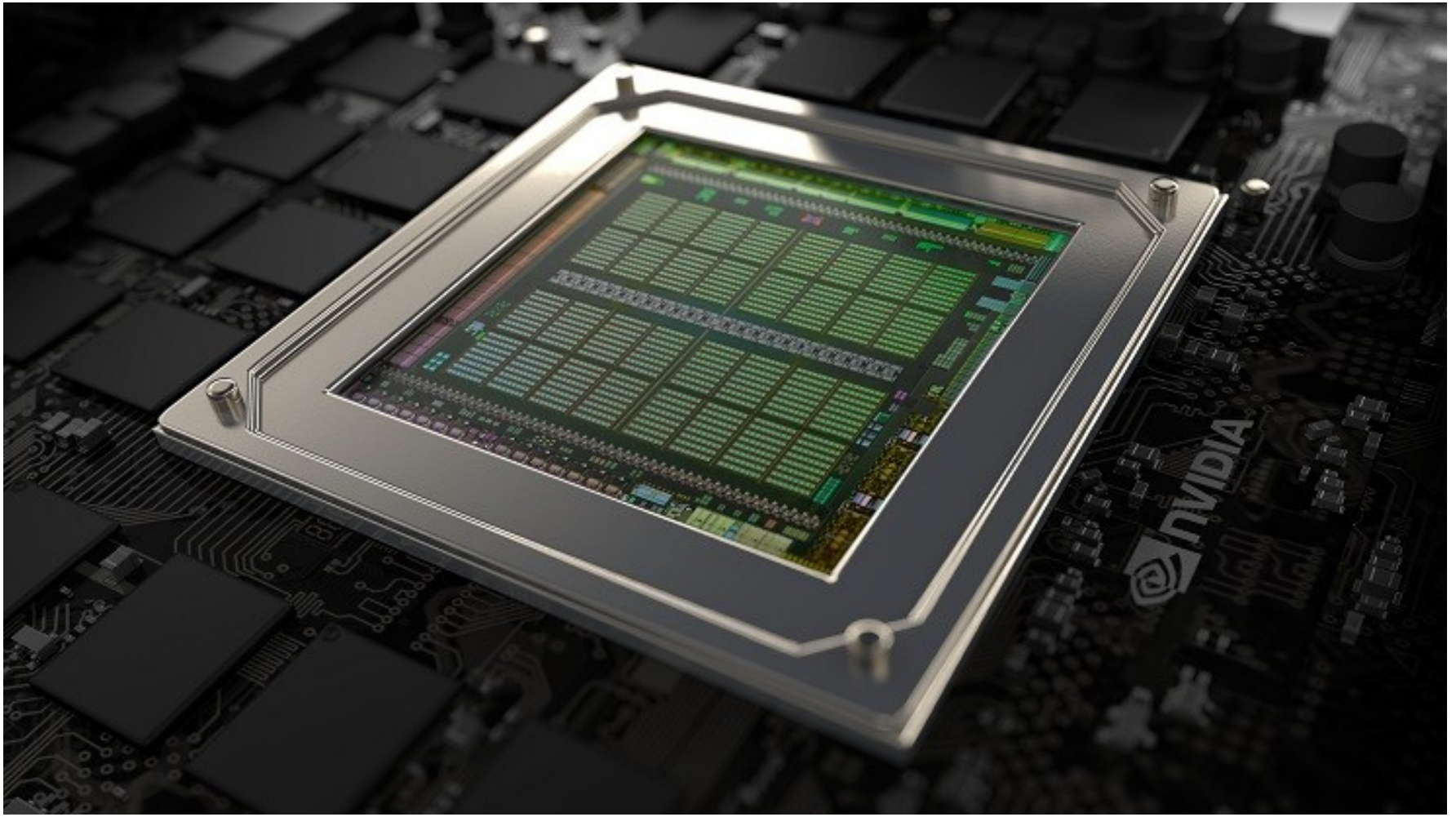
Parallel programming: Introduction to GPU architecture

Sylvain Collange
Inria Rennes – Bretagne Atlantique
sylvain.collange@inria.fr

GPU internals

- What makes a GPU tick?



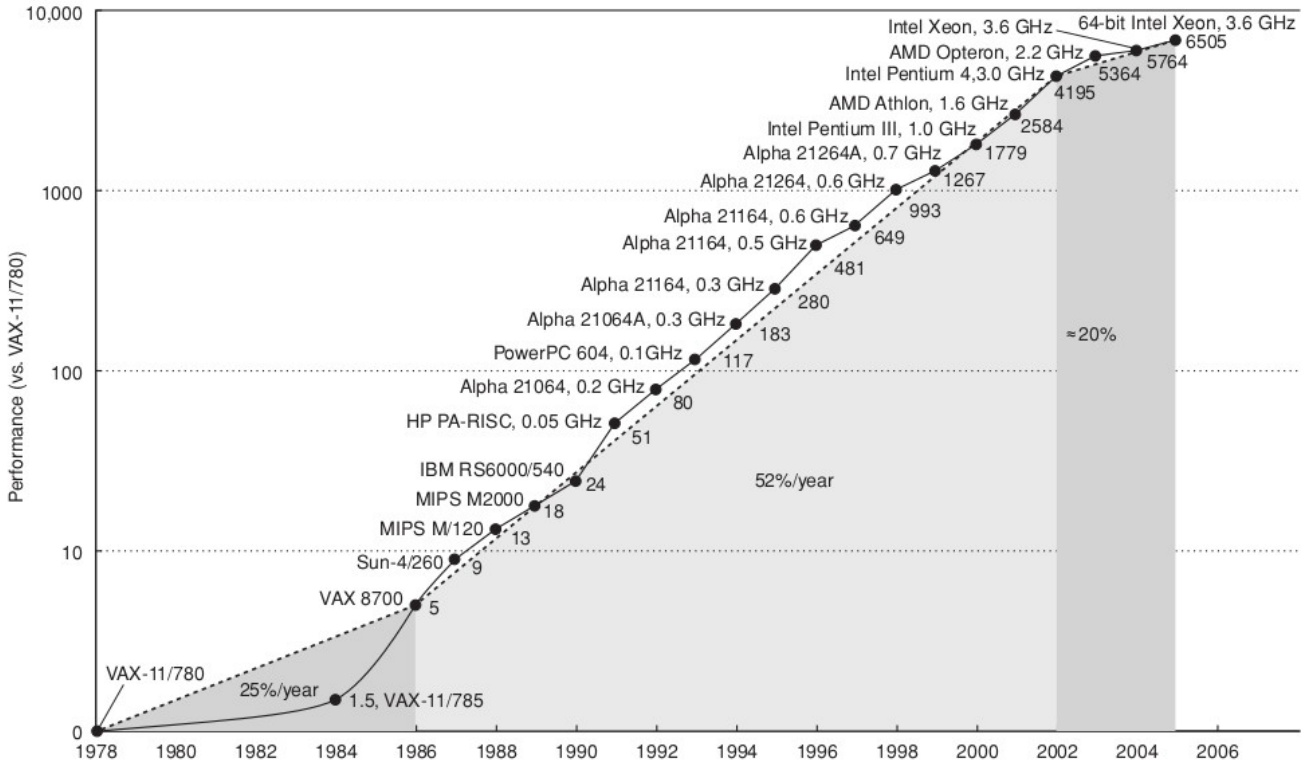
NVIDIA GeForce GTX 980 Maxwell GPU. Artist rendering!

Outline

- Computer architecture crash course
  - The simplest processor
  - Exploiting instruction-level parallelism
- GPU, many-core: why, what for?
  - Technological trends and constraints
  - From graphics to general purpose
- Forms of parallelism, how to exploit them
  - Why we need (so much) parallelism: latency and throughput
  - Sources of parallelism: ILP, TLP, DLP
  - Uses of parallelism: horizontal, vertical
- Let's design a GPU!
  - Ingredients: Sequential core, Multi-core, Multi-threaded core, SIMD
  - Putting it all together
  - Architecture of current GPUs: cores, memory

The free lunch era... was yesterday

- 1980's to 2002: *Moore's law*, *Dennard scaling*, micro-architecture improvements
  - Exponential performance increase
  - Software compatibility preserved



Hennessy, Patterson. Computer Architecture, a quantitative approach. 4th Ed. 2006

- Do not rewrite software, buy a new machine!

Computer architecture crash course

- How does a processor work?
  - Or was working in the 1980s to 1990s:
    modern processors are much more complicated!
  - An attempt to sum up 30 years of research in 15 minutes

Machine language: instruction set

- Registers
  - State for computations
  - Keeps variables and temporaries
- Instructions
  - Perform computations on registers,
    move data between register and memory, branch…
- Instruction word
  - Binary representation of an instruction
- Assembly language
  - Readable form of machine language
- Examples
  - Which instruction set does your laptop/desktop run?
  - Your cell phone?

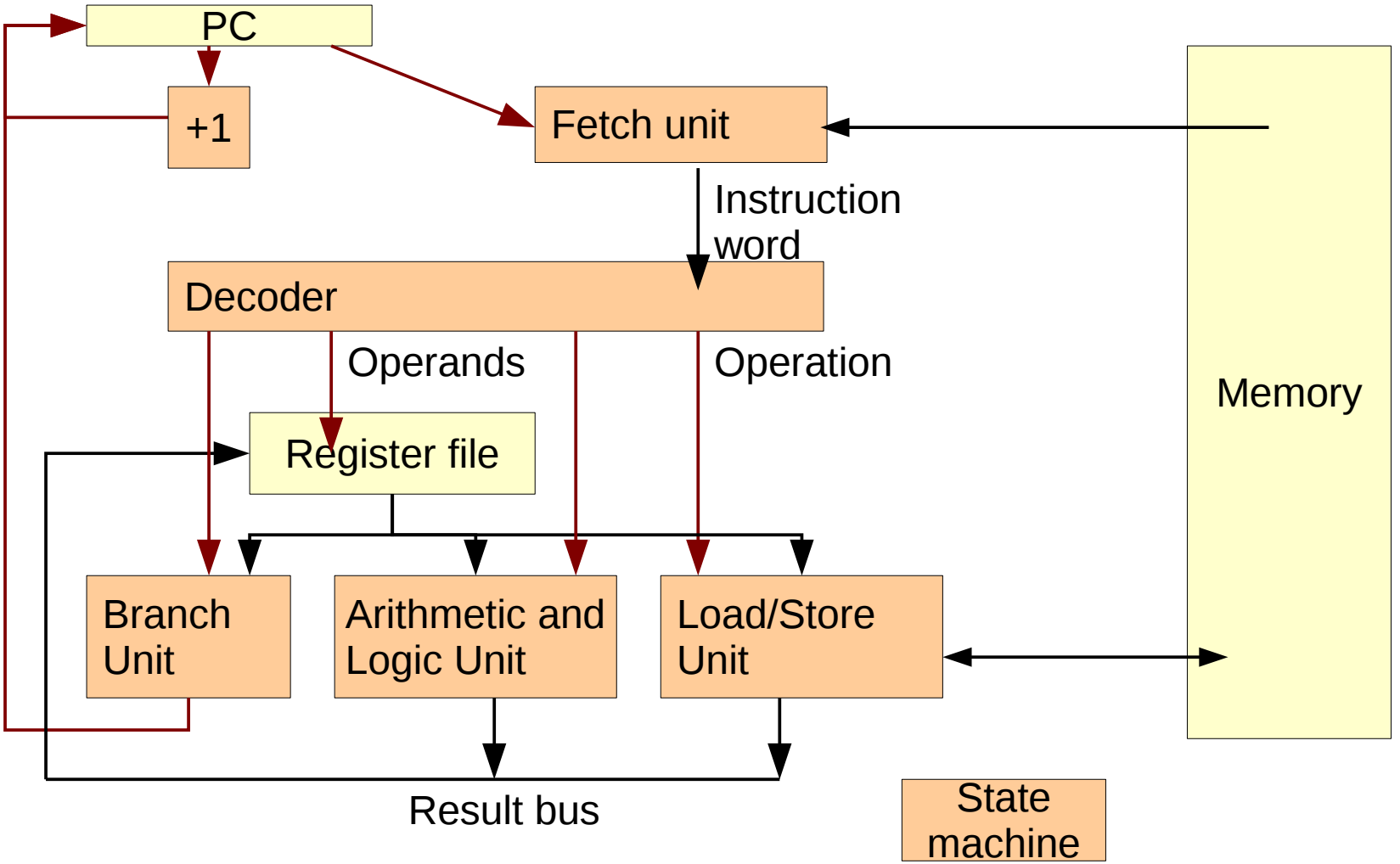
Example

R0, R1, R2, R3... R31

01100111

ADD R1, R3

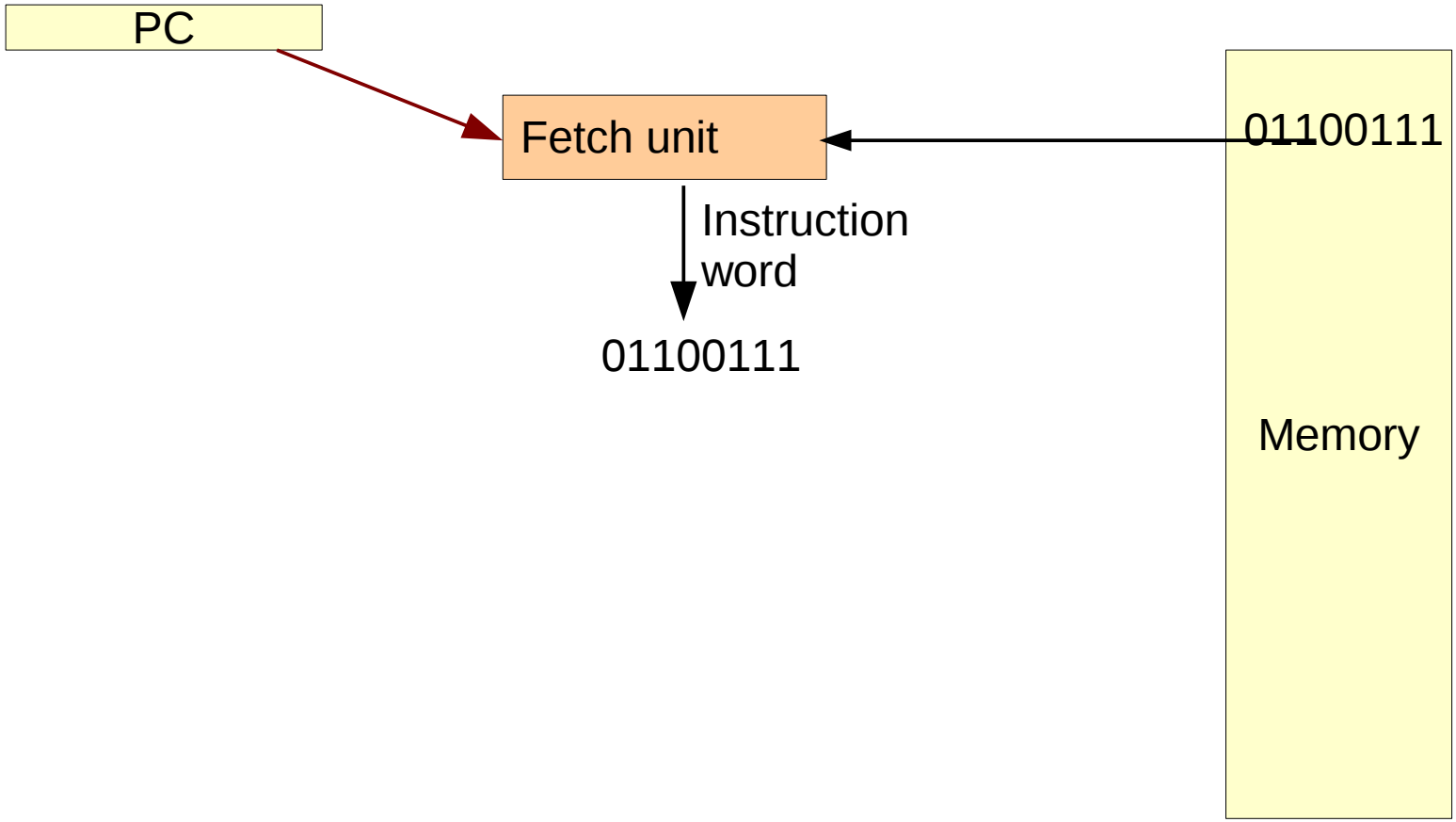
The Von Neumann processor



- Let's look at it step by step

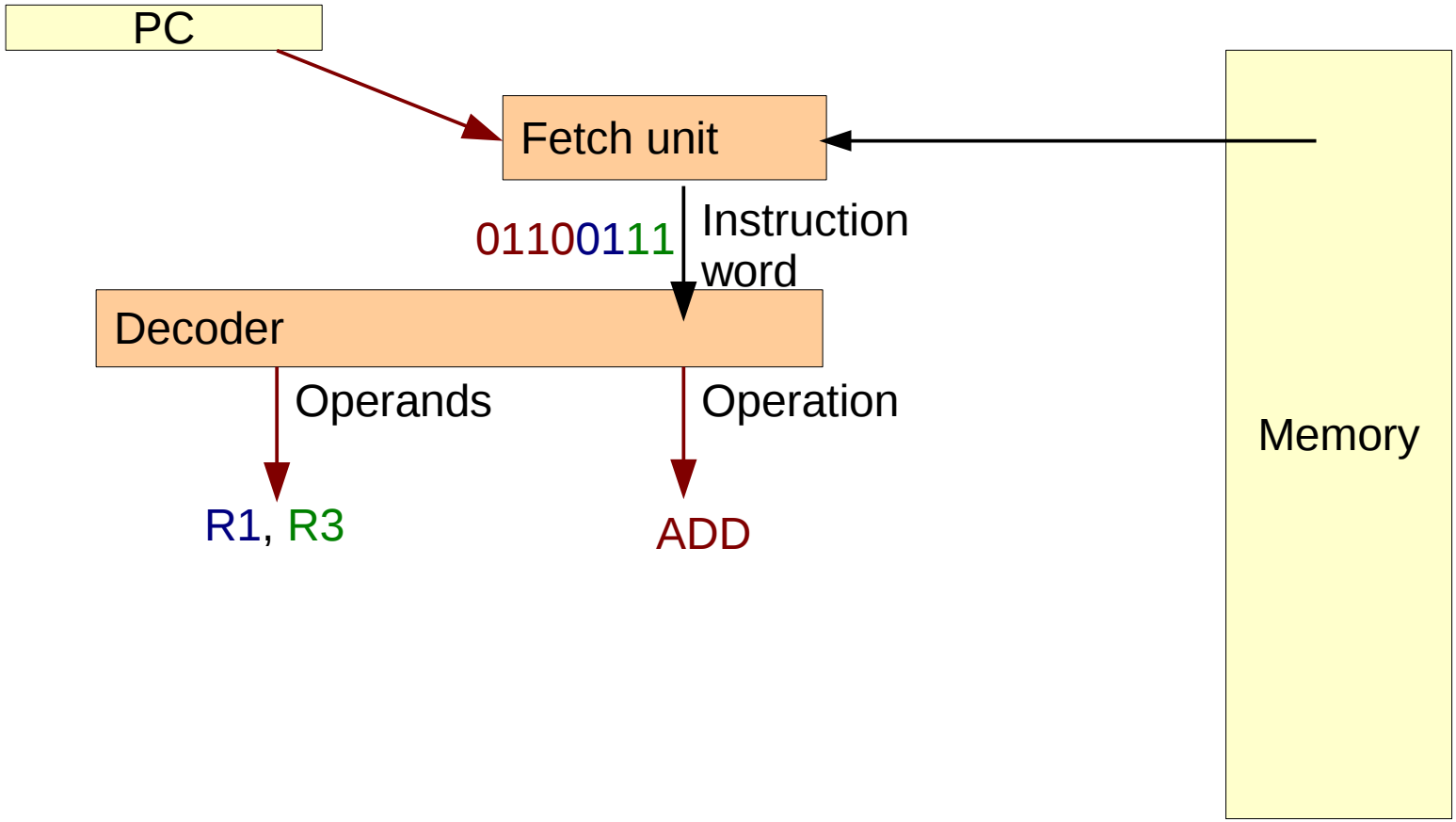
Step by step: Fetch

- The processor maintains a Program Counter (PC)
- Fetch: read the instruction word pointed by PC in memory



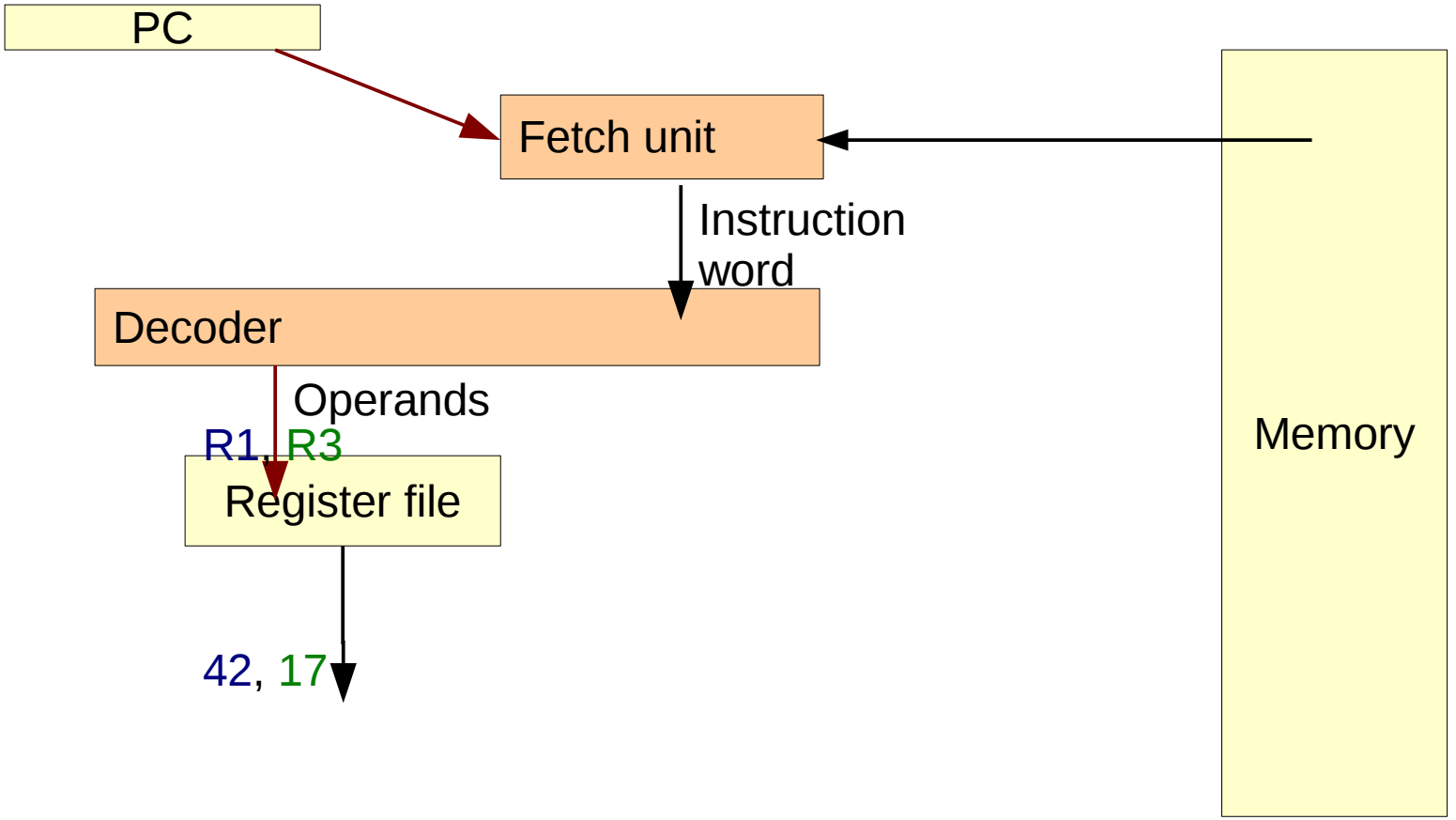
Decode

- Split the instruction word to understand what it represents
  - Which operation? → ADD
  - Which operands? → R1, R3



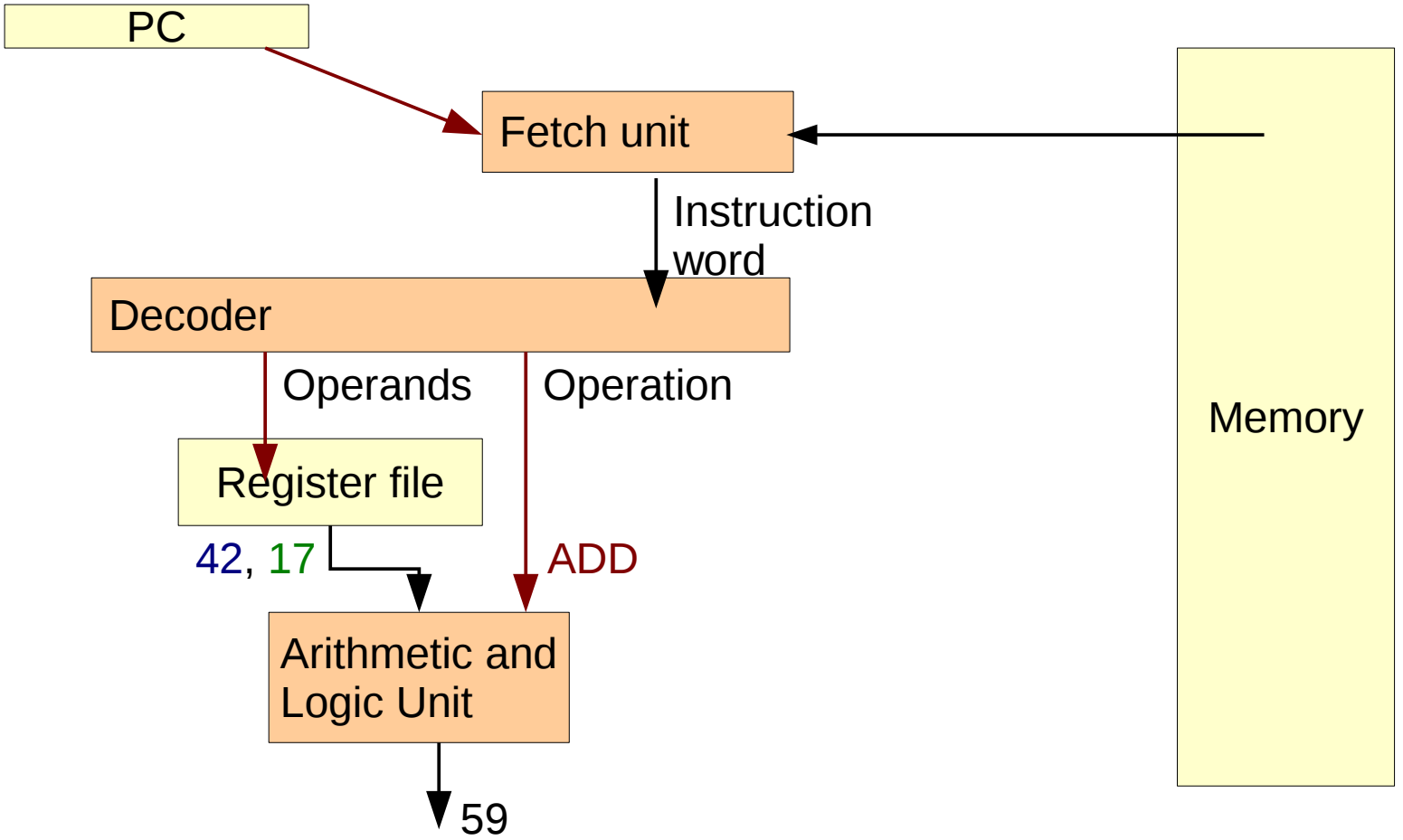
Read operands

- Get the value of registers R1, R3 from the register file



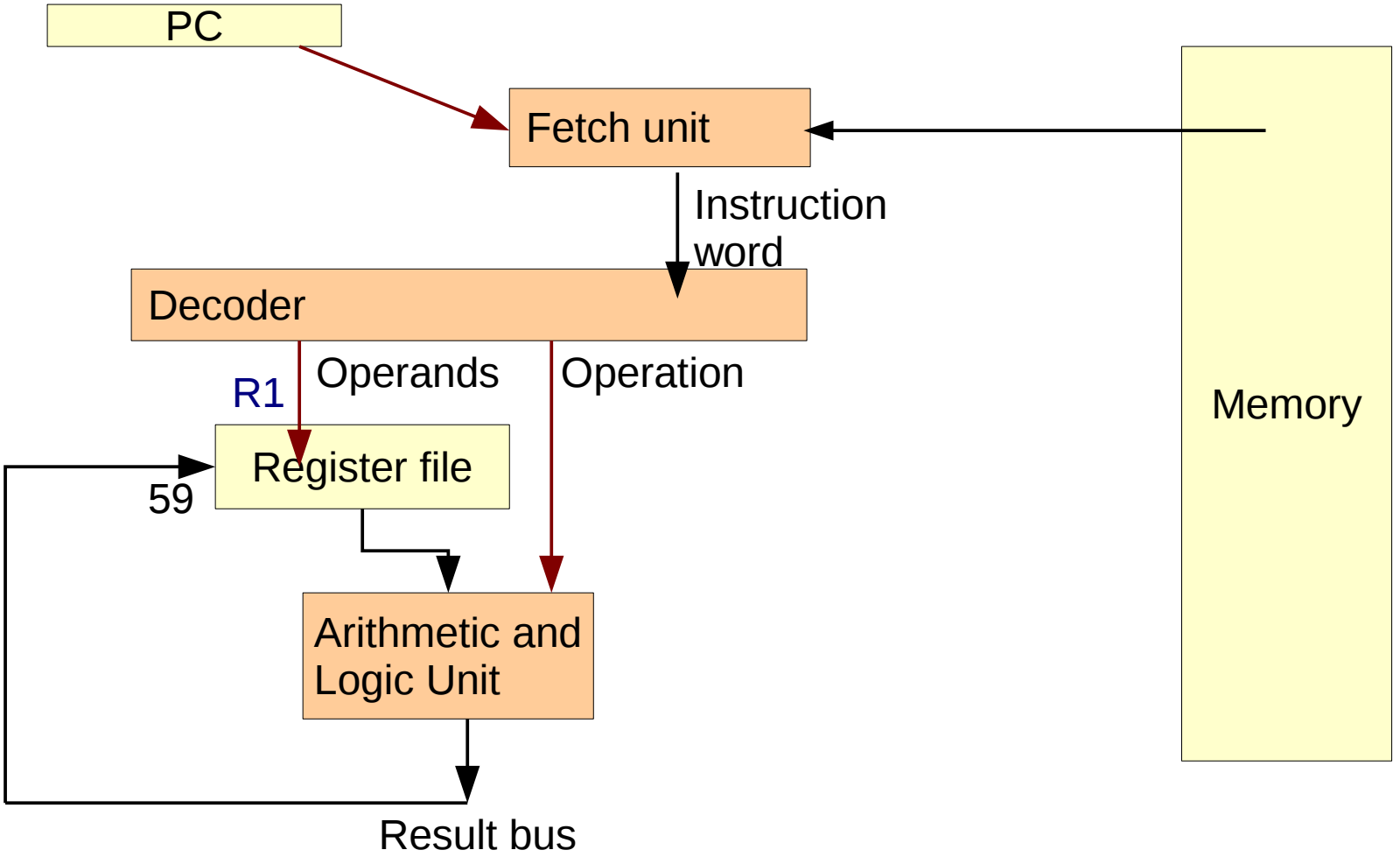
Execute operation

- Compute the result: 42 + 17

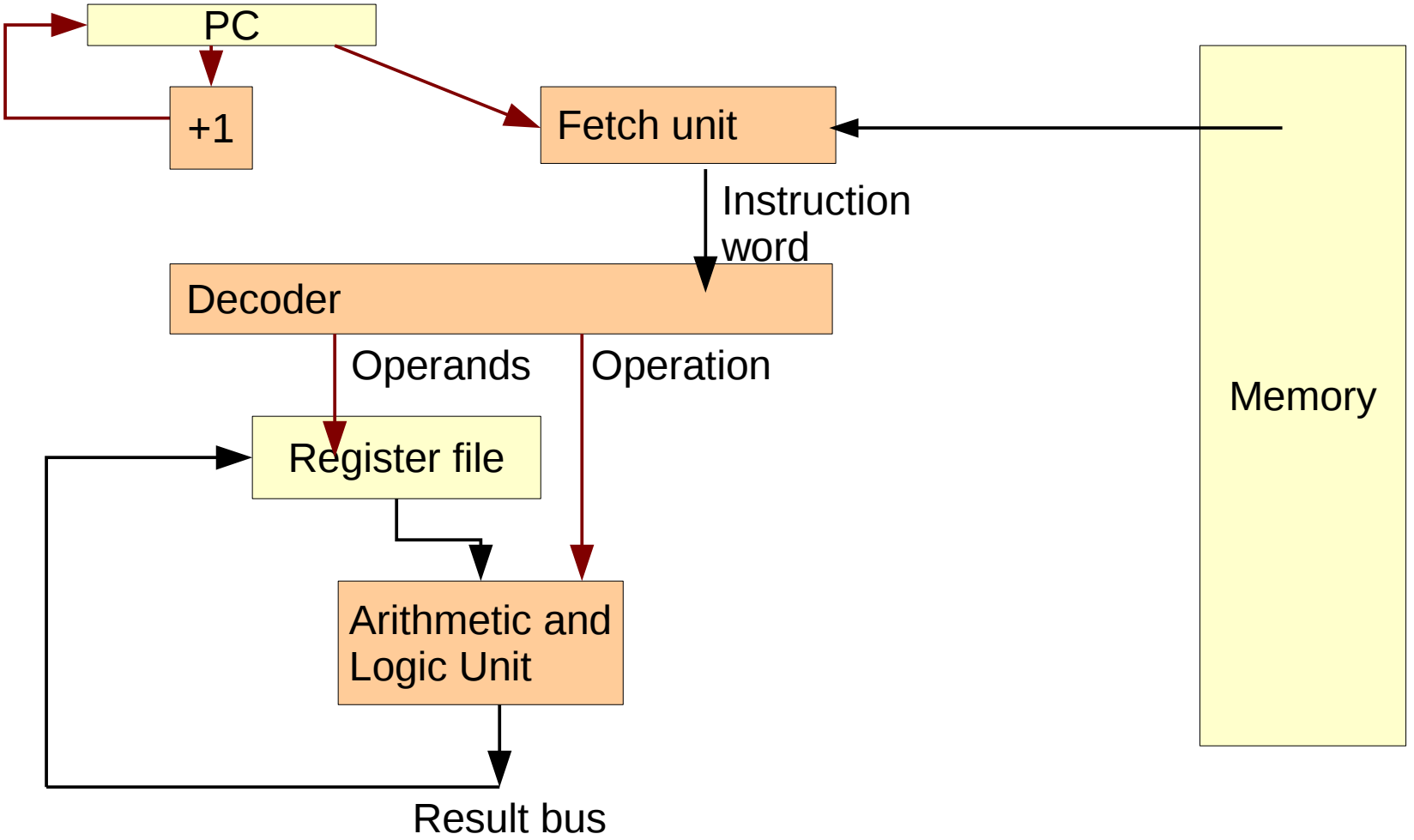


Write back

- Write the result back to the register file

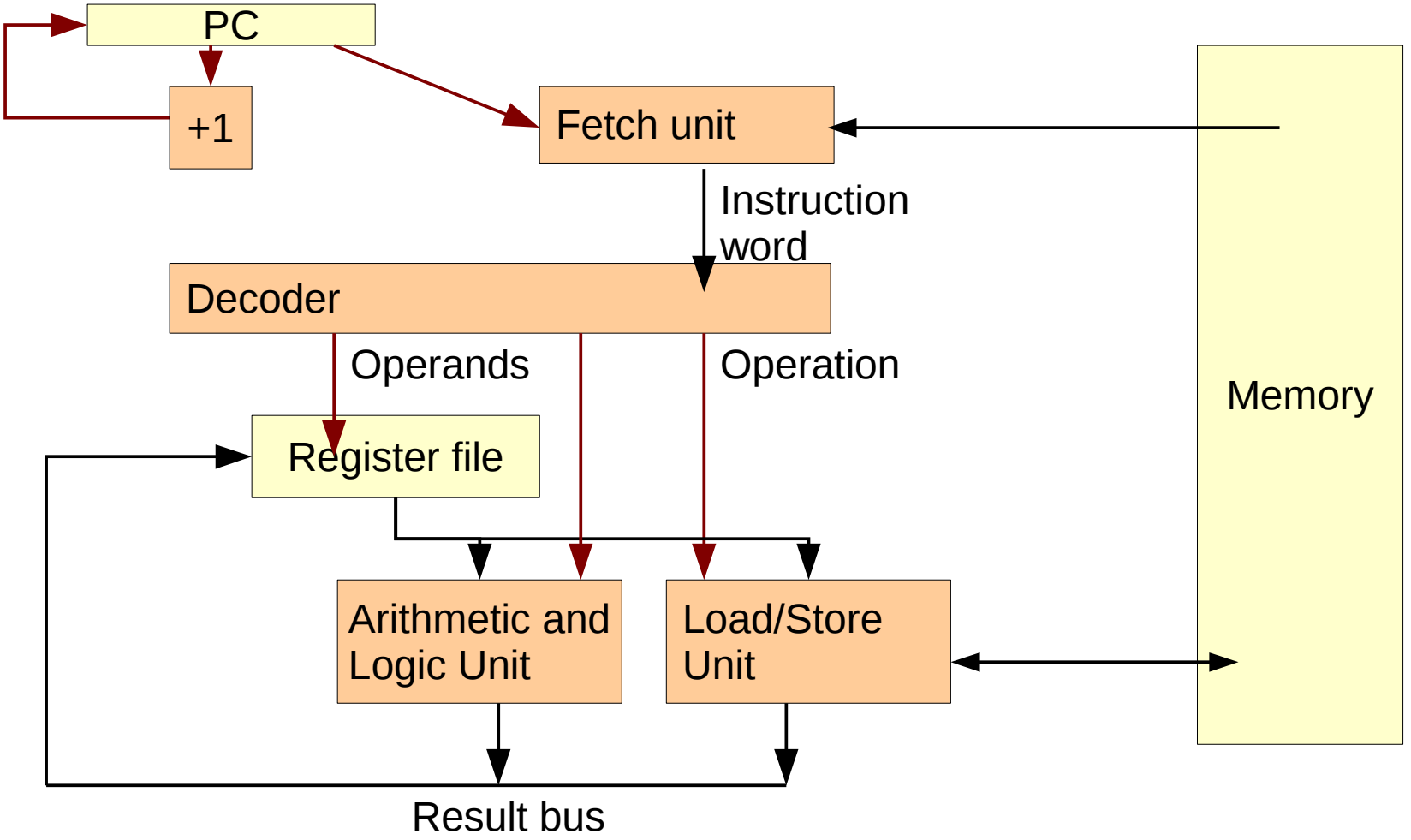


Increment PC



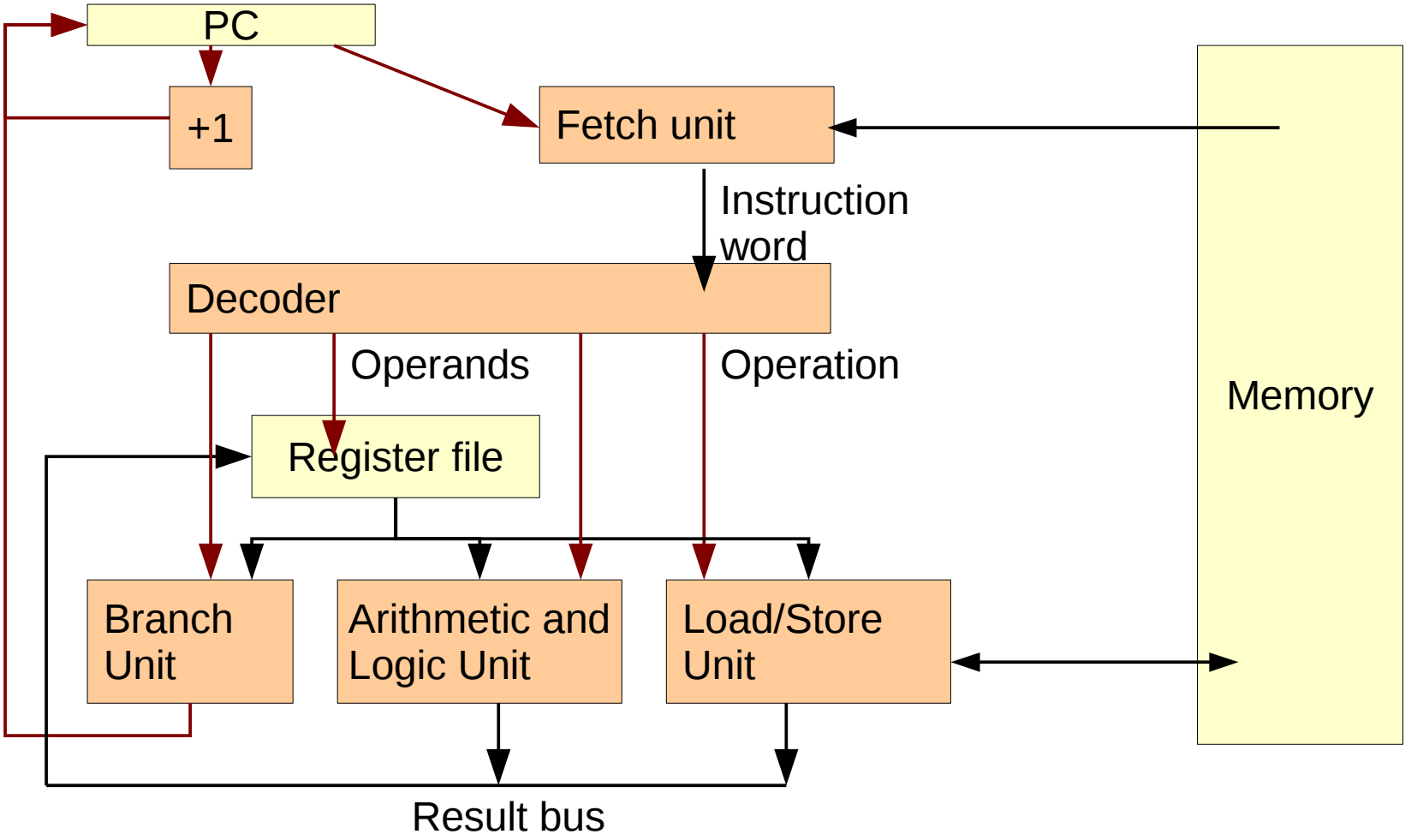
Load or store instruction

- Can read and write memory from a computed address



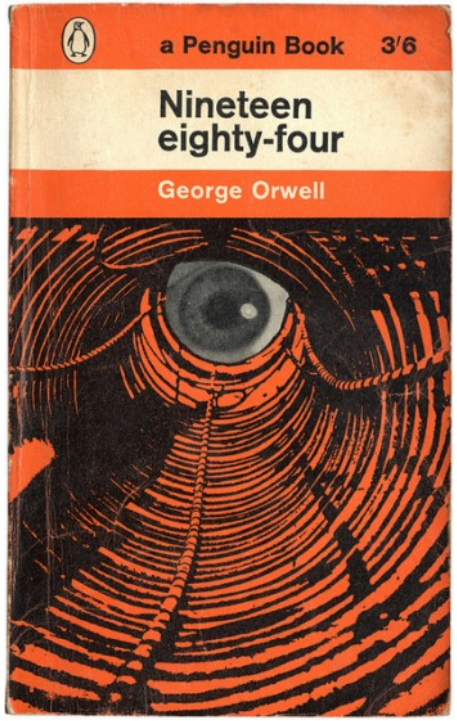
Branch instruction

- Instead of incrementing PC, set it to a computed value

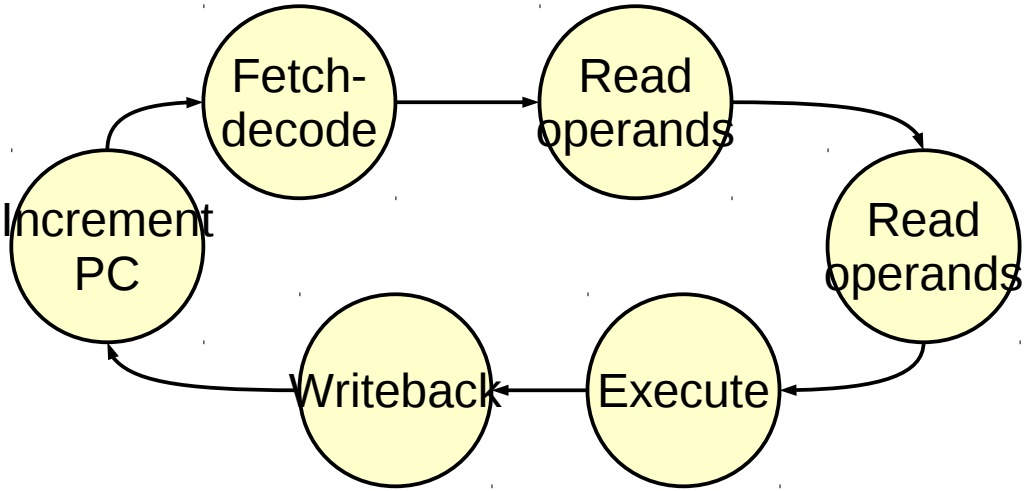


What about the state machine?

- The state machine controls everybody

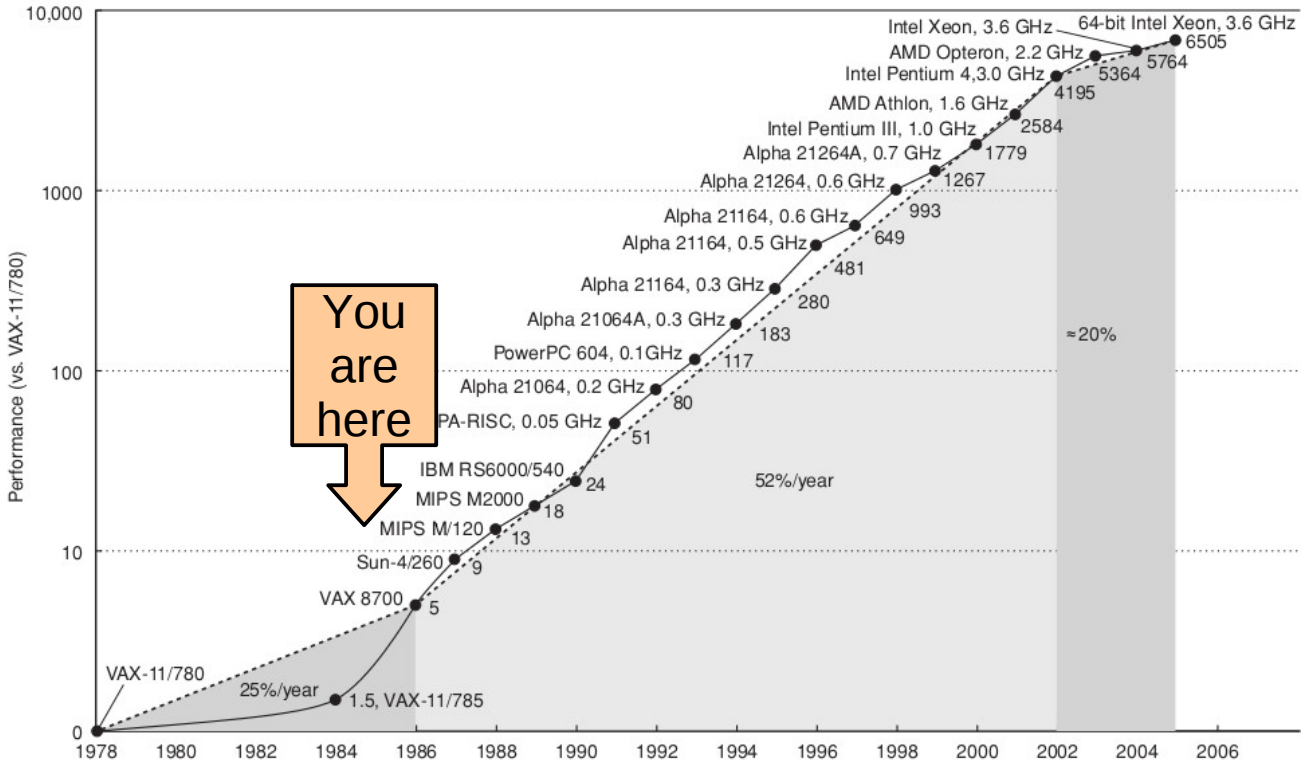


- Sequences the successive steps
  - Send signals to units depending on current state
  - At every clock tick, switch to next state
  - Clock is a periodic signal (as fast as possible)



Recap

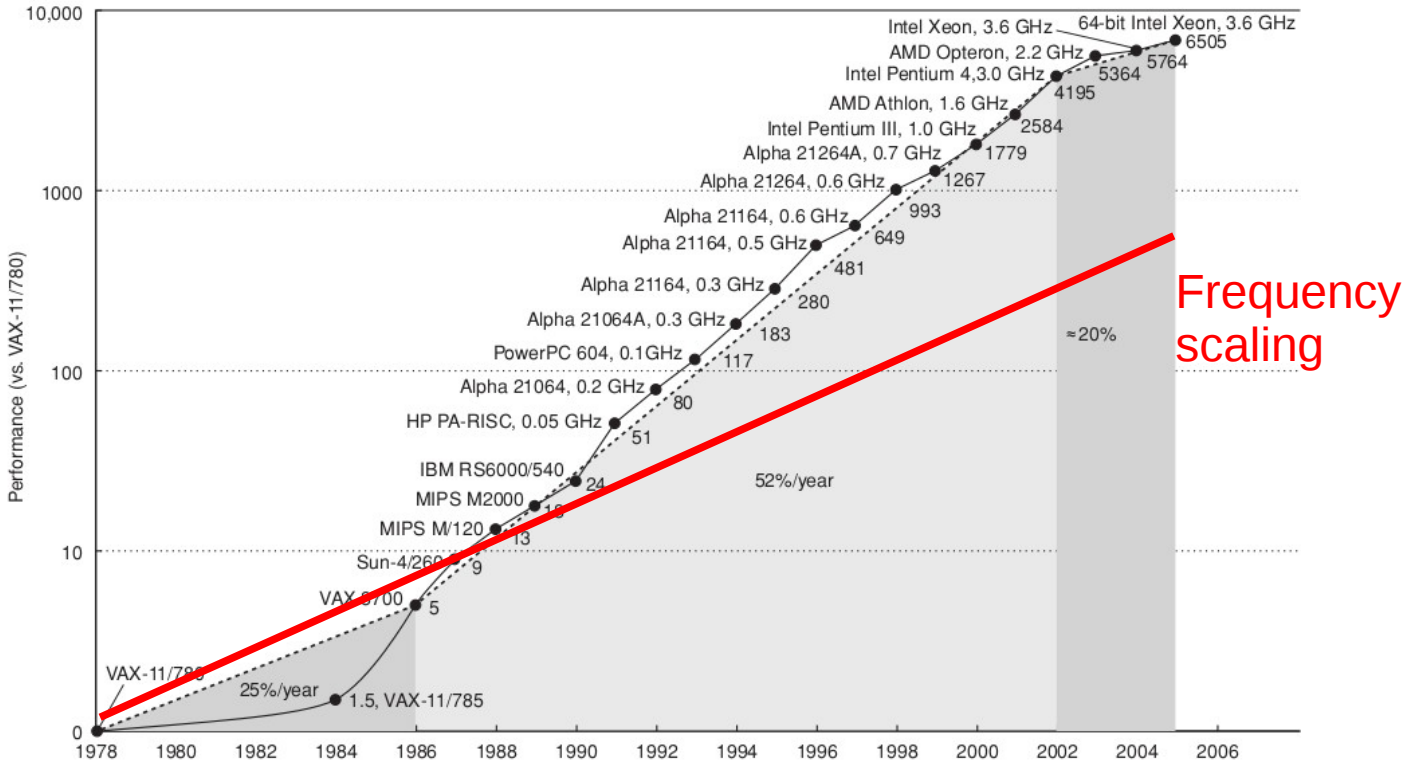
- We can build a real processor
  - As it was in the early 1980's



- How did processors become faster?

Reason 1: faster clock

- Progress in semiconductor technology allows higher frequencies



- But this is not enough!

Outline

- Computer architecture crash course
  - The simplest processor
  - Exploiting instruction-level parallelism
- GPU, many-core: why, what for?
  - Technological trends and constraints
  - From graphics to general purpose
- Forms of parallelism, how to exploit them
  - Why we need (so much) parallelism: latency and throughput
  - Sources of parallelism: ILP, TLP, DLP
  - Uses of parallelism: horizontal, vertical
- Let's design a GPU!
  - Ingredients: Sequential core, Multi-core, Multi-threaded core, SIMD
  - Putting it all together
  - Architecture of current GPUs: cores, memory

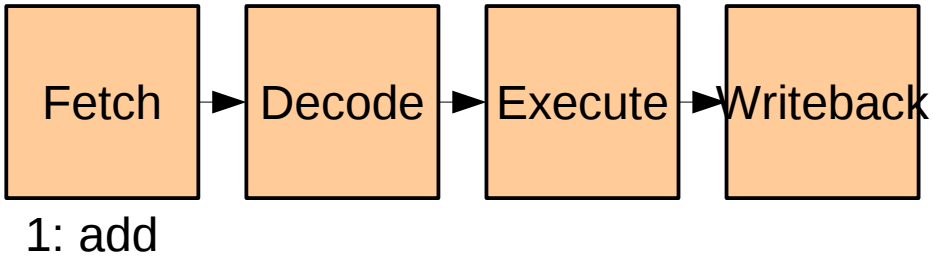
Going faster using ILP: pipeline

- Idea: we do not have to wait until instruction *n* has finished to start instruction *n*+1
- Like a factory assembly line
  - Or the *bandeijão*

Pipelined processor

Program

```
1: add, r1, r3
2: mul r2, r3
3: load r3, [r1]
```

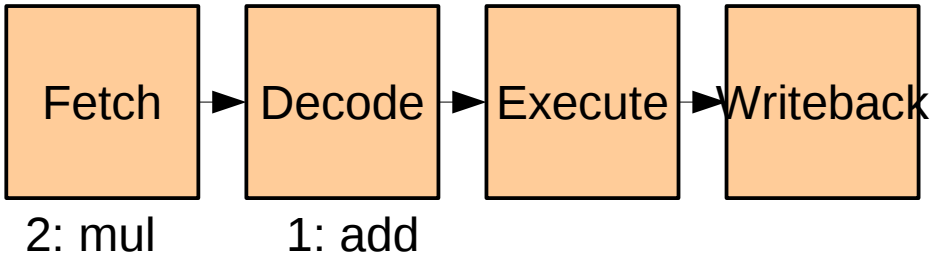


- Independent instructions can follow each other
- Exploits ILP to hide instruction latency

Pipelined processor

Program

```
1: add, r1, r3
2: mul r2, r3
3: load r3, [r1]
```

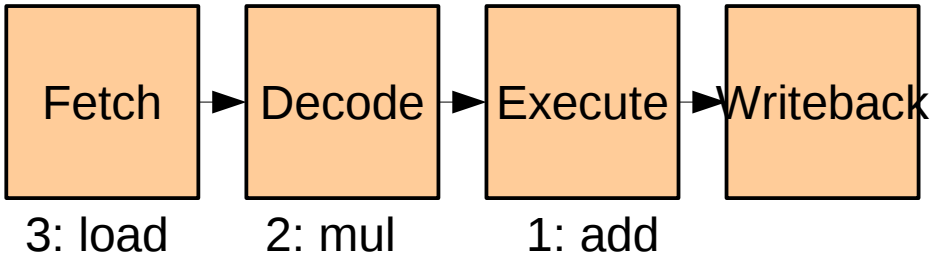


- Independent instructions can follow each other
- Exploits ILP to hide instruction latency

Pipelined processor

Program

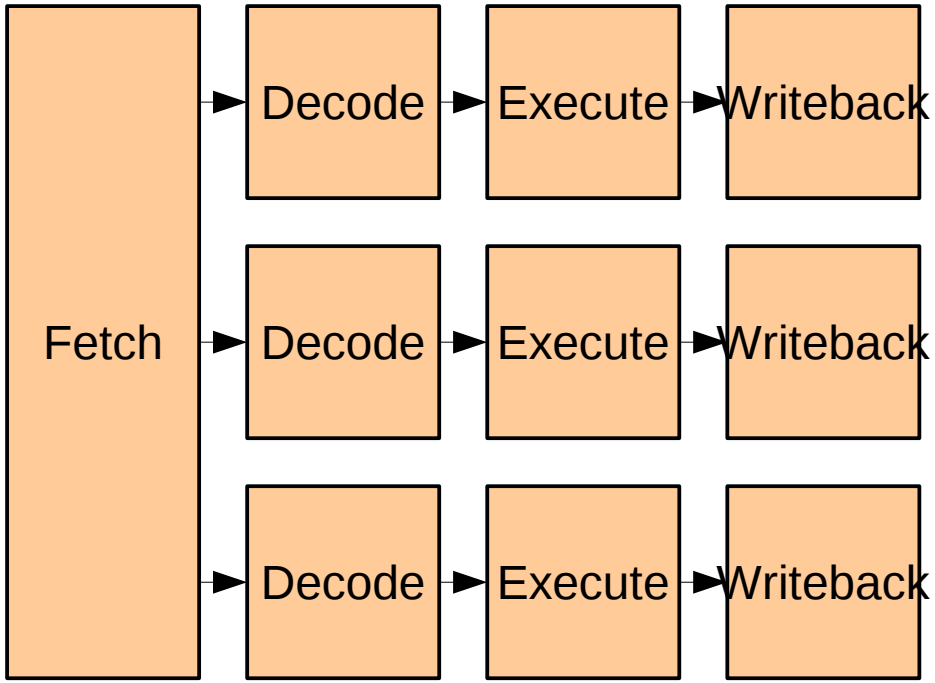
```
1: add, r1, r3
2: mul r2, r3
3: load r3, [r1]
```



- Independent instructions can follow each other
- Exploits ILP to hide instruction latency

Superscalar execution

- Multiple execution units in parallel
  - Independent instructions can execute at the same time



- Exploits ILP to increase throughput

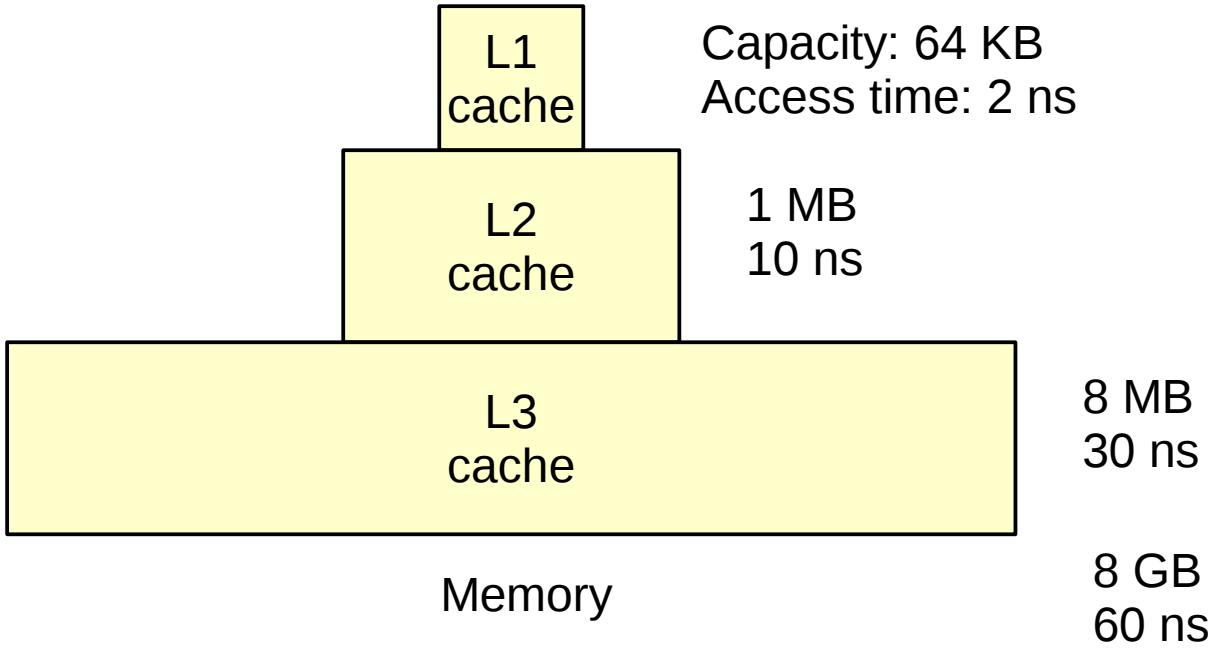
Locality

- Time to access main memory: ~200 clock cycles
- One memory access every few instructions
- Are we doomed?

- Fortunately: principle of locality
  - ~90% of memory accesses on ~10% of data
  - Accessed locations are often the same
- Temporal locality
  Access the same location at different times
- Spacial locality
  Access locations close to each other

Caches

- Large memories are slower than small memories
  - The computer theorists lied to you:
    in the real world, access in an array of size *n* costs O(log *n*), not O(1)!
  - Think about looking up a book in a small or huge library
- Idea: put frequently-accessed data in small, fast memory
  - Can be applied recursively: hierarchy with multiple levels of cache



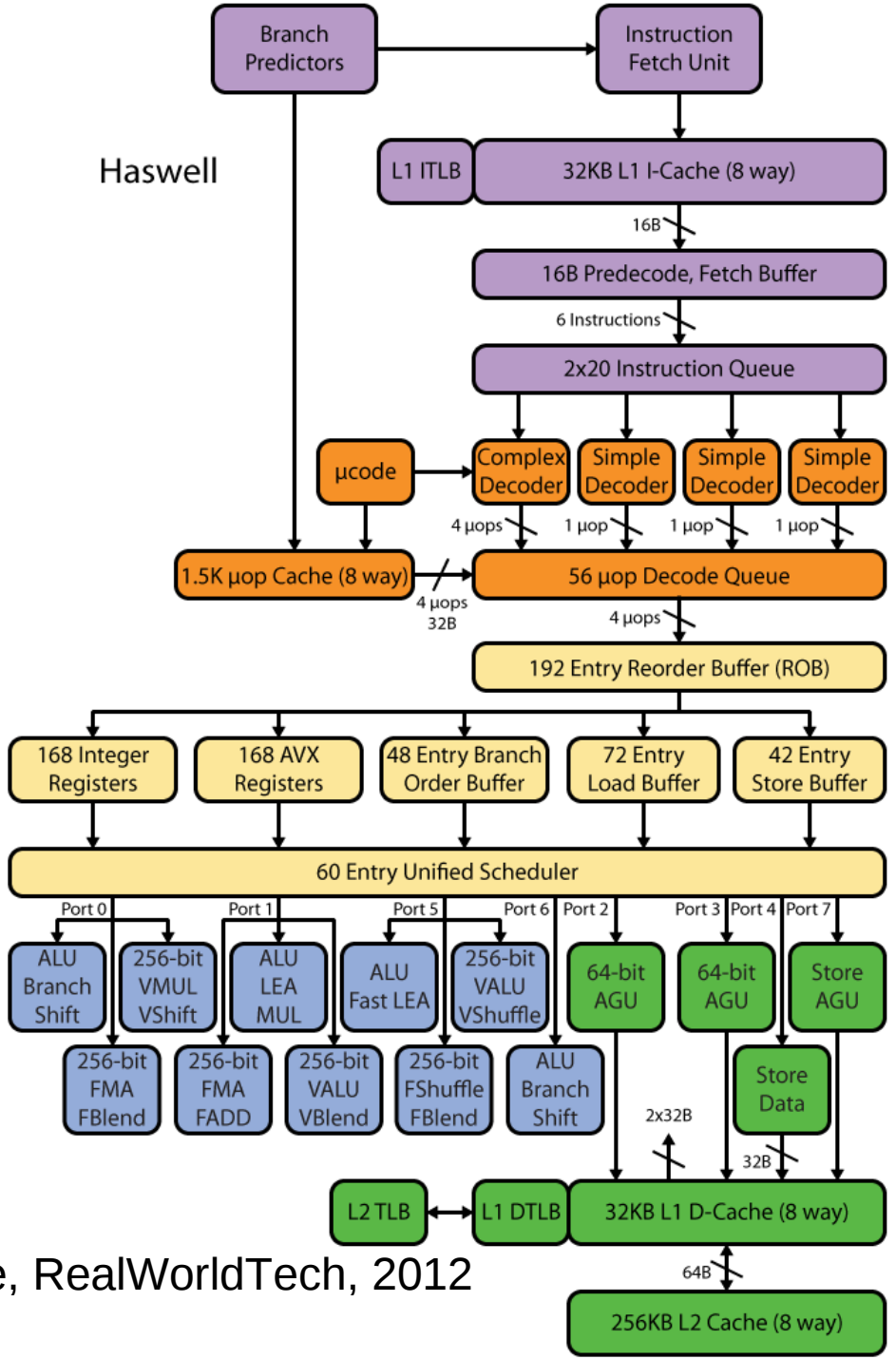
Branch prediction

- What if we have a branch?
  - We do not know the next PC to fetch from until the branch executes
- Solution 1: wait until the branch is resolved
  - Problem: programs have 1 branch every 5 instructions on average
  - We would spend most of our time waiting
- Solution 2: predict (guess) the most likely direction
  - If correct, we have bought some time
  - If wrong, just go back and start over
- Modern CPUs can correctly predict over 95% of branches
  - World record holder: 1.691 mispredictions / 1000 instructions
- General concept: *speculation*

P. Michaud and A. Seznec. "Pushing the branch predictability limits with the multi-poTAGE+ SC predictor." JWAC-4: Championship Branch Prediction (2014).

Example CPU: Intel Core i7 Haswell

- Up to 192 instructions in flight
  - May be 48 predicted branches ahead
- Up to 8 instructions/cycle executed out of order
- About 25 pipeline stages at ~4 GHz
  - Quizz: how far does light travel during the 0.25 ns of a clock cycle?
- Too complex to explain in 1 slide, or even 1 lecture



David Kanter, Intel's Haswell CPU architecture, RealWorldTech, 2012
http://www.realworldtech.com/haswell-cpu/

Recap

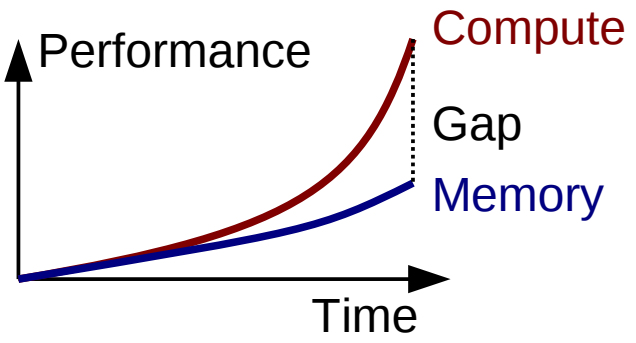
- Many techniques to run sequential programs as fast as possible
  - Discovers and exploits parallelism between instructions
  - Speculates to remove dependencies
- Works on existing binary programs, without rewriting or re-compiling
  - Upgrading hardware is cheaper than improving software
- Extremely complex machine

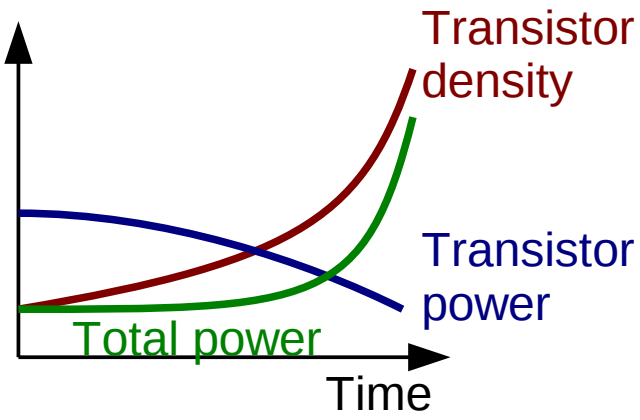
Outline

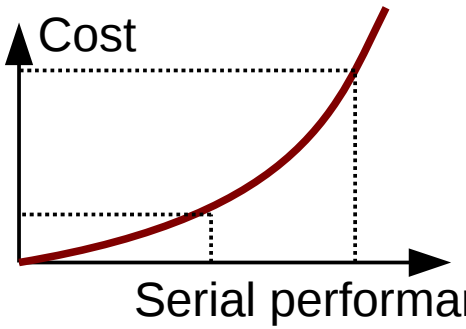
- Computer architecture crash course
  - The simplest processor
  - Exploiting instruction-level parallelism
- GPU, many-core: why, what for?
  - Technological trends and constraints
  - From graphics to general purpose
- Forms of parallelism, how to exploit them
  - Why we need (so much) parallelism: latency and throughput
  - Sources of parallelism: ILP, TLP, DLP
  - Uses of parallelism: horizontal, vertical
- Let's design a GPU!
  - Ingredients: Sequential core, Multi-core, Multi-threaded core, SIMD
  - Putting it all together
  - Architecture of current GPUs: cores, memory

Technology evolution

- Memory wall
  - Memory speed does not increase as fast as computing speed
  - More and more difficult to hide memory latency
- Power wall
  - Power consumption of transistors does not decrease as fast as density increases
  - Performance is now limited by power consumption
- ILP wall
  - Law of diminishing returns on Instruction-Level Parallelism
  - Pollack rule: cost ≃ performance²







Usage changes

- New applications demand **parallel processing**
  - Computer games : 3D graphics
  - Search engines, social networks… “big data” processing
- New computing devices are **power-constrained**
  - Laptops, cell phones, tablets…
    - Small, light, battery-powered
  - Datacenters
    - High power supply and cooling costs

Latency vs. throughput

- Latency: time to solution
  - CPUs
  - Minimize time, at the expense of power
- Throughput: quantity of tasks processed per unit of time
  - GPUs
  - Assumes unlimited parallelism
  - Minimize energy per operation

Amdahl's law

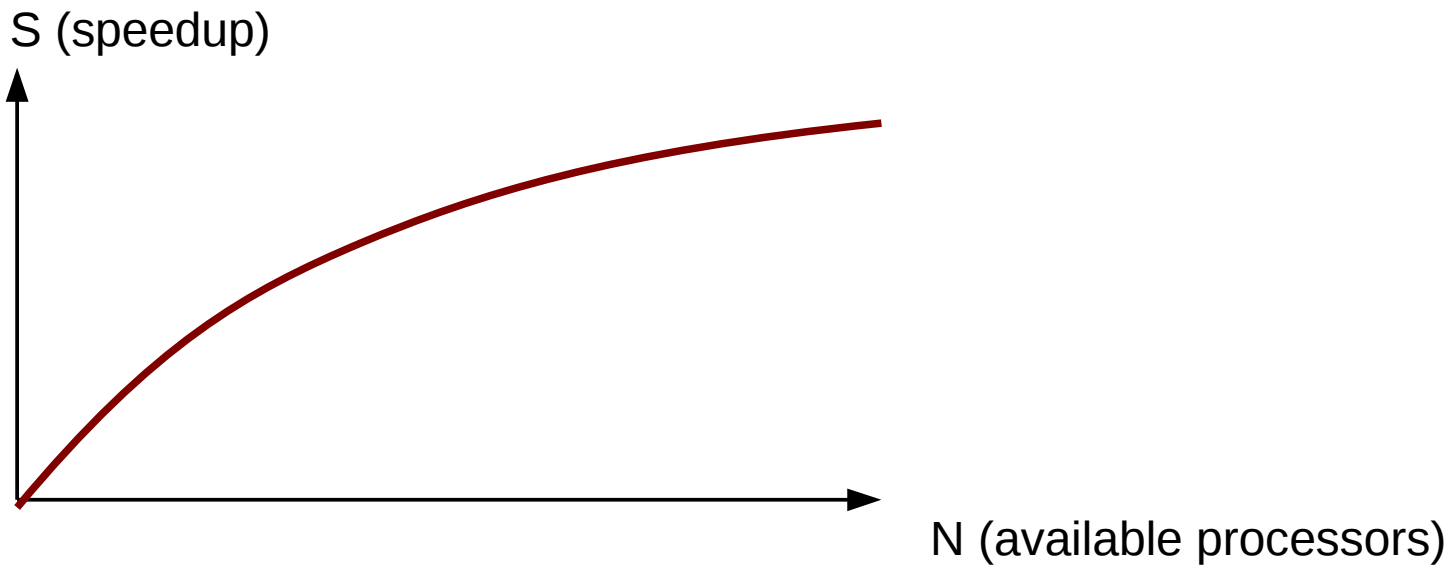
- Bounds speedup attainable on a parallel machine

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

Time to run sequential portions

Time to run parallel portions

S	Speedup
P	Ratio of parallel portions
N	Number of processors



G. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. AFIPS 1967.

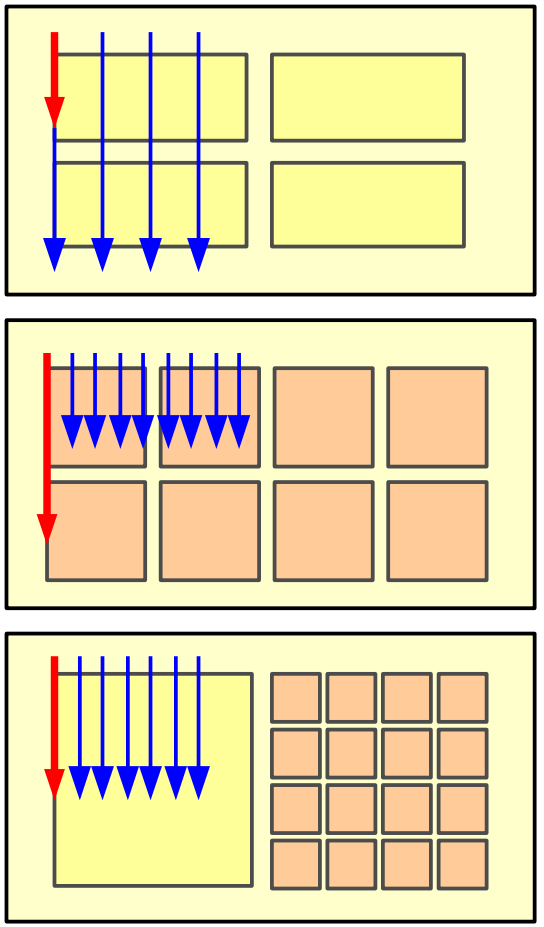
Why heterogeneous architectures?

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

Time to run sequential portions

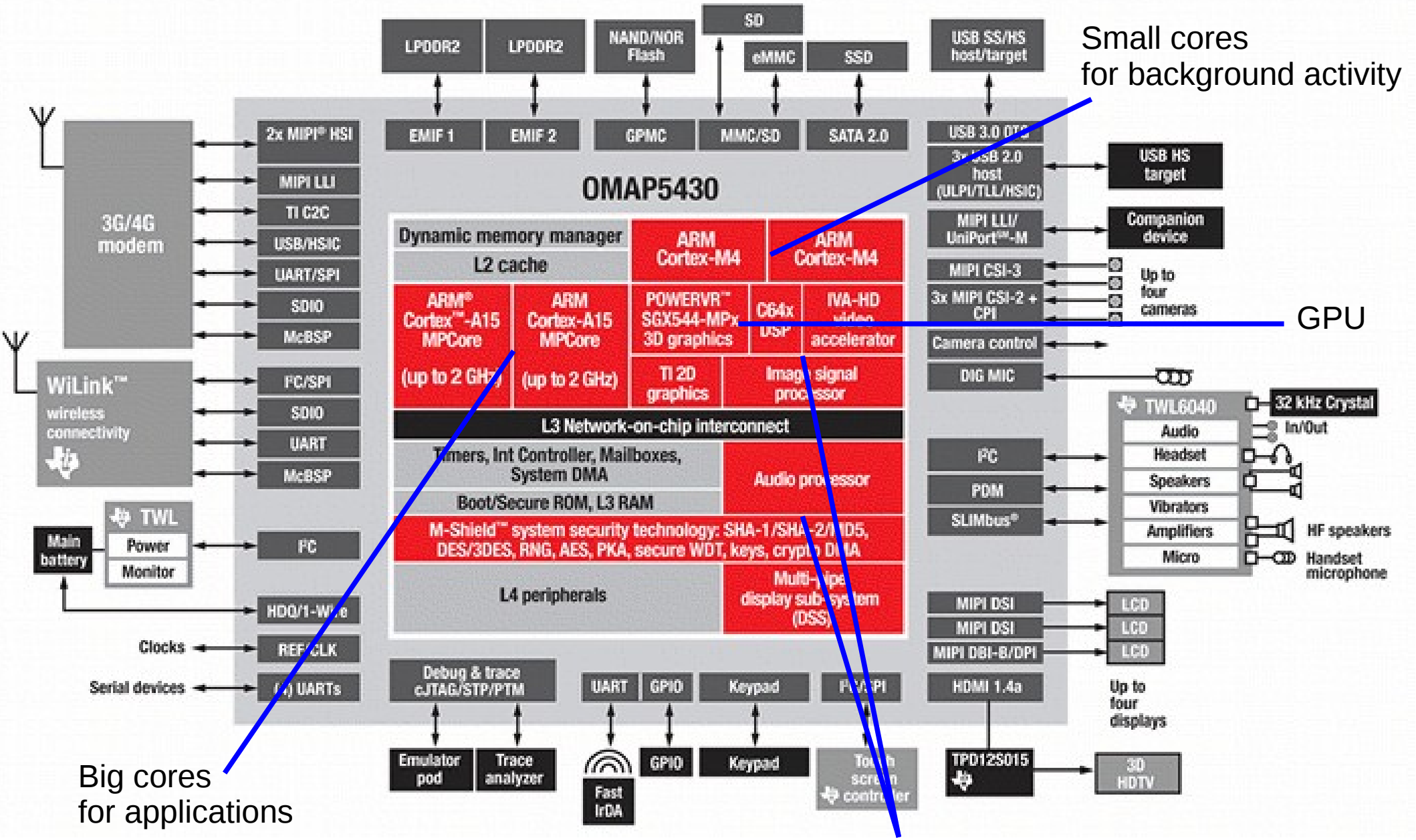
Time to run parallel portions

- Latency-optimized multi-core (CPU)
  - Low efficiency on parallel portions: spends too much resources
- Throughput-optimized multi-core (GPU)
  - Low performance on sequential portions
- Heterogeneous multi-core (CPU+GPU)
  - Use the right tool for the right job
  - Allows aggressive optimization for latency **or** for throughput



M. Hill, M. Marty. Amdahl's law in the multicore era. IEEE Computer, 2008.

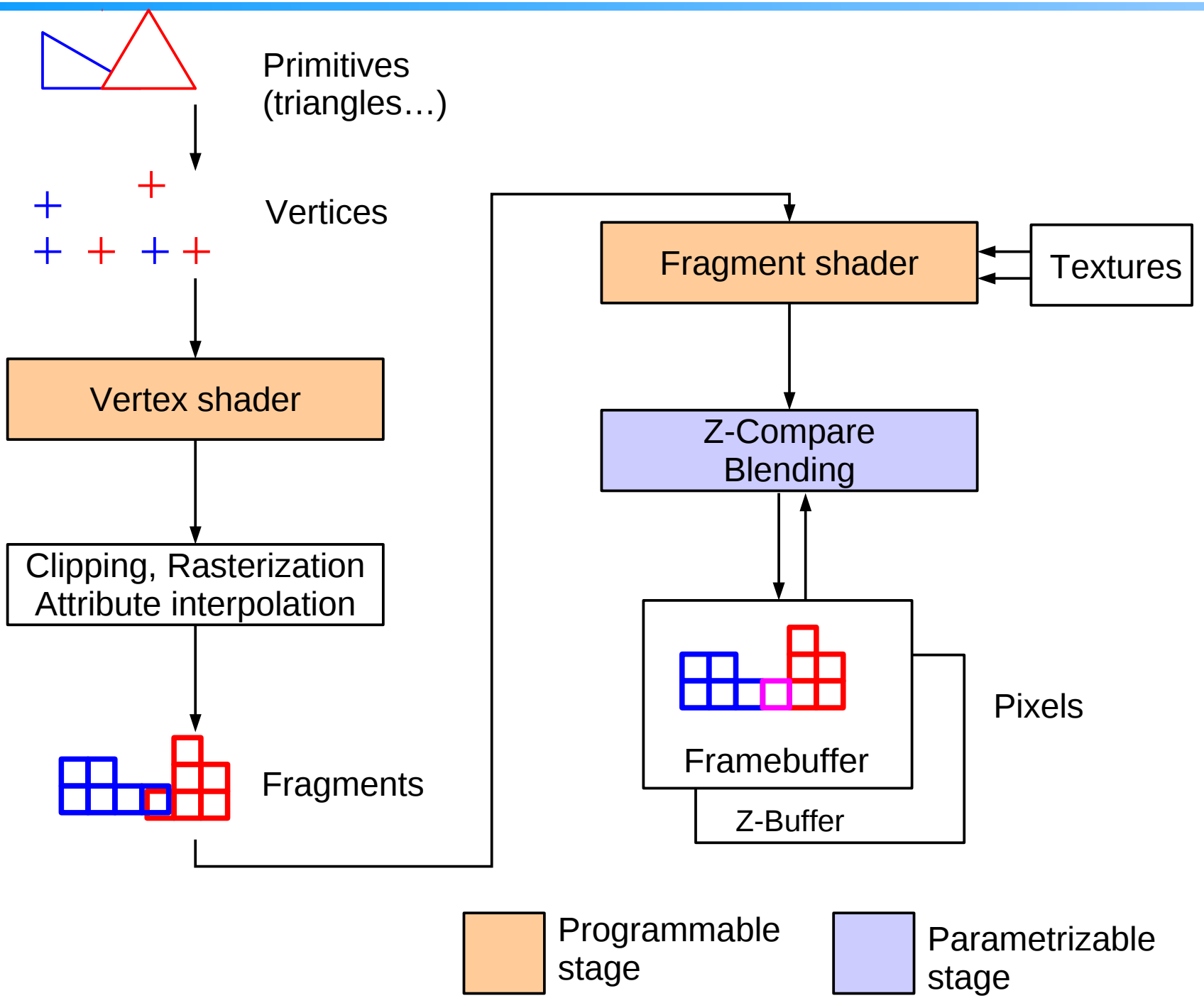
Example: System on Chip for smartphone



Outline

- Computer architecture crash course
  - The simplest processor
  - Exploiting instruction-level parallelism
- GPU, many-core: why, what for?
  - Technological trends and constraints
  - From graphics to general purpose
- Forms of parallelism, how to exploit them
  - Why we need (so much) parallelism: latency and throughput
  - Sources of parallelism: ILP, TLP, DLP
  - Uses of parallelism: horizontal, vertical
- Let's design a GPU!
  - Ingredients: Sequential core, Multi-core, Multi-threaded core, SIMD
  - Putting it all together
  - Architecture of current GPUs: cores, memory

The (simplest) graphics rendering pipeline



How much performance do we need

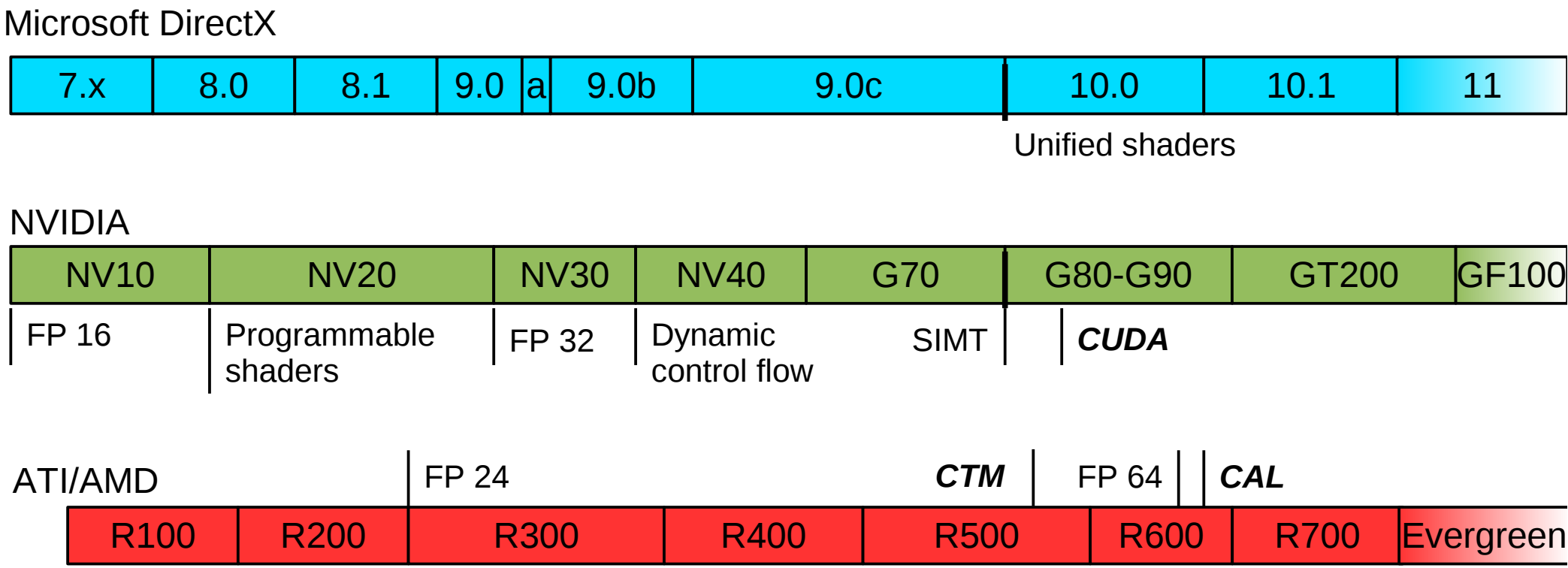
- … to run 3DMark 11 at 50 frames/second?

Element	Per frame	Per second
Vertices	12.0M	600M
Primitives	12.6M	630M
Fragments	180M	9.0G
Instructions	14.4G	**720G**

- Intel Core i7 2700K: 56 Ginsn/s peak
  - We need to go 13x faster
  - Make a special-purpose accelerator

Source: Damien Triolet, Hardware.fr

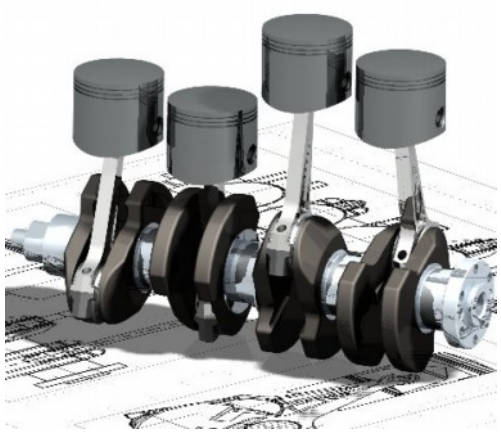
Beginnings of GPGPU

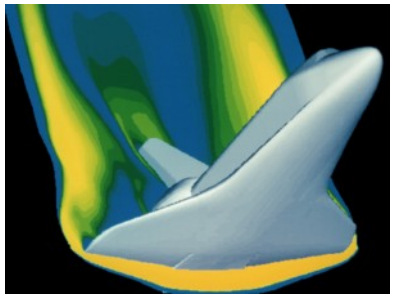


Today: what do we need GPUs for?

1. 3D graphics rendering for games
   - Complex texture mapping, lighting computations…
2. Computer Aided Design workstations
   - Complex geometry
3. GPGPU
   - Complex synchronization, data movements

- One chip to rule them all
  - Find the common denominator

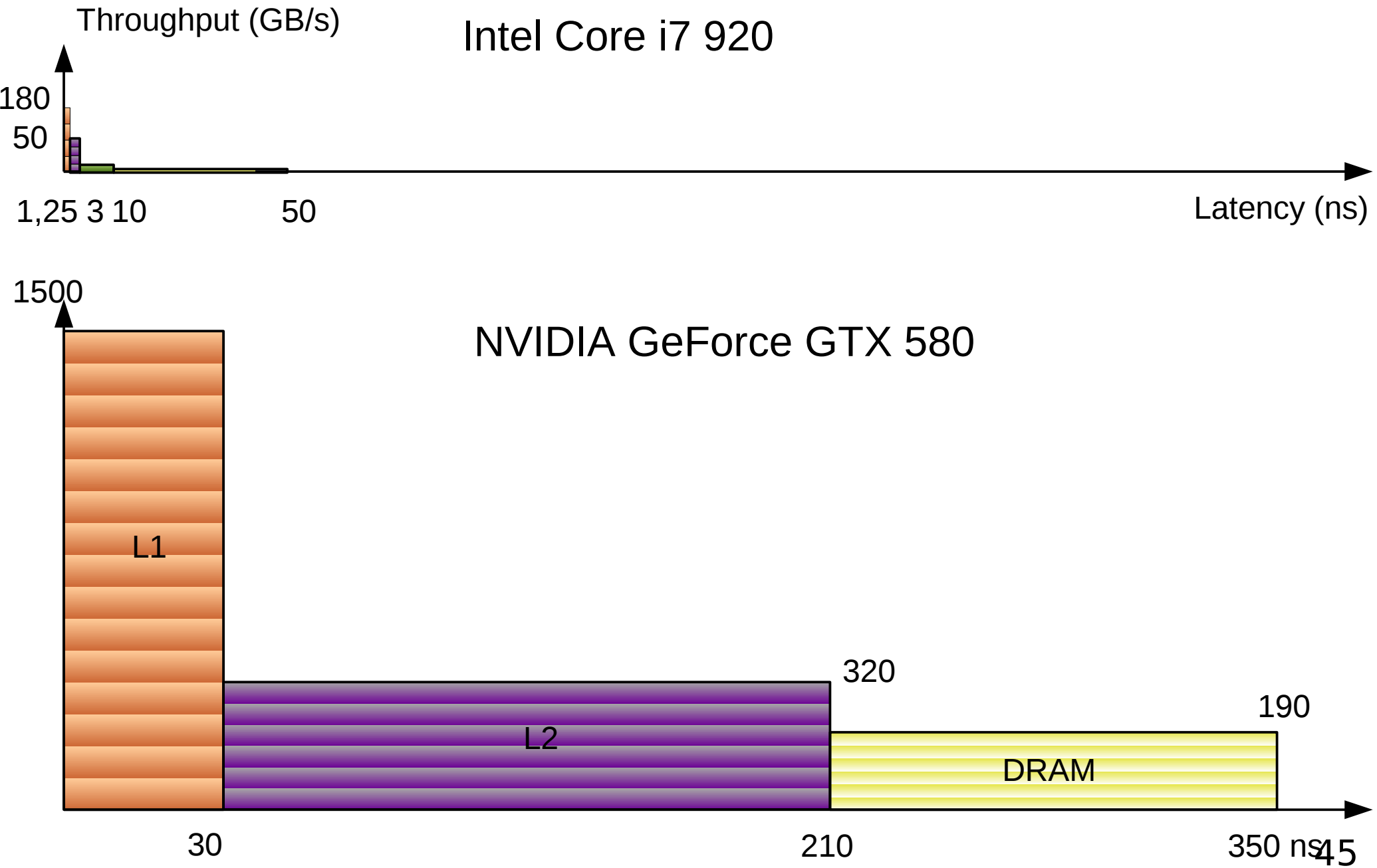




Outline

- Computer architecture crash course
  - The simplest processor
  - Exploiting instruction-level parallelism
- GPU, many-core: why, what for?
  - Technological trends and constraints
  - From graphics to general purpose
- Forms of parallelism, how to exploit them
  - Why we need (so much) parallelism: latency and throughput
  - Sources of parallelism: ILP, TLP, DLP
  - Uses of parallelism: horizontal, vertical
- Let's design a GPU!
  - Ingredients: Sequential core, Multi-core, Multi-threaded core, SIMD
  - Putting it all together
  - Architecture of current GPUs: cores, memory

Little's law: data=throughput×latency

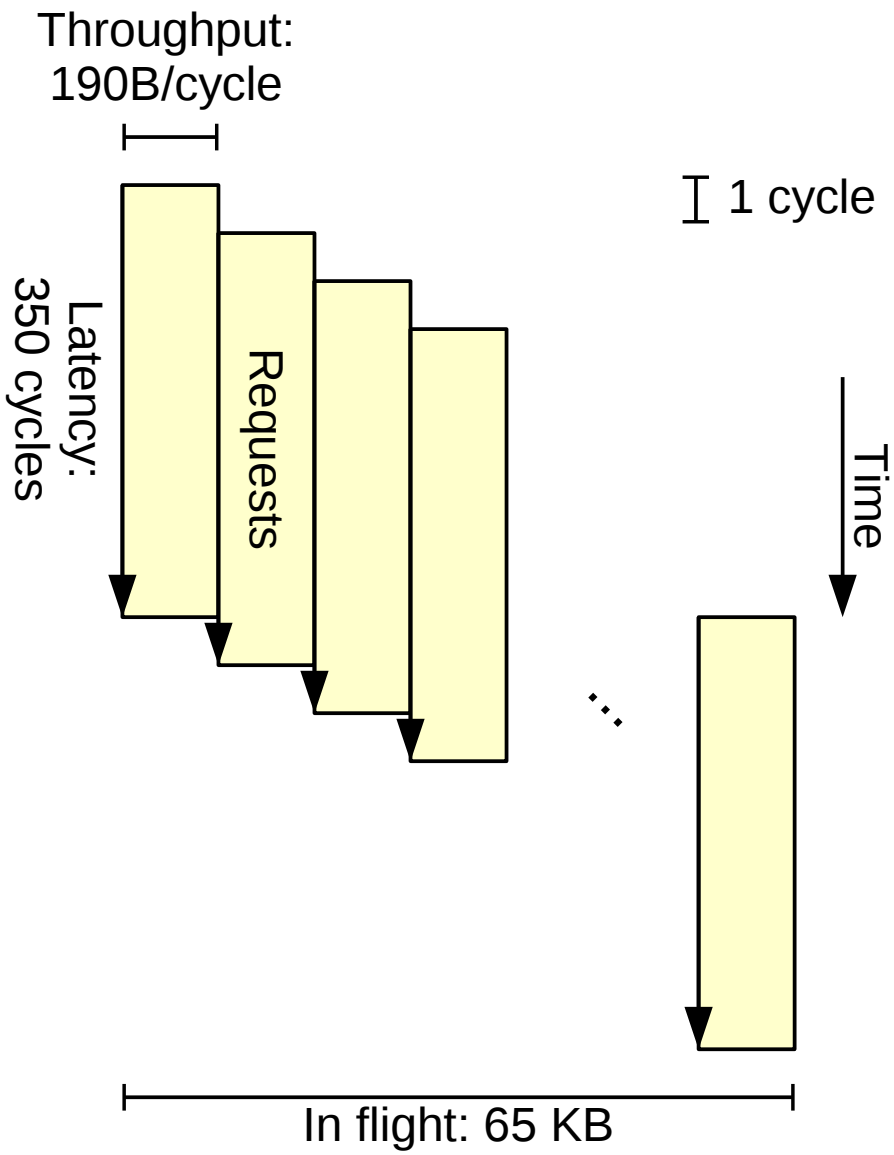


J. Little. A proof for the queuing formula L= λ W. JSTOR 1961.

Hiding memory latency with pipelining

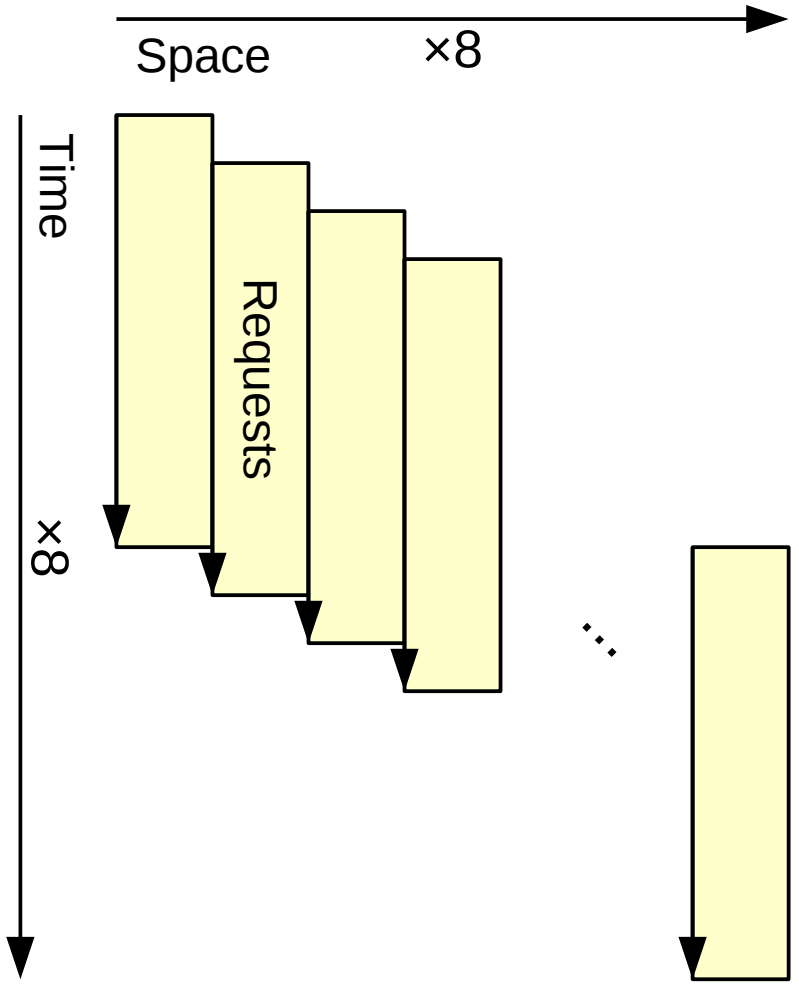
- Memory throughput: 190 GB/s
- Memory latency: 350 ns
- Data in flight = 66 500 Bytes

- At 1 GHz:
  190 Bytes/cycle,
  350 cycles to wait



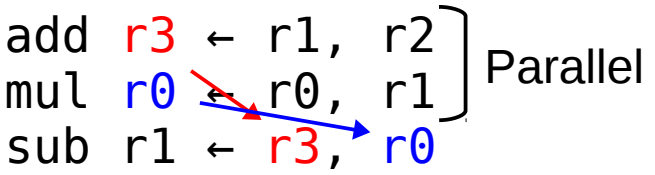
Consequence: more parallelism

- GPU vs. CPU
  - 8× more parallelism to feed more units (throughput)
  - 8× more parallelism to hide longer latency
  - 64× more total parallelism
- How to find this parallelism?

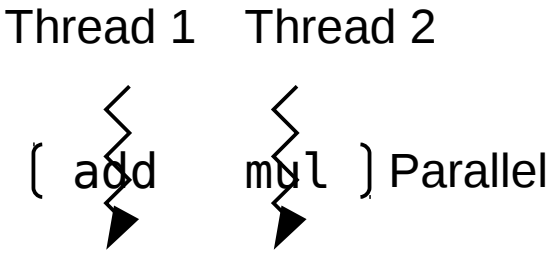


Sources of parallelism

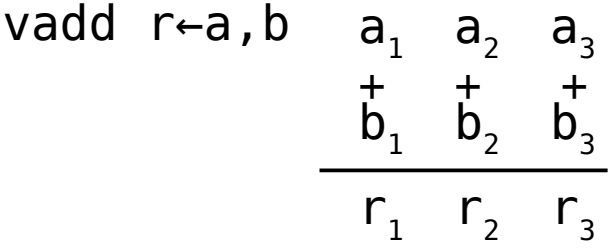
- ILP: Instruction-Level Parallelism
  - Between independent instructions in sequential program



- TLP: Thread-Level Parallelism
  - Between independent execution contexts: threads



- DLP: Data-Level Parallelism
  - Between elements of a vector: same operation on several elements



Example: X ← a×X

- In-place scalar-vector product: X ← a×X

Sequential (ILP)

```
For i = 0 to n-1 do:
   X[i] ← a * X[i]
```

Threads (TLP)

```
Launch n threads:
   X[tid] ← a * X[tid]
```

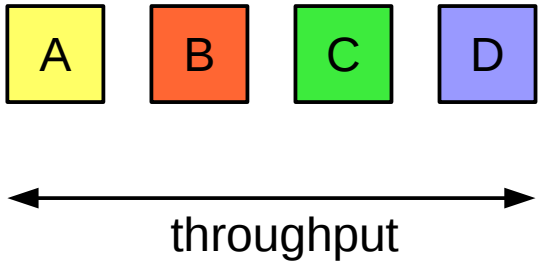
Vector (DLP)

```
   X ← a * X
```

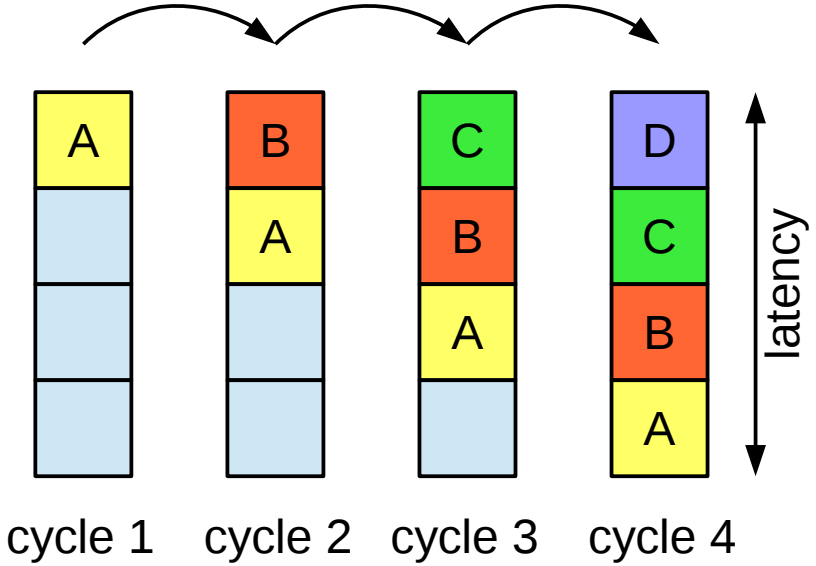
- Or any combination of the above

Uses of parallelism

- “Horizontal” parallelism for throughput
  - More units working in parallel



- “Vertical” parallelism for latency hiding
  - Pipelining: keep units busy when waiting for dependencies, memory



How to extract parallelism?

	Horizontal	Vertical
ILP	Superscalar	Pipelined
TLP	Multi-core SMT	Interleaved / switch-on-event multithreading
DLP	SIMD / SIMT	Vector / temporal SIMT

- We have seen the first row: ILP
- We will now review techniques for the next rows: TLP, DLP

Outline

- Computer architecture crash course
  - The simplest processor
  - Exploiting instruction-level parallelism
- GPU, many-core: why, what for?
  - Technological trends and constraints
  - From graphics to general purpose
- Forms of parallelism, how to exploit them
  - Why we need (so much) parallelism: latency and throughput
  - Sources of parallelism: ILP, TLP, DLP
  - Uses of parallelism: horizontal, vertical
- Let's design a GPU!
  - Ingredients: Sequential core, Multi-core, Multi-threaded core, SIMD
  - Putting it all together
  - Architecture of current GPUs: cores, memory

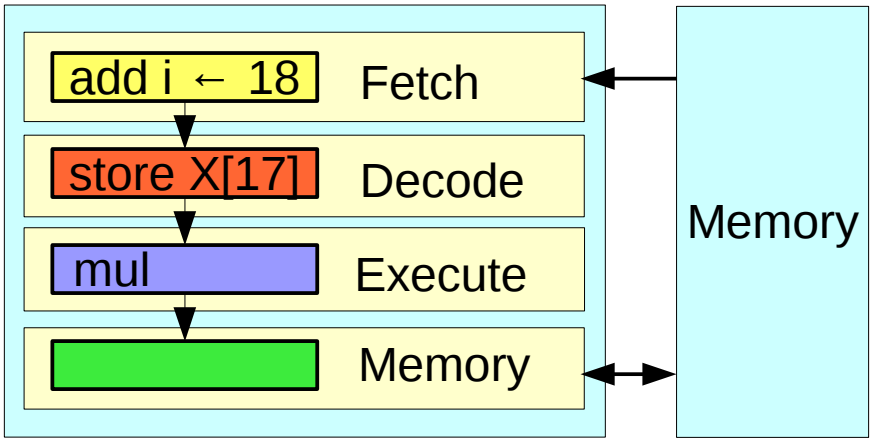
Sequential processor

```
for i = 0 to n-1
   X[i] ← a * X[i]
```

Source code

```
   move  i ← 0
loop:
   load  t ← X[i]
   mul   t ← a×t
   store X[i] ← t
   add   i ← i+1
   branch i<n? loop
```

Machine code

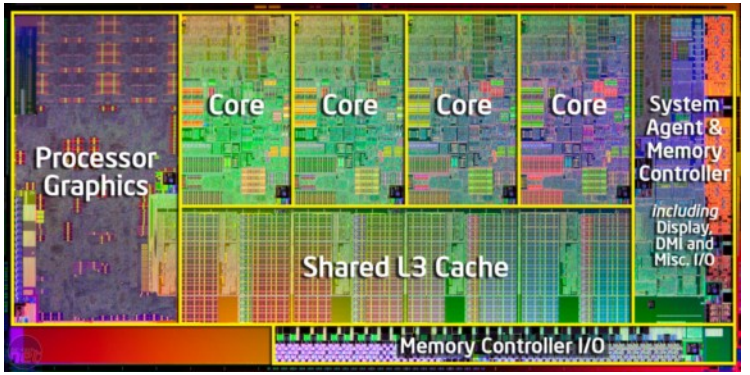


Sequential CPU

- Focuses on instruction-level parallelism
  - Exploits ILP: vertically (pipelining) and horizontally (superscalar)

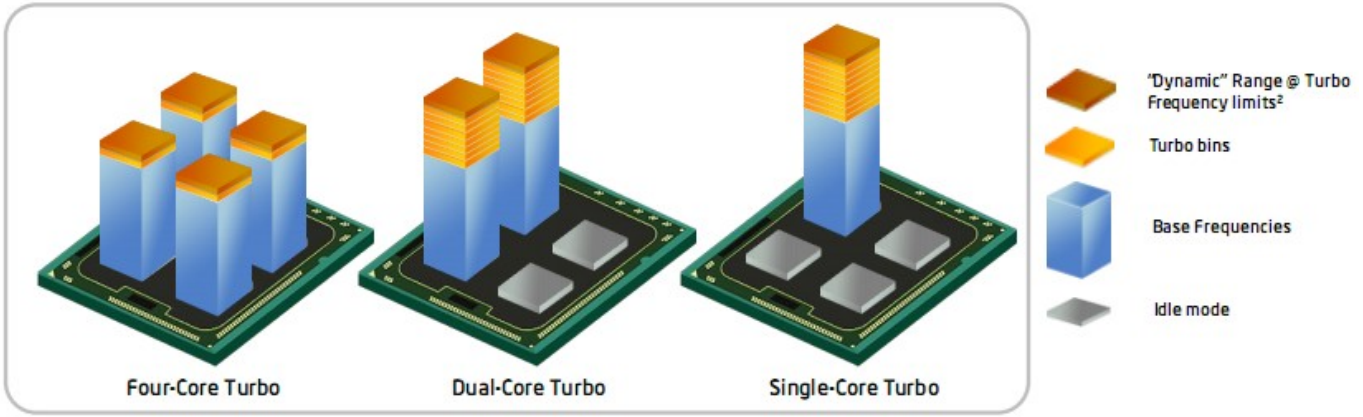
The incremental approach: multi-core

- Several processors
  on a single chip
  sharing one memory space



Intel Sandy Bridge

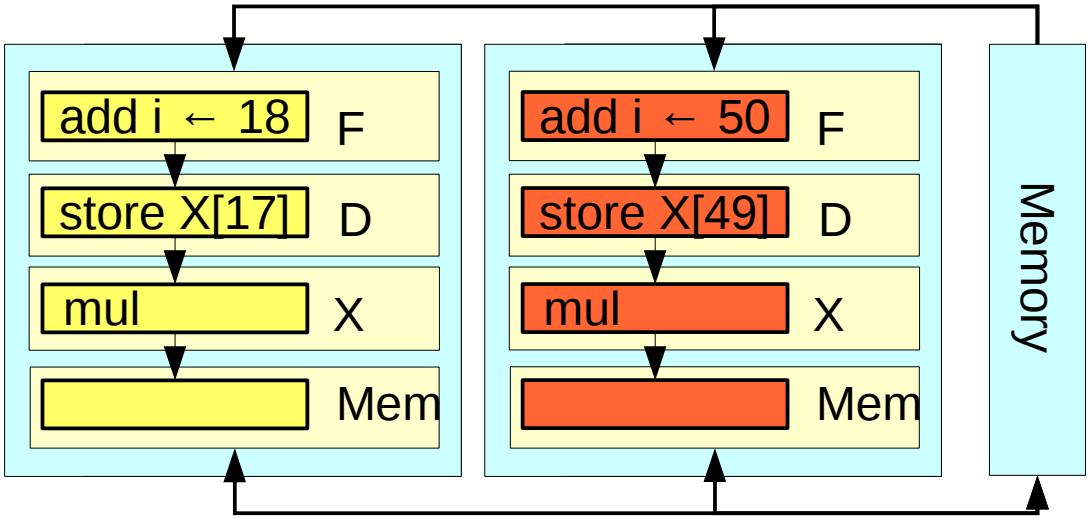
- Area: benefits from Moore's law
- Power: extra cores consume little when not in use
  - e.g. Intel Turbo Boost



Source: Intel

Homogeneous multi-core

- Horizontal use of thread-level parallelism

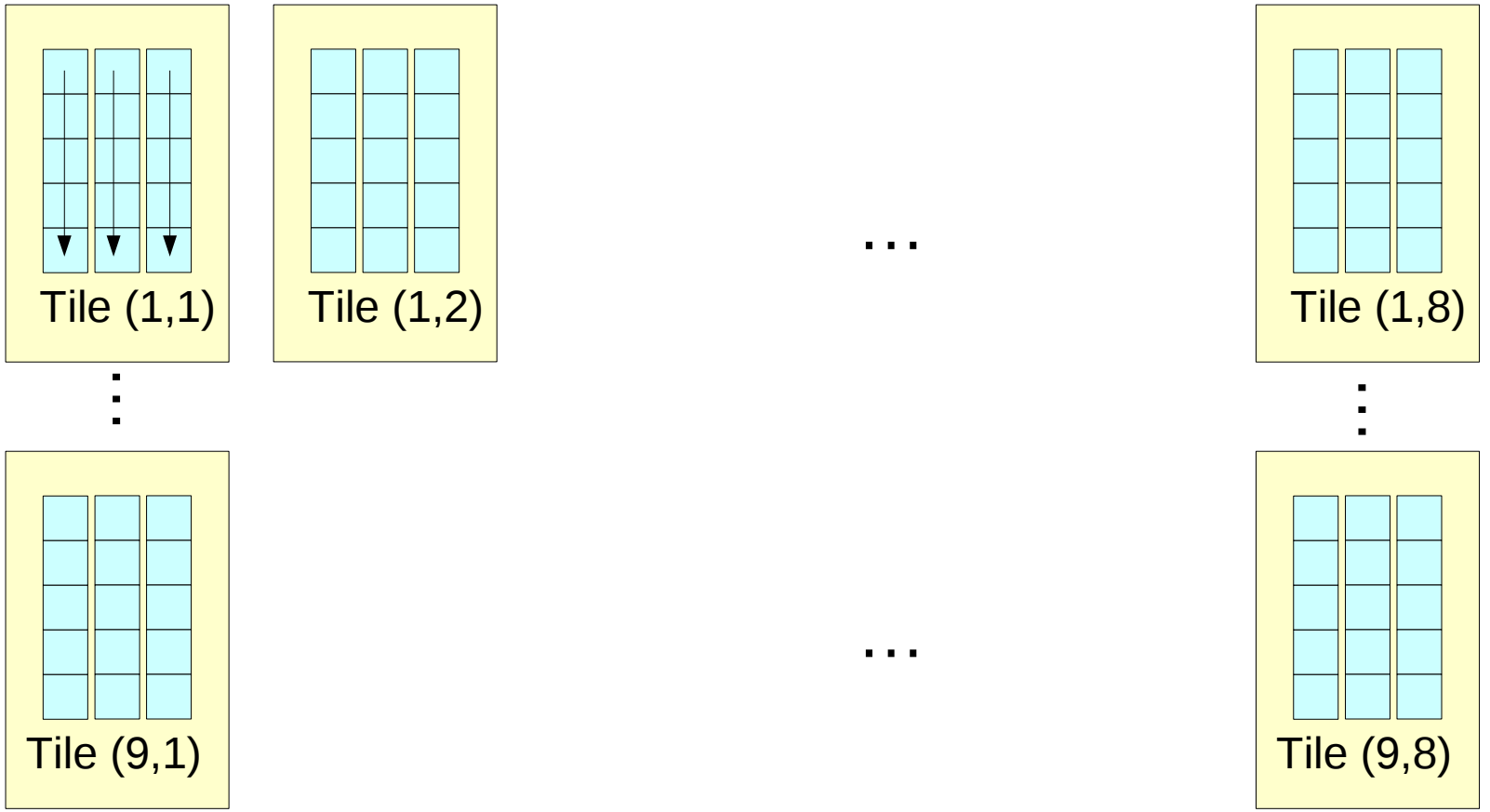


Threads: T0 T1

- Improves peak throughput

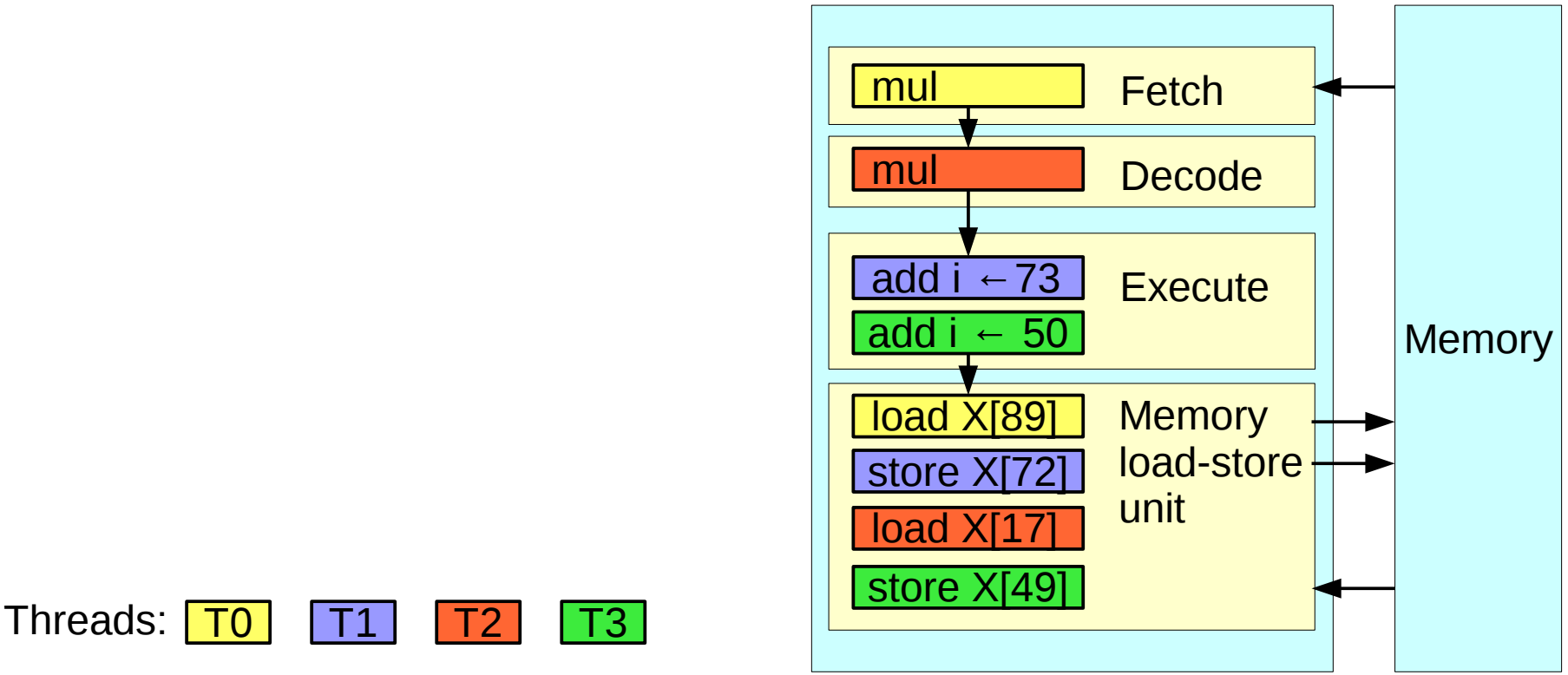
Example: Tilera Tile-GX

- Grid of (up to) 72 tiles
- Each tile: 3-way VLIW processor, 5 pipeline stages, 1.2 GHz



Interleaved multi-threading

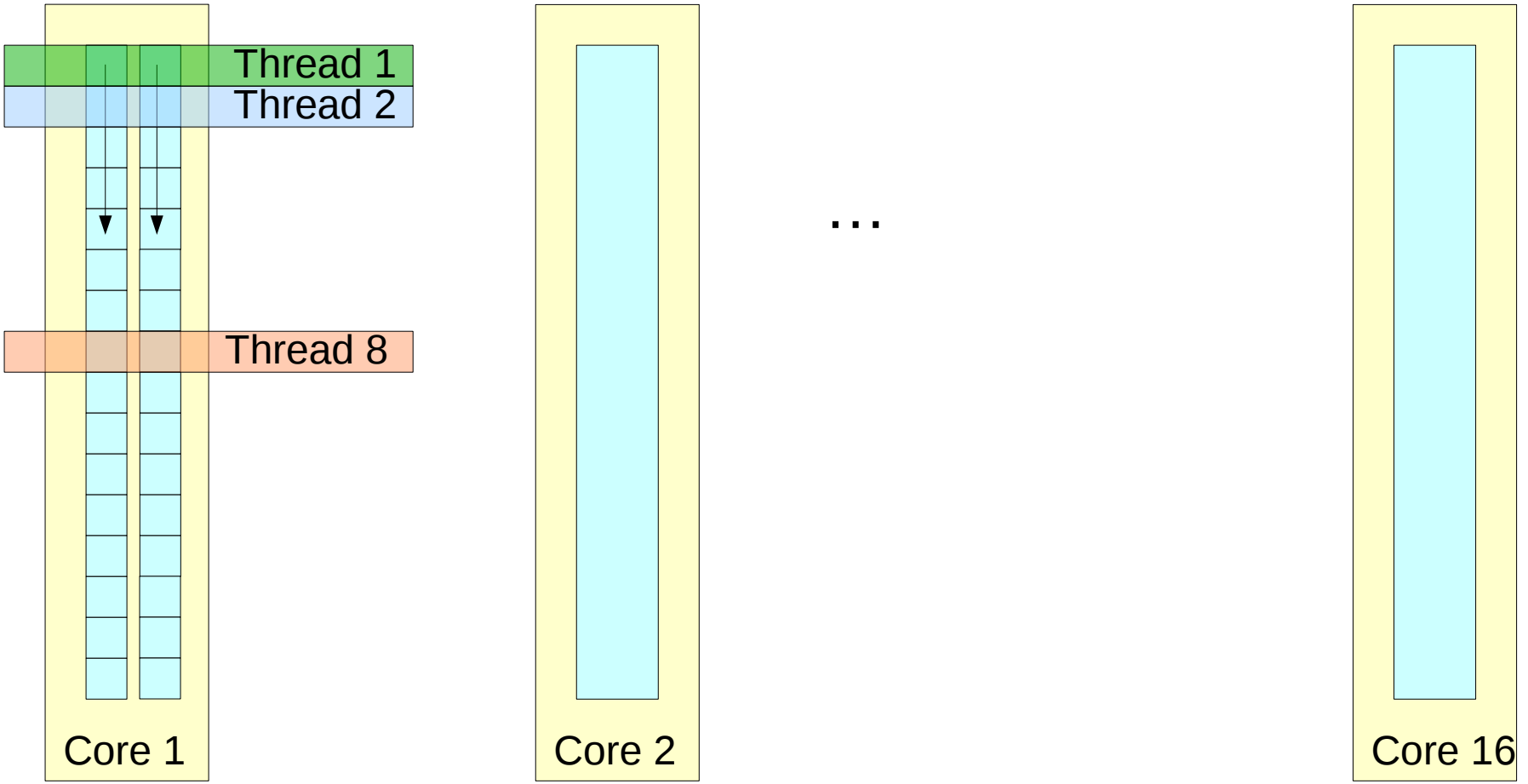
- Vertical use of thread-level parallelism



- Hides latency thanks to explicit parallelism improves achieved throughput

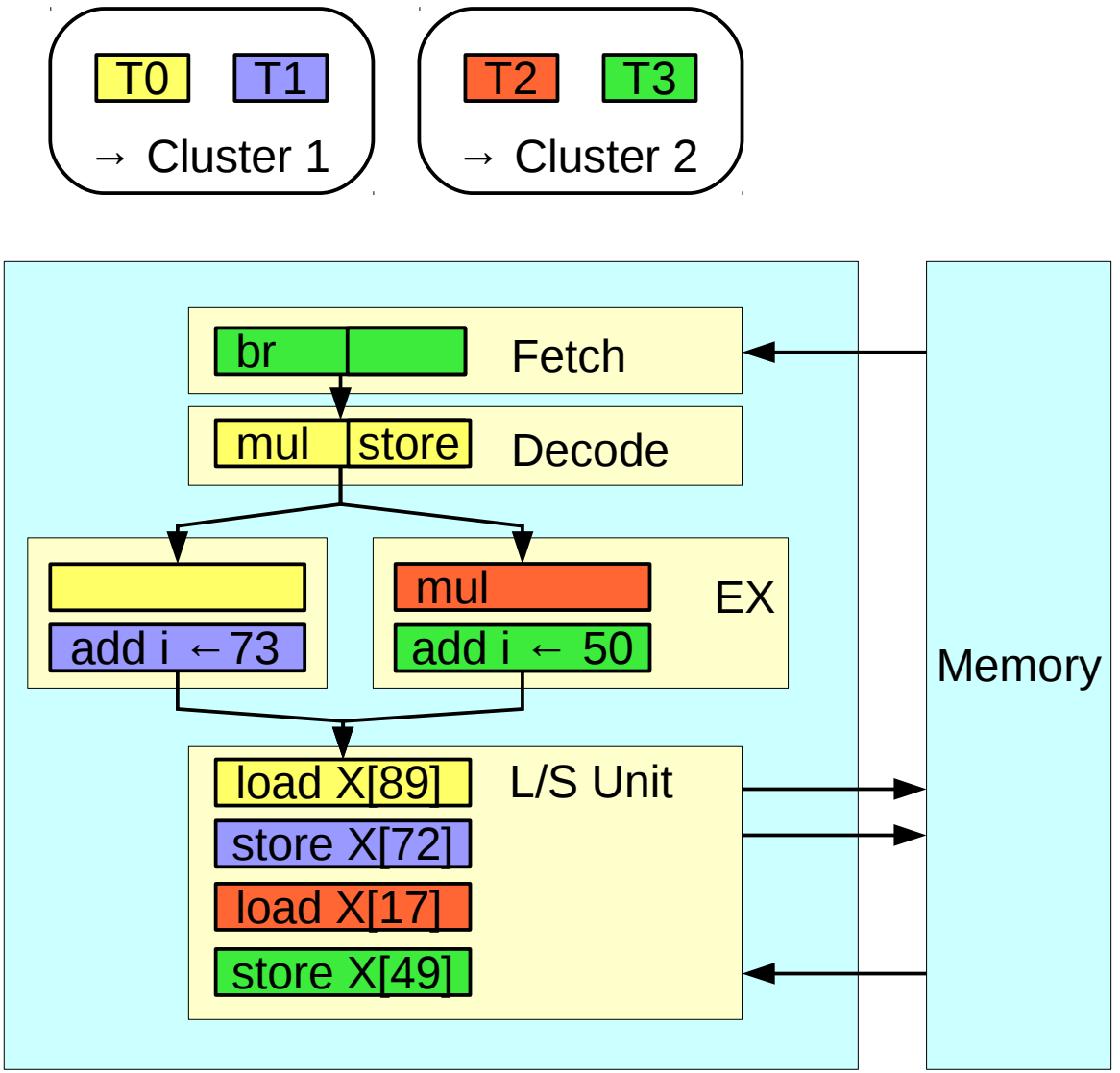
Example: Oracle Sparc T5

- 16 cores / chip
- Core: out-of-order superscalar, 8 threads
- 15 pipeline stages, 3.6 GHz



Clustered multi-core

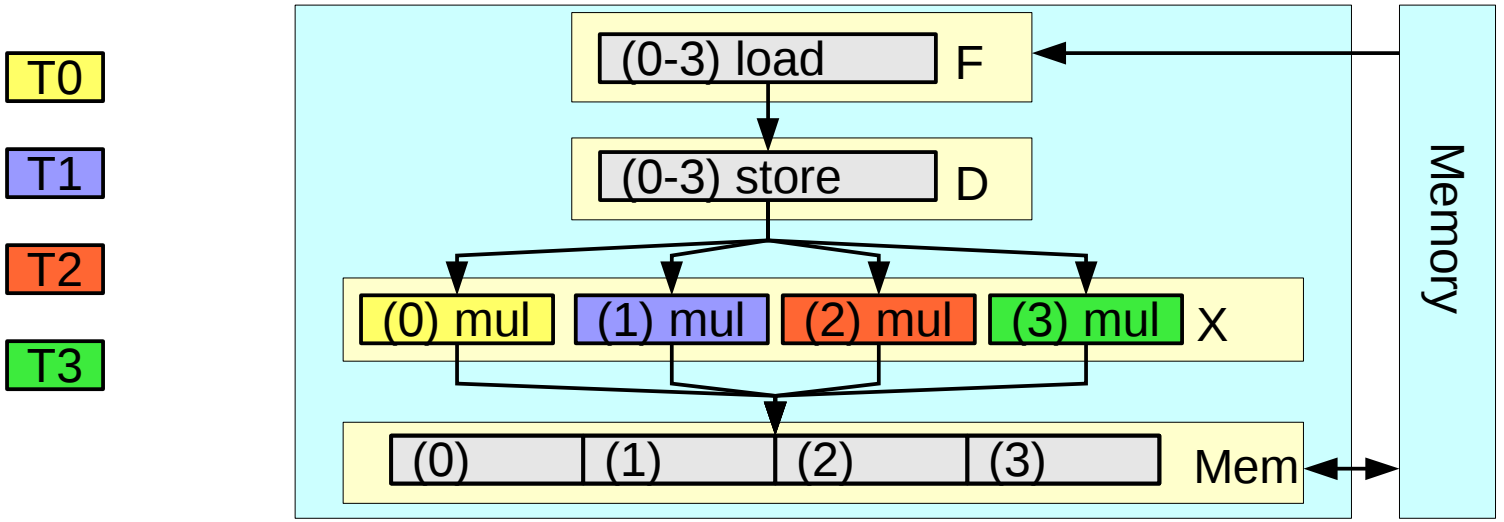
- For each individual unit, select between
  - Horizontal replication
  - Vertical time-multiplexing
- Examples
  - Sun UltraSparc T2, T3
  - AMD Bulldozer
  - IBM Power 7



- Area-efficient tradeoff
- Blurs boundaries between cores

Implicit SIMD

- Factorization of fetch/decode, load-store units
  - Fetch 1 instruction on behalf of several threads
  - Read 1 memory location and broadcast to several registers



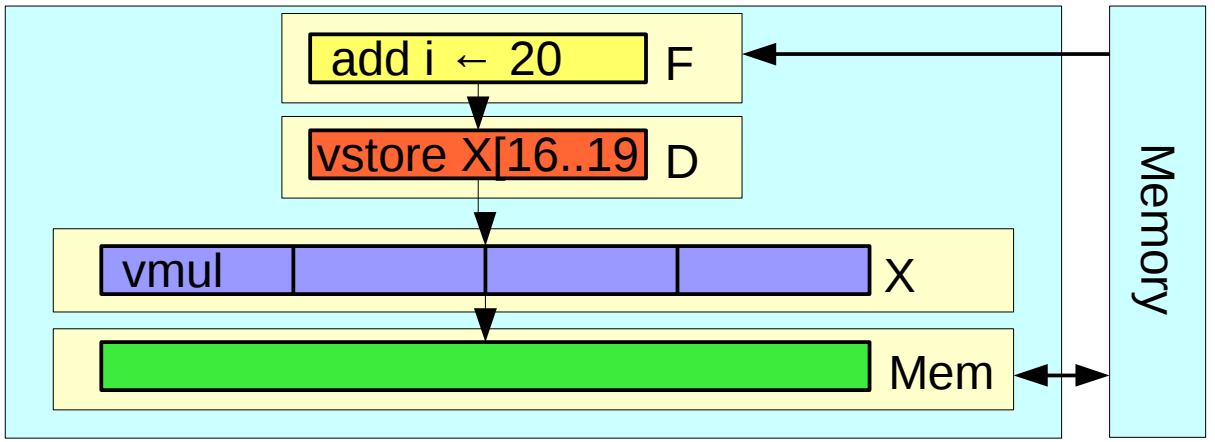
- In NVIDIA-speak
  - SIMT: Single Instruction, Multiple Threads
  - Convoy of synchronized threads: *warp*
- Extracts DLP from multi-thread applications

Explicit SIMD

- Single Instruction Multiple Data
- Horizontal use of data level parallelism

```
loop:
    vload  T ← X[i]
    vmul   T ← a×T
    vstore X[i] ← T
    add    i ← i+4
    branch i<n? loop
```

Machine code



SIMD CPU

- Examples
  - Intel MIC (16-wide)
  - AMD GCN GPU (16-wide×4-deep)
  - Most general purpose CPUs (4-wide to 8-wide)

Quizz: link the words

Parallelism

- ILP
- TLP
- DLP

Use

- Horizontal:
  more throughput
- Vertical:
  hide latency

Architectures

- Superscalar processor
- Homogeneous multi-core
- Multi-threaded core
- Clustered multi-core
- Implicit SIMD
- Explicit SIMD

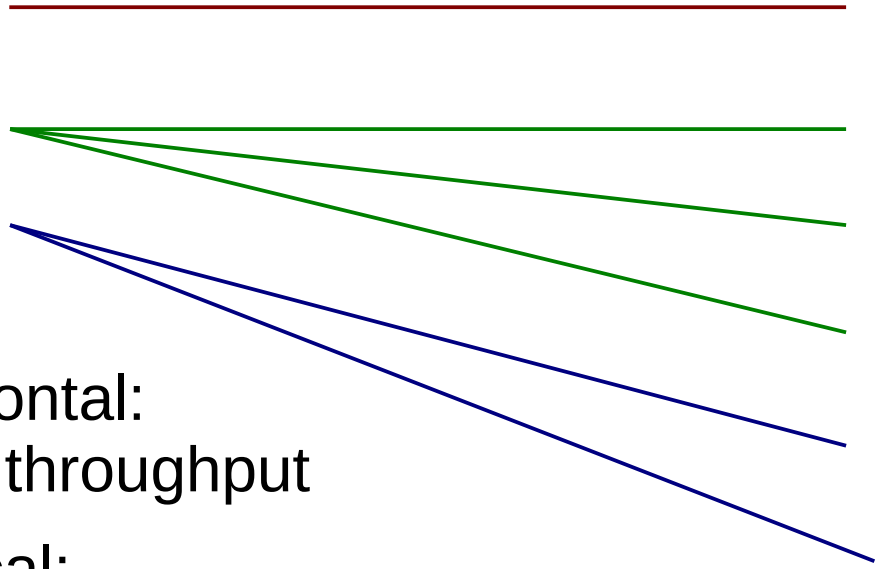
Quizz: link the words

Parallelism

- ILP
- TLP
- DLP

Use

- Horizontal:
  more throughput
- Vertical:
  hide latency



Architectures

- Superscalar processor
- Homogeneous multi-core
- Multi-threaded core
- Clustered multi-core
- Implicit SIMD
- Explicit SIMD

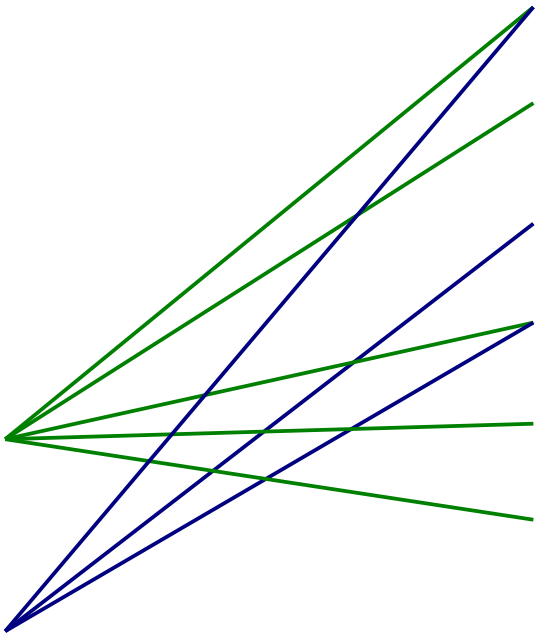
Quizz: link the words

Parallelism

- ILP
- TLP
- DLP

Use

- Horizontal:
  more throughput
- Vertical:
  hide latency



Architectures

- Superscalar processor
- Homogeneous multi-core
- Multi-threaded core
- Clustered multi-core
- Implicit SIMD
- Explicit SIMD

Outline

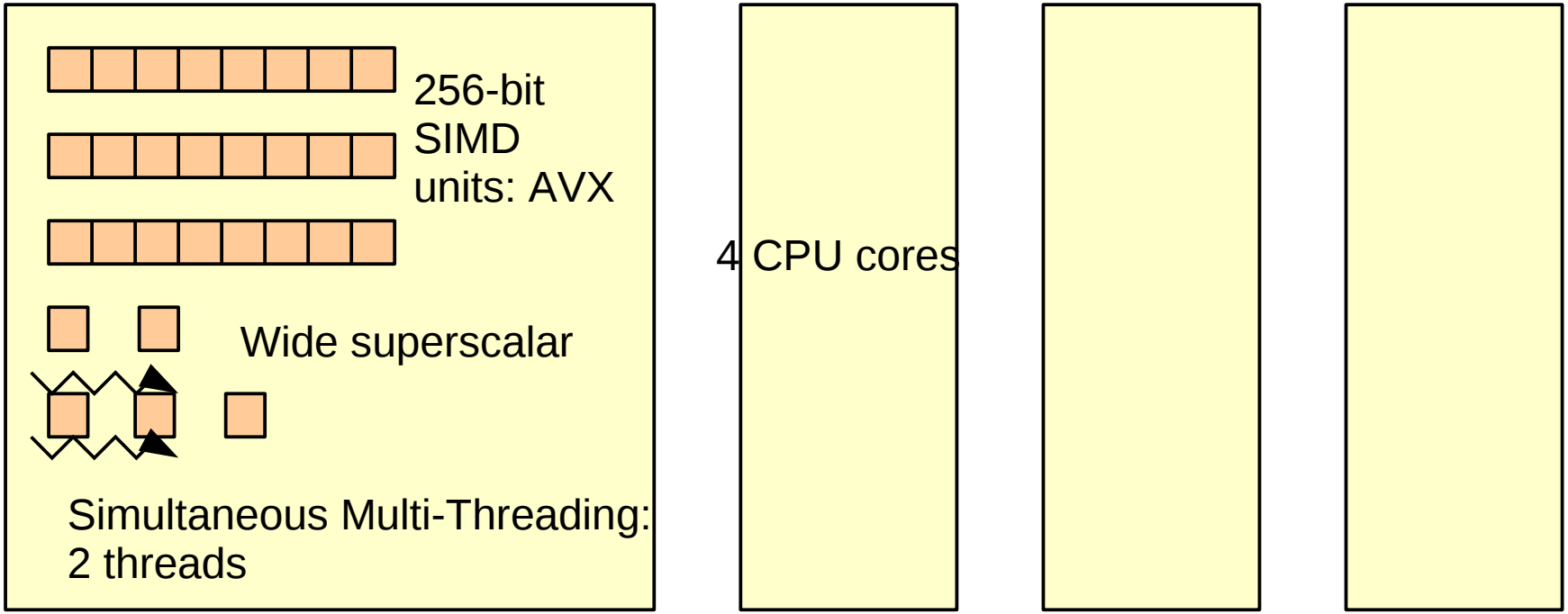
- Computer architecture crash course
  - The simplest processor
  - Exploiting instruction-level parallelism
- GPU, many-core: why, what for?
  - Technological trends and constraints
  - From graphics to general purpose
- Forms of parallelism, how to exploit them
  - Why we need (so much) parallelism: latency and throughput
  - Sources of parallelism: ILP, TLP, DLP
  - Uses of parallelism: horizontal, vertical
- Let's design a GPU!
  - Ingredients: Sequential core, Multi-core, Multi-threaded core, SIMD
  - Putting it all together
  - Architecture of current GPUs: cores, memory

Hierarchical combination

- Both CPUs and GPUs combine these techniques
  - Multiple cores
  - Multiple threads/core
  - SIMD units

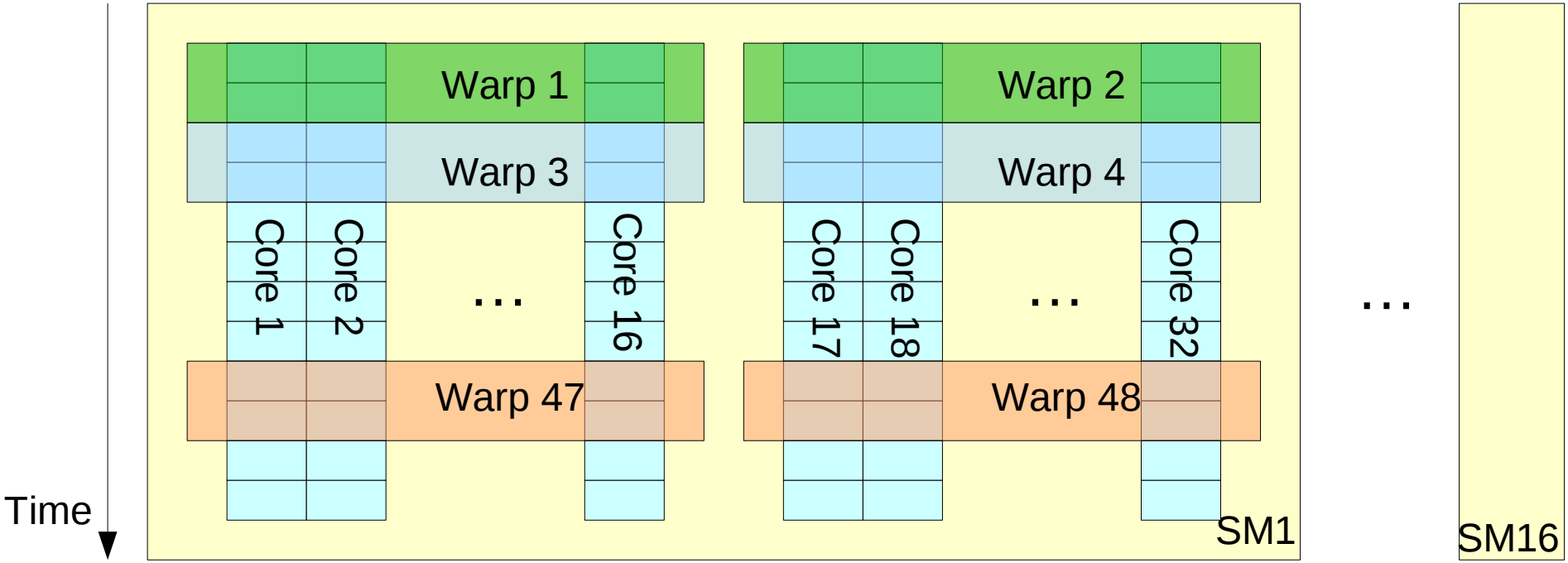
Example CPU: Intel Core i7

- Is a wide superscalar, but has also
  - Multicore
  - Multi-thread / core
  - SIMD units
- Up to 117 operations/cycle from 8 threads



Example GPU: NVIDIA GeForce GTX 580

- SIMT: warps of 32 threads
- 16 SMs / chip
- 2×16 cores / SM, 48 warps / SM



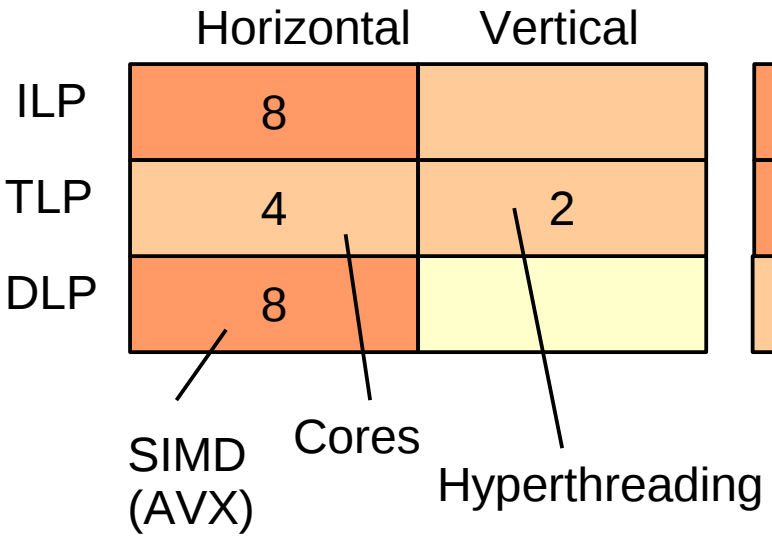
- Up to 512 operations per cycle from 24576 threads in flight

Taxonomy of parallel architectures

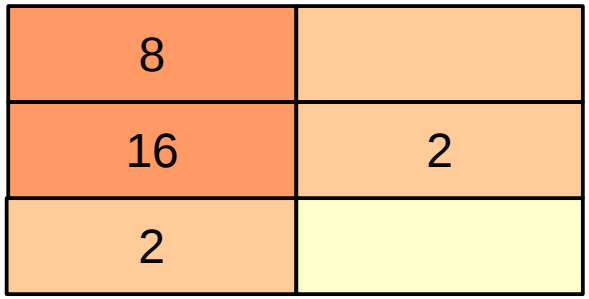
	Horizontal	Vertical
ILP	Superscalar / VLIW	Pipelined
TLP	Multi-core SMT	Interleaved / switch-on-event multithreading
DLP	SIMD / SIMT	Vector / temporal SIMT

Classification: multi-core

Intel Haswell

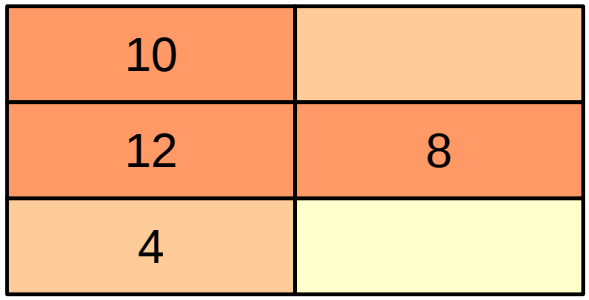


Fujitsu SPARC64 X

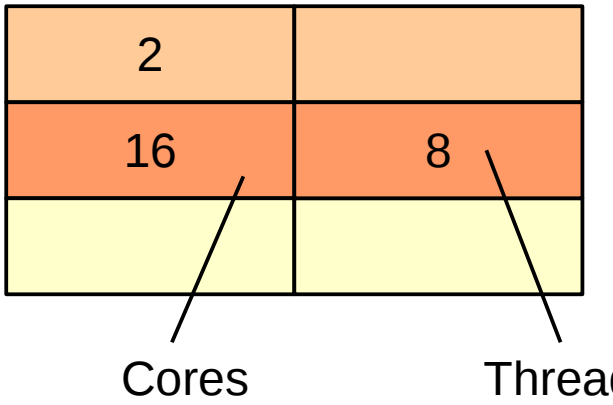


General-purpose multi-cores:
balance ILP, TLP and DLP

IBM Power 8

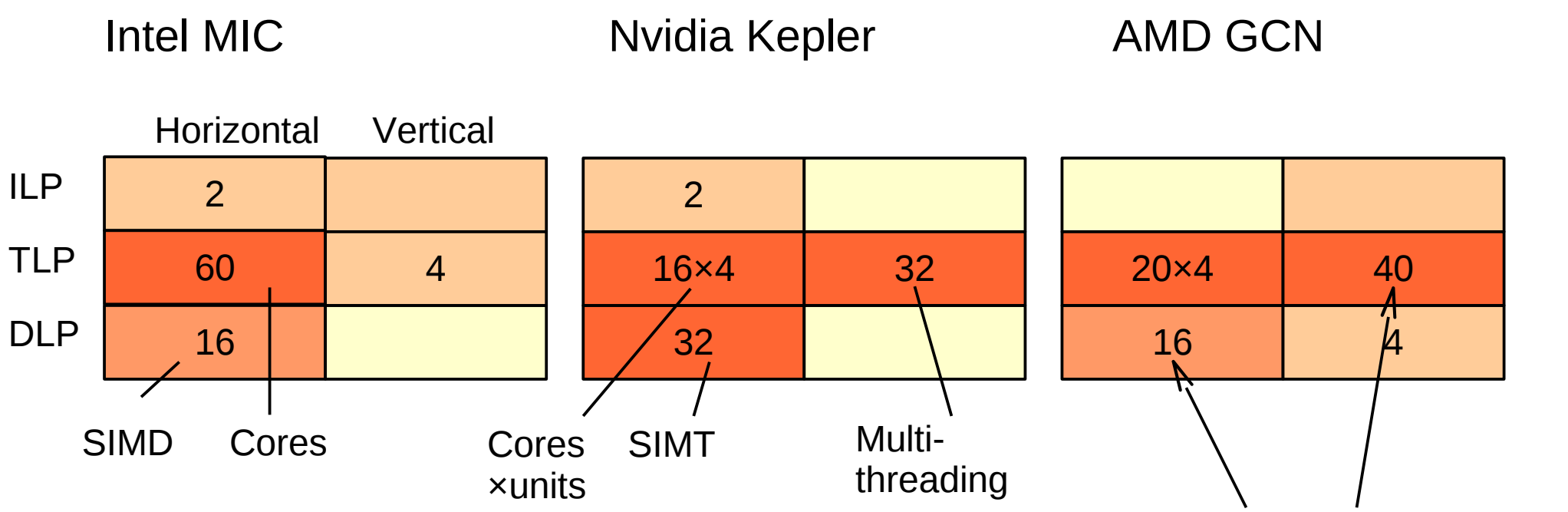


Oracle Sparc T5

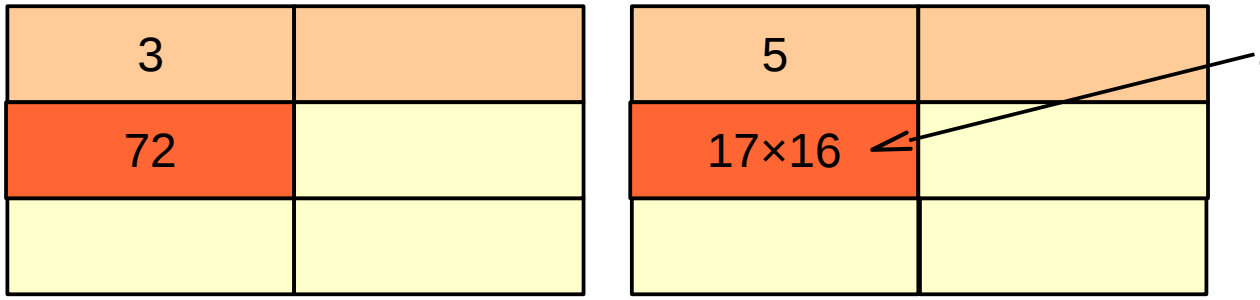


Sparc T:
focus on TLP

Classification: GPU and many small-core



GPU: focus on DLP, TLP horizontal and vertical



Many small-core: focus on horizontal TLP

Takeaway

- All processors use hardware mechanisms to turn parallelism into performance
- GPUs focus on Thread-level and Data-level parallelism

Outline

- Computer architecture crash course
  - The simplest processor
  - Exploiting instruction-level parallelism
- GPU, many-core: why, what for?
  - Technological trends and constraints
  - From graphics to general purpose
- Forms of parallelism, how to exploit them
  - Why we need (so much) parallelism: latency and throughput
  - Sources of parallelism: ILP, TLP, DLP
  - Uses of parallelism: horizontal, vertical
- Let's design a GPU!
  - Ingredients: Sequential core, Multi-core, Multi-threaded core, SIMD
  - Putting it all together
  - Architecture of current GPUs: cores, memory

Computation cost vs. memory cost

- Power measurements on NVIDIA GT200

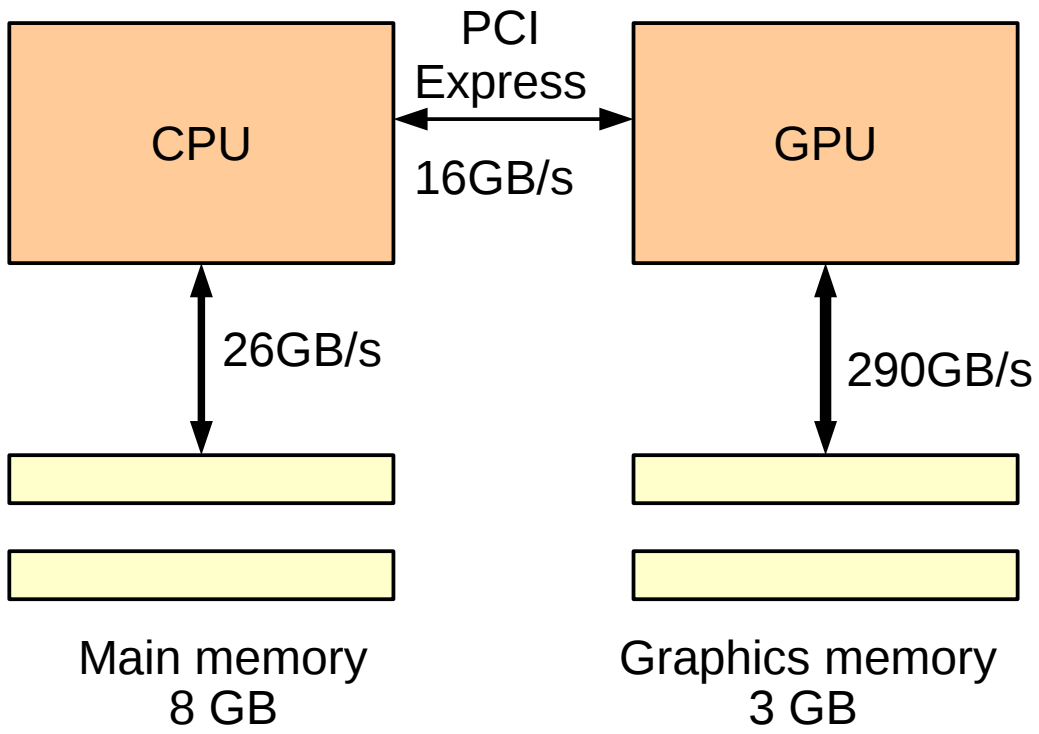
	Energy/op (nJ)	Total power (W)
Instruction control	1.8	18
Multiply-add on a 32-wide warp	3.6	36
Load 128B from DRAM	80	90

- With the same amount of energy
  - Load 1 word from external memory (DRAM)
  - Compute 44 flops
- Must optimize memory accesses first!

External memory: discrete GPU

Classical CPU-GPU model

- Split memory spaces
- Highest bandwidth from GPU memory
- Transfers to main memory are slower

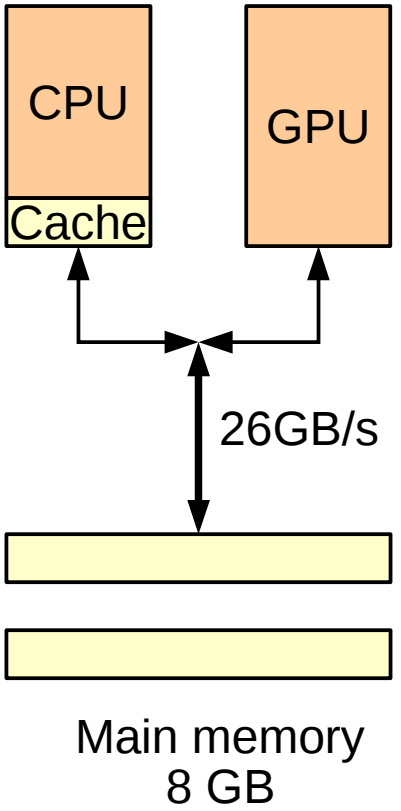


Ex: Intel Core i7 4770, Nvidia GeForce GTX 780

External memory: embedded GPU

Most GPUs today

- Same memory
- May support memory coherence
  - GPU can read directly from CPU caches
- More contention on external memory



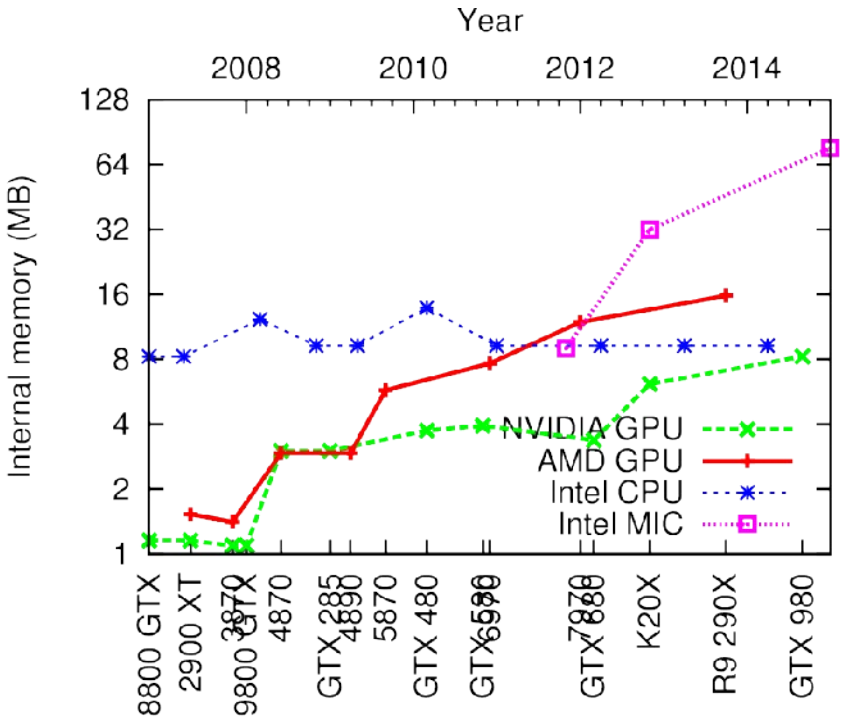
GPU: on-chip memory

- Conventional wisdom
  - Cache area in CPU vs. GPU according to the NVIDIA CUDA Programming Guide:

Figure 1-2. The GPU Devotes More Transistors to Data Processing

- But... if we include registers:

GPU	Register files + caches
NVIDIA GM204 GPU	8.3 MB
AMD Hawaii GPU	15.8 MB
Intel Core i7 CPU	9.3 MB



- GPU/accelerator internal memory exceeds desktop CPUs

Registers: CPU vs. GPU

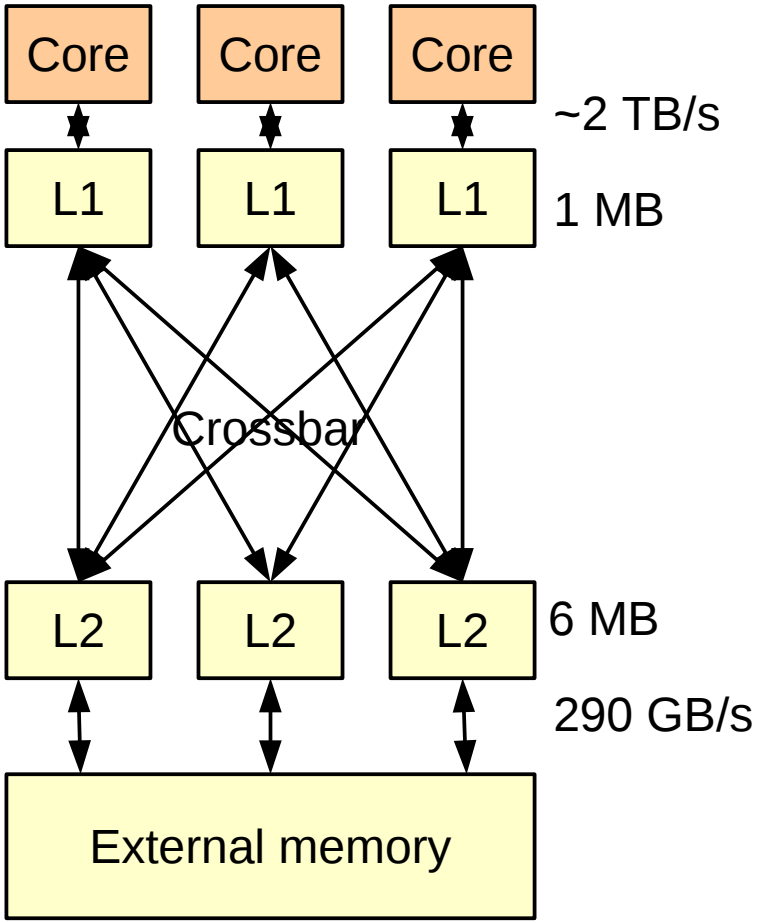
- Registers keep the contents of local variables
- Typical values

	CPU	GPU
Registers/thread	32	32
Registers/core	256	65536
Read / Write ports	10R/5W	2R/1W

- GPU: many more registers, but made of simpler memory

Internal memory: GPU

- Cache hierarchy
  - Keep frequently-accessed data
  - Reduce throughput demand on main memory
  - Managed by hardware (L1, L2) or software (shared memory)



Caches: CPU vs. GPU

	CPU	GPU
Latency	Caches, prefetching	Multi-threading
Throughput		Caches

- On CPU, caches are designed to avoid memory latency
  - Throughput reduction is a side effect
- On GPU, multi-threading deals with memory latency
  - Caches are used to improve throughput (and energy)

GPU: thousands of cores?

- Computational resources

NVIDIA GPUs	G80/G92 (2006)	GT200 (2008)	GF100 (2010)	GK104 (2012)	GK110 (2012)	GM204 (2014)
Exec. units	128	240	512	1536	2688	2048
SM	16	30	16	8	14	16

AMD GPUs	R600 (2007)	R700 (2008)	Evergreen (2009)	NI (2010)	SI (2012)	VI (2013)
Exec. Units	320	800	1600	1536	2048	2560
SIMD-CU	4	10	20	24	32	40

- Number of clients in interconnection network (cores) stays limited

Takeaway

- Result of many tradeoffs
  - Between locality and parallelism
  - Between core complexity and interconnect complexity
- GPU optimized for throughput
  - Exploits primarily DLP, TLP
  - Energy-efficient on parallel applications with regular behavior
- CPU optimized for latency
  - Exploits primarily ILP
  - Can use TLP and DLP when available

Next time

- Next Tuesday, 1:00pm, room 2014
  CUDA
  - Execution model
  - Programming model
  - API
- Thursday 1:00pm, room 2011
  Lab work: what is my GPU and when should I use it?
  - There may be available seats even if you are not enrolled