INF3201- Parallel Programming
Assignment 3 – CUDA C

Zulfiqar Ali, Muhammad Nauman Ali

Introduction

The report discusses the implementation to find the greatest number in an array of integers of different sizes. For this, one serial and one parallel solution are implemented. The serial solution is performed using a simple loop and the parallel solution is performed in Compute Unified Device Architecture (CUDA) and then runs on the Nvidia Graphics Processing Unit (GPU). The technique used to find the greatest number in the parallel solution is by reduction tree algorithm.

A GPU is a special processor that is responsible for speeding up the rendering of graphics[1]. GPU's are based on parallel processing architecture which allows it to execute several computations at the same time. This architecture also enables the GPU to cut down complicated computations into millions of sub tasks and perform calculations all together at once. CUDA is an application programming interface by Nvidia that provides direct access to the elements of parallel calculations to perform certain tasks and works on the guidelines of single instruction multiple threads (SIMT)[2]. SIMT guidelines are used to implement parallel solutions together with multithreading on GPUs. A thread is a single unit and is a set of instructions that is used to run kernels on the GPU. A block is the group of threads that can communicate with each other. Similarly, the grid is the group of blocks with threads to launch a kernel. When the group of 32 threads are working together in a block is called warp. Each core in CUDA is called the streaming processor (SP) and 8 SP combine together to form a streaming multiprocessor (SM). All the 8 SPs in the SM execute the same instruction. The global memory in the GPU is the main memory of the GPU which is continuous across kernel calls. A shared memory is a fast memory which is spread across each block and every thread in a block can access that memory. Finally, the registers are the fastest memory in the GPU that are limited to each thread. The variables in the kernel use these registers to perform computations.

Method

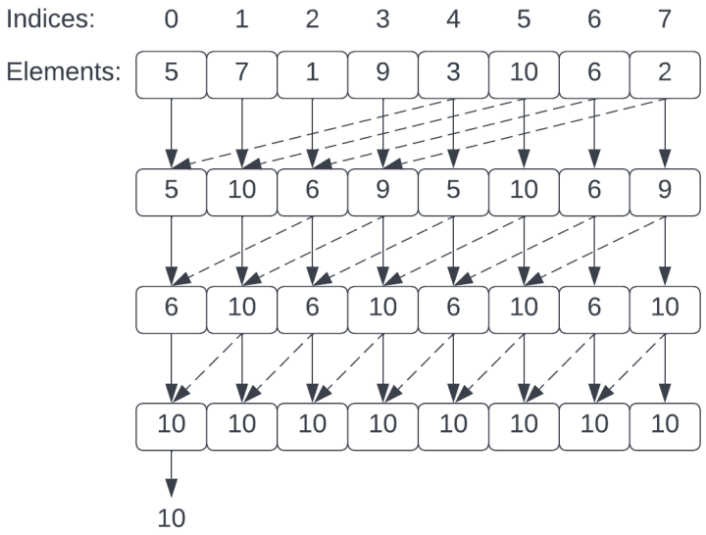

Figure 1: Solution design for parallel reduction.

The method shown above is used as a reduction tree in order to parallelize the problem. It is a tree based technique that is used within each thread block. For each iteration the size of the array is divided by half and each half of the thread would compare the other half to find the maximum value. The maximum value is stored in the first index of the array. For example in the example above the array is divided into half and for the first iteration the 0 index is compared with 4th index. The maximum of both is 5 and it is stored on 0 index. The iterations are repeated for every block until every block has a maximum in its shared memory. This local maximum is compared with a global variable to return the maximum number. The reason for choosing this method was keeping in mind the shared memory bank conflicts. That was the main reason to choose this approach.

Implementation

Two versions were implemented to find the greatest number in an array and are discussed below. In order to compile and run the code, execute the following commands in the cluster terminal on a compute node with an Nvidia GPU for example on compute-4-2.

Command to compile: **nvcc main.cu -o main timer.cpp**
Command to run: ./main

The current solution will work for the size of 402653184 of the array. As the solution is not automated therefore the array size can be changed manually by changing the value of x variable on line 49 of the code.

Serial version:

```
int arraySize;
int j=0
int myArray[arraySize]
int maximum= myArray[0]
 while(j<arraySize){
         if(myArray[j] > maximum) maximum = myArray[j]
     j++
 }
```

Figure 2: Pseudo code for serial version.

The above pseudo code shows the serial version implementation of the code. A while loop is used to iterate through each element of an array and compare with the maximum. Initially the maximum is the first element of the array. The maximum is updated each time if the value in a particular index is greater than the current maximum. Hence, a maximum number is found when execution of the while loop is completed.

Parallel version:

```
unsigned int threadIndX = threadIdx.x;
unsigned int wid = threadIndX % 32;
unsigned int index = threadIndX + blockIdx.x * blockDim.x;
__shared__ int local[threadNumberPerBlock];
 // call intra-block synchronization function
//performing the warp reduction
for (int oSet = 32; oSet > 0; oSet /= 2) {
    int value = __shfl_down_sync(mask, local[threadIndX], oSet);
    if(value > local[wid]) {
        local[wid] = value;
    }
   // call intra-block synchronization function
   //finding max using atomicMax function
   if(threadIndX == 0){
    atomicMax(deviceMaximum, local[0]);
   }
```

Figure 3: Pseudo code for parallel version.

The above pseudo is the parallel version implementation to find the greatest number using reduction. Initially, for every thread a shared memory is declared by the name of local. Every block has its own shared memory which is loaded from global memory. Then, every thread will load one element from the global to the shared memory. An intra-block synchronization function is called in order to confirm that all the elements were loaded successfully. After this, warp level reduction is performed which uses the function called __shfl_down_sync() in order to calculate the sum of the variable called value that is held by every thread in the warp. As we know that every wrap has 32 threads and every thread occupies one lane. The function __shfl down sync(mask, local[threadIndX], oSet) retrieves the value for a thread at lane Y in the warp from thread at lane Y+oSet of the same warp. For every iteration the local[wid] is updated with the value variable if it is greater. At last, atomicMax function is used to put the block's greatest number into the deviceMaximum global memory and copied back to the host.

Experiments and results

As mentioned in the assignment text 30 experiments were conducted for time recordings of serial and parallel versions.

```
Device 0: "Quadro P2000"
  CUDA Driver Version / Runtime Version          11.7 / 10.2
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 5059 MBytes (5304483840 bytes)
  ( 8) Multiprocessors, (128) CUDA Cores/MP:     1024 CUDA Cores
  GPU Max Clock rate:                            1481 MHz (1.48 GHz)
  Memory Clock rate:                             3504 Mhz
  Memory Bus Width:                              160-bit
  L2 Cache Size:                                 1310720 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
```

Figure 4: The hardware detail for all the experiments.

Serial version

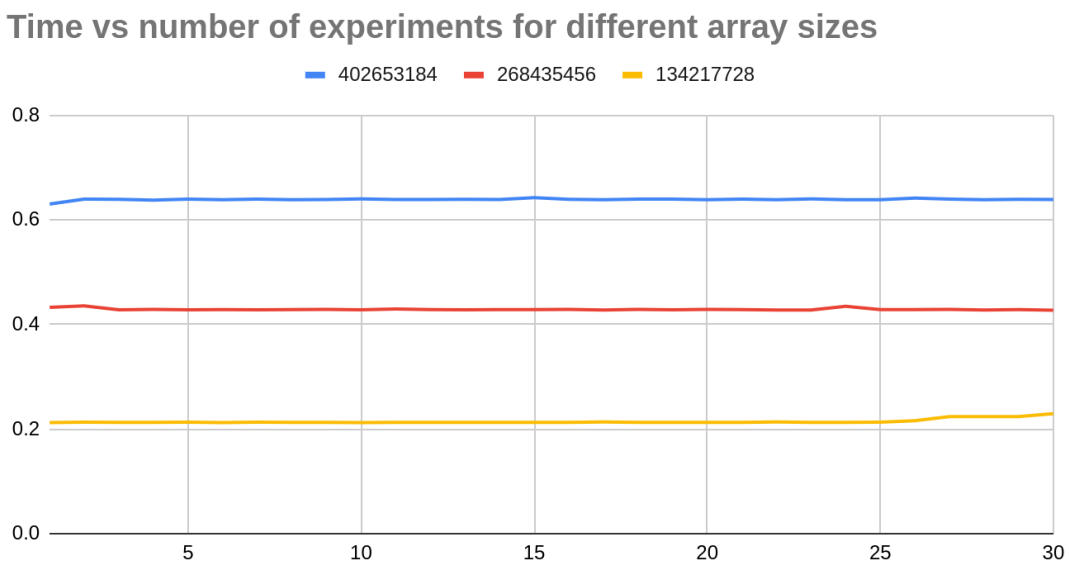

Figure 5: Time in seconds of 30 experiments for serial version with different array sizes.

Array Size	Minimum time in seconds	Mean Time
402653184	0.6301	0.6390
268435456	0.4273	0.4289
134217728	0.2124	0.2146

Figure 6: Mean and Minimum time for different array sizes.

It can be observed from the graph that as the number of array sizes increases the time taken by the serial version to find the maximum number also increases. Secondly, it can be seen that the minimum time and the mean time are almost the same uptill two decimal digits which depicts that the solution is working fine.

Parallel version

Same approach as the serial was used to perform the experiments but this time the copy time from the host to the device was also recorded.

For array size of 402653184

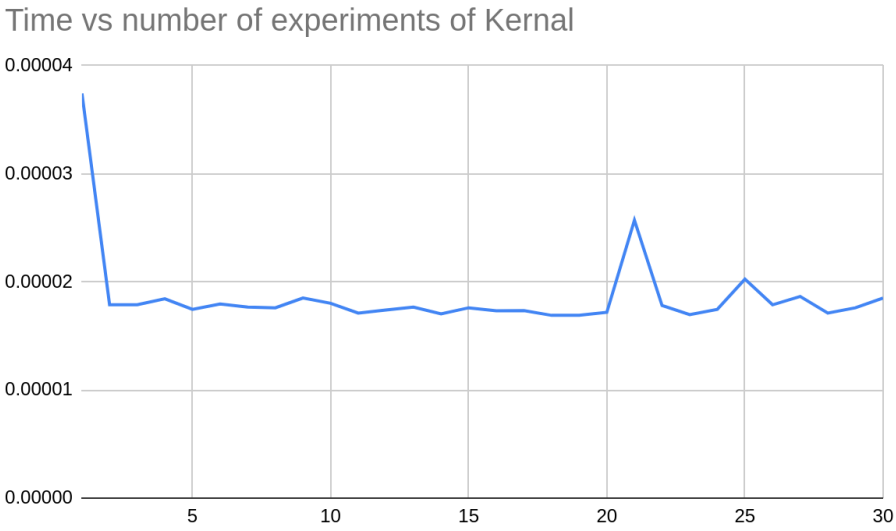

Figure 7: Kernel execution time in seconds with number of experiments.

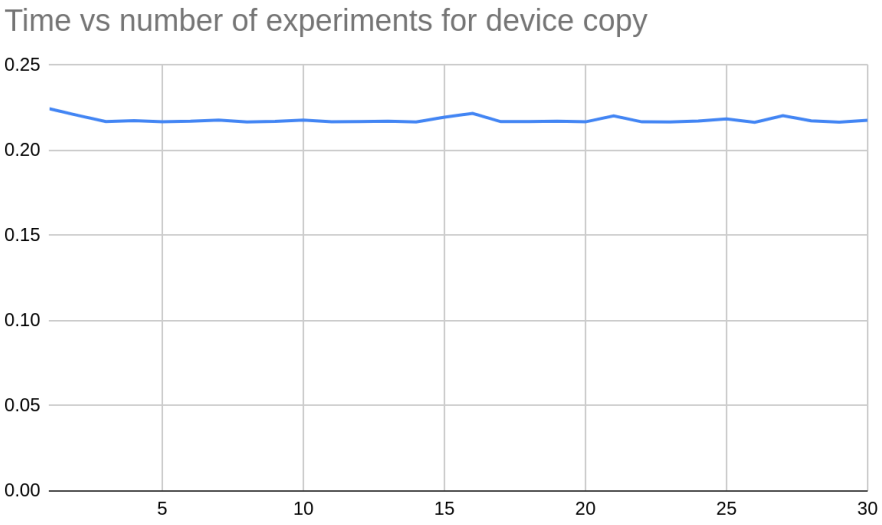

Figure 8: Device copy time in seconds with number of experiments.

Dimension	Minimum Time in seconds	Mean Time
Kernel Time	0.000016902	0.000018641
Device Copy Time	0.216534072	0.2179272377

Figure 9: Mean and Minimum time for kernel and device copy.

For array size of 268435456

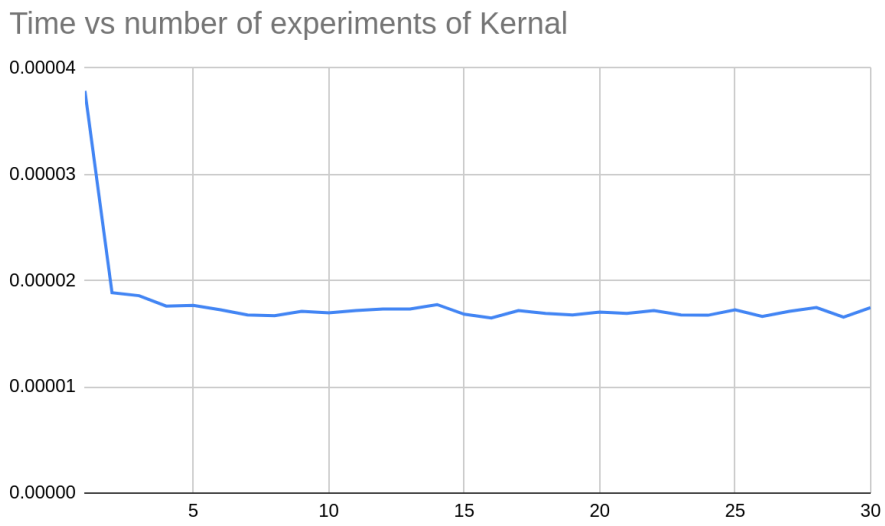

Figure 10: Kernel execution time in seconds with number of experiments.

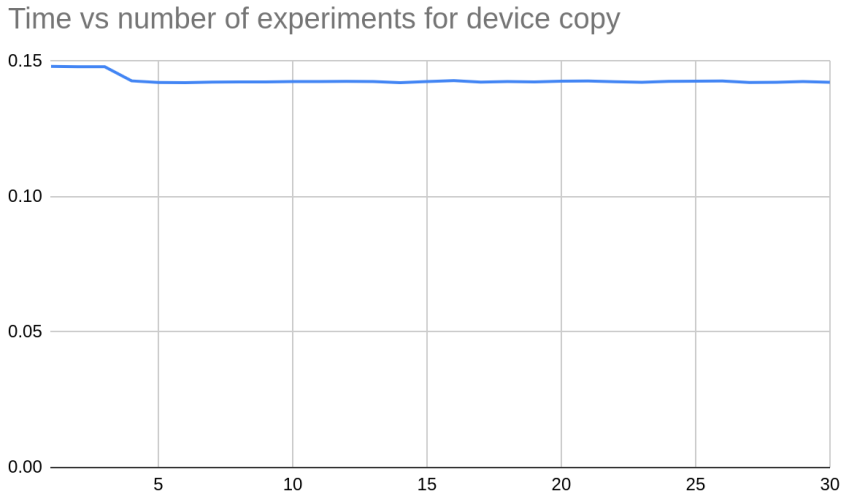

Figure 11:Device copy time in seconds with number of experiments.

Dimension	Minimum Time in seconds	Mean Time
Kernel Time	0.000016482	0.0000178717
Device Copy Time	0.142118036	0.1430123436

Figure 12: Mean and Minimum time for kernel and device copy.

For array size of 134217728

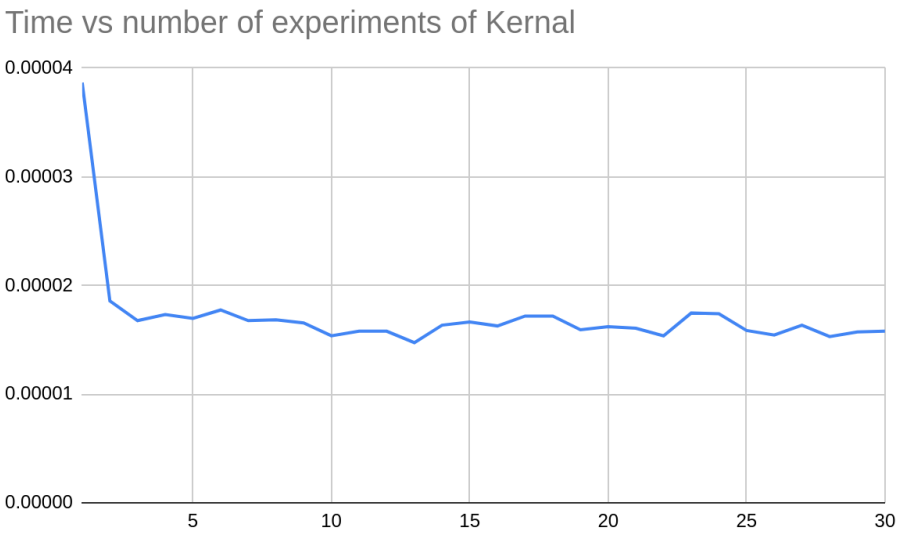

Figure 13: Kernel execution time in seconds with number of experiments.

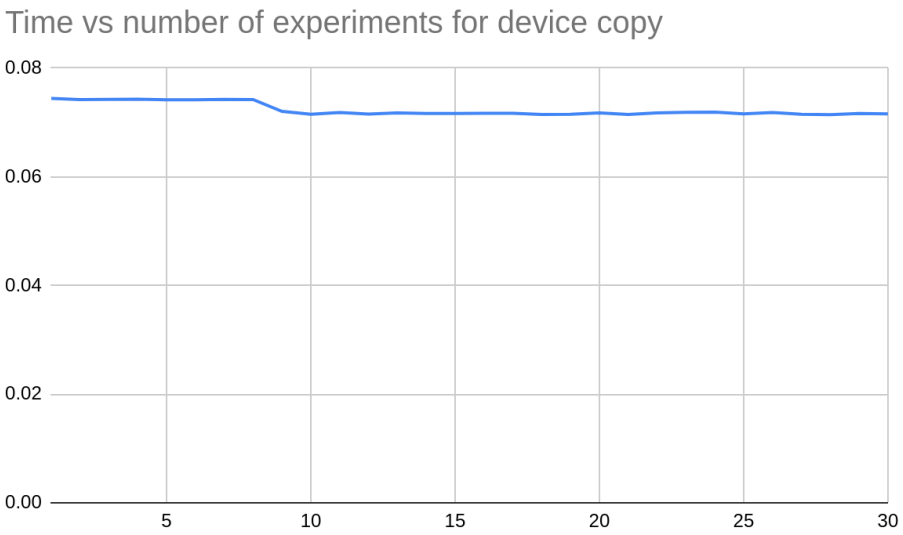

Figure 14: Device copy time in seconds with number of experiments.

Dimension	Minimum Time in seconds	Mean Time
Kernel Time	0.000014737	0.0000171435
Device Copy Time	0.071435356	0.07235625297

Figure 15: Mean and Minimum time for kernel and device copy.

GeForce GTX 2060 Vs Quadro P2000 for the array size of 402653184

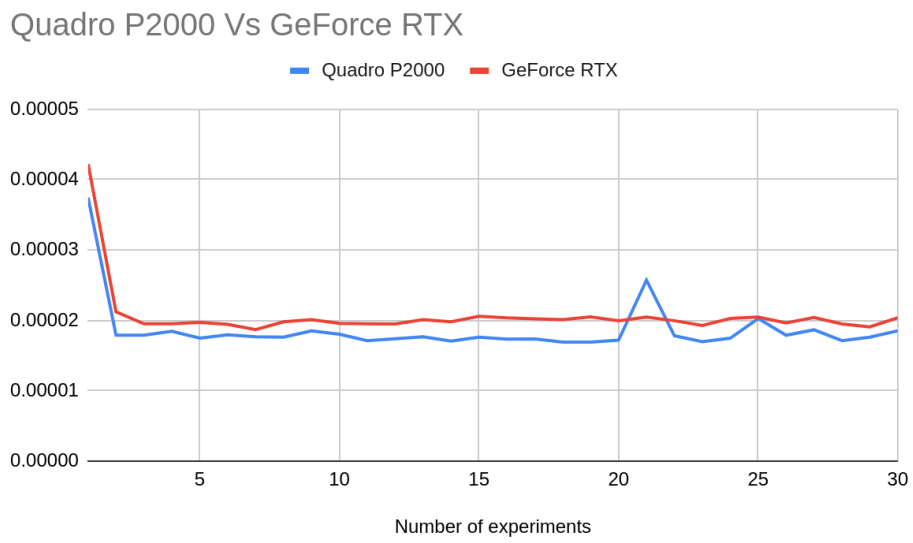

Figure 16: Environment comparison with number of experiments and time in seconds.

```
Device 0: "NVIDIA GeForce RTX 2060"
  CUDA Driver Version / Runtime Version          11.7 / 10.2
  CUDA Capability Major/Minor version number:    7.5
  Total amount of global memory:                 5934 MBytes (6222577664 bytes)
  (30) Multiprocessors, ( 64) CUDA Cores/MP:     1920 CUDA Cores
  GPU Max Clock rate:                            1710 MHz (1.71 GHz)
  Memory Clock rate:                             7001 Mhz
  Memory Bus Width:                              192-bit
  L2 Cache Size:                                 3145728 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 3 copy engine(s)
```

Figure 17: The hardware detail of GeForce RTX.

As seen from the graph above the Quadro P2000 performs slightly better as it has more CUDA cores than the GeForce RTX.

Discussion

From the experiments conducted it can be observed that for smaller array sizes the sequential version outperforms the parallel version as the copy from host data to the device data takes a lot of time. On the other hand as the array size gets bigger the parallel version gets better as the array size increases due to the reason that the load is distributed across threads. The experiments

also showed that the parallel version is 70% - 75% better than the sequential version and as the array size gets bigger the efficiency increases.

As it can be seen from the graph that the kernel takes a lot of time in the first experiment of each array size this is due to the reason that in the first iteration most of the threads are idle and are not performing but later on the solution gets stable and less time is consumed. This problem can be solved by replacing one load with two loads and the first addition of reduction.

Conclusion

The report explains the implementation to find the maximum number in sequential and parallel versions. The sequential version uses a basic loop to find the maximum number and the parallel version uses a sequential addressing method using the parallel reduction. It can be observed from the implementation and experiments that as the array size gets bigger the parallel version outperforms the serial version due to the reason that the loads get distributed along different threads. But for smaller array sizes the serial version is better.

References

1. https://www.intel.com/content/www/us/en/products/docs/processors/what-is-a-gpu.html
2. https://en.wikipedia.org/wiki/CUDA