Homework_1

September 28, 2014

1 CSE891 Homework 1

Camille Welcher

```
In [3]: import pandas as pd
        import numpy as np
        import seaborn as sns
```

1.1 Problem 1

Consider a multi-core CPU with p cores. The peak single-precision performance of each core is 10 GFlops/s. The peak bandwidth for the memory bus is 60 GB/s. Assume that you are evaluating a polynomial of the form, $x = y^2 + z^3 + yz$, which can be implemented as follows:

```
float x[N], y[N], z[N];
for (i=0; i < N; ++i)
    x[i] = y[i]*y[i] + z[i]*z[i]*z[i] + y[i]*z[i];
```

1.1.1 Part A

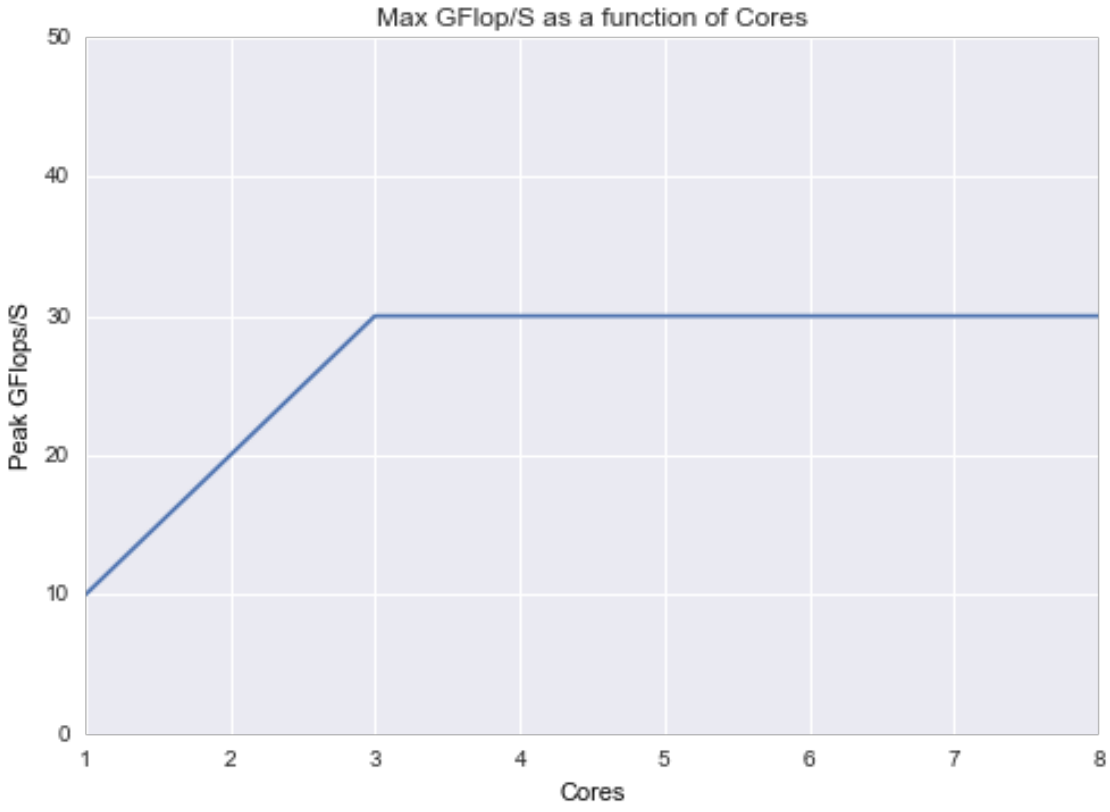
What is the arithmetic intensity (flops/byte) of each iteration of the loop? Think of how many bytes need to be transferred over the memory bus (both reads and writes) and how many floating point operations are performed.

For each iteration, there are 6 floating point ops (4 mults, 2 adds). There is 1 write (the assignment of `x[i]`) and there are 2 reads (`y[i]` and `z[i]`), assuming that once we've fetched a value from memory, we can use the cached value for repeated uses; with single precision, each r/w is 4 bytes, so we get 12 bytes total. This means that our arithmetic intensity is $6/12 = .5$ flops/byte.

1.1.2 Part B

Plot the theoretical peak performance of the system (GFlops/s) as a function of the number of cores, where $1 \leq p \leq 8$. For what values of p would this computation's performance be compute-bound and for what values would it be memory-bound?

```
In [4]: X_p = np.arange(1,9)
        flops_func = np.vectorize(lambda x: min(x * 10, 60 * .5))
        Y_f = flops_func(X_p)
        plot(X_p, Y_f)
        xlabel('Cores')
        ylabel('Peak GFlops/S')
        title('Max GFlop/S as a function of Cores')
        axis(ymax=50, ymin=0)
        savefig('1a.png')
```



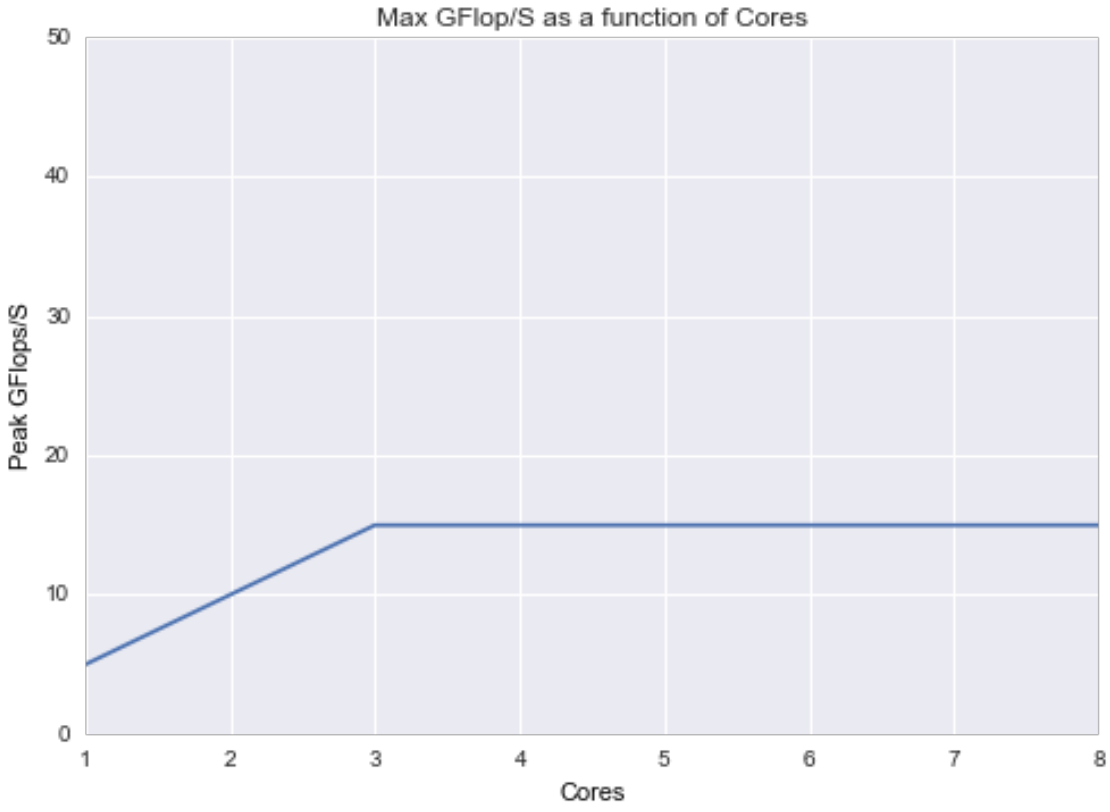
In a completely ideal system, GFlops/S would continue to scale with the number of cores until saturating the memory bus; there are 2 bytes of memory used per 1 flop, so we saturate the memory bus at 3 cores and 30 GFlops/S. Under 3 cores, we are compute bound, and over, we are memory bound.

1.1.3 Part C

Repeat A and B for the same computation but this time in double-precision (i.e. double x[N], y[N], z[N]). The peak double-precision performance per core is half of its single- precision peak - 5 GFlops/s.

Now, $q = 0.25$, as we've doubled the number of bytes. We still saturate the memory bus at 3 cores, but peak performance is now limited to 15 GFlops/S.

```
In [5]: X_p = np.arange(1,9)
        flops_func = np.vectorize(lambda x: min(x * 5, 60 * .25))
        Y_f = flops_func(X_p)
        plot(X_p, Y_f)
        xlabel('Cores')
        ylabel('Peak GFlops/S')
        title('Max GFlop/S as a function of Cores')
        axis(ymax=50, ymin=0)
        savefig('1c.png')
```



1.2 Problem 2

Consider the same architecture as above - p cores, each with 5 GFlops/s peak double- precision performance and 60 GB/s memory bandwidth. Let there be only a single level of cache with a total bandwidth of 300 GB/s (from cache to all cores). Now assume that your computational kernel has an arithmetic intensity of 0.125 flops/byte and a cache hit ratio of 50%. Plot the performance of the system as a function of the number of cores p. Denote the ranges for p, where the peak performance is bound by the CPU, the memory bandwidth and the cache bandwidth (if applicable).

```
In [6]: B_cache = 300.0
        B_mem = 60.0
        F_p = 5.0
        q = .125

        X_p = np.arange(1,9)

        def flops_func(ncores, cmr=.5):

            cpu_perf = ncores * F_p
            bytes = float(cpu_perf) / q
            max_cache_bytes = B_cache * cmr
            if bytes < max_cache_bytes:
                return cpu_perf
            bytes_to_mem = bytes - max_cache_bytes
            if bytes_to_mem < B_mem:
```

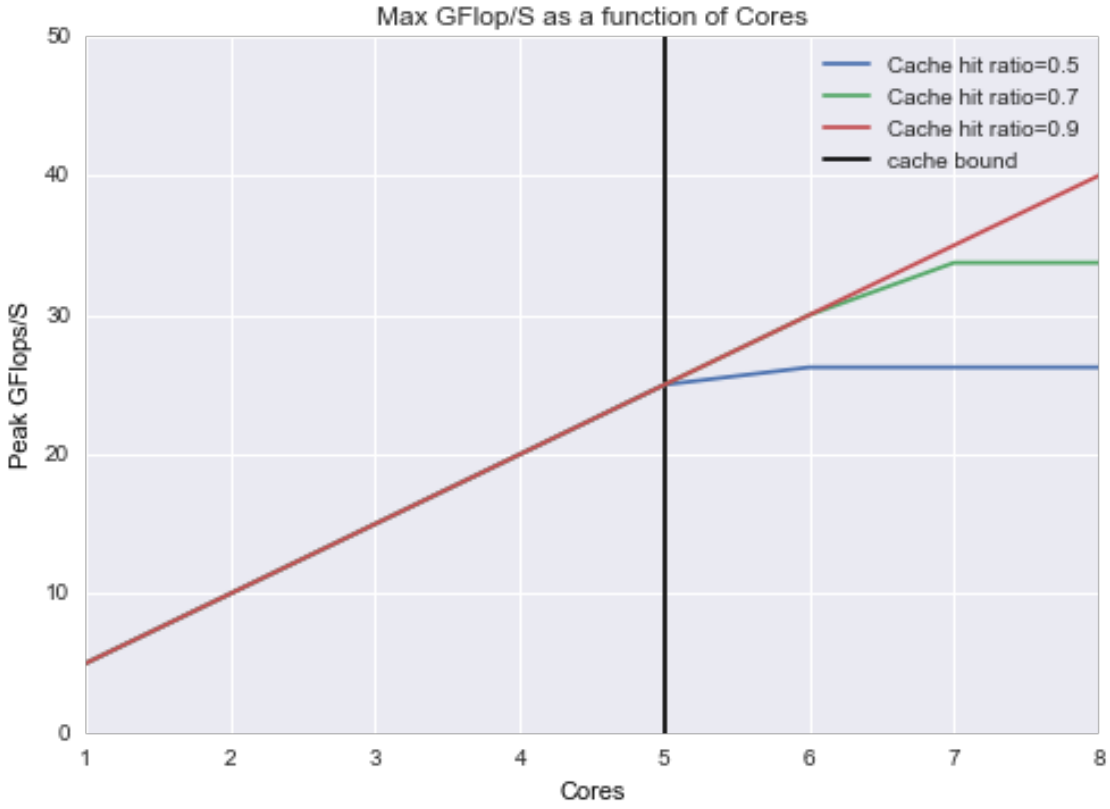
```
            return max_cache_bytes * q + bytes_to_mem * q
        else:
            return max_cache_bytes * q + B_mem * q

v_flops_func = np.vectorize(flops_func)

for cmr in [.5, .7, .9]:
    Y_f = v_flops_func(X_p, cmr=cmr)
    plot(X_p, Y_f, label='Cache hit ratio={}'.format(cmr))

vlines(5, 0, 50, label='cache bound')

xlabel('Cores')
ylabel('Peak GFlops/S')
title('Max GFlop/S as a function of Cores')
axis(ymax=50, ymin=0)
legend()
savefig('2.png')
```



1.3 Problem 3

Assume you are the director of Simulation University's HPC center and you are given a $1 million budget (you may go over the budget by up to 2% if you have a good justification to do so) to purchase a new system. You did some market research and found out that the compute nodes you like cost $5,000/node and each interconnect link (together with the necessary network hardware) costs $1000/piece. You did a survey among

the HPC center! users and (on average) they rated the utility of an additional compute node as 5 and the utility of increasing the bisection bandwidth of the network by one link as 3. Think of the network topologies we discussed in class - bus, ring, 2D/3D torus (or k-D torus), hypercube,tree or a fat-tree. How many nodes would you buy and how would you connect them? What if the users rate the compute vs. communicate utilities as 5 vs 1?

1.3.1 Utility = (5,3)

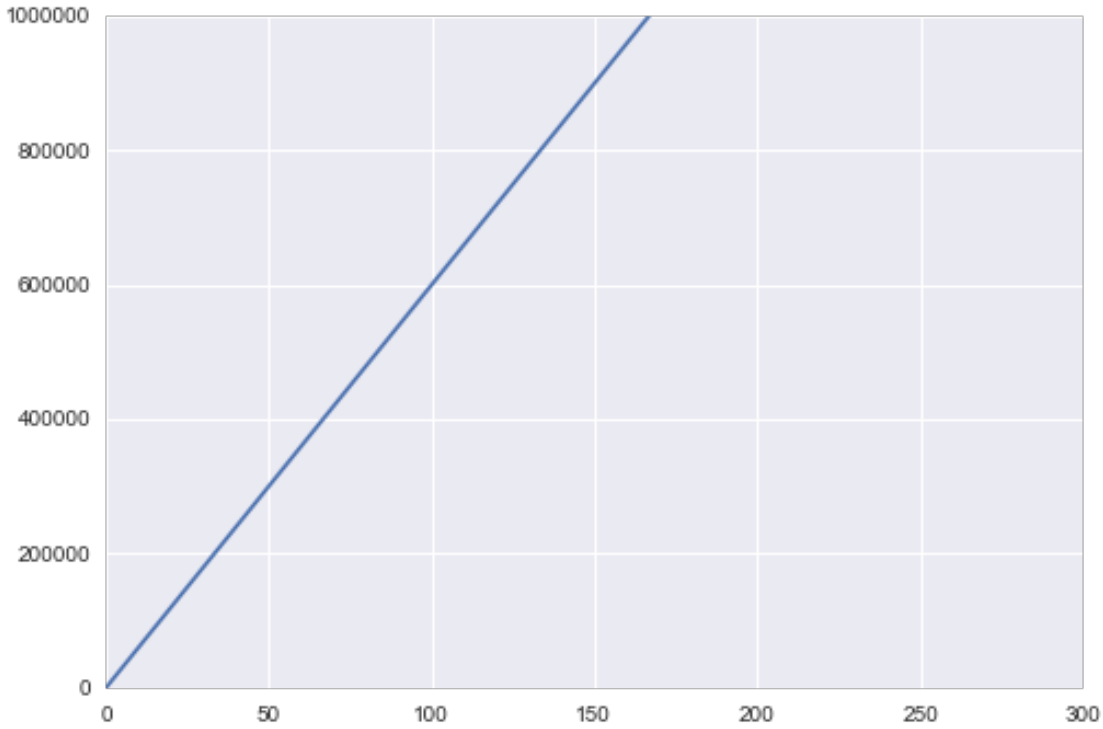
```
In [11]: B = 1000000.0

In [12]: def utility(D, b, u_d=5, u_b=3):
             return D * u_d + b * u_b

In [13]: def cost_ring(p, cp=5000, cl=1000):
             return int(p) * cp + cl * p
         # return D, edges, bisect
         def stat_ring(p):
             return int(p) / 2, p, 2
         v_cost_ring = np.vectorize(cost_ring)

In [14]: X_p = np.arange(0,300)
         c = v_cost_ring(X_p)
         plot(X_p, c)
         axis(ymax=B)
         max_p_ring = np.argmax(c[c <= B])
         print 'Max nodes for budget:', max_p_ring
         savefig('3a.png')
```

```
Max nodes for budget: 166
```



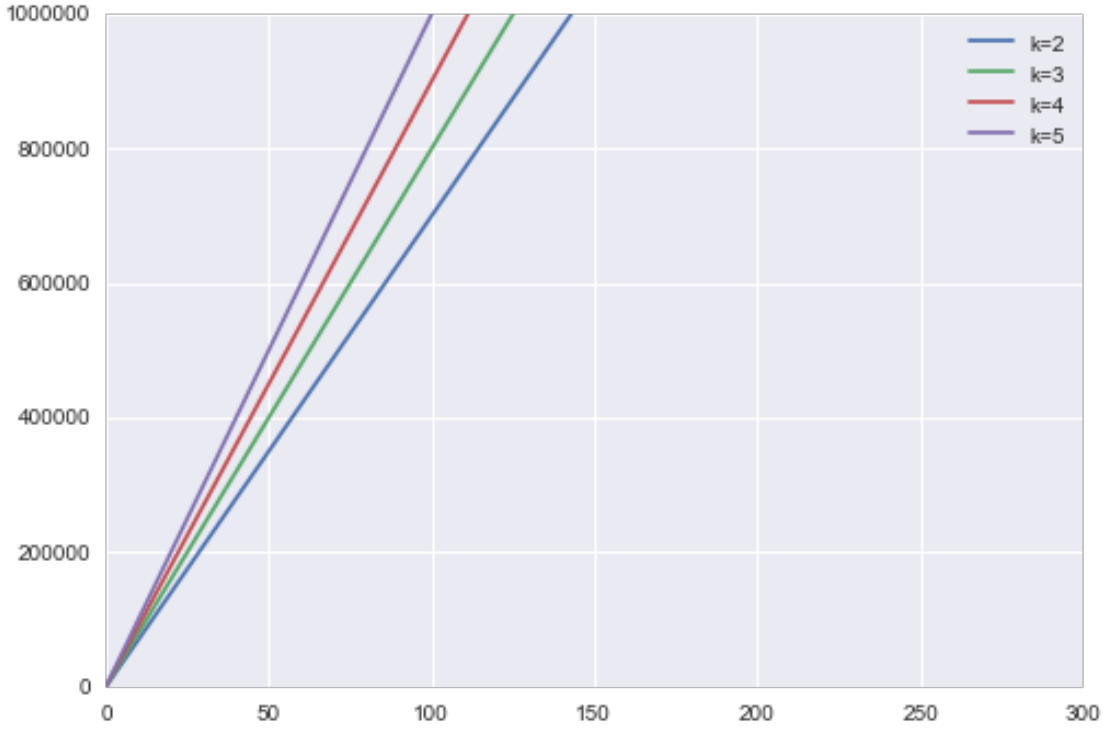
```
In [15]: D, e, b = stat_ring(max_p_ring)
         print 'Utility of ring with {} nodes:'.format(max_p_ring), utility(D, b)
```

```
Utility of ring with 166 nodes: 421
```

```
In [16]: def cost_torus(p, k, cp=5000, cl=1000):
             return int(p) * cp + k * cl * p
         # return D, edges, bisect
         def stat_torus(p, k):
             return int(k) * int(p)/2, int(k) * int(p), 2 * p
         v_cost_torus = np.vectorize(cost_torus)
```

```
In [17]: X_p = np.arange(0,300)
         max_p_torus = []
         for k in range(2,6):
             c = v_cost_torus(X_p, k)
             plot(X_p, c, label='k={}'.format(k))
             m = np.argmax(c[c <= B])
             max_p_torus.append((m, k))
             print 'Max nodes for budget at k={}:'.format(k), m
         axis(ymax=B)
         legend()
         savefig('3b.png')
```

```
Max nodes for budget at k=2: 142
Max nodes for budget at k=3: 125
Max nodes for budget at k=4: 111
Max nodes for budget at k=5: 100
```

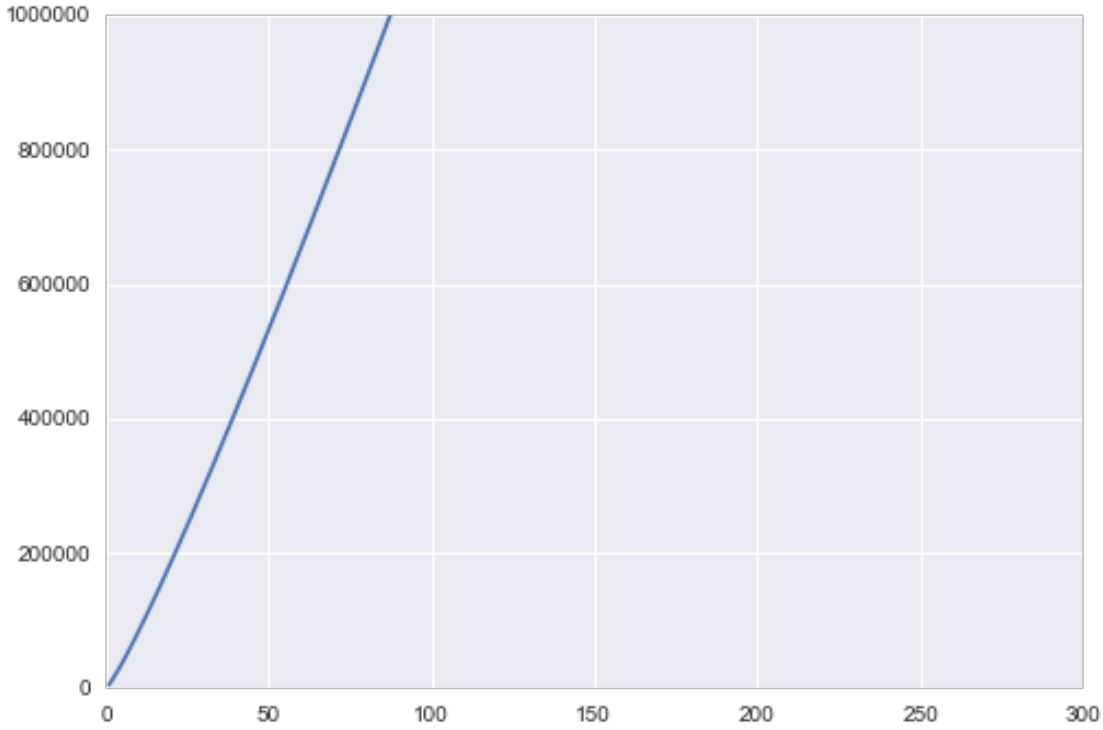


```
In [18]: for p, k in max_p_torus:
             D, e, b = stat_torus(p, k)
             print 'Utility of torus with {} nodes, k={}:'.format(p, k), utility(D, b)
```

```
Utility of torus with 142 nodes, k=2: 1562
Utility of torus with 125 nodes, k=3: 1685
Utility of torus with 111 nodes, k=4: 1776
Utility of torus with 100 nodes, k=5: 1850
```

```
In [19]: def cost_tree(p, cp=5000, cl=1000):
             return int(p) * cp + cl * p * log2(p)
         # return D, edges, bisect
         def stat_tree(p):
             return 2 * log2(p), int(p) * int(log2(p)), int(p/2)
         v_cost_tree = np.vectorize(cost_tree)
```

```
In [20]: X_p = np.arange(0,300)
         c = v_cost_tree(X_p)
         plot(X_p, c)
         axis(ymax=B)
         max_p_tree = np.argmax(c[c <= B])
         savefig('3c.png')
```




```
In [21]: D, e, b = stat_tree(max_p_tree)
         print 'Utility of fat tree with {} nodes:'.format(max_p_tree), utility(D, b)
```

```
Utility of fat tree with 86 nodes: 193.262647547
```

With our budget and the given utility values, I would choose a 5-D 100-ary torus; this is 100 nodes and 500 links.

1.3.2 Utility = (5,1)

```
In [22]: D, e, b = stat_ring(max_p_ring)
         print 'Utility of ring with {} nodes:'.format(max_p_ring), utility(D, b, u_b=1)

Utility of ring with 166 nodes: 417

In [23]: for p, k in max_p_torus:
             D, e, b = stat_torus(p, k)
             print 'Utility of torus with {} nodes, k={}:'.format(p, k), utility(D, b, u_b=1)

Utility of torus with 142 nodes, k=2: 994
Utility of torus with 125 nodes, k=3: 1185
Utility of torus with 111 nodes, k=4: 1332
Utility of torus with 100 nodes, k=5: 1450

In [24]: D, e, b = stat_tree(max_p_tree)
         print 'Utility of fat tree with {} nodes:'.format(max_p_tree), utility(D, b, u_b=1)

Utility of fat tree with 86 nodes: 107.262647547
```

The 5-D torus still wins out, although lower dimensions inch up in value. This analysis also doesn't factor in some of the complexity of constructing torus networks, or the potential variable bisection bandwidths of a fat-tree.

1.4 Problem 4

The outer product is a commonly used tensor operation. The input is two vectors of length N and M. The result is a matrix of dimensions N by M. In code it would look as follows:

```
double x[N], y[M], A[N][M];
    for (i=0; i<N; ++i)
        for (j=0; j<M; ++j)
            A[i][j] = x[i]*y[j];
```

1.4.1 Part A

Assume that there is only a single level of cache of size 64 KBs, a cache line is 64 bytes and the Least Recently Used (LRU) policy is adopted for cache eviction. For $N = M = 1024$, how many cache misses would the implementation above incur?

```
In [25]: print 'cache holds', 64000 / 8, 'doubles, cache line is 8 doubles'

cache holds 8000 doubles, cache line is 8 doubles
```

`x[N]` and `y[N]` can both reside entirely in cache, consuming 2048 of our 8000 doubles. Misses should only occur when writing into `A`. With a 64 byte cache line that can store 8 doubles, we get $(M * N)/8$ misses, or 131,072.

1.4.2 Part B

Repeat A, but now for $N = M = 10240$.

Now we cannot store the entirity of `x` and `y` in cache. Each of the N iterations of the outer loop will cause a miss, and we'll have $M/8$ misses for accessing `y` and $M/8$ misses for writing into `A`, giving a total of $N * (M/4)$ misses, or 26,214,400 total.

1.4.3 Part C

Based on your insights from 4.A and 4.B, give the pseudo-code for a high performance outer product implementation with a reduced number of cache misses.

Pseudocode:

```
S = floor((cache_size / 8) / 3)
for s_i in 0..N/S:
    loadCache(x[s_i*S:(i+1)*S])
    for s_j in 0..M/S:
        loadCache(y[s_j*S:(j+1)*S])
        loadCache(A[s_i*S:(s_i+1)*S, s_j*S:(s_j+1)*S])
        for ii in s_i*S..(s_i+1)*S:
            for jj in s_j*S..(s_j+1)*S:
                A[ii,jj] = x[ii] * y[jj]
```

Thus, the idea is to subdivide `A` into square submatrices, such that we can load the ranges of `x` and `y` and single row of the submatrix of `A` into memory for each submatrix. Traversal order would follow the pattern of the following matrix:

```
In [26]: N = 25
         A = np.zeros((N,N))
         S = 5
         z = 0
         for s_i in range(N/S):
             for s_j in range(N/S):
                 z += 1
                 for ii in range(s_i * S, (s_i + 1) * S):
                     for jj in range(s_j * S, (s_j + 1) * S):
                         A[ii,jj] = z

In [27]: A[0:10,0:10]

Out[27]: array([[ 1.,  1.,  1.,  1.,  1.,  2.,  2.,  2.,  2.,  2.],
                [ 1.,  1.,  1.,  1.,  1.,  2.,  2.,  2.,  2.,  2.],
                [ 1.,  1.,  1.,  1.,  1.,  2.,  2.,  2.,  2.,  2.],
                [ 1.,  1.,  1.,  1.,  1.,  2.,  2.,  2.,  2.,  2.],
                [ 1.,  1.,  1.,  1.,  1.,  2.,  2.,  2.,  2.,  2.],
                [ 6.,  6.,  6.,  6.,  6.,  7.,  7.,  7.,  7.,  7.],
                [ 6.,  6.,  6.,  6.,  6.,  7.,  7.,  7.,  7.,  7.],
                [ 6.,  6.,  6.,  6.,  6.,  7.,  7.,  7.,  7.,  7.],
                [ 6.,  6.,  6.,  6.,  6.,  7.,  7.,  7.,  7.,  7.],
                [ 6.,  6.,  6.,  6.,  6.,  7.,  7.,  7.,  7.,  7.]])
```