Tối ưu mã nguồn C/C++

Tại sao phải tối ưu mã lệnh?

Sự ra đời của các trình biên dịch hiện đại đã giúp lập trình viên cải thiện đáng kể thời gian và công sức phát triển phần mềm. Một vấn đề đáng quan tâm là xu hướng phát triển phần mềm theo hướng trực quan nhanh và tiện dụng dần làm mặt bằng kĩ năng viết mã lệnh của các lập trình viên giảm rõ rệt vì họ trông cậy hoàn toàn vào sự hỗ trợ của trình biên dịch. Khi phát triển một hệ thống phần mềm có tần suất xử lý cao, ví dụ các sản phẩm có chức năng điều phối hoạt động dây chuyền sản xuất trong nhà máy, thì bên cạnh sự hỗ trợ của một trình biên dịch mạnh còn cần đến kĩ năng tối ưu mã lệnh của lập trình viên. Kĩ năng tốt sẽ biến công việc lập trình khô khan, với các đoạn code tưởng chừng lạnh lùng trở nên sinh động. Một đoạn mã lệnh tốt sẽ tận dụng tối đa ưu điểm của ngôn ngữ và khả năng xử lý của hệ thống, từ đó giúp nâng cao đáng kể hiệu suất hoạt động của hệ thống.

Để chương trình hoạt động tối ưu, điều đầu tiên là tận dụng những hỗ trợ sẵn có của trình biên dịch thông qua các chỉ thị (directive) giúp tối ưu mã lệnh, tốc độ và kích thước chương trình. Hầu hết các trình biên dịch phổ biến hiện nay đều hỗ trợ tốt việc tối ưu mã khi biên dịch. Tuy nhiên, để đạt được hiệu quả tốt nhất, lập trình viên cần tập cho mình thói quen tối ưu mã lệnh ngay từ khi bắt tay viết những chương trình đầu tay. Bài viết này trình bày một số gợi ý rất cơ bản và kinh nghiệm thực tế tối ưu trong lập trình bằng ngôn ngữ C/C++.

Tinh giản các biểu thức toán học

Các biểu thức toán học phức tạp khi được biên dịch có thể sinh ra nhiều mã dư thừa làm tăng kích thước và chậm tốc độ thực hiện của chương trình. Do đó khi viết các biểu thức phức tạp lập trình viên cần nhớ một số đặc điểm cơ bản sau để giúp tinh giản biểu thức:

- CPU xử lý các phép tính cộng và trừ nhanh hơn các phép tính chia và nhân.

Ví dụ:

+ Biểu thức Total = (A*B + A*C + A*D) cần 2 phép cộng và 3 phép nhân. Ta có thể nhóm các phép cộng và viết thành Total = A*(B+C+D), tốc độ tính nhanh hơn vì giảm đi một phép tính nhân.

+ Biểu thức Total = (B/A + C/A) cần 2 phép chia có thể viết thành Total = (B+C)/A, giúp giảm đi một phép chia.

- CPU xử lý tính toán với các số nguyên (integer) chậm hơn với số thực (float, double), và tốc độ xử lý float nhanh hơn double.

- Trong một số trường hợp nhân hoặc chia số nguyên, sử dụng toán tử dời bit (bit shifting) sẽ nhanh hơn toán tử nhân chia.

Ví dụ:

Biểu thức (A *= 128) có thể tận dụng toán tử dời bit sang trái thành (A <<= 7).

Một số trình biên dịch có khả năng tối ưu mã khi biên dịch như Visual C++ 6 hoặc .Net 2003, biểu thức (A *= 128) và (A <<= 7) đều được biên dịch thành:

```
mov eax, A

shl eax, 7 (toán tử shl được dùng thay vì mul/imul)

mov A, eax
```

Ta có thể tối ưu bằng cách sử dụng mã assembly trực tiếp trong mã C/C++ như sau (xem thêm thủ thuật tận dụng thế mạnh của C/C++ bên dưới):

```
__asm shl A, 7;
```

Tối ưu việc sử dụng biến tạm

Đối với một số biểu thức tính toán số học phức tạp, trình biên dịch thường tạo các biến tạm trong bộ nhớ để chứa kết quả tính toán và cuối cùng mới gán giá trị này cho biến kết quả. Việc sử dụng biến tạm làm giảm tốc độ tính toán do phải cấp phát vùng nhớ, tính toán và thực hiện việc gán kết quả cuối cùng. Để tránh việc sử dụng biến tạm, ta có thể thực hiện việc tách các biểu thức phức tạp thành các biểu thức nhỏ hơn, hoặc sử dụng các mẹo cho việc tính toán.

Xem một ví dụ cộng các số nguyên sau:

```
A = B + C
```

Về cơ bản, khi thực hiện biểu thức này trình biên dịch tạo một biến tạm rồi thực hiện cộng 2 giá trị B, C vào biến tạm này, cuối cùng sẽ gán kết quả cho A.

Ta có thể viết lại biểu thức trên như sau để tránh sử dụng biến tạm làm chậm việc tính toán:

```
A = B;

A += C;
```

Trong lập trình hướng đối tượng, theo thói quen đôi khi lập trình viên sử dụng các biến tạm không cần thiết như trong ví dụ sau:

```
int MyFunc(const MyClass &A)

{

MyClass B;

B = A;

return B.value;

}
```

Trong hàm trên, khi biến tạm B kiểu MyClass được khởi tạo thì constructor mặc định sẽ được thực hiện. Sau đó B được gán giá trị của biến A thông qua việc sử dụng toán tử =, khi đó copy constructor sẽ được gọi. Tuy nhiên với yêu cầu của bài toán thì việc này không cần thiết, ta có thể viết lại như sau:

```
int MyFunc(const MyClass &A)

{

return A.value;

}
```

Dưới đây là một ví dụ khác cho bài toán hoán vị giá trị 2 số nguyên A và B. Thông thường, yêu cầu này sẽ được viết như sau:

```
int A = 7, B = 8;

int nTemp; //biến tạm

nTemp = A;

A = B;

B = nTemp;
```

Tuy nhiên, bạn có thể sử dụng mẹo sau để tránh sử dụng biến tạm và tăng tốc tính toán:

+ Sử dụng toán tử XOR:

```
A = A^B;

B = A^B;

A = A^B;
```

+ Sử dụng phép cộng, trừ

```
nX = nX + nY;

nY = nX - nY;

nX = nX - nY;
```

Bạn hãy chạy thử đoạn mã lệnh trên sẽ thấy điều bất ngờ thú vị khi bài toán hoán vị được giải quyết hết sức đơn giản.

Thủ thuật tránh sử dụng biến tạm cần áp dụng linh động tùy thuộc kiểu dữ liệu, đặc biệt các loại dữ liệu phức tạp như kiểu structure, string... có cơ chế lưu trữ và xử lý riêng. Đối với các trình biên dịch hiện đại, việc tối ưu theo cách này đôi khi không cần thiết vì trình biên dịch đã hỗ trợ sẵn cơ chế tối ưu này khi biên dịch mã lệnh.

Tối ưu các biểu thức điều kiện và luận lý

Biểu thức điều kiện là thành phần không thể thiếu ở hầu hết các chương trình máy tính vì nó giúp lập trình viên biểu diễn và xử lý được các trạng thái của thế giới thực dưới dạng các mã lệnh máy tính. Những điều kiện dư thừa có thể làm chậm việc tính toán và gia tăng kích thước mã lệnh, thậm chí có những đoạn mã có xác suất xảy ra rất thấp. Một trong những tiêu chí quan trọng của việc tối ưu các biểu thức điều kiện là đưa các điều kiện có xác suất xảy ra cao nhất, tính toán nhanh nhất lên đầu biểu thức.

Đối với các biểu thức luận lý, ta có thể linh động chuyển các biểu thức điều kiện đơn giản và xác suất xảy ra cao hơn lên trước, các điều kiện kiểm tra phức tạp ra sau.

Ví dụ: Biểu thức logic ((A || B) && C) có thể viết thành (C && (A || B)) vì điều kiện C chỉ cần một phép kiểm tra TRUE, trong khi điều kiện (A || B) cần đến 2 phép kiểm tra TRUE và một phép OR (||). Như vậy trong trường hợp C có giá trị FALSE, biểu thức logic này sẽ có kết quả FALSE và không cần kiểm tra thêm giá trị (A || B).

Đối với các biểu thức kiểm tra điều kiện phức tạp, ta có thể viết đảo ngược bằng cách kiểm tra các giá trị cho kết quả không thoả trước, giúp tăng tốc độ kiểm tra.

Ví dụ: Kiểm tra một giá trị thuộc một miền giá trị cho trước.

```
if (p <= max && p >= min && q <= max && q >= min)

{

//thực hiện khi thoả miền giá trị

}

else //không thoả

{

//thực hiện khi không thoả miền giá trị

}
```

Có thể viết thành:

```
if (p > max || p < min || q > max || q < min)

{

}

else

{

}
```

Tránh các tính toán lặp lại trong biểu thức điều kiện

Ví dụ:

```
if ((mydata->MyFunc() ) < min)

{

// ...

}

else if ((mydata->MyFunc() ) > max)

{

// ...

}
```

Ta có thể chuyển hàm MyFunc ra ngoài biểu thức điều kiệu như sau:

```
int temp_value = mydata->MyFunc();

if (temp_value < min)

{

// ...

}

else if (temp_value > max)

{

// ...

}
```

Đối với biểu thức điều kiện dạng switch...case: nếu các giá trị cho case liên tục nhau, trình biên dịch sẽ tạo ra bảng ánh xạ (còn gọi là jump table) giúp việc truy xuất đến từng điều kiện nhanh hơn và giảm kích thước mã lệnh. Tuy nhiên khi các giá trị không liên tục, trình biên dịch sẽ tạo một chuỗi các phép toán so sánh, từ đó gây chậm việc xử lý:

Ví dụ sau cho kết quả truy xuất tối ưu khi sử dụng switch...case:

```
switch (my_value)
{
case A:
...
break;
case B:
...
break;
case C:
...
break;
case D:
...
default:
```

```
...
}
```

Trong trường hợp các giá trị dùng cho case không liên tục, ta có thể viết thành các biểu thức if...elseif...else như sau:

```
switch (my_value)
{
case A:
...
break;
case F:
...
break;
case T:
}

Có thể viết thành:
if (my_value == A)
{
// xử lý cho trường hợp A
}
else if (my_value == F)
{
// xử lý cho trường hợp F
}
else
{
// các trường hợp khác
}
```

Ø Tối ưu vòng lặp

Vòng lặp cũng là một thành phần cơ bản phản ánh khả năng tính toán không mệt mỏi của máy tính. Tuy nhiên, việc sử dụng máy móc vòng lặp là một trong những nguyên nhân làm giảm tốc độ thực hiện của chương trình. Một số thủ thuật sau sẽ giúp lập trình viên tăng tốc vòng lặp của mình:

- Đối với các vòng lặp có số lần lặp nhỏ, ta có thể viết lại các biểu thức tính toán mà không cần dùng vòng lặp. Nhờ vậy tiết kiệm được khoảng thời gian quản lý và tăng biến đếm trong vòng lặp.

Ví dụ cho vòng lặp sau:

```
for( int i = 0; i < 4; i++ )

{

array[i] =MyFunc(i);

}

có thể viết lại thành:

array[0] = MyFunc(0);

array[1] = MyFunc(1);
```

```
array[2] = MyFunc(2);

array[3] = MyFunc(3);
```

- Đối với các vòng lặp phức tạp có số lần lặp lớn, cần hạn chế việc cấp phát các biến nội bộ và các phép tính lặp đi lặp lại bên trong vòng lặp mà không liên quan đến biến đếm lặp.

Ví dụ cho vòng lặp sau:

```
int students_number = 10000;

for( int i = 0; i < students_number; i++ )

{

//hàm MyFunc mất nhiều thời gian thực hiện

double sample_value = MyFunc(students_number);

CalcStudentFunc(i, sample_value);

}
```

Trong ví dụ trên, biến sample_value được tính ở mỗi vòng lặp một cách không cần thiết vì hàm MyFunc có thể tốn rất nhiều thời gian, ta có thể dời đoạn mã tính toán này ra ngoài vòng lặp như sau:

```
int students_number = 10000;

double sample_value = MyFunc(students_number);

for( int i = 0; i < students_number; i++ )

{

CalcStudentFunc(i, sample_value);

}
```

- Đối với vòng lặp từ 0 đến n phần tử như sau:

```
for( int i = 0; i < max_number; i++ )
```

Nên thực hiện việc lặp từ giá trị max_number trở về 0 như sau:

```
for( int i = max_number - 1; i >=0 ; -- i )
```

Vì khi biên dịch thành mã máy, các phép so sánh với 0 (zero) sẽ được thực hiện nhanh hơn với các số nguyên khác. Do đó phép so sánh ở mỗi vòng lặp là (i >=0) sẽ nhanh hơn phép so sánh (i < max_number).

- Trong vòng lặp lớn, các toán tử prefix dạng (--i hoặc ++i) sẽ thực hiện nhanh hơn toán tử postfix (i-- hoặc i++). Nguyên nhân là do toán tử prefix tăng giá trị của biến

trước sau đó trả kết quả về cho biểu thức, trong khi toán tử postfix phải lưu giá trị cũ của biến vào một biến tạm, tăng giá trị của biến và trả về giá trị của biến tạm.

Tối ưu việc sử dụng bộ nhớ và con trỏ

Con trỏ (pointer) có thể được gọi là một trong những "niềm tự hào" của C/C++, tuy nhiên thực tế nó cũng là nguyên nhân làm đau đầu cho các lập trình viên, vì hầu hết các trường hợp sụp đổ hệ thống, hết bộ nhớ, vi phạm vùng nhớ... đều xuất phát từ việc sử dụng con trỏ không hợp lý.

- Hạn chế pointer dereference: pointer dereference là thao tác gán địa chỉ vùng nhớ dữ liệu cho một con trỏ. Các thao tác dereference tốn nhiều thời gian và có thể gây hậu quả nghiêm trọng nếu vùng nhớ đích chưa được cấp phát.

Ví dụ với đoạn mã sau:

```
for( int i = 0; i < max_number; i++ )

{

SchoolData->ClassData->StudentData->Array[i] = my_value;

}
```

Bằng cách di chuyển các pointer dereference nhiều cấp ra ngoài vòng lặp, đoạn mã trên có thể viết lại như sau:

```
unsigned long *Temp_Array = SchoolData->ClassData->StudentData-
>Array;

for( int i = 0; i < max_number; i++ )

{

Temp_Array[i] = my_value;

}
```

- Sử dụng tham chiếu (reference) cho đối tượng dữ liệu phức tạp trong các tham số hàm. Việc sử dụng tham chiếu khi truyền nhận dữ liệu ở các hàm có thể giúp tăng tốc đáng kể đối với các cấu trúc dữ liệu phức tạp. Trong lập trình hướng đối tượng, khi một đối tượng truyền vào tham số dạng giá trị thì toàn bộ nội dung của đối tượng đó sẽ được sao chép bằng copy constructor thành một bản khác khi truyền vào hàm. Nếu truyền dạng tham chiếu thì loại trừ được việc sao chép này. Một điểm cần lưu ý khi sử dụng tham chiếu là giá trị của đối tượng có thể được thay đổi bên trong hàm gọi, do đó lập trình viên cần sử dụng thêm từ khóa const khi không muốn nội dung đối tượng bị thay đổi.

Ví dụ: Khi truyền đối tượng dạng giá trị vào hàm để sử dụng, copy constructor sẽ được gọi.

```
void MyFunc(MyClass A) //copy constructor của A sẽ được gọi

{
```

```
int value = A.value;

}
```

Khi dùng dạng tham chiếu, đoạn mã trên có thể viết thành:

```
void MyFunc(const MyClass &A) //không gọi copy constructor

{

int value = A.value;

}
```

- Tránh phân mảnh vùng nhớ: Tương tự như việc truy xuất dữ liệu trên đĩa, hiệu năng truy xuất các dữ liệu trên vùng nhớ động sẽ giảm đi khi bộ nhớ bị phân mảnh. Một số gợi ý sau sẽ giúp giảm việc phân mảnh bộ nhớ.

+ Tận dụng bộ nhớ tĩnh. Ví dụ: như tốc độ truy xuất vào một mảng tĩnh có tốc độ nhanh hơn truy xuất vào một danh sách liên kết động.

+ Khi cần sử dụng bộ nhớ động, tránh cấp phát hoặc giải phóng những vùng nhớ kích thước nhỏ. Ví dụ như ta có thể tận dụng xin cấp phát một mảng các đối tượng thay vì từng đối tượng riêng lẻ.

+ Sử dụng STL container cho các đối tượng hoặc các cơ chế sử dụng bộ nhớ riêng có khả năng tối ưu việc cấp phát bộ nhớ. STL cung cấp rất nhiều thuật toán và loại dữ liệu cơ bản giúp tận dụng tối đa hiệu năng của C++. Các bạn có thể tìm đọc các sách về STL sẽ biết thêm nhiều điều thú vị.

- Sau khi cấp phát một mảng các đối tượng, tránh nhầm lẫn khi sử dụng toán tử delete[] và delete: với C++, toán tử delete[] sẽ chỉ định trình biên dịch xóa một chuỗi các vùng nhớ, trong khi delete chỉ xóa vùng nhớ mà con trỏ chỉ đến, do đó có thể gây hiện tượng "rác" và phân mảnh bộ nhớ.

Ví dụ:

```
int *myarray = new int[50];

delete []myarray;
```

Với delete[], trình biên dịch sẽ phát sinh mã như sau:

```
mov ecx, dword ptr [myarray]

mov dword ptr [ebp-6Ch], ecx

mov edx, dword ptr [ebp-6Ch]

push edx

call operator delete[] (495F10h) //gọi toán tử delete[]

add esp,4
```

Trong khi với đoạn lệnh:

```
int *myarray = new int[50];

delete myarray;
```

Trình biên dịch sẽ phát sinh mã như sau:

```
mov ecx, dword ptr [myarray]

mov dword ptr [ebp-6Ch], ecx

mov edx, dword ptr [ebp-6Ch]

push edx

call operator delete (495F10h) //gọi toán tử delete

add esp,4
```

Sử dụng hợp lý cơ chế bẫy lỗi try...catch

Việc sử dụng không hợp lý các bẫy lỗi có thể là sai lầm tai hại vì trình biên dịch sẽ thêm các mã lệnh kiểm tra ở các đoạn mã được cài đặt try...catch, điều này làm tăng kích thước và giảm tốc độ xử lý của chương trình, đồng thời gây khó khăn trong việc sửa chữa các lỗi logic. Thống kê cho thấy các đoạn mã có sử dụng bẫy lỗi thì hiệu xuất thực hiện giảm từ 5%-10% so với đoạn mã thông thường được viết cẩn thận. Để hạn chế điều này, lập trình viên chỉ nên đặt bẫy lỗi ở những đoạn mã có nguy cơ lỗi cao và khả năng dự báo trước thấp.

Tận dụng đặc tính xử lý của CPU

Để đảm báo tốc độ truy xuất tối ưu, các bộ vi xử lý (CPU) 32-bit hiện nay yêu cầu dữ liệu sắp xếp và tính toán trên bộ nhớ theo từng offset 4-byte. Yêu cầu này gọi là memory alignment. Do vậy khi biên dịch một đối tượng dữ liệu có kích thước dưới 4-byte, các trình biên dịch sẽ bổ sung thêm các byte trống để đảm bảo các dữ liệu được sắp xếp theo đúng quy luật. Việc bổ sung này có thể làm tăng đáng kể kích thước dữ liệu, đặc biệt đối với các cấu trúc dữ liệu như structure, class...

Xem ví dụ sau:

```
class Test

{

bool a;

int c;

int d;

bool b;

};
```

Theo nguyên tắc alignment 4-byte (hai biến "c" và "d" có kích thước 4 byte), các biến "a" và "b" chỉ chiếm 1 byte và sau các biến này là biến int chiếm 4 byte, do đó trình biên dịch sẽ bổ sung 3 byte cho mỗi biến này. Kết quả tính kích thước của lớp Test bằng hàm sizeof(Test) sẽ là 16 byte.

Ta có thể sắp xếp lại các biến thành viên của lớp Test như sau theo chiều giảm dần kích thước:

```
class Test

{

int c;

int d;

bool a;

bool b;

};
```

Khi đó, hai biến "a" và "b" chiếm 2 byte, trình biên dịch chỉ cần bổ sung thêm 2 byte sau biến "b" để đảm bảo tính sắp xếp 4-byte. Kết quả tính kích thước sau khi sắp xếp lại class Test sẽ là 12 byte.

Tận dụng một số ưu điểm khác của C++

- Khi thiết kế các lớp (class) hướng đối tượng, ta có thể sử dụng các phương thức "inline" để thực hiện các xử lý đơn giản và cần tốc độ nhanh. Theo thống kê, các phương thức inline thực hiện nhanh hơn khoảng 5-10 lần so với phương thức được cài đặt thông thường.

- Sử dụng ngôn ngữ cấp thấp assembly: một trong những ưu điểm của ngôn ngữ C/C++ là khả năng cho phép lập trình viên chèn các mã lệnh hợp ngữ vào mã nguồn C/C++ thông qua từ khóa __asm { ... }. Lợi thế này giúp tăng tốc đáng kể khi biên dịch và khi chạy chương trình.

Ví dụ:

```
int a, b, c, d, e;

e = a*b + a*c;
```

Trình biên dịch phát sinh mã hợp ngữ như sau:

```
mov eax, dword ptr [a]

imul eax, dword ptr [b]

mov ecx, dword ptr [a]

imul ecx, dword ptr [c]
```

```
add eax, ecx

mov dword ptr [e], eax
```

Tuy nhiên, ta có thể viết rút gọn giảm được 1 phép imul (nhân), 1 phép mov (di chuyển, sao chép):

```
__asm

{

mov eax, b;

add eax, c;

imul eax, a;

mov e, eax;

};
```

- Ngôn ngữ C++ cho phép sử dụng từ khóa "register" khi khai báo biến để lưu trữ dữ liệu của biến trong thanh ghi, giúp tăng tốc độ tính toán vì truy xuất dữ liệu trong thanh ghi luôn nhanh hơn truy xuất trong bộ nhớ.

Ví dụ:

```
for (register int i; i <max_number; ++i )

{

// xử lý trong vòng lặp

}
```

- Ngôn ngữ C/C++ hỗ trợ các collection rất mạnh về tốc độ truy xuất như các bảng map, hash_map,... nên tận dụng việc sử dụng các kiểu dữ liệu này thay cho các danh sách liên kết bộ nhớ động (linked list).

Kết luận

Một chương trình được đánh giá tốt khi tất cả các bộ phận tham gia vào hoạt động của chương trình đạt hiệu suất cao nhất theo yêu cầu của người sử dụng. Một dòng lệnh đơn giản tưởng chừng sẽ hoạt động trong tích tắc có thể làm hệ thống trở nên chậm chạp khi được gọi hàng ngàn, hàng triệu lần trong khoảng thời gian ngắn. Do vậy, trong suốt qui trình hình thành sản phẩm phần mềm, giai đoạn cài đặt mã lệnh chiếm vai trò hết sức quan trọng và cần kĩ năng tối ưu hóa cao nhất. Để đạt được điều đó, không cách nào khác hơn là lập trình viên cần tự rèn luyện thật nhiều để thông thạo ngôn ngữ mình chọn lựa, trình biên dịch mình sử dụng. Khi đó lập trình không còn là việc tạo những đoạn mã khô khan, mà là một nghệ thuật.

Nguyễn Văn Sơn

Global CyberSoft Vietnam
sonnv@cybersoft-vn.com