Compiladores

Aula 23: Otimização de Código

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Maio de 2026

Sumário

Objetivos da Aula

- Compreender a **filosofia** e os **limites** das Otimizações de Código.
- Entender o conceito de *Peephole Optimization* (Otimização de Visor).
- Explorar otimizações algébricas e *Strength Reduction*.
- Identificar e construir **Blocos Básicos** e o **Grafo de Fluxo de Controle**.
- Representar um Bloco Básico como **DAG** e utilizá-lo para otimizações.
- Aplicar *Constant Folding*, *Constant Propagation*, CSE e DCE locais.
- Reconhecer quando e por que estas técnicas **falham ou não se aplicam**.

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.
- O hardware moderno é complexo demais para um humano otimizar manualmente.

A Arte da Otimização

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

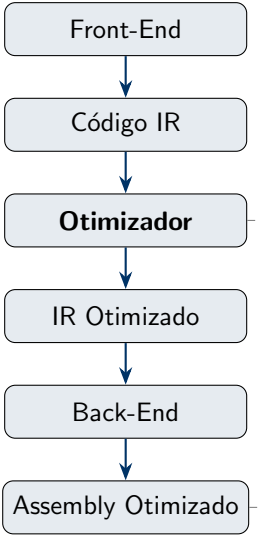
Mas, no nível do **Compilador**, otimizar é vital. O compilador deve gerar o código mais limpo e rápido possível, com uma **regra de ouro intransponível**:

Preservação Semântica

Não importa quão complexa seja a otimização, o programa otimizado deve ter **exatamente o mesmo comportamento observável** que o programa original (para entradas válidas).

Otimizações se dividem em três escopos: **Peephole** (instruções vizinhas), **Local** (Blocos Básicos) e **Global** (Grafos de Fluxo inteiros) – e ainda **Interprocedural** (múltiplas funções).

Onde o Compilador Otimiza?



- **Local**: dentro de um Bloco Básico
- **Global**: entre blocos (CFG)
- **Interprocedural**: entre funções

- **Peephole**: janela de 2–5 instruções
- **Alocação de registradores**

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.

Atenção

Uma otimização que parece óbvia pode ser **insegura** em casos especiais (arredondamento de ponto flutuante, ponteiros com aliasing, código multi-threaded, etc.).

O que é Peephole Optimization?

O compilador desliza uma pequena "janela" (*peephole*) sobre o código assembly ou IR, analisando **2 a 5 instruções adjacentes** por vez.

- 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.

Analogia

Imagine revisar um texto procurando **palavras redundantes** numa frase de cada vez – você não precisa re-ler o livro inteiro para remover "muito muito".

Peephole: Load/Store Redundantes

Antes da Otimização

```
1 store r1, [x]
2 load  [x], r2
```

Problema: `r2` será carregado com o valor que **acabou de ser salvo** por `r1`. O `load` é inútil – o valor já está em `r1`.

Após Otimização

```
1 store r1, [x]
2 mov   r1, r2
```

Um `mov` de registrador para registrador é muito mais rápido que um acesso à memória.

Atenção

O `store` é mantido, pois `x` pode ser lida depois (por exemplo, em outro bloco).

Peephole: Saltos para Saltos (Jump Chaining)

Antes da Otimização

```
1     jmp L1
2     ...
3 L1: jmp L2
4     ...
5 L2: <codigo>
```

O código salta para L1, que imediatamente salta para L2. Ciclos desperdiçados.

Após Otimização

```
1     jmp L2
2     ...
3 L1: jmp L2
4     ...
5 L2: <codigo>
```

O compilador encurta o caminho, indo direto ao alvo final. L1 pode se tornar código morto e ser removido.

Peephole: Código Morto Após Salto Incondicional

Antes

```
1     jmp L1
2     add r2, r3    ; nunca executa!
3     mul r4, r5    ; nunca executa!
4 L1: <codigo>
```

Instrucoes entre um `jmp` incondicional e o próximo rótulo são **inatingíveis** e podem ser removidas.

Após Otimização

```
1     jmp L1
2 L1: <codigo>
```

Quando isso ocorre?

Comum após inlining de funções, simplificação de *branches* condicionais ou geração de código com `return` seguido de mais instruções.

Simplificações Algébricas

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

Identidades com 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 / Bit a Bit

- `x = x && true` $\longrightarrow$ `x`
- `x = x || false` $\longrightarrow$ `x`
- `x = x XOR x` $\longrightarrow$ `x = 0`
- `x = x AND 0` $\longrightarrow$ `x = 0`
- `x = x OR -1` $\longrightarrow$ `x = -1`

Strength Reduction (Redução de Força)

Objetivo: Substituir operações de **alto custo** de hardware por operações **equivalentes mais baratas**.

Multiplicação por potência de 2

- `x = y * 2` → `x = y << 1`
- `x = y * 4` → `x = y << 2`
- `x = y * 8` → `x = y << 3`
- `x = y / 2` → `x = y >> 1`
- `x = y % 8` → `x = y & 7`

Exponenciação

- `x = y^2` → `x = y * y` *(evita libm)*
- `x = y^3` → `t=y*y; x=t*y`

Custo típico em ciclos (hardware)

Operação	Ciclos
Shift	1
Soma/Sub	1–3
Mul	3–5
Div	20–40

Strength Reduction: Cuidados

Ponto Flutuante – Cuidado!

`x / 2.0` **não** é equivalente a `x * 0.5` em todos os casos de ponto flutuante por causa de regras de arredondamento IEEE 754. A substituição só é segura se o compilador tiver a flag `-ffast-math` (ou equivalente).

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.

Revisão: Bloco Básico

Definição Formal

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!

Como Identificar Blocos Básicos?

Algoritmo de Identificação de 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.

Exemplo: Identificando Líderes

```
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
```

Identificando Líderes:

- **(1):** Primeira instrução → líder
- **(2):** Alvo do salto em (11) → líder
- **(3):** Alvo do salto em (9) → líder
- **(10):** Após o salto em (9) → líder
- **(12):** Após o salto em (11) → líder
- **(13):** Alvo do salto em (17) → líder
- **(18):** Após o salto em (17) → líder

Blocos Resultantes:

Bloco	Instruções
B_1	(1)
B_2	(2)
B_3	(3)–(9)
B_4	(10)–(11)
B_5	(12)

Grafo de Fluxo de Controle (CFG)

Definição

O **Grafo de Fluxo de Controle** (Control Flow Graph – CFG) é um grafo dirigido onde:

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

Quando há aresta $B_i \rightarrow B_j$?

- B_j é o **alvo de um salto** (condicional ou não) que está no fim de B_i.
- B_j **segue imediatamente** B_i no código, e B_i não termina com um salto incondicional.

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

DAG de um Bloco Básico

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

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!

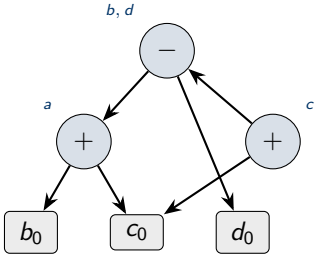
Construindo um DAG: Passo a Passo

Código do Bloco Básico

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

Regras de Construção

1. Para cada instrução $x = y\ op\ z$:
   - Encontre (ou crie) o nó para y e z.
   - Verifique se já existe um nó op com esses filhos.
   - Se sim: **reutilize** esse nó; se não: **crie** um novo.
   - Remova x dos rótulos de outros nós; adicione x ao rótulo do nó resultado.



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.

Otimizações Locais (Em Blocos Básicos)

Otimizações **locais** têm a visão de um **Bloco Básico** inteiro.

Revisão: Por que Blocos Básicos são seguros?

Uma sequência de instruções de código linear, possuindo uma única entrada (primeira instrução) e uma única saída (saltos no final). O código executa do início ao fim sem interrupções. Por isso, a análise dentro do bloco é exata – sem fluxo alternativo.

- Como não há bifurcações, podemos modelar as instruções como um **DAG**.
- Isso permite otimizações como:
  - *Constant Folding* e *Constant Propagation*.
  - *Common Subexpression Elimination* (CSE) local.
  - *Dead Code Elimination* (DCE) de variáveis temporárias.

Constant Folding

Definição

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

Antes

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

Após Constant Folding

```
1 x = 5;      // calculado em compile-time
2 y = 50;     // x era constante => fold
3 z = 6.28318; // idem
```

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

Constant Propagation

Definição

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

Código Original

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

Propagação + Folding

```
1 a = 2;
2 b = 3;
3 c = 5;       // 2+3 foldado
4 d = 5 * x;   // c propagado
5 if (true) ... // a>0 => 2>0
```

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

Constant Propagation: 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`.

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** (Aula 25).

Local Common Subexpression Elimination (CSE)

Código com Redundância

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

- `b + c` é avaliada **duas vezes**.
- Como `b` e `c` não mudam entre as duas avaliações, o resultado será idêntico.

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 e tanto a quanto e apontam para ele. Ao reemitir o código, o compilador vê que e não precisa recalcular nada.

CSE: Quando NÃO aplicar

Invalidade de CSE

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

Não é CSE!

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

Aqui `b + c` na linha 1 usa o valor original de `b`, e na linha 3 usa `b = 99`. São valores distintos.

E com chamadas de função?

```
a = f() + c;
// f() pode ter efeitos colaterais
e = f() + c;
```

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

Dead Code Elimination (DCE) Local

Definição

Dead Code: código cujo resultado **nunca é utilizado** após o ponto de cálculo – dentro do bloco ou na saída do bloco.

Código com variável morta

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

`t` é calculado mas nunca lido. Sua instrução pode ser removida.

Após DCE

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

No DAG, nós cujos rótulos são variáveis **não-vivas** na saída do bloco são candidatos à remoção. O compilador verifica as variáveis **ao vivo** (*live variables*) – tema da análise de fluxo.

DCE: Variáveis ao Vivo (Live Variables)

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.

- Variáveis ao vivo na saída de um bloco: informação necessária para DCE **local**.
- A análise de variáveis ao vivo em todo o CFG é chamada de **análise de vivacidade** (*liveness analysis*) – técnica de análise de fluxo de dados (Aula 24).
- Para DCE **local**, podemos usar a heurística: todas as variáveis que são **usadas fora do bloco** (ou que não são temporárias do compilador) são consideradas ao vivo.

Exercício: Aplicando Todas as Otimizações

Dado o bloco básico abaixo, aplique: Constant Propagation, Constant Folding, CSE local e DCE.

```
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;
// Saida: h, f
// (g nao e' usado fora)
```

Passo 1: Constant Propagation + Folding

```
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
```

Exercício: Continuação

Após Constant Folding

```
1  a = 4;
2  b = 2;
3  c = 6;
4  d = 18;
5  e = 6;      // CSE: mesmo que c
6  f = 24;
7  g = 18;     // CSE: mesmo que d
8  h = 34;
```

Passo 2: CSE + DCE

- `e = 6` → `e = c` (CSE com `c`)
- `g = 18` → `g = d` (CSE com `d`)
- `g` não é vivo na saída → **DCE**
- `a, b, c, d, e` podem ser eliminadas se não forem vivas na saída (apenas `h` e `f` são)

```
1  f = 24;
2  h = 34;
3  // 8 instrucoes -> 2!
```

Otimizações que Veremos nas Próximas Aulas

O que estudamos hoje é apenas uma fatia. O compilador moderno realiza dezenas de otimizações:

Otimizações Globais (Aula 25)

- *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)

Loop Invariant Code Motion (Prévia)

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

```
1 for (i = 0; i < n; i++) {
2     x = a * b;    // invariante
          !
3     arr[i] = x + i;
4 }
```

`a * b` é recalculado n vezes, mas nunca muda.

```
1 x = a * b;    // movido para fora
2 for (i = 0; i < n; i++) {
3     arr[i] = x + i;
4 }
```

Reduz de n multiplicações para 1. Para $n = 10^6$, o ganho é enorme!

Loop Unrolling (Prévia)

Loop Unrolling: reduz a sobrecarga de controle do loop (teste e incremento) e facilita a vetorização.

Antes

```
1 for (i = 0; i < 4; i++) {
2     sum += a[i];
3 }
```

4 iterações, 4 testes, 4 incrementos.

Após Unrolling (fator 4)

```
1 sum += a[0];
2 sum += a[1];
3 sum += a[2];
4 sum += a[3];
```

Zero testes e incrementos! O CPU pode até **paralelizar** as somas.

GCC/Clang: Níveis de Otimização

Os compiladores reais agrupam as otimizações em **níveis**:

Flag	O que ativa
`-O0`	Sem otimizações (debug). Cada instrução C vira instruções simples de IR.
`-O1`	Peephole, DCE básico, CP, 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 FP).

Vendo a Diferença: -O0 vs -O2

Código C

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

Assembly -O0 (x86-64)

```
push rbp
mov  rbp, rsp
mov  [rbp-4], edi    ; a
mov  [rbp-8], esi    ; b
mov  eax, [rbp-4]
add  eax, [rbp-8]
mov  [rbp-12], eax   ; x = a+b
mov  eax, [rbp-4]
add  eax, [rbp-8]    ; y = a+b (again!)
mov  [rbp-16], eax
mov  eax, [rbp-12]
add  eax, [rbp-16]   ; x + y
pop  rbp
ret
```

Vendo a Diferença: Resultado com -O2

Código C

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

Assembly -O2 (x86-64)

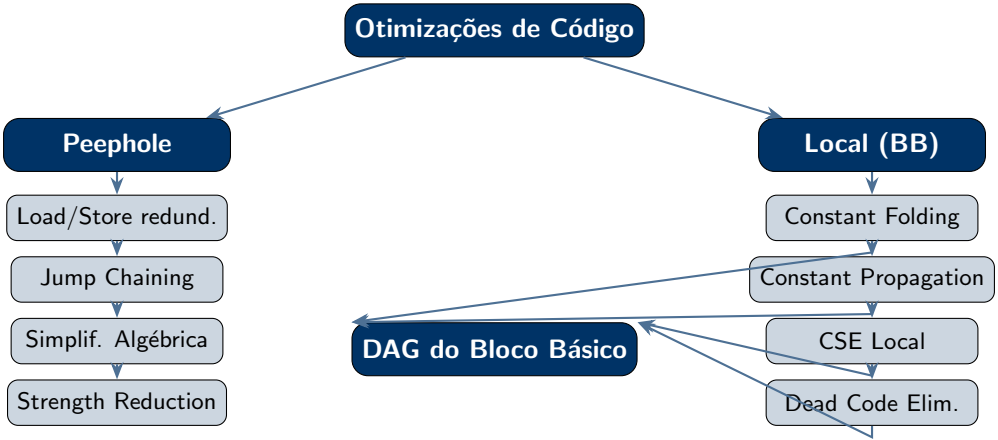
```
lea   eax, [rdi + rsi]    ; a + b
add   eax, eax            ; (a+b)*2
ret
```

O compilador aplicou:

- **CSE**: `a+b` calculado uma vez
- **Strength Reduction**: `x+y = 2*(a+b)` virou `add eax, eax`
- **Register Allocation**: sem acesso à memória

14 instruções $\rightarrow$ 3!

Mapa Mental da Aula



Pontos-Chave para Fixar

1. **Preservação Semântica** é inviolável. Toda otimização deve manter o comportamento observável.
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).
4. **Blocos Básicos** são a unidade de análise local: sem bifurcações internas, a análise é exata.
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.

Referências

- **AHO, A. V. et al.** *Compiladores: Princípios, Técnicas e Ferramentas*. 2ª Edição. (Dragão Roxo) – Cap. 8 e 9.
- **COOPER, K.; TORCZON, L.** *Engineering a Compiler*. 2ª Edição. – Cap. 8 (Local Optimization).
- **APPEL, A. W.** *Modern Compiler Implementation in C*. – Cap. 17.
- **GCC Documentation:** `https://gcc.gnu.org/onlinedocs/gccint/` – Passes e Plugins do GCC.
- **LLVM Reference:** `https://llvm.org/docs/Passes.html` – Lista de passes de otimização do LLVM.

Obrigado!

Email: alessio@cefetmg.br
Web: alessiojr.com

Próxima Aula – Aula 24

Análise de Fluxo de Dados: *Liveness Analysis*, *Reaching Definitions* e *Available Expressions* – a base das otimizações globais.