ASSIGNMENT NO.01

TITLE: STUDY OF OPENMP AND CUDA FRAMEWORK FOR DESIGNING PARALLEL AGLORITHMS

PROBLEM STATEMENT:

A) **Implement Parallel Reduction using Min, Max, Sum and Average operations**.

B) **Write a CUDA program that, given an N-element vector, find following things :**

1) The maximum element in the vector

2) The minimum element in the vector

3) The arithmetic mean of the vector

4) The standard deviation of the values in the vector.

Test for input N and generate a randomized vector V of length N (N should be large). The program should generate output as the two computed maximum values as well as the time taken to find each value.

OBJECTIVES:

- To study about OpenMP and CUDA.
- To learn how to perform reduction operations such as min, max, mean, sum, average in OpenMP as well as CUDA.

SOFTWARE & HARDWARE REQUIREMENTS:

1. Windows Operating System/64-bit Open source Linux or its derivative

2. GCC Compiler

3. Graphics Card

4. NVIDIA

THEORY:

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran,[3] on most platforms, instruction set architectures and operating systems, including Solaris, AIX, HP-UX, Linux, macOS, and Windows. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behaviour.

OpenMP is managed by the non-profit technology consortium *OpenMP Architecture Review Board* (or *OpenMP ARB*), jointly defined by a group of major computer hardware and software vendors, including AMD, IBM, Intel, Cray, HP, Fujitsu, Nvidia, NEC, Red Hat, Texas Instruments, Oracle Corporation, and more.

OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer.

OpenMP is an implementation of multithreading, a method of parallelizing whereby a *master* thread (a series of instructions executed consecutively) *forks* a specified number of *slave* threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors. *OpenMP* is an alternative to writing shared memory parallel programs

The section of code that is meant to run in parallel is marked accordingly, with a compiler directive that will cause the threads to form before the section is executed. Each thread

has an *id* attached to it which can be obtained using a function (called omp_get_thread_num()). The thread id is an integer, and the master thread has an id of *0*. After the execution of the parallelized code, the threads *join* back into the master thread, which continues onward to the end of the program.

By default, each thread executes the parallelized section of code independently. *Work-sharing constructs* can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both task parallelism and data parallelism can be achieved using OpenMP in this way.

The runtime environment allocates threads to processors depending on usage, machine load and other factors. The runtime environment can assign the number of threads based on environment variables, or the code can do so using functions. The OpenMP functions are included in a header file labelled omp.h in C/C++.

In C/C++, OpenMP uses #pragmas.

Pros:

- Portable multithreading code (in C/C++ and other languages, one typically has to call platform-specific primitives in order to get multithreading).
- Simple: need not deal with message passing as MPI does.
- Data layout and decomposition is handled automatically by directives.
- Scalability comparable to MPI on shared-memory systems.
- Incremental parallelism: can work on one part of the program at one time,
- no dramatic change to code is needed.

  Unified code for both serial and parallel applications: OpenMP constructs are treated as comments when sequential compilers are used.

- Both coarse-grained and fine-grained parallelism are possible and can be used on various accelerators such as GPGPU.

Cons:

- Risk of introducing difficult to debug synchronization bugs and race conditions
- Requires a compiler that supports OpenMP.
- Scalability is limited by memory architecture.
  No support for compare-and-swap.
- Reliable error handling is missing.
- Lacks fine-grained mechanisms to control thread-processor mapping.

  High chance of accidentally writing false sharing code.

OpenMP reduction operations

OpenMP reduction operations can be used for simple cases, such as incrementing a shared numeric variable or the summation of an array into a shared numeric variable. To implement a reduction operation, add the reduction clause within a parallel region to instruct the compiler to perform the summation operation in parallel using the specified operation and variable. The reduction operations that can be performed in openmp are min, max, sum, avg, etc.

The parallel C/C++ code after adding #include <omp.h> and #pragma omp parallel for reduction:

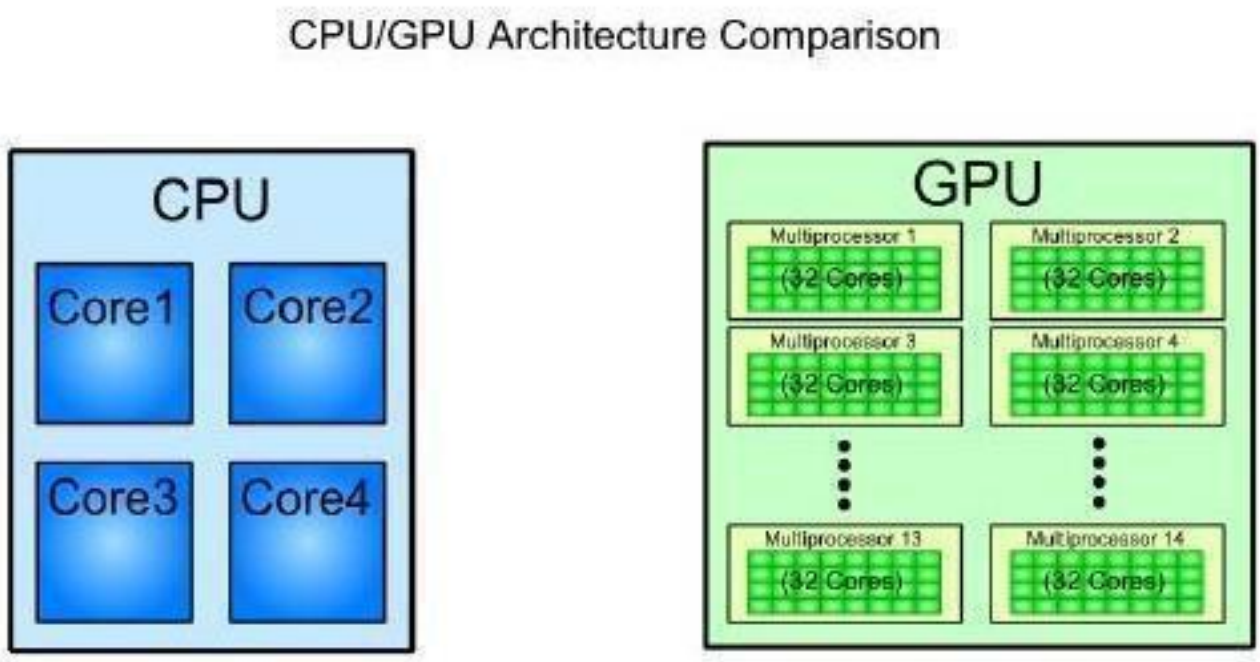
```
#include <omp.h> //prevents a load-time problem with a .dll not being found
  int i, n=500000;
 float *array, total=0.0;
    ...
   #pragma omp parallel for reduction(+:total)

      for (i=0; i <n ; ++i
      {
        total+ = array[i];
      }
```

CUDA(Compute Unified Device Architecture)

CUDA is a parallel computing platform and an API model that was developed by Nvidia. Using CUDA, one can utilize the power of Nvidia GPUs to perform general computing tasks, such as multiplying matrices and performing other linear algebra operations, instead of just doing graphical calculations. Using CUDA, developers can now harness the potential of the GPU for general purpose computing (GPGPU).

A CPU is very good when it comes to executing a single instruction on a single datum, but not so much when it comes to processing a large chunk of data. A CPU has a larger instruction set than a GPU, a complex ALU, a better branch prediction logic, and a more sophisticated caching/pipeline schemes. The instruction cycles are also a lot faster. The other paradigm is many-core processors that are designed to operate on large chunks of data, in which CPUs prove inefficient. A GPU comprises many cores (that almost double each passing year), and each core runs at a clock speed significantly slower than a CPU's clock. GPUs focus on execution throughput of massively-parallel programs.

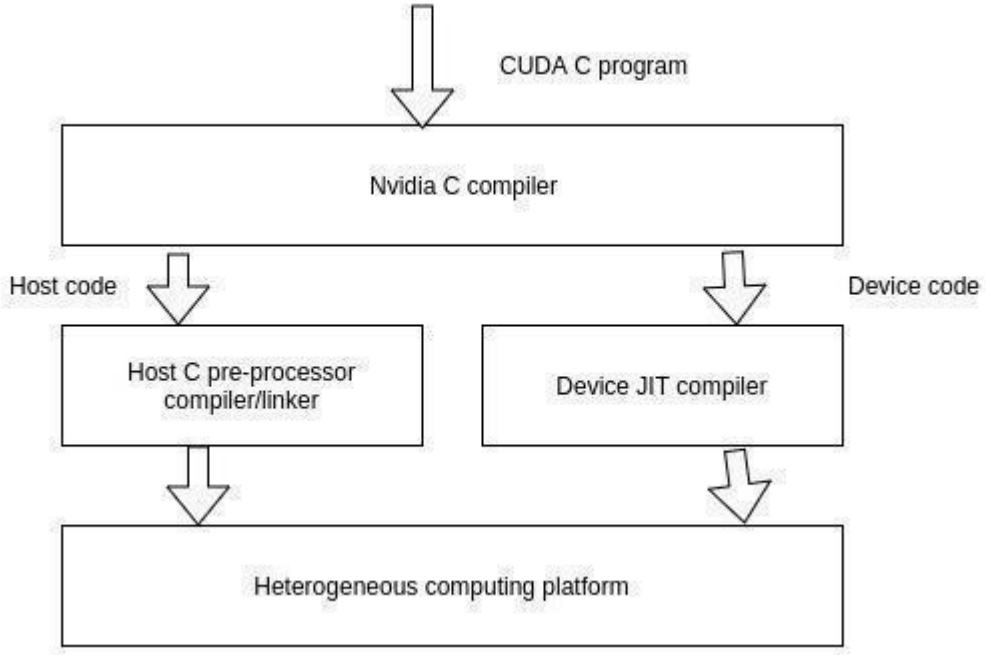


PROGRAM STRUCTURE IN CUDA-

A typical CUDA program has code intended both for the GPU and the CPU. By default, a traditional C program is a CUDA program with only the host code. The CPU is referred to as the host, and the GPU is referred to as the device. Whereas the host code can be compiled by a traditional C compiler as the GCC, the device code needs a special compiler to understand

the api functions that are used. For Nvidia GPUs, the compiler is called the NVCC (Nvidia C Compiler).

The device code runs on the GPU, and the host code runs on the CPU. The NVCC processes a CUDA program, and separates the host code from the device code. To accomplish this, special CUDA keywords are looked for. The code intended to run of the GPU (device code) is marked with special CUDA keywords for labelling data-parallel functions, called 'Kernels'. The device code is further compiled by the NVCC and executed on the GPU.



Flow of Execution of CUDA Program

The following keywords are used while declaring a CUDA function. As an example, while declaring the kernel, we have to use the **__global__** keyword. This provides a hint to the compiler that this function will be executed on the device and can be called from the host.

	EXECUTED ON THE −	CALLABLE FROM −
__device__ float function()	GPU (device)	CPU (host)
__global__ void function()	CPU (host)	GPU (device)
__host__float function()	GPU (device)	GPU (device)

The reduction operations in CUDA can be done similar to OpenMP. The lines of code for finding min or max value i.e. the computational part are included in the GPU. The values are accepted in the main() function. After allocating the memory space in GPU using cudaMalloc(), the accepted values are copied to GPU memory using cudamemcpy() function. The respective function is then called and the control is passed to the GPU. After the value is returned by the GPU, the memory is released.

Note that only computations are performed in GPU. Input Output are not processed in GPU.

Similar process can be carried out for various reduction operations.

Compilation of CUDA programs:

$nvcc filename.cu

$./a.out

CONCLUSION: