1. Consider the following program:

```c
int A[1000];
int B[1000][1000];
for (j = 1; j < 1000; j++) {
    for (k = j; k < 1000; k++) {
    }
}
```

(a) Draw the iteration space for this loop.

(b) Write the access functions for each of the accesses in this loop, in the form of $F_i + f$.
(c) For each pair of accesses which might have a data dependence (e.g. they refer to the same array and are not both read operations), write down the integer linear programs which need to be solved to identify a data dependence between those accesses.

(d) Which pairs of accesses have a data dependence? For each dependence, give example iterations $i$ and $i'$ where this data dependence occurs.

(e) Find an affine partition mapping for the loop body statement which maximizes the degree of communication-free parallelism in the loop. This mapping should be of the form $C_i + c$. 
(f) Now, it is time to get familiar with the Omega calculator. Refer to the Omega tutorial posted on the class website. Use the Omega calculator to generate an SPMD code that uses the affine partition mapping obtained in part (e). Show your input file and the code generation result you get from the Omega calculator.
2. Consider the following LU-decomposition program:

```c
for (i= 1; i < 10000; i++) { /* Loop1 */
    for (j = i + 1; j < 10000; j++) { /* Loop2 */
        A[j,i] = A[j,i]/A[i,i]; /* (s1) */
    }
}
for (i= 1; i < 10000; i++) { /* Loop1 */
    for (j = i + 1; j < 10000; j++) { /* Loop2 */
        A[j,i] = A[j,i]/A[i,i]; /* (s1) */
    }
}
    }
}
    }
```

(a) Show that Loop 1 is not parallelizable.

(b) Use the Omega calculator to generate an SPMD code for Loop2. Show your input file and the result obtained.
3. With the help of the Omega Calculator, parallelize the following program:

```c
for (i = 0; i < 1000; i++) {
    for (j = 0; j < 1000; j++) {
        for (k = 0; k < 1000; k++) {
            A[i,j,k] = A[i+1,j-1,k] + 1; /* (s1) */
            B[i,j,k] = B[i,j,k-1] + A[i-1,j-1,k-1]; /* (s2) */
        }
    }
}
```

Show the input file you use to generate the parallelized SPMD code as well as the results obtained.
4. Consider a programming language such as C that lays out matrix data in row-major order. That is, consecutive elements in a matrix row are contiguous in memory. Now consider the following program:

```c
int A[10000,512];
for (i = 0; i < 512; i++) {
    for (j = 0; j < 9999; j++) {
    }
}
```

An access \( A[j,i] \) refers to row \( j \), column \( i \) of the matrix. Each entry in the matrix uses one word in memory. Suppose the machine we will execute this program has a tiny cache with a 1024 word capacity, where cache lines are 4 words in length (i.e., each cache miss loads 4 contiguous words into the cache). Also suppose that the cache is fully associative and LRU cache replacement policy is used (i.e., a least recently used cache line will be replaced when the cache is full).

(a) If we execute this code, how many cache misses are there?

(b) Describe how this code can be transformed to improve its cache locality so that the number of cache misses can be reduced by at least \( 2 \times \).

(c) How many cache misses are there in the optimized program?