20[redacted]
Castor, John Danielle T
5-20-23

For reference, the specifications of the machine used includes:

Base speed:	3.30 GHz
Sockets:	1
Cores:	6
Logical processors:	12
Virtualization:	Enabled
L1 cache:	384 KB
L2 cache:	3.0 MB
L3 cache:	16.0 MB

I. MMM w/o vs. w/ blocking

A. Large matrix outputs- n=512, N=32

```
MMM w/o blocking completed in 10.056355 seconds
MMM with blocking completed in 10.722050 seconds
Verifying if blocking and nonblocking algorithms have equal results...
OK!
```

```
MMM w/o blocking completed in 12.787135 seconds
MMM with blocking completed in 11.196037 seconds
Verifying if blocking and nonblocking algorithms have equal results...
OK!
```

```
MMM w/o blocking completed in 10.502210 seconds
MMM with blocking completed in 10.734766 seconds
Verifying if blocking and nonblocking algorithms have equal results...
OK!
```

*N=8

```
MMM w/o blocking completed in 13.336075 seconds
MMM with blocking completed in 10.007984 seconds
Verifying if blocking and nonblocking algorithms have equal results...
OK!
MMM w/o blocking completed in 12.850117 seconds
MMM with blocking completed in 9.971657 seconds
Verifying if blocking and nonblocking algorithms have equal results...
OK!
```

```
MMM w/o blocking completed in 13.013801 seconds
MMM with blocking completed in 10.059013 seconds
Verifying if blocking and nonblocking algorithms have equal results...
OK!
```

*N=128

```
MMM w/o blocking completed in 11.887924 seconds
MMM with blocking completed in 14.284716 seconds
Verifying if blocking and nonblocking algorithms have equal results...
OK!
MMM w/o blocking completed in 12.732181 seconds
MMM with blocking completed in 14.337713 seconds
Verifying if blocking and nonblocking algorithms have equal results...
OK!
```

```
MMM w/o blocking completed in 12.759716 seconds
MMM with blocking completed in 14.407070 seconds
Verifying if blocking and nonblocking algorithms have equal results...
OK!
```

*N=1

```
MMM w/o blocking completed in 11.043046 seconds
MMM with blocking completed in 11.135987 seconds
Verifying if blocking and nonblocking algorithms have equal results...
OK!
```

*N=256

```
MMM w/o blocking completed in 12.614729 seconds
MMM with blocking completed in 23.965489 seconds
Verifying if blocking and nonblocking algorithms have equal results...
OK!
```

B. Small matrix outputs- n=32, N=4

```
MMM w/o blocking completed in 0.002998 seconds
MMM with blocking completed in 0.002999 seconds
Verifying if blocking and nonblocking algorithms have equal results...
OK!
MMM w/o blocking completed in 0.002999 seconds
MMM with blocking completed in 0.003001 seconds
Verifying if blocking and nonblocking algorithms have equal results...
OK!
```

```
MMM w/o blocking completed in 0.003000 seconds
MMM with blocking completed in 0.003002 seconds
Verifying if blocking and nonblocking algorithms have equal results...
OK!
```

*N=2

```
MMM w/o blocking completed in 0.002996 seconds
MMM with blocking completed in 0.002004 seconds
Verifying if blocking and nonblocking algorithms have equal results...
OK!
```

```
MMM w/o blocking completed in 0.002005 seconds
MMM with blocking completed in 0.001999 seconds
Verifying if blocking and nonblocking algorithms have equal results...
OK!
```

```
MMM w/o blocking completed in 0.002997 seconds
MMM with blocking completed in 0.003001 seconds
Verifying if blocking and nonblocking algorithms have equal results...
OK!
```

*N=16

```
MMM w/o blocking completed in 0.003000 seconds
MMM with blocking completed in 0.005000 seconds
Verifying if blocking and nonblocking algorithms have equal results...
OK!
MMM w/o blocking completed in 0.002996 seconds
MMM with blocking completed in 0.004001 seconds
Verifying if blocking and nonblocking algorithms have equal results...
OK!
```

```
MMM w/o blocking completed in 0.003000 seconds
MMM with blocking completed in 0.004998 seconds
Verifying if blocking and nonblocking algorithms have equal results...
OK!
```

*Questions

1. Increasing N will decrease block size. N simply is the number where n (square matrix size) is divided to obtain block size.
2. For small matrices, the MMM w/ blocking algorithm still somehow performs better, but the improvements are small.
3. Blocking improves MMM as traversals are done w/ shorter strides, minimizing cache trashing/pollution per floating point operation.
4. For large matrices, MMM w/ blocking shows significant improvements over ijk MMM. The N value must be appropriately chosen though to see improvements; too large of an N value will worsen runtimes, too small will show no improvements.
5. Ideally, N=n * √(3/M), where M is the cache size. If too large of N values are used, the cache might not be fully utilized (just use a small portion of its capacity). If smaller N values are used, more data will be required to be put on cache, and if cache becomes congested, means more penalties and cache misses.

II. MMM w/ blocking vs. BLAS

A. Large matrix outputs- n=512, N=32

```
MMM with blocking completed in 10.754184 seconds
Numpy matmul() completed in 0.018868 seconds
MMM with blocking completed in 10.728648 seconds
Numpy matmul() completed in 0.018716 seconds
```

```
MMM with blocking completed in 10.710383 seconds
Numpy matmul() completed in 0.018999 seconds
```

*N=8, better runtimes for MMM w/ blocking

```
MMM with blocking completed in 9.961703 seconds
Numpy matmul() completed in 0.019098 seconds
```

B. Small matrix outputs- n=32, N=4

```
MMM with blocking completed in 0.002999 seconds
Numpy matmul() completed in 0.000000 seconds
MMM with blocking completed in 0.002999 seconds
Numpy matmul() completed in 0.000000 seconds
```

```
MMM with blocking completed in 0.001999 seconds
Numpy matmul() completed in 0.000000 seconds
```

*Questions-

1. OpenBlas64 library is what Numpy uses.
2. For my machine, OpenBlas64 library is also used.

2. The performances wrt. Runtimes of the said 2 algorithms are shown above. Clearly, the difference is very significant, almost 1000x better, in favor of matmul().
3. Yes.
4. Reason is that BLAS is a standard library for common operations, optimized and standardized according to the machine and its architecture used.
5. It would be so difficult to write a comparable code w/o using external libraries.
6. If a program involves lots of linear algebra solving, reordering scalar-matrix and vector-matrix multiplications to MMMs will yield runtime improvements.

III. Faster MMMs

*Own implementation vs. previous algs. Output comparison

```
MMM with blocking completed in 1.197063 seconds
Numpy matmul() completed in 0.004999 seconds
Own implementation completed in 0.004665 seconds
c_block
[[64.69837784698568, 60.61316058491624, 62.84946296149736, 70.0471232985917, 61.708760330863846, 67.1130614526930 5, 62.25434119760566, 69.1133843945812, 61.72395427259624, 67.800927379(

c_np
[[64.69837785 60.61316058 62.84946296 ... 67.71468957 68.93012681
  64.29908184]
 [63.30335919 63.63983966 58.41819556 ... 65.82095508 66.04483768
  62.88815457]
 [58.80360249 56.37033396 55.09429161 ... 59.86988855 60.60671825
  55.46959919]
 ...
 [60.74169219 60.32919767 56.83897    ... 59.94979409 61.75785398
  58.26216974]
 [64.9045398  63.35349122 59.59760007 ... 66.37244226 67.14046056
  62.75118319]
 [63.72974107 61.91047708 60.27629087 ... 64.99048441 65.35289208
  62.63167235]]

c_own
[[64.69838  60.61318  62.849464 ... 67.71466  68.930084 64.29908 ]
 [63.303364 63.639847 58.418194 ... 65.820946 66.044846 62.888126]
 [58.803604 56.37033  55.09429  ... 59.869892 60.60671  55.469593]
...
 [60.74169  60.329197 56.838955 ... 59.949802 61.757847 58.26216 ]
 [64.90454  63.353485 59.597595 ... 66.37246  67.14047  62.751198]
 [63.729767 61.910465 60.276287 ... 64.990524 65.3529   62.63165 ]]

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

A. Large matrix outputs- n=512, N=8

```
MMM with blocking completed in 9.818000 seconds
Numpy matmul() completed in 0.017995 seconds
Own implementation completed in 0.014998 seconds
MMM with blocking completed in 9.841331 seconds
Numpy matmul() completed in 0.019004 seconds
Own implementation completed in 0.014996 seconds
```

```
MMM with blocking completed in 9.921428 seconds
Numpy matmul() completed in 0.019001 seconds
Own implementation completed in 0.015999 seconds
```

B. Small matrix outputs- n=32, N=4

```
MMM with blocking completed in 0.003002 seconds
Numpy matmul() completed in 0.000000 seconds
Own implementation completed in 0.000000 seconds
MMM with blocking completed in 0.001995 seconds
Numpy matmul() completed in 0.000000 seconds
Own implementation completed in 0.000000 seconds
```

```
MMM with blocking completed in 0.003000 seconds
Numpy matmul() completed in 0.000000 seconds
Own implementation completed in 0.000000 seconds
```

*Questions

1. An improved way than matmul() is to directly use BLAS. First is to import scipy, then use the syntax scipy.linalg.blas.sgemm(<float scalar>, <matrix 1>, <matrix 2>). This will improve the runtime as it is a direct subroutine call, and that the data might already be prepared/constructed to be used w/ BLAS.
2. Yes, quite significantly especially for large matrices.