Hardware :-

Tesla M2070.
Compute capability – 2.0
Shared Memory per block – 48 KB
SMs – 14 Registers per SM - 32K

Problem 1:-

The naïve implementation is to use the global memory and trying to distribute one thread per column of computation of B. But there is extensive data reuse in the computation of B. All the computation of given row of B require the same row of A. Hence, we try to copy a row into shared memory to exploit the data reuse and the effective shared memory which offers us high bandwidths. We observe a increase in performance.

Tried to analyze the performance was constrained by the barrier synchronization. Eliminated a barrier by allocating more shared memory. Barrier was eliminated but the performance degraded. This is because of unavailable warp parallelism. Only 3 blocks can reside at a time because 16KB of memory is required to store one row of A and shared memory available per block is 48KB. Double the sharing memory to eliminate the barrier, allowed only 1 block to reside in an SM.

Version	**Work Distribution**	**Block/Grid dimensions**	**Performance**
Global Memory	One thread per column.	(128,1) and (16,1)	7.4 GF
Shared-A	One thread per column.	(2048/64, 1) (64,1)	11.3 GF
Shared-AB	One thread per column. Tile size- (8,64)	(N/64,1) (64,1)	19.2 GF
Shared-AB. Separating triangular and rectangular loops	One thread per column. Tile size- (8,64)	(N/64,1) (64,1)	25.7 GF
Shared-AB- 2D block	One thread block per a tile-column block of B.	(N/32,1) – (32,32)	78.6 GF

Computation of successive elements along a column require common data. Hence, storing B into shared memory also would help us with the bandwidth issues. We consider a tile of B as the work-load for a thread-block.

We launch 1-dimensional horizontal thread-blocks of shape (n,1). Each thread iteratively brings in all the elements of the tile it needs to write to. In consecutive iterations, all the threads of the block bring tiled-rows of A and tiled-block of B and multiply them. We do observe a increase in performance than the earlier version but the increase is minor. After some initial profiling, we observe that the achieved occupancy is low. We observe separating triangular and rectangular loops also improves performance. This improve might be because the compiler is now able to apply some of the optimizations.

To improve occupancy, we launch more warps as 2D thread blocks of the shape (n,n). All the threads collectively bring the block of computation it writes to. (Note that here we are hiding the dependencies because we copy the block of B to the shared memory location). We iteratively bring tiled blocks of A and B, compute the result. We observe an increased achieved occupancy of 5 warps/cycle on an average. We observe tremendous increase in performance. This can be attributed to the improved occupancy and the warp-level parallelism.

We also measure performance excluding the time taken to copy memory from host to device which is **86.6 GF** for the best kernel.

Programming Optimizations:-

We use “short int” data type to store loop iterator because we know the no. of rows < 65536. This helps us save some registers and allows more warps to reside at a time in an SM and avoids register spilling into local memory. We let the compiler take the decision to unroll by adding the #pragma unroll. We observe no bank conflicts and command line profiler also gives us this info and the global memory accesses are coalesced.

Best Kernel :-

```
#define n 32
   short int it = blockDim.y * by;  short int jt = blockDim.x * bx, kt = it,k;
   __shared__ double a[n][n], b[n][n];

  for( ;it < N; it+= n*gridDim.y)
  {  //  kt = it  // Load Asub and Bsub to the shared memory
     bsum = 0;
     kt = it;
       a[ty][tx] = A[(it + ty)*N + kt + tx];
       b[ty][tx] = B[(kt + ty)*N + jt + tx];
       __syncthreads();
       // Compute the result
       for( k = ty ; k < n ; k++)
          bsum += a[ty][k] * b[k][tx];
       __syncthreads();

//#pragma unroll
    for( kt = it+n; kt < N; kt += n)
    {
       // Load Asub and Bsub to the shared memory
       a[ty][tx] = A[(it + ty)*N + kt + tx];
       b[ty][tx] = B[(kt + ty)*N + jt + tx];
       __syncthreads();
       // Compute the result
#pragma unroll
       for( k = 0 ; k < n ; k++)
          bsum += a[ty][k] * b[k][tx];
       __syncthreads();
    }
    // Write result to B
    B[(it+ty)*N + jt+tx ] += bsum;
  }
```

Problem 2:-

There are no dependencies in the computation. Hence, the most naïve implementation to load the computation of one element of C to one thread. 2D Grid and 2D block look natural. We use only use global memory in this version.
Half the threads have no work because we only compute the upper triangular portion of C. Please note that though it appears there are large number of warps affected by thread divergence, but there is only one warp per row that is affected by thread divergence, i.e the warp that is cut by the diagonal.

We observe there is extensive reuse of the same data by the elements along a row or column of C. Hence, data reuse can be exploited by using shared memory. As, shared memory is limited (48KB), we tile the loops such that we can fit a tile into the shared-memory.

Version	**Work Distribution**	**Block and Grid dimensions**	**Performance**
Global Memory	One thread per one element of C	(N/32,N/32) - (32,32)	8.9 GF
Tile-Shared memory	One thread per one element of C. Tile (8,8,8)	(N/8,N/8) – (8,8)	13.7 GF
Tile-Shared Mem-Opt1	Two threads per one element of C	(N/8,N/8) – (8,8)	19.9 GF
Tile-Shared Mem-Opt1,2,3	Two threads per one element of C. Tile-(16,2,32)	(N/16,N/16) – (16,16)	49.2 GF
Tile-Shared Mem-Opt 1,2,3,4	2* Two threads per one element of C	(N/16,N/16) – (16,16,2)	Upper bound - 42.0 GF

Opt1 - Optimized Work-distribution :-
We observe half the threads in the grid have no work to do. As non-rectangular grids are not allowed by cuda, we cannot avoid creating them. But, work can be shared among them, i.e two thread blocks can work on a tile of C. The loop-body in this kernel allows to divide the computation of tile into two easily.

C[k][l] = 0.5*(A[l][i][j]*B[k][i][j] + A[k][i][j]*B[l][i][j])

into

D[k][l] = 0.5*(A[l][i][j]*B[k][i][j]) and D[l][k] = 0.5*(A[k][i][j]*B[l][i][j])

C[k][l] += D[k][l] + D[l][k]

The post-process step “C[k][l] += D[k][l] + D[l][k]” is done in the host (CPU) after the GPU-kernel ends.

We are forced to use a store the partial results in a temporary location, D because of

1. If both the blocks try to update their result to a common location, atomicAdd for doubles is not supported with GPU architectures/CUDA yet. (float could be tried but precision has to be compromised)
2. If one block tried to get the partial result from its symmetrically mirrored block and update the result, inter-block communication is available in CUDA by design.

We could have some hacky ways that allows us to have inter-block synchronization. But not sure, it would run for varying block sizes/SMs. Even the future versions of compiler might remove such loops as part of optimizations, because CUDA wants programs to written independent of the number of blocks as the program should be to scale for architectures that have more SMs.

More over, it takes 0.0027 milli-secs to compute the above step. This is 0.1% of the total computation time and 0.2% of total wall clock time (including memory transfer). Hence, this is extremely minute in its resource requirements and almost negligible in its contribution to the overall time/performance.

Opt2 – Bank Conflicts

Removed bank conflicts in the kernel. We pad the stride (which is even) by 1 which maps elements accessed by consecutive threads to different blocks because the greatest common divisor of 32 (no. of banks in devices of compute capability 2.0) and any odd number is 1.

Opt3 - Global Memory coalesced accesses:-

We initially tried with tile sizes (8,8,8). Tiling along 'J' by 8 is not efficient because this request would not fill the cache-line and is not a fully coalesced access. Hence, we would transferring unused data. There is a L1-cache but cannot rely on it. Alternately, tiling 'J' by '16' creates a 128-byte request. (16 * 8) which is equal to the cache-line. Please note, we do not need a warp (32) of threads access consecutive elements because when double (64 bits) data types are accessed it is broken down to 2 global memory transactions. (cache line is 128 bytes only can only fit 16 doubles)

Opt4:-

We try to create 3-Dimensional thread blocks to improve SM occupancy. We now have all threads along z-dimension to compute one element of C. Hence, there should be reduction step or an atomicAdd step. But, we do observe a degrade in both the cases (tried atomicAdd with float compromising precision). We tried to observe whether the degrade in performance is because of the reduction or atomicAdd. We remove the step and try to write results (though incorrect) to get an upper bound on the performance. We observe it to be less than the performance observed in Opt2&3 and hence we ignore this optimization.

Thread block sizes, grid dimensions and tile sizes have not been extensively experimented with. Doing so, might give us higher Gflops.

We observe a performance of **87 GFlop** without the memory copies to and from the device. A little profiling indicates that we spent 45% of total time trying to copy data to and from the device

Best Kernel :-

```
#define n 16
#define tileSizeA 16
#define tI 2
#define tJ 16
__global__ void mul_globalMem(double *A, double *B,double *C)
{
   short int bidx = blockIdx.x, bidy = blockIdx.y;
   short int tidx = threadIdx.x, tidy = threadIdx.y;

   short int i,j,it,jt;
   short int k = bidy * blockDim.y + tidy;
   short int l = bidx * blockDim.x + tidx;
```

```
    short int l1 = bidx * blockDim.x + tidy;

    __shared__ double a[tileSizeA * (tileSizeI*tileSizeJ+1)];
    __shared__ double b[tileSizeA* ((tileSizeI*tileSizeJ)+1) ];

    double sum = 0;
    for(it = 0; it< N; it+=tileSizeI)
    {
       for(jt = 0; jt< N; jt+=tileSizeJ)
       {
// Load data into shared memory. Note (tI*tJ + 1). "+1" is padded to avoid bank conflicts.
// This is a 2xi-unroll  version.
                a[ ((tI*tJ+1)* (tidy)) + tidx ] = A[get3(k ,it,jt+tidx)];
                a[ ((tI*tJ+1)* (tidy)) + tJ+ tidx ] = A[get3(k ,it+1,jt+tidx)];

                b[ ((tI*tJ+1)* (tidy)) + tidx ] = B[get3(l1 ,it,jt+tidx)];
                b[ ((tI*tJ+1)* (tidy)) + tJ+ tidx ] = B[get3(l1 ,it+1,jt+tidx)];

          __syncthreads();
             for (i = 0; i < tileSizeI; i++)
                for (j = 0; j < tileSizeJ; j++)
                   sum += (a[ ((tI*tJ +1)* tidy) + (tJ*i) + j]*b[ ((tI*tJ+1) * tidx) + (tJ*i) + j]) ;
          __syncthreads();
       }
    }
    C[get2(k,l)] = 0.5*(sum);
}
```

End Remarks :-

Tried to avoid non-coalesced global memory accesses and bank conflicts. Tried to exploit the warp parallelism to hide memory latency. As the data elements we are dealing with are "double" data typed (8 bytes per element), it is costly to store them. Using float, would definitely give us higher GFlops because we can achieve more warp parallelism (because of reduced shared memory requirements, occupancy increases) or double the data could be fit into shared memory (bigger tiles).