Estrutura do tema Avaliação de Desempenho (IA-32)

1. A avaliação de sistemas de computação (métricas)
2. Técnicas de otimização de *hardware*
    1. *Hierarquia de memória*
    2. *Exploração de paralelismo*
3. Técnicas de otimização de código
4. Outras técnicas de otimização
5. Medição de tempos …

Técnicas de otimização de código
Introdução

- **Melhorar desempenho das aplicações**
  1. A otimização é possível a dois níveis:
     - **Programador**: através da escolha do algoritmo e estruturas de dados
     - **Compilador:** geração de código otimizado
       - Pode beneficiar da ajuda do programador
  2. A otimização é um compromisso entre legibilidade/abstração e eficiência
     - Como melhorar o desempenho do programa sem destruir a modularidade e generalidade?
     - Os programas fortemente otimizados são bastante mais complexos de manter/depurar
  3. Potencialidades e limitações dos compiladores
     - Quais as otimizações possíveis nos compiladores atuais e quais não são possíveis?

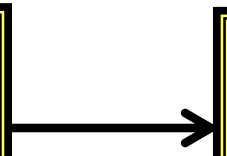
Técnicas de otimização de código otimizações do compilador

- **Objetivos**
  1. Reduzir o número de instruções executadas
     - Evitar repetir cálculos / cálculos desnecessários
     - Substituir instruções “lentas” (CPI elevado: multiplicação)
  2. Evitar saltos
     - Tornar mais fácil a previsão de saltos
     - Desdobrar os ciclos
  3. Reduzir impacto dos acessos à memória
     - Manter valores em registos sempre que possível
     - Usar padrões de acesso “amigáveis” da *cache*
- Limites
  - Não consegue reduzir a complexidade do algoritmo!
  - Tem que garantir a correção (em TODOS o casos)

Otimização de código : movimentação de código (1)

- **Exemplo 1: Movimentação de código**
  - Objetivo: minimizar repetição de cálculos
    - se produzir sempre o mesmo resultado
    - especialmente retirar código do interior de ciclos
  - **Exemplo**: definir uma matriz `a[n,n]` em que todas as colunas são iguais e cada coluna é igual ao vetor `b[n]`

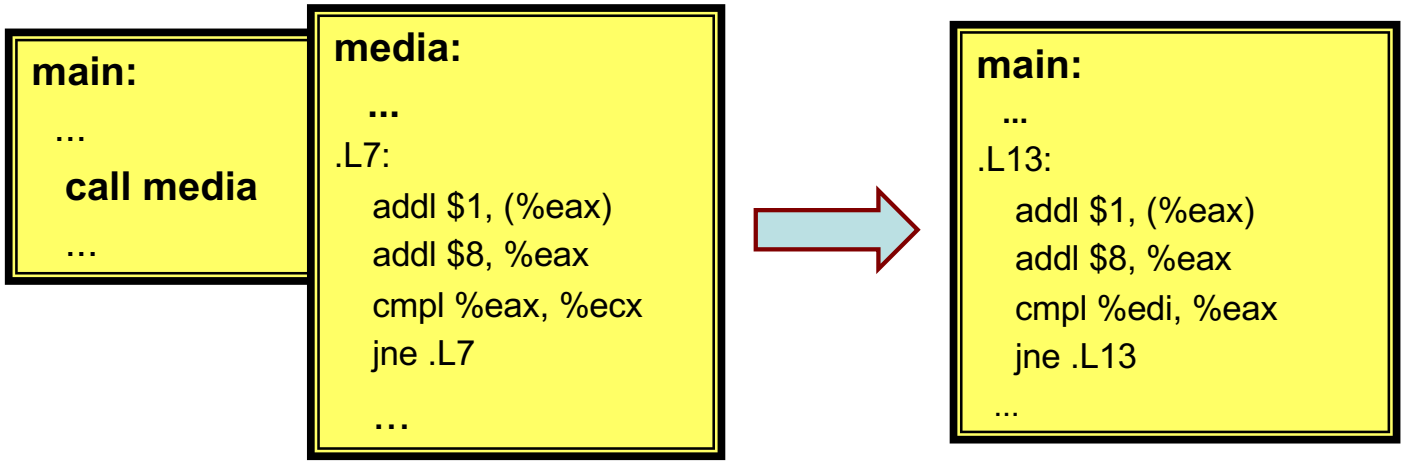
```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```



```
for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```

Otimização de código : expansão em linha(2)

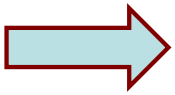
- **Exemplo 2: Expansão em linha (*Inlining*)**
  - Copia o código da função invocada
    - Cria oportunidades para outras otimizações
    - O código pode ficar maior!!!!
  - **Exemplo**: TPC9: chamada a *media*



Otimização de código : desdobramento de ciclos (3)

- **Exemplo 3: Desdobramento de ciclos**
  - Combina múltiplas iterações num só ciclo
    - Amortiza a sobrecarga dos ciclos
    - Requer código adicional no fim do ciclos para lidar com as restantes iterações
    - O código ficar maior!!!!

```
media:
   ...
.L7:
   addl $1, (%eax)
   addl $8, %eax
   cmpl %eax, %ecx
   jne .L7
   ...
```

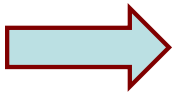


```
media:
   ...
.L7:
   addl $1, (%eax)
   addl $1, 8(%eax)
   addl $16, %eax
   cmpl %eax, %ecx
   jne .L7
   ...
```

- **Exemplo 4: Manter valores em registos**
  - Reduz o número de acessos à memória

```
  movl    $0, -4(%ebp)        # i =0

.L7:
  movl    -4(%ebp), %eax      # i
  movl    8(%ebp), %edx       # v
  incl    (%edx,%eax,4)       # v[i]++
  addl    $1, -4(%ebp)        # i++
  cmpl    $99, -4(%ebp)       # i<99?
  jle     .L7
  ...
```



```
  xorl    %eax, %eax          # i=0
  movl    8(%ebp), %edx       # v
.L7:


  incl    (%edx,%eax,4)       # v[i]++
  incl    %eax                # i++
  cmpl    $99, %eax
  jle     .L7
  ...
```

Limitações às otimizações: memory aliasing

- **Limites das otimizações:** Memory Aliasing

```
/* Sum rows of n X n matrix a and store in vector b. */
void sum_rows1(double *a, double *b, int n) {
    int i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
        movl    $0, (%esi)
        pxor    %xmm0, %xmm0
.L4:
        addsd   (%edi), %xmm0
        movsd   %xmm0, (%esi)
        addl    $8, %edi
        cmpl    %ecx, %edi
        jne     .L4
```

- O código atualiza `b[i]` em cada iteração
- Porque razão b[i] não foi colocado em registo?

Limitações às otimizações: memory aliasing

- **Limites das otimizações:** Memory Aliasing

```
/* Sum rows of n X n matrix a and store in vector b. */
void sum_rows1(double *a, double *b, int n) {
    int i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
```

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16},
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

```
double A[9] =
  { 0,   1,   2,
    3,  22, 224},
   32,  64, 128};
```

Valor de B:

```
init:  [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

```
i = 1: [3, 22, 16]
```

```
i = 2: [3, 22, 224]
```

- O código atualiza b[i] em cada iteração
- Porque razão b[i] não foi colocado em registo? Sobreposição entre A e B!

Limitações às otimizações: memory aliasing

- **Evitar a penalização** Memory Aliasing

```
/* Sum rows of n X n matrix a and store in vector b. */
void sum_rows2(double *a, double *b, int n) {
    int i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

```
        pxor    %xmm0, %xmm0
.L4:
        addsd   (%edi), %xmm0
        addl    $8, %rdi
        cmpl    %eax, %edi
        jne     .L4
        movsd   %xmm0, (%esi)
```

  - Declarar uma variável local

Limitações às otimizações: efeitos colaterais

- **Limites das otimizações:** Efeitos colaterais (na chamada a funções)
  - Substituir func1 por func2?

```
int func1 (x)
{
  return f(x)+f(x)+f(x)+f(x);
}
```

```
int func2 (x)
{
  return 4*f(x);
}
```

E se f(x) for?

```
int counter=0;
f(x) { return(counter++); }
```

- O compilador tem que ser cauteloso com funções que alteram estado global
  - Solução (limitada a algumas funções): **Expansão em linha (*inline*)**

Resumo

- **Resumo**
  - Fases de desenvolvimento
    1. Selecionar o melhor algoritmo
       - Utilizar a análise de complexidade para comparar algoritmos
    2. Escrever código legível e fácil de gerir
    3. Eliminar bloqueadores de otimizações
    4. Medir o perfil de execução
       - Otimizar as partes críticas para o desempenho (*hot-spots*)
         - » Operações repetidas muitas vezes (e.g., ciclos interiores)
  - Código com otimizações é mais complexo de ler, manter e de garantir a correção

- ***Common compiler optimizations***
- Loops
  - Identify **induction variables** that are increased or decreased by a fixed amount on every iteration of a loop (e.g., j = i*4 +1 => j+= 5)
  - **Fission** - break a loop into multiple loops but each taking only a part of the loop's body
  - **Fusion** – combine loops to reduce loop overhead
  - **Inversion** - changes a standard *while* loop into a *do/while*
  - **Interchange** - exchange inner loops with outer loops
  - **Loop-invariant** code motion
  - **Loop unrolling** - duplicates the body of the loop multiple times
  - **Loop splitting** - breaks into multiple loops which have the same bodies but iterate over different contiguous portions of the index range

- ***Common compiler optimizations***
- Data flow
  - **Common sub-expression elimination/sharing**
  - **Reduction in strength -** expensive operations are replaced with less expensive operations
  - **Constant folding** - replaces expressions of constants (e.g., 3 + 5) with their final value (8)
  - **Dead store elimination** - removal of assignments to variables that are not read
- Code generation
  - **Register allocation** - most frequently used variables are kept in processor registers
  - **Instruction selection** – select one of several different ways of performing an operation
  - **Instruction scheduling** – avoid pipeline stalls
  - **Re-materialization** - recalculates a value instead of loading it from memory

- ***Common compiler optimizations***
- Other optimizations
  - **Bounds-checking elimination**
  - **Code-block reordering** – alters the order of basic blocks
  - **Dead code elimination**
  - **Inline expansion** - insert the body of a procedure inside the calling code

- Limitations
  - Memory aliasing & side effects of functions
  - Compilers do not typically improve the algorithmic complexity
  - A compiler typically only deals with a part of a program at a time
  - Time overhead of compiler otimisations