ECE 451-566 PDC, Rutgers University, Fall 2020. Instructor: Maria Striki

Project 3: Issued: 11-25-20 Due: Monday Dec 12-14-20, 9:00 pm (60 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. Please put together a detailed report that displays the results of your experimentation, describes your code, answers questions.

A group with size >=2 will do 1 problem to completion and sketch a second problem, i.e. document the algorithm, the code or pseudo-code, write the insights, but do not have to do the experiments, do not have to do it to completion.

A group with size 1 will do 1 problem to completion. If you want to work together with a group of size 1 or 2 I will allow you to work together.

Remark: At the back of this page there are details about the ORBIT CUDA environment set-up and use. I will be also supplying additional resources to read and samples of CUDA code.

PROBLEM 1 (35 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 (35 points) (MATRIX MULTIPLICATION in different flavors)

Part 1: (Native MM): Write a CUDA program that does 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:

a.) only your CPU
b.) GPU and Global Memory only,
c.) GPU and Shared Memory only (Global where needed).

The cuBLAS Library provides a GPU-accelerated implementation of the basic linear algebra subroutines (BLAS). cuBLAS accelerates AI and HPC applications with drop-in industry standard BLAS APIs highly optimized for NVIDIA GPUs. The cuBLAS library contains extensions for batched operations, execution across multiple GPUs, and mixed and low precision execution. Using cuBLAS, applications automatically benefit from regular performance improvements and new GPU architectures. The cuBLAS library is included in both the NVIDIA HPC SDK and the CUDA Toolkit.

1) In all cases profile your implementation and output the estimated amount of time taken.
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.

Comparison: Compare the results of Parts 1-2 and of course compare results of GPU vs. CPU execution. For Parts 1-3 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 (PARALLEL PREFIX SUM) (35 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}.

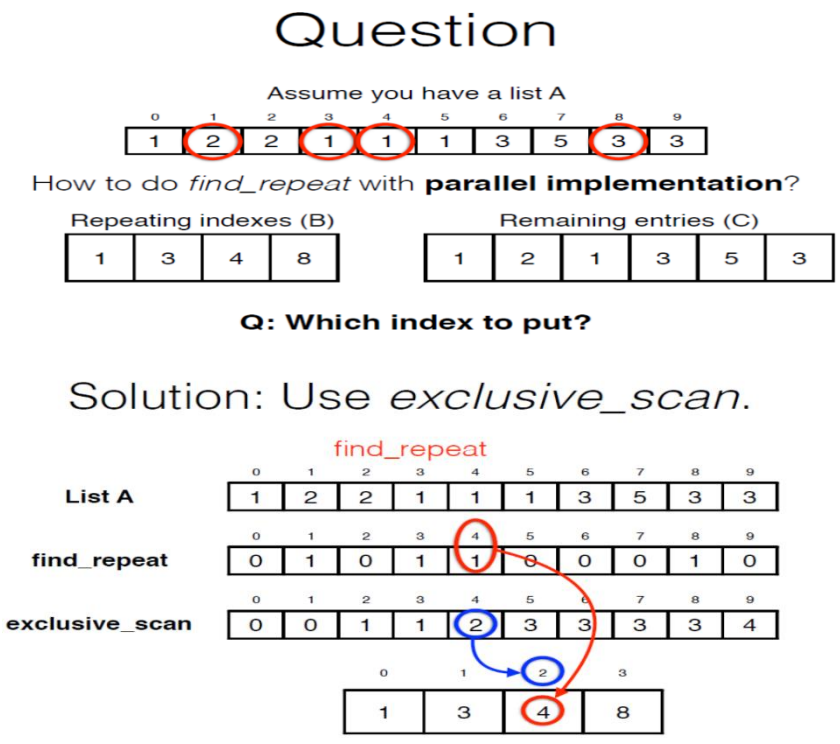
3. Once you've written exclusive_scan, implement function find_repeats, using exclusive_scan. This will involve writing more device code, in addition to one or more calls to exclusive_scan. Your code should output the array B and array C, and write the list of array B and C in two separate files, and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

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

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



PROBLEM 4: PASSWORD CRACKER (35 points):

Note: Only the Groups that implemented Password Cracker in Project 2 can select this project.

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 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 Pthreads/ISPC. Write a table where you report a comparative performance evaluation of timings: CUDA vs. Pthreads. 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: But also check yourselves because there are many updates every couple of months.

Available Cards

Model	Architecture	# of CUDA Cores	Memory (GB)	Memory Bandwidth (GB/s)	Single-Precision Performance	Double-Precision Performance	Compute Capability	More Info
GeForce GT 640 GDDR5	Kepler	384	1	40.1	803 Gflops	~33 Gflops *(unofficial)*	3.5	1 2
Quadro K5000	Kepler	1536	4	173	2150 Gflops	~90 Gflops *(unofficial)*	3.0	1 2 3
Tesla K40	Kepler	2880	12	288	4.29 Tflops	1.43 Tflops	3.5	1 2
Tesla K80	Kepler	4992	24	480	8.73 Tflops	2.91 Tflops	3.7	1 2 3
Tesla P100 12GB	Pascal	4992	12	549	9.3 Tflops	4.7 Tflops	6.0	1 2 3
Tesla V100 16GB	Volta	5120	16	900	13 Tflops	7 Tflops	7.0	1 2 3

Domains with CUDA Capabilities

Domain	Node: CUDA Card(s)
sb4.orbit-lab.org	node2-1: GeForce GT 640 GDDR5
grid.orbit-lab.org	node21-1 to node21-7: Tesla K80 node21-9 to node21-32: Tesla P100
COSMOS	see COSMOS testbed for details

You may be reserving the sandboxes separately per group, but I will keep reserving the ORBIT grid for the class.

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 know ASAP. I will contact Prof. 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.