**Project #1 CPU Benchmarking**

Name: Peitong Shi

SFSU ID: 915604994

CSC641 – 01

Instructor: Dr. Jozo J. Dujmović

September 18th 2017

**Table of Contents**

1. **Introduction……………………………………………**

**3**

3

4

4

4

5

5

6

7

8

**9**

**9**

1. Workload………………………………………..............
2. Program Structure………………………………………
   1. Idea……………………………………………….
   2. Testing Environment and Expected Result………
3. Measurements………………………………………......
   1. Speed Mark……………………………………....
   2. Speed Mark for Multiple Operations…………….
   3. Comparison………………………………………
4. Conclusion………………………………………….......
5. **Appendix……………………………………………….**
   1. **Source Code……………………………………..**
6. **Introduction**

With the development of the technology, the speed of hardware updating is getting faster and faster. CPU, as one of the most important part in the computer, is also becoming more advanced. Nevertheless, even though the CPU manufacturers always claims their new generation item is faster than old one to an extent, it is hard to see the differences without having any true comparison. For example, it is hard to tell if the multiple cores provide improvement. The purpose of this program is to design a CPU speed benchmark program to test the performance of CPUs under a relatively high workload situation and present and visualize its speed level as a number mark, which makes the user easy to understand the real performance of their CPU.

1. **Workload**

We will be having two types of entity in our program testing, which is integer arithmetic operations, and floating point arithmetic operations. The reason why we are using both integer and floating point arithmetic, is because comparing to integer calculation, floating point calculation is much complicated. In order to get a more accurate result, having both operations and get their harmonic mean of their result will make this project more precise and rigorous.

1. **Program Structure**

**3.a. Idea**

The idea of this project is to have the CPU fully work on complicated calculations, and we collect the data such as the time spent and workloads. In this case, we are choosing matrices multiplications.

The basic idea is to have to 100\*100 matrices with random elements in it and perform multiplication. We will first generate 10000 random number between 1-9 for integer matrices, and (1.0 – 9.9) for floating point matrices, then make the calculation up to 1000 times. During the calculation, CPU clock time, and the real time is keep going. Then we record the running time, and we calculate the speeds.

In short, we have start time T1, end time T2, total operations N = 1000 times. Then speed which represents that how many operations would be done in one minute. In order to visualize the speed I make a formula to calculate the mark, which is

where clock2 is clock end time, clock1 is clock start time, clock\_per\_minute is how many clock times spent in one minute when making operation. Since the value is small, simply times 150000 to make it around 100 – 1000, which is helpful to make comparison.

**3.b. Testing Environment and Expected Result**

Before I test the program on others PC or laptop, I made a test on my MacBook and I got 229 for integer arithmetic, and 246 for floating point arithmetic. According to the definition of harmonic mean, I got M = . My MacBook processor information is 1.1 GHz Intel Core m3, and my PC has Intel® Core™ i7-6700K @ 4.0GHz. Simply consider the differences between the Gigahertz, the speed mark of my PC should have about 3.6 times the score my MacBook has, which would be around

.

Second, my MacBookIntel® Core™ i7-6700K has 4 cores and it should be perfectly running 4 programs simultaneously.

1. **Measurements**

**4.a. Speed Mark**

After running the program on PC which has Intel® Core™ i7-6700K, the result is 603. While the program was running, I check the CPU usage before the program started has 0.9 – 1.1 GHz was occupied. It may be occupied by Windows or other system software. Which means there is only around 3.0 GHz left. Thus, the result shows only ¾ of the speed level of the CPU, the real performance of the CPU may be around 800. According to 3.b., the program gave 211 for Intel Core m3 @ 1.1GHz. , while . Consider the differences between their architectures and the CPU MacBook is mobile version. With (1 (1 - error is tolerable since the i7-6700k is not fully functioning only for calculation (760/0.75 = 1013).

**4.b. Speed Mark for Multiple Operations**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SM | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | Total | improve |
| 1 | 603 |  |  |  |  |  |  |  | 603 | 100% |
| 2 | 586 | 586 |  |  |  |  |  |  | 1172 | 1.94 |
| 3 | 558 | 550 | 541 |  |  |  |  |  | 1649 | 2.73 |
| 4 | 471 | 492 | 486 | 494 |  |  |  |  | 1943 | 3.2 |
| 5 | 456 | 444 | 441 | 453 | 449 |  |  |  | 2243 | 3.7 |
| 6 | 418 | 410 | 406 | 402 | 408 | 414 |  |  | 2458 | 4.07 |
| 7 | 389 | 381 | 377 | 376 | 376 | 384 | 389 |  | 2639 | 4.37 |
| 8 | 356 | 351 | 346 | 341 | 340 | 345 | 350 | 370 | 2799 | 4.64 |

After running the program on the same environment as 4.a., the speed mark result shows as chart in the previous page.

We can see that , which means there is 94% improvement obtained from multiple cores. When it reaches to 4 SM programs running simultaneously, the improvement become less and steadily.

In the chart above, it is not very clear but we can still see the slope goes down since the x-value reaches 4, which means more than 4 programs that is run by the Intel® Core™ i7-6700K will have less improvement. Max SM speed = 2799, Clock rate = 4.0GHz.

Thus, SM parallelism ratio is .

**4.c. Comparison**

When the SM program is being tested on MacBook, data was also collected, now we are going to compare two set of result. Here is the result of MacBook.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SM | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | Total | improve |
| 1 | 237 |  |  |  |  |  |  |  | 237 | 100% |
| 2 | 240 | 233 |  |  |  |  |  |  | 473 | 1.99 |
| 3 | 183 | 179 | 180 |  |  |  |  |  | 542 | 2.28 |
| 4 | 138 | 137 | 138 | 141 |  |  |  |  | 554 | 2.33 |
| 5 | 118 | 120 | 113 | 112 | 122 |  |  |  | 585 | 2.47 |
| 6 | 104 | 103 | 105 | 100 | 113 | 102 |  |  | 627 | 2.64 |

In this case, we see that , which means multiple cores provides 99% improvement. However, the improvement is clearly getting lower after there are more than 2 programs running in the same time.

SM parallelism ratio is

1. **Conclusion**

In conclusion, according to the data we collected and the result we calculated, it seems both Intel® Core™ i7-6700K and Intel Core m3 @ 1.1GHz have incredible improvement obtained from multiple cores. And it clearly shows that with better Gigahertz and numbers of cores (in this case we assume the architectures are the same), CPUs can perform much better.

1. **Appendix**

**6.a. Source Code**

// main.cpp

// SpeedBenchmark

// Created by PEITONG SHI on 15/09/2017.

// Copyright © 2017 PEITONG SHI. All rights reserved.

#include <iostream>

#include <time.h>

#include <chrono>

using namespace std;

//function to make an int type matrix with argument rows and columns, elements are random

int\*\* make\_Matrix(int row, int column){

int\*\* matrix = 0;

matrix = new int\* [row];

srand(time(NULL));

for (int i = 0; i < row; i++) {

matrix[i] = new int [column];

for (int j = 0; j <column; j++) {

matrix[i][j] = rand() % 10;

}

}

return matrix;

}

//method to make a matrix with double elemnts

double \*\*make\_float\_Matrix(int row, int column){

double\*\* matrix = 0;

matrix = new double\* [row];

srand(time(NULL));

for (int i = 0; i < row; i++) {

matrix[i] = new double [column];

for (int j = 0; j <column; j++) {

matrix[i][j] = rand() % 10+0.001;

}

}

return matrix;

}

//method for two int type matrices multiplication,elements are random

void intMatrix\_mul(int\*\* m1, int\*\* m2, int row, int column) {

int\*\* result = new int\* [row];

for (int i = 0; i < 100; i++) {

result[i] = new int [column];

for (int j = 0; j<100;j++) {

for (int k = 0; k < row; k++) {

result[i][j] = result[i][j] + (2 \*(m1[i][k]) / 2 \* m2[k][j]);

}

}

}

}

//method for two double type matrices multiplication

void doubleMatrix\_mul(double\*\* m1, double\*\* m2, int row, int column){

double \*\* result = new double\*[row];

for (int i = 0; i < 100; i++) {

result[i] = new double [column];

for (int j = 0; j<100;j++) {

for (int k = 0; k < row; k++) {

result[i][j] = result[i][j] + (2.0\*m1[i][k] / 2.0 \*m2[k][j]);

}

}

}

}

//main start here

int main(int argc, const char \* argv[]) {

cout<<"Test Start..."<<endl;

cout<<"Testing Int type matrices"<<endl;

//initialize two int type matrices with 100\*100 elements

int\*\* Matrix2 = make\_Matrix(100, 100);

int\*\* Matrix3 = make\_Matrix(100, 100);

//time record

int realTime3 = chrono::system\_clock::to\_time\_t((chrono::system\_clock::now()));

int clockTime3 = clock();

//1000 times mulitiplication for 100\*100 matrices,

cout<<"Please Wait\n";

for (int i = 1; i<1000;i++) {

intMatrix\_mul(Matrix2, Matrix3, 100, 100);

if(i %10 ==0 ) {

cout<<"|";

}

}

cout<<endl;

//time record after calculation

int realTime4 = chrono::system\_clock::to\_time\_t((chrono::system\_clock::now()));

int clockTime4 = clock();

int Int\_speed = 60\*(1000/(realTime4-realTime3));

cout<<"\nTesting floating point matrices"<<endl;

//To initialize two double type matrices with 100\*100 elements

double\*\* Matrix4 = make\_float\_Matrix(100,100);

double\*\* Matrix5 = make\_float\_Matrix(100,100);

//To record cputime and clocktime before calculation

int realTime5 = chrono::system\_clock::to\_time\_t((chrono::system\_clock::now()));

int clockTime5 = clock();

//1000 times mulitiplication for 100\*100 matrices

cout<<"Please Wait\n";

for (int i = 0; i<1000;i++) {

doubleMatrix\_mul(Matrix4, Matrix5, 100, 100);

if(i %10 ==0 ) {

cout<<"|";

}

}

cout<<endl;

//To record cputime and clocktime after calculation

int realTime6 = chrono::system\_clock::to\_time\_t((chrono::system\_clock::now()));

int clockTime6 = clock();

int Double\_speed = 60\*(1000/(realTime6-realTime5));

int int\_point = 1/(double(clockTime4-clockTime3)/CLOCKS\_PER\_SEC\*60)\*150000;

int float\_point = 1/(double(clockTime6-clockTime5)/CLOCKS\_PER\_SEC\*60)\*150000;

int harmonic\_mean = (2\* (int\_point\*float\_point)) / (int\_point+float\_point);

//in this case, the more point it gets, the more operations is done in one minute,

// i.e., less time it takes.

cout<<"| Speed Benchmark for int type matrix operation is " << int\_point << " |" <<

" \n| Speed Benchmark for double type matrix operation is " << float\_point << " |"<< endl;

cout<<"| According to the definition of Harmonic Mean, the final mark is " <<harmonic\_mean<<" |"<<endl;

cout<<"| ^^^ |";

cout<<"\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*End\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n\n\n\n\n\n\n";

return 0;

}