CS 243 Homework 5 Solutions

Winter 2024

Due: February 28, 2024 at 11:59pm

Directions:

- This homework includes Gradiance quizzes and some written questions. The written part should be submitted 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.
- This assignment is an individual assignment. You are allowed to discuss the homework with others, but you must write the solution individually. If you look up any material in the textbook or online, you should cite it appropriately.

Problem 1. Short answer questions:

1. One way to schedule a do-all loop is to fully unroll it and utilize instruction scheduling. The other way is to use software pipelining. For both the techniques, when would you prefer to use one technique over the other and why?

   Fully unrolling a loop is more beneficial for loops that have a short number of iterations. For loops which may have a large number of iterations, fully unrolling them would lead to a large code size, which will potentially cause multiple instruction cache misses. In those cases, software pipelining is better.

2. Consider the following software pipelining algorithm for do-all loops:

```
1. Find lower bound of initiation interval
2. For T = T0, T0+1, ... until a schedule is found
3.     For each node n in topological order
4.         s0 = earliest n can be scheduled
5.         For each s = s0, s0+1,..., s0+T-1
6.             if NodeScheduled(n, s) break;
7.         if (n cannot be scheduled) break;
8.
9. NodeScheduled(n, s):
10.  Check resources of n at s in modulo resource reservation table
```

   (a) What is the maximum `T` in line 2 that we may need to iterate over?

   Number of cycles between the first instruction of two successive iterations. (This is equivalent to length of scheduling one iteration including any `nop`s in the end)

   (b) If we replace line 2 with a binary search over `T0 to T_max` (from your answer above), does the algorithm produce the optimal schedule (regardless of the time it takes to find the optimal schedule)?

   No, it will not work. Acceptable solutions should be around one or some of these reasonings:

   i. We may end up settling at some higher initiation interval because it is easier to accommodate a software pipeline at a higher initiation interval.

   ii. It is possible that we are not able to find a schedule for some initiation interval of `T`, in which case we move to the right half, however they may be a lower `T'` for which we can get a schedule.

   (c) In line 5, if we replace the upper bound of the loop with `s0 + T`, can it potentially find an optimal schedule for the initiation interval in cases where it was not able to before?

   No, for an initiation interval of `T`, `s0 + T` simply wraps around back to `s0`, so we do not find any new optimal schedule.

(d) If we replace line 5 with a binary search similar to the one discussed in part (b) above, does the new algorithm produce the optimal schedule (regardless of the time it takes to find the optimal schedule)?

No. By using a binary search and scheduling it, we will not get an optimal schedule. Reasoning is similar to part (b). We want to schedule it as early as possible and binary search will not help in that case. Also, we do not have a well-defined decision on which half to select in binary search: if we cannot schedule at `s0 + T/2`, do we move left or right? Any one or more relevant reasoning among all of these can be accepted for full credit.

3. After successfully software pipelining a loop with an initiation interval T, registers `R1` and `R2` in the original program have live ranges of $3 \times T$ and $4 \times T$ cycles, respectively. Suppose you have 10 registers to devote to `R1` and `R2`, how many times would you unroll the steady state of the loop?

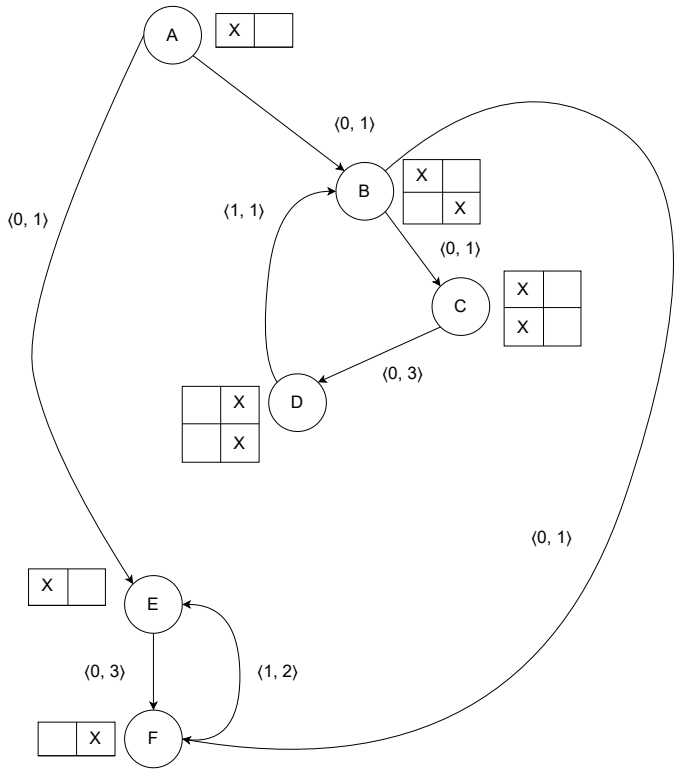
   4 unrolls. This follows from the formula discussed in class $\max_r\left(\frac{\text{lifetime}_r}{T}\right)$.

4. True or False. A single-issue machine will not benefit from global instruction scheduling because it cannot issue more than one instruction per cycle.

   False. As an example, the program can benefit from moving operations with a latency of more than one clock cycle earlier.

Problem 2. Software Pipelining

Consider the following dependence graph for a single iteration of a loop, with resource constraints:



1. What is the bound on the initiation interval T according to the precedence and resource constraints for this program?

   Both the B–C–D–B and E–F–E cycles impose a bound of 5 due to data dependence, as computed by dividing cycle length by iteration difference. The left resource imposes a bound of 5 and the right resource 4. So overall, $T \geq 5$.

2. What is the minimum initiation interval? Show a modulo reservation table for an optimal software pipelined schedule. Also show the code and schedule for an iteration in the source loop.

   The minimum initiation interval is 5. Modulo reservation table:

A	
E	D
B	D
C	B
C	F

   Single iteration: A, E, B, C, F, nop, D

3. Can the scheduling algorithm described in class produce the optimal schedule for this loop? If not, show the modulo reservation table and code schedule generated by the algorithm.

   The scheduling algorithm described in class will backtrack only within strongly connected components, and schedule greedily across them. Therefore it will not produce an optimal schedule: it will pack the $\{B, C, D\}$ SCC immediately after A (in the first available spot), eliminating the ability interleave the $\{E, F\}$ SCC in-between.

A	D
B	D
C	B
C	
E	

   Here, we are stuck, since we cannot schedule F 3 clocks after E.

   However, the algorithm will be able to schedule instructions with an initiation interval of 6. Modulo reservation table:

A	D
B	F
C	B
C	
E	
	D

   Single iteration: A, B, C, nop, E, D, nop, F

Problem 3. Software Pipelining with Register Allocation

Consider the following do-all loop program:

```
for (i = 0; i < 1000; i++)
    C[i] = (A[i] - b) * A[i];
```

One iteration of the loop can be written in assembly as:

```
1.  LD   R5, 0(R1++)      // R1 = &A[i]; R5 = *R1; R1++
2.  SUB  R6, R5, R2       // R2 = b; R6 = R5 - R2
3.
4.  MUL  R6, R6, R5       // R6 = R6 * R5
5.
6.  ST   R6, 0(R3++)      // R3 = &C[i]; *R3 = R6; R3++
```

Optimize this program using software pipelining for a machine with the following specifications:

- The processor can issue at most one `LD` instruction, one `ST` instruction, *and* one arithmetic instruction within the same clock cycle.
- Each arithmetic operation has a two-cycle latency, but can be pipelined.
- The processor supports auto-incrementing addressing and hardware loop operations.

Answer the questions below using your optimized program:

1. What is the minimum initiation interval? 2

2. Write down the pipelined portion of the loop using actual registers. Annotate each instruction with the iteration it's associated with (e.g., i, $i+1$, $i+2$, etc.). Resolve any anti-dependencies across loop iterations by unrolling the loop as described in class.

```
ST  R6, 0(R3++) (i)      MUL R16, R16, R15 (i+1)    LD  R5, 0(R1++) (i+3)
                         SUB  R6, R5, R2   (i+3)
ST R16, 0(R3++) (i+1)    MUL R26, R26, R25 (i+2)    LD R15, 0(R1++) (i+4)
                         SUB R16, R15, R2  (i+4)
ST R26, 0(R3++) (i+2)    MUL  R6, R6, R5   (i+3)    LD R25, 0(R1++) (i+5)
                         SUB R26, R25, R2  (i+5)
```

Here's the full program for reference (not required for full credit):

```
        i               i+1               i+2               i+3               i+4               i+5               i+6
  LD   R5, 0(R1++)
  SUB  R6, R5, R2
                   LD  R15, 0(R1++)
                   SUB R16, R15, R2
  MUL  R6, R6, R5                    LD  R25, 0(R1++)
                                     SUB R26, R25, R2
> ST   R6, 0(R3++)   MUL R16, R16, R15                   LD  R5, 0(R1++)
>                                                       SUB R6, R5, R2
>                    ST  R16, 0(R3++)    MUL R26, R26, R25                 LD  R15, 0(R1++)
>                                                                          SUB R16, R15, R2
>                                        ST  R26, 0(R3++)    MUL R6, R6, R5                    LD  R25, 0(R3++)
>                                                                                              SUB R26, R25, R2
                                                            ST  R6, 0(R3++)   MUL R16, R16, R15                   LD  R5, 0(R3++)
                                                                                                                  SUB R6, R5, R2
```

3. How many more registers does software pipelining require compared to the unpipelined code? Think of a way we can reduce the number of registers used without decreasing the throughput. (You do not have to show the generated code.) *Hint*: check the live ranges for all the duplicated registers.

   We need 4 more registers (R15, R16, R25, R26). There are two ways to reduce registers:

   (a) Notice that R25 isn't strictly needed, since the live ranges of R5 and R25 do not overlap. We can remove the need for R25 by unrolling the loop 6 times rather than 3 with the following register assignment:

      i. R5 – R6
      ii. R15 – R16
      iii. R5 – R26
      iv. R15 – R6
      v. R5 – R16
      vi. R15 – R26

      In general, we can reduce register pressure by unrolling n times, where n is the least common multiple of $\lceil \text{lifetime}_r/T \rceil$ for all registers r. This is in contrast with the algorithm in lecture 8, which suggests unrolling $\max_r \lceil \text{lifetime}_r/T \rceil$ times.

   (b) Alternatively, we can change the program a bit to break up the live range of R6:

```
LD   R5, 0(R1++)          LD   R5, 0(R1++)
SUB  R6, R5, R2           SUB  R6, R5, R2

MUL  R6, R6, R5     →     MUL  R7, R6, R5

ST   R6, 0(R3++)    →     ST   R7, 0(R3++)
```

      Because of this, we now only need to unroll twice and create another copy each of R5 and R6. Compared to the original program, this also adds 3 registers (R7 and the two copies).

```
       i              i+1              i+2              i+3              i+4              i+5
  LD   R5, 0(R1++)
  SUB  R6, R5, R2
                    LD  R15, 0(R1++)
                    SUB R16, R15, R2
  MUL  R7, R6, R5                    LD  R5, 0(R1++)
                                     SUB R6, R5, R2
> ST   R7, 0(R3++)  MUL  R7, R16, R15                 LD  R15, 0(R1++)
>                                                     SUB R16, R15, R2
>                   ST   R7, 0(R3++)  MUL R7, R6, R5                   LD  R5, 0(R1++)
>                                                                      SUB R6, R5, R2
                                     ST  R6, 0(R3++)  MUL  R7, R16, R15                 LD  R15, 0(R3++)
                                                                                        SUB R16, R15, R2
                                                      ST   R7, 0(R3++)
```

Problem 4. Dependency Analysis and Parallelization

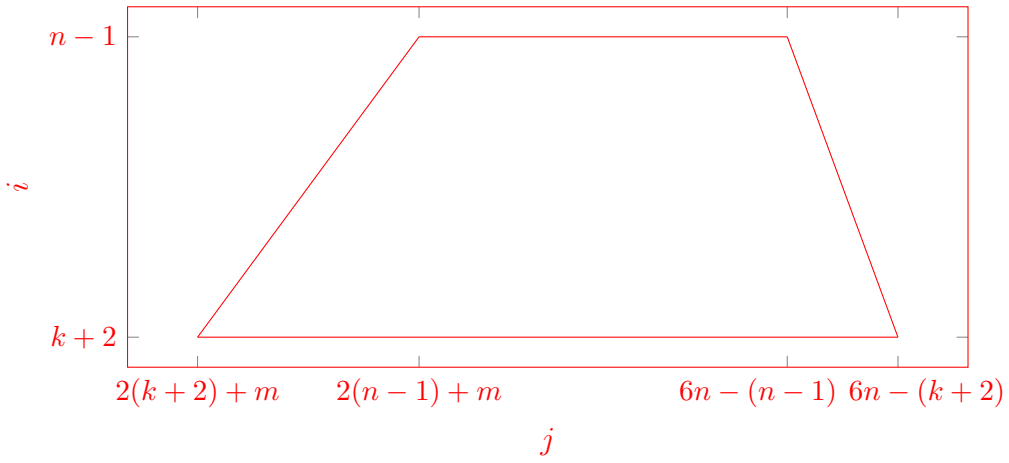
Consider the C-style program:

```
for (i = k+2; i < n; i++) {
    for (j = 2*i+m; j < 6*n-i; j++) {
        X[i,   4*j-2] = X[i, 4*j+1] + Y[i, j] -(1)
        Y[i+1,   j-3] = X[i, 2*j]   + Y[i, j] -(2)
    }
}
```

Assuming $0 \leq m \leq k \ll n$ and that X and Y are non-overlapping arrays, answer the following questions.

1. Draw the iteration space for this loop. *Hint*: The axes should be situated such that program execution first goes from left to right, and next from bottom to top.

   (Either axis labels should be present or line equations should be present for full credit. Also, slope for j >= 2i + m should look shallower than j < 6n + 1)



2. What are the data dependencies in this loop? Categorize each as a true dependency, output dependency, or an anti-dependency. Include all dependency, even if they can be ruled out using simple tests like the GCD test. *Hint*: a data dependency is something of the form "read/write $A[i]$, read/write $B[j]$."

   (a) True dependency: write $X[i, 4j-2]$, read $X[i, 4j+1]$

   (b) True dependency: write $X[i, 4j-2]$, read $X[i, 2j]$

   (c) True dependency: write $Y[i+1, j-3]$, read $Y[i, j]$

   (d) Output dependency: write $Y[i+1, j-3]$, write $Y[i+1, j-3]$

   (e) Output dependency: write $X[i, 4j-2]$, write $X[i, 4j-2]$

   (f) Anti dependency: read $X[i, 4j+1]$, write $X[i, 4j-2]$

   (g) Anti dependency: read $X[i, 2j]$, write $X[i, 4j-2]$

(h) Anti dependency: read $Y[i, j]$, write $Y[i + 1, j - 3]$

Note: True/Anti dependences are opposite of each other. If a dependence is written something like: "True/Anti dependency: write $X[i, 4j - 2]$, read $X[i, 4j + 1]$", we have given them full credit.

3. Formulate the data dependence tests for the given loop nest.

Note: We have graded part 3 and 4 together. Overall, they should present the following for the dependencies listed in part 2 above:

- explicitly state that we consider two distinct iterations (i, j) and (i', j')
- eliminate output dependences based on the fact that $i = i'$ and $j = j'$
- develop loop bound constraints
- for each dependence, get equations for i, i' and j, j'.
- Prove no dependence using tests like GCD/out-of-loop-bounds or prove dependence using Fourier-Motzkin.
- Finally explicitly state the data dependent acceses

Solution: Inside the nested loop, we have two statements. Let those statements be marked as (1) and (2) respectively.

First, we can rule out output dependencies in this case. Consider two distinct iterations (i, j) and (i', j'). For an output dependency, they both should be writing to the same location.

- For an output dependency on writing to $X[i, 4j - 2]$ in statement 1, we would get $i = i'$ and $4j - 2 = 4j' - 2 \implies j = j'$. Hence, this output dependency does not exist.
- For an output dependency on writing to $Y[i + 1, j - 3]$ in statement 2, we would get $i + 1 = i' + 1 \implies i = i'$ and $j - 3 = j' - 3 \implies j = j'$. Hence, this output dependency does not exist.

Now, we can talk about true and anti-dependencies. They would occur when a write index and a read index is the same for two different iterations (i, j) and (i', j'). We would simply solve to get a relationship between (i, j) and (i', j'), and based upon that, decide whether it is a true dependency or anti-dependency or any of them or none of them. We would essentially use Integer Linear Programming and GCD tests for finding the relationship between the iterations.

Loop bound constraints: The eight trivial inequalities that we have for iterations (i, j) and (i', j') coming from the loop constraints are:

- $i - k - 2 \geq 0$ - (i)
- $-i + n > 0 \implies -i + n - 1 \geq 0$ - (ii)
- $j - 2i - m \geq 0$ - (iii)

- $-j - i + 6n > 0 \implies -j - i + 6n - 1 \geq 0$ - (iv)
- $i' - k - 2 \geq 0$ - (v)
- $-i' + n > 0 \implies -i' + n - 1 \geq 0$ - (vi)
- $j' - 2i' - m \geq 0$ - (vii)
- $-j' - i' + 6n > 0 \implies -j' - i' + 6n - 1 \geq 0$ - (viii)

Dependency formulation: In addition to the loop bound constraints, we will split the various dependencies into the following cases:

(a) *Case 1: Write $X[i, 4j-2]$ in statement (1) in iteration (i, j), Read $X[i', 4j'+1]$ in statement (1) in iteration (i', j'):*
Here, for this case to be a dependency, the read-write should happen at the same location. Hence, we get the following extra equations:

$$i = i'$$
$$4j - 2 = 4j' + 1 \implies 4(j - j') = 3$$

The above equation is unsolvable for integer values of j and j'. This can also be explained by GCD test. Hence, no dependency exists from this case.

(b) *Case 2: Write $X[i, 4j-2]$ in statement (1) in iteration (i, j), read from $X[i, 2j]$ in statement(2) in iteration (i', j'):*
Here, for this case to be a dependency, the read-write should happen at the same location. Hence, we get the following extra equations:

$$i = i'$$
$$4j - 2 = 2j' \implies j' = 2j - 1$$

Substitute i' and j' in loop bounds (v) - (viii):

$k+2 \leq i'$ and $i' \leq n-1 \implies k+2 \leq n-1$ (Fourier-Motzkin): True since $k \ll n$

$$2i' + m \leq j' \text{ and } j' \leq 6n - i' - 1$$
$$2i + m \leq 2j - 1 \text{ and } 2j - 1 \leq 6n - i - 1$$
$$2i + m + 1 \leq 2j \text{ and } 2j \leq 6n - i$$
$$2k + 4 + m + 1 \leq 2j \text{ and } 2j \leq 6n - k - 2 \text{ (from inequality (i))}$$
$$2k + m + 5 \leq 6n - k - 2 \text{ (Fourier-Motzkin)}$$
$$3k + m + 7 \leq 6n : \text{ True since } m \leq k \ll n$$

Therefore, the inequalities are satisfied and there exist such pairs with dependencies.

$$\text{Also, } j \geq 2i + m \geq 2k + 4 + m > 1 \text{ from inequalities (i) and (ii)}$$

$$\implies j' = 2j - 1 = j + j - 1 > j$$

Therefore, we can see that iteration (i, j) comes before iteration (i', j'). Hence, we can say that there exist dependencies between iterations of indices (i, j) [write X] and $(i, 2j - 1)$ [read X] such that all of these iterations are within loop bounds. Since the write (i, j) is visited before the read $(i, 2j - 1)$, these are true dependencies.

(c) Case 3: Write to $Y[i + 1, j - 3]$ in statement (2) in iteration (i, j), read from $Y[i', j']$ in statement (1) / statement (2) in iteration (i', j').
Here, for this case to be a dependency, the read-write should happen at the same location. Hence, we get the following extra equations:

$$i + 1 = i'$$
$$j - 3 = j'$$

Substitute i' in loop bounds (v) - (vi):

$$k + 2 \leq i' \text{ and } i' \leq n - 1$$
$$\implies k + 2 \leq i + 1 \text{ and } i + 1 \leq n - 1$$
$$\implies k + 1 \leq i \text{ and } i \leq n - 2$$
$$\implies k + 1 \leq n - 2 \text{ (Fourier-Motzkin): True since } k \ll n$$

Substitute j' in loop bounds (vii) - (viii):

$$2i' + m \leq j' \text{ and } j' \leq 6n - i' - 1$$
$$2i + 2 + m \leq j - 3 \text{ and } j - 3 \leq 6n - i - 1 - 1$$
$$2i + 5 + m \leq j \text{ and } j \leq 6n - i + 1$$
$$2k + 4 + 5 + m \leq j \text{ and } j \leq 6n - k - 1$$
$$2k + 9 + m \leq j \text{ and } j \leq 6n - k - 1$$
$$2k + 9 + m \leq 6n - k - 1 \text{ (Fourier-Motzkin)}$$
$$3k + m + 10 \leq 6n : \text{ True since } m \leq k \ll n$$

Therefore, the inequalities are satisfied and there exist such pairs with dependencies.

$$\text{Also, } i' = i + 1 \implies i' > i$$

Therefore, we can see that iteration (i, j) comes before iteration (i', j'). Hence, we can say that there exist dependencies between iterations of indices (i, j) [write Y] and $(i + 1, j - 3)$ [read Y] such that all of these iterations are within loop bounds. Since the write (i, j) is visited before the read $(i + 1, j - 3)$, these are true dependencies.

4. Which pairs of accesses are data dependent?

   (a) Iterations (i, j) and $(i, 2j-1)$ have the true dependency: write $X[i, 4j-2]$, read $X[i', 2j']$

   (b) Iterations (i, j) and $(i+1, j-3)$ have a true dependency: write $Y[i+1, j-3]$, read $Y[i', j']$

Problem 5. Affine Transforms

Apply affine transform to find the largest degree of outermost loop parallelism. Show the transformed code, and mark the loops that are parallelizable. Assume $N \geq 100$.

```
for (i = 1; i <= N; i++) {
    for (j = 1; j <= N; j++) {
        A[i,j] = A[i-1,j] + X[i,j];
    }
}
for (i = 4; i <= N; i++) {
    for (j = i+1; j <= N; j++) {
        for (k = 1; k <= N; k++) {
            B[i,k] = B[i-1,k] + Y[j];
        }
    }
}
for (i = 1; i <= N; i++) {
    C[i] = A[3,i];
}
```

Solution:

```
for (pj = 1; pj <= N; pj++) { // parallelizable
    for (pi = 1; pi <= N; pi++) {
        A[pi,pj] = A[pi-1,pj] + X[pi,pj];
        if (pi >= 4) {
            for (j = pi+1; j <= N; j++) {
                B[pi,pj] = B[pi-1,pj] + Y[j];
            }
        }
        if (pi == 3) {
            C[pj] = A[pi,pj];
        }
    }
}
```

This uses the following index mappings:

$$\text{for } A \text{ loop}: \quad \begin{bmatrix} p_i \\ p_j \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_A \\ j_A \end{bmatrix},$$

$$\text{for } B \text{ loop}: \quad \begin{bmatrix} p_i \\ p_j \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_B \\ k_B \end{bmatrix},$$

$$\text{for } C \text{ loop}: \quad \begin{bmatrix} p_i \\ p_j \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} i_C + \begin{bmatrix} 3 \\ 0 \end{bmatrix}.$$

There is only one degree of outermost loop parallelism.

Other acceptable solutions:

1. It's acceptable for the B loop to not be fused together with A and C, since there is no dependency between the second loop and the other two loops.

```
for(p = 1; p <= N; p++){
    for (i = 1; i <= N; i++) {
        A[i,p] = A[i-1,p] + X[i,p];
        if(i == 3){
            C[p] = A[3, p];
        }
    }
    for (i = 4; i <= N; i++) {
        for (j = i+1; j <= N; j++) {
            B[i,p] = B[i-1,p] + Y[j];
        }
    }
}
```

2. Noticing that B and A/C loops are completely independent, another way to do the transform is to execute B completely in parallel with A/C:

```
for (pk = 1; pk <= 2; pk++) {       // parallelizable
    for (pj = 1; pj <= N; pj++) {   // parallelizable
        for (pi = 1; pi <= N; pi++) {
            if (pk == 1) {
                A[pi,pj] = A[pi-1,pj] + X[pi,pj];
            }
            if (pk == 2 && pi >= 4) {
                for (j = pi+1; j <= N; j++) {
                    B[pi,pj] = B[pi-1,pj] + Y[j];
                }
            }
            if (pk == 1 && pi == 3) {
                C[pj] = A[pi,pj];
            }
        }
    }
}
```

This uses the following processor mapping:

$$\text{for } A \text{ loop}: \quad \begin{bmatrix} p_i \\ p_j \\ p_k \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_A \\ j_A \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix},$$

$$\text{for } B \text{ loop}: \quad \begin{bmatrix} p_i \\ p_j \\ p_k \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_B \\ k_B \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix},$$

$$\text{for } C \text{ loop}: \quad \begin{bmatrix} p_i \\ p_j \\ p_k \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} i_C + \begin{bmatrix} 3 \\ 0 \\ 1 \end{bmatrix}.$$

Caveats:

1. Notice that the `j` loop in the `B` loop nest is not actually parallelizable, since two independent iterations of `j` can write to the same location in memory.

   (You can reduce the `j` loop to the last iteration of `j` since that's the only one that sticks, but that's not something that can be automatically done using an affine transform.)