Module 12: CS31003: Compilers

Loop Optimization

Partha Pratim Das & Pralay Mitra

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in ; pralay@cse.iitkgp.ac.in

November 02, 2021

Module Outline

What is a Loop?

- In source language – designated by loop construct
  - `for`
  - `while`
  - `do-while`
  - goto with jump back?
  - ...

What is a Loop?

- In CFG
  - A loop is a back edge in the control-flow graph from a node l to a node h that *dominates* l.
  - We call h the *header node* of the loop.
  - The loop itself then consists of the nodes on a path from h to l.
  - It is convenient to organize the code so that a loop can be identified with its header node.
  - We then write $loop(h; l)$ if line l is in the loop with header h.
  - When loops are nested
    - (Generally) optimize the inner loops before the outer loops.
    - Inner loops are likely to be executed more often.
    - Inner loops could move computation to an outer loop from which it is hoisted further when the outer loop is optimized and so on.

In a CFG, a **Dominator** for a node n is a node d such that any path from the entry of the CFG to n goes through d. In this case we also say that d dominates n.

Source: Lecture Notes on Loop Optimizations by Frank Pfenning

Loop Invariant: Rules

An (**pure**[1]) expression is **loop invariant** if its value does not change throughout the loop. We define a set of rules to compute loop invariant using the predicates below:

$inv(h,\ p)$:	A pure expression p is invariant in loop h
$const(c)$:	c is a constant
$loop(h, I)$:	Instruction at I belongs to (dominated by) loop (header) h
$def(I, x)$:	Instruction at I defines (that is, writes to) variable x

R1: **Constant**

$$\frac{const(c)}{inv(h,\ c)}$$

R2: **Out-of-loop Definition**

$$\frac{def(I,\ x)\ \wedge\ \neg loop(h,\ I)}{inv(h,\ x)}$$

R3: **Composition** ($\oplus$ is $+$, etc.)

$$\frac{inv(h,\ s_1)\ \wedge\ inv(h,\ s_2)}{inv(h,\ s_1\ \oplus\ s_2)}$$

R4: **Propagation**

$$\frac{I\!:\ t\ \leftarrow\ p\ \wedge\ inv(h,\ p)\ \wedge\ loop(h,\ I)}{inv(h,\ t)}$$

[1] Expression with no side-effect – repeated evaluations with the same operands produce the same value

Loop Invariant: Pre-Header

- In order to hoist loop invariant computations out of a loop we should have a loop *pre-header* in the control-flow graph, which immediately dominates the loop header.
- When then move all the loop invariant computations to the pre-header, in order.

Hoisting Loop-Invariant: Example

- Consider the example of hoisting loop invariant computation in a loop to initialize all elements of a two-dimensional array (`A` is the *linearized view* of the array):

```
for (int i = 0; i < width * height; i++)
    A[i] = 1;
```

- We show the relevant part of the abstract assembly on the left. In the right is the result of hoisting the multiplication, enabled because both `width` and `height` are loop invariant (*Rule R2*) and therefore their product is (*Rule R3*), and hence `t1` is invariant (*Rule R4*)
- In TAC:

```
    i_0 = 0              // Pre-header

    goto loop (i_0)
loop (i_1):
    t1 = width * height // Hoist candidate
    if i_1 >= t1 goto exit
    ...
    i_2 = i_1 + 1
    goto loop (i_1)
exit:
```

```
    i_0 = 0              // Pre-header
    t1 = width * height // Hoisted
    goto loop (i_0)
loop (i_1):

    if i_1 >= t1 goto exit
    ...
    i_2 = i_1 + 1
    goto loop (i_1)
exit:
```

Hoisting Loop-Invariant: Caveat

- Consider the following code segment:

```
for (i = 0; i < n; ++i) {
    m = p + q;
    a[i] = m;
}
b = m;
```

- Clearly `m = p + q` is a loop invariant. And we hoist it:

```
i = 0;      // Pre-header
m = p + q;  // Loop invariant hoisted
for (; i < n; ++i) {
    a[i] = m;
}
b = m;
```

- Is it correct? No. If `n <= 0`, b will get a wrong value. As the loop does not iterate even once, b should have got the value of m before the loop. To fix, start with:

```
i = 0;         // Pre-header
if (i < n) {   // Guard for Loop invariant hoisting
    m = p + q; // Hoisted if the loop will run at least one
}
for (; i < n; ++i) {
    a[i] = m;
}
b = m;
```

Hoisting Loop-Invariant: Caveat

- Consider a function to search a character in a string:

```
int search(char *str, char c) {
    for (int i = 0; i < strlen(str); i++)
        if (str[i] == c) return i;
    return -1;
}
```

- Clearly `strlen(str)` is a loop invariant. And the following code would perform significantly better:

```
int search(char *str, char c) {
    int len = strlen(str);
    for (int i = 0; i < len; i++)
    // ...
}
```

- But the rules we have stated would not be able to detect it as it involves a function call. However, we can help the compiler:

```
int strlen(const char *str);              // Cannot change str

int search(const char *str, char c) {     // Cannot change str
    for (int i = 0; i < strlen(str); i++) // Easy to detect loop invariant
    // ...
}
```

- Now it gets easy to conclude that `strlen(str)` is a loop invariant. Note the importantce of having `const` declarations

Induction Variables

- *Hoisting loop invariant* computation is significant
- Optimizing computation which *changes by a constant amount each time around the loop* is probably even more important. We call such variables **basic induction variables**.
- A **derived induction variable** has the form `a * i + b`, where `i` is an induction variable and `a` and `b` are loop invariant
- Numbering induction variables:
  - Over a loop an induction variable (say, `i`) is sub-scripted with the iteration number to designate updates to the same variable in different iterations
  - Hence, `i_0` is the initial value (in *pre-header*, before entry to the loop)
  - `i_1` is the value in the *current iteration* (set from `i_0` on entry), and
  - `i_2` is the value in the *next iteration*
  - When the control jumps back, `i_2` is assigned to `i_1` for the next iteration to work

Induction Variables

- Consider an example of a function to check if a given array is sorted in ascending order.

```
bool is_sorted(int[] A, int n) { //@requires 0 <= n && n <= \length(A);
    for (int i = 0; i < n-1; i++) //@loop_invariant 0 <= i && i <= n-1;
        if (A[i] > A[i+1]) return false;
    return true;
}
```

- In TAC (`M` is the array in memory):

```
is_sorted(A, n):
    i_0 = 0                  // Pre-header
    goto loop (i_0)
loop (i_1):                  // Basic Induction Variable
    t0 = n - 1
    if i_1 >= t0 goto rtrue
    t1 = 4 * i_1             // Derived Induction Variable
    t2 = A + t1
    t3 = M[t2]
    t4 = i_1 + 1             // Derived Induction Variable
    t5 = 4 * t4
    t6 = A + t5
    t7 = M[t6]
    if t3 > t7 goto rfalse
    i_2 = i_1 + 1
    goto loop (i_2)
rtrue :
    return 1
rfalse :
    return 0
```

Induction Variables

- Let us consider t4 first. We see that common subexpression elimination applies. However, we would like to preserve the basic induction variable i_1 and its version i_2, so we apply code motion and then eliminate the second occurrence of i_1 + 1
- In TAC:

```
is_sorted(A, n):
    i_0 = 0
    goto loop (i_0)
loop (i_1):
    t0 = n - 1
    if i_1 >= t0 goto rtrue
    t1 = 4 * i_1
    t2 = A + t1
    t3 = M[t2]

    t4 = i_1 + 1
    t5 = 4 * t4
    t6 = A + t5
    t7 = M[t6]
    if t3 > t7 goto rfalse
    i_2 = i_1 + 1
    goto loop (i_2)
rtrue :
    return 1
rfalse :
    return 0
```

```
is_sorted(A, n):
    i_0 = 0
    goto loop (i_0)
loop (i_1):
    t0 = n - 1
    if i_1 >= t0 goto rtrue
    t1 = 4 * i_1
    t2 = A + t1
    t3 = M[t2]
    i_2 = i_1 + 1
    t4 = i_2
    t5 = 4 * t4
    t6 = A + t5
    t7 = M[t6]
    if t3 > t7 goto rfalse

    goto loop (i_2)
rtrue :
    return 1
rfalse :
    return 0
```

Induction Variables

- Next we look at the derived induction variable `t1 = 4 * i_1`. The idea is to see how we can calculate `t1` at a subsequent iteration from `t1` at a prior iteration. In order to achieve this effect, we add a new induction variable to represent `4 * i_1`. We call this `j` and add it to our loop variables.
- In TAC:

```
is_sorted(A, n):
    i_0 = 0

    goto loop (i_0)
loop (i_1):
    t0 = n - 1
    if i_1 >= t0 goto rtrue
    t1 = 4 * i_1
    t2 = A + t1
    t3 = M[t2]
    i_2 = i_1 + 1

    t4 = i_2
    t5 = 4 * t4
    t6 = A + t5
    t7 = M[t6]
    if t3 > t7 goto rfalse
    goto loop (i_2)
rtrue :
    return 1
rfalse :
    return 0
```

```
is_sorted(A, n):
    i_0 = 0
    j_0 = 4 * i_0             // Ensures j_0 = 4 * i_0
    goto loop (i_0, j_0)
loop (i_1, j_1):              // Requires j_1 = 4 * i_1
    t0 = n - 1
    if i_1 >= t0 goto rtrue
    t1 = j_1                  // Asserts j_1 = 4 * i_1
    t2 = A + t1
    t3 = M[t2]
    i_2 = i_1 + 1
    j_2 = 4 * i_2             // Ensures j_2 = 4 * i_2
    t4 = i_2
    t5 = 4 * t4
    t6 = A + t5
    t7 = M[t6]
    if t3 > t7 goto rfalse
    goto loop (i_2, j_2)
rtrue :
    return 1
rfalse :
    return 0
```

Induction Variables

- Crucial here is the invariant that `j_1 = 4 * i_1` when label `loop(i_1; j_1)` is reached. Now we calculate `j_2 = 4 * i_2 = 4 * (i_1 + 1) = 4 * i_1 + 4 = j_1 + 4` so we can express `j_2` in terms of `j_1` without multiplication. This is an example of strength reduction.
- Similarly: `j_0 = 4 * i_0 = 0` since `i_0 = 0`, which is an example of constant propagation followed by constant folding. In TAC:

```
is_sorted(A, n):
    i_0 = 0
    j_0 = 4 * i_0
    goto loop (i_0, j_0)
loop (i_1, j_1):
    t0 = n - 1
    if i_1 >= t0 goto rtrue
    t1 = j_1
    t2 = A + t1
    t3 = M[t2]
    i_2 = i_1 + 1
    j_2 = 4 * i_2
    t4 = i_2
    t5 = 4 * t4
    t6 = A + t5
    t7 = M[t6]
    if t3 > t7 goto rfalse
    goto loop (i_2, j_2)
rtrue :
    return 1
rfalse :
    return 0
```

```
is_sorted(A, n):
    i_0 = 0
    j_0 = 0                    // Ensures j_0 = 4 * i_0
    goto loop (i_0, j_0)
loop (i_1, j_1):               // Requires j_1 = 4 * i_1
    t0 = n - 1
    if i_1 >= t0 goto rtrue
    t1 = j_1                   // Asserts j_1 = 4 * i_1
    t2 = A + t1
    t3 = M[t2]
    i_2 = i_1 + 1
    j_2 = j_1 + 4              // Ensures j_2 = 4 * i_2
    t4 = i_2
    t5 = 4 * t4
    t6 = A + t5
    t7 = M[t6]
    if t3 > t7 goto rfalse
    goto loop (i_2, j_2)
rtrue :
    return 1
rfalse :
    return 0
```

Induction Variables

- With some copy propagation, and noticing that `n - 1` is loop invariant, we next get:
- In TAC:

```
is_sorted(A, n):
    i_0 = 0
    j_0 = 0

    goto loop (i_0, j_0)
loop (i_1, j_1):
    t0 = n - 1
    if i_1 >= t0 goto rtrue
    t1 = j_1
    t2 = A + t1
    t3 = M[t2]
    i_2 = i_1 + 1
    j_2 = j_1 + 4
    t4 = i_2
    t5 = 4 * t4
    t6 = A + t5
    t7 = M[t6]
    if t3 > t7 goto rfalse
    goto loop (i_2, j_2)
rtrue :
    return 1
rfalse :
    return 0
```

```
is_sorted(A, n):
    i_0 = 0
    j_0 = 0                 // Ensures j_0 = 4 * i_0
    t0 = n - 1
    goto loop (i_0, j_0)
loop (i_1, j_1):            // Requires j_1 = 4 * i_1

    if i_1 >= t0 goto rtrue

    t2 = A + j_1
    t3 = M[t2]
    i_2 = i_1 + 1
    j_2 = j_1 + 4           // Ensures j_2 = 4 * i_2

    t5 = 4 * i_2
    t6 = A + t5
    t7 = M[t6]
    if t3 > t7 goto rfalse
    goto loop (i_2, j_2)
rtrue :
    return 1
rfalse :
    return 0
```

Induction Variables

- With common subexpression elimination (noting the additional assertions we are aware of), we can replace `4 * i_2` by `j_2`. We combine this with copy propagation.
- In TAC:

```
is_sorted(A, n):
    i_0 = 0
    j_0 = 0
    t0 = n - 1
    goto loop (i_0, j_0)
loop (i_1, j_1):
    if i_1 >= t0 goto rtrue
    t2 = A + j_1
    t3 = M[t2]
    i_2 = i_1 + 1
    j_2 = j_1 + 4
    t5 = 4 * i_2
    t6 = A + t5
    t7 = M[t6]
    if t3 > t7 goto rfalse
    goto loop (i_2, j_2)
rtrue :
    return 1
rfalse :
    return 0
```

```
is_sorted(A, n):
    i_0 = 0
    j_0 = 0                   // Ensures j_0 = 4 * i_0
    t0 = n - 1
    goto loop (i_0, j_0)
loop (i_1, j_1):              // Requires j_1 = 4 * i_1
    if i_1 >= t0 goto rtrue
    t2 = A + j_1
    t3 = M[t2]
    i_2 = i_1 + 1
    j_2 = j_1 + 4             // Ensures j_2 = 4 * i_2

    t6 = A + j_2
    t7 = M[t6]
    if t3 > t7 goto rfalse
    goto loop (i_2, j_2)
rtrue :
    return 1
rfalse :
    return 0
```

Induction Variables

- We observe another derived induction variable, namely `t2 = A+j_1`. We give this a new name (`k_1 = A+j_1`) and introduce it into our function. Again we just calculate: `k_2 = A + j_2 = A + j_1 + 4 = k_1 + 4` and `k_0 = A + j_0 = A`
- In TAC:

```
is_sorted(A, n):
    i_0 = 0
    j_0 = 0

    t0 = n - 1
    goto loop (i_0, j_0)
loop (i_1, j_1):
    if i_1 >= t0 goto rtrue
    t2 = A + j_1
    t3 = M[t2]
    i_2 = i_1 + 1
    j_2 = j_1 + 4

    t6 = A + j_2
    t7 = M[t6]
    if t3 > t7 goto rfalse
    goto loop (i_2, j_2)
rtrue :
    return 1
rfalse :
    return 0
```

```
is_sorted(A, n):
    i_0 = 0
    j_0 = 0                 // Ensures j_0 = 4 * i_0
    k_0 = A + j_0           // Ensures k_0 = A + j_0
    t0 = n - 1
    goto loop (i_0, j_0, k_0)
loop (i_1, j_1, k_1):       // Requires j_1 = 4 * i_1 && k_1 = A + j_1
    if i_1 >= t0 goto rtrue
    t2 = k_1
    t3 = M[t2]
    i_2 = i_1 + 1
    j_2 = j_1 + 4           // Ensures j_2 = 4 * i_2
    k_2 = k_1 + 4           // Ensures k_2 = A + j_2
    t6 = k_2
    t7 = M[t6]
    if t3 > t7 goto rfalse
    goto loop (i_2, j_2, k_2)
rtrue :
    return 1
rfalse :
    return 0
```

Induction Variables

- After more round of constant propagation, common subexpression elimination, and dead code elimination we get:
- In TAC:

```
is_sorted(A, n):
    i_0 = 0
    j_0 = 0
    k_0 = A + j_0
    t0 = n - 1
    goto loop (i_0, j_0, k_0)
loop (i_1, j_1, k_1):
    if i_1 >= t0 goto rtrue
    t2 = k_1
    t3 = M[t2]
    i_2 = i_1 + 1
    j_2 = j_1 + 4
    k_2 = k_1 + 4
    t6 = k_2
    t7 = M[t6]
    if t3 > t7 goto rfalse
    goto loop (i_2, j_2, k_2)
rtrue :
    return 1
rfalse :
    return 0
```

```
is_sorted(A, n):
    i_0 = 0
    j_0 = 0                     // Ensures j_0 = 4 * i_0
    k_0 = A                     // Ensures k_0 = A + j_0
    t0 = n - 1
    goto loop (i_0, j_0, k_0)
loop (i_1, j_1, k_1):           // Requires j_1 = 4 * i_1 && k_1 = A + j_1
    if i_1 >= t0 goto rtrue

    t3 = M[k_1]
    i_2 = i_1 + 1
    j_2 = j_1 + 4               // Ensures j_2 = 4 * i_2
    k_2 = k_1 + 4               // Ensures k_2 = A + j_2

    t7 = M[k_2]
    if t3 > t7 goto rfalse
    goto loop (i_2, j_2, k_2)
rtrue :
    return 1
rfalse :
    return 0
```

Induction Variables

- With neededness analysis we can say that j_0, j_1, and j_2 are no longer needed and can be eliminated.
- Neededness Analysis is similar to Liveness and def-use Analysis; however, it ascertains if a computation is at all needed. We can see that j_0 and j_1 are live at `j_2 = j_1 + 1`, but is not needed as it does nothing other than updating j that is not used in any other computation.
- In TAC:

```
is_sorted(A, n):
    i_0 = 0
    j_0 = 0
    k_0 = A
    t0 = n - 1
    goto loop (i_0, j_0, k_0)
loop (i_1, j_1, k_1):
    if i_1 >= t0 goto rtrue
    t3 = M[k_1]
    i_2 = i_1 + 1
    j_2 = j_1 + 4
    k_2 = k_1 + 4
    t7 = M[k_2]
    if t3 > t7 goto rfalse
    goto loop (i_2, j_2, k_2)
rtrue :
    return 1
rfalse :
    return 0
```

```
is_sorted(A, n):
    i_0 = 0

    k_0 = A              // Ensures k_0 = A + j_0
    t0 = n - 1
    goto loop (i_0, k_0)
loop (i_1, k_1):         // Requires k_1 = A + j_1
    if i_1 >= t0 goto rtrue
    t3 = M[k_1]
    i_2 = i_1 + 1

    k_2 = k_1 + 4        // Ensures k_2 = A + j_2
    t7 = M[k_2]
    if t3 > t7 goto rfalse
    goto loop (i_2, k_2)
rtrue :
    return 1
rfalse :
    return 0
```

Induction Variables

- Unfortunately, i_1 is still needed, since it governs a conditional jump. In order to eliminate that we would have to observe that
  `i_1 >= t0 iff A + 4 * i_1 >= A + 4 * t0`.
  If we exploit this we obtain:
- In TAC:

```
is_sorted(A, n):
    i_0 = 0
    k_0 = A
    t0 = n - 1
    goto loop (i_0, k_0)
loop (i_1, k_1):
    if i_1 >= t0 goto rtrue
    t3 = M[k_1]
    i_2 = i_1 + 1
    k_2 = k_1 + 4
    t7 = M[k_2]
    if t3 > t7 goto rfalse
    goto loop (i_2, k_2)
rtrue :
    return 1
rfalse :
    return 0
```

```
is_sorted(A, n):
    i_0 = 0
    k_0 = A                    // Ensures k_0 = A + j_0
    t0 = n - 1
    goto loop (i_0, k_0)
loop (i_1, k_1):               // Requires k_1 = A + j_1
    if k_1 >= A + 4 * t0 goto rtrue
    t3 = M[k_1]
    i_2 = i_1 + 1
    k_2 = k_1 + 4              // Ensures k_2 = A + j_2
    t7 = M[k_2]
    if t3 > t7 goto rfalse
    goto loop (i_2, k_2)
rtrue :
    return 1
rfalse :
    return 0
```

Induction Variables

- Now i_0, i_1, and i_2 are no longer needed and can be eliminated. Moreover, `A+4 * t0` is loop invariant and can be hoisted.
- In TAC:

```
is_sorted(A, n):
    i_0 = 0
    k_0 = A
    t0 = n - 1


    goto loop (i_0, k_0)
loop (i_1, k_1):
    if k_1 >= A + 4 * t0 goto rtrue
    t3 = M[k_1]
    i_2 = i_1 + 1
    k_2 = k_1 + 4
    t7 = M[k_2]
    if t3 > t7 goto rfalse
    goto loop (i_2, k_2)
rtrue :
    return 1
rfalse :
    return 0
```

```
is_sorted(A, n):

    k_0 = A             // Ensures k_0 = A + j_0
    t0 = n - 1
    t8 = 4 * t0
    t9 = A + t8
    goto loop (k_0)
loop (k_1):             // Requires k_1 = A + j_1
    if k_1 >= t9  goto rtrue
    t3 = M[k_1]

    k_2 = k_1 + 4       // Ensures k_2 = A + j_2
    t7 = M[k_2]
    if t3 > t7 goto rfalse
    goto loop (k_2)
rtrue :
    return 1
rfalse :
    return 0
```

Induction Variables

- Final Code.
- We can avoid two memory accesses per iteration by unrolling the loop once. (Homework)
- In TAC:

```
is_sorted(A, n):
    k_0 = A
    t0 = n - 1
    t8 = 4 * t0
    t9 = A + t8
    goto loop (k_0)
loop (k_1):
    if k_1 >= t9  goto rtrue
    t3 = M[k_1]
    k_2 = k_1 + 4
    t7 = M[k_2]
    if t3 > t7 goto rfalse
    goto loop (k_2)
rtrue :
    return 1
rfalse :
    return 0
```

Loop Optimization: Practice Problem

Using the peep-hole and GCSE optimized TAC of Bubble Sort (Tutorial 7)

- Perform loop optimization using loop invariant and induction variable
- Perform Live Variable analysis on the loop optimized code and draw the CFG
- Perform Global Optimal Register Allocation (you should need less number of registers - without or with spilling)
- Flatten the CFG to generate the target code
- Optimize load-store in the target code
- Comment on the quality of the target code before and after loop optimization
- Can loop unrolling improve the memory access further?

Induction Variables: Without and With Opt. by VC++


```
; 5 : for (int i = 0; i < n - 1; i++)
mov DWORD PTR _i$1[ebp], 0
jmp SHORT $LN4@is_sorted
$LN3@is_sorted:
mov eax, DWORD PTR _i$1[ebp]
add eax, 1
mov DWORD PTR _i$1[ebp], eax
$LN4@is_sorted:
mov eax, DWORD PTR _n$[ebp]
sub eax, 1
cmp DWORD PTR _i$1[ebp], eax
jge SHORT $LN2@is_sorted

; 6 : if (A[i] > A[i + 1]) return 0;
mov eax, DWORD PTR _i$1[ebp]
mov ecx, DWORD PTR _A$[ebp]
mov edx, DWORD PTR _i$1[ebp]
mov esi, DWORD PTR _A$[ebp]
mov eax, DWORD PTR [ecx+eax*4]
cmp eax, DWORD PTR [esi+edx*4+4]
jle SHORT $LN1@is_sorted
xor al, al
jmp SHORT $LN5@is_sorted
$LN1@is_sorted:

; 7 : return 1;
jmp SHORT $LN3@is_sorted
$LN2@is_sorted:
mov al, 1
$LN5@is_sorted:
```

```
is_sorted(A, n):
; _A$ = ecx
; _n$dead$ = edx

; 5 : for (int i = 0; i < n - 1; i++)
xor eax, eax
$LL4@is_sorted:

; 6 : if (A[i] > A[i + 1]) return 0;
mov edx, DWORD PTR [ecx+eax*4]
cmp edx, DWORD PTR [ecx+eax*4+4]
jg SHORT $LN8@is_sorted

; 5 : for (int i = 0; i < n - 1; i++)
inc eax
cmp eax, 11 ; 0000000bH
jl SHORT $LL4@is_sorted

; 7     :  return 1;

mov al, 1
ret 0
$LN8@is_sorted:

; 6     :  if (A[i] > A[i + 1]) return 0;

xor al, al
ret 0
```

In-Class Assignment 1: 04-Nov-2021

- Consider the following code segment (`sizeof(int) = 4`): **Marks 10**

```
int a, b, c, i, n;
int p[100];

// ...

for(i = 0; i < n; ++i) {
    a = b + c;
    p[i] = a * a + 4 * i;
}
```

- Generate the TAC, draw CFG, and optimize for LCSE, GCSE, jumps, strength reduction etc.
- Optimize with loop invariant and induction variable
- Ensure correctness of your optimized code
- You may write your solution on notepad or on paper
- Submit by email to ppd@cse.iitkgp.ac.in within class hours (9:55am)
- Mention your name and roll number