Compiler Optimization Example

Source Code looks like:

```
float A[4][4];              // A is stored in ROW major order

for (int i=0;i<4;i++) {
        A[i][j] = A[i][j]+1.0;   // increment every element in jth column of A
}
```

Un-optimized assembly code looks like:

```
        LD      R1,#0                   // index variable i
        LD      R2,j                    // value of j (some constant btw 0  and 4)
        LD      F1,#1.0                 // floating point constant to add

LOOP:  LD       R3,A                    // base address of A from symbol table
        MULI    R4,R2,#16               // offset of current row
        ADD     R3,R4,R3                // R3 now has addr of first elt in row
        MUL     R4,R1,#4                // offset into row of current col
        ADD     R3,R4,R3                // R3 now has addr of A[i][j]

        LD.F    F2,(R3)
        ADD.F   F2,F2,F1
        ST.F    F2,(R3)

        ADDI    R1,#1                   // increment i

        SUBI    R5,R1,#4                // compare R1 and 4
        BNZ     R5,LOOP                 // if R1 != 3, loop back and do another
```

The compiler can optimize away those slow multiplies in the effective address calculation

```
        LD R1,#0                        // R1 now holds i*4, the offset of curr elt
        LD R2,j*#16                     // R2 holds offset to the current row
        LD F1,#1.0

LOOP:  LD R3,A
        ADD R3,R3,R2
        ADD R3,R3,R1                    // R3 has A + 16*j+4*i, the effective addr

        LD.F    F2,(R3)                 // "meat" of the loop is unchanged
        ADD.F   F2,F2,F1
        ST.F    F2,(R3)
```

```
ADD    R1,R1,#4                // increment R1 by size of an element
SUB    R5,R1,#16               // now check if R1 has 4*4    in it
BNZ    R5, LOOP
```

The important thing to note is that had the array effective address calculation not been explicit (the 5 instructions following the LOOP: label in the unoptimized version), the compiler could not have done anything with it.