CS259 Lab2 Report

Yaying Ye yye33@g.ucla.edu 605034025

Introduction

The goal of this lab is to learn and get familiar with HLS and AFI on AWS given a practical machine learning algorithm. In this lab, the Knn algorithm is applied to handwriting recognition of digits. Given a 7*7 bitmap representation of a handwritten digit, the distance is the sum of difference in the bitmap and 1800 instance of the 10 digit is used to train the Knn model. Our goal is to optimize the performance of this algorithm by adding HLS syntax or rewriting the algorithm to be more computational efficient. The first part of the report analyzes the original code and find out possible optimization. The second part of the report discusses detail of re-written algorithm and its enhanced performance.

Original Code Analysis:

In the original knn algorithm, there is three nested for loops (iterate through 0-10 digits and each 1800 training instance of each digit) with 18000 iterations each and all of them contains memory access that introduce high latency. Our attempt to optimize is placed on reducing time of iterations of these nested loop, reducing the memory read latency and enabling parallel computation. The original algorithm is re-written to enhance performance in the following way.

Optimization:

1) Memory optimization
   To reduce the latency spent on memory read, the optimized algorithm allocates 180 burst size of local memory and use double buffeting to load and compute at the same time. The memory is load with HLS pipeline and the computation is done in local memory.

   A snippet of memory load function as below:

```
void Load (const bool enable,unsigned long* data_dram, unsigned long* data_local){
#pragma HLS inline off
    if(enable){
load:
    for(int i=0;i<kBurstSize;++i){
#pragma HLS pipeline
        data_local[i]=data_dram[i];
```

2) Combination of three nested loop
   To reduce iteration in total, three nested for loop can be combined in to one 10*1800 nested for loop. Since the diff, dist and update of knn_mat of one instance is independent to the other instance. The rewritten compute function includes diff: Xoring with test image, dist: calculating distance and update: update the knn_mat if dis is smaller than existing knn_mat. After this optimization, the total iteration can be reducing 3 times of the original code.
3) Unroll computation
   As mention in above, the computation function of one instance is independent to the other instances. Parallelism can be applied to increase throughput of pipelining. The optimized algorithm use tile size of 10 to unroll the 180 burst size of computation in pipeline.

   A snippet of compute function as below:

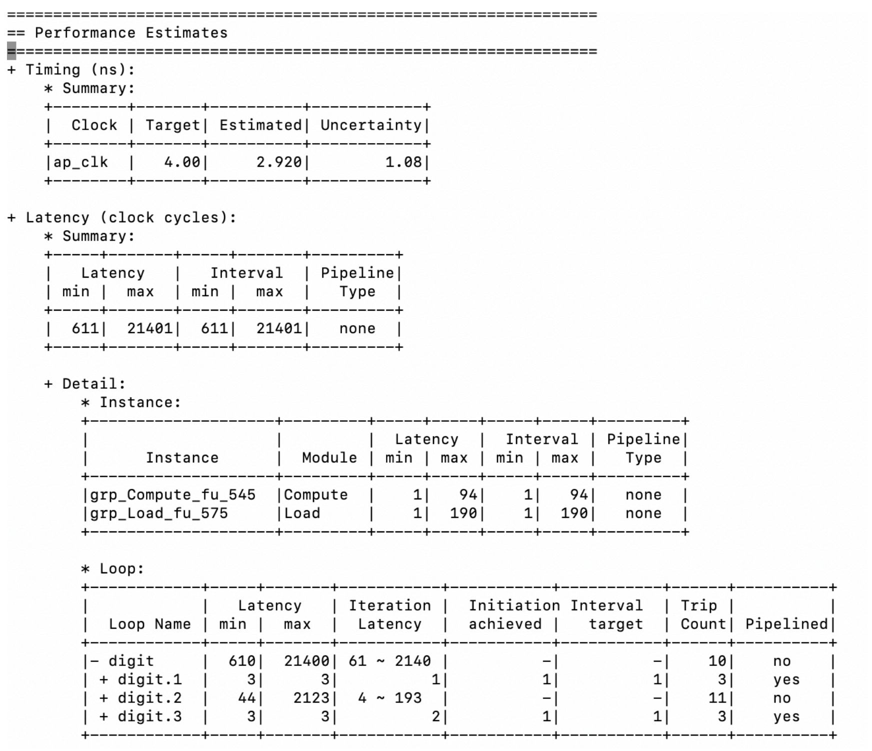
```
#pragma HLS inline off
    if(enable){
    for(int i=0;i<kBurstSize/kTileSize;++i) {
#pragma HLS pipeline
        for(int j=0;j<kTileSize;++j){
#pragma HLS unroll
            data_local[i*kTileSize+j]=data_local[i*kTileSize+j]^test_image;
```

4) Knn_mat algorithm rewrite
   In the original code, the knn_mat is iterated at each instance to update the min distance in knn_mat. This yield high latency since knn_mat is read from memory every time. Instead of loop through the knn_mat every instance, we store three minimum value in local array min[3] and we update the array in each iteration of instance. Finally, we do one memory write outside of the 1800 for loop to write the min values to knn_mat of each digit. After re-written the algorithm, the original process of the init loop is no long necessary.

Result

The original max latency is over 40,000. With the optimization applied above, the new max latency is 21401.

Performance report as follow:



```
================================================================
== Performance Estimates
================================================================
+ Timing (ns):
    * Summary:
    +--------+-------+----------+------------+
    |  Clock | Target| Estimated| Uncertainty|
    +--------+-------+----------+------------+
    |ap_clk  |   4.00|     2.920|        1.08|
    +--------+-------+----------+------------+

+ Latency (clock cycles):
    * Summary:
    +-----+-------+-----+-------+---------+
    |   Latency    |   Interval   | Pipeline|
    | min |  max  | min |  max  |   Type  |
    +-----+-------+-----+-------+---------+
    |  611|  21401|  611|  21401|   none  |
    +-----+-------+-----+-------+---------+

    + Detail:
        * Instance:
        +--------------------+---------+-----+-----+-----+-----+---------+
        |                    |         |  Latency  |  Interval | Pipeline|
        |      Instance      |  Module | min | max | min | max |   Type  |
        +--------------------+---------+-----+-----+-----+-----+---------+
        |grp_Compute_fu_545  |Compute  |    1|   94|    1|   94|   none  |
        |grp_Load_fu_575     |Load     |    1|  190|    1|  190|   none  |
        +--------------------+---------+-----+-----+-----+-----+---------+

        * Loop:
        +------------+-----+-------+-----------+-----------+-----------+-----+----------+
        |            |    Latency  | Iteration |  Initiation Interval  | Trip |          |
        |  Loop Name | min |  max  |  Latency  |  achieved |   target  | Count| Pipelined|
        +------------+-----+-------+-----------+-----------+-----------+-----+----------+
        |- digit     |  610|  21400| 61 ~ 2140 |          -|          -|   10|    no    |
        | + digit.1  |    3|      3|          1|          1|          1|    3|    yes   |
        | + digit.2  |   44|   2123|  4 ~ 193  |          -|          -|   11|    no    |
        | + digit.3  |    3|      3|          2|          1|          1|    3|    yes   |
        +------------+-----+-------+-----------+-----------+-----------+-----+----------+
```