CS6023 : GPU Programming

Assignment 3

Submission deadline :31st March–3rd April 2022 23:55 on Moodle

Problem Statement:

Multicore Task Scheduling:

In a system, there are M cores. There are N tasks which are present in the ready queue. Each task has a different execution time (in seconds). Each core can execute a single task at a time. Multiple cores can execute tasks simultaneously.
Each task has a priority assigned to it. The first task with a particular priority decides on which core it can execute (say core 2). It depends on which core is free. When two or more cores are free, a task should be executed on the core with the smaller ID. E.g. When core 3 and core 5 are free, then the task will get executed on core 3. **Other tasks with the same priority execute on the same core (eg. core 2) as the first task of that priority**.
Whenever a core completes execution of a task, i.e., whenever it is free, a new task is taken from the ready queue depending upon the task's priority. If a task with certain priority does not belong to that core, and if the core is free, it will be idle until the next appropriate task is taken out. If a task with a certain priority belongs to a core and the core is busy, the task has to wait until that particular core is free.
Since we have only one queue in the system, a task is popped out of the queue only when the previous task is out. Thus, the ready queue is a blocking structure.

Assumptions:

- All tasks are in the ready queue at time zero.
- All tasks are independent of each other.
- Context switching time between tasks is zero.
- Scheduling time is zero.
- The number of cores are at least as many as the number of different priorities.

Tasks:

Print the total time (waiting time in queue + execution time on core) spent in the system for each task.

Format:

Input Format

First line contains M, N respectively.
The next line contains execution times T(i) of each task, in an order present in the ready queue.
The next line contains priorities Pr(i) of each task.

Output Format:

Total time taken by each task.

Input Constraints:

1 ≤ M ≤ 1000
1 ≤ N ≤ 10^5
1 ≤ T(i) ≤ 100
0 ≤ Pr(i) < M

Example:

Sample input:

```
3 10
1 3 2 5 4 1 2 3 1 2
1 0 0 1 1 0 0 0 2 2
```

Sample output:

```
1 3 5  12 9 11 14 12 14
```

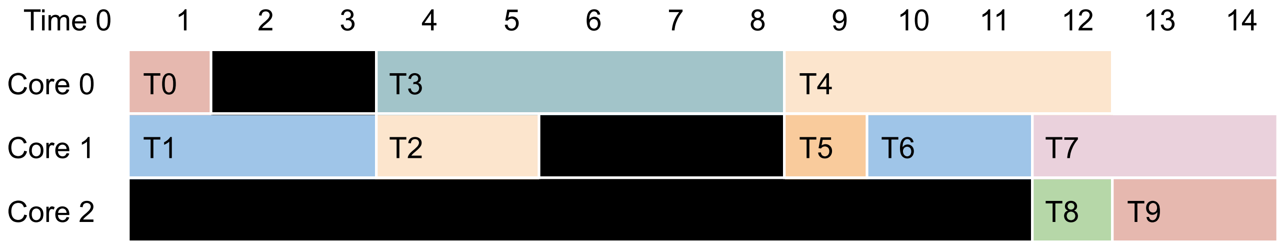
Explanation:

Initially, all cores are free. So smaller ID core i.e. core 0 starts executing task 0. Since task 0 has priority 1, all other tasks which have priority 1 will get executed on core 0 when their turn comes up.
Now next task 1 is having priority as 0 and two cores are free. Again a smaller ID core i.e. core 1 will execute task 1. So, all tasks with priority 0 will get executed on core 1.
At time 0, Core 2 is free and task 2 is at the front of the queue. But since it has priority 0, it must be executed on the core where the first task of priority 0 was executed (that is, Core 1 which executed task 1 of priority 0). Since Core 1 is not free, task 2 is not popped out of the queue.

Meanwhile, when task 0 is completed on core 0, task 2 has not yet been popped out of the queue. So it will wait until task 2 is popped. At time 3 seconds, task 1 completes and frees Core 1. Thus task 2 with priority 0 is popped and executed on core 1. At the same time, task 3 with priority 1 is also scheduled for execution on core 0.

Similarly, tasks 4,5,6,7 execute. When task 8, which is the first task with priority 2, is taken out from the ready queue, none of the other cores are free, so it will get executed on core 2. And so does task 9 which executes on core 2.

Please find the Gantt chart below for easy understanding.



Black color represents idle time of the cores.

Implementation Details

- Queue can be implemented as an array. All GPU threads should access it in parallel – thus retrieving a task from the queue would need synchronization.
- Task execution need not be simulated (e.g., a thread waiting for a few seconds). You can simply calculate its turnaround time.
- Sequential implementations would fetch zero marks.
- You are free to launch the kernel with the number of threads equal to the number of cores or number of tasks.

Points to be noted

- The provided ‘main.cu’ contains the code, which takes care of file reading, writing and computation of time taken by ‘operations’ function. You need to implement the given ‘operations’ function.
- You can write your own code from scratch. In this case, output should be written in exactly the same manner as given in ‘main.cu’ file. Also, make sure that input file and output files are taken as arguments.
- You need to write your own kernels/parallelized code inside the ‘operations’ function.
- The sequential implementation leads to ‘0’ marks on the assignment. Please keep this in mind.
- Test your code on large inputs.

Submission Guidelines

- You can either use the code provided by us or can write the entire code on your own from scratch based on your choice.
- Rename the file ‘main.cu’, which contains the implementation of the above-described functionality, to <ROLL_NO>.cu
- For example, if your roll number is CS20S052, then the name of the file you submit on the Moodle should be CS20S052.cu (submit only the <ROLL_NO>.cu file).
- After submission, download the file and make sure it was the one you intended to submit.

Learning Suggestion

- Write a CPU-version of code achieving the same functionality. Time the CPU code and GPU code separately for large inputs and compare the performances.