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]
```

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

In the original loop, all three arrays are read column-by-column, which is bad for data locality. An improvement can be made for array C by swapping the loops over i and k to read it row-by-row. It so happens that this also makes the reads of the other arrays row-by-row, so all three arrays have good data locality.

(B)

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

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

The i->j order is favorable for array A (reads it row by row), but unfavorable for arrays B and C (reads it column by column). It will be overall faster to use the j->i ordering, which preserves data locality on arrays B and C, while sacrificing it on array A.
(C)

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]

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

Arrays A and C use the same indices in different orders, so we can’t have good data locality with both at the same time. They have different orders of m and i, but we will choose to iterate m before i because both B and C use that order. For both A and C, the index j comes before m, so we should iterate that index first. Finally, the index k comes after indices m and i for array B’s accesses, so we should iterate over that index last. This leaves us with an ordering of j-m-i-k.
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?

This loop iterates over the data in A column by column. Each iteration of the outer loop completely replaces the data in the cache with data from column j. Each line brought in to the cache brings in 3 adjacent words, but those are in different columns so they cannot be used in the same iteration of the outer loop. That line will be gone by the next iteration of the outer loop, so this data locality cannot be taken advantage of. Thus, every single element in the array is a cache miss, for a total of $32 \times 32 = 1024$ misses.

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?

This loop iterates over the data in A row by row. Each line brought in to the cache brings in 3 adjacent words, which are the next 3 words to be summed. Only every 4 words is a cold miss, for a total of $32 \times 32 / 4 = 256$ misses.
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?

For a given iteration of the j loop, A[i][j] will remain the same and will be accessed for each iteration of the k loop. This value will be frequently accessed and can thus only be missed once at the beginning but then will definitely be cached in one of the ways for the remaining 31 iterations of the k loop.

A[i][j] will be brought in with 3 adjacent words. Since A[i][j] is frequently used, it (and the 3 other adjacent words) will not be kicked out of the cache by the time A[i][j+1] is requested, so on the next iteration of j, there will be 0 misses for A[i][j] rather than 1.

So, overall, 1 in 128 access of A are misses.

For the B array, there are sequential accesses only. For these accesses, each brings in 3 other adjacent words that will be accessed directly in the next iteration, so only 1 in 4 accesses for B are misses. There is not enough space in the cache to keep the entire B array (32 * 32), so when another i loop is started there will be new misses again.

In total, there are 32 * 32 * 32 = 32768 access of each array. For A, 32768/128 = 256 of these are cache misses. For B, 32768/4 = 8192 of them are cache misses.

Overall, this is 256 + 8192 = 8448 cache misses.
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]
```

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

(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>
<tbody>
<tr>
<td>mm</td>
<td>T</td>
<td>1</td>
<td>1</td>
<td>T</td>
</tr>
<tr>
<td>kk</td>
<td>T</td>
<td>1</td>
<td>T</td>
<td>$T^2$</td>
</tr>
<tr>
<td>jj</td>
<td>T</td>
<td>T</td>
<td>$T^2$</td>
<td>$T^2$</td>
</tr>
<tr>
<td>ii</td>
<td>$T^2$</td>
<td>$T^2$</td>
<td>$T^2$</td>
<td>$T^2$</td>
</tr>
</tbody>
</table>
(C) What cache size would prevent the maximum number of cache misses within the subloops?

4T^2

We need space for all four arrays. If one of the arrays had only cold misses, it might be possible to omit its space requirement (i.e. in a case where each element of the array is only accessed once). However, this is not the case for our loop. With a tile size of T, T^2 elements of each array are accessed.
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.

```c
// arr is an 1024-element array of ints
int sum = 0;
for (int i = 0; i < 1024; i+=4) {
    sum += arr[i];
    sum += arr[i+1];
    sum += arr[i+2];
    sum += arr[i+3];
}
```

(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: _______1024_______

Unrolled Code: _______256_______
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.

```java
// arr is an 1027-element array of ints
int sum = 0;
for (int i = 0; i < 1024; i+=4) {
    sum += arr[i];
    sum += arr[i+1];
    sum += arr[i+2];
    sum += arr[i+3];
}
sum += arr[1024];
sum += arr[1025];
sum += arr[1026];
```

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

<table>
<thead>
<tr>
<th>Code Size</th>
<th>Increase</th>
<th>No Change</th>
<th>Decrease</th>
</tr>
</thead>
<tbody>
<tr>
<td>More lines of code must be written in the loop body, and after the loop if there are an uneven number of iterations for the unrolling factor.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Instructions Executed</th>
<th>Increase</th>
<th>No Change</th>
<th>Decrease</th>
</tr>
</thead>
<tbody>
<tr>
<td>There will be fewer branching instructions executed, since multiple loop bodies are combined.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Readability</th>
<th>Increase</th>
<th>No Change</th>
<th>Decrease</th>
</tr>
</thead>
<tbody>
<tr>
<td>The code can become harder to follow with multiple loop bodies in one loop and fixup code for uneven iterations.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction cache misses</th>
<th>Increase</th>
<th>No Change</th>
<th>Decrease</th>
</tr>
</thead>
<tbody>
<tr>
<td>A larger code size, and particularly a larger loop body, can lead to more instruction cache misses.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data cache misses</th>
<th>Increase</th>
<th>No Change</th>
<th>Decrease</th>
</tr>
</thead>
<tbody>
<tr>
<td>The same data is accessed in the same order.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
For some loop bodies, and with high loop unrolling factors, the number of variables inside the loop increases beyond what registers can handle. Thus these must be spilled to memory.

Consider the following C code:

```c
// 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.

```c
// 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+=4) {
    sum += A[i] * B[i];
    sum += A[i+1] * B[i+1];
    sum += A[i+2] * B[i+2];
    sum += A[i+3] * B[i+3];
}
```

(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.

```c
// 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+=4) {
    sum += dot_vec4(&A[i], &B[i]);
}
```
Problem 5: Multithreading ★

Consider the following C code:

```
// 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.

```
for (int i = t*(N/8); i < (t+1)*(N/8); ++i) {
    for (int j = 0; j < N; ++j) {
        for (int k = 0; k < N; ++k) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Thread t handles the indices of i from t*(N/8) to (t+1)*(N/8).

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

We can split this loop across a maximum of 64 threads, since the outer iteration goes up to 64. If we split the inner loops across threads, we could use even more threads and still benefit from parallelism.

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

The same locations in array B will be accessed by different threads. This does not cause any problems these locations are only read, not written, during the lifetime of these threads. The values in these locations will not change so there are no race conditions/synchronization issues between threads.
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.

```c
// N = 64
for (int k = t*(N/8); k < (t+1)*(N/8); ++k) {
    for (int j = 0; j < N; ++j) {
        for (int i = 0; i < N; ++i) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

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

These threads share array $C$. This can cause problems when two threads which are both modifying the same $C[i][j]$ interleave. The loop body is really two separate statements:

```c
temp = C[i][j] + A[i][k] * B[k][j];
C[i][j] = temp;
```

Two threads running this program could have the following interleaving:

```c
temp = C[i][j] + A[i][k] * B[k][j];  (Thread 1)
temp = C[i][j] + A[i][k] * B[k][j];  (Thread 2)
C[i][j] = temp;  (Thread 1)
C[i][j] = temp;  (Thread 2)
```

$C[i][j]$ is read by both threads, then the multiplication happens, then both threads write to $C[i][j]$. $C[i][j]$ ends up getting the value from Thread 2, which does not include the update that Thread 1 made.

More details to come in the synchronization lecture!