Directions:

- This homework includes one Gradiance quiz, two written question, and one programming question. The written part and the programming part should be submitted separately via Gradescope. No late days can be applied to the Gradiance quiz.

- You may use up to two of your remaining late days for this assignment, for a late deadline of March 10, 2023 at 11:59pm.

- You can complete the programming part in pairs, but the written part should be done individually. If you look up any material in the textbook or online, you should cite it appropriately.
1 Written Questions: Pipelined Parallelism

Problem 1. Consider the following program.

```java
for (int i = 1; i <= n; i++) {
    A[i, 0] = A[i-1, 0] * 2;
    for (int j = 1; j < n; j++) {
    }
}
```

1. Convert the program into a 2-deep fully permutable loop nest and write it down. Then draw the iteration space for the converted program. Include arrows to denote data dependencies that exist across iterations. (You don’t need to draw out all the arrows, just enough to get a clear picture of the dependencies.)

```java
for (int i = 1; i <= n; i++) {
    for (int j = 0; j < n; j++) {
        if (j == 0)
            A[i, 0] = A[i-1, 0] * 2;
        else
    }
}
```
2. Apply pipelined parallelism with blocking to your program. Assume your machine has 8 single-core processors, you choose blocking size $B = 16$ based on the cache properties of your machine. Also assume $n = 4096$. Show the generated code for each processor (use $p$ as the processor ID, assume $p$ ranges from 0 to 7) that is optimized for cache locality. You may use sync variables as presented in slide 8 of lecture 11. You may describe in text how sync variables are initialized without writing out the code. Your solution should have reasonable synchronization overhead.

```
// t[0]..t[7] are sync variables initialized to 0.
// begin of code executed on the processor of ID = p
if (p==0) p_synch = 7 else p_synch = p-1;
for (int ii = 1+16*p; ii <= 4096; ii += 128) { // parallelizable loop
    for (int jj = 0; jj < 4096; jj += 16) {
        if (ii == 1 || wait(t[p_synch]) >= t[p]+16) { // +16 is necessary
            for (int i = ii; i < ii+16; i++) {
                for (int j = jj; j < jj+16; j++) {
                    if (j == 0)
                        A[i, 0] = A[i-1, 0] * 2;
                    else
                }
            }
        }
    }
    t[p]+=16;
}
// end of code executed on the processor of ID = p
```

Since we only have 8 processors, parallelizing the outer loop is enough to maximize the parallelism. Parallelizing both loops will introduce more synchronization overhead.
Problem 2. Consider the following program.

```java
for (int i = 2; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
    }
}
```

1. Draw the iteration space for the program. Use arrows to mark data-dependencies between iterations. You don’t need to draw out all the arrows, just enough to get a clear picture of the dependencies. Explain why the loop nest is not fully permutable as is.

![Iteration Space Diagram]

The loop nest is not fully permutable because of the backward-pointing arrows with vector \((-2, 1)\). For a concrete example, consider iterations \((i = 2, j = 3)\) and \((i = 3, j = 1)\). The former writes to \(A[2, 3]\) while the latter reads from \(A[i - 1, j + 2] = A[2, 3]\), so there is a true dependency from \((i = 2, j = 3)\) to \((i = 3, j = 1)\). However, if we permute the two loops, then the iteration \((i = 3, j = 1)\) will run before \((i = 2, j = 3)\), violating the constraint.
2. Transform the program into a fully permutable loop nest and remove any “if guard” if exists. You do NOT need to perform blocking or use sync variables. Show the generated loop code.

Let \( t = xi + yj + z \) be the time partitioning formula for iteration \((i, j)\). From the previous part, we know that it needs to satisfy the following dependencies:

Whenever \( i = i' - 1, j = j' + 2, xi + yj + z \leq xi' + yj' + z \)
Whenever \( i = i' - 2, j = j' - 1, xi + yj + z \leq xi' + yj' + z \)

Solving the constraints above gives:

\[
\begin{align*}
    x - 2y & \geq 0, \\
    2x + y & \geq 0,
\end{align*}
\]

meaning that any time partitions with coefficients that satisfy these constraints can work.

For simplicity, we choose \((x, y, z) = (1, 0, 0)\) and \((2, 1, 0)\), meaning we have the time mappings

\[
    t = \begin{bmatrix} 1 & 0 \\ j & i \end{bmatrix}, \quad t = \begin{bmatrix} 2 & 1 \\ j & i \end{bmatrix}.
\]

These two vectors form a complete basis in the iteration space. So we can use the 2D time mapping

\[
    \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}.
\]

Calling \( t_1 \) the new \( i' \) and \( t_2 \) the new \( j' \), we can then generate code as follows:

\[
\begin{align*}
    &\text{for (int } i' = 2; i' \leq m; i'++) \{ \\
    &\quad \text{for (int } j' = 2i' + 1; j' \leq 2*i' + n; j'++) \{ \\
    &\quad\quad \text{A}[i, j'-2i'] = A[i'-1, j'-2i'+2] + A[i'-2, j'-2i'-1]; \\
    &\quad \}
    &\}
\end{align*}
\]

This loop nest is now fully permutable.

For a more graphical approach, you can think of the matrix as being sheared horizontally by the matrix \( \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} \). (Notice that the matrix is written in order \( i, j \), while the \( x \)-axis on the diagram is variable \( j \), not \( i \).) Thus, the previously top-right-pointing vectors are now much more slanted to the right with vector \((\Delta i', \Delta j') = (2, 5)\), while the previous backwards-pointing vectors are now pointing straight up \((\Delta i', \Delta j') = (1, 0)\).

Note that this answer is not unique. Another solution that works is \( i' = i, j' = 3i + j \). That transformation would skew the program even further.