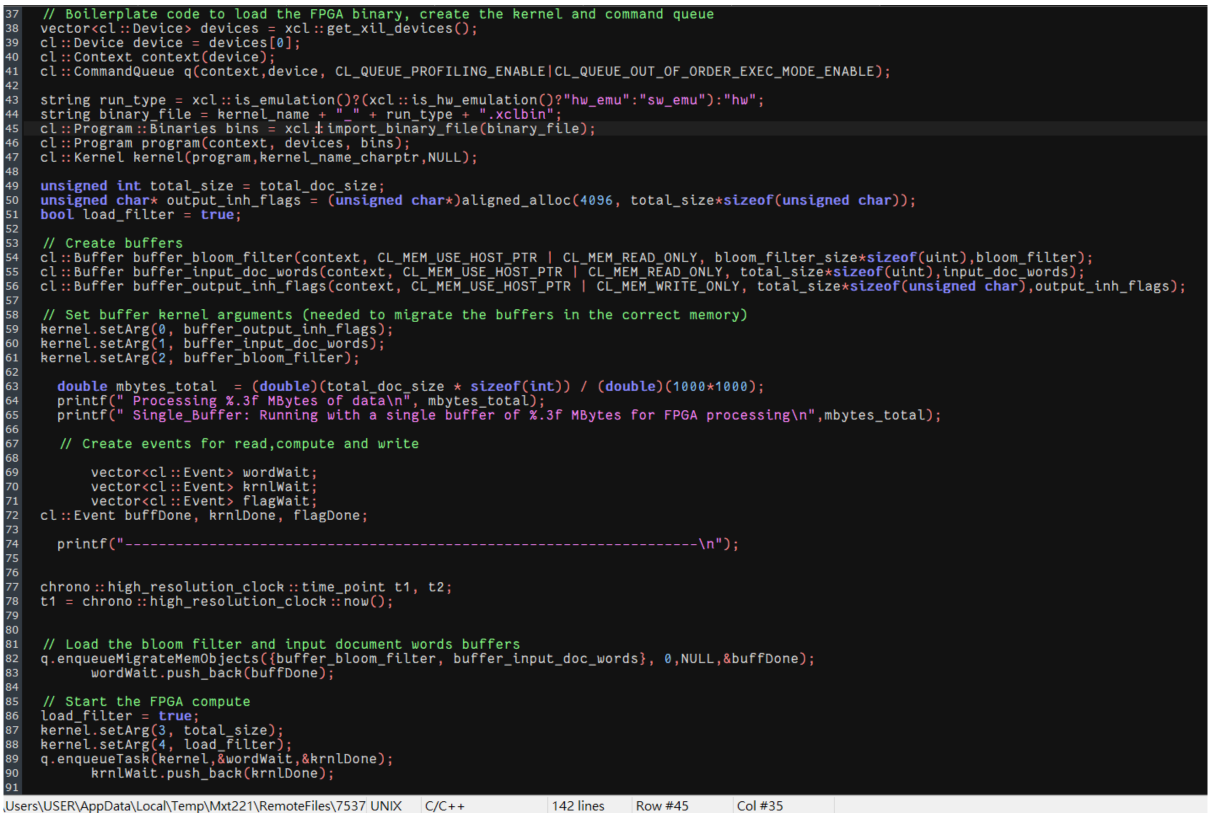
```
extern "C"
{

  void runOnfpga (
          ap_uint<512>*  output_flags,
          ap_uint<512>*  input_words,
          unsigned int*  bloom_filter,
          unsigned int   total_size,
          bool           load_filter)
  {
  #pragma HLS INTERFACE ap_ctrl_chain port=return          bundle=control
  #pragma HLS INTERFACE m_axi         port=output_flags    bundle=maxiport0   offset=slave
  #pragma HLS INTERFACE m_axi         port=input_words     bundle=maxiport0   offset=slave
  #pragma HLS INTERFACE m_axi         port=bloom_filter    bundle=maxiport1   offset=slave

    static unsigned int bloom_filter_local[PARALLELISATION][bloom_filter_size];
  #pragma HLS ARRAY_PARTITION variable=bloom_filter_local complete dim=1
printf("From runOnfpga : Total_size = %d\n", total_size);

    if(load_filter==true)
    {
      read_bloom_filter: for(int index=0; index<bloom_filter_size; index++) {
  #pragma HLS PIPELINE II=1
        unsigned int tmp = bloom_filter[index];
        for (int j=0; j<PARALLELISATION; j++) {
          bloom_filter_local[j][index] = tmp;
        }
      }
    }

    compute_hash_flags_dataflow(
      output_flags,
      input_words,
      bloom_filter_local,
      total_size);
  }
}
```

The host function includes two functions runOnFPGA & runOnCPU, the former will instantiate the kernel “runOnfpga” and perform data transmission between host and kernel:



```
    std::cout << "Initializing data"<< endl;
    block_size = num_iter*64;
    setupData();

     //printf ("Sending data on FPGA  Doc_sizes = %lu\n ", doc_sizes.size());
     //std::cout << "Sending data on FPGA  Doc_sizes " <<  doc_sizes.size() << endl;
     //std::cout << "Input Doc Words " <<  input_doc_words.size() << endl;
    runOnFPGA(
        doc_sizes.data(),
        input_doc_words.data(),
        bloom_filter.data(),
        profile_weights.data(),
        fpga_profileScore.data(),
        total_num_docs,
        size,
        num_iter) ;

     runOnCPU(
        doc_sizes.data(),
        input_doc_words.data(),
        bloom_filter.data(),
        profile_weights.data(),
        cpu_profileScore.data(),
        total_num_docs,
        size) ;

    printf("--------------------------------------------------------------------\n");

    for (unsigned doci = 0; doci < total_num_docs; doci++)
    {
        if (cpu_profileScore[doci] != fpga_profileScore[doci]) {
            std::cout << " Verification: FAILED "<< endl  << " : doc[" << doci << "]" << " score: CPU = " << cpu_profileScore[doci]<< ", FPGA = "<< fpga_profileScore[doci] <<  endl;
            return 0;
        }
    }
    cout << " Verification: PASS" << endl;
    cout << endl;

    return 0;
}
```

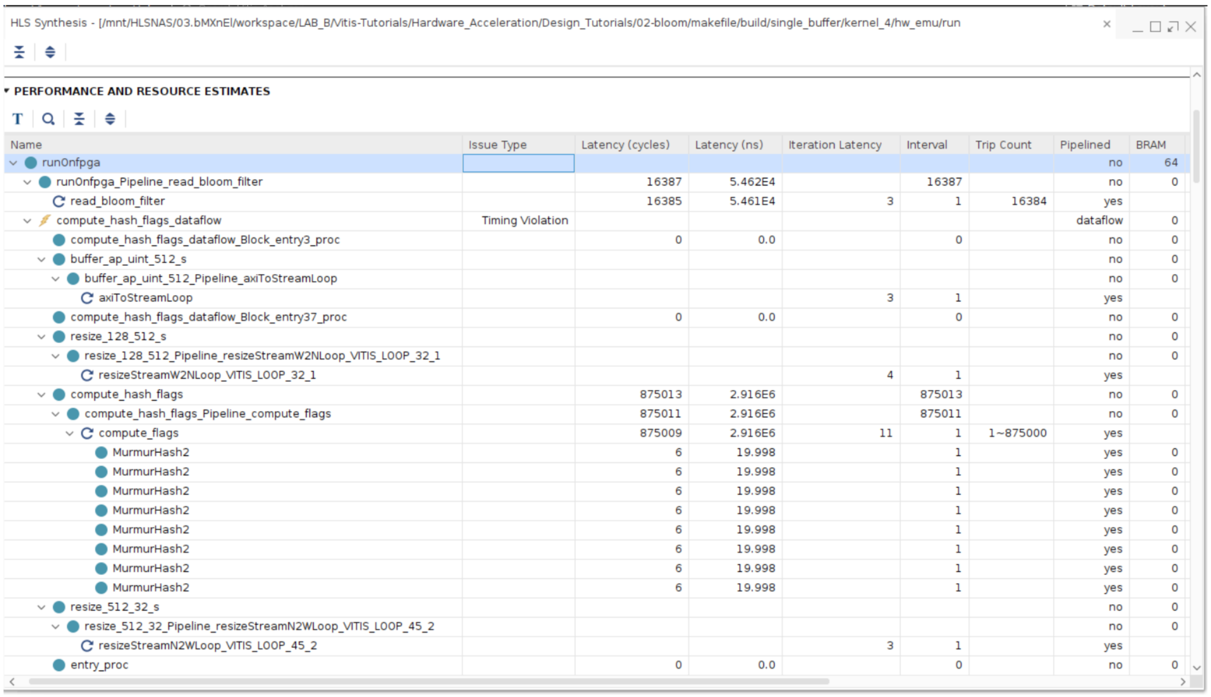
runOnFPGA creates buffer and setups kernel:



```
 // Boilerplate code to load the FPGA binary, create the kernel and command queue
vector<cl::Device> devices = xcl::get_xil_devices();
cl::Device device = devices[0];
cl::Context context(device);
cl::CommandQueue q(context,device, CL_QUEUE_PROFILING_ENABLE|CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE);

string run_type = xcl::is_emulation()?(xcl::is_hw_emulation()?"hw_emu":"sw_emu"):"hw";
string binary_file = kernel_name + "_" + run_type + ".xclbin";
cl::Program::Binaries bins = xcl::import_binary_file(binary_file);
cl::Program program(context, devices, bins);
cl::Kernel kernel(program,kernel_name_charptr,NULL);

unsigned int total_size = total_doc_size;
unsigned char* output_inh_flags = (unsigned char*)aligned_alloc(4096, total_size*sizeof(unsigned char));
bool load_filter = true;

 // Create buffers
cl::Buffer buffer_bloom_filter(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, bloom_filter_size*sizeof(uint),bloom_filter);
cl::Buffer buffer_input_doc_words(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, total_size*sizeof(uint),input_doc_words);
cl::Buffer buffer_output_inh_flags(context, CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY, total_size*sizeof(unsigned char),output_inh_flags);

 // Set buffer kernel arguments (needed to migrate the buffers in the correct memory)
kernel.setArg(0, buffer_output_inh_flags);
kernel.setArg(1, buffer_input_doc_words);
kernel.setArg(2, buffer_bloom_filter);

  double mbytes_total  = (double)(total_doc_size * sizeof(int)) / (double)(1000*1000);
  printf(" Processing %.3f MBytes of data\n", mbytes_total);
  printf(" Single_Buffer: Running with a single buffer of %.3f MBytes for FPGA processing\n",mbytes_total);

  // Create events for read,compute and write

      vector<cl::Event> wordWait;
      vector<cl::Event> krnlWait;
      vector<cl::Event> flagWait;
cl::Event buffDone, krnlDone, flagDone;

  printf("--------------------------------------------------------------------\n");


chrono::high_resolution_clock::time_point t1, t2;
t1 = chrono::high_resolution_clock::now();


 // Load the bloom filter and input document words buffers
q.enqueueMigrateMemObjects({buffer_bloom_filter, buffer_input_doc_words}, 0,NULL,&buffDone);
      wordWait.push_back(buffDone);

 // Start the FPGA compute
load_filter = true;
kernel.setArg(3, total_size);
kernel.setArg(4, load_filter);
q.enqueueTask(kernel,&wordWait,&krnlDone);
      krnlWait.push_back(krnlDone);
```

Let’s use the following HLS synthesis report to see latencies of the functions

- The compute_hash_flags latency reported is 875,011 cycles, because there are in total 35,000,000 words, computed with 4 words in parallel, and with task-level parallelism pipelining we can achieve about 35M/4 cycles.
- The read_bloom_filter task takes about 16000 cycles because the loop is iterated over 16,000 cycles reading 32-bits data from the Bloom filter coefficients. This is also due to pipelining with the code #pragma HLS PIPELINE II=1 when loading the filter coefficients.

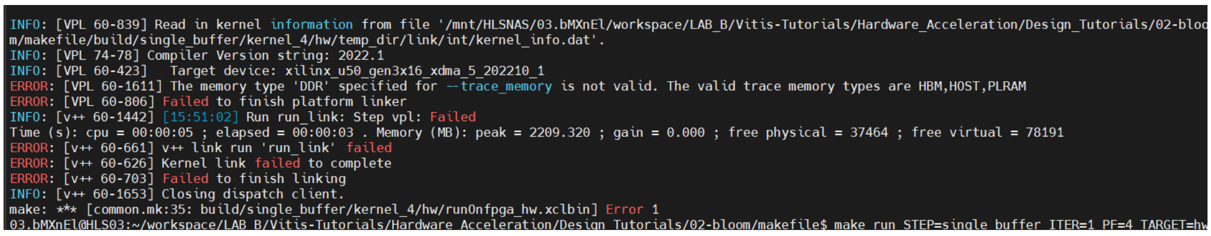


HLS Synthesis - [/mnt/HLSNAS/03.bMXnEl/workspace/LAB_B/Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/02-bloom/makefile/build/single_buffer/kernel_4/hw_emu/run

PERFORMANCE AND RESOURCE ESTIMATES

Name	Issue Type	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM
runOnfpga							no	64
runOnfpga_Pipeline_read_bloom_filter		16387	5.462E4		16387		no	0
read_bloom_filter		16385	5.461E4	3	1	16384	yes	
compute_hash_flags_dataflow	Timing Violation						dataflow	0
compute_hash_flags_dataflow_Block_entry3_proc		0	0.0		0		no	0
buffer_ap_uint_512_s							no	0
buffer_ap_uint_512_Pipeline_axiToStreamLoop							no	0
axiToStreamLoop				3	1		yes	
compute_hash_flags_dataflow_Block_entry37_proc		0	0.0		0		no	0
resize_128_512_s							no	0
resize_128_512_Pipeline_resizeStreamW2NLoop_VITIS_LOOP_32_1							no	0
resizeStreamW2NLoop_VITIS_LOOP_32_1				4	1		yes	
compute_hash_flags		875013	2.916E6		875013		no	0
compute_hash_flags_Pipeline_compute_flags		875011	2.916E6		875011		no	0
compute_flags		875009	2.916E6	11	1	1~875000	yes	
MurmurHash2		6	19.998		1		yes	0
MurmurHash2		6	19.998		1		yes	0
MurmurHash2		6	19.998		1		yes	0
MurmurHash2		6	19.998		1		yes	0
MurmurHash2		6	19.998		1		yes	0
MurmurHash2		6	19.998		1		yes	0
MurmurHash2		6	19.998		1		yes	0
MurmurHash2		6	19.998		1		yes	0
resize_512_32_s							no	0
resize_512_32_Pipeline_resizeStreamN2WLoop_VITIS_LOOP_45_2							no	0
resizeStreamN2WLoop_VITIS_LOOP_45_2				3	1		yes	
entry_proc		0	0.0		0		no	0

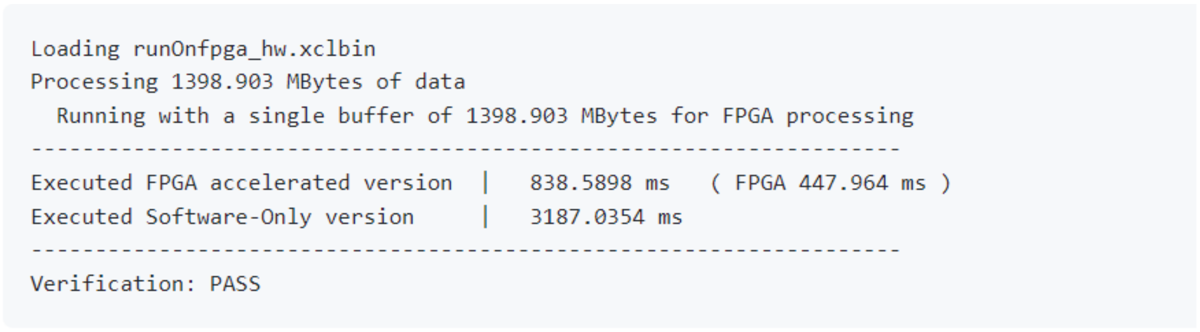
There are some issues when performing this lab with U50 (originally use U200), similar problem has been discussed in the thread https://github.com/Xilinx/Vitis-Tutorials/issues/95, and the solution was also provided.



```
INFO: [VPL 60-839] Read in kernel information from file '/mnt/HLSNAS/03.bMXnEl/workspace/LAB_B/Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/02-bloo
m/makefile/build/single_buffer/kernel_4/hw/temp_dir/link/int/kernel_info.dat'.
INFO: [VPL 74-78] Compiler Version string: 2022.1
INFO: [VPL 60-423]   Target device: xilinx_u50_gen3x16_xdma_5_202210_1
ERROR: [VPL 60-1611] The memory type 'DDR' specified for --trace_memory is not valid. The valid trace memory types are HBM,HOST,PLRAM
ERROR: [VPL 60-806] Failed to finish platform linker
INFO: [v++ 60-1442] [15:51:02] Run run_link: Step vpl: Failed
Time (s): cpu = 00:00:05 ; elapsed = 00:00:03 . Memory (MB): peak = 2209.320 ; gain = 0.000 ; free physical = 37464 ; free virtual = 78191
ERROR: [v++ 60-661] v++ link run 'run_link' failed
ERROR: [v++ 60-626] Kernel link failed to complete
ERROR: [v++ 60-703] Failed to finish linking
INFO: [v++ 60-1653] Closing dispatch client.
make: *** [common.mk:35: build/single_buffer/kernel_4/hw/runOnfpga_hw.xclbin] Error 1
03.bMXnEl@HLS03:~/workspace/LAB_B/Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/02-bloom/makefile$ make run STEP=single_buffer ITER=1 PF=4 TARGET=hw
```

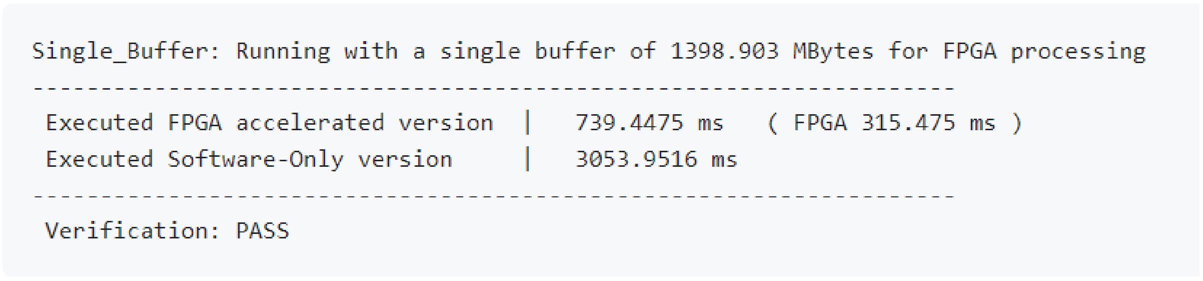
Use the information provided on the lab website, after running on hardware:

1. Parallelization = 4 words simultaneously for computing their hashes:



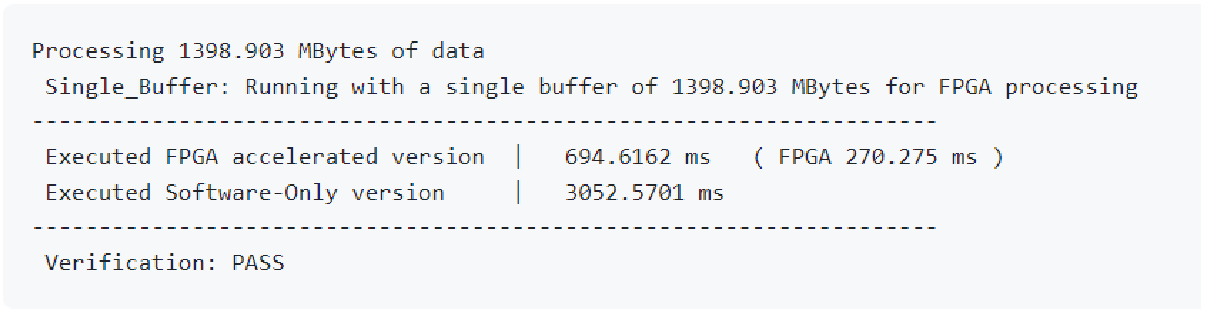
```
Loading runOnfpga_hw.xclbin
Processing 1398.903 MBytes of data
  Running with a single buffer of 1398.903 MBytes for FPGA processing
--------------------------------------------------------------------
Executed FPGA accelerated version  |   838.5898 ms   ( FPGA 447.964 ms )
Executed Software-Only version     |   3187.0354 ms
--------------------------------------------------------------------
Verification: PASS
```

2. Parallelization = 8 words simultaneously for computing their hashes:



```
Single_Buffer: Running with a single buffer of 1398.903 MBytes for FPGA processing
--------------------------------------------------------------------
 Executed FPGA accelerated version  |   739.4475 ms   ( FPGA 315.475 ms )
 Executed Software-Only version     |   3053.9516 ms
--------------------------------------------------------------------
 Verification: PASS
```

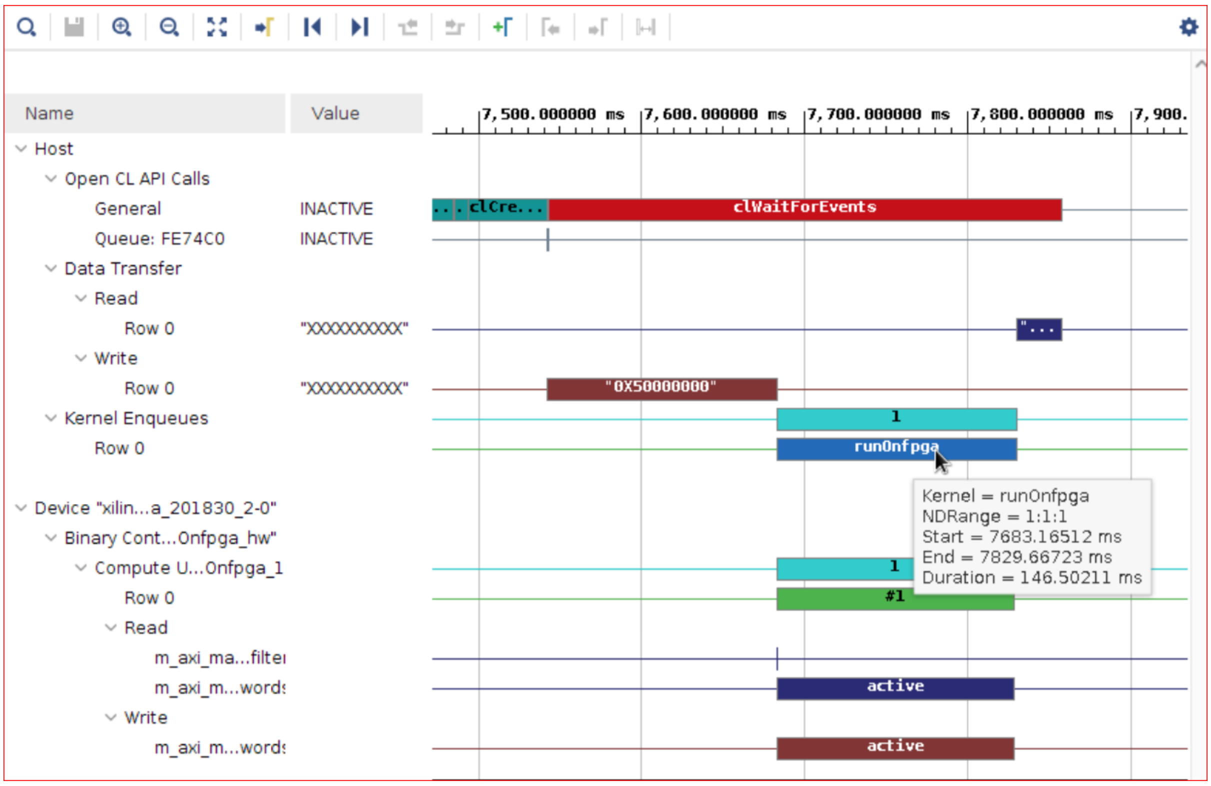
3. Parallelization = 16 words simultaneously for computing their hashes:



```
Processing 1398.903 MBytes of data
 Single_Buffer: Running with a single buffer of 1398.903 MBytes for FPGA processing
--------------------------------------------------------------------
 Executed FPGA accelerated version  |   694.6162 ms   ( FPGA 270.275 ms )
 Executed Software-Only version     |   3052.5701 ms
--------------------------------------------------------------------
 Verification: PASS
```

we can see there is a significant improvement of execution time using FPGA compared to software-only version.

timeline trace of Parallelization = 8 words:



Kernel = runOnfpga
NDRange = 1:1:1
Start = 7683.16512 ms
End = 7829.66723 ms
Duration = 146.50211 ms

The figure also shows a similar task flow as in fig.1.

From this, we can see the tasks are still sequentially implemented, therefore if we divide the “write” into several parts, overlapping “write” and “runOnfpga”, we can probably get better execution time. Next, we’ll explore using multiple sub-buffers for better performance.

Using multiple sub-buffers (split buffers):

The idea of splitting the buffer into several sub-buffers is shown in fig.3, where the input document buffer is split into 2 sub-buffers.

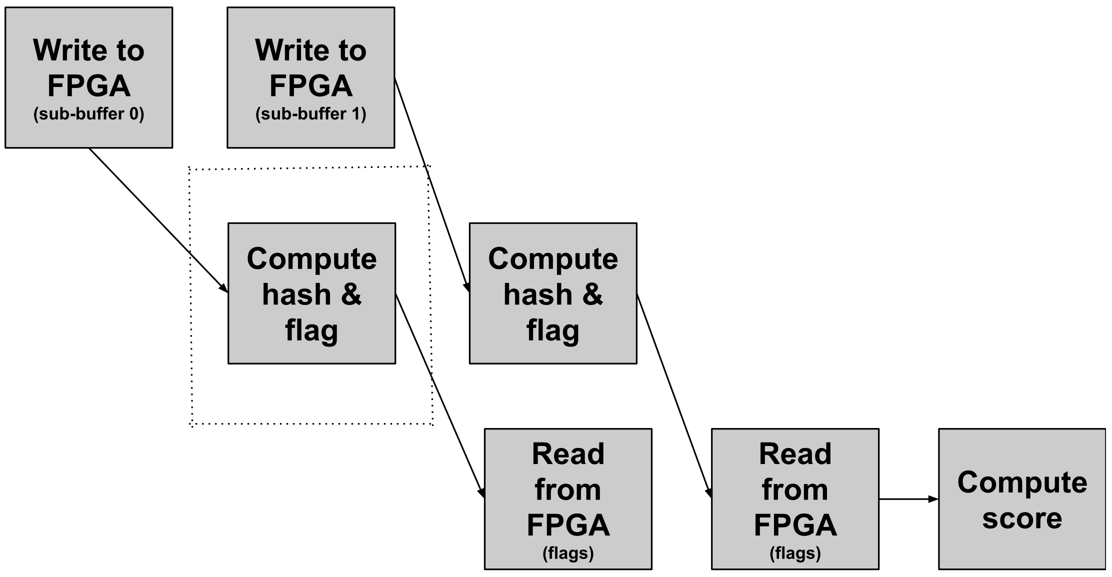
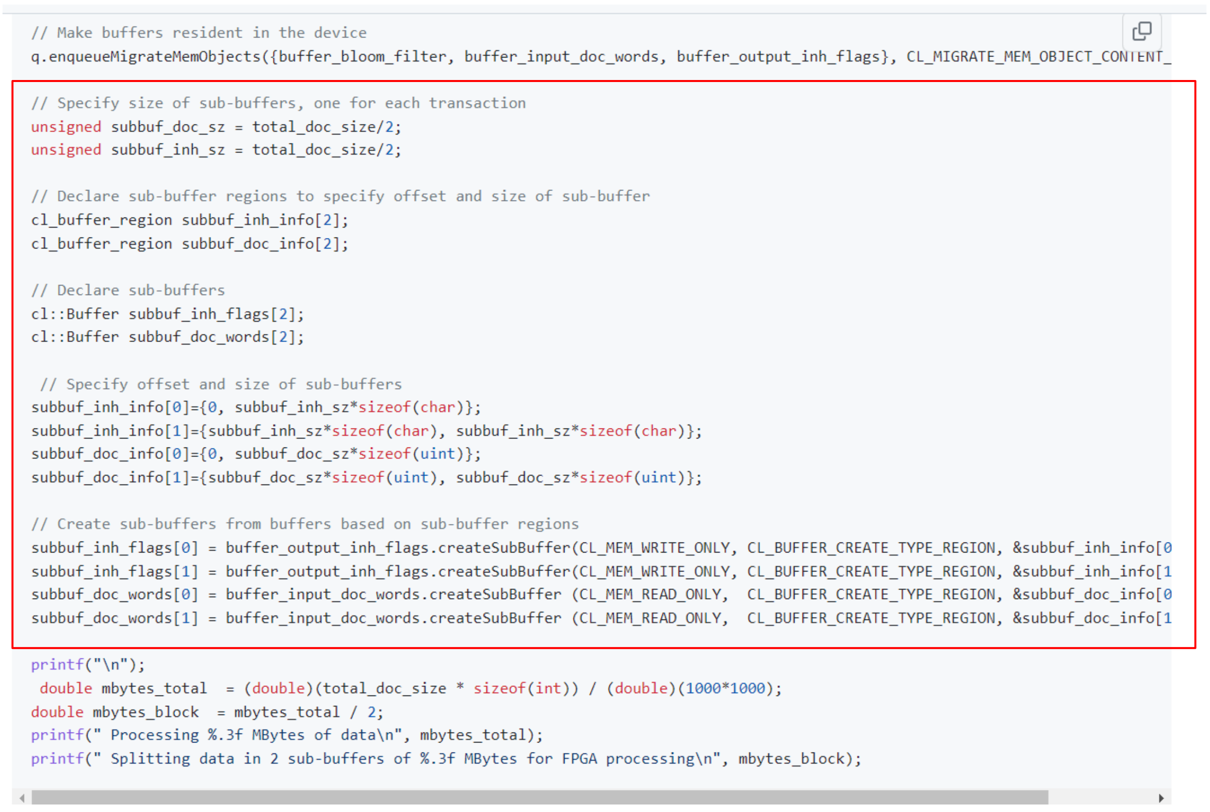


fig.3 implementation with multiple sub-buffer

The following code declares sub-buffer regions to specify offset and size of sub-buffer and creates two sub-buffers from buffers based on sub-buffer regions:



```
// Make buffers resident in the device
q.enqueueMigrateMemObjects({buffer_bloom_filter, buffer_input_doc_words, buffer_output_inh_flags}, CL_MIGRATE_MEM_OBJECT_CONTENT_

// Specify size of sub-buffers, one for each transaction
unsigned subbuf_doc_sz = total_doc_size/2;
unsigned subbuf_inh_sz = total_doc_size/2;

// Declare sub-buffer regions to specify offset and size of sub-buffer
cl_buffer_region subbuf_inh_info[2];
cl_buffer_region subbuf_doc_info[2];

// Declare sub-buffers
cl::Buffer subbuf_inh_flags[2];
cl::Buffer subbuf_doc_words[2];

 // Specify offset and size of sub-buffers
subbuf_inh_info[0]={0, subbuf_inh_sz*sizeof(char)};
subbuf_inh_info[1]={subbuf_inh_sz*sizeof(char), subbuf_inh_sz*sizeof(char)};
subbuf_doc_info[0]={0, subbuf_doc_sz*sizeof(uint)};
subbuf_doc_info[1]={subbuf_doc_sz*sizeof(uint), subbuf_doc_sz*sizeof(uint)};

// Create sub-buffers from buffers based on sub-buffer regions
subbuf_inh_flags[0] = buffer_output_inh_flags.createSubBuffer(CL_MEM_WRITE_ONLY, CL_BUFFER_CREATE_TYPE_REGION, &subbuf_inh_info[0
subbuf_inh_flags[1] = buffer_output_inh_flags.createSubBuffer(CL_MEM_WRITE_ONLY, CL_BUFFER_CREATE_TYPE_REGION, &subbuf_inh_info[1
subbuf_doc_words[0] = buffer_input_doc_words.createSubBuffer (CL_MEM_READ_ONLY,  CL_BUFFER_CREATE_TYPE_REGION, &subbuf_doc_info[0
subbuf_doc_words[1] = buffer_input_doc_words.createSubBuffer (CL_MEM_READ_ONLY,  CL_BUFFER_CREATE_TYPE_REGION, &subbuf_doc_info[1
printf("\n");
 double mbytes_total  = (double)(total_doc_size * sizeof(int)) / (double)(1000*1000);
double mbytes_block  = mbytes_total / 2;
printf(" Processing %.3f MBytes of data\n", mbytes_total);
printf(" Splitting data in 2 sub-buffers of %.3f MBytes for FPGA processing\n", mbytes_block);
```

```
// Create Events to co-ordinate read,compute and write for each iteration
vector<cl::Event> wordWait;
vector<cl::Event> krnlWait;
vector<cl::Event> flagWait;
```

First iteration for setting kernel arguments, reading, enqueuing kernel and writing:

```
//  Set Kernel Arguments, Read, Enqueue Kernel and Write for first iteration
total_size = total_doc_size/2;
load_filter=false;
kernel.setArg(3, total_size);
kernel.setArg(4, load_filter);
kernel.setArg(0, subbuf_inh_flags[0]);
kernel.setArg(1, subbuf_doc_words[0]);
q.enqueueMigrateMemObjects({subbuf_doc_words[0]}, 0, &wordWait, &buffDone);
wordWait.push_back(buffDone);
q.enqueueTask(kernel, &wordWait, &krnlDone);
krnlWait.push_back(krnlDone);
q.enqueueMigrateMemObjects({subbuf_inh_flags[0]}, CL_MIGRATE_MEM_OBJECT_HOST, &krnlWait, &flagDone);
flagWait.push_back(flagDone);
```

Second iteration:

```
//  Set Kernel Arguments, Read, Enqueue Kernel and Write for second iteration
total_size = total_doc_size/2;
load_filter=false;
kernel.setArg(3, total_size);
kernel.setArg(4, load_filter);
kernel.setArg(0, subbuf_inh_flags[1]);
kernel.setArg(1, subbuf_doc_words[1]);
q.enqueueMigrateMemObjects({subbuf_doc_words[1]}, 0, &wordWait, &buffDone);
wordWait.push_back(buffDone);
q.enqueueTask(kernel, &wordWait, &krnlDone);
krnlWait.push_back(krnlDone);
q.enqueueMigrateMemObjects({subbuf_inh_flags[1]}, CL_MIGRATE_MEM_OBJECT_HOST, &krnlWait, &flagDone);
flagWait.push_back(flagDone);
```

Use wait() function to wait until results are transmitted to host:

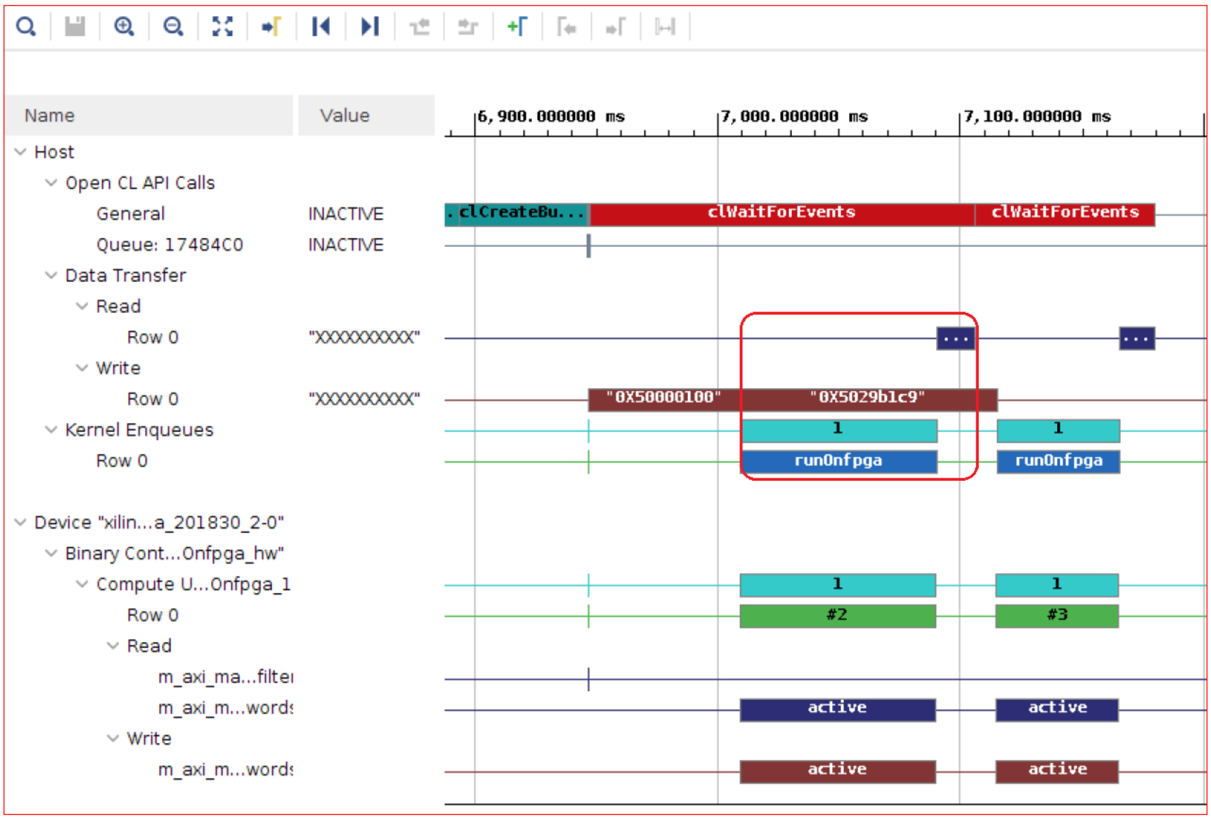
```
// Wait until all results are copied back to the host before doing the post-processing
 flagWait[0].wait();
 flagWait[1].wait();
```

Running this optimization with Parallelization = 8 words yields:

```
Processing 1398.903 MBytes of data
Split_buffer : Splitting data in 2 sub-buffers of 699.452 MBytes for FPGA processing
--------------------------------------------------------------------
Executed FPGA accelerated version  |   734.0995 ms   ( FPGA 262.363 ms )
Executed Software-Only version     |   3246.2145 ms
--------------------------------------------------------------------
Verification: PASS
```

This improves the FPGA execution time from 315 ms to 262 ms, while the overall performance does not improve much (739 ms vs 734 ms)

timeline trace:



The timeline trace is as expected as the idea in fig.3.

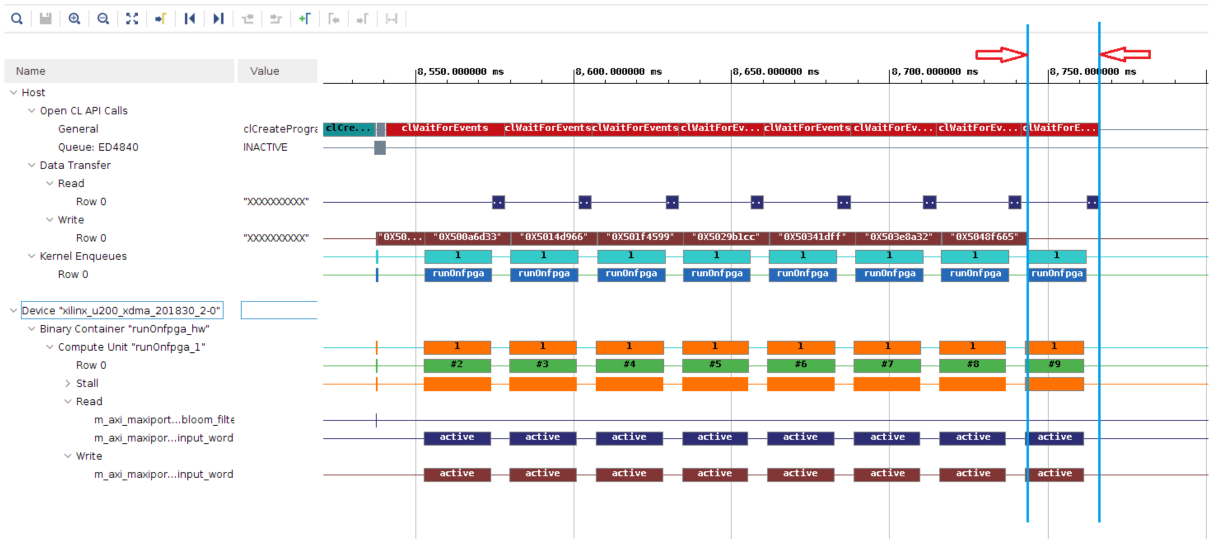
Now, we can change the code into more generic form, which allow us to split the buffer into desired number of sub-buffers:

```
// Specify size of sub buffers for each iteration
unsigned subbuf_doc_sz = total_doc_size/num_iter;
unsigned subbuf_inh_sz = total_doc_size/num_iter;

// Declare sub buffer regions to specify offset and size for each iteration
cl_buffer_region subbuf_inh_info[num_iter];
cl_buffer_region subbuf_doc_info[num_iter];

// Declare sub buffers
cl::Buffer subbuf_inh_flags[num_iter];
cl::Buffer subbuf_doc_words[num_iter];

// Define sub buffers from buffers based on sub-buffer regions
for (int i=0; i<num_iter; i++)  {
  subbuf_inh_info[i]={i*subbuf_inh_sz*sizeof(char), subbuf_inh_sz*sizeof(char)};
  subbuf_doc_info[i]={i*subbuf_doc_sz*sizeof(uint), subbuf_doc_sz*sizeof(uint)};
  subbuf_inh_flags[i] = buffer_output_inh_flags.createSubBuffer(CL_MEM_WRITE_ONLY, CL_BUFFER_CREATE_TYPE_REGION, &subbuf_inh_in
  subbuf_doc_words[i] = buffer_input_doc_words.createSubBuffer (CL_MEM_READ_ONLY,  CL_BUFFER_CREATE_TYPE_REGION, &subbuf_doc_in
}

printf("\n");
double mbytes_total  = (double)(total_doc_size * sizeof(int)) / (double)(1000*1000);
double mbytes_block  = mbytes_total / num_iter;
printf(" Processing %.3f MBytes of data\n", mbytes_total);
if (num_iter>1) {
  printf(" Splitting data in %d sub-buffers of %.3f MBytes for FPGA processing\n", num_iter, mbytes_block);
}
```

```
// Set Kernel arguments. Read, Enqueue Kernel and Write for each iteration
for (int i=0; i<num_iter; i++)
{
  cl::Event buffDone, krnlDone, flagDone;
  total_size = subbuf_doc_info[i].size / sizeof(uint);
  load_filter = false;
  kernel.setArg(0, subbuf_inh_flags[i]);
  kernel.setArg(1, subbuf_doc_words[i]);
  kernel.setArg(3, total_size);
  kernel.setArg(4, load_filter);
  q.enqueueMigrateMemObjects({subbuf_doc_words[i]}, 0, &wordWait, &buffDone);
  wordWait.push_back(buffDone);
  q.enqueueTask(kernel, &wordWait, &krnlDone);
  krnlWait.push_back(krnlDone);
  q.enqueueMigrateMemObjects({subbuf_inh_flags[i]}, CL_MIGRATE_MEM_OBJECT_HOST, &krnlWait, &flagDone);
  flagWait.push_back(flagDone);
}
```

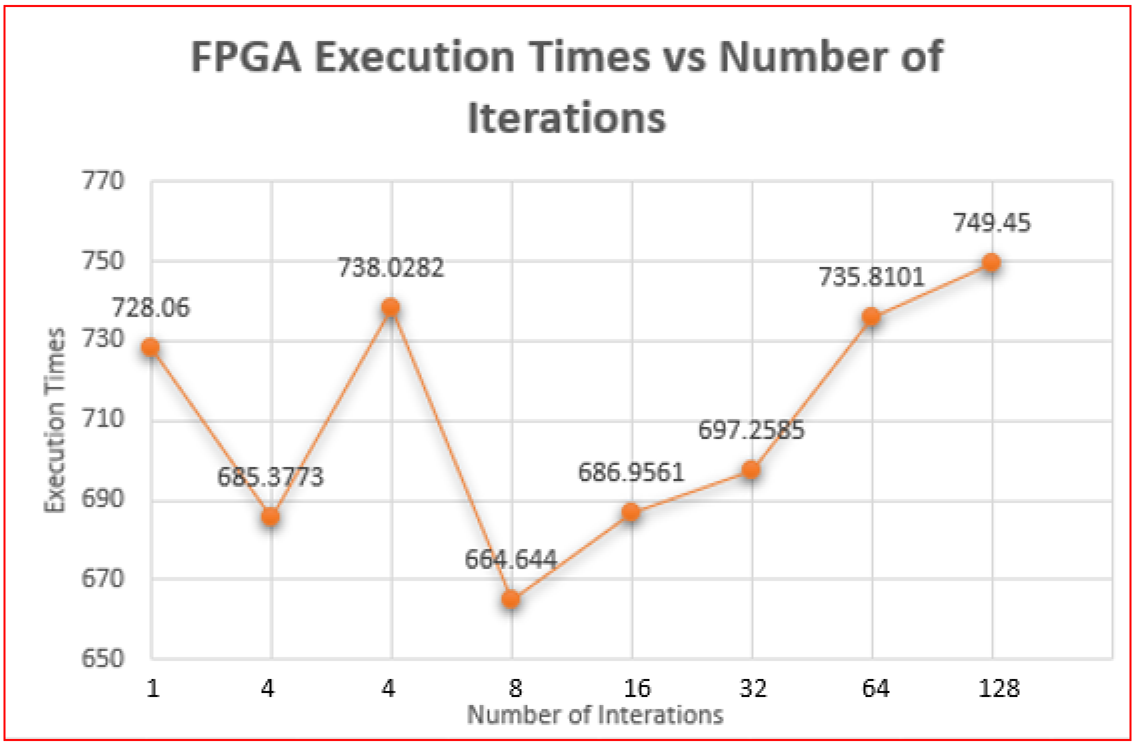
```
// Wait until all results are copied back to the host before doing the post-processing
for (int i=0; i<num_iter; i++)
{
  flagWait[i].wait();
}
```

The timeline trace of splitting into 8 sub-buffers:



We can see the red flags clWaitForEvents last the whole kernel execution time, a further optimization idea is that we don't want to wait for the whole events, but compute score right after reading a subbuffer of flags.

We can run multiple sub-buffers cases and plot the following FPGA execution time v.s. number of split buffers(sub-buffers):



source:https://github.com/Xilinx/Vitis-Tutorials/blob/2022.1/Hardware_Acceleration/Design_Tutorials/02-bloom/5_data-movement.md

We can see the case occurs when we split the buffer into 8 parts.

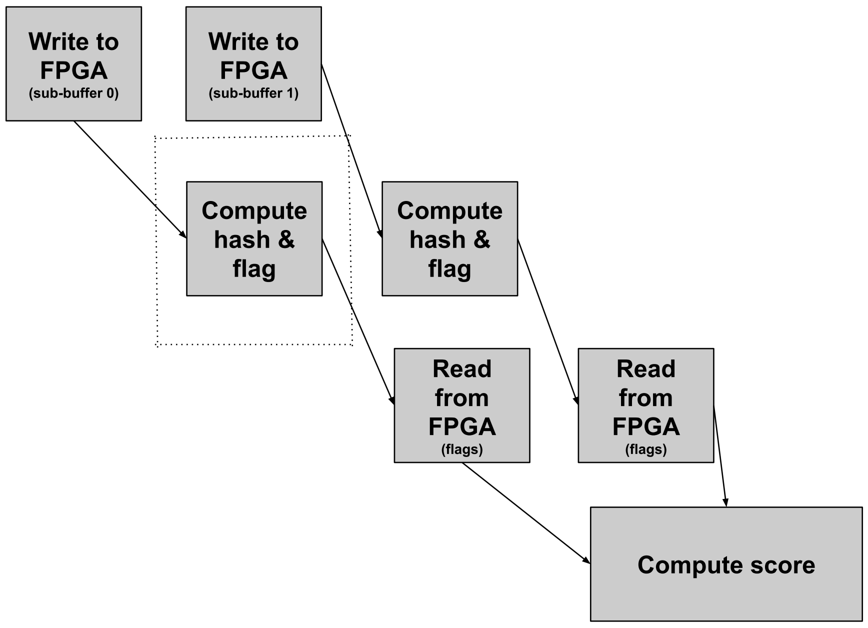
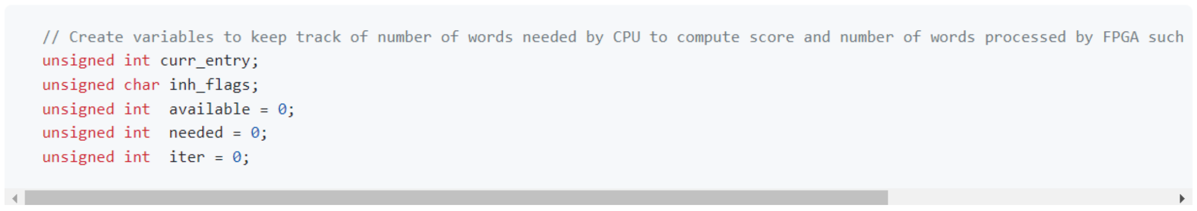


fig.4 implementation with multiple sub-buffer, overlapping compute score and read from FPGA

modify the compute score part with the following variables added:



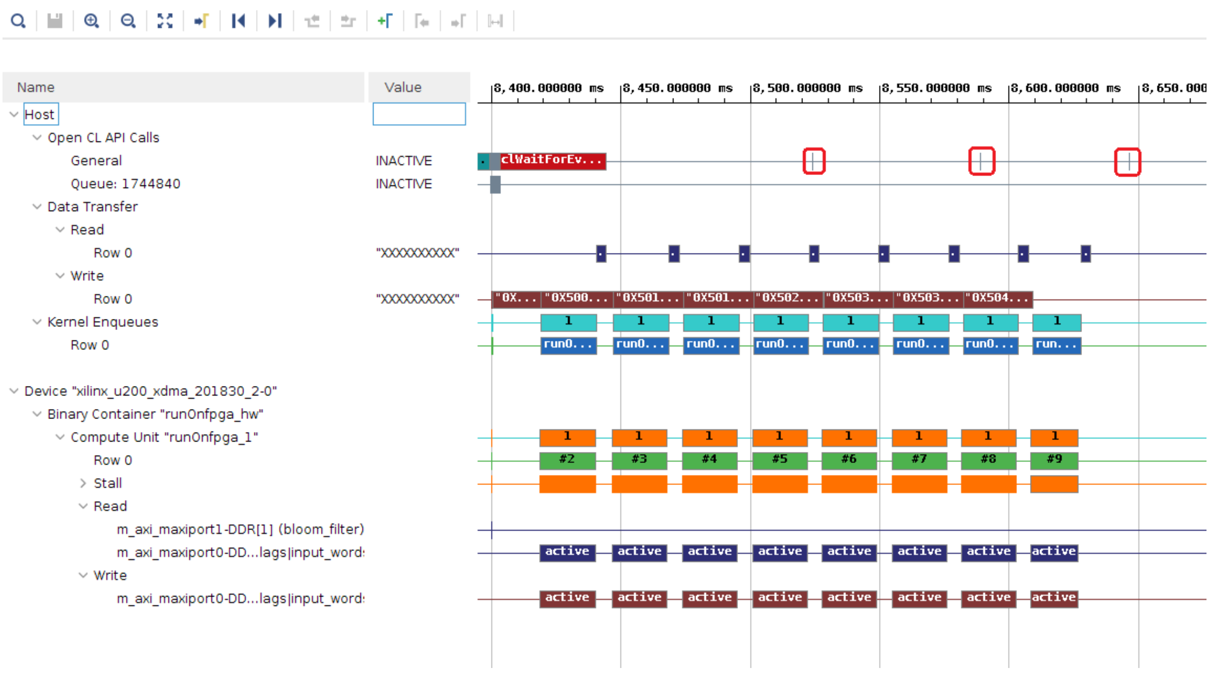
```
// Create variables to keep track of number of words needed by CPU to compute score and number of words processed by FPGA such
unsigned int curr_entry;
unsigned char inh_flags;
unsigned int  available = 0;
unsigned int  needed = 0;
unsigned int  iter = 0;
```

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

  // Calculate size by needed by CPU for processing next document score
  needed += size;

  // Check if flags processed by FPGA is greater than needed by CPU. Else, block CPU
  // Update the number of available words and sub-buffer count(iter)

  if (needed > available)
  {
    flagWait[iter].wait();
    available += subbuf_doc_info[iter].size / sizeof(uint);
    iter++;
  }

  for (unsigned i = 0; i < size ; i++, n++)
  {
    curr_entry = input_doc_words[n];
    inh_flags  = output_inh_flags[n];

    if (inh_flags)
    {
      unsigned frequency = curr_entry & 0x00ff;
      unsigned word_id = curr_entry >> 8;
      ans += profile_weights[word_id] * (unsigned long)frequency;
    }
  }
  profile_score[doc] = ans;
}
```

The needed variable indicates the number of flags needed (accumulated) to score the current document.

The available variable indicates the number of flags that has been generated.

needed > available means we should wait until more output flags are generated.



```
Processing 1398.905 MBytes of data
Splitting data in 8 sub-buffers of 174.863 MBytes for FPGA processing
--------------------------------------------------------------------
Executed FPGA accelerated version  |   427.1341 ms   ( FPGA 230.345 ms )
Executed Software-Only version     |   3057.6307 ms
--------------------------------------------------------------------
Verification: PASS
```

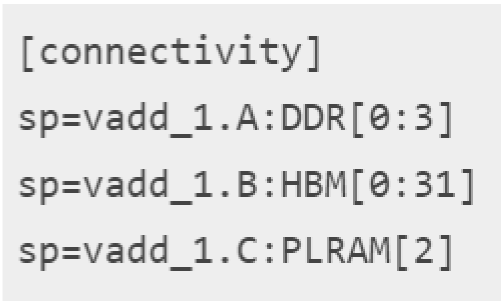
We can see the wait time in timeline trace is largely reduced, and it gives much better performance.

Using multiple DDR banks:

Since there are memory contentions because the host and kernel both accessed the same bank at the same time, we can further optimize the performance by using multiple DDR banks in ping-pong fashion, meaning that:

- The host write words to DDR bank 1 and bank 2 alternatively
- when the host is writing DDR bank 1, the kernel is reading DDR bank 2
- when the host is writing DDR bank 2, the kernel is reading DDR bank 1

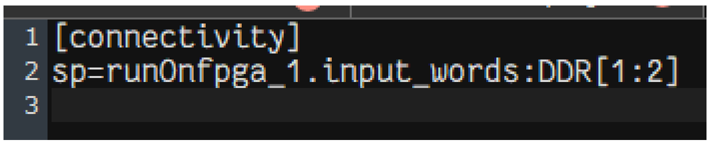
v++ linking option --connectivity.sp <arg> can be used to connect kernel arguments to specific memory resources. This option can be specified in a configuration file under the [connectivity] section head using the following format:



```
[connectivity]
sp=vadd_1.A:DDR[0:3]
sp=vadd_1.B:HBM[0:31]
sp=vadd_1.C:PLRAM[2]
```

NOTE: Any argument not explicitly mapped to a memory resource through the --connectivity.sp option is automatically connected to an available memory resource during the build process.

In this lab, this option is specified in connectivity.cfg:



```
[connectivity]
sp=runOnfpga_1.input_words:DDR[1:2]

```

The -sp option instructs the v++ linker that input_words is connected to both DDR banks 1 and 2. We should rebuild the kernel since the connectivity is changed.

There are two Xilinx extension pointer objects (cl_mem_ext_ptr_t) created. The corresponding flags will determine which DDR bank the buffer will be sent to, so that the kernel can access it.

```
cl_mem_ext_ptr_t buffer_words_ext[2];

buffer_words_ext[0].flags = 1 | XCL_MEM_TOPOLOGY; // DDR[1]
buffer_words_ext[0].param = 0;
buffer_words_ext[0].obj   = input_doc_words;
buffer_words_ext[1].flags = 2 | XCL_MEM_TOPOLOGY; // DDR[2]
buffer_words_ext[1].param = 0;
buffer_words_ext[1].obj   = input_doc_words;
```

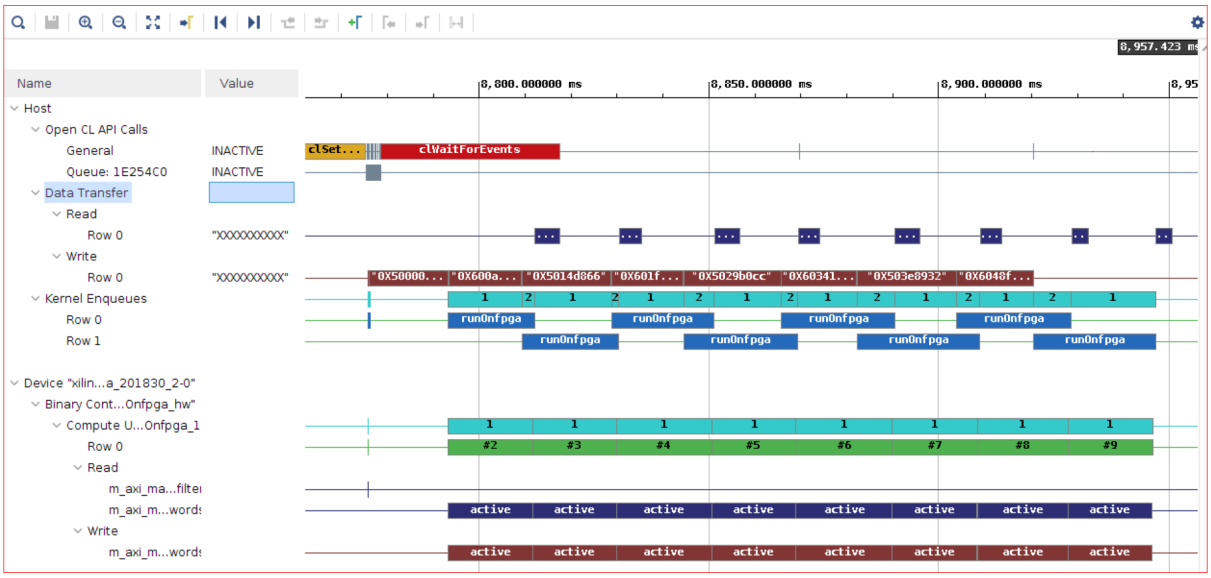
modify the code as following:

```
buffer_doc_words[0] = cl::Buffer(context, CL_MEM_EXT_PTR_XILINX | CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, total_size*sizeof(uint), &buffer_words_ext[0]);
buffer_doc_words[1] = cl::Buffer(context, CL_MEM_EXT_PTR_XILINX | CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, total_size*sizeof(uint), &buffer_words_ext[1]);
buffer_inh_flags    = cl::Buffer(context, CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY, total_size*sizeof(char),output_inh_flags);
buffer_bloom_filter = cl::Buffer(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, bloom_filter_size*sizeof(uint),bloom_filter);

// Set buffer kernel arguments (needed to migrate the buffers in the correct memory)
kernel.setArg(0, buffer_inh_flags);
kernel.setArg(1, buffer_doc_words[0]);
kernel.setArg(2, buffer_bloom_filter);

// Make buffers resident in the device
q.enqueueMigrateMemObjects({buffer_bloom_filter, buffer_doc_words[0], buffer_doc_words[1], buffer_inh_flags}, CL_MIGRATE_MEM_OBJE

// Create sub-buffers, one for each transaction
unsigned subbuf_doc_sz = total_doc_size/num_iter;
unsigned subbuf_inh_sz = total_doc_size/num_iter;

cl_buffer_region subbuf_inh_info[num_iter];
cl_buffer_region subbuf_doc_info[num_iter];
cl::Buffer subbuf_inh_flags[num_iter];
cl::Buffer subbuf_doc_words[num_iter];

for (int i=0; i<num_iter; i++) {
    subbuf_inh_info[i]={i*subbuf_inh_sz*sizeof(char), subbuf_inh_sz*sizeof(char)};
    subbuf_doc_info[i]={i*subbuf_doc_sz*sizeof(uint), subbuf_doc_sz*sizeof(uint)};
    subbuf_inh_flags[i] = buffer_inh_flags.createSubBuffer(CL_MEM_WRITE_ONLY, CL_BUFFER_CREATE_TYPE_REGION, &subbuf_inh_info[i]);
    // The doc words sub-buffers will be alternating in DDR[1] and DDR[2]
    subbuf_doc_words[i] = buffer_doc_words[i%2].createSubBuffer (CL_MEM_READ_ONLY,  CL_BUFFER_CREATE_TYPE_REGION, &subbuf_doc_inf
}
```

The doc words sub-buffers will be alternating in DDR[1] and DDR[2].

```
Processing 1398.905 MBytes of data
MultiDDR- Splitting data in 8 sub-buffers of 174.863 MBytes for FPGA processing
--------------------------------------------------------------------
Executed FPGA accelerated version  |   426.6388 ms   ( FPGA 175.113 ms )
Executed Software-Only version     |   3058.8499 ms
--------------------------------------------------------------------
Verification: PASS
```

we can see the FPGA execution time has improved.



In WRITE transactions, we can observe that when the host is writing DDR bank 1 and bank 2 alternatively.

What has been learned:

In this lab, I not only learned what is bloom filter but also learned a lot of optimization techniques, plus, Opencl API is reviewed thoroughly in the lab. I think this certainly gives me much intuition when design my own work.