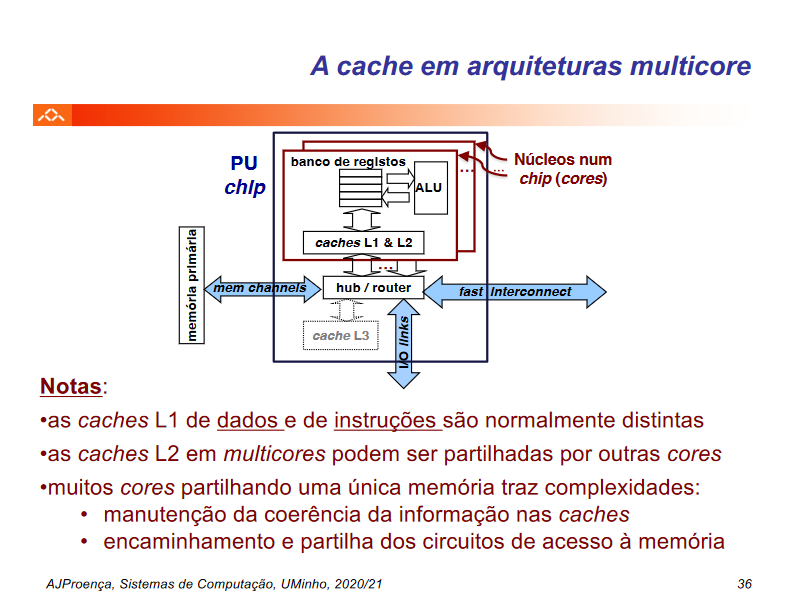
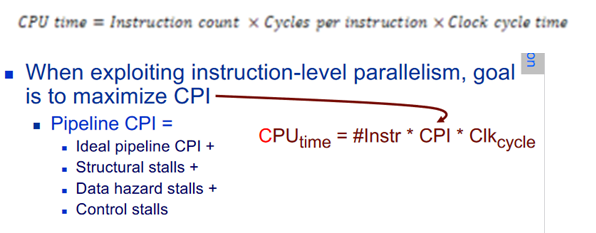
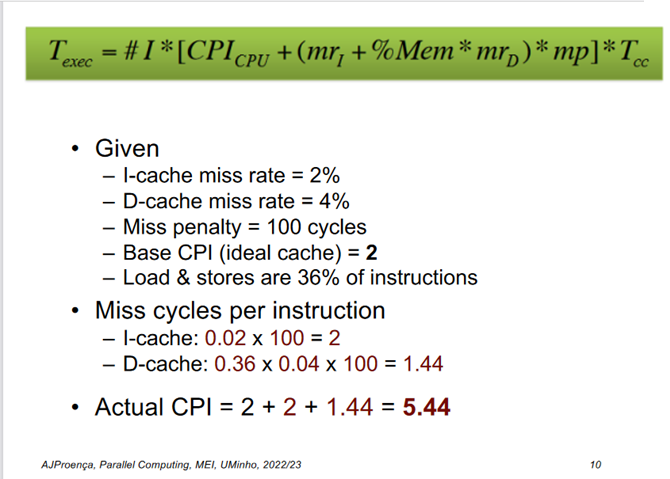
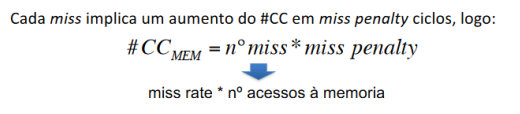
Amdhal’s Law



Paralelismo ajuda a reduzir CPI

**LEI gustafson’s law** (strong scalability analysis) :: aumenta-se o tamanho do problema como o número de PU (unidades de processamento)

Falta ver os outros exercícios



1GHz = 1 \*10^9

1 MB = 10^6 bytes

Paralelismo no processador Pipeline Problemas:

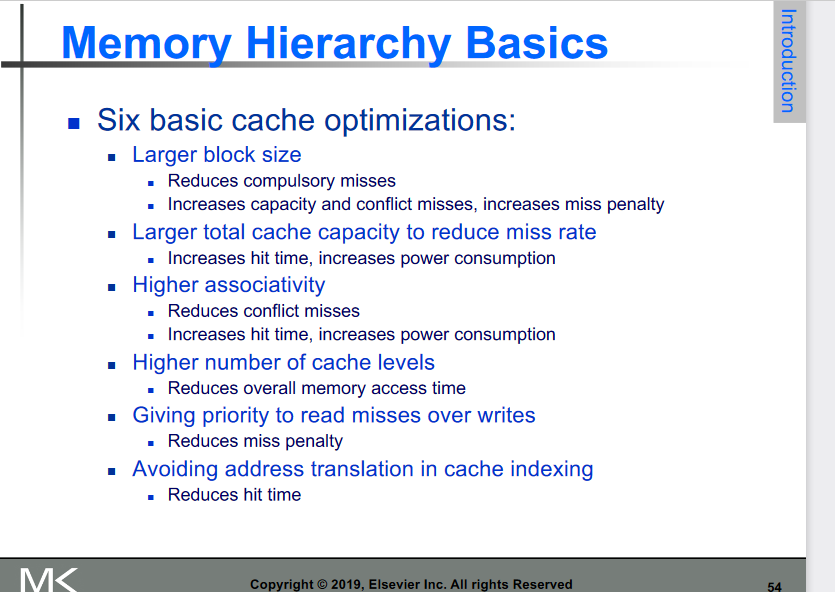
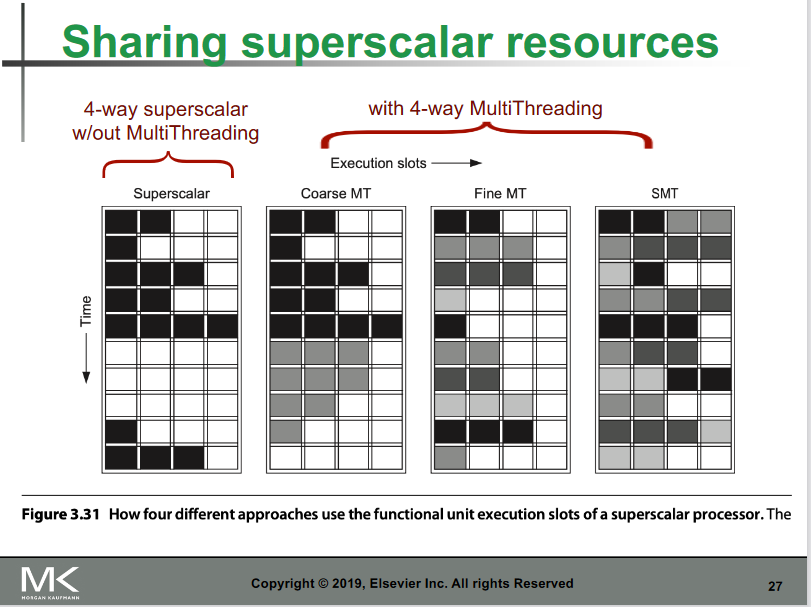
•dependências de dados

•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



Ter mais que uma thread num core aumenta o tempo de cada thread, pois tem de trocar registos e tudo mais, mas melhora o tempo global, porque quando uma thread está à espera de coisas da memória, a outra pode fazer coisas. Assim, o tempo global é melhorado

2 -> Só troca de thread quando o tempo de espera é grande, tipo L2 misses;

3-> Troca ciclo a ciclo, logo precisa de várias threads, é mau por estar sempre mais à espera que uma thread acabe.

4-> Tenta ocupar ao máximo, também -------

3. "as várias alterações que tiveram que ser feitas nessa arquitetura"

Assim, o ponto de partida era "uma arquitetura básica elementar" (portanto sem cache) e respetiva lista de operações a executar, as quais incluíam ir buscar a instrução à memória (que iria levar vários ciclos de clock e não mostrados na figura, o que já pressupõem uma execução em pipeline entre a busca de instruções e a sua respetiva execução), e ter de ir buscar operandos à memória (como neste caso, e sem cache levaria muitos ciclos de clock e não apenas 3 como indicado na figura). Aqui já estão 2 melhorias que tiveram de ser implementadas. Outras:

- em cada ciclo de clock é possível arrancar com a execução de mais que uma operação em simultâneo, isto é superescalaridade;

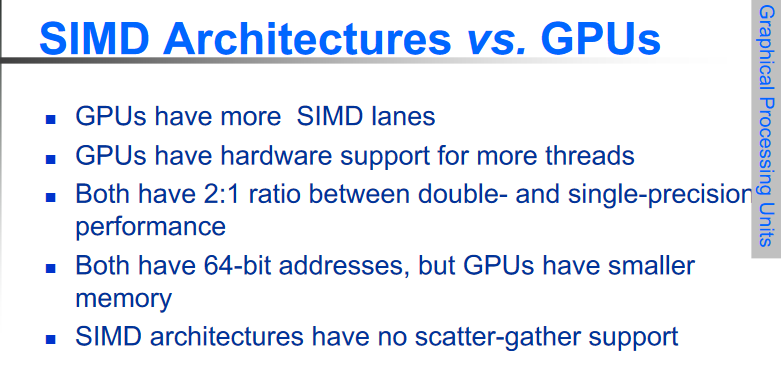
- em cada ciclo de clock é possível iniciar uma nova operação de load sem a anterior ter concluído, isto é pipeline na execução do load;

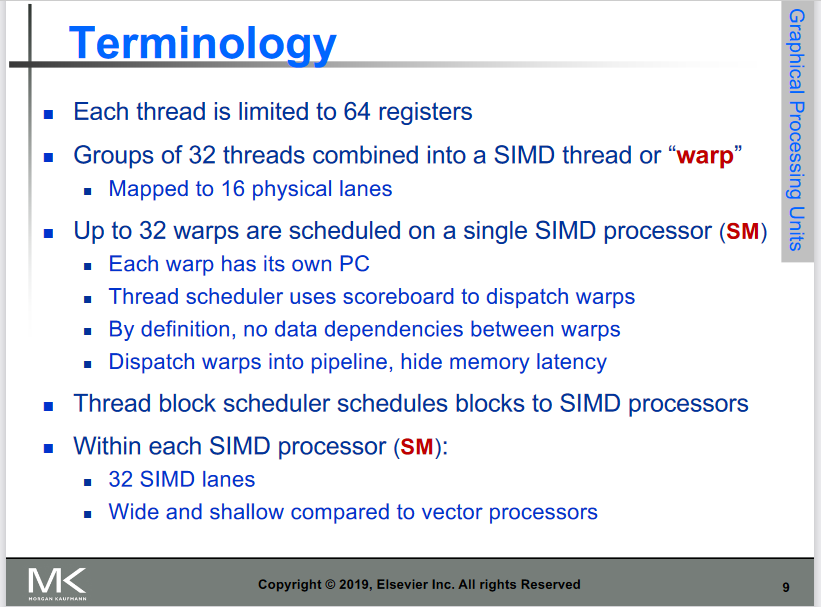
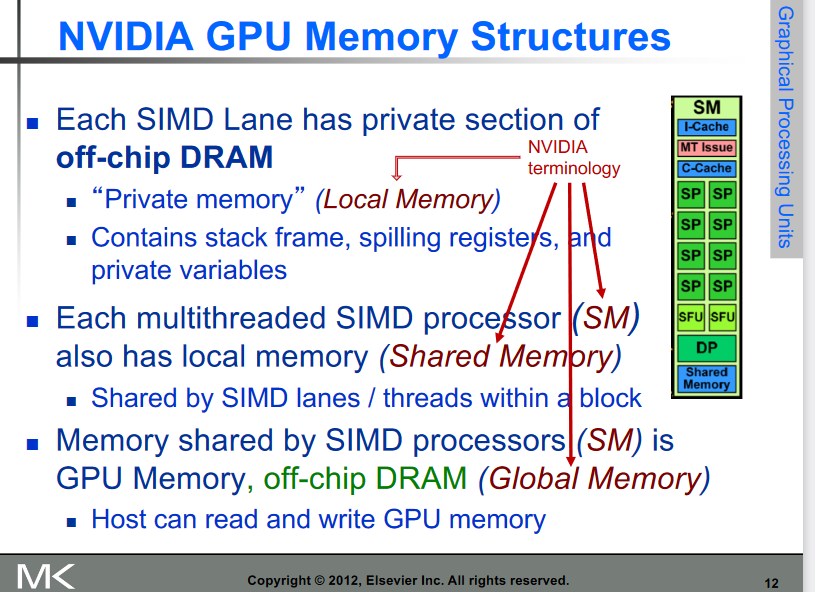
- a duplicação de unidades funcionais: há pelo menos 2 unidades para operar com inteiros, o que permite que em cada ciclo de clock possam ser executadas 2 operações com inteiros;

- as instruções de teste de fim do ciclo estão a ser executadas ainda antes de o load ter terminado, i.e., execução de instruções fora de ordem;

- uma nova iteração do ciclo é iniciada mesmo antes de se saber se o ciclo for já chegou ao fim ou não, isto é

execução especulativa de código.

GPU

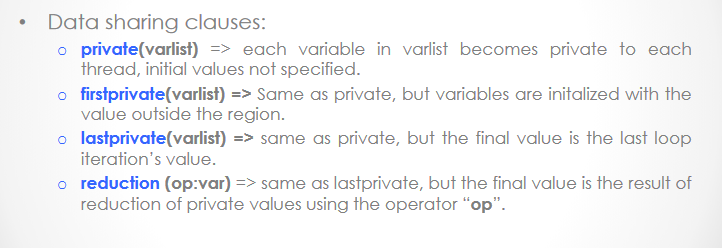


Código

void saxpy\_serial(int n, float a, float \*x. float \*y)  
{  
for (int 1 = 0; i < n; ++ii)  
  y[i] = a\*x[i] + y[i]:    
}

// Invoke serial SAXPY kerne]  
saxpy\_serial(n, 2.0, x, y);

\_\_[globa]\_\_ void saxpy\_parallel(int n, float a, float \*x. float \*y)  
{  
int i = blockIdx.x\*blockDim.x + threadId.x:  
if (i < n) y[i] =  a\*x[i] + y[i«i];   
}  
// Invoke parallel SAXPY kernel with 256 threads/block  
int nblocks = (n + 255) / 256;  
saxpy\_parallel<<<nblocks, 256>>>(n, 2.0, x, y);



Falta MPI

