Benchmarking CPU Performance

641-01 Computer Performance Evaluation

Author: Thanh Dip

03/08/2019

Content

Introduction

The issue is how to measure and compare CPU performance between different processors and even operating systems with a program that is run on the computer which is being tested. A couple of issues that can come up is measuring elapsed time, compiling the program to run on different computers, and making sure the processor is properly stressed. The program has to be robust and works the CPU's memory and computation power with integer and floating-point operations. The final score outputted by the program and given to the CPU will be a measure of those integer operations and floating-point operations. While normal CPU workloads will probably be not just equal amounts of integer and floating-point operations, it is a good start and can be used to compared raw performance between two CPUs. Like horsepower in an engine, while you can compare the two, it won't necessarily let you know which engine is better for a consumer, just which is faster.

The following report will detail how the program was designed and written as well as the performance measured between 3 computers (Custom Desktop, Dell Ultrabook, MacBook Air). Testing was done on the three computers as well as between a debug and release version and between Windows and Linux (Arch Distro). Multicore performance was also measured as well on the difference machines highlighting how parallelism can improve performance.

Proposal

The program will be written in C++ using two functions that run massive amounts of integer and floating-point operations to measure a CPU's speed. All the program should do is run those two functions for about 10 seconds each, measure the processor speed in operations, and then use both to compute the average score. The average displayed will be the final score that the CPU gets. In general, the program will be like the one described on iLearn:

```
int NINT=0, NFLOAT=0;
double START, VINT, VFLOAT, AverageSpeed;

START = sec( );
while(sec( ) < START+10) {Minv( ); NFLOAT++;}
VFLOAT = 60*NFLOAT/(sec( )-START);

START = sec( );
while(sec( ) < START+10) {Qsort( ); NINT++;}
VINT = 60*NINT/(sec( )-START);

AverageSpeed = 2*VFLOAT*VINT/(VFLOAT+VINT);
Display VFLOAT, VINT, AverageSpeed;
```

Numeric Workload

The integer operation function will be quick sort based on Hoare partition scheme. The workload will be a sawtooth pattern data where it is just repeats of integers 1 to N (1,2,3,…n,1,2,3,…n,1,2….). This creates an unsorted list with data that can be generated consistently on any machine.

Nonnumeric Workload

The floating-point function will be matrix inversion based on LU decomposition. The LU decomposition makes it simple to write an algorithm to solve for the inverse of the matrix. The workload will be a large matrix of floating-point numbers filled with 1.001 except on the diagonal which will be filled with 2.001. This makes sure the matrix is not singular or in other words invertible.

Implementation

The full implantation in code can be found in the appendix near the end of the report or at https://github.com/thanhdip/CPU-Benchmark. The implementation does not require any special library's and builds straight from the C++11 standard library using <iostream> for printing to the console, <vector> for all the data structure, and <chrono> for the time functions to record time intervals.

<vector> is used for both array and matrix because it is the most efficient and easiest data structure in most cases. We are also computing a large amount of data as well so it should be store on

the heap and not the stack which raw array implementations do not do unless we explicitly do so but in doing so will create unneeded complexity with data management which might affect the program's results. So <vector> was just the easiest and best choice.

<chrono> and specifically steady_clock was used to measure the time intervals between function calls. This seems to be the best choice since it is like a stopwatch and is made only to measure accurate intervals and not just time()-start.

No other part of the standard library was needed and the implementation is somewhat robust since most modern computers these days can compile and run the C++11 standard.

Structure

The program will need only two files and one header file. The first file will be main.cpp which will generate the data needed, call each of the functions and loop them each for 10 seconds. It will also handle computing the score and outputting to the console.

The specifics of the algorithms will be mentioned in the files section on the next page.

Files

There are two main files, benchmarkFunctions.cpp and main.cpp:

-benchmarkFunctions.cpp----------Holds quicksort and matrix invert functions

-main.cpp----------Calls the functions above and measures the time and score outputting it to screen

benchmarkFunctions.cpp

QuickSort

The benchmarkFunctions.cpp file holds the follow functions for **quick sort**:

```
int partition(std::vector<int> &array, int lo, int hi)
void quickSort(std::vector<int> &array, int lo, int hi)
void INTEGERQuickSort(std::vector<int> &array)
```

Quicksort algorithm was written based on the Hoare partition scheme. partition() is called by quickSort() to split the array into two so that both parts can then be quicksorted again. This is done recursively down until each individual part of the array is sorted. The pivot is the average half of the size of the

given array. This is easier to implement than a random pivot point and does relatively well against other pivot choices.

INTEGERQuickSort() is called by the main function in main.cpp so only a vector needs to be passed. This is all done by reference to only changed the vector given rather then return a whole new vector.

Vector is used to store the list since it is simple, robust, and fast.

Matrix Inversion

The benchmarkFunctions.cpp file holds the follow functions for **matrix inversion**:

```
void LUDecompose(vector<vector<double>> &matrix, int dimension)
vector<vector<double>> LUMatrixInvert(vector<vector<double>> &matrix, int dimension)
void DOUBLEmatrixInv(std::vector<std::vector<double>> &matrix, int rc)
```

The matrix inversion is based on LU decomposition where some matrix A = LU. That is a matrix A is decomposed into two matrix L and U where L is the lower triangle of the matrix and U is the upper. This decomposition makes it easy to write an algorithm which computes the inverse. Since L and U are triangular, it turns a complex matrix into a basic substitution algorithm for functions.

$$A = \begin{bmatrix} 1 & 2 & 4 \\ 3 & 8 & 14 \\ 2 & 6 & 13 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 4 \\ 0 & 2 & 2 \\ 0 & 0 & 3 \end{bmatrix}$$

The matrix L can then be solved by Lx = i where i is just corresponding identity vector.

LUDecmpose() changes a matrix into it's decomposition by looping through the lower and upper part of the matrix dividing the subsequent vector part and subtracting the subsequent vector parts so that everything zeros out on each respective triangle L or U.

LUMatrixInvert() splits the identity matrix into each column vector and loops through the Lower and Upper triangle to compute the inverse of L and U. After that it multiples inverse L and inverse U together to get inverse of A or the inverse of the original matrix.

Like quicksort DOUBLEmatrixInv() just calls the two functions above and sets things up to be called by main.cpp. And again everything is done by reference but LUMatrixInvert() returns a matrix because it was easier to implement and store the inverse than to do it on the matrix itself.

Matrix is a vector of vectors as mentioned before because vector implementation is simple, robust, and is as fast or only a little bit slower than any other data structure.

main.cpp

main.cpp generates the data to be ran and calls the functions above. It calculates the time interval between each function and outputs the score based on the results. Again, it follows the basic pseudo code given on iLearn.

```
int NINT=0, NFLOAT=0;
double START, VINT, VFLOAT, AverageSpeed;

DateGEN
START = sec( );
while(sec( ) < START+10) {Minv( ); NFLOAT++;}
VFLOAT = 60*NFLOAT/(sec( )-START);

DateGEN
START = sec( );
while(sec( ) < START+10) {Qsort( ); NINT++;}
VINT = 60*NINT/(sec( )-START);

AverageSpeed = 2*VFLOAT*VINT/(VFLOAT+VINT);
Display VFLOAT, VINT, AverageSpeed;
```

Data Gen

Data gen for quick sort: (1, 2, 3, 4,…,n) Union (1, 2, 3, 4,…,n) Union (1, 2, 3, 4,…,n) Union (1, 2, 3, 4,…,n)

This generates data in a sort of sawtooth which gives reproducible results and does not rely on randomness or any random functions.

Data gen for matrix inversion: matrix A[i][j] = 1.0001 when i ! = j AND A[i][j] = 2.0001 when i == j

This generates a known inversible matrix that is reproducible and scalable to a large point.

Time Measurement

Time measurement is from the std::chrono::steady_clock::now() function which returns a time_point that can be compared with another time_point to get the interval. This does not return a time itself like 3:39 p.m but a time_point which can be converted to a time.

The time function is used before, within, and after the while loop. This calculates the total time it too for the loop to finish and keeps the loop running for only about 10 seconds each for both functions.

Score

```
int score = (2*vfloat*vint)/(vfloat+vint);
```

The final score is just an average of both the scores.

Results

All benchmarks were running with the follow amount of data:

QuickSort: 5000 concatenations of 1,2,3,4,…,800; Making a list of 4,000,000 entries.

Matrix Inversion: Matrix of 475x475

The follow machines were benchmarked:

Custom PC:	i9-7900x @ 3.3ghz base	10 Cores 20 Threads	Windows 10 LTSC Build 17763
Dell XPS 13:	i7-8550U @ 1.80ghz base	4 Cores 8 Threads	Windows 10 Home 17763 | Arch Linux 5.0.0
MacBook Air:	i7-5650U @ 2.20ghz base	2 Cores 4 Threads	macOS 10.14

One thing to note is that each clock speed is only the base clock speed an not the boost clock speed which can range from the base to 4ghz depending on the machine above. So while the XPS 13 has a lower base clock, it has a much higher boost clock.

Debug vs Release

Preformed with Visual Studio Code 2017 on i9-7900x @ 3.3ghz base :

Debug score = 0 | vfloat = .546 minv/min | vint = 1.378 sort/min

Release score = 408 | vfloat = 360.539 minv/min | vint = 470.195 sort/min

Speedup: 408/.782 = 521x

Release was 521 times faster than the debug version.

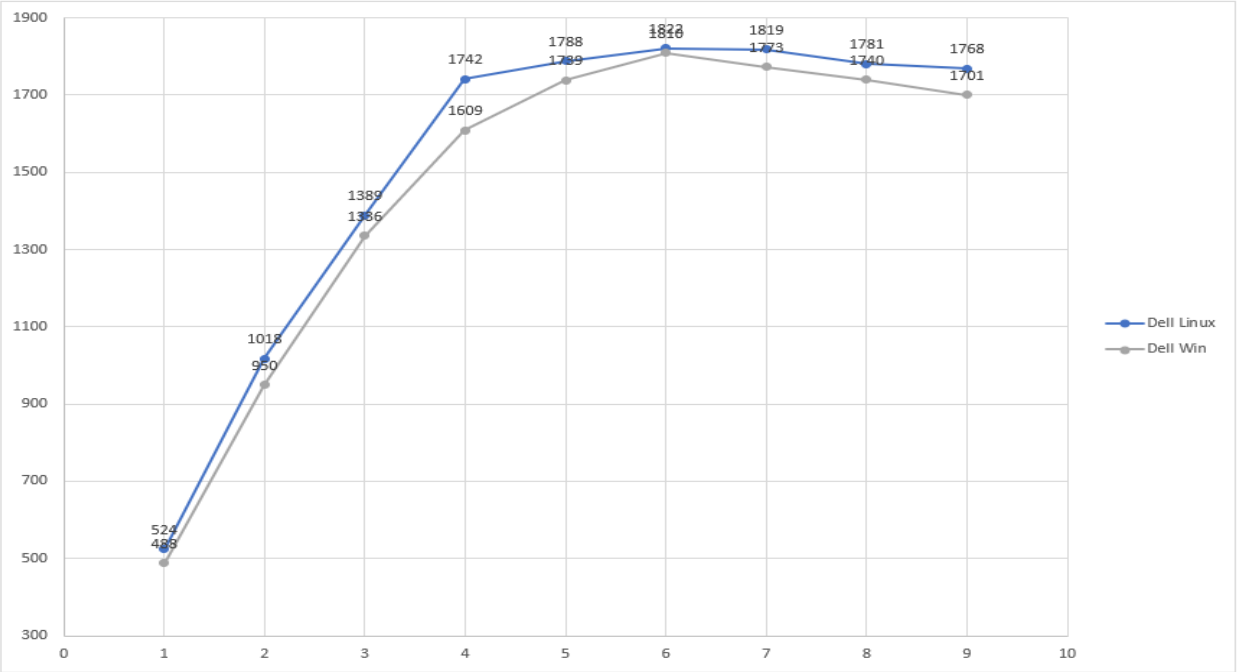
The debug score was abysmal since the size of the list and matrix was extremely large and each time the compiler needed to insert debug flags and code most likely increasing the program size by a significant amount.

Windows vs Linux

Data

Preformed on Dell XPS 13 which has i7-8550U @ 1.80ghz base

Graph: Sum of all scores vs number of programs running



Linux scores on same machine:

#	Dell Linux	4 cores	8 threads						
1	524								
2	509	509							
3	463	463	463						
4	434	437	436	435					
5	409	278	284	409	408				
6	380	263	262	386	264	267			
7	242	241	241	367	243	241	244		
8	224	223	220	227	222	222	222	221	
9	192	188	203	195	195	219	189	191	196

Windows scores on same machine:

	Dell Win								
1	488								
2	475	475							
3	445	445	446						
4	403	406	398	402					
5	325	368	332	372	342				
6	302	332	322	310	265	279			
7	256	244	268	243	280	241	241		
8	217	219	217	216	217	220	216	218	
9	187	197	183	199	187	194	186	197	171

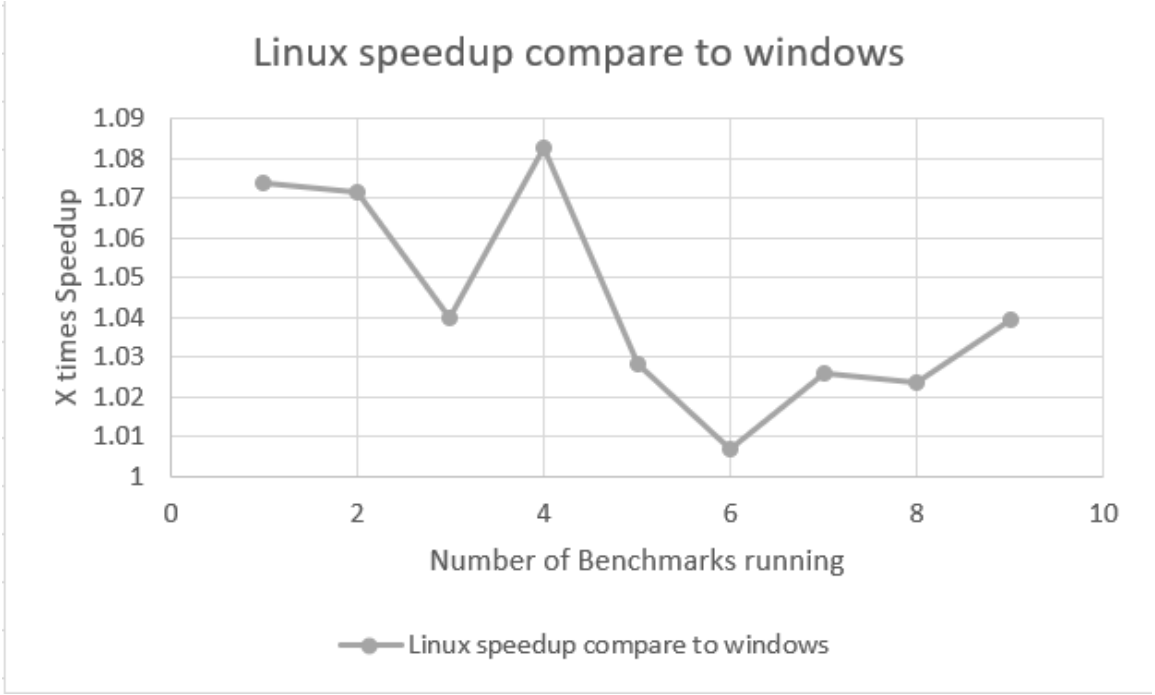
Explanation

The Linux implementation is a bit faster than the windows implantation. This is mostly likely do to the fact that Windows has more overhead than most Linux distributions. Arch Linux specifically is a barebones Linux distro where a user can build the experience from scratch i.e choose how the UI is rendered or even to not even have UI at all. So with less overhead the benchmark program has more system resources and can be just a little faster than the Windows version.

Linux average sum of score was 1516.778

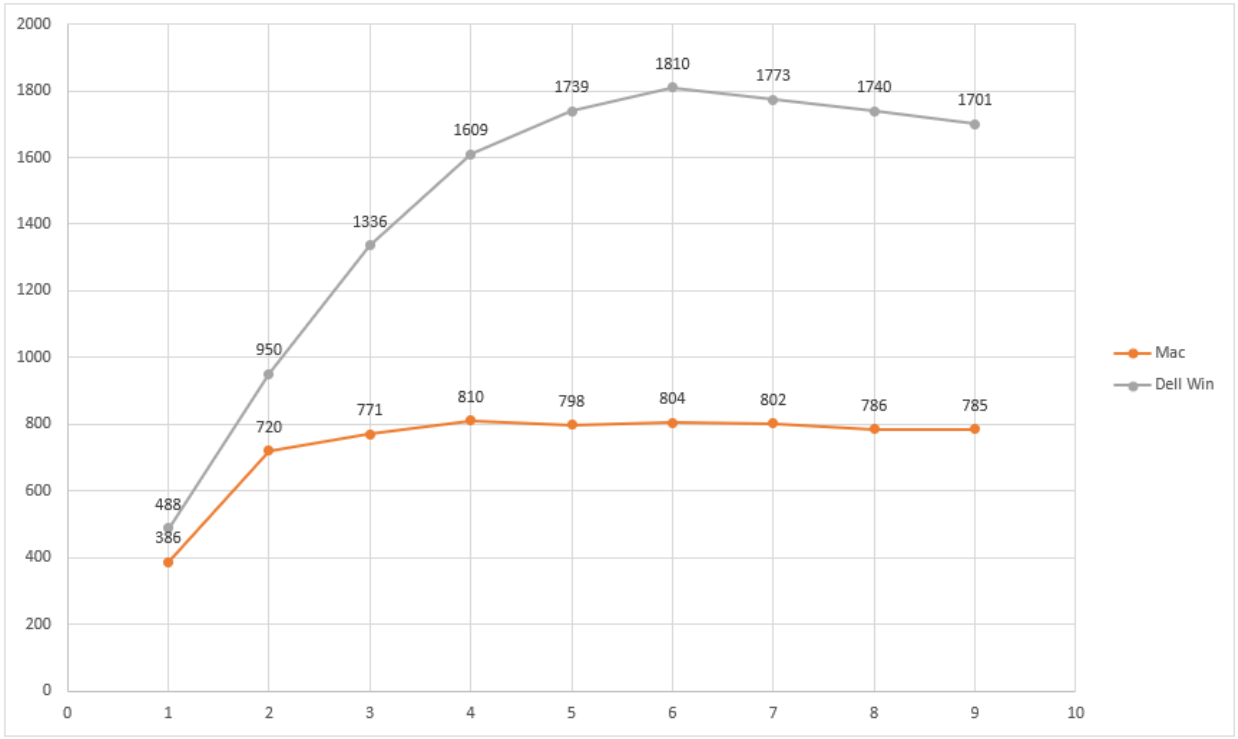
Windows average sum of score was 1460.667

Linux was about 4% faster on average.

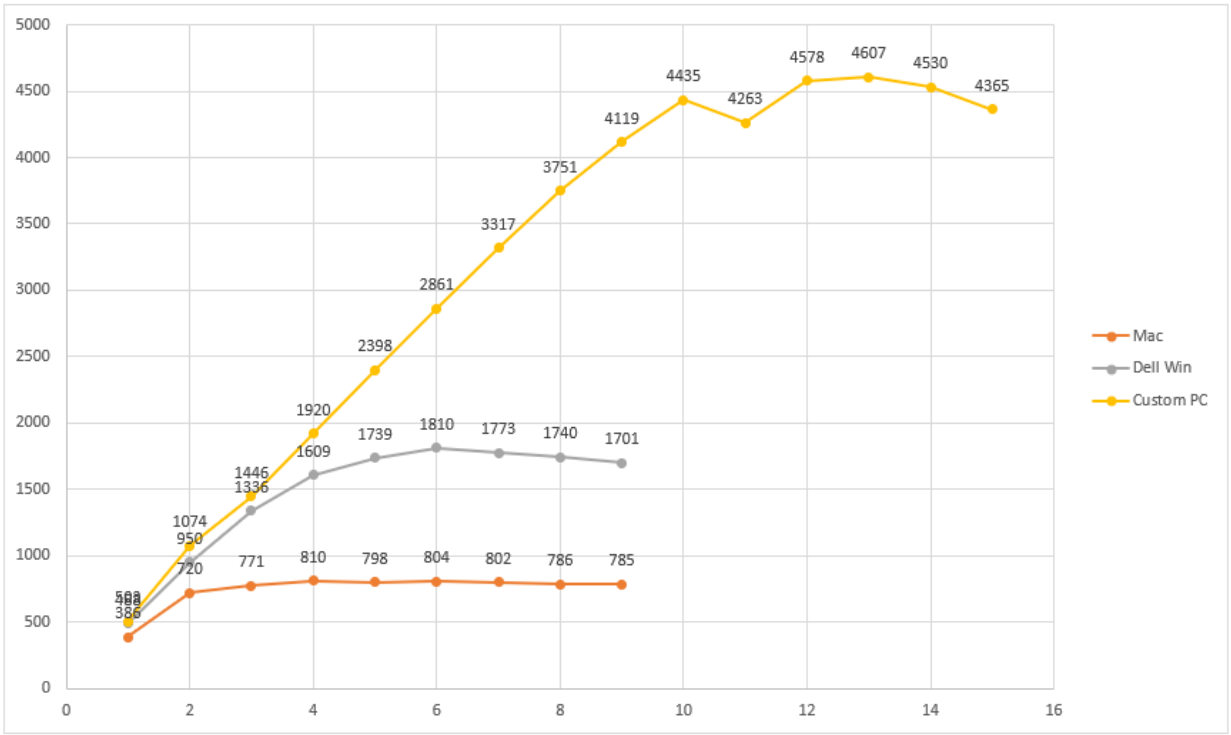


Multicore

Performance between the Mac and Dell machine only:



Performance between all 3 machines:



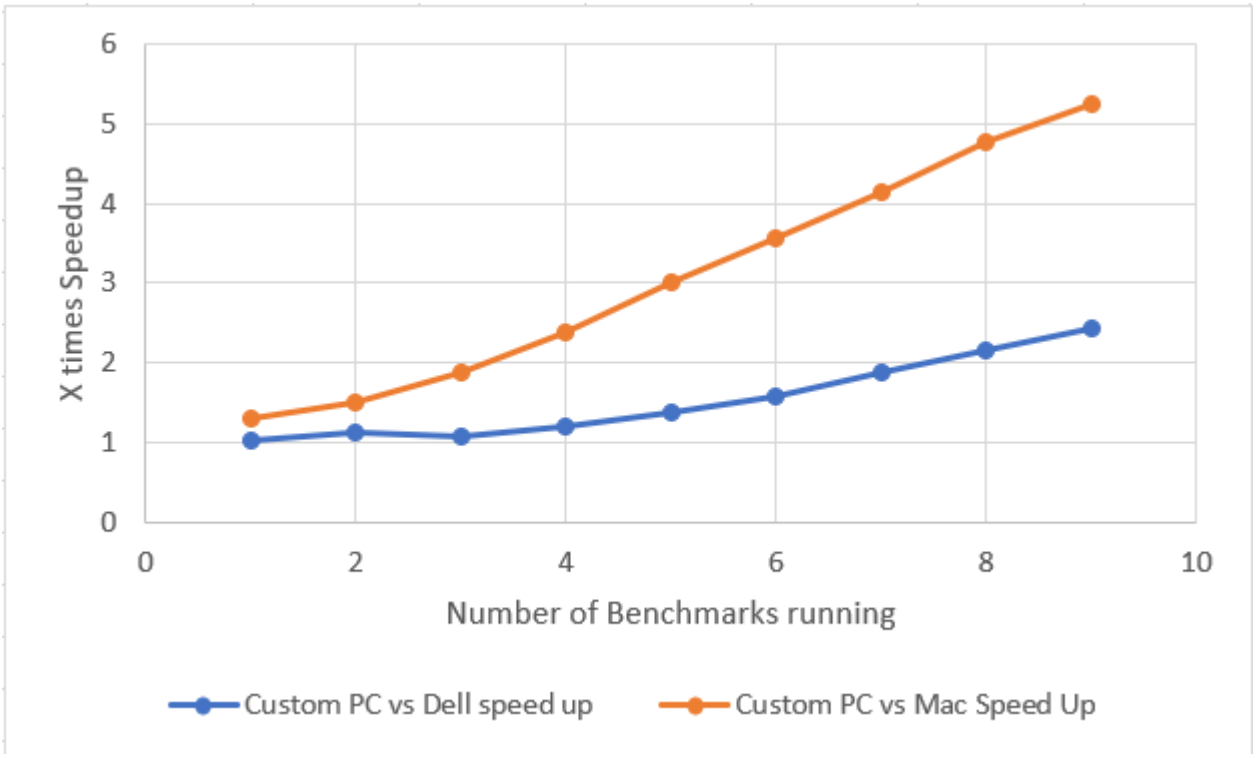
Explanation

The first multi-core graph only shows the performance of the Dell on windows and the Mac os. One has 4 physical cores while the other has 2. Initially the graph shows relatively linear growth in performance until the number of programs running is equivalent to the number of physical cores. Hyper-threading allows more than one thread to run on a physical core which increases a CPU's ability to multitask. This is why performance does not act asymptotically right away.

The above is mostly true for the second multi-core graph as well which also includes the performance of the custom PC with 10 physical cores and 20 threads. However once the custom PC runs more 10 benchmark programs at once performance becomes strange. This is a windows issue. Windows does not allow for a user to highlight more than 15 programs and run them all at once. This can be fixed by editing the registry but windows itself seems to not be able to handle the mutlithreading itself. Testing on Linux would need to be done.

Another thing to note is the i9 is not faster than the i7 in single threaded performance where only the clock speed matters. This seems to be the case for most modern processors where clock speed in most high-end processors are only around 3ghz to 4ghz. Anything higher needs a large cooling system so the heat generated by the processor can be managed and run without issues for a longer period of time.

Speedup graph, Custom PC vs Dell and Mac



Conclusion

The implementation was not difficult to do and quite simple because of the algorithms and data used. The only tweaks that needed sometime was how much to crunch at a time.

For the most part the experimental results were as expected. The custom PC was the fastest followed by the Dell and the MacBook came last. Performance depended on the CPU's boost clock or the highest clock speed the CPU can output and then the number of cores the CPU has. Linux is also faster than windows in performance as well because of the lower overhead in most Linux distributions. Debug versions of programs are significantly slower since the compiler as to add machine code in between the actual program code so depending on the complexity of the program it can be a lot slower.

CPU performance these days are very similar for single threaded performance. Since most processors have about 3ghz to 4ghz clock speed on an individual core. Where performance differs is on the multi-core systems. More cores can handle a lot more processing and multi-tasking.

The program itself worked well however it was written and tested on the Custom PC which may have affected the range of the score. Single core performance was around 500 and was not granular enough to really see large differences in performance between the computers. Each CPU was also very close in family which each other and the results where more of how performance changed generation to generation of Intel CPUs. Possible improvements would be to make the Benchmark more granular and have single threaded scores of around 1000 rather than 500. Different CPUs should be tested instead of relatively high ends CPUs so Intel i3, i5 and AMD CPU's should also be tested.

Appendix

Code

main.cpp

```
#include <iostream>
#include <chrono>
#include "benchmarkFunctions.h"

void printArray(std::vector<int> array);
void printMatrix(std::vector<std::vector<double>> matrix);

/* Using std::vector because allocating on heap is easier to allocate large amount of memory. Vector is
also really fast as well.
 */

int main()
{
    //PRINT AT BEGINING
    std::cout << "BENCHMARK PROGRAM STARTING..." << std::endl;
    std::cout << "This should run for about 20 seconds. Please give it some time." << std::endl;
    std::cout << "Running...." << std::endl;

    /* ------------------------------------------------------------------------------------
     *  INTEGER
     */

        //Data Gen
        // 10000 * 1000 = 1.51s 10000 * 500 = .67s 10000 * 800 = 1.12s
        // 1000*800 runs just enough to match with matrix inversion to get 500ish avg score on an Intel
Core i9-7900X @ 3.30Ghz NO BOOST
        // 5000*800 is perfect for -O3 optimization.
    int multa = 5000;
    int multb = 800;
    std::vector<int> array;
    for(int i = 0; i < multa; i++)
    {
        for(int j = 0; j < multb; j++)
        {
            array.push_back(j);
        }
    }

        //Sort
    auto starti = std::chrono::steady_clock::now();
    int nint = 0;
    std::cout << "Sort...." << std::endl;
    while(auto t = std::chrono::duration_cast<std::chrono::seconds>( std::chrono::steady_clock::now() -
starti ).count() < 10)
    {
        std::vector<int> arr = array;
        INTEGERQuickSort(arr);
        nint++;
    }

    auto endi = std::chrono::steady_clock::now();
    std::chrono::duration<double> runtimeInt = endi-starti;
    double vint = (60*nint)/runtimeInt.count();

    /* ------------------------------------------------------------------------------------
     *  DOUBLE
     */

        //Data Gen
        // 475 = 1.07 always a little above 1 sec.
```

```
        // 225 runs just enough to match with sort to get 500ish avg score on an Intel Core i9-7900X @
3.30Ghz NO BOOST
        // 475 is baseline used for -O3 optimization to match with above.
    int rc = 475;
    std::vector<std::vector<double>> matrix(rc, std::vector<double>(rc));
    for(int i = 0; i < rc; i++)
    {
        for(int j = 0; j < rc; j++)
        {
            if(i == j)
            {
                matrix.at(i).at(j) = 2.0001;
            }
            else
            {
                matrix.at(i).at(j) = 1.0001;
            }
        }
    }

        //Invert
    auto startf = std::chrono::steady_clock::now();
    int nfloat = 0;
    std::cout << "Invert...." << std::endl;
    while(auto t = std::chrono::duration_cast<std::chrono::seconds>( std::chrono::steady_clock::now() -
startf ).count() < 10)
    {
        std::vector<std::vector<double>> mat = matrix;
        DOUBLEmatrixInv(mat, rc);
        nfloat++;
    }

    auto endf = std::chrono::steady_clock::now();
    std::chrono::duration<double> runtimeFloat = endf-startf;
    double vfloat = (60*nfloat)/runtimeFloat.count();

    /*-------------------------------------------------------------------------------------------------------
---------------------------------
     * PRINT AT END
     */
    std::cout << "DONE!!!\n" << std::endl;
    int score = (2*vfloat*vint)/(vfloat+vint);
    std::cout << "YOUR BENCHMARK SCORE IS:" << std::endl;
    std::cout << "!!!! " << score << " !!!!" << std::endl;
    std::cout << "vfloat = " << vfloat << " minv/min " << "|_| vint = " << vint << " sort/min" <<
std::endl;

    std::cout << "\nEND BENCHMARK" << std::endl;
    std::cout << "Press ENTER to exit..." << std::endl;
    std::cin.get();
    return 0;
}
```

benchmarkFunctions.cpp

```
#include "benchmarkFunctions.h"
#include <iostream>

/* QUICKSORT AND HELPER FUNCTIONS
 * Uses vector and function based on Hoare partition scheme
 */
        //Partions the array and swaps to sort
int partition(std::vector<int> &array, int lo, int hi)
{
    int pivot = array[(lo+hi)/2];
    int i = lo - 1;
    int j = hi + 1;

    for(;;)
```

```
    {
        do
        {
            i = i + 1;
        } while (array[i] < pivot);

        do
        {
            j = j - 1;
        } while (array[j] > pivot);

        if(i >= j)
        {
            return j;
        }
        //Swap
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

}
        //Sorts by paritioning then recursively doing so
void quickSort(std::vector<int> &array, int lo, int hi)
{
    if(lo < hi)
    {
        int p = partition(array, lo, hi);
        quickSort(array, lo, p);
        quickSort(array, p+1, hi);
    }
}

void INTEGERQuickSort(std::vector<int> &array)
{
    int end = array.size();
    quickSort(array, 0, end-1);
}

/* MATRIX INVERSION AND HELPER FUNCTIONS
 * Uses vector and based on LU decomposition
 * I.e some matrix A is split into two matrix L and U where they are the upper and lower triangle of the
square matrix.
 * This makes solving for the inverse easier.
 */
        //Decomposes matrix M = LU where L is lower and U is upper triangle of the square matrix
void LUDecompose(std::vector<std::vector<double>> &matrix, int dimension)
{
    double tem;
    for(int k = 0; k <= dimension - 1; k++)
    {
        for(int j = k + 1; j <= dimension; j++)
        {
            tem = matrix[j][k] / matrix[k][k];
            for(int i = k; i <= dimension; i++)
            {
                matrix[j][i] = matrix[j][i] - tem * matrix[k][i];
            }
            matrix[j][k] = tem;
        }
    }
}
        //Uses the LU decomposition to then go through both upper and lower triangle and invert the matrix
std::vector<std::vector<double>> LUMatrixInvert(std::vector<std::vector<double>> &matrix, int dimension)
{
    std::vector<double> identityVec(dimension + 1, 0.0);
    std::vector<double> y(dimension + 1, 0.0);
    std::vector<std::vector<double>> s(dimension + 1, std::vector<double>(dimension + 1, 0.0));
    double tem;
```

```
    for(int m = 0; m <= dimension; m++)
    {
        for(int x = 0; x < dimension; x++)
        {
            identityVec[x] = 0.0;
        }
        identityVec[m] = 1.0;

        for(int i = 0; i <= dimension; i++)
        {
            tem = 0.0;
            for(int j = 0; j <= i - 1; j++)
            {
                tem = tem + (matrix[i][j] * y[j]);
            }
            y[i] = identityVec[i] - tem;
        }

        for(int i = dimension; i >= 0; i--)
        {
            tem = 0.0;
            for(int j = i + 1; j <= dimension; j++)
            {
                tem = tem + (matrix[i][j] * s[j][m]);
            }
            s[i][m] = (y[i] - tem) / matrix[i][i];
        }
    }

    return s;
}

void DOUBLEmatrixInv(std::vector<std::vector<double>> &matrix, int rc)
{
    std::vector<int> piv;

    LUDecompose(matrix, rc - 1);
    matrix = LUMatrixInvert(matrix, rc - 1);
}
```