CSE415 Homeowork 2
Created by: Ian McGregor
2/10/2018

1) Performance Modeling: Assume you are evaluating a polynomial of the form, $x = y^2 + z^3 + yz$, which can by 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];
```

Here i is an integer and x, y and z are single precision floating point arrays.

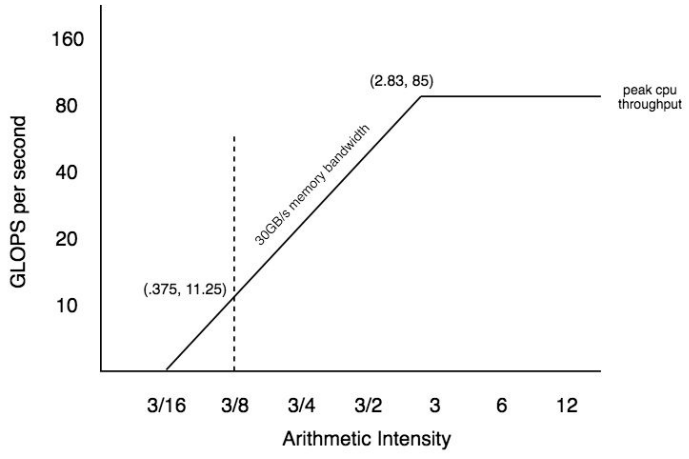
a. What is the arithmetic intensity of this kernel?

6N FLOPS = 4 multiply + 2 add operations per iteration
16N bytes to/from memory =
(3N floats loaded from memory (x[N] + y[N] + z[N)
+ 1N floats stored back to memory (x[N]))
* 4 bytes (size of floats)
Arithmetic intensity = 3 / 8

b. Assume this kernel is to be executed on a processor that has 30 GB/sec of memory bandwidth. Under what conditions will this kernel be memory bound, and under what conditions will it be compute bound?

Memory bound at peak computational throughput < 11.25 GFLOPS/s
Compute bound at peak computational throughput > 11.25 GFLOPS/s

c. Develop a roofline model for this processor, assuming it has a peak computational throughput of 85 GFLOP/sec. Mark the arithmetic intensity of the kernel on your plot and determine its expected performance.



2) [75 pts] Memory hierarchy: Matrix Vector Multiplication

A simple nested loop is sufficient to multiply an NxM matrix with a Mx1 vector resulting in an Nx1 vector:

```
for (i = 0; i < N; i++) {
   for (j = 0; j < M; j++) {
      output[i] += input_matrix[i][j]*input_vector[j];
   }
}
```

However, as we discussed in class, this simple implementation can not make efficient use of the memory hierarchy. In particular, assuming row-major storage, while the accesses to the input matrix can achieve ideal cache miss rates, the accesses to the output matrix may result in high number of cache misses.

a. Implementation: "src/optMultiplication.c" contains a cache optimized version of the naive matrix multiplication algorithm in "src/naiveMultiplication.c". The implementation relies on two additional outer loops which track the current block index on each dimension of the matrix. Each outer loop iterates from 0 to the index of the last whole block division of its respective dimension as determined by the length (N or M) * 4 (bytes per float) / B (block size in bytes).The two inner loops are modified from the naive algorithm to start at the front of the current block and iterate only over the items in that block. Remaining items left by an uneven matrix/block size division are handled by separate structures which begin at the end of the last whole block division and end once the index has reached the end of the dimension length (N or M). So, there are 4 end cases handled by this logic: fullBlockN x fullBlockM, fullBlockN x partialBlockM, partialBlockN x fullBlockM, and partialBlockN x partialBlockM.

b. Performance Analysis: Performance data can be found in the "./performanceData" folder and is further organized by order of magnitude of the input_matrix N dimension. Where $N= 10^i$, the file "i/output.txt" contains the output of a performance execution for each order of M from 10^1 to (10^8/N), for each binary ordered block size B from 2^4 to 2^{10}. In general the data says that a smaller matrix is less susceptible to influence by the cache blocking. A smaller number in either dimension seems to reduce the standard deviation of performance increase or decrease of the cache blocked output. All tests follow the same trend of increase and decrease for changes in block size but the larger matrices see greater improvement for optimal block sizes and the detriment for the suboptimal.

c. Cache Performance Measurement: The folder "./tauData" contains the performance output of each run followed by the tau performance data for time, L1 data cache misses, L2 total cache misses, and L3 total cache misses, organized in the same structure as "./performanceData". The root cause for the behavior explained above is that any small N or M has less iterations to exploit a difference between the two source codes. Whether a bad or good block size, the worst that can happen on a small matrix is a few lines of overhead. The extreme is a matrix that fits entirely within the L1 cache in which case any instruction beyond the naive solution is unnecessary.

d. Inference about the Memory Hierarchy: L1 misses see the most improvement at 128x128 byte blocks which would likely mean a 32K cache since the matrix alone is 16K, which such an increase in performance the cache should be holding onto the two vectors as well for each block. This matches the hardware specs, however I can't find any data to support improvement due to cache blocking for the 256K L2 cache, mostly the difference in L2 cache misses is equivalent to the L1 ratio.