

Arquitetura de Computadores

Licenciatura em Engenharia Informática

Optimização de Desempenho
(técnicas dependentes da máquina)

João Luís Ferreira Sobral

Optimização de Desempenho

- **Técnicas independentes da máquina**
 - Simplificação de expressões
 - Utilização de um único cálculo de uma expressão utilizada em vários locais
 - Redução do nº de vezes que um cálculo é efectuado
 - Alocação de variáveis a registos
 - Expansão em linha de funções/procedimentos
- **Técnicas dependentes da máquina**
 - Apontadores
 - Desdobramento de ciclos
 - Desdobramento de ciclos em paralelo

Optimização de Desempenho

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

```
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;
}
```

- **CPE de 2,00 (após várias optimizações, compilado com -O2)**

Optimização de Desempenho

- **Vector ADT**

- Extensão para tipos genéricos (*templates* em C++, pseudo-código)

```
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;
}
```

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

- **Tipos de dados <T>:**
 - **int, float, double**
- **Operações <OP> / elemento neutro <IDENT>**
 - ‘+’ / 0
 - ‘x’ / 1

Optimização de Desempenho

- **vector ADT**
 - Optimizações independentes da máquina (optimização -O2)
 - Comparação do CPE das várias versões
 - Ganho quando os valores são acumulados em variáveis temporárias (evitando acessos à memória)
 - Combine 3 (1 LD + 1 ST)

	Method	Integer		Floating Point	
		+	*	+	*
combine1	Abstract -g	29.02	29.21	27.40	27.36
combine1	Abstract -O2	12.00	12.00	12.00	13.00
combine2	Move vec_length	8.03	8.09	10.09	12.09
combine3	data access	3.00	3.00	3.00	5.03
combine4	Accum. in temp	2.00	3.00	3.00	5.00

```
.L24:                                # Loop:  
    addl (%eax,%ecx,4),%edx      # aux += data[i]  
    movl %edx,(%ebx)            # *dest = aux;  
    inc %ecx                   # i++  
    cmpl %ecx,%edi            # i:length  
    jl .L24                   # if < goto Loop
```

Optimização de Desempenho

- **Vector ADT**

- Utilização de apontadores em alternativa a referências a vectores

```
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 = sum + data[i];
    *dest = sum;
}
```



```
void combine4p(vec_ptr v, int *dest)
{
    int length = vec_length(v);
    int *data = get_vec_start(v);
    int *dend = data+length;
    int sum = 0;
    while (data < dend) {
        sum += *data;
        data++;
    }
    *dest = sum;
}
```

- CPE = 3,00 (!!!)
 - O compilador gera código mais eficiente

4 instruções em 2 ciclos

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

4 instruções em 3 ciclos

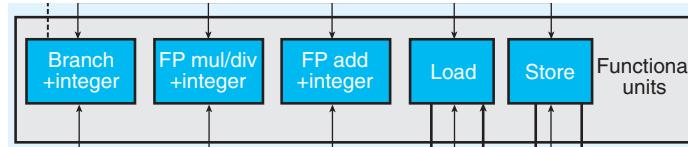
```
.L30:                      # Loop:
    addl (%eax),%ecx        # sum += *data
    addl $4,%eax            # data ++
    cmpl %edx,%eax          # data:dend
    jb .L30                 # if < goto Loop
```

Optimização de Desempenho

- **Características do i7 (Nehalem)**

- **Unidades de execução**

- 1 load
 - 1 store
 - 3 integer
 - One can be branch
 - One can be FP Addition
 - One can be FP Multiplication or Division



- **Latência e Instruções por ciclo (IPC = 1/CPI)**

Instruction	RAW Latency	IPC
Load / Store	3	1
Integer	1	3
Integer Multiply	3	1
Integer Divide	11 a 21	1/5 a 1/13
Double/Single FP Add	3	1
Double/Single FP Multiply	5	1
Double/Single FP Divide	10 a 23	1/6 a 1/19

Optimização de Desempenho

- **Transformação de instruções na arquitetura Intel**

- Cada instrução tipicamente gera, na fase de ID, 1-3 instruções primitivas (designadas por uOp)
 - Operações aritméticas sobre memória geram 1 *load* + 1 operação
- Os registos são convertidos em etiquetas
 - Elimina as falsas dependências (WAR e WAW)
 - Serve de base para a renomeação de registos

```
.L24:
```

```
imull (%eax,%edx,4),%ecx
incl %edx
cmpl %esi,%edx
j1 .L24
```

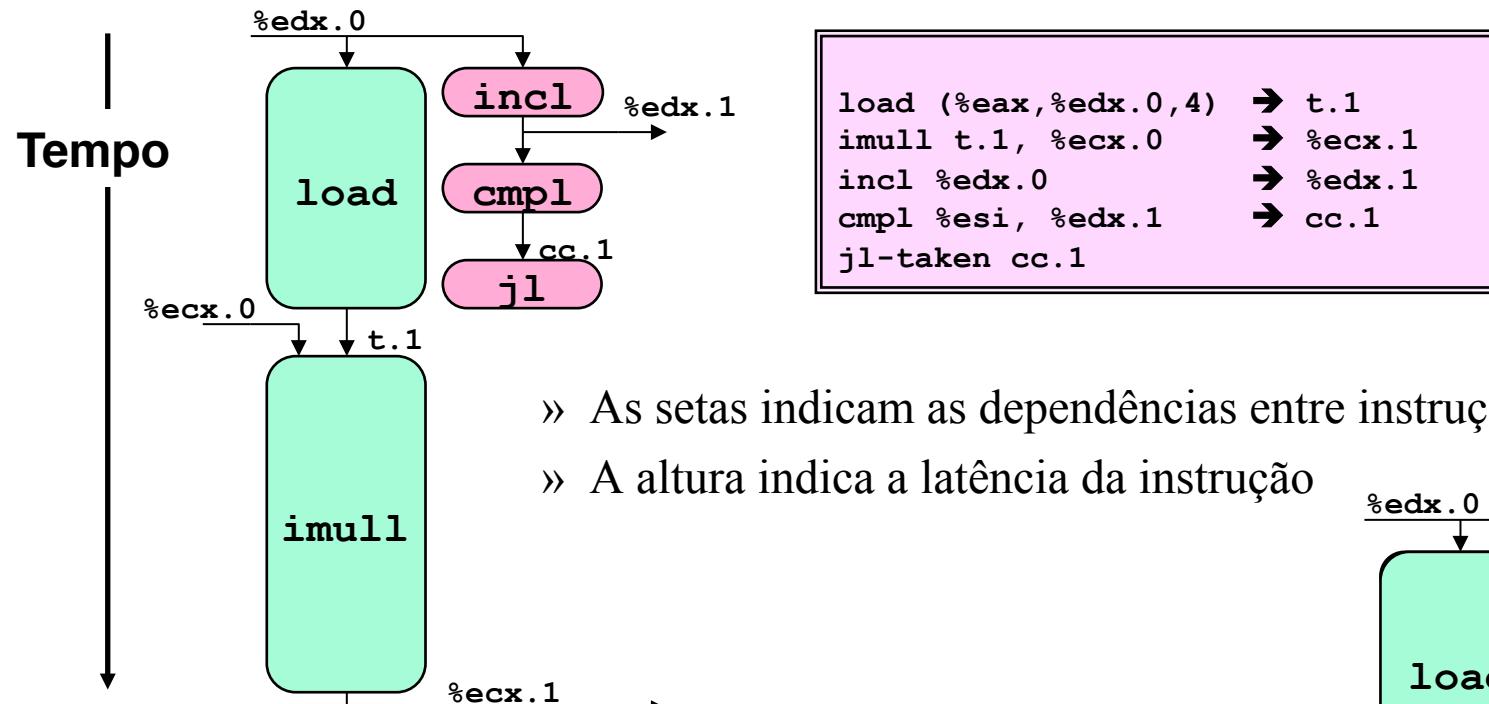


```
load (%eax,%edx.0,4)    → t.1
imull t.1, %ecx.0        → %ecx.1
incl %edx.0               → %edx.1
cmpl %esi, %edx.1        → cc.1
j1-taken cc.1
```

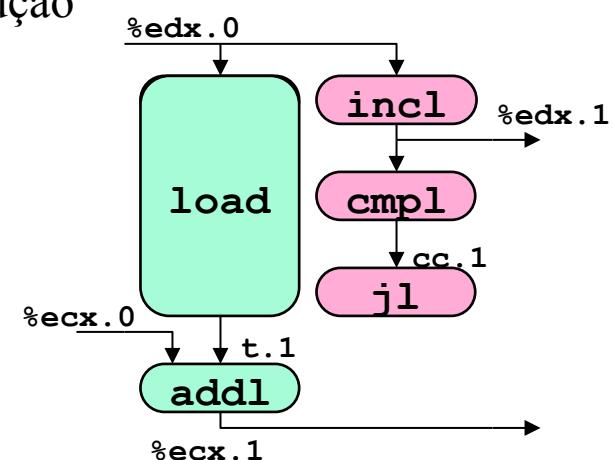
- `load` lê o valor da memória para a “tag” `t.1`
 - Encaminhamento de dados entre `load` e `imull`
- O valor de `eax` é fixo durante o ciclo
 - copiado do banco de registos durante a descodificação
- Cada escrita em registo gera uma nova “tag” do registo
 - `ecx.0, ecx.1, ...` => neste caso, corresponde ao valor em cada iteração

Optimização de Desempenho

- Visualização da execução de instruções

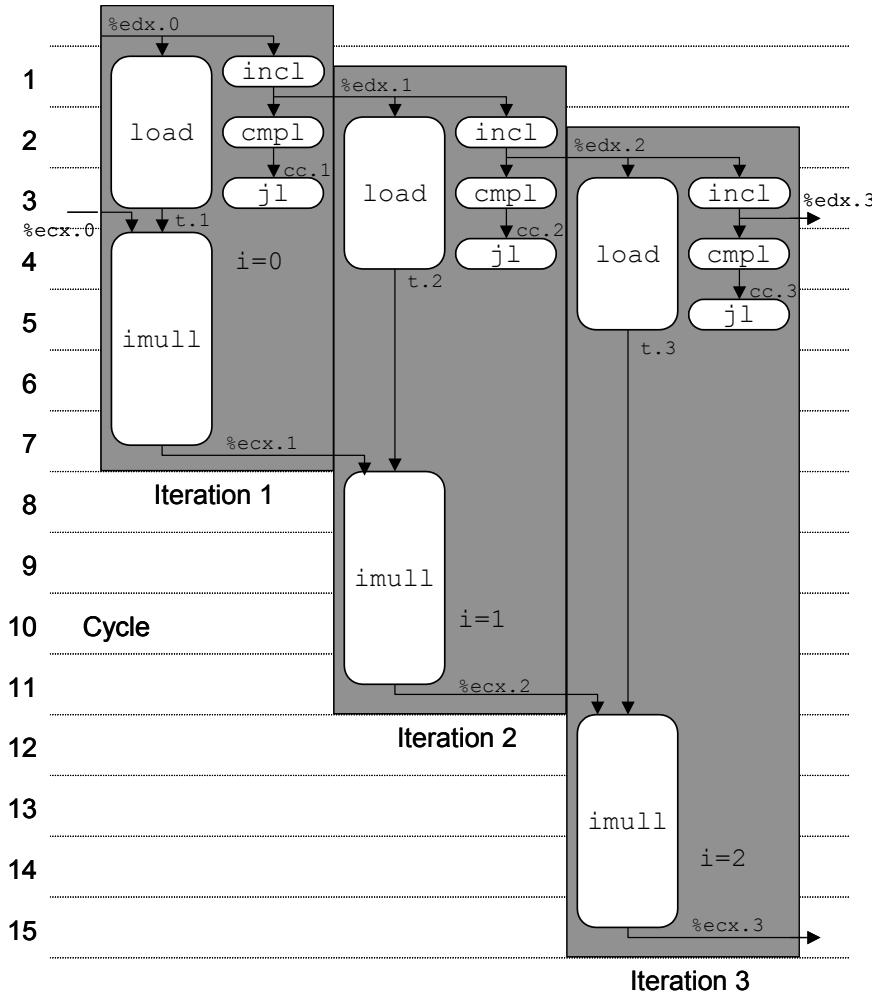


- » As setas indicam as dependências entre instruções
- » A altura indica a latência da instrução



Optimização de Desempenho

- Execução de três iterações (produto dos elementos)



Análise para recursos ilimitados

- Operação começa assim que os recursos estão disponíveis
- Execução simultânea de instruções de várias iterações

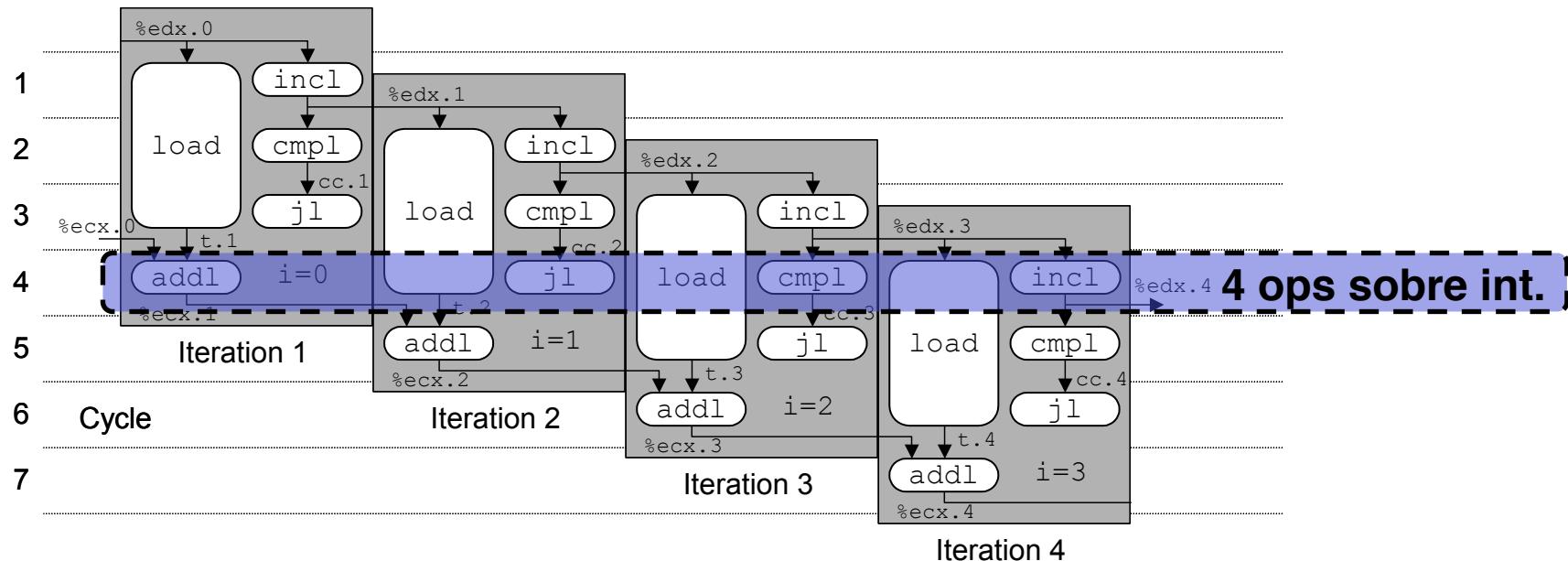
Desempenho

- O factor limitativo é a latência da multiplicação de inteiros (4 ciclos neste exemplo)

$$\text{CPE} = 4,0$$

Optimização de Desempenho

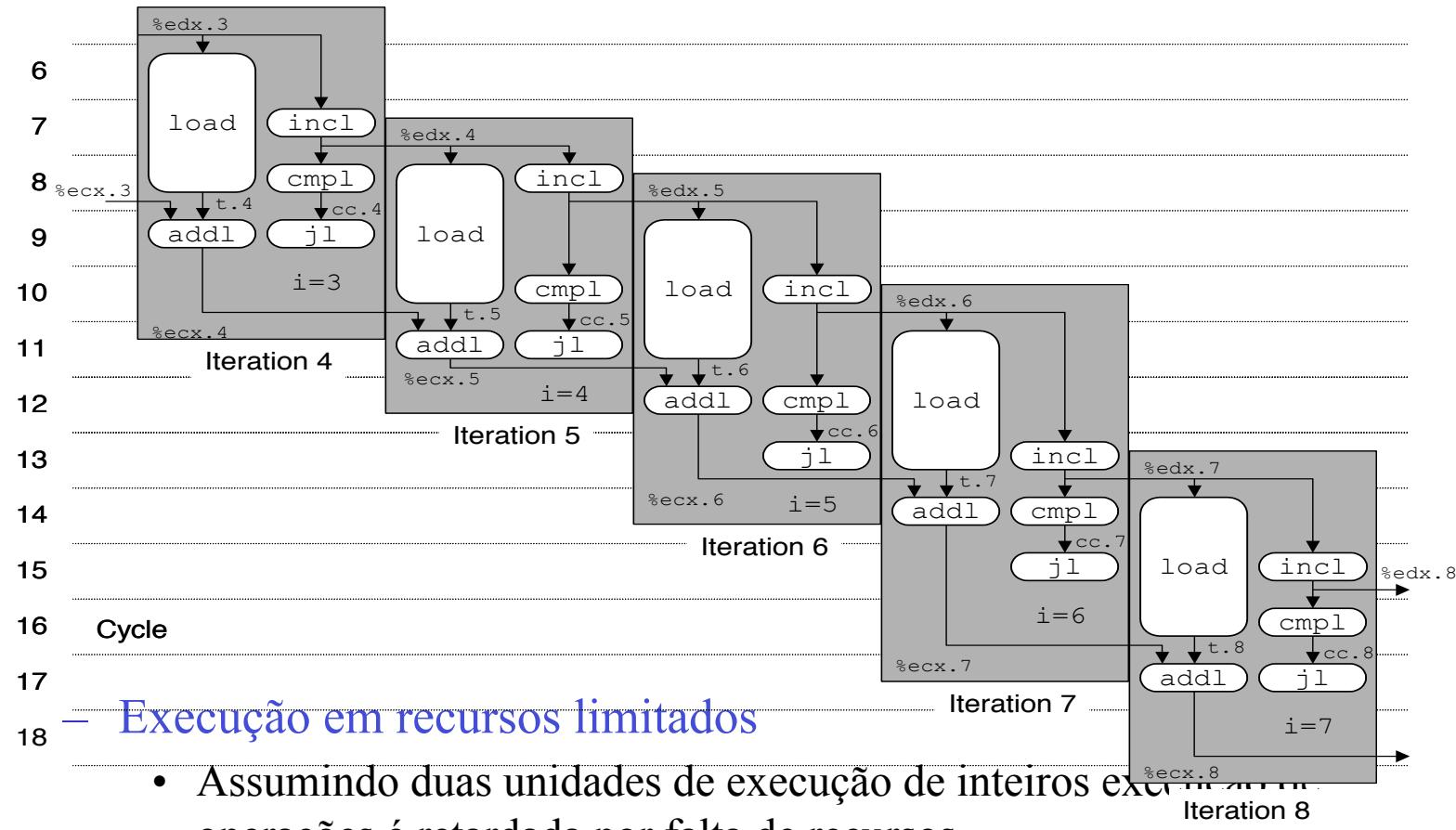
- Execução de quatro iterações (soma dos elementos)



- Análise para recursos ilimitados
 - Inicia uma iteração por ciclo
 - CPE = 1.00 em condições ideais
 - Requer a capacidade para executar 4 operações sobre inteiros

Optimização de Desempenho

- Execução de quatro iterações (soma dos elementos)



Optimização de Desempenho

- **Desdobramento de ciclos**

- Combina múltiplas iterações num só ciclo
- Amortiza a sobrecarga dos ciclos
- Requer código adicional no fim do ciclos para lidar com as restantes iterações
- CPE = 1,00 (desdobramento de grau 4)

- Exemplo para grau 3:

```
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 = sum + data[i];
    *dest = sum;
}
```

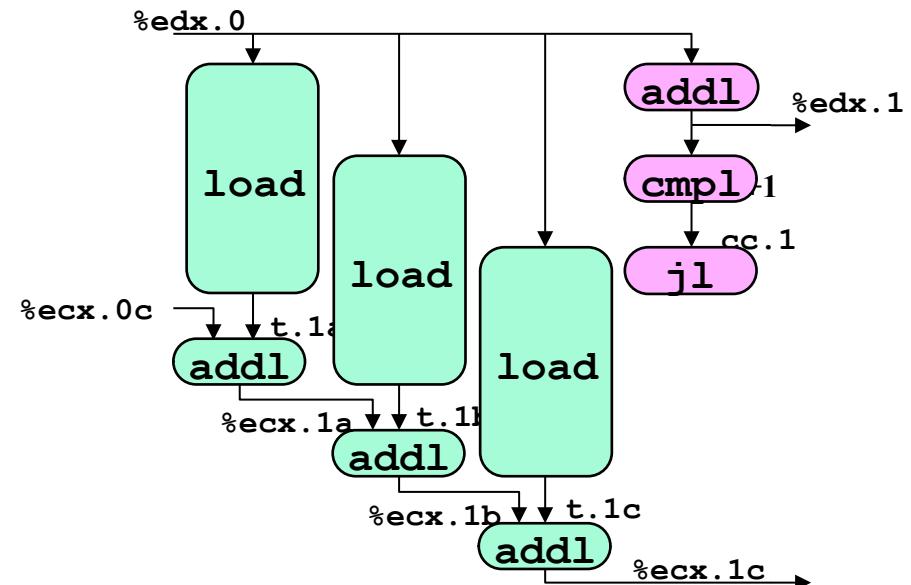


```
void combine5(vec_ptr v, int *dest)
{
    int i;
    int length = vec_length(v);
    int limit = length-2;
    int *data = get_vec_start(v);
    int sum = 0;
    /* Combine 3 elements at a time */
    for (i = 0; i < limit; i+=3) {
        sum += data[i] + data[i+1] + data[i+2];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        sum += data[i];
    }
    *dest = sum;
}
```

Optimização de Desempenho

- Visualização do desdobramento dos ciclos (3x)

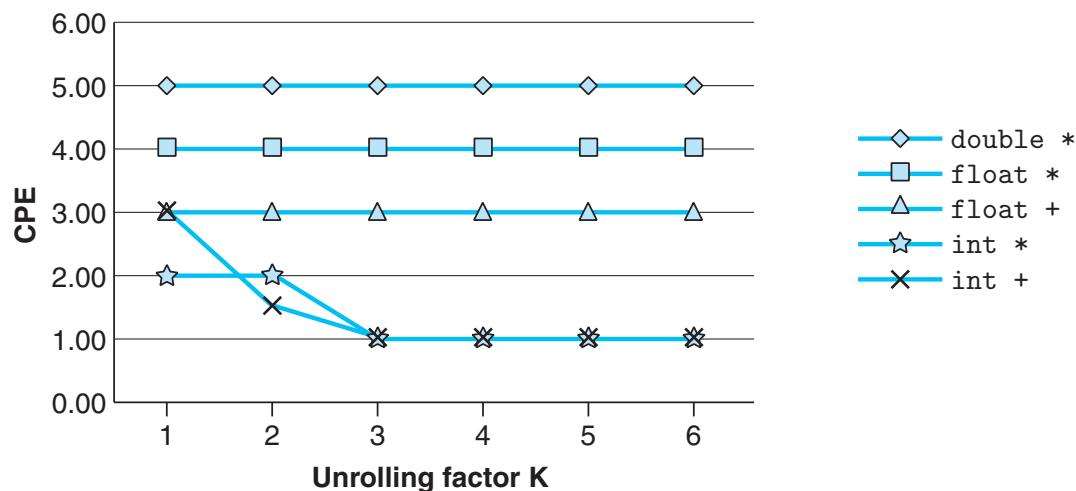
```
load (%eax,%edx.0,4)    → t.1a
iaddl t.1a, %ecx.0c     → %ecx.1a
load 4(%eax,%edx.0,4)   → t.1b
iaddl t.1b, %ecx.1a     → %ecx.1b
load 8(%eax,%edx.0,4)   → t.1c
iaddl t.1c, %ecx.1b     → %ecx.1c
iaddl $3,%edx.0          → %edx.1
cmpl %esi, %edx.1       → cc.1
jl-taken cc.1
```



- Não há dependências entre os “load”
- Apenas um conjunto de operações para controlo do ciclo

Optimização de Desempenho

- Impacto do desdobramento de ciclos no desempenho



- Apenas as operações sobre inteiros beneficiam com o desdobramento
 - Nos outros casos a limitação é a latência das unidades funcionais
 - Exceção para o int “x”!!!! (ver slides seguintes)
- O impacto do desdobramento não é linear
 - Existem vários factores que causam impacto no escalonamento

Optimização de Desempenho

- **Computação série versus paralela**

- Série

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

- Desempenho

- N elementos, D ciclos por operação, N*D ciclos

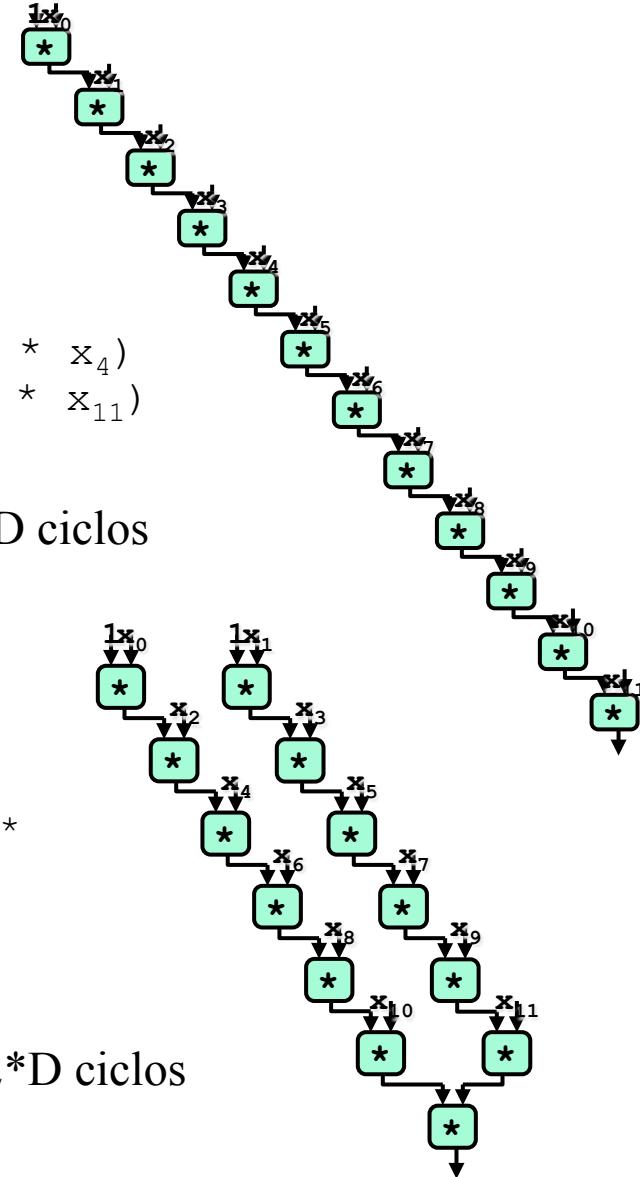
- Paralela

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

- Desempenho

- N elementos, D ciclos por operação, N/2*D ciclos

- Ganho de $\sim 2X$



Optimização de Desempenho

- **Desdobramento de ciclos em paralelo**
 - Acumula os produtos em duas variáveis
 - Junta os resultados no fim do ciclo

```
void combine5(vec_ptr v, int *dest)
{
    int i;
    int length = vec_length(v);
    int limit = length-1;
    int *data = get_vec_start(v);
    int acc = 1;
    /* Combine 3 elements at a time */
    for (i = 0; i < limit; i+=2) {
        acc *= data[i] * data[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc *= data[i];
    }
    *dest = acc;
}
```

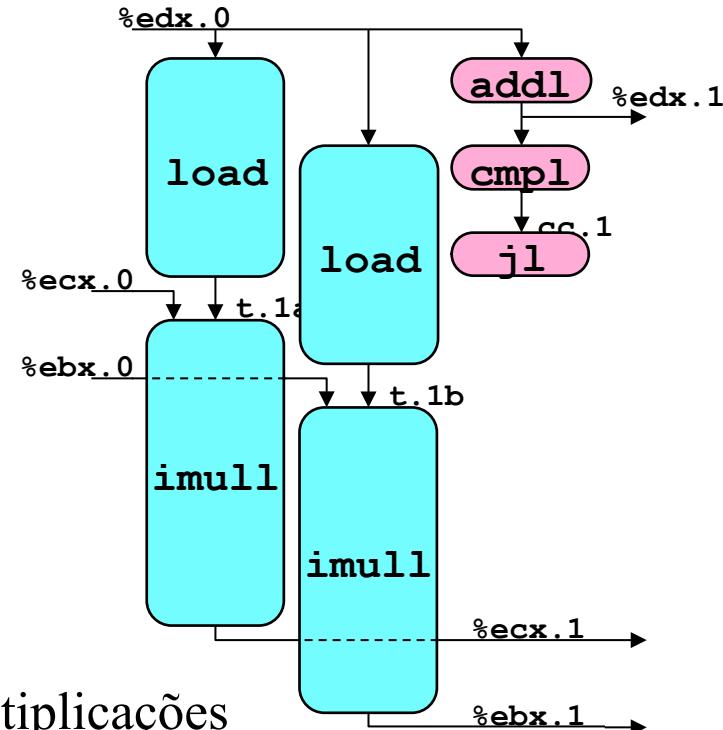


```
void combine6(vec_ptr v, int *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    int *data = get_vec_start(v);
    int x0 = 1;
    int x1 = 1;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x0 *= data[i];
        x1 *= data[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 *= data[i];
    }
    *dest = x0 * x1;
}
```

Optimização de Desempenho

- Visualização do desdobramento dos ciclos em paralelo

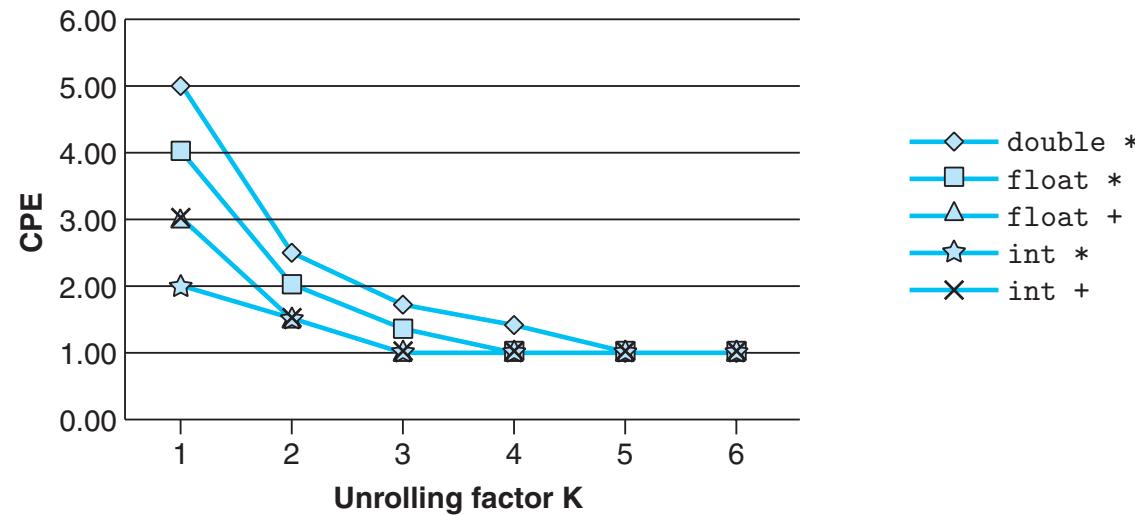
```
load (%eax,%edx.0,4)    → t.1a
imull t.1a, %ecx.0      → %ecx.1
load 4(%eax,%edx.0,4)   → t.1b
imull t.1b, %ebx.0      → %ebx.1
iaddl $2,%edx.0          → %edx.1
cmpl %esi, %edx.1       → cc.1
jl-taken cc.1
```



- Não há dependências entre as multiplicações
 - Podem executar de forma encadeada
- Limitação =>
 - número de registos disponíveis no ISA
 - renomeação de registos não ajuda

Optimização de Desempenho

- **Impacto do desdobramento de ciclos em paralelo**



- Permite atingir o limite teórico do desempenho das unidades funcionais
 - Exceção para o caso de soma de inteiros (3 por ciclo)

Optimização de Desempenho

- **Divisão de registos (*register spilling*)**

- Variáveis locais partilham os registos
- Exemplo: Desdobrar o ciclo 8 vezes e calcular 8 multiplicações em paralelo
 - As variáveis locais são armazenadas na pilha e carregadas para %edi
- x86-64 adicionou 8 registos ao ISA

Machine	Degree of unrolling					
	1	2	3	4	5	6
IA32	2.12	1.76	1.45	1.39	1.90	1.99
x86-64	2.00	1.50	1.00	1.00	1.01	1.00

```
.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
j1 .L165
```

Optimização de Desempenho

- **Resumo**

- Fases de desenvolvimento
 1. Selecionar o melhor algoritmo
 - Utilizar a análise de complexidade para comparar algoritmos
 2. Escrever código legível e fácil de gerir
 3. Eliminar bloqueadores de optimizações
 4. Medir o perfil de execução
 - Optimizar as partes críticas para o desempenho
 - » Operações repetidas muitas vezes (e.g., ciclos interiores)
- Código com optimizações é mais complexo de ler, manter e de garantir a correção

Common compiler optimizations

- Loops
 - Identify **induction variables** that are increased or decreased by a fixed amount on every iteration of a loop (e.g., $j = i*4 + 1 \Rightarrow j += 5$)
 - **Fission** - break a loop into multiple loops but each taking only a part of the loop's body
 - **Fusion** – combine loops to reduce loop overhead
 - **Inversion** - changes a standard *while* loop into a *do/while*
 - **Interchange** - exchange inner loops with outer loops
 - **Loop-invariant code motion**
 - **Loop unrolling** - duplicates the body of the loop multiple times
 - **Loop splitting** - breaks into multiple loops which have the same bodies but iterate over different contiguous portions of the index range
- Data flow
 - **Common sub-expression elimination/sharing**
 - **Reduction in strength** - expensive operations are replaced with less expensive operations
 - **Constant folding** - replaces expressions of constants (e.g., $3 + 5$) with their final value (8)
 - **Dead store elimination** - removal of assignments to variables that are not read

Common compiler optimizations

- Code generation
 - **Register allocation** - most frequently used variables are kept in processor registers
 - **Instruction selection** – select one of several different ways of performing an operation
 - **Instruction scheduling** – avoid pipeline stalls
 - **Re-materialization** - recalculates a value instead of loading it from memory
- Other optimizations
 - **Bounds-checking elimination**
 - **Code-block reordering** – alters the order of basic blocks
 - **Dead code elimination**
 - **Inline expansion** - insert the body of a procedure inside the calling code
- Limitations
 - Memory aliasing & side effects of functions
 - Compilers do not typically improve the algorithmic complexity
 - A compiler typically only deals with a part of a program at a time
 - Time overhead of compiler optimisations