Computer Architecture

IE 496 Lecture 3

Reading for this lecture

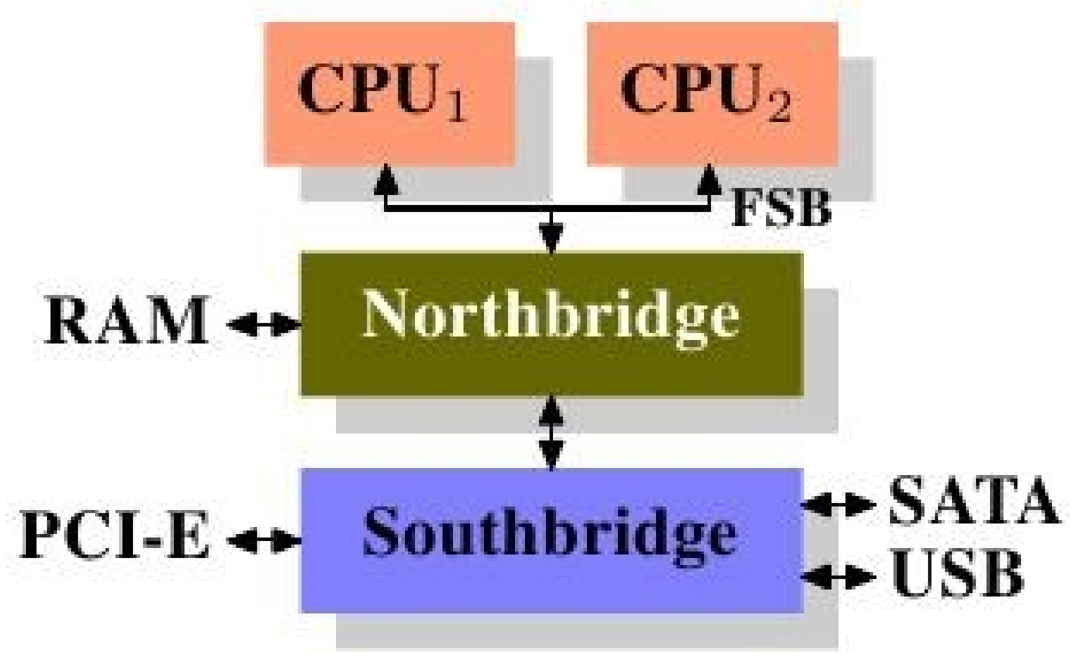
- “All You Ever Wanted to Know About Memory”, Ulrich Drepper

Modern Architectures

- Recent trends have led to the dominance of commodity hardware over specialized architectures.
- This might seems to obviate the need to learn about specialized parallel architectures and topologies.
- The increases in clock frequency of processors that largely followed Moore's Law have slowed dramatically.
- For the foreseeable future, additional capacity will be in the form of processors with multiple computing *cores.*
- This means in essence that all computers will be parallel computers.

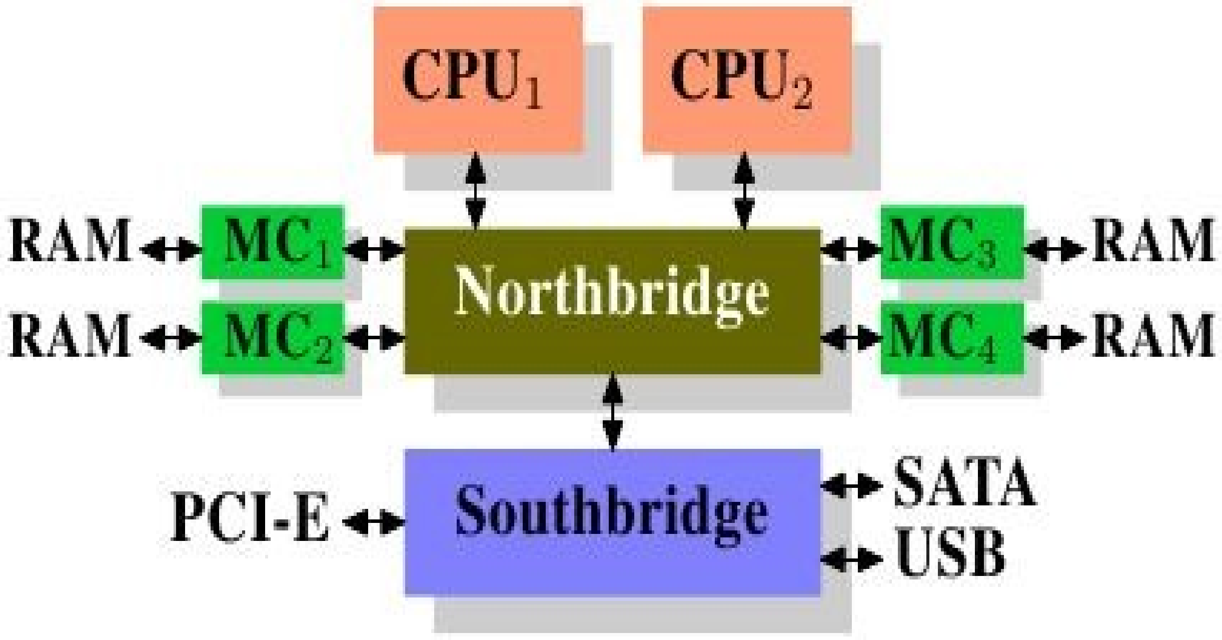
A More Realistic View

- This is a more detailed view of a modern architecture.
- Note the obvious bottleneck between CPU and memory.



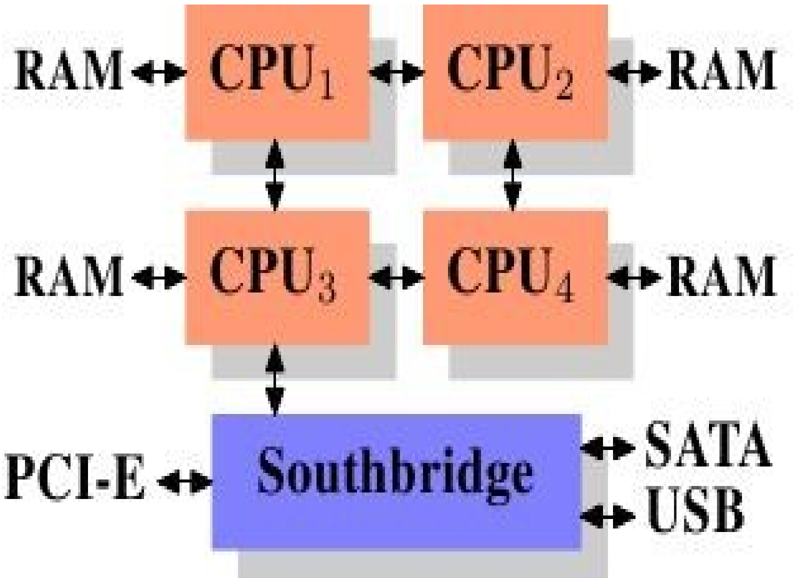
Overcoming the Bottleneck

- The bottleneck can be partially overcome with additional memory controllers.
- This increases complexity and expense.



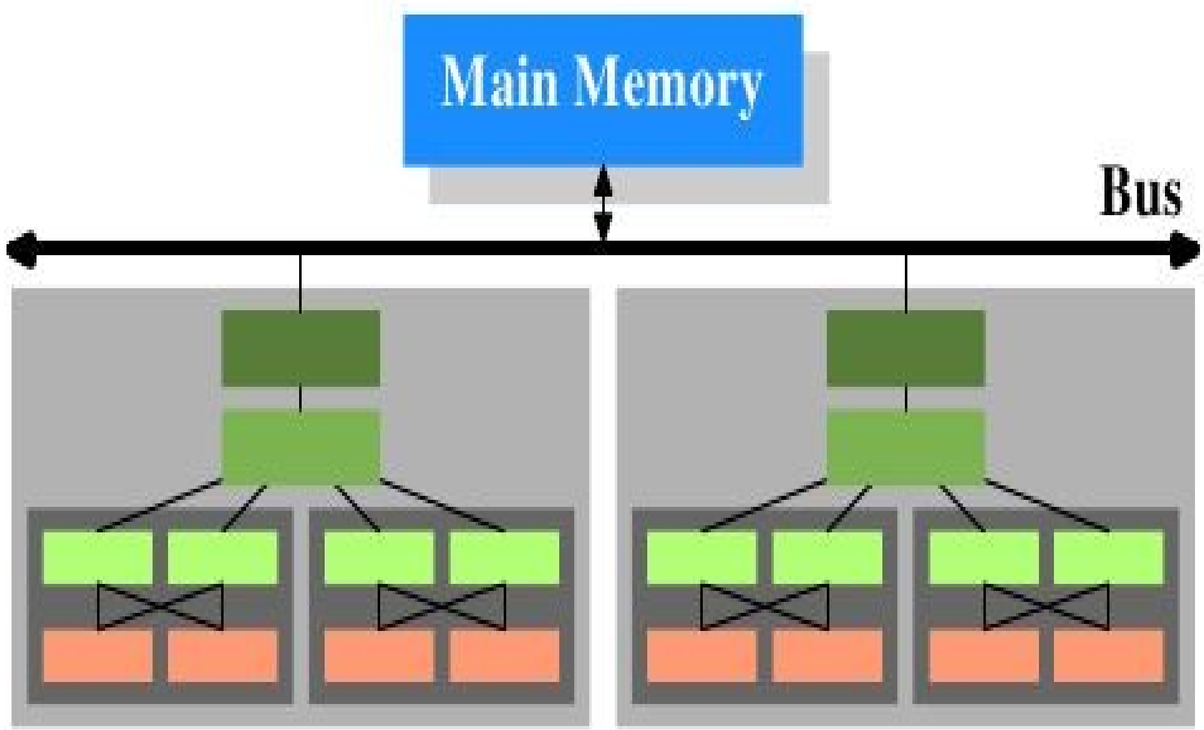
Another Option

- A second option is to attach each CPU to local memory.
- This creates a small parallel architecture with an associated interconnection topology.
- All memory appears local, but access times are not uniform (called a NUMA architecture).



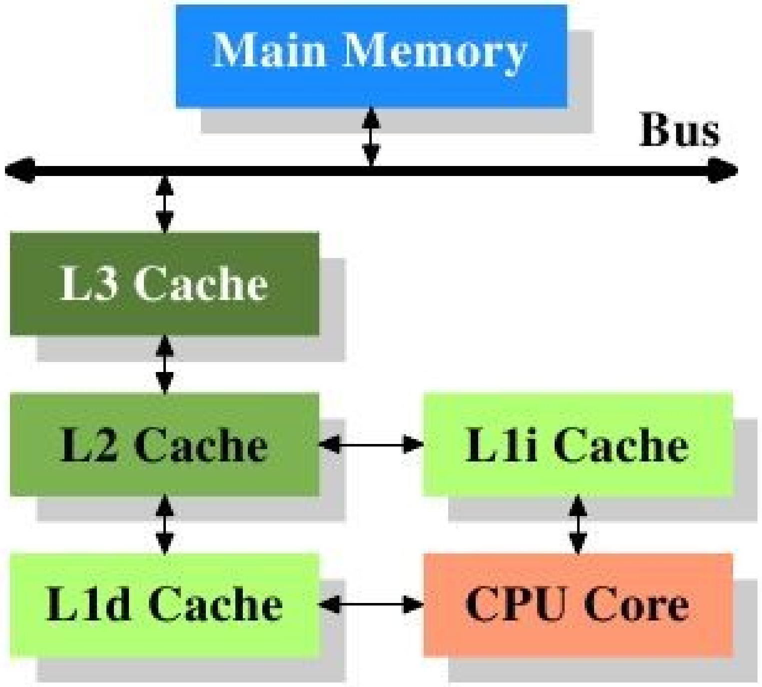
Putting it Together

- Today's architectures consist of multiple processors, each with multiple cores.
- The resulting memory hierarchy is very complex.



Memory Hierarchy Revisited

- To overcome the gap between processor and memory speeds, additional levels of cache can be added.
- There may be separate caches for instructions and data.

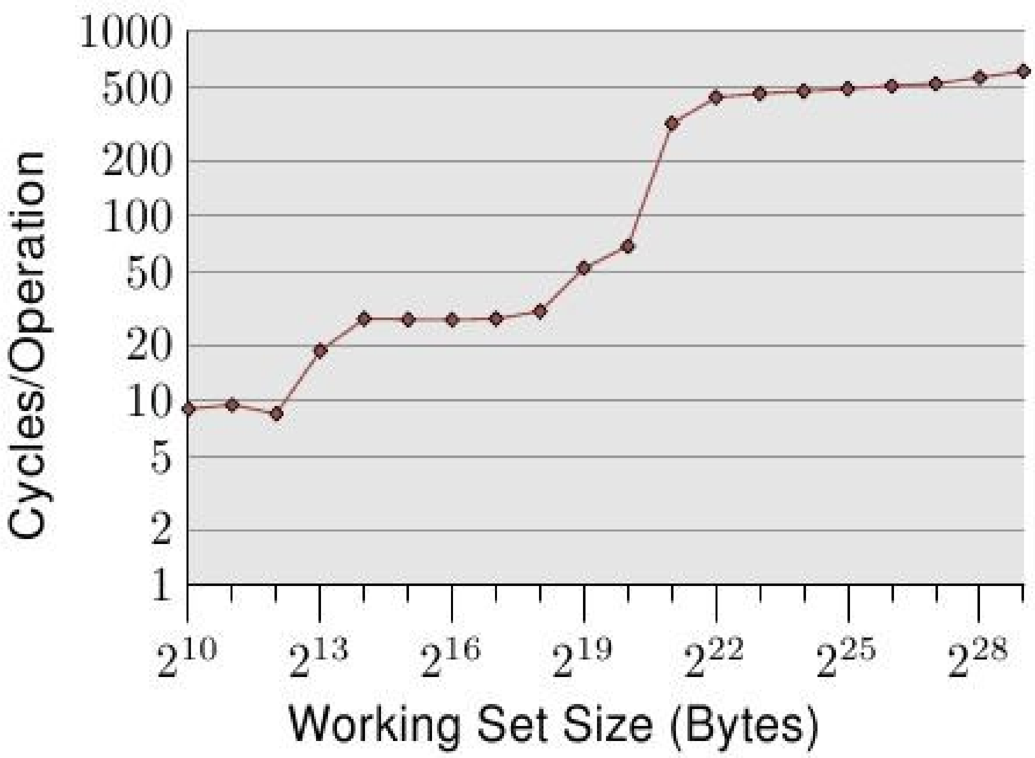


Access Times

- Here are some representative access times
  - Register: 1 cycle
  - L1d: 3 cycles
  - L2: 14 cycles
  - Main Memory: 240 cycles
- It is easy to see why it's important to understand the hierarchy.

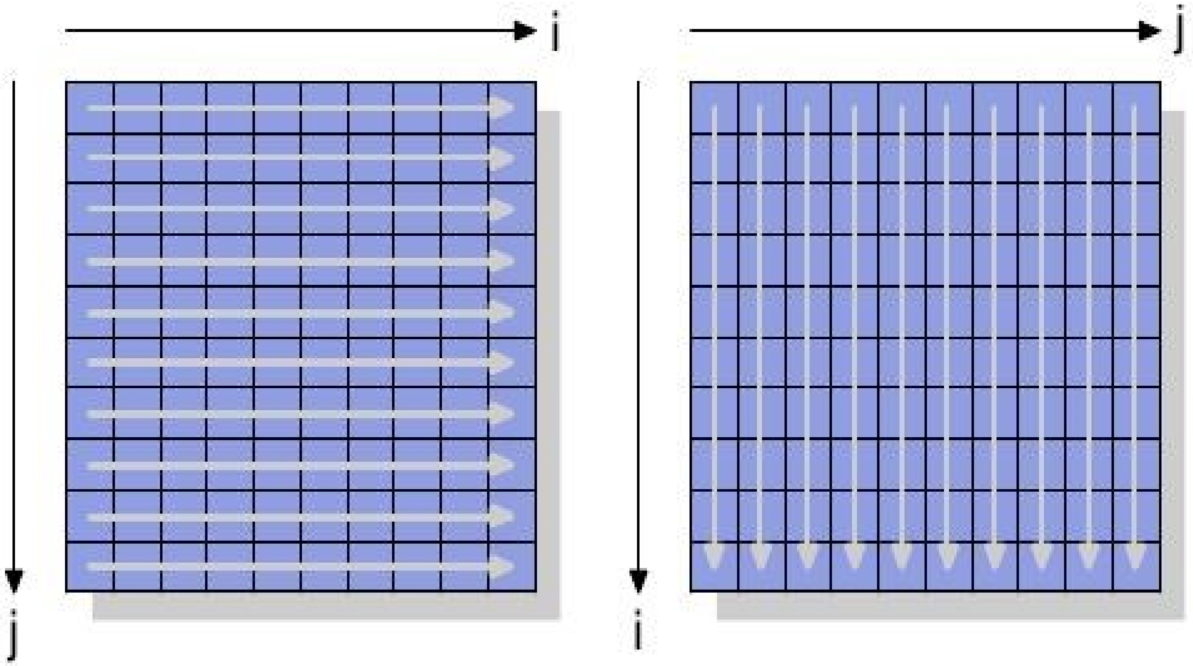
Example

- The following data were generated by random accesses into memory blocks of different sizes
- The plateaus correspond to the sizes of the caches



A Second Example

- Consider the time to initialize a matrix.
- The matrix is stored row-wise.
- We consider initializing column-wise and row-wise.



Results

- Here are the average access times

	Inner Loop Increment	
	Row	Column
Normal	0.048s	0.127s
Non-Temporal	0.048s	0.160s

- Nontemporal writes bypass the cache and go directly to memory.
- When writing the matrix column-wise, the cache actually hurts us.
- The data we need to access next will never be in the cache.

A Third Example

- Now consider multiplying two matrices.
- A straightforward implementation would be

```
for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
      for (k = 0; k < N; ++k)
        res[i][j] += mul1[i][k] * mul2[k][j];
```

- It is natural to access one matrix row-wise and the other one column-wise, but this is bad.
- A solution is to transpose one of the matrices first.
- Does this make sense?

The Improved Code

- Here is the new code

```
double tmp[N][N];
 for (i = 0; i < N; ++i)
   for (j = 0; j < N; ++j)
     tmp[i][j] = mul2[j][i];
 for (i = 0; i < N; ++i)
   for (j = 0; j < N; ++j)
     for (k = 0; k < N; ++k)
       res[i][j] += mul1[i][k] * tmp[j][k];
```

- And the comparison

	Original	Transposed
Cycles	16,765,297,870	3,922,373,010
Relative	100%	23.4%

A Little More on Caching

- We can do even better if we understand exactly how caches work.
- Data is cached in lines that have a fixed length.
- Therefore, if we copy one element from an array into the cache, we will also get the next few elements for free.
- To get maximum performance, we should use all the data in the cache that we can before it gets evicted.
- In the matrix example, this means that we should do several inner products at the same time.

A Third Implementation

- We can get the size of a cache line in gcc as follows

```
gcc -DCLS=$(getconf LEVEL1_DCACHE_LINESIZE) ...
```

- The improved code is then

```
#define SM (CLS / sizeof (double))

for (i = 0; i < N; i += SM)
    for (j = 0; j < N; j += SM)
        for (k = 0; k < N; k += SM)
            for (i2 = 0, rres = &res[i][j], rmul1 = &mul1[i][k]; i2 < SM;
                 ++i2, rres += N, rmul1 += N)
                for (k2 = 0, rmul2 = &mul2[k][j]; k2 < SM; ++k2, rmul2 += N)
                    for (j2 = 0; j2 < SM; ++j2)
                        rres[j2] += rmul1[k2] * rmul2[j2];
```

- This looks a bit messy, but it's efficient! 10x speedup

	Original	Transposed	Sub-Matrix	Vectorized
Cycles	16,765,297,870	3,922,373,010	2,895,041,480	1,588,711,750
Relative	100%	23.4%	17.3%	9.47%

Cache Coherency

- A challenge for shared memory architectures is to maintain “cache coherency.”
- Since each core may have its own cache, there may be multiple copies of the memory location.
- If a cached copy is written, then it becomes “dirty” and other cached copies are invalidated.
- This can lead to inefficiency if different cores are trying to access the same memory locations simultaneously.

Processes and Threads

- In general, a computer can have multiple *processes* executing simultaneously.
- These processes have separate memory address spaces and have no direct means of communicating.
- A process can, however, have multiple *threads* that execute independently but memory.
- In a multi-core system, different threads from the same process can execute on different cores.

Hyperthreading

- Hyperthreading is a technique for allowing multiple threads to execute efficiently using the same core.
- When one thread is idle due to a cache miss (i.e., waiting for data to be retrieved), other threads can be run.
- In practice, this may create speed-ups similar to what one would observe with multiple cores.