REPORT FOR ASSIGNMENT 1

Tried all the below experiment on matrix dimension 1024 * 1024.

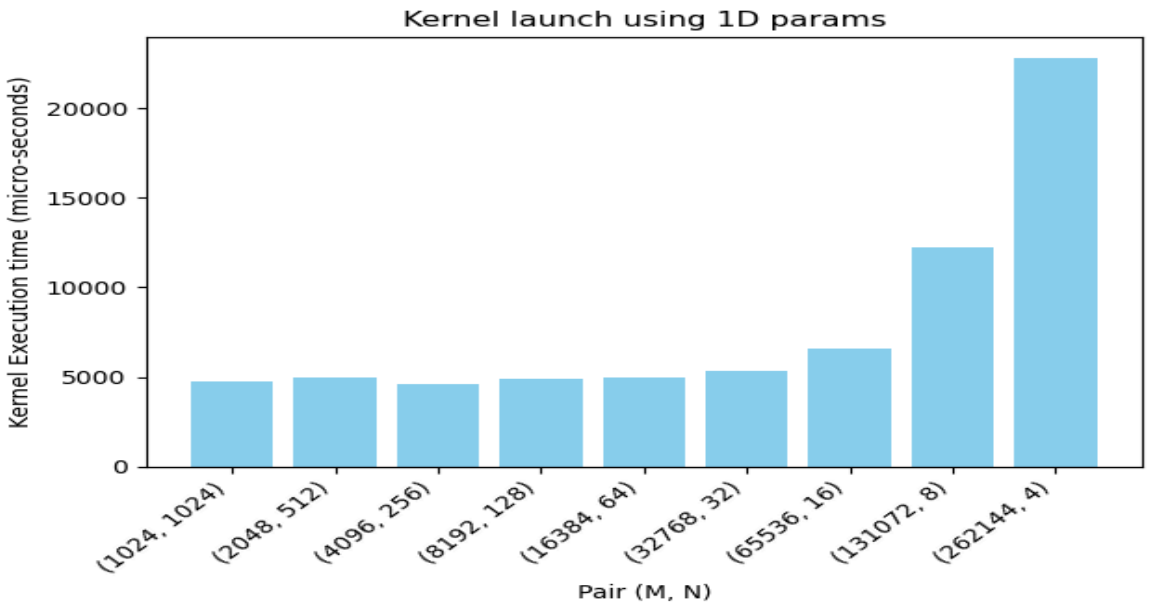
Question 1.1: Basic Version of Matrix Multiplication(Kernel launch using 1D params)

Tried this experiment on matrix dimension 1024 * 1024.

Trial No	#Thread Blocks (M)	Block Size i.e. #Threads in a block (N)	Kernel Execution time (microsecs)	Your insights on the execution time
1	1024	1024	4748.735 ms	Since #thread blocks is less so, less #global mem transaction compare to others
2	2048	512	4945.184 ms	Comparatively #thread blocks is less so, less #global mem transaction compare to others
3	4096	256	4584.000 ms	Runs in less time since it balances with all parameters
4	8192	128	4932.064 ms	Increase in #thread block results increase of thread blocks in waiting and execution time also increases
5	16384	64	4995.520 ms	Increase in #thread block results increase

				of thread blocks in waiting and execution time also increases
6	32768	32	5356.640 ms	Increase in #thread block results increase of thread blocks in waiting and execution time also increases
7	65536	16	6583.584 ms	Takes slightly more time than avg since the #thread blocks more
8	131072	8	12263.968 ms	Takes more time than avg since more #thread blocks in the scheduling
9	262144	4	22808.737 ms	Takes more time than avg since more #thread blocks in the scheduling

Result: (4096, 256) has less execution of 4584.000 micro seconds.



Question 1.2 Basic Version of Matrix Multiplication(Kernel launch using 2D params)

Tried this experiment on matrix dimension 1024 * 1024.

Trial No	2D Grid Size (X1,Y1)	2D Block Size (X2,Y2)	Kernel Execution time (microsecs)	Your insights on the execution time
1	(32, 32)	(32, 32)	4755.616	Executed fast but still not optimal.
2	(64, 64)	(16, 16)	4516.799	Since row size is less and column size is more, a block brought from global mem is utilized more. So less execution time.

3	(128, 128)	(8, 8)	5154.879	Since reducing block dim and increasing #thread blocks, resulting in more global mem transaction per block.
4	(256, 256)	(4, 4)	8260.576	Since reducing block dim and increasing #thread blocks, resulting in more global mem transaction per block.
5	(512, 512)	(2, 2)	23737.664	Execution time is too large since couple of threads per block which results in more memory transactions and scheduling overhead.
6	(1024, 1024)	(1, 1)	90515.835	Execution time is exponentially more since only one thread per block which results in more memory transaction and scheduling overhead.
7	(64, 16)	(16, 64)	4219.903	Since row size is less and column size is more, a block brought from global mem is utilized more compared to (16 * 16). So less execution time.

8	(128, 8)	(8, 128)	4703.0083	Combination results in avg execution time.
9	(1, 1024)	(1024, 1)	4399.456	Very less compared to all since the data in cache are utilized maximally and better memory coalescing.
10	(1024, 1)	(1, 1024)	24598.623	Too huge execution time since worst memory coalescing.

Result: BlockDim: (64, 16) and ThreadDim: (16, 64) have less execution of 4219.903 micro seconds.

Question 2: Tiled Version of Matrix Multiplication

Tried this experiment on matrix dimension 1024 * 1024.

Here , the #memory accesses are calculated manually by splitting each of the accesses.
#MA = number of memory access
SM = shared memory
#SMA = shared memory access

Trial No	Tile Size	#Global Memory Access	#Shared Memory Access	Kernel Execution time (microsecs)	Your insights on the execution time
1	0	**(No tiling)** **> total #MA:** **(2^21 + 1) * 2^20**	**0**	9245.600	Since no tiling, more global mem transaction cost.
2	32	**> per tile #MA per slide in mat A/B:** 32 * 32 = 2^10	**SHARED MEM A&B:** **> #MA in assigning values to SM A/B per**	5016.895996	Tile size is not optimal, not more memory

		> total #MA in mat A/B across slides: 2^10 * (2^5 slides) **> #MA in A:** (2^10 * 2^5)*(2^10 #tiles) **> #MA in B:** (2^10 * 2^5)*(2^10 #tiles) **>#MA in mat C = rows * cols of C:** 2^10* 2^10 **> total #MA:** **65 * 20^20**	**tile per slide:** 2^10 **> #MA in fetching values from SM A/B per tile per slide:** 32*32*32 = 2^15 **> total #SMA in A/B per tile across slides:** (2^10 + 2^15) * (32 slides) **> total #SMA in A/B:** (2^10 + 2^15) * (32 slides) * (2^10 tiles) **<u>SHARED MEM C:</u>** **> #MA in assigning values to SM C per tile per slide:** 2^10 **> total #SMA in C per tile across slides:** 2^10 * (32 slides) **> total #SMA in C:** 2^10 * (32 slides) * (2^10 tiles) **> total #MA: 67 * 2^25**		coalescing and tile size is more, it takes more time in coping elements from/to global memory to/from shared memory per block. When the tile size is small this overhead also reduces per block.
3	16	**> per tile #MA per slide in mat A/B:** 16 * 16 = 2^8 **> total #MA in mat A/B across slides:** 2^8 * (2^6 slides) **> #MA in A:** (2^8 * 2^6) * (2^12 #tiles) **> #MA in B:** (2^8 * 2^6) * (2^12 #tiles) **>#MA in mat C = rows * cols of C:**	**<u>SHARED MEM A&B:</u>** **> #MA in assigning values to SM A/B per tile per slide:** 2^8 **> #MA in fetching values from SM A/B per tile per slide:** 16*16*16 = 2^12 **> total #SMA in A/B per tile across slides:** (2^8 + 2^12) * (2^6 slides) **> total #SMA in A/B:** (2^8 + 2^12) * (2^6	4672.28808 6	Takes less execution time since its proper memory coalescing and less global memory transaction cost is more.

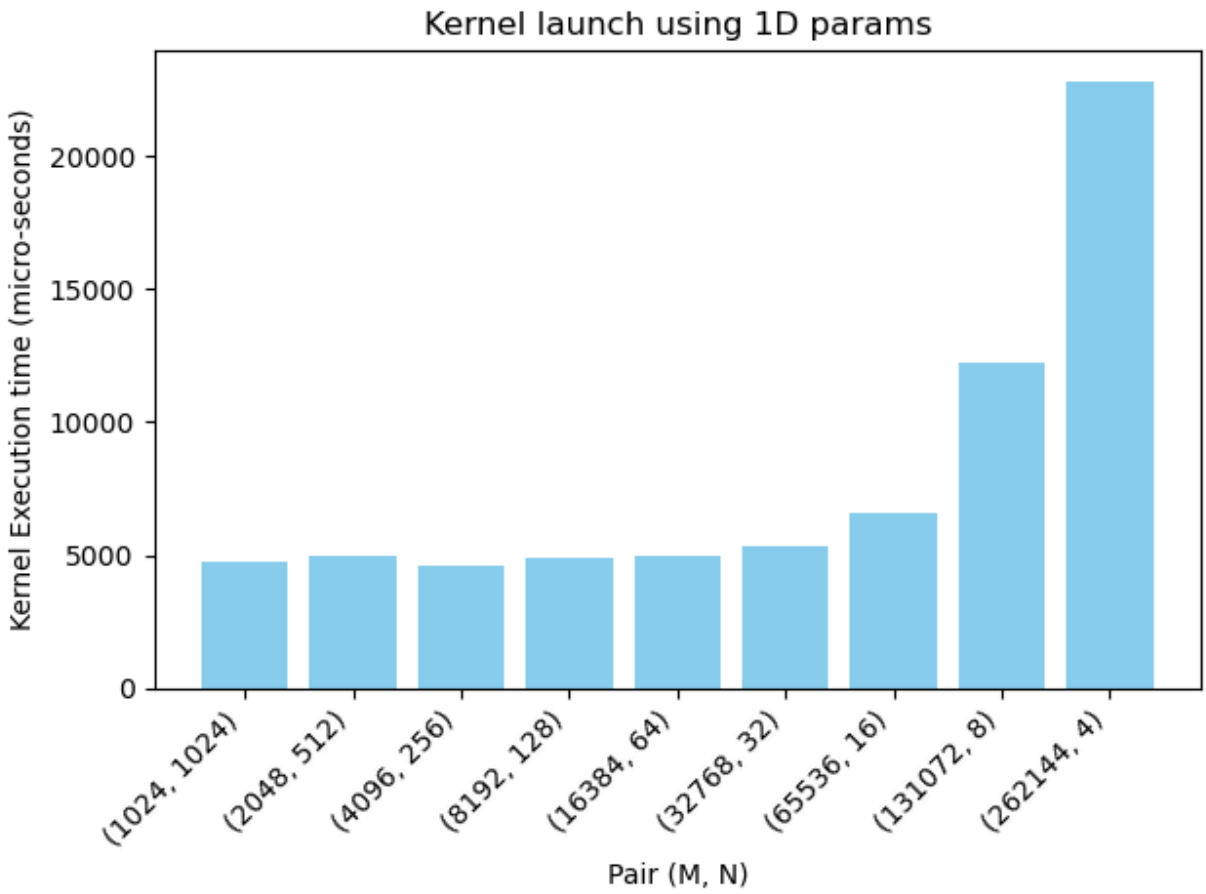
		2^10* 2^10 **> total #MA:** (2^6 + 2^6 + 1)*(2^20) = **129 * (20^20)**	slides) * (2^12 tiles) **<u>SHARED MEM C:</u>** **> #MA in assigning values to SM C per tile per slide:** 2^8 **> total #SMA in C per tile across slides:** 2^8 * (2^6 slides) **> total #SMA in C:** 2^8 * (2^6 slides) * (2^12 tiles) **> total mem.acc:** (1 + 2^4 + 1 + 2^4 + 1) * 2^26= **70 * 2^25**		
4	8	**> per tile #MA per slide in mat A/B:** 8 * 8 = 2^6 **> total #MA in mat A/B across slides:** 2^6 * (2^7 slides) **> #MA in A:** (2^6 * 2^7) * (2^14 #tiles) **> #MA in B:** (2^6 * 2^7) * (2^14 #tiles) **>#MA in mat C = rows * cols of C:** 2^10* 2^10 **> total #MA:** (2^7 + 2^7 + 1)*(2^20) = **257 * (20^20)**	**<u>SHARED MEM A&B:</u>** **> #MA in assigning values to SM A/B per tile per slide:** 2^6 **> #MA in fetching values from SM A/B per tile per slide:** 8*8*8 = 2^9 **> total #SMA in A/B per tile across slides:** (2^6 + 2^9) * (2^7 slides) **> total #SMA in A/B:** (2^6 + 2^9) * (2^7 slides) * (2^14 tiles) **<u>SHARED MEM C:</u>** **> #MA in assigning values to SM C per tile per slide:** 2^6 **> total #SMA in C per tile across slides:** 2^6 * (2^7 slides)	5621.535645	Execution time is too large since too small tile size is the worst use of memory coalescing and takes more time to execute.

			> total #SMA in C: 2^6 * (2^7 slides) * (2^14 tiles) **> total mem.acc:** (1 + 2^3 + 1+ 2^3 + 1) * 2^27= **76 * 2^25**		
5	4	**> per tile #MA per slide in mat A/B:** 4 * 4 = 2^4 **> total #MA in mat A/B across slides:** 2^4 * (2^8 slides) **> #MA in A:** (2^4 * 2^8) * (2^16 #tiles) **> #MA in B:** (2^4 * 2^8) * (2^16 #tiles) **>#MA in mat C = rows * cols of C:** 2^10* 2^10 **> total #MA:** (2^8 + 2^8 + 1)*(2^20) = **513 * (20^20)**	**<u>SHARED MEM A&B:</u>** **> #MA in assigning values to SM A/B per tile per slide:** 2^4 **> #MA in fetching values from SM A/B per tile per slide:** 4*4*4 = 2^6 **> total #SMA in A/B per tile across slides:** (2^4 + 2^6) * (2^8 slides) **> total #SMA in A/B:** ((2^4 + 2^6) * (2^8 slides) * (2^16 tiles) **<u>SHARED MEM C:</u>** **> #MA in assigning values to SM C per tile per slide:** 2^4 **> total #SMA in C per tile across slides:** 2^4 * (2^8 slides) **> total #SMA in C:** 2^4 * (2^8 slides) * (2^16 tiles) **> total mem.acc:** (1 + 2^2 + 1+ 2^2 + 1)* 2^28= **152 * 2^25**	13212.063477	Execution time is too large since too small tile size is the worst use of memory coalescing and takes more time to execute.
6	2	**> per tile #MA per**	**<u>SHARED MEM A&B:</u>**	80259.2890	Execution

		slide in mat A/B: 2 * 2 = 2^2 **> total #MA in mat A/B across slides:** 2^2 * (2^89slides) **> #MA in A:** (2^2 * 2^9) * (2^18 #tiles) **> #MA in B:** (2^2 * 2^9) * (2^18 #tiles) **>#MA in mat C = rows * cols of C:** 2^10* 2^10 **> total #MA:** (2^9 + 2^9 + 1)*(2^20) = **1025 * (20^20)**	**> #MA in assigning values to SM A/B per tile per slide:** 2^2 **> #MA in fetching values from SM A/B per tile per slide:** 2*2*2 = 2^3 **> total #SMA in A/B per tile across slides:** (2^2 + 2^3) * (2^9 slides) **> total #SMA in A/B:** ((2^2 + 2^3) * (2^9 slides)) * (2^18 tiles) **<u>SHARED MEM C:</u>** **> #MA in assigning values to SM C per tile per slide:** 2^2 **> total #SMA in C per tile across slides:** 2^2 * (2^9 slides) **> total #SMA in C:** 2^2 * (2^9 slides) * (2^18 tiles) **> total mem.acc:** (1 + 2^1 + 1+ 2^1 + 1)* 2^29 = **112 * 2^25**		time is too large since too small tile size is the worst use of memory coalescing and takes more time to execute.
7	30	**> per tile #MA per slide in mat A/B:** 30 * 30 = 900 **> total #MA in mat A/B across slides:** 900 * (35 slides) **> #MA in A:** (900 * 35) * (1225 #tiles)	**<u>SHARED MEM A&B:</u>** **> #MA in assigning values to SM A/B per tile per slide:** 30 *30 = 900 **> #MA in fetching values from SM A/B per tile per slide:** 30*30*30 = 27000	57533.949219	Since tile size is more, it takes more time in coping elements from/to global memory to/from shared memory per block. When

		> #MA in B: (900 * 35) * (1225 #tiles) **>#MA in A&B:** 73.599 * 2^20 **>#MA in mat C = rows * cols of C:** 2^10* 2^10 **> total #MA:** (73.599 + 1)*(2^20) = **74.599 * (20^20)**	**> total #SMA in A/B per tile across slides:** (900 + 27000) * (35 slides) **> total #SMA in A/B:** ((900 + 27000) * (35 slides)) * (1255 tiles)= 1,168.7 * 2^20 **<u>SHARED MEM C:</u>** **> #MA in assigning values to SM C per tile per slide:** 30*30 = 900 **> total #SMA in C per tile across slides:** 900 * (35 slides) **> total #SMA in C:** 900 * (35 slides) * (1255 tiles)= 37.7 * 2^20 **> total mem.acc:** **74.22 * 2^25**		the tile size is small this overhead also reduces per block.
8	20	**> per tile #MA per slide in mat A/B:** 20 * 20 = 400 **> total #MA in mat A/B across slides:** 400 * (52 slides) **> #MA in A:** (400 * 52) * (2704 #tiles) **> #MA in B:** (400 * 52) * (2704 #tiles) **>#MA in A&B:** 107.275 * 2^20 **>#MA in mat C = rows**	**<u>SHARED MEM A&B:</u>** **> #MA in assigning values to SM A/B per tile per slide:** 20 * 20 = 400 **> #MA in fetching values from SM A/B per tile per slide:** 20*20*20 = 8000 **> total #SMA in A/B per tile across slides:** (400 + 8000) * (52 slides) **> total #SMA in A/B:** ((400 + 8000) * (52 slides)) * (2704 tiles)=	55647.390625	Since tile size is more, it takes more time in coping elements from/to global memory to/from shared memory per block. When the tile size is small this overhead also reduces per block.

		*** cols of C:** 2^10* 2^10 **> total #MA:** (107.275 + 1)*(2^20) = **108.275 * (20^20)**	2,252.782 * 2^20 **<u>SHARED MEM C:</u>** **> #MA in assigning values to SM C per tile per slide:** 20*20 = 400 **> total #SMA in C per tile across slides:** 400 * (52 slides) **> total #SMA in C:** 400 * (52 slides) * (2704 tiles)= 53.637 * 2^20 **> total mem.acc:** **72.075 * 2^25**		

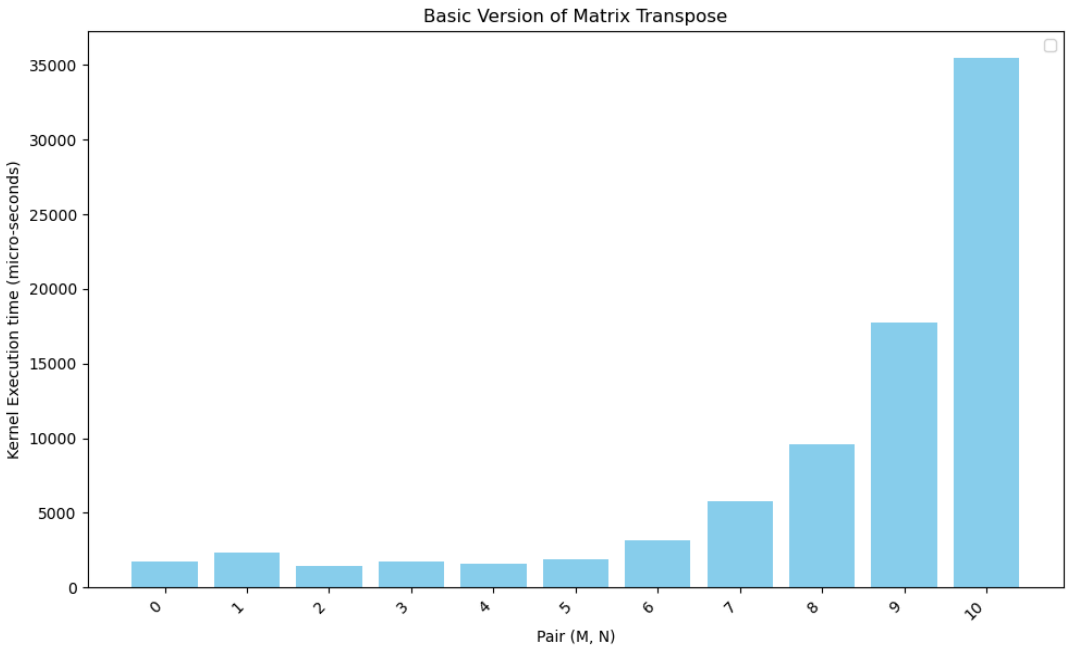


Result: Tile width: 16 has less execution of 4672.288 micro-seconds.

Question 3: Basic Version of Matrix Transpose

Trial No	#Thread Blocks (M)	Block Size i.e. #Threads in a block (N)	Kernel Execution time (microsecs)	Your insights on the execution time
1	1024	1024	1740.192017	Since #thread blocks is less so, less #global mem transaction compare to others
2	2048	512	2316.319824	Comparatively #thread blocks is less so, less #global mem transaction compare to others
3	4096	256	1468.703979	Runs in less time since it balances with all parameters
4	8192	128	1703.871948	Increase in #thread block results increase of thread blocks in waiting and execution time also increases
5	16384	64	1627.776001	Increase in #thread block results increase of thread blocks in waiting and execution time also increases
6	32768	32	1915.776001	Increase in #thread block results increase of thread blocks in waiting and execution time also increases

7	65536	16	3161.983887	Takes slightly more time than avg since the #thread blocks more
8	131072	8	5789.184082	Takes more time than avg since more #thread blocks in the scheduling
9	262144	4	9571.488281	Takes more time than avg since more #thread blocks in the scheduling
10	524288	2	17721.279297	Since very few threads per block the memory coalescing is poor and more memory transaction cost, it results in more execution time.
11	1048576	1	35480.351562	Since one thread per block the memory coalescing is poor and more memory transaction cost, it results in more execution time.



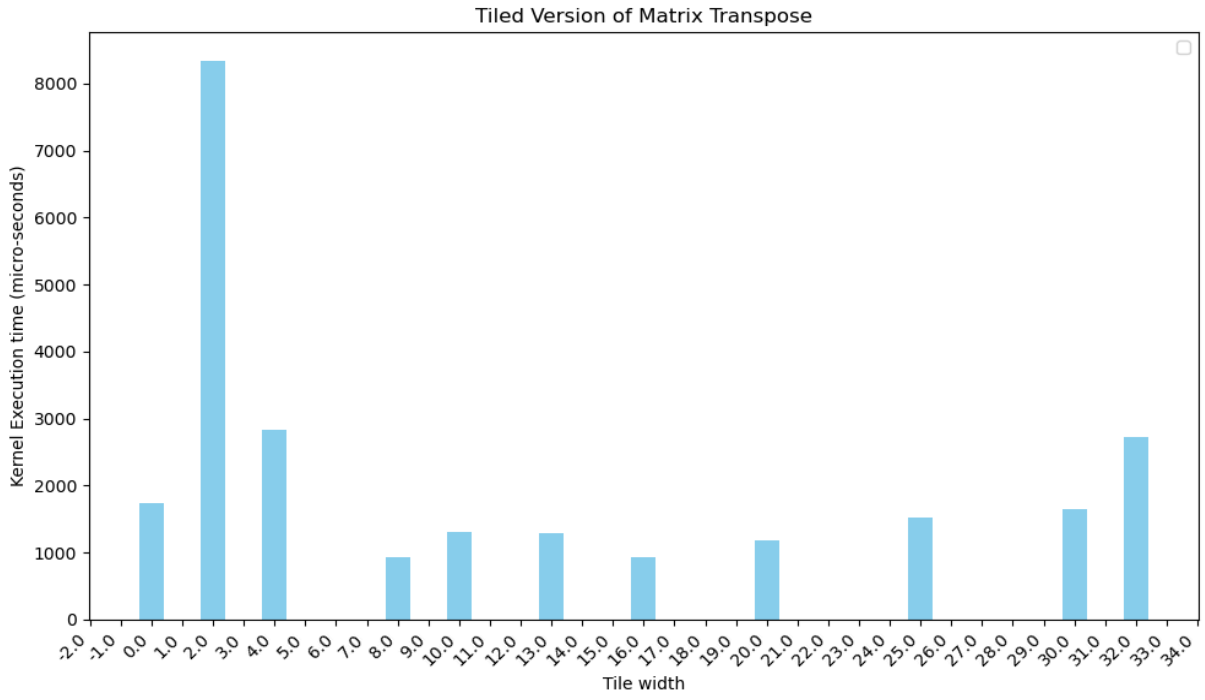
Result: Grid and thread dim (4096, 256**) has less execution of** 1468.703 **micro-seconds.**

Question 4: Tiled Version of Matrix Transpose

Trial No	Tile Size	Kernel Execution time (microsecs)	Your insights on the execution time
1	0 (No tiling)	1740.192017	Since no tiling, more global mem transaction cost.
2	32	2726.975830	Memory coalescing is not good.
3	16	924.767944	Better use of shared

			memory and better memory coalescing and results in less execution time.
4	8	922.432007	Better use of shared memory and better memory coalescing and results in less execution time.
5	4	2834.080078	Less use of shared memory and thus results in more global memory
6	2	8343.392578	Less use of shared memory and thus results in more global memory
7	30	1650.367920	Since tile size is more, it takes more time in coping elements from/to global memory to/from shared memory per block. When the tile size is small this overhead also reduces per block.
8	25	1515.488037	Since tile size is more, it takes more time in coping elements from/to global memory to/from shared memory per block. When the tile size is small this overhead also reduces per block.
9	20	1189.919922	Since tile size is more, it takes more time in coping elements from/to global memory

			to/from shared memory per block. When the tile size is small this overhead also reduces per block.



Result: Tile width: 16 has less execution of 924.767 **micro-seconds.**

*****End of report****