Operating Systems, Assignment 5

This assignment includes two programming problems, one in C and one in CUDA for the ARC system that you made an account for on assignment 4. As always, be sure your code compiles and runs on the EOS Linux machines before you submit (or on the arc system for the second program). Also, remember that your work needs to be done individually and that your programs need to be commented and consistently indented. You **must** document your sources and, if you use sources, you must still write at least half your code yourself from scratch.

1. (40 pts) We're going to implement a multi-threaded Unix client/server program in C using TCP/IP sockets for communication. The server will let users query what score a given word is worth in the game of scrabble. Each user will be able to post a word. The server will remember the most recent word posted by each user and will be able to report all the users' most recent words, sorted by their scrabble score.

   I'm providing you a skeleton of the server to help get you started, `scrabbleServer.c`. You'll complete the implementation of this server, adding support for multi-threading and synchronization, and building some representation for scoring words according to scrabble letter values and for storing the most recent word submitted by each user. You won't need to write a client program; for that, we're going to use a general-purpose program for network communication, `nc`.

   Once started, your server will run perpetually, accepting connections from clients until the user kills it with ctrl-C. The program we're using as a client, `nc`, will take the server's hostname and port number as command-line arguments. The sample execution at the end of this problems shows how to run `nc` like this. After connecting to the server, `nc` will just echo to the screen any text sent by the server and send any text the user enters to the server.

   Scrabble Word Scores

   The game of scrabble requires players to make words by arranging tiles on a game board. Each tile has a single letter, with a score for that letter. The score for a word is the sum of scores for all its letters. In a real game of scrabble, there are other factors, like special squares on the game board that increase the value of words. We're not going to use features like that. For this program, the score for a word will just be the total score for all of its letters.

   A word can consist of up to 24 letters. Either capital or lower-case is fine, but a word can only contain letters (no spaces, punctuation, digits, etc). The following table gives the score for each letter (either capital or lower-case):

Letter	score
A, E, I, O, U, L, N, S, T or R	1
D or G	2
B, C, M or P	3
F, H, V, W or Y	4
K	5
J or X	8
Q or Z	10

   Client / Server Interaction

   When a client first connects to the server, it will prompt the user for a username, using the prompt, "`username>` ". The username can be any string of 1 to 10 letters. After the user enters a name, the server will repeatedly prompt for a command using the prompt "`cmd>` ". In the starter code, the server

just echoes each command back to the client, but you're going to modify it to support the following commands:

- `query` *word*
  This asks the server for the score for the given word. The server should report the score on the next output line.
- `submit` *word*
  The server maintains a list of words submitted by users (one word per user). This list should be able to contain words for any number of users, so it may need to use something like a linked list or a resizable array to store arbitrarily many entries.

  The submit command records the given word as the most recent word submitted by the current user. If the user has previously submitted a word, this replaces it with the given word.
- `report`
  This command lists the most recent word submitted by each user. The list should be sorted by word score, least to greatest. If two words have the same score, they can be reported in any order. If a given user hasn't submitted any word yet, they should be omitted from the report.

  The report should have a line for each user, giving the user's name, right aligned in an 10-character field, followed by a space, then the user's most recently submitted word, right aligned in a 24-character field. Then, after another space the line should give the score for the word, right aligned in a 3-character field. Examples of this output format are included in the sample execution below.
- `quit`
  This is a request for the server to terminate the connection with this client. This behavior has already been implemented for you, but see the "NC Hanging" section below; we seem to get different behavior on different `nc` clients, depending on the OS and installation.

Invalid Client Input

If the client sends an invalid user name at the start (one that's too long or contains something other than capital letters), the server will print a line saying "Invalid username" and terminate the connection with the client right away. See the note below about the behavior of `nc` on some systems. Like the quit command described above, this also causes a problem when the user enters an invalid username.

After successfully entering the username, if the client sends an invalid command, the server will print out a line saying "Invalid command", ignore that input line and then prompt for the next command. So, a bad username terminates the connection right away, but bad commands after that don't terminate the connection.

You can assume each user command is given on a single line of input. You don't have to worry about dealing with multiple commands given on the same line, or a single command spread over multiple input lines. We won't test your program with inputs like that. This should make the user input easier to parse.

NC Client

For this program, we won't need to write a separate client program. The server just needs a client that can open a socket connection, send user input over the socket and write anything it reads from the socket to the terminal. There are standard programs that do this kind of thing. In fact, the predecessor to ssh, 'telnet', did exactly this. It's no longer installed on the university machines, but we can use a the program 'nc' to do some of the same thing.

You can run nc as follows. Here, hostname is the name of the system where you're running your server. The port-number is the unique port number your server is using for listening for connections (see below).

nc *hostname port-number*.

Client/Server Interaction

Client/server communication is all done in text. In the server, we're using fdopen() to create a FILE pointer from the socket file descriptor. This lets us send and receive messages the same way we would write and read files using the C standard I/O library. Of course, we could just read and write directly to the socket file descriptor, but using the C I/O functions should make sending and receiving text a little bit easier.

The partial server implementation just repeatedly prompts the client for commands and terminates the connection when the client enters "quit". You will extend the server to handle the commands described above. There's a sample execution below to help demonstrate how the server is expected to respond to these commands.

Multi-Threading and Synchronization

In the starter code, the server uses just the main thread to accept new client connections and to communicate with the client. It can only interact with one client at a time. You're going to fix this by making it a multi-threaded server. Each time a client connects, you will create a new thread to handle interaction with that client. When it's done talking to the client, that thread can terminate. Use the pthread argument-passing mechanism to give the new thread the socket associated with that client's connection.[1]

With each client having its own thread on the server, there could be race conditions when threads try to access any state stored in the server (e.g., when they report or modify the most recent word submitted by each user). You'll need to add synchronization support to your server to prevent potential race conditions. You can use POSIX semaphores or POSIX monitors for this. That way, if two clients enter a command at the same time, the server will make sure their threads can't access shared server state at the same time.

Detaching Threads

Previously, we've always had the main thread join with the threads it created. Here, we don't need to do that. The main thread can just forget about a thread once it has created it. Each new thread can communicate with its client and then terminate when it's done.

For this to work properly, after creating a thread, the server needs to detach it using `pthread_detach()`. This tells pthreads to free memory for the given thread as soon as it terminates, rather than keeping it around for the benefit of another thread that's expected to join with it.

Buffer Overflow

In its current state, the server is vulnerable to buffer overflow where it reads the username or commands from the client. You'll need to fix this vulnerability in the existing code and make sure you don't have potential buffer overflows in any new code you add.[2] Your server only needs to be able to handle the commands listed above. If the user tries to type in a really long name, a really long command or a really long word, you should detect this, ignore it and print the "Invalid command" message.

At the start of a connection, if the user tries to enter a username that's too long, you should print the "Invalid username" message and terminate the connection.

[1] Remember that it's easy to make a mistake in how you pass an argument to a new thread. Be careful here.

[2] As you know, a buffer overflow is really bad in a server. It could permit an attacker anywhere in the world to gain access to the system the server is running on.

Input Flushing

Because of the way we're creating our file pointer, it looks like buffered input is discarded when the server starts writing output to the socket. This ends up being fairly convenient, but its something to keep in mind as you're writing the code to parse and respond to user commands. For example, you'll need to read all parts of the user's command before you start printing a response to it (otherwise, you'll lose the parts you haven't read yet). Likewise, imagine the the user types in a really word as part of a query command. Once you've detected that the word is too long, you can print out the "Invalid command" message and the rest of their command will be discarded. You wont' have to write code to read and discard the rest of the command.

Port Numbers

Since we may be developing on multi-user systems, we need to make sure we all use different port numbers our servers can listen on. That way, a client written by one student won't accidentally try to connect to a server being written by another student. To help prevent this, I've assigned each student their own port number. Visit the following URL to find out what port number your server is supposed to use.

`people.engr.ncsu.edu/dbsturgi/class/info/246/`

NC Hanging

I haven't really seen `nc` hang, but I've seen what might look like nc hanging. If the server terminates the socket connection to the nc client, I've seen nc continue to run. This is what happens if you type in a bad username at the start of a client connection, or if you type in the quit command. The server will close the client's socket connection and let the client's thread terminate, but the nc client won't exit. When this happens, you can press ctrl-C to terminate the nc client. This works, but it's inconvenient to have to take this extra step to end the `nc` client. There are other client programs that automatically exit when the socket connection is closed (putty for example), but it's not clear that there are better alternatives installed on the NC State machines.

The `nc` program seems to work fine on my MacBook, but, on the university machines it exhibits this behavior. if you're running nc on a university machine, be prepared to type ctrl-C to get nc to exit, even after the server closed the socket connection.

Sad Truths about Sockets on EOS Linux Hosts

Since we're using TCP/IP for communication, we can finally run our client and server programs on different hosts. You should be able to run your server on one host and then start nc on any other Internet-connected system in the world, giving it the server's hostname and your port number on the command line. However, if you try this on a typical university Linux machine, you'll probably be disappointed. These systems have a software firewall that blocks most IP traffic. As a result, if you want to develop on a university system, you will still need to run the client and server on the same host. If you have your own Linux machine, you should be able to run your server on it and connect from anywhere (depending on how your network is set up).

If you'd like to use a machine in the Virtual Computing Lab (VCL), you should be able to run commands as the administrator, so you can disable the software firewall. I had success with a reservation for a CentOS 7 Base (64 bit VM) system. After logging in, uploading and building my server, I ran the following command to permit TCP connections via my port. Then, if I ran a server on the virtual machine, I was able to connect to it, even from off campus.

sudo /sbin/iptables -I INPUT 1 -p tcp --dport *my-port-number* -j ACCEPT

Also, if your server crashes, you may temporarily be unable to bind to your assigned port number, with an error message "Can't bind socket." Just wait a minute or two and you should be able to run the server again.

Implementation

You will complete your server by extending the file, `scrabbleServer.c` from the starter. You'll need to compile as follows:

```
gcc -Wall -g -std=gnu99 scrabbleServer.c -o scrabbleServer -lpthread
```

Sample Execution

You should be able to run your server with any number of concurrent clients, each handled by its own thread in the server. Below, we're looking at commands run in three terminal sessions, with the server running in the left-hand column and clients running in the other two columns (be sure to use our own port number, instead of 28123). I've interleaved the input/output to illustrate the order of interaction with each client, and I've added some comments to explain what's going on. If you try this out on your own server, don't enter the comments, since the server doesn't know what to do with them.

Here, I'm running all three programs on the same host, but if you use a machine where you can control the firewall, you should be able to run this example on three different hosts.

```
$ ./scrabbleServer

                  # Run one client and query a word.
                  $ nc localhost 28123
                  username> alice
                  cmd> query counterrevolutionary
                  28

                  # Submit a word
                  cmd> submit skillfully

                                          # Run another client, and
                                          # submit a word.
                                          $ nc localhost 28123
                                          username> bob
                                          cmd> submit monopolize

                                          # The report contains words from both clients
                                          cmd> report
                                               alice              skillfully  20
                                                 bob              monopolize  23

                  # Disconnect as alice, and connect
                  # as a different user.
                  cmd> quit
                  ctrl-C (if you need this)
```

```
$ nc localhost 28123
username> cheryl

# Submit a word for this user and view
# the (sorted) report
cmd> submit The
cmd> report
    cheryl                      The   6
     alice              skillfully  20
       bob              monopolize  23

# Change my word and get a new
# report
cmd> submit Theoretically
cmd> report
     alice              skillfully  20
    cheryl           Theoretically  21
       bob              monopolize  23

                          # Try some invalid commands
                          cmd> query InvalidWordBecauseItIsTooLong
                          Invalid command
                          cmd> fskdsljdfk fjsdkslfsj
                          Invalid command
                          cmd> submit invalid-chars-123
                          Invalid command

                          # Quit both clients.
                          cmd> quit
                          ctrl-C (if you need this)

cmd> quit
ctrl-C (if you need this)

# Kill the server.
ctrl-C
```

Submitting your Work

When you're done, submit your `scrabbleServer.c` using the Moodle link for Homework_5.

2. (40 points) For this problem, you're going to implement the zerosum program from assignments 1, 2 and 3 on a GPU, with lots and lots of threads. We're going to use the CUDA framework on the arc cluster at NC State. You know about arc; you should have activated your arc account as part of assignment 4. Arc has a collection of about 100 hosts running Linux, each with a general-purpose CPU and a GPU serving as a highly parallel co-processor.

Arc Filesystem

Arc doesn't share user accounts or filesystems with other university machines. To connect to it, you'll need to ssh in from another university system (e.g., remote.eos.ncsu.edu), just like you did when you

activated your arc account for assignment 4. To successfully connect, you'll need to use the same machine where you made the your key pair on assignment 4, or some machine that has access to AFS if you made your key pair on a machine with AFS.

Due Date Distribution

The arc system has a lot of nodes, but it doesn't have enough for all of us to have one at the same time. To make sure you get a chance to complete this problem, I'm implementing a soft due date for this assignment. You can earn up to 8 points of **extra credit** by completing this problem early. Or, if you submit your solution late, the late penalty will be applied gradually rather than all at once. I'm hoping this will help to make sure we aren't all trying to use the arc system on the same evening. Arc has a lot of compute nodes, but if we're all trying to use it at the same time, some users are going to be disappointed.

The following table shows the extra credit or late penalty based on how early or late you turn in your solution.

Bonus / Penalty	Submission Time
+8	At least 96 hours early
+6	At least 72 hours early
+4	At least 48 hours early
+2	At least 24 hours early
-2	At most 24 hours late
-4	At most 48 hours late

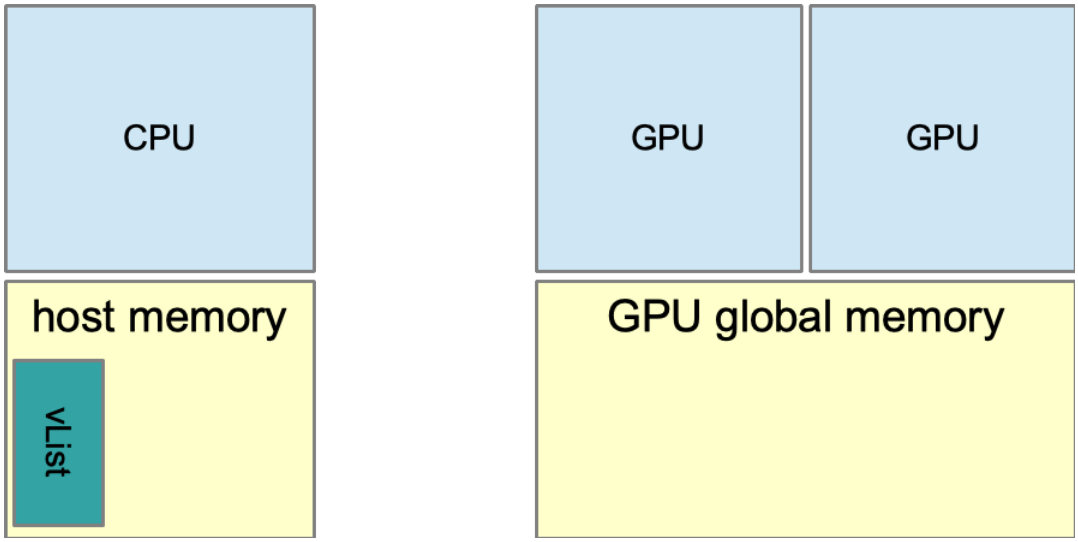
Work Partitioning

For this assignment, we're going to divide work statically among the threads. You can divide up the work however you want, and, unlike on homework 3, you will have access to the whole sequence of values before any of your worker threads start running. Each thread has a unique index, i. It computes this index based on the grid and block dimensions right after the thread starts up. You could have thread number i look for all zero-sum sequences starting at position i in the sequence, or maybe it could check all sequences ending at position i, or something else if you have a better idea. If the report option is given on the command line, your threads should report ranges as they find them, like we've done on previous assignments. CUDA has support for this, even though the GPU threads are running on a different device.

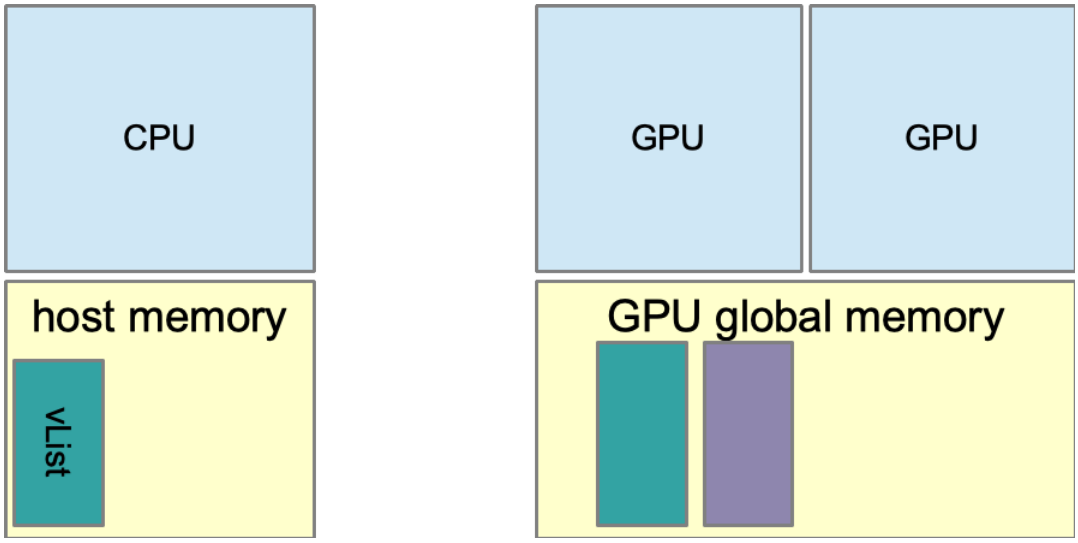
When a worker is done, it will copy the count of zero-sum ranges it finds to element i of an array allocated in CUDA global memory. After all the threads are finished, code from main will copy all the counts to host memory and add up the results to get an overall total.

For previous assignments, we made each process or thread responsible for several elements in the sequence. We didn't want to create too many workers. This won't be a problem on a GPU. It can handle lots of threads at once, and the CUDA software will take care of sharing the processing elements among threads if we ask for more threads than the GPU can execute at once.

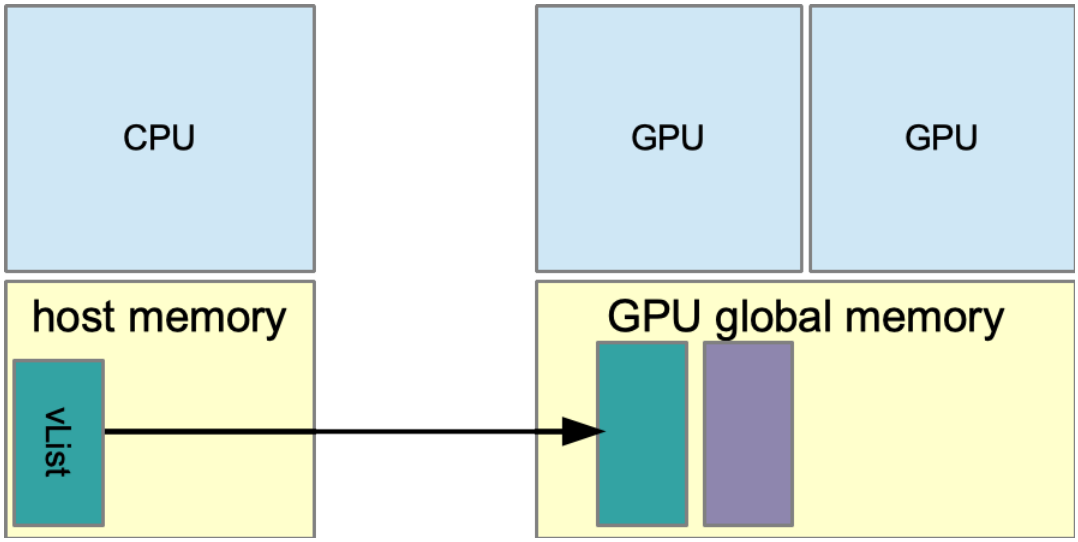
I'm providing you with a skeleton, `zerosum.cu`, to get you started. This program reads the sequence into an array on the host, `vList`.



You'll need to add code to allocate two arrays on the device, one array for holding a copy of the host's sequence and another array of integers for holding the counts computed by each thread.

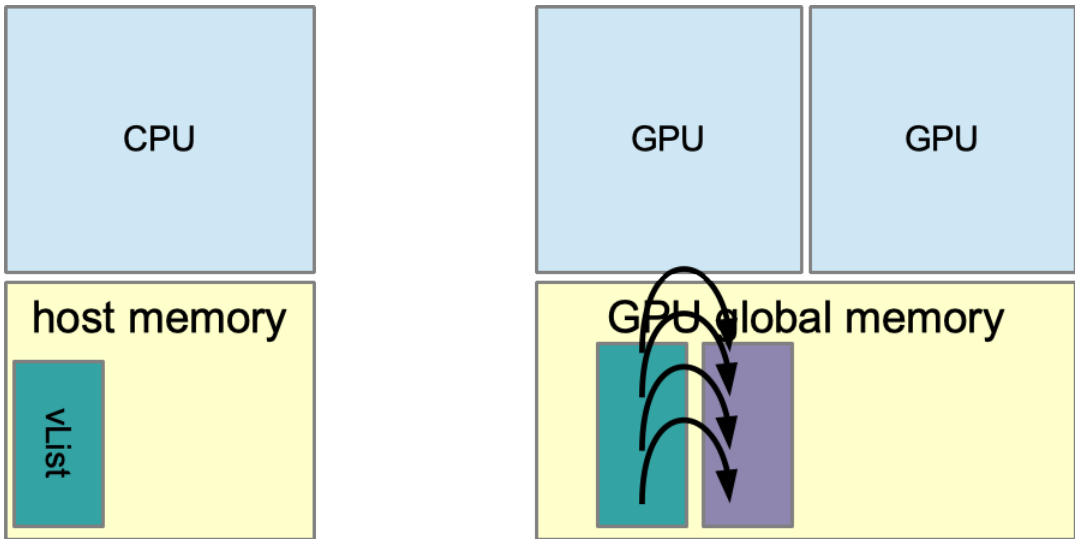


Then you'll need code to copy the sequence from the host over to the device, so threads on the GPU can access it.

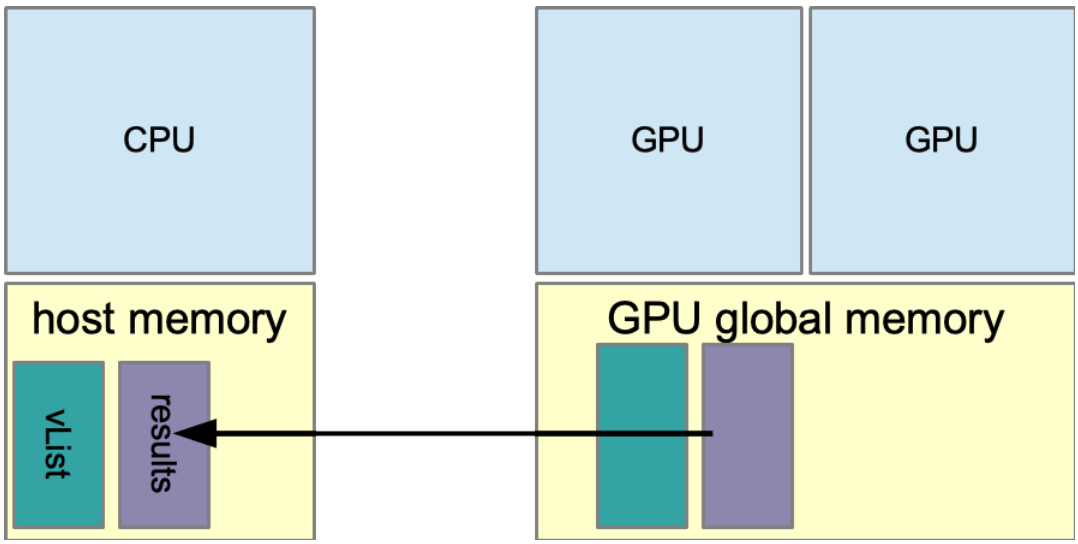


Now that the device is ready, you'll need to complete the implementation of the checkSum() kernel. Each thread will be responsible for checking for some ranges in the sequence. When it's done, it will store its total count in the i^{th} index of the results array in device memory.

When you run this kernel from the host, it will start a thread for each element and wait for them to finish computing their counts.



Then, the host will need to allocate host memory (malloc) and copy the results into it.



Then, the main thread on the host can quickly add up the counts from all the threads and print out a total count in the format:

```
Total: 225
```

Thread Implementation

You get to write most of the checkSum() function, the code that actually runs on the GPU. I've written some starter code for this function. Right now, each thread just computes a unique index and makes sure it has a value to work on[3] and then returns. You'll need to add parameters to the kernel function for passing in pointers to the input and output arrays, looking for zero-sum ranges and counting how many each thread finds.

Working on Arc

To access arc, you'll have to log in from another university machine, where you've stored the key pair that you used to activate your arc account (probably remote.csc.ncsu.edu). Use your ssh client (e.g., putty or mobaXterm or ssh) to connect to remote.csc.ncsu.edu, then you can get to arc from there.

Uploading your Files

The nodes on the arc cluster share a filesystem, but they don't use the university AFS. To get files onto the machine, you'll need to upload them from another university machine. You can just upload your files to one of the university Linux machines and then transfer them from there using the `sftp` program. From one of the remote Linux machines (not from ARC), commands like the following should let you upload files. Starting from a directory containing your zerosum.cu and the input files:

```
sftp arc.csc.ncsu.edu

sftp> put zerosum.cu
Uploading zerosum.cu ...
```

[3] If the block size doesn't evenly divide the number elements in the sequence, there may be some threads in the last block that don't actually have anything to do.

```
sftp> put input-1.txt
Uploading input-1.txt
sftp> put input-2.txt
Uploading input-2.txt
<you get the idea>
sftp> bye
```

Logging in on Arc

After logging in to one of the machines at remote.eos.ncsu.edu, you should be able to enter the following to log in to the head node on arc:

```
ssh arc.csc.ncsu.edu
```

Editing, Compiling and Running

When you log in to arc, you connect to a head node that doesn't even have a GPU. From the head node, you'll need to connect to one of the compute nodes. Arc uses batch processing software to assign jobs to the compute nodes. The following commands let you connect from the head node to one of the compute nodes. Each time you want to work on arc, you should just need to run one of these commands. It will give you an interactive prompt on one of the compute nodes as soon as one is available. The comments below tell you what kind of GPU you'll get on each node. Depending on how busy the system is, you may have to wait a little while to get a prompt. Also be sure you run this from an a shell prompt (not from inside sftp command described above):

```
# there are 13 nodes with an nVidia RTX 2060
srun -p rtx2060 --pty /bin/bash

# there are 17 nodes with an nVidia rtx2070.
# Once, when I tried to use one of the rtx2070 systems, it failed
# on my first call to a cuda function.  Later, when I logged back
# in to another rtx2070 system, it worked fine.  This makes me think
# there may have been one of these nodes that had a bad GPU or maybe
# it wasn't configured correctly.  If you have this problem, you may
# want to try another system.
srun -p rtx2070 --pty /bin/bash

# there are 2 nodes with an nVidia RTX 2080
srun -p rtx2080 --pty /bin/bash

# there are 21 nodes with an nVidia RTX 2060 Super
srun -p rtx2060super --pty /bin/bash

# there are 2 nodes with an nVidia RTX 2080 Super
# I also got an error from my first CUDA call when I tried one of
# these systems.
srun -p rtx2080super --pty /bin/bash

# there are still some systems with the older, nVideo gtx480 graphics
# card.  If you can't connect to one of the newer systems above, you
# could try one of these older systems instead.
```

```
srun -p gtx480 --pty /bin/bash
```

Once you have a prompt on a compute node, you should still be able to see the files you uploaded. Arc used to require you to enter the following command before you could do any work with CUDA, but it seems like this is no longer required. If you try to run the nvcc compiler and you get a complaint that your shell can't find a program with that name, you could try running this command to see if that fixes it.

```
module load cuda
```

For editing your program, you can make edits in AFS and just use sftp to copy the modified file to arc every time you make a change. If you're comfortable editing directly on Linux machine, it may be easier for you to edit your source file directly on the arc machine. Editors vim and emacs are there, but it doesn't look like the really simple editors like pico or nano are installed (vim and emacs are better editors anyway, but it takes a little space in your brain to be ready to use them).

Once you're ready to build your program, you can compile as follows:

```
nvcc -I/usr/local/cuda/include -L/usr/local/cuda/lib64  zerosum.cu -o zerosum
```

When you're ready to run, be sure you're on a compute node (i.e., you've run one of the svrun commands above). You can run your program just like you would an ordinary program. This version of the program will still have the optional report flag, but it won't need to be told how many threads to use. It just makes a thread for every value in the input sequence. As usual, if report is enabled, then your threads will report ranges as they find them, so they may get printed in any order.

```
$ ./zerosum report < input-2.txt
1 .. 2
12 .. 14
5 .. 9
9 .. 13
3 .. 7
1 .. 7
6 .. 15
0 .. 12
4 .. 18
7 .. 19
2 .. 17
Total: 11
```

The time command is available when you want to measure the execution time for your program (especially on the larger inputs).

```
$ time ./zerosum < input-5.txt
Total: 2500985

real ?m?.???s
user ?m?.???s
sys ?m?.???s
```

Recording Execution Time

At the top of the zerosum.cu file, I've put comments for recording running times for this program on the input-5.txt file. Once your program is working, time its execution on this input file and enter the (real) execution time you got in this comment. Also, report what type of compute node you were using (i.e., what type of GPU it has). Don't use the report flag when you're measuring execution time; that would just slow it down. You'll probably notice that start-up time seems to be significant, so pushing work out to the GPU pays off most with the largest input files. The different compute nodes on arc may have different capabilities (at least, that's been the case in the past), so don't worry if you get different execution times from some of your classmates. Either way, I hope you'll find these runtimes to be faster than what you got on a general-purpose CPU.

Logging Out

When you're ready to log out, you'll have several levels to log out from. From a compute node, you can type `exit` to fall back to the head node of the arc cluster. Typing `exit` from there will drop you back to whatever machine you connected from (probably remote.eos.ncsu.edu). One more exit should get you out.

Submitting your Work

When you're done, submit a copy of your working program, `zerosum.cu`, with the runtime filled in at the top. Use the Homework 5 link in Moodle.