

Técnicas básicas de otimização de código em C

Técnicas de otimização

Regras gerais

- Seleção dos melhores algoritmos
- Uso de bibliotecas eficientes (não esquecendo da portabilidade)
- Escolha da melhor estrutura de dados
- Programar com boas práticas de programação (otimizações manuais)
- Usar otimizações do compilador SEMPRE!!!

Otimização manual

- O utilizador tem maior controle sobre as dependências dos dados
- Pode reduzir a legibilidade do programa
- Algumas otimizações podem ser benéficas nalgumas arquiteturas e maléficas noutras

Otimização automática

- O compilador possui um conhecimento limitado das dependências dos dados
- Praticamente impossível em C se forem utilizados ponteiros indiscriminadamente
- Não altera a legibilidade do código
- Resultados ótimos de acordo com a arquitetura e a habilidade do compilador
- Pode aumentar consideravelmente o tempo de compilação

Otimização na prática

- Realizar otimizações manuais básicas e deixar outras por conta do compilador
- Utilizar otimizações avançadas somente para as arquiteturas predominantes

Otimização manual

- Forma mais trabalhosa porém a mais efetiva
 - o programador deve ter controle sobre o que deve ser otimizado
- Baseia-se em
 - utilizar boas práticas de programação durante a fase de concepção do software
 - colocar em prática o ciclo de desempenho:



Técnicas Básicas de Otimização

Reduções

Original

$2 * k$ (k é um valor inteiro)

Melhorado

$k + k$

Original

$2 * f$ (f é um real em ponto flutuante)

Alternativa (modificação sem benefícios)

$f + f$

a adição e a multiplicação de ponto flutuante consomem o mesmo número de ciclos

Original

$\text{pow}(x, 2.0)$

Melhorado

$x * x$

Eliminação de sub-expressões

Original

s1 = a + b + c;

s2 = a + b - c;

a ordem das operações inibe as otimizações por parte do compilador

Melhorado

no mínimo substituir por:

s1 = (a + b) + c;

s2 = (a + b) - c;

ou ainda melhor:

t = a + b;

s1 = t + c;

s2 = t - c;

Eliminação de sub-expressões

Original

```
r = 2.0 * X[k];
```

```
s = X[k] - 1;
```

Melhorado

```
t = X[k];
```

```
r = 2.0 * t;
```

```
s = t - 1;
```

Eliminação de sub-expressões

Original

```
x = acos(-1.0) + pow(acos(-1.0), 2);
```

Melhorado

```
t = acos(-1.0);
```

```
x = t + pow(t, 2);
```

ou ainda melhor:

```
t = acos(-1.0);
```

```
x = t + t * t;
```


Expressões constantes em ciclos

Original

```
for (k = 0; k < N; k++)
```

```
  A[k] = r * A[k] * s;
```

ordem das operações inibindo as otimizações por parte do compilador

Melhorado

no mínimo, substituir por:

```
for (k = 0; k < N; k++)
```

```
  A[k] = r * s * A[k];
```

ou ainda melhor:

```
t = r * s;
```

```
for (k = 0; k < N; k++)
```

```
  A[k] = t * A[k];
```

Conversão de sinais

- A conversão de sinais consome alguns ciclos de processamento
- A omissão das conversões, em operações de ponto flutuante, podem alterar o resultado das operações,
 - ou seja, o compilador não tenta fazer tais transformações em níveis baixos de otimização

Original

```
A[k] = -(1.0 - 0.50 * B[k]);
```

Melhorado

```
A[k] = -1.0 + 0.50 * B[k];
```

Propagação e avaliação de constantes

Original

```
t = 2.0;
```

```
x = 3.0 * t * y;
```

Melhorado

```
x = 6.0 * y;
```

Simplificação no cálculo de endereços

Original

```
float A[4][100]; // array de 2 dimensões com 4 linhas e 100 colunas
for (k = 0; k < 100; k++)
    x = x + A[0][k];
```

O endereço de $A[0][k]$ em relação ao elemento $A[0][0]$ é $4 * (k+1) - 4$

Melhorado (transformar um array de 2 dimensões num array de 1 dimensão)

```
float A[4*100]; // array de 1 dimensão com 400 elementos
indice = 0;
for (k = 0; k < 100; k++)
{
    x = x + A[indice];
    indice = indice + 4;
}
```

Aqui o endereço é calculado por uma simples adição

In-lining

- Evocação (chamada) de um subprograma (função) dentro de um ciclo

```
...  
float dist (float x, float y)  
{  
    float dist;  
    dist = sqrt(x*x + y*y);  
    return dist;  
}  
...  
for (k = 0; k < max; k++)  
    R[k] = dist(X[k], Y[k]);  
...
```

In-lining

- A técnica de “in-lining” consiste em substituir as chamadas de subprogramas escrevendo-as explicitamente onde elas são chamadas

```
...  
for (k = 0; k < max; k++)  
    R[k] = sqrt(X[k]*X[k], Y[k]*Y[k]);  
...
```

- Os compiladores possuem **flags** específicos para realizar esse tipo de substituição baseado nalguns critérios
- No entanto, quando efetuamos essa substituição explicitamente, facilitamos o trabalho do compilador