Chuong VII

Code tuning

I. Code tuning

1. Hiệu năng của chương trình và Code Tuning
2. Các phương pháp Code Tuning

1.1. Hiệu năng

Sau khi áp dụng các kỹ thuật xây dựng CT PM:

- CT đã có tốc độ đủ nhanh
  - Không nhất thiết phải quan tâm đến viêc tối ưu hóa hiệu năng
  - Chỉ cần giữ cho CT đơn giản và dễ đọc
- Hầu hết các thành phần của 1 CT có tốc độ đủ nhanh
  - Thường chỉ một phần nhỏ làm cho CT chạy chậm
  - Tối ưu hóa riêng phần này nếu cần
- Các bước làm tăng hiệu năng thực hiện CT
  - Tính toán thời gian thực hiện của các phần khác nhau trong CT
  - Xác định các “hot spots” – đoạn mã lệnh đòi hỏi nhiều thời gian thực hiện
  - **Tối ưu hóa phần CT đòi hỏi nhiều thời gian thực hiện**
  - Lặp lại các bước nếu cần

Tối ưu hóa hiệu năng của CT ?

- Cấu trúc dữ liệu tốt hơn, giải thuật tốt hơn
  - Cải thiện độ phức tạp tiệm cận (*asymptotic complexity)*
    - Tìm cách khống chế tỉ lệ giữa số phép toán cần thực hiện và số lượng các tham số đầu vào
    - Ví dụ: thay giải thuật sắp xếp có độ phức tạp $O(n^2)$ bằng giải thuật có độ phức tạp $O(n \log n)$
  - Cực kỳ quan trọng khi lượng tham số đầu vào rất lớn
  - Đòi hỏi LTV phải nắm vững kiến thức về CTDL và giải thuật
- Mã nguồn tốt hơn: viết lại các đoạn lệnh sao cho chúng có thể được trình dịch tự động tối ưu hóa và tận dụng tài nguyên phần cứng
  - Cải thiện các yếu tố không thể thay đổi
    - Ví dụ: Tăng tốc độ tính toán bên trong các vòng lặp: từ *1000n* thao tác tính toán bên trong vòng lặp xuống còn *10n* thao tác tính toán
  - Cực kỳ quan trọng khi 1 phần của CT chạy chậm
  - Đòi hỏi LTV nắm vững kiến thức về phần cứng, trình dịch và quy trình thực hiện CT

→ Code tuning

1.2. Code tuning (tinh chỉnh mã nguồn) là gì ?

- Thay đổi mã nguồn đã chạy thông theo hướng hiệu quả hơn nữa
- Chỉ thay đổi ở phạm vi hẹp, ví dụ như chỉ liên quan đến 1 CTC, 1 tiến trình hay 1 đoạn mã nguồn
- Không liên quan đến việc thay đổi thiết kế ở phạm vi rộng, nhưng có thể góp phần cải thiện hiệu năng cho từng phần trong thiết kế tổng quát

1.3. Cải thiện hiệu năng thông qua cải thiện mã nguồn

- Có 3 cách tiếp cận để cải thiện hiệu năng thông qua cải thiện mã nguồn
  - Lập hồ sơ mã nguồn (profiling): chỉ ra những đoạn lệnh tiêu tốn nhiều thời gian thực hiện
  - Tinh chỉnh mã nguồn (code tuning): tinh chỉnh các đoạn mã nguồn
  - Tinh chỉnh có chọn lựa (options tuning): tinh chỉnh thời gian thực hiện hoặc tài nguyên sử dụng để thực hiện CT
- Khi nào cần cải thiện hiệu năng theo các hướng này
  - Sau khi đã kiểm tra và gỡ rối chương trình
    - Không cần tinh chỉnh 1 CT chạy chưa đúng
    - Việc sửa lỗi có thể làm giảm hiệu năng CT
    - Việc tinh chỉnh thường làm cho việc kiểm thử và gỡ rối trở nên phức tạp
  - Sau khi đã bàn giao CT
    - Duy trì và cải thiện hiệu năng
    - Theo dõi việc giảm hiệu năng của CT khi đưa vào sử dụng

1.4. Quan hệ giữa hiệu năng và tinh chỉnh mã nguồn

- Việc giảm thiểu số dòng lệnh viết bằng 1 NNLT bậc cao KHÔNG có nghĩa là :
  - Làm tăng tốc độ chạy CT
  - làm giảm số lệnh viết bằng ngôn ngữ máy

```
for (i = 1;i<11;i++)  a[i] = i;
```

```
a[ 1 ] = 1 ; a[ 2 ] = 2 ;
a[ 3 ] = 3 : a[ 4 ] = 4 ;
a[ 5 ] = 5 ; a[ 6 ] = 6 ;
a[ 7 ] = 7 ; a[ 8 ] = 8 ;
a[ 9 ] = 9 ; a[ 10 ] = 10 ;
```

Language	*for*-Loop Time	Straight-Code Time	Time Savings	Performance Ratio
Visual Basic	8.47	3.16	63%	2.5:1
Java	12.6	3.23	74%	4:1

Quan hệ giữa hiệu năng và tinh chỉnh mã nguồn

- Luôn định lượng được hiệu năng cho các phép toán
- Hiệu năng của các phép toán phụ thuộc vào:
  - Ngôn ngữ lập trình
  - Trình dịch / phiên bản sử dụng
  - Thư viện / phiên bản sử dụng
  - CPU
  - Bộ nhớ máy tính
- Hiệu năng của việc tinh chỉnh mã nguồn trên các máy khác nhau là khác nhau.

Quan hệ giữa hiệu năng và tinh chỉnh mã nguồn

- 1 số kỹ thuật viết mã hiệu quả được áp dụng để tinh chỉnh mã nguồn
- Nhưng nhìn chung không nên vừa viết chương trình vừa tinh chỉnh mã nguồn
  - Không thể xác định được những nút thắt trong chương trình trước khi chạy thử toàn bộ chương trình
  - Việc xác định quá sớm các nút thắt trong chương trình sẽ gây ra các nút thắt mới khi chạy thử toàn bộ chương trình
  - Nếu vừa viết chương trình vừa tìm cách tối ưu mã nguồn, có thể làm sai lệch mục tiêu của chương trình

Hiệu suất công nghệ

- Băng thông thiết bị (Tốc độ tăng dần):user input device, tape drives, network, CDROM, hard drive, memory mapped local BUS device (graphics memory), uncached main memory, external cached main memory, local/CPU cached memory, local variables (registers.)
- Tốc độ thực hiện các phép toán : Lượng giác->Căn > % > / > * > - > + > << > >>
- Tốc độ thực hiện lệnh : indirect function calls, switch() statements, fixed function calls, if() statements, while() statements

2. Các kỹ thuật tinh chỉnh mã nguồn

- Tinh chỉnh các biểu thức logic
- Tinh chỉnh các vòng lặp
- Tinh chỉnh việc biến đổi dữ liệu
- Tinh chỉnh các biểu thức
- Tinh chỉnh dãy lệnh
- Viết lại mã nguồn bằng ngôn ngữ assembler
- Lưu ý: Càng thay đổi nhiều thì càng không cải thiện được hiệu năng

2.1. Tinh chỉnh các biểu thức logic

- Không kiểm tra khi đã biết kết quả rồi
  - Initial code

```
if ( 5 < x ) && ( x < 10 ) ….
```

  - Tuned code

```
if ( 5 < x )
    if ( x < 10 )
     ….
```

2.1. Tinh chỉnh các biểu thức logic

- Không kiểm tra khi đã biết kết quả rồi
- Ví dụ: tinh chỉnh như thế nào ???

```
negativeInputFound = False;
for ( i = 0; i < iCount; i++ ) {
        if ( input[ i ] < 0 ) {
        negativeInputFound = True;
    }
}
```

Dùng break:

Language	Stra
C++	4.27
Java	4.85

```
input[iCount]=-1;
i=0;
while (input[i] >=0) i++;
If (i<iCount) negativeInput=true;
else negativeInput=false;
```

2.1. Tinh chỉnh các biểu thức logic

- Sắp xếp thứ tự các phép kiểm tra theo tần suất xảy ra kết quả đúng
  - Initial code

```
Select inputCharacter
    Case "+", "="
        ProcessMathSymbol( inputCharacter )
    Case "0" To "9"
        ProcessDigit( inputCharacter )
    Case ",", ".", ":", ";", "!", "?"
        ProcessPunctuation( inputCharacter )
    Case " "
        ProcessSpace( inputCharacter )
    Case "A" To "Z", "a" To "z"
        ProcessAlpha( inputCharacter )
    Case Else
        ProcessError( inputCharacter )
End Select
```

2.1. Tinh chỉnh các biểu thức logic

Language	Straight Time	Code-Tuned Time	Time Savings
C#	0.220	0.260	-18%
Java	2.56	2.56	0%
Visual Basic	**0.280**	**0.260**	**7%**

```
Select inputCharacter
    Case "A" To "Z", "a" To "z"
        ProcessAlpha( inputCharacter )
    Case " "
        ProcessSpace( inputCharacter )
    Case ",", ".", ":", ";", "!", "?"
        ProcessPunctuation( inputCharacter )
    Case "0" To "9"
        ProcessDigit( inputCharacter )
    Case "+", "="
        ProcessMathSymbol( inputCharacter )
    Case Else
        ProcessError( inputCharacter )
End Select
```

2.1. Tinh chỉnh các biểu thức logic

- Sắp xếp thứ tự các phép kiểm tra theo tần suất xảy ra kết quả đúng
  - Tuned code: chuyển lệnh switch thành các lệnh if - then - else

Language	Straight Time	Code-Tuned Time	Time Savings
C#	0.630	0.330	48%
Java	0.922	0.460	50%
Visual Basic	**1.36**	**1.00**	**26%**

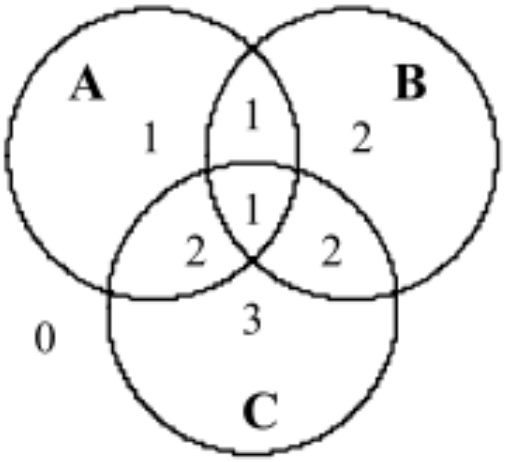
2.1. Tinh chỉnh các biểu thức logic

- So sánh hiệu năng của các lệnh có cấu trúc tương đương

Language	*case*	*if-then-else*	Time Savings	Performance Ratio
C#	0.260	0.330	-27%	1:1
Java	2.56	0.460	82%	6:1
Visual Basic	0.260	1.00	258%	1:4

2.1. Tinh chỉnh các biểu thức logic

- Thay thế các biểu thức logic phức tạp bằng bảng tìm kiếm kết quả

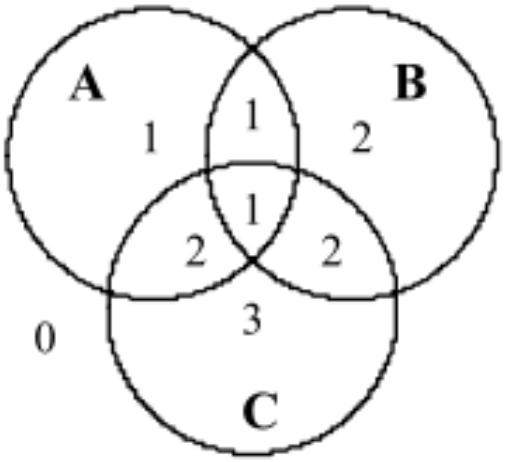


Initial code

```
if ( ( a && !c ) || ( a && b && c ) ) {
        category = 1;
}
else if ( ( b && !a ) || ( a && c && !b
) ) {
        category = 2;
}
else if ( c && !a && !b ) {
        category = 3;
}
else {
        category = 0;
}
```

2.1. Tinh chỉnh các biểu thức logic

- Thay thế các biểu thức logic phức tạp bằng bảng tìm kiếm kết quả



Tuned code

```
// define categoryTable
static int categoryTable[2][2][2] = {
// !b!c  !bc  b!c  bc
    0,   3,   2,   2,   //   !a
    1,   2,   1,   1    //    a
};
...

category = categoryTable[ a ][ b ][ c ];
```

2.1. Tinh chỉnh các biểu thức logic

- Lazy evaluation: 1 trong các kỹ thuật viết mã chương trình hiệu quả đã học

2.2. Tinh chỉnh các vòng lặp

- Loại bỏ bớt việc kiểm tra điều kiện bên trong vòng lặp
  - Initial code

```
for ( i = 0; i < count; i++ ) {
    if ( sumType == SUMTYPE_NET ) {
        netSum = netSum + amount[ i ];
    }
    else {
        grossSum = grossSum + amount[ i ];
    }
}
```

```
if ( sumType == SUMTYPE_NET ) {
    for ( i = 0; i < count; i++ ) {
```

Language	Straight Time	Code-Tuned Time	Time Savings
C++	**2.81**	**2.27**	**19%**
Java	3.97	3.12	21%
Visual Basic	2.78	2.77	<1%
Python	8.14	5.87	28%

2.2. Tinh chỉnh các vòng lặp

- Nếu các vòng lặp lồng nhau, đặt vòng lặp xử lý nhiều công việc hơn bên trong
  - Initial code

```
for ( j = 0; j < 1000000000; j++ ) {
    for ( i = 0; i < 5; i++ ) {
        sum = sum + i*j;
    }
}
```

  - ?

```
for (c = 0; c < 5; c++ ) {
    for (r = 0; r < 1000000000; r++) {
        sum = sum + table[ r ][ c ];
    }
}
```

```
for ( i = 0; i < 5; i++ ) {
   for ( j = 0; j < 1000000000; j++ ) {
        sum = sum + i*j;
    }
} i
```

Ban dau ;
1ty ktra j
1ty ++
1ty gan I
5ty ss I
5ty ++
5ty lap

Sau tc ;
5 ktra i
5 ++
5 gan I
5ty ss j
5ty ++
5ty lap

3ty – 15 !!!!

2.2. Tinh chỉnh các vòng lặp

- Một số kỹ thuật viết các lệnh lặp hiệu quả đã học
  - Ghép các vòng lặp với nhau
  - Giảm thiểu các phép tính toán bên trong vòng lặp nếu có thể

```
for (i=0; i<n; i++) {
    balance[i] += purchase->allocator->indiv->borrower;
    amounttopay[i] = balance[i]*(prime+card)*pcentpay;
}
newamt = purchase->allocator->indiv->borrower;
payrate = (prime+card)*pcentpay;
for (i=0; i<n; i++) {
    balance[i] += newamt;
    amounttopay[i] = balance[i]*payrate;
}
```

Thay the phep nhan trong vong lap = phep cong

```
for (i=0; i<n; i++)
    a[i] = i*conversion;
```

=>

```
sum = 0;
for (i=0; i<n; i++) {
    a[i] = sum;
    sum += conversion;
}
```

Better :

```
a[0] = 0;
for (i=1; i<n; i++)
   a[i] = a[i-1]+conversion;
```

```
for (i=0; i<r; i++)
    for (j=0; j<c; j++)
        A[j] = B[j] + C[i];
```

=>

```
for (i=0; i<r; i++) {
    temp = C[i];
    for (j=0; j<c; j++)
        A[j] = B[j] + temp;
}
```

2.3. Tinh chỉnh việc biến đổi dữ liệu

- Một số kỹ thuật viết mã hiệu quả đã học:
  - Sử dụng kiểu dữ liệu có kích thước nhỏ nếu có thể
  - Sử dụng mảng có số chiều nhỏ nhất có thể
  - Đem các phép toán trên mảng ra ngoài vòng lặp nếu có thể
  - Sử dụng các chỉ số phụ
  - Sử dụng biến trung gian
  - Khai báo kích thước mảng = 2^n

2.4. Tinh chỉnh các biểu thức (đã học)

- Thay thế phép nhân bằng phép cộng
- Thay thế phép lũy thừa bằng phép nhân
- Thay việc tính các hàm lượng giác bằng cách gọi các hàm lượng giác có sẵn
- Sử dụng kiểu dữ liệu có kích thước nhỏ nếu có thể
  - long int → int
  - floating-point → fixed-point, int
  - double-precision → single-precision
- Thay thế phép nhân đôi / chia đôi số nguyên bằng phép bitwise : <<, >>
- Sử dụng hằng số hợp lý
- Tính trước kết quả
- Sử dụng biến trung gian

2.5. Tinh chỉnh dãy lệnh (đã học)

- Sử dụng các hàm inline

2.6. Viết lại mã nguồn bằng ngôn ngữ assembler

- Viết chương trình hoàn chỉnh bằng 1 NNLT bậc cao
- Kiểm tra tính chính xác của toàn bộ chương trình
- Nếu cần cải thiện hiệu năng thì áp dụng kỹ thuật lập hồ sơ mã nguồn để tìm “hot spots” (chỉ khoảng 5 % CT thường chiếm 50% thời gian thực hiện, vì vậy ta có thể thường xác định đc 1 mẩu code như là hot spots)
- Viết lại những mẩu nhỏ các lệnh = assembler để tăng tốc độ thực hiện

Giúp trình dịch làm tốt công việc của nó

- Trình dịch có thể thực hiện 1 số thao tác tôi ưu hóa tự động
  - Cấp phát thanh ghi
  - Lựa chọn lệnh để thực hiện và thứ tự thực hiện lệnh
  - Loại bỏ 1 số dòng lệnh kém hiệu quả
- Nhưng trình dịch không thể tự xác định
  - Các hiệu ứng phụ (side effect) của hàm hay biểu thức: ngoài việc trả ra kết quả, việc tính toán có làm thay đổi trạng thái hay có tương tác với các hàm/biểu thức khác hay không
  - Hiện tượng nhiều con trỏ trỏ đến cùng 1 vùng nhớ (memory aliasing)
- Tinh chỉnh mã nguồn có thể giúp nâng cao hiệu năng
  - Chạy thử từng đoạn chương trình để xác định “hot spots”
  - Đọc lại phần mã viết bằng assembly do trình dịch sản sinh ra
  - Xem lại mã nguồn để giúp trình dịch làm tốt công việc của nó

Khai thác hiệu quả phần cứng

- Tốc độ của 1 tập lệnh thay đổi khi môi trường thực hiện thay đổi
- Dữ liệu trong thanh ghi và bộ nhớ đệm được truy xuất nhanh hơn dữ liệu trong bộ nhớ chính
  - Số các thanh ghi và kích thước bộ nhớ đệm của các máy tính khác nhau
  - Cần khai thác hiệu quả bộ nhớ theo vị trí không gian và thời gian
- Tận dụng các khả năng để song song hóa
  - Pipelining: giải mã 1 lệnh trong khi thực hiện 1 lệnh khác
    - Áp dụng cho các đoạn mã nguồn cần thực hiện tuần tự
  - Superscalar: thực hiện nhiều thao tác trong cùng 1 chu kỳ đồng hồ (clock cycle)
    - Áp dụng cho các lệnh có thể thực hiện độc lập
  - Speculative execution: thực hiện lệnh trước khi biết có đủ điều kiện để thực hiện nó hay không

Kết luận

- Hãy lập trình một cách thông minh, đừng quá cứng nhắc
  - Không cần tối ưu 1 chương trình đủ nhanh
  - Tối ưu hóa chương trình đúng lúc, đúng chỗ
- Tăng tốc chương trình
  - Cấu trúc dữ liệu tốt hơn, giải thuật tốt hơn: hành vi tốt hơn
  - Các đoạn mã tối ưu: chỉ thay đổi ít
- Các kỹ thuật tăng tốc chương trình
  - Tinh chỉnh mã nguồn theo hướng
    - Giúp đỡ trình dịch
    - Khai thác khả năng phần cứng

1 số ví dụ tăng hiệu năng

- Before:

  for(i=0;i<1000000000;i++) {

  printf("%d\n",i*10); } ???

Before:

char x; int y;

y = x; ???

So sánh phép toán trên int và char

char sum_char(char a, char b) {
char c; c = a + b; return c; }
int sum_int(int a, int b) {
int c; c = a + b; return c; }

Gọi hàm 1 sẽ có các thao tác sau:

1.1 Chuyển tham số thứ 2 thành int (C và C++ theo thứ tự ngược nhau)
1.2 Push parameter on the stack as b.
1.3 Convert the first parameter into an int .
1.4 Push parameter on to the stack as a.
1.5 a + b và kq được cast to a char rồi gán cho c.
1.6 c lại đc chuyển thành int
1.7 c được trả về cho lời gọi hàm.
1.8 Giá trị trả về lại đc chuyển thành char.
1.9 The result is stored.

Gọi hàm 2 sẽ thực hiện các thao tác sau:

2.1 Push int b on stack
2.2 Push int a on stack
2.3 a+b và kq đc gán cho c
2.4 c đc trả về to caller.
2.5 The called function stores the returned value

Before

```
a = c * (3*x + 2*y);
d = (3*x + 2*y) >> 1;
y = 2 * x;
```

```
a = b * 15;      ???
```

```
for (i = 1; i < n; i++) {
  k = i * 4 + m;
  c += 2 * a[k];
}
```

```
Int tg=n*4+m;
for(k=m+4;k<tg;k+=4)
  c+=a[k]<<1;
```

```
for (i = 0; i < n; i++)
    x[i] = a[i] + b[i];
for (i = 0; i < n; i++)
    y[i] = a[i] + c[i];
```

Cache

```
for (c = 0; c < n; c++)
      for (r = 0; r < m; r++)
         arr[r][c] = arr[r][c] + 1;
```

sẽ chậm hơn đoạn code sau:

```
for (x = 0; x < m; x++)
   for (y = 0; y < n; y++)
      arr[x][y] = arr[x][y] + 1;
```

Trường hợp thứ nhất, các phần tử của mảng không được truy cập tuần tự ==> xác xuất cache missing rất lớn. Trong đoạn mã nguồn thứ 2, các phần tử được truy cập tuần tự nên tốc độ sẽ nhanh hơn.

Cache

```
double _3dVectorX[1000], _3dVectorY[1000], _3dVectorZ[1000];
    _3dVectorX[0] += _3dVectorX[20];
    _3dVectorY[0] += _3dVectorY[20];
    _3dVectorZ[0] += _3dVectorZ[20];
```

Vì các giá trị nằm ở trong 3 mảng khác nhau nên ta không sử dụng được tính chất của cache. Nếu ta thiết kế lại như sau thì kết quả sẽ khác hẳn:

```
struct _3dVector {
   double X, Y, Z;
};
struct _3dVector arr[1000];
arr[0].X += arr[20].X;
arr[0].Y += arr[20].Y;
arr[0].Z += arr[20].Z;
```

ở đây 3 giá trị arr[20].X, arr[20].Y, arr[20].Z sẽ nằm liên tiếp nhau trong bộ nhớ, và xác xuất chúng nằm trong 1 cache line là rất lớn.

Short Boolean Expression Evaluation

- Trong nhiều ngôn ngữ, những biểu thức logic dạng *Exp = (Exp1 OR Exp2 OR Exp3)* sẽ được thực hiện lần lượt từ trái qua phải. Trong biểu thức trên, nếu Exp1 có giá trị TRUE thì cả biểu thức Exp sẽ có giá trị TRUE, và chương trình sẽ không tính giá trị của Exp2 và Exp3 nữa
- Ví dụ :

```
int letter_count(const char *buf, int size)
    {
      int count, i;

      count = 0;
      for (i = 0; i < size; i++) {
        if ((buf[i] >= 'A' && buf[i] <= 'Z') ||
            (buf[i] <= 'z' && buf[i] >= 'a'))
          count++;
      }
      return count;
    }

    ????
```

(**Thông thường các ký tự thường thường gặp nhiều hơn các ký tự hoa !)**

Biến mảng 2 chiều thành 1 chiều

```
int arr[500][300];
   for (i = 0; i < 500; i++)
      for (j = 0; j < 300; j++)
         arr[i][j] += 1;
```

??? (arr[0][301] = ?)

Loop

Loop Unrolling

Đây là một thủ thuật hay được các lập trình viên sử dụng . Chúng ta hãy xem xét đoạn mã nguồn sau:

```
for (i = 0; i < 2*n; i++) {
  a[i] = b[i] + c[i];
}
```

đoạn mã nguồn này có thể được viết dưới dạng khác như sau:

```
for (i = 0; i < 2*n; i += 2) {
  a[i] = b[i] + c[i];
  a[i+1] = b[i+1] + c[i+1];
}
```

Reverse Loop Counting

Do cấu tạo đặc biệt, một số CPU thực hiện loop nhanh hơn nếu như thay vì tăng dần chỉ số, ta giảm dần nó đến 0. Ví dụ:

```
sum = 0;
for (i = 1; i <= n; i++) sum += i;
```

ta có thể viết theo kiểu khác như sau:

```
sum = 0;
for (i = count; i > 0; i--) sum += i;
```

- **Loop Flipping**
  Thực chất của kỹ thuật này là ta chuyển việc kiểm tra điều kiện kết thúc vòng lặp từ đầu (top) đến cuối (bottom). Ví dụ:

```
sum = 0;
i = 1;
while (i <= count) {
   sum += i;
   i++;
}
```

  nếu như ta bảo đảm là count lúc nào cũng >= 1 thì có thể dùng vòng lặp do...while để thay thế:

```
sum = 0;
i = 1;
do {
   sum += i;
   i++;
} while (i <= count);
```

- Đối với các kỹ thuật tối ưu đơn lẻ thì hầu hết các trình biên dịch tự biết áp dụng
- Khi kết hợp cả 3 kỹ thuật Loop Unrolling, Loop Reverse Counting và Loop Flipping lại với nhau thì tốc độ nhanh hơn đáng kể (khoảng 20%):

```
sum = 0;
i = count;
do {
   sum += i;
   i--;
} while (i > 0);
```

Accesses by Index, Accesses by Pointer, and Counting Down

- Thông thường ta thường dùng vòng lặp tăng dần cho chỉ số của mảng như ví dụ sau :

```
for (index = 0; index < count; ++index)  {
    sum[index] = left[index] + right[index];
}
```

- Đôi khi ta dùng con trỏ để truy cập tới các phần tử mảng:

```
for (index = 0; index < count; ++index) {
    *sum = *left + *right;
    ++right;
    ++left;
    ++sum;
}
```

- Một cách khác, ta dùng vòng lặp giảm dần :

```
for (; count > 0; --count) {
    *sum = *left + *right;
    ++right;
    ++left;
    ++sum;
}
```

- Cả 3 cách trên đều cho kết quả như nhau, nhưng cách cuối cùng làm hiệu năng tăng rõ rệt

Index Access to an Array

```
SUB.W R6,R6
LoopTop:
MOV.W R6,R5 ; convert the
ADD.W R5,R5 ; index.
MOV.W @(SP),R0 ; calculate
ADD.W R5,R0 ; the address.
.
. ; repeat.
.
MOV.W @R1,R1 ; load the
MOV.W @R2,R2 ; array values.
ADD.W R2,R1
MOV.W R1,@R0 ; store the sum.
ADDS.W #1,R6 ; increment the
CMP.W R4,R6 ; index and loop
BCS LoopTop
```

Pointer Access to an Array

```
SUB.W R6,R6
LoopTop:
MOV.W @R3,R0 ; load the
MOV.W @R5,R1 ; array values.
ADD.W R1,R0
MOV.W R0,@R4 ; store the sum.
ADDS.W #2,R5 ; increment the
ADDS.W #2,R3 ; pointers
ADDS.W #2,R4
ADDS.W #1,R6 ; increment the
MOV.W @(SP),R0 ; index and loop
CMP.W R0,R6
BCS LoopTop
```

```
LoopTop:
MOV.W @R4,R0 ; load the
MOV.W @R3,R1 ; array values.
ADD.W R1,R0
MOV.W R0,@R5 ; store the sum.
ADDS.W #2,R3 ; increment the
ADDS.W #2,R4 ; pointers.
ADDS.W #2,R5
SUBS.W #1,R6 ; decrement the
CMP.W R6,R6 ; count and loop
BCS LoopTop
```