Go Multicore Series:

Understanding Memory in a Multicore World, Part 1:

Does Your Thread See What I See?

Joe Hummel, PhD | http://www.joehummel.net/freescale.html

FTF 2014: FTF-SDS-F0098

External Use

Agenda

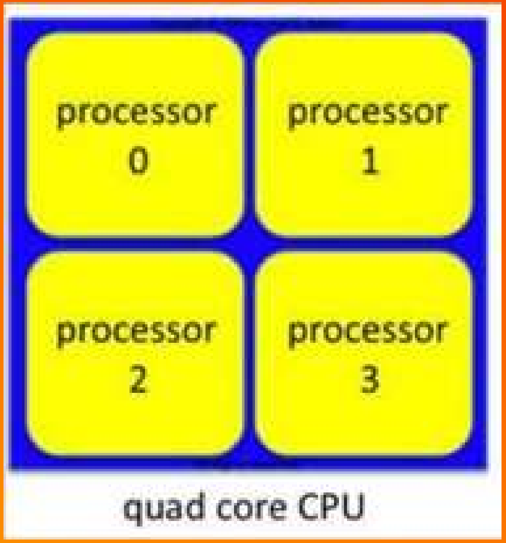
- Why multicore?
- Motivation by example: matrix multiplication
- Shared memory and caching
- High Performance Computing = …

Introductions

- Your speaker…

- Joe Hummel, PhD
  - ***PhD***: *UC-Irvine, in High Performance Computing*
  - ***Professor***: *U. of Illinois, Chicago*
  - ***Trainer***: *Pluralsight, LLC*
  - ***Consultant***: *Joe Hummel, Inc.*
  - *Microsoft MVP C++*
  - *Married, one daughter adopted from China (just turned 12!)*
  - *Avid sailor*

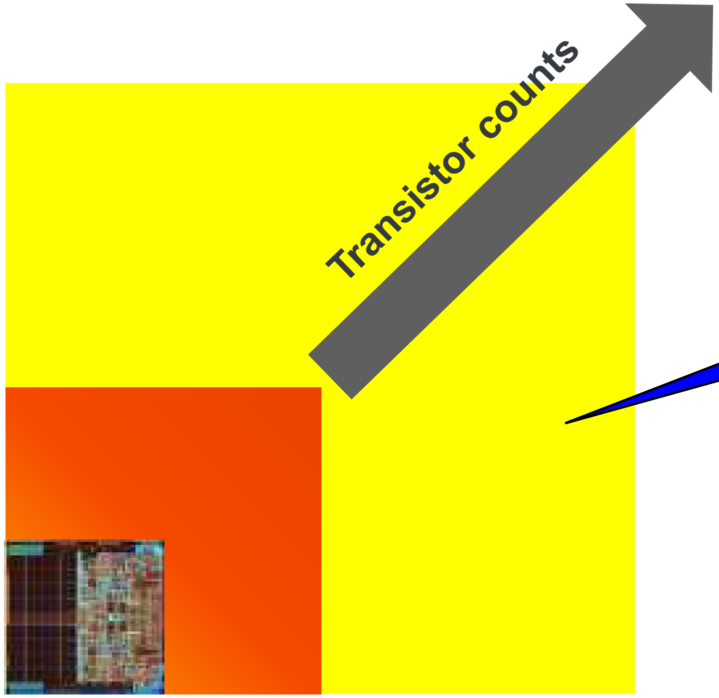
Multicore



✓ How did we get here?

Moore's law

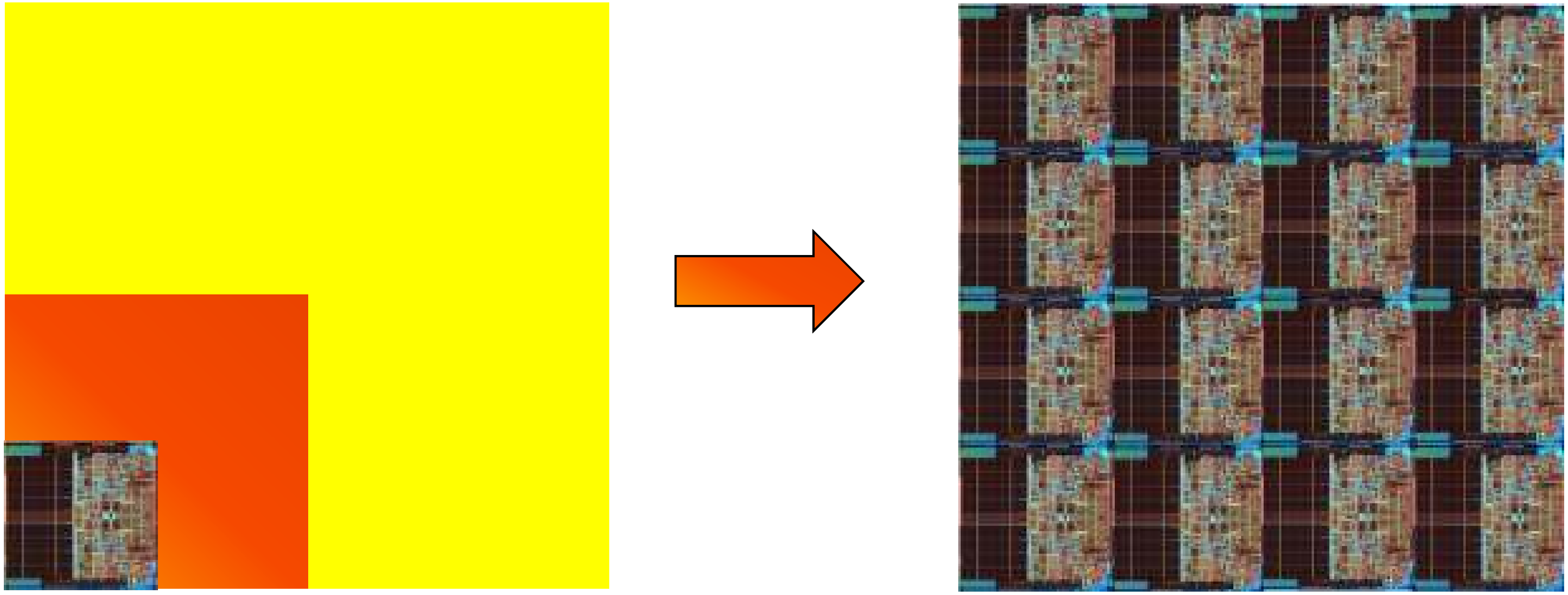
- **Moore's law continues to serve us well — e.g. transistor counts…**



What to do with all these transistors?

Solution

- **Multicore!**
  - *copy-paste processors :-)*



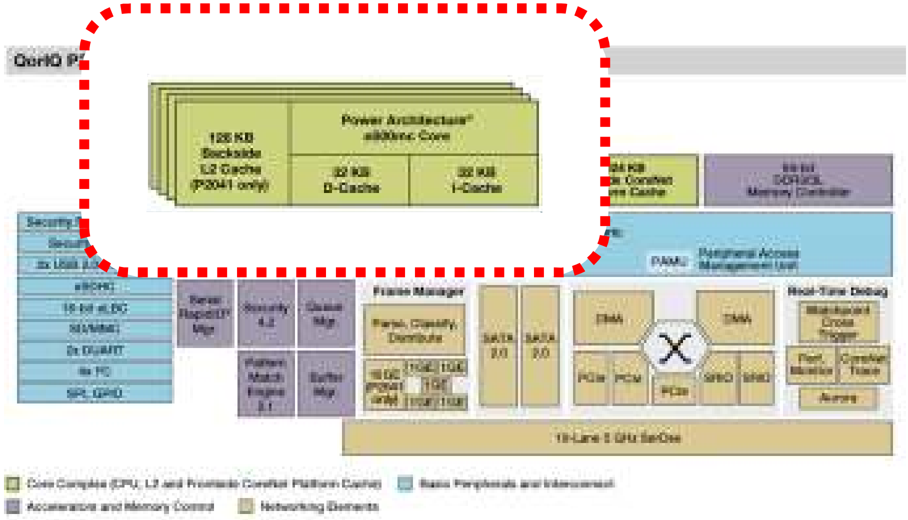
Representative multicore examples…

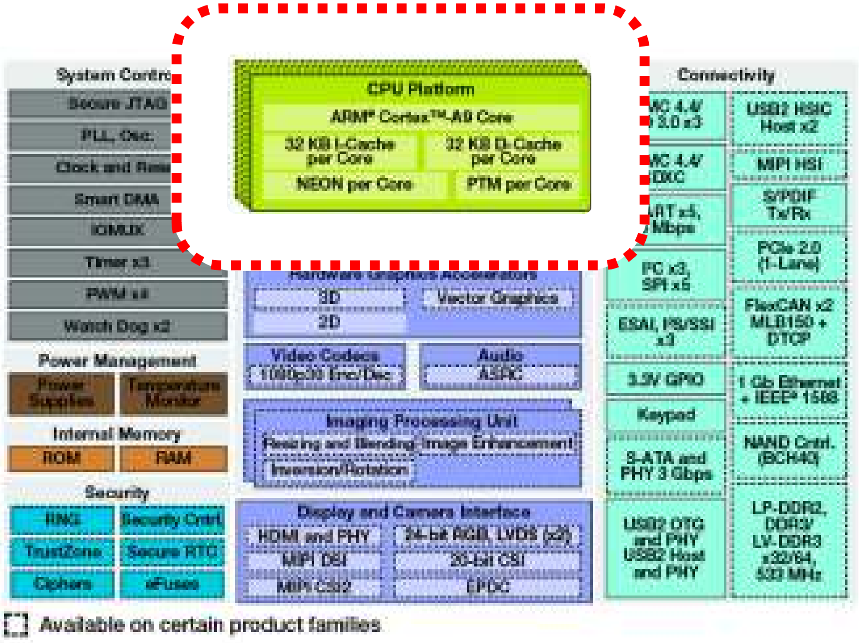
QorIQ P2040/2041

- *Power architecture, quad-core*

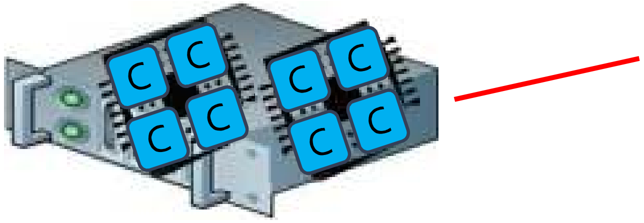
i.MX 6Q

- *ARM architecture, quad-core*





Theoretical performance



8 x 2GHz cores have the computing equivalent of **1 x 16GHz core**, without the heat and exponential power requirement

Taking advantage of multicore

1. **AMP**
2. **SMP**

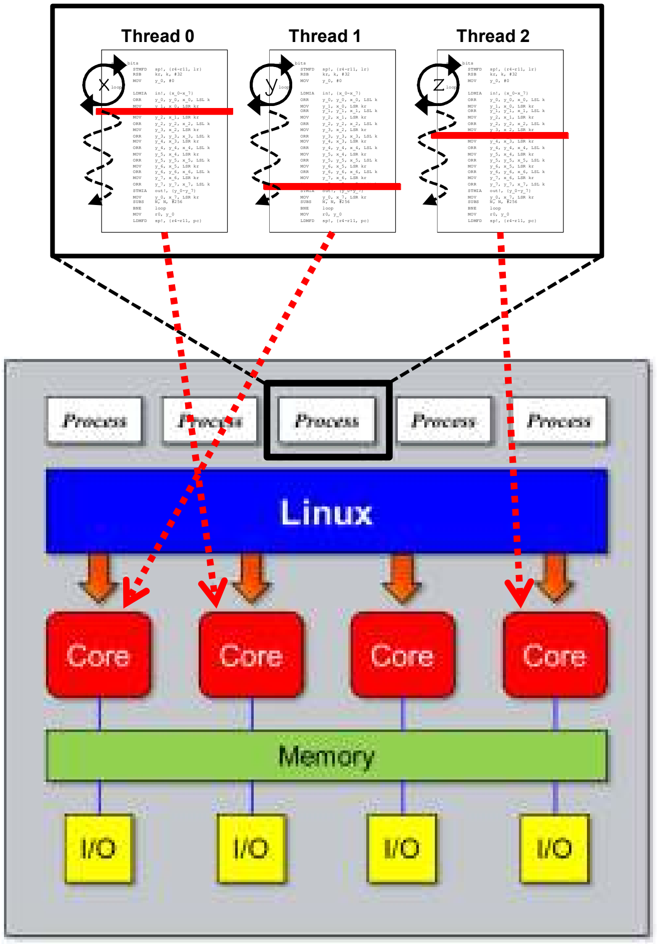
mostly about configuration…

3. **Multithreading**
4. **Multiprogramming**

mostly about programming…

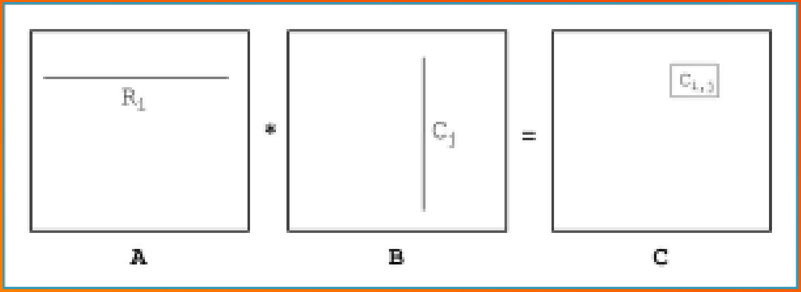
Multithreading

- **Multithreading == multiple threads working together**
  - *To handle more work (throughput) or complete faster (performance)*
  - *OS automatically maps threads across available cores*



- **Disadvantages?**
  - *Increased application complexity*
  - *Dangers of shared memory programming model*
    - race conditions, one thread crash will crash them all, …

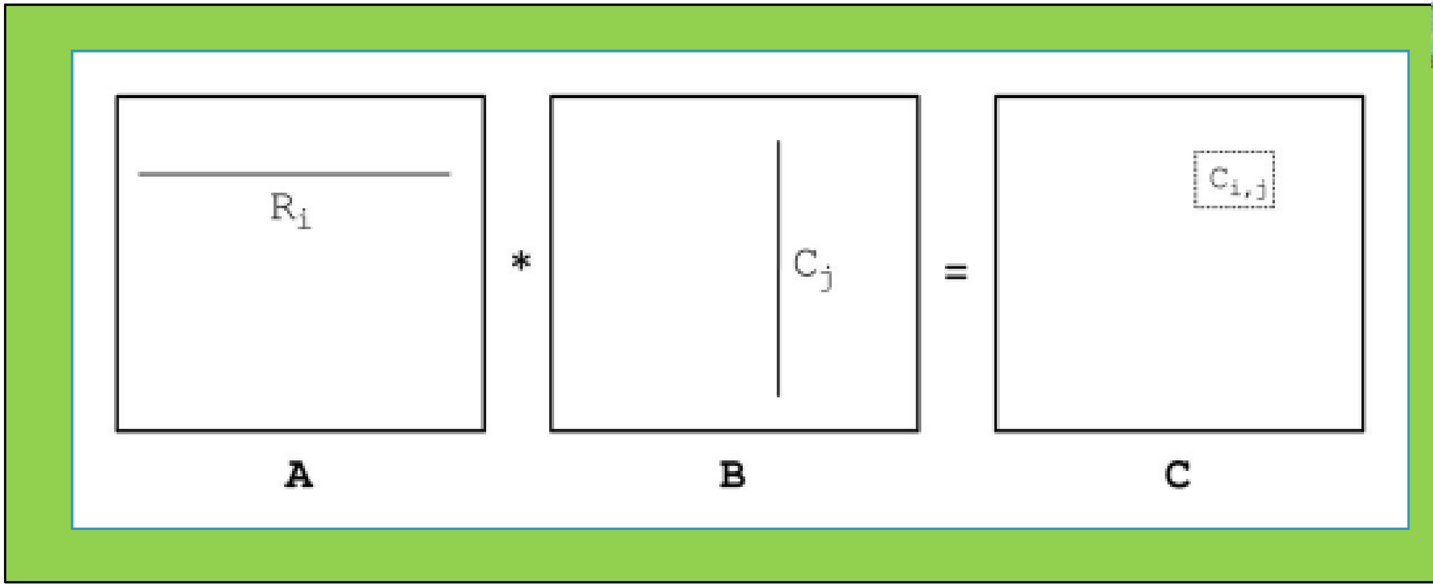
Example

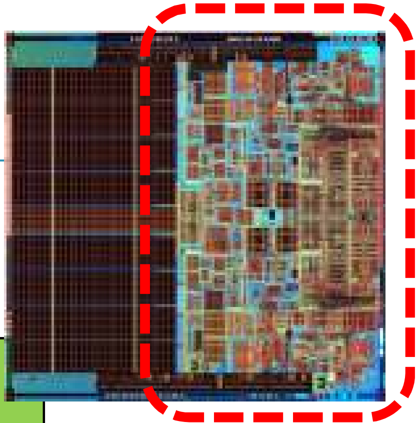


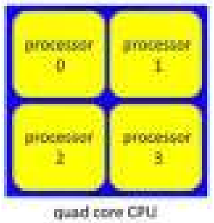
- ✓ **Matrix multiplication…**

Motivation by example

- **Matrix Multiplication**

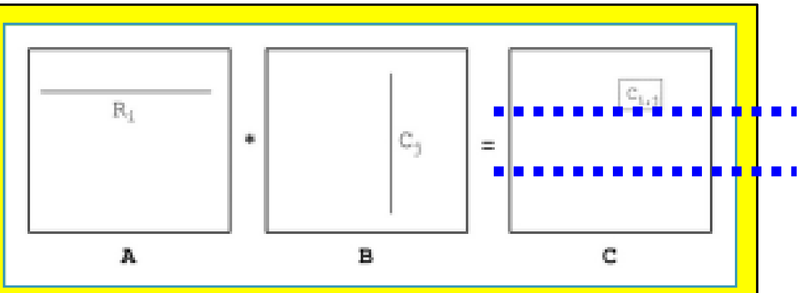


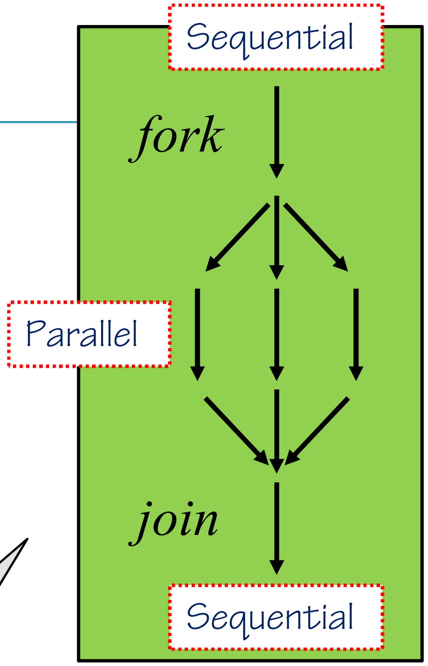




Sequential to parallel…

```
//
// Naïve, triply-nested sequential solution:
//
for (int i = 0; i < N; i++)
{
   for (int j = 0; j < N; j++)
   {
      C[i][j] = 0.0;

      for (int k = 0; k < N; k++)
         C[i][j] += (A[i][k] * B[k][j]);
   }
}
```



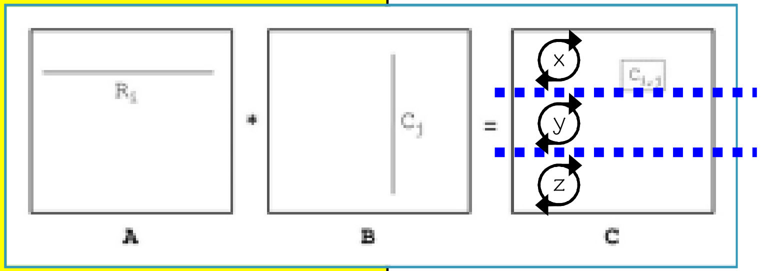


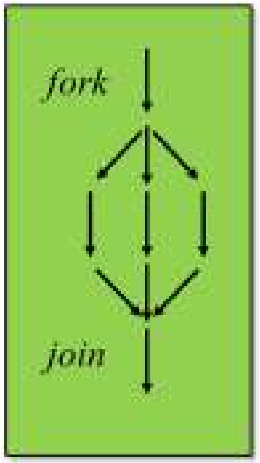
***Common pattern for parallelism**:*
Structured (or "Fork-Join")
Parallelism

Multithreading using OpenMP

- OpenMP == Open Multiprocessing standard
  - *Provide directive, and compiler multithreads for you…*

```
//
// Naïve parallel solution using OpenMP:  result is structured parallelism, with
// static division of workload by row.
//
#pragma omp parallel for   // parallelize outer loop  ==>  by rows:
for (int i = 0; i < N; i++)
{
   for (int j = 0; j < N; j++)
   {
      C[i][j] = 0.0;

      for (int k = 0; k < N; k++)
         C[i][j] += (A[i][k] * B[k][j]);
   }
}
```

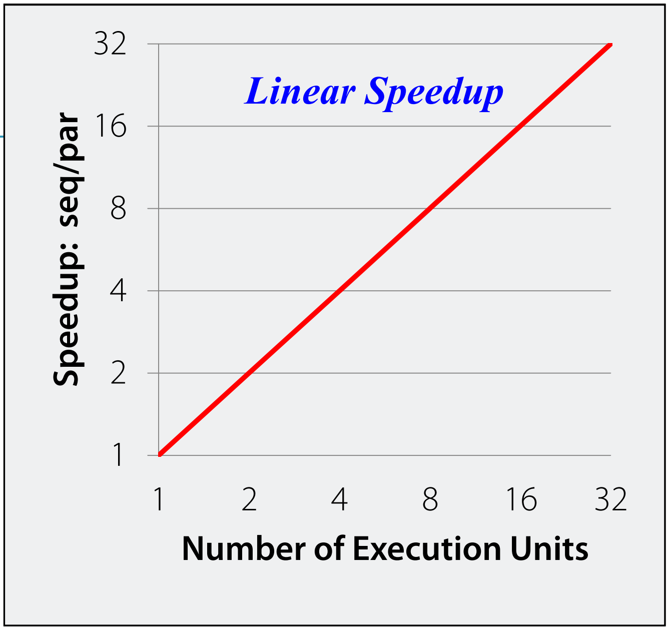




Results?

- **Very good!**
  - *matrix multiplication is "embarrassingly parallel"*
  - *linear speedup — 2x on 2 cores, 4x on 4 cores, …*

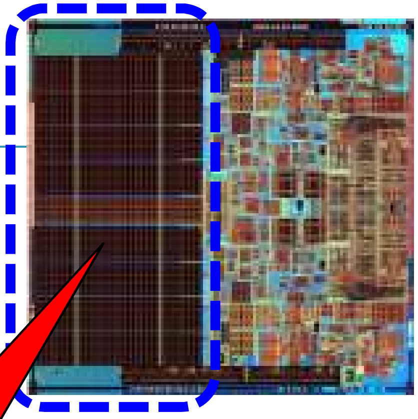
Version	Cores	Time (secs)	Speedup
Sequential	1	30	
OpenMP	4	7.6	**3.9**



But wait…

- **What's the other half of the chip?**
  - *cache!*

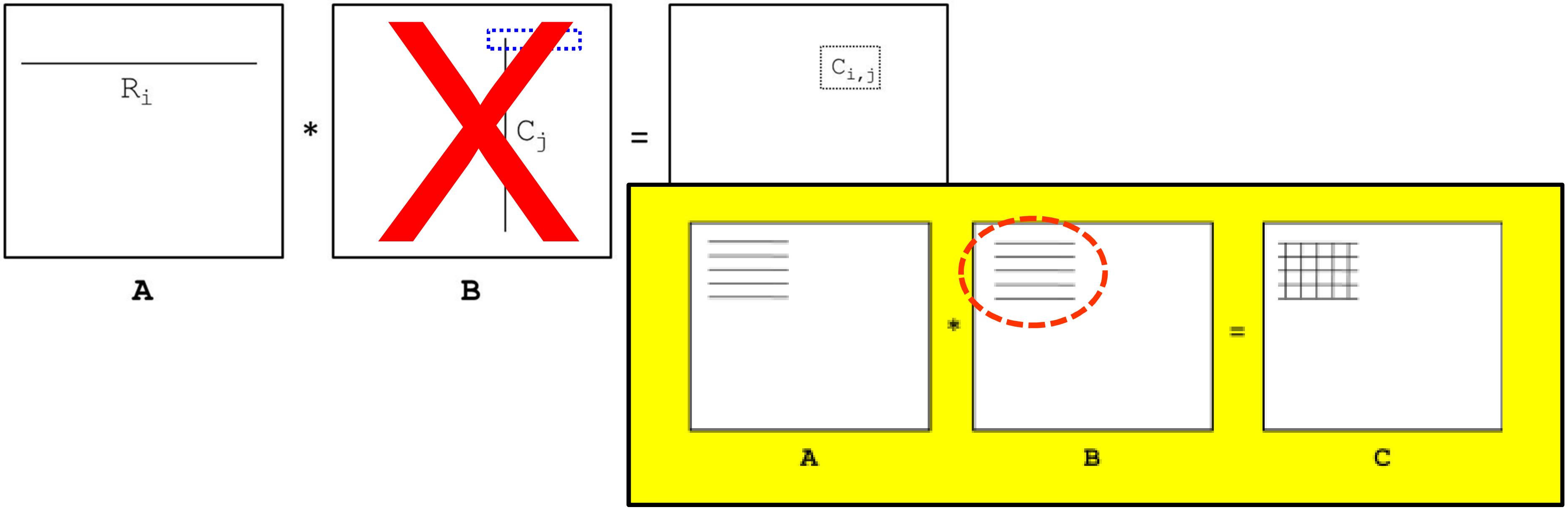
- **Are we using it effectively?**
  - *we are not…*



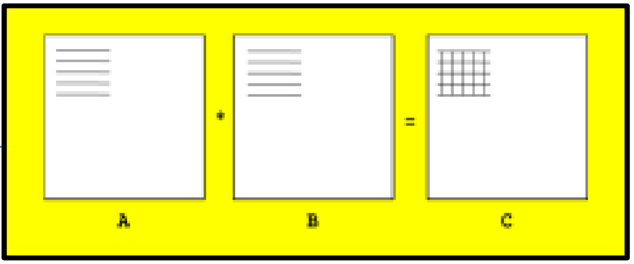
Memory cache…

Cache-friendly matrix multiplication

- **No one solves MM using the naïve algorithm**
  - *horrible cache behavior*



Step 1: loop interchange



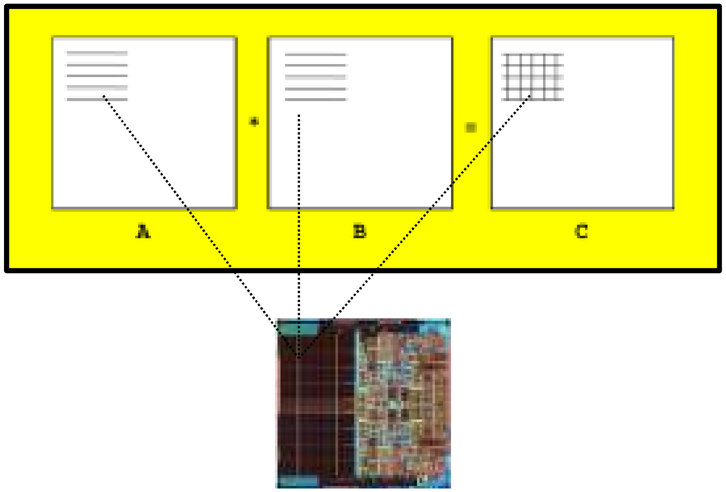
```
   for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
         C[i][j] = 0.0;

#pragma omp parallel for
   for (int i = 0; i < N; i++)
      for (int k = 0; k < N; k++)
         for (int j = 0; j < N; j++)
            C[i][j] += (A[i][k] * B[k][j]);
```

Another factor of 2-10x improvement!

Step 2: blocking

- **Block size based on size of cache closest to core — level 1 ("L1")**
  - *largest integer BS such that*

$$\frac{BS * BS * 3 * sizeof(double)}{cache\ size} < 1$$



```
#pragma omp parallel for
for (int jj=0; jj<N; jj+=BS)   // for each column block:
{
  int jjEND = Min(jj+BS, N);

  // initialize:
  for (int i=0; i<N; i++)
    for (int j=jj; j < jjEND; j++)
      C[i][j] = 0.0;

  // block multiply:
  for (int kk=0; kk<N; kk+=BS)   // for each row block:
  {
    int kkEND = Min(kk+BS, N);

     for (int i=0; i<N; i++)
       for (int k=kk; k < kkEND; k++)
         for (int j=jj; j < jjEND; j++)
           C[i][j] += (A[i][k] * B[k][j]);
  }
}
```

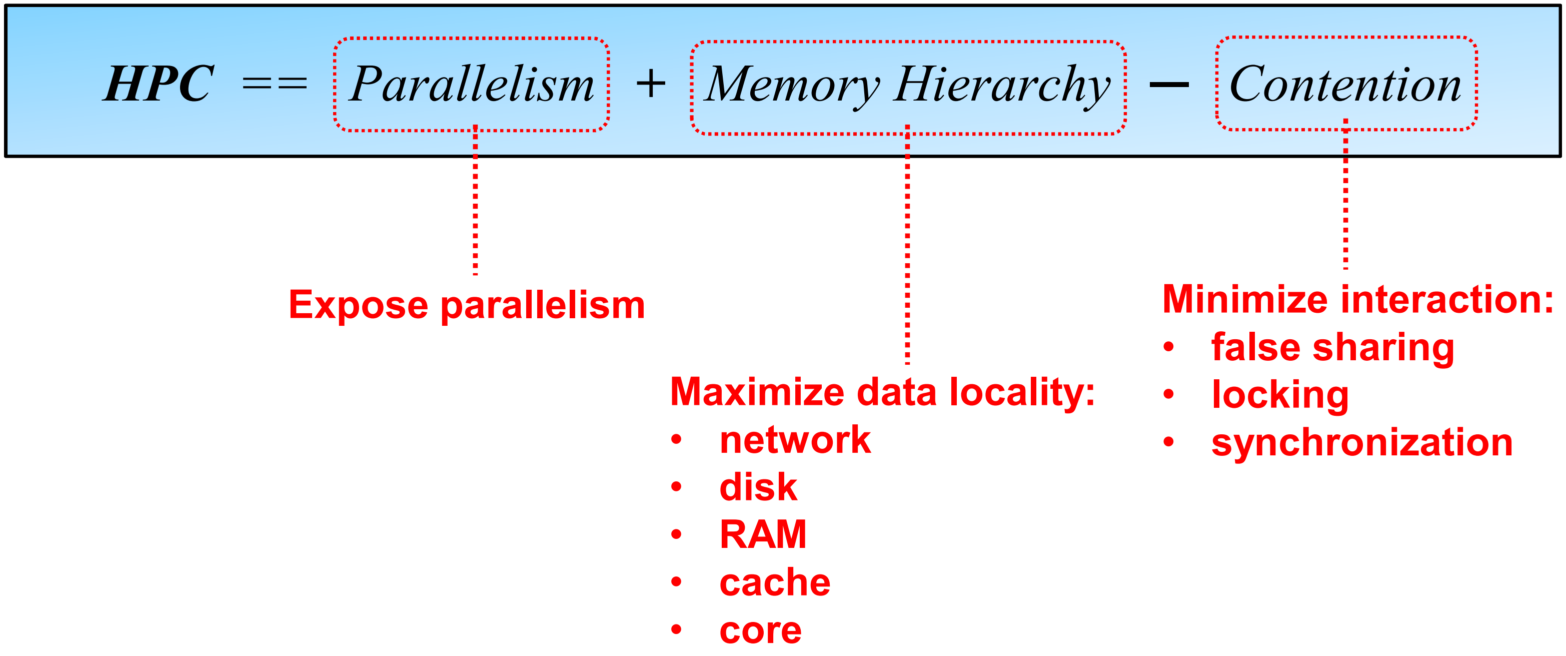
Results?

- **Caching impacts all programs, sequential and parallel…**

Version	Cores	Time (secs)	Speedup
Sequential			
Naive	1	30	
Blocked	1	3	**10**
OpenMP			
Naïve	4	7.6	**3.9**
Blocked	4	0.8	**37.5**

High-Performance Computing

- HPC
- Parallelism alone is not enough…



$$HPC == \boxed{Parallelism} + \boxed{Memory\ Hierarchy} - \boxed{Contention}$$

Expose parallelism

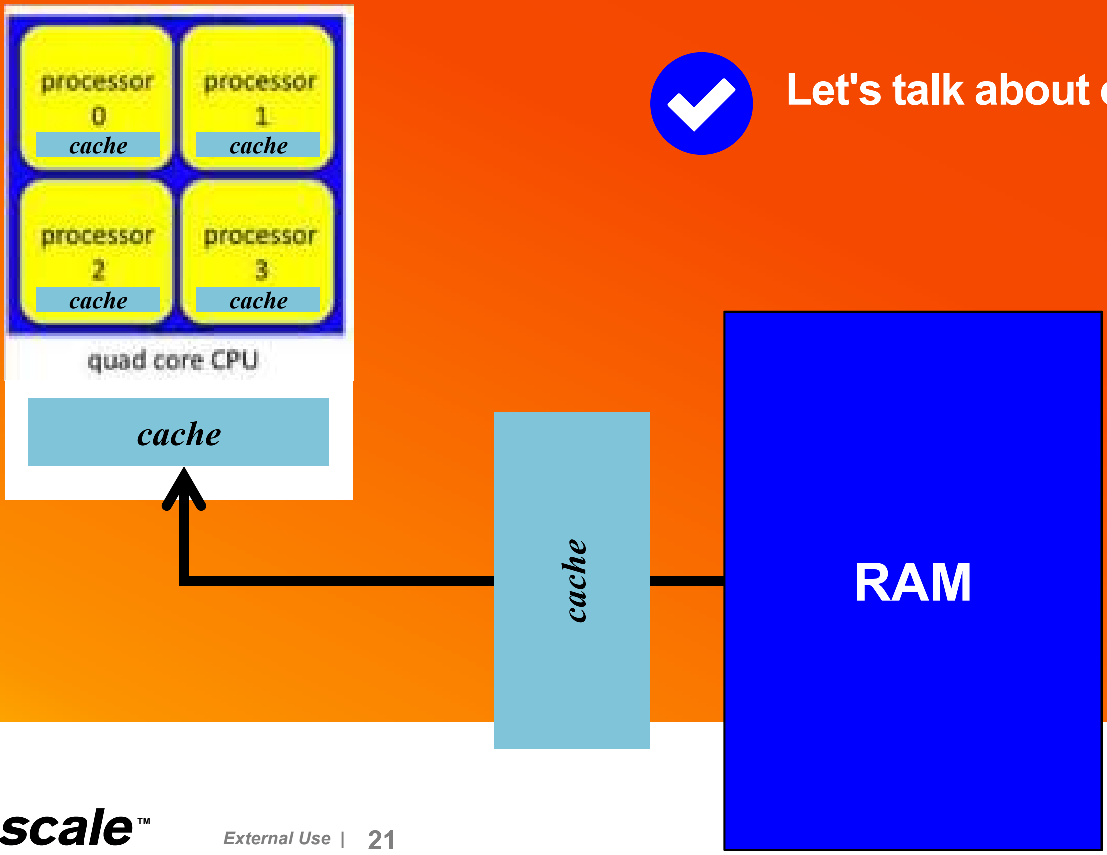
Maximize data locality:

- network
- disk
- RAM
- cache
- core

Minimize interaction:

- false sharing
- locking
- synchronization

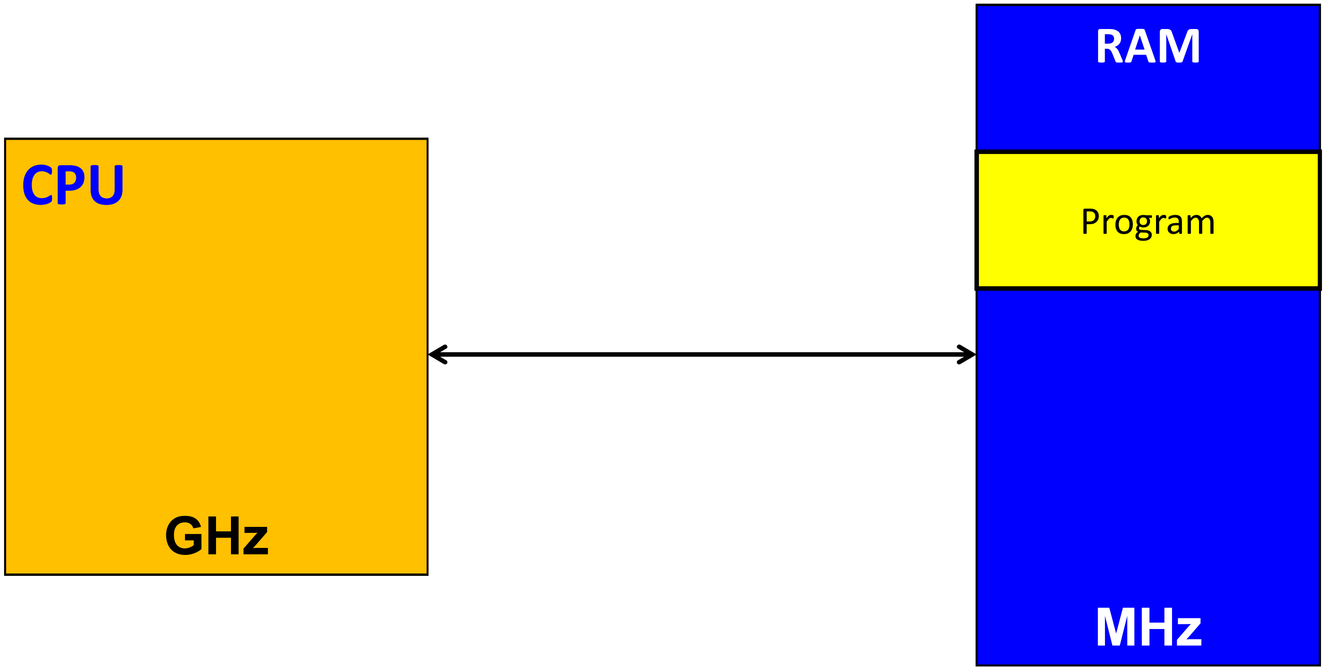
Caching in a multicore world



✓ **Let's talk about caching…**

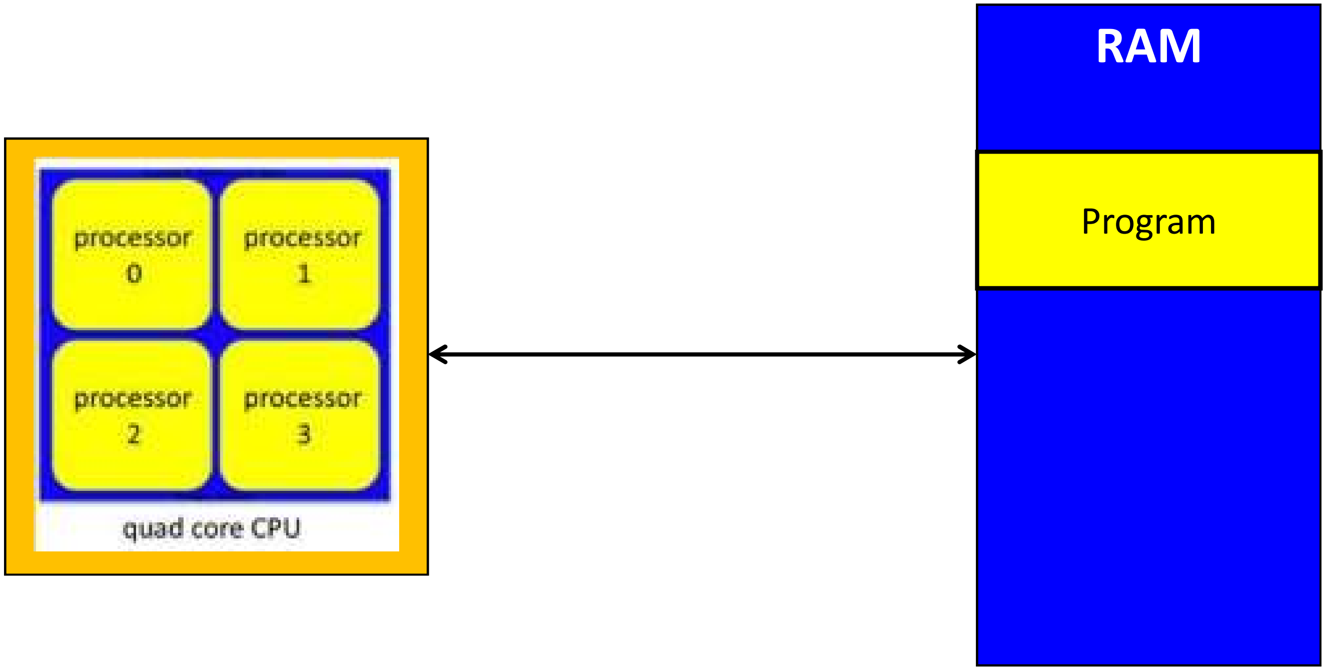
The memory bottleneck

- **Memory latency is *the* biggest performance problem for HW designers**
  - *latency on the order of 100's to 1,000's of CPU cycles*



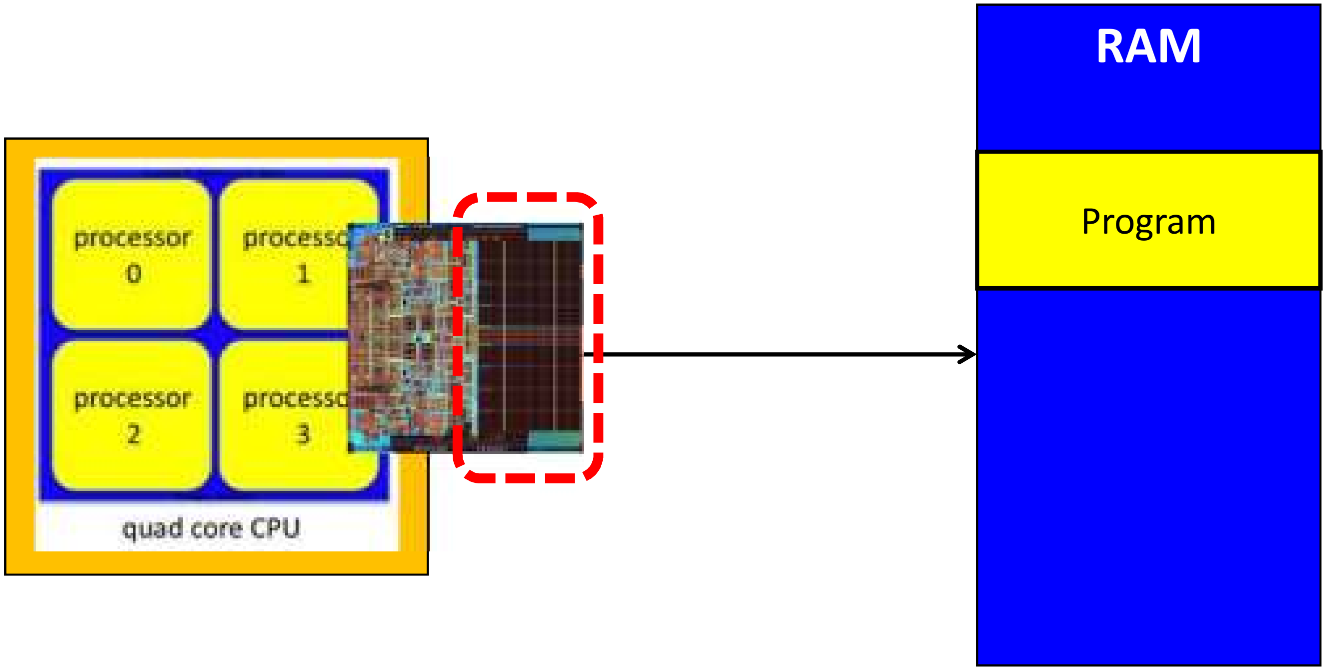
How about multicore?

- In a multicore world, the memory bottleneck is even worse
  - *more mouths to feed :-)*



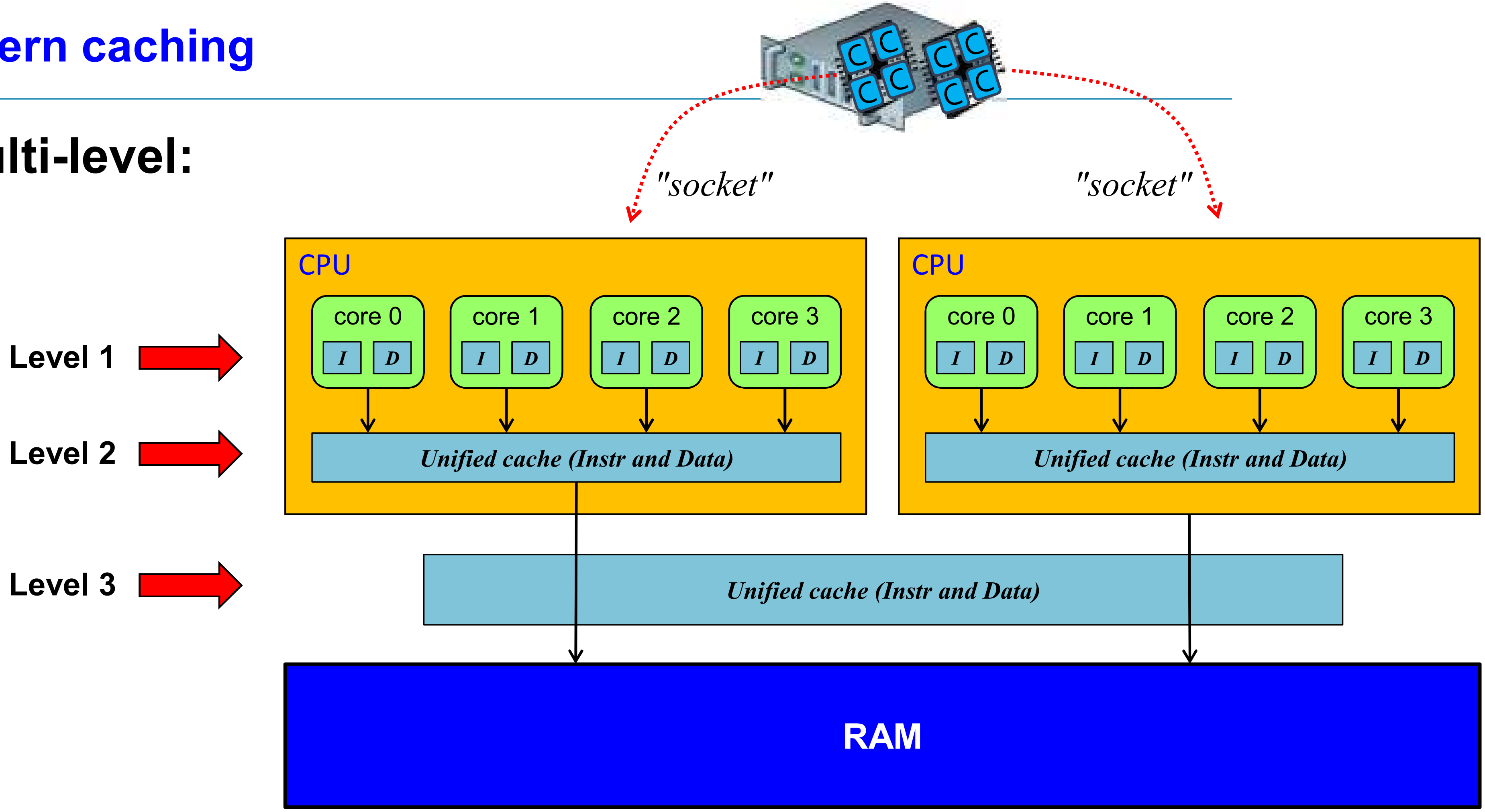
The most common solution?

- **Caching!**
  - *keeping copy of RAM closer to the cores…*



Modern caching

- **Multi-level:**



"socket"

"socket"

Level 1

Level 2

Level 3

CPU

core 0 I D

core 1 I D

core 2 I D

core 3 I D

Unified cache (Instr and Data)

CPU

core 0 I D

core 1 I D

core 2 I D

core 3 I D

Unified cache (Instr and Data)

Unified cache (Instr and Data)

RAM

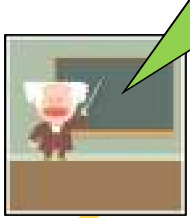
Design question #1

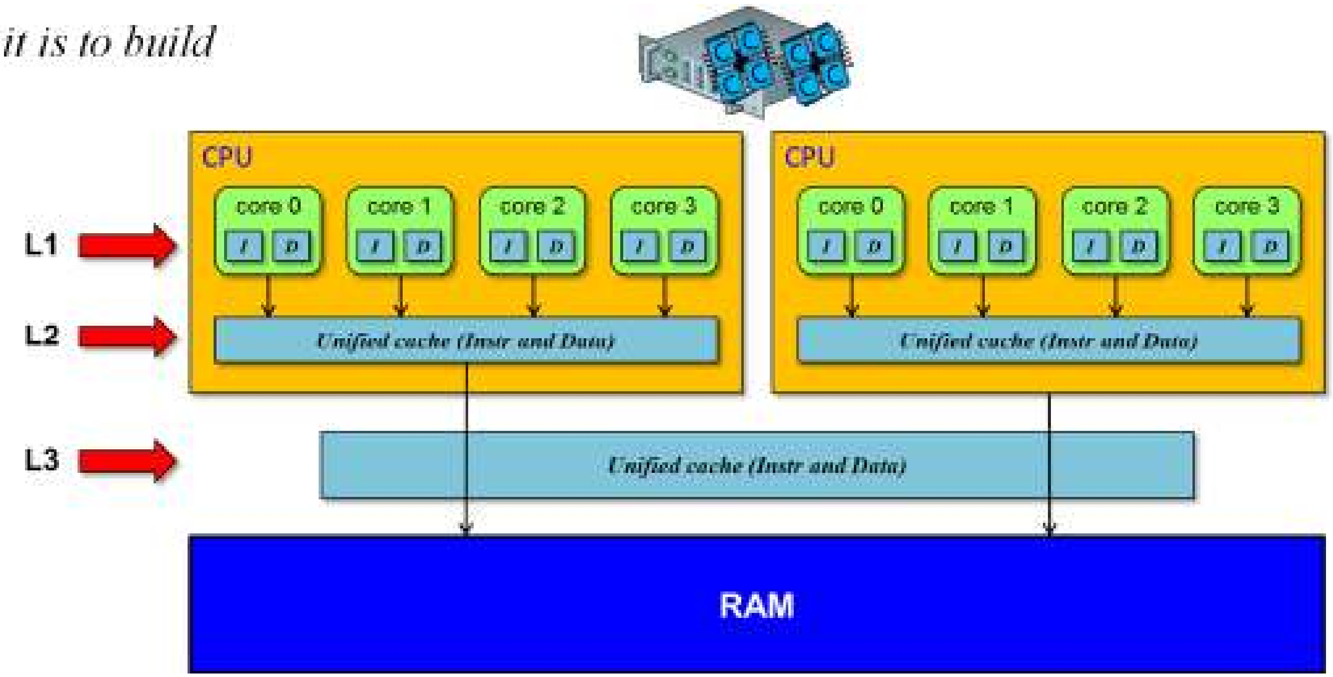
- **Cache speed vs. size**
  - *the closer to the core, the faster we need the cache to be*
  - *the faster the cache, the harder it is to build*

∴ The faster the cache, the smaller it is.

Typical sizes and access times (in CPU cycles):

L1: 32KB, 1 cycle
L2: 512KB, 10 cycles
L3: 4MB, 100 cycles
RAM: GBs, 1,000 cycles

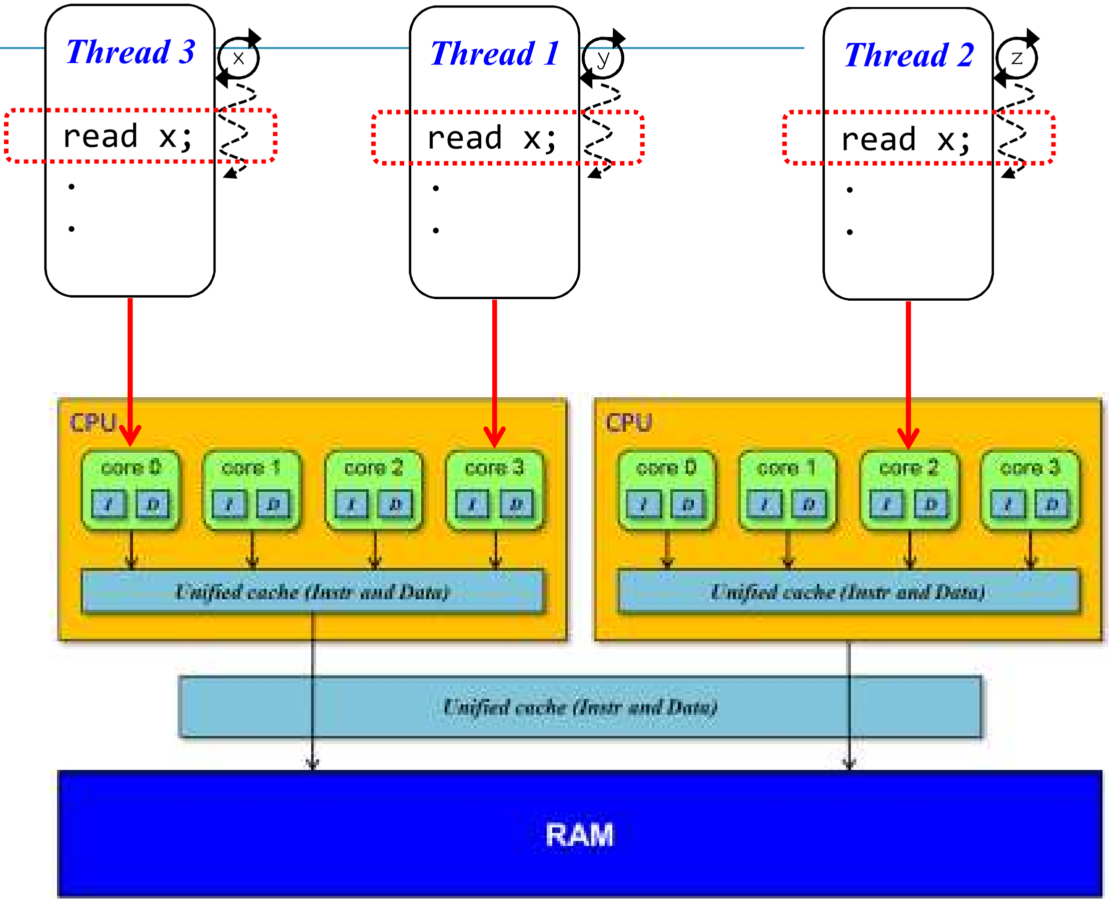



L1
L2
L3
CPU
core 0
core 1
core 2
core 3
I
D
Unified cache (Instr and Data)
CPU
core 0
core 1
core 2
core 3
I
D
Unified cache (Instr and Data)
Unified cache (Instr and Data)
RAM

Design question #2

- What happens as different cores read memory?

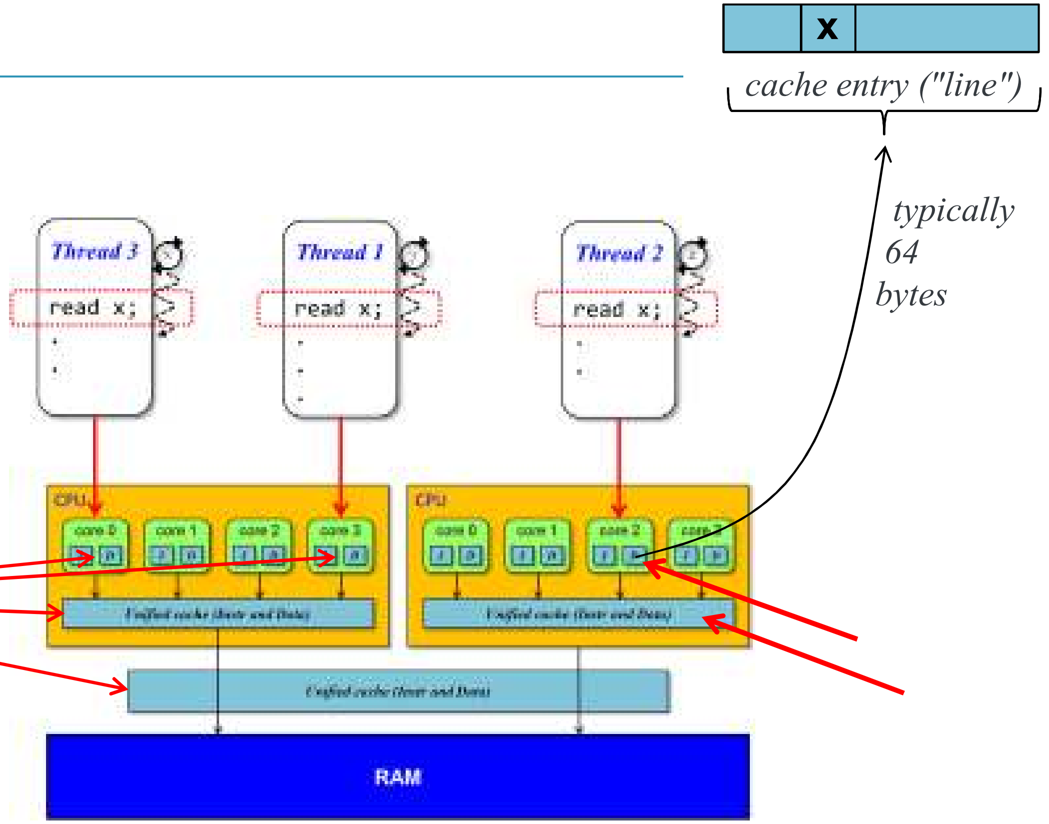
What do you think?



Reads…

- **Each core caches a copy**
- **Read policies:**
  - *inclusive => copy @ all levels*
  - *exclusive => copy @ most one level*

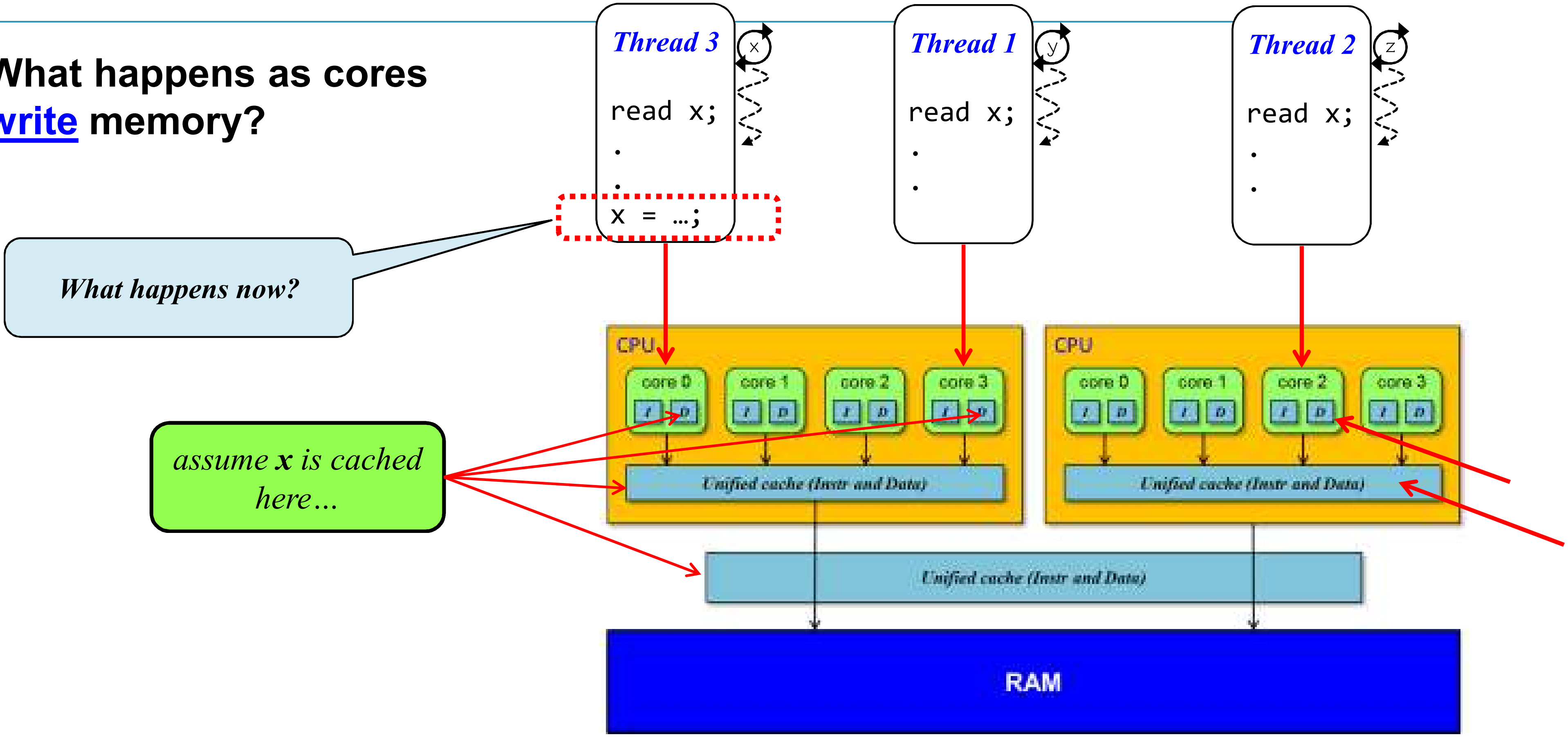
Assuming inclusive, x is cached at L3, both L2's, and 3 L1's…



- *inclusive is simpler / faster, exclusive holds more data overall*

Design question #3

- What happens as cores write memory?



Thread 3

```
read x;
.
.
x = …;
```

Thread 1

```
read x;
.
.
```

Thread 2

```
read x;
.
.
```

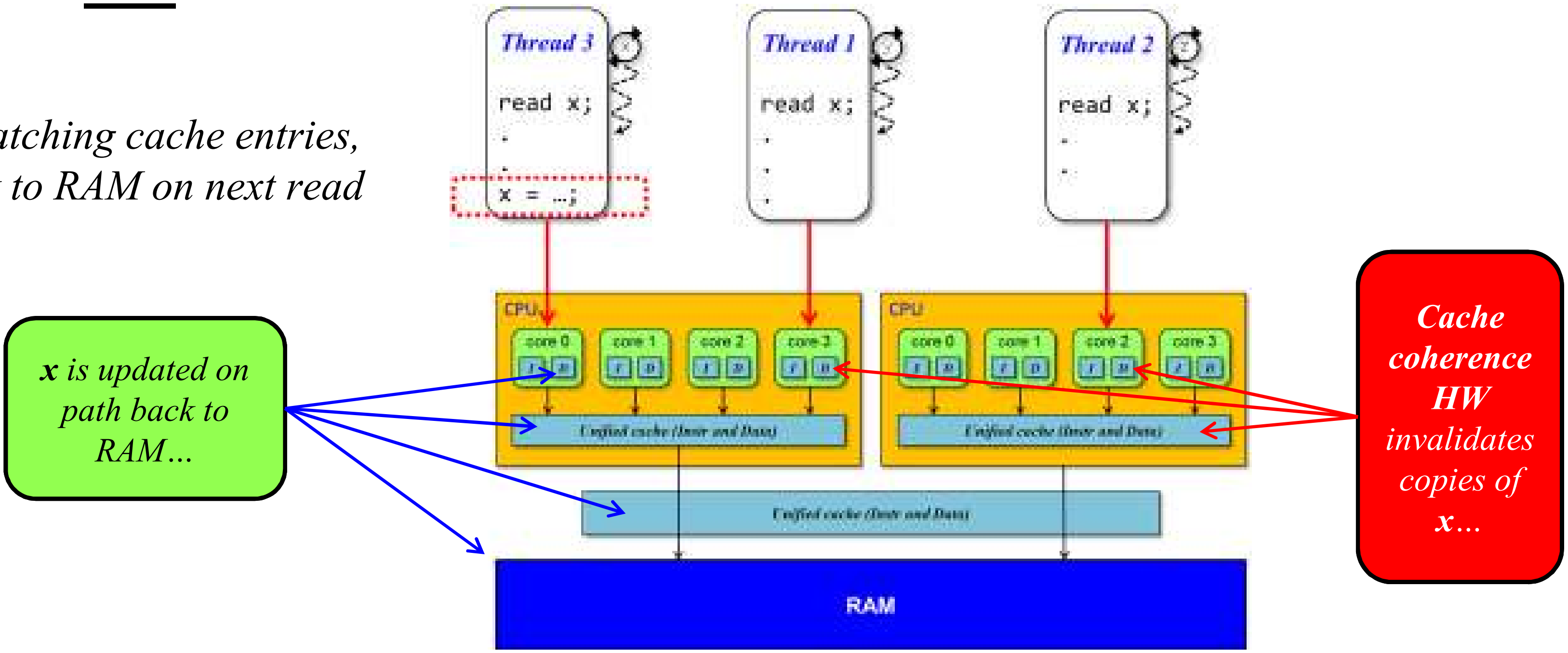
What happens now?

*assume **x** is cached here…*

x
y
z
CPU
core 0
core 1
core 2
core 3
I
D
Unified cache (Instr and Data)
CPU
core 0
core 1
core 2
core 3
Unified cache (Instr and Data)
Unified cache (Instr and Data)
RAM

Writes…

- **Cache coherence!**
- **On a write, HW ensures that all cores will eventually see <u>new</u> value**
- **How?**
  - *by invalidating matching cache entries, sending core back to RAM on next read*

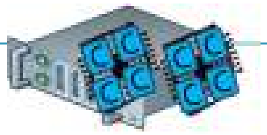


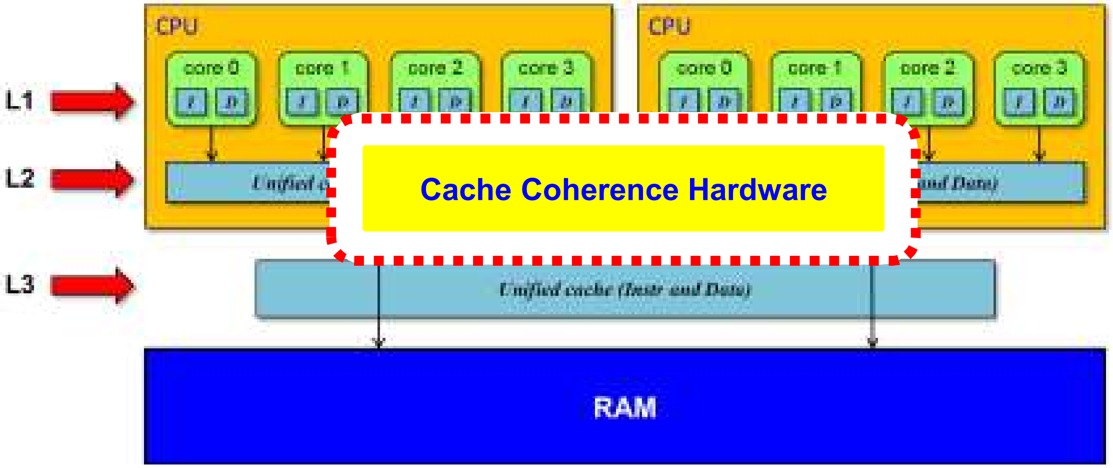
x *is updated on path back to RAM…*

Cache coherence HW *invalidates copies of* ***x****…*

Cache coherence…

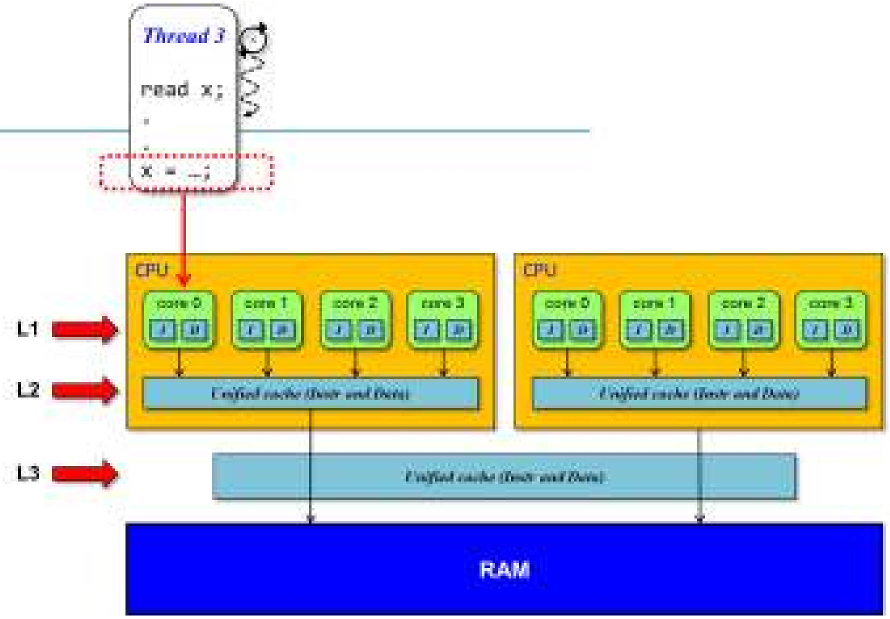
- **… is not cheap!**
  - *complex interconnections*
  - *must be fast*





When does write happen?

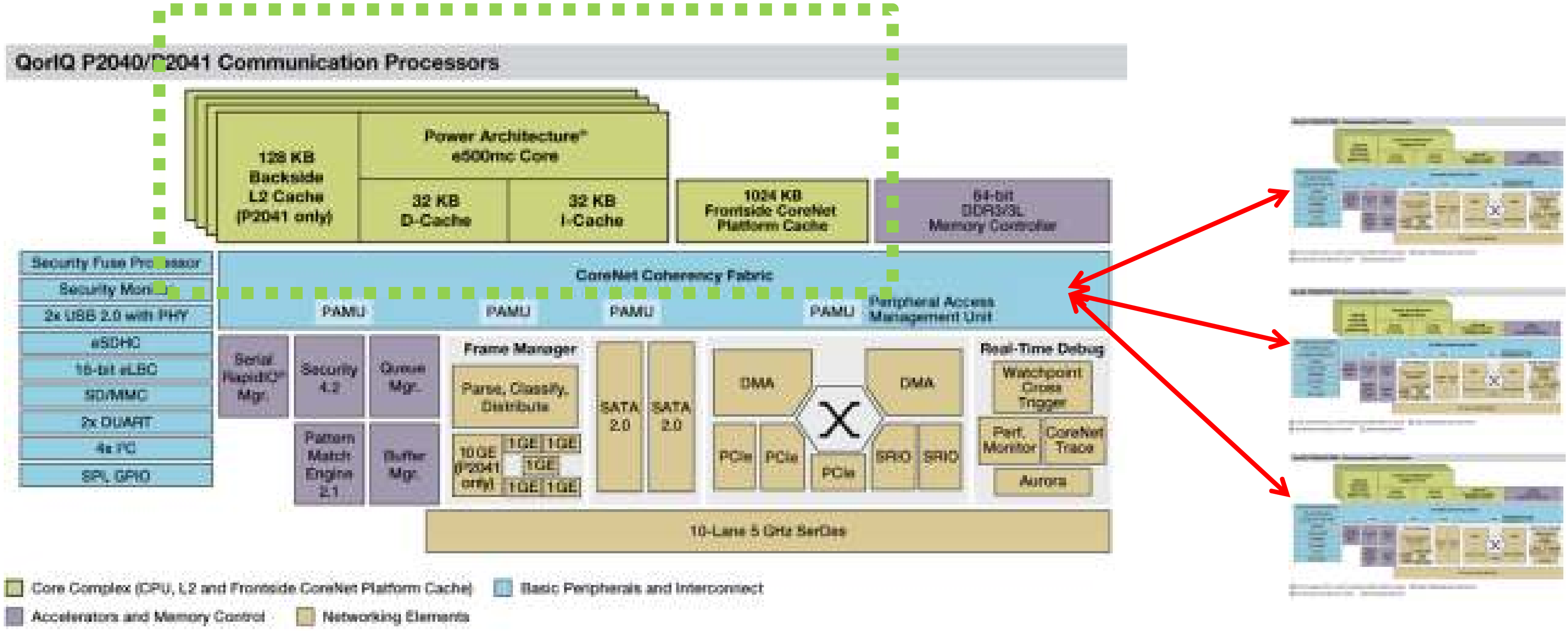
- **Often configurable**
- **Write policies:**
  - *write-through* => *write to RAM immediately*
  - *write-back* => *buffer and write later*



  - *Write-through is simpler / updates RAM sooner, write-back yields higher memory bandwidth*

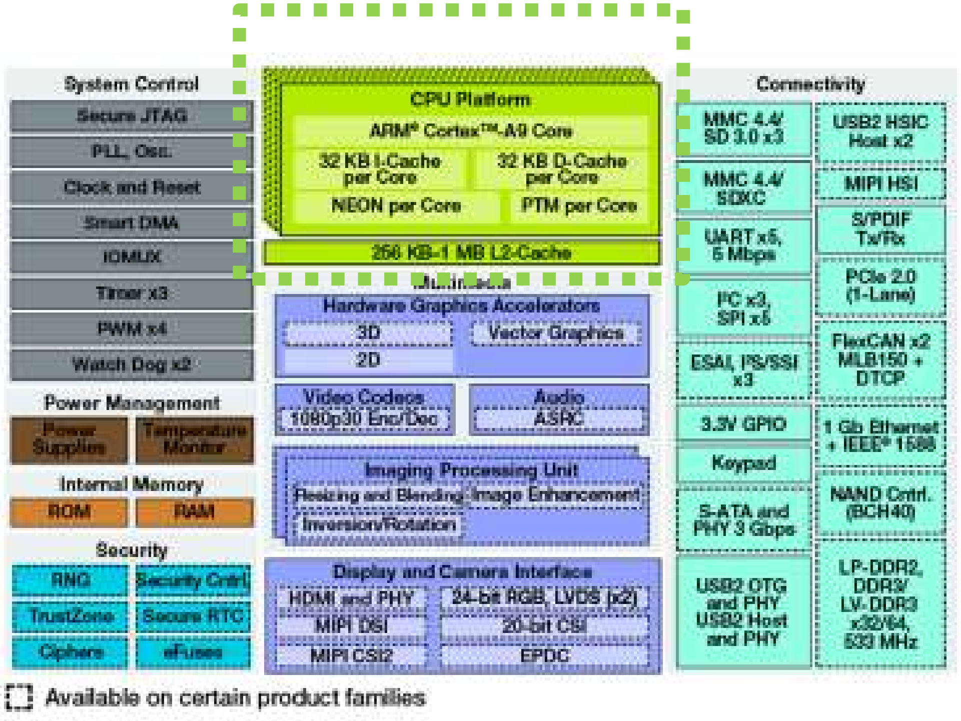
Example: QorIQ P2040/2041

- **Multi-level caching: L1, L2, L3**
  - *Configurable write-back or write-through*
- **Corenet switch fabric:**
  - *Fully interconnected (crossbar), cache coherent*
  - *Scalable...*



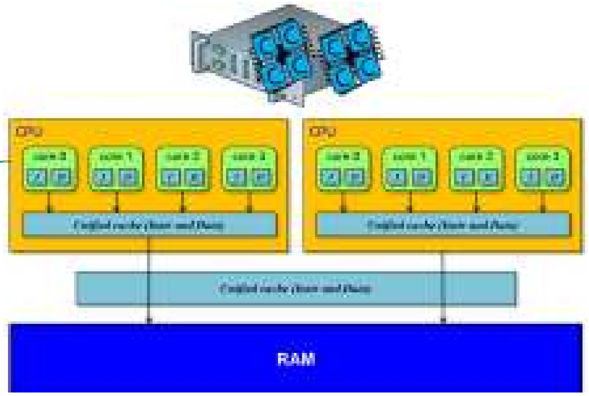
Example: i.MX 6Q

- **Multi-level caching: L1, L2**
- **Fully cache coherent**



Summary

Summary


CPU
core 0
core 1
core 2
core 3
Unified cache (Instr and Data)
CPU
core 0
core 1
core 2
core 3
Unified cache (Instr and Data)
Unified cache (Instr and Data)
RAM

- Multicore is the path forward for better performance

- But good performance is not easy:

HPC *==* *Parallelism* + *Memory Hierarchy* – *Contention*

- Thank you for attending!
  - *Email:* *joe@joehummel.net*
  - *Materials:* *http://www.joehummel.net/freescale.html*

www.Freescale.com