

Avaliação de Desempenho no IA-32 (2)



Estrutura do tema Avaliação de Desempenho (IA-32)

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

Otimização de código: técnicas independentes da máquina



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

Algumas técnicas de otimização:

- 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 subexpressões comuns

Metodologia a seguir:

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

Otimizações independentes da máquina: movimentação de código (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];  
}
```

Otimizações independentes da máquina: movimentação de código (2)



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

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

```
imull %ebx,%eax      # i*n colocado em %eax  
movl 8(%ebp),%edi    # apont p/ array a em %edi  
leal (%edi,%eax,4),%edx # p=a+n*i (ajustado 4*) em %edx  
.L40:               # Ciclo interior  
movl 12(%ebp),%edi   # apont p/ array b em %edi  
movl (%edi,%ecx,4),%eax # b+j (ajustado 4*) em %eax  
movl %eax,(%edx)     # *p=b[j] (%edx aponta para b+j)  
addl $4,%edx         # p++ (ajustado 4*)  
incl %ecx            # j++  
jl .L40             # loop if j<n
```

Otimizações independentes da máquina: simplificação de cálculos



- Substituir operações “caras” por outras +simples

- *shift* ou *add* em vez de *mul* ou *div* por potências de 2

- $16 * x \rightarrow x \ll 4$

- escolha pode ser dependente da máquina

Já visto!

- reconhecer sequência de produtos

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



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

Otimizações independentes da máquina: partilha de cálculos



- **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;
```

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

```
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
```

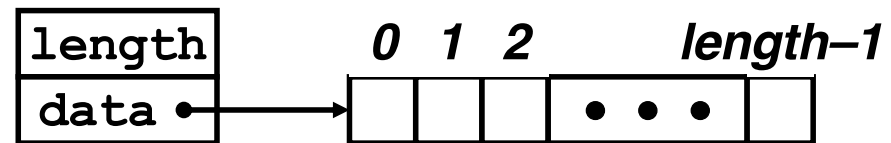
```
int inj = i*n + j;  
up =    val[inj - n];  
down =  val[inj + n];  
left =  val[inj - 1];  
right = val[inj + 1];  
sum = up + down + left + right;
```

1 multiplicação: $i*n$

Análise detalhada de um exemplo: introdução ao Abstract Data Type (ADT)



O vetor ADT:



- **Funções associadas**

```
vec_ptr new_vec(int len)
```

- cria vetor com o comprimento especificado

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

- recolhe um elemento do vetor 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 vetor

- Idêntico às implementações de *arrays* em Pascal, ML, Java
 - i.e., faz sempre verificação de limites (*bounds*)

Análise detalhada de um exemplo: o procedimento a otimizar (1)



```
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 vetor
 - guarda o resultado numa localização “destino”
 - estrutura e operações do vetor definidos via ADT
- **Tempos de execução: que/como medir?**

Análise detalhada de um exemplo: tempos de execução (1)



```
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;
    }
}
```

Tempos de execução: que medir e 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 medir o CPE**: fazer várias medições de tempo para dimensões variáveis de ciclos, e calculá-lo através do traçado gráfico; o CPE é o declive da reta *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];
    }
}
```

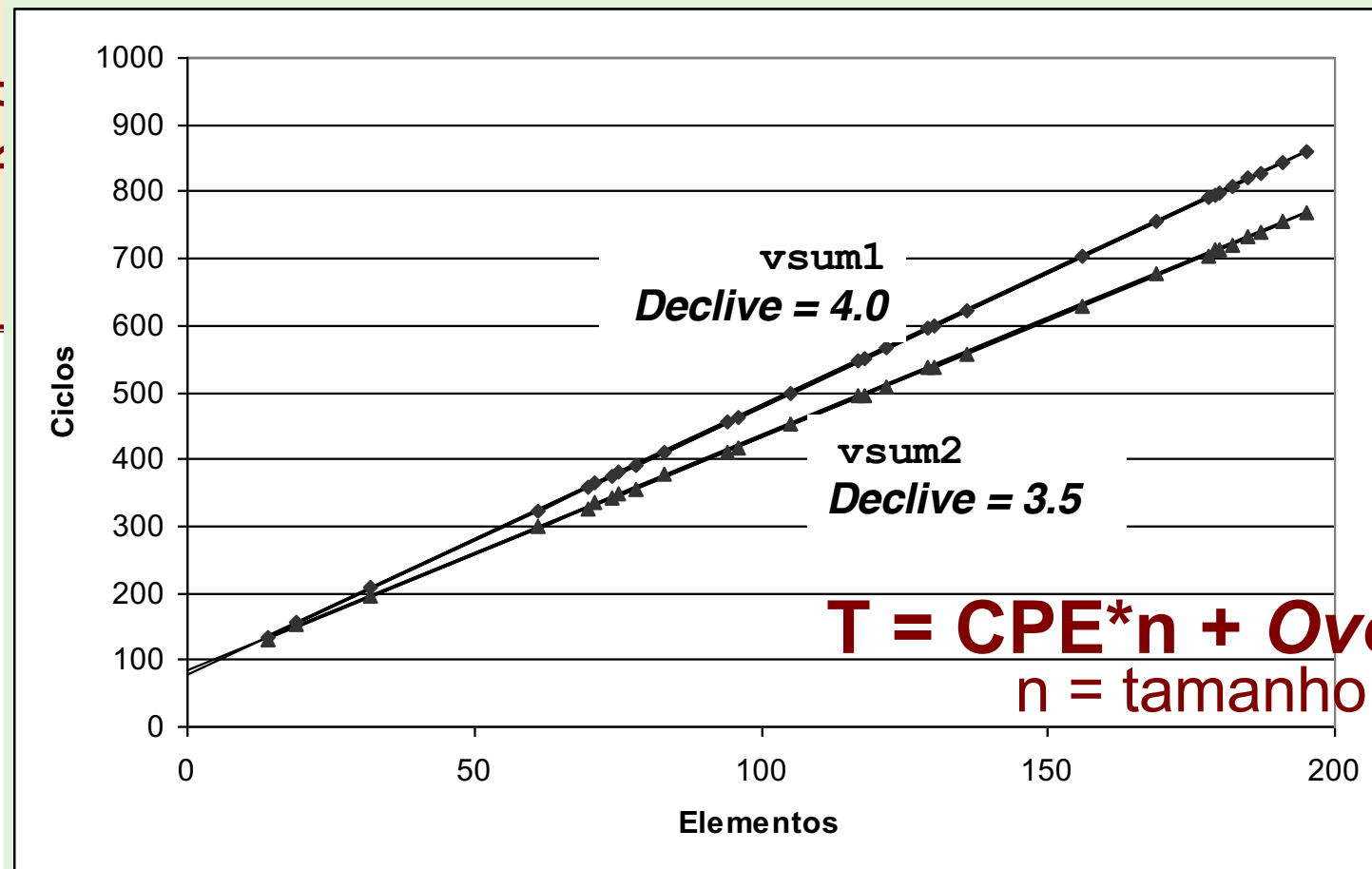
Análise detalhada de um exemplo: tempos de execução (3)



```
void vsum1(int n)
{
    int
    for
}
```

```
void vsum2(int n)
```

```
{
    [i];
    [i+1];
}
```



$$T = CPE * n + \text{Overhead}$$

n = tamanho do vetor

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



```
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 vetor
 - guarda o resultado numa localização destino
 - estrutura e operações do vetor definidos via ADT
- **Tempo de execução (inteiros) :**
 - compilado sem qq otimização: 42.06 CPE
 - compilado com **-O2**: 31.25 CPE

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



Versão
goto

```
void combine1-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!

Análise detalhada do exemplo: movimentação de código



Otimização 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;
    }
}
```

Bloqueadores de otimização: a invocação de procedimentos/funções



Por que razão o compilador não moveu `vec_1en` para fora do ciclo?

- a função pode ter efeitos colaterais
 - por ex., alterar 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_1en`?

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

Aviso:

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

Análise detalhada do exemplo: simplificação de cálculos



Otimização 2:

- evitar invocação de `get_vec_element` para ir buscar cada elemento do vetor
 - 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, mas tem riscos dispensá-lo
 - 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];
    }
}
```


Análise detalhada do exemplo: eliminar referências desnecessárias à memória



Otimização 3:

- não é preciso guardar resultado em `dest` a meio dos cálculos
 - adiciona-se a variável local `sum` que é alocada a um registo
 - poupam-se 2 acessos à memória por ciclo (1 leitura + 1 escrita)
- **CPE:** de 6.00 para **2.00** (compilado com **-O2**)
 - acessos à memória 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;
}
```

Análise detalhada do exemplo: como detetar referências desnecessárias à memória



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 em cada iteração

– Combine4

- 4 instruções em 2 ciclos de *clock*

Bloqueadores de otimização: aliasing de memória



- **Aliasing**

- 2 referências distintas à memória especificam a mesma localização

- **Example**

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

- **Observações**

- fácil de acontecer em C, porque esta linguagem permite
 - operações aritméticas com endereços
 - acesso direto a valores armazenados em estruturas de dados
- 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***

Análise detalhada do exemplo: forma genérica e abstrata de *combine*



```
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

- declarações distintas para **data_t**
 - int
 - float
 - double

Operações

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

Otimizações independentes da máquina: resultados experimentais com o programa exemplo



Otimizações

- 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 IA-32
 - memória usa formato com 64-bit, registo usa 80
 - os dados causaram *overflow* com 64 bits, mas não com 80