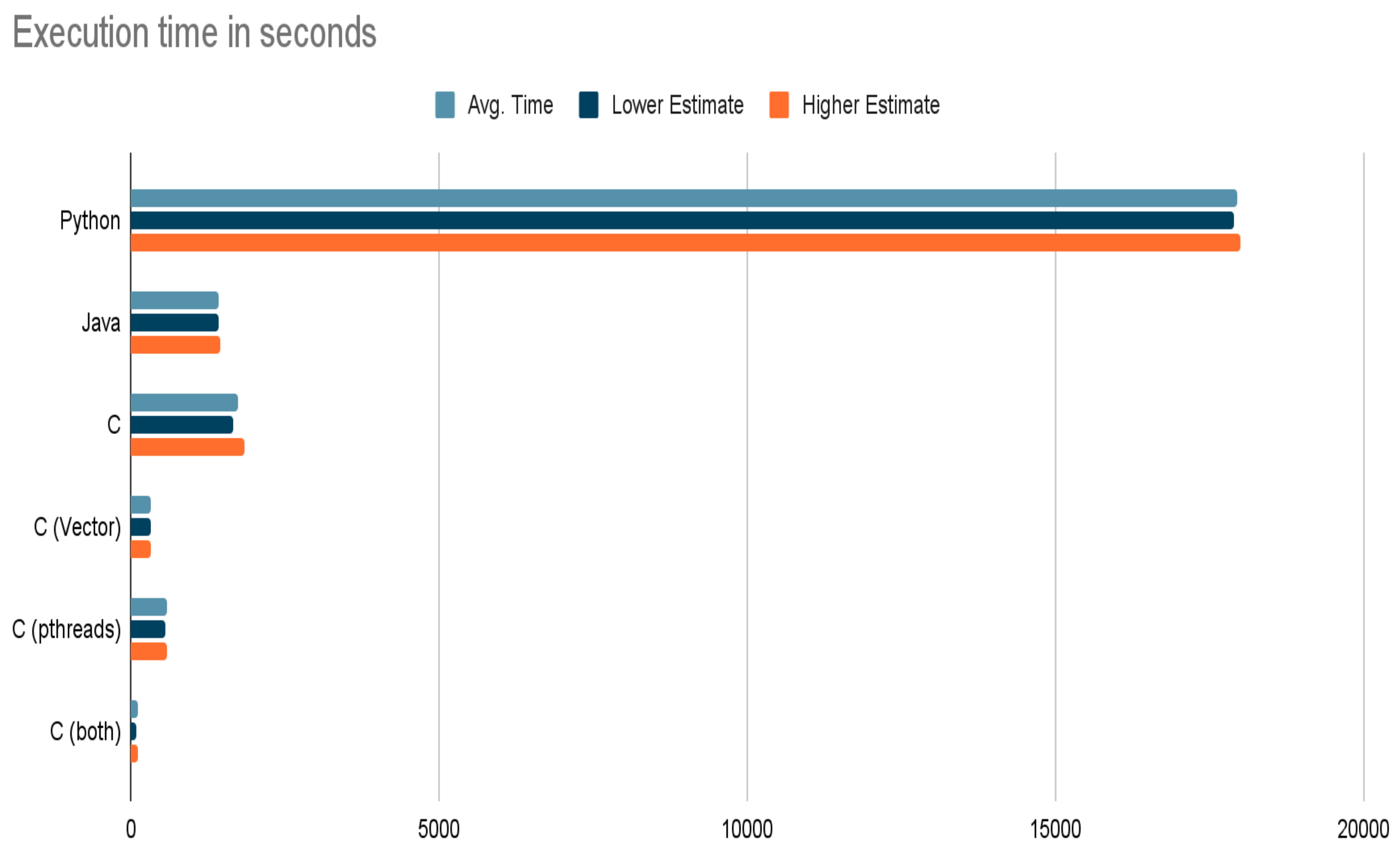
Matrix Multiplication Speedup Comparison
By
Sarmad Siddique

CPU Information:

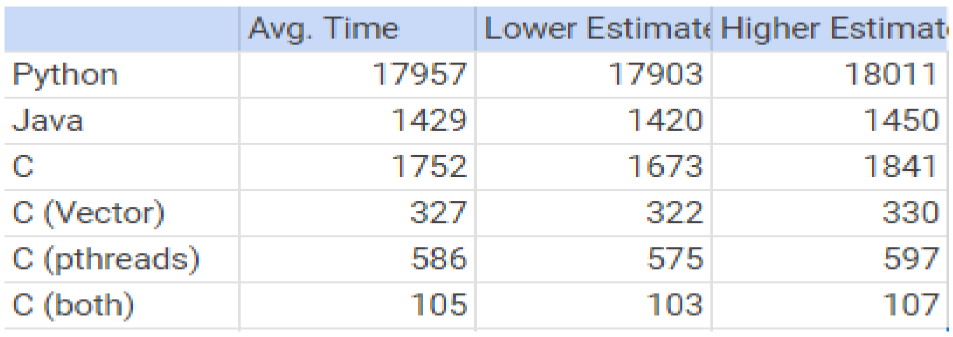
Mode name:	Intel(R) Core i5-9300H
Processor Family:	Sky Lake
Processor Vendor:	GenuineIntel
Vector Capabilities:	AVX2
No. of cores:	4
# of hyper-threads:	8 (VM shows 4, but actual performance seems to be of 8)
RAM amount:	4 GB
Disk Type:	SSD
CPU Utilization:	1% (of VM)
Memory Utilization:	0.4% (of VM)

***All codes were run on a VM, which probably increased the actual time by a considerable amount.**

Graph:



Readings for Graph:



	Avg. Time	Lower Estimat	Higher Estimat
Python	17957	17903	18011
Java	1429	1420	1450
C	1752	1673	1841
C (Vector)	327	322	330
C (pthreads)	586	575	597
C (both)	105	103	107

Notes Regarding Graph:

- The most surprising result in the graph is that Java executes matrix multiplication faster than C. But there is one caveat, while testing the code I was not aware that by default java JIT compiler optimizes the code which in turn gives the time it takes to run the optimized java code, not the actual java code. So, it's fair that we see what happens when we let the C compiler optimize the code, and as it turns out C is faster and it's not even close. The C code optimized by GCC gives us a time of just **27 seconds.**
- Another thing to note is that these times are for 32 bit integer matrices only. For normal C code (and with pthreads), the difference between different data types is negligible. However, for python and C with vector instructions the difference between data types is quite clear. For python, double is almost 5400 seconds slower, while for C with vector instructions, the code is almost 2x slower for 64 bit int and 64 bit floats.

General Insights:

- To tackle underflow/overflow situations, I have assumed the given data will not have any underflow/overflow scenarios. The attached Create_input.c file also generates random values that will never result in any underflow/overflow situation. The underflow/overflow can be detected, however the added overhead will slow down the time, thus it was not used.
- Underflow/Overflow in vector instructions can be dealt with using saturation, however that still would result in inaccuracies.
- For C program with threading, code executed in 767 seconds when using 4 threads, thus the optimal threads for maximum speedup were 8, same as the actual threads available to the processor. With 10 threads, the code executed in 723 seconds. Thus the performance deteriorated by adding more threads than available.
- One possible reason why pthreads is slower than vector instructions could be due to the small amount of RAM available to the VM.

Part-5:

(a)

- This assignment allowed me to learn different ways to speed up code. Also gave me understanding of how different factors affect code speed up.
- One interesting tidbit is that actual code also matters. This actual matrix multiplication is done using the naive algorithm. However, by just rearranging the loop order we can make sure that there are fewer cache misses. This allows us to bring the execution time of C code with pthreads and vector instructions from **103 seconds to just 26 seconds**, which is faster than execution time of serial C code optimized by GCC compiler. This modified code for 32 bit integers is given at the end.

(b)

- The best speedup was going from python to C. Changing just the language allowed us to speed up the code by almost 13 times. However, we don't necessarily have to change the language, we can use the numpy library and achieve extremely fast times as it is written in C and provides other ways to speed up matrix multiplication.

(c)

- Assuming that language change is 1 processor added and addition of pthreads or vectorization adds another processor then the quality of the speedup as mentioned in the graph is decreasing and is plateauing as shown in the graph that the graph is following a flattening pattern. This shows that the code that can be sped up by parallelization methods cannot be sped up more (or atleast the speed up will not be as drastic) by adding more parallelization or more processors for that matter.

***The following rearranged code of pthreads(8) + vector instructions provides the fastest time(26 seconds).**

```
for (i = arguments->start_row; i < arguments->end_row; i++)
{
    for (k = 0; k < 4096; k++)
    {
        for (j = 0; j < 4096; j+=8)
      {
            __m256i sum = _mm256_setzero_si256();

            __m256i bc_mat1 = _mm256_set1_epi32(arguments->matrix1[i][k]);
            __m256i vec_mat2 = _mm256_loadu_si256((__m256i*)&arguments->matrix2[k][j]);
            __m256i prod = _mm256_mullo_epi32(bc_mat1, vec_mat2);

            __m256i vec_mat3 = _mm256_loadu_si256((__m256i*)&arguments->matrix3[i][j]);
            sum = _mm256_add_epi32(vec_mat3, prod);
            _mm256_storeu_si256((__m256i*)&arguments->matrix3[i][j], sum);
      }
    }
}
```