Matrix Multiplication in CUDA

-G. Sai Sudheer

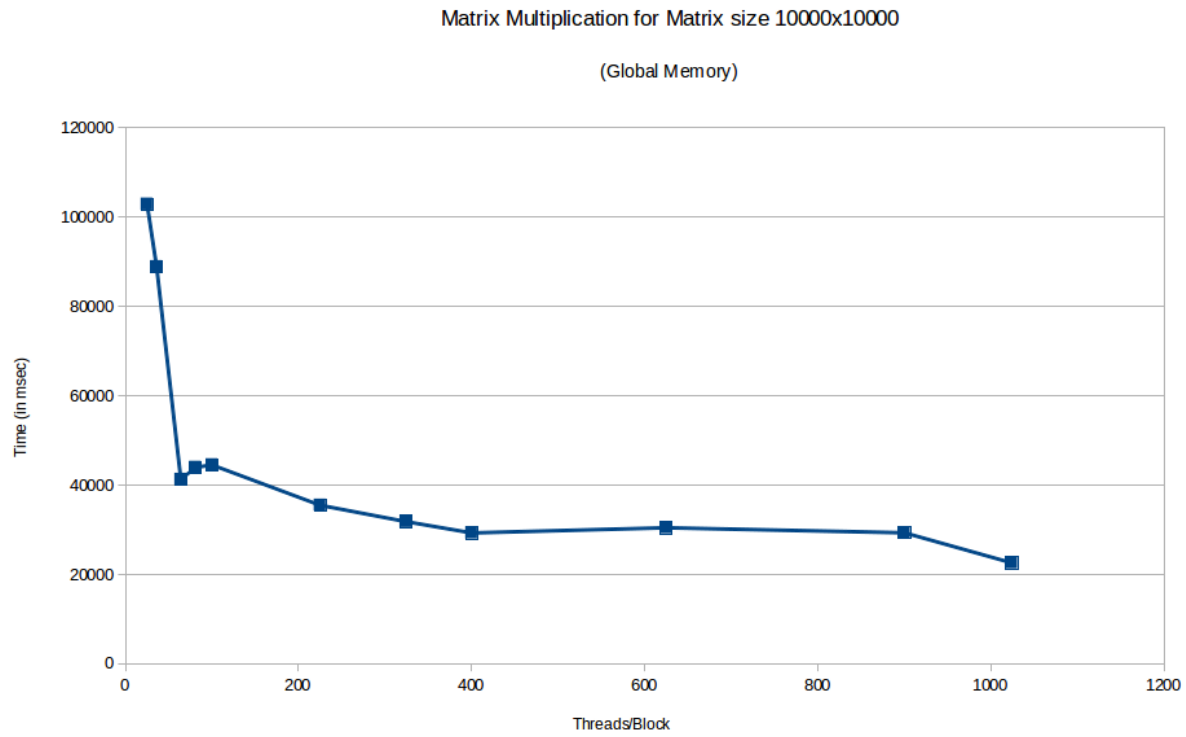
Reg. No. 13556

In this report, we show the results obtained with matrix multiplication implemented in CUDA. There were two ways in which the matrix multiplication was implemented. In the first version, the entire matrix was copied to the global memory of the device (GPU) and each thread computed one element of the output matrix by taking one row and one column of the input matrices A and B respectively. In the second version, a tiled matrix multiplication was implemented. In this, a tile is copied into the shared memory of the block. The shared memory is per block and all the threads belonging to only that block have the access to it. Therefore, each thread in the block contributes to the partial sum of an element of the output matrix.

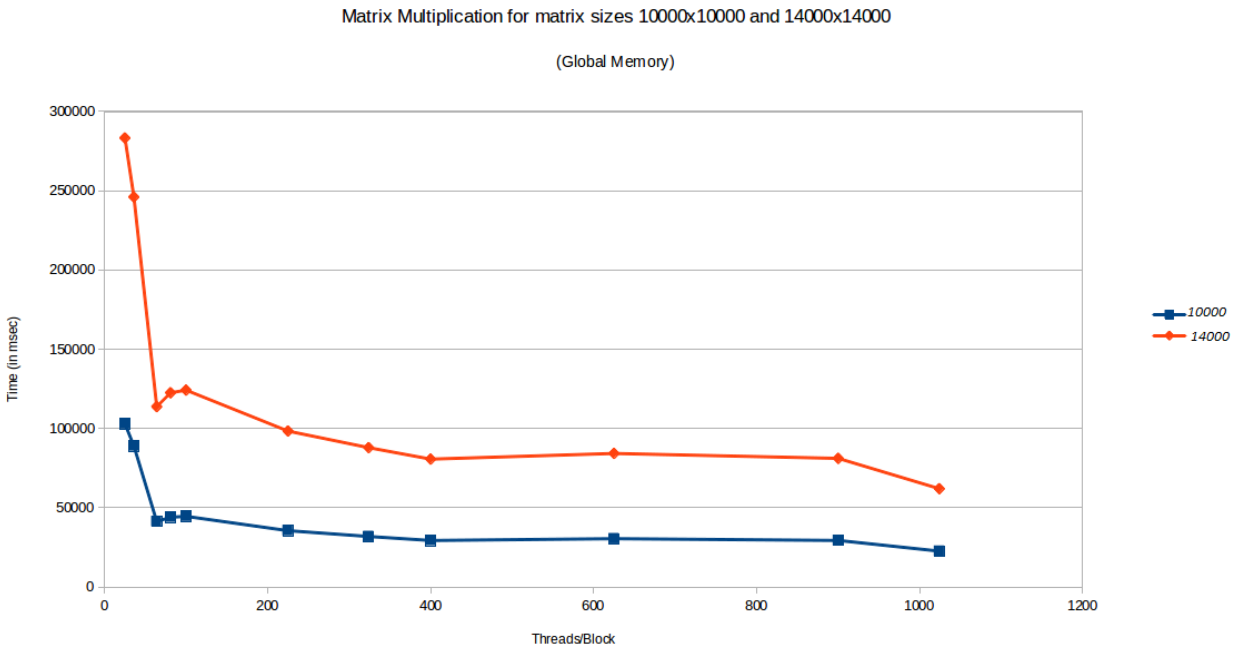
Global Memory:

In the following table, we show the time taken for matrices of different sizes and various number of threads/block. Below, we plot the time taken for matrix of size 10000x10000 for various number of threads/block. We can see that as the number of threads/block increases, the performance of the application also increases as expected.

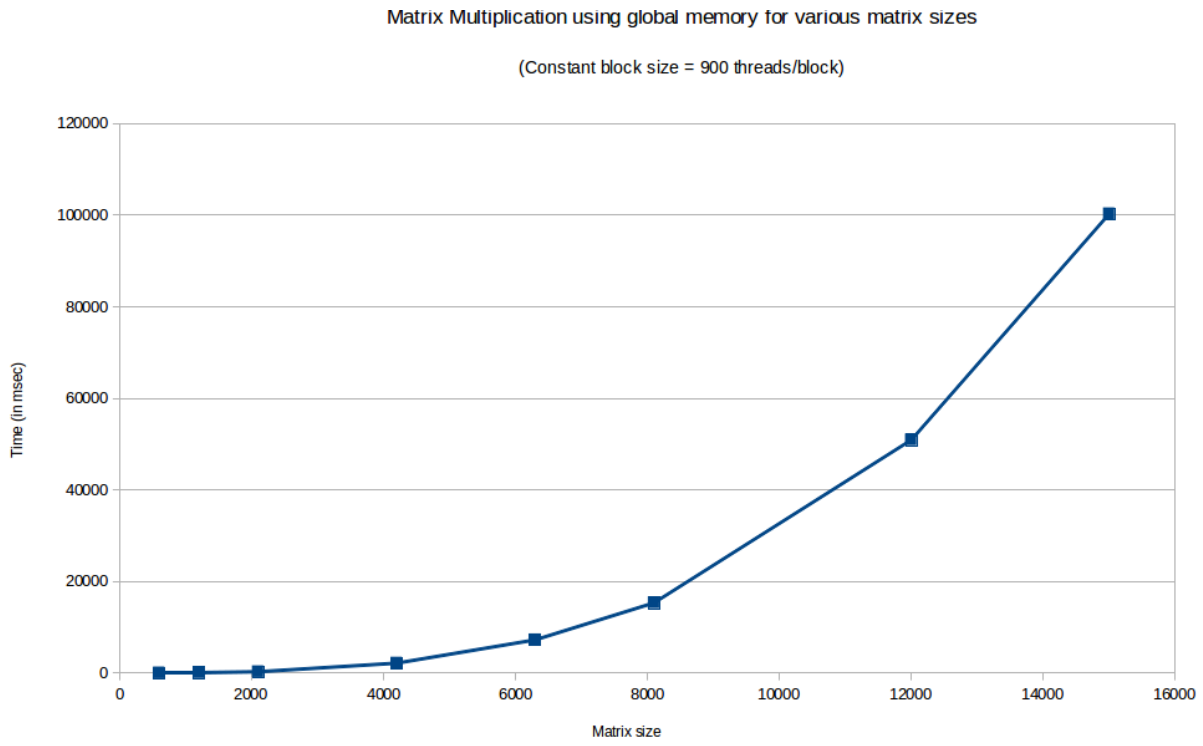
size/TPB	25	36	64	81	100	225	324	400	625	900	1024
500	11.463264	11.183136	5.096448	5.461728	5.41968	4.600896	4.06224	3.71936	3.875712	3.891136	2.83696
1000	101.606689	88.030624	40.882401	42.990143	42.140064	35.151234	31.731009	29.609505	30.395935	30.120129	24.229248
2000	813.30011	703.440308	329.853485	340.123901	338.769684	278.708069	251.450363	234.120132	238.350525	233.702148	182.958344
4000	6520.799316	5640.797852	2636.415039	2749.604248	2751.988525	2248.412598	2006.790039	1875.527344	1899.927856	1867.124634	1390.842773
6000	22103.726562	19055.207031	8896.929688	9297.380859	9356.821289	7576.967285	6791.752441	6325.845703	6470.660156	6245.531738	4891.601074
8000	52691.855469	45510.554688	21115.523438	22460.630859	22559.945312	18324.341797	16304.301758	15158.84082	15501.282227	14924.34082	11160.50293
10000	102901.757812	88895.117188	41278.730469	43986.003906	44539.117188	35507.375	31845.181641	29310.857422	30459.412109	29338.796875	22601.105469
12000	178188.171875	155197.6875	71531.171875	77282.960938	77700.0625	62670.308594	55546.019531	51429.171875	52851.859375	50887.765625	37528.628906
14000	283310.9375	246075.703125	113758.640625	122528.890625	124183.648438	98365.890625	87903.484375	80652.289062	84232.984375	81126.609375	62002.433594



In the following plot, we compare the time taken to multiply two matrices of sizes 14000x14000 and 10000x10000. We see that as the matrix size increases, the time taken for multiplying them also increases and this is similar to sequential execution in the sense that an increase in the matrix size results in an increase in the execution time.



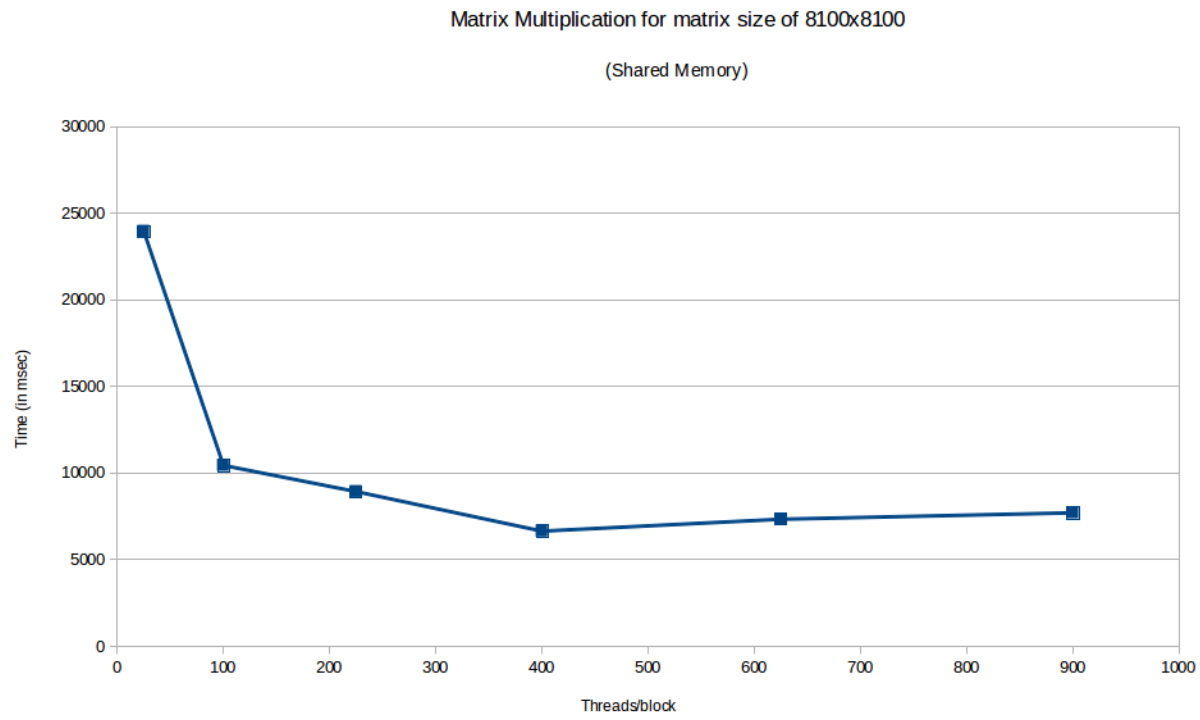
Next, we show how the execution time increases across various matrix sizes for a constant number of threads/block. We fixed the threads/block to be 900.



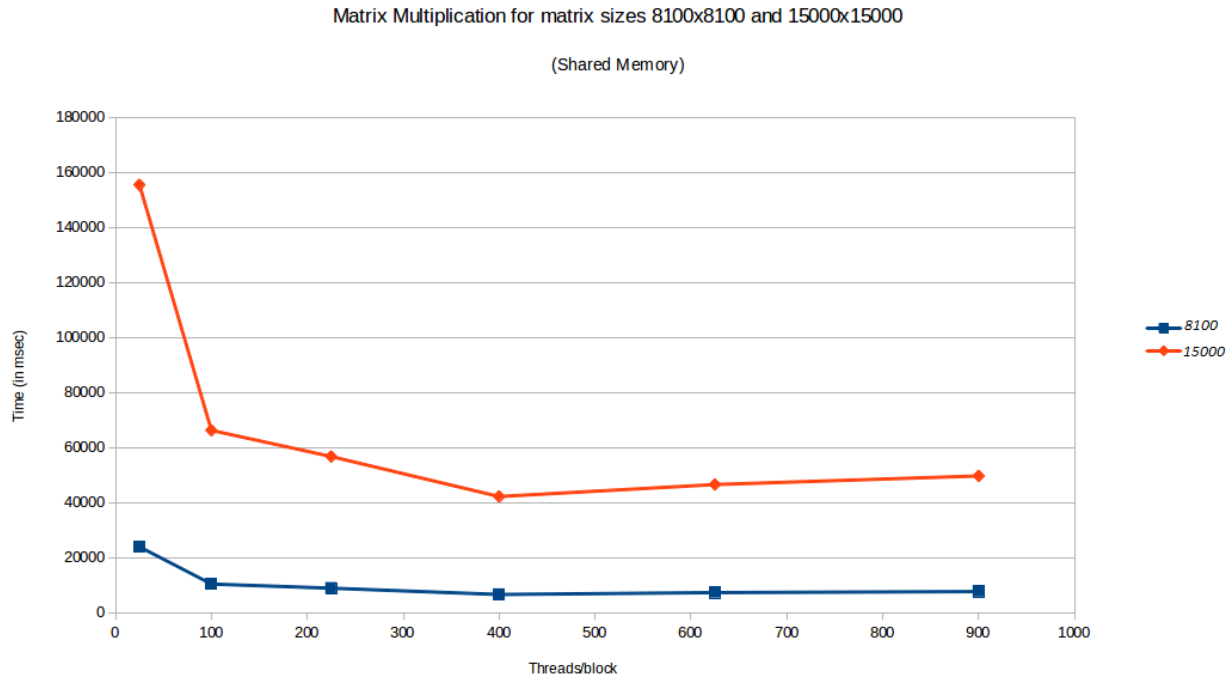
Shared Memory:

In the following table, we show the time taken for matrices of different sizes and various tile dimensions. We show in the following figure, how the matrix multiplication performs for the matrix of size 8100x8100 and various tile sizes. Note that the dimension of the tile is the same as the dimension of the block.

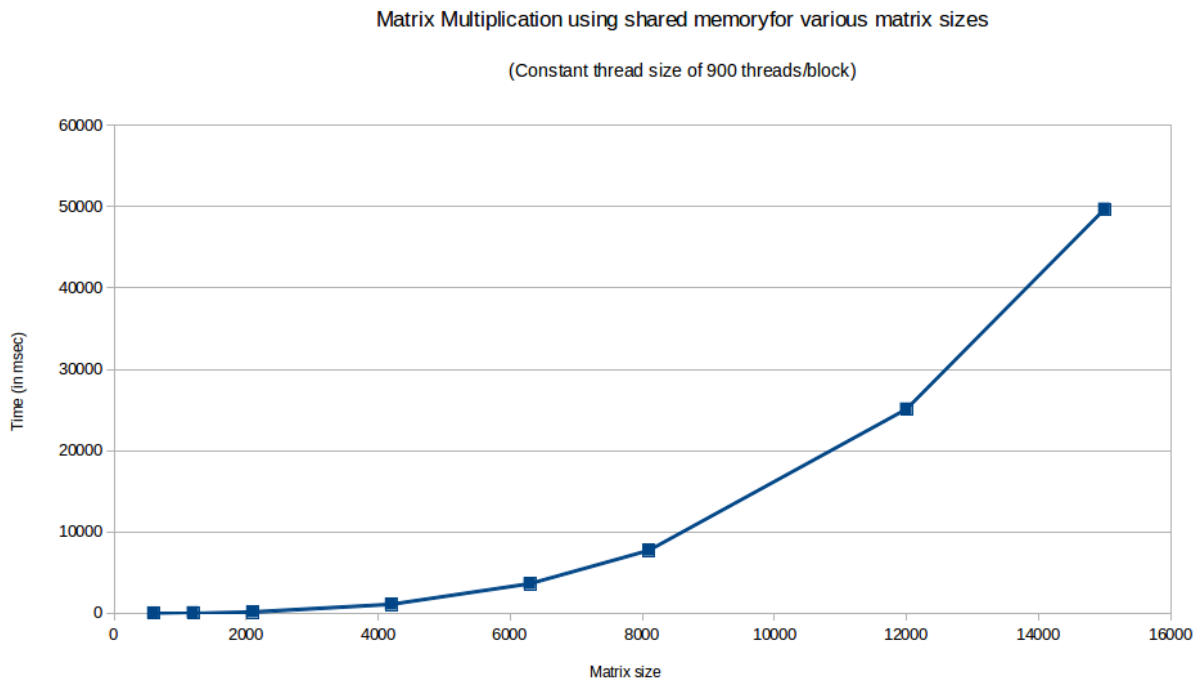
size/TPB	25	100	225	400	625	900
600	9.548864	4.309888	3.74896	2.837344	3.136032	3.289888
1200	77.64579	33.5728	29.468479	21.950945	24.279072	25.221279
2100	416.53894	179.654816	157.05687	116.776192	128.344604	134.256165
4200	3337.723389	1447.803101	1250.994019	929.835449	1026.381836	1075.181274
6300	11261.375977	4909.989258	4212.429199	3131.282471	3455.955566	3615.382324
8100	23963.357422	10450.112305	8936.963867	6656.250488	7343.584961	7712.338867
12000	80000.59375	34021.820312	29065.449219	21645.8125	24102.150391	25097.056641
15000	155392.046875	66257.640625	56739.980469	42228.679688	46569.238281	49706.792969



In the following plot, we compare the time taken to multiply matrices of sizes 8100x8100 and 15000x15000 using shared memory. We see that the time taken to multiply the matrix of larger size is larger (as expected).



Next, we show how the time taken increases when the tile size is kept constant and the matrix size is varied.

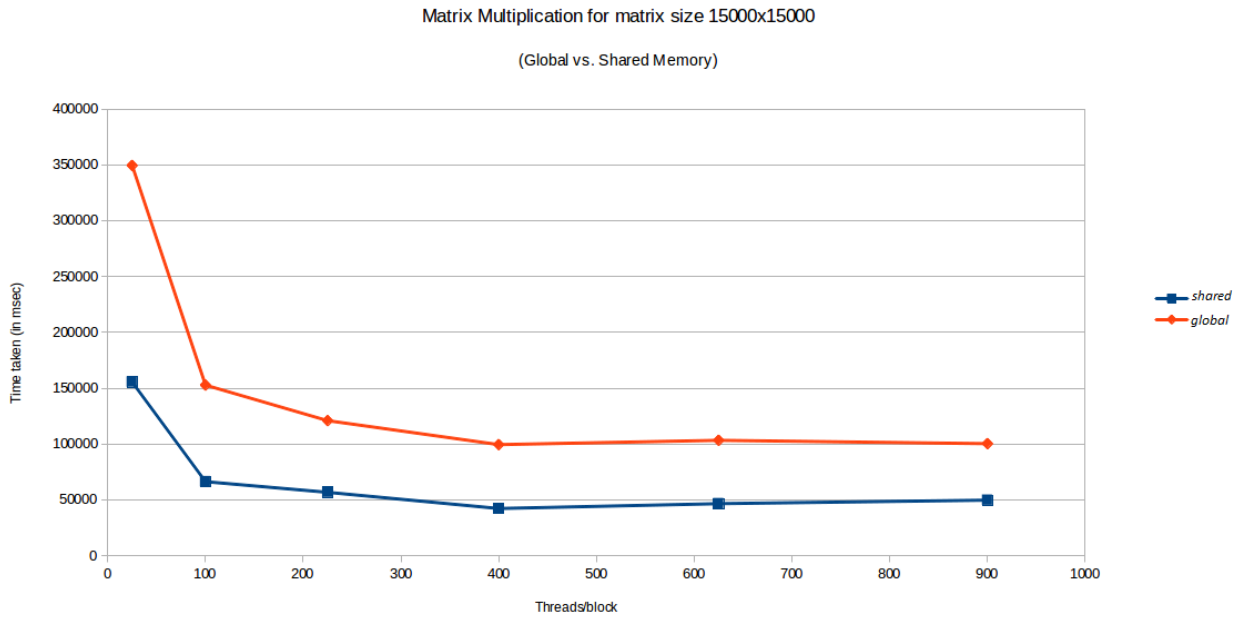


Comparison of Matrix Multiplication using Global Memory vs. Shared Memory:

For simplicity, we ran the matrix multiplication code for the matrix sizes as in the table for shared memory. The results are given below:

size/TPB	25	100	225	400	625	900
600	21.97648	9.26672	7.780832	6.47296	6.648064	6.601312
1200	175.96019	72.799133	60.260414	50.799553	51.977249	50.70512
2100	944.473389	391.046265	321.380096	271.015198	276.27713	269.305878
4200	7568.222168	3166.142822	2575.622803	2167.076904	2204.481934	2146.421143
6300	25658.503906	10783.952148	8738.320312	7312.354492	7466.025879	7232.29248
8100	54776.726562	23208.240234	18652.914062	15559.789062	16027.863281	15318.920898
12000	178347	77693.828125	62617.804688	51418.742188	52870.296875	50916.371094
15000	349340.21875	152719.484375	120884.320312	99430.195312	103294.75	100312.570312

We plot and compare the time taken to multiply two matrices using global memory and shared memory. We see that using shared memory for computing the product of two matrices is much faster than using global memory.



Below, we plot how the two versions, i.e. multiplying the matrices using global and shared memories work when we keep the number of threads/block constant, i.e. using a block size of 900x900. We see that as the size of the matrix increases, using shared memory will result in a lower execution time than using global memory.