**Class:** Final Year (Computer Science and Engineering)

**Year:** 2022-23 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 10**

**Exam Seat No:** 2019BTECS00070

**Name:** Prathmesh Killedar

**Title of practical:**

Implementation of Matrix-matrix Multiplication, 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

**Information #:**

#include <stdio.h>

#define N 64

\_\_global\_\_ void matrixMulGPU( int \* a, int \* b, int \* c )

{

int val = 0;

int row = blockIdx.x \* blockDim.x + threadIdx.x;

int col = blockIdx.y \* blockDim.y + threadIdx.y;

if (row < N && col < N)

{

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

val += a[row \* N + k] \* b[k \* N + col];

c[row \* N + col] = val;

}

}

void matrixMulCPU( int \* a, int \* b, int \* c )

{

int val = 0;

for( int row = 0; row < N; ++row )

for( int col = 0; col < N; ++col )

{

val = 0;

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

val += a[row \* N + k] \* b[k \* N + col];

c[row \* N + col] = val;

}

}

int main()

{

int \*a, \*b, \*c\_cpu, \*c\_gpu;

int size = N \* N \* sizeof (int); // Number of bytes of an N x N matrix

// Allocate memory

cudaMallocManaged (&a, size);

cudaMallocManaged (&b, size);

cudaMallocManaged (&c\_cpu, size);

cudaMallocManaged (&c\_gpu, size);

// Initialize memory

for( int row = 0; row < N; ++row )

for( int col = 0; col < N; ++col )

{

a[row\*N + col] = row;

b[row\*N + col] = col+2;

c\_cpu[row\*N + col] = 0;

c\_gpu[row\*N + col] = 0;

}

dim3 threads\_per\_block (16, 16, 1); // A 16 x 16 block threads

dim3 number\_of\_blocks ((N / threads\_per\_block.x) + 1, (N / threads\_per\_block.y) + 1, 1);

matrixMulGPU <<< number\_of\_blocks, threads\_per\_block >>> ( a, b, c\_gpu );

cudaDeviceSynchronize(); // Wait for the GPU to finish before proceeding

// Call the CPU version to check our work

matrixMulCPU( a, b, c\_cpu );

// Compare the two answers to make sure they are equal

bool error = false;

for( int row = 0; row < N && !error; ++row )

for( int col = 0; col < N && !error; ++col )

if (c\_cpu[row \* N + col] != c\_gpu[row \* N + col])

{

printf("FOUND ERROR at c[%d][%d]\n", row, col);

error = true;

break;

}

if (!error)

printf("Success!\n");

// Free all our allocated memory

cudaFree(a);

cudaFree(b);

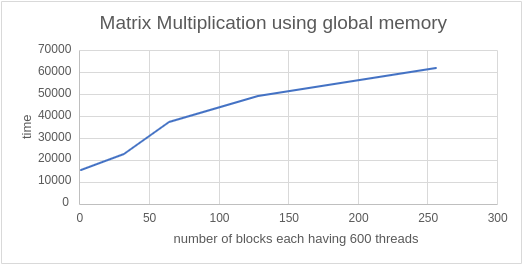
cudaFree( c\_cpu );

cudaFree( c\_gpu );

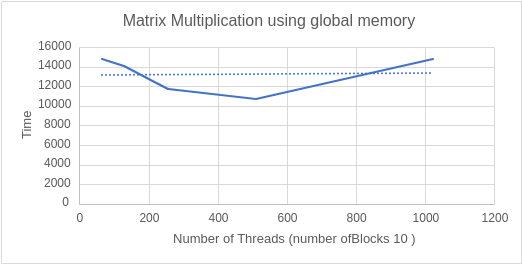
}

**Serial execution time: 0.000224 second**

| **Number of blocks (600 threads each)** | **Time required** | **speedup** |
| --- | --- | --- |
| 1 | 15168 | 14.2857 |
| 32 | 22687 | 9.8734 |
| 64 | 37343 | 5.9984 |
| 128 | 48991 | 4.5722 |
| 256 | 61983 | 3.6138 |



| **Number of Threads with constant block size 10** | **Time required** | **speedup** |
| --- | --- | --- |
| 64 | 14816 | 15.1187 |
| 128 | 14048 | 15.9453 |
| 256 | 11776 | 19.0217 |
| 512 | 10656 | 21.021 |
| 1024 | 14816 | 15.1187 |



**Conclusion:**

1. For constant number of threads we have concluded that the execution time is increasing with the increasing number of blocks
2. For constant number of block we have concluded that the execution time is decreasing until a certain point and after that it is increasing due to communication overhead by increasing the number of threads per block

## 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 a huge amount of time to execute.

**Information #:**

#include <stdio.h>

#include <math.h>

#define TILE\_WIDTH 2

/\*matrix multiplication kernels\*/

// shared

\_\_global\_\_ void

MatrixMulSh( float \*Md , float \*Nd , float \*Pd , const int WIDTH )

{

//Taking shared array to break the MAtrix in Tile widthand fetch them in that array per ele

\_\_shared\_\_ float Mds [TILE\_WIDTH][TILE\_WIDTH] ;

\_\_shared\_\_ float Nds [TILE\_WIDTH][TILE\_WIDTH] ;

// calculate thread id

unsigned int col = TILE\_WIDTH\*blockIdx.x + threadIdx.x ;

unsigned int row = TILE\_WIDTH\*blockIdx.y + threadIdx.y ;

for (int m = 0 ; m<WIDTH/TILE\_WIDTH ; m++ ) // m indicate number of phase

{

Mds[threadIdx.y][threadIdx.x] = Md[row\*WIDTH + (m\*TILE\_WIDTH + threadIdx.x)] ;

Nds[threadIdx.y][threadIdx.x] = Nd[ ( m\*TILE\_WIDTH + threadIdx.y) \* WIDTH + col] ;

\_\_syncthreads() ; // for synchronizing the threads

// Do for tile

for ( int k = 0; k<TILE\_WIDTH ; k++ )

Pd[row\*WIDTH + col]+= Mds[threadIdx.x][k] \* Nds[k][threadIdx.y] ;

\_\_syncthreads() ; // for synchronizing the threads

}

}

// main routine

int main ()

{

const int WIDTH = 500;

float array1\_h[WIDTH][WIDTH] ,array2\_h[WIDTH][WIDTH], M\_result\_array\_h[WIDTH][WIDTH] ;

float \*array1\_d , \*array2\_d ,\*result\_array\_d ,\*M\_result\_array\_d ; // device array

int i , j ;

//input in host array

for ( i = 0 ; i<WIDTH ; i++ )

{

for (j = 0 ; j<WIDTH ; j++ )

{

array1\_h[i][j] = (i + 2\*j) %500 ;

array2\_h[i][j] = (i + 3\*j) %500 ;

}

}

//create device array cudaMalloc ( (void \*\*)&array\_name, sizeofmatrixinbytes) ;

cudaMalloc((void \*\*) &array1\_d , WIDTH\*WIDTH\*sizeof (int) ) ;

cudaMalloc((void \*\*) &array2\_d , WIDTH\*WIDTH\*sizeof (int) ) ;

//copy host array to device array; cudaMemcpy ( dest , source , WIDTH , direction )

cudaMemcpy ( array1\_d , array1\_h , WIDTH\*WIDTH\*sizeof (int) , cudaMemcpyHostToDevice ) ;

cudaMemcpy ( array2\_d , array2\_h , WIDTH\*WIDTH\*sizeof (int) , cudaMemcpyHostToDevice ) ;

//allocating memory for resultent device array

cudaMalloc((void \*\*) &result\_array\_d , WIDTH\*WIDTH\*sizeof (int) ) ;

cudaMalloc((void \*\*) &M\_result\_array\_d , WIDTH\*WIDTH\*sizeof (int) ) ;

MatrixMulSh<<<512,32>>> ( array1\_d , array2\_d ,M\_result\_array\_d , WIDTH) ;

// all gpu function blocked till kernel is working

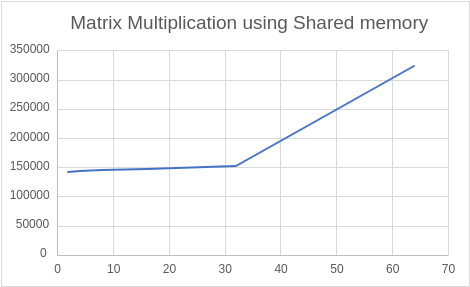
//copy back result\_array\_d to result\_array\_h

cudaMemcpy(M\_result\_array\_h , M\_result\_array\_d , WIDTH\*WIDTH\*sizeof(int) ,cudaMemcpyDeviceToHost) ;

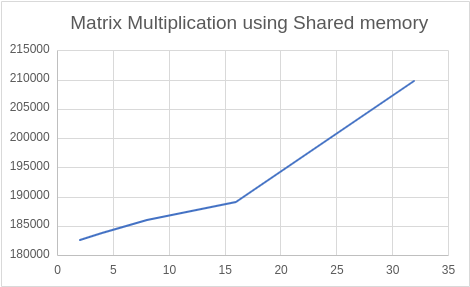
printf("Multiplication Successful using shared Memory");

}

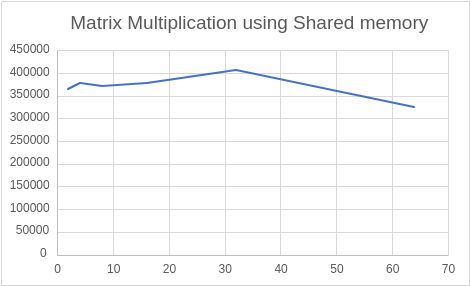
| **Number of Threads with constant block size 256** | **Time required** |
| --- | --- |
| 2 | 141692 |
| 4 | 143356 |
| 8 | 145020 |
| 16 | 146556 |
| 32 | 151196 |
| 64 | 323192 |



| **Number of Threads with constant block size 512** | **Time required** |
| --- | --- |
| 2 | 182555 |
| 4 | 183771 |
| 8 | 185947 |
| 16 | 189019 |
| 32 | 209755 |



| **Number of Threads with constant block size 1024** | **Time required** |
| --- | --- |
| 2 | 364726 |
| 4 | 377142 |
| 8 | 372630 |
| 16 | 378135 |
| 32 | 406197 |
| 64 | 324248 |



**Conclusion:**

1. For constant number of blocks we have concluded that the execution time is increasing with the increasing number of threads
2. For constant number of threads per block at a partic

## 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.

**Information #:**

#include <stdio.h>

void initWith(float val, float \*arr, int N)

{

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

{

arr[i] = val;

}

}

\_\_global\_\_

void prefixSum(float \*arr, float \*res, float \*ptemp, float\* ttemp, int N)

{

int threadId = blockIdx.x \* blockDim.x + threadIdx.x;

int totalThreads = gridDim.x \* blockDim.x;

int elementsPerThread = ceil(1.0 \* N / totalThreads);

int start = threadId \* elementsPerThread;

int count = 0;

float \*sums = new float[elementsPerThread];

float sum = 0;

for (int i = start; i < N && count < elementsPerThread; i++, count++) {

sum += arr[i];

sums[count] = sum;

}

float localSum;

if (count)

localSum = sums[count - 1];

else

localSum = 0;

ptemp[threadId] = localSum;

ttemp[threadId] = localSum;

\_\_syncthreads();

if (totalThreads == 1) {

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

res[i] = sums[i];

} else {

int d = 0; // log2(totalThreads)

int x = totalThreads;

while (x > 1) {

d++;

x = x >> 1;

}

x = 1;

for (int i = 0; i < 2\*d; i++) {

int tsum = ttemp[threadId];

\_\_syncthreads();

int newId = threadId / x;

if (newId % 2 == 0) {

int nextId = threadId + x;

ptemp[nextId] += tsum;

ttemp[nextId] += tsum;

} else {

int nextId = threadId - x;

ttemp[nextId] += tsum;

}

x = x << 1;

}

\_\_syncthreads();

float diff = ptemp[threadId] - localSum;

for (int i = start, j = 0; i < N && j < count; i++, j++) {

res[i] = sums[j] + diff;

}

}

}

void checkRes(float \*arr, float \*res, int N, float \*ptemp, float\* ttemp)

{

float sum = 0;

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

{

sum += arr[i];

if (sum != res[i])

{

printf("FAIL: res[%d] - %0.0f does not equal %0.0f\n", i, res[i], sum);

exit(1);

}

}

printf("SUCCESS! All prefix sums added correctly.\n");

}

int main()

{

const int N = 1000000;

size\_t size = N \* sizeof(float);

float \*arr;

float \*res;

cudaMallocManaged(&arr, size);

cudaMallocManaged(&res, size);

initWith(2, arr, N);

initWith(0, res, N);

int blocks = 1;

int threadsPerBlock = 32;

int totalThreads = blocks \* threadsPerBlock;

float \*ptemp;

float \*ttemp;

cudaMallocManaged(&ptemp, totalThreads \* sizeof(float));

cudaMallocManaged(&ttemp, totalThreads \* sizeof(float));

prefixSum<<<blocks, threadsPerBlock>>>(arr, res, ptemp, ttemp, N);

cudaDeviceSynchronize();

checkRes(arr, res, N, ptemp, ttemp);

cudaFree(arr);

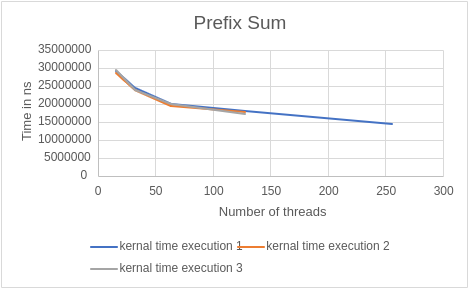
cudaFree(res);

cudaFree(ttemp);

cudaFree(ptemp);

}

| **No of threads (1 block)** | **kernal time execution 1** | **kernal time execution 2** | **kernal time execution 3** |
| --- | --- | --- | --- |
| 16 | 29185007 | 28690201 | 29448042 |
| 32 | 24658464 | 23898575 | 24076521 |
| 64 | 20158082 | 19436169 | 20129785 |
| 128 | 18123523 | 17824168 | 17406672 |
| 256 | 14619018 |  |  |



**Conclusion:**

As there is lack of synchronisation in blocks but there is synchronisation in threads, so for the prefix sum problem we consider only one block with varying number of threads. So by observing the above graph we have concluded that as the number of threads increases execution time is decreasing.

### Problem Statement 4:

Implement 2D Convolution using shared memory 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.

**Information #:**

#include <stdio.h>

#define MASK\_DIM 7

#define MASK\_OFFSET (MASK\_DIM / 2)

\_\_constant\_\_ int mask[7 \* 7];

\_\_global\_\_ void convolution\_2d(int \*matrix, int \*result, int N)

{

// Calculate the global thread positions

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

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

// Starting index for calculation

int start\_r = row - MASK\_OFFSET;

int start\_c = col - MASK\_OFFSET;

// Temp value for accumulating the result

int temp = 0;

// Iterate over all the rows

for (int i = 0; i < MASK\_DIM; i++)

{

// Go over each column

for (int j = 0; j < MASK\_DIM; j++)

{

// Range check for rows

if ((start\_r + i) >= 0 && (start\_r + i) < N)

{

// Range check for columns

if ((start\_c + j) >= 0 && (start\_c + j) < N)

{

// Accumulate result

temp += matrix[(start\_r + i) \* N + (start\_c + j)] \* mask[i \* MASK\_DIM + j];

}

}

}

}

// Write back the result

result[row \* N + col] = temp;

}

void init\_matrix(int \*m, int n)

{

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

{

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

{

m[n \* i + j] = rand() % 100;

}

}

}

void verify\_result(int \*m, int \*mask, int \*result, int N)

{

int temp;

int offset\_r;

int offset\_c;

// Go over each row

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

{

// Go over each column

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

{

// Reset the temp variable

temp = 0;

// Go over each mask row

for (int k = 0; k < MASK\_DIM; k++)

{

// Update offset value for row

offset\_r = i - MASK\_OFFSET + k;

// Go over each mask column

for (int l = 0; l < MASK\_DIM; l++)

{

// Update offset value for column

offset\_c = j - MASK\_OFFSET + l;

// Range checks if we are hanging off the matrix

if (offset\_r >= 0 && offset\_r < N)

{

if (offset\_c >= 0 && offset\_c < N)

{

// Accumulate partial results

temp += m[offset\_r \* N + offset\_c] \* mask[k \* MASK\_DIM + l];

}

}

}

}

// Fail if the results don't match

if (result[i \* N + j] != temp)

{

printf("Check failed");

return;

}

}

}

}

int main()

{

int N = 1 << 10; // 2^10

size\_t bytes\_n = N \* N \* sizeof(int);

size\_t bytes\_m = MASK\_DIM \* MASK\_DIM \* sizeof(int);

int \*matrix;

int \*result;

int \*h\_mask;

cudaMallocManaged(&matrix, bytes\_n);

cudaMallocManaged(&result, bytes\_n);

cudaMallocManaged(&h\_mask, bytes\_m);

init\_matrix(matrix, N);

init\_matrix(mask, MASK\_DIM);

cudaMemcpyToSymbol(mask, h\_mask, bytes\_m);

// Calculate grid dimensions

//int THREADS = 64;

//int BLOCKS = (N + THREADS - 1) / THREADS;

// Dimension launch arguments

//dim3 block\_dim(THREADS, THREADS);

//dim3 grid\_dim(BLOCKS, BLOCKS);

//printf("%d %d",grid\_dim.y,block\_dim.y);

convolution\_2d<<<128, 1024>>>(matrix, result, N);

verify\_result(matrix, h\_mask, result, N);

printf("COMPLETED SUCCESSFULLY!");

cudaFree(matrix);

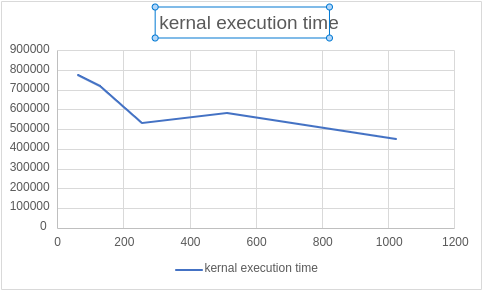
cudaFree(result);

cudaFree(h\_mask);

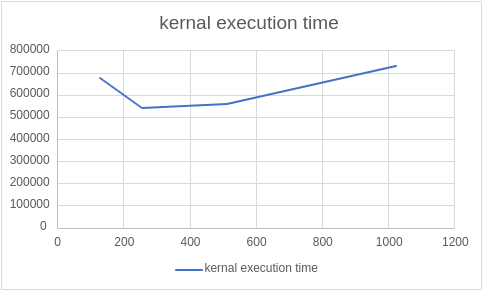
return 0;

}

| **Blocks (thread constant 128)** | **kernal execution time** |
| --- | --- |
| 64 | 777744 |
| 128 | 720112 |
| 256 | 533781 |
| 512 | 582900 |
| 1024 | 448567 |



| **Threads(Block Constant 128)** | **kernal execution time** |
| --- | --- |
| 128 | 674482 |
| 256 | 539349 |
| 512 | 559860 |
| 1024 | 731410 |



**Github Link:**[**https://github.com/killedar27/HPC-assignments/tree/main/assignment10**](https://github.com/killedar27/HPC-assignments/tree/main/assignment10)