# CSE 504
# Assignment 3

Sanaz Sheikhi 111733554

## Overview

In this report I try to explain the logic behind each solution . On the other hand as the code is huge (700 lines), I try to point out to some key functions and their implementation.
Tip: My implementation supports multi level loops, that is more than 2 loops say 3 or 4 or more nested loops. But in this report for convenience I explain everything with 2 level nested loops. But in my test cases in question 4 I test 3 and 4 level nested loops.

## Question 1:
## perfectly nested loops

### Logic:

For perfectly nested loop we Consider consecutive loop pairs. We call two consecutive loops perfectly nested if there is no code (command) between the headers of two loop. After making sure the loops are nested (from previous assignment), the mechanism for perfect loop detection works as follows.
Each loop is consist of three main block, first block is for initializing loop index, the second one is body of the loop with the instruction inside it and the third one calculates the loop condition and gets back to the beginning of it.
When we have two perfectly nested loops after the first block of the outer loop there is no block for body of it and with a intermediary block to load the index of the inner loop it jumps to the first block of the inner loop. Also, after the body block of the inner loop the blocks for calculating the indexes of the loops show up and no other block comes in between.
In two conditions the two loops are not nested:
First, there is command after the header of the outer loop and before the body inner loop . So, in IR code, after the initialization block of the outer loop a block containing the command shows up and then the initialization block of inner loop follows it .
Second, a command comes after the body of the inner loop . So, in the IR code, after the condition calculation block of the inner loop we see a block containing instruction related to this command .

The following figures show the IR structure of the perfectly nested and not perfectly nested loops.

Perfectly Nested

```
For( int i=0; i<100;i++){
   For( int j=0;j<100;j++){
                              B++;}}
```

IR

Block1 : Handle index i. Branch to Block 2.
Block2: Store th index i. Branch to Block 3.
Block3: Handle index j. Branch to Block 4.
Block4: Handle body of inner loop (B++). Branch to Block 5.
Block5: Calculate condition of inner loop.Branch to either Block 3 or Block 6.
Block 6: Branch to Block 7
Block 7: Calculate condition of outer loop. Branch to Block 1.

Fig.1

Not Perfectly Nested

```
For( int i=0; i<100;i++){
          A++;
   For( int j=0;j<100;j++){
                              B++;}}
```

IR

Block1 : Handle index i. Branch to Block 2.
Block2: Handle A++. Branch to Block3
Block3: Handle index j. Branch to Block 4.
Block4: Handle body of inner loop (B++). Branch to Block 5.
Block5: Calculate condition of inner loop.Branch to either Block 3 or Block 6.
Block 6: Branch to Block 7
Block 7: Calculate condition of outer loop. Branch to Block 1.

Fig.2

Not Perfectly Nested

```
For( int i=0; i<100;i++){
   For( int j=0;j<100;j++){
                                   B++;}
           C++;
}
```

IR
Block1 : Handle index i. Branch to Block 2.
Block2: Handle index j. Branch to Block 3.
Block3: Handle body of inner loop (A++). Branch to Block 4.
Block4: Calculate condition of inner loop. Branch to either Block 2 or Block 5.
Block 5: Handle C++. Branch Block 6.
Block 6: Calculate condition of outer loop. Branch to Block 1.

Fig.3

Tip: Suppose we have the following nested loop:

```
for (int i=0; i< 300; i++)
      for (int j=0; j< 300; j++)
           for (int k=0; k< 300; k++)
```

Based on this logic (I,j) and (j,k) are consecutive pairs that are perfectly nested but (I,k) that are not consecutive are not perfectly nested.

Implementation:
Based on assignment 2 implementation I have a map of block numbers to some block indexes in addition to number of instructions in each block. Whenever two blocks are considered as nested, then in perfect_loop_detector(int outer, int inner) function I look at the block after the block of outer loop , if it has more than a store and br instructions (that are jump to block of inner loop) it means this block is implementation of a command. So, it is obvious there is a command between two loops. So, the two loops are not perfectly nested . Fig.2 shows this logic.
With the same logic I look at the number of instructions in the loop after block of condition of inner loop.If this block contains more than one br instructionit means it is implementation of a command.So, obviously there is a command after the inner loop. So, the two loops are not perfectly nested.Fig.3 shows this logic.

## Question 2
## Loops Independence

### Logic

To calculate dependence between two loops, we calculate the distance vector . It is distance between index of arrays based on define and use of the indexes of the loops .

X = USE(loop index) – Def(loop index)

if X > 0 ----> >
if X < 0 ----> <

```
 if X == 0 --->   =
```

Based on the result , we calculate dir(outermost index, innermost index) vector

```
for i
   for j
      for k
         A[i][j][k] = A[i-2][j+1][k-4]

(i-2) - i = -2    <
(j+1) - j =  1    >
(k-4) - k =  -4   <
```

** only loops with  case (<,>) are not interchangeable. In this example loops i and j are not interchangeable

## Implementation

If two loops are perfectly nested I call dependency_check(int outer, int inner) function to check whether they are dependent or not. The process is as follows:

1) For checking dependency I need the information of all the instructions including the opcode, the operands and the register containing the result of each instruction. For this means at the beginning of the pass , when I iterate over each block and visit each instruction I use getOpcode() and getOperand() functions and  extract all the information for each instruction, convert them to integer and store them in a global table called operands[][]. Operands is a big two dimensional array. Each instruction is assigned to one row. The columns are block index, opcode, result register, operands.
2) I find the initialization blocks related to inner and outer block indexes and keep track of the registers that loop indexes ,say i and j, are stored in.
3) I find the block that is related to body of inner loop in which the array is being calculated.  Lets consider the following array assignment:

   A[i][j] = A[i-1][j-1]

   What happens in this block is that an array assignment instruction is considered as two parts the left hand side of assignment and right hand side of it. First the right hand side gets calculated. Index of outer loop is loaded into a register, based on the index of the array (adding/subtracting a constant value to I) index of first dimension array  gets calculated , using getelementptr instruction one dimension gets calculated and repeating this process the second dimension also gets calculated . The same way the left hand side of the array gets calculated. The following IR code snippet indicates the block related to array.

```
; <label>:13:                                     ; preds = %10
  %14 = load i32, i32* %4, align 4
  %15 = sub nsw i32 %14, 1
  %16 = sext i32 %15 to i64
  %17 = getelementptr inbounds [200 x [100 x i32]], [200 x [100 x i32]]* %3, i64 0, i64 %16
  %18 = load i32, i32* %5, align 4
  %19 = sub nsw i32 %18, 1
  %20 = sext i32 %19 to i64
  %21 = getelementptr inbounds [100 x i32], [100 x i32]* %17, i64 0, i64 %20
  %22 = load i32, i32* %21, align 4
  %23 = load i32, i32* %4, align 4
  %24 = sext i32 %23 to i64
  %25 = getelementptr inbounds [200 x [100 x i32]], [200 x [100 x i32]]* %3, i64 0, i64 %24
  %26 = load i32, i32* %5, align 4
  %27 = sext i32 %26 to i64
  %28 = getelementptr inbounds [100 x i32], [100 x i32]* %25, i64 0, i64 %27
  store i32 %22, i32* %28, align 4
  br label %29
```

Fig.4

4) What I have done for dependency checking is I start from the store command in the block and keep track of it's operands. The first operand corresponds to A[i-1][j-1] (right hand side) and it's second operand corresponds to A[i][j] (left hand side). For each one I go up and investigate related getelementptr and load instructions for each of the separately to until I get the register related to array index calculation (for example ( i-1) or (j-1)). I repeat this process until I get all for array indexes ( i, j, i-1, j-1). Meanwhile I take care that they are all related to the same array.

5) Based on results from part 4 and logic explained at the beginning of this part I calculate the distance between each pair of indexes related to each dimension of the array and check if direction vector is like (<,>) . If so these two loops are dependent and **are** not interchangeable. Otherwise, they are interchangeable.
It is worth notice that Based on class slides , I only consider dir(<,>) as dependency. I don't check if we interchange we reach to (>,<).

6) W̌hen I find out two loops are interchangeable I keep the information required to swap them including the block indexes that should be changed, the instructions in each block that should be changed, and the operands that should change. I keep all information in a global table called instruction_changes[][] to be used in question 3.

7) What my program takes care of :
- Multi level nested loops (more than 3 level )
- Having several arrays in the body of the loop
-Having several separate nested loops in the program
- Detecting non perfect loops when the extra instruction comes either before or after the inner loop

**Question 3**
**Modify IR code**

At the beginning, before dependency check, when I am extracting information of each instruction and storing them in the form of integer values to be comparable and usable at the same time I store the Value* type of each operand in a map called value_star_map[][] to be used in this question.
To change the IR:

- I iterate over all block and their instructions once again
- using the information stored in instruction_changes derived at dependency check phase and the values stored at value_star_map table and the function setOperand() I change each required instruction in IR file.
-

For loop interchange the following instructions change :

✔ constant value operand in "icmp " instruction in condition check block of both loops
✔ register operand in "load " instruction in the array management block which is paired with getelementptr command.

- ✔ Register operand in “load” instruction in the index increment blocks of the loops which come after the body block

Tip: a very important issue is that in this question 3 we are asked two interchange any two pairs that are identified independent. But here there is a problem. Consider a nested loop with 3 loops say I, j ,k . And (I,j) and (j,k) are the pairs that are independent and can be interchange. So if we interchange pair (I,j) there is no guarantee that new pair (I,k) are still independent and interchangeable. This issue has been mentioned in the question 2. So, after running the program on a loop with more than 2 levels you can not track the IR modification to see if it is correct or not . For example for a 3 level loop if 2 pairs of them are independent then 2 consecutive IR modification get applied to IR . So tracking it so hard.
I suggest for testing this question please use a two level loop so only one time modification happens and you see it or use a 3 level nested loop in which only one pair is independent and can be interchange.

**Question 5**
**Profitability of loop interchange**

If two loops are independent and interchangeable , in the IR we can move the vectorizable loops into inner most positions of the nested loop and move parallelizable loops into outer most positions. In addition by reordering execution of statements in loops we can move dependence cycle and increase data locality and use of cache when accessing array elements. Because when CPU wants to access an element of array it reads one block of data that may contain other elements that may be referenced so soon.
Example: Consider the following nested loop

```
for (int j=0; j < 10; j++)
    for(int i=0; i < 10;i++)
      A[i][j] +=3
```

usually data sits in disk in row-major . It means when an array is stored on disk, first the row 0 comes, then row 1 comes , and so on. So, in the case of above loop where the inner loop iterates on the rows each time CPU should read one complete block from disk to memory to access an element in row 0 to access A[0][0], then read another complete block to access an element in row1 say A[0][1] and so on to access
If we interchange the loops we have :

```
for (int i=0; i < 10; i++)
    for(int j=0; j < 10;j++)
      A[i][j] +=3
```

This way by reading each block from each row we access data in the same row but in next columns . So , we don’t need to read from different rows to acess specific columns . This is data locality and is much faster.

**Question 4**
**Test Cases**

Example 1: Combination of all cases :)

The following test case contains a 4 level loop that some pairs are not perfectly nested because of an instruction between them (I cover both cases of not being perfectly nested explained in part 1). Also, there is two array assignment . The first one makes the loops completely independent and hence interchangeable and the second one makes it dependent and not interchangeable. As a result the loops are not interchangeable.

```
int a = 0;
    int A[40][30][20][10];
    int B[40][30][20][10];

    for(int i =0; i < 40;i++){
        a+=10;
        for(int j=0; j < 30;j++){
            for(int k=0; k < 20;k++ ){
                for(int l=0; l <10;l++){
                    A[i][j][k][l]=A[i-1][j-1][k-1][l-1];
                    B[i][j][k][l]=B[i-1][j-1][k+1][l+1];
                }
                a+=5;
            }
        }
    }
```

```
Output:

loop 4 is nested within loop 1. Not Perfectly Nested

loop 7 is nested within loop 1. Not Perfectly Nested

loop 10 is nested within loop 1. Not Perfectly Nested

loop 7 is nested within loop 4. Perfectly Nested

Loops are Dependent and Can Not be interchanged

loop 10 is nested within loop 4. Not Perfectly Nested

loop 10 is nested within loop 7. Not Perfectly Nested
```

In the above example loop 1 is loop i, loop 4 is loop j , loop 7 is loop k and loop 10 is loop l.
The direction vector for array A is (<,<,<,<) and the direction vector of array B is (<,<,>,>).
So based on array A dir vector we can interchange any perfectly nested loop. And based on dir vector of array B we can't interchange loops j, k (<,>) that are loops 4 and 7 (the red lines).
Also, because of a+=10 , loops I,j that are loops 1 and 4 are not perfectly nested. In addition, because of a+=5 loops k,l that are loops 7 and 10 are not perfectly nested.

Example 2: Nested loop with 4 levels

```
int array[40][30][20][10];

    for(int i =0; i < 40;i++){
         for(int j=0; j < 30;j++){
              for(int k=0; k < 20;k++ ){
                   for(int l=0; l <10;l++){
                        array[i][j]=array[i-1][j-1][k+1][l-1];
                   }}}}
```

Output of program is :

```
loop 4 is nested within loop 1. Perfectly Nested

Loops are Not Dependent and Can be interchanged


loop 7 is nested within loop 1. Not Perfectly Nested


loop 10 is nested within loop 1. Not Perfectly Nested


loop 7 is nested within loop 4. Perfectly Nested

Loops are Dependent and Can Not be interchanged


loop 10 is nested within loop 4. Not Perfectly Nested


loop 10 is nested within loop 7. Perfectly Nested

Loops are Not Dependent and Can be interchanged
```

In the above example loop 1 is loop i, loop 4 is loop j , loop 7 is loop k and loop 10 is loop l.
The direction vector is (<,<,>,<). So :
loop I,j ( loop 1 , loop 4) perfectly nested and are not dependent and interchangeable.
Loop j,k (loop 4, loop 7)are perfectly nested and dependent and not interchangeable .
Loop k,l (loop 7, loop 10 ) are perfectly nested and not dependent and interchangeable.
Other pairs of loops are not perfectly nested.

## Example 3 : IR modification

```
Int A[100][50];
for(int i=0; i<100;i++)
            for(int j=0;j<50;j++)
                                    A[i][j] = A[i-1]
[j-1];
Dir(<,<)
Loops are independent.
```

Before Modification of IR:

```
; <label>:6:                            ; preds = %33, %0
  %7 = load i32, i32* %4, align 4
  %8 = icmp slt i32 %7, 100
  br i1 %8, label %9, label %36

; <label>:9:                            ; preds = %6
  store i32 0, i32* %5, align 4
  br label %10

; <label>:10:                            ; preds = %29, %9
  %11 = load i32, i32* %5, align 4
  %12 = icmp slt i32 %11, 50
  br i1 %12, label %13, label %32

; <label>:13:                            ; preds = %10
  %14 = load i32, i32* %4, align 4
  %15 = sub nsw i32 %14, 1
  %16 = sext i32 %15 to i64
  %17 = getelementptr inbounds [100 x [100 x i32]], [100 x
[100 x i32]]* %3, i64 0, i64 %16
  %18 = load i32, i32* %5, align 4
  %19 = sub nsw i32 %18, 1
  %20 = sext i32 %19 to i64
  %21 = getelementptr inbounds [100 x i32], [100 x i32]* %17,
i64 0, i64 %20
  %22 = load i32, i32* %21, align 4
  %23 = load i32, i32* %4, align 4
  %24 = sext i32 %23 to i64
  %25 = getelementptr inbounds [100 x [100 x i32]], [100 x
[100 x i32]]* %3, i64 0, i64 %24
  %26 = load i32, i32* %5, align 4
  %27 = sext i32 %26 to i64
  %28 = getelementptr inbounds [100 x i32], [100 x i32]* %25,
i64 0, i64 %27
  store i32 %22, i32* %28, align 4
  br label %29

; <label>:29:                            ; preds = %13
  %30 = load i32, i32* %5, align 4
  %31 = add nsw i32 %30, 1
  store i32 %31, i32* %5, align 4
  br label %10

; <label>:32:                            ; preds = %10
  br label %33

; <label>:33:                            ; preds = %32
  %34 = load i32, i32* %4, align 4
  %35 = add nsw i32 %34, 1
  store i32 %35, i32* %4, align 4
  br label %6
```

Before Modification of IR:

```
; <label>:6:                            ; preds = %33, %0
  %7 = load i32, i32* %5, align 4
  %8 = icmp slt i32 %7, 50
  br i1 %8, label %9, label %36

; <label>:9:                            ; preds = %6
  store i32 0, i32* %4 align 4
  br label %10

; <label>:10:                            ; preds = %29, %9
  %11 = load i32, i32* %4, align 4
  %12 = icmp slt i32 %11, 100
  br i1 %12, label %13, label %32

; <label>:13:                            ; preds = %10
  %14 = load i32, i32* %5, align 4
  %15 = sub nsw i32 %14, 1
  %16 = sext i32 %15 to i64
  %17 = getelementptr inbounds [100 x [100 x i32]], [100 x
[100 x i32]]* %3, i64 0, i64 %16
  %18 = load i32, i32* %4, align 4
  %19 = sub nsw i32 %18, 1
  %20 = sext i32 %19 to i64
  %21 = getelementptr inbounds [100 x i32], [100 x i32]*
%17, i64 0, i64 %20
  %22 = load i32, i32* %21, align 4
  %23 = load i32, i32* %5, align 4
  %24 = sext i32 %23 to i64
  %25 = getelementptr inbounds [100 x [100 x i32]], [100 x
[100 x i32]]* %3, i64 0, i64 %24
  %26 = load i32, i32* %4, align 4
  %27 = sext i32 %26 to i64
  %28 = getelementptr inbounds [100 x i32], [100 x i32]*
%25, i64 0, i64 %27
  store i32 %22, i32* %28, align 4
  br label %29

; <label>:29:                            ; preds = %13
  %30 = load i32, i32* %4, align 4
  %31 = add nsw i32 %30, 1
  store i32 %31, i32* %5, align 4
  br label %10

; <label>:32:                            ; preds = %10
  br label %33

; <label>:33:                            ; preds = %32
  %34 = load i32, i32* %5, align 4
  %35 = add nsw i32 %34, 1
  store i32 %35, i32* %4, align 4
  br label %6
```

What to do to test :
You can run the MakFile using “make” command. If you use it don’t forget to change name of your program in the MakeFile. In MakeFile the name of the program is code.cpp.

Files included:
1- Skeleton.cpp
2-MakeFile
3-code.cpp (testcase)
4-code1.cpp (testcase)
5-code.ll