正如注释所示，这段代码计算元素 i，j 的地址为 $x_A+4(n\cdot i)+4j=x_A+4(n\cdot i+j)$。这个地址的计算类似于定长数组的地址计算(参见 3.8.3 节)，不同点在于 1)由于增加了参数 n，寄存器的使用变化了；2)用了乘法指令来计算 $n\cdot i$(第 2 行)，而不是用 leaq 指令来计算 $3i$。因此引用变长数组只需要对定长数组做一点儿概括。动态的版本必须用乘法指令对 i 伸缩 n 倍，而不能用一系列的移位和加法。在一些处理器中，乘法会招致严重的性能处罚，但是在这种情况中无可避免。

在一个循环中引用变长数组时，编译器常常可以利用访问模式的规律性来优化索引的计算。例如，图 3-38a 给出的 C 代码，它计算两个 $n\times n$ 矩阵 A 和 B 乘积的元素 i，k。GCC 产生的汇编代码，我们再重新变为 C 代码(图 3-38b)。这个代码与固定大小数组的优化代码(图 3-37)风格不同，不过这更多的是编译器选择的结果，而不是两个函数有什么根本的不同造成的。图 3-38b 的代码保留了循环变量 j，用以判定循环是否结束和作为到 A 的行 i 的元素组成的数组的索引。

```c
/* Compute i,k of variable matrix product */
int var_prod_ele(long n, int A[n][n], int B[n][n], long i, long k) {
    long j;
    int result = 0;

    for (j = 0; j < n; j++)
        result += A[i][j] * B[j][k];

    return result;
}
```

a）原始的C代码

```c
/* Compute i,k of variable matrix product */
int var_prod_ele_opt(long n, int A[n][n], int B[n][n], long i, long k) {
    int *Arow = A[i];
    int *Bptr = &B[0][k];
    int result = 0;
    long j;
    for (j = 0; j < n; j++) {
        result += Arow[j] * *Bptr;
        Bptr += n;
    }
    return result;
}
```

b）优化后的C代码

图 3-38　计算变长数组的矩阵乘积的元素 i，k 的原始代码和优化后的代码。编译器自动执行这些优化

下面是 var_prod_ele 的循环的汇编代码：

```
   Registers: n in %rdi, Arow in %rsi, Bptr in %rcx
              4n in %r9, result in %eax, j in %edx
   .L24:                                    loop:
     movl     (%rsi,%rdx,4), %r8d             Read Arow[j]
     imull    (%rcx), %r8d                    Multiply by *Bptr
```