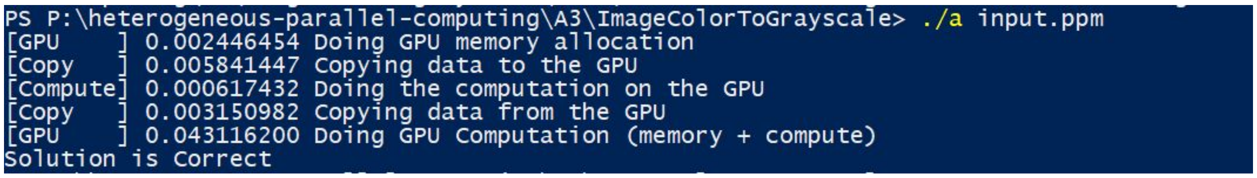
Report - A3

Dibyadarshan Hota (16CO154)
Omkar Prabhu (16CO233)

Q1)

Output:
Checked using serial code computation of output

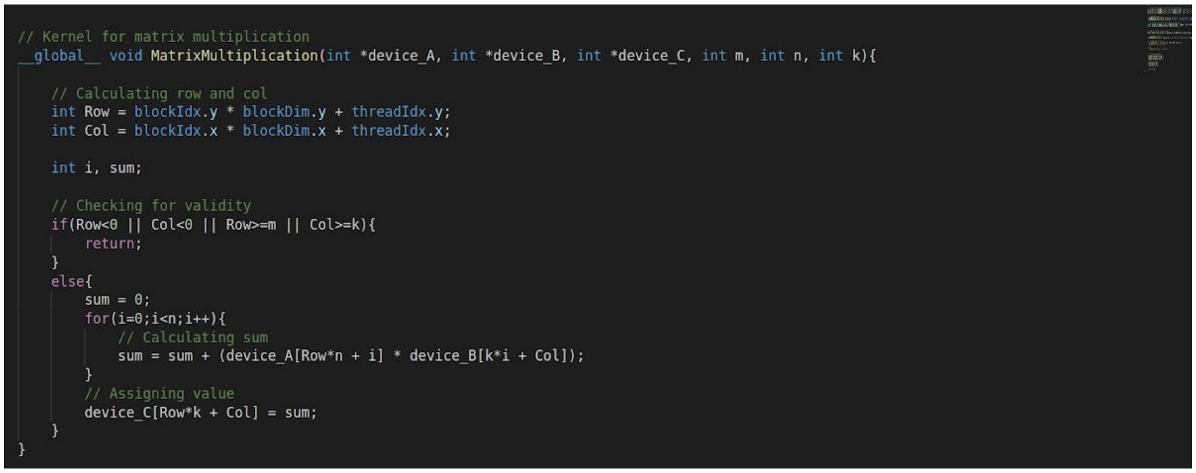


```
PS P:\heterogeneous-parallel-computing\A3\ImageColorToGrayscale> ./a input.ppm
[GPU    ] 0.002446454 Doing GPU memory allocation
[Copy   ] 0.005841447 Copying data to the GPU
[Compute] 0.000617432 Doing the computation on the GPU
[Copy   ] 0.003150982 Copying data from the GPU
[GPU    ] 0.043116200 Doing GPU Computation (memory + compute)
Solution is Correct
```

1. **How many floating operations are being performed in your color conversion kernel?**
   => 5 * imageHeight * imageWidth
2. **Which format would be more efficient for color conversion: a 2D matrix where each entry is an RGB value or a 3D matrix where each slice in the Z axis represents a color. i.e. is it better to have color interleaved in this application? Can you name an application where the opposite is true?**
   ==> For the application of conversion to grayscale a 2D representation would be more efficient as the reading in three consecutive values from a image would be faster due to them being next thus having spatial locality of reference than reading the three values of the 3D (which would be represented as 1D with RGB values far apart)
   Using a 3D representation is helpful in application of image blur where for each RGB channel the neighboring elements for a pixel in a channel is used. So 2D representation won't have good spatial locality of reference thus would be more efficient
3. **How many global memory reads are being performed by your kernel?**
   The number of reads are:
   => 3*imageHeight*imageWidth
4. **How many global memory writes are being performed by your kernel?**
   The number of writes are:
   => imageHeight*imageWidth
5. **Describe what possible optimizations can be implemented to your kernel to achieve a performance speedup.**
   Speed up can be achieved by launching 3 threads per pixel in a block and copying the data into shared memory. Reason is that there are 3 global reads made by each pixel, we can speed up the process by making use of shared memory instead. But we will need additional threads to write these values into the shared memory.

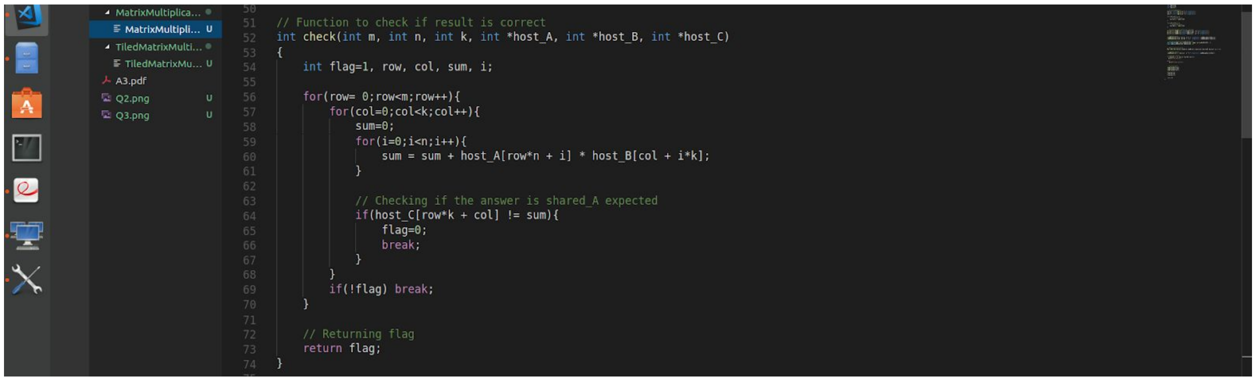
Q2)

Kernel



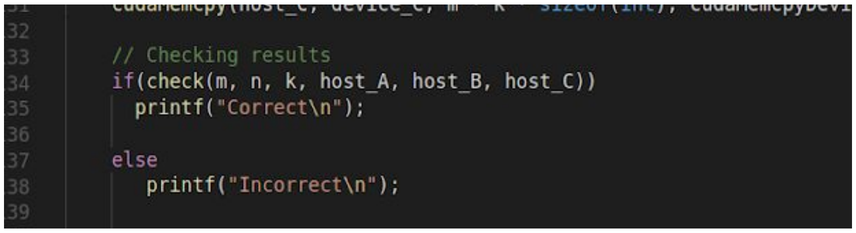
```
// Kernel for matrix multiplication
__global__ void MatrixMultiplication(int *device_A, int *device_B, int *device_C, int m, int n, int k){

    // Calculating row and col
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    int Col = blockIdx.x * blockDim.x + threadIdx.x;

    int i, sum;

    // Checking for validity
    if(Row<0 || Col<0 || Row>=m || Col>=k){
        return;
    }
    else{
        sum = 0;
        for(i=0;i<n;i++){
            // Calculating sum
            sum = sum + (device_A[Row*n + i] * device_B[k*i + Col]);
        }
        // Assigning value
        device_C[Row*k + Col] = sum;
    }
}
```

Function to check



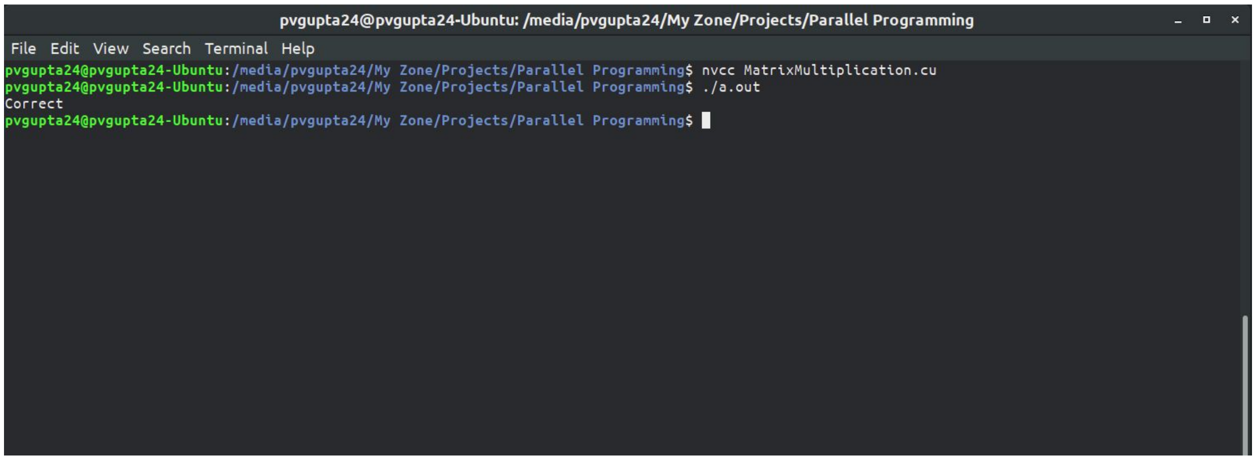
```
// Function to check if result is correct
int check(int m, int n, int k, int *host_A, int *host_B, int *host_C)
{
    int flag=1, row, col, sum, i;

    for(row= 0;row<m;row++){
        for(col=0;col<k;col++){
            sum=0;
            for(i=0;i<n;i++){
                sum = sum + host_A[row*n + i] * host_B[col + i*k];
            }

            // Checking if the answer is shared_A expected
            if(host_C[row*k + col] != sum){
                flag=0;
                break;
            }
        }
        if(!flag) break;
    }

    // Returning flag
    return flag;
}
```

Printing



```
    cudaMemcpy(host_C, device_C, m * k * sizeof(int), cudaMemcpyDevi

    // Checking results
    if(check(m, n, k, host_A, host_B, host_C))
      printf("Correct\n");

    else
       printf("Incorrect\n");

```

Output



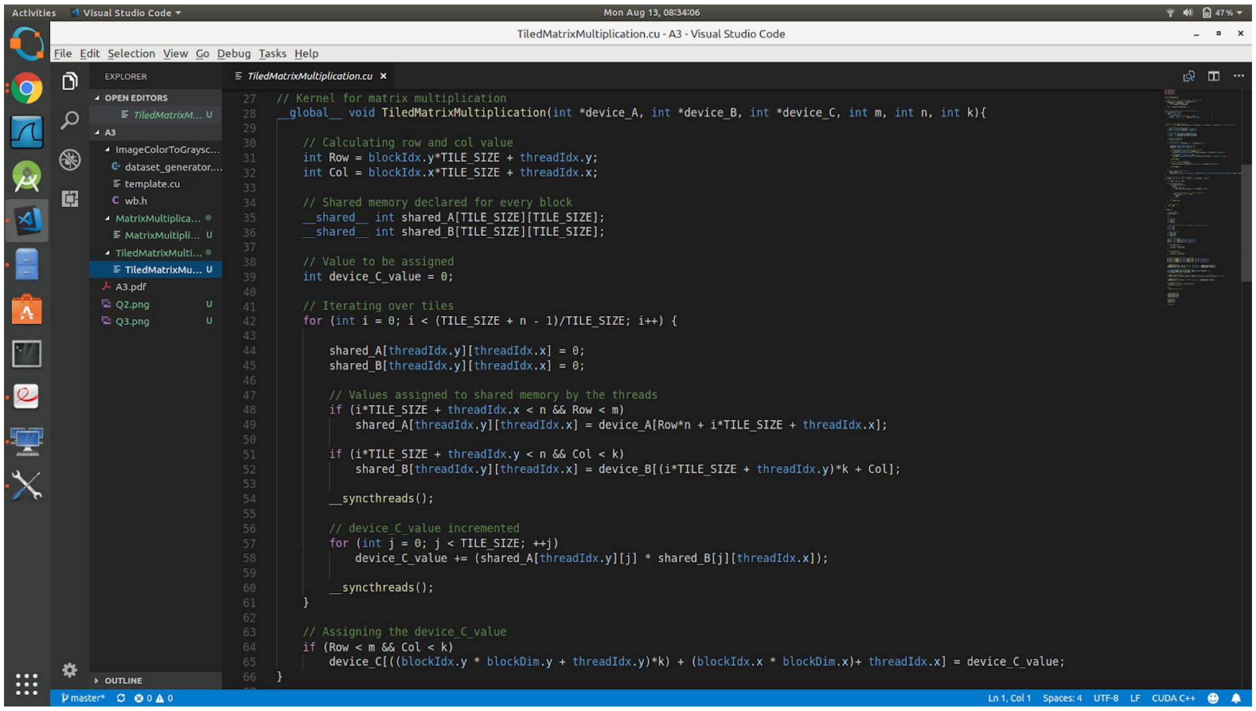
```
pvgupta24@pvgupta24-Ubuntu: /media/pvgupta24/My Zone/Projects/Parallel Programming
File  Edit  View  Search  Terminal  Help
pvgupta24@pvgupta24-Ubuntu:/media/pvgupta24/My Zone/Projects/Parallel Programming$ nvcc MatrixMultiplication.cu
pvgupta24@pvgupta24-Ubuntu:/media/pvgupta24/My Zone/Projects/Parallel Programming$ ./a.out
Correct
pvgupta24@pvgupta24-Ubuntu:/media/pvgupta24/My Zone/Projects/Parallel Programming$ 
```

Assume A => dimension (M*N), B => dimension (N*K) and C = A * B

1. **How many floating operations are being performed in your matrix multiply kernel?**
   Since matrix A has dimensions M*N and matrix B has dimensions N*K, the number of floating operations will be (M*K)*N
   => (512*512)*1024 = 268,435,456
2. **How many global memory reads are being performed by your kernel?**
   Matrix A and B are being read, hence reads are
   => M*N + N*K
   => 512*1024 + 512*1024 = 1,048,576
3. **How many global memory writes are being performed by your kernel?**
   Matrix C is being written into, hence
   => M*K
   => 512*512 = 262,144
4. **Describe what possible optimizations can be implemented to your kernel to achieve a performance speedup.**
   One possible way of speeding up the algorithm would be to form tiles in the input matrices, copy these into shared memory one by one and carry out operations.
   This would be faster because for each (input matrix) tile, a thread has to copy only 2 values into shared memory (rest are done by the adjacent threads) and then using this shared memory calculations are carried out. Accessing shared memory is faster than accessing global memory.

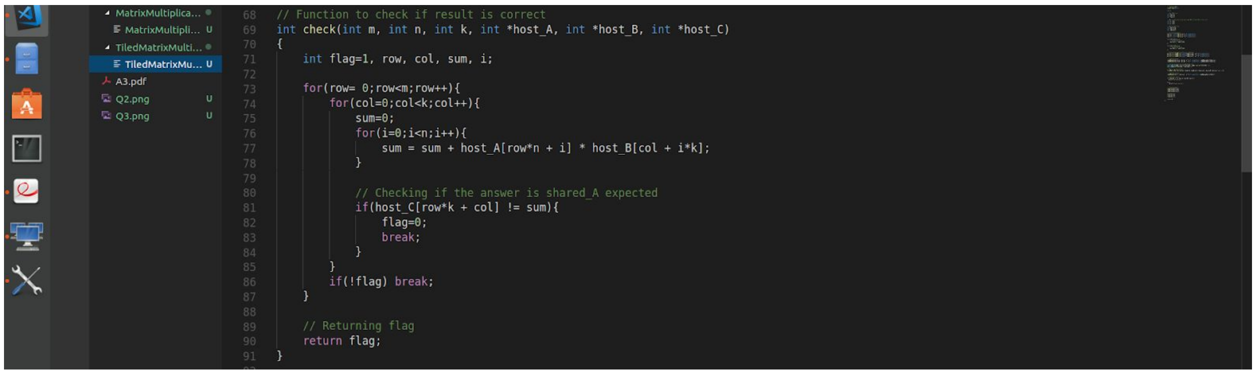
Q3)

Kernel



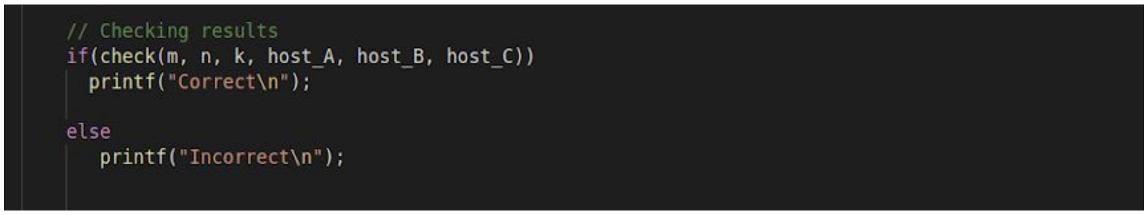
```c
// Kernel for matrix multiplication
__global__ void TiledMatrixMultiplication(int *device_A, int *device_B, int *device_C, int m, int n, int k){

    // Calculating row and col value
    int Row = blockIdx.y*TILE_SIZE + threadIdx.y;
    int Col = blockIdx.x*TILE_SIZE + threadIdx.x;

    // Shared memory declared for every block
    __shared__ int shared_A[TILE_SIZE][TILE_SIZE];
    __shared__ int shared_B[TILE_SIZE][TILE_SIZE];

    // Value to be assigned
    int device_C_value = 0;

    // Iterating over tiles
    for (int i = 0; i < (TILE_SIZE + n - 1)/TILE_SIZE; i++) {

        shared_A[threadIdx.y][threadIdx.x] = 0;
        shared_B[threadIdx.y][threadIdx.x] = 0;

        // Values assigned to shared memory by the threads
        if (i*TILE_SIZE + threadIdx.x < n && Row < m)
            shared_A[threadIdx.y][threadIdx.x] = device_A[Row*n + i*TILE_SIZE + threadIdx.x];

        if (i*TILE_SIZE + threadIdx.y < n && Col < k)
            shared_B[threadIdx.y][threadIdx.x] = device_B[(i*TILE_SIZE + threadIdx.y)*k + Col];

        __syncthreads();

        // device_C_value incremented
        for (int j = 0; j < TILE_SIZE; ++j)
            device_C_value += (shared_A[threadIdx.y][j] * shared_B[j][threadIdx.x]);

        __syncthreads();
    }

    // Assigning the device_C_value
    if (Row < m && Col < k)
        device_C[((blockIdx.y * blockDim.y + threadIdx.y)*k) + (blockIdx.x * blockDim.x)+ threadIdx.x] = device_C_value;
}
```

Function to check



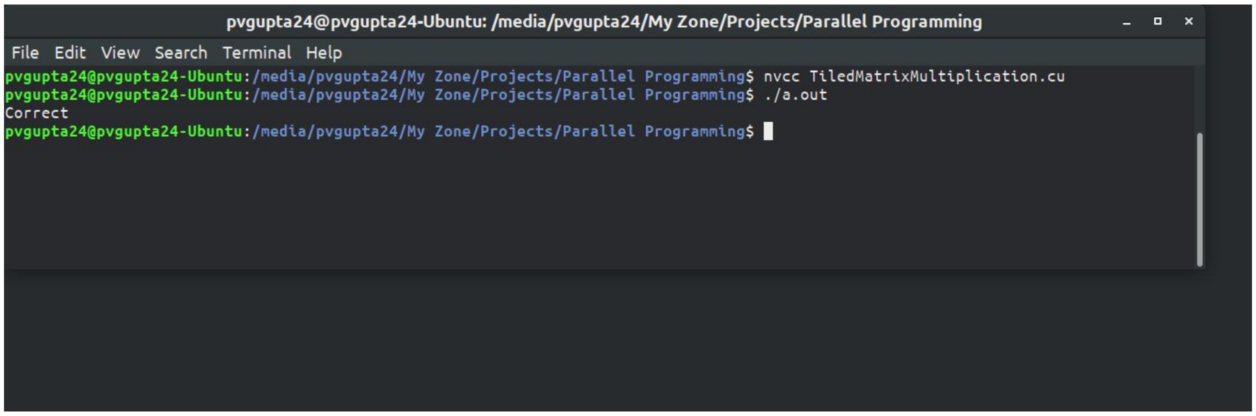
```c
// Function to check if result is correct
int check(int m, int n, int k, int *host_A, int *host_B, int *host_C)
{
    int flag=1, row, col, sum, i;

    for(row= 0;row<m;row++){
        for(col=0;col<k;col++){
            sum=0;
            for(i=0;i<n;i++){
                sum = sum + host_A[row*n + i] * host_B[col + i*k];
            }

            // Checking if the answer is shared_A expected
            if(host_C[row*k + col] != sum){
                flag=0;
                break;
            }
        }
        if(!flag) break;
    }

    // Returning flag
    return flag;
}
```

Printing



```c
// Checking results
if(check(m, n, k, host_A, host_B, host_C))
  printf("Correct\n");

else
   printf("Incorrect\n");
```

Output



```
pvgupta24@pvgupta24-Ubuntu:/media/pvgupta24/My Zone/Projects/Parallel Programming$ nvcc TiledMatrixMultiplication.cu
pvgupta24@pvgupta24-Ubuntu:/media/pvgupta24/My Zone/Projects/Parallel Programming$ ./a.out
Correct
pvgupta24@pvgupta24-Ubuntu:/media/pvgupta24/My Zone/Projects/Parallel Programming$
```

Assume A => dimension (M*N), B => dimension (N*K) and C = A * B

1. **How many floating operations are being performed in your matrix multiply kernel? Explain.**
   Operations => M*N*K
2. **How many global memory reads are being performed by your kernel? Explain.**
   Reads
   => Number of tiles in C * Globally read (input) tiles per tile in C * Elements per tile
   => (M/Tile_size * K/Tile_size) * (2 * N/Tile_size) * (Tile_size * Tile_size)
3. **How many global memory writes are being performed by your kernel? Explain.**
   Writes => M*K
4. **Describe what further optimizations can be implemented to your kernel to achieve a performance speedup.**
   Strassen's algorithm could have been used for optimising the time complexity of matrix multiplication.
5. **Compare the implementation difficulty of this kernel compared to the previous MP. What difficulties did you have with this implementation?**
   Implementation is slightly more complex. Calculating the number of tiles to be read per output tile, corner cases etc had to be taken care of.
6. **Suppose you have matrices with dimensions bigger than the max thread dimensions. Sketch an algorithm that would perform matrix multiplication algorithm that would perform the multiplication in this case.**
   We need an algorithm in which storage requirements remain constant. Cannon's algorithm or Scalable Universal Matrix Multiplication Algorithm could have been used for that purpose.
7. **Suppose you have matrices that would not fit in global memory. Sketch an algorithm that would perform matrix multiplication algorithm that would perform the multiplication out of place.**
   Since, we cannot fit the matrices into global memory at once we have to split matrix A and matrix B into X partitions, each of size X_size * X_size. Then load these partitions one by one and find the resultant matrix.

Algorithm would go as follows, for each partition of resultant matrix, load partitions of size X_size * X_size from both matrix A and B. Add these values to the result matrix and then load the next partitions from A and B, so on.