Project 4: Tree reduction and stencil computation in GPU (100 points)

Submission guidelines:

You will submit **2 files** in Canvas.

- a PDF report file (filename: **Project_4_Report.pdf**)
- a ZIP file (filename: **LastnameFirstname_Project4.zip**) containing:
  - Problem_1/
    - gpu_mining_problem1.cu
    - nonce_kernel.cu (This is provided)
    - hash_kernel.cu (Write this)
    - Makefile
    - support.cu (If you use it for timing)
    - support.h (If you use it for timing)
  - Problem_2/
    - gpu_mining_problem2.cu
    - nonce_kernel.cu (This is provided)
    - hash_kernel.cu (This is from Problem 1)
    - reduction_kernel.cu. (Write this)
    - Makefile
    - support.cu (If you use it for timing)
    - support.h (If you use it for timing)
  - Problem_3/
    - convolution_CUDA.cu
    - kernel.cu
    - Makefile
    - support.cu (If you use it for timing)
    - support.h (If you use it for timing)
  - Problem_4/ (If graduate)
    - convolution_maxpooling_CUDA.cu
    - kernel.cu
    - Makefile
    - support.cu (If you use it for timing)
    - support.h (If you use it for timing)

Test data:

The test data are provided in your zip file and on Schooner at:
”/home/oucspdn_ta/PDN2022/test_data/Project_4_Tests/”.

GPEL Machines:

For this project, you are given access to several remote GPEL machines. Accessing these machines is very similar to Schooner, except you do not need to run your programs using batch jobs. You may run any program, CPU or GPU, on the command line. Refer to the *gpel-howto.txt* document or the instructional video on Canvas to learn how to access them. Please keep in mind that if you do wish to use Schooner, your program runtimes may be slower than the alternative.

Problem 1. For CS4473 (40 points) and CS5473 (20 points)

We are going to develop a new cryptocurrency, called PDNcoin™. At the Initial Coin Offering (ICO), everyone will be able to buy one PDNcoin for one US dollar. Only a trillion PNDcoins will be offered to create artificial scarcity. We expect that one PDNcoin will be worth $1000 in 5 years given the current inflation rate of the fiat money. Please help us develop the PDNcoins and Make the World a Better Place!

Mining the PDNcoins will be similar to mining the bitcoins. Please read the blog post below to understand the concepts like **block**, **transactions**, **nonce**, **hash**, and **target**.
https://medium.com/coinmonks/complex-puzzle-in-bitcoin-mining-aint-complex-no-more-9035b25b2a10

You may look at a concrete example of mining a block in bitcoin
https://www.blockchain.com/btc/block/00000000000000000003d05ef31993d1ddb80b6ef5632d0ae939ea1b22a24e150
This block contains 1068 transactions. The winning nonce value is 554,703,974. The hash of this winning nonce value contains 18 leading zeros, which is smaller than the target. The miner who hit this nonce value earned about half a million dollars for successful mining this block.

You are provided with a serial C program, `serial_mining.c`, that mines the PDNcoins. This program takes a block of transactions as an input file and tries a certain number of randomly-generated nonce values. For each nonce value, the program computes a hash value from the block of transaction using a simple hash function. This hash function generates a hash given from the transactions, nonces, and hash-array index. The index was used in order to attain more hash-value 'randomness' in the program. For simplicity, a transaction is represented by an integer and a block of transaction is a list of integers. The nonce values are also integers. The hash function is based on the modulus of a large prime number. To successfully mine a block of PDNcoin transactions, the program must find a nonce value that can generate a hash value less than the target (e.g., 10).

After loading a block of transactions, a PDNcoin mining program attempts to mine it in 3 steps:
Step 1: generate the nonce values using a random number generator
Step 2: generate the hash values using the hash function
Step 3: find the nonce with the minimum hash value

Let us accelerate this PDNcoin mining program using GPU in this project. You are provided with a starter CUDA program, `gpu_mining_starter.cu`. This starter CUDA program off-loads the Step 1 to the GPU using a kernel function from `nonce_kernel.cu.` It still performs the Step 2 and Step 3 on the CPU using the same code as the serial program. In this project, let's off-load the Step 2 to GPU in the Problem 1 and then off-load Step 3 to GPU in Problem 2.

In the Problem 1, please write a kernel function called `hash_kernel.cu` to generate the hash value array on the GPU. Please change the CUDA program to `gpu_mining_problem1.cu` which performs Step 1 and Step 2 on the GPU and Step 3 on the CPU. The gpu mining program should carry out the following procedure for off-loading the hash computation

1. Keep the nonce value array on the device memory
2. Copy the transaction array to the device memory
3. Allocate the hash value array on the device memory
4. Launch the hash kernel function to compute the hash value array
5. Copy the hash value array to the system memory, which allows you to use the serial code to find the min hash and min nonce

Your CUDA program should be run as the starter CUDA program like this:

```
gpu_mining transactions.csv n_transactions trials out.csv
time.csv
```

Where:

- `transactions.csv`: A block of transactions
- `n_transactions`: The number of transactions in this block (20,000 for the provided transaction.cvs in the test data)
- `trials`: the number of nonce values to try
- `out.csv`: the minimum hash value and its nonce value found in this run
- `time.csv`: the runtime

Please first compile and run the starter CUDA program to make sure that you have the correct programming environment setup. Again, it is recommended that you use the GPEL machines for your computation.

Learning outcome:
This is designed to practice the embarrassingly parallel pattern on GPU.

What to submit:

In the ZIP file, please include submit all the code as specified on the first page, including the new hash_kernel.cu and gpu_mining_problem1.cu.

Please benchmark the performance of your program using the provided transaction block file that contains 20,000 transactions. In the report, please provide the wall-clock runtimes for 5 millions trials and 10 million trials.

Implementation	**Steps on GPU**	**5 million trials**	**10 million trials**
serial_mining.c	None		
gpu_mining_starter.cu	Step 1		
gpu_mining_problem1.cu	Steps 1 and 2		

In the report, please discuss the speedup provided by the CUDA parallelization and the parallelization overhead incurred by CUDA, in comparison with the serial program.

Grading rubrics
CS4473
- 25 points for CUDA parallelization
- 15 points for the report

CS5473
- 15 points for CUDA parallelization
- 5 points for the report

Problem 2. For CS4473 (40 points) and CS5473 (40 points)

To achieve the maximum acceleration, we need to off-load as much computation as possible to GPU and minimize the data transfer between the device memory and the system memory. In this problem, please off-load the Step 3 of our PDNcoin mining program to the GPU.

Please write a kernel function called `reduction_kernel.cu` to find the min hash and min nonce from the hash value array and the nonce value array on the GPU. Please change the CUDA program to `gpu_mining_problem2.cu` which performs Step 1, Step 2 and Step 3 on the GPU and Step 3 on the CPU. For simplicity, you just need to do one round of reduction on the GPU, which will reduce the size the array by a factor of 2 times the block size. The remaining reduction can be done by the host thread. The gpu mining program should carry out the following procedure for off-loading the reduction computation

1. Keep both the nonce value array and the hash value array on the device memory
2. Launch the reduction kernel to find the local min hash values and local min nonce values. If there are T trials and the block size is B, the reduction kernel should find T/(2*B) local min hash values and T/(2*B) local nonce values.
3. Copy these local min hash values and local min nonce values to the system memory
4. Find the global min hash values and min nonce values serially using the CPU.

Hint: please review the CUDA reduction program in Chapter 3 - Section 3 and Exercise 2.3.2. You may also read the example code in the code-reduction folder on Canvas. This program performs one round of reduction from the original array to an output array containing the partial sums and then adds up the partial sums to the global sum in the host code. You just need to change the reduction from the addition operation on one array to the minimum operation on two arrays. Below are a few caveats when you upgrade the sum reduction to the min reduction.

- Please keep track of both the minimum hash value and its corresponding nonce value during the reduction
- The initialization value for the reduction should be 0 for addition and UINT_MAX for minimum of unsigned int.

Your program should be run as the CUDA starter program with the same parameter list.

Learning outcome:
This is designed to practice the **tree reduction pattern** on GPU and the use of shared memory.

What to submit:

In the ZIP file, please include submit all the code as specified on the first page, including the new reduction_kernel.cu and gpu_mining_problem2.cu.

Please benchmark the performance of your program using the provided transaction block file that contains 20,000 transactions. In the report, please provide the wall-clock runtimes for 5 millions trials and 10 million trials.

Implementation	**Steps on GPU**	**5 million trials**	**10 million trials**
serial_mining.c	None		
gpu_mining_starter.cu	Step 1		
gpu_mining_problem1.cu	Steps 1 and 2		
gpu_mining_problem2.cu	Steps 1, 2 and 3		

In the report, please discuss the speedup (or lack thereof) provided by this implementation over the previous one in Problem 1.

Grading rubrics

- 30 points for the CUDA implementation
- 10 points for the report

Problem 3. For CS4473 (20 points) and CS5473 (20 points)

GPU has been used extensively to speed up deep learning. Convolution is a key operation in convolutional neural networks (CNN) (for an intuitive explanation, please read https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Please implement a simple convolution layer for a gray-scale image using CUDA. We provided a serial C program for this. The input gray-scale image is a matrix of unsigned integers. In a CNN model, the values in the convolution filers are learned from the training data. For simplicity, let us use a fixed convolution filter in this exercise. Below is the convolution filter that can highlight the diagonal strip patterns in the input image:

1	0	0	0	1
0	1	0	1	0
0	0	1	0	0
0	1	0	1	0
1	0	0	0	1

Please use a stride size of 1 and the zero padding around the four sides of the image. You will need to pad 2 extra rows or columns on each side of an image to accommodate the entire convolution filter when computing on the border pixels.

Below is a pseudo-code for the CUDA convolution layer implementation.

1. **Transfer the input image (the A matrix) to the device memory**
2. **Transfer the convolution filter (the K matrix) to the device memory**
3. **Launch the convolution kernel to compute the filter map (the B matrix) by applying the convolution to every pixel in the input image.**
4. **Transfer the filter map (the B matrix) from the device memory to the system memory.**

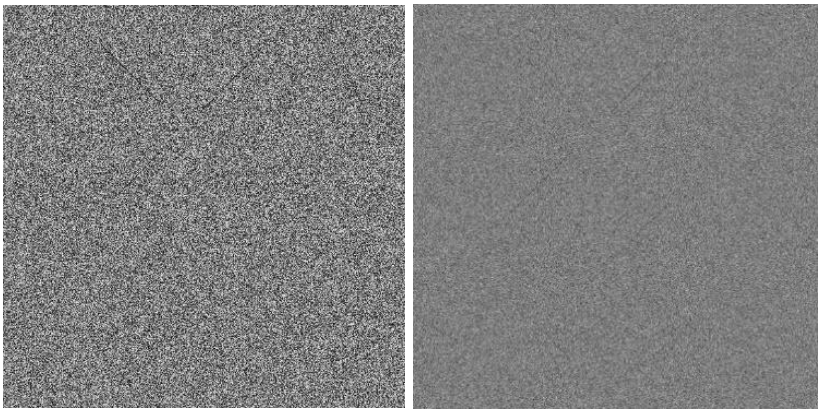
Data transfer between the device memory and the system memory is a major overhead for off-loading computation to the GPU. Please benchmark the wall-clock times for [1] transferring the input data from the system memory to the device memory in steps 1 and 2, [2] running the kernel computation in step 3, and [3] transferring the output data from the device memory to the system memory in step 4. Remember that the host thread returns immediately after the kernel launch. In order to time the kernel computation, you need a synchronization barrier after launching the kernel and before taking the finish time.

Hint: Please review the image blurring CUDA program covered in Chapter 2. The blurring operation is basically a convolution operation using the convolution filter below:

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

This blurring program also used a stride size of 1 and the zero padding around the four edges of the image.

Input image: Output image:



Full images are available in test_data/ folder. If you zoom in, you will be able to see some diagonal strips in the input image. The output image has less noise and still retains the diagonal strips.

Images were obtained after using make_image.py code. This program converts the csv file to a jpg image.
Use like this, e.g.:
python make_image.py mat_input.csv mat_input.jpg

You are provided with a serial implementation. You have an input matrix (**mat_input.csv**) in the test_data/ folder. It has the dimension of 2048x2048.

Your program should be run like this:

```
convolution_CUDA_parallel n_row n_col mat_input.csv
mat_output_prob3.csv time_prob3_CUDA.csv
```

Where:

- n_row, n_col: number of rows and columns, respectively
- mat_input.csv: matrix input in csv format
- mat_output_prob3.csv: matrix output in csv format
- time_prob3_CUDA.csv: You would output 3 wall-time times in csv format (1 column, 3 rows) to fill in the time table in the report

e.g.
0.5
0.7
0.5

Learning outcome:

This is designed to practice the **stencil computation pattern** on GPU. The convolution filter considered here is a 2D 25-point stencil.

What to submit:

In the ZIP file, please include your implementation:

- convolution_CUDA.cu: the main function of your CUDA program
- kernel.cu: the kernel function of your CUDA program

In the report, please provide the runtime for the provided C serial program and your parallel CUDA program.

Implementation	Wall-clock runtime
Original serial program	
CUDA: Copying data from host to device	
CUDA: Launching kernel	
CUDA: Copying data from device to host	

Please review Chapter 3 and do the following. First, estimate the compute-to-global-memory-access ratio of your CUDA program. Second, provide the pseudo-code for using the tiling technique in your kernel function to take advantage of the shared memory. Third, estimate the compute-to-global-memory-access ratio of your tiled convolution kernel.

Grading rubrics

- 15 points for CUDA implementation
- 5 points for the report

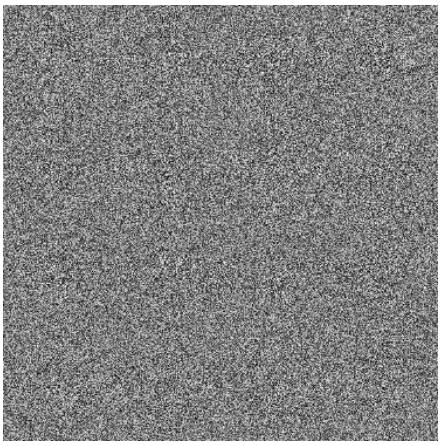
Problem 4. Only For CS5473 (20 points)

In CNN, the convolution operation is typically followed by the max-pool operation. Max-pool is also explained in https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

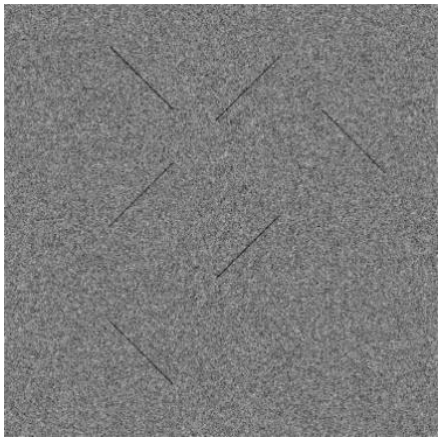
Here, please expand your CUDA program from Problem 3 and add the max-pool operation. The output matrix from the convolution operation would become the input matrix for the max-pool operation. For simplicity, please use a max-pool window of size 5x5 and a stride size of 1. Please see the provided serial program for more information about the max-pool operation.

Below is the example output. The diagonal strips become much clearer after convolution and max-pooling.

Original input image:



Output image after convolution and max-pooling:



The input matrix and full images are available in test_data/ folder.

Your program should be run like this:

```
convolution_maxpooling_CUDA n_row n_col mat_input.csv
mat_output_prob4.csv time_prob4_CUDA.csv
```

Where:

- n_row, n_col: number of rows and columns, respectively
- mat_input.csv: matrix input in csv format
- mat_output_prob4.csv: matrix output in csv format
- time_prob4_CUDA.csv: You would submit 4 runtimes in csv format (1 column, 4 rows) as below:

0.05
0.07
0.05
0.07

Learning outcome:
You will learn how to perform 2 consecutive stencil computation on GPU.

What to submit:

In the ZIP file, please include your implementation:

- convolution_maxpooling_CUDA.cu: the main function of your CUDA program
- kernel.cu: the kernel functions of your CUDA program

In the report, please provide the runtime for your convolution maxpooling CUDA program.

Implementation	**Wall-clock runtime**
Original serial program	
CUDA: Copying data from host to device	
CUDA: Running kernel 1 for convolution	
CUDA: Running kernel 2 for max-pooling	
CUDA: Copying data from device to host	

Please also think about how to use tiling to improve these two kernels. First, estimate the compute-to-global-memory-access ratio of your CUDA program with the two kernels. Second, provide the pseudo-code for using the tiling technique in the two kernel functions to take advantage of the shared memory. Third, estimate the compute-to-global-memory-access ratio of your tiled convolution and max-pool kernels.

Grading rubrics

- 15 points for CUDA implementation
- 5 points for the report