Note: A subset of problems are marked with a red star (★). We especially encourage you to try these out before recitation.

Problem 1: Loop Reordering ★

Assume all arrays are stored in row-major order. Reorder the following loops so they have the best data locality.

(A)

```python
for i in range(0, N):
    for j in range(0, N):
        for k in range(0, N):
            C[k][i] += A[k][j] * B[j][i]
```

(B)

```python
for i in range(0, N):
    for j in range(0, M):
        C[j][i] = A[i][j] + B[j][i]
```
for i in range(0, N):
    for j in range(0, N):
        for k in range(0, N):
            for m in range(0, N):
                A[i][j][m] += B[m][i][k] * C[j][m][i]
Problem 2: Loop Reordering & Caches ★

Consider the following loop:

```python
sum = 0
for j in range(0, 32):
    for i in range(0, 32):
        sum += A[i][j]
```

Consider running this program using a direct mapped data cache of 64 words with a block size of 4. This data cache is separate from the instruction cache.

(A) How many data cache misses will there be during execution of this program?

Now consider reordering the loop:

```python
sum = 0
for i in range(0, 32):
    for j in range(0, 32):
        sum += A[i][j]
```

(B) How many data cache misses will there be during execution of this program?
Now consider the following loop:

```python
sum = 0
for i in range(0, 32):
    for j in range(0, 32):
        for k in range(0, 32):
            sum += A[i][j] * B[j][k]
```

Now run this program on a 2-way set associative data cache with 128 total words, and a block size of 4. The replacement policy is LRU.

(C) How many cache misses will there be in the execution of this code?
Problem 3: Loop Tiling ★

(A) Tile the following loops using a tile size of T.

```python
for i in range(0, N):
    for j in range(0, N):
        for k in range(0, N):
            for m in range(0, N):
                A[i][m] += B[i][j] * C[j][k] * D[k][m]
```

(B) For each of the new subloops introduced by tiling, report the number of elements accessed for each array.

<table>
<thead>
<tr>
<th>Loop</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
</table>
(C) What cache size would prevent the maximum number of cache misses within the subloops?
Problem 4: Loop Unrolling ★

Consider the following C code:

```c
// arr is an 1024-element array of ints
int sum = 0;
for (int i = 0; i < 1024; i++) {
    sum += arr[i];
}
```

(A) Perform loop unrolling on the code with an unrolling factor of 4.

(B) Assume the code is translated to RISC-V assembly in a direct manner. How many times is a branch instruction encountered in the original code? In the unrolled code?

Original Code: _______________

Unrolled Code: _______________
Now consider the code, modified as follows:

```java
// arr is an 1027-element array of ints
int sum = 0;
for (int i = 0; i < 1027; i++) {
    sum += arr[i];
}
```

(C) Perform loop unrolling on the code with an unrolling factor of 4.

(D) For each of the following, circle the effect loop unrolling can have.

- Code Size: (Increase) (No Change) (Decrease)
- Number of Instructions Executed: (Increase) (No Change) (Decrease)
- Readability: (Increase) (No Change) (Decrease)
- Instruction cache misses: (Increase) (No Change) (Decrease)
- Data cache misses: (Increase) (No Change) (Decrease)
- Values Stored in Memory: (Increase) (No Change) (Decrease)
Consider the following C code:

```
// A is an 1024-element array of ints (32-bit)
// B is an 1024-element array of ints (32-bit)
int sum = 0;
for (int i = 0; i < 1024; i++) {
    sum += A[i] * B[i];
}
```

(E) Perform loop unrolling on the code with an unrolling factor of 4.

(F) Now assume we have access to a SIMD instruction dot_vec4, which takes in a memory location for two arrays, multiplies each of four 32-bit ints pairwise, and sums the result. Rewrite the loop above using this instruction.
Problem 5: Multithreading ★

Consider the following C code:

```c
// N = 64
for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
        for (int k = 0; k < N; ++k) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

(A) If this loop is multithreaded with 8 threads using `#pragma omp parallel for`, what would the loop that thread t runs look like? Assume that the threads are produced *statically* (this is not true in general) and split the iterations of i into 8 even chunks.

(B) What is the maximum number of threads that we could use to split this loop if we use `#pragma omp parallel for` to parallelize the loop? Is there some other way we can benefit from parallelism using more threads?

(C) What resource(s) do these threads share? Does this cause any problems?
Now consider reordering the loops:

```c
// N = 64
for (int k = 0; k < N; ++k) {
    for (int j = 0; j < N; ++j) {
        for (int i = 0; i < N; ++i) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

As discussed before, this causes issues with data locality, but it also has an impact on multithreading.

(D) If this loop is multithreaded with 8 threads using `#pragma omp parallel for`, what would the loop that thread `t` runs look like? Assume that the threads are produced *statically* (this is not true in general) and split the iterations of `i` into 8 even chunks.

(E) What resource(s) do these threads share? Does this cause any problems?