AMD Processor Scalability: Ryzen 1800x vs. Ryzen 1950x (Threadripper)

Ryan Galloway, Dan Benett, Lucian Turk

May 15, 2019

Summary

When looking at the scalability of the Ryen 1800x vs the Threadripper 1950x, we found that the 1950x scaled better after certain thresholds for both strong and weak scaling were met with regards to number of threads used to paralleize the matrix multiplication algorithm. When looking at the Cinebench scores, the programs scaled equally until the Threadripper surpassed the 1800x by simply having access to more physical cores and threads.

Introduction

The purpose of this experiment is to determine which chipset scales better the AMD Ryzen 1800x, or the Ryzen Threadripper 1950x. Do you see a 1:1 ratio of performance increase when increasing the number of threads for the same size problem ? When increasing the problem size does the execution time remain constant when also increasing the number of threads? Various performance metrics were used to collect data on the performance of each processor. The data was then analyzed to determine which processor had better weak and strong scaling, and whether either had good performance with regards to weak and strong scaling.

Tools & Hardware

The experiments were conducted on:

- Ryzen 1950x Threadripper with 32 GB of DRAM, Aorsus X399 Motherboard, water
- Ryzen 1800x with 16 GB 3200mhz DRAM, an Asrock X370 Taichi Motherboard, water

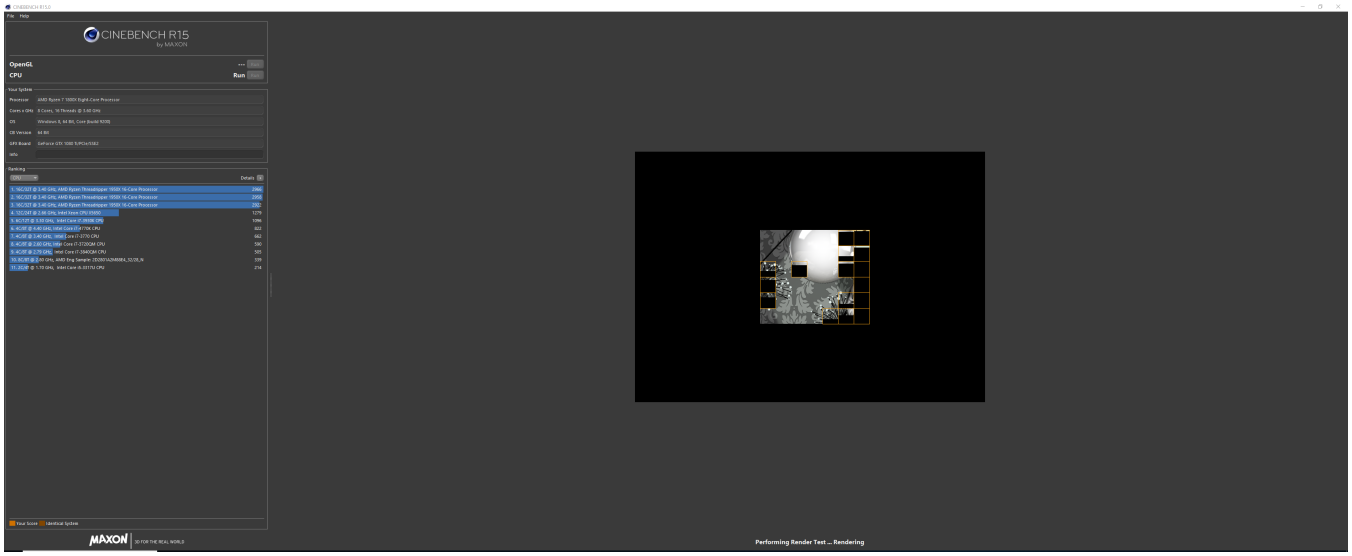
The software used for testing was:

- Linux Ubuntu
- Windows 10 Home edition
- Linux perf commands
- Cinebench R15
- Vim, g++, and Open MP for parallelization

Methods

We conducted two separate tests for our purposes. The first was to run a commercial program called Cinbench R15 and track the scores and time for each processor and thread count. For the second test we created a matrix multiplication program to test specific scalability of each processor.

For the Cinebench tests we operated in a windows environment using Cinebench R15 to conduct the performance tests for both systems. We conducted 5 trials on each thread, and since Cinebench doesn't track time we conducted our own "wall clock time" tracking via a stop watch. The following is a sample shot of one of the tests conducted:



For the matrix multiplication we were looking for strong and weak scalability of the processors. The following is the code we used to conduct the tests starting with the functions used to dynamically allocate and destroy a 2d array:

```
#include <stdio.h>      /* printf, scanf, puts, NULL */
#include <stdlib.h>     /* srand, rand */
#include <time.h>       /* time */
#include <iostream>
#include "omp.h"

using namespace std;

double** create_matrix(int rows, int cols)
{
    double** mat = new double* [rows]; //Allocate rows.
    for (int i = 0; i < rows; ++i)
    {
        mat[i] = new double[cols](); //Allocate each row and zero initialize..
    }
    for(int j = 0; j < rows; j++)
            for(int k = 0; k < cols; k++)
            {
                    mat[j][k] = rand() % 10 + 1; // generate random matrix values...
            }
    return mat;
}

void destroy_matrix(double** &mat, int rows)
{
    if (mat)
    {
        for (int i = 0; i < rows; ++i)
        {
            delete[] mat[i]; //delete each row..
        }

        delete[] mat;  //delete the rows..
        mat = nullptr;
    }
}
```

The next image is the code in the main function:

```
int main(int argc, char *argv[])
{
    int m = atoi(argv[1]);
    int n = atoi(argv[2]);
    int l = atoi(argv[3]);


    int rows = m;
    int cols = n;

    double** matA = create_matrix(rows, cols);
    double** matB = create_matrix(rows, cols);
    double** matC = create_matrix(rows, cols);
    omp_set_num_threads(l);
    //Multiplication
    int i, j, k;
    #pragma omp parallel for private(i ,j ,k)
        for(int i = 0; i < rows; ++i)
        {
            for(int j = 0; j < cols; ++j)
            {
                for(int k = 0; k < cols; ++k) //ColsA..
                {
                    matC[i][j] += matA[i][k] * matB[k][j];
                }
            }
        }
    destroy_matrix(matA, rows);
    destroy_matrix(matB, rows);
    destroy_matrix(matC, rows);

    return 0;
}
```

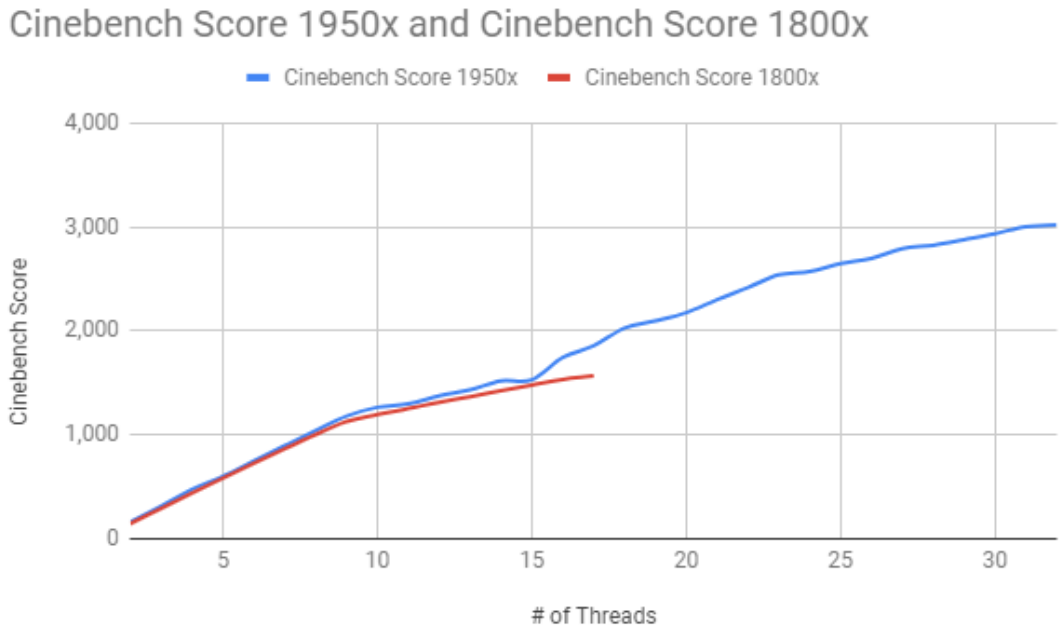
The code that begins with #pragma is the code that actually parallelizes the matrix multiplication using multiple threads to do the calculations. The line that contains ,omp_set_num_threads(l), is a function call for omp that allows us to set the number of threads used with the command line argument, l.

Take specific note of the declaration of the loop counter variables prior to the loop statements, and also the private statement in the #pragma paralleization statement. We ran into significant problems when having these declared within each loop statement, and not having declared the inner loop counters private. We ran into what is called a r̈ace condition;̈ and the threads were updating the same loop counters at the same time which caused significant errors in the matrix multiplications.

To run the tests we created dual boot systems for both machines with a Linux Ubuntu. We did this order to gain access to the perf commands Linux offers to calculate number of cycles, number of instructions, L1 cache loads, etc. through the perf stat command. his allowed us to calculate time and speedup of both processors for our strong and weak scaling experiments.

Data and Analysis

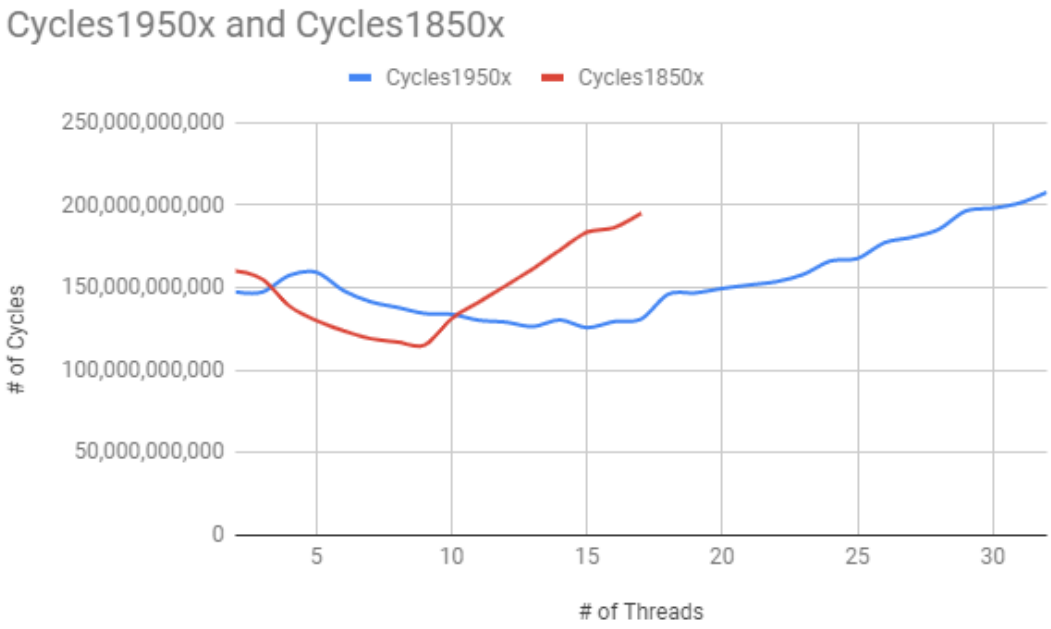
Cinebench

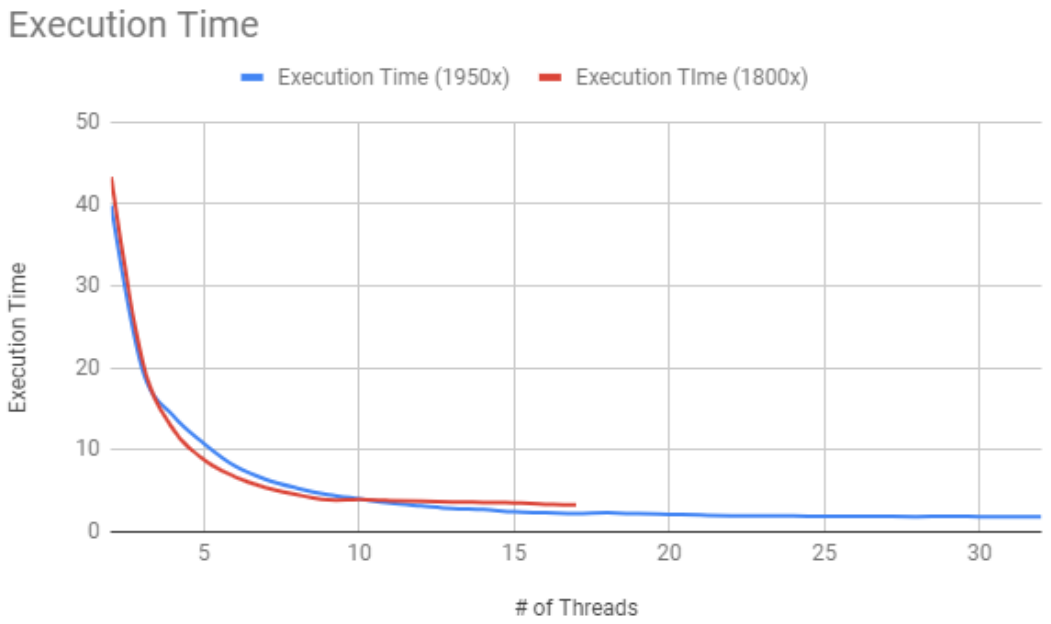


Cinebench is an industry standard for testing CPU performances that uses floating point calculations to render an image as seen in the figure in the methods section, but it does have it's flaws. It tries to separate the CPU processes form the system dependent resource and background processes running from the operating system. For our purposes it worked well as a snap shot of the CPU performance for each chip. As seen on the graph the processors are neck and neck from 1-8 threads, but as soon as the Ryzen 1800x is using more threads than it has physical cores we see a leveling out of the performance increases we saw when increasing threads from 1-8 initially. The 1950x excels from 8 cores on and sees a double increase in score at 32 threads vs. the 1800x's 16 threads. So for this type of problem the 1950x is the clear winner.

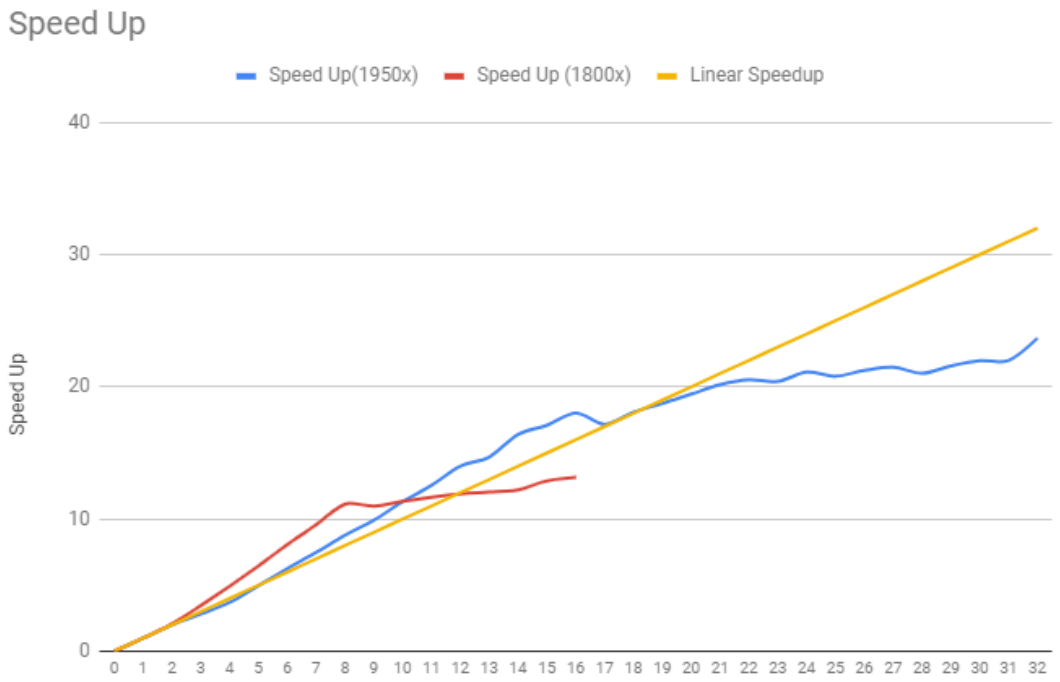
Strong Scaling

For this experiment we used a set matrix size of 1500 x 1500, and increased the number of threads without changing the problem size. The following is the data we collected for this experiment:





Strong Scaling(cont.)



The strong scaling experiments produced some pleasant surprises. Using the formula:

$$S = \frac{T_s}{T_p}$$

In the formula speedup S equals the time of serial execution T_s over the time of parallelized execution T_p. We had to assume 100% parallelization due to the fact we had know way of calculating the portion of our program that was parallelized. As you can see from the charts on speedup and efficiency,

Strong Scaling(cont.)

we observed superlinear speedup and efficiency greater than 100% at 2 threads for the 1800x and at 6 threads for the 1950x.

Both processors saw significant superlinear speedup from their breakout points at 2 and 6 threads respectively, but when they reached the limit of their physical cores we saw their efficiency and speedup gains per thread decrease. The superlinear speedup then went into sub-linear speedup.

This could be partly due to Ahmdahl's Law, which states that the maximum speedup you can achieve when increasing the number of resources working on a paralleized problem is directly proportional to the amount of work that can be parallelized in the program. We do not believe we hit the plateau of Ahmdahl's Law, but rather when we hit the physical core limit the speedup decreased because we were no longer adding more lower level cahce resources. We were instead sharing the existing physical cores with new threads.

In terms of which processor has more scalability, it boils down to what the cost is in terms of power, increased cost for the 1950x chip, and increase in performance we saw. Though speedup and efficiency continued past 16 threads with the 1950x, the execution time decrease wasn't very significant. The 1950x requires a 180v socket vs. the 1800x's 95V socket. The thermal max of 68C for the 1950x would also mean our cooling costs would increase to maintain that temperature vs the 1800x's max of 95C. With all of these factors added in the 1800x wins the scalability fight in terms of strong scaling.

Weak Scaling

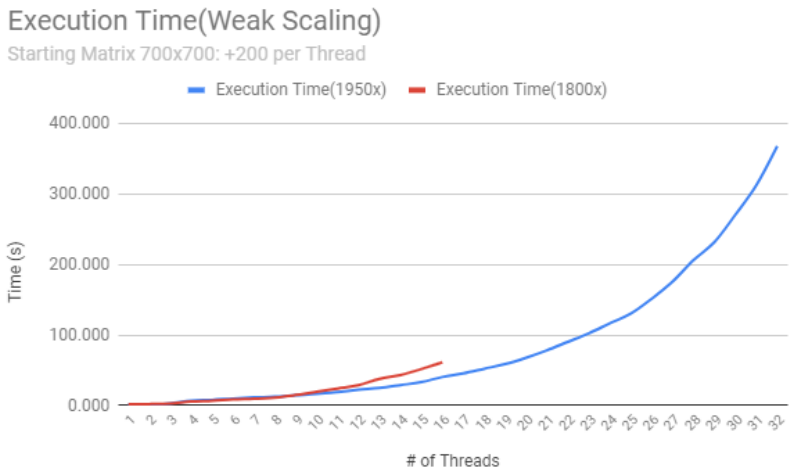
The group ran into to some self induced issues with weak scaling. The definition we obtained for measuring weak scaling led us to believe that doubling the matrix size per thread was the way to go to test weak scaling. Due to time constraints forced upon us by problems with the test code, by the time we ran the tests and analyzed the data we had lost access to both systems. One due to it going back to it's owner, and the other due to a hard drive failure.

Also, we did not know the exact portion of our program that was serialized which made it hard to calculate speedup with Gustafson's Law that states:

$$scaled\ speedup = s + p * N$$

We would obtain a false speedup, and also a false efficiency %. So we were left with execution time as a metric. The following is the data we obtained for execution time give a starting size 700x700 matrix at thread 1, and increasing the matrix size by 200 per thread:

Weak Scaling(cont.)



What we would want to see is execution time remain almost constant, but we saw exponential growth in execution time. The reasons for this are we were attempting to double the matrix size per thread. This is not how you should conduct weak scaling.

We should have attempted to double the problem size (not matrix size) when doubling the number of threads. Each matrix multiplication uses the dot product to compute the resulting matrix entries. So with our increase of 200 x 200 we were increasing our problem size by an amount much larger than double *per thread*!

Looking at the execution times with our improper workload it is clear that the 1950x handled the increased problem sizes much better than the 1800x. The 1950x kept execution time increase lower owing to the fact that the 1950x has double the number of physical cores of the 1800x. This allowed the 1950x to scale better with our problem size increases. They both had a sharp increase in execution times when surpassing their physical core number which continued to rise exponentially with the more threads we added after reaching the physical core thread count.

Conclusions

Both processors performed well with regards to Cinebench Scores, Strong, and Weak scaling. However, our data collection methods were rather primitive. The 1950x wins the scalability wars when only taking into account increased speedup and decreased execution times due to having almost double the cores. It does come at a price though. The 1950x uses more power, can't handle heat as well, and is significantly more money to buy. The ultimate decision as to which chip is more scalable lies with the user and their needs, the probelm type, and the cost to benefit gains you receive for each chipset.

References

Li, Xin,(2018,November 09)*Scalability : strong and weak scaling* retrieved from URL https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/

Measuring Parallel Scaling Performance. (n.d.). Retrieved from https://www.sharcnet.ca/help/index.php/Measuring_Parallel_Scaling_Performance