Programming Assignment Report 4

**Matrix Multiplication with Linux Performance Tools**

**Approach:**

The instructions were very clear, and all the instructions were followed step by step:

1. First, the perf tools were installed
2. Next, the CPU counters were observed
3. Then the naïve and interchange algorithms were run with the performance tools for matrix size 500 and 1000
4. Next, the hotspots were accessed, and the highest cost functions were recorded (anything in red since that meant the largest time)

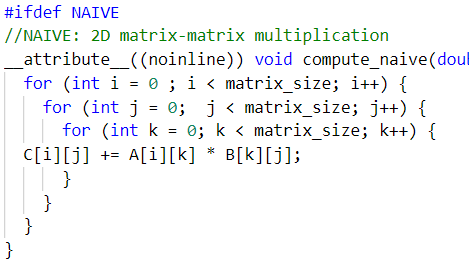
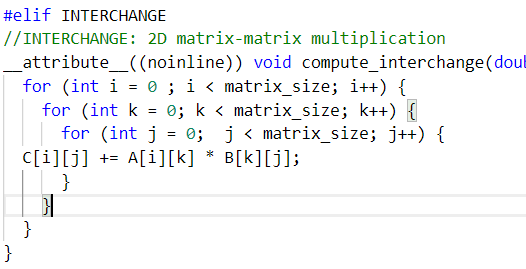
**Insight Gained and Problems:**

The biggest problem was having root permission on fusion. Since I had access to the student account, the perf record and perf report steps would not produce the right symbols; this was overlooked since enough information was there to navigate to the right process and see function costs.

Some insights gained were that this tool is very helpful in quantifying how good the optimization of the algorithms is. Even though I knew row-based matrix access in the inner loop is better than column-based since the cache is loaded rows when it loads in bulk, seeing the number of misses is reduced to a quantifiable amount was interesting. There were more CPU counters on the machine we could have used to do further analysis which was not used in this lab (per instruction). I would like to try a few and see more quantifiable results. Also, there exists a triangular optimization for the multiplication which I would also like to see would pan out versus interchange.

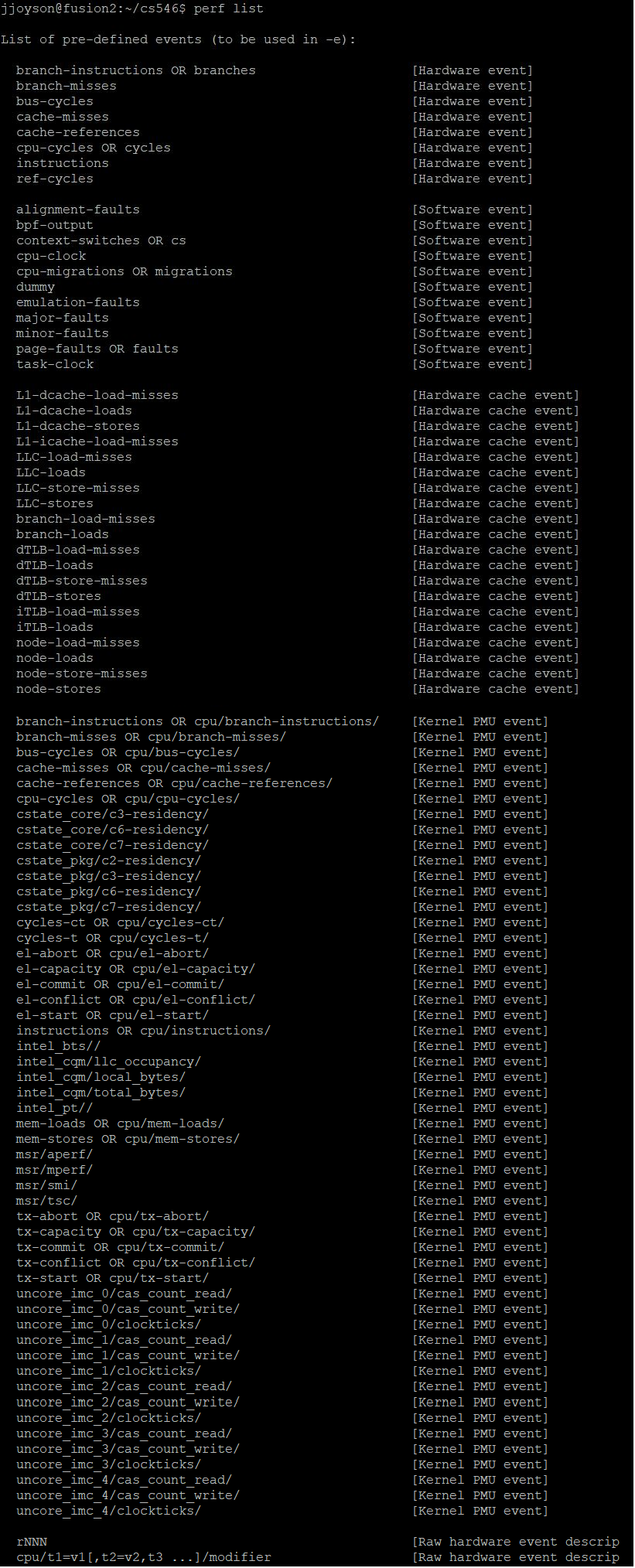
**Optimizations:**

The code was already provided so no optimizations were done. Three algorithms were provided: naïve, Interchange and Triangular. Only naïve and Interchange were used:

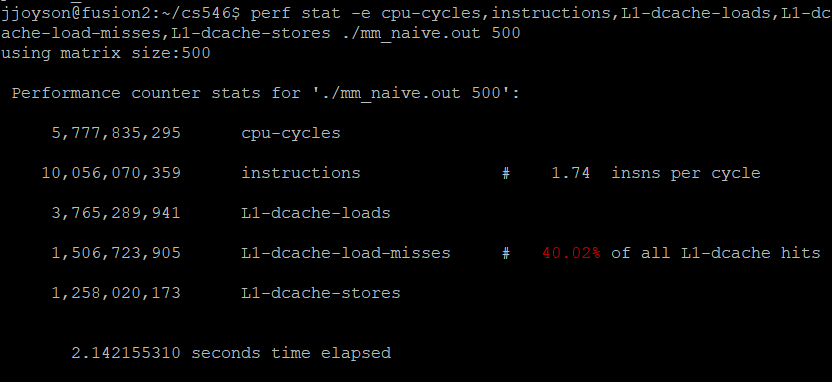
**Results & Analysis:**

Exercise 1

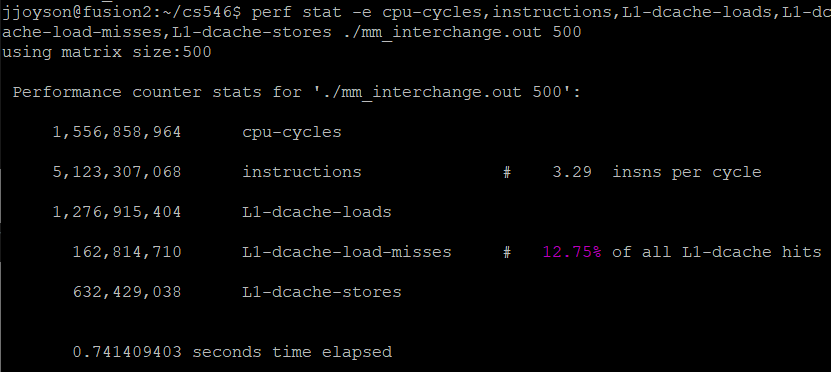
****

Exercise 2









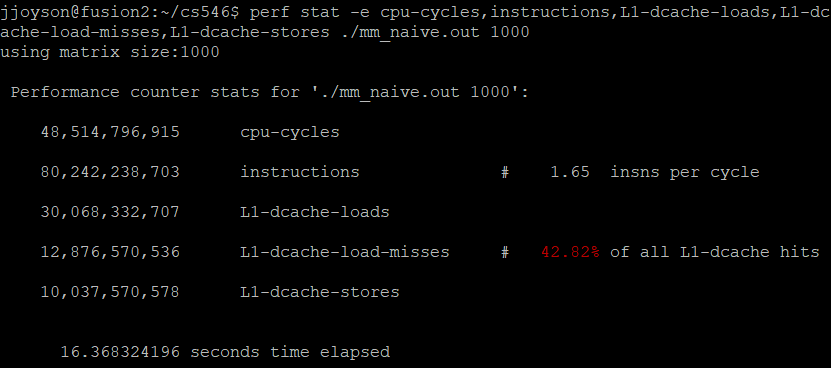
1. Compare the numbers for both cases.

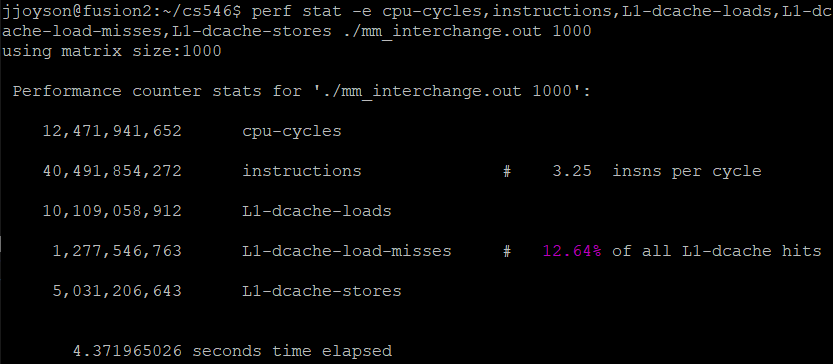
The CPU cycles are much lower (27% of naïve version) for the interchange version. Which leads to lower elapsed time. The number of instructions is much lower (51% of naïve version) for the interchange version. Therefore, there are fewer loads. There is much lower (11% of naïve version) LL1 load misses, which indicates better data locality; the loads and stores are significantly less due to this.

Based on the code, the interchange function has the inner most loop access one value (A[i][k]) and 2 matrix size values (B[k][j] and C[i][j]). Thus, A[i][k] is always accessed since the inner loop only increments j (impacting B[k][j] and C[i][j]). The cache doesn’t need to clear A[i][k] and can have better locality since the row of the matrix access on B & C is static (in B[k][j] and C[i][j] only j changes in the inner loop).

The naïve functions also has the innermost loop access 2 matrix size values (A[i][k] and B[k][j]) and on value (C[i][j]). Once again C[i][j] is always accessed since the inner loop only increments k (impacting A[i][k] and B[k][j]). The difference here is that even though there is row-based access on A (in A[i][k] only the column is impacted by the iterations) for a better locality, but access on B is horrible (B[k][j] has its row being varied in each access). This is because when the matrix is loaded into the cache, the temporal locality is increased when the elements are the same rows are access since the rows are loaded together. When the rows are varied, we cannot take advantage of temporal locality and more misses are produced (the cache needs to be cleared for new loading).

Exercise 3

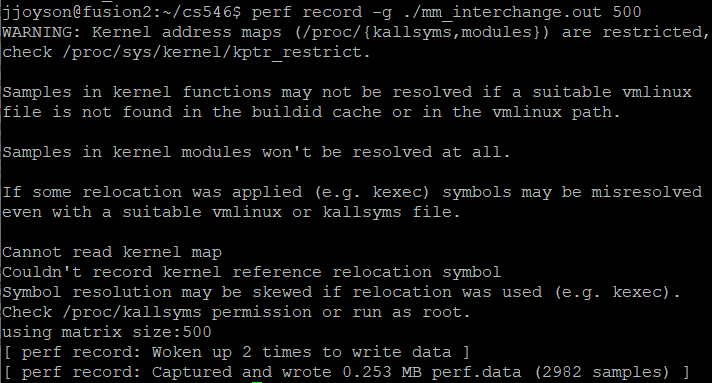




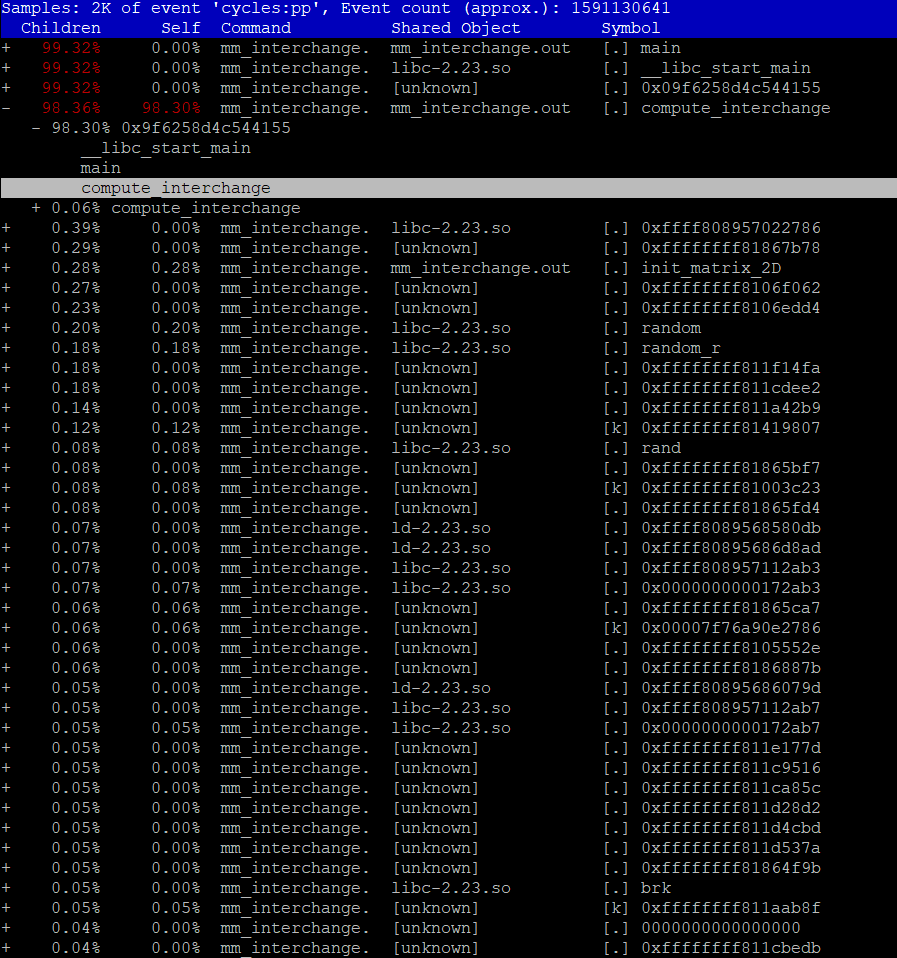
The CPU cycles are much lower (26% of naïve version) for the interchange version. Which leads to lower elapsed time. The number of instructions is much lower (50% of naïve version) for the interchange version. Therefore, there are fewer loads. There is much lower (10% of naïve version) LL1 load misses, which indicates better data locality; the loads and stores are significantly less due to this.

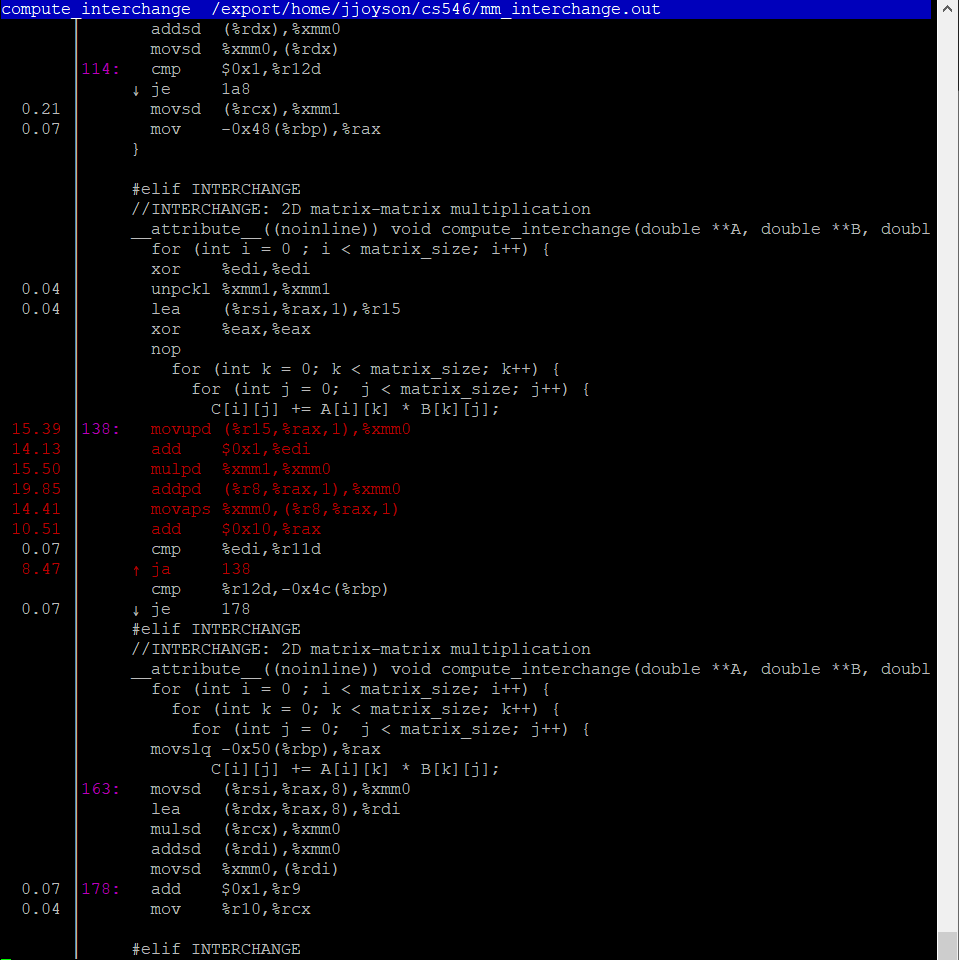
As the matrix sizes increased, the percent of the difference between naïve and interchange are the same. This means the Interchange algorithm optimization is scalable.

**Exercise 4**



Not root user so symbol naming problems but doesn’t matter in cost calculations





Ordering these by cost (most costliest to least):

1. addpd (19.85)
2. mulpd (15.50)
3. movupd (15.39)
4. movaps (14.41)
5. add (14.13, 10.51)
6. ja (8.47)

All the other functions are below 1.00 so the cost is negligible.