Lab 12

NAME: Mehmet Turhan

Part I: Dynamic Memory Allocation

N	**Time (s)**
512	.006
1024	.009
2048	.020
4096	.062

Part II: Matrix Addition

N	**Time (s)**
512	.011
1024	.016
2048	.048
4096	.174

Part III: Matrix Multiplication

N	**Time (s)**
128	.015
256	.078
512	.589
1024	5.313
2048	116.916

Part IV: Spatial Locality in Matrix Multiplication (*N* = 2048)
First Run:

Version	**Time (s)**	**Rank**
ijk	116.916	4
ikj	20.680	2
jik	104.666	3
jki	160.776	6
kij	20.364	1
kji	122.907	5

Second Run:

Version	**Time (s)**	**Rank**
ijk	118.310	4
ikj	20.506	1
jik	76.804	3
jki	140.488	5
kij	20.588	2
kji	151.0160	6

Justification for your ranking:

In the first run, kij was ranked first and and jki was ranked last. But in the second run ikj was ranked first and kji was ranked last. So, ikj and kij have the best ranking because these functions` for loops go row by row instead of columns, this increases the efficiency of a better spacial locality which runs the program faster. Jki and kji have the worst ranking because these functions` for loops go column by column instead of rows, which is worse for the spacial locality. Therefor jki and kji has the longest time. When you access it row by row, you have a pattern in memory, and it’s easier to load those addresses into cache. So there is a higher percentage of hitting the cache. But, if you access it column by column, you won’t have a pattern so it’s going to be harder to load those memory addresses. There is a lower percentage of hitting the cache, so we will go to the main memory more frequently to hit it, which will make it run slower.

[Optional] Part V: Block Matrix Multiplication (N = 2048)

b	**Time (s)**	**Rank**
32		
64		
128		
256		
512		