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.

Compulsory misses will only occur at a$[0][0]$ and b$[0][0]$ since the cache is initially empty. The problem statement is set up conveniently to allow us to pull the entirety of matrix A and B in. We can do that at the start so, 2 misses.

Since the L1 cache is "small", each iteration causes new data to be pulled in and successive accesses to a$[i][j]$ and b$[i][j]$ are going to compete with existing cache sets. There are $128 * 128$ iterations, so 16384 total iterations. Since we are pulling from two sets, there are $16384 * 2 = 32768$ load operations. Due to the finite space of our cache, there is going to be $32768 - 2 = 32766$ conflict misses.

b.

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

$$\text{AMAT} = 1 + \left(\frac{32766}{32768} * 20\right) = 20 \text{ cycles}$$

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.

```
#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;
    }

}
```