All exercises must be done by yourself. You may discuss questions and potential solutions with your classmates, but you may not look at their code. If in doubt, ask the instructor.

Acknowledge all sources you found useful.

Your code should compute the correct results.

Partial credit is available, so attempt all exercises.

Submit your answers as a PDF file.

The compressed archive (e.g. ZIP) file you upload to Blackboard should have your name in the filename, e.g. `JRandomStudentA3.zip`

Exercise 1

(Parallel Binary Search) Write a program to search through an array $haystack$ (of size H), for numbers from another array $needle$ (of size N). Your program will be invoked like this:

```
parbinsearch haystackfile.txt needlefile.txt outputfile.txt
```

Both `haystackfile.txt` and `needlefile.txt` contain an array of numbers. The first line contains the total count of numbers in the file, and the remaining lines contain the numbers. This is the same file format used by `reduction` in Assignment 1, and you can reuse that code in your solution.

Each line of `outputfile.txt` should be in the format n pos, where n is a number from $needle$ and pos is the position (array index) in $haystack$ where it was found. Use -1 if the number was not found.

For example, assuming the haystack file looks like this (it will always be sorted):

```
3
1
2
3
```

and the needle file contents look like this:

```
2
5
1
```

Your output should be:

```
5 -1
1 0
```

1. Write a GPU implementation of parallel binary search. Call this `pbs`. Distribute work so that one GPU thread searches for one needle. Report time taken on the supplied inputs for the kernel only (use `nvprof`).

2. Sort needle (on the CPU) before performing parallel binary search. Call this `pbs-sort`. You can use standard library functions to perform the sorting. Report the time taken for the kernel on the supplied inputs.

3. Distribute work so that each warp handles one needle. Call this `pbs-warp`. One way to do this is to let the first thread of each warp perform the binary search while the others are idle. Report the time taken for the kernel now on the supplied inputs.

4. *(Extra credit)* Examine the metrics that *nvprof* provides, and report metrics that measure warp divergence and predication information for each variant above.

Exercise 2

Maximum of an array Write a program to calculate the maximum element of an array. You can build on the chunked maximum example discussed in class.

IMPORTANT: All calculations must be done on the GPU. For example, you *should not* execute the last serial part on the CPU.

You can do this either as a two-step procedure (one parallel and one serial [i.e. a GPU kernel running one thread in total]) or as a multistep tree reduction (multiple parallel and one serial). Note that the GPU does not support global barriers (i.e. between thread blocks). However, two kernels launched on the same stream have an implicit barrier between them.

Your program will be called like this:

```
maxprogram arrayofnumbers.txt
```

The file `arrayofnumbers.txt` will have the same file format as in Exercise 1. Your program should print out the following maximum value, assuming it was passed the example needle file from Exercise 1.

```
5
```

1. Write a variant called `gpumax` which operates on one element per thread. Report the time taken for the kernel using `nvprof`.
2. Write a variant called `gpumax-multi` which operates on multiple elements per thread. Report the time taken for the kernel using `nvprof`. Note, a correct solution will use fewer barriers than the `gpumax` for the same input. You can fix the number of elements per thread at compile time if you wish.

END.