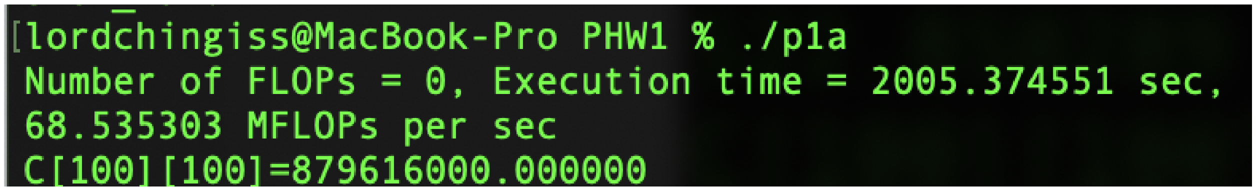
CS/EE 451 PHW1

Name: Jack Li
ID: 1321-1056-14

P1a

Running on Mac OS once, the output is:

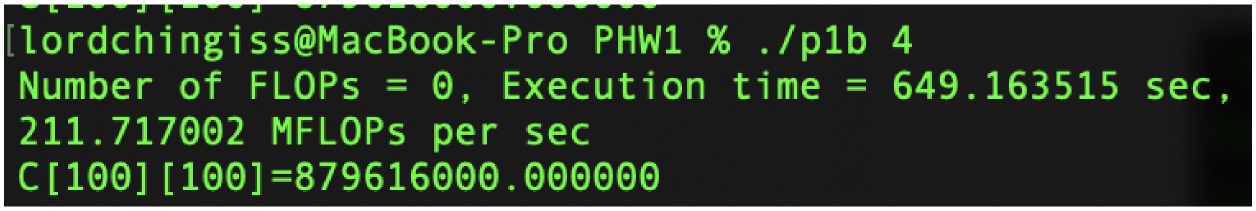


```
[lordchingiss@MacBook-Pro PHW1 % ./p1a
 Number of FLOPs = 0, Execution time = 2005.374551 sec,
 68.535303 MFLOPs per sec
 C[100][100]=879616000.000000
```

P1b

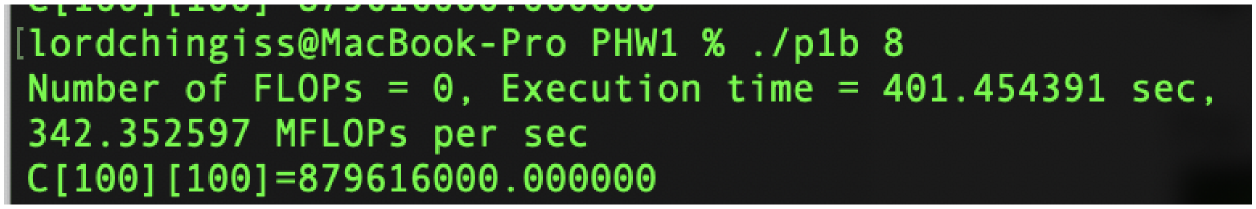
Running on Mac OS once per block size

For a block size of 4:



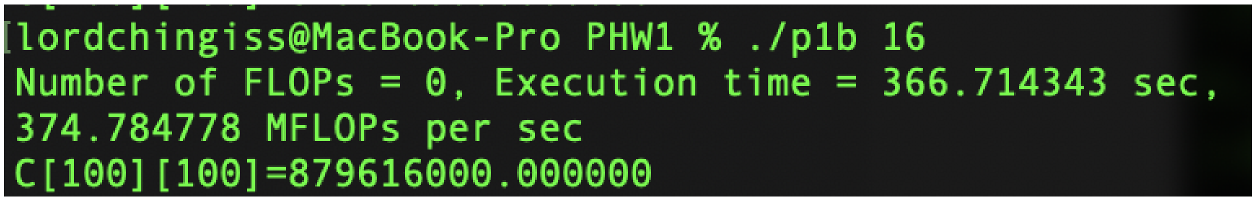
```
[lordchingiss@MacBook-Pro PHW1 % ./p1b 4
 Number of FLOPs = 0, Execution time = 649.163515 sec,
 211.717002 MFLOPs per sec
 C[100][100]=879616000.000000
```

For a block size of 8, the output is:



```
[lordchingiss@MacBook-Pro PHW1 % ./p1b 8
 Number of FLOPs = 0, Execution time = 401.454391 sec,
 342.352597 MFLOPs per sec
 C[100][100]=879616000.000000
```

For a block size of 16, the output is:

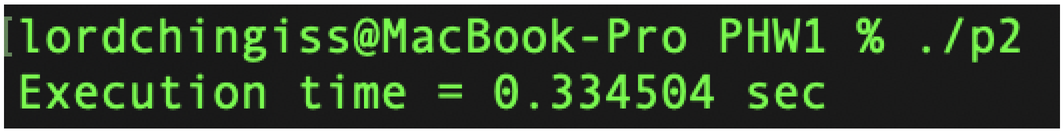


```
[lordchingiss@MacBook-Pro PHW1 % ./p1b 16
 Number of FLOPs = 0, Execution time = 366.714343 sec,
 374.784778 MFLOPs per sec
 C[100][100]=879616000.000000
```

The general trend is that performance improves (shorter execution time) with increasing block size. It's also important to note that there is not much improvement from using block size of 16 vs 8. This is because the cache capacity has almost been reached; too big of a block size eventually will not improve (and probably hurt) performance. In all cases, performing block matrix multiplication is more efficient than the naive implementation since more data elements can be brought into the cache, taking advantage of temporal locality. Also note that the number of FLOPS is 2 * n * n * n for both naive and blocked implementations but shows as 0 due to overflow; since n is 4096, the number of FLOPS is 137438953472.

P2:

Running on Mac OS once, the output is:



```
[lordchingiss@MacBook-Pro PHW1 % ./p2
Execution time = 0.334504 sec
```

And the output image is: