Mestrado em Informática

2007/08

A.J.Proença

Tema

Revisitando os Sistemas de Computação (3)

Estrutura do tema Avaliação de Desempenho (IA32)

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

"*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

$$CPU_{Time} = N^{o}_{instr} * CPI * T_{clock}$$

e . . .

– **. . . construi-los mais rápidos**

– **. . . melhorar o desempenho das aplicações**

"*Análise do desempenho*": para quê?

– **. . . melhorar o desempenho das aplicações**
- 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 expr em vários locais
- reduz o nº de vezes um cálculo é efectuado
- tiram 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 (IA32)

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 optimizar
- 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

```
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];
}
```


- 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
  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
```


- **Substituir operações "caras" por outras +simples**
  - **`shift`** ou **`add`** em vez de **`mul`** ou **`div`**
    - **`16*x`** → **`x<<4`**
    - escolha pode ser dependente da máquina
  - 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++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```


- **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;
```

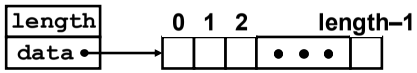
3 multiplicações: i*n, (i–1)*n, (i+1)*n

```
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

```
  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?

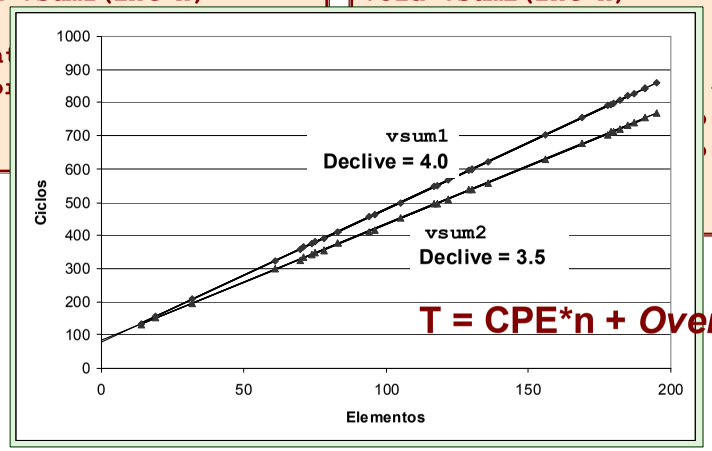
- **que medir**: em programas iterativos (com ciclos), uma medida útil é a duração da operação para cada um dos elementos da iteração:
  - **ciclos** (de *clock*) **por elemento, CPE**
- **como medi-lo**: efectuar várias medições de tempo para dimensões variáveis de ciclos, e calculá-lo através do traçado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


```
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];
  }
}
```

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

```
void vsum2(int n)
{
  [i];
  [i+1];
```



T = CPE*n + *Overhead*


```
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
- **Tempo de execução** (inteiros) **:**
  - compilado sem qq optimização: 42.06 CPE
  - compilado com **-O2**: 31.25 CPE

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!

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`?

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

Warning:

- o compilador trata invocação de procedimentos como uma *black box*
- as optimizações 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 mem dentro do ciclo

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

- Anomalia no desempenho
  - cálculos de produtos de FP excepcional/ lento com todos
  - aceleração considerável qdo. 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

Estrutura do tema Avaliação de Desempenho (IA32)

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

Estrutura do tema Avaliação de Desempenho (IA32)

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

Análise de técnicas de optimização (*s/w*)

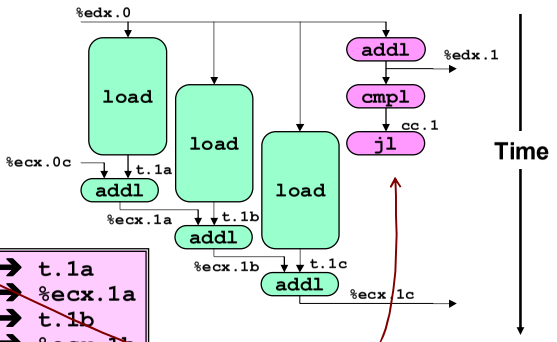
- técnicas de optimização de código (indep. máquina)
  - *já visto...*
- **técnicas de optimização de código (dep. máquina)**
  - análise sucinta de um CPU actual, P6 (*já visto...*)
  - ***loop unroll* e *inline functions***
  - **identificação de potenciais limitadores de desempenho**
  - dependentes da hierarquia da memória
- outras técnicas de optimização (*a ver adiante...*)
  - na compilação: optimizações efectuadas pelo `Gcc`
  - na identificação dos "gargalos" de desempenho
    - *program profiling* e uso dum *profiler* p/ apoio à optimização
    - lei de Amdahl


```
void combine5(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int limit = length-2;
  int *data = get_vec_start(v);
  int sum = 0;
  int i;
  /* junta 3 elem's no mesmo ciclo */
  for (i = 0; i < limit; i+=3) {
    sum += data[i] + data[i+1]
           + data[i+2];
  }
  /* completa os restantes elem's */
  for (; i < length; i++) {
    sum += data[i];
  }
  *dest = sum;
}
```

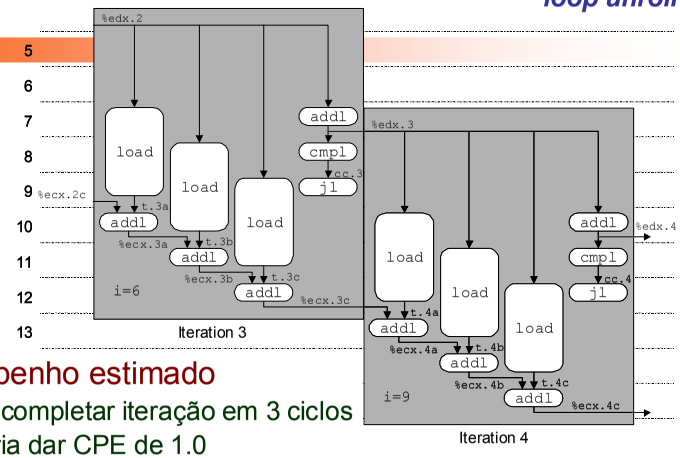
Optimização 4:

- juntar várias (**3**) iterações num simples ciclo
- amortiza *overhead* dos ciclos em várias iterações
- termina extras no fim
- **CPE: 1.33**

- **loads** podem encadear, uma vez que não há dependências
- apenas um conjunto de instruções de controlo de ciclo




```
load (%eax,%edx.0,4)   → t.1a
iaddl t.1a, %ecx.0c    → %ecx.1a
load 4(%eax,%edx.0,4)  → t.1b
iaddl t.1b, %ecx.1a    → %ecx.1b
load 8(%eax,%edx.0,4)  → t.1c
iaddl t.1c, %ecx.1b    → %ecx.1c
iaddl $3,%edx.0        → %edx.1
cmpl %esi, %edx.1      → cc.1
jl-taken cc.1
```

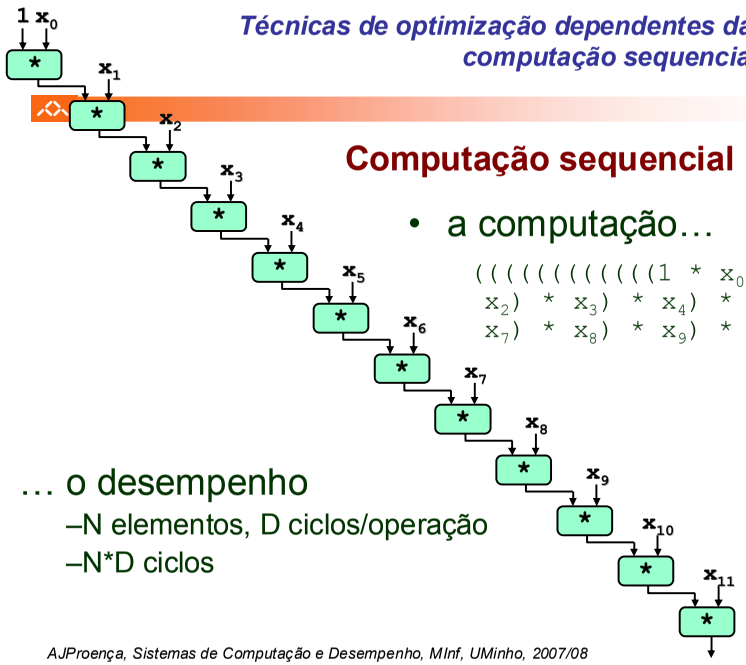


- Desempenho estimado
  - pode completar iteração em 3 ciclos
  - deveria dar CPE de 1.0
- Desempenho medido
  - CPE: 1.33
  - 1 iteração em cada 4 ciclos

Valor do **CPE** para várias situações de *loop unroll*:

Grau de *Unroll*		1	2	3	4	8	16
Inteiro	Soma	2.00	1.50	1.33	1.50	1.25	1.06
Inteiro	Produto	4.00					
fp	Soma	3.00					
fp	Produto	5.00					

- apenas melhora nas somas de inteiros
  - restantes casos há restrições com a latência da unidade
- efeito não é linear com o grau de *unroll*
  - há efeitos subtis que determinam a atribuição exacta das operações



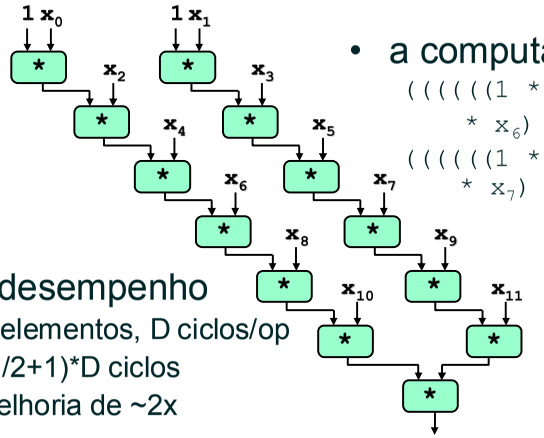
Computação sequencial *versus* ...

- a computação…

$$((((((((((1 * x_0) * x_1) * x_2) * x_3) * x_4) * x_5) * x_6) * x_7) * x_8) * x_9) * x_{10}) * x_{11})$$

… o desempenho
- –N elementos, D ciclos/operação
- –N*D ciclos

Computação sequencial ... *versus* paralela!



- a computação…

$$((((((1 * x_0) * x_2) * x_4) * x_6) * x_8) * x_{10}) * ((((((1 * x_1) * x_3) * x_5) * x_7) * x_9) * x_{11})$$

… o desempenho
- –N elementos, D ciclos/op
- –(N/2+1)*D ciclos
- –melhoria de ~2x

```
void combine6(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int limit = length-1;
  int *data = get_vec_start(v);
  int x0 = 1;
  int x1 = 1;
  int i;
  /* junta 2 elem's de cada vez */
  for (i = 0; i < limit; i+=2) {
    x0 *= data[i];
    x1 *= data[i+1];
  }
  /* completa os restantes elem's */
  for (; i < length; i++) {
    x0 *= data[i];
  }
  *dest = x0 * x1;
}
```

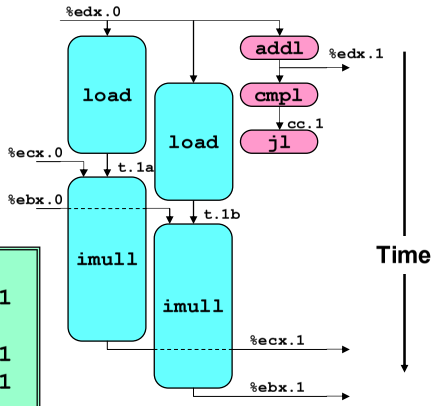
... *versus* paralela!

Optimização 5:

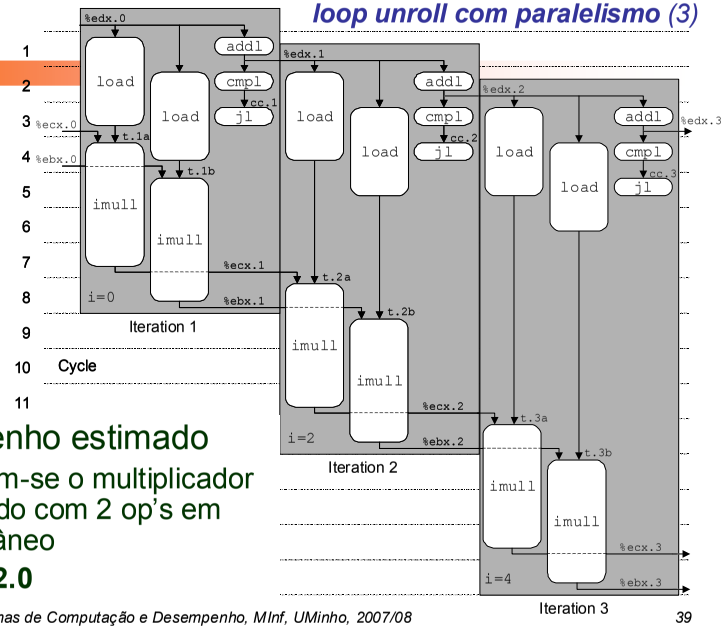
- acumular em 2 produtos diferentes
  - pode ser feito em paralelo, se OP fôr associativa!
- juntar no fim

- Desempenho
  - **CPE: 2.0**
  - melhoria de 2x

- os dois produtos no interior do ciclo não dependem um do outro…
- e é possível encadeá-los
- *iteration splitting*, na literatura




```
load (%eax,%edx.0,4)   → t.1a
imull t.1a, %ecx.0     → %ecx.1
load 4(%eax,%edx.0,4)  → t.1b
imull t.1b, %ebx.0     → %ebx.1
iaddl $2,%edx.0        → %edx.1
cmpl %esi, %edx.1      → cc.1
jl-taken cc.1
```

Desempenho estimado

- mantém-se o multiplicador ocupado com 2 op's em simultâneo
- **CPE: 2.0**

Método	Inteiro		Real (precisão simples)	
	+	*	+	*
Abstract -g	42.06	41.86	41.44	160.00
Abstract -O2	31.25	33.25	31.25	143.00
Move `vec_length`	20.66	21.25	21.15	135.00
Acesso aos dados	6.00	9.00	8.00	117.00
Acum. em temp	2.00	4.00	3.00	5.00
Unroll 4x	1.50	4.00	3.00	5.00
Unroll 16x	1.06	4.00	3.00	5.00
Unroll 2x, paral. 2x	1.50	2.00	2.00	2.50
Unroll 4x, paral. 4x	1.50	2.00	1.50	2.50
Unroll 8x, paral. 4x	1.25	1.25	1.50	2.00
Optimização Teórica	1.00	1.00	1.00	2.00
Pior : Melhor	*39.7*	*33.5*	*27.6*	*80.0*

- Precisa de muitos registos!
  - para guardar somas/produtos
  - apenas 6 registos (p/ inteiros) disponíveis no IA32
    - tb usados como apontadores, controlo de ciclos, …
  - 8 registos de fp
  - quando os registos são insuficientes, temp's vão para a *stack*
    - elimina ganhos de desempenho
      (ver *assembly* em produto inteiro com *unroll* 8x e paralelismo 8x)
  - re-nomeação de registos não chega
    - não é possível referenciar mais operandos que aqueles que o *instruction set* permite
    - … principal inconveniente do *instruction set* do IA32
- Operações a paralelizar têm de ser associativas
  - a soma e multipl de fp num computador não é associativa!
    - (3.14+1e20)-1e20 nem sempre é igual a 3.14+(1e20-1e20)…

- **`combine`**
  - produto de inteiros
  - *unroll* 8x e paralelismo 8x
  - 7 variáveis locais partilham 1 registo (`%edi`)
    - observar os acessos à *stack*
    - melhoria desempenho é comprometida...
    - *register spilling* na literatura

```
.L165:
      imull (%eax),%ecx
      movl -4(%ebp),%edi
      imull 4(%eax),%edi
      movl %edi,-4(%ebp)
      movl -8(%ebp),%edi
      imull 8(%eax),%edi
      movl %edi,-8(%ebp)
      movl -12(%ebp),%edi
      imull 12(%eax),%edi
      movl %edi,-12(%ebp)
      movl -16(%ebp),%edi
      imull 16(%eax),%edi
      movl %edi,-16(%ebp)
    …
      addl $32,%eax
      addl $8,%edx
      cmpl -32(%ebp),%edx
      jl .L165
```


Estrutura do tema Avaliação de Desempenho (IA32)

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

Análise de técnicas de optimização (*s/w*)

- técnicas de optimização de código (indep. máquina)
  - *já visto...*
- técnicas de optimização de código (dep. máquina)
  - dependentes do processador *(já visto...)*
- **outras técnicas de optimização**
  - **na compilação: optimizações efectuadas pelo `Gcc`**
  - **na identificação dos "gargalos" de desempenho**
    - ***code profiling***
    - **uso dum *profiler* para apoio à optimização**
    - **lei de Amdahl**
  - **dependentes da hierarquia da memória**
    - **a localidade espacial e temporal dum programa**
    - **influência da *cache* no desempenho**

Options That Control Optimization

These options control various sorts of optimizations:

-O
-O1
Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function. (...)
With -O, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

-O2
Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. (...) this option increases both compilation time and the performance of the generated code.
-O2 turns on all optional optimizations except for loop unrolling, function inlining, and register renaming.

-O3
Optimize yet more. -O3 turns on all optimizations specified by -O2 and also turns on the -finline-functions and -frename-registers options.

-O0
Do not optimize.

-Os
Optimize for size. -Os enables all -O2 optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.

3.10 Options That Control Optimization

These options control various sorts of optimizations.

`-O`

`-O1` Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.
With '`-O`', the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

`-O2` Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. The compiler does not perform loop unrolling or function inlining when you specify '`-O2`'. As compared to '`-O`', this option increases both compilation time and the performance of the generated code.

`-O3` Optimize yet more. '`-O3`' turns on all optimizations specified by '`-O2`' and also turns on the '`-finline-functions`', '`-funswitch-loops`' and '`-fgcse-after-reload`' options.

`-O0` Do not optimize. This is the default.

`-Os` Optimize for size. '`-Os`' enables all '`-O2`' optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.

Optimizações para código com *arrays* e *loops*:

-funroll-loops
Unroll loops whose number of iterations can be determined at compile time or upon entry to the loop. -funroll-loops implies both -fstrength-reduce and -frerun-cse-after-loop. This option makes code larger, and may or may not make it run faster.

-funroll-all-loops
Unroll all loops, even if their number of iterations is uncertain when the loop is entered. This usually makes programs run more slowly. -funroll-all-loops implies the same options as -funroll-loops,

-fprefetch-loop-arrays
If supported by the target machine, generate instructions to prefetch memory to improve the performance of loops that access large arrays.

-fmove-all-movables
Forces all invariant computations in loops to be moved outside the loop.

-freduce-all-givs
Forces all general-induction variables in loops to be strength-reduced.

Optimizações para inserção de funções em-linha:

-finline-functions
Integrate all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way. If all calls to a given function are integrated, and the function is declared static, then the function is normally not output as assembler code in its own right.

-finline-limit=*n*
By default, gcc limits the size of functions that can be inlined. This flag allows the control of this limit for functions that are explicitly marked as inline (ie marked with the inline keyword ...) *n* is the size of functions that can be inlined in number of pseudo instructions (not counting parameter handling). The default value of *n* is 600. Increasing this value can result in more inlined code at the cost of compilation time and memory consumption. Decreasing usually makes the compilation faster and less code will be inlined (which presumably means slower programs).

Acção

```
gcc -O2 -pg prog. -o prog
./prog
```

- executa como habitual/, mas tb gera o ficheiro `gmon.out`

```
gprof prog
```

- GNU *profiler*: a partir de `gmon.out` gera informação que caracteriza o perfil do programa

Factos

- calcula (aproximadamente) o tempo gasto em cada função
- método para cálculo do tempo (*mais detalhe adiante*)
  - periodicamente (~ cada 10ms) interrompe o programa
  - determina que função está a ser executada nesse momento
  - incrementa o seu temporizador de um intervalo (por ex., 10ms)
- para cada função mantém ainda um contador (nº de vezes que foi invocada)

Uso do `GProf` em 3 passos:

–compilar com indicação explícita (`-pg`)

- ex.: análise do `combine1_sum_int` (vector com 10^7 elementos)

```
gcc -O2 -pg combine1_sum_int.c -o comb1
```

–executar o programa

```
./comb1
```

- vai gerar automaticamente o ficheiro `gmon.out`

–invocar o GProf para analisar os dados em `gmon.out`

```
gprof comb1.exe [ > comb1.txt ]
```

- análise parcial do ficheiro `comb1.txt` a seguir…

Análise da primeira parte de `comb1.txt`:

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
39.33	2.58	2.58				_mcount
38.57	5.11	2.53	20000000	0.00	0.00	get_vec_element
12.65	5.94	0.83				mcount
6.40	6.36	0.42	2	0.21	1.57	combine1
3.05	6.56	0.20	20000002	0.00	0.00	vec_length
0.00	6.56	0.00	2	0.00	0.00	access_counter
0.00	6.56	0.00	1	0.00	0.00	get_counter
0.00	6.56	0.00	1	0.00	0.00	new_vec
0.00	6.56	0.00	1	0.00	0.00	start_counter

Análise em árvore da execução do prog. (em `comb1.txt`):

index	% time	self	children	called	name
		0.42	2.73	2/2	main [2]
[1]	100.0	0.42	2.73	2	combine1 [1]
		2.53	0.00	20000000/20000000	get_vec_element [3]
		0.20	0.00	20000002/20000002	vec_length [4]

					<spontaneous>
[2]	100.0	0.00	3.15		main [2]
		0.42	2.73	2/2	combine1 [1]
		0.00	0.00	1/1	new_vec [11]
		0.00	0.00	1/1	start_counter [12]
		0.00	0.00	1/1	get_counter [10]

		2.53	0.00	20000000/20000000	combine1 [1]
[3]	80.3	2.53	0.00	20000000	get_vec_element [3]

		0.20	0.00	20000002/20000002	combine1 [1]
[4]	6.3	0.20	0.00	20000002	vec_length [4]

		0.00	0.00	1/2	start_counter [12]
		0.00	0.00	1/2	get_counter [10]
[9]	0.0	0.00	0.00	2	access_counter [9]

…

Vantagens

- ajuda a identificar os gargalos de desempenho
- particularmente útil em sistemas complexos com muitos componentes

Limitações

- apenas analisa o desempenho para o conjunto de dados de teste
- a metodologia de medição de tempos é rudimentar
  - apenas usável em programas com tempos de exec > 3 seg

O ganho no desempenho – *speedup* - obtido com a melhoria do tempo de execução de uma parte do sistema, está limitado pela fracção de tempo que essa parte do sistema pode ser utilizada.

$$\textit{Overall speedup} = \frac{\text{Tempo\_exec}_{antigo}}{\text{Tempo\_exec}_{novo}} = \frac{1}{(1-f) + f/s}$$

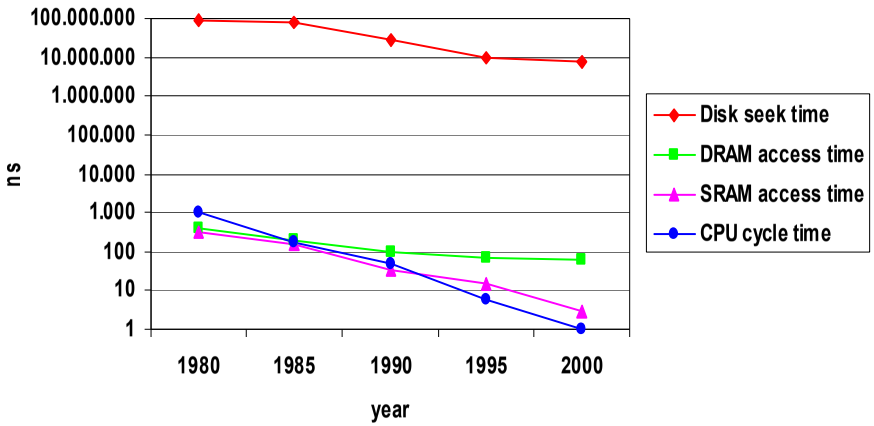
em que **f** – fracção de um programa que é melhorado,
s – *speedup* da parte melhorada

Ex.1
Se 10% de um prog executa 90x mais rápido, então

***Overall speedup* = 1.11**

Ex.2
Se 90% de um prog executa 90x mais rápido, então

***Overall speedup* = 9.09**



Princípio da Localidade:

– programas tendem a reusar dados e instruções próximos daqueles que foram recentemente usados, ou que foram recentemente referenciados por eles

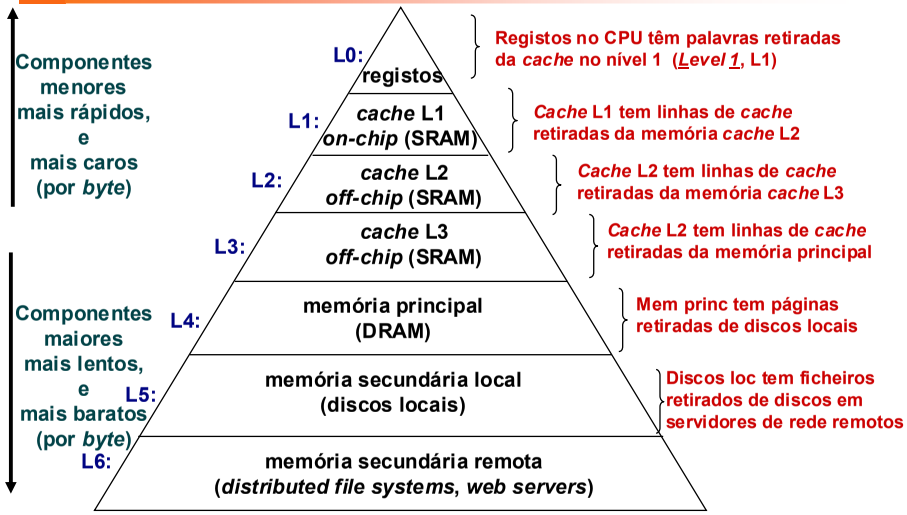
- **Localidade Espacial** : itens em localizações contíguas tendem a ser referenciados em tempos próximos
- **Localidade Temporal** : itens recentemente referenciados serão provavelmente referenciados no futuro próximo

Exemplo da Localidade :

```
sum = 0;
for (i = 0; i < n; i++)
   sum += a[i];
return sum;
```

- **Dados**
  - os elementos do *array* são referenciados em instruções sucessivas: **Localidade Espacial**
  - a variável `sum` é acedida em cada iteração: **Localidade Temporal**
- **Instruções**
  - as instruções são acedidas sequencial/: **Localidade Espacial**
  - o ciclo é repetidamente acedido: **Localidade Temporal**

Organização hierárquica da memória



A cache numa hierarquia de memória: métricas de desempenho

Miss Rate
- percentagem de referências à memória que não tiveram sucesso na *cache* (*misses* / acessos)
- valores típicos:
  - 3-10% para L1
  - pode ser menor para L2 (< 1%), dependendo do tamanho, etc.

Hit Time
- tempo para a *cache* entregar os dados ao processador (inclui o tempo para verificar se a linha está na *cache*)
- valores típicos :
  - 1 ciclo de *clock* para L1
  - 3-8 ciclos de *clock* para L2

Miss Penalty
- tempo extra necessário para ir buscar uma linha após *miss*
  - tipicamente 25-100 ciclos para aceder à memória principal

A cache numa hierarquia de memória: regras na codificação de programas

Referenciar repetidamente uma variável é positivo!
(localidade temporal)

Referenciar elementos consecutivos de um *array* é positivo!
(localidade espacial)

Exemplos:

– ***cache* fria, palavras de 4-*bytes*, blocos (linhas) de *cache* com 4-palavras**

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = 1/4 = 25%

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = 100%

Avaliação de Desempenho no IA32 (6)

Estrutura do tema Avaliação de Desempenho (IA32)

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

Os próximos slides foram adaptados da aula do Prof. Bryant em 2002