Problem 1 (15%): Compute Cache performance is a factor of several parameters. For each of these, describe the issues that arise if their value is either too small or too large:

a. Cache size
b. Line size
c. Associativity

a. Cache Size

   If the cache size is too small, it risks not being able to hold frequently accessed data or instructions. I.e., the miss rate increases, and the processor will have to fetch data from lower and slower levels of cache and memory more often.

   As the cache size increases, relative access time generally increases. I.e., the time it takes to search the cache for a specific piece of data increases, the hit time. Additionally, there's the concept of diminishing returns due to the principle of locality since most programs tend to hit smaller, localized regions of space at a time rather than uniformly.

b. Line Size

   If the line size is too small, more blocks are needed to fill the cache which creates a lot of overhead since the processor will keep needing to go to lower, slower levels of cache and memory to fetch the needed data. Additionally, for programs that need to continuously access a localized region of space, it might miss adjacent data that'll need to be accessed which further reduces performance and efficiency.

   If the line size is too big, you risk wasting space and bandwidth since you're prefetching data that may not be needed.

c. Associativity

   If associativity is too small, it leads to high miss rates if multiple blocks of memory frequently accessed map to the same cache block since you're constantly having to switch them out.

   On the other hand, if the associativity is too high, you may have to search the entire cache for a block, making it slower.

Problem 2 (50%): Consider the matrix_add function shown below:

```
int matrix_add(int a[128][128], int b[128][128], int c[128][128])
{
   int i, j;
   for(i = 0; i < 128; i++)
   for(j = 0;j < 128; j++)
      c[i][j] = a[i][j] + b[i][j];
      return 0;
}
```

In each iteration, the compiled code will load a[i][j] first, and then load b[i][j]. After performing the addition of a[i][j] and b[i][j], the result will be stored to c[i][j]. The processor has a 64KB, 2-way, 64Byte-block L1 data cache, and the cache uses LRU policy once a set if full. The L1 data cache is write-back and write-allocate. If the addresses of array a, b, c are 0x10000, 0x20000, 0x30000 and the cache is currently empty, please answer the following questions:

a. What is the L1 D-cache miss rate of the matrix_add function? How many misses are contributed by compulsory miss? How many misses are conflict misses?

b. If the L1 hit time is 1 cycle, and the L1 miss penalty is 20 cycles. What is the average memory access time?

a.

To calculate compulsory misses, we need to figure out how many times the processor must reach into the cache to fetch unknown data. This occurs at each iteration when we pull in new rows for matrix A and matrix B.

The problem statement states the cache is 64 KB, 2-way, and 64-byte blocks. This means that each cache block holds 16 integers, and each cache set holds 32 integers. Each matrix row needs 4 cache sets, or 8 cache blocks. First, we need to focus on the inner loop. At $i = 0$, there are going to be 128 iterations that are compulsory since this'll be the first time the cache sees matrix B. So, $128 * 8 = 1024$ compulsory misses. It also pulls in Matrix C to manipulate, so that's another 1024 compulsory misses. Next, the outer loops. On the first loop, it sees all of matrix B, but by focusing on just the outer loop itself, we see that there are also $128 * 8 = 1024$ compulsory misses. Therefore, there are 3072 compulsory misses.

For conflict misses, we need to figure out how many cache sets are available.

$$64\text{ kb} * \frac{1024\text{ bytes}}{1\text{ kb}} * \frac{1\text{ block}}{64\text{ bytes}} * \frac{1\text{ set}}{2\text{ blocks}} = 512\text{ sets}$$

We know that each cache set holds 2 lines; however, there are 3 matrices. If a set's lines are already occupied and a third one maps to the set, there will be a conflict miss. Each iteration pulls in 128 new numbers or fills in 8 cache blocks. Since each set can only hold 2 blocks from the same matrix, the next 6 cache blocks will be conflict misses.

Therefore, with 384 matrix rows (128 from each matrix), there'll be $384 * 6 = 2304$ conflict misses.

b.

$$\text{AMAT} = \text{HT} + (\text{MR} * \text{MP})$$

$$\text{AMAT} = 1 + \left(\frac{2304 + 3072}{3 * 128 * 128} + 20\right) = 3.44$$

Problem 3 (35%): You are given a cache that has 16 byte blocks, 512 rows, and is 2-way set associative. Integers are 4 bytes. Give the C code for a loop that has a 100% miss rate in this cache but whose hit rate rises to almost 100% if you double the size of the cache.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char* argv[]) {
    // Cache Parameters
    // 16 Bytes per block
    // 512 rows
    // 2-Way Set Associative
    // 16 KB then 32 KB, need 4096 and 8192 ints

    int array_size = (16 * 512 * 2) * 2;
    int arr[array_size];
    for (int i = 0; i < array_size; i++) {
        arr[i] = i;
    }

}
```