GPU Project 3: Parallel Radix Partition

Instructor: Dr. Yi-Cheng Tu
Course: CIS6930/4930 Massive Parallel Computing
Submission Deadline: November 15, 2023 at 11:59 PM

1 Overview

In this project you will implement a GPU version of the parallel radix partition using CUDA. Radix partition is a partition mechanism commonly used in parallel hash join. It reorders the input keys so that the keys have the same radix value (or hash value) and are gathered into a contiguous memory space that forms a partition. The output array consists of all the partitions and the partitions with smaller radix values are located in front of those with larger radix values.

To begin with, each key of the input array needs to determine its radix value using hash function $h1$. $h1$ can be as simple as a bit field extraction which takes little time but does the job fairly. As a result, the keys that have the same radix value are clustered consecutively in the memory space. In order to efficiently partition the input in parallel, first we need to compute a histogram of the radix values by scanning the array of keys so that we know the number of keys that belong to each partition. This is similar to computing the histogram of distance in project 1 and 2. After we obtain the histogram, we perform an exclusive prefix scan on the histogram to get the prefix sum of the bucket counters. The prefix sum serves as the output offset of each partition for each thread block so the threads can work independently to reorder the keys they processed when building the histogram.

The main objective of this project is to practice your CUDA programming skills. It is essential that your code produces accurate results. While performance is not the primary goal of this project, you still need to ensure that your code runs faster than a threshold value (to be announced later). You have the freedom to choose any techniques you learned from class to improve your program, these include (but are not limited to): using shared memory to hold input data, manipulating input data layout for coalesced memory access, managing thread workload to reduce code divergence, atomic operations, private copies of output, and shuffle instructions. You can use a combination of different techniques that may help improve the performance. The following papers address the implementation and optimization of parallel hash join algorithms that use the parallel radix partition. They may help you in understanding the basic idea and designing your code.

Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. "*Relational Query Coprocessing on Graphics Processors*". ACM Trans. Database Systems, 34(4), Article 21, Dec. 2009.

C. Balkesen, J. Teubner, G. Alonso, and M. T. Ozsu. "*Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware*". In Procs. of ICDE. pp. 362373, 2013.

C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. "*Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs*". Proc. VLDB Endow., vol. 2, no. 2, pp. 13781389, Aug. 2009.

Ran Rui and Yi-Cheng Tu. "*Fast Equi-Join Algorithms on GPUs: Design and Implementation*". In Proceedings of the 29th International Conference on Scientific and Statistical Database Management.

2 Tasks to Perform

Write a CUDA program to implement the radix partition, perform different experiments, and write a short report about your project. There should be three kernels implemented in your code:

1) histogram

Input: the input array of keys, the target number of partitions.
Output: the histogram of the partitions.

Algorithm 1 histogram

```
for each k in keys do
    h = bit extract(k,number of bits);
    histogram[h]++;
end for
return histogram;
```

2) prefix scan

Input: the histogram of the partitions.
Output: the exclusive prefix sum of the histogram.

Algorithm 2 prefix_scan

```
for i from 0 to number of partitions do
    if i == 0 then
        sum[i] = 0;
    else
        sum[i] = histogram[i-1]+sum[i-1];
    end if
end for
return sum;
```

3) reorder

Input: the array of keys, the prefix sum of the histogram.
Output: the reordered array of keys.

Algorithm 3 reorder

```
for each k in keys do
    h = bit extract(k,number of bits);
    offset = atomicAdd(&sum[h],1);
    output[offset] = k;
end for
```

To lower the overall difficulty of the project, kernel 2) is optional as you may choose to borrow the prefix scan code from the CUDA sample code located in /apps/cuda/cuda-10.2/samples/6_Advanced/shfl_scan. You will receive a 10% extra credit on this project if you implement the prefix scan by yourself. The final partition results and running time of the kernel(s) should be displayed as output. The input array consists of uniform-distributed random integers. We will provide the data generator and the GPU bit extraction function in the code template. Your program should work with a minimum input array size of one million elements and be able to process 2 to 1024 partitions (your code only needs to support the power of 2).

Input/Output of Your Program: Your program should take care of bad or missing inputs and throw appropriate errors in a reasonable way. This command should launch your program:

```
/apps/GPU_course/runScript.sh {filename} {#of_elements_in_array}  {#of_partitions}
```

where filename is assumed to be the name of your CUDA project file. The two arguments should be the size of the input array and the number of partitions respectively. Unlike project 2, we don't need the block size here. You have to determine the optimal block size in your code by experiments.

The output of your program should print out the pointer offset and the number of keys of each partition as the final result. Following the result, you should add a line to report the performance of your kernel, it should look like the following sample.

```
******** Total Running Time of All Kernels = 2.0043 sec *******
```

The time measurement mechanism is the same as Project 2.

Project Report

Write a report to explain clearly how you implemented the GPU kernels, with a discussion on what techniques you used to optimize performance. There is no specific format required for the report; just be sure to clearly explain your implementation.

3 Environment

All projects will be tested on the GAIVI cluster. If you prefer to work on your own computer, make sure your project can be executed on the GAIVI computers.

4 Instructions to Submit Project

You should submit one .cu file containing your implementation of the SDH algorithm and a separate .pdf file containing the report. You may include the timing.cuh file as well, if you choose to use it. Put all the files in a folder, zip the folder and name the zipped file proj3-xxx.zip (or proj3-xxx.tar if you prefer using TAR for compression), where xxx is your USF NetID. Submit the zipped file only via assignment link in Canvas. E-mail or any other form of submission will not be graded. Once you submit your file to Canvas, try to download the submitted file and open it in your machine to make sure no data transmission problems occurred. For that, we suggest you finish the project and submit the file way before the deadline.

5 Rules for Grading

Following are some of the rules that you should keep in mind while submitting the project. Functionality of project will be graded by a set of test cases we run against your code.

- All programs that fail to compile will get zero points. We expect your submission can be compiled by running a simple line of command as in Project 1&2.
- If, after the submission deadline, any changes are to be made to make the main code work, they should be done within 3 lines of code. This will incur a 30% penalty.
- All the required kernels must run on GPU. Use of any tricks to make it run on one thread or only on CPU will result in zero points.
- We will check the code related to performance measurement carefully. Any tricks played to report shorter running time than the actual one will be treated as cheating and incur a heavy penalty.