IoT final project

**Objective：**

* Master the multi-core usage of low-power development boards (GAP8)
* Understand the basic neural network structure
* Learn how to analyze the performance

**Marking Criteria:**

* Master GAP8 multi-core control (run different functions on different cores)
* Realize a simple neural network (see un small example at https://github.com/stephanie3han/IoT.git)
* Implement the neural network on GAP8
* Use simulated data for training and prediction
* Performance analysis of GAP8 (cycles, frequency, time… Please find the attached document)

**Tips:** You can use the **/gap\_sdk/examples/….../helloworld.c** example as a framework and template to realize the functions and compile on GAP8.

GAP8 does not provide a complete implementation of the standard C language library, so for initialization of weight and bias, you can create your own random function or just give it a value as beginning on GAP8.

There will be a 15 mins report on December 18th for each group, and you need to prepare a report and PPT (Please upload your project to your GitHub account and indicate the project link in the report).

**Bonus:**

* Use flex sensor data as training and test data (see Teams group file)
* Multi-core asynchronous control, such as training on core1, generating random prediction data sets on core2, and implementing the prediction function on core3...

**Attachment 1**

**Performance counters**

The cluster and the FC have a range of performance counters that can be used to measure cycle accurate performance on the GVSOC simulator and on the GAP8 hardware.

The following counters are supported on GAP8:

**PI\_PERF\_CYCLES** Total number of cycles (also includes the cycles where the core is sleeping). Be careful that this event is using a timer shared within the cluster, so resetting, starting or stopping it on one core will impact other cores of the same cluster.

**PI\_PERF\_ACTIVE\_CYCLES** Counts the number of cycles the core was active (not sleeping).

**PI\_PERF\_INSTR** Counts the number of instructions executed.

**PI\_PERF\_LD\_STALL** Number of load data hazards.

**PI\_PERF\_JR\_STALL** Number of jump register data hazards.

**PI\_PERF\_IMISS** Cycles waiting for instruction fetches, i.e., number of instructions wasted due to non-ideal caching.

**PI\_PERF\_LD** Number of data memory loads executed. Misaligned accesses are counted twice.

**PI\_PERF\_ST** Number of data memory stores executed. Misaligned accesses are counted twice.

**PI\_PERF\_JUMP** Number of unconditional jumps (j, jal, jr, jalr).

**PI\_PERF\_BRANCH** Number of branches. Counts both taken and not taken branches.

**PI\_PERF\_BTAKEN** Number of taken branches.

**PI\_PERF\_RVC** Number of compressed instructions executed.

**PI\_PERF\_LD\_EXT** Number of memory loads to EXT executed. Misaligned accesses are counted twice. Every non-TCDM access is considered external (cluster only).

**PI\_PERF\_ST\_EXT** Number of memory stores to EXT executed. Misaligned accesses are counted twice. Every non-TCDM access is considered external (cluster only).

**PI\_PERF\_LD\_EXT\_CYC** Cycles used for memory loads to EXT. Every non-TCDM access is considered external (cluster only).

**PI\_PERF\_ST\_EXT\_CYC** Cycles used for memory stores to EXT. Every non-TCDM access is considered external (cluster only).

**PI\_PERF\_TCDM\_CONT** Cycles wasted due to TCDM/log-interconnect contention (cluster only).

**Pi\_time\_us():** a function in PMSIS library, Returns an integer value representing the current time in microseconds.

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

// define neural network paramaters

#define INPUT\_SIZE 2

#define OUTPUT\_SIZE 1

#define LEARNING\_RATE 0.01

#define EPOCHS 10000

// define ReLU activated function）

double activation(double x) {

return (x > 0) ? x: 0.0;

}

// define neural network weights and bias

double weights[INPUT\_SIZE];

double bias;

// initialize NN paramaters

void initialize() {

for (int i = 0; i < INPUT\_SIZE; i++) {

weights[i] = ((double)rand() / RAND\_MAX) \* 2 - 1;// initialize random weights

}

bias = ((double)rand() / RAND\_MAX) \* 2 - 1; // initialize random bias

}

// neural network forward propagation

double predict(double inputs[]) {

double output = 0;

for (int i = 0; i < INPUT\_SIZE; i++) {

output += weights[i] \* inputs[i];

}

output += bias;

return activation(output);

}

// Train a neural network

void train(double inputs[], double target) {

double prediction = predict(inputs);

double error = target - prediction;

for (int i = 0; i < INPUT\_SIZE; i++) {

weights[i] += LEARNING\_RATE \* error \* inputs[i];

}

bias += LEARNING\_RATE \* error;

}

int main() {

// Initialize the neural network

initialize();

// prepare training data

/\*This is just a example, you can add your own train data here\*/

double training\_data[][INPUT\_SIZE] = {{0.1, 0.12}, {1.1, 0.9}, {2.1,1.98}, {2.89, 3.2}};

double targets[] = {0, 1, 2, 3};

/\* add your test\_data here \*/

// Train a neural network

for (int epoch = 0; epoch < EPOCHS; epoch++) {

for (int i = 0; i < sizeof(training\_data) / sizeof(training\_data[0]); i++) {

train(training\_data[i], targets[i]);

}

}

// Test neural network

/\*Here you should use your test data added before\*/

for (int i = 0; i < sizeof(training\_data) / sizeof(training\_data[0]); i++) {

double prediction = predict(training\_data[i]);

printf("Input: [%lf, %lf], Target: %lf, Prediction: %lf\n",

training\_data[i][0], training\_data[i][1], targets[i], prediction);

}

return 0;

}