Experiment 3

Sinhgad Institutes

Date 11 1.

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 values in the vector.

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.

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 well as time taken to find each value.

outcome: Students will understand the implementation of sequential program augmented with compiler directives to specify parallelism.

Theory :

OpenMP -

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 with setting up and co-ordinating threads.

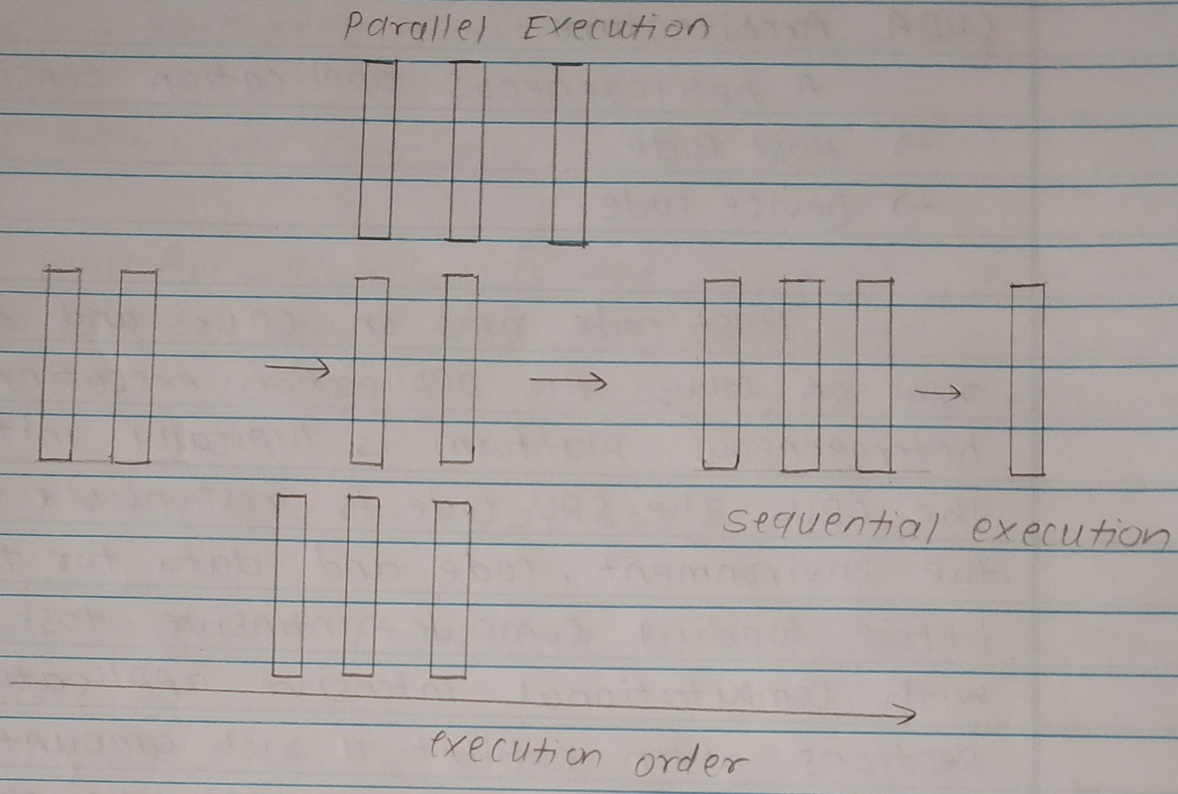
Parallel Programming :

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 largly 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 express · 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
→ 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 task on the device. With computational intensive applications, program sections often exhibit a rich amount of data parallelism. 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.

NVIDIA's CUDA nvcc compiler seperates the device code from the host code during the compilation process. The device code is written using CUDA 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, ~~no~~ named hello from GPU, to Print String of "Hello World from GPU!" as follows:

```
__global__ void helloFromGPU(void)
{
 printf("Hello World from GPU!\n");
}
```

The qualifier _global_ tells the compiler that the function will be called from the GPU and executed on the GPU. Launch the function with the following code :

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

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 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

Experiment 4

Sinhgad Institutes | Date 1 1 | 1.

Title: Design & implementation of Parallel (CUDA) algorithm to Add two large vector and Matrix multiplication using CUDA C

Problem statement: Design & implementation of CUDA algorithm to add two large vector, Multiply vector and matrix and multiply two N x N arrays using n^2.

Prerequisite: Strassen's matrix multiplication Algorithm and CUDA Tutorials.

Programming Tools: Ubuntu 14.04, GPU Driver 352.68, CUDA Toolkit 8.0, CUDNN Library V5.0

objective: To offload parallel computations to the graphics card, when it is appropriated to do so, and to give some idea of how to think about code running.

outcome: Students should understand the basic of GPU computing in the CUDA environment.

Theory:

Introduction -

It has become increasingly common to see supercomputing applications harness the massive Parallelism of graphics cards to speed up computations. one platform for doing so is NVIDIA's

compute unified Device Architecture. We use the example of matrix multiplication to introduce the basics of GPU computing in the CUDA environment.

Matrix Multiplication is a fundamental building block for scientific computing. Moreover, the algorithmic patterns of matrix multiplication are representative. Many other algorithms share similar optimization techniques as matrix multiplication. Therefore, matrix multiplication is one of the most important examples in learning Parallel programming.

A kernel that allows host code to offload matrix multiplication to the GPU. The kernel function is :

```
--global-- void MatMulKernel (Matrix A, Matrix B, Matrix C){
 // Each thread computes one element of C
 // by accumulating results into Cvalue
float Cvalue = 0;
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
if (row > A.height || col > B.width) return;
for (int e=0; e< A.width ; ++e)
    Cvalue += A.elements [row * A.width + e] *
                                B.elements [e* B.width + col];
C.elements [row * C.width + col ] = Cvalue;
}
```

The first line contain __global__ keyword declaring that this is an entry-point function for running code on the device. the declaration float Cvalue=0 sets aside a register to hold this float value where we will accumulate the product of the row and column entries. The next two lines help the thread to discover its row and column within the matrix. It is a good idea to make sure you understand those two lines before moving on.

The if statement in the next line terminates the thread if its row or column place it outside the bounds of the product matrix. This will happen only in those blocks that overhang either the right or bottom side of the matrix.

The next three lines loop over entries of the row of A and the column of B needed to compute the row, col-entry of the product, and the sum of these products are accumulated in the Cvalue variable. Matrices A and B are stored in the device's global memory in row major order, meaning that the matrix is stored as a one-dimensional array, with the first row followed by the second row, and so on. Thus to find the index in this linear array of the (i,j) entry of matrix A. Finally, the last line of the kernal copies this product into the appropriate element of the product matrix C, in the device's global memory.

In light of memory hierarchy described above, each thread loads (2 x A.width) elements in kernel. From global memory two for each iteration through the loop, one from matrix A and one from matrix B.

Matrix A is shown on the left and matrix B is shown on the top, with matrix C, their product on the bottom-right. This is a nice way to layout the matrices visually, since each element of C is the product of row to its left in A and the column above it. in B.

BLOCK_SIZE and will assume that the dimensions of A and B are all multiples of BLOCK_SIZE. Again, each thread will be responsible for computing one element of the product matrix C.

It decomposes matrices A and B in to non-overlapping submatrices of size BLOCK_SIZE x BLOCK_SIZE. It shows in red row and red column. It passes through the same number of these submatrices, since they are of equal length. If it load the left-most of those submatrices of matrix A into shared memory, and the top most of those submatrices of matrix B into shared memory. then it compute the BLOCK_SIZE and continue adding the term-by-term products to our value in C.

Applications :

1. Fast video Trans-coding & Enhancement
2. Medical Imaging
3. Neural Networks
4. Gate-level VLSI simulation

Conclusion :

Strassen's Matrix multiplication algorithm have been implemented parallel using GPU computing in the CUDA environment.