Distributed and Parallel Systems — Due on Sunday, March 19, 2017

Assignment 2

CS4402B / CS9535B — *University of Western Ontario*

Submission instructions. This assignment consists of a single problem which asks to write a `CUDA` program for performing a counting sort.

Format: The submission should consist of

- the source code of this program together with
- a `Makefile` for compiling it, similarly to the simple examples posted on the course web site.

Your `CUDA` program should contain a test applying your counting sort code to an input array containing n elements with random values between 0 and 256. The integer n will be assumed to be a power of 2. You should test your program:

- $n = 1024$ for debugging/verification purposes and print the result on the screen,
- $n = 2^{20}$, $N = 2^{21}$, $N = 2^{22}$, $N = 2^{23}$, $N = 2^{24}$ for performance measurements, comparing the running time of your `CUDA` program against a serial version written in `C`.

Submission: The assignment should be submitted through the OWL website of the class.

Collaboration. You are expected to do this assignment *on your own* without assistance from anyone else in the class. However, you can use the literature. Be careful! You might find on the web solutions to our problem which are not appropriate. For instance, because the parallelism model is different. So please, avoid those traps and work out the solutions by yourself. You should not hesitate to contact me if you have any questions regarding this assignment. I will be more than happy to help.

Marking. This assignment will be marked out of 100. A 10 % bonus will be given if your program is clearly organized and documented. Messy or undocumented code may give rise to a 10 % malus.

PROBBLEM 1. [100 points] The goal of this problem is to realize a `CUDA` implementation of the counting sort algorithm We use the same notations as in the Wikipedia page:

`https://en.wikipedia.org/wiki/Counting_sort`

We assume that k is small, say $k \in \{2^6, 2^7, 2^8\}$ (we assume that the entries are positive integers in the range $1 \cdots k$) while n is large, say $n \in \{2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}\}$. We propose the following algorithm targeting a `CUDA` implementation using a one-dimensional grid and one-dimensional thread-blocks. Let B and p be two powers of 2 such that each thread-block has B threads, and the input array `Input` is divided in p consecutive sub-arrays such that the i-th thread-block works with the i sub-array, for $0 \leq i < p$. In the proposed Kernel 1 below, each thread in each thread-block sorts B elements from `Input`. Hence, each of these sub-arrays is regarded as the row-major layout of a matrix with B columns and B rows. B is intended to be small, say $B = 2^5$ or $B = 2^6$. Therefore, we have $p = n/B^2$.

Kernel 1: Each thread, in each thread-block, uses a "private" `Count` array. In fact, this `Count` array should be a row of a shared array with B rows and k columns. This latter array resides in shared memory and is shared among the threads of a given thread-block.

Step 1.1 : For all $0 \leq i \leq p-1$, for all $0 \leq j \leq B-1$, the j-th thread of the i-th thread-block scans the elements $jB \cdots (j+1)B - 1$ of the i sub-array and fills up its private `Count` array. It is true that, in this design, accesses will not be coalesced. You are free to fix that, which is easy, by modifying the way data is layed out in global memory.

Step 1.2 : the B threads (of the i-th thread-block, for all $0 \leq i \leq p-1$) together add the "private" `Count` array**s**. That is, these B threads add those arrays component-wise into a single `Count` array.

Step 1.3 : Thread 0 applies the *prefix sum* algorithm to `Count` (in order to calculate the starting index for each key). It is true that, in this design, hardware occupancy is low during that step. However, B is small and accessing shared memory is fast. Hence, this is only a minor concern in terms of performance.

Step 1.4 : Thread 0 computes (and writes to global memory) the `Output` array, which correspond to the entries that were read by the threads of its thread-block, from the `Input` array. Since B is small and memory accesses are coalesced, this step is **again** a minor concern in terms of (low) hardware occupancy.

Hence at the end of this kernel, n/p `Output` arrays have been written to the global memory.

Kernel 2: Its goal is to merge the `Output` arrays in $O(\log(p))$ calls. Recall that the `Output` arrays have been sorted by Kernel 1.

- During its first call, this kernel merges p `Output` arrays of size n/p into $p/2$ `Output` arrays of size $2n/p$.

- During its second call, this kernel merges $p/2$ `Output` arrays of size $2n/p$ into $p/4$ `Output` arrays of size $4n/p$.
- etc.
- Merging is done with the `merge` procedure of *MergeSort* algorithm. You are free to use either a serial `merge` or a parallel one, based on the ideas seen in class.
- For this kernel to make a good use of the hardware, one can have each thread-block merging several pairs, but this is not required.