

Lic. Eng<sup>a</sup> Informática

1º ano

2009/10

A.J.Proença

**Tema****Avaliação de Desempenho (IA-32)***Análise do desempenho  
na execução de aplicações (1)***"Análise do desempenho": para quê?**

– para avaliar Sistemas de Computação

- identificação de métricas
  - latência, velocidade, ...
- ligação entre métricas e factores na arquitectura que influenciam o desempenho de um núcleo

$$\text{Core}_{\text{time}} = N^{\circ}_{\text{instr}} * \text{CPI} * T_{\text{clock}}$$

e ...

– ... **construí-los mais rápidos**– ... **melhorar a eficiência de execução de appl****Estrutura do tema Avaliação de Desempenho (IA-32)**

1. A avaliação de sistemas de computação
2. Técnicas de optimização de código (IM)
3. Técnicas de optimização de *hardware*
4. Técnicas de optimização de código (DM)
5. Outras técnicas de optimização
6. Medição de tempos

$$\text{Core}_{\text{time}} = N^{\circ}_{\text{instr}} * \text{CPI} * T_{\text{clock}}$$

**Análise dos componentes da fórmula:**

- **Core<sub>time</sub>**
  - inclui tempo de execução no CPU, acessos à memória, ...
- **N<sup>o</sup><sub>instr</sub>**
  - efectivamente executadas; depende essencialmente de:
    - eficiência do compilador
    - do *instruction set*
- **CPI (Clock-cycles Per Instruction)**
  - tempo médio de exec de 1 instr, em ciclos; depende essencialmente de:
    - complexidade da instrução (e acessos à memória ...)
    - paralelismo na execução da instrução
- **T<sub>clock</sub>**
  - período do *clock*; depende essencialmente de:
    - complexidade da instrução (ao nível dos sistemas digitais)
    - micro-electrónica

## "Análise do desempenho": para quê?

### – . . . melhorar a eficiência de execução de app

- análise de técnicas de optimização
  - algoritmo / **codificação / compilação** / *assembly*
  - compromisso entre legibilidade e eficiência...
  - potencialidades e limitações dos compiladores...
  - técnicas independentes / dependentes da máquina
  - uso de *code profilers*
- técnicas de medição de tempos
  - escala microscópica / macroscópica
  - uso de *cycle counters* / *interval counting*

### – um compilador moderno já inclui técnicas que

- explora oportunidades para simplificar expressões
- usa um único cálculo de expressão em vários locais
- reduz o nº de vezes um cálculo é efectuado
- tira partido de algoritmos sofisticados para
  - alocação eficiente dos registos
  - selecção e ordenação de código
- ... **mas** está limitado por factores, incluindo
  - nunca modificar o comportamento correcto do programa
  - limitado conhecimento do programa e seu contexto
  - necessidade de ser rápido!

### – certas optimizações estão-lhe vedadas...

### – certas optimizações estão vedadas aos compiladores:

- pode trocar `twiddle1` por `twiddle2` ?

```
void twiddle1(int *xp,int *yp)
{
  *xp += *yp;
  *xp += *yp;
}
```

```
void twiddle2(int *xp,int *yp)
{
  *xp += 2* *yp;
}
```

teste: `xp` igual a `yp`: que acontece?

- pode trocar `func1` por `func2` ?

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

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

teste: e se `f` for...?

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

## Estrutura do tema Avaliação de Desempenho (IA-32)

1. A avaliação de sistemas de computação
2. Técnicas de optimização de código (IM)
3. Técnicas de optimização de *hardware*
4. Técnicas de optimização de código (DM)
5. Outras técnicas de optimização
6. Medição de tempos

**"Independentes da máquina": aplicam-se a qualquer processador / compilador**

**Algumas técnicas de optimização:**

- movimentação de código
  - reduzir frequência de execução (compiladores têm limitações)
- simplificação de cálculos
  - substituir operações por outras mais simples
- partilha de cálculos
  - identificar e explicitar sub-expressões comuns

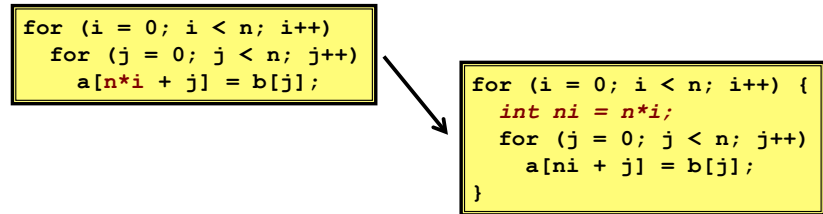
**Metodologia a seguir:**

- apresentação de alguns conceitos
- análise de um programa exemplo a otimizar
- introdução de uma técnica de medição de desempenho

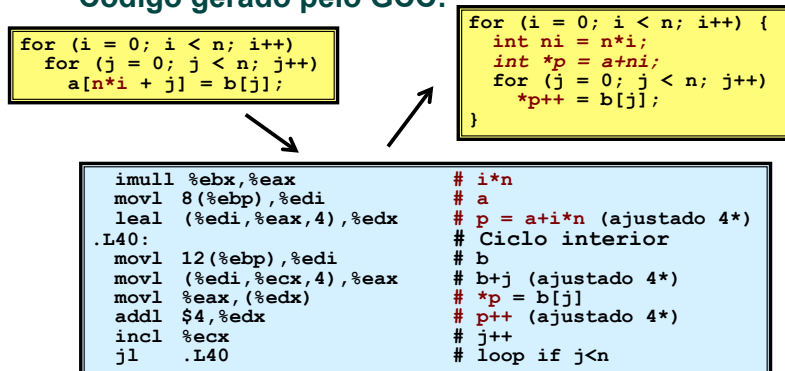


**• Movimentação de código**

- Reduzir a frequência da realização de cálculos
  - se produzir sempre o mesmo resultado
  - especialmente retirar código do interior de ciclos

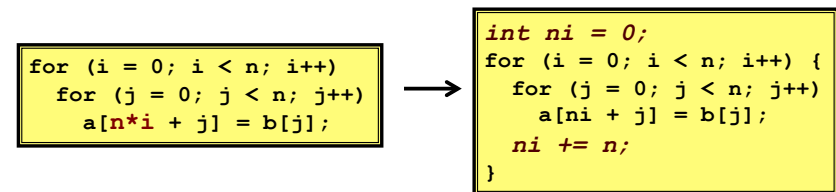


- A maioria dos compiladores é eficiente a lidar com código com *arrays* e estruturas simples com ciclos
- Código gerado pelo GCC:



**• Substituir operações "caras" por outras +simples**

- **shift** ou **add** em vez de **mul** ou **div**
  - $16*x \rightarrow x \ll 4$
  - escolha pode ser dependente da máquina
- reconhecer sequência de produtos



• Partilhar sub-expressões comuns

- reutilizar partes de expressões
- compiladores não são particularmente famosos a explorar propriedades aritméticas

```
/* Soma vizinhos de i,j */
up = val[(i-1)*n + j];
down = val[(i+1)*n + j];
left = val[i*n + j-1];
right = val[i*n + j+1];
sum = up + down + left + right;
```

```
int inj = i*n + j;
up = val[inj - n];
down = val[inj + n];
left = val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

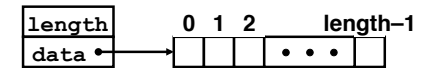
3 multiplicações:  $i*n$ ,  $(i-1)*n$ ,  $(i+1)*n$

1 multiplicação:  $i*n$

```
leal -1(%edx),%ecx # i-1
imull %ebx,%ecx # (i-1)*n
leal 1(%edx),%eax # i+1
imull %ebx,%eax # (i+1)*n
imull %ebx,%edx # i*n
```



O vector ADT:



• Procedimentos associados

`vec_ptr new_vec(int len)`

- cria vector de comprimento especificado

`int get_vec_element(vec_ptr v, int index, int *dest)`

- recolhe um elemento do vector e guarda-o em \*dest
- devolve 0 se fora de limites, 1 se obtido com sucesso

`int *get_vec_start(vec_ptr v)`

- devolve apontador para início dos dados do vector

• Idêntico às implementações de arrays em Pascal, ML, Java

- i.e., faz sempre verificação de limites (bounds)

```
void combine1(vec_ptr v, int *dest)
{
    int i;
    *dest = 0;
    for (i = 0; i < vec_length(v); i++) {
        int val;
        get_vec_element(v, i, &val);
        *dest += val;
    }
}
```

• Procedimento

- calcula a soma de todos os elementos do vector
- guarda o resultado numa localização destino
- estrutura e operações do vector definidos via ADT

• Tempos de execução: que/como medir?



```
void combine1(vec_ptr v, int *dest)
{
    int i;
    *dest = 0;
    for (i = 0; i < vec_length(v); i++) {
        int val;
        get_vec_element(v, i, &val);
        *dest += val;
    }
}
```

Tempos de execução: que/como medir?

- **que medir:** em programas iterativos (com ciclos), uma medida útil é a duração da operação para cada um dos elementos da iteração:
  - **ciclos** (de clock) **por elemento, CPE**
- **como medi-lo:** efectuar várias medições de tempo para dimensões variáveis de ciclos, e calculá-lo através do traçado gráfico: o declive da recta *best fit*, obtida pelo método dos mínimos quadrados:
  - análise gráfica de um exemplo...

Análise detalhada de um exemplo:  
tempos de execução (2)

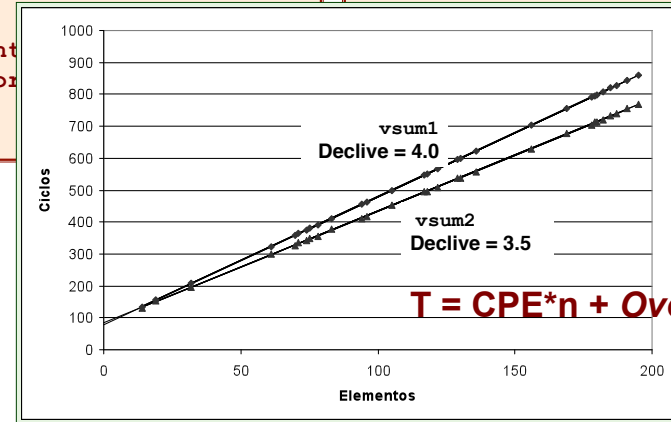
```
void vsum1(int n)
{
    int i;
    for (i=0; i<n; i++)
        c[i] = a[i] + b[i];
}
```

```
void vsum2(int n)
{
    int i;
    for (i=0; i<n; i+=2){
        c[i] = a[i] + b[i];
        c[i+1]= a[i+1]+ b[i+1];
    }
}
```

Análise detalhada de um exemplo:  
tempos de execução (3)

```
void vsum1(int n)
{
    int i;
    for (i=0; i<n; i++)
        c[i] = a[i] + b[i];
}
```

```
void vsum2(int n)
{
    int i;
    for (i=0; i<n; i++)
        c[i] = a[i] + b[i];
}
```



Análise detalhada de um exemplo:  
o procedimento a otimizar (2)

```
void combinel(vec_ptr v, int *dest)
{
    int i;
    *dest = 0;
    for (i = 0; i < vec_length(v); i++) {
        int val;
        get_vec_element(v, i, &val);
        *dest += val;
    }
}
```

- **Procedimento**
  - calcula a soma de todos os elementos do vector
  - guarda o resultado numa localização destino
  - estrutura e operações do vector definidos via ADT
- **Tempo de execução (inteiros) :**
  - compilado sem qq optimização: 42.06 CPE
  - compilado com -O2: 31.25 CPE

Análise detalhada do exemplo:  
à procura de ineficiências...

Versão  
goto

```
void combinel-goto(vec_ptr v, int *dest)
{
    int i = 0;
    int val;
    *dest = 0;
    if (i >= vec_length(v)) goto done;

    loop:
    get_vec_element(v, i, &val);
    *dest += val;
    i++;
    if (i < vec_length(v))
        goto loop;
    done:
}
```

} 1 iteração

**Ineficiência óbvia:**

- função `vec_length` invocada em cada iteração
- ... mesmo sendo para calcular o mesmo valor!



### Optimização 1:

- mover invocação de `vec_length` para fora do ciclo interior
  - o valor não altera de iteração para iteração
- **CPE:** de 31.25 para **20.66** (compilado com `-o2`)
  - `vec_length` impõe um *overhead* constante, mas significativo

```
void combine2(vec_ptr v, int *dest)
{
    int i;
    int length = vec_length(v);
    *dest = 0;
    for (i = 0; i < length; i++) {
        int val;
        get_vec_element(v, i, &val);
        *dest += val;
    }
}
```



### Por que razão o compilador não moveu `vec_len` para fora do ciclo?

- a função pode ter efeitos colaterais
  - altera o estado global de cada vez que é invocada
- a função poderá não devolver os mesmos valores consoante o `arg`
  - depende de outras partes do estado global

### Por que razão o compilador não analisou o código de `vec_len`?

- otimização interprocedimental não é usada extensivamente devido ao seu elevado custo

### Warning:

- o compilador trata invocação de procedimentos como uma *black box*
- as otimizações são pobres em redor de invoc de procedimentos



### Optimização 2:

- evitar invocação de `get_vec_element` para ir buscar cada elemento do vector
  - obter apontador para início do *array* antes do ciclo
  - dentro do ciclo trabalhar apenas com o apontador
- **CPE:** de 20.66 para **6.00** (compilado com `-o2`)
  - invocação de funções é dispendioso
  - validação de limites de *arrays* é dispendioso

```
void combine3(vec_ptr v, int *dest)
{
    int i;
    int length = vec_length(v);
    int *data = get_vec_start(v);
    *dest = 0;
    for (i = 0; i < length; i++) {
        *dest += data[i];
    }
}
```



### Optimização 3:

- não é preciso guardar resultado no `dest` a meio dos cálculos
  - a variável local `sum` é alocada a um registo
  - poupa 2 acessos à mem por ciclo (1 leitura + 1 escrita)
- **CPE:** de 6.00 para **2.00** (compilado com `-o2`)
  - acessos à mem são dispendiosos

```
void combine4(vec_ptr v, int *dest)
{
    int i;
    int length = vec_length(v);
    int *data = get_vec_start(v);
    int sum = 0;
    for (i = 0; i < length; i++)
        sum += data[i];
    *dest = sum;
}
```

### Combine3

```
.L18:
    movl (%ecx,%edx,4),%eax
    addl %eax, (%edi)
    incl %edx
    cmpl %esi,%edx
    jl .L18
```

### Combine4

```
.L24:
    addl (%eax,%edx,4),%ecx

    incl %edx
    cmpl %esi,%edx
    jl .L24
```

### Desempenho comparativo

- Combine3
  - 5 instruções em 6 ciclos de *clock*
  - `addl` tem de ler e escrever na memória
- Combine4
  - 4 instruções em 2 ciclos de *clock*

### • Aliasing

- 2 referências distintas à memória especificam a mesma localização

### • Example

- `v: [3, 2, 17]`
- `combine3(v, get_vec_start(v)+2) --> ?`
- `combine4(v, get_vec_start(v)+2) --> ?`

### • Observações

- fácil de acontecer em C, por permitir
  - operações aritméticas com endereços
  - acesso directo a valores armazenados de *structures*
- criar o hábito de usar variáveis locais
  - para acumular resultados dentro de ciclos
  - como forma de avisar o compilador para não se preocupar com *aliasing*

```
void abstract_combine4(vec_ptr v, data_t *dest)
{
    int i;
    int length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP data[i];
    *dest = t;
}
```

### Tipos de dados

- Usar declarações distintas para `data_t`
  - `int`
  - `float`
  - `double`

### Operações

- Usar definições diferentes para `OP` e `IDENT`
  - `+` / `0`
  - `*` / `1`

### Optimizações

- reduzir invocação `func` e acessos à memória dentro do ciclo

Método	Inteiro		Real (prec simp)	
	+	*	+	*
<i>Abstract -g</i>	42.06	41.86	41.44	160.00
<i>Abstract -O2</i>	31.25	33.25	31.25	143.00
<i>Move vec_length</i>	20.66	21.25	21.15	135.00
<i>Acesso aos dados</i>	6.00	9.00	8.00	117.00
<i>Acum. em temp</i>	2.00	4.00	3.00	5.00

### • Anomalia no desempenho

- cálculos de produtos de FP excepcional/ lento com todos
- aceleração considerável quando acumulou em `temp`
- causa: unidade de FP do IA32
  - mem usa formato com 64-bit, registo usa 80
  - os dados causaram *overflow* com 64 bits, mas não com 80