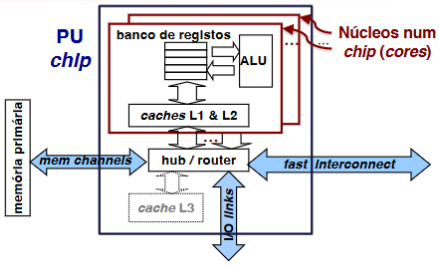
Amdhal's Law $$Speedup = \frac{1}{(1-p) + p/N}$$

A cache em arquiteturas multicore



Notas:

- as *caches* L1 de dados e de instruções são normalmente distintas
- as *caches* L2 em *multicores* 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

Paralelismo ajuda a reduzir CPI

Falta ver os outros exercícios

$$CPU\ time = Instruction\ count \times Cycles\ per\ instruction \times Clock\ cycle\ time$$

- When exploiting instruction-level parallelism, goal is to maximize CPI
  - Pipeline CPI =
    - Ideal pipeline CPI +
    - Structural stalls +
    - Data hazard stalls +
    - Control stalls

CPU_{time} = #Instr * CPI * Clk_{cycle}

Cada *miss* implica um aumento do #CC em *miss penalty* ciclos, logo:

$$\#CC_{MEM} = n^{o} miss * miss\ penalty$$

miss rate * nº acessos à memoria

$$T_{exec} = \#I * [CPI_{CPU} + (mr_I + \%Mem * mr_D) * mp] * T_{cc}$$

- Given
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = **2**
  - Load & stores are 36% of instructions
- Miss cycles per instruction
  - I-cache: 0.02 x 100 = 2
  - D-cache: 0.36 x 0.04 x 100 = 1.44
- Actual CPI = 2 + 2 + 1.44 = **5.44**

Sharing superscalar resources

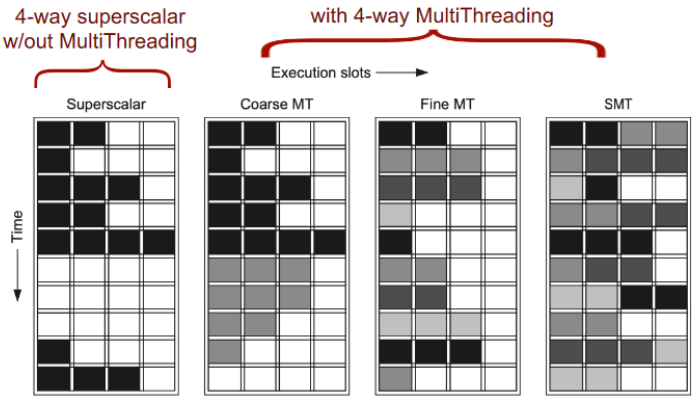


Figure 3.31 How four different approaches use the functional unit execution slots of a superscalar processor. The

Copyright © 2019, Elsevier Inc. All rights Reserved

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

Memory Hierarchy Basics

- Six basic cache optimizations:
  - Larger block size
    - Reduces compulsory misses
    - Increases capacity and conflict misses, increases miss penalty
  - Larger total cache capacity to reduce miss rate
    - Increases hit time, increases power consumption
  - Higher associativity
    - Reduces conflict misses
    - Increases hit time, increases power consumption
  - Higher number of cache levels
    - Reduces overall memory access time
  - Giving priority to read misses over writes
    - Reduces miss penalty
  - Avoiding address translation in cache indexing
    - Reduces hit time

Copyright © 2019, Elsevier Inc. All rights Reserved

GPU

Terminology

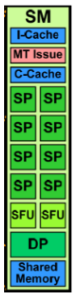
- Each thread is limited to 64 registers
- Groups of 32 threads combined into a SIMD thread or "**warp**"
  - Mapped to 16 physical lanes
- Up to 32 warps are scheduled on a single SIMD processor (**SM**)
  - Each warp has its own PC
  - Thread scheduler uses scoreboard to dispatch warps
  - By definition, no data dependencies between warps
  - Dispatch warps into pipeline, hide memory latency
- Thread block scheduler schedules blocks to SIMD processors
- Within each SIMD processor (**SM**):
  - 32 SIMD lanes
  - Wide and shallow compared to vector processors

Copyright © 2019, Elsevier Inc. All rights Reserved

NVIDIA GPU Memory Structures

- Each SIMD Lane has private section of **off-chip DRAM**
  - "Private memory" *(Local Memory)*
  - Contains stack frame, spilling registers, and private variables
- Each multithreaded SIMD processor *(SM)* also has local memory *(Shared Memory)*
  - Shared by SIMD lanes / threads within a block
- Memory shared by SIMD processors *(SM)* is GPU Memory, off-chip DRAM *(Global Memory)*
  - Host can read and write GPU memory

NVIDIA terminology



Copyright © 2012, Elsevier Inc. All rights reserved.

SIMD Architectures *vs.* GPUs

- GPUs have more SIMD lanes
- GPUs have hardware support for more threads
- Both have 2:1 ratio between double- and single-precision performance
- Both have 64-bit addresses, but GPUs have smaller memory
- SIMD architectures have no scatter-gather support

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);
```

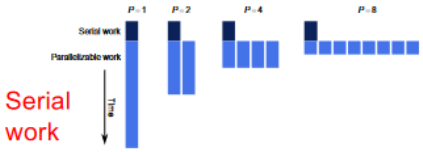
- Data sharing clauses:
  - **private(varlist)** => each variable in varlist becomes private to each thread, initial values not specified.
  - **firstprivate(varlist) =>** Same as private, but variables are initalized with the value outside the region.
  - **lastprivate(varlist) =>** same as private, but the final value is the last loop iteration's value.
  - **reduction (op:var)** => same as lastprivate, but the final value is the result of reduction of private values using the operator "**op**".

Falta MPI

- **Strong scalability** analysis:
  - speed-up increase with PU for a fixed problem data size
  - ideal speed-up is proportionally to PU
- **Weak scalability** analysis:
  - Increase problem data size as the number of PU increases
  - Ideally the execution time should remain constant

(some reasons) why parallel applications do not have an ideal speed-up?

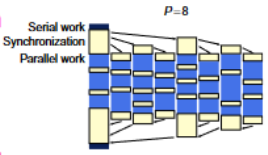
1. % of serial work (**Amdahl's law**)
2. Memory wall
   - Serializes memory accesses



Serial work

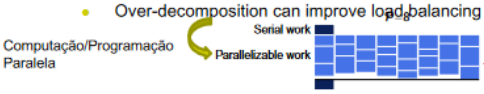
3. Parallelism/task granularity
   - Additional work performed in the parallel application (tasks management, redundant computations, etc)
4. Synchronisation overhead
   - Might also serialize execution (e.g. critical)
     - Includes (serial) calls to external routines (e.g., malloc)

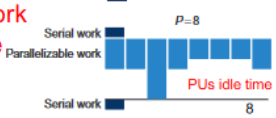
Parallelism overhead



Serial work /idle time

5. Load imbalance
   - Over-decomposition can improve load balancing





a