ECE 451-566 Parallel and Distributed Computing, Rutgers University, Fall 2018.

Instructor: Maria Striki **TA: George Chantzialexiou**

Project 3: Issued: 11-23-18 Due: Monday Dec 10th, 10:00 pm (120 points)

When you run your programs, you may compare them against other nodes in your group, if the specifications of other machines (nodes) are different. Again, please produce a detailed report that displays the results of your experimentation, describes your code, and answers all questions addressed.

TWEAK: I will allow 2 or 3 groups (depending on the number of members) to formulate super-groups and collaborate on the first 4 tasks only, but each groups will work separately towards continuing with her own password cracker.

Super-group Allocations:

1) Group 2, Group 7 and Group 5,
2) Group 4 and Group 8,
3) Group 3 and Group 6.

Remark 1: At the end of Project no 3 we expect single reports from the super-group members where the contribution of each member and the correspondence to the number and parts of the problems that they got involved with appears very clearly. In the same report, we expect you to include 2 or 3 different versions of the password cracker (problem 5), one for each separate group.

Remark 2: Although we will accept one single report from 2 or 3 group members, students will be graded based roughly: 1) mostly on their individual contribution and quality of individual work, and 2) somewhat also on the quality of the combined work. So, if a student has contributed to no problems or has provided a very superficial solution, will be graded differently than a student who has worked thoroughly on one or more problems. Since we do not have too much time, I am merging you into super-groups to allow you to partition work and have less load. The fact that you can partition work still makes all of you accountable for the quality of your report. You must regularly meet and check up on each other on the progress on their assignment and jump in to help if things do not go as expected.

Remark 3: At the back of this page there are details about the ORBIT CUDA environment set-up and use. I will also supply additional resources to read and samples of CUDA code, for you to start with.

PROBLEM 1 (20 points) MAX-MIN-MEAN-SDT DEV ELEMENTS

Part 1: Write a CUDA program that finds the maximum element in an N-element vector. Your input should be N, where N: 50,000,000 – 100,000,000 -150,000,000 – 200,000,000 and randomly generate a vector of real numbers (double-precision floats). Calculate the maximum value both in GPU and CPU, output the two values. Are they different or the same? Measure the time taken to produce the output in every case and justify the difference (if there is any). In the case of GPU measure the time taken to carry out memory operations as well (i.e., from the host to the GPU and back). However, when comparing the two methods, the overhead of copying the vector to the GPU and back should be neglected so that only execution times can be compared.

Part 2: In the same manner report the following statistics as well: minimum value of the vector, arithmetic mean of the vector, standard deviation of vector values.

Part 3: Do all the above concurrently in order to further enhance GPU performance. Compare the GPU vs. CPU performance now.

Questions: Please write down your reported results, produce the corresponding graphics if that facilitates the interpretation, and provide an interpretation of results for Part1-Part2-Part3. When addressing all Part1, Part2, Part3, for which elements action do you find the largest and for which the smallest difference: max, min, mean, standard deviation? And under which handling (separate or concurrent calculations)? Please provide adequate justification.

Remark: Floating-point addition is neither commutative nor associative. Therefore, it may happen that computing the mean on the GPU will produce a different result than on the CPU. The mean value computed by the GPU may be significantly more accurate for large vector sizes.

Problem 2 (25 points) (MATRIX MULTIPLICATION in different flavors)

Part 1: (Native MM): Write a CUDA program that conducts matrix multiplication by loading two matrices of random dimensions (but suitable for application and large enough for your GPU application to produce comparatively superior results BUT MAKE SURE THEY BOTH FIT IN GLOBAL MEMORY) and calculating the product: C = AB.

You should implement the matrix multiplication using both in GPU and CPU. Implement the matrix multiplication using the shared memory of the GPU.

1) In both cases profile your implementation and output the estimated amount of time in a similar fashion as in Part 1.
2) Calculate the estimated GFLOPS for both cases and compare.
3) Implement the matrix multiplication using the cuBLAS library. Also profile this implementation and compare your output to the result of the cuBLAS gemm function (using Mean Square Error function MSE).

Remarks: Perform the multiplication using cuBLAS and verify your result (even in the CPU implementation). It is preferable that your results match cuBLAS w.r.t. the mean squared error (MSE): $MSE = \sum(g[i] - x[i])\ 2\ N\ i$=1 where the array x is an array containing every element in the matrix. You can produce this function by looping through the final matrices and calculating the difference between each element, and squaring the result. This value should be very close to zero. It may be preferable to implement the multiplication in global memory first and verify the result before you continue with your shared memory implementation.

Part 2: (Tiled MM): There are many optimizations that can be made to this simple matrix multiplication program to ensure faster execution of your code and many other optimization benefits. The most popular optimization technique is Tiled Multiplication. You are expected to read about how tiled multiplication is conducted and solve the problem of part 1 and answer the questions of part 1 using Tiled Multiplication. Please show via experimentation and justify using theory what are the best sizes for the tiles you have used in your implementation and why. Compare the two methods of Part 1 and Part 2. Show by means of number and graphs what sort of improvement has been achieved and on what resources and performance metrics.

Part 3 (Loop Unrolling for MM). Yet another very popular technique for MM optimization is Loop Unrolling. You are expected to research and report about how Loop Unrolling is conducted and solve the problem of part 1 and answer the questions of part 1 using Loop Unrolling. Please show via experimentation and justify using theory what the optimal loop unrolling decisions you have made in your implementation are and explain why. Compare the methods of Part 1, Part 2, and Part 3. Show by means of number and graphs what sort of improvement has been achieved and on what resources and performance metrics.

Part 4 (Pick 1 more optimizations for MM and implement it). Conduct your own research in the Theory techniques for Faster/Optimized Matrix Multiplications and come up with a third optimization technique. Describe it and refer to your sources. Unless it is a technique of your own inspiration. Then implement it and conduct experiments and measurements as these in Part 1, Part 2, Part 3. Finally compare the results among the three optimization methods and of course compare results of GPU vs. CPU execution.

For Parts 1-4 in addition to your code and questions asked in the problem description, please produce a very detailed report that describes and explains your steps very clearly.

Problem 3 (PARALLELIZED BREAD FIRST SEARCH – BFS and Triangle Counting) (35 points)

Part 1 (BFS – 17 pts): As discussed already in class, highly recursive algorithms do not perform this well in parallel execution platforms. You're expected to alter the recursive code for the **Bread First Search (BFS)** to a non-recursive one (you may easily figure out how this can be done yourself or you are welcome to do some research on parallelizing Tree search algorithms) and implement it using CUDA. Your input should be a 2-dimensional matrix of dimensions > O(100) each, that represents the vertices and edges of a graph. The values of all vertices of the BFS tree should be non-negative integer values between 0 and 100. Hence, you may implement the absence of a node by setting its value to -1. Test your program for a number of random graphs. At the end, in addition to your code, provide the random graphs used in a file and also screenshots of your results.

1) Implement the sequential BFS code on a CPU using two versions: recursive and looped.
2) Implement the version of BFS code tailored to the GPU
3) Compare the speed of the GPU algorithm both to the CPU recursive and CPU looped.
4) Why does BFS lends itself more simply than DFS to a parallelized implementation? Please do some research and provide the main reasons, and also clearly describe what challenges we would deal with if trying to parallelize DFS, and which resources would be impacted by such implementation.

Part 2 (Parallelize Prim's Algorithm-18 points): Now that you had a first warm-up with modifying a traditional recursive algorithm to be suitable for efficient parallel execution, let's look at another traditional problem that could be modified to be efficiently handled by a massive parallel framework: Prim's Mininum Spanning Tree. You are requested to modify the original sequential algorithm and also the Pthreads parallel handling on Prim's SMT to run it as efficiently as you can on the CUDA parallel environment.

1) Implement the sequential Prim's code on a CPU using two versions.
2) Implement the version of Prim's code tailored to the GPU.
3) Compare the speed of the GPU algorithm both to the sequential and to the Pthreads parallel.
4) Why does Prim's lends itself more easily than Kruskal's algorithm for a parallelized implementation? Please do some research and provide the main reasons, and also clearly describe what challenges we would deal with if trying to parallelize Kruskal's, and which resources would be impacted by such implementation.
5) Theoretical question: What is Boruvka's algorithm? In what way does it relate to Prim's or Kruskal's? Please explain in details. What do we need to do to parallelize Boruvka's? Do not provide pseudo-code, just describe the main points you need to take care of to parallelize this algorithm.

Problem 4 (PARALLEL PREFIX SUM) (20 points):

You're asked to implement a parallel version of the function find_repeats using GPU cards, which given an array A of 1,000,000 integers, returns a list of all indices i for which A[i] == A[i+1]. For example, given the array A = {1,2,2,1,1,1,3,5,3,3}, your program should output the array B ={1,3,4,8} which has all the repeating indexes and array C={1,2,1,3,5,3} which eliminates all the repeating entries. Specifically, you need to complete the following steps:

1. You need to first generate the array A. Each array element is a random integer between 0 and 100. You don't need to parallelize this step.

2. First implement exclusive_scan, which takes an array A and produces a new array output that has, at each index i, the sum of all elements up to but not including A[i]. For example, given the array {1,4,6,8,2}, the output of exclusive prefix sum output={0,1,5,11,19}.

3. Once you've written exclusive_scan, implement function find_repeats, using exclusive_scan. This will involve writing more device code, in addition to one or more calls to exclusive_scan. Your code should output the array B and array C, and write the list of array B and C in two separate files, and then return the size of the output list. This can be executed on CPU.

4. Output: please output the exclusive_scan result of array A, find_repeats results B and C in three separate files (from CPU). On the display, please output the last element of the find_repeats result.

Your code will be tested for correctness and performance on random input arrays.

Important Notation: Parallel prefix sum and parallel exclusive scan will be covered in our lectures shortly, but along with the project I am attaching the basic sources which describe these two methodologies in detail.

http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html

http://www.cs.princeton.edu/courses/archive/fall13/cos326/lec/23-parallel-scan.pdf

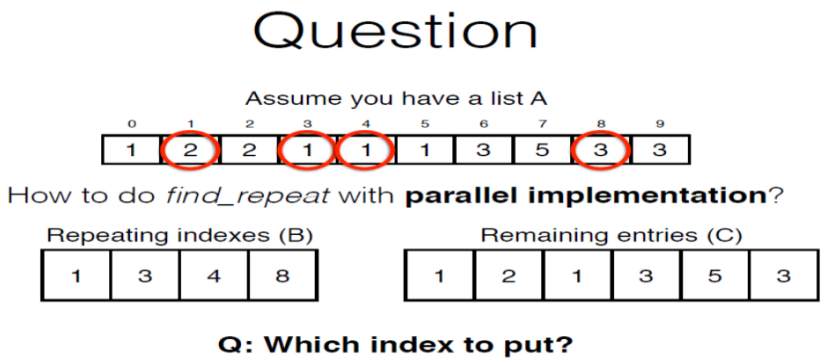
https://en.wikipedia.org/wiki/Prefix_sum

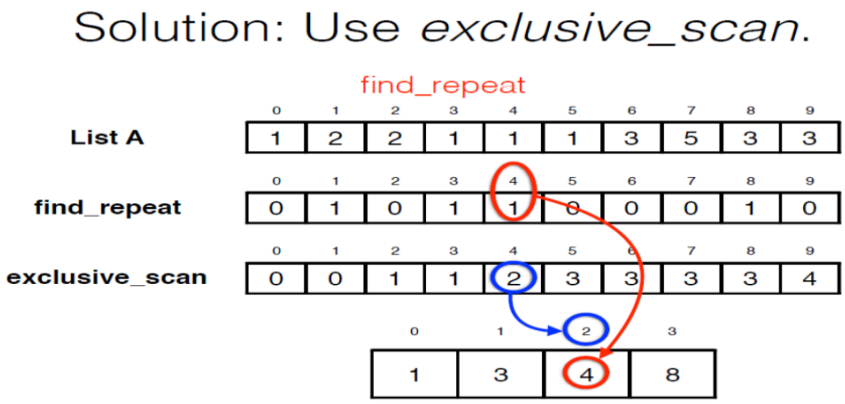
https://www.cs.cmu.edu/~guyb/papers/Ble93.pdf

http://www.eecs.umich.edu/courses/eecs570/hw/parprefix.pdf

And there are many more related resources on the web. The above are just a few indicative ones.

Notation for Part 4:





Report Requirement:

Turn in a report that uses two different GPU cards for executing your GPU codes and comparing against the CPU codes. Along with your results please provide performance analysis that justifies your findings.

PROBLEM 5: PASSWORD CRACKER (20 points):

Implement a simple **password cracker** using CUDA. You are going to make a password which is between 1-8 ascii characters. You will then use one of the popular hash functions to convert your password to a hash. You may use the same algorithm and implementation details for the password cracker you built for Project 1 (ISPC) and Project 2 (Pthreads), only now you are implementing your Password Cracker in CUDA. The attacker (you-the group-the programmer) is exposed only to the hash of your password and to no other information. You are to implement your own password cracker based on the rules above. You are provided the range of the password length, which is to be between 1-8 characters. Provided this information, your group

should brute force the password, i.e., take every possible combination of words between the mentioned length, convert it into a hash and compare it with the provided hash. This task can be easily parallelized.

Note 1: In your report please mention clearly what the design and implementation differences are with CUDA when compared to your prior password cracker with ISPC and Pthreads (or whichever of the two you have managed to implement, if any). Write a table where you report a comparative performance evaluation of timings: CUDA vs. Pthreads vs. ISPC. Which version of your password cracker is faster? Why?

Note 2: Also experiment with a number of different password characters and report the difference in the timing results.

All groups in addition to your own imaginary passwords, also use the following to compare against all groups and all varying implementations: bv37qi#f. Apply a number of popular hashes and obtain a number of versions of the password. And you take it from there…

Environment Setup:

This assignment requires an NVIDIA GPU with CUDA or GPU that can run OpenCL. Most NVIDIA cards support CUDA, so if your computer has one, you can download the SDK and develop in your computer. Most MacBook pros have NVIDIAs, as well as a lot of mid-high end laptops, it may be worth checking. However, the ORBIT environment may still be optimal as it contains several very powerful machines with multicore GPUs. Details can be found in the following URL:

http://www.orbit-lab.org/wiki/Hardware/fDevices/gCUDA#CUDA

Below, I provide illustrated the types of GPU machines supported in ORBIT and the grid/sandboxes these are located:

Model	# of CUDA Cores	Memory (GB)	Memory Bandwidth (GB/s)	Single-Precision Performance	Double-Precision Performance	More Info
GeForce GT 640 GDDR5	384	1	40.1	803 Gflops	*unknown*	1 2
GeForce GTX 750 Ti	640	2	86.4	1306 Gflops	*unknown*	1 23

Quadro 2000D	192	1	41.6	480 Gflops	*unknown*	1 23
Quadro K5000	1536	4	173	2.1 Tflops	*unknown*	1 23
Tesla C2050	448	3	144	1.03 Tflops	515 Gflops	1 23
Tesla K40	2880	12	288	4.29 Tflops	1.43 Tflops	1 2
Tesla K80	4992	24	480	8.73 Tflops	2.91 Tflops	1 23

Domains with CUDA Capabilities

Domain	Node: CUDA Card(s)
SB1	node2-1: GeForce GTX 750 Ti
SB4	node2-1: GeForce GT 640 GDDR5
SB8	node2-1: Quadro K5000
SB9	node2-1: Quadro 2000D
Grid	node21-1: 2x Tesla K40, node21-2: Tesla K80, node21-3: 2x Tesla C2050

Hence, we have 1 machine per sandbox for four sandboxes, and 3 machines in Grid.

In total we have 7 machines supporting NVIDIA.

You are 11 groups. This is to say that you can make an arrangement among each group to be reserving specific sandboxes every day in a round robin fashion perhaps.

Remark: This time I will make reservations to the grid only but you are responsible for making reservations to the sandboxes that contain NVIDIA equipped machines.

Please respect your colleagues and do not reserve the sandboxes for very long hours. Also, consider that these resources are being used by individuals, institutes and scientific labs around the world.

Try to work and experiment on all resources listed above, and if you find that there are issues with any of them (they are not working, they do not have the proper hw/sw installed) please let me and Mr George Chantzialexiou know ASAP. I will contact Mr Ivan Seskar immediately and he will see if he can fix the problem ASAP.

Environment:

Please check the wiki page in ORBIT under:

http://www.orbit-lab.org/wiki/Courseware/aParDist/Project3

There you can find the details of how we downloaded the required tools to NVIDIA so that now both CUDA and OpenCL are installed and example programs are already compiled.

The downloads and installations may no longer be saved under the base mariassecondimage. So, if you find that mariassecondimage does not have the proper folders and files you need, please use mariasfirstimage to download and install the required folders. And if you need extra material to upload on this image, do not forget to replicate the image to another name and store your work there.