Tools and techniques for optimization and debugging

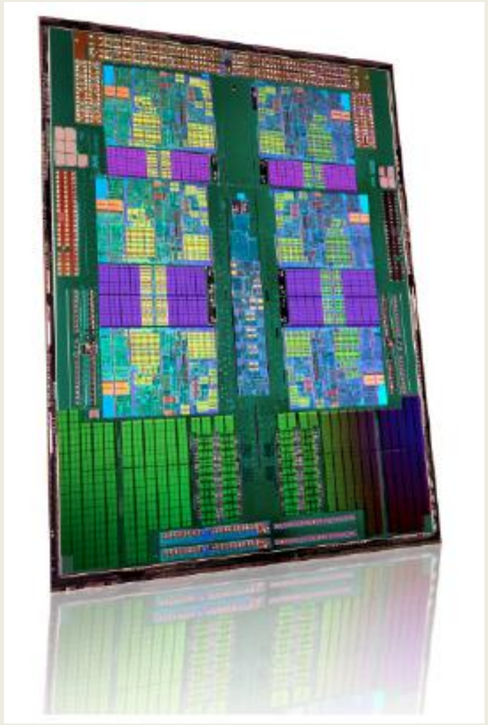
Fabio Affinito, Andrew Emerson

November 2016

Fundamentals of computer architecture

Serial architectures

Introducing the CPU

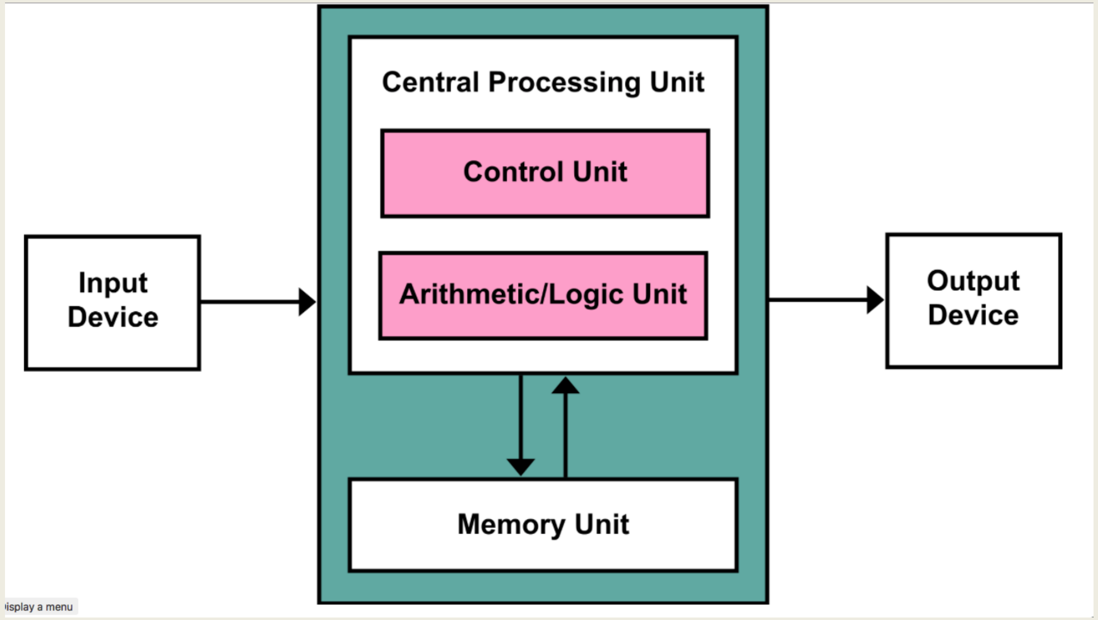


It's a complex, modular object, made of different units...

We want to understand how it works.

Before consider all the features, we will start with a very important abstraction.

Von Neumann architecture



Elements are:

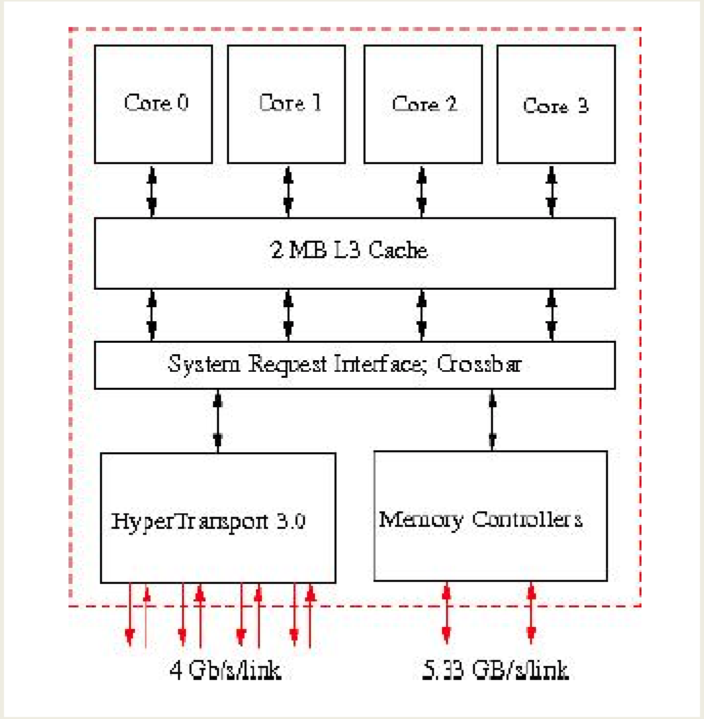
- I/O devices
- a Central Processing Unit
- a memory

Things are more complicated

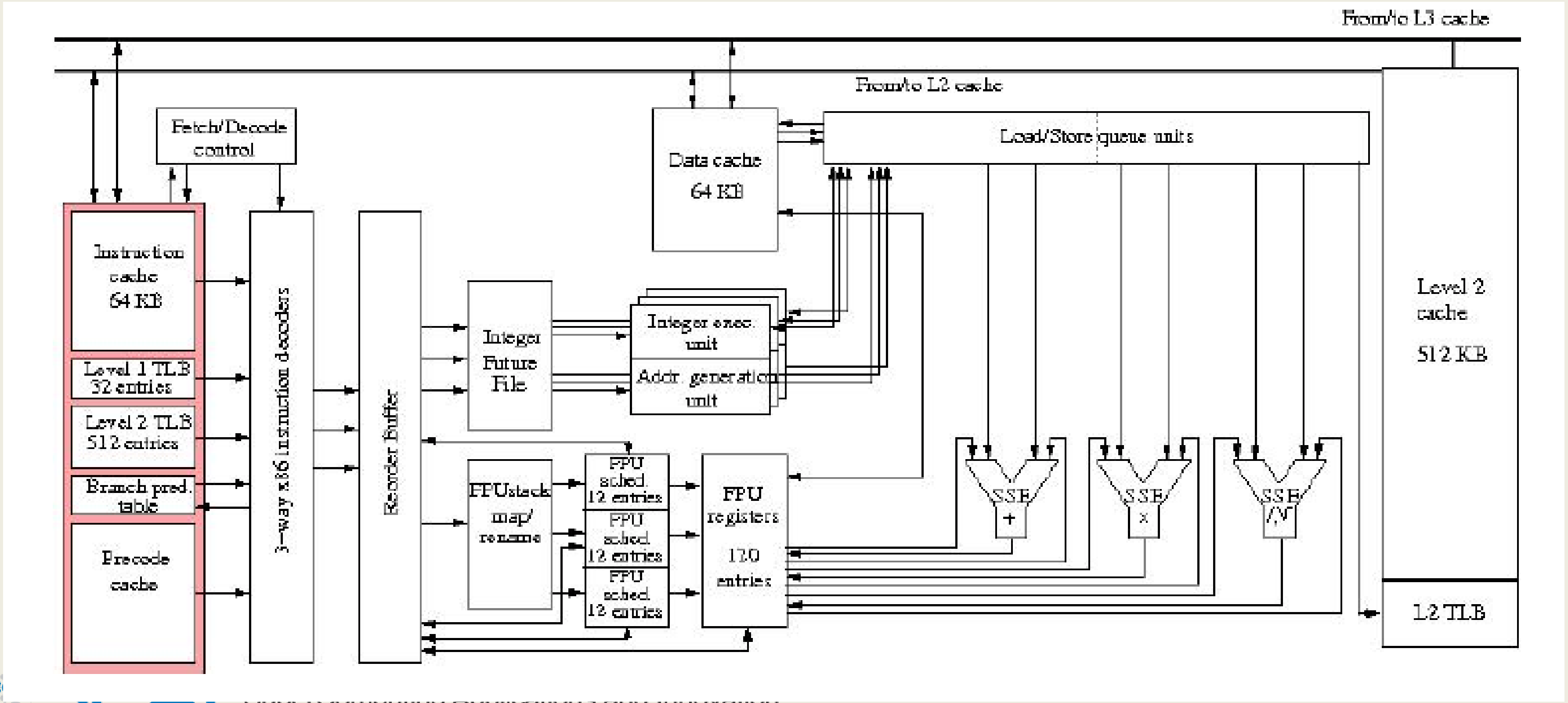
CPU is organized in different units, each of them is specialized in a specific kind of operation (depending on the type of operand)

Memory is hierarchically structured, from main memory to registers.

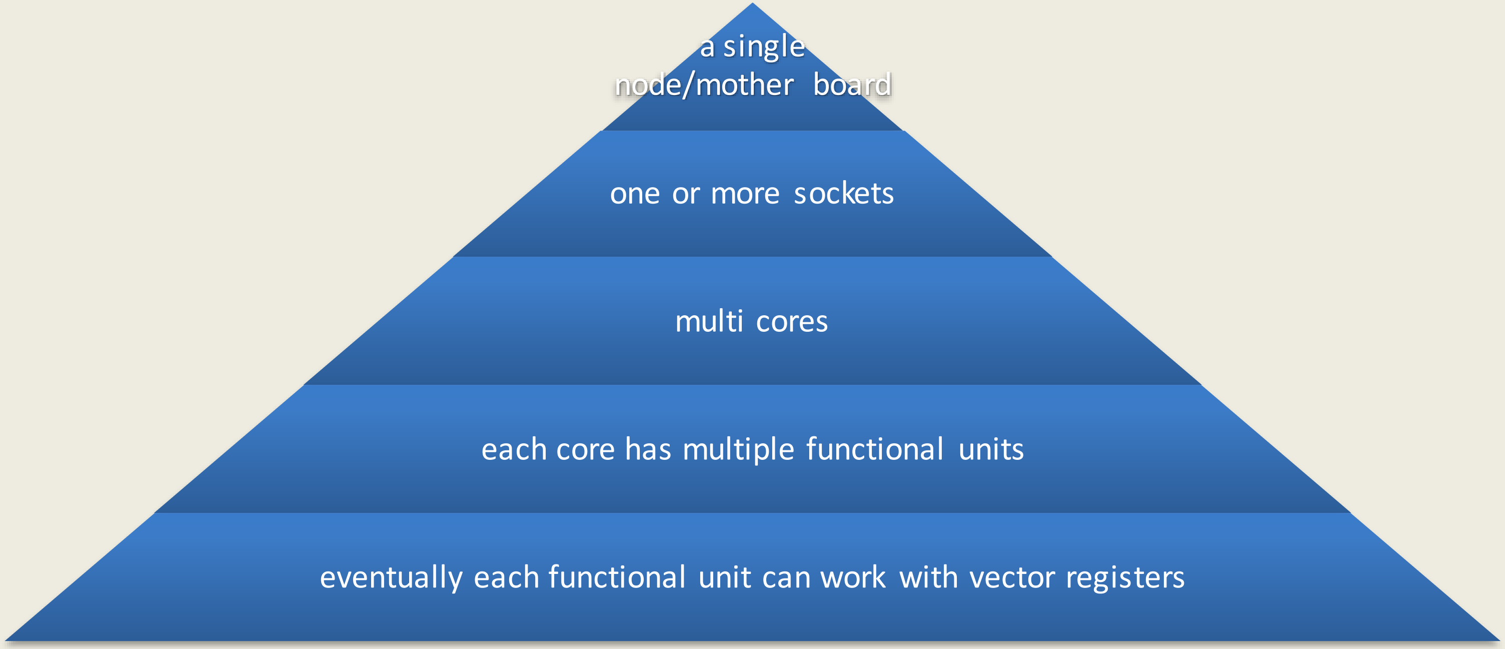
Operations can be performed simultaneously and not in a prefixed order



What's inside a core?



Putting things together


a single node/mother board
one or more sockets
multi cores
each core has multiple functional units
eventually each functional unit can work with vector registers

What about the performance?

The performance of a computer depends on several factors:

- clock frequency
- access bandwidth to the memory
- number of cores
- efficiency of functional units
- vector registers (SIMD)
- quality of the software

Clock frequency

The clock frequency defines the most elementary step for the CPU. Any given operation performed by the CPU can require one or more than one clock cycle.

In order to not waste performance, it is essential that the CPU has always work to do. This can be enforced using a proper pipelining.

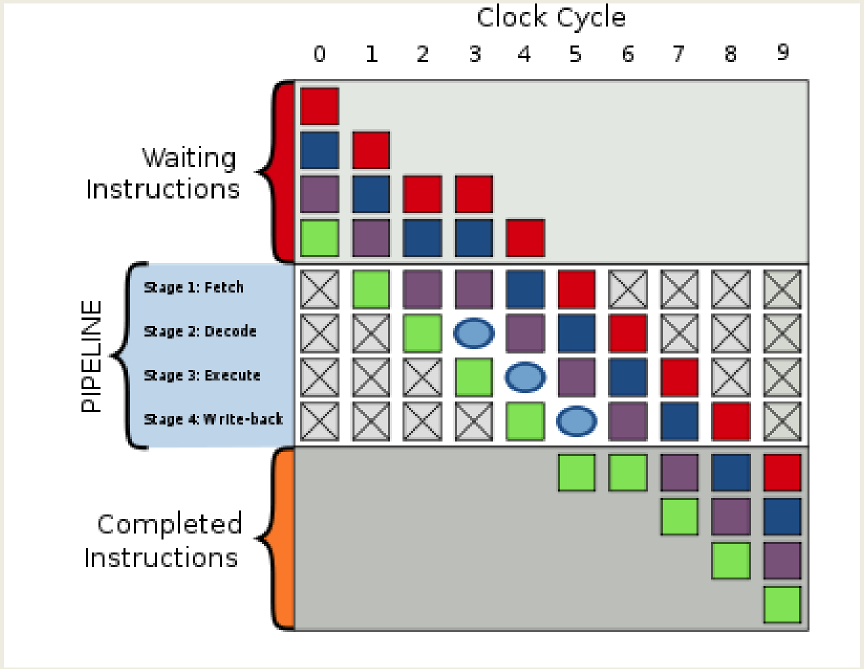
Pipelining

An instruction pipeline is a technique used in the design of computers to increase their instruction throughput (the number of instructions that can be executed in a unit of time). The basic instruction cycle is broken up into a series called a pipeline. Rather than processing each instruction sequentially (one at a time, finishing one instruction before starting the next), each instruction is split up into a sequence of steps so different steps can be executed concurrently (at the same time) and in parallel (by different circuitry).

(from Wikipedia)

Pipelining

If the flux of instructions to the CPU is high enough, the pipelining ensures that the processor is always busy and that there are not stall cycles.



In-order vs out-of-order

To increase the efficiency of the processor, sometimes it could be useful to interchange the order of the execution of the instructions (preserving the semantics). Processor that are able to do so, are called “out-of-order” (OOO) processors.

If this feature is not present, the processor is called “in-order”

Memory hierarchies

Typically, a large memory is also very slow. It is totally inefficient to operate directly to the main memory.

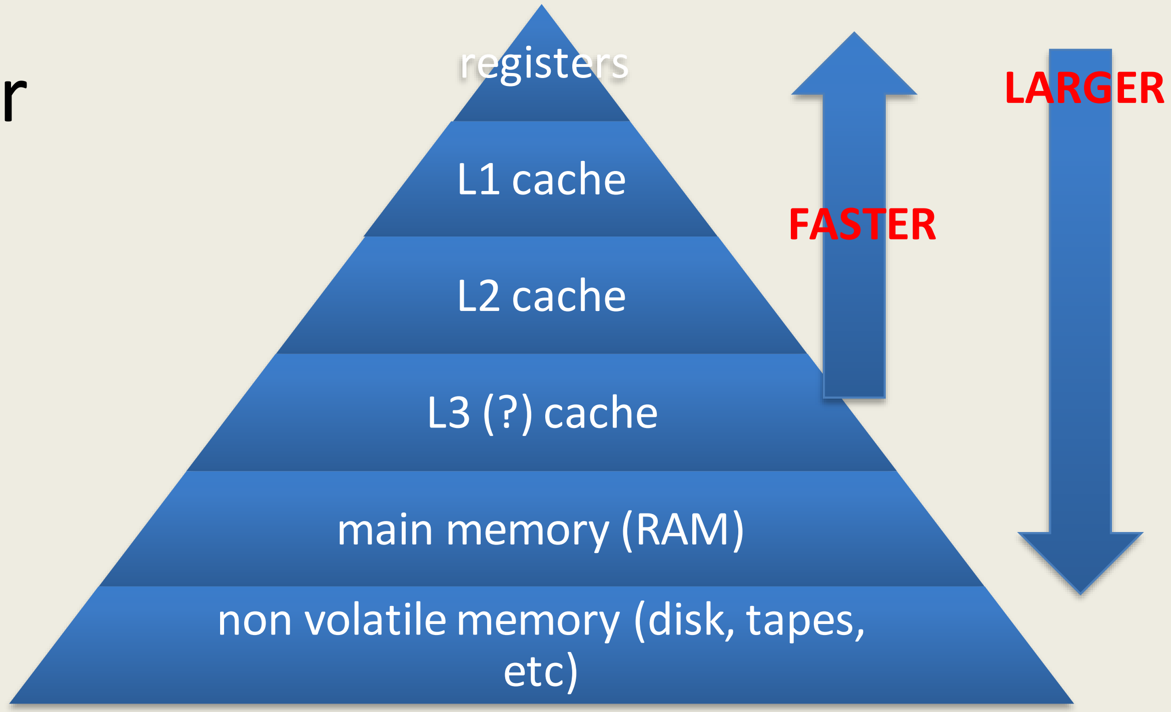
100 to 1000 cycles can be necessary to move data from the memory to the CPU

Caches are smaller portions of memory that are accessible in very fast way.

Registers are very, very small memories, but they are directly accessible by the functional units.

Memory hierarchies

- larger = slower
- closer = faster



Memory performance

Two of the crucial factor affecting memory performance are:

- Latency: how long does it take to retrieve a data from the memory?
- Bandwidth: how many data can be moved from/to the memory? what is the data rate that can be sustained?

Memory subsystem in detail

Registers: Highest bandwidth, lowest latency memory that a modern processor can access.

They are built inside the CPU

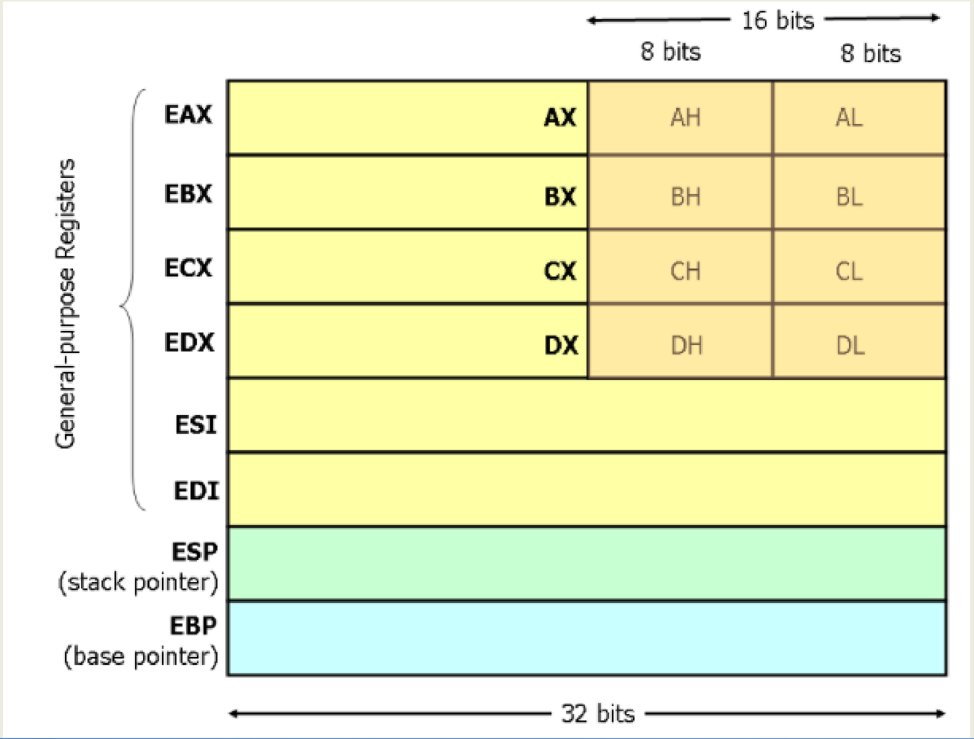
They can contain a very small amount of data.

Specialized according to the kind of data (integers, floats, logical)

Functional units can operate directly (and exclusively) on data contained on registers

Memory subsystem in detail

x86 registers



A bit of assembly

Assembly code, describes the data movement and data operations acting on the registers.

For example:

```
mov eax, [ebx]              ; Move the 4 bytes in memory at the address contained in EBX into EAX
mov WORD PTR [ebx], 2  ; Move the 16-bit integer representation of 2 into the 2 bytes starting at the address in EBX
```

32bit vs 64bit

Registers contain also the addresses in which data are stored in memory.

Historically first registers were 32-bit wide. This meant that the largest address contained in a register was 2^{32}-1= 4096MB

Hence, 32-bits architectures cannot address a memory larger than 4GB

Data caches

Data caches are necessary to "fill the gap" between the main memory (large and slow) and the registers (small and fast).

Different levels of caches do exist.

L1 is the closest to the registers, L2 (and eventually L3) are larger than L1 but slower.

L2 is typically 10 to 100 times larger than L1

Data caches

Caches are organized in lines.

A cache line is the smallest unit of data that can be moved btw the main memory and the cache (or btw different levels of cache).

A cache line is made of N (N=8, 16) sequentially stored words

Multi-core systems

In the beginning we had the single core – single CPU. Performance depended mostly on the clock frequency. The increase of the clock frequency became soon not sustainable (power consumptions scales with the square of the clock frequency).

Number of computing units began to increase (power consumption scales linearly). At the same time, the density also increased. This was known as multi-core era.

Then number of cores grew again (from 2-4 to 8-16 per chip, and even more for GPUs). This is knows as many-core era.

Multi-core and caches

Node-board contains more than one socket, each of them containing in turn more than one core (multi-core nodes).

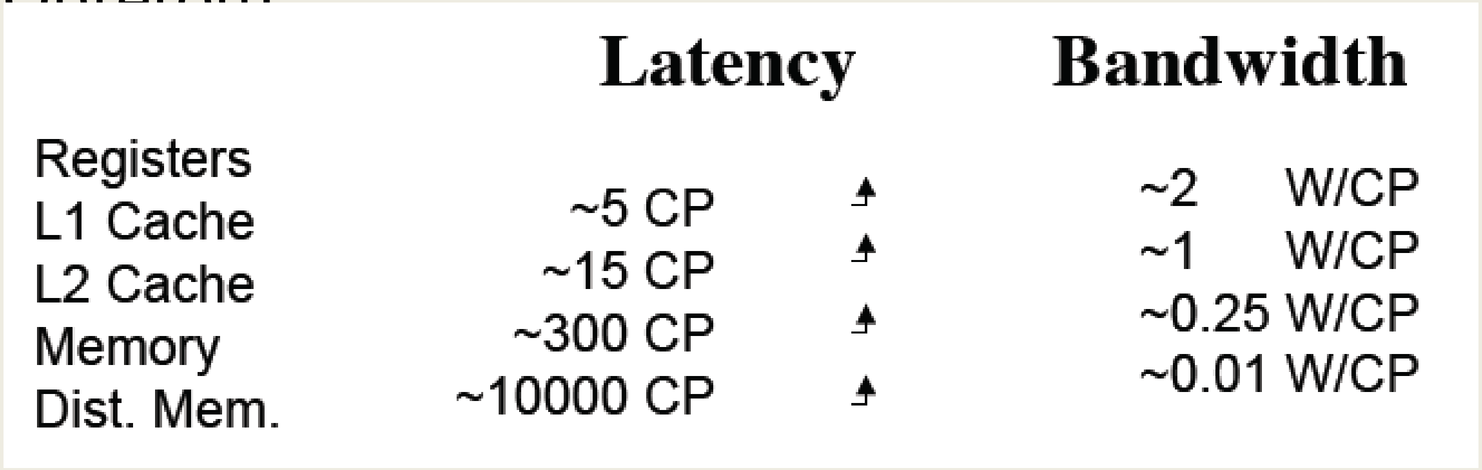
A multicore, usually, shares the main memory and the L2 caches.

The L1 cache is usually not shared among different cores

Cache coherency problem: conflicting access to duplicated cache line

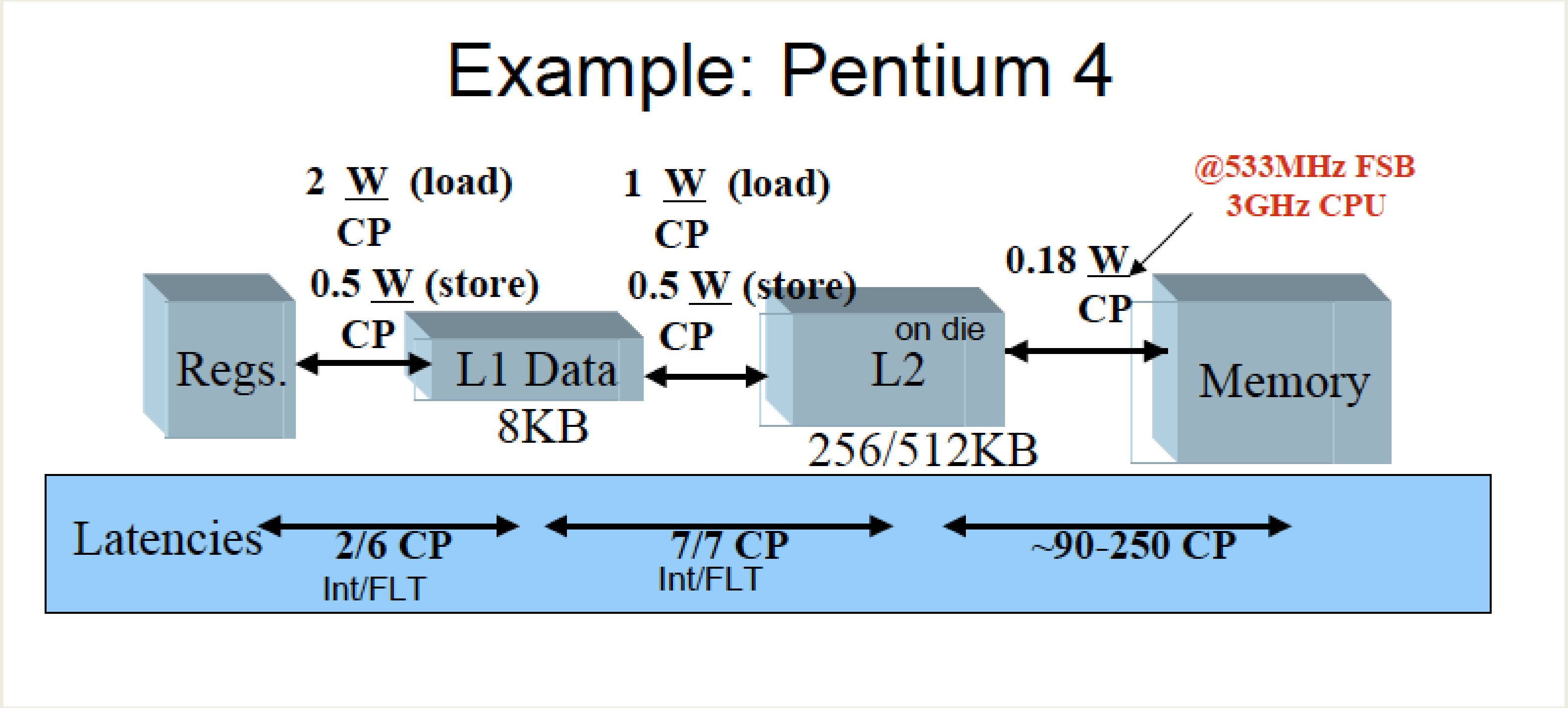
Examples

Typical values of latency and bandwidth (AMD Opteron):



	Latency		Bandwidth
Registers			
L1 Cache	~5 CP	↑	~2 W/CP
L2 Cache	~15 CP	↑	~1 W/CP
Memory	~300 CP	↑	~0.25 W/CP
Dist. Mem.	~10000 CP	↑	~0.01 W/CP

Examples


Example: Pentium 4
2 W/CP (load)
0.5 W/CP (store)
1 W/CP (load)
0.5 W/CP (store)
0.18 W/CP
@533MHz FSB
3GHz CPU
Regs.
L1 Data
8KB
on die
L2
256/512KB
Memory
Latencies
2/6 CP
Int/FLT
7/7 CP
Int/FLT
~90-250 CP

Cache access

Data can be stored in

- main memory
- L2/L1 caches

and this is unknown “a priori” to the programmer.

Data is fetched into the registers from the highest level where it can be found. If the data is not found in a given level, it will be searched in the lowest one.

Cache access

Access is transparent to the programmer...

... adopting a set of measures, you can make it better.

Cache access

- if you access two data that are stored far apart, then it is likely that those data are not kept together into the cache
- if you use an element twice, don't wait too long between the two events
- if you loop over data, try to take chunks of less than cache size

Data locality

When you write a code, try to keep the same variable as long as possible in the registers or in the cache.

For example, y[i]:

```
for (i=0; i<m; i++){
  for (j=0; j<n; j++){
    y[i]=y[i]+a[i][j]*x[j];
  }
}
```

Hits, misses, thrashing

- Cache hit: the data needed in the registers is found in the cache
- Cache miss: the data is not found in the cache, so it should be searched in the next level cache or in the memory
- Cache trashing: two needed data elements are not contained in the same cache line. In order to load the second one, the cache must be emptied and reloaded. This affects badly the performances.

Cache mapping

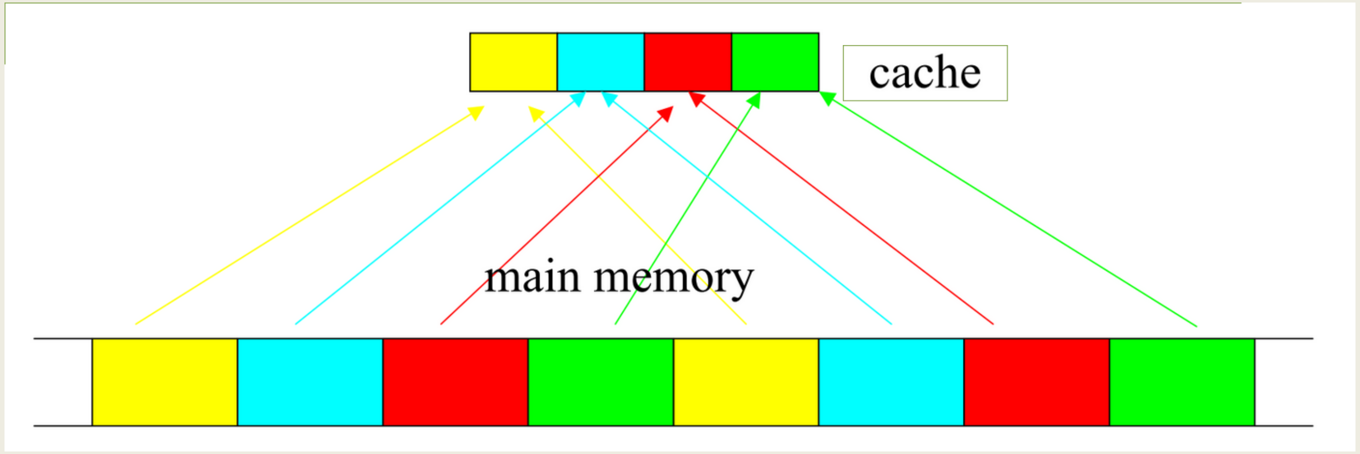
There is a need for a correspondance (or mapping) between the cache and next closer level of memory.

There are 3 types of mapping:

- direct
- set associative
- fully associative

Direct mapping

A given block from the main memory can be placed only in a given place in the cache.



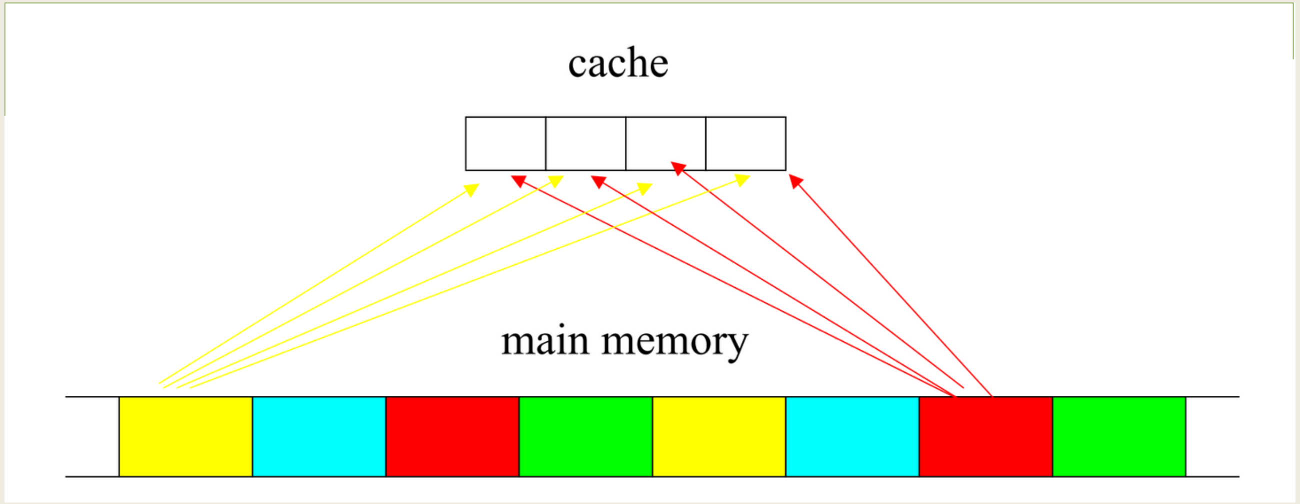
Direct mapping

```
Double a[8192], b[8192];
for (i=0; i<n; i++){
    a[i]=b[i];
}
```

-Example: cache size 64k = 2^{16} bytes = 8192 words
- a[0] and b[0] are exactly one whole cache apart. When use direct mapping, they are mapped to the same cache location
- Cache line is 4 words, so:
- b[0], b[1], b[2], b[3] are loaded
- when a[0], a[1], a[2], a[3] are loaded the first 4 values of b are thrashed
- but then b[1] is requested, so b[0], b[1], b[2], b[3] are loaded again
- a[1] is requested so a[0], a[1], a[2], a[3] are loaded and b[0], b[1], b[2], b[3] are thrased....

Fully associative

A block from main memory can be mapped to any location in the cache. It requires a lookup table.

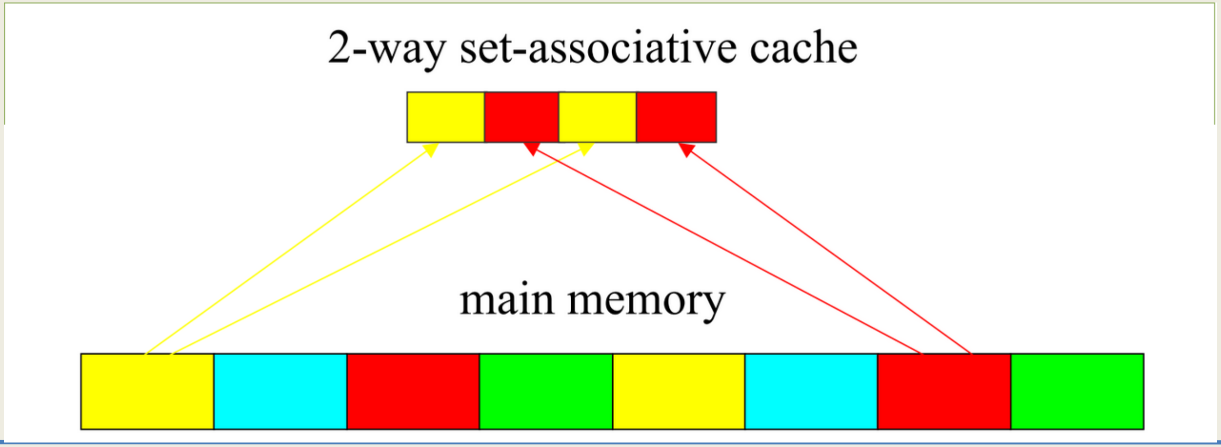


Fully associative

It looks as the ideal solution where any memory location can be associated with any cache line but it has a cost proihibitive.

Set associative

It is a compromise between full associative and direct mapped caches. In a n-way set associative cache, a block from the main memory can be placed in n (where n is at least 2) locations of the cache.



Set associative

- Direct-mapped caches are 1-way set associative caches
- In a k-way set associative cache, each memory region can be associated with k cache lines
- Fully associative is k-way with k equal to the number of cache lines

Intel Woodcrest caches

- L1
  - 32 KB
  - 8-way set associative
  - 64 byte line size
- L2
  - 4 MB
  - 8-way set associative
  - 64 byte line size

TLB

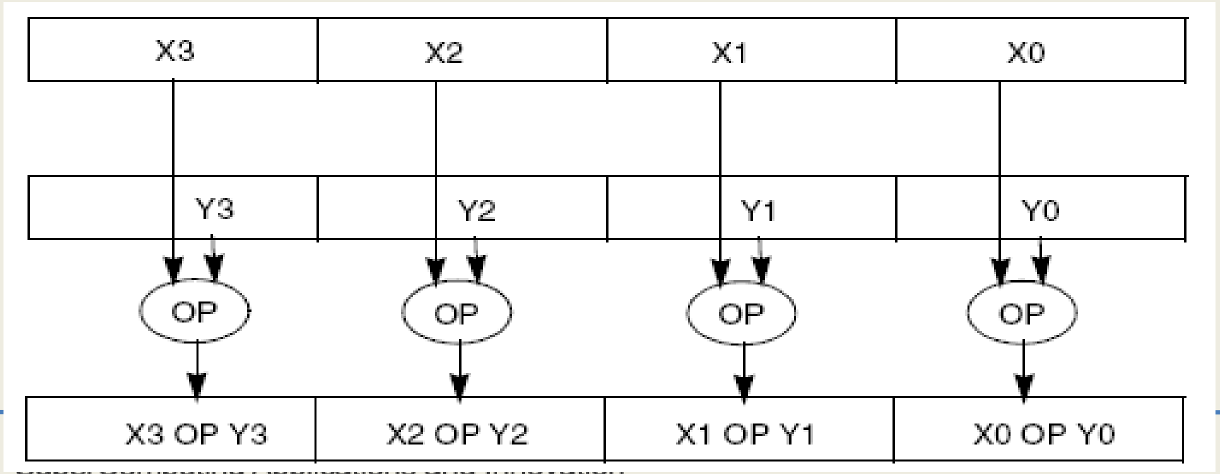
- Translation Look-aside Buffer
- Maps between logical space and actual memory addresses
- Memory is organized in ‘small pages’ of few KB in size
- Memory requests go through the TLB, normally very fast
- Pages that are not tracked through the TLB can be found in the ‘page table’: much slower
- Jumping between more pages than those contained in the TLB has a strong performance penalty
- Again “data locality” is a must!

Prefetch

- Hardware can be able to detect if you regularly load the same stream of data:
- “prefetch stream”
- Prefetching can be also suggested by software instructions

Vector registers

Vector registers are registers able to contain more than a single data. Then, the processing unit can perform a single instruction on multiple data (SIMD). They can be exploited only with a set of assembly instructions (SSE, AVX, AVX2..)



Vectorization

Usually, loops can benefit by SIMD operations, but only under certain conditions (no loop dependencies, numerable loops, etc.) that are checked by the compiler. If such conditions are satisfied, the compiler can generate SIMD assembly code (containing, for example, AVX instructions).

Vectorization is a factor that strongly contribute to enhance the peak performance of a processor.

Peak performance vs actual performance

The peak performance is given by:

- clock
- number of processing units
- memory bandwith
- vector registers

But only if all this features are exploited at the best. Actual algorithms can only exploit a part of those possibilities (for example for cache misses or CPU stalls). This is what gives the actual performance.

Data reuse

One of the factors affecting the performance is the data transfer rate:

High performance if data items are used multiple times

Example: vector addition x(i)=x(i)+y(i) needs 1 operation and 3 memory accesses

Example: inner product s = s + x(i)*y(i) needs 2 operations and 2 memory accesses (keeping s in the registers)

Data reuse

Matrix-matrix multiplication: $2N^3$ operations, $2N^2$ data

```
for (i=0; i<n; i++){
  for (j=0; j<n; j++){
    s=0;
    for (k=0; k<n; k++){
      s=s+a[i][k]*b[k][j];
    }
    c[i][j]=s;
  }
}
```

Can we improve this algorithm?

Data reuse

- Matrix-matrix multiplication: $2N^3$ operations, $2N^2$ data
- Since the number of of data is less than the number of operations, in principle, data should be reused, overcoming the problems due to bandwith/cpu speed
- But typically the naive implemenation is totally inefficient
- This is a good reason to use libraries when possible (MKL, Lapack, etc)

Reuse analysis

Matrix-vector multiplication

```
for (i=0; i<m; i++) {
  for (j=0; j<n; j++) {
    y[i]=y[i]+a[i][j]*x[j]
  }
}
```

y(i) behaves as a scalar, but the y array is loaded twice and fills a cache line!

Reuse analysis

Matrix-vector multiplication

```
for (i=0; i<m; i++) {
  for (j=0; j<n; j++) {
    y[i]=y[i]+a[i][j]*x[j]
  }
}
```

y(i) behaves as a scalar, but the y array is loaded twice and fills a cache line!

```
for (i=0; i<m; i++) {
  s=0;
  for (j=0; j<n; j++) {
    s = s+ a[i][j]*x[j];
  }
  y[i] = s;
}
```

now s is a scalar and can be kept in the registers

Reuse analysis

Matrix-vector multiplication

```
for (j=0; j<n; j++) {
  for (i=0; i<m; i++) {
    y[i]=y[i]+a[i][j]*x[j]
  }
}
```

x[j] is reused but there are a lot more of multiple load/store of y[i]

Reuse analysis

Matrix-vector multiplication

```
for (j=0; j<n; j++) {
  for (i=0; i<m; i++) {
    y[i]=y[i]+a[i][j]*x[j]
  }
}
```

x[j] is reused but there are a lot more of multiple load/store of y[i]

```
for (j=0; j<n; j++) {
  t=x[j]
  for (i=0; i<m; i++) {
    y[i]=y[i]+a[i][j]*t
  }
}
```

different behavior if matrix is stored by rows (row-major) or by columns (column-major): C and Fortran differ!

A note on arrays in Fortran and C..

Consider this matrix. How is it stored in memory? It depends..

$$A = \begin{pmatrix} a11 & a12 \\ a21 & a22 \end{pmatrix}$$

For FORTRAN it is *column-major order*

a11	a21	a12	a22

But C/C++ is *row-major order*

a11	a12	a21	a22

Reuse analysis

Matrix-vector multiplication

```
for (i=0; i<m; i+=2) {
   s1=0.; s2=0.;
   for (j=0; j<n; j++) {
      s1 = s1 + a[i][j]*x[j];
      s2 = s2 + a[i+1][j]*x[j];
   }
   y[i] = s1; y[i+1] = s2;
}
```

Loop tiling:

- x is loaded m/2 times rather than m
- Register usage for y as before
- Loop overhead half less
- Pipelined operations exposed
- Prefetch streaming

Data locality

High performance computing requires to take care of data locality:

- Temporal locality: group references to the same item close together
- Spatial locality: group references to nearby memory items close together

Temporal locality

Use an item, use it again before it is flushed away from registers or from the cache:

- Use item
- Use a small amount of other data
- Use item again

Temporal locality

```
for (loop=0; loop<10; loop++){
  for (i=0; i<N; i++){
    ... = .. x[i]...
  }
}
```

Long time between different uses of x...

Temporal locality

```
for (loop=0; loop<10; loop++){
  for (i=0; i<N; i++){
    ... = .. x[i]...
  }
}
```

Long time between different uses of x...

```
for (i=0; i<N; i++){
  for (loop=0; loop<10; loop++){
      ... = .. x[i]...
  }
}
```

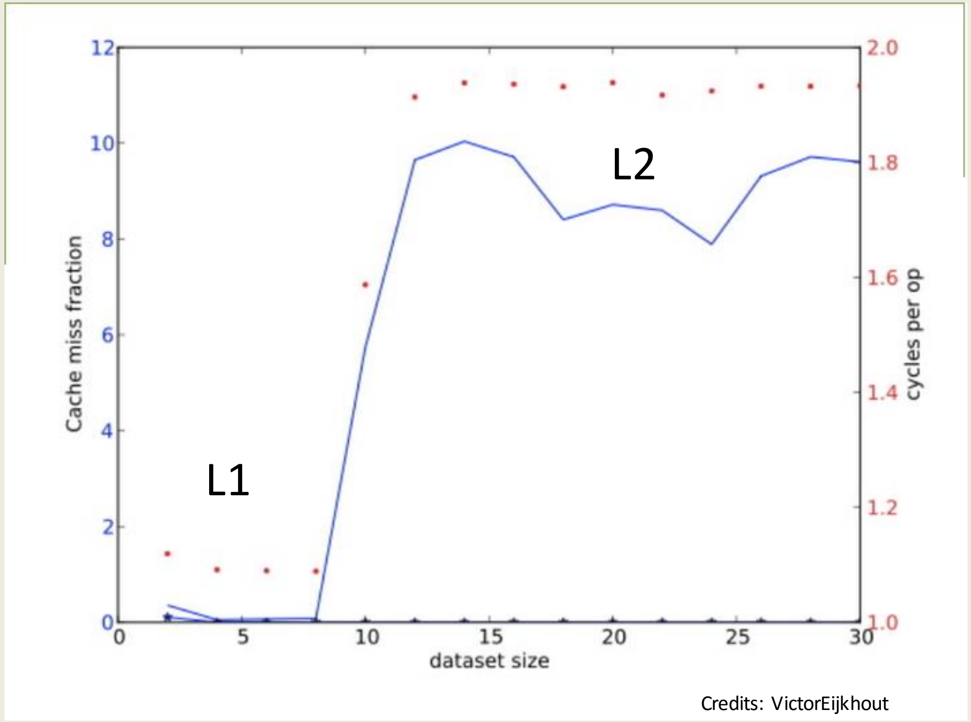
Loop interchange: here x is reused.

Spatial locality

- Use items close together
- Cache lines: try to use all the elements contained in a cache line before this is thrashed
- TLB: don’t make jumps larger than 512 words too many times

Cache size

```
for (i=0; i<NRUNS; i++){
  for (j=0; j<size; j++){
    array[j]=2.3*array[j]+1.2;
  }
}
```



Cache blocking

```
for (i=0; i<NRUNS; i++){
  blockstart = 0;
  for (b=0; b<size/l1size; b++){
    for (j=0; j<l1size; j++){
      array[blockstart+j]=2.3*array[blockstart+j]+1.2;
    }
  }
}
```

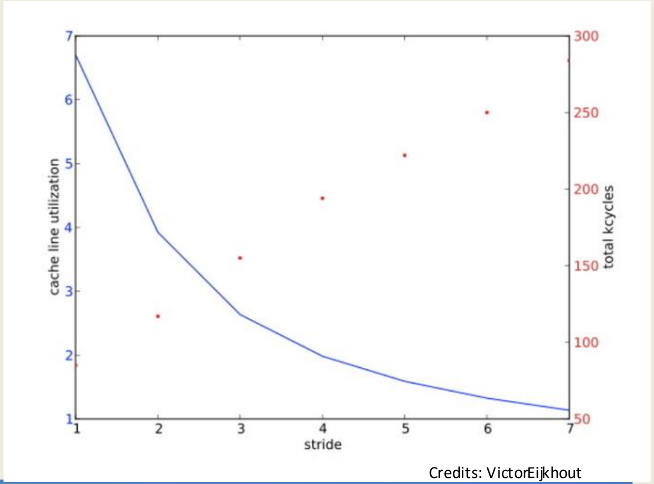
The cache-blocking techniques permit to arrange data in order to properly fit the cache size

Cache line utilization

```
for (i=0; n=0; i<L1WORDS; i++; n+=stride)
  array[n]= 2.3*array[n]+1.2;
```

The amount of data doesn't change but the stride access does.

Increasing the stride: more lines of cache are loaded, slower execution



Credits: VictorEijkhout

Fundamentals of computer architecture

Parallel architectures

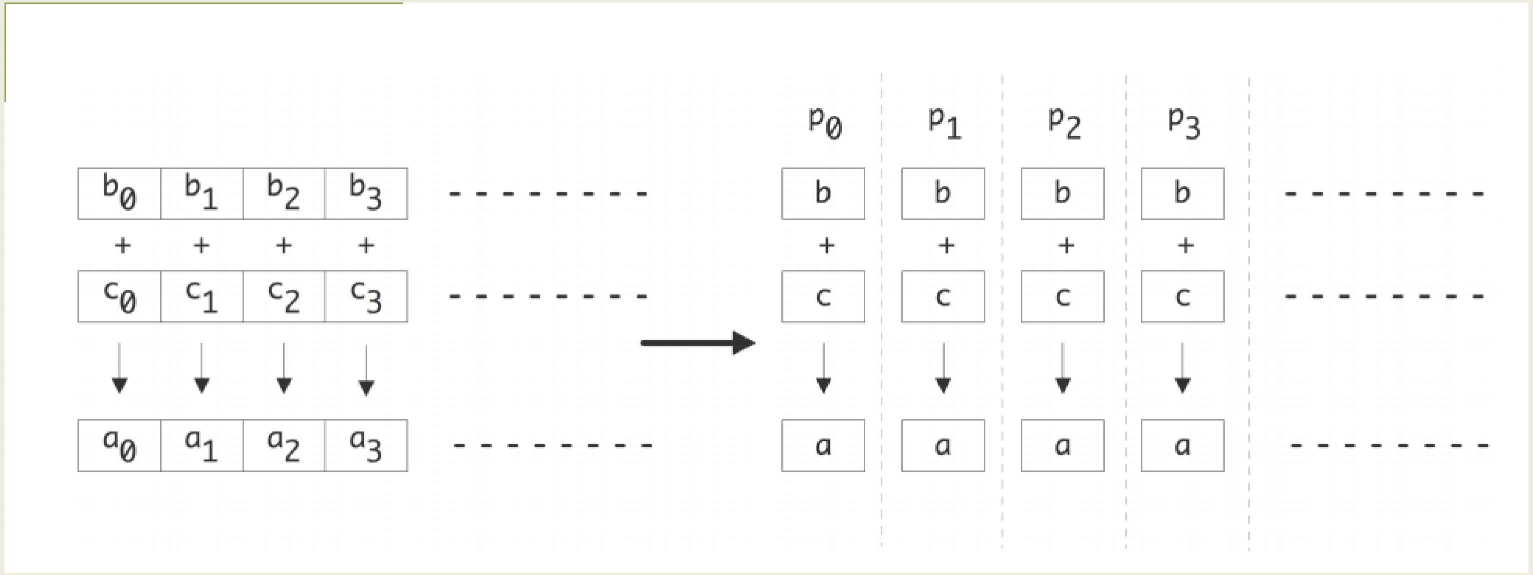
Basic concepts

Use multiple computing units in order to accelerate the computation or to extend domain of a problem.

```
for (i=0; i<N; i++)
  a[i]=b[i]+c[i]
```

Ideally, each processing unit can work on one (or more) array element

Basic concepts

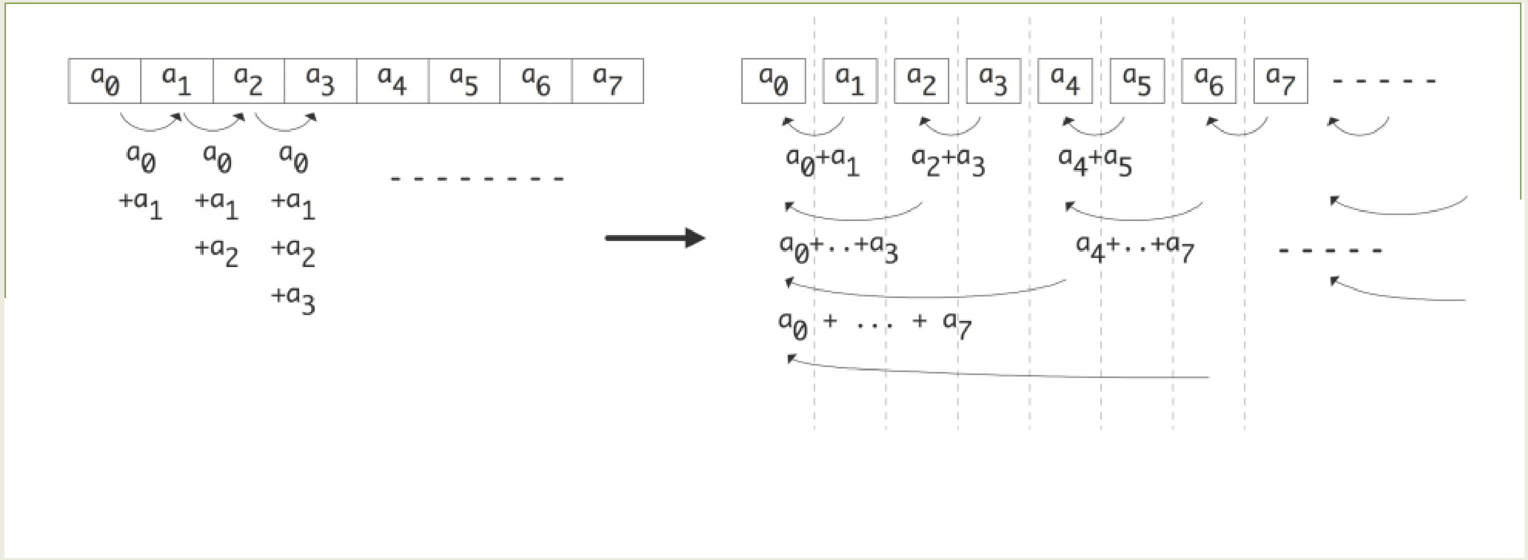


Basic concepts

- Spread operations over many processors
- If n operations take time t on 1 processor,
- Does it always become t/p on p processors?

```
s = sum ( x[i], i=0,n-1 )
```

Basic concepts



a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7
a_0 $+a_1$
a_0 $+a_1$ $+a_2$
a_0 $+a_1$ $+a_2$ $+a_3$
a_0+a_1 a_2+a_3 a_4+a_5
$a_0+..+a_3$ $a_4+..+a_7$
$a_0 + ... + a_7$

Basic concepts

- Spread operations over many processors
- If n operations take time t on 1 processor,
- Does it always become t/p on p processors?

```
s = sum ( x[i], i=0,n-1 )
```

N operations can be done with N/2 processors, in total time $\log_2 N$

Can we do faster?

Some theory…

- Best case: P processes give $T_p=T_1/P$
- Speedup is defined as $S_P=T_1/T_p < P$
- Superlinear speedup is not possible, in theory, but sometimes happens in practice
- Perfect speedup can be reached in “embarassingly parallel applications” where processes are independent
- Less than optimal can be due to overhead, sequential parts, interdependencies...
- Efficiency is defined as $E_p=S_p/P$

Amdahl’s Law

Some parts of the code can not be parallel, so they ultimately become a bottleneck

Amdahl’s law states that the serial part of a code pose a constraint on the maximum parallel efficiency of the code.

For instance, if 5% of your application is serial, you cannot get a speedup over 20. No matter how many processors you use.

Flavours of parallelism -Flynn taxonomy

Flynn‘s(1966) taxonomy is a first way to classify parallel computers into one of four types:

– (SISD) Single instruction, single data

- Your desktop (if it is at least 5 years old, maybe…)

– (SIMD) Single instruction, multiple data:

- all the machines based on vector registers
- GPUs

– (MISD) Multiple instruction, single data

- never covered

– (MIMD) Multiple instruction, multiple data

- Nearly all of today’s parallel machines

Memory models

Top500 list is dominated by clusters and MPPs: so the most successful model was the MIMD

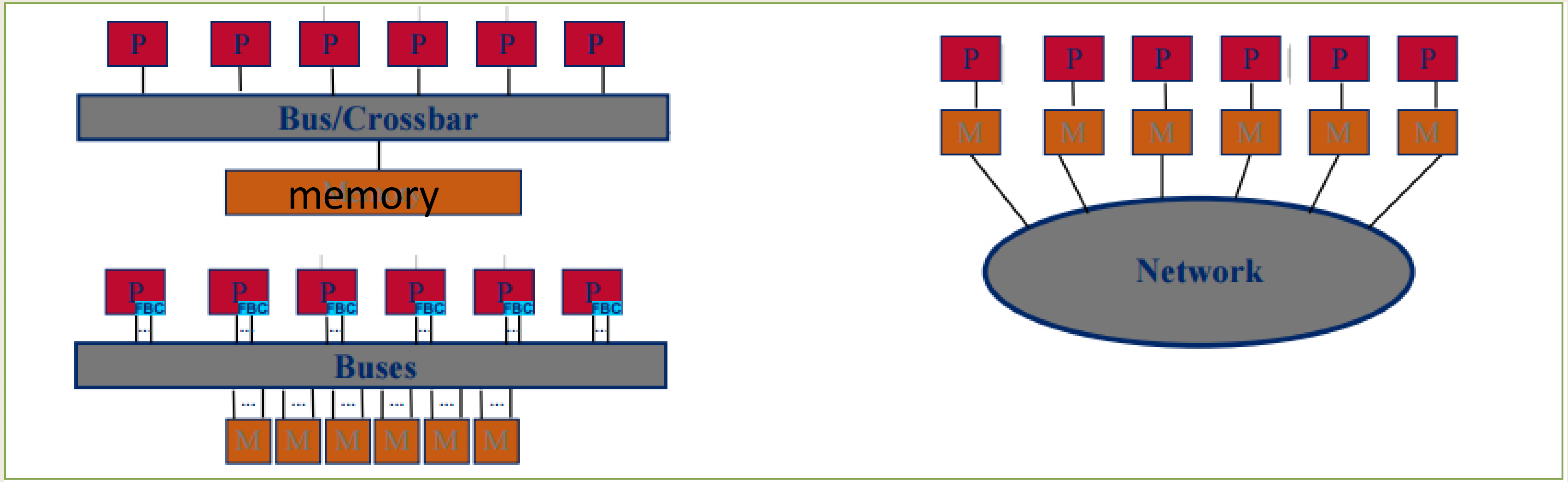
A much more useful way to classify is according to memory models:

- shared memory
- distributed memory

Memory models

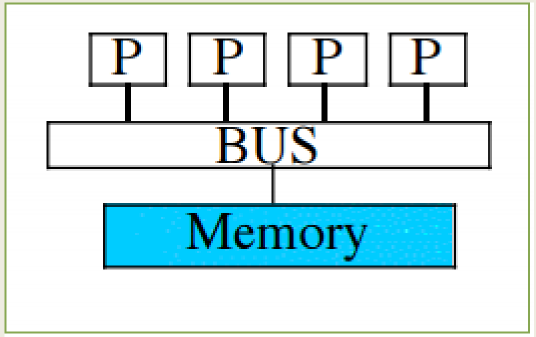
- Shared memory: all the processors share the same memory address space
  - OpenMP: directive based programming
  - PGAS languages (Co-array Fortran,UPC, Titanium, X10...)
- Distributed memory: every processor has a private memory address space
  - MPI: Message Passing Interface

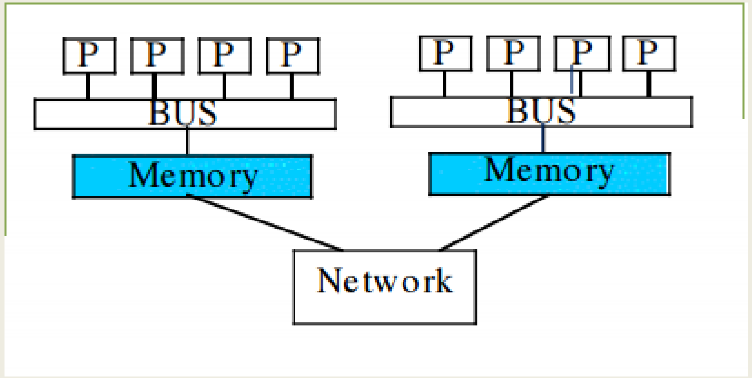
Memory models



Shared memory: UMA and NUMA

- UMA: Uniform Memory Access. Each processor has uniform access time to the memory (also known as SMP)
- NUMA: Non-Uniform Memory Access. Time for memory access depends on location of data.



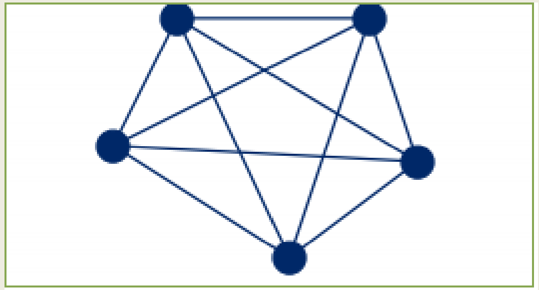


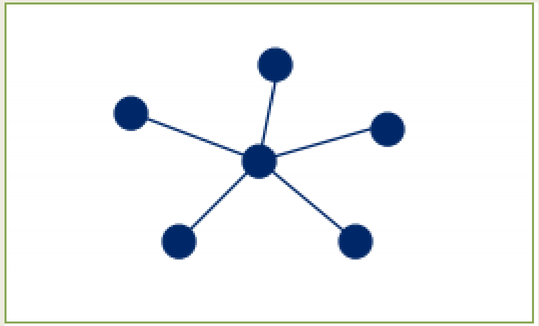
Interconnects

What is the actual shape of an interconnection network? Mesh? Ring? Something more elaborate?

Topologies

- Completely connected topology: each processor has direct access to every other processor
- Star connected topology: middle processor is the central processor. Every other processor is connected to it.

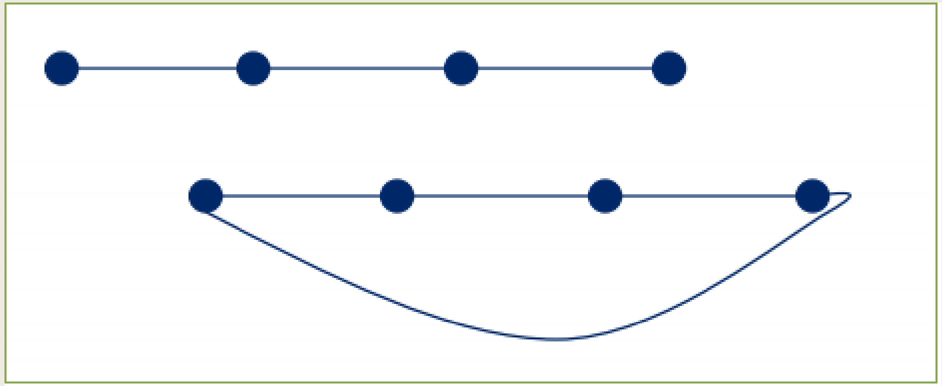




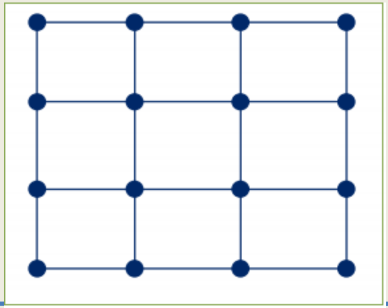
Topologies

Linear array

Ring

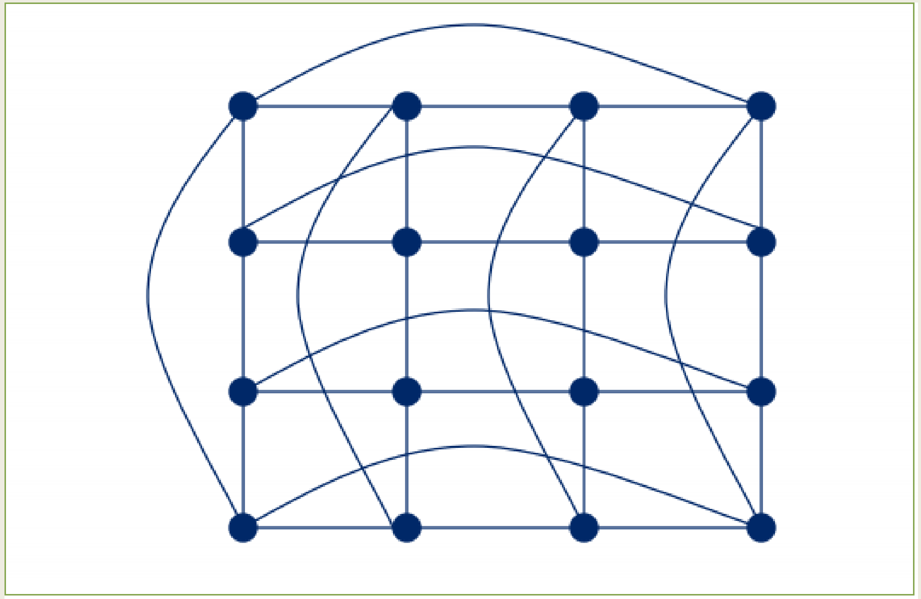


Mesh (2D array)



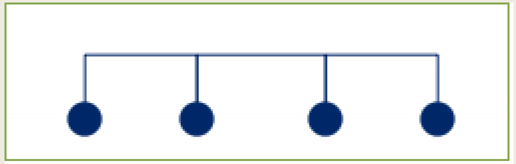
Topologies

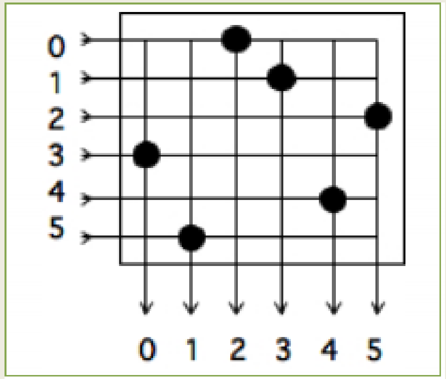
Torus (2D ring..)



Hubs and crossbars

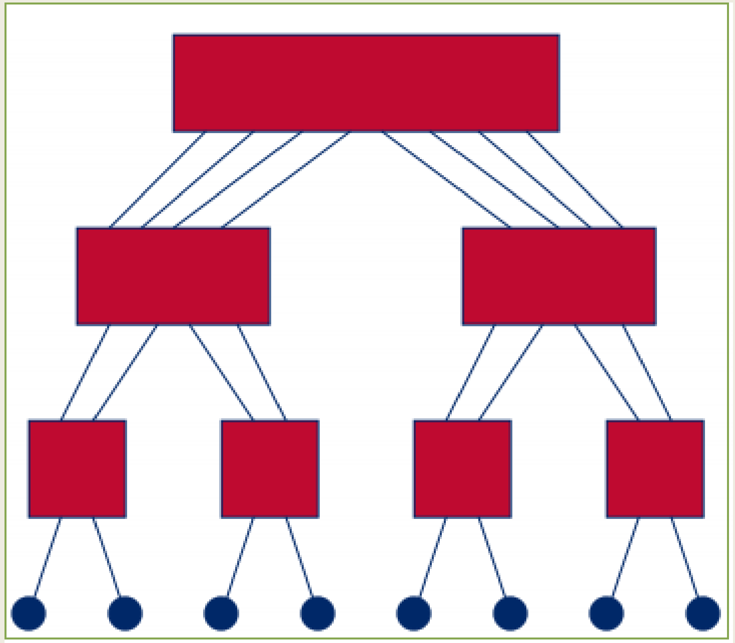
- Hubs/buses: every processor shares the communication link
- Crossbar switches: every processor connects to the switch which routes the communications to their destinations





Fat trees

- Multiple switches
- Each level has the same number of links in as out
- Increasing the number of links at each level
- Gives full bandwith between the links
- Added latency the higher you go



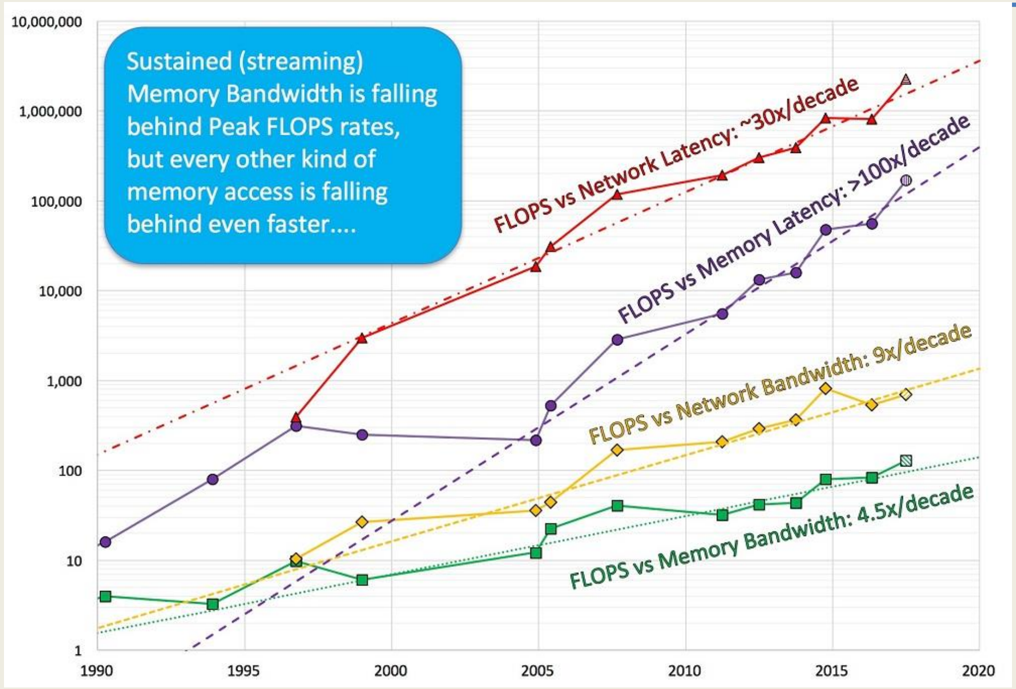
Practical issues

- Latency: how long does it take to start sending a “message”? (ms or us)
- Bandwidth: what data rate can be sustained once the message is started? (MB/s or GB/s)
  - Both point-to-point and aggregate bandwith are quantities of interest
- Multiple wires, multiple latencies, same bw

Summary

- Every computer is a parallel computer now
- Good serial computing skills are central prerequisite to good parallel computing
- Clusters and MPPs are at large similar to desktops, laptops, etc.
- Focus on:
  - (multiple?) Processing units
  - Memory hierarchies (registers, caches, main memory)
  - Internal interconnect
- Good hardware is wasted without good programming practice

Final word



Trends in the relative performance of floating-point arithmetic and several classes of data access for select HPC servers over the past 25 years. Source: John McCalpin

→ Pointless having fast processors if access to memory or network is slow