CA Assignment 2

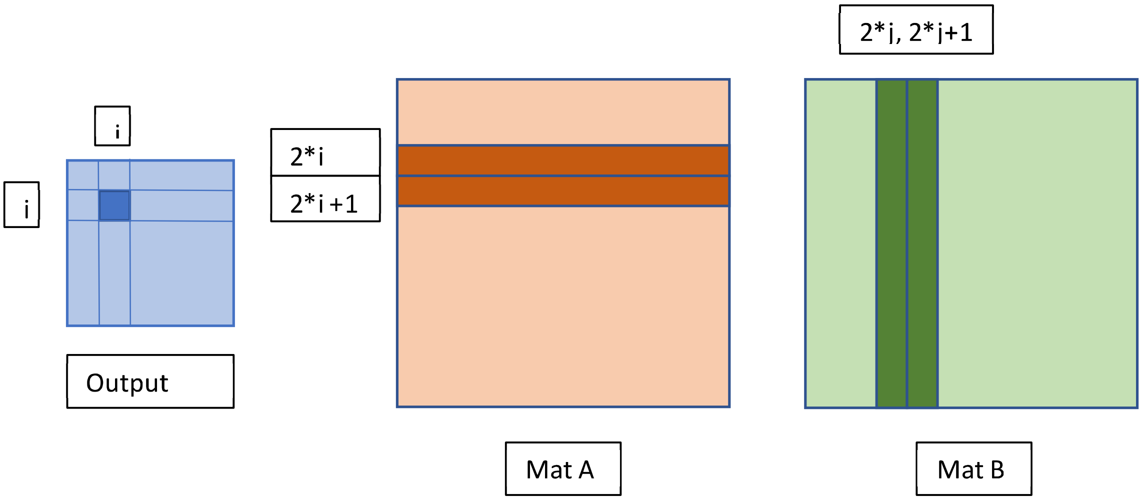
Name: Abhradip Ghosh
Link - https://github.com/abhradipg/CAassign2/

In reduced matrix multiplication we multiply two matrices of size N*N and produce a matrix of size (N/2) * (N/2), here N=2*k; k>=2.

Naive implementation of RMM is given as -

```
for(int rowA = 0; rowA < N; rowA +=2) {
   for(int colB = 0; colB < N; colB += 2) {
     int sum = 0;
     for(int iter = 0; iter < N; iter++) {
        sum += matA[rowA * N + iter] * matB[iter * N + colB];
        sum += matA[(rowA+1) * N + iter] * matB[iter * N + colB];
        sum += matA[rowA * N + iter] * matB[iter * N + (colB+1)];
        sum += matA[(rowA+1) * N + iter] * matB[iter * N + (colB+1)];
     }
     // compute output indices
     int rowC = rowA>>1;
     int colC = colB>>1;
     int indexC = rowC * (N>>1) + colC;
     output[indexC] = sum;
   }
}
```

In this implementation the innermost loop computes the value of an element of output matrix [i,j] by traversing rows 2*i and 2*i+1 of matrix A and traversing columns 2*j and 2*j+1 of matrix B.



We are traversing the Matrix B in column wise manner in long strides, which is not cache friendly.

Reference Execution Time -

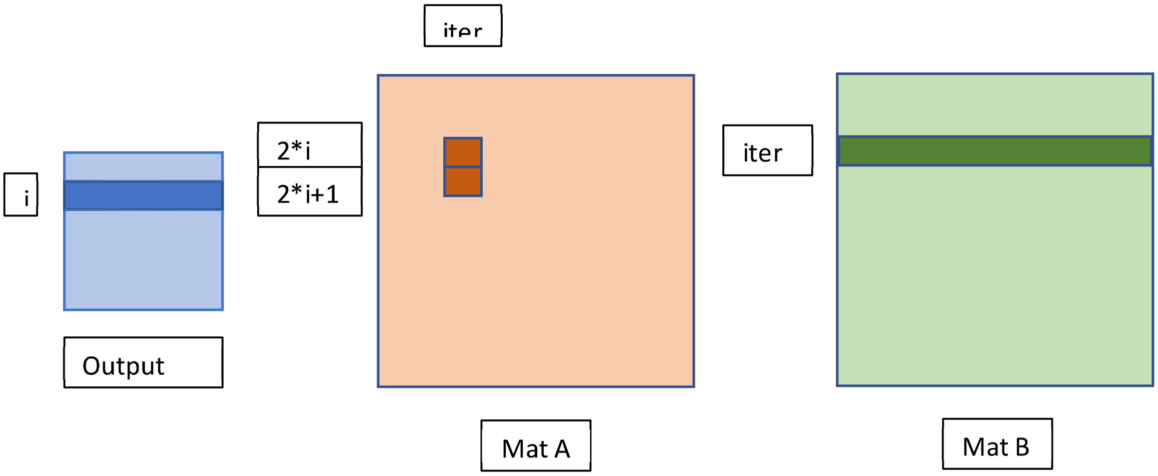
Matrix Size	16*16	1024*1024	4096*4096	8192*8192	16384*16384
Execution Time (mS)	0.028	2911.35	533015	5.85628e+06	5.09691e+07

Part A -

Optimization 1-

We can change the order of our loop, let outermost loop traverse on rowA, innerloop traverse on iter and our innermost loop traverse on colC.

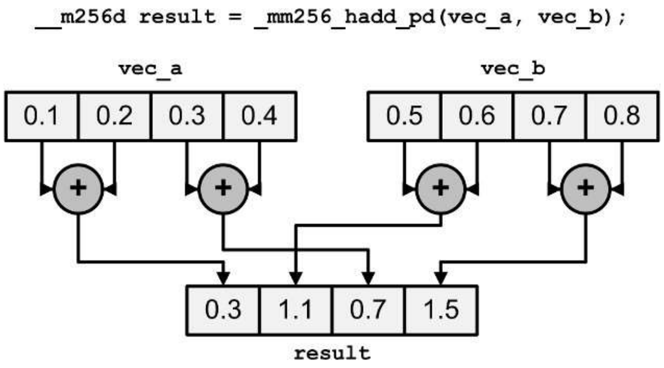
In this approach the innermost loop calculates partial result of row i of output matrix by using one element of MatA and traversing over whole row of MatB.



Since we are accessing all the matrices in which they are stores in memory, we can use AVX instructions to compute 8 elements of output matrix in one iteration, by loading two 8 element long data from MatB and two elements of MatA and using this to compute partial result of 8 elements of output matrix.

We need to first initialize output matrix with 0. Then we can compute output matrix.

```
for(int rowA = 0, rowC = 0; rowA < N; rowA+=2,rowC+=noOfRowC){
    rowFirstIndex=rowA * N;
    rowSecondIndex=(rowA+1) * N;
    for(int iter = 0; iter < N; iter++){
        rowFirst = _mm256_set1_epi32(matA[rowFirstIndex + iter]);
        rowSecond = _mm256_set1_epi32(matA[rowSecondIndex + iter]);
        rowBIndex=iter * N;
        for(int colB = jj, colC = jj>>1; colB < jj+32; colB+=16, colC+=8){
            indexC=rowC + colC;
            colFirst = _mm256_loadu_si256((__m256i *)&matB[rowBIndex + colB]);
            colSecond = _mm256_loadu_si256((__m256i *)&matB[rowBIndex + (colB+8)]);
            sum1 = _mm256_add_epi32(_mm256_mullo_epi32(rowFirst,colFirst),
                       _mm256_mullo_epi32(rowSecond,colFirst));
            sum2 = _mm256_add_epi32(_mm256_mullo_epi32(rowFirst,colSecond),
                    _mm256_mullo_epi32(rowSecond,colSecond));
            sumTotal = _mm256_hadd_epi32(sum1,sum2);
            _mm256_storeu_si256((__m256i *)&output[indexC],
            _mm256_add_epi32(_mm256_loadu_si256((__m256i *)&output[indexC]),
                                                                sumTotal));
            }
      }
}
```



```
__m256d result = _mm256_hadd_pd(vec_a, vec_b);
```

Hadd of two vectors

Since we are using hadd instruction to combine 2 vector to get 8 elements of output we need to permute the elements in output matrix to get correct results.

```
__m256i permuteVector=_mm256_setr_epi32(0,1,4,5,2,3,6,7);

 for(indexC=0;indexC<sizeC;indexC+=8){
    _mm256_storeu_si256((__m256i *)&output[indexC],_mm256_permutevar8x32_epi32
                        (_mm256_loadu_si256((__m256i *)&output[indexC]), permuteVector));
}
```

Execution Time after using AVX instruction -

Matrix Size	16*16	1024*1024	4096*4096	8192*8192	16384*16384
Execution Time (mS)	0.009	562.482	36929.7	295188	2.40012e+06
Speedup	3.112	5.17	14.433	19.839	21.236

Here, we are using matrix of size N*N to compute output matrix.

We can add two consecutive columns of B to get a matrix of size N*N/2 and similarly we can add two consecutive rows of A to operate on smaller matrix of size N/2*N.

Optimization 2-

Execution Time operating on smaller matrix (only A) including time to compute smaller matrices-

Matrix Size	16*16	1024*1024	4096*4096	8192*8192	16384*16384
Execution Time (mS)	0.006	369.432	24211.8	194634	1.56757e+06
Speedup	4.667	7.88	22.014	30.088	32.514

Optimization 3-

Execution Time operating on smaller matrices (both A and B) including time to compute smaller matrices-

Matrix Size	16*16	1024*1024	4096*4096	8192*8192	16384*16384
Execution Time (mS)	0.005	250.275	16442.2	132253	1.0656e+06
Speedup	5.6	11.632	32.417	44.28	47.831

Optimization 4-

Next we can try unrolling the loop, by unrolling the iter loop we can work on different rows of matB in single iteration of innermost loop so that we can reuse data by storing it in registers as much as possible, so that it does not have to be repeatedly loaded from the caches. In addition, we are accumulating results in registers as long as possible, so that it reduces the times of storing data into the cache memory.

We are loading 8 elements of matA at once using AVX instruction and using this to compute output matrix value.

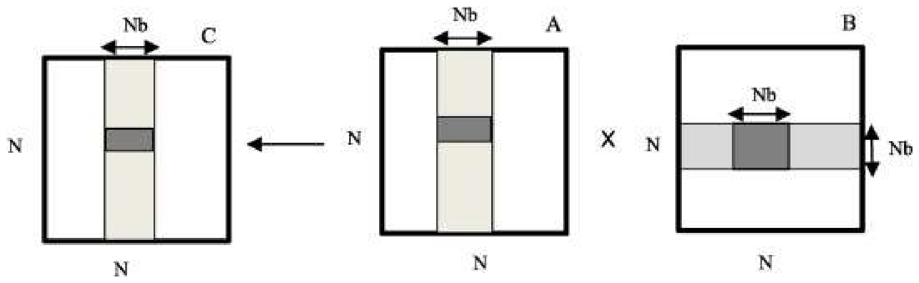
```
for(int colB = 0, indexC = rowC; colB < noOfRowC; colB+=8, indexC+=8)
{
    colC=_mm256_loadu_si256((__m256i *)&output[indexC]);

    rowB=rowBIndex;
     colFirst1 = _mm256_loadu_si256((__m256i *)&Bnew[rowB + colB]);
     sumTotal1 = _mm256_mullo_epi32(row1,colFirst1);
     colC=_mm256_add_epi32(colC,sumTotal1);

      rowB+=noOfRowC;
     colFirst2 = _mm256_loadu_si256((__m256i *)&Bnew[rowB + colB]);
     sumTotal2 = _mm256_mullo_epi32(row1,colFirst2);
     colC=_mm256_add_epi32(colC,sumTotal2);
     /*
            Do this 5 more times                              */

     rowB+=noOfRowC;
     colFirst2 = _mm256_loadu_si256((__m256i *)&Bnew[rowB + colB]);
     sumTotal2 = _mm256_mullo_epi32(row1,colFirst2);
     _mm256_storeu_si256((__m256i *)&output[indexC],_mm256_add_epi32(colC,sumTotal8));
}
```

Matrix Size	16*16	1024*1024	4096*4096	8192*8192	16384*16384
Execution Time (mS)	0.004	180.356	11817.2	91950.3	785192
Speedup	7	16.142	45.105	63.689	64.912

Optimization 5 (Maybe ?) -

Next, we can use tiling to make better use of memory hierarchy by working on tile which fits in cache working on it and then move to next tile.

We are using tile of size 16*16 of MatB and using it to compute value of whole column of width 16 of output matrix using column of A of width 16. Use we are using block of 16*16 of MatB and then then never use it again.



```
for(int kk = 0;kk < N; kk+=16){
      for(int jj = 0;jj < noOfRowC; jj+=16 ){
           for(int rowA = 0, rowC = 0; rowA < N; rowA+=2,rowC+=noOfRowC){
               /*
                   Calculate row index of Mat a
                                            */
                 for(int iter = kk; iter < kk+16; iter +=8){
                     /*
                          Fetching  8 columns of matA from computed index
                                                  */
                       for(int colB = jj, col = jj; colB < jj+16 ; colB+=8, col+=8) {
                            /*
                                  Computethe values of Output Matrix
                                                  */
                       }
                   }
                }}}
```

Using tilting we should get better performance but in my implementation its performance slightly worse than performance in optimization 3 (writing report on last day (lazy to use perf), so don't want to find why ?).

Matrix Size	16*16	1024*1024	4096*4096	8192*8192	16384*16384
Execution Time (mS)	0.003	216.473	22671.5	191547	1.69764e+06
Speedup	9.33	13.449	23.51	30.573	30.022

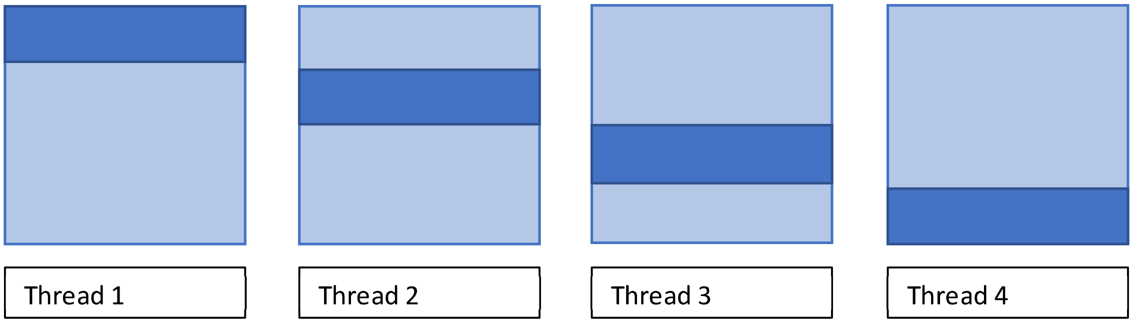
There might be other implementation of tiling which could give better performance.

Optimization 6-

We could use _mm_prefetch() instruction to prefetch the values that we will use in tile to get better execution time. Since I am writing report on last day don't have time to implement it (future work).

Multi-Threading -

For multi thread execution we will break the output matrix into group of into disjoint sets of rows and give each set to different threads, so each thread can compute independently.



We are running the code in 12 – Logical core CPU, so we will get least execution time for 12 threads. Resulting execution times confirms this.

Execution time and comparison with naïve AVX single thread implementation -

#Threads	Matrix Size	16*16	1024*1024	4096*4096	8192*8192	16384*16384
8	Execution Time (mS)	0.514	131.608	6733.81	54040.5	437312
	Speedup	<1	4.27	5.484	5.463	5.488
12	Execution Time (mS)		94.474	5665.9	45692.8	387698
	Speedup		5.953	6.517	6.460	6.190
16	Execution Time (mS)		106.357	5754.64	47290.9	399737
24	Execution Time (mS)		93.625	5760.37	47763.8	392799

Execution time and comparison with smaller matrix (only A) single thread implementation -

#Threads	Matrix Size	16*16	1024*1024	4096*4096	8192*8192	16384*16384
8	Execution Time (mS)	0.474	86.547	4636.17	36816.4	297941
	Speedup	<1	4.268	5.22	5.286	5.261
12	Execution Time (mS)		63.097	3902.1	30256.5	245417
	Speedup		5.854	6.204	6.432	6.387
16	Execution Time (mS)		70.533	4170.21	42048.9	374190
24	Execution Time (mS)		63.134	4290.61	38948.9	367352

Execution time and comparison with smaller matrices (both A and B) single thread implementation -

#Threads	Matrix Size	16*16	1024*1024	4096*4096	8192*8192	16384*16384
8	Execution Time (mS)	0.62	59.685	3040.37	26836.1	201361
	Speedup	<1	4.193	5.407	5.051	5.291
12	Execution Time (mS)		43.081	2531.4	19696.4	163402
	Speedup		5.809	6.495	6.882	6.521
16	Execution Time (mS)		51.333	3111.96	36256.1	370185
24	Execution Time (mS)		49.146	3562.59	36014.9	366321

Execution time and comparison with unrolled single thread implementation -

#Threads	Matrix Size	16*16	1024*1024	4096*4096	8192*8192	16384*16384
8	Execution Time (mS)	0.656	56.897	3013.2	24047	195347
	Speedup	<1	3.169	3.921	3.823	4.019
12	Execution Time (mS)		40.825	2480.71	20901.5	163728
	Speedup		4.417	4.763	4.399	4.795
16	Execution Time (mS)		46.174	2568.44	22735.4	202566
24	Execution Time (mS)		46.431	2495.8	22083.4	201403

Conclusion we found that unrolled version of implementation will have best performance and thread of size 12 is idle.

Part B -

For GPU we will make one thread for each element of output matrix. We make block of size 32*32 as 32 threads will form a warp and will execute in lock step manner.

So, if we make block of size 16*16, the warp will work on 16 column of row i and 16 column of row i+1 , which are not in consecutive location.

Thus, by using block size of 32*32 all 32 threads of a warp will have consecutive memory location so GPU can benefit from coalesced memory accesses.

Code for all implementation is present in gpu_thread.h file (commented in end).

Execution time (just Kernel) for GPU (Naïve implementation) -

Matrix Size	128*128	1024*1024	4096*4096	8192*8192	16384*16384
Execution Time (nS)	30,592	5,72,770	3,09,63,446	21,76,15,467	3,19,34,48,117

Optimization 1 -

We can use int2 and int4 data type to load multiple elements at a time and unroll the loop 4 times to compute on loaded data.

Execution time (just Kernel) for GPU (Unrolled implementation) and speedup compared to naïve implementation -

Matrix Size	128*128	1024*1024	4096*4096	8192*8192	16384*16384
Execution Time (nS)	22,401	3,32,323	2,54,40,482	18,66,07,282	1,96,86,15,325
Speedup	1.36	1.72	1.21	1.66	1.62

Optimization 2 -

We can use shared memory to store a block of data and compute on that then we can move to next block. Since Shared memory is in L1 cache the access to it will be faster than accessing memory.

We use shared memory of sizes A[32][64] and B[64][32], each thread in thread block will load their corresponding element into shared memory and iterate on it.

Execution time (just Kernel) for GPU (Shared memory implementation) and speedup compared to naïve implementation -

Matrix Size	128*128	1024*1024	4096*4096	8192*8192	16384*16384
Execution Time (nS)	13,760	1,85,057	87,90,084	7,03,80,284	64,90,24,485
Speedup	2.22	3.09	3.52	3.09	4.92