Experiment 3

Title	Implement Min, Max, Sum and Average operations using Parallel Reduction.
Problem statement	Implement Parallel Reduction using Min, Max, Sum and Average operations. b) Write a CUDA program that given an N-element vector, find- • The maximum element in the vector • The minimum element in the vector • The arithmetic mean of the vector • 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.
Prerequisite	64 bit Open source Linux or its derivative, Programming Languages: C/C++,CUDA Tutorials
CO mapped	3
Programming Tools	Ubuntu 14.04, GPU Driver 352.68, CUDA Toolkit 8.0

Objective: To study and implementation of directive based parallel programming model. To study and implement the operations on vector, generate o/p as two computed max values as wellas time taken to find each value

Outcome: Students will understand the implementation of sequential program augmented withcompiler directives to specify parallelism. Students will understand the implementation of operations on vector, generate o/p as two computed max with respect to time.

Pre-requisites: 64-bit Open source Linux or its derivative, Programming Languages: C/C++,CUDA Tutorials.

Hardware Specification: x86_64 bit, 2 – 2/4 GB DDR RAM, 80 - 500 GB SATA HD, 1GBNIDIA TITAN X Graphics Card.

Software Specification: Ubuntu 14.04, GPU Driver 352.68, CUDA Toolkit 8.0, CUDNNLibrary v5.0

Theory:

Open MP:

OpenMP is a set of C/C++ pragmas (or FORTRAN equivalents) which provide the programmer a high-level front-end interface which get translated as calls to threads (or other similar entities). The key phrase here is "higher-level"; the goal is to better enable the programmer to "think parallel," alleviating him/her of the burden and distraction of dealing withsetting up and coordinating threads. For example, the OpenMP directive.

Parallel Programming:

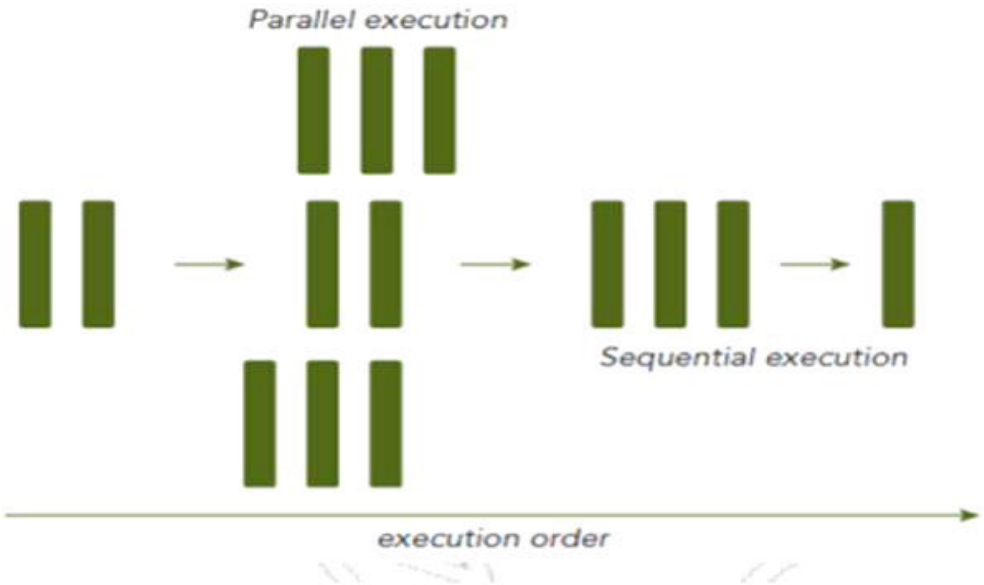
There are two fundamental types of parallelism in applications:

➤Task parallelism

➤Data parallelism

Task parallelism arises when there are many tasks or functions that can be operated independently and largely in parallel. Task parallelism focuses on distributing functions across multiple cores.

Data parallelism arises when there are many data items that can be operated on at the same time.Data parallelism focuses on distributing the data across multiple cores.



CUDA:

CUDA programming is especially well-suited to address problems that can be expressedas data-parallel computations. Any applications that process large data sets can use a data- parallel model to speed up the computations. Data-parallel processing maps data elements to parallel threads. The first step in designing a data parallel program is to partition data across threads, with each thread working on a portion of the data.

CUDA Architecture:

A heterogeneous application consists of two parts:

➤Host code

➤Device code

Host code runs on CPUs and device code runs on GPUs. An application executing on a heterogeneous platform is typically initialized by the CPU. The CPU code is responsible for managing the environment, code, and data for the device before loading compute-intensive taskson the device. With computational intensive applications, program sections often exhibit a rich amount of data parallelism. GPUs are used to accelerate the execution of this portion of data parallelism. When a hardware component that is physically separate from the CPU is used to accelerate computationally intensive sections of an application, it is referred to as a hardware accelerator. GPUs are arguably the most common example of a hardware accelerator. GPUs must operate in conjunction with a CPU-based host through a PCI-Express bus, as shown in Figure.

NVIDIA's CUDA nvcc compiler separates the device code from the host code during the compilation process. The device code is written using CUDA C extended with keywords for labeling data-parallel

functions, called kernels . The device code is further compiled by Nvcc. During the link stage, CUDA runtime libraries are added for kernel procedure calls and explicit GPU device manipulation. Further kernel function, named hello From GPU, to print the string of"Hello World from GPU!" as follows:

```
__global____void hello From GPU(void)
{
printf("Hello World from GPU!\n");
}
```

The qualifier __global_tells the compiler that the function will be called from the CPU and exe-cuted on the GPU. Launch the kernel function with the following code:

```
Hello From GPU <<<1,10>>>();
```

Triple angle brackets mark a call from the host thread to the code on the device side. A kernel is executed by an array of threads and all threads run the same code. The parameters within the triple angle brackets are the execution configuration, which specifies how many threads will execute the kernel. In this example, you will run 10 GPU threads.

A typical processing flow of a CUDA program follows this pattern:

1.Copy data from CPU memory to GPU memory.
2.Invoke kernels to operate on the data stored in GPU memory.
3.Copy data back from GPU memory to CPU memory.

Conclusion: We have implemented parallel reduction using Min, Max, Sum and Average Operations. We have implemented CUDA program for calculating Min, Max, Arithmetic mean and Standard deviation Operations on N-element vector.