GPU Programming – Lab 4 26.11.21 – 2021/22

SYNCRONIZATION IN PARALLEL PROGRAMS

Goals:

- To acquire confidence in the development and analysis of parallel programs for GPUs
- To analyze the effect and need of thread synchronization directives in parallel programs
- Use atomic operations for memory coherency in parallel programs

---

Exercise 1:

Analyze the following conceptual kernel performing the next operations distributed among the threads in a block:

1) Initialize a variable with X + 1, where X is an input value for global memory
2) Initialize a variable with (X+1)/2;
3) Finds the cos((X+1)/2);
4) Determines the operation cos((X+1) / 2)*((X+1) / 2) + X + 1

```
__global__ void simple_kernel(float* x, float* y)
{
        int i = threadIdx.x + blockDim.x * blockIdx.x;
        if ( (i % blockDim.x) == 0)
                x[i] = x[i] + 1;
        if ( (i % blockDim.x) == 1)
                y[i] = x[i-1] / 2.0;
        if ( (i % blockDim.x) == 2)
                x[i-1] = __cos(y[i-1]);
        if ( (i % blockDim.x) == 3)
                y[i-1] = x[i-2] * y[i-2] + x[i-3];
}
```

1) Determine if thread synchronization is needed. Then, add the thread synchronization assessment (**__syncthreads()**) to the code in the required places if needed.

2) Verify the execution of the kernel using a parallel configuration with 8 blocks and 32 threads per block.

Is it possible to affirm that the kernel describes a fully embarrassingly parallel application? Why?

Is it possible to affirm that the kernel produces one of more intra-warp divergences in the parallel application? Why?

Are there issues in the memory management in the GPU when executing the kernel?

Is there a way to optimize and increase the performance of the application?

Exercise 2:

Consider scheme of a simple 1D convolution. In this algorithm, a set of values from the input vector **N** are multiplied with a constant filter **M**, so producing an output **P** following the next equation:

$$P[j] = \sum_{i=0}^{K} N[i+x] * M[i]$$

Where ***K*** is the width of the filter **M**, and ***x*** is the offset in the input vector **N**. This equation is applied in windows with size ***K*** on all sets of input values from **N**.

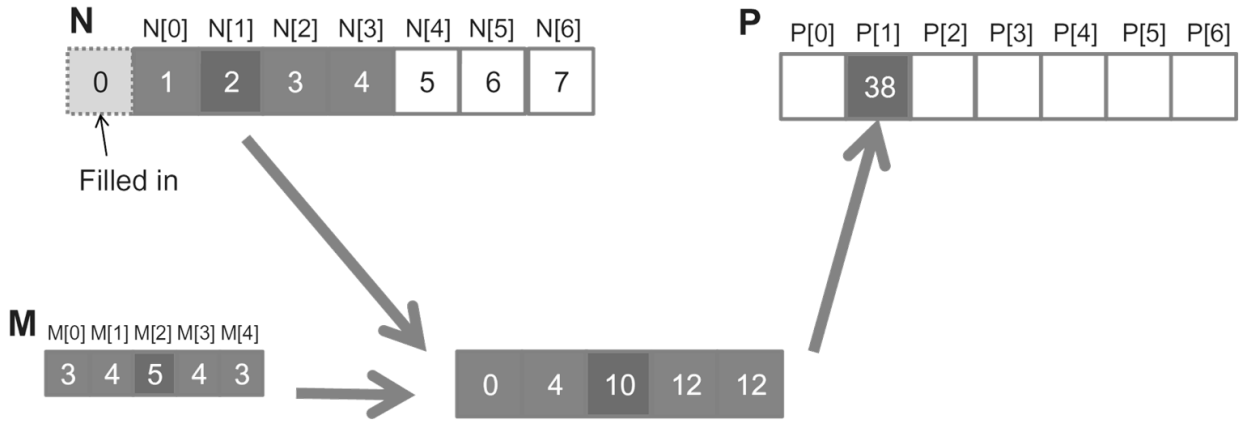

Figure 1. A general scheme of a 1D convolution

A simple parallel description of a 1D convolution is presented in Listing 1. In this case, each thread performs the convolution between the inputs **N** and **M**.

Listing 1. 1D convolution kernel

```
1  __global__ void convolution_1(float *M, float *P, float *N, int Mask_Width, int vector_Width)
2  {
3          int tid= blockIdx.x * blockDim.x + threadIdx.x;
4          float Pvalue = 0;
5          int N_start_point = tid - (Mask_Width/2);
6
7          for (int j = 0; j < Mask_Width; j++)
8          {
9                  if (N_start_point + j >= 0 && N_start_point+ j < vector_Width)
10                 {
11                         Pvalue += N[N_start_point+ j] * M[j];
12                 }
13         }
14         P[i] = Pvalue;
15 }
```

1) Describe the code for the host in order to use the *convolution_1* kernel. Check the operation with an input vector **N** of 2048 values and a filter **M** of 8 values.
2) Optimize the operation of the *convolution_1* kernel by storing the filter **M** in the constant memory of the GPU device (**Hint:** adapt the kernel to use the constant memory).
3) Optimize the operation of the *convolution_1* kernel by storing the values of the input vector **N** in the shared memory (**Hint:** adapt the kernel to use the constant and shared memories). Please, consider the use of thread synchronization (*__syncthreads()*) when addressing the shared memory and processing the convolution.
   Is the execution of the kernel equal with and without thread synchronization? Why?

Exercise 3:

In signal processing applications, waveforms are preliminary analyzed to extract several features, such as the maximum value, minimum value, etc. GPU accelerators can optimize this process by processing the input signals in parallel and extracting several parameters in the process.

Consider the case of a 1D waveform as depicted in Figure 2. The analysis can be performed by dividing the waveform in parts processed by blocks in a GPU.

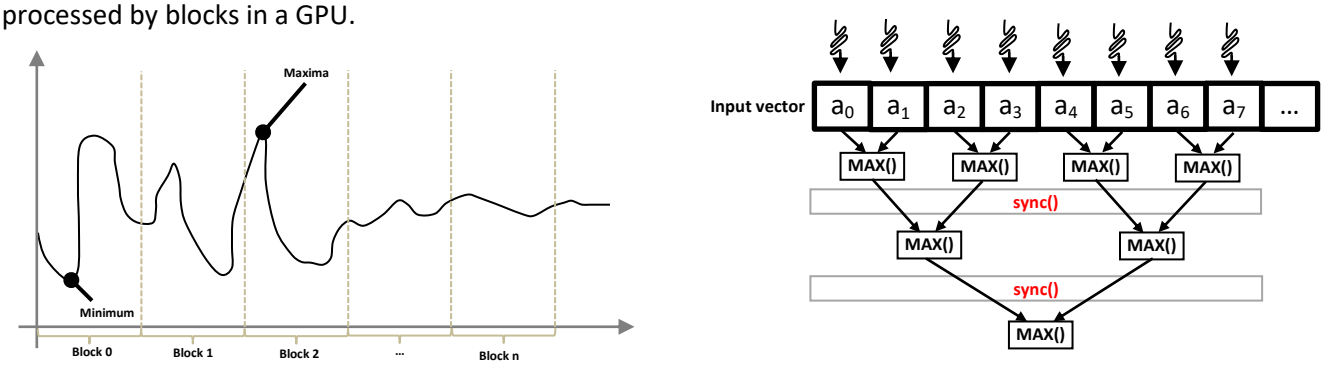

Figure 2. A 1D waveform divided in blocks for GPU processing (left), parallel scan approach to apply the max operation to an input vector (right).

1) Describe a kernel to determine the maximum and minimum values and their locations of one input 1D float waveform. Use a parallel scan approach to describe the kernel and take advantage of the available memory resources to increase the performance. Consider to include synchronization assessment after loading and execution operations.

Hint: analyze the input waveform in blocks and then determine the max and min values among the blocks (Use atomic operations and the global memory to synchronize and handle the results among blocks).

2) **Optional:** Extend the kernel description to analyze multi-signal inputs organized as a vector with the following organization:

```
typedef struct
{
        float signal_1_value;
        float signal_2_value;
        float signal_3_value;
} MULTI_SIGNAL_VECTOR;

MULTI_SIGNAL_VECTOR input_vector[10000];
```