Arquitetura de Computadores

Optimização de Desempenho

(técnicas independentes da máquina)

João Luís Ferreira Sobral
jls@...

Conteúdos	**4 – Optimização do Desempenho**
	4.1 – Capacidades e Limitações dos Compiladores
	4.2 – Técnicas Independentes do Processador: optimização de ciclos, invocação de procedimentos e acessos a memória
Resultados de Aprendizagem	R4.1 – Descrever, aplicar e avaliar técnicas de optimização de desempenho

Optimização de Desempenho

- **Melhorar o desempenho das aplicações**
  - Optimização é possível a dois níveis:
    - **Programador**: através da escolha do algoritmo e estruturas de dados
    - **Compilador:** geração de código optimizado
      - Pode beneficiar da ajuda do programador
  - A optimização é um compromisso entre legibilidade/abstração e eficiência
    - Como melhorar o desempenho do programa sem destruir a modularidade e generalidade?
    - Os programas fortemente optimizados são bastante mais complexos de manter/depurar
  - Potencialidades e limitações dos compiladores
    - Quais as optimizações possíveis nos compiladores atuais e quais não são possíveis?
  - As optimizações podem ser **independentes** ou **dependentes da arquitetura**

Optimização de Desempenho

- **Potencialidades e limitações dos compiladores**

  - **Técnicas utilizadas em compiladores atuais**
    - Simplificação de expressões
    - Utilização de um único cálculo de uma expressão em vários locais
    - Redução do nº de vezes que um cálculo é efectuado
    - Algoritmos sofisticados para:
      - Alocação de registos, seleção e ordenação de código, eliminação de pequenas ineficiências
  - **Limitações**
    - Garantir o comportamento correto do programa (mesmo para casos pouco frequentes!)
    - Conhecimento do programa
      - Pode ser óbvio para o programador mas ofuscado pela linguagem de programação
    - Tempo de compilação
      - Confina as optimizações a blocos de código (e.g., procedimentos)
    - Análise estática do programa
      - Alguns parâmetros podem só ser conhecidos durante a execução

  - **Os bloqueadores de optimizações limitam as optimizações aplicáveis pelo compilador:**
    - "aliasing" nos acessos à memória
    - efeitos co-laterais nos procedimentos (limita as optimizações inter-procedimentos)

Optimização de Desempenho

- **Bloqueador de optimizações: *memory aliasing***
  - **Trocar** twiddle1 **por** twiddle2

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

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

  - **o que acontece se xp for igual a yp?**
    - twiddle1 calcula (*xp)*4
    - twiddle2 calcula (*xp)*3
  - O compilador tem que ser cauteloso com optimizações que envolvem apontadores porque o seu valor concreto só é conhecido durante a execução

Optimização de Desempenho

- **Bloqueador de optimizações: efeitos colaterais**

  – **Trocar** func1 **por** func2

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

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

  – **o que acontece se f(x) for:**

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

- O compilador tem que ser cauteloso com funções que alteram estado global, não podendo substituir expressões que envolvem a função por versões optimizadas

Optimização de Desempenho

- **Técnicas independentes da máquina**
  - **Aplicam-se a qualquer processador / compilador**
  - **Algumas técnicas**
    - movimentação de código
      - reduzir frequência de execução do bloco de código
    - simplificação de cálculos
      - substituir operações por outras mais simples
    - partilha de cálculos
      - identificar e explicitar sub-expressões comuns
    - utilizar registos para armazenar variáveis
      - diminui o número de acessos à memória

Optimização de Desempenho

- **Movimentação de código**
  - Reduz a frequência de realização de cálculos
    - Aplicável se a expressão produz sempre os mesmos resultados
    - Utilizado frequentemente para mover código para fora de ciclos

```
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];
}
```

  - **Código gerado pelo gcc**

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

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

Optimização de Desempenho

- **Substituir operações por outras mais simples**
  (reduction in Strength)

  – ***shift* ou *add* em vez de *mul* ou *div***
    - 16*x = x<<4
    - escolha pode ser dependente da máquina

  – **reconhecer uma 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++) {
        for (j = 0; j < n; j++)
                a[ni + j] = b[j];
        ni += n;
}
```

Optimização de Desempenho

- **Partilhar sub-expressões comuns**
  - **Reutilizar partes de expressões**

```
/* 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;
```

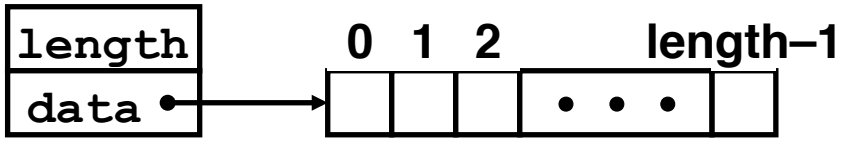
```
leal -1(%edx),%ecx          # i-1
imull %ebx,%ecx             # (i-1)*n
leal 1(%edx),%eax           # i+1
imull %ebx,%eax             # (i+1)*n
imull %ebx,%edx             # i*n
```

Optimização de Desempenho

- **Utilizar registos para armazenar variáveis**
  - **Evita acessos à memória**
  - **Condicionado pelo número de registos disponíveis**
    - **IA-32 apenas dispõe de 8 registos genéricos**
      - **Como tirar partido da renomeação de registos?**
    - **Máquinas com conjuntos de instruções mais recentes (e.g., RISC, IA-64) têm mais registos (e.g. 32)**
  - **Possibilidade de *aliasing***
  - **Nem sempre é possível determinar se um valor pode ser armazenado em registo**
    - **Em C é possível indicar que a variável deve ficar em registo**
      - **register int a;**
  - **Problemas em máquinas "multi-core"**
    - **Modelos de consistência da memória (a analisar mais tarde)**

Optimização de Desempenho

- **Exemplo: vector ADT**
  - **Procedimentos**




```
vec_ptr new_vec(int len)
```
  - Create vector of specified length

```
int get_vec_element(vec_ptr v, int index, int *dest)
```
  - Retrieve vector element, store at *dest
  - Return 0 if out of bounds, 1 if successful

```
int *get_vec_start(vec_ptr v)
```
  - Return pointer to start of vector data

  - **Implementações similares aos *vectors* em Java**
    - E.g., always do bounds checking

Optimização de Desempenho

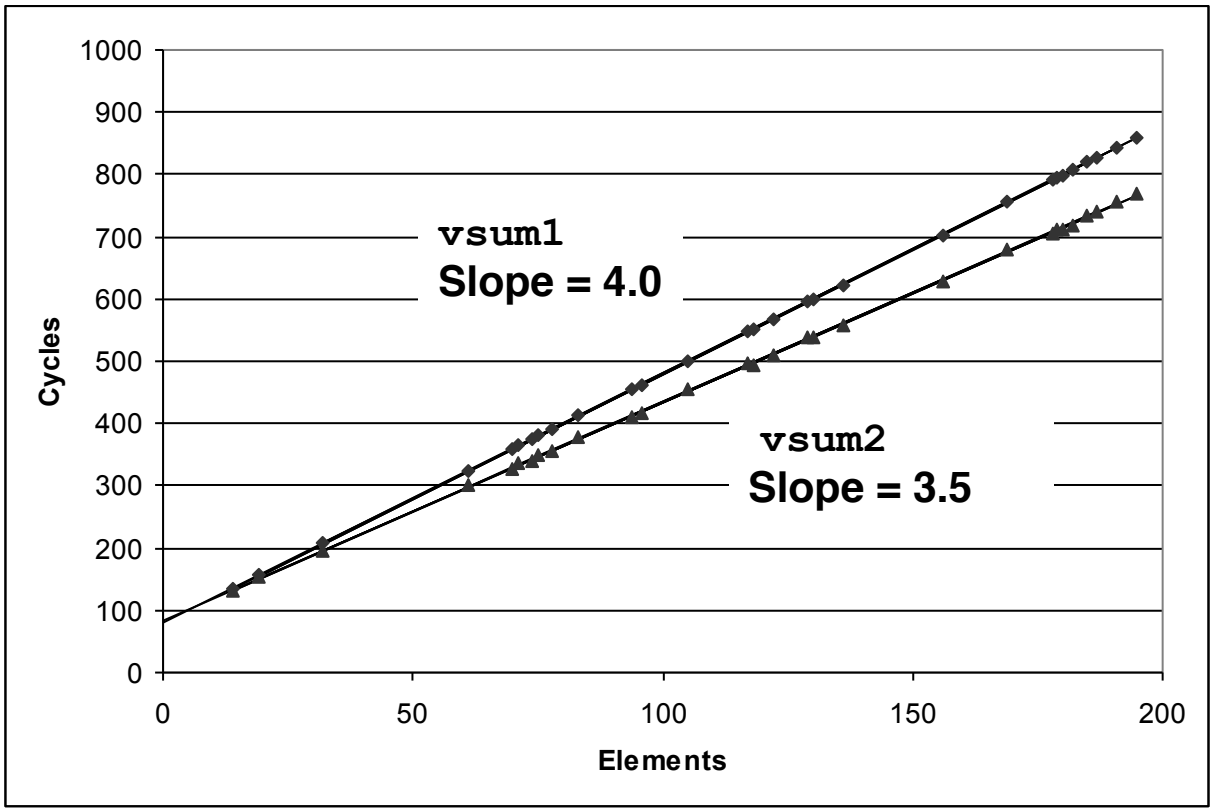
- **Exemplo: vector ADT**

  - **Procedimento**
    - **Calcular a soma de todos os elementos**
    - **Guardar o resultado numa localização especificada**

```
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;
  }
}
```

Optimização de Desempenho

- **Exemplo: vector ADT – Métrica de desempenho**
  - **Ciclos necessários para processar cada elemento (CPE)**
    - **Num. Ciclos = CPE*n + sobrecarga** (***n = Nº de elementos***)
    - **CPE (máquina i7, gcc 4.2) = 28,90 (-g), 11,45 (–O3)**

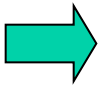


Optimização de Desempenho

- **Exemplo: vector ADT**

  - **Optimização 1 – Movimentação de código**
    - **Mover *vec_length(v)* para fora do ciclo**
    - **CPE = 7,11 (-O3) (versus 11,45 de combine1)**

```
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;
  }
}
```



```
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;
  }
}
```

Optimização de Desempenho

- **Exemplo: vector ADT**

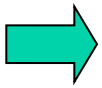
  - **Bloqueador de optimizações: chamadas a procedimentos**
    - **O procedimento pode gerar efeitos colaterais**
      - **Alterar o estado global quando invocado**
    - **A função pode não retornar sempre o mesmo valor para determinados parâmetros de entrada**
      - **O valor pode depender do estado**
    - **Porque não foi analisado o conteúdo de "vec_length"?**
      - **Pode ser substituída por outra versão pelo "linker"**
        - » **Declarar a função com "static"**
        - » **Funções ligadas dinamicamente?**
      - **A análise inter-procedimentos tem custos elevados**

    - **O compilador trata as chamadas a procedimentos como "caixas pretas"**
      - **Optimizações limitadas nos procedimentos e antes/depois das chamadas**

Optimização de Desempenho

- **Exemplo: vector ADT**
  - **Optimização 2 – Partilha de cálculos (*reduction in strength*)**
    - **Obter um apontador para o vector fora do ciclo**
    - **Não é tão "limpo" em termos de abstração de dados**
    - **CPE = 1,65 (-O3)**
      - **As chamadas a procedimentos e verificação de limites são operações caras!**

```
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;
  }
}
```

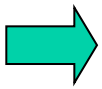


```
void combine3(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    *dest += data[i];
}
```

Optimização de Desempenho

- **Exemplo: vector ADT**
  - **Optimização 3 – Variáveis locais (em registos)**
    - **Manter o valor da soma numa variável local (em registo?)**
    - **Reduz o número de acessos à memória (1 rd + 1 wr)**
    - **CPE = 1,36 (-O3)**

```
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];
}
```



```
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;
}
```

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

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

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

Optimização de Desempenho

- **Bloqueador de optimização – "memory aliasing"**
  - **Duas referências à memória podem especificar a mesma localização**
  - **Exemplo:**
    - `v: [3, 2, 17]`
    - `combine3(v, get_vec_start(v)+2) -->  ?`
    - `combine4(v, get_vec_start(v)+2) -->  ?`
  - **Frequente em programas em C**
    - **Aritmética de endereços**
    - **Acesso directo a estruturas de dados**
  - **Aritmética de endereços não é permitida em Java**