1byte = 8 bits

2GHz = 2×10^9 Hz

• Read after Write (RAW): a instrução lj que depende de li tem que ser executada depois desta última.
• Write After Read (WAR): a instrução lj depende de li pode ser executada em simultâneo com esta última mas não antes
• Write After Write (WAW): a instrução lj depende de li e tem de ser executada depois desta última

• #pragma omp single: faz com que apenas uma única thread execute o código associado enquanto os outros ficam bloqueados e esperam (a não ser que nowait seja usado)
• #pragma omp master: faz com que a thread 0 (thread master) execute o bloco enquanto os outros threads continuam o resto do código
• #pragma omp parallel for: paraleliza automaticamente o loop for dividindo iterações por threads para que o for seja executado em paralelo (Δ a variável do 1º for é privada)
• #pragma omp parallel for reduction (:): combina variáveis parciais calculadas por múltiplos threads em uma única variável global. Permite que cada thread tenha a sua própria variável que será combinada no final por uma operação específica (+, *, max, min, etc.)
• #pragma omp atomic: garante que operações simples em memória partilhada sejam executadas de forma indivisível e sem interrupção (previne race conditions). O #pragma omp critical é preciso no entanto torna a execução sequencial.
• schedule(static): divide o loop de forma fixa e previsível; usa quando todas as iterações demoram o mesmo tempo
• schedule(dynamic): divide o loop sobre demanda; usa quando as iterações têm tempos variáveis para obter um balanceamento de carga dinâmico.

$T_{exec} = \frac{CPI \times \#I}{f}$

$CPI = \frac{\#CC}{\#I}$ $\quad CPI_{processador\ p} = \frac{\#CC_p}{\#I_p}$

$CPI_{global} = \frac{\sum_{p=0}^{P-1} \#CC_p}{\sum_{p=0}^{P-1} \#I_p}$

$CPI_{percepcionado} = \frac{max(\#CC_p)}{\sum_{p=0}^{P-1} \#I_p}$

• SpeedUp: $S_p = T_1 / T_p$
• p: nº de processadores; T_1: tempo de execução p=1, T_p: tempo de execução com p processadores

• Eficiência:
$E_p = S_p / p$
• p: número de processadores; S_p: SpeedUp com p processadores

• Quando falamos de ciclos:
#I = nº de vezes que o ciclo se repete × o nº de instruções que cada ciclo tem

Código não vetorizável:
- uncountable loop (for i=0; a[i]!=0; i++)
- estruturas condicionais grandes (if else, etc.)
- invocações de funções dentro do ciclo
- dependência RAW

```
Ex: for(i=1; i<size; i++){
c = a[i-1]*2; a[i]=(c>0 ? c:1);
} d = i-(i-1) = 1
a[1] = a[0]*2:1   RAW
a[2] = a[1]*2:1   não vetorizável
Ex: for(i=size-1; i>0; i--){
c = a[i-1]*2; a[i] = (c>0 ? c:1);
} d = -(i-(i-1)) = -i+i-1 = -1
a[size-1] = a[size-2]*2:1   WAR
a[size-2] = a[size-3]*2:1   vetorizável
```

• O ganho de desempenho obtido com a vetorização do código relativamente à respetiva versão escalar deve-se à diminuição do número total de instruções executadas (#I).
• As unidades de processamento gráfico minimizam o impacto dos acessos à memória no desempenho dos programas comutando rapidamente entre grupos de threads ocupando o tempo de acesso de algumas threads com execução de outras

• O código "for(i=1; i<S; i+=1) a[i] = a[i-1]/(i>10 ? 2:3);" não vetoriza devido a uma dependência RAW entre iterações
• Considere o código: #define size ... float a[size]; main(){ int i; for(i=1; i<Size; i++){ a[i] = powf(a[i], 2)+10.0/a[i]-a[i-1]; }}
• O código não vetoriza porque tem uma dependência RAW (a[i] = ... a[i-1]) e utiliza funções dentro do loop (powf)
• Correção do código com vetorização parcial:

```
float a[size], aux[size];
main(){ int i; for(i=1; i<Size; i++){
aux[i] = a[i]*a[i]+10.0/a[i]; }
for(i=1; i<Size; i++){
a[i] = aux[i]-a[i-1]; }}
```

• Considere o programa abaixo:

```
I1: mov (%ebx, %ecx), %esi
I2: add $20 %esi
I3: mov %esi (%ebx, %ecx)
I4: dec %ecx
I5: jnz I1
```

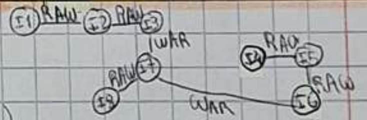
• (CPI=1, out of order, saltos previstos como tomados)

ciclo	UF1(mem)	UF2(mem)	UF3(int)	UF4(int)
1	I1-1			
2			I2-1	
3	I3-1		I4-1	
4	I1-2		I5-1	
5			I2-2	
6	I3-2		I4-2	
7			I5-2	

• SupeE: CPI (reduz) → #I (igual) → Texec (reduzido)
• loop unrolling: CPI (reduz) → #I (pode aumentar) → Texec (reduzido)
• Vetorização: CPI (pode aumentar pela mem) → #I (muito reduzido) → Texec (muito reduzido)
• Paralelismo: CPI (igual) → Nº de Instruções (igual ou mais) → Texec (reduzido)

loop unrolling = 2:

```
I1: mov (%ebx, %ecx), %esi
I2: add $20 %esi
I3: mov %esi (%ebx, %ecx)
I4: mov -1(%ebx, %ecx), %edi
I5: add $20 %edi
I6: mov %edi, -1(%ebx, %ecx)
I7: subl $2, %ecx
I8: jnz I1
```



ciclo	UF1(mem)	UF2(mem)	UF3(int)	UF4(int)
1	I1	I4		
2			I2	I5
3	I3	I6		
4			I8	

```
I1: movl (%ebx, %edx, 4), %esi
I2: addl %esi, %eax
I3: incl %edx
I4: decl %ecx
I5: jnz I1
```

ciclo	static inorder		dynamic outoforder	
ciclo	UF1(int)	UF2(mem)	UF1(int)	UF2(mem)
1		I1	I3	I1
2	I2		I4	
3	I3		I2	I5
4	I4			
6		I5		
	CPI = 5/5 = 1		CPI = 3/5 = 0,6	

• Faz mais sentido utilizar schedule(dynamic) porque nada nos diz que a matriz é quadrada e então é melhor dividir o loop por demanda

• Código abaixo explora Thread Level Parallelism (OpenMP):

```
#define W 400000  int a[W][W]; int sum, i, j;
# pragma omp parallel for
for (i=0; i<W; i++){ sum=0;
for(j=1; j<=i; j++) sum+=a[i][j];
a[i][0]=sum; }
```

• Erros e correções: Este programa produz comportamento indefinido porque existem race conditions causados pelo acesso e alteração imprevisível a dados partilhados pelos vários threads. É aproveitado o facto das somas dos elementos de linhas de uma matriz serem independentes em cada linha para paralelizar esta tarefa. No entanto, existe informação erroneamente partilhada entre os threads: as variáveis globais sum e j. No contexto do programa não faz sentido tal acontecer. Existem duas soluções: - Declarar variáveis dentro do ciclo exterior (int sum=0 antes do ciclo interior em vez da declaração global, e int j=1 antes do ciclo interior ou for(int j=1...) em vez da declaração global. - Usar a keyword private da API OpenMP (# pragma omp parallel for private(sum, j))

⚠ $mp_{cc} = mp_T \times f$!

ganho$_{b \to a}$ = $\frac{T_{exec\,a}}{T_{exec\,b}}$

- $T_{exec} = \frac{CPI \times \#I}{f} = \frac{\#cc}{f}$
- $CPI = \frac{\#cc}{\#I}$ $T_{cc} = \frac{1}{f}$
- $CPI = CPI_{CPU} + CPI_{MEM}$
- $CPI_{MEM} = (mri + mrD \times \%Mem) \times mpcc$

$\%Mem = \frac{n^o\ I\ \text{acedem à memória}}{n^o\ \text{total de instruções}}$

- $CPI_{mem\text{-}I} = mri \times mpcc$
- $CPI_{memD} = \%Mem \times mrD \times mpcc$

tag	set	offset

(m bits)

- tag = m − b − s
- block offset = $\log_2 B$ (identifica o byte dentro do bloco)
- set = $\log_2 S$ (identificam o set dentro da cache)
- S=4 E=1 B=2 m=5 → nº de bits do endereço
  - S: nº de sets; E: nº de linhas por set; B: nº de bytes do block offset; (mapeamento direto)

- $T_{cc} = max(T_{estágios}) + T_{registo}$ (latência)
- T_{inst} = nº estágios × T_{cc}
- $T_{estágio}$ = soma de todos os blocos desse estágio

- **Mapeamento direto**: nº de sets = nº de linhas. Uma linha por set (E=1); potencial para elevado nº de colisões; hit time reduzido pois apenas 1 posição compara a tag
- **Mapeamento Completamente associativo**: um único set (S=1), reduzido nº de colisões, hit time aumenta pois é necessário comparar com todas as tags
- **Mapeamento n-way set associative**: S sets com E linhas cada; compromisso no nº de colisões, comparação de tags de cada set; compromisso no hit time

- Aumentar f (diminuir Tcc) implica frequentemente um aumento do CPI. Tcc diminui, mas o tempo de acesso à memória mantém-se são necessários mais ciclos para aceder à memória (Tcc diminui ⇒ CPI aumenta)
- Diminuir o #I recorrendo a instruções mais complexas (necessitam de mais ciclos) resulta num aumento do CPI
- Localidade temporal: um elemento de memória acedido pelo CPU será acedido de novo num futuro próximo
- Localidade espacial: se um elemento é acedido pelo CPU, os elementos com endereços próximos são acedidos num futuro próximo

```
for(i=0; i<N; i++) a[i]=b[100*i]*2;...
```

→ exibe localidade espacial nos acessos a a[]
→ exibe localidade temporal nos acessos a i

- A técnica de pipelining relativamente a uma arq. sequencial de ciclo único, acelera o desempenho de um processador pois resulta numa diminuição do período de relógio, uma vez que deve apenas ser tão longo quanto o estágio mais demorado do pipeline

Pipeline: resumo

- execução de n instruções simultaneamente em diferentes estágios
- permite aumentar a frequência do relógio
- Dependência de dados: stalling (injeção de bolhas (Nops)); realimentação elimina penalizações
- Dependências de Controlo: saltos condicionais implicam execução especulativa (previsão); previsão errada implica stalling do pipeline

decimal	binário
10^3 Kilobyte KB	2^{10} Kibibyte KiB
10^6 Megabyte MB	2^{20} Mebibyte MiB
10^9 Gigabyte GB	2^{30} Gibibyte GiB
10^{12} Terabyte TB	2^{40} Tebibyte TiB
10^{15} Petabyte PB	2^{50} Pebibyte PiB
10^{18} Exabyte EB	2^{60} Exbibyte EiB
10^{21} Zettabyte ZB	2^{70} Zebibyte ZiB

1 ps (picosegundo)	10^{-12}
1 ns (nanosegundo)	10^{-9}
1 µs (microsegundo)	10^{-6}
1 ms (milisegundo)	10^{-3}

→ Cache
- nº total de linhas: S×E
- Capacidade da cache: S×E×B

	#	Instrução	Fetch hit	Fetch miss	Acesso à Mem hit	Acesso à Mem miss
mov $4 %ebx	1	mov $4 ebx		x		
mov $2, %ecx	2	mov $2 ecx	x			
loop: mov(%ebx,%ecx,4),%eax	3	mov(...) eax		x		x
add %eax, %edx	4	add	x			
dec %ecx	5	dec		x		
jne loop	6	jne loop	x			
	7	mov(...)	x		x	
	8	add	x			
	9	dec	x			
	10	jne	x			

Addr	mapeamento direto S=4 E=1 B=2 m=5	totalmente associativo S=1 E=4 B=2 m=5	m-way set associativo (2-way set associativo) S=2 E=2 B=2 m=5
5(00101)	set 2/linha 0 miss	set 0/linha 0 miss	set 0/linha 0 miss
14(01110)	set 3/linha 0 miss	set 0/linha 1 miss	set 1/linha 0 miss
10(01010)	set 1/linha 0 miss	set 0/linha 2 miss	set 1/linha 1 miss
29(11101)	set 2/linha 0 miss	set 0/linha 3 miss	set 0/linha 1 miss
4(00101)	set 2/linha 0 miss	set 0/linha 0 hit//	set 0/linha 0 hit
21(10101)	set 2/linha 0 miss	set 0/linha 1 miss	set 0/linha 1 miss
16(10000)	set 0/linha 0 miss	set 0/linha 2 miss	set 0/linha 0 miss
5(00101)	set 2/linha 0 miss	set 0/linha 0 hit//	set 0/linha 1 miss
Miss rate	100%	75%	87.5

I1: mov $10, %eax
I2: sub $5, %eax
I3: jz I1
I4: add $10 %eax

- Dependência de controlo: jumps: prevê-se que o salto é sempre dado; A correção da previsão é feita posteriormente; Se a instrução estiver errada as instruções entretanto lidas são convertidas em bolhas, em injeção resulta num desperdício de ciclos igual ao nº de bolhas

→ recorrendo aos stallings

	1	2	3	4	5	6	7	8	9	10	11
I1	F	D	E	W							
I2		F	D	D	D	E	W				
I3			F	F	F	D	E	W			
B1						B_1^F	B_1^D	B_1^E	B_1^W		
B2							B_2^F	B_2^D	B_2^E	B_2^W	
I4								F	D	E	W

→ prevê os saltos como tomados + data forwarding

	1	2	3	4	5	6	7	8	9
I1	F	D	E	W					
I2		F	D	E	W				
I3			F	D	E	W			
I1				F	D	(E)	(W)		
I2					F	(D)	(E)	(W)	
I4						F	D	E	W

- Data-forwarding: Passa o valor necessário (na dependência de dados) do estágio onde está disponível (E ou W) para o estágio de decode

- São necessários 9 ciclos. A dependência de dados entre I1 e I2 é resolvida por data forwarding, o salto é mal previsto resultando numa penalização de 2 ciclos