CS243 Homework 6

Winter 2021

Due: March 10th, 2021 at 4:00 pm

Directions:

- Submit via Gradescope.
- You may use up to two of your remaining late days for this assignment, for a late deadline of March 12th, 2021 at 4:00 pm.
- There is **no** Gradiance assignment for this homework.
- This 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. Pipelined Parallelism

Part 1. Consider the following program.

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

1. Convert the program into a 2-deep fully permutable loop nest. Show the converted program, and briefly justify why it preserves all the data dependencies in the original program.

```
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;
     if (j >= 1)
        A[i, j] = A[i-1, j] + A[i, j-1];
  }
}
```

If we look at a single instruction, all its iterations are still executed in the same order, so intra-instruction dependencies are preserved. In addition there is a inter-instruction WR dependency where A[i, j - 1] of instruction 2 reads from A[i, 0] of instruction 1 when j = 1. This relation is preserved by the if guard: A[i, 0] will always be written before A[i, j - 1] is read for all j $\geq$ 1.

2. Apply pipelined parallelism with blocking to your program. Use blocking size B = 16. You may assume n can be divided by B. Show the generated code for each processor that is optimized for cache locality. (You may use sync variables as presented in slide 4 of lecture 11. You may describe in text how sync variables are initialized without writing out the code.)

```
// t[0]..t[n/B - 1] are sync variables initialized to 0.

for (int i = 0; i < n/B; i ++) { // parallelizable loop
  // begin of code executed on processor i.
  for (int j = 0; j < n/B; j ++) {
     if (i == 0 or wait(t[i-1]) >= j+1) ) {
         for (int ii = i*B + 1; ii < (i+1)*B + 1; ii++) {
            for (int jj = j*B; jj < (j+1)*B; jj++) {
                if (jj == 0)
```

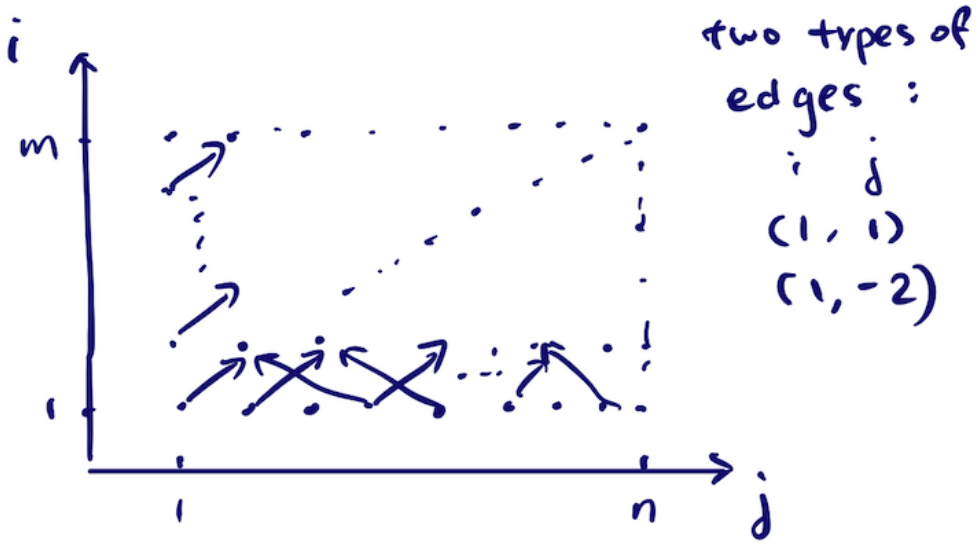
```
                    A[ii, 0] = A[ii - 1, 0] * 2;
                if (jj >= 1)
                    A[ii, jj] = A[ii - 1, jj] + A[ii, jj-1];
            }
        }
      }
      t[i]++;
    }
    // end of code executed on processor i
  }
```

Remark: We are only parallelizing the outermost loop here for simplicity's sake. You can also parallelize both i and j loops. These two strategies both make full use of cache locality offered by ii and jj loops, and both are given full credit.

Part 2. Consider the following program.

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

3. 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.



4. Transform the program into a fully permutable loop nest and show the generated

code. You should apply optimizations to remove unnecessary loops or conditions in the if statements, but you do NOT need to perform blocking or use sync variables.

Let $t = c_2 * i + c_1 * j + c_0$ be the time partitioning for iteration (i, j). It needs to satisfy the following dependencies:

Whenever $i = i' - 1, j = j' + 2$, $c_2 * i + c_1 * j + c_0 \leq c_2 * i' + c_1 * j' + c_0$

Whenever $i = i' - 1, j = j' - 1$, $c_2 * i + c_1 * j + c_0 \leq c_2 * i' + c_1 * j' + c_0$

Solving for the constraint above gives:

$c_2 - 2c_1 \geq 0, c_2 + c_1 \geq 0$.

We can find two time mapping solutions that form a basis in the iteration space of i and j:

$t_1 = i, t_2 = 2i + j$. (Solution is not unique)

Use i' and j' to denote the loop variables corresponding to t_1 and t_2.

Then $i' = i, j' = 2i + j, j = j' - 2i'$.

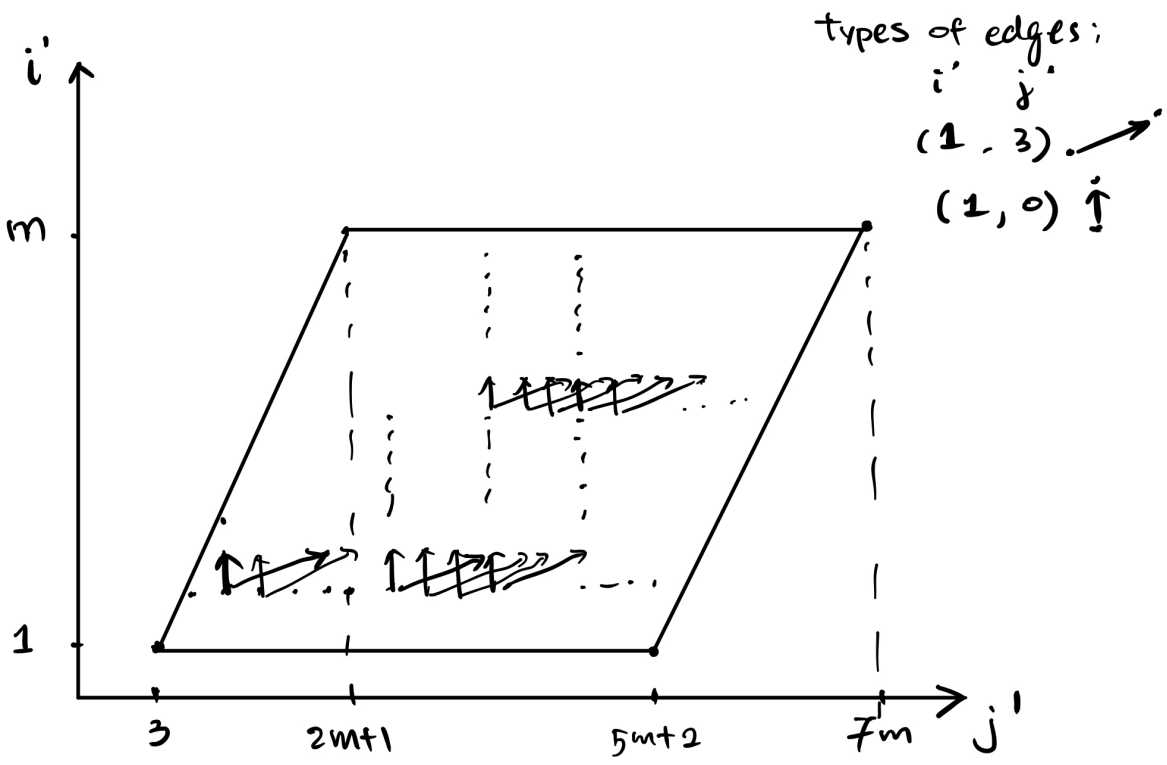
```
for (int i' = 1; i' <= m; i'++) {
  for (int j' = 3; j' <= 2m + n; j'++) {
      if (1 <= j' - 2 * i' && j' - 2 * i' <= n) {
          A[i',j'-2*i'] = A[i'-1, j'-2*i' + 2] + A[i'-1, j'-2*i' - 1]
      }
  }
}
```

After optimization:

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

5. Draw the iteration space for the transformed program when n = 5m. Use arrows to mark data-dependencies between iterations. Again, you don't need to draw out all the arrows, just enough to get a clear picture of the dependencies.

   (The iteration space and arrows may vary depending on solution to 4.)


types of edges:
i' j'
(1, 3)
(1, 0)
i'
m
1
3
2m+1
5m+2
7m
j'

Problem 2. Pointer Analysis

Perform pointer analysis on the following Java snippet, and answer the following questions. Each allocation site in the program is labeled with the object name (h1-h4).

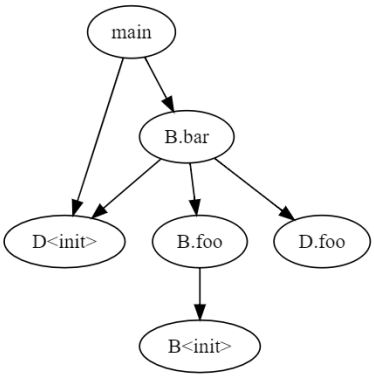
```
public class B {
    public B foo(B x) {
        return new B(); // h1
    }

    public static B bar(B c, B d) {
        c = c.foo(c);
        d = d.foo(d);
        if (c != d) {
            c = new D(); // h2
        }
        return c;
    }
}

public class D extends B {
    public B foo(B y) {
        return y;
    }
}

public class Main {
    public static void main(String[] args) {
        B a = new D(); // h3
        B b = new D(); // h4
        a = B.bar(a, b);
    }
}
```

1. If you have a perfect pointer analysis, what is the set of heap objects that `a` can refer to during the execution of this program?

   {h3, h2}.

   h3 from the first allocation to a, h2 from "c = new D()" in B.bar.

2. What is the call graph that can be determined a priori, without pointer analysis? You need to assume that a virtual method call can invoke all virtual methods defined.



3. What is the set of heap objects that `a` can refer to after performing a context-insensitive, flow-insensitive pointer analysis using your worst-case call graph from the previous part?

   {h1, h2, h3, h4}.

   pts(a, h3) pts(b, h4) pts(c, h3) pts(d, h4)

   pts(x, h3) pts(c, h1) pts(x, h1) pts(y, h1) pts(y, h3) pts(x, h4) pts(d, h1) pts(y, h4) pts(c, h4) pts(d, h3) [h1 comes from the worst-case call graph; h4 comes from context-insensitivity.]

   pts(c, h2) pts(x, h2) pts(y, h2) pts(d, h2)

   pts(a, h2) pts(a, h1) pts(a, h4)

4. What is the set of heap objects that `a` can refer to after performing a context-insensitive, flow-insensitive pointer analysis, assuming that the call graph is now calculated on-the-fly? By "on-the-fly", we mean that you should calculate which methods can be called (conservatively), based on the pts-to-relations that you generate when doing your pointer analysis.

   Hint: You can calculate which methods can be called (conservatively), using the pts-to-relations on the calling variable, where pts(x, h) indicates that a variable x can point to a heap object h. Note that if x can point to multiple heap objects of different types, to be safe, we need to assume that any virtual method call based on x can refer to methods from any of the different heap objects that x could point to.

   {h2, h3, h4}.

   pts(a, h3) pts(b, h4) pts(c, h3) pts(d, h4)

   pts(y, h3) pts(c, h3) pts(y, h4) pts(d, h4) pts(c, h4) pts(d, h3) [h4 still comes from context-insensitivity, but we can get rid of h1, since from main, we never allocate anything other than D's.]

   pts(c, h2) pts(y, h2) pts(x, h2) pts(d, h2)

pts(a, h2) pts(a, h4)

5. What is the set of heap objects that `a` can refer to after performing a context-sensitive, flow-insensitive pointer analysis using the worst-case call graph you computed previously?

   {h1, h2, h3}.

   pts(a, h3) pts(b, h4) pts(c, h3) pts(d, h4)

   pts(x_1, h3) pts(c, h1) pts(x_1, h1) pts(y_1, h1) pts(y_1, h3) pts(x_2, h4) pts(d, h1) pts(x_2, h1) pts(y_2, h1) pts(y_2, h4) [We still have h1 because of our worst-case call graph.]

   pts(c, h2) pts(x_1, h2) pts(y_1, h2)

   pts(a, h1) pts(a, h2)

6. What is the set of heap objects that `a` can refer to after performing a context-sensitive, flow-insensitive pointer analysis, assuming that the call graph is now calculated on-the-fly?

   {h2, h3}.

   pts(a, h3) pts(a, h4) pts(c, h3) pts(d, h4)

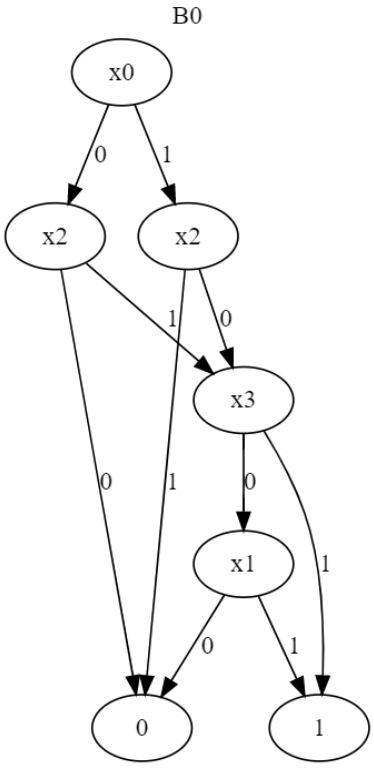
   pts(y_1, h3) pts(c, h3) pts(y_2, h4) pts(d, h4)

   pts(c, h2) pts(y_1, h2)

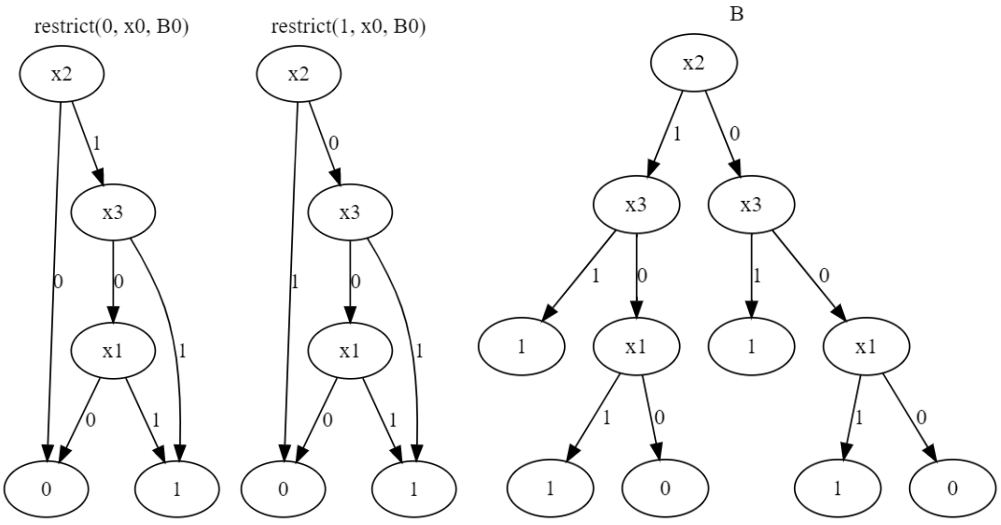
   pts(a, h2)

Problem 3. Binary Decision Diagram

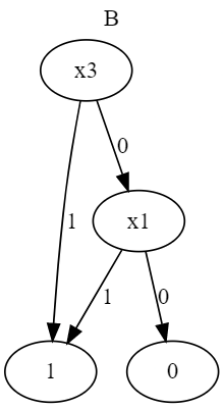
1. Draw the BDD for $B_0 =$ `(x0 XOR x2)` $\wedge$ `(x1` $\vee$ `x3)` with the following variable order: x0 $\geq$ x2 $\geq$ x3 $\geq$ x1.



2. Draw the BDD for $B =$ `exists(x0, (x0 XOR x2)` $\wedge$ `(x1` $\vee$ `x3))` using the diagram for B_0 (you do not need to simplify the BDD in this part).



3. Reduce the number of nodes in your BDD to create a compact representation.


B
x3
0
1
x1
1
0
1
0