Problem 1:

Solution: Part 1:

Let us understand the computation environment and the computation to be performed:

We have one grid a containing one block of 32* 32 threads

So, we have 1024 threads in a block that will be implemented as a warp of 32 threads with 32 warps creating 1 Block. Then we are considering the fact that input array LHS and RHS which are of dimension 32 * 128 and 128*32 respectively will be brought into the shared block memory for ease access of the elements by the threads. We have the output 32 * 32-dimension result matrix where each coordinate of the matrix will be computed by each thread. Hence, we have to visualize that these LHS and RHS input matrices are stored in the column major order.

LHS - 32 * 128 - Col1 32 elements à Col2 32 elements à Col3 32 elements .....till column Col128

RHS – 128 * 32 – Col1 128 elements à Col2 128 elements à Col3 128 elements .....till column Col32

So the above representation shows the representation of the LHS matrix and RHS matrix in the shared memory in the column major order. Now as we saw earlier also in the refresher and the lecture also that we have the data arranged in the 32 Banks with I % 32 gives the bank of the address of the element.

First 32 elements-

0 -> 1 -> 2 ->.....31

Second 32 elements-

Arranged in the same bank as shown above

0 -> 1 -> 2 -> 31

So, this way our elements of the above shown column major based linearly stored 2 D matrices in the shared memory. The purpose of the Bank is that as our warp constitutes of 32 threads working for 32 output elements running the kernel code as given above:

```
int i = threadIdx.x;
int j = threadIdx.y;
for (int k = 0; k < 128; k += 2) {
        output[i + 32 * j] += lhs[i + 32 * k] * rhs[k + 128 * j];
        output[i + 32 * j] += lhs[i + 32 * (k + 1)] * rhs[(k + 1) + 128 * j];
}
```

Here, Thread denoted by x,y of the 2D Block
Now the point is that 128 elements are accessed by each thread from LHS as well as RHS.

Based on the arrangement of the LHS and RHS (Linear Column Based) mapped onto the Bank based system of the shared memory we will see that in case of LHS:

32 elements belonging to 32 banks
32 elements belonging to 32 banks
.
.
.
.
.
128 times

So here we have each of the 32 threads accessing the different Banks so a whole warp can access all memory locations in one clock cycle because each thread is getting the element from different banks.

But in case of RHS:
Same as above arrangement will occur but one column of 128 elements get divided into the Banks in below arrangement:

32 elements belonging to 32 banks
.
32 elements belonging to 32 banks
.
32 elements belonging to 32 banks
.
32 elements belonging to 32 banks

Now in this case if 32 threads in the warp access the elements for running the kernel and outputting their corresponding element. Every other thread will be accessing the same bank for first , second .. so on , elements access .

Hence here will be the bank conflict and hence each thread has to access the element one by one in each clock cycle. So, it seems 32 cycles per warp to get each thread access element.

Solution Part 2a:

In order to understand the solution , we need to see what this sequential is doing.

We have 1024 * 1024 size **A array** which is an input array and 1024 size **B array** which is an output array.

Let us understand the working of this code:

B0----------B1024

********** (consider represent the 1024 elements row)
****
****
****
.
.
.
***
**
*

This is the row wise access pattern of the i , j

And what we are doing—

Adding the elements represented by an arrow are being added and put to the corresponding output element.

So Here we can see that we can create the partition as :

Divide the 1023 output elements as threads where the input to each thread will be the diagonal elements as shown. Here the kernel assigned to each thread will take the sum of the diagonal elements as shown and hence put them into the corresponding output.

It is to be noted that we are accessing the input and the output elements form the global space

Part 2b:

Memory coalescing of global space means that when the consecutive elements are accessed utilizing the bandwidth of the global memory access. But in our case as we can see that we are accessing the diagonal elements so the input memory access pattern has a stride but in case of output array access pattern we can write consecutively.

Sample Kernel Pseudo code can be like this :
i= thread_id.x
row =0 , col = i
while col > 0:
b[i]+=a[row][col]
row+=1
col-=1

Problem 2
Solution Part 1:

1. ***Global Memory***

```
#include <stdio.h>
#include <math.h>
#define THREADS_PER_BLOCK 256

__global__ void MatrixMul( float *Md , float *Nd , float *Pd , const int WIDTH )
{
int COL = threadIdx.x + blockIdx.x * blockDim.x;
int ROW = threadIdx.y + blockIdx.y * blockDim.y;

 if (ROW < WIDTH && COL < WIDTH) {
     for (int i = 0; i < WIDTH; i++) {
        Pd[ROW * WIDTH + COL] += Md[ROW * WIDTH + i] * Nd [i * WIDTH + COL];
      }
  }
}

int main(int arg0, char *arg1[]){
 cudaThreadSynchronize();
 int WIDTH = atoi(arg1[1]);
 int sqrtThreads = sqrt(THREADS_PER_BLOCK);
 int nBlocks = WIDTH/sqrtThreads;

 if (WIDTH % sqrtThreads != 0) {
          nBlocks++;
  }

 dim3 grid(nBlocks, nBlocks, 1);
 dim3 block(sqrtThreads, sqrtThreads, 1);
 float *a_h, *b_h, *c_h, *d_h, *a_d, *b_d, *c_d;
```

```
int size;

cudaEvent_t start;
cudaEvent_t stop;
float elapsed1;

size = WIDTH * WIDTH * sizeof(float);

a_h = (float*) malloc(size);
b_h = (float*) malloc(size);
c_h = (float*) malloc(size);
d_h = (float*) malloc(size);

for (int i = 0; i < WIDTH; i++){
        for (int j = 0; j < WIDTH; j++){
                a_h[i * WIDTH + j] = i;
                b_h[i * WIDTH + j] = i;
        }
}

cudaMalloc((void**)&a_d, size);
cudaMalloc((void**)&b_d, size);
cudaMalloc((void**)&c_d, size);

cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
cudaMemcpy(b_d, b_h, size, cudaMemcpyHostToDevice);
cudaMemcpy(c_d, c_h, size, cudaMemcpyHostToDevice);

cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);

MatrixMul<<<grid, block>>>(a_d, b_d, c_d, WIDTH);

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsed1, start, stop);

printf("%f\n", elapsed1/1000);
cudaMemcpy(c_h, c_d, size, cudaMemcpyDeviceToHost);
```

```
 free(a_h);
 free(b_h);
 free(c_h);
 free(d_h);
 cudaFree(a_d);
 cudaFree(b_d);
 cudaFree(c_d);

 cudaEventDestroy(start);
 cudaEventDestroy(stop);

 return 0;
}
```

2. Shared Memory

```
#include <stdio.h>
#include <math.h>
#define THREADS_PER_BLOCK 64
const int TILE_WIDTH = 8;

__global__ void multi(float *a, float *b, float *c, int width) {
__shared__ float s_a[TILE_WIDTH][TILE_WIDTH];
__shared__ float s_b[TILE_WIDTH][TILE_WIDTH];

int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
float result = 0;

for (int p = 0; p < width/TILE_WIDTH; p++){
        s_a[threadIdx.y][threadIdx.x] = a[row*width + (p*TILE_WIDTH + threadIdx.x)];
        s_b[threadIdx.y][threadIdx.x] = b[(p*TILE_WIDTH + threadIdx.y)*width + col];

        __syncthreads();

        for (int i = 0; i < TILE_WIDTH; i++){
                result += s_a[threadIdx.y][i] * s_b[i][threadIdx.x];
        }
        __syncthreads();
}
 c[row * width + col] = result;
}
```

```
int main(int arg0, char **arg1) {
  cudaThreadSynchronize();

  int width = atoi(arg1[1]);

  int sqrtThreads = sqrt(THREADS_PER_BLOCK);
  int nBlocks = width/sqrtThreads;
  if (width % sqrtThreads != 0){
          nBlocks++;
  }

  dim3 grid(nBlocks, nBlocks, 1);
  dim3 block(sqrtThreads, sqrtThreads, 1);

  float *a_h, *b_h, *c_h, *d_h, *a_d, *b_d, *c_d;
  int size;

  cudaEvent_t start;
  cudaEvent_t stop;
  float elapsed1;

  size = width * width * sizeof(float);

  a_h = (float*) malloc(size);
  b_h = (float*) malloc(size);
  c_h = (float*) malloc(size);
  d_h = (float*) malloc(size);

  for (int i = 0; i < width; i++){
          for (int j = 0; j < width; j++){
                  a_h[i * width + j] = i;
                  b_h[i * width + j] = i;
          }
}


  cudaMalloc((void**)&a_d, size);
  cudaMalloc((void**)&b_d, size);
  cudaMalloc((void**)&c_d, size);

  cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, b_h, size, cudaMemcpyHostToDevice);
```

```
 cudaMemcpy(c_d, c_h, size, cudaMemcpyHostToDevice);

 cudaEventCreate(&start);
 cudaEventCreate(&stop);
 cudaEventRecord(start, 0);

 multi<<<grid, block>>>(a_d, b_d, c_d, width);
 cudaDeviceSynchronize();

 cudaEventRecord(stop, 0);
 cudaEventSynchronize(stop);
 cudaEventElapsedTime(&elapsed1, start, stop);

 printf("%f\n", elapsed1/1000);

 cudaMemcpy(c_h, c_d, size, cudaMemcpyDeviceToHost);

 free(a_h);
 free(b_h);
 free(c_h);
 free(d_h);
 cudaFree(a_d);
 cudaFree(b_d);
 cudaFree(c_d);

 cudaEventDestroy(start);
 cudaEventDestroy(stop);

 return 0;
}
```

Solution Part 2. Part 1:

Below is the new kernel program. We have generalised the program to support any dimensions of M and N :

```
#define BLOCK_DIM_X 16
#define BLOCK_DIM_Y 16
#define TILE_DIM ((BLOCK_DIM_X+BLOCK_DIM_Y)/2)

__global__ outer_product_kernel (float* u, float* v, float* A, unsigned int M, unsigned int N){
        /* perform the outer product of u and v
        * u is of size M,             v is of size N,     A is of size M x N
        */
        // Declaring the number of rows and cols for each u and v vector
        unsigned int uRows = M;
        unsigned int uCols = 1;
        unsigned int vRows = 1;
        unsigned int vCols = N;

        unsigned int row = blockIdx.y * TILE_DIM + threadIdx.y;
        unsigned int col = blockIdx.x * TILE_DIM + threadIdx.x;
        float aValue = 0;                                                   // Local variable

        // Shared memory declaration
          __shared__ float uS[TILE_DIM][1];
          __shared__ float vS[1][TILE_DIM];

          for (int k = 0; k < (TILE_DIM + uCols - 1)/TILE_DIM; k++) {

            if (k*TILE_DIM + threadIdx.x < uCols && row < uRows)
              uS[threadIdx.y][1] = u[row*uCols + k*TILE_DIM + threadIdx.x];
            else
              uS[threadIdx.y][1] = 0.0;

            if (k*TILE_DIM + threadIdx.y < vRows && col < vCols)
              vS[1][threadIdx.x] = v[(k*TILE_DIM + threadIdx.y)*vCols + col];
            else
              vS[1][threadIdx.x] = 0.0;

            __syncthreads();

            for (int n = 0; n < TILE_DIM; ++n)
              aValue += uS[threadIdx.y][n] * vS[n][threadIdx.x];

            __syncthreads();
          }
```

```
        if (row < M && col < N)
            A[((blockIdx.y * blockDim.y + threadIdx.y)*N) +
              (blockIdx.x * blockDim.x)+ threadIdx.x] = aValue;
}


void outer_product (float* u, float* v, float* A, unsigned int M, unsigned int N)
{
    dim3 blockDim (BLOCK_DIM_X, BLOCK_DIM_Y, 1);
    dim3 gridDim ((N-1)/BLOCK_DIM_X + 1, (M-1)/BLOCK_DIM_X + 1, 1);
        outer_product_kernel <<< gridDim, blockDim >>> (u, v, A, M, N);
}
```

Part 2. Part 2:

1. In the original version of the code, each element of u is loaded N times because we have to multiply the element from each row of u to element from each column of v as u is of size Mx1 and v is of size 1xN.
2. In the tiled version, each element of u needs to be loaded only N/TILE_DIM times because we have used shared memory and hence we are loading elements from vector u of size TILE_DIM size. Hence, each element from the tile of u will be multiplied by each element from the tile of v and since the number of tiles from vector v can be N/TILE_DIM, we will have to load each element from u, N/TILE_DIM tiles.

Part 2. Part 3:

1. In the original version of the code, each element of v is loaded M times because we have to multiply the element from each column of v to element from each row of u as u is of size Mx1 and v is of size 1xN.
2. In the tiled version, each element of v needs to be loaded only M/TILE_DIM times because we have used shared memory and hence we are loading elements from vector v of size TILE_DIM size. Hence, each element from the tile of v will be multiplied by each element from the tile of u and since the number of tiles from vector u can be M/TILE_DIM, we will have to load each element from v, N/TILE_DIM tiles.