Name: Mrunal Khade

PRN No: 2020BTECS00057

High Performance Computing Lab

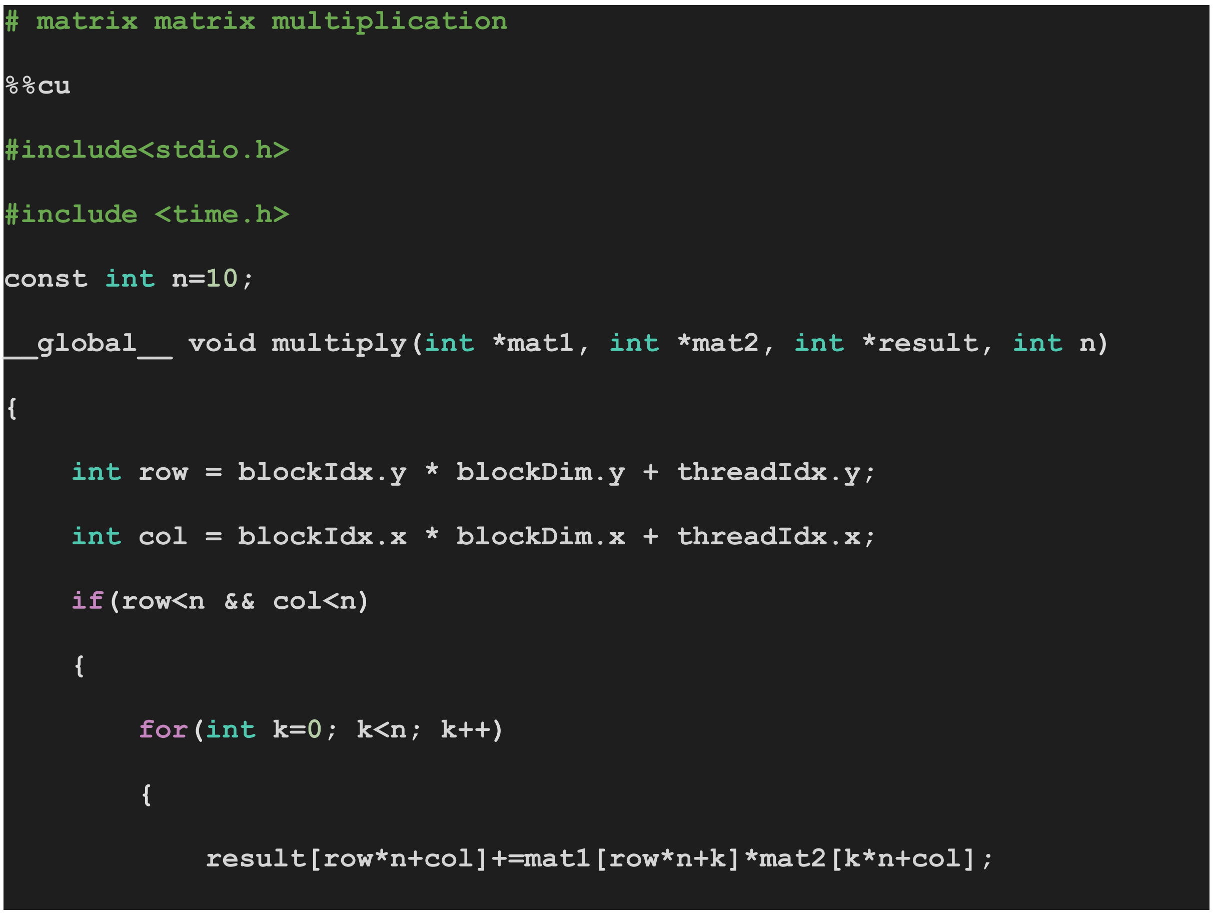
Practical No. 9

Title of practical: Implementation of Matrix-matrix Multiplication (global and shared Memory), Prefix sum, 2D Convolution using CUDA C

Problem Statement 1:
Implement Matrix-matrix Multiplication using global memory in CUDA C. Analyze and tune the program for getting maximum speed up. Do Profiling and state what part of the code takes the huge amount of time to execute.

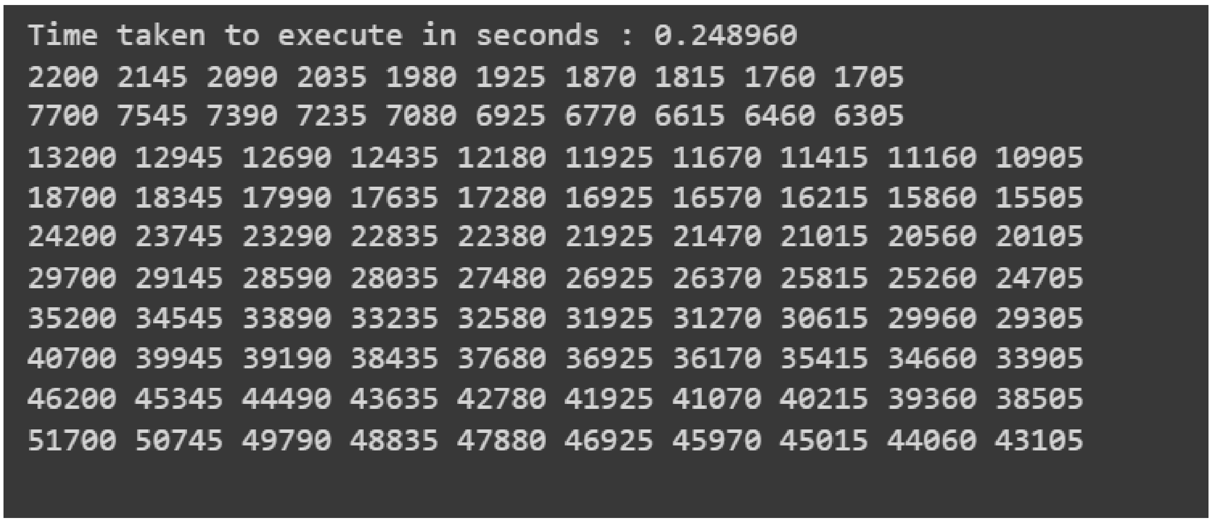
Code:

Screenshots:



```
# matrix matrix multiplication

%%cu

#include<stdio.h>

#include <time.h>

const int n=10;

__global__ void multiply(int *mat1, int *mat2, int *result, int n)

{

    int row = blockIdx.y * blockDim.y + threadIdx.y;

    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if(row<n && col<n)

    {

        for(int k=0; k<n; k++)

        {

            result[row*n+col]+=mat1[row*n+k]*mat2[k*n+col];
        }
    }
}



int main()
{
    clock_t start, end;
    start = clock();
    int mat1[n*n];
    int mat2[n*n];
    for(int i=0;i<n*n;i++)
    {
        mat1[i]=i+1;
        mat2[i]=n*n-i;
    }
    int result[n*n];
    int *d_mat1, *d_mat2, *d_result;
    cudaMalloc(&d_mat1, n*n*sizeof(int));
    cudaMalloc(&d_mat2, n*n*sizeof(int));
    cudaMalloc(&d_result, n*n*sizeof(int));



    cudaMemcpy(d_mat1, mat1, n*n*sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_mat2, mat2, n*n*sizeof(int), cudaMemcpyHostToDevice);



    int b_size=2;
    int g_size=ceil(n/2.0);
    dim3 threads(b_size, b_size);

    dim3 blocks(g_size, g_size);

    multiply<<<blocks,threads>>>(d_mat1,d_mat2,d_result,n);

    cudaDeviceSynchronize();



                    cudaMemcpy(result,     d_result,     n*n*sizeof(int),
cudaMemcpyDeviceToHost);



    end = clock();

    double duration = ((double)end - start) / CLOCKS_PER_SEC;

    printf("\nTime taken to execute in seconds : %f\n", duration);



    for(int i=0;i<n;i++)

    {

        for(int j=0;j<n;j++)

        {

            printf("%d ",result[i*n+j]);

        }

        printf("\n");

    }

    return 0;

}
```

Output:



```
Time taken to execute in seconds : 0.248960
2200 2145 2090 2035 1980 1925 1870 1815 1760 1705
7700 7545 7390 7235 7080 6925 6770 6615 6460 6305
13200 12945 12690 12435 12180 11925 11670 11415 11160 10905
18700 18345 17990 17635 17280 16925 16570 16215 15860 15505
24200 23745 23290 22835 22380 21925 21470 21015 20560 20105
29700 29145 28590 28035 27480 26925 26370 25815 25260 24705
35200 34545 33890 33235 32580 31925 31270 30615 29960 29305
40700 39945 39190 38435 37680 36925 36170 35415 34660 33905
46200 45345 44490 43635 42780 41925 41070 40215 39360 38505
51700 50745 49790 48835 47880 46925 45970 45015 44060 43105
```

Analysis:

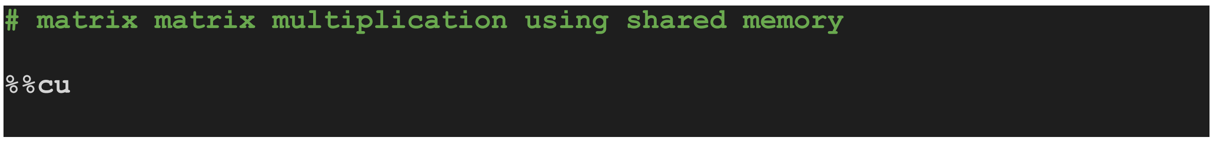
Number of threads	**Data Size (n)**	**Execution time**
$O(n^2)$	10	0.195431
$O(n^2)$	100	0.192159
$O(n^2)$	500	0.213233

For performing the matrix-matrix multiplication, we have used the block of n^2 threads as we have to calculate n^2 elements in the resultant matrix. For calculating each element of the result matrix, row of the first matrix and one column of the second matrix is needed. So, we have assigned the unique row and column to each thread. We can get the unique row and column by using the block indexes and thread indexes as they are unique as a combination.

Problem Statement 2:
Implement Matrix-matrix Multiplication using shared memory in CUDA C. Analyze and tune the program for getting maximum speed up. Do Profiling and state what part of the code takes the huge amount of time to execute.

Code:

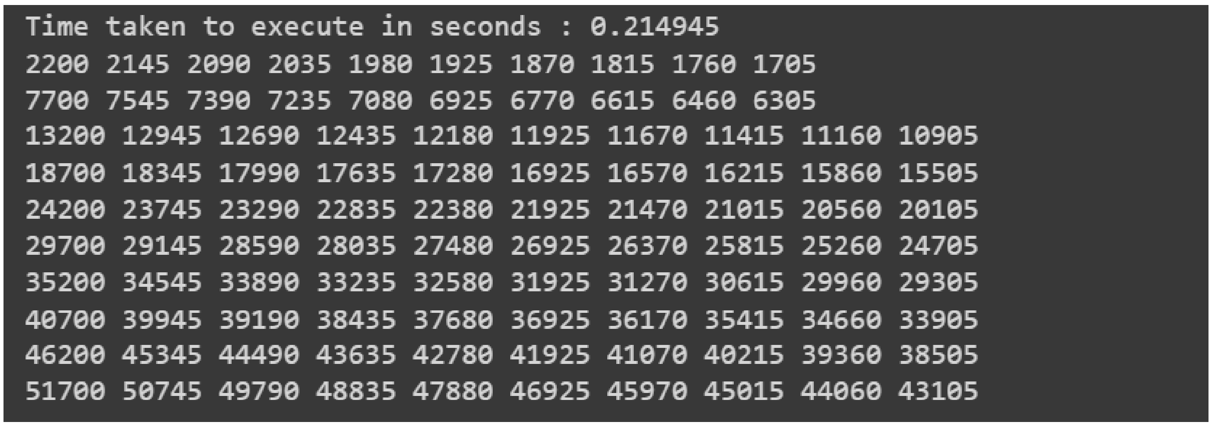
Screenshots:



```
# matrix matrix multiplication using shared memory

%%cu
```

```
#include<stdio.h>

#include <time.h>

const int n =10;

__global__ void multiply(int *mat1, int *mat2, int *result, int n)

{

    int row = blockIdx.y;

    int col = blockIdx.x;

    int k = threadIdx.x;



    __shared__ int temp[1000];

    temp[k]=mat1[row*n+k]*mat2[k*n+col];

    __syncthreads();



    for(int i=0;i<n;i++)

    {

        result[row*n+col]+=temp[i];

    }

}



int main()

{

    clock_t start, end;

    start = clock();

    int mat1[n*n];

    int mat2[n*n];

    for(int i=0;i<n*n;i++)
    {

        mat1[i]=i+1;

        mat2[i]=n*n-i;

    }

    int result[n*n];

    int *d_mat1, *d_mat2, *d_result;

    cudaMalloc(&d_mat1, n*n*sizeof(int));

    cudaMalloc(&d_mat2, n*n*sizeof(int));

    cudaMalloc(&d_result, n*n*sizeof(int));



    cudaMemcpy(d_mat1, mat1, n*n*sizeof(int), cudaMemcpyHostToDevice);

    cudaMemcpy(d_mat2, mat2, n*n*sizeof(int), cudaMemcpyHostToDevice);



    dim3 blocks(n,n);

    multiply<<<blocks,n>>>(d_mat1,d_mat2,d_result,n);

    cudaDeviceSynchronize();



                    cudaMemcpy(result,     d_result,     n*n*sizeof(int),
cudaMemcpyDeviceToHost);



    end = clock();

    double duration = ((double)end - start) / CLOCKS_PER_SEC;

    printf("\nTime taken to execute in seconds : %f\n", duration);
    for(int i=0;i<n;i++)

    {

        for(int j=0;j<n;j++)

        {

            printf("%d ",result[i*n+j]);

        }

        printf("\n");

    }



    return 0;

}
```

Output:



```
Time taken to execute in seconds : 0.214945
2200 2145 2090 2035 1980 1925 1870 1815 1760 1705
7700 7545 7390 7235 7080 6925 6770 6615 6460 6305
13200 12945 12690 12435 12180 11925 11670 11415 11160 10905
18700 18345 17990 17635 17280 16925 16570 16215 15860 15505
24200 23745 23290 22835 22380 21925 21470 21015 20560 20105
29700 29145 28590 28035 27480 26925 26370 25815 25260 24705
35200 34545 33890 33235 32580 31925 31270 30615 29960 29305
40700 39945 39190 38435 37680 36925 36170 35415 34660 33905
46200 45345 44490 43635 42780 41925 41070 40215 39360 38505
51700 50745 49790 48835 47880 46925 45970 45015 44060 43105
```

Analysis:

Analysis:

Number of threads	**Data Size (n)**	**Execution time**
O(n^3)	10	0.214945
O(n^3)	100	0.196268
O(n^3)	500	0.287709

For performing the matrix-matrix multiplication, we have used the n^2 blocks of n threads each as we have to calculate n^2 elements in the resultant matrix. In this computation, for calculating each element of the result matrix, we need n threads.

In this implementation, the matrix multiplication is computed using a grid of thread blocks. Each thread block is responsible for computing a sub-matrix of the resulting matrix. Threads within a block cooperate to load the necessary data into shared memory and perform the matrix multiplication. The use of shared memory minimizes global memory accesses, leading to significant speedups.

Analysis by comparing shared and global version:

While using the global memory version of the program, one element of the result matrix is calculated by the one thread. But, in case of the shared memory each element is calculated by one block consisting of n threads each.

We can get the significant speedup in case of the shared memory program as it reduces the global memory accesses and increases the local memory accesses.

Problem Statement 3:

Implement Prefix sum using CUDA C. Analyze and tune the program for getting maximum speed up. Do Profiling and state what part of the code takes the huge amount of time to execute.

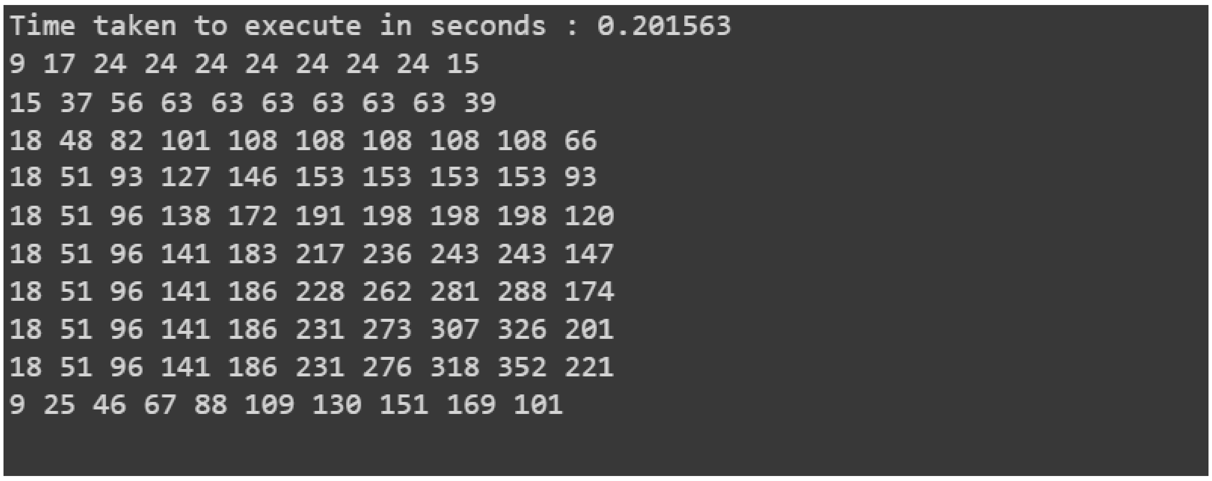
Code:

```
%%cu
#include<stdio.h>
#include <time.h>
const int n =10;
__global__ void calculate2DConvolution(int *image, int *mask, int
*result, int n, int maskdim)
{
    int offset = maskdim/2;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if(row<n && col<n)
    {
        int start_row = row - offset;
        int start_col = col - offset;
        for(int i=0;i<maskdim;i++)
        {
            for(int j=0;j<maskdim;j++)
            {
                if(start_row+i>=0 && start_row+i<n && start_col+j>=0 &&
start_col+j<n)
                {
                    int cr = start_row+i, cc = start_col+j;

result[row*n+col]+=image[cr*n+cc]*mask[i*maskdim+j];
                }
            }
        }
    }
}
int main()
{
    clock_t start, end;
    start = clock();
    int maskdim =3;
    int image[n*n];
    for(int i=0;i<n*n;i++)
    {
        int x=i/n;
        int y=i%n;
        image[i]=min(x,y);
    }
    int mask[]={1,2,3,4,5,6,7,8,9};
    int result[n*n];
    int *d_image, *d_mask, *d_result;
    cudaMalloc(&d_image,n*n*sizeof(int));
    cudaMalloc(&d_mask, maskdim*maskdim*sizeof(int));
    cudaMalloc(&d_result, n*n*sizeof(int));
    cudaMemcpy(d_image, image, n*n*sizeof(int),
cudaMemcpyHostToDevice);
    cudaMemcpy(d_mask, mask, maskdim*maskdim*sizeof(int),
cudaMemcpyHostToDevice);
    int thread=2;
    int block=ceil((n*1.0)/thread);
    dim3 blocks(block,block);
    dim3 threads(thread,thread);
```

```
calculate2DConvolution<<<blocks,threads>>>(d_image,d_mask,d_result,n,ma
skdim);
    cudaDeviceSynchronize();
    cudaMemcpy(result, d_result, n*n*sizeof(int),
cudaMemcpyDeviceToHost);


    end = clock();
    double duration = ((double)end - start) / CLOCKS_PER_SEC;
    printf("\nTime taken to execute in seconds : %f\n", duration);
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            printf("%d ",result[i*n+j]);
        }
        printf("\n");
    }
    return 0;
}
```

Screenshots:

For input size: n=10:



```
Time taken to execute in seconds : 0.201563
9 17 24 24 24 24 24 24 24 15
15 37 56 63 63 63 63 63 63 39
18 48 82 101 108 108 108 108 108 66
18 51 93 127 146 153 153 153 153 93
18 51 96 138 172 191 198 198 198 120
18 51 96 141 183 217 236 243 243 147
18 51 96 141 186 228 262 281 288 174
18 51 96 141 186 231 273 307 326 201
18 51 96 141 186 231 276 318 352 221
9 25 46 67 88 109 130 151 169 101
```

Analysis:

When implementing 2D convolution using shared memory in CUDA, the computation is distributed among threads within a block, with each thread responsible for computing an element of the output matrix. The implementation involves loading the necessary data into shared memory to minimize global memory transactions and maximize memory access efficiency.

Number of threads	**Data Size (n)**	**Execution time**
O(n^2)	10	0.201563
O(n^2)	100	0.224688
O(n^2)	1000	