Chapter 1

Otimização de Código: Peephole e Local

1.1 Objetivos

⋆ title=Objetivos da Aula

Ao final deste capítulo, o estudante deverá ser capaz de:

- Compreender a **filosofia** e os **limites** das Otimizações de Código, incluindo os três critérios que toda otimização deve satisfazer.
- Entender o conceito de *Peephole Optimization* e aplicar seus padrões (Load/Store, Jump Chaining, identidades algébricas).
- Aplicar *Strength Reduction* e reconhecer casos onde ela é **insegura** (ponto flutuante, inteiros negativos).
- Identificar e construir **Blocos Básicos** e o **Grafo de Fluxo de Controle (CFG)**.
- Representar um Bloco Básico como **DAG** e utilizá-lo para identificar CSEs, código morto e oportunidades de folding.
- Aplicar *Constant Folding*, *Constant Propagation*, CSE local e DCE em cascata.
- Reconhecer quando e por que estas técnicas **falham ou não se aplicam**.

1.2 O Compilador Como Engenheiro de Performance

O programador descreve **o quê** o programa faz. O compilador decide **como** isso será executado na máquina. Um bom compilador realiza duas tarefas inseparáveis:

- **Correção:** O código gerado deve ser semanticamente equivalente ao original.
- **Eficiência:** O código gerado deve usar o mínimo de recursos (ciclos de CPU, registradores, memória).

Por que otimizar no compilador? Código fonte é escrito para ser lido por humanos, não máquinas. Um loop pode gerar dezenas de instruções redundantes, e o hardware moderno é complexo demais para que um programador o otimize manualmente.

A Regra de Ouro: Preservação Semântica

> *"A otimização prematura é a raiz de todos os males na programação."* – Donald Knuth

Mas, no nível do compilador, otimizar é vital. A regra intransponível: **o programa otimizado deve ter exatamente o mesmo comportamento observável que o original** (para entradas válidas). Não importa quão drástica seja a transformação.

1.3 O Escopo das Otimizações

As otimizações são separadas pelo seu raio de visão (escopo) para manter o tempo de compilação factível:

1. **Peephole (Visor):** Analisa apenas 2 a 5 instruções vizinhas sequenciais. Ocorre no Back-End, sobre o assembly gerado.
2. **Local:** Analisa um **Bloco Básico** isolado por vez. Opera sobre o IR.
3. **Global:** Analisa todos os blocos de uma única função integrados via **Grafo de Fluxo de Controle (CFG)**.
4. **Interprocedural:** Analisa o fluxo do programa transitando entre diferentes chamadas de função.

1.3.1 Critérios de uma Boa Otimização

Toda otimização aplicada por um compilador deve obedecer a três critérios:

1. **Segura (Safe):** Nunca altera o comportamento observável do programa.
2. **Lucrativa (Profitable):** O benefício em tempo/espaço supera o custo de compilação para encontrar e aplicar a otimização.

3 **Aplicável (Applicable):** As condições de guarda (pré-condições) da otimização são satisfeitas no trecho de código analisado.

△ Importante

Uma otimização que parece óbvia pode ser **insegura** em casos especiais: arredondamento de ponto flutuante (IEEE 754), ponteiros com *aliasing*, código multi-threaded, etc. O compilador deve verificar pré-condições antes de aplicar qualquer transformação.

1.4 Peephole Optimization (Otimização de Visor)

O nome *Peephole* ("olho mágico" ou visor) reflete a natureza do algoritmo: uma **janela deslizante** que expõe apenas 2 a 5 instruções contíguas do código gerado. Conforme a janela percorre o programa instrução por instrução, o compilador tenta aplicar **casamento de padrões** (*pattern matching*).

Propriedades fundamentais:

- A janela **não cruza** fronteiras de Bloco Básico.
- Cada padrão reconhecido é substituído por uma versão mais eficiente.
- A janela desliza e o processo se repete até que não haja mais melhorias.

1.4.1 Load/Store Redundantes

Frequente após a seleção crua de instruções, o código pode armazenar e reler o mesmo valor desnecessariamente:

Eliminação de Load Redundante

Antes:

```
store r1, [x]
load  [x], r2    ; redundante !
```

Após Peephole:

```
store r1, [x]
mov   r1, r2     ; apenas copia de reg
```

O `store` é mantido, pois `x` pode ser lida depois (por exemplo, em outro bloco). Apenas o `load` lento de memória é substituído por um `mov` de registrador (1 ciclo vs. dezenas de ciclos).

1.4.2 Jump Chaining (Saltos para Saltos)

Estruturas lógicas mal aninhadas podem gerar cadeias de saltos. Se a janela capta um desvio que aponta para um bloco que apenas contém outro desvio, ela encurta o salto inicial para o alvo final:

Eliminação de Jump Chain

Antes:

```
    jmp L1
    ...
L1: jmp L2
    ...
L2: <codigo>
```

Após Peephole:

```
    jmp L2      ; encurtado!
    ...
L1: jmp L2
    ...
L2: <codigo>
```

`L1` pode se tornar código morto e ser removido na passagem seguinte.

1.4.3 Código Morto Após Salto Incondicional

Instruções entre um `jmp` incondicional e o próximo rótulo são **inatingíveis** e podem ser removidas. Ocorre comumente após inlining de funções ou simplificação de *branches* condicionais:

Remoção de Código Inatingível

```
    jmp L1
    add r2, r3    ; NUNCA executa -- removido!
    mul r4, r5    ; NUNCA executa -- removido!
L1: <codigo>
```

1.4.4 Simplificações Algébricas

A janela *Peephole* aplica identidades matemáticas básicas para eliminar operações desnecessárias:

- **Identidades com elementos neutros:**
  - `x = x + 0` $\longrightarrow$ removido (*Nop*)
  - `x = x - 0` $\longrightarrow$ removido (*Nop*)
  - `x = x * 1` $\longrightarrow$ removido (*Nop*)
  - `x = x / 1` $\longrightarrow$ removido (*Nop*)
  - `x = x * 0` $\longrightarrow$ `x = 0`
  - `x = 0 - x` $\longrightarrow$ `x = -x` (*neg*)
- **Identidades booleanas e bit a bit:**
  - `x = x && true` $\longrightarrow$ `x`
  - `x = x XOR x` $\longrightarrow$ `x = 0`
  - `x = x AND 0` $\longrightarrow$ `x = 0`
  - `x = x OR -1` $\longrightarrow$ `x = -1`

1.4.5 Strength Reduction (Redução de Força)

Objetivo: Substituir operações de **alto custo** de hardware por operações **equivalentes mais baratas**. A ALU e a FPU possuem custos de ciclo drasticamente assimétricos:

Operação	Ciclos típicos	Observação
Shift («, »)	1	Mais barata disponível
Soma / Subtração	1–3	Rápida
Multiplicação	3–5	Moderada
Divisão inteira	20–40	Muito cara
Divisão float	10–25	Depende do hardware

Substituições clássicas:

- `x = y * 2` → `x = y « 1` (shift left)
- `x = y * 4` → `x = y « 2`
- `x = y / 2` → `x = y » 1` (para inteiros sem sinal)
- `x = y % 8` → `x = y & 7` (módulo por pot. de 2)
- `x = y^2` → `x = y * y` (evita `libm/pow()`)
- `x = y^3` → `t = y*y; x = t*y`

△ Importante

Ponto Flutuante – IEEE 754: `x / 2.0` **não** é equivalente a `x * 0.5` em todos os casos por causa de regras de arredondamento. A substituição só é segura com a flag `-ffast-math`.
Inteiros com Sinal – Divisão: Em C, `x » 1` para inteiros **negativos** com sinal pode não corresponder a `x / 2` por causa da semântica de arredondamento ("round toward zero" vs. "arithmetic shift"). O compilador precisa verificar o tipo antes de aplicar.

1.5 Blocos Básicos e Grafo de Fluxo de Controle

1.5.1 Definição de Bloco Básico

Bloco Básico

Um **Bloco Básico** é uma sequência **máxima** de instruções de código intermediário tal que:

1 O fluxo de controle entra apenas pelo **início** da sequência.

2 O fluxo de controle sai apenas pelo **fim** da sequência (possível salto).

Propriedade-chave: Se a primeira instrução de um bloco executa, **todas** as demais executam, na mesma ordem. Isso nos dá liberdade para reorganizar e substituir instruções dentro do bloco.

1.5.2 Algoritmo de Identificação de Blocos Básicos

O algoritmo opera em duas fases: identificar **líderes** (primeiras instruções de cada bloco) e depois delimitar os blocos.

Regras para identificar líderes:

1 A **primeira instrução** do programa é um líder.

2 Qualquer instrução que seja **alvo de um salto** (condicional ou não) é um líder.

3 Qualquer instrução **imediatamente após** um salto (condicional ou não) é um líder.

Cada líder inicia um novo Bloco Básico. O bloco vai do líder até a instrução **imediatamente anterior** ao próximo líder.

Identificando Blocos Básicos

Dado o código intermediário abaixo:

```
1  (1)   i = 1
2  (2)   j = 1
3  (3)   t1 = 10 * i
4  (4)   t2 = t1 + j
5  (5)   t3 = 8 * t2
6  (6)   t4 = t3 - 88
7  (7)   a[t4] = 0.0
8  (8)   j = j + 1
9  (9)   if j <= 10 goto (3)
10 (10)  i = i + 1
11 (11)  if i <= 10 goto (2)
12 (12)  i = 1
13 (13)  t5 = i - 1
14 (14)  t6 = 88 * t5
15 (15)  a[t6] = 1.0
16 (16)  i = i + 1
17 (17)  if i <= 10 goto (13)
18 (18)  halt
```

Líderes identificados: **(1)** (primeira instrução), **(2)** (alvo de 11), **(3)** (alvo de 9), **(10)** (após salto em 9), **(12)** (após salto em 11), **(13)** (alvo de 17), **(18)** (após salto em 17).
Blocos resultantes:

Bloco	Instruções
B_1	(1)
B_2	(2)
B_3	(3)–(9)
B_4	(10)–(11)
B_5	(12)
B_6	(13)–(17)
B_7	(18)

1.5.3 Grafo de Fluxo de Controle (CFG)

Control Flow Graph (CFG)

O **Grafo de Fluxo de Controle** é um grafo dirigido onde:

- Cada **nó** é um Bloco Básico.
- Existe uma **aresta** $B_i \to B_j$ se B_j pode ser executado imediatamente após B_i.

Há aresta $B_i \to B_j$ quando: B_j é alvo de um salto no fim de B_i, **ou** B_j segue imediatamente B_i no código e B_i não termina com salto incondicional.

O CFG é a base para todas as **otimizações globais** (análise de alcance de definições, análise de fluxo de dados, etc.) – tema das aulas seguintes.

1.6 Representação por DAG (Directed Acyclic Graph)

Dentro de um Bloco Básico (sem bifurcações), podemos representar as computações como um **DAG – Directed Acyclic Graph** (Grafo Acíclico Dirigido).

1.6.1 Estrutura do DAG

- **Folhas:** variáveis ou constantes (entradas do bloco).
- **Nós internos:** operadores (+, *, etc.).
- **Rótulos dos nós:** lista de variáveis que recebem aquele valor.
- **Arestas:** de um operador para seus operandos.

A grande vantagem: se a mesma subexpressão ocorre duas vezes no bloco, o DAG a **representa com um único nó** – revelando automaticamente as CSEs!

1.6.2 Algoritmo de Construção do DAG

Para cada instrução $x = y \ op \ z$:

1 Encontre (ou crie) o nó para y e para z.

2 Verifique se já existe um nó op com esses mesmos filhos.

3 Se sim: **reutilize** esse nó; se não: **crie** um novo nó op.

4 Remova x dos rótulos de outros nós; adicione x ao rótulo do nó resultado.

Construindo o DAG de um Bloco Básico

Bloco básico:

```
1 a = b + c
2 b = a - d
3 c = b + c
4 d = a - d
```

O nó (`-`, a, d_0) é instanciado uma única vez, mas recebe os rótulos b *e* d (pois `d = a - d` produz o mesmo resultado que `b = a - d`). O DAG expõe essa subexpressão comum automaticamente.

1.6.3 O que o DAG Revela

Com o DAG construído, o compilador pode automaticamente identificar:

1 **CSE (Common Subexpression Elimination):** Nós com mais de um rótulo (ou usados por mais de uma instrução) são subexpressões comuns – computadas apenas uma vez.

2 **Dead Code:** Nós cujos rótulos **não são variáveis vivas** na saída do bloco podem ser eliminados.

3 **Ordem de Avaliação:** O DAG sugere uma **ordem topológica** para reemitir o código, potencialmente melhor para o *scheduler* de instruções do processador.

4 **Constant Folding:** Quando ambos os operandos de um nó são constantes, o compilador pode calcular o resultado em tempo de compilação e criar um nó folha com o valor calculado.

1.7 Otimizações Locais em Blocos Básicos

Otimizações **locais** têm a visão de um Bloco Básico inteiro. Como não há bifurcações internas, a análise dentro do bloco é **exata** – sem fluxo alternativo. O compilador pode modelar as instruções como um DAG e aplicar as técnicas a seguir.

1.7.1 Constant Folding

Constant Folding

Constant Folding: avaliar em **tempo de compilação** qualquer expressão cujos operandos são todos constantes conhecidas.

Exemplo de Constant Folding

Antes:

```
x = 2 + 3;
y = x * 10;
z = 3.14159 * 2.0;
```

Após Constant Folding:

```
x = 5;        // compile-time!
y = 50;       // x era constante
z = 6.28318; // idem
```

Nenhum ciclo de CPU é gasto em tempo de execução para fazer essas contas.

1.7.2 Constant Propagation

Constant Propagation

Constant Propagation: substituir **usos** de uma variável pelo seu valor constante, quando o compilador pode garantir que a variável não mudou desde sua última atribuição.

Algoritmo simplificado:

1 Mantenha uma tabela `val[v]` para cada variável v no bloco.

2 Inicialize `val[v] = UNKNOWN` para todas as variáveis.

3 Para cada instrução $v = e$ no bloco (em ordem):

- Avalie e substituindo variáveis pelos seus valores em `val`.
- Se e se reduz a uma constante c: `val[v] = c` e substitua e por c.
- Caso contrário: `val[v] = NOTCONSTANT`.

Propagation e Folding trabalham em **sinergia**: a propagação de constantes gera novas expressões com operandos constantes, que por sua vez são foldadas:

Constant Propagation + Folding em Cascata

Original:

```
a = 2;
b = 3;
c = a + b;
d = c * x;
if (a > 0) ...
```

Após Propagation:

```
a = 2;
b = 3;
c = 2 + 3;
d = c * x;
if (2 > 0) ...
```

Após Folding:

```
a = 2;
b = 3;
c = 5;
d = 5 * x;
if (true) ...
```

O `if (true)` abre caminho para a **eliminação de branch**: o compilador pode remover o bloco condicional inteiro!

△ Importante

Limitação local: Se v pode receber valores de **arestas diferentes do CFG** (ex: ϕ-funções em SSA, ou uniões de fluxo), a análise local não é suficiente. Será necessária a **análise global de fluxo de dados**.

1.7.3 Common Subexpression Elimination Local (CSE)

CSE Local

Uma **subexpressão comum** é uma expressão que aparece mais de uma vez no bloco, com os mesmos operandos e sem modificação dos operandos entre as duas ocorrências. O compilador pode calcular a expressão uma única vez e reutilizar o resultado.

CSE Local

Com Redundância:

```
1 a = b + c;
2 d = a - 10;
3 e = b + c;    // redundante!
4 f = e * 2;
```

`b + c` avaliada duas vezes desnecessariamente.

Após CSE Local:

```
1 a = b + c;
2 d = a - 10;
3 e = a;        // reutiliza!
4 f = e * 2;
```

No DAG, o nó `+` é criado uma única vez; tanto `a` quanto `e` apontam para ele.

Quando NÃO aplicar CSE:

- Se uma variável que compõe a expressão é **modificada** entre as duas ocorrências, a expressão **não** é mais uma subexpressão comum:

```
1 a = b + c;
2 b = 99;          // b mudou!
3 e = b + c;       // valor diferente -- NAO e CSE!
```

- Funções com **efeitos colaterais** (E/S, acesso a globais) não podem participar de CSE sem análise adicional de pureza da função.

1.7.4 Dead Code Elimination Local (DCE)

Dead Code

Dead Code: código cujo resultado **nunca é utilizado** após o ponto de cálculo – nem dentro do bloco, nem na saída do bloco (variável não está viva).

DCE Local

Com variável morta:

```
a = b + c;
t = a * 2;   // t nunca usado !
d = a + 1;
```

Após DCE:

```
a = b + c;
d = a + 1;
```

Uma variável x é **ao vivo** (*live*) em um ponto p do programa se existe algum caminho de p até um **uso** de x sem uma redefinição de x no meio. Para DCE local, variáveis usadas fora do bloco (ou que não são temporárias do compilador) são conservadoramente consideradas ao vivo. A análise exata de vivacidade em todo o CFG é o tema da análise de fluxo de dados.

1.8 Exercício Resolvido: Otimizações em Cascata

Exercício – Aplicando todas as técnicas em cascata

Dado o bloco básico abaixo, aplique Constant Propagation, Constant Folding, CSE local e DCE. As variáveis vivas na saída são **apenas** `h` e `f`.

```
a = 4;
b = 2;
c = a + b;    // (1)
d = c * 3;    // (2)
e = a + b;    // (3)
f = d + e;    // (4)
g = c * 3;    // (5)
h = f + 10;
```

Passo 1 – Constant Propagation + Folding:
Substituímos `a` por 4 e `b` por 2 em todos os usos, e dobramos as expressões constantes:

```
a = 4;    b = 2;
c = 6;    // 4+2
d = 18;   // 6*3
e = 6;    // 4+2
f = 24;   // 18+6
g = 18;   // 6*3
h = 34;   // 24+10
```

Passo 2 – CSE local:

- `e = 6` é igual a `c` ⇒ `e = c`
- `g = 18` é igual a `d` ⇒ `g = d`

Passo 3 – DCE:

- `g` não é vivo na saída ⇒ eliminado
- `a, b, c, d, e` não são vivos na saída ⇒ eliminados

Resultado final: 8 instruções → **2 instruções**!

```
f = 24;
h = 34;
```

1.9 Panorama de Outras Otimizações

O que estudamos neste capítulo é apenas uma fatia. O compilador moderno realiza dezenas de otimizações em múltiplos escopos:

- **Otimizações Globais** (próximas aulas):

- *Global CSE* (via análise de alcance de definições)
- *Loop Invariant Code Motion* (LICM)
- *Induction Variable Elimination*
- *Global Constant Propagation* (via SSA)

- **Otimizações Avançadas:**
  - *Inlining* de funções
  - *Loop Unrolling*
  - *Vectorization* (SIMD)
  - *Tail Call Optimization*
  - *Partial Redundancy Elimination* (PRE)

1.9.1 Loop Invariant Code Motion (LICM) – Prévia

Uma das otimizações mais lucrativas: mover cálculos que **não mudam dentro de um loop** para fora dele:

LICM

Antes:

```
for (i = 0; i < n; i++) {
    x = a * b;    // invariante!
    arr[i] = x + i;
}
```

n multiplicações desnecessárias.

Após LICM:

```
x = a * b;    // movido para fora
for (i = 0; i < n; i++) {
    arr[i] = x + i;
}
```

Para $n = 10^6$: de 10^6 muls para **1**.

1.9.2 Níveis de Otimização no GCC/Clang

Os compiladores reais agrupam as otimizações em níveis ativados por flags:

Flag	O que ativa
`-O0`	Sem otimizações (debug). Cada instrução C vira instruções simples de IR.
`-O1`	Peephole, DCE básico, Constant Propagation, algumas CSEs locais.
`-O2`	LICM, Global CSE, inlining leve, otimização de loops.
`-O3`	Vetorização, loop unrolling, inlining agressivo, otimizações interprocedurais.
`-Os`	Otimiza para **tamanho** de código (variante de `-O2`).
`-Ofast`	`-O3` + quebra IEEE 754 (risco em cálculos de ponto flutuante).

Diferença entre -O0 e -O2 em Assembly (x86-64)

Função C:

```c
int soma(int a, int b) {
    int x = a + b;
    int y = a + b;  // CSE com x
    return x + y;
}
```

Com `-O0`: 14 instruções (acessa memória, recalcula `a+b` duas vezes).
Com `-O2`: apenas 3 instruções:

```
lea   eax, [rdi + rsi]    ; a + b
add   eax, eax            ; (a+b)*2 -- strength reduction de x+y
ret
```

O compilador aplicou: CSE (`a+b` uma vez), Strength Reduction (`x+y = 2*(a+b)` vira `add eax, eax`) e eliminação de acesso à memória.

1.10 Resumo

✓ title=Pontos-Chave

1. **Preservação Semântica** é inviolável. Toda otimização deve manter o comportamento observável do programa.
2. **Peephole** age localmente (2–5 instruções adjacentes): elimina redundâncias de load/store, encurta cadeias de saltos e aplica identidades algébricas.
3. **Strength Reduction** troca operações caras (mul, div, exp) por operações baratas (shift, add) – mas requer cuidado com tipos e semântica.
4. **Blocos Básicos** são a unidade de análise local: sem bifurcações internas, a análise é exata. O CFG conecta os blocos e habilita otimizações globais.
5. O **DAG** de um bloco básico é a estrutura que automaticamente revela CSEs, permite DCE e guia a reordenação de instruções.
6. **Constant Folding + Propagation + CSE + DCE** frequentemente se combinam em cascata, reduzindo drasticamente o número de instruções (no exercício: $8 \rightarrow 2$).

1.11 Referências

- **AHO, A. V.; LAM, M. S.; SETHI, R.; ULLMAN, J. D.** *Compiladores: Princípios, Técnicas e Ferramentas*. 2ª ed. Pearson, 2008. Cap. 8 e 9.

- **COOPER, K.; TORCZON, L.** *Engineering a Compiler*. 2ª ed. Morgan Kaufmann, 2011. Cap. 8 (Local Optimization).
- **APPEL, A. W.** *Modern Compiler Implementation in C*. Cambridge University Press, 1998. Cap. 17.