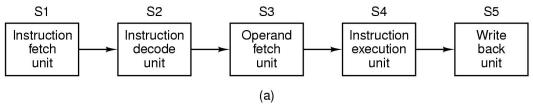
Estrutura do tema Avaliação de Desempenho (IA-32)

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

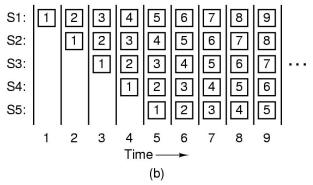
Optimização do desempenho (no *h/w*)

- com introdução de **paralelismo**
  - ao nível do processo (sistemas *multicore*/distribuídos)
  - ao nível da instrução (***I**nstruction **L**evel **P**arallelism*)
    - só nos dados (processadores vectoriais)
    - paralelismo desfasado (*pipeline*)
    - paralelismo "real" (superescalar)
  - no acesso à memória
    - paralelismo desfasado (*interleaving*)
    - paralelismo "real" (maior largura do *bus*)
- com introdução de **hierarquia de memória**
  - *cache* dedicada/partilhada *on/off chip*, UMA/NUMA...

Exemplo de *pipeline*



(a)



(b)

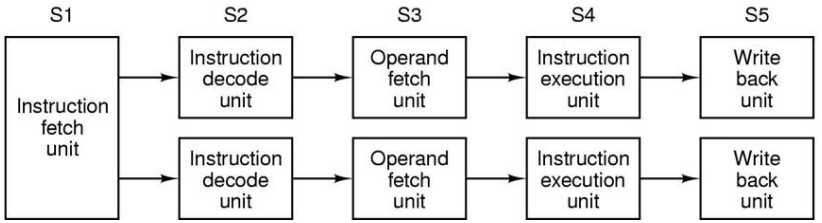
Objectivo

- CPI = 1

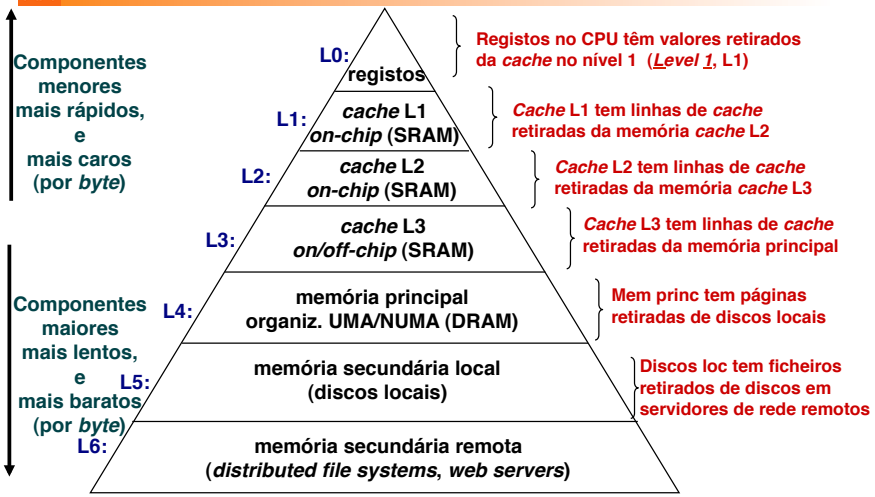
Problemas:

- dependências de dados
- latências nos acessos à memória
- saltos condicionais; propostas de solução para minimizar perdas:
  - executar sempre a instrução "que se segue"
  - usar o historial dos saltos anteriores (1 ou mais bits)
  - executar os 2 percursos alternativos até à tomada de decisão

Exemplo de superescalaridade (nível 2)



Organização hierárquica da memória



Sucesso da hierarquia de memória: o princípio da localidade

Princípio da Localidade:

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

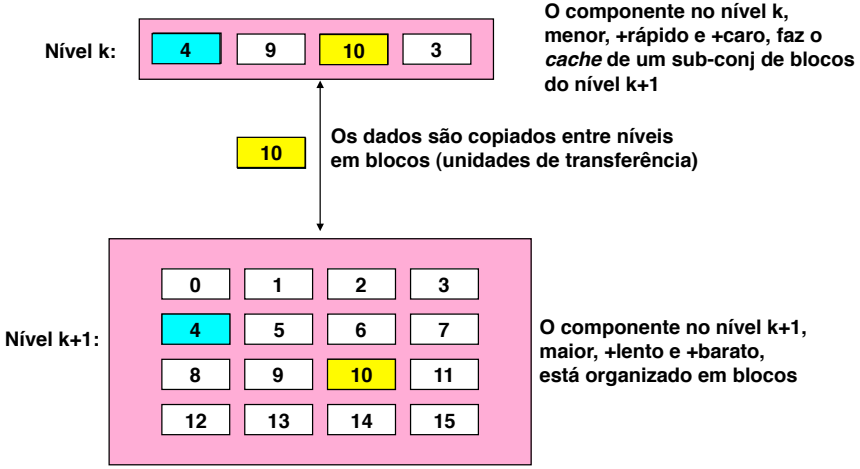
- **Localidade Espacial**: itens em localizações contíguas tendem a ser referenciados em tempos próximos
- **Localidade Temporal**: itens recentemente referenciados serão provavelmente referenciados no futuro próximo

Exemplo da Localidade :

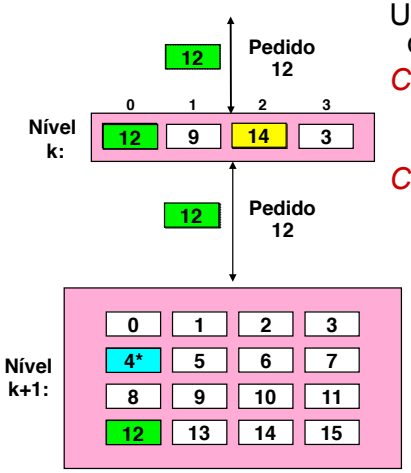
```
sum = 0;
for (i = 0; i < n; i++)
   sum += a[i];
return sum;
```

- **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 sequencialmente: **Localidade Espacial**
  - o ciclo é repetidamente acedido: **Localidade Temporal**

A cache numa hierarquia de memória: introdução



A cache numa hierarquia de memória: conceitos



Um programa pede pelo objecto `d`, que está armazenado num bloco `b`

Cache hit

– o programa encontra `b` na *cache* no nível k. Por ex., bloco 14

Cache miss

– `b` não está no nível k, logo a *cache* do nível k deve buscá-lo do nível k +1. Por ex., bloco 12

– se a *cache* do nível k está cheia, então um dos blocos deve ser substituído (retirado); qual?

- *Placement policy*: onde colocar o novo bloco? Por ex., `b mod 4`
- *Replacement policy*: que bloco deve ser retirado? Por ex., LRU

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 :
  - 1-2 ciclos de *clock* para L1
  - 3-10 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

Referenciar repetidamente uma variável é positivo!

(localidade temporal)

Referenciar elementos consecutivos de um *array* é positivo!

(localidade espacial)

Exemplos:

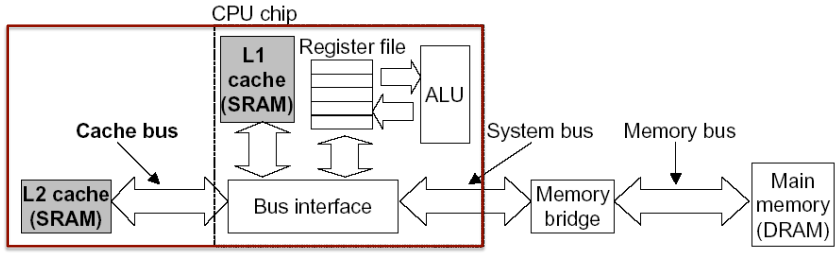
– ***cache* fria, palavras de 4-*bytes*, blocos (linhas) de *cache* com 4-palavras**

```
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%

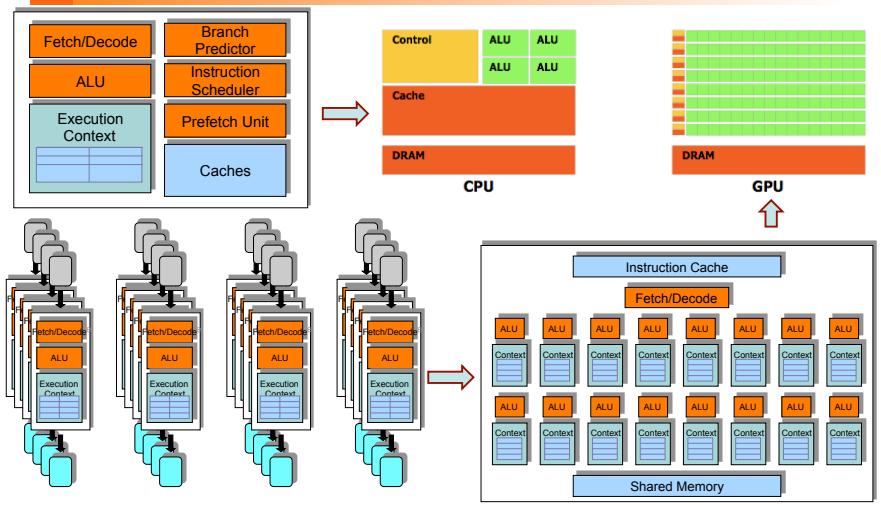
```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

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

Miss rate = 100%

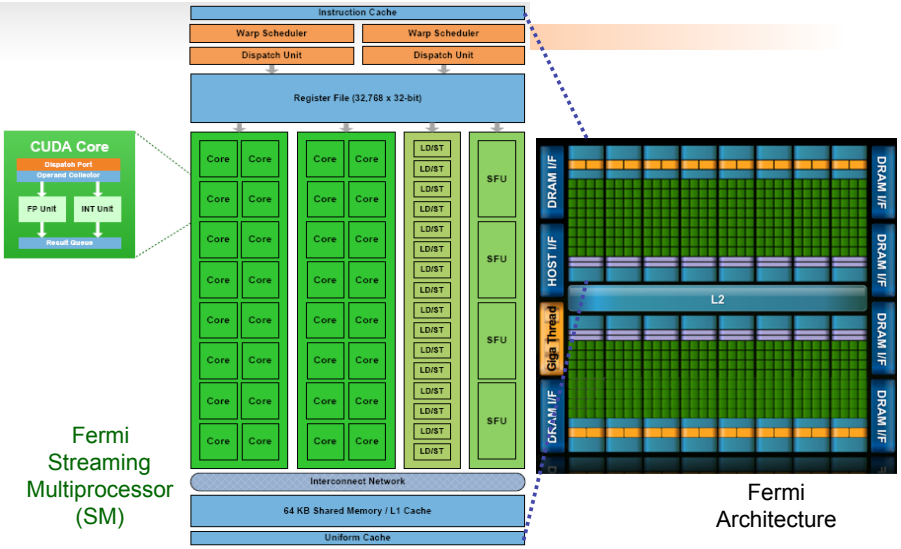


Notas:

- as *caches* L1 de dados e de instruções são normalmente distintas
- as *caches* L2 em *multi-cores* podem ser partilhadas por outras *cores*
- muitos *cores* partilhando uma única memória traz complexidades:
  - manutenção da coerência da informação nas *caches*
  - encaminhamento e partilha dos circuitos de acesso à memória



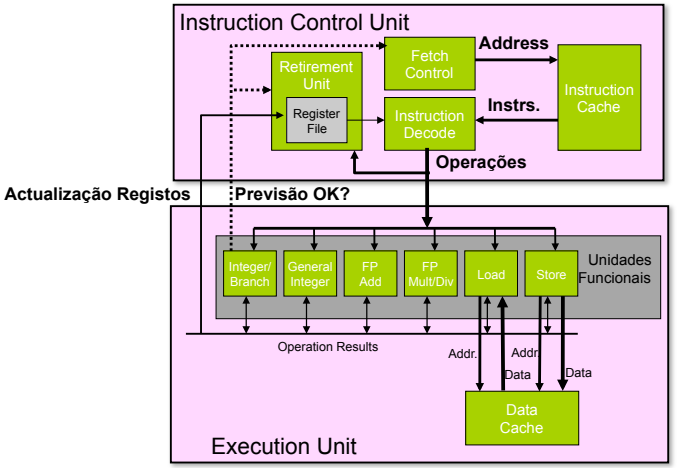
The new NVidia Fermi architecture



Fermi Streaming Multiprocessor (SM)

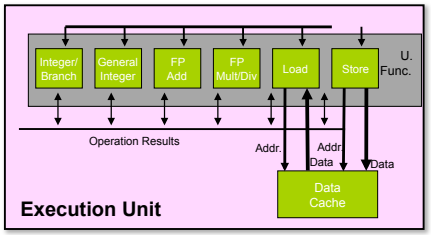
Fermi Architecture

A arquitectura interna dos processadores Intel P6



Algumas potencialidades do Intel P6

- **Execução paralela de várias instruções**
  - 2 `integer` (1 pode ser `branch`)
  - 1 `FP Add`
  - 1 `FP Multiply ou Divide`
  - 1 `load`
  - 1 `store`



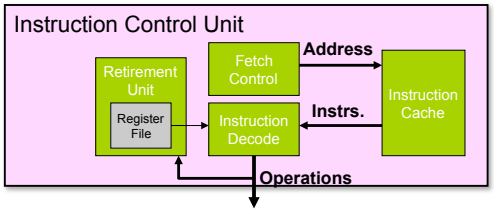
- Algumas instruções requerem > 1 ciclo, mas podem ser encadeadas

Instrução	Latência	Ciclos/Emissão
`Load / Store`	3	1
`Integer Multiply`	4	1
`Integer Divide`	36	36
`Double/Single FP Add`	3	1
`Double/Single FP Multiply`	5	2
`Double/Single FP Divide`	38	38

A unidade de controlo de instruções do Intel P6

Papel da ICU:

- Lê instruções da *InstCache*
  - –baseado no IP + previsão de saltos
  - –antecipa dinamicamente (por *h/w*) se salta/não_salta e (possível) endereço de salto



- Traduz Instruções em *Operações*
  - – *Operações*: designação da Intel para instruções tipo-RISC
  - – instrução típica requer 1–3 operações
- Converte referências a Registos em *Tags*
  - – *Tags*: identificador abstracto que liga o resultado de uma operação com operandos-fonte de operações futuras

Conversão de instruções com registos para operações com tags

- **Versão de `combine4`**
  - tipo de dados: *inteiro* ; operação: *multiplicação*

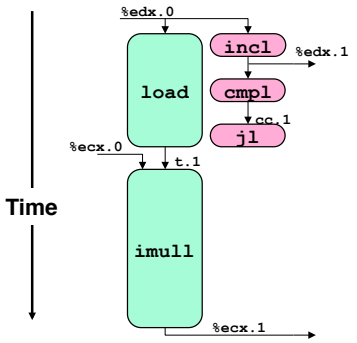
```
.L24:                           # Loop:
  imull (%eax,%edx,4),%ecx      # t *= data[i]
  incl  %edx                    # i++
  cmpl  %esi,%edx               # i:length
  jl    .L24                    # if < goto Loop
```

- **Tradução da 1ª iteração**

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

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

```
load (%eax,%edx.0,4)  ➔ t.1
imull t.1, %ecx.0     ➔ %ecx.1
incl %edx.0           ➔ %edx.1
cmpl %esi, %edx.1     ➔ cc.1
jl    -taken cc.1
```

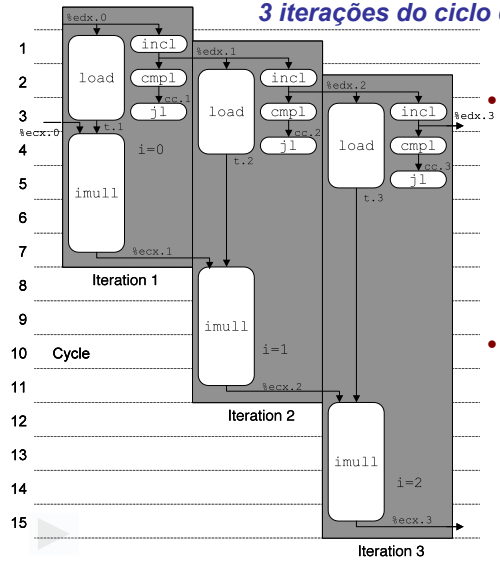

Análise visual da execução de instruções no P6: 1 iteração do ciclo de produtos em `combine`




```
load (%eax,%edx.0,4)  ➔ t.1
imull t.1, %ecx.0     ➔ %ecx.1
incl %edx.0           ➔ %edx.1
cmpl %esi, %edx.1     ➔ cc.1
jl    -taken cc.1
```

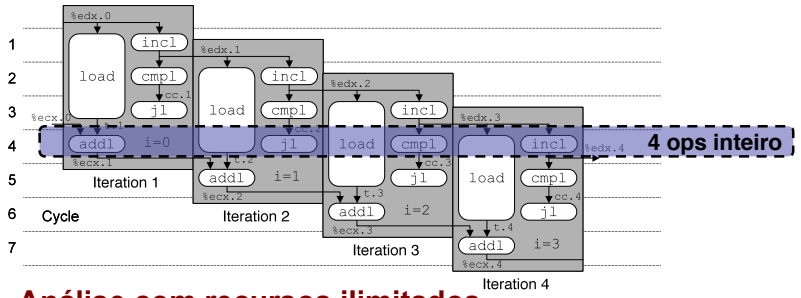
- **Operações**
  - a posição vertical dá uma indicação do tempo em que é executada
    - uma operação não pode iniciar-se sem os seus operandos
  - a altura traduz a latência
- **Operandos**
  - os arcos apenas são representados para os operandos que são usados no contexto da *execution unit*

Análise visual da execução de instruções no P6: 3 iterações do ciclo de produtos em `combine`

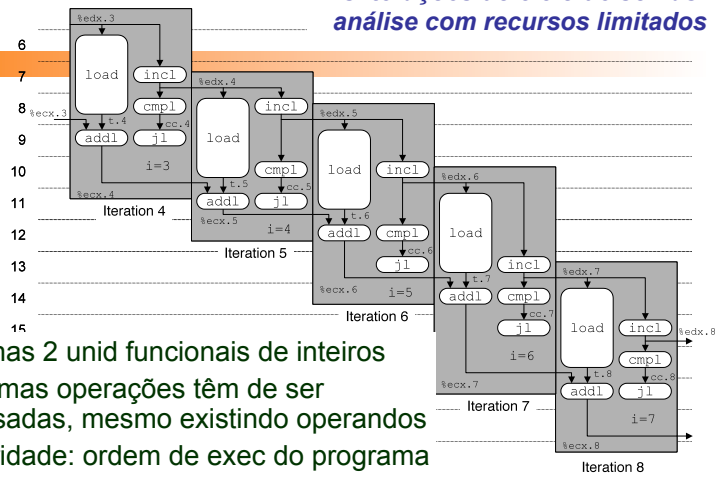


- **Análise com recursos ilimitados**
  - execução paralela e encadeada de operações na EU
  - execução *out-of-order* e especulativa
- **Desempenho**
  - factor limitativo: latência da multipl. de inteiros
  - CPE: 4.0

Análise visual da execução de instruções no P6: 4 iterações do ciclo de somas em `combine`



- **Análise com recursos ilimitados**
- **Desempenho**
  - pode começar uma nova iteração em cada ciclo de *clock*
  - valor teórico de CPE: 1.0
  - requer a execução de 4 operações c/ inteiros em paralelo



- apenas 2 unid funcionais de inteiros
- algumas operações têm de ser atrasadas, mesmo existindo operandos
- prioridade: ordem de exec do programa

- **Desempenho**
  - CPE expectável: 2.0

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

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

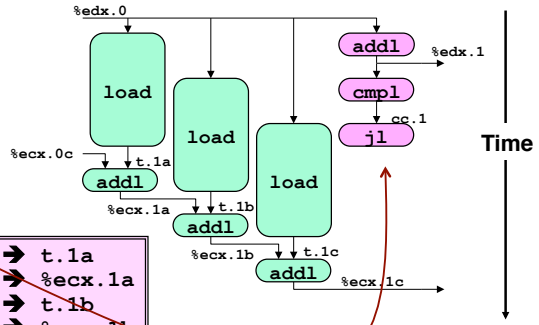
- técnicas de optimização de código (indep. máquina)
  - *já visto...*
- **técnicas de optimização de código (dep. máquina)**
  - análise sucinta de um CPU actual, P6 (*já visto...*)
  - ***loop unroll* e *inline functions***
  - **identificação de potenciais limitadores de desempenho**
  - dependentes 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;
}
```

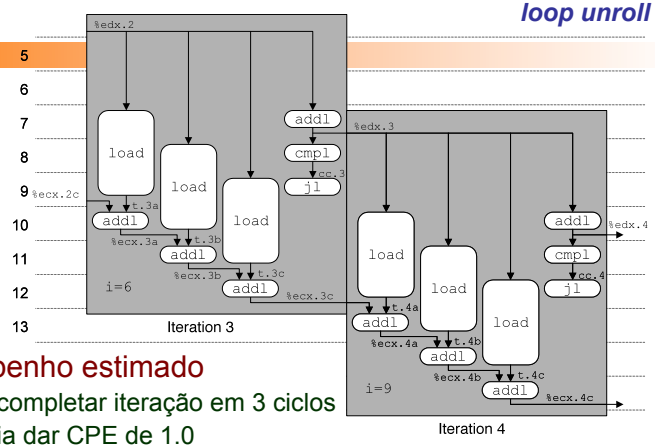
Optimização 4:

- 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
```

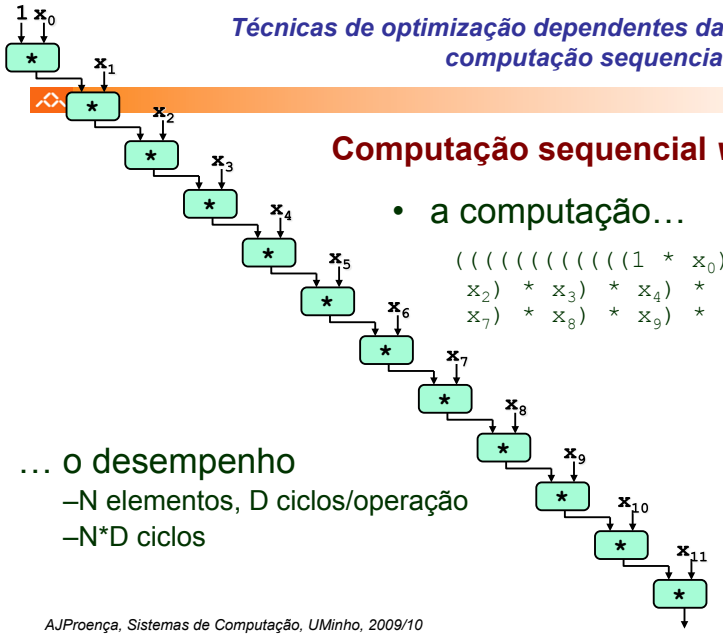



- Desempenho estimado
  - pode completar iteração em 3 ciclos
  - deveria dar CPE de 1.0
- Desempenho medido
  - CPE: 1.33
  - 1 iteração em cada 4 ciclos

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



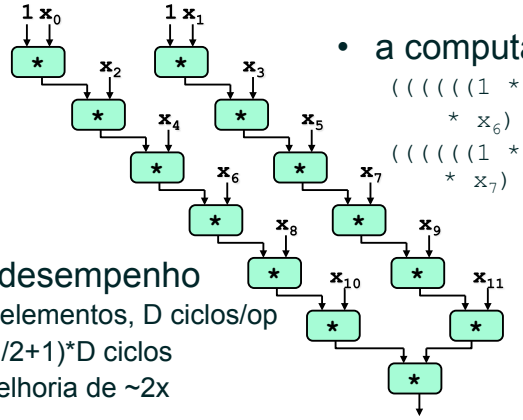
Computação sequencial *versus* ...

- 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…

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

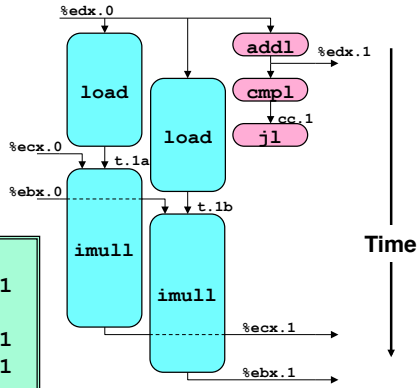
… o desempenho
- –N elementos, D ciclos/op
- –(N/2+1)*D ciclos
- –melhoria de ~2x


```
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;
}
```

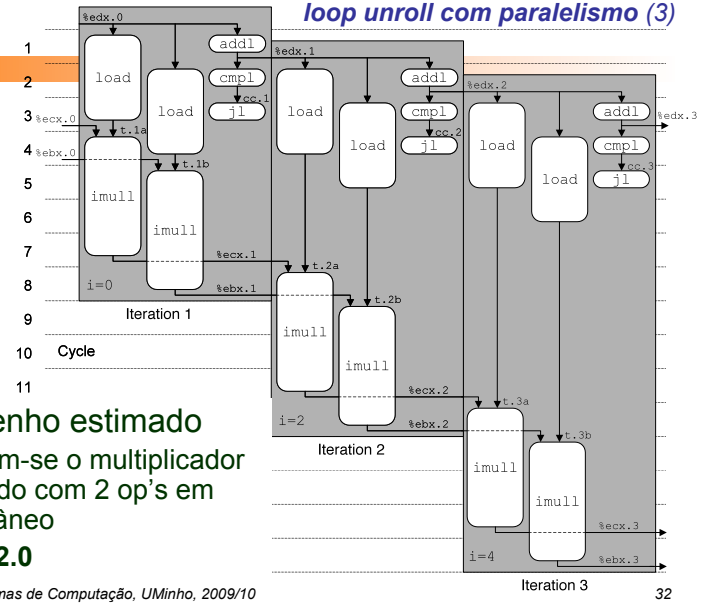
... *versus* paralela!

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 (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

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**

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.

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

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**