Cloud Computing Programming Assignment 1

Design Document

**CPU Benchmark:**

Following are the steps for the designing of the CPU benchmark:

1] Run the benchmarking for the following thread counts automatically one after another.

a) 1 Thread

b) 2 Thread

c) 4 Thread

d) 8 Thread

2] Before running any operations i am doing some initial setups and memory alocation using the initialization function. Bellow is the some code snippet regarding it:

void initialization(char threadOption) {

switch (threadOption) {

case 'a':

threadCount = 1;

threadArray = malloc(sizeof(pthread\_t)\*threadCount);

printf("Using 1 threads.");

fprintf(f,"Using 1 threads.");

break;

case 'b':

threadCount = 2;

threadArray = malloc(sizeof(pthread\_t)\*threadCount);

printf("Using 2 threads.");

fprintf(f,"Using 2 threads.");

break;

case 'c':

threadCount = 4;

threadArray = malloc(sizeof(pthread\_t)\*threadCount);

printf("Using 4 threads.");

fprintf(f,"Using 4 threads.");

break;

case 'd':

threadCount = 8;

threadArray = malloc(sizeof(pthread\_t)\*threadCount);

printf("Using 8 threads.");

fprintf(f,"Using 8 threads.");

break;

default:

printf("Invalid Option\n");

}

}

3] This benchmark contains the benchmarking using FLOPS and IOPS. So, this program will save the output in CPUPerformaceAVX.txt file for each thread.

4] To run the FLOPS and IOPS benchmarking i am giving good amount of floatng and int operations to the CPU. I am counting the time taken to run such operations to calculate the benchmark. Here, i am using the AVX instructions.

Code snippets:

void \*flopOperations() {

\_\_m256 vecfa = \_mm256\_set\_ps(6.99999, 7.99999, 8.99999, 9.99999, 10.99999, 11.99999, 12.99999, 13.999999);

\_\_m256 vecfb = \_mm256\_set\_ps(14.45685679, 15.54687899, 16.7687899, 17.768689, 18.6786789, 19.7869, 20.9, 21.99969);

\_\_m256 vecfc = \_mm256\_set\_ps(134.456879, 14345.5463487899, 1346.763487899, 1347.76834689, 1834.6786789, 1349.7869, 3420.9, 23431.949969);

\_\_m256 vecfr = \_mm256\_set\_ps(22.5675889, 23.658879, 24.9656454645, 25.546564569, 26.4564564569, 27.5665449, 28.54656459, 29.546456459);

int i=0;

for(i=0; i<noOfIterations; i++) {

//printf("%d sd", i);

\_mm256\_add\_ps(vecfa, vecfb);

\_mm256\_add\_ps(vecfc, vecfb);

\_mm256\_add\_ps(vecfr, vecfb);

\_mm256\_add\_ps(vecfr, vecfa);

}

}

void \*iopOperations() {

\_\_m256i veca = \_mm256\_set\_epi32(13236, 24234246, 3324426, 4324326, 5246, 64326, 72346, 823423326);

\_\_m256i vecb = \_mm256\_set\_epi32(92341, 102342, 1133, 1342424, 13436, 142346, 123456, 132466);

\_\_m256i vecc = \_mm256\_set\_epi32(492341, 4102342, 1324133, 13342424, 123436, 1242346, 4123456, 4132466);

\_\_m256i vecr = \_mm256\_set\_epi32(173431, 14482, 13493, 2342204, 34216, 223426, 23436, 24436);

int i=0;

for(i=0; i<noOfIterations; i++) {

\_mm256\_add\_epi32(veca, vecb);

\_mm256\_add\_epi32(vecb, vecc);

\_mm256\_add\_epi32(vecr, veca);

\_mm256\_add\_epi32(vecr, vecb);

}

}

5] All output is saved in CPUPerformaceAVX.txt file.

**Memory Benchmark:**

Following are the steps for the designing of the Memory benchmark:

1] Run the benchmarking for the following thread counts automatically one after another.

a) 1 Thread

b) 2 Thread

c) 4 Thread

d) 8 Thread

Consider the different block sizes and run the following operations for each block:

a) Sequential read write

b) Sequential write

c) Random write

2] Initialize all the values based on the above block option and thread option.

3] Based on selection create threads and run the operations given by user.

code snippet for create thread:

for(i=0; i<threadCount; i++) {

pthread\_create(&threadArray[i],NULL,seqReadWrite, "");

}

code snippet for operations:

void \*seqReadWrite() {

int indexVal = 0;

int i=0;

for (i=0; i<noOfIterations;i++){

memcpy(&memSection1[indexVal], &memSection2[indexVal], blockSize);

indexVal = (indexVal + blockSize) % (allocationSize);

}

}

void \*seqWrite() {

int indexVal = 0;

int i=0;

for (i=0; i<noOfIterations;i++){

memset(&memSection1[indexVal], '$', blockSize);

indexVal = (indexVal + blockSize) % (allocationSize);

}

}

void \*randomWrite() {

int indexVal = 0;

int i=0;

for (i=0; i<noOfIterations;i++){

memset(&memSection1[indexVal], '$', blockSize);

indexVal = rand() % (allocationSize - (blockSize + 1)) ;

}

}

4] Evaluated througput and latency for above performed operations.

5] Save the results in file.

**Disk Benchmark:**

Following are the steps for the designing of the Disk benchmark:

1] Run the benchmarking for the following thread counts automatically one after another.

a) 1 Thread

b) 2 Thread

c) 4 Thread

d) 8 Thread

Consider the different block sizes and run the following operations for each block:

a) Sequential read write

b) Sequential write

c) Random write

d) Sequential read

e) Random read

2] Initialize all the values based on the above block option and thread option.

3] Based on selection create threads and run the operations given by user.

code snippet for create thread:

for(i=0; i<threadCount; i++) {

pthread\_create(&threadArray[i],NULL,seqReadWrite, "");

}

sample code snippet for operations:

void \*seqWrite() {

int i=0;

for (i=0; i<noOfIterations; i++){

fwrite(buff, 1, blockSize, f1);

}

}

void \*randomWrite() {

int i=0;

for(i=0; i<noOfIterations; i++) {

fseek (f1, (rand()% ((allocationSize \* mbSize) - (blockSize + 1))), SEEK\_SET);

fwrite(buff, 1, blockSize, f1);

}

}

void \*seqRead() {

int i=0;

for (i=0; i<noOfIterations; i++){

fread(buff, 1, blockSize, f1);

}

}

void \*randomRead() {

int i=0;

for(i=0; i<noOfIterations; i++) {

fseek (f1, (rand()% ((allocationSize \* mbSize) - (blockSize + 1))), SEEK\_SET);

fread(buff, 1, blockSize, f1);

}

}

void \*seqReadWrite() {

int i=0;

f2 = f1;

fseek (f2, (rand()% ((allocationSize \* mbSize) - (noOfIterations\*blockSize\*threadCount + 1))), SEEK\_SET);

for (i=0; i<noOfIterations; i++){

fread(buff, 1, blockSize, f1);

fwrite(buff, 1, blockSize, f2);

}

}

4] Evaluated througput and latency for above performed operations.

5] Save the results in file.

* ***Network Benchmarking***
  + Aim: To benchmark loopback and node to node communication within a network and compare the same with iperf LINPACK benchmark
  + Procedure Notes:
    - The basic idea is to send a 64K packet of data from client to server and back.
    - The time is recorded from the start of data transfer from the client till the time the data is received back to the client.
    - **For TCP Connection:**
    - TCP connection maintains connection state hence we create a dedicated connection thread on the server and client.
    - Each thread on the server side waits for a socket from the client side and the data transfer is forwarded to another thread with the socket object.
    - The data size here is fixed at 4 GB and packet size of 64 KB
    - The TCPClient class handles the receiving and sending of data, a bytegenerator generates random 64KB packets to send to server
    - We have considered round trip time and evaluated accordingly
    - There is one serversocket which has various sockets mapped to each thread.
    - At times due to the server always accepting the data from the receiver, the connection closes after reading and before writing back to the client which results in a connection reset exception.
    - **For UDP Connection:**
    - UDP is stateless hence we have a single server accepting from multiple threads on the client side.
    - We may notice an increase in the performance because UDP has no Acks and don’t resend the packets incase of packet drops
    - Incase of UDP the Individual threads send packets of data and one server sends it back to where it came from.
    - The UDPClient class handles the receiving and sending of data, a bytegenerator generates random 64KB packets to send to server
  + After execution we calculate the throughput and latency by using the formula:
    - Throughput = Total Data(4 GB) \* (round trip) 2 \* (bits conversion) 8 / Total time for all threads
    - Latency = Total time for all threads / Total Data(4 GB) \* (round trip) 2 \* (bits conversion) 8
* ***GPU Benchmarking***
  + Aim: To benchmark operations per second for a GPU by assuming 1 thread per core for double precision, single precision, half precision and quarter precision. Compare the same with the LINPACK benchmark
  + Procedure:
    - There are four functions for 4 data types to evaluate for double for 64 bit precision, int for 32 bit precision, short for 16 bit and char for 8 bit.
    - Each thread runs on one core of the GPU
    - To calculate the GFLOPS we use:
      * GFLOPS = Number of times loop executes \* Number of cores \* Number of operations (32) \* 10 (Number of active threads is 10 times the GPU Cores) / Total time in seconds
    - Here we are assuming that each core runs one thread but ideally there are atleast threads equal to 10 times the number of cores running simultaneously.
    - To get sufficient time we are executing 32 operations in one device function.
    - The number of loops to execute in each cuda device is got by user input which is fixed at 30.