Programming Assignment 4

Due Monday, April 14 at 11am
Note the change in the due date

For programming assignment 4 you should write a C program that uses CUDA to implement a parallel solution to the Longest Common Subsequence Problem.

Input to the program will consist of a collection of positive ints entered on stdin:

- m : the length of the first sequence,
- $A = (a_0, a_1, \ldots, a_{m-1})$: the elements of the first sequence,
- n : the length of the second sequence, and
- $B = (b_0, b_1, \ldots, b_{n-1})$: the elements of the second sequence.

The program should find a sequence $C = (c_0, c_1, \ldots, c_{q-1})$, such that

- C is a subsequence of both A and B, and
- if $D = (d_0, d_1, \ldots, d_{r-1})$ is also a subsequence of both A and B, then $r \leq q$.

In other words C is a longest common subsequence of A and B.

The command line for the program should specify the number of thread blocks and the number of threads per block. For example, with the command line:

```
$ ./lcs 2 64
```

the program should run with 2 thread blocks and 64 threads in each block.

An optional single character command line argument, `'n'`, will suppress all output except the longest common subsequence and the elapsed time.

If the `'n'` option is not present, your program should print the `L` array (see below), the longest common subsequence, and the elapsed time for the computation. If the `'n'` option is present, your program should just print the longest common subsequence and the elapsed time.

Terminology

A *sequence* is an *ordered* collection of objects. So a sequence is different from a set. For example, the set $\{1, 2, 5\}$ is the same as the set $\{5, 1, 2\}$, but the sequence $(1, 2, 5)$ is different from the sequence $(5, 1, 2)$, since its elements are in a different order.

A *subsequence* of a sequence can be obtained by removing zero or more elements of the sequence. For example, if S is the sequence $(1, 3, 1, 5, 2, 7)$, then $T = (1, 1, 5)$ and $U = (1, 5, 2, 7)$ are subsequences, but $V = (1, 2, 1, 5)$, $W = (3, 5, 4, 7)$, and $X = (1, 1, 1, 5)$ are not.

Solving the Longest Common Subsequence Problem

For assignment 4, you'll need to solve the two-sequence longest common subsequence problem. A nice discussion of this is contained in Sections 2.1–2.5 of the Wikipedia Article "Longest Common Subsequence Problem" (`http://en.wikipedia.org/wiki/Longest_common_subsequence_problem`).

The basic idea is that we create a data structure L that can be defined in terms of subsequences of A and B. So suppose that $0 \leq i < m$ and $0 \leq j < n$. Then A_i is the subsequence $(a_0, a_1, \ldots, a_i)$ and B_j is the subsequence $(b_0, b_1, \ldots, b_j)$. If $i < 0$, then $A_i = \emptyset$, the empty sequence, and if $j < 0$, $B_j = \emptyset$.

Now L is defined by the following formula:

$$L(A_i, B_j) = \begin{cases} \emptyset, & i < 0 \text{ or } j < 0 \\ L(A_{i-1}, B_{j-1}) + a_i, & a_i = b_j \\ \text{Longer}(L(A_i, B_{j-1}), L(A_{i-1}, B_j)), & a_i \neq b_j \end{cases} .$$

In the second formula the plus sign (+) denotes concatenation: to get $L(A_i, B_j)$ we concatenate a_i to $L(A_{i-1}, B_{j-1})$. In the last formula "Longer" denotes that we choose the longer of the two subsequences $L(A_i, B_{j-1})$ and $L(A_{i-1}, B_j))$.

The basic ideas are as follows:

- If $i < 0$ or $j < 0$, then at least one of the sequences A_i, B_j is empty, and the only common subsequence is the empty sequence.
- If $a_i = b_j$, then we can extend the longest common subsequence of A_{i-1} and B_{j-1} by concatenating a_i (which, of course, is the same as b_j).
- If $a_i \neq b_j$, then the longest common subsequence of A_i and B_j is going to be either the longest common subsequence of A_i and B_{j-1}, or it will be the longest common subsequence of A_{i-1} and B_j : whichever of these two sequences is longer.

So a *serial* solution to the longest common subsequence problem can be obtained by using a doubly nested `for` loop:

```
/* Careful of i-1, j-1 < 0! */
for (i = 0; i < m; i++)
   for (j = 0; j < n; j++)
      if (A[i] == B[j])
         /* L[i,j] = L(A_i, B_j) */
         L[i,j] = L[i-1,j-1] + A[i];  // Concatenate A[i] to L[i-1,j-1]
      else {
         len1 = Length(L[i,j-1]);
         len2 = Length(L[i-1,j]);
         if (len1 > len2)
```

```
            L[i,j] = L[i,j-1];
        else
            L[i,j] = L[i-1,j];
    }
```

In order to obtain *all* of the longest common subsequences, you need to allow multiple sequences in `L[i,j]`. That is, when `len1 = len2`, you need to add both `L[i,j-1]` and `L[i-1,j]` to `L`, but in this assignment you only need to find *one* longest common subsequence. In order to insure that everyone gets the same answer, you should choose `L[i-1,j]` when `len1 = len2`.

Parallelization

The serial algorithm has very little parallelism. Since the computation of `L[i,j]` depends on the computation of `L[i,j-1]`, we can't compute `L[i,j]` until we've computed `L[i,j-1]`. In general, in order to compute `L[i,j]` we need to know the three values, `L[i-1,j-1]`, `L[i-1,j]`, and `L[i,j-1]`. So a parallel computation can simultaneously compute all the values lying along "diagonals" running from an element in the first row to an element in the first column. For example, consider the following diagram:

`L[0,0]`	`L[0,1]`	`L[0,2]`	`L[0,3]`	`L[0,4]`	`L[0,5]`
`L[1,0]`	`L[1,1]`	`L[1,2]`	`L[1,3]`	`L[1,4]`	`L[1,5]`
`L[2,0]`	`L[2,1]`	`L[2,2]`	`L[2,3]`	`L[2,4]`	`L[2,5]`
`L[3,0]`	`L[3,1]`	`L[3,2]`	`L[3,3]`	`L[3,4]`	`L[3,5]`
`L[4,0]`	`L[4,1]`	`L[4,2]`	`L[4,3]`	`L[4,4]`	`L[4,5]`

1. Once we've computed `L[0,0]`, we can simultaneously compute `L[1,0]` and `L[0,1]`.
2. Once we've computed `L[0,1]` and `L[1,0]`, we can simultaneously compute `L[2,0]`, `L[1,1]`, and `L[0,2]`.
3. Once we've computed `L[2,0]`, `L[1,1]`, and `L[0,2]`, we can simultaneously compute `L[3,0]`, `L[2,1]`, `L[1,2]`, and `L[0,3]`.
4. Etc.

So if we divide the elements of a diagonal among the threads, each thread can compute some of the entries in the diagonal (or be idle).

Note that we can't proceed to the next diagonal until we've completed computation of the current diagonal. So we'll need to synchronize the threads after each computation of a diagonal.

All of this suggests that we create a CUDA kernel that computes a single element of `L`, and we invoke the kernel so that it starts at least one thread for each element on the current diagonal. Pseudo-code might look something like this:

```
Allocate d_A, d_B on device;
Copy A into d_A, B into d_B;
Allocate L on the host;
Initialize L;
Allocate d_L on the device;
Copy L into d_L;
diag_count = m + n - 1;
max_diag_len = min(m,n);
for (diag = 0; diag < diag_count; diag++) {
   /* Call the kernel */
   Find_L_entry <<blocks, threads_per_block>> (d_L, d_A, m, d_B, n,
      max_diag_len, diag);
   cudaThreadSynchronize();
}
Copy d_L into L;
Free d_A, d_B, L, d_L;
```

You can assume that the product

`blocks*threads_per_block >= max_diag_len.`

So the number of threads running the kernel can always be at least as large as the number of elements in the diagonal. Also before starting computation of the element of `d_L`, we can determine which row and column the thread is responsible for, and whether they correspond to a "valid" element of `d_L`:

```
int j = threadIdx.x + blockIdx.x*blockDim.x;
int i = diag - j;  /* i + j = diag */
if (i >= 0 && i < n)
   Compute d_L[i,j];
```

Data Structures

It may be tempting to define a struct that represents subsequences, and then define `L` and `d_L` to be arrays of this struct. You *may* do this if you want. However, it is *very* difficult to copy complex data structures from host to device and vice-versa. So I advise you to instead make `L` and `d_L` large arrays of `int` and simply write functions or macros to access individual elements. For example, suppose that $m = 4$ and $n = 6$. Then the maximum possible number of elements in any subsequence will be 4. So we can allocate $4 \times 6 \times (4 + 1)$ ints for `L`: the "+1" is for storing the actual length of the subsequence. Then, the length of the subsequence in row 2 and column 3 might be stored in

`L[2*n*(4+1) + 3*(4+1)]`

and the element in slot 2 in the subsequence in row 2 and column 3 would be stored in

```
L[2*n*(4+1) + 3*(4+1) + 1 + 2].
```

Here, the "+1" will skip over the entry that stores the length. So with these values for m and n, we might write the following access functions:

```
/* entry_sz = max subsequence length + 1 */
int Subseq_length(int Arr[], int n, int entry_sz,
      int i, int j) {
   int sub = i*n*entry_sz;
   sub += j*entry_sz;
   return sub;
}

int Subseq_element(int Arr[], int n, int entry_sz
      int i, int j, int k) {
   int sub = i*n*entry_sz;
   sub += j*entry_sz;
   sub += (1 + k);
   return sub;
}
```

Of course, once the program has been debugged these should be replaced by macros to speed execution.

A further complication to the data structures arises from thread divergence. Recall that in CUDA if two threads in the same warp execute different branches of an `if-else`, then their execution is serialized. For example, suppose we execute the code

```
if (x > 0)
   code1;
else
   code2;
```

Now suppose that on thread zero `x = 1` while on thread one `x = 0`, and the two threads are in the same warp. Then CUDA will not execute `code1` and `code2` simultaneously. Rather, it will execute one and then the other. So the advantage of having two threads is lost.

The problem here is that the computations in the kernel will depend on whether `i-1` and/or `j-1` are negative. If they are, then of course there is no valid entry in `L` or `d_L`, and we would run the risk of thread divergence every time we checked whether these are valid. There are several ways to deal with this. One of the simplest is to just add an extra row and column to the data structures so that a row or column subscript of -1 will be valid. For example, we might declare a "big" data structure with $m + 1$ rows and $n + 1$ columns:

```
int *big_L = (int*) malloc((m+1)*(n+1)*(max_subseq_len+1)*sizeof(int));
```

Then `L` can be made to "point to" `big_L[1,1]`:

```
int *L = &big_L[1*(n+1)*entry_sz + 1*entry_sz];
```

Now subscripts of -1 are valid for `L` and the only change that needs to be made to the preceding discussion is that instead of using `n` for element access, we need to use `n+1`.

Timing

It's easiest to take the timings on the host. If you do this, you should start the timer after copying the data structures from the host to the device. After the final computation on the device (i.e., after computing the last diagonal), you should stop the timer. So your timing shouldn't include the time it takes to copy data structures either from the host to the device or from the device to the host.

Cuda does provide a class called `cudaEvent_t` and objects of this class can be used for timing. However, I've found that the results they give can be unreliable on some systems. So I recommend that you use `timer.h` and the macro `GET_TIME`.

Using CUDA

If you have an Nvidia graphics card installed on your computer, you should be able to install CUDA. See

```
https://developer.nvidia.com/cuda-downloads
```

for further information. If you do this, be sure that your program compiles and runs on the CS department systems, since these are the systems I'll use to test your program.

Alternatively, you can use CUDA on one of the CS lab machines. You'll need to log on to stargate:

```
$ ssh stargate.cs.usfca.edu
```

When you get onto stargate, you can find the machines that are booted into Linux by running

```
$ rusers -a
```

Then you can log on to any of these machines.

To use CUDA on one of the lab machines, you'll need `/usr/local/cuda/bin` in your `PATH` environment variable. To check if it's already there, type

```
$ echo $PATH
```

If `/usr/local/cuda/bin` is not in the output of this command, you can add it to your `PATH` by typing

```
$ export PATH=/usr/local/cuda/bin:$PATH
```

A better solution is to add this command to the file `.bash_profile` in your home directory. Once this is done, the CUDA binaries will be automatically added to your path every time you log on.

It may also be necessary to add `/usr/local/cuda/lib64` to your `LD_LIBRARY_PATH`. Type

```
$ echo $LD_LIBRARY_PATH
```

If `/usr/local/cuda/lib64` is not listed, you can add it by typing

```
$ export LD_LIBRARY_PATH=/usr/local/cuda/lib64:$LD_LIBRARY_PATH
```

As before, a better solution is to add this command to your `.bash_profile`.

When you compile your program on the lab machines, it's a good idea to use the `-arch` and `-G` command line options:

```
$ nvcc -g -G -arch=sm_21 -o lcs lcs.cu
```

(Note that `nvcc` doesn't recognize the `-Wall` option.) The `-g` option generates debug information for the host, and the `-G` option generates debug information for the device. The `-arch=sm_21` option allows your source to use the more advanced features available with our GPU's. (For example, you can execute `printf` in device code.)

Submission

You should submit your source file(s) to your SVN `cs625/prog4` directory by 11 am on Monday, April 14, and you should get your hardcopy to me by 2pm.

Grading

1. Correctness will be 75% of your grade. Does your code operate as required?
2. Documentation will be 10% of your grade. Does your header documentation include the author's name, the purpose of the program, and a description of how to use the program? Are the identifiers meaningful? Are any obscure constructs clearly explained? Does the function header documentation explain the purpose of the function, its arguments and its return value?
3. Source format will be 5% of your grade. Is the indentation consistent? Have blank lines been used so that the program is easy to read? Is the use of capitalization consistent? Are there any lines of source that are longer than 80 characters (i.e., wrap around the screen)?
4. Quality of solution will be 10% of your grade. Is the code clear and efficient?

Academic Honesty

Remember that you cannot look at anyone else's source code. This includes source code that you find on the internet. If you have any doubt about whether what you'd like to do is legal, you should ask me first.