Introduction to Parallel Computer Architecture
Final Exam

Instructor: Prof. Naga Kandasamy
ECE Department
Drexel University

December 4, 2015

The exam is due by noon on December 12, 2015. It comprises of two programming problems. You may work on this exam in a team of up to two people.

1. **(25 points)** Given a function $f(x)$ and end points a and b, where $a < b$, we wish to estimate the area under this curve; that is, we wish to determine $\int_a^b f(x)\,\mathrm{d}x$.

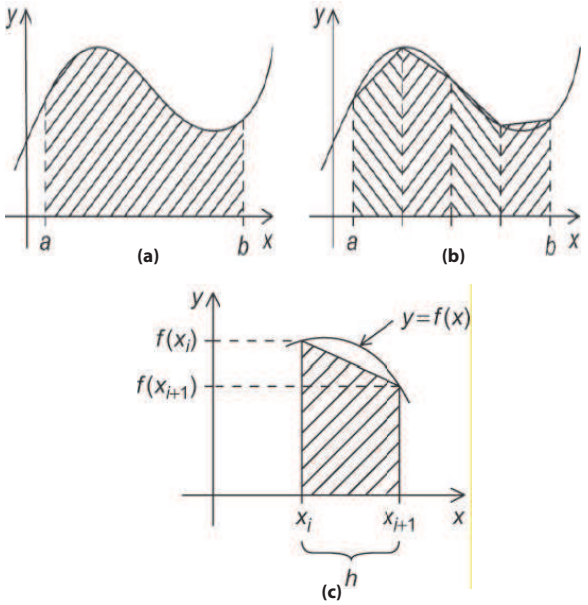

Figure 1: Illustration of the trapezoidal rule: (a) area to be estimated; (b) approximate area using trapezoids; and (c) area under one trapezoid.

The area between the graph of $f(x)$, the vertical lines $x = a$ and $x = b$, and the x-axis can be estimated as shown in Fig. 1 (b) by dividing the interval $[a, b]$ into n subintervals and approximating the area over each subinterval by the area of a trapezoid. Fig. 1(c) shows one such trapezoid where the base of the trapezoid is the subinterval, its vertical sides are the vertical lines through the

endpoints of the subinterval, and the fourth side is the secant line joining the points where the vertical lines cross the graph. If the endpoints of the subinterval are x_i and x_{i+1}, then the length of the subinterval is $h = x_{i+1} - x_i$, and if the lengths of the two vertical segments are $f(x_i)$ and $f(x_{i+1})$, then the area of a single trapezoid is $\frac{h}{2}[f(x_i) + f(x_{i+1})]$. If each subinterval has the same length then $h = (b - a)/n$. Also, if we call the leftmost endpoint x_0 and the rightmost endpoint x_n, we have

$$x_0 = a, x_1 = a + h, x_2 = a + 2h, \ldots, x_{n-1} = a + (n - 1)h, x_n = b,$$

and our approximation of the total area under the curve will be

$$\int_a^b f(x)\,\mathrm{d}x = h[f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2].$$

Thus, the pseudo-code for a serial algorithm might look something like the following.

```
1: procedure TRAP(a, b, n)
2: h := (b − a)/n;
3: sum := (f(a) + f(b))/2.0;
4: for i := 1 to n − 1 step 1 do
5:     x_i := a + i × h;
6:     sum := sum + f(x_i);
7: end for
8: sum := h × sum;
```

The program provided to you accepts no arguments. The values for a, b, and n are defined within the program and so is the function $f(x)$ (within the file `trap_gold.cpp`). A CPU implementation generates a reference solution which will be compared with your GPU program's output. Edit the `compute_on_device()` function within the file `trap.cu` to complete the functionality of the trapezoidal rule on the GPU. To achieve this functionality, you may add multiple kernels to the `trap_kernel.cu` file.

Upload all of the files needed to run your code as a single zip file on BBLearn. Also, provide a brief report describing: (1) the design of your kernel(s) including the optimization techniques used (provide code or pseudocode to clarify the discussion); (2) the speedup achieved over the serial version; and (3) sensitivity of the kernel to thread-block size in terms of the execution time. Ignore the overhead due to CPU/GPU communication when reporting speedup.

2. **(25 points)** Consider the problem of solving a system of linear equations of the form In matrix

$$\begin{array}{lllll} a_{0,0}x_0 & + a_{0,1}x_1 & + \cdots & + a_{0,n-1}x_{n-1} & = b_0, \\ a_{1,0}x_0 & + a_{1,1}x_1 & + \cdots & + a_{1,n-1}x_{n-1} & = b_1, \\ \cdot & \cdot & & \cdot & \cdot \\ \cdot & \cdot & & \cdot & \cdot \\ a_{n-1,0}x_0 & + a_{n-1,1}x_1 & + \cdots & + a_{n-1,n-1}x_{n-1} & = b_{n-1}. \end{array}$$

notation, the above system is written as $Ax = b$ where A is a dense $n \times n$ matrix of coefficients such that $A[i, j] = a_{i,j}$, b is an $n \times 1$ vector $[b_0, b_1, \ldots, b_{n-1}]^T$, and x is the desired solution vector $[x_0, x_1, \ldots, x_{n-1}]^T$. From here on, we will denote the matrix elements $a_{i,j}$ and x_i by $A[i, j]$ and $x[i]$, respectively. A system of equations $Ax = b$ is usually solved in two stages. First, through a set of algebraic manipulations, the original system of equations is reduced to an upper triangular system of the form We write the above system as $Ux = y$, where U is an upper-triangular matrix,

$$\begin{array}{lllll} x_0 & + u_{0,1}x_1 & + u_{0,2}x_2 & + \cdots & + u_{0,n-1}x_{n-1} & = y_0, \\ & x_1 & + u_{1,2}x_2 & + \cdots & + u_{1,n-1}x_{n-1} & = y_1, \\ & & & & \cdot & \cdot \\ & & & & \cdot & \cdot \\ & & & & x_{n-1} & = y_{n-1}. \end{array}$$

that is, one where the subdiagonal entries are zero and all principal diagonal entries are equal to one. More formally, $U[i, j] = 0$ if $i > j$, otherwise $U[i, j] = u_{i,j}$, and furthermore, $U[i, i] = 1$ for $0 \leq i < n$. In the second stage of solving a system of linear equations, the upper-triangular system is solved for the variables in reverse order, from $x[n - 1]$ to $x[0]$ using a procedure called back-substitution.

A serial implementation of a simple Gaussian elimination algorithm is shown in the next page. The algorithm converts the system of linear equations $Ax = b$ into a unit upper-triangular system $Ux = y$. We assume that the matrix u shares storage with A and overwrites the upper-triangular portion of A. So, the element $A[k, j]$ computed in line 5 of the code is actually $U[k, j]$. Similarly, the element $A[k, k]$ that is equated to 1 in line 8 is $U[k, k]$. Also, our program assumes that $A[k, k] \neq 0$ when it is used as a divisor in lines 5 and 7. So, our implementation is numerically unstable, though it should not be a concern for this assignment. For k ranging from 0 to $n - 1$, the Gaussian elimination procedure systematically eliminates the variable $x[k]$ from equations $k+1$ to $n - 1$ so that the matrix of coefficients becomes upper-triangular. In the $k^{\texttt{th}}$ iteration of the outer loop (line 3), an appropriate multiple of the $k^{\texttt{th}}$ equation is subtracted from each of the equations $k + 1$ to $n - 1$.

Develop a parallel formulation of `GAUSS_ELIMINATE` for the GPU. Complete the functionality of Gaussian elimination on the GPU by editing the `gauss_eliminate_on_device()` function in `gauss_eliminate.cu`. You may develop additional kernels as needed. The program given to you accepts no arguments. The upper-diagonal matrix generated by the GPU is compared against the CPU result and if the solutions match within a certain tolerance, the application will print out "Test PASSED" to the screen before exiting. Upload all of the files needed to run your code as a single zip file on BBLearn. This question will be graded on the following parameters:

```
1: procedure GAUSS_ELIMINATE(A, b, y)
2: int i, j, k;
3: for k := 0 to n − 1 do
4:    for j := k + 1 to n − 1 do
5:       A[k, j] := A[k, j]/A[k, k];      /* Division step. */
6:    end for
7:    y[k] := b[k]/A[k, k];
8:    A[k, k] := 1;
9:    for i := k + 1 to n − 1 do
10:      for j := k + 1 to n − 1 do
11:         A[i, j] := A[i, j] - A[i, k] × A[k, j];      /* Elimination step. */
12:      end for
13:      b[i] := b[i] − A[i, k] × y[k];
14:      A[i, k] := 0;
15:   end for
16: end for
```

- **(10 points)** At the very least, use GPU global memory to get your code working. Also note that you made need to use double precision on the GPU to pass the acceptance test with respect to the reference result.
- **(15 points)** Optimize the performance of your GPU code via efficient use of the GPU memory hierarchy, appropriately sizing the thread granularity, etc.

Provide a short report describing how you designed your kernel (use code or pseudocode if that helps the discussion) and the amount of speedup obtained over the serial version for the following matrix sizes: 512×512, 1024×1024, and 2048×2048. Ignore the overhead due to CPU/GPU communication when reporting speedup.