The University of Birmingham
School of Computer Science

Assignment 02 CUDA — THIS IS AN ASSESSED ASSIGNMENT

1 Summary

The deadline for this assignment is **Monday December 2nd at 9:00**

This assignment will be marked out of 10 and contributes

- 10% of the total module marks for 06-26945 Distributed and Parallel Computing
- 10% of the total module marks for 06-26944 Distributed and Parallel Computing (Extended)

In this paper, Mark Harris describes the implementation of **Exclusive** Sum Scan (Sum Prescan) on NVidia GPUs:

- Web version: `https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.html`
- PDF version: `http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/scan/doc/scan.pdf`

Write a CUDA program that implements the work efficient **exclusive** scan described and demonstrate it by applying it to a large vector of ***integers***. Note that the code in the papers is for a float scan, whereas you are required to implement an integer scan. The integers should be in the range 0 to 9 inclusive (use `rand()%10` to generate the integers).

Your full large vector scan should be written as a host function that, depending on the size of the input vector, chooses whether to do a 1 level block scan, a 2 level or a 3 level scan.

Marks will be assigned as follows:

06-26945 Distributed and Parallel Computing:

- 3 marks for getting the basic work-efficient, **exclusive** block scan working **as described in the paper**.
- 4 marks for getting full scan for large arrays working (second and third level scans required, test vector should be of length 10,000,000)
- 3 marks for getting the shared memory bank conflict avoidance optimization working: note that this should be for GPUs, such as those in the school labs, that have 32 banks, instead of the 16 banks of the GPUs described in Harris' paper.
- Note that there are no extra marks for supporting data segment sizes in the scans that are twice the size of the thread block size, although students are encouraged to try this if they have the time and inclination — just make sure you keep your original version without this feature to submit if you run into difficulty with it.

06-26944 Distributed and Parallel Computing (Extended):

- 2 marks for getting the basic work-efficient, **exclusive** block scan working **as described in the paper**.
- 4 marks for getting full scan for large arrays working (second and third level scans required, test vector should be of length 10,000,000)
- 2 marks for getting the shared memory bank conflict avoidance optimization working: note that this should be for GPUs, such as those in the school labs, that have 32 banks, instead of the 16 banks of the GPUs described in Harris' paper.
- 2 marks for supporting data segment sizes in the scans that are twice the size of the thread block size.

Your submission should include:

- Your implementation as a single "`.cu`" program code file
- A comment at the top of the file that reports:
  - Your name and student id number
  - Which of the assignment goals you achieved:
    - block scan
    - full scan for large vectors
    - Bank conflict avoidance optimization (BCAO)
  - Your time, in milliseconds to execute the different scans on a vector of 10,000,000 entries:
    - Block scan without BCAO
    - Block scan with BCAO
    - Full scan without BCAO
    - Full scan with BCAO

  - The model of CPU and GPU that you ran your timings on
  - A short description of any implementation details or performance improvement strategies that you successfully implemented and which improve upon a base level implementation of the target goals.

1.1 Notes

- The scan should be an **integer** scan that generates an **integer** result vector. This means you can easily test the scanned result without having to worry about round-off errors.
- Note that you can get the marks for the BCAO part even if you don't get the full scan working and vice versa
- You can get half marks (2/4) for getting the 2 level full scan working (on a smaller input vector) even if you can not get the 3 level version working.
- Optimise and fine tune your program to get the best speedup possible. Note that this includes re-structuring your code as well as trying different configuration parameters.
- You may use any code from the referenced paper, any of your own code from your previous assignments and any code from the lecture materials. You may **NOT** use code from any other source and you may **NOT** collaborate with any other person in developing your solution.
- Your submitted code **MUST** calculate and print to standard out the times in milliseconds of the different scans you implement.
- Your code **MUST** use host functions to calculate host versions of all of the GPU scans you implement and **MUST** compare the host version result array against the GPU version result array to check correctness for each scan that you implement and report the results to standard out.
- Remember, when doing your final timings, to compile and run your program in **RELEASE** mode, not in **DEBUG** mode.
- Upload your final code with the comment reporting your results via Canvas

1.2 Further Points

- The program must NOT take any command line args or read input from any input file or stream
- The program must do the following, IN ORDER AND WITHOUT requiring multiple runs or any changes to constants, values or variables:
  - The program MUST time and report the timing for **HOST ONLY** sequential **BLOCK** scan on a 10M array of random ints in the range 0 to 10. Only calling it on a single block or segment is NOT sufficient.
  - The program MUST time and report the timing for **HOST ONLY** sequential **FULL** scan on a 10M array of random ints in the range 0 to 10. Only calling it on a single block or segment is NOT sufficient.
  - The program MUST time, check for correctness, print the result of the check (i.e. passed or failed), and report the timing for a GPU block scan on a 10M array of random ints in the range 0 to 10. Only calling it on a single block or segment is NOT sufficient.
  - The program MUST time, check for correctness, print the result of the check (i.e. passed or failed), and report the timing for a GPU block scan with BCAO on a 10M array of random ints in the range 0 to 10. Only calling it on a single block or segment is NOT sufficient.
  - The program MUST time, check for correctness, print the result of the check (i.e. passed or failed), and report the timing for a GPU full scan on a 10M array of random ints in the range 0 to 10
  - The program MUST time, check for correctness, print the result of the check (i.e. passed or failed), and report the timing for a GPU full scan with BCAO on a 10M array of random ints in the range 0 to 10
- Timings for GPU kernels or sequences of GPU kernels should be done with cuda event timers.
- Timings for purely host code should be done with the sdkTimer methods.
- For all timings, make sure you do not include ANY computation other than the intended host function or the intended sequence of GPU kernel calls in what gets timed.
- You must NOT pad the size of the input array: your code should work on an input array as given up to and including an array of 10,000,000 entries. If called on an array of any size up to that max limit, it must NOT increase the size of it in any way. Your code is essentially a library function which can be called by the user on the user's input array. You should **NOT** slow down your code by copying it into a larger array and you cannot restrict the user to use an array with size which is a multiple of your chosen block size or to a size which is a power of 2.
- Avoid complicated use of macros to organise your code or to avoid repeating code. C++ provides plenty of structure to organise your code well without macros. In general, use macros to define constants, to handle cuda errors and to switch on or off sections of debug code. Other than that, do not use them.

2 Bugs and Errors in the Paper

There are a number of bugs and errors in the paper. The most serious one is that the BCAO macros the paper has a bug. It should be:

```
#define NUM_BANKS 32
#define LOG_NUM_BANKS 5

#define ZERO_BANK_CONFLICTS
#ifdef ZERO_BANK_CONFLICTS
#define CONFLICT_FREE_OFFSET(n) ( ((n) >> LOG_NUM_BANKS) + ((n) >> (2 * LOG_NUM_BANKS)) )
#else
#define CONFLICT_FREE_OFFSET(n) ((n) >> LOG_NUM_BANKS)
#endif

// You need extra shared memory space if using BCAO because of
// the padding. Note this is the number of WORDS of padding:

#define EXTRA (CONFLICT_FREE_OFFSET((BLOCK_SIZE * 2 - 1))
```

In practice, it turns out the `ZERO_BANK_CONFLICTS` version does more calculations for very little gain, so you will be better off leaving it undefined and just using the simpler version of the macro.

Otherwise, note that the block scan described in the paper **ONLY** works for a **SINGLE** block's worth of data. You will need to fix it so that it does a block scan for **EVERY** block of data in the input array, and make sure that it handles arrays whose length is not a multiple of the block size: it is **NOT** acceptable to pad the array to such a multiple: you must handle the array of the given size.

Be very careful with the fact that you are doing an **integer** scan, not a float scan: missing places where you get the type wrong causes subtle and difficult to find bugs.

Finally, note that there are one or two other small bugs in the paper - make sure you understand exactly what the code is doing and why, inspect the code with suspicion, and you should be okay.