ECEC-622: Introduction to Parallel Computer Architecture Final Exam

Prof. Naga Kandasamy, ECE Department, Drexel University

June 6, 2013

The exam is due on June 17, 2013. You may work on this exam in groups of up to two people.

1. **(25 points)** Convolution filtering is used in a wide range of image processing tasks, including smoothing, sharpening, and edge detection. This question asks you to implement a 2D separable convolution filter on the GPU.

A convolution filter is a scalar product of the filter weights with the input pixels within a window surrounding each of the corresponding output pixels. More specifically, given a vector s and a convolution kernel k of size n, the convolution $r(i)$ of the i^{th} element of the vector is given by

$$r(i) = \sum_n s(i-n)k(n).$$

The elements at the boundaries, that is, elements that are "outside" the vector s are treated as if they had the value zero. Convolution can be easily extended into two dimensions by adding indices for the second dimension. For example, given a 2D matrix s and a convolution filter k of dimension $n \times m$, where n is the width and m is the height of the filter, the convolution of element (i, j) in s is given by

$$\sum_n \sum_m s(i-n, j-m)k(n, m).$$

A 2D convolution filter requires $n \times m$ multiplications for each output element. Separable filters are a special type of filter that can be expressed as the composition of two 1D filters, one on the rows of the matrix and one on the columns of the matrix. So, a separable filter can be divided into two consecutive 1D convolution operations on the data (row-wise and column-wise), requiring only $n + m$ multiplications for each output element.

The program provided to you accepts no arguments. It creates a random Gaussian kernel of specified width as well as an input matrix consisting of random floating-point values. The size of the matrix is guaranteed to be 8192×8192 elements. A CPU implementation of separable convolution generates a reference solution which will be compared with your GPU program's output. If the solutions match within a certain tolerance, the program prints out "Test PASSED" to the screen before exiting.

Edit the `compute_on_device()` function within the file `separable_convolution.cu` to complete the functionality of 2D separable convolution on the GPU. To achieve this functionality, you may add multiple kernels to the `separable_convolution_kernel.cu` file. The CUDA source files for this question are available in a zip file called `problem_1.zip`.

E-mail me all of the files needed to run your code as a single zip file called `solution_1.zip`.

This question will be graded on the following parameters:

- **(10 points)** You may just use GPU global memory to get the program working correctly without any performance-related optimizations.

- **(15 points)** Improve the performance of your code by using the GPU's memory hierarchy (shared memory, and/or texture memory, and/or constant memory) efficiently to maximize the speedup over the serial version. Other optimizations may include loop unrolling, appropriately sizing the thread granularity, etc.

Provide a brief write-up (two/three pages) 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)** 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 $\int_a^b f(x)\,\mathrm{d}x$.

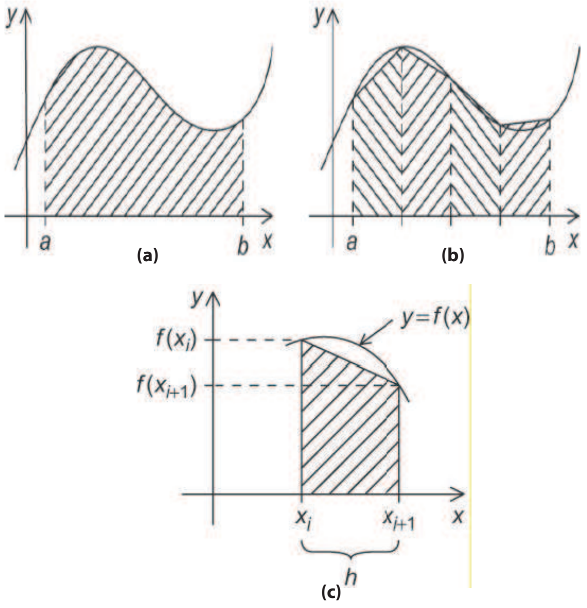

Figure 1: Illustration of the trapezoidal rule: (a) area to be estimated; (b) approximate area using trapezoids; and (c) area under one trapezoid.

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.

```
procedure TRAP($a$, $b$, $n$)
$h := (b - a)/n$;
$sum := (f(a) + f(b))/2.0$;
for $i := 1$ to $n - 1$ step 1 do
    $x_i := a + i \times h$;
    $sum := sum + f(x_i)$;
end for
$sum := h \times sum$;
```

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. The CUDA source files for this question are available in a zip file called `problem_2.zip`.

- **(10 points)** You may just use GPU global memory to get the program working correctly without any performance-related optimizations.
- **(15 points)** Improve the performance of your code by using the GPU's memory hierarchy (shared memory, and/or texture memory, and/or constant memory) efficiently to maximize the speedup over the serial version. Also, quantify the precision issues related to floating point arithmetic on the GPU (when compared to the CPU), and improve the precision of the GPU's result using the Kahan summation technique.

E-mail me all of the files needed to run your code as a single zip file called `solution_2.zip`. Also, provide a brief write-up (two/three pages) 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.