**EE590-A : Applied GPU Computing : Homework-4 (**Due 2/12/2017 by 11:59 PM**)**

**A. Growth of functions analysis**

1. Show your solutions to the following three problems in two ways (a) mathematically using analytical equations; (b) graphically using plots to visually explain the behaviors.
   1. Show that is by directly finding the constants using the definition of “big-Theta Notation” given on slide #8 of Lecture-4. Show your work.

0 <= x + 1 <= 3x2 for x > 1

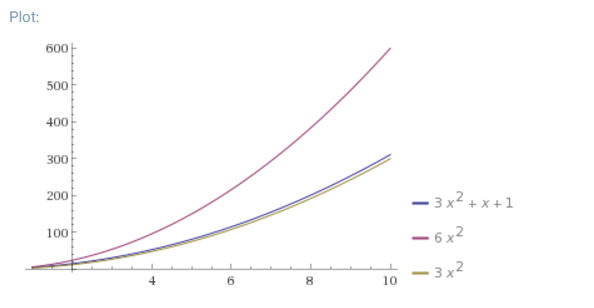
3x2 + x + 1 <= 2\*3x2 for x > 1

C2 = 2 n0 = 1

0 <= 3x2 <= 3x2 + x + 1 for x > 1

C1 = 1, n0 = 1

*THUS*

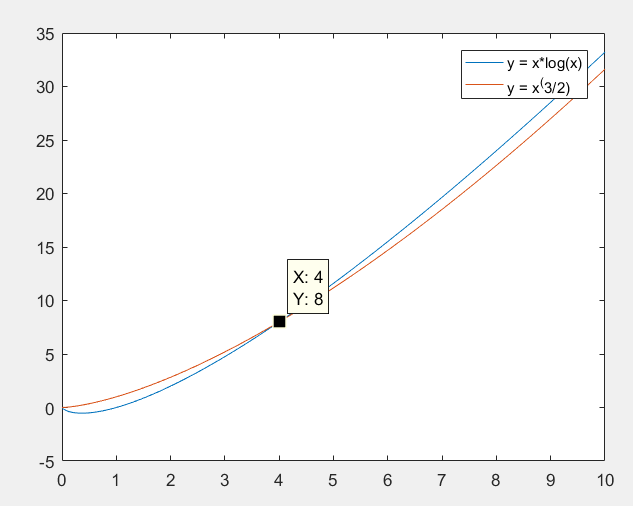


* 1. Suppose you have two different algorithms for solving a problem. To solve a problem of size *n*, the first algorithm uses exactly operations and the second algorithm uses exactly operations. As *n* grows, which algorithm uses fewer operations?

As n > 4, n(log n) is greater.

For n = 6, n log(n) = 15.51

N ^ (3/2) = 14.7



x = 0:.01:10;

y = x.\*log2(x);

y2 = x.^(3/2);

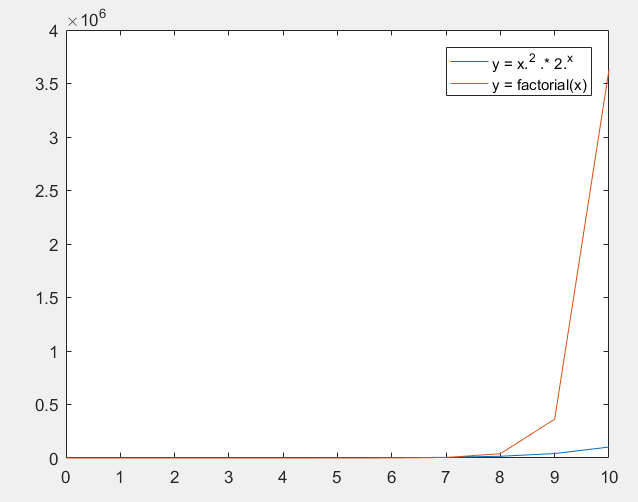
plot(x,y,x,y2)

legend('y = x\*log(x)','y = x^(3/2)');

* 1. Suppose you have two different algorithms for solving a problem. To solve a problem of size *n*, the first algorithm uses exactly operations and the second algorithm uses exactly operations. As *n* grows, which algorithm uses fewer operations?

use fewer operations.

,



>> x = 0:10;

>> y = x.^2 .\* 2.^x;

>> y2 = factorial(x);

>> plot(x,y,x,y2)

>> legend('y = x.^2 .\* 2.^x','y = factorial(x)');

**B. Find Arithmetic Intensity**

For each algorithm, compute W : the work (flops), Q : bytes memory access, AI : arithmetic intensity. Show your work for each step to get full credit. Assume the writes generate same amount of memory traffic as reads (not always the case, see *write-allocate caches*, which GPUs often use).

For each, write the order of the AI function, using big-Theta notation.

1. Outer product
   1. Two vectors of data type short, a is Mx1, b is Nx1

W = (M \* N) Q = (2M + 2N) + 2(M \* N) = 2M + 2N + 2MN

AI = W/Q = (M\*N) / (2M + 2N + 2MN), as M & N grows

* 1. Two vectors of data type double, a is Mx1, b is Nx1

W = (M \* N) Q = (8M + 8N) + 8(M \* N) = 8M + 8N + 8MN

AI = W/Q = (M\*N) / (8M + 8N + 8MN), as M & N grows

1. General matrix-matrix product
   1. C = AB, where matrices A is MxN, B is NxL, and output C is MxL, all are type float

W = (N + N – 1) \* M \* L Q = 4MN + 4NL + 4ML

AI = W/Q = =, as M, N & L grows

~

1. Real quadratic form:
   1. Where A is a real symmetric NxN matric, and vector x is Nx1, all are doubles.

W = ((N + N – 1) \* N) + ((N + N - 1) \* 1) = 2N2 + N - 1

Q = (8N2 + 8N) + 8 = 8N2 + 8N + 8

AI = W/Q =

**C. Evaluating speedup and efficiency, and Amdahl’s Law**

1. Suppose you want to achieve a speedup of 90 using 100 processors. What percentage of the original computation can be sequential (non-parallelizable) code?

Let be the non-parallelizable serial fraction of the total work. Then the following inequalities hold:

Substitute Sp = 90, P = 100

* 1. What is the resulting efficiency? 90/100 = 90%

1. Suppose you want to perform two sums: one is a scalar sum of 10 variables, and one is a matrix sum of a pair of two-dimensional arrays, with dimensions 10 x 10. Assume only the matrix sum is parallelizable. What speedup do you get with 10 versus 40 processors?

Speed up of 1 for scalar sum, because nothing is parallelizable. Increasing number of processors will only increase the overhead of setting up, and does not contribute to solving the problem.

Speed up of 4 for matrix sum, because assuming scalar sum of 10 variables cannot be parallelizable, and each row of the matrix cannot be parallelize. This matrix have 10 rows, having 4 times the processors would allow it to tackle 4 rows when 10 processors finished 1 row.

* 1. Next calculate the speedups if the matrices grow to 20 x 20.

Speed up of 1 for scalar sum.

Speed up of 16 for matrix sum, because 20x20 matrix can be broken down to four 10 x 10 matrix. And since 40 processors would bring 4 times the speed up to 10 x 10 matrix, four 10 x 10 matrix would result of 4 \* 4 = 16 times speed up.

**D. Applying the Roofline performance model**

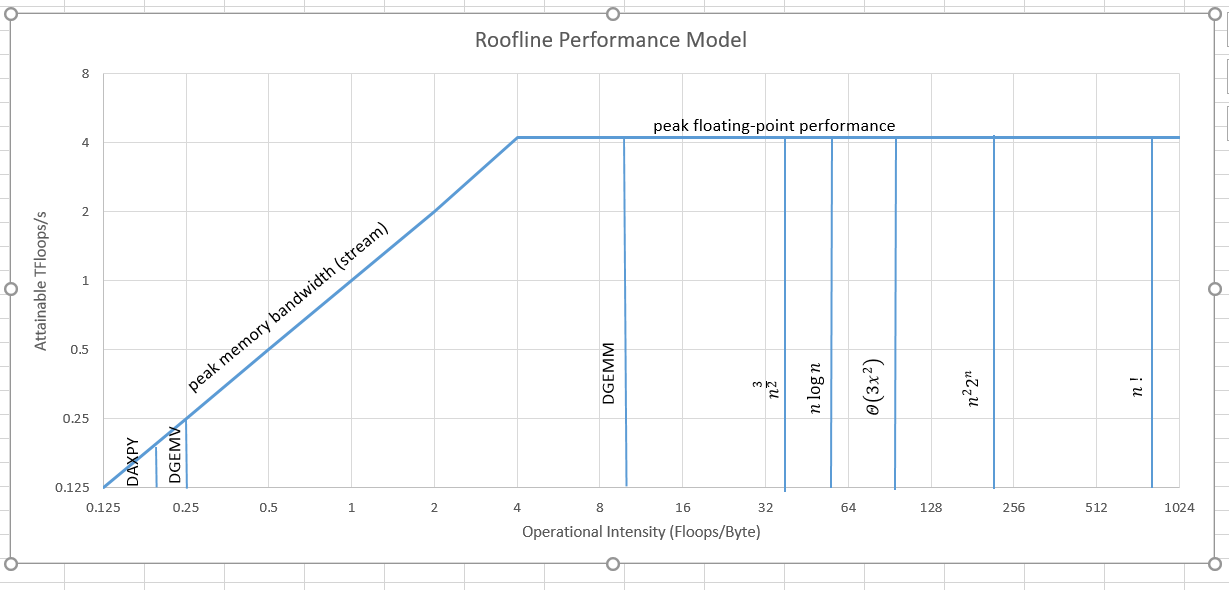
1. Using the published specs (memory bandwidth, peak single-precision floating-point performance) for the AMD Radeon FirePro W8100 GPU

<http://www.amd.com/en-us/products/graphics/workstation/firepro-3d/8100>

Create a Roofline plot for the device. Plot the vertical lines for the AIs of the kernels computed in section A of the homework. Also plot the vertical lines for the AIs of the three BLAS kernels we analyzed in class during Lecture-4 (DAXPY, DGEMV, DGEMM). Remember the Roofline AI is defined in terms of *bytes* of memory traffic.

Memory bandwidth: 320 Gb/s

peak single-precision floating-point performance: 4.2 TFLOPS



1. Next go back to Lecture-3 slides at look at Slide 37 which shows the specs for our HD 530 Intel Processor Graphics GPU. If you also refer to slide 43, you can do a similar computation for the HD 530 GPU performance in terms of FLOPS/sec, using the appropriate # EUs and use the given graphics max dynamic frequency for the peak clock rate.
   1. Add the Roofline lines for the HD 530 to the graph along with the AMD FirePro.

