Name:Vikas Mane
Class:BE-B
Roll Number:B1921152
PRN Number:72000291F

Experiment No: 6

Title: Vector and Matrix Operations-
Design parallel algorithm to
1. Add two large vectors
2. Multiply Vector and Matrix
3. Multiply two N × N arrays using n^2 processors

Aim: Implement *nxn* matrix parallel addition, multiplication using CUDA, use shared memory.

Prerequisites:
- Concept of matrix addition, multiplication.

- Basics of CUDA programming

Objectives:
Student should be able to learn parallel programming, CUDA architecture and CUDA processing flow

Theory:
A straightforward matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs

• Leave shared memory usage until later

• Local, register usage

• Thread ID usage

• *P = M * N of size WIDTH x WIDTH*

• One thread handles one element of P

• M and N are loaded WIDTH times from global memory

Matrix Multiplication steps
1. Matrix Data Transfers
2. Simple Host Code in C
3. Host-side Main Program Code
4. Device-side Kernel Function
5. Some Loose Ends

Step 1: Matrix Data Transfers

// Allocate the device memory where we will copy M
to Matrix Md;
Md.width = WIDTH;
Md.height = WIDTH;
Md.pitch = WIDTH;
int size = WIDTH * WIDTH * sizeof(float);
cudaMalloc((void**)&Md.elements, size);
// Copy M from the host to the device
cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyHostToDevice);
// Read M from the device to the host into P cudaMemcpy(P.elements,
Md.elements, size, cudaMemcpyDeviceToHost);
...
// Free device memory
cudaFree(Md.elements);

Step 2: Simple Host Code in C

// Matrix multiplication on the (CPU) host in double precision

// for simplicity, we will assume that all dimensions are equal

```
void MatrixMulOnHost(const Matrix M, const Matrix N, Matrix P)
{
for (int i = 0; i < M.height; ++i)
for (int j = 0; j < N.width; ++j) { double sum = 0;
for (int k = 0; k < M.width; ++k) {
double a = M.elements[i * M.width + k]; double b = N.elements[k * N.width +
j]; sum += a * b;
}
P.elements[i * N.width + j] = sum;
}
```

Multiply Using One Thread Block

• One Block of threads compute matrix P – Each thread computes one element of P

• Each thread – Loads a row of matrix M

– Loads a column of matrix N

– Perform one multiply and addition for each pair of M and N elements

– Compute to off-chip memory access ratio close to 1:1 (not very high)

• Size of matrix limited by the number of threads allowed in a thread block

Step 3: Host-side Main Program Code

```
int main(void) {
// Allocate and initialize the matrices
Matrix M = AllocateMatrix(WIDTH, WIDTH, 1); Matrix N = AllocateMatrix(WIDTH,
WIDTH, 1); Matrix P = AllocateMatrix(WIDTH, WIDTH, 0);
// M * N on the device MatrixMulOnDevice(M, N, P);
// Free matrices FreeMatrix(M)
;
FreeMatrix(N)
;
```

```
FreeMatrix(P);

return 0;

}
```

Host-side code

```
// Matrix multiplication on the device
void MatrixMulOnDevice(const Matrix M, const Matrix N, Matrix P)
{
// Load M and N to the device Matrix Md = AllocateDeviceMatrix(M);
CopyToDeviceMatrix(Md, M);
Matrix Nd = AllocateDeviceMatrix(N); CopyToDeviceMatrix(Nd, N);
// Allocate P on the device
// Setup the execution configuration dim3 dimBlock(WIDTH, WIDTH);
dim3 dimGrid(1, 1);
// Launch the device computation threads! MatrixMulKernel<<<dimGrid,
dimBlock>>>(Md, Nd, Pd);
// Read P from the device CopyFromDeviceMatrix(P, Pd);
// Free device matrices

FreeDeviceMatrix(Md); FreeDeviceMatrix(Nd); FreeDeviceMatrix(Pd);
}
```

Step 4: Device-side Kernel Function

```
// Matrix multiplication kernel – thread specification
global void MatrixMulKernel(Matrix M, Matrix N, Matrix P)
{
// 2D Thread ID
int tx = threadIdx.x; int ty = threadIdx.y;
// Pvalue is used to store the element of the matrix
// that is computed by the thread float Pvalue = 0;
for (int k = 0; k < M.width; ++k)
{
float Melement = M.elements[ty * M.pitch + k]; float Nelement =
Nd.elements[k * N.pitch + tx]; Pvalue += Melement * Nelement;
}
// Write the matrix to device memory;
// each thread writes one element P.elements[ty * P.pitch + tx] = Pvalue;
}
```

Step 5: Some Loose Ends

- Free allocated CUDA memory

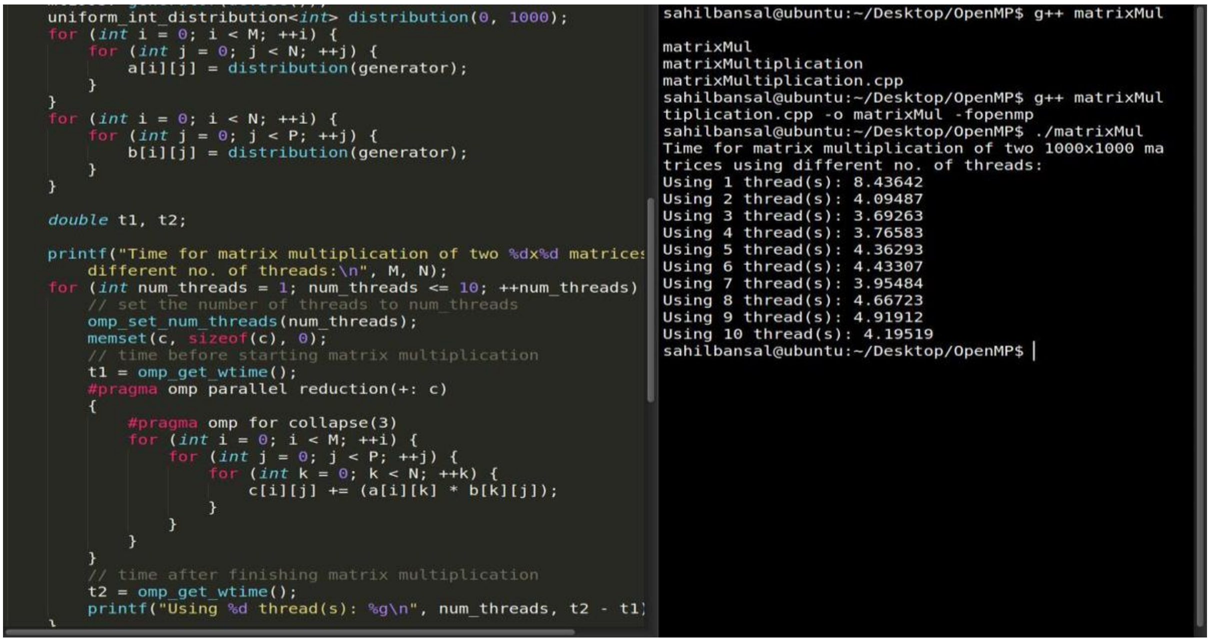
Facilities:

Latest version of 64 Bit Operating Systems, CUDA enabled NVIDIA Graphics card

Input:

Two matrices

Output:



```
    uniform_int_distribution<int> distribution(0, 1000);
    for (int i = 0; i < M; ++i) {
        for (int j = 0; j < N; ++j) {
            a[i][j] = distribution(generator);
        }
    }
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < P; ++j) {
            b[i][j] = distribution(generator);
        }
    }

    double t1, t2;

    printf("Time for matrix multiplication of two %dx%d matrices
        different no. of threads:\n", M, N);
    for (int num_threads = 1; num_threads <= 10; ++num_threads)
        // set the number of threads to num_threads
        omp_set_num_threads(num_threads);
        memset(c, sizeof(c), 0);
        // time before starting matrix multiplication
        t1 = omp_get_wtime();
        #pragma omp parallel reduction(+: c)
        {
            #pragma omp for collapse(3)
            for (int i = 0; i < M; ++i) {
                for (int j = 0; j < P; ++j) {
                    for (int k = 0; k < N; ++k) {
                        c[i][j] += (a[i][k] * b[k][j]);
                    }
                }
            }
        }
        // time after finishing matrix multiplication
        t2 = omp_get_wtime();
        printf("Using %d thread(s): %g\n", num_threads, t2 - t1)
```

```
sahilbansal@ubuntu:~/Desktop/OpenMP$ g++ matrixMul

matrixMul
matrixMultiplication
matrixMultiplication.cpp
sahilbansal@ubuntu:~/Desktop/OpenMP$ g++ matrixMul
tiplication.cpp -o matrixMul -fopenmp
sahilbansal@ubuntu:~/Desktop/OpenMP$ ./matrixMul
Time for matrix multiplication of two 1000x1000 ma
trices using different no. of threads:
Using 1 thread(s): 8.43642
Using 2 thread(s): 4.09487
Using 3 thread(s): 3.69263
Using 4 thread(s): 3.76583
Using 5 thread(s): 4.36293
Using 6 thread(s): 4.43307
Using 7 thread(s): 3.95484
Using 8 thread(s): 4.66723
Using 9 thread(s): 4.91912
Using 10 thread(s): 4.19519
sahilbansal@ubuntu:~/Desktop/OpenMP$
```

Conclusion:

We learned parallel programming with the help of CUDA architecture.