Assignment 1

Q1

F1 instances include 16 nm Xilinx UltraScale Plus FPGA.

F1 FPGA specification

Name	Amount
SLR	3
System Logic Cells (K)	2,586
CLB LUTs (K)	1,182
CLB LUTs (K)	1,182
DSP Slices	6,840
URAM	270.0Mb
Total Block RAM (Mb)	75.9
DDR total capacity	64GB (4x16GB)
DDR Total BW	68GB/s
PCI Express	Gen3x16

F1 CPU specification

```
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              8
On-line CPU(s) list: 0-7
Thread(s) per core:  2
Core(s) per socket:  4
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               79
Model name:          Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz
Stepping:            1
CPU MHz:             1497.788
CPU max MHz:         3000.0000
CPU min MHz:         1200.0000
BogoMIPS:            4600.00
Hypervisor vendor:   Xen
Virtualization type: full
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            46080K
NUMA node0 CPU(s):   0-7
```

Q2

The PCIe report state that

```
[ec2-user@ip-172-31-17-89 vitis_intro_lab]$ xbutil dmatest
--------------------------------------------------------------------
Deprecation Warning:
    The given legacy sub-command and/or option has been deprecated
    to be obsoleted in the next release.

    Further information regarding the legacy deprecated sub-commands
    and options along with their mappings to the next generation
    sub-commands and options can be found on the Xilinx Runtime (XRT)
    documentation page:

    https://xilinx.github.io/XRT/master/html/xbtools_map.html

    Please update your scripts and tools to use the next generation
    sub-commands and options.
--------------------------------------------------------------------
INFO: Found total 1 card(s), 1 are usable
INFO: DMA test on [0]: xilinx_aws-vu9p-f1_shell-v04261818_201920_2
Total DDR size: 65536 MB
Buffer Size: 16 MB
Reporting from mem_topology:
Data Validity & DMA Test on bank0
Host -> PCIe -> FPGA write bandwidth = 8500.711852 MB/s
Host <- PCIe <- FPGA read bandwidth = 12200.716077 MB/s
INFO: xbutil dmatest succeeded.
```

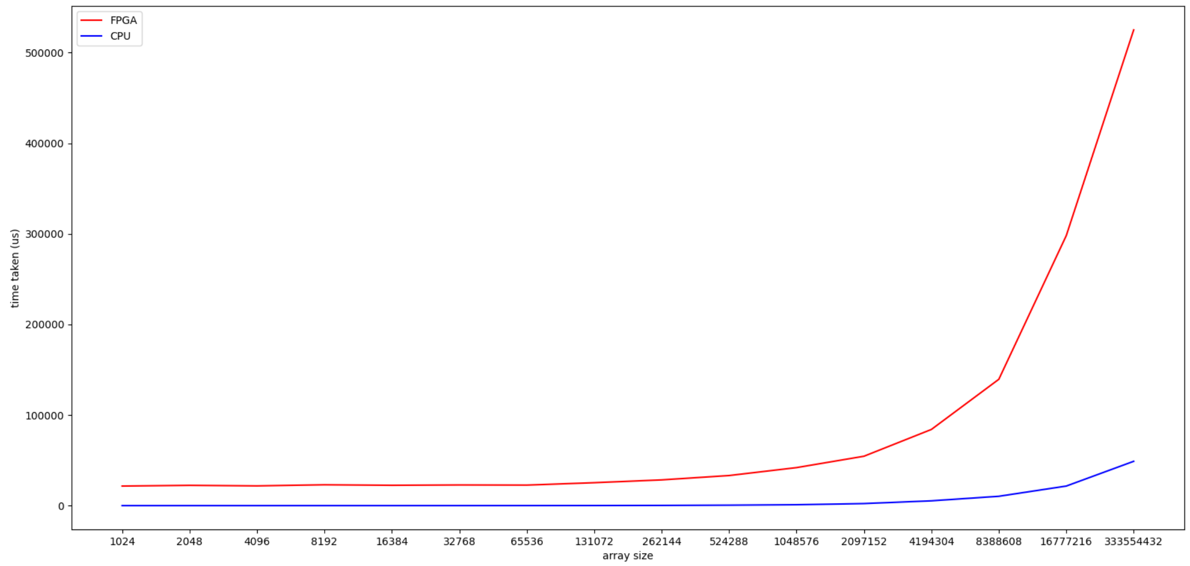
Q3

A

The results are obtained uses Chrono high-resolution clock. This result are obtained using the vadd code

Vadd CPU time vs FPGA time

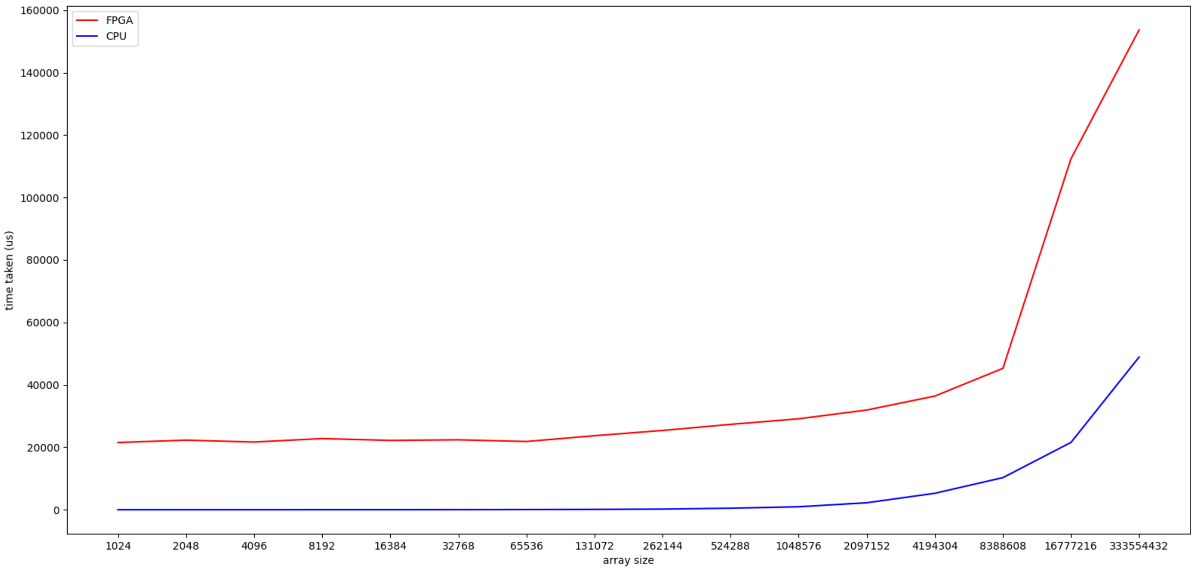
vector size	Data transfer time millisecond	FPGA time (microsecond)	CPU time microsecond
1024	22.759	150	5
2048	21.146	192	6
4096	21.972	214	7
8192	22.893	287	11
16384	21.955	344	15
32768	22.334	550	30
65536	22.545	983	58
131072	23.856	1834	114
262144	25.556	3525	230
524288	26.462	6728	492
1048576	27.561	14484	960
2097152	28.122	25934	2245
4194304	30.453	54112	5277
8388608	33.315	107125	10291
16777216	84.634	211362	21576
33554432	103.225	422904	48914



In wide add we use uint512_t which enable us to directly take 512 bits at a time. Also wide add use memory banking which help in reducing the congestion of reading the memory and different variables can read data in parallel.

wide add CPU time vs FPGA time

Aa vector size	# Data transfer time millisecond	# FPGA time (microsecond)	# CPU time microsecond
1024	21.449	97	5
2048	22.156	141	6
4096	21.562	129	7
8192	22.693	112	11
16384	22.065	146	15
32768	22.235	169	30
65536	21.652	215	58
131072	23.452	266	114
262144	24.856	545	230
524288	26.462	892	492
1048576	27.451	1687	960
2097152	28.562	3381	2245
4194304	29.953	6464	5277
8388608	32.315	12985	10291
16777216	86.654	25797	21576
33554432	102.255	51437	48914



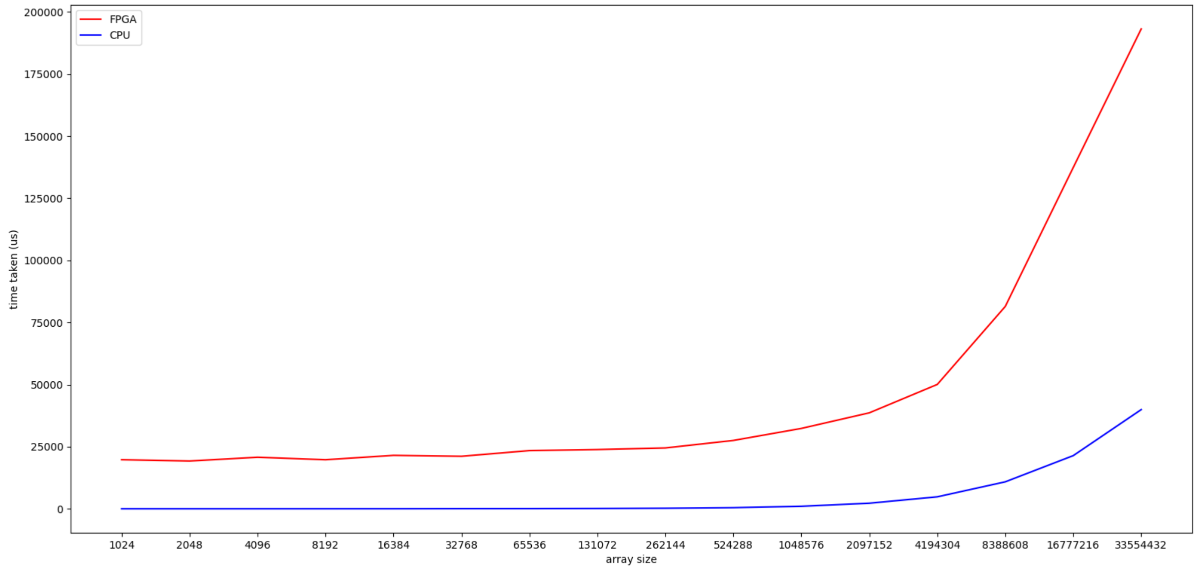
Here we can see that we are able to obtain better results compared to simple vadd code.

B

Using the wide add code we modified it so that is able to read floating point data type and instead of addition we do multiplication.

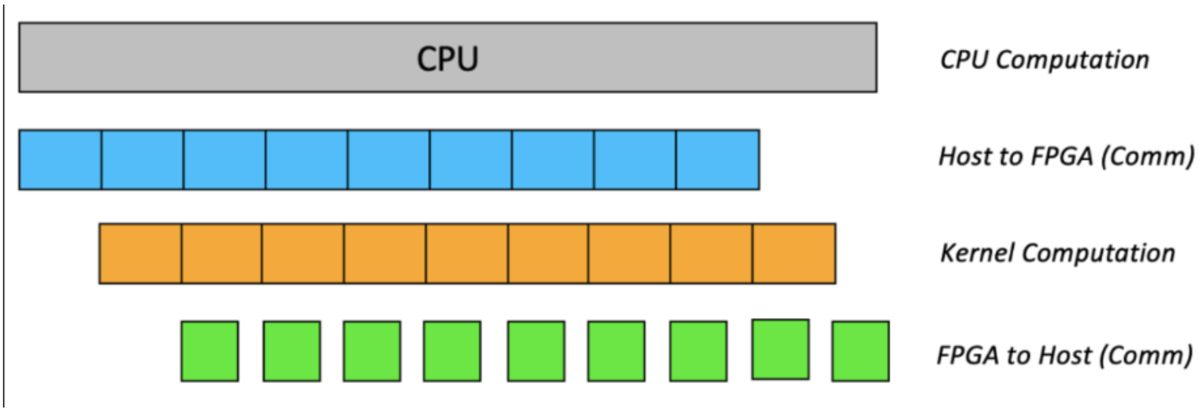
float mult CPU time vs FPGA time

Aa vector size	# Data transfer time millisecond	# FPGA time (microsecond)	# CPU time microsecond
1024	19.65	105	2
2048	19.145	129	2
4096	20.617	133	4
8192	19.627	120	7
16384	21.363	145	14
32768	20.988	154	47
65536	23.206	217	58
131072	23.548	301	114
262144	24.045	447	225
524288	26.696	817	455
1048576	30.811	1541	1011
2097152	32.68	2951	2244
4194304	38.241	5820	4816
8388608	41.911	11541	10855
16777216	62.401	22981	21400
33554432	75.247	45926	39931



Q4

Using the code of wide add instead of sending the data and then reading the result all at once we divide the data into n chunks. this pre

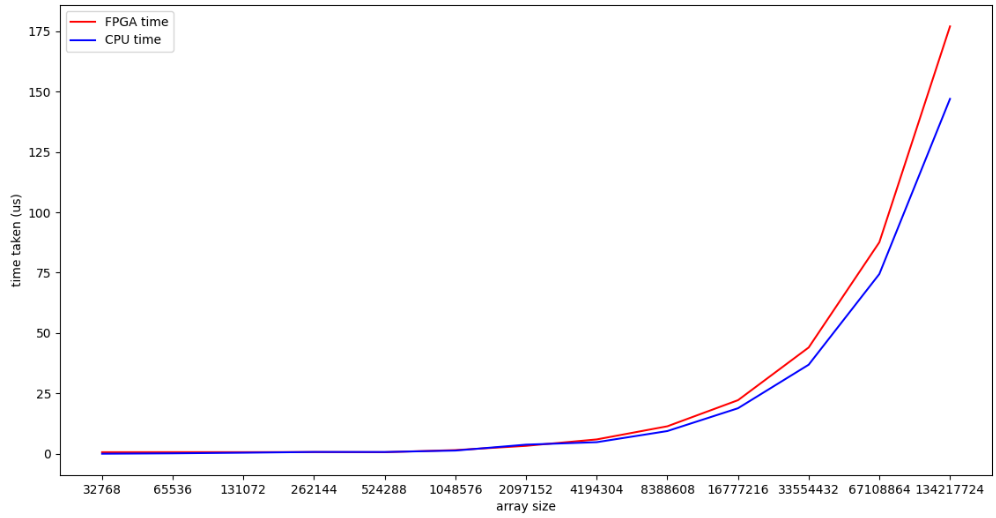


This help us in reducing the total time taken to compute the data

Parallel CPU vs FPGA

Array size	FPGA time (ms)	Communication time (ms)	CPU time(ms)
32768	0.74	20.34	0.105
65536	0.761	19.817	0.263
131072	0.752	20.976	0.521
262144	0.809	22.369	0.832
524288	0.743	23.57	0.812
1048576	1.615	28.952	1.418
2097152	3.368	30.03	3.857
4194304	6.026	35.132	4.886
8388608	11.461	43.015	9.488
16777216	22.264	75.487	18.929
33554432	44.067	135.372	36.941
67108864	87.6	254.892	74.462

Array size	FPGA time (ms)	Communication time (ms)	CPU time(ms)
134217724	176.932	450.909	146.944



Q5

Kernel code

```
#include <stdio.h>
#include <string.h>

extern "C"
{
    void matrix_multiply(float *data, float *query, int N, float *res)
    {
#pragma HLS INTERFACE m_axi port = data max_read_burst_length = 32 offset = slave bundle = gmem
#pragma HLS INTERFACE m_axi port = query max_read_burst_length = 32 offset = slave bundle = gmem1
#pragma HLS INTERFACE m_axi port = res max_write_burst_length = 32 offset = slave bundle = gmem2
#pragma HLS INTERFACE s_axilite port = data bundle = control
#pragma HLS INTERFACE s_axilite port = query bundle = control
#pragma HLS INTERFACE s_axilite port = res bundle = control
#pragma HLS INTERFACE s_axilite port = N bundle = control
#pragma HLS INTERFACE s_axilite port = return bundle = control
        float query_local[256]; // 4 queries
        float query_cal;
        // float data_val[256];
        int n = N;
        Load_Query: for (int i = 0; i < 256; i++)
        {
      #pragma HLS pipeline
      #pragma HLS unroll factor=16
            query_local[i] = query[i] ;
        }

    Main:
        for (int j = 0; j < n; j++)
        {
          //#pragma HLS pipeline
      #pragma HLS LOOP_TRIPCOUNT min = 16 max = 1024

        query_cal =0 ;

        M_cp:
            for (int k = 0; k < 256; k++)
            {
        #pragma HLS pipeline
              #pragma HLS unroll factor=16

                query_cal += data[k + j * 256] * query_local[k] ;
            }
```

```
            res[j] =query_cal ;
        }
    }
}
```

Host code

```
#include <vector>
#include <random>
#include "xcl2.hpp"
#include <algorithm>
// #include <cstring>
#include <string>
#include <iostream>
#include <chrono>

using namespace std;

int getans(float *data , float *query , int N ){
  int ans = 0;
  float val = 0 ;
  std::cout<< N << std::endl ;
  for (int i =0 ; i< N ;i++){
    float tem = 0 ;
    for (int j= 0 ; j < 256 ; j++){
      tem += data[i* 256 + j ] * query[j] ;
    }
    if (tem  >  val){
      val = tem;
      ans = i ;
    }
    //std::cout << tem << std::endl;
  }
  return ans ;
}

int subdivide_buffer(std::vector<cl::Buffer> &data_bufs, cl::Buffer data_in,
    std::vector<cl::Buffer>&res_bufs , cl::Buffer res_in , uint &num_div , int N){
  num_div = (N +511)/512;
  cl_buffer_region r1, r2 ;
  int err;
  r1.origin = 0  ;
  r2.origin = 0 ;

  r1.size = 256 * 4*16;
  r2.size = 4 *16 ;
  for(uint i =0 ;i< num_div  ; i++){
    if (i == num_div -1 ){
      r1.size =data_in.getInfo<CL_MEM_SIZE>() - i*16*4*256;
      r2.size =res_in.getInfo<CL_MEM_SIZE>() - i*16*4;
    }
    data_bufs.push_back(data_in.createSubBuffer(static_cast<cl_mem_flags>(CL_MEM_READ_ONLY) , CL_BUFFER_CREATE_TYPE_REGION, &r1, &err)
    if (err != CL_SUCCESS){
      exit(-1) ;
    }
    res_bufs.push_back(res_in.createSubBuffer(static_cast<cl_mem_flags>(CL_MEM_WRITE_ONLY), CL_BUFFER_CREATE_TYPE_REGION, &r2, &err));
    if (err != CL_SUCCESS){
      exit(-1) ;
    }
    r1.origin += r1.size;
    r2.origin += r2.size ;
  }
  return 0 ;
}

int enqueue_subbuf_vadd(cl::CommandQueue &q, cl::Kernel &krnl,
    cl::Event &event, cl::Buffer a, cl::Buffer b, cl::Buffer c)
{
    // Get the size of the buffer
    cl::Event k_event, m_event;
    std::vector<cl::Event> krnl_events;

    static std::vector<cl::Event> tx_events, rx_events;
    std::vector<cl::Memory> c_vec;
    size_t size;
    size = a.getInfo<CL_MEM_SIZE>();

    std::vector<cl::Memory> in_vec;
    in_vec.push_back(a);
    in_vec.push_back(b);
    q.enqueueMigrateMemObjects(in_vec, 0, &tx_events, &m_event);
    krnl_events.push_back(m_event);
    tx_events.push_back(m_event);
```

```
    if (tx_events.size() > 1)
    {
        tx_events[0] = tx_events[1];
        tx_events.pop_back();
    }

    krnl.setArg(0, a);
    krnl.setArg(1, b);
    krnl.setArg(3, c);
    krnl.setArg(2, (int)(size / (4 * 256)));
    q.enqueueTask(krnl, &krnl_events, &k_event);
    krnl_events.push_back(k_event);
    if (rx_events.size() == 1)
    {
        krnl_events.push_back(rx_events[0]);
        rx_events.pop_back();
    }
    c_vec.push_back(c);
    q.enqueueMigrateMemObjects(c_vec, CL_MIGRATE_MEM_OBJECT_HOST, &krnl_events, &event);
    rx_events.push_back(event);

    return 0;
}

int main(int argc, char **argv)
{
    if (argc < 2 || argc > 4)
    {
        cout << "Incorrect format" << endl;
        return EXIT_FAILURE;
    }
    char *binaryFile = argv[1];
    int N  ;
    if (argc == 3)
    {
        try
        {
            N = stoi(argv[2]);
        }
        catch (invalid_argument val)
        {
            cerr << "Invalid argument" << endl;
            return -1;
        }
    }
    else
    {
        N = 256;
    }


    cl::Device device;
    cl::Context context;
    cl::CommandQueue q;
    cl::Program program;
    cl::Kernel krnl;
    cl_int err;

    auto devices = xcl::get_xil_devices();

    auto fileBuf = xcl::read_binary_file(binaryFile);
    cl::Program::Binaries bins{{fileBuf.data(), fileBuf.size()}};
    // bool valid_device = false;
    for (unsigned int i = 0; i < devices.size(); i++)
    {
        device = devices[i];
        // Creating Context and Command Queue for selected Device
        OCL_CHECK(err, context = cl::Context(device, NULL, NULL, NULL, &err));
        OCL_CHECK(err,
                  q = cl::CommandQueue(context, device,
                                      CL_QUEUE_PROFILING_ENABLE |
                                          CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE,
                                      &err));
        std::cout << "Trying to program device[" << i
                  << "]: " << device.getInfo<CL_DEVICE_NAME>() << std::endl;
        cl::Program program(context, {device}, bins, NULL, &err);
        if (err != CL_SUCCESS)
        {
            std::cout << "Failed to program device[" << i << "] with xclbin file!\n";
        }
        else
        {
            std::cout << "Device[" << i << "]: program successful!\n";
            // Creating Kernel
            OCL_CHECK(err, krnl = cl::Kernel(program, "matrix_multiply", &err));
            // valid_device = true;
            break; // we break because we found a valid device
```

```
        }
    }

    cl::Buffer data_buf(context,
           static_cast<cl_mem_flags>(CL_MEM_READ_ONLY),
           N*256 * sizeof(float),
           NULL,
           NULL);
  cl::Buffer query_buf(context,
           static_cast<cl_mem_flags>(CL_MEM_READ_ONLY),
           256 * sizeof(float),
           NULL,
           NULL);
  cl::Buffer res_buf(context,
           static_cast<cl_mem_flags>(CL_MEM_READ_WRITE),
           N* sizeof(float),
           NULL,
           NULL);
    krnl.setArg(0, data_buf);
  krnl.setArg(1, query_buf);
  krnl.setArg(2 ,N)  ;
  krnl.setArg(3, res_buf);
    float *data = (float *)q.enqueueMapBuffer(data_buf,
                         CL_TRUE,
                         CL_MAP_WRITE,
                         0,
                         N*256 * sizeof(float));
  float *query = (float *)q.enqueueMapBuffer(query_buf,
                         CL_TRUE,
                         CL_MAP_WRITE,
                         0,
                         256* sizeof(float));


    // Filling the data
    float temp_sum = 0;
    for (int i = 0; i < N; i++)
    {
        temp_sum = 0 ;
        for (int j = 0; j < 256; j++)
        {
            data[i * 256 + j] = static_cast<float>(rand()) / static_cast<float>(RAND_MAX);
            temp_sum += data[i * 256 + j] * data[i * 256 + j];
        }
        temp_sum = sqrt(temp_sum);
        for (int j = 0; j < 256; j++)
        {
            data[i * 256 + j] /= temp_sum;
        }
    }
    temp_sum = 0 ;
    for (int j = 0; j < 256; j++)
    {
        query[j] = static_cast<float>(rand()) / static_cast<float>(RAND_MAX);
        temp_sum +=query[j]*query[j];
    }
    temp_sum = sqrt(temp_sum);
    for (int j = 0; j < 256; j++)
    {
      query[j] /= temp_sum;
    }

    std::chrono::high_resolution_clock::time_point st = std::chrono::high_resolution_clock::now() ;
    int ans_val = getans(data , query , N);
    std::chrono::duration<float, std::milli> duration = std::chrono::high_resolution_clock::now() - st;


    std::cout << "CPU time:\t" << duration.count()<< " ms "<<endl;
    q.enqueueUnmapMemObject(data_buf, data);
  q.enqueueUnmapMemObject(query_buf,query);

  std::vector<cl::Buffer> data_bufs , res_bufs;
  uint num_div;
  subdivide_buffer(data_bufs, data_buf, res_bufs, res_buf, num_div, N);
  cl::Event  kernel_events[num_div];

  for (uint i = 0; i < num_div; i++)
  {
    cl::Buffer t2 =data_bufs[i];
    cl::Buffer te = res_bufs[i]  ;
    enqueue_subbuf_vadd(q, krnl, kernel_events[i], t2 , query_buf, te);
  }
  st = std::chrono::high_resolution_clock::now() ;
  clWaitForEvents(num_div, (const cl_event *)&kernel_events);
  duration = std::chrono::high_resolution_clock::now() - st;
```

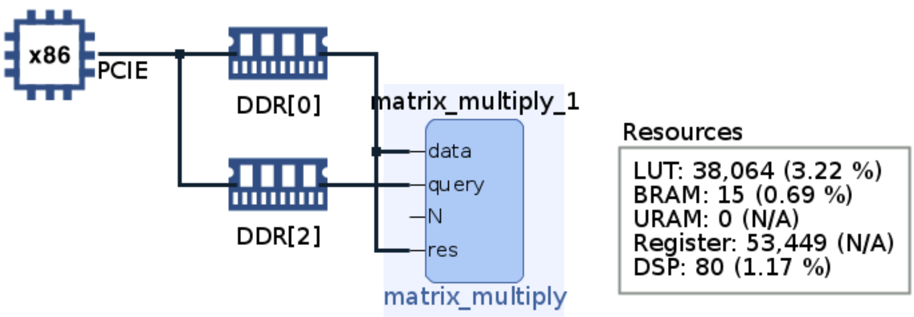
```
      std::cout << "FPGA time:\t" << duration.count()<< " ms "<<endl;
    int ker_ans =0 , tem_ans =0  ;
    float *res = (float *)q.enqueueMapBuffer(res_buf,
                           CL_TRUE,
                           CL_MAP_READ,
                           0,
                           N* sizeof(float));
    //calculate ans
    for (int i= 0 ; i< N ; i++){
      if (tem_ans < res[i]){
        tem_ans = res[i] ;
        ker_ans = i ;
      }
    }

    q.enqueueUnmapMemObject(res_buf , res) ;

      std::cout<< "Finish \n" ;
      return 0  ;
  }
```

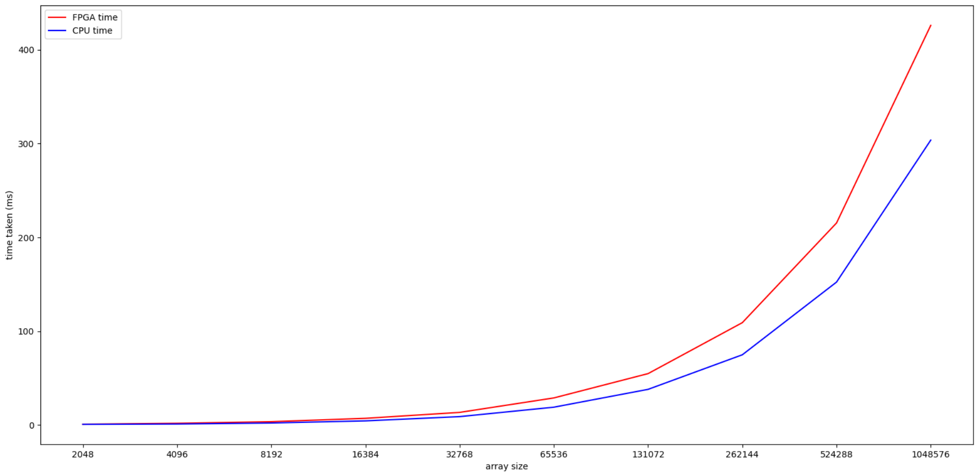
Let us go through the code

- We see that computing dot product is the most compute-intensive task and we create a kernel for it.
- First, we load the query on the local memory for fast io. (Load_Query)
- As we know that at max 512 bits can be transferred at a time we unroll the loop by a factor of 16 and then pipeline it.
- To reduce the data transfer time we divide the buffer into 512 size chunks.
- We also bank the data in different ddr so that we can access them in parallel.



Code stastics

Database size	Latency (ms)	Throughput (Mb/s)
2048	0.864931	9.03251
4096	1.85606	8.41835
8192	3.58039	8.7281
16384	7.16698	8.72055
32768	13.4871	9.26808
65536	28.9076	8.64824
131072	54.8245	9.12001
262144	109.119	9.16434
524288	215.357	9.28689
1048576	425.91	9.39165



Roofline analysis

For now, let us assume all floating-point multiplication and accumulation happen in DSPs. In the AWS F1 instance, the FPGA has 6800 DSPs. These DSPs compute 1 Mac operating in 1 clock cycle.

Clock frequency = 250 MHz

Peak computation performance = 6800* 250 * 10^6= 1.7 TFlops

The DDR memory bandwidth = 16 GB/s

The Operational Intensity = 2(multiplication + addition)/ 4 (4 as we are reading or writing 4byte in parallel) = $1/2$

Attainable GFlops = $1/2 * 16 = 8$ GFlops

So the code is memory-bound as 4GFlops is very less compared to 1.7 TFlops this is assuming we are doing the best designing we can.

The current avg throughput is 8.977 Mb/s