GPU Programming – Lab 2 28.10.21 – 2021/22

PROFILING PARALLEL PROGRAMS

Goals:

- To acquire confidence in the development and analysis of parallel programs for GPUs
- To analyze the performance effect of different workloads and levels of parallelism in GPU programs
- To practice with the profiler utilities in the Jetson Nano GPU platform

Exercise 1:

Preliminary tutorial:

For this exercise, listing 1 describes a naïve implementation of a parallel algorithm to sort elements in a vector. As can be observed, the parallel program uses two input arguments (*data* and *num_elem*). The first argument allows addressing the input vector for sorting but also stores the outputs. On the other hand, the second argument defines the limit number of elements to be sorted by the program.

Each thread in the program evaluates a set of two consecutive elements in the vector and organizes them according to their magnitude. An offset is employed to select an even or odd set of elements for the evaluation. This sorting procedure is repeated the number of times defined by the second input argument.

Listing 1. Parallel sorting algorithm.

```
__global__ void parallel_sort(int *data, int num_elem)
{
        unsigned int tid = (blockIdx.x * blockDim.x) + threadIdx.x;
        unsigned int tid_idx;
        unsigned int offset = 0;
        unsigned int d0, d1;
        unsigned int  tid_idx_max = (num_elem - 1);
        for (int i = 0; i < num_elem; i++)
        {
                tid_idx = (tid * 2) + offset;
                if (tid_idx < tid_idx_max)
                {
                        d0 = data[tid_idx];
                        d1 = data[tid_idx + 1];
                        if (d0 > d1)
                        {
                                data[tid_idx] = d1;
                                data[tid_idx + 1] = d0;
                        }
                }
                if (offset == 0)
                        offset = 1;
                else
                        offset = 0;
        }
}
```

Profiling steps:

1) Describe the Host code to employ the *parallel_sort* kernel and verify the operation with an input vector of 32 elements.

   Hint: use the same procedures from Lab1 to describe the Host code.

   According to the description in listing 1, is it possible to affirm that the sorting algorithm is a fully embarrassingly parallel kernel? Why?

Is it possible to affirm that all threads in the *parallel_sort* kernel would execute the same instructions? Are all threads active during the execution of the kernel? Why?

2) Use the profiling tool (*nvprof*) to determine the execution time of the parallel_sort kernel.

   Hint: from command line, use: **nvprof ./your_app.out** to observe the summary mode of the parallel program. In this mode, *nvprof* outputs a single result line for each kernel and each type of CUDA memory copy/set performed by the application. For each kernel, *nvprof* outputs the total time of all instances of the kernel and memory movements.

   According to the profiler output, is the execution time the same for the kernel after every running? Where is the main bottleneck for the parallel application under the selected configuration?

NOTE: in Jetson Nano platform, *nvprof* may have no system privileges, so use administrator privileges (***sudo su***) or a make file to perform the profiling of an application.

3) Determine the number of registers per thread, the static shared memory used per block, and the dynamic shared memory used per block using the *(nvprof)* tool.

   Hint: from command line, use: **nvprof --print-gpu-trace ./your_app.out** to observe the running kernel, as well as the grid dimensions used for each launch. Moreover, the report also includes the registers and shared memory occupancy and the duration of each kernel.

NOTE: try and analyze the effect of **nvprof --print-api-trace ./your_app.out**

4) Analyze the effect in performance for the kernel and the memory copy operations when the workload varies from 32 to 2048 elements in the vector to be sorted. For this purpose, determine the relationship between the execution time vs. the number of elements to process by the kernel.

   Hint: Perform the code changes to use the kernel and sort different size elements in the vector: 32, 64, 128, 256, 512, 1024, 2048. Then, use the profiler to calculate the execution time of each kernel.

   Are the parallel algorithm and the compilation correctly working for a vector with 2048 elements?

   Please use the documentation of Jetson Nano to understand the behavior for a configuration with 2048 elements. Then, adopt the kernel parameters to sort a vector with 2048 elements.

5) Analyze the effect in performance for the kernel and the memory copy operations when the workload varies from 32 to 2048 elements in the vector to be sorted. For this purpose, determine the relation between the execution time vs. the number of elements to process by the kernel.

   Hint: Perform the changes in code to use the kernel and sort different size elements in the vector: 32, 64, 128, 256, 512, 1024, 2048. Then, use the profiler to calculate the execution time of each kernel.

   Is the parallel algorithm and the compilation correctly working for a vector with 2048 elements?

   Please, use the documentation of Jetson Nano to understand the behavior for a configuration with 2048 elements. Then, adopt the kernel parameters to sort a vector with 2048 elements.

NOTE: use the concept of Threads-per-Block and Blocks-per-Grid to adapt the kernel parameters.

6) Determine the parallel metric traces for the application configured to sort 1024 elements in a vector. Analyze the number of IPCs (Instructions per cycle), the number of integer instructions, number of floating-point operations (single precision), number of control-flow instructions, number of load/store instructions, and number of instructions per warp.

   Hint: use **nvprof –metrics all --log-file your_file.log ./your_app.out**

   Is the number of integer and floating-point instructions from the traces somehow related? Why?

7) Determine the parallel event traces for the sorting application and analyze the number of warps launched by the kernel, the number of unconditional instructions executed per thread (not predicated), and the number of CTAs launched per SM. For this purpose, evaluate the parallel configurations:

   a) 1 block-per-grid and 1024 threads-per-block
   b) 2 blocks-per-grid and 512 threads-per-block

   Hint: use **nvprof –events all --log-file your_file.log ./your_app.out**

   After running both kernels with the different configurations, are the output results equal? Why?
   Is there any relation between the CTAs and the blocks-per-grid?

Exercise 2:

Repeat steps 2 to 6 from Exercise 1 using the vector Add kernel described in Lab1.

Exercise 3:

The degree of parallelism can also impact the performance and resource usage of a parallel application. To observe and evaluate the effect of different levels of parallelism:

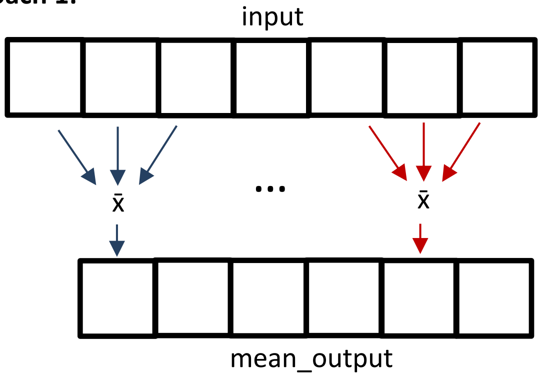
1) Design two parallel programs (*kernels*) to calculate the local mean (mean among three consecutive elements of a vector) using the two approaches as depicted in the following figure.

   In the first approach, the kernel performs one operation and store the result in a new vector (*mean_output*). In the second approach, the kernel performs two consecutive operations and stores the two results in a new vector (*mean_output*).

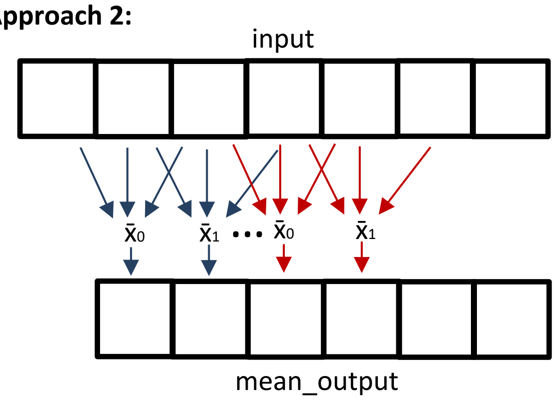
   Hint: Use the following template to define each kernel:

```
__global__ void mean_XXX(float *input, float *mean_output, int total_elements)
{
   …
}
```

Approach 1:



Approach 2:



$\bar{X} = \frac{a_0+a_1+a_2}{3}$, where a_0, a_1 *and* a_2 are the consecutive elements from the input vector

2) Configure and verify the operation of both kernels to operate with an input vector of a) 256, b) 512, and c) 1024 elements.

3) Determine the performance, IPC, and the number of registers used per thread for both kernels.

4) Determine and compare the number of integer, floating-point, and control-flow instructions for both kernels.

5) Calculate the dependency analysis for both kernels and analyze the main cause of latency (*waiting time*).

   Hint: use the option **--dependency-analysis** in **nvprof** tool.