



SRI RAMAKRISHNA ENGINEERING COLLEGE
[Educational Service: SNR Sons Charitable Trust]
[Autonomous Institution, Reaccredited by NAAC with 'A+' Grade]
[Approved by AICTE and Permanently Affiliated to Anna University, Chennai]
[ISO 9001:2015 Certified and all eligible programmes Accredited by NBA]
VATTAMALAIPALAYAM, N.G.G.O. COLONY POST, COIMBATORE – 641 022.



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

BONAFIDE CERTIFICATE

Certified that this is the bonafide record of works done by Mr./Mrs.

_____ **in 20CS257 – HIGH PERFORMANCE**

COMPUTING (Theory Cum Lab) LABORATORY of this Institution for VI

Semester during the Academic Year 2022 – 2023.

Faculty In-Charge

(Dr. M. S. GEETHA DEVASENA, Prof/CSE)

HOD – CSE

(Dr.A.GRACE SELVARANI, Prof/Head)

Date:

Register Number: _____

Submitted for the VI Semester B.E.-CSE Practical Examination held on

_____ **during the Academic Year 2022 – 2023.**

Internal Examiner

External Examiner

INDEX

SL. NO.	DATE	TITLE	PAGE NO.	STAFF'S SIGNATURE
1	28.12.2022	INTRODUCTION TO OPENMP	1	
2	28.12.2022	PARALLELIZATION USING OPENMP WITH/ WITHOUT THREAD	3	
3	28.12.2022	LOOP PARALLELIZATION USING OPENMP	5	
4	03.01.2023	MATRIX MULTIPLICATION ON ROW MAJOR AND COLUMN MAJOR MATRICES	6	
5	10.01.2023	MATRIX MULTIPLICATION USING OPENMP	10	
6	24.01.2023	MAXIMUM AMONG THE ELEMENTS OF N MATRICES	13	
7	14.02.2023	VECTOR ADDITION AND MULTIPLICATION USING CUDA C	15	
8	21.02.2023	VECTOR SQUARING USING CUDA C	20	
9	28.02.2023	THREADS AND BLOCKS USING CUDA C	22	
10	07.03.2023	MATRIX MULTIPLICATION USING CUDA C	25	

Ex. No: 01	INTRODUCTION TO OPENMP
28.12.2022	

AIM:

To do a study on OpenMP programming.

OPENMP

- OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran, on many platforms, instruction-set architectures and operating systems, including Solaris, AIX, FreeBSD, HP-UX, Linux, macOS, and Windows. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.
- An OpenMP program has sections that are sequential and sections that are parallel. In general an OpenMP program starts with a sequential section in which it sets up the environment, initializes the variables, and so on.
- When run, an OpenMP program will use one thread (in the sequential sections), and several threads (in the parallel sections).
- There is one thread that runs from the beginning to the end, and it's called the *master thread*. The parallel sections of the program will cause additional threads to fork. These are called the *slave threads*.

STEPS TO CREATE A PARALLEL PROGRAM

1. Include the header file. Include OpenMP header for our program along with standard header files.

```
#include <omp.h>
```

2. Specify the parallel region.

In OpenMP, we need to mention the region which we are going to make it as parallel using the keyword **pragma omp parallel**. The **pragma omp parallel** is used to fork additional threads to carry out the work enclosed in the parallel.

```
# pragma omp parallel
```

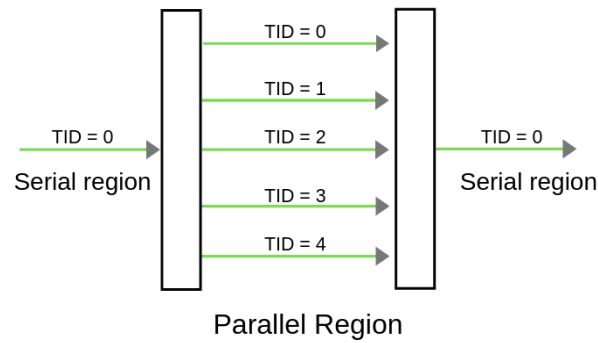
```
{
```

```
//Parallel region code
```

```
}
```

3. Set number of threads.

```
omp_set_num_threads(any number);
```



Once the parallel region ends, all threads will get merged into the master thread.

COMPILATION

1. Create an object file
`gcc -o objectfilename -fopenmp cfilename.c`
2. Execute the program
`./objectfilename`

NOTE:

Order of statement execution in the parallel region won't be the same for all executions.
`omp_get_thread_num()` will return the thread number associated with the thread.

```
#include<omp.h>
#include<stdio.h>
int main(int argc,char* argv[]){
    #pragma omp parallel
    {
        //parallel region
    }
}
```

RESULT:

Thus, the study on OpenMP programming was done successfully.

Ex. No: 02	PARALLELIZATION USING OPENMP WITH/WITHOUT THREAD
28.12.2022	

AIM:

To write a C program to print Hello World with or without thread.

1. WITHOUT THREAD

ALGORITHM:

Step 1: Start

Step 2: Include header files

Step 3: Specify main function

Step 4: Print Hello World by declaring it inside the parallel region

Step 5: Compile the program using gcc command

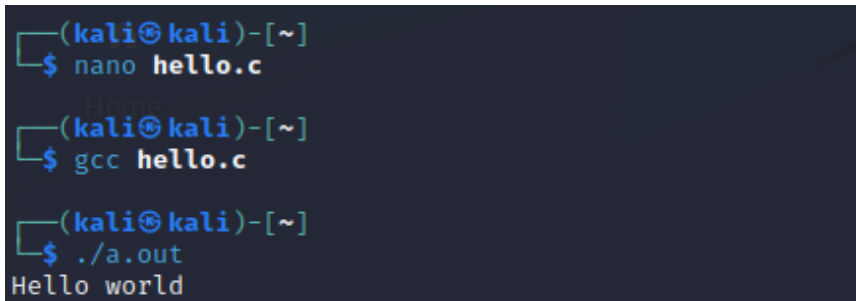
Step 6: Run the object file and display the result

Step 7: Stop

PROGRAM CODE:

```
#include<omp.h>
#include<stdio.h>
#include<stdlib.h>
int main(int argc,char* argv[]){
printf("Hello world \n");
return 0;
}
```

OUTPUT:



```
(kali㉿kali)-[~]
$ nano hello.c

(kali㉿kali)-[~]
$ gcc hello.c

(kali㉿kali)-[~]
$ ./a.out
Hello world
```

2. WITH THREAD

ALGORITHM:

Step 1: Start

Step 2: Include header files

Step 3: Specify parallel region in main function

Step 4: Print Hello World by declaring it inside the parallel region

Step 5: Compile the program using gcc command

Step 6: Run the object file and display the result

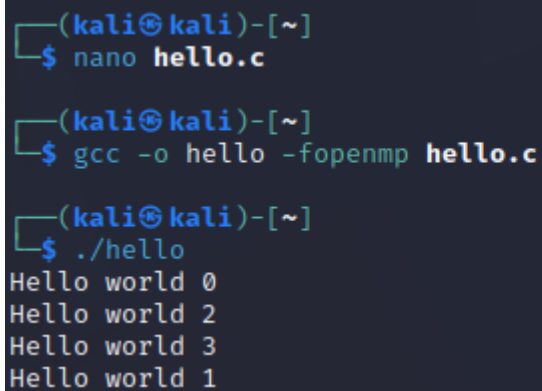
Step 7: Stop

PROGRAM CODE:

```
#include<omp.h>
#include<stdio.h>
#include<stdlib.h>
//#define OMP_NUM_THREADS=4
int main(int argc,char* argv[]){
omp_set_num_threads(4);
#pragma omp parallel
{
printf("Hello world %d\n",omp_get_thread_num());

}
return 0;
}
```

OUTPUT:

A terminal window with a dark background and light blue text. It shows three commands being executed in a Kali Linux environment. The first command is 'nano hello.c'. The second command is 'gcc -o hello -fopenmp hello.c'. The third command is './hello', which produces four lines of output: 'Hello world 0', 'Hello world 2', 'Hello world 3', and 'Hello world 1' in a different order.

```
(kali㉿kali)-[~]
└─$ nano hello.c

(kali㉿kali)-[~]
└─$ gcc -o hello -fopenmp hello.c

(kali㉿kali)-[~]
└─$ ./hello
Hello world 0
Hello world 2
Hello world 3
Hello world 1
```

RESULT:

Thus, the program to print Hello World with/ without thread was executed successfully and the output was verified.

Ex. No: 03	LOOP PARALLELIZATION USING OPENMP
28.12.2022	

AIM:

To write a C program to execute for loop parallelly using OpenMp.

ALGORITHM:

Step 1: Start

Step 2: Include header files

Step 3: Specify main function with a parallel region

Step 4: Iterate a for loop from 0 to 10 and print thread number.

Step 5: Compile the program using gcc command

Step 6: Run the object file and display the result

Step 7: Stop

PROGRAM:

```
#include<omp.h>
#include<stdio.h>
int main(int argc,char* argv[]){
#pragma omp parallel for num_threads(4)
for(int i=0;i<10;i++)
printf("The thread %d executes i=%d\n",omp_get_thread_num(),i);
return 0;}
```

OUTPUT:

```
~$ gcc -o l -fopenmp loop.c
~$ ./l
The thread 0 executes i=0
The thread 0 executes i=1
The thread 0 executes i=2
The thread 2 executes i=6
The thread 2 executes i=7
The thread 3 executes i=8
The thread 3 executes i=9
The thread 1 executes i=3
The thread 1 executes i=4
The thread 1 executes i=5
```

RESULT:

Thus, the program to execute for loop parallelly was done successfully.

Ex. No: 04	MATRIX MULTIPLICATION ON ROW MAJOR AND COLUMN MAJOR MATRICES
03.01.2023	

AIM:

To write a C program to run matrix multiplication on row major and column major matrices using OpenMP parallelly.

ALGORITHM:

Step 1: Start

Step 2: Include header files

Step 3: Specify main function with a parallel region

Step 4: Declare column major and row major matrices and give random inputs to it.

Step 5: Share matrix elements and keep dimensions private.

Step 6: Calculate the elapsed time for both row major and column major matrices.

Step 7: Stop

ROW MAJOR MATRICES:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <sys/time.h>
#define N 600

int A[N][N];
int B[N][N][N];
int C[N][N][N];
int main()
{
    int i,j,k;
    struct timeval tv1, tv2;
    struct timezone tz;
    double elapsed;
    omp_set_num_threads(4);
    for (i= 0; i< N; i++){
        for (j= 0; j< N; j++)
        {
            for (k= 0; k< N; k++)
            {
```



```

        A[i][j] = 2;
        B[i][j][k] = 4;
    }
}

gettimeofday(&tv1, &tz);
#pragma omp parallel for private(i,j,k) shared(A,B,C)
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        for (k = 0; k < N; k++) {
            C[i][j][k] += A[i][k] * B[k][j][i];
        }
    }
}

gettimeofday(&tv2, &tz);
elapsed = (double) (tv2.tv_sec-tv1.tv_sec) + (double) (tv2.tv_usec-tv1.tv_usec) * 1.e-6;
printf("Elapsed time = %f seconds.\n", elapsed);

/*for (i= 0; i< N; i++)
{
    for (j= 0; j< N; j++)
    {
        printf("%d\t",C[i][j]);
    }
    printf("\n");
}*/
}

```

OUTPUT:

```

(kali㉿kali)-[~]
$ nano rowmajor.c

(kali㉿kali)-[~]
$ gcc -o a -fopenmp rowmajor.c

(kali㉿kali)-[~]
$ ./a
Elapsed time = 13.098900 seconds.\n

```

COLUMN MAJOR MATRICES:

```
#include <pthread.h>
```

```

#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <sys/time.h>
int main(int argc, const char* argv[]) {
    int rows=2;
    int cols=3;
    int i,j,k;
    struct timeval tv1, tv2;
    struct timezone tz;
    double elapsed;
    omp_set_num_threads(4);

    int A[rows*rows];
    int B[rows*cols];
    int res[rows*cols];
    A[0]=1;
    A[1]=2;
    A[2]=3;
    A[3]=4;
    B[0]=5;
    B[1]=6;
    B[2]=2;
    B[3]=3;
    B[4]=1;
    B[5]=7;
    gettimeofday(&tv1, &tz);
    #pragma omp parallel for private(i,j,k) shared(A,B)
    //multiplication as column major
    for (int i=0;i<rows;i++){
        for (int j=0;j<cols-1;j++){
            res[i+j*rows]=0;
            for (int k=0;k<rows;k++){
                res[i+j*rows]+=A[i+k*rows]*B[k+j*cols];
            }
        }
    }
    gettimeofday(&tv2, &tz);
    elapsed = (double) (tv2.tv_sec-tv1.tv_sec) + (double) (tv2.tv_usec-tv1.tv_usec) * 1.e-6;
    printf("Elapsed time = %f seconds.\n", elapsed);
    for (int i=0;i<rows*(cols-1);i++){
        printf("/nB[%d]=%d",i,res[i]);
    }
}

```

```
}  
return 0;  
}
```

OUTPUT:

```
(kali㉿kali)-[~]  
$ nano columnmajor.c  
  
(kali㉿kali)-[~]  
$ gcc -o a -fopenmp columnmajor.c  
  
(kali㉿kali)-[~]  
$ ./a  
10 2 5  
34 4 6  
6 2 3  
10 4 1  
5 1 5  
23 3 6  
3 1 3  
6 3 1  
Elapsed time = 0.003632 seconds.\n/nB[0]=23/nB[1]=34/nB[2]=6/nB[3]=10
```

RESULT:

Thus, the C program to run matrix multiplication on row major and column major matrices using OpenMP parallelly was executed successfully and the output was verified.

Ex. No: 05	MATRIX MULTIPLICATION USING OPENMP
10.01.2023	

AIM:

To write a C program to run matrix multiplication using OpenMP parallelly.

ALGORITHM:

Step 1: Start

Step 2: Include header files

Step 3: Specify main function with a parallel region

Step 4: Declare matrices and give random inputs to it.

Step 5: Share matrix elements and keep dimensions private.

Step 6: Calculate the elapsed time and analyze for both with or without threads.

Step 7: Stop

PROGRAM CODE:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <sys/time.h>
#define N 100
int A[N][N];
int B[N][N];
int C[N][N];
int main()
{
    int i,j,k;
    struct timeval tv1, tv2;
    struct timezone tz;
    double elapsed;
    omp_set_num_threads(100);
    for (i= 0; i< N; i++){
        for (j= 0; j< N; j++){
            A[i][j] = 2;
            B[i][j] = 2;
        }
    }
    gettimeofday(&tv1, &tz);
```

```

#pragma omp parallel for private(i,j,k) shared(A,B,C)
for (i = 0; i < N; ++i) {
    for (j = 0; j < N; ++j) {
        for (k = 0; k < N; ++k) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
gettimeofday(&tv2, &tz);
elapsed = (double) (tv2.tv_sec-tv1.tv_sec) + (double) (tv2.tv_usec-tv1.tv_usec) * 1.e-6;
printf("Elapsed time = %f seconds.\n", elapsed);
}

```

OUTPUT:

```

(kali㉿kali)-[~]
$ nano mul.c

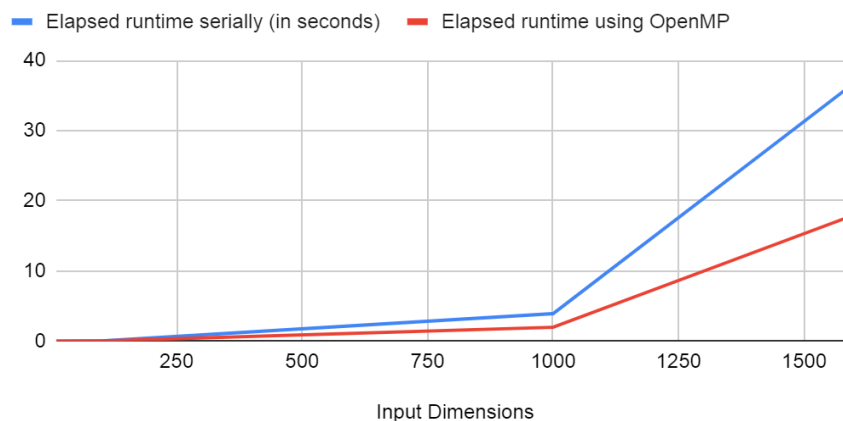
(kali㉿kali)-[~]
$ gcc -o a -fopenmp mul.c

(kali㉿kali)-[~]
$ ./a
Elapsed time = 17.975703 seconds.\n

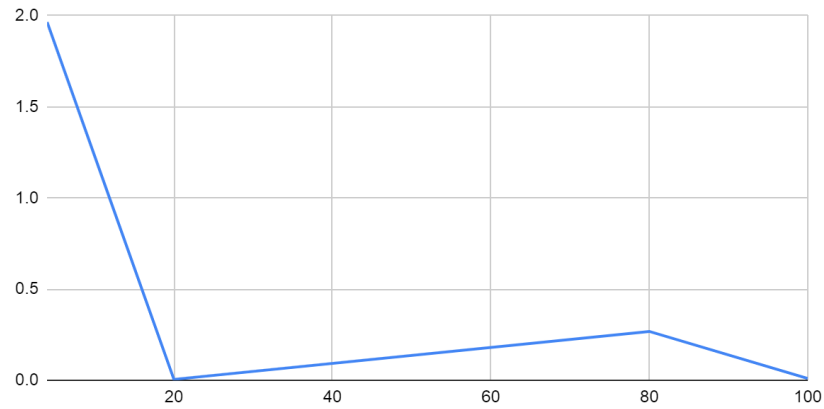
```

Input Dimensions	Elapsed runtime serially (in seconds)	Elapsed runtime using OpenMP (in seconds)
10	0.000019	0.0000931
100	0.027293	0.029195
1000	3.898065	1.964642
1600	36.779024	17.975703

Number of threads=4



Input Dimensions	No. of threads	Elapsed runtime using OpenMP (in seconds)
1000	4	1.964642
1000	20	0.004737
1000	80	0.026771
1000	100	0.009510



OBSERVATIONS:

- There is always a hardware limitation due to fixed cores.
- With increase in the number of dimensions parallel computation using threads becomes efficient.

RESULT:

Thus, the analysis of matrix multiplication using OpenMP was done successfully and the output was verified.

Ex. No: 06	MAXIMUM AMONG THE ELEMENTS OF N MATRICES
24.01.2023	

AIM:

To write a C program to find maximum among the elements of n matrices using OpenMP parallelly.

ALGORITHM:

Step 1: Start

Step 2: Include header files

Step 3: Specify main function with a parallel region

Step 4: Declare matrices and give random inputs to it.

Step 5: Share matrix elements and keep dimensions private.

Step 6: Calculate the elapsed time and print the maximum element in n matrices.

Step 7: Stop

PROGRAM CODE:

```
#include <stdio.h>
#include<omp.h>
#include<stdlib.h>
#include <sys/time.h>
#define N 4
#define M 4
#define num 100
// Driver code
int main(int argc, char* argv[])
{
    // matrix
    int mat[num][N][M];
    struct timeval tv1, tv2;
    struct timezone tz;
    double elapsed;
    omp_set_num_threads(4);
    for(int i=0;i<num;i++){
        for(int j=0;j<N;j++){
            for(int k=0;k<M;k++){
                int temp = rand();
                mat[i][j][k]= temp % 100;}
        }}
    gettimeofday(&tv1,&tz);
```

```

int maxElement[num];
for(int i=0;i<num;i++){
    maxElement[i] = 0;}
#pragma omp parallel for private(i,j,k) shared(mat)
for (int k = 0; k < num; k++) {
    for (int j = 0; j < N; j++) {
        for(int i=0;i<M;i++){
            if (mat[k][i][j] > maxElement[k]) {
                maxElement[k] = mat[k][i][j];
            }
        }
    }
}
int max=maxElement[0];
for(int i=1;i<num;i++){
    if(maxElement[i]>max){
        max=maxElement[i];
    }
}
printf("%d",max);
gettimeofday(&tv2, &tz);
elapsed = (double) (tv2.tv_sec-tv1.tv_sec) + (double) (tv2.tv_usec-tv1.tv_usec) * 1.e-6;
printf("Elapsed time = %f seconds.\n", elapsed);
return 0;
}

```

OUTPUT:



```

(kali㉿kali)-[~]
$ nano max.c
(kali㉿kali)-[~]
$ gcc -o a -fopenmp max.c
(kali㉿kali)-[~]
$ ./a
99
Elapsed time = 0.000513 seconds.\n

```

RESULT:

Thus, the C program to find maximum among the elements of n matrices using OpenMP parallelly was executed successfully and the output was verified.

Ex. No: 07	VECTOR ADDITION AND MULTIPLICATION USING CUDA C
14.02.2023	

AIM:

To write a Cuda C program to perform vector addition and vector multiplication.

CUDA C:

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA for use with their GPUs (graphics processing units). CUDA allows developers to write programs in C, C++, and Fortran that can run on the GPU for parallel processing.

To program in CUDA C, you need to have a GPU that supports CUDA. You also need to have the CUDA toolkit installed on your machine, which includes the CUDA compiler (NVCC) and the CUDA runtime libraries. In CUDA C, you write both host code and device code. Host code runs on the CPU and is responsible for managing the memory and launching the device code. Device code runs on the GPU and performs the parallel computations.

To write device code, you use CUDA-specific keywords and syntax, such as global functions, which are kernel functions that can be called from host code and run on the GPU, and threadIdx and blockIdx, which are built-in variables that represent the thread index and block index in the CUDA execution grid. To launch a kernel function, you specify the number of blocks and threads to use in the execution grid. The number of threads in each block should be a multiple of the warp size, which is the number of threads that can be executed in parallel on the GPU.

CUDA also provides features for managing memory on the GPU, such as CUDA malloc for allocating memory and cudaMemcpy for copying data between the CPU and GPU.

ALGORITHM:

Step 1: Define the code that will run on the device in a separate function, called the kernel function.

Step 2: In the main program running on the host's CPU:

Allocate memory on the host for the data arrays.

Initialize the data arrays in the host's memory.

Allocate separate memory on GPU device for the data arrays.

Copy data arrays from the host memory to the GPU device memory.

Step 3: On the GPU device, execute the kernel function that computes new data values given the original arrays. Specify how many blocks and threads per block to use for this computation.

Step 4: After the kernel function completes, copy the computed values from the GPU device memory back to the host's memory.

Step 5: Display the results

VECTOR ADDITION

PROGRAM CODE:

```
#include "stdio.h"

#include <iostream>

#include <cuda.h>

#include <cuda_runtime.h>

// Defining number of elements in Array
#define N 5

// Defining Kernel function for vector addition
__global__ void gpuAdd(int *d_a, int *d_b, int *d_c)
{
    // Getting block index of current kernel
    int tid = blockIdx.x; // handle the data at this index
    if (tid < N)
        d_c[tid] = d_a[tid] + d_b[tid];
}

int main(void)
{
    // Defining host arrays
    int h_a[N], h_b[N], h_c[N];

    // Defining device pointers
    int *d_a, *d_b, *d_c;

    // allocate the memory
    cudaMalloc((void**)&d_a, N * sizeof(int));
    cudaMalloc((void**)&d_b, N * sizeof(int));
    cudaMalloc((void**)&d_c, N * sizeof(int));

    // Initializing Arrays
    for (int i = 0; i < N; i++) {
        h_a[i] = 2*i*i;
```

```

    h_b[i] = i ;
}

// Copy input arrays from host to device memory
cudaMemcpy(d_a, h_a, N * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, N * sizeof(int), cudaMemcpyHostToDevice);
// Calling kernels with N blocks and one thread per block, passing
// device pointers as parameters
gpuAdd <<<N, 1 >>>(d_a, d_b, d_c);
// Copy result back to host memory from device memory
cudaMemcpy(h_c, d_c, N * sizeof(int), cudaMemcpyDeviceToHost);
printf("Vector addition on GPU \n");
// Printing result on console
for (int i = 0; i < N; i++) {
    printf("The sum of %d element is %d + %d = %d\n",
        i, h_a[i], h_b[i], h_c[i]);
}
// Free up memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
return 0;
}

```

OUTPUT:

```

Vector addition on GPU
The sum of 0 element is 0 + 0 = 0
The sum of 1 element is 2 + 1 = 3
The sum of 2 element is 8 + 2 = 10
The sum of 3 element is 18 + 3 = 21
The sum of 4 element is 32 + 4 = 36

```

VECTOR MULTIPLICATION

PROGRAM CODE:

```
#include<stdio.h>
#include<cuda.h>
__global__ void VecMul(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i < N)
        C[i] = A[i]*B[i];
}
int main()
{
    int i, N = 10;
    size_t size = N * sizeof(float);

    // Allocating host and initializing
    float A[N],B[N],C[N];
    for(i=0;i<N;i++) {
        A[i] = B[i] = i;
    }

    // Allocating device and copying to device
    float *d_A, *d_B, *d_C;
    cudaMalloc((void **)&d_A, size);
    cudaMalloc((void **)&d_B, size);
    cudaMalloc((void **)&d_C, size);

    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

    // Invoking kernel
    int threadsPerBlock = 8;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

    VecMul<<<blocksPerGrid, threadsPerBlock>>>>(d_A, d_B, d_C, N);

    // Copy result from device to host
    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

    for(i=0;i<N;i++)
        printf("%f\n", C[i]);
}
```

OUTPUT:

```
0.000000
1.000000
4.000000
9.000000
16.000000
25.000000
36.000000
49.000000
64.000000
81.000000
```

RESULT:

Thus, the Cuda C program to demonstrate vector addition and vector multiplication was executed successfully and the output was verified.

Ex. No: 08	VECTOR SQUARING USING CUDA C
21.02.2023	

AIM:

To write a Cuda C program to perform vector squaring.

ALGORITHM:

Step 1: Define the code that will run on the device in a separate function, called the kernel function.

Step 2: In the main program running on the host's CPU:

Allocate memory on the host for the data arrays.

Initialize the data arrays in the host's memory.

Allocate separate memory on GPU device for the data arrays.

Copy data arrays from the host memory to the GPU device memory.

Step 3: On the GPU device, execute the kernel function that computes new data values given the original arrays. Specify how many blocks and threads per block to use for this computation.

Step 4: After the kernel function completes, copy the computed values from the GPU device memory back to the host's memory.

Step 5: Display the results

PROGRAM CODE:

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>

// Defining number of elements in Array
#define N 5

// Kernel function for squaring number
__global__ void gpuSquare(float *d_in, float *d_out)
{
    // Getting thread index for current kernel
    int tid = threadIdx.x; // handle the data at this index
    float temp = d_in[tid];
    d_out[tid] = temp*temp;
}

int main(void)
{
    // Defining Arrays for host
    float h_in[N], h_out[N];
    float *d_in, *d_out;
    // allocate the memory on the cpu
    cudaMalloc((void**)&d_in, N * sizeof(float));
```

```

cudaMalloc((void**)&d_out, N * sizeof(float));
// Initializing Array
for (int i = 0; i < N; i++) {
    h_in[i] = i;
}
// Copy Array from host to device
cudaMemcpy(d_in, h_in, N * sizeof(float), cudaMemcpyHostToDevice);
// Calling square kernel with one block and N threads per block
gpuSquare <<<1, N >>>(d_in, d_out);
// Coping result back to host from device memory
cudaMemcpy(h_out, d_out, N * sizeof(float), cudaMemcpyDeviceToHost);
// Printing result on console
printf("Square of Number on GPU \n");
for (int i = 0; i < N; i++) {
    printf("The square of %f is %f\n", h_in[i], h_out[i]);
}
// Free up memory
cudaFree(d_in);
cudaFree(d_out);
return 0;
}

```

OUTPUT:

```

Square of Number on GPU
The square of 0.000000 is 0.000000
The square of 1.000000 is 1.000000
The square of 2.000000 is 4.000000
The square of 3.000000 is 9.000000
The square of 4.000000 is 16.000000

```

RESULT:

Thus, the Cuda C program to demonstrate vector squaring was executed successfully and the output was verified.

Ex. No: 9	THREADS AND BLOCKS USING CUDA C
28.02.2023	

AIM:

To write a Cuda C program to perform parallel addition using threads and blocks.

ALGORITHM:

Step 1: Define the size of the arrays to be added and allocate memory for them on the device using the cudaMalloc() function.

Step 2: Initialize the arrays with random values using the curand() function.

Step 3: Define the number of threads per block and the number of blocks per grid using the dim3 data type.

Step 4: Launch the kernel function that will perform the addition in parallel. This function should take in the two arrays, the size of the arrays, and any other necessary parameters.

Step 5: Inside the kernel function, use the threadIdx.x, blockIdx.x, and blockDim.x variables to calculate the index of the current thread and block.

Step 6: Use a loop to add the corresponding elements of the two arrays using the calculated indices.

Step 7: Synchronize the threads using the __syncthreads() function to ensure that all threads have finished before copying the result back to the host.

Step 8: Copy the result array from the device back to the host using the cudaMemcpy() function.

Step 9: Display the results

PARALLEL ADDITION USING BLOCKS

PROGRAM CODE:

```
#include<stdio.h>
#define N 512
__global__ void add( int* a, int* b, int* c ) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
int main( void ) {
    int *a, *b, *c; // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c; // device copies of a, b, c
    int size = N *sizeof( int); // we need space for 512 integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );
    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( size );
    *a = 2;
    *b = 7;
    // copy inputs to device
```



```

cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice);
// launch add() kernel with N parallel blocks
add<<< N, 1 >>>( dev_a, dev_b, dev_c);
// copy device result back to host copy of c
cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost);

printf("SUM IS: %d", *c);
free( a ); free( b ); free( c );
cudaFree( dev_a);
cudaFree( dev_b);
cudaFree( dev_c);
return 0;
}

```

OUTPUT:

```
SUM IS: 9
```

PARALLEL ADDITION USING THREADS

PROGRAM CODE:

```

#include<stdio.h>
global__ voidadd( int* a, int* b, int* c ) {
c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
#define N 512
int main( void ) {
int *a, *b, *c; //host copies of a, b, c
int *dev_a, *dev_b, *dev_c; //device copies of a, b, c
int size = N * sizeof( int); //we need space for 512 integers
// allocate device copies of a, b, c
cudaMalloc( (void**)&dev_a, size );
cudaMalloc( (void**)&dev_b, size );
cudaMalloc( (void**)&dev_c, size );
a = (int*)malloc( size );
b = (int*)malloc( size );
c = (int*)malloc( size );
*a = 2;
*b = 7;
// copy inputs to device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice);

```

```
// launch add() kernel with N
add<<<1,N>>>( dev_a, dev_b, dev_c);
// copy device result back to host copy of c
cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost);

printf("%d", *c);
free( a ); free( b ); free( c );
cudaFree( dev_a);
cudaFree( dev_b);
cudaFree( dev_c);
return 0;
}
```

OUTPUT:

SUM IS : 9

RESULT:

Thus, the Cuda C program to demonstrate parallel addition using thread and blocks was executed successfully and the output was verified.

Ex. No: 10	MATRIX MULTIPLICATION USING CUDA C
07.03.2023	

AIM:

To write a Cuda C program to perform matrix multiplication.

ALGORITHM:

Step 1: Define the size of the matrices to be multiplied and allocate memory for them on the device using the cudaMalloc() function.

Step 2: Initialize the matrices with random values using the curand() function.

Step 3: Define the number of threads per block and the number of blocks per grid using the dim3 data type.

Step 4: Launch the kernel function that will perform the matrix multiplication in parallel. This function should take in the two matrices, the size of the matrices, and any other necessary parameters.

Step 5: Inside the kernel function, use the threadIdx.x, blockIdx.x, and blockDim.x variables to calculate the index of the current thread and block.

Step 6: Use nested loops to perform the matrix multiplication using the calculated indices.

Step 7: Synchronize the threads using the __syncthreads() function to ensure that all threads have finished before copying the result back to the host.

Step 8: Copy the result matrix from the device back to the host using the cudaMemcpy() function.

PROGRAM CODE:

```
#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#define TILE_WIDTH 2
/*matrix multiplication kernels*/
//non shared
__global__ void
MatrixMul( float *Md , float *Nd , float *Pd , const int WIDTH )
{
    // calculate thread id
    unsigned int col = TILE_WIDTH*blockIdx.x + threadIdx.x ;
    unsigned int row = TILE_WIDTH*blockIdx.y + threadIdx.y ;
    for (int k = 0 ; k<WIDTH ; k++ )
    {
        Pd[row*WIDTH + col]+= Md[row * WIDTH + k ] * Nd[ k * WIDTH + col ] ;
    }
}
// shared
__global__ void MatrixMulSh( float *Md , float *Nd , float *Pd , const int WIDTH )
{
    //Taking shared array to break the MATrix in Tile widht and fatch 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 synchronizeing 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 synchronizeing the threads
    }
}
// main routine
int main ()
{
    const int WIDTH = 100 ;
    struct timeval tv1, tv2;
    struct timezone tz;
    double elapsed;

    float array1_h[WIDTH][WIDTH] ,array2_h[WIDTH][WIDTH],
        result_array_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] = 1 ;
            array2_h[i][j] = 2 ;
        }
    }
    gettimeofday(&tv1, &tz);
    //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) ) ;
    //calling kernal
    dim3 dimGrid ( WIDTH/TILE_WIDTH , WIDTH/TILE_WIDTH , 1 ) ;
    dim3 dimBlock( TILE_WIDTH, TILE_WIDTH, 1 ) ;
    // Change if 0 to if 1 for running non shared code and make if 0 for shared memory code

```

