
Parallel programming: Introduction to GPU architecture

Caroline Collange
she/her

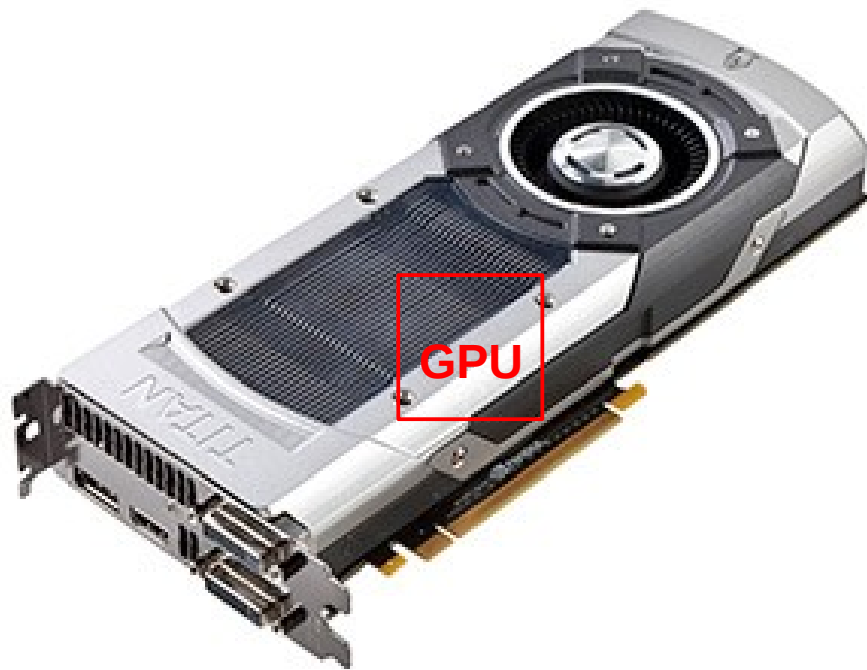
caroline.collange@inria.fr
<https://team.inria.fr/pacap/members/collange/>

Master 1
PPAR - 2022

Outline of the course

- February 27: Introduction to GPU architecture
 - ◆ Parallelism and how to use it
 - ◆ Performance models
- March 7: GPU programming
 - ◆ The software side
 - ◆ Programming model
- March 14: Performance optimization
 - ◆ Possible bottlenecks
 - ◆ Common optimization techniques
- 4 lab sessions, March 8, 15, 22/24, 29
 - ◆ Labs 1&2: computing $\log(2)$ the hard way
 - ◆ Labs 3&4: yet another Conway's Game of Life

Graphics processing unit (GPU)



or



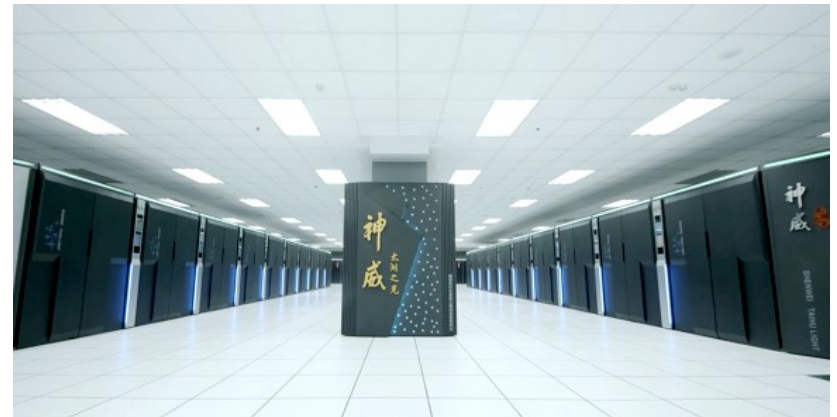
- Graphics rendering accelerator for computer games
 - ♦ Mass market: low unit price, amortized R&D
 - ♦ Increasing programmability and flexibility
- Inexpensive, high-performance parallel processor
 - ♦ GPUs are everywhere, from cell phones to supercomputers
- ➔ General-Purpose computation on GPU (GPGPU)

GPUs in high-performance computing

- GPU/accelerator share in Top500 supercomputers
 - ◆ In 2010: 2%
 - ◆ In 2020: 30%
- 2016+ trend:
Heterogeneous multi-core processors influenced by GPUs



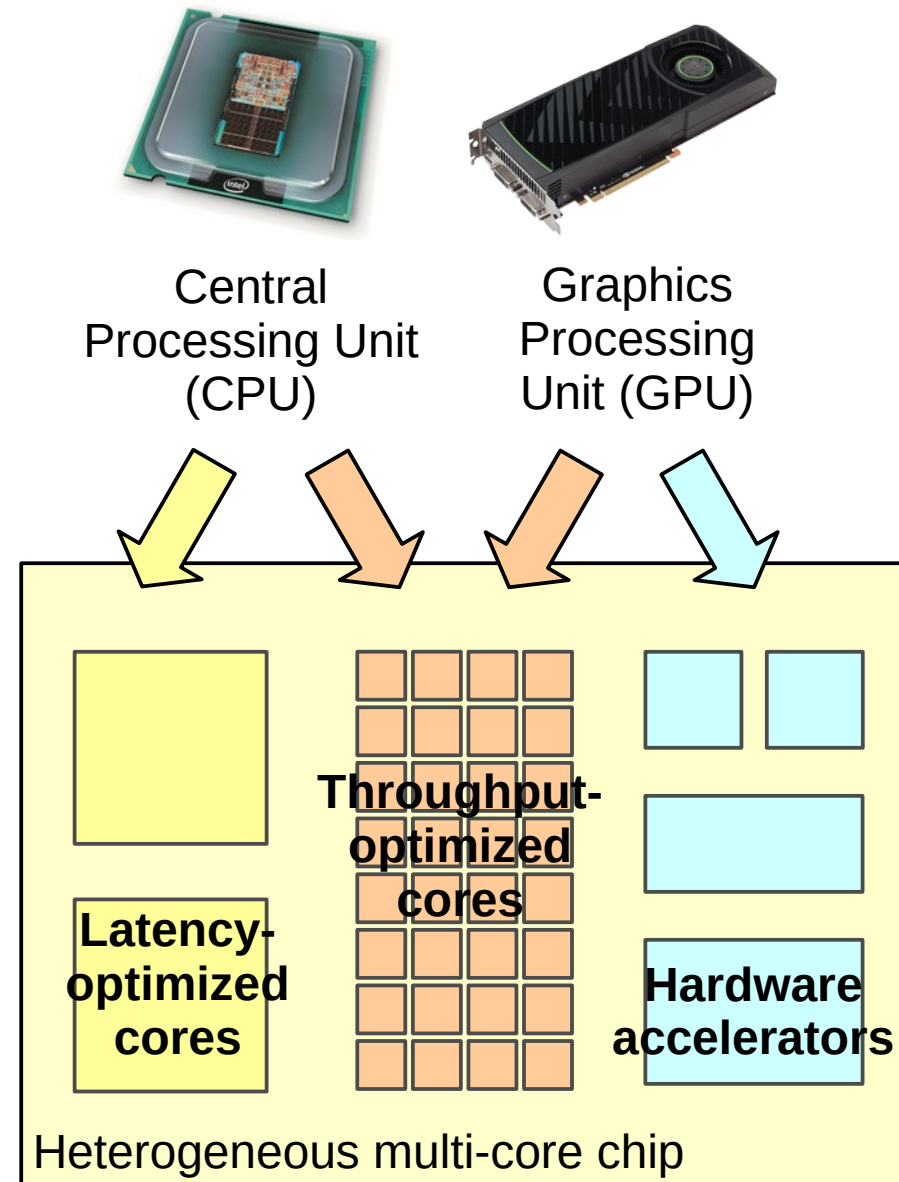
#2 Summit (USA)
4,608 × (2 Power9 CPUs + 6 Volta GPUs)



#4 Sunway TaihuLight (China)
40,960 × SW26010 (4 big + 256 small cores)

GPUs in the future?

- Yesterday (2000-2010)
 - ♦ Homogeneous multi-core
 - ♦ Discrete components
- Today (2011-...)
Chip-level integration
 - ♦ CPU cores and GPU cores on the same chip
 - ♦ Still different programming models, software stacks
- Tomorrow
Heterogeneous multi-core
 - ♦ GPUs to blend into throughput-optimized, general purpose cores?

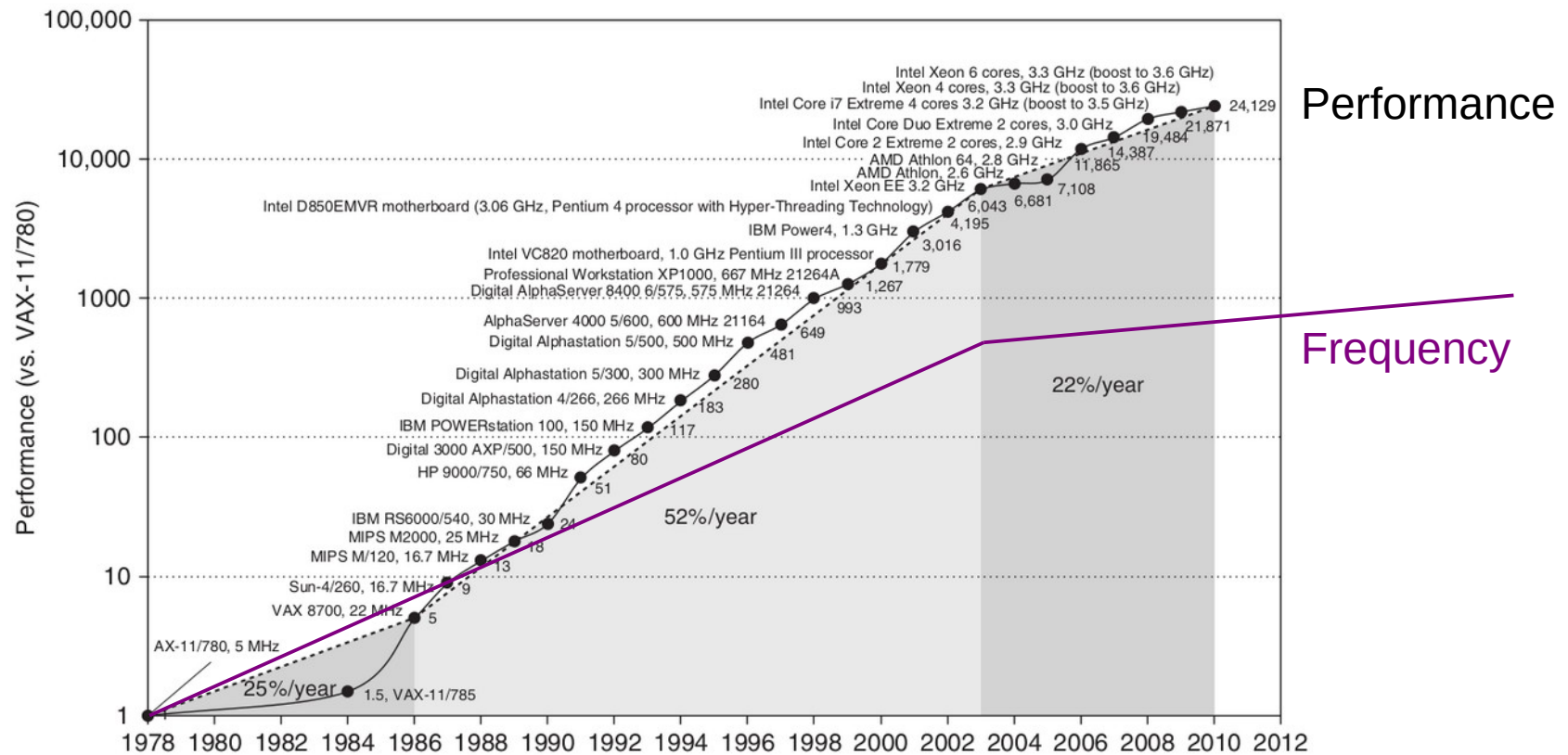


Outline

- GPU, many-core: why, what for?
 - ◆ Technological trends and constraints
 - ◆ From graphics to general purpose
 - ◆ Hardware trends
- 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

Last century: the free lunch era

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

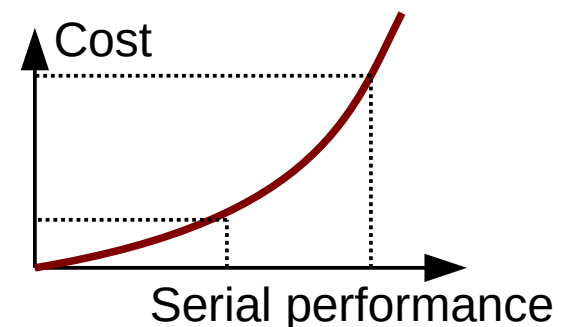
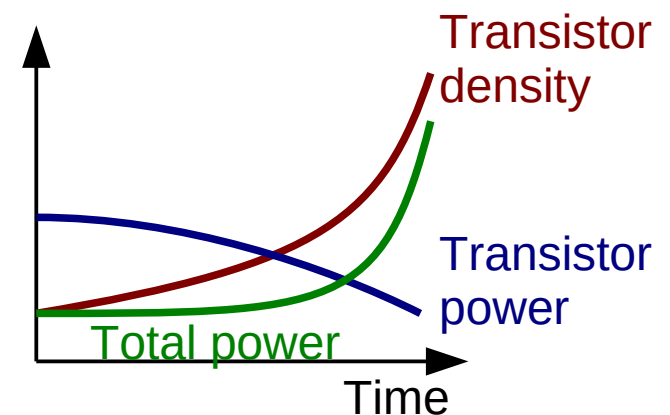
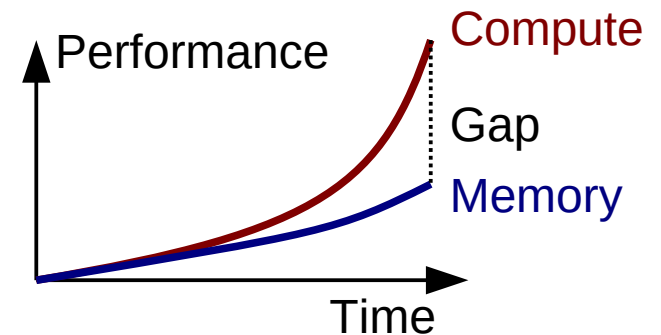


Hennessy, Patterson. Computer Architecture, a quantitative approach. 5th Ed. 2010

- Do not rewrite software, buy a new machine!

Technology evolution

- Memory wall
 - ◆ Memory speed does not increase as fast as computing speed
 - ◆ Harder 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: $\text{cost} \approx \text{performance}^2$



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
 - ♦ And incidentally, climate crisis!



Latency vs. throughput

- **Latency**: time to solution
 - ◆ Minimize time, at the expense of power
 - ◆ Metric: time
e.g. seconds
- **Throughput**: quantity of tasks processed per unit of time
 - ◆ Assumes unlimited parallelism
 - ◆ Minimize energy per operation
 - ◆ Metric: operations / time
e.g. Gflops / s
- CPU: optimized for latency
- GPU: optimized for throughput



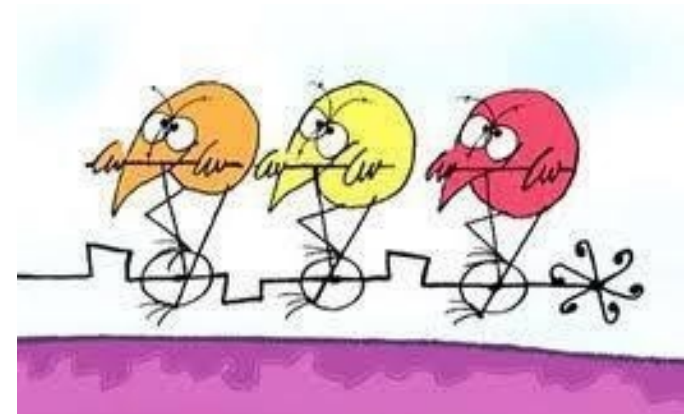
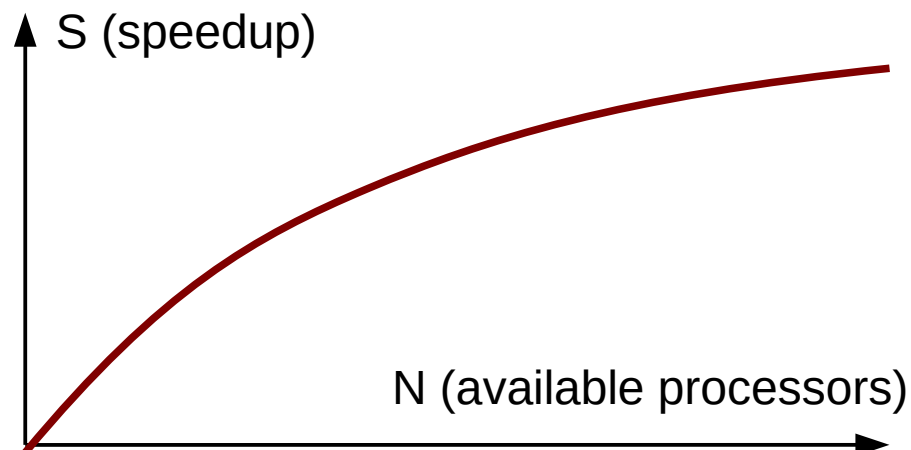
What is the difference between Latency and 1/Throughput?

Amdahl's law

- Bounds speedup attainable on a parallel machine

Time to run sequential portions \rightarrow $S = \frac{1}{(1 - P) + \frac{P}{N}}$ \leftarrow Time to run parallel portions

S Speedup
 P Ratio of parallel portions
 N Number of processors



"The more you are, the least faster you go"

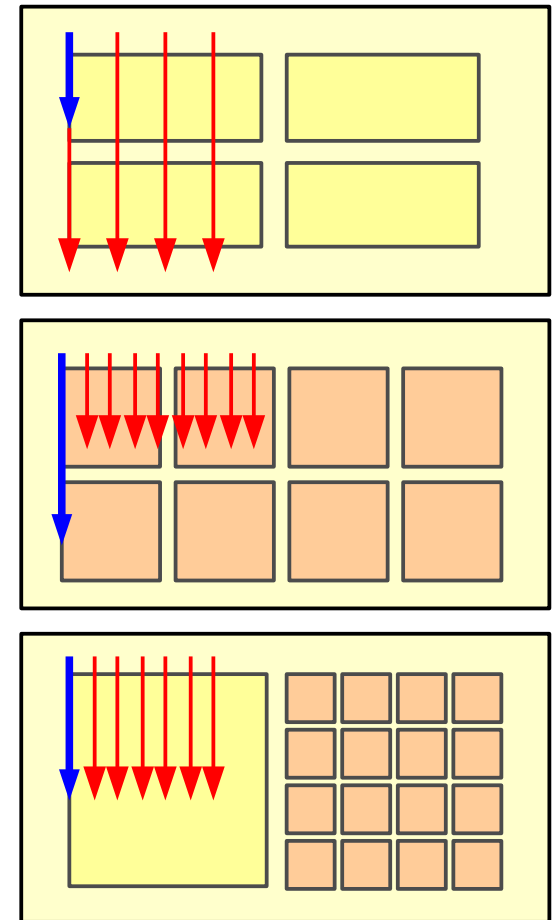
Why heterogeneous architectures?

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

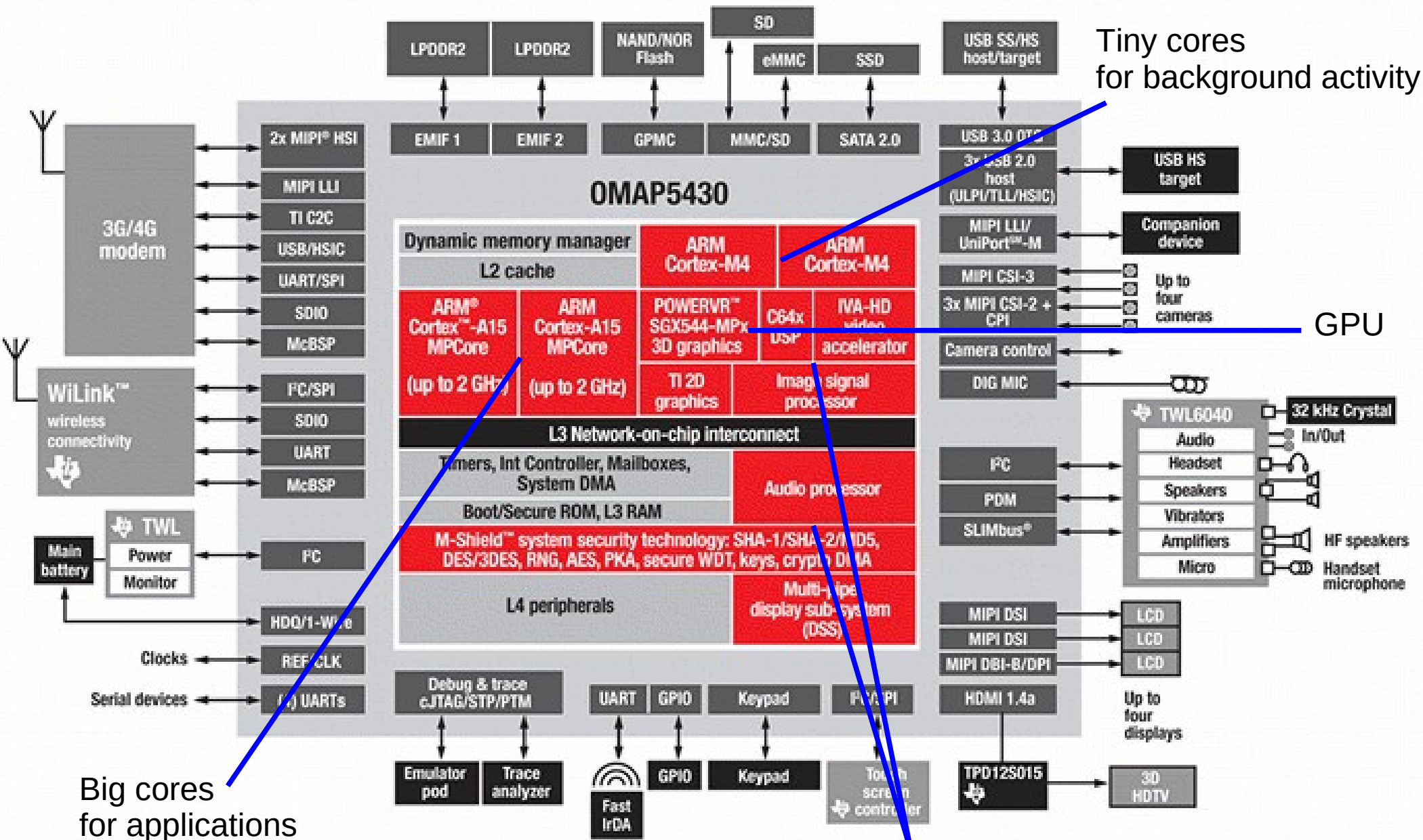
Time to run sequential portions \rightarrow $(1-P)$

$\frac{P}{N}$ \leftarrow 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



Example: System on Chip for smartphone



Tiny cores
for background activity

GPU

Big cores
for applications

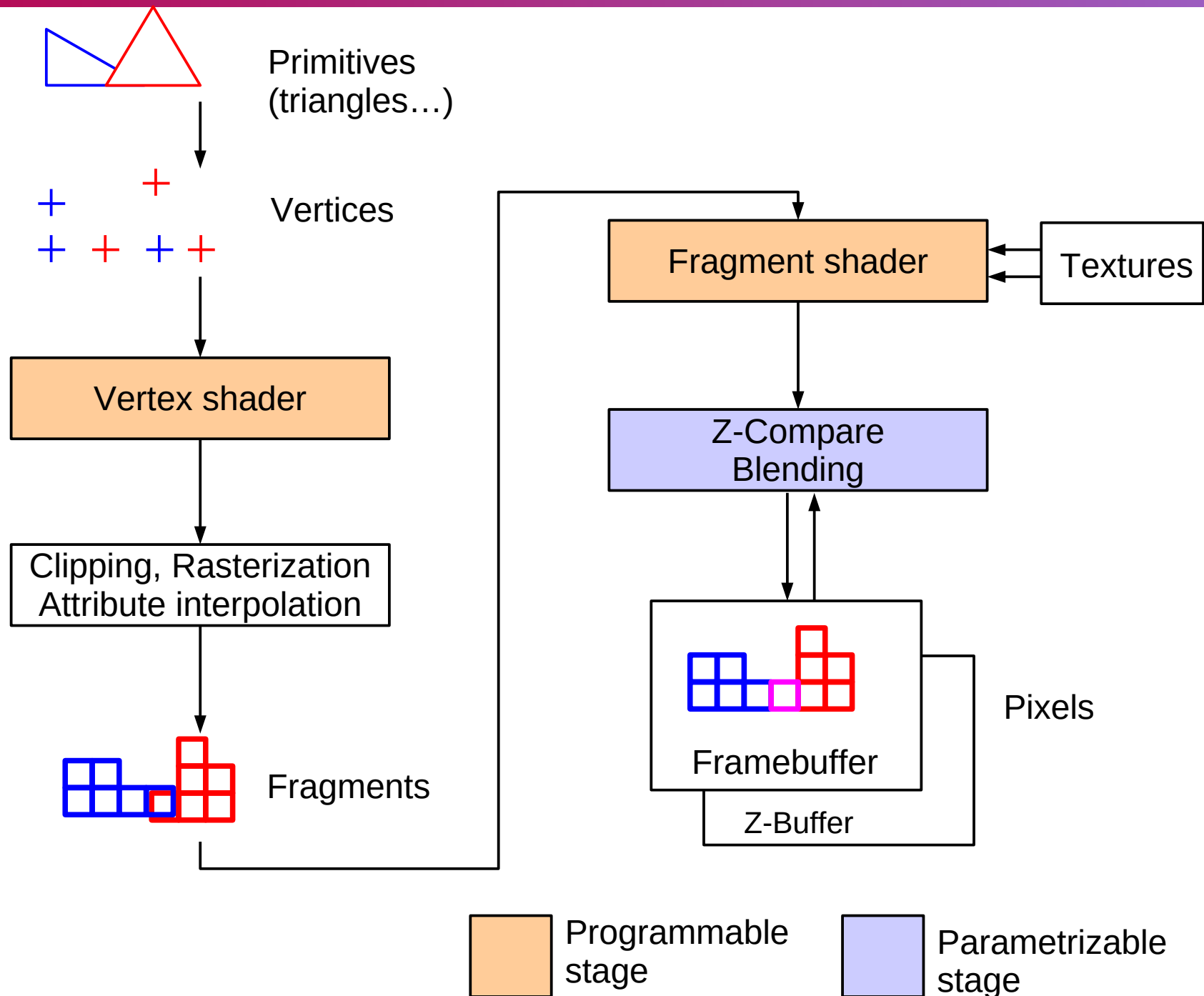
Lots of interfaces

Special-purpose
accelerators

Outline

- GPU, many-core: why, what for?
 - ◆ Technological trends and constraints
 - ◆ From graphics to general purpose
 - ◆ Hardware trends
- 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

Aside: GPU as an out-of-order pipeline

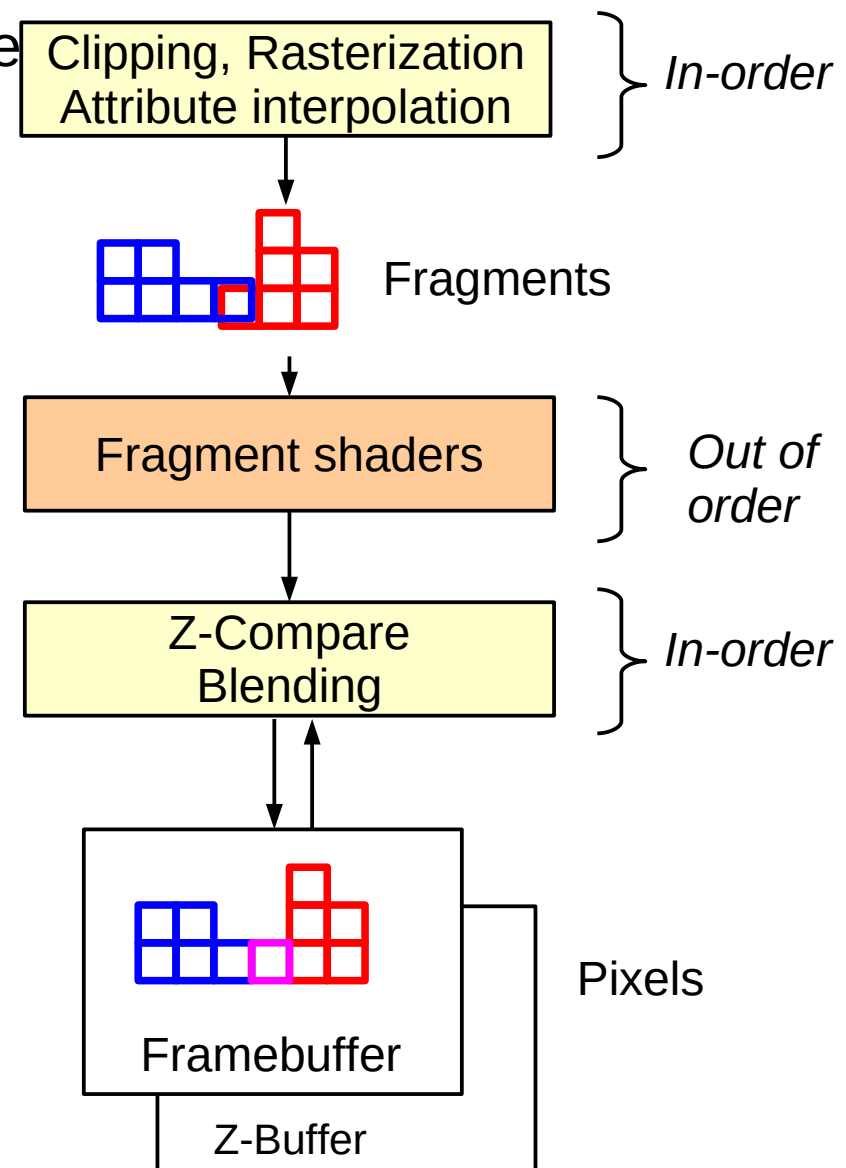
- Graphics APIs demand that primitives are drawn in submission order

- e.g. back-to-front rendering

- Shaders proceed out of order

- 10 000s fragments in flight
 - Shaders render fragments out of order
 - Raster ops put fragments back in order for framebuffer update
 - Various binning and tiling techniques to identify independent fragments

- General-purpose compute pipeline is much **simpler** than graphics pipeline



GPGPU: General-Purpose computation on GPUs

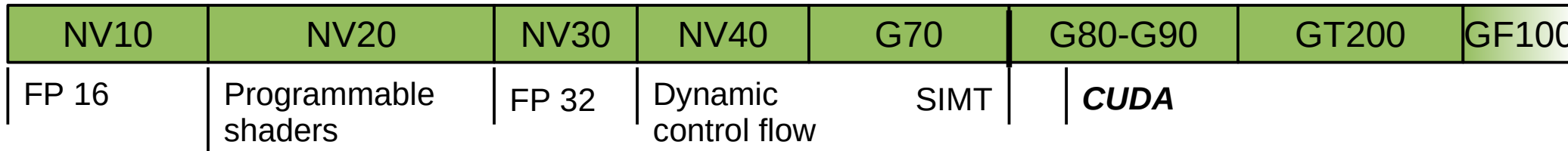
Early 2000: software and hardware evolution enable general-purpose programming

Microsoft DirectX

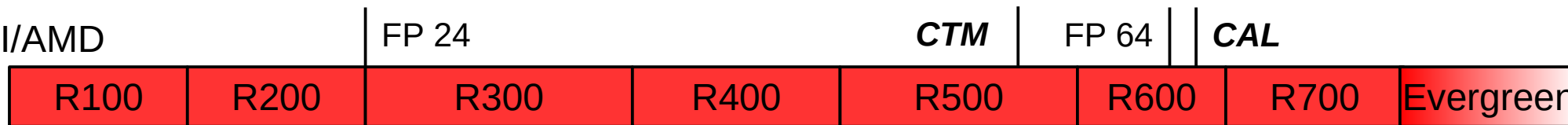


Unified shaders

NVIDIA



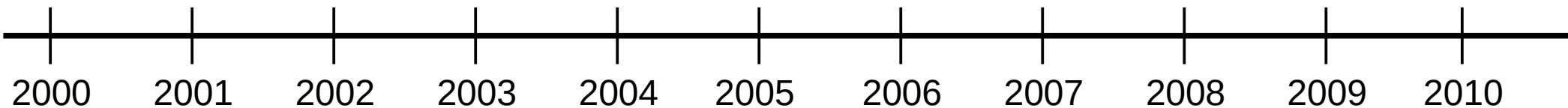
ATI/AMD



Mark Harris coins
"GPGPU"

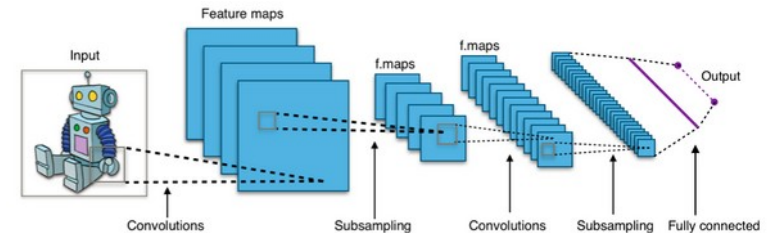
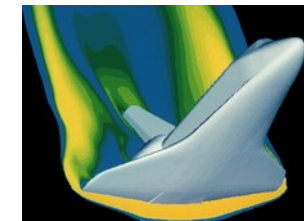
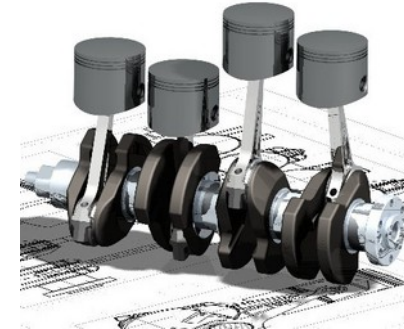
GPUs in #1
supercomputer

GPGPU traction



Today: what do we need GPUs for?

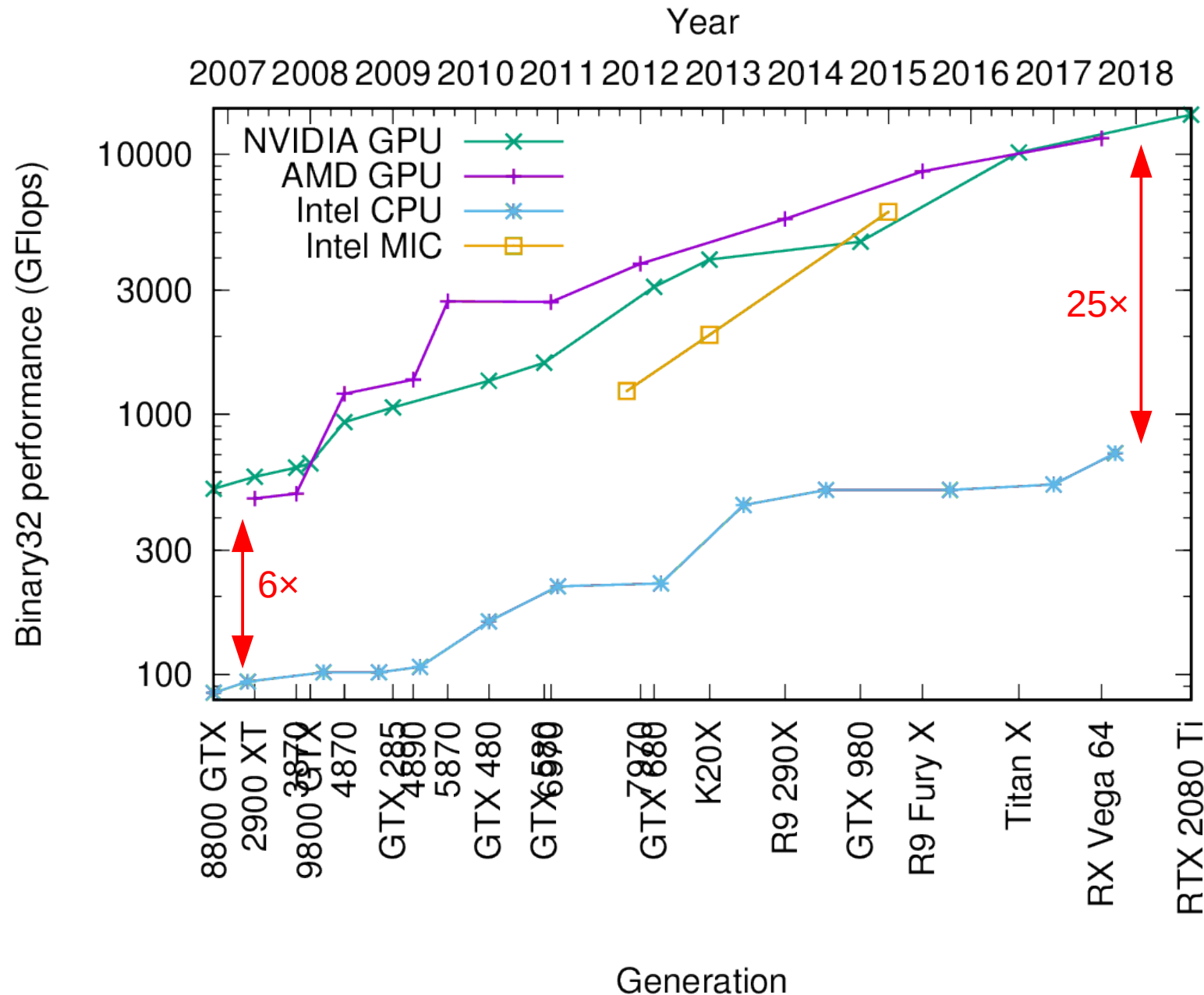
1. 3D graphics rendering for games
 - ❖ Complex texture mapping, lighting computations...
 2. Computer Aided Design workstations
 - ❖ Complex geometry
 3. High-performance computing
 - ❖ Complex synchronization, off-chip data movement, high precision
 4. Convolutional neural networks
 - ❖ Complex scheduling of low-precision linear algebra
- One chip to rule them all
 - ➔ Find the common denominator



Outline

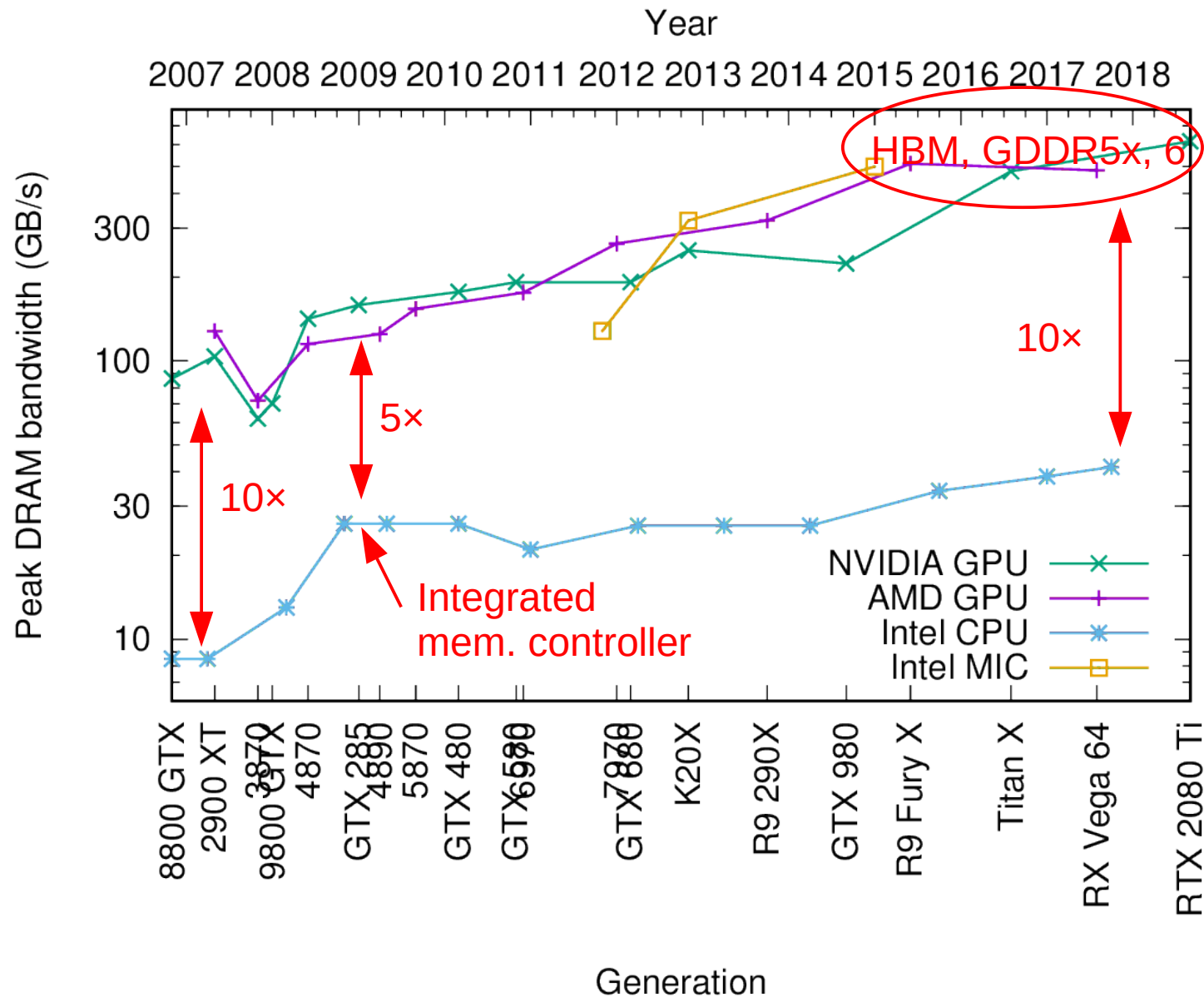
- GPU, many-core: why, what for?
 - ◆ Technological trends and constraints
 - ◆ From graphics to general purpose
 - ◆ Hardware trends
- Forms of parallelism, how to exploit them
 - ◆ Why we need (so much) parallelism: latency and throughput
 - ◆ Sources of parallelism: ILP, TLP, DLP
 - ◆ Uses of parallelism: 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

Trends: compute performance

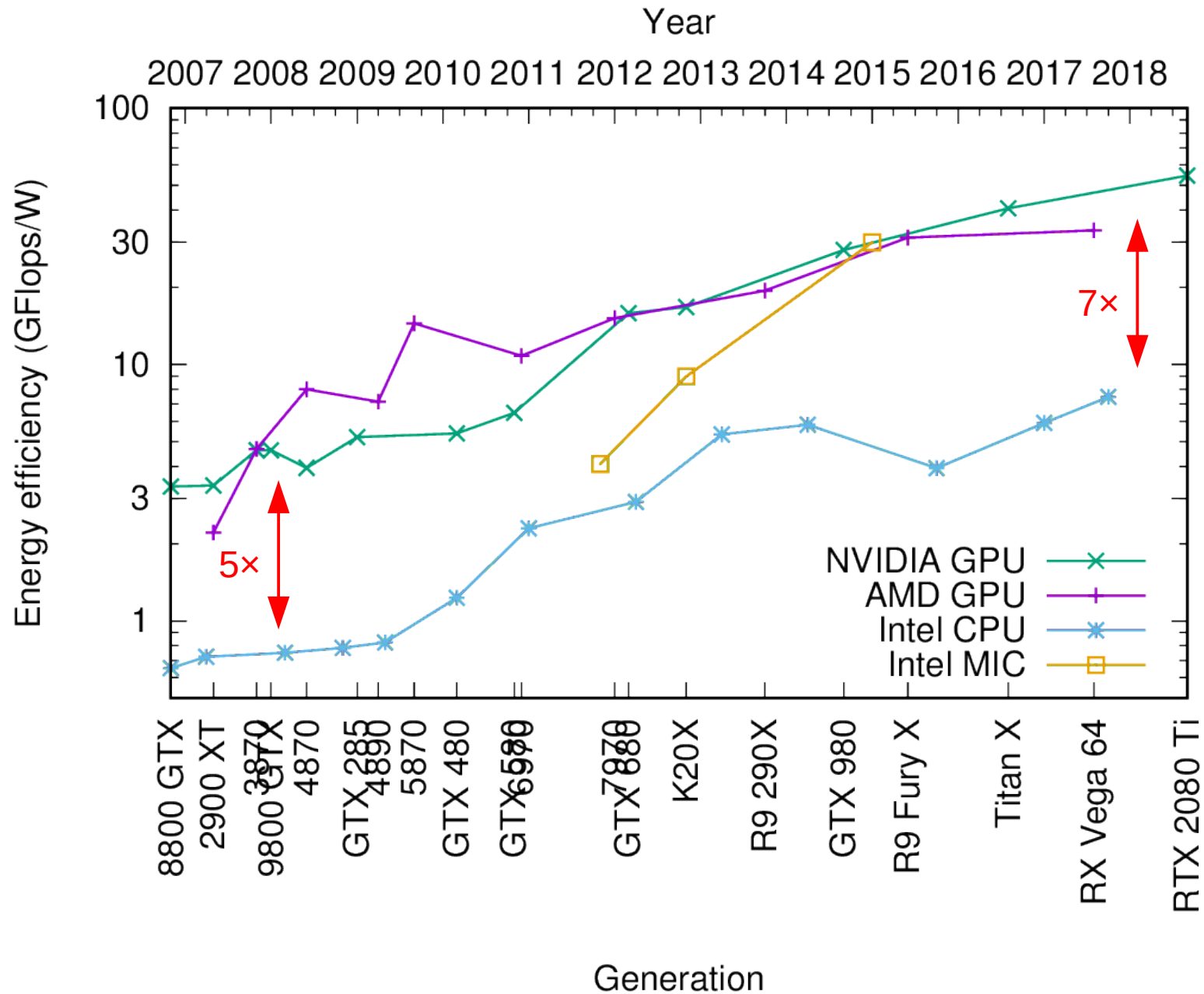


Caveat: only considers **desktop** CPUs. Gap with server CPUs is “only” 4x!

Trends: memory bandwidth



Trends: energy efficiency



Outline

- GPU, many-core: why, what for?
 - ◆ Technological trends and constraints
 - ◆ From graphics to general purpose
 - ◆ Hardware trends
- 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

What is parallelism?

Parallelism: independent operations which execution can be overlapped
Operations: memory accesses or computations

How much parallelism do I need?

- Little's law in queuing theory

- ♦ Average customer arrival rate λ

- ♦ Average time spent W

← throughput

- ♦ Average number of customers

← latency

$$L = \lambda \times W$$

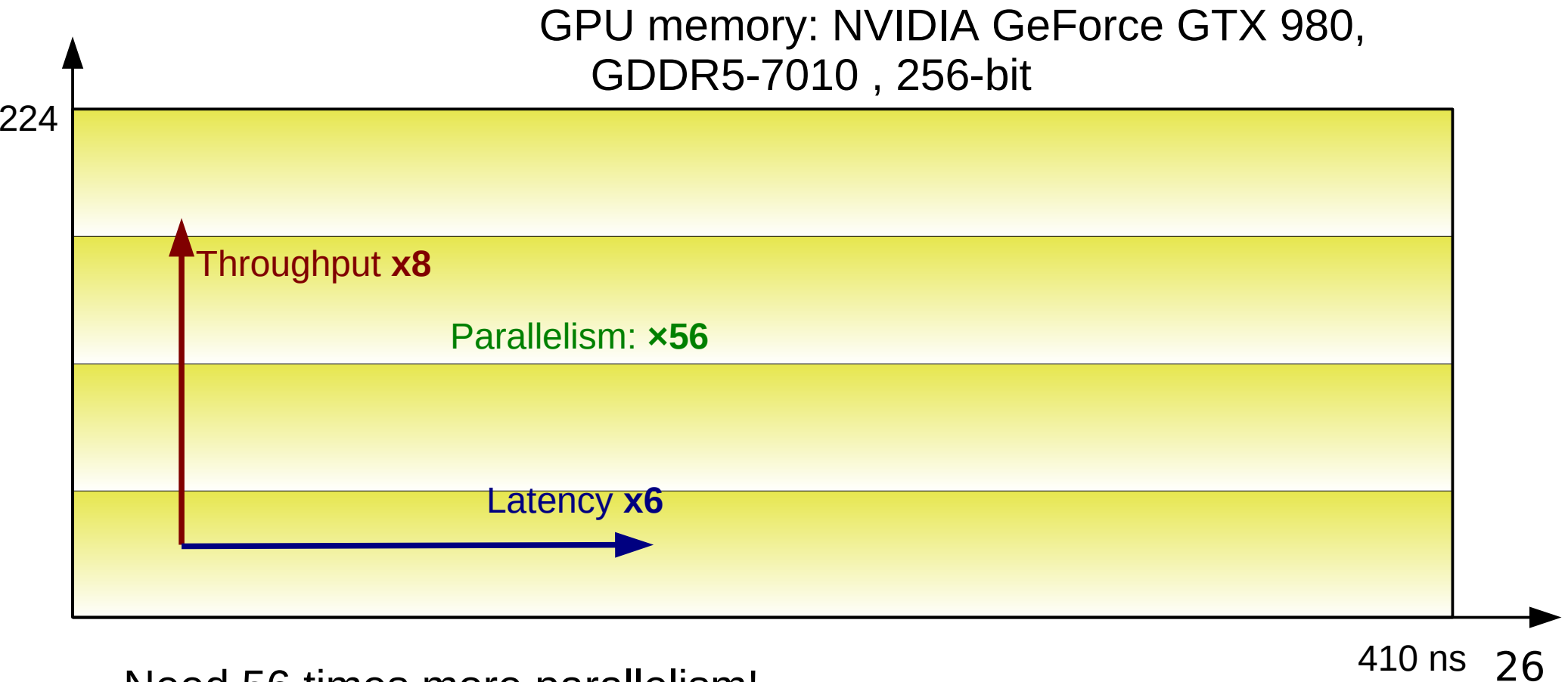
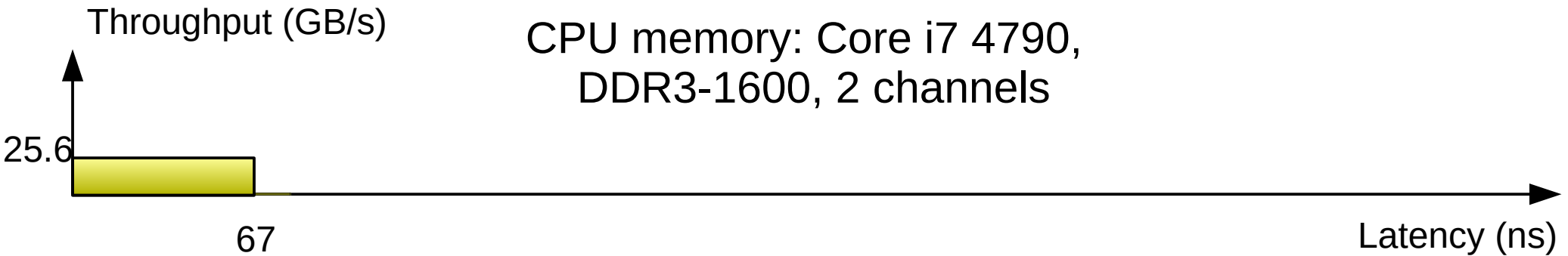
← **Parallelism** = throughput \times latency

- Units

- ♦ For memory: $B = \text{GB/s} \times \text{ns}$

- ♦ For arithmetic: $\text{flops} = \text{Gflops/s} \times \text{ns}$

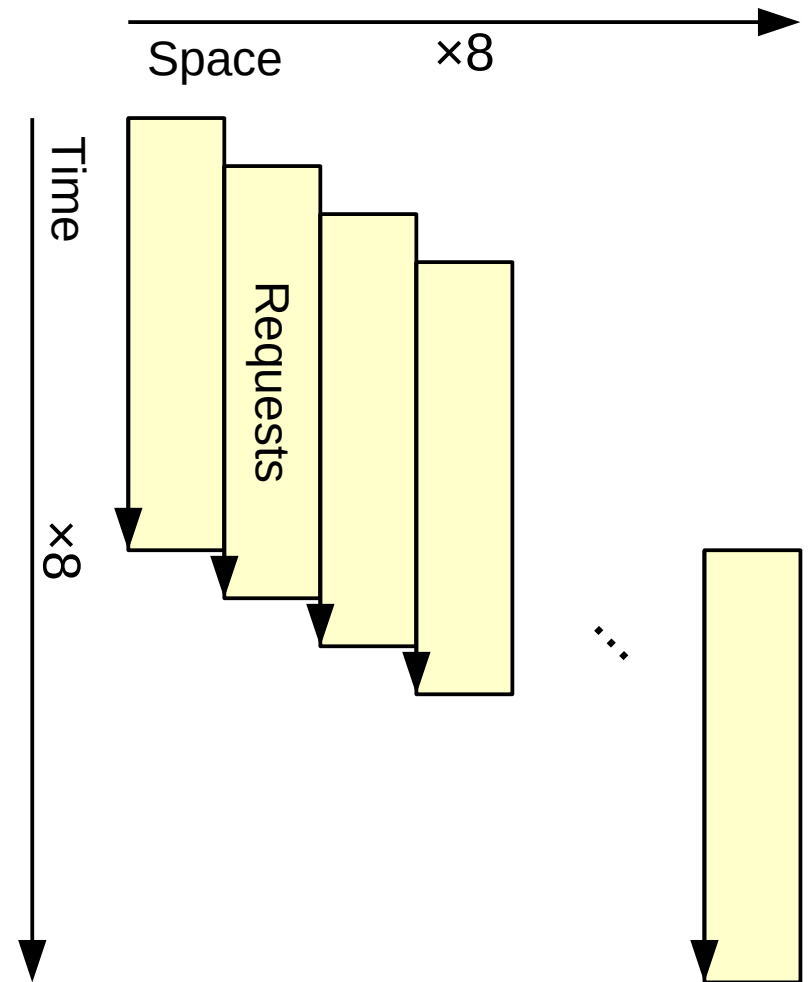
Throughput and latency: CPU vs. GPU



→ Need 56 times more parallelism!

Consequence: more parallelism

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



Sources of parallelism

- ILP: Instruction-Level Parallelism

- Between independent instructions in sequential program

```
add  r3 ← r1, r2  
mul  r0 ← r0, r1  
sub  r1 ← r3, r0
```

} Parallel

- TLP: Thread-Level Parallelism

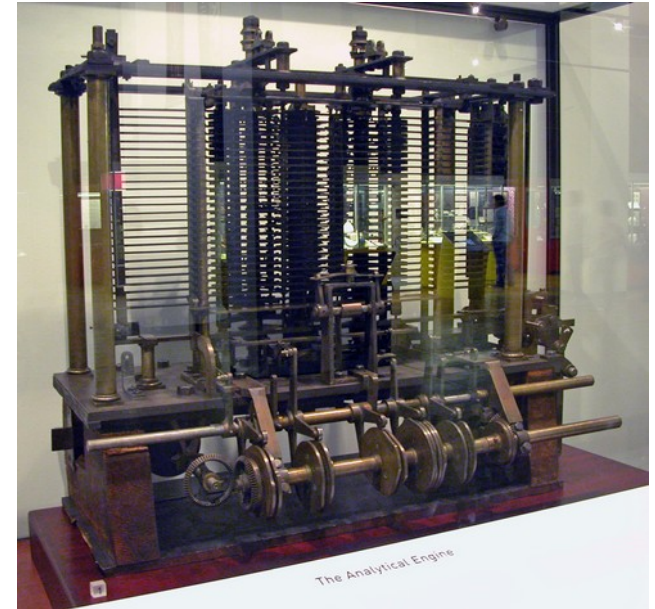
- Between independent execution contexts: threads

Thread 1 Thread 2

```
( add    mul ) Parallel
```

Avoiding instruction redundancy

- 1840s: Charles Babbage's *Analytical Engine* programmed using punched cards
 - ◆ Instructions : *operation cards*
 - ◆ Operands : *variable cards*
- One instruction (e.g. multiplication) can operate on multiple data
 - ◆ Ada Lovelace:
“The **same operation** would be performed on different **subjects of operation**.”
 - ◆ (Vectors had not been invented yet)



L. F. Menabrea and A. A. Lovelace. *Sketch of the Analytical Engine invented by Charles Babbage*. 1842

Sources of parallelism

- ILP: Instruction-Level Parallelism

- Between independent instructions in sequential program

```

add  r3 ← r1, r2
mul  r0 ← r0, r1
sub  r1 ← r3, r0
    
```

Parallel

- TLP: Thread-Level Parallelism

- Between independent execution contexts: threads

Thread 1 Thread 2

```

( add      mul ) Parallel
    
```

- DLP: Data-Level Parallelism

- Between elements of a vector: same operation on several elements

```

vadd r ← a, b
    
```

a_1	a_2	a_3
$+$	$+$	$+$
b_1	b_2	b_3
<hr/>		
r_1	r_2	r_3

Example: $X \leftarrow a \times X$

- In-place scalar-vector product: $X \leftarrow a \times X$

Sequential (ILP)

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

Threads (TLP)

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

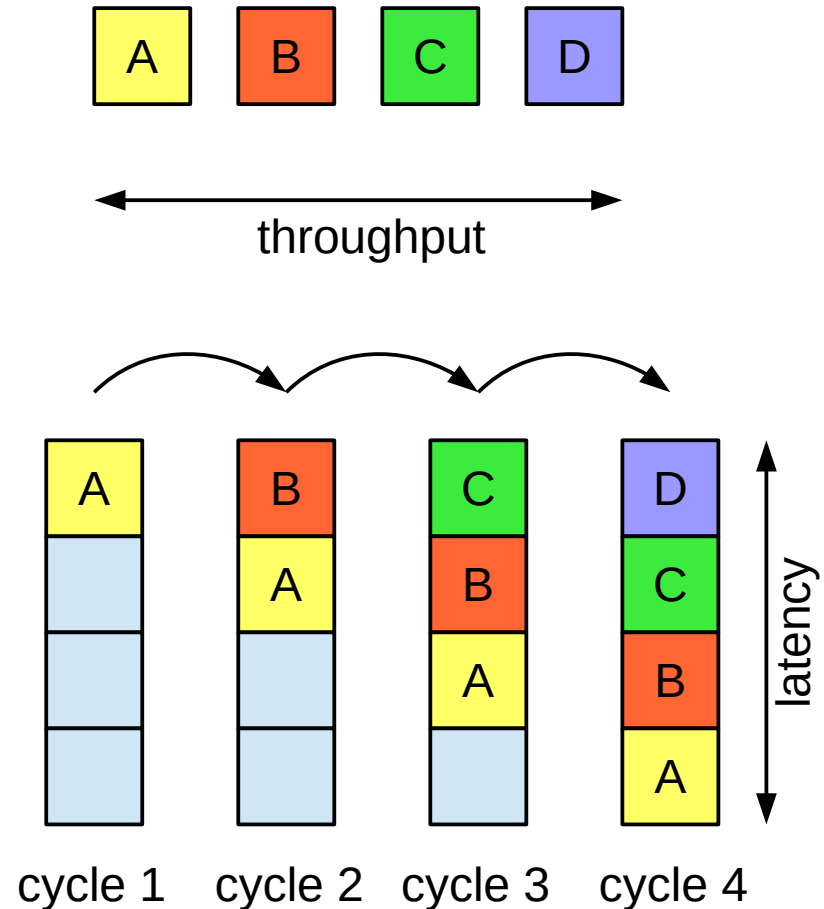
Vector (DLP)

```
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

Break

Outline

- GPU, many-core: why, what for?
 - ◆ Technological trends and constraints
 - ◆ From graphics to general purpose
 - ◆ Hardware trends
- 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

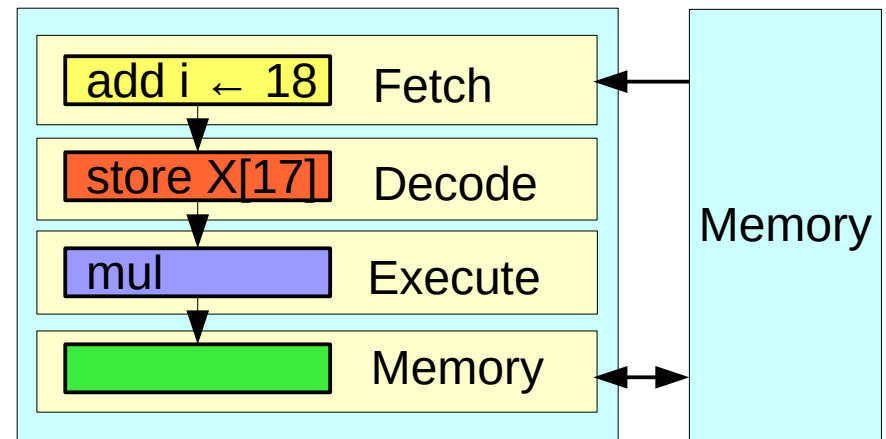
Sequential processor

```
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

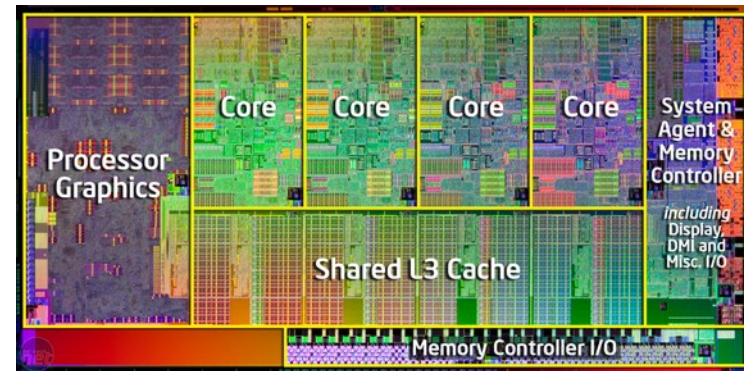


Sequential CPU

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

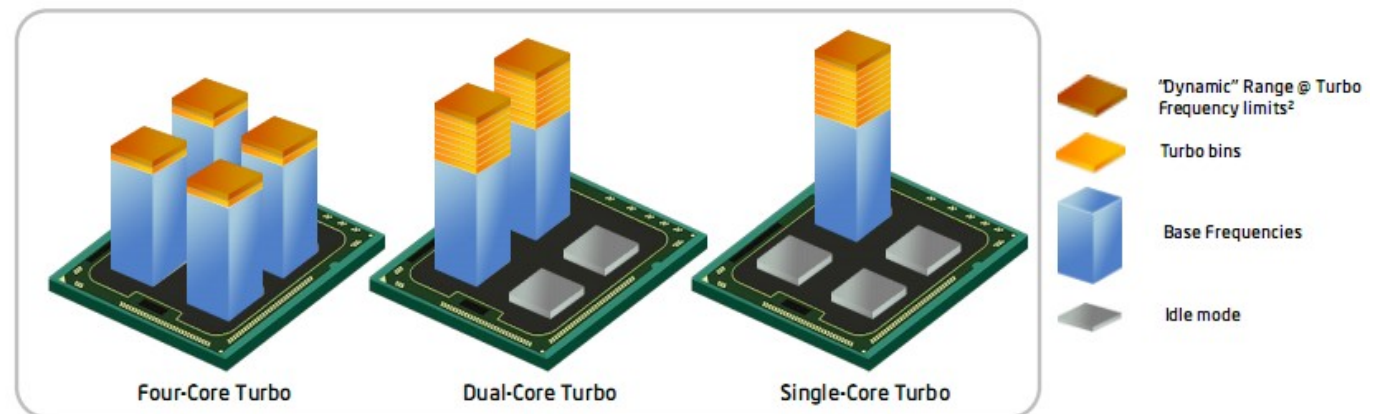
The incremental approach: multi-core

- Several processors on a single chip sharing one memory space



Intel Sandy Bridge

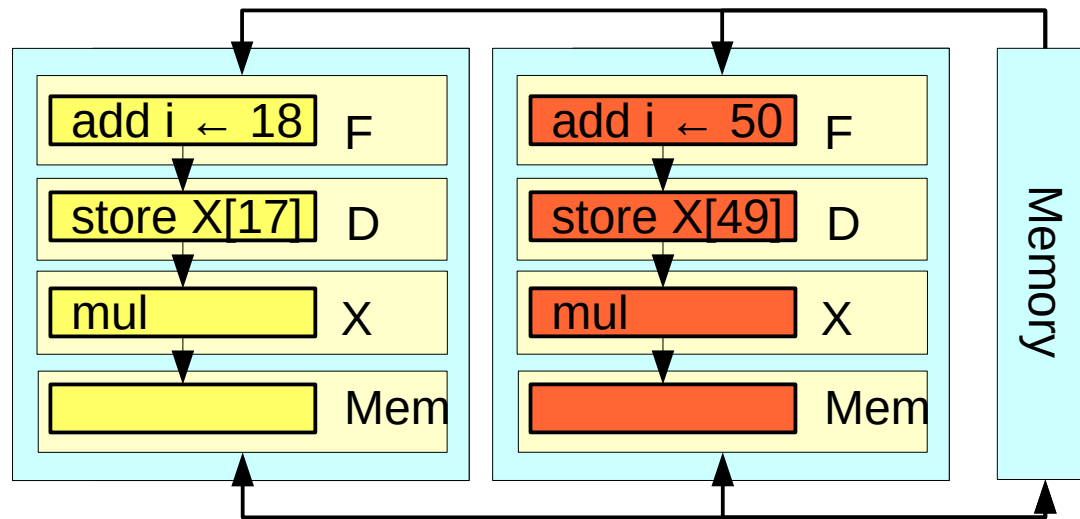
- Area: benefits from Moore's law
- Power: extra cores consume little when not in use
 - ◆ e.g. Intel Turbo Boost



Source: Intel

Homogeneous multi-core

- Horizontal use of thread-level parallelism

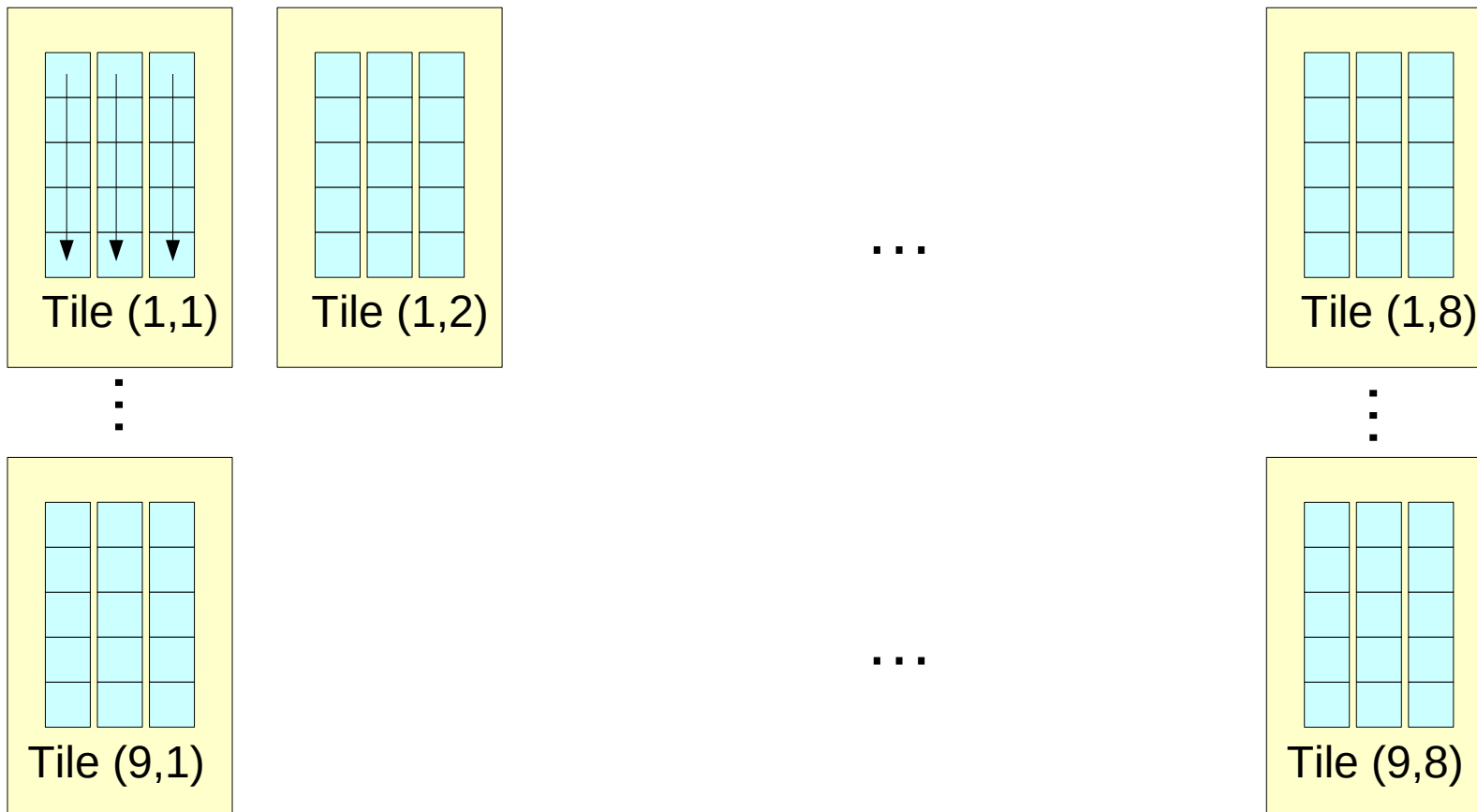


Threads: T0 T1

- Improves peak throughput

Example: Tileria Tile-GX

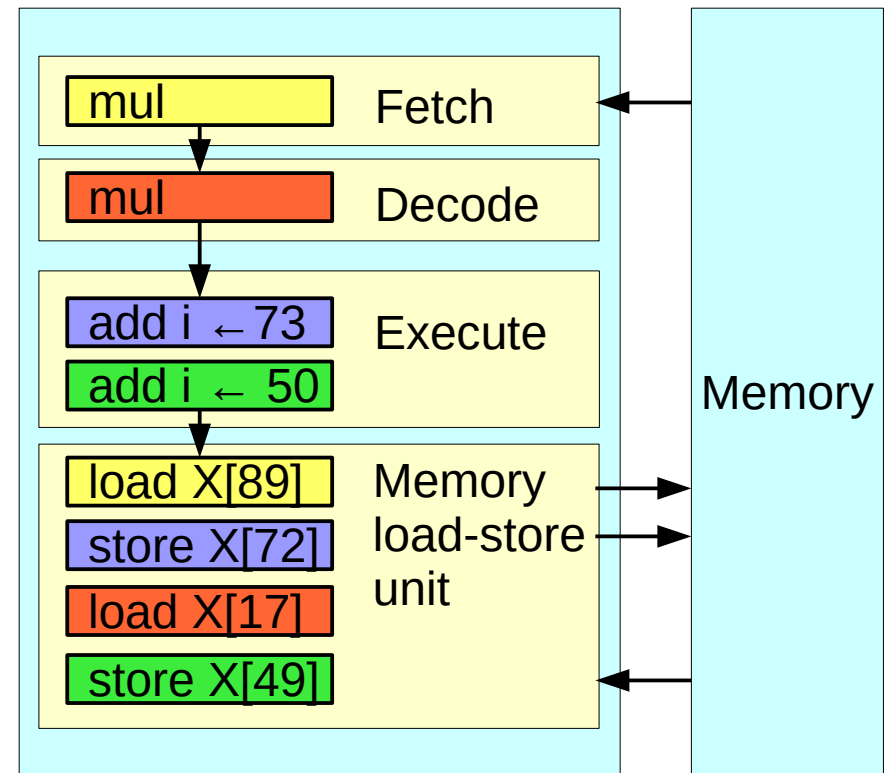
- Grid of (up to) 72 tiles
- Each tile: 3-way VLIW processor, 5 pipeline stages, 1.2 GHz



Interleaved multi-threading

- Vertical use of thread-level parallelism

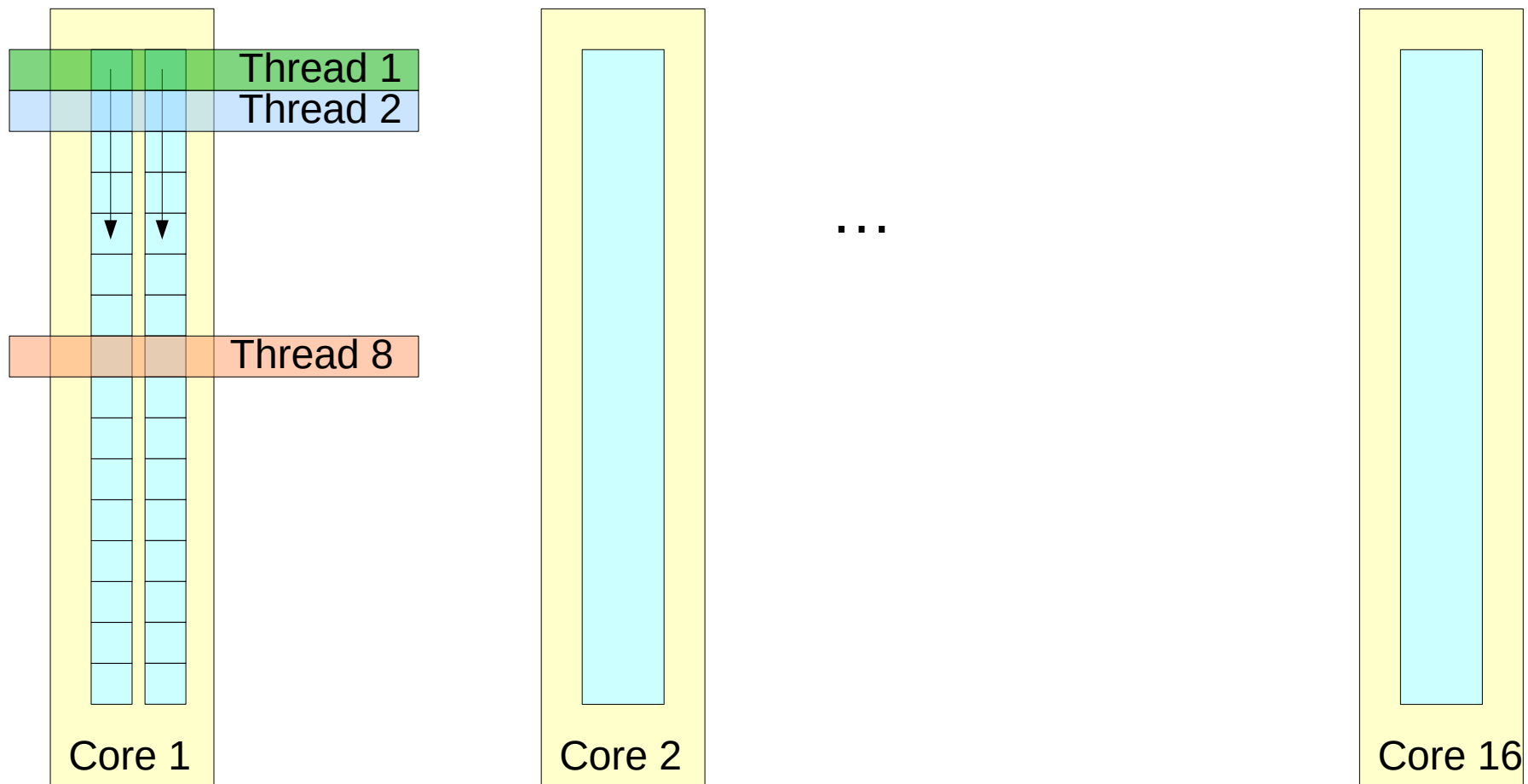
Threads: T0 T1 T2 T3



- Hides latency thanks to explicit parallelism
improves achieved throughput

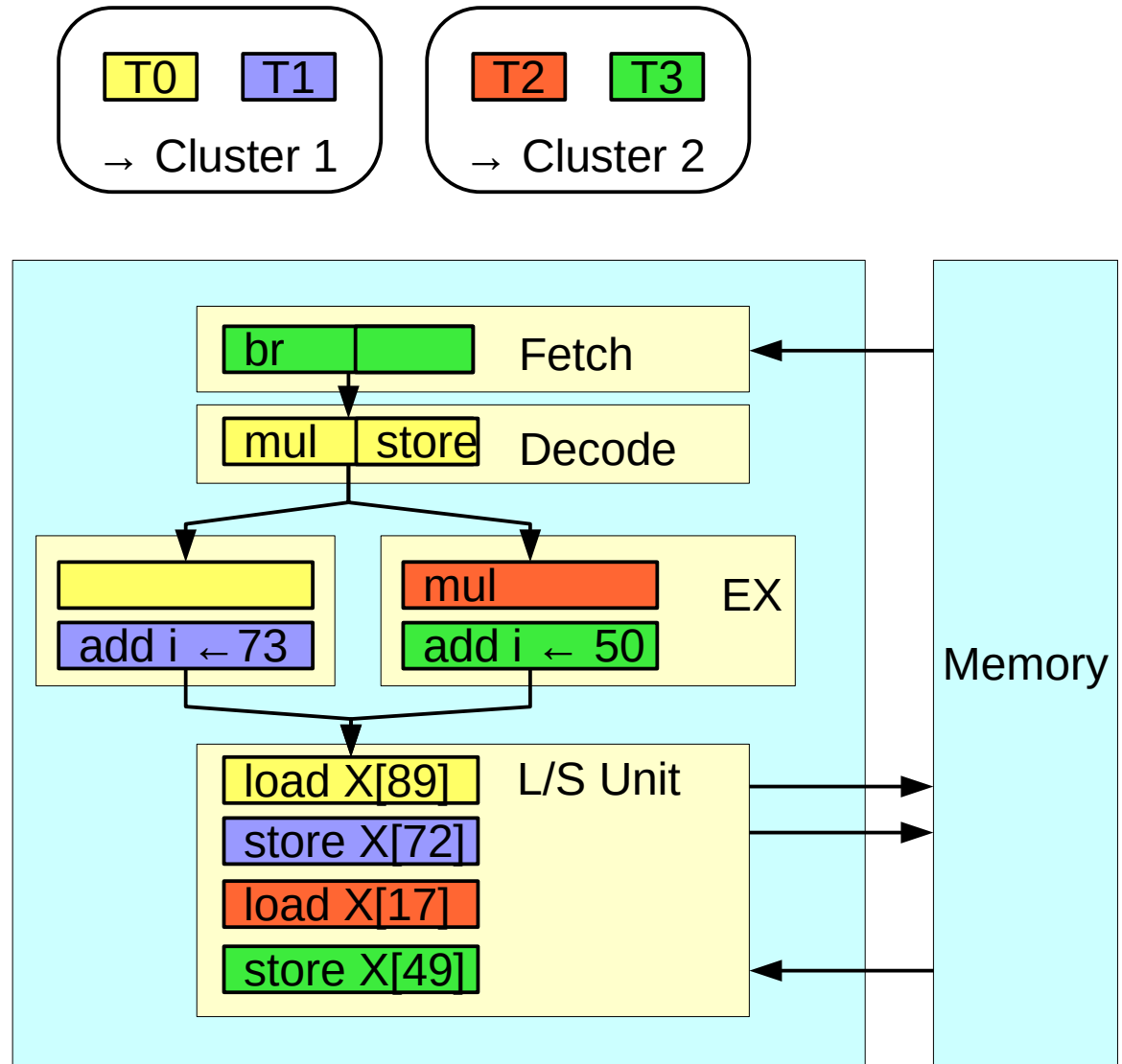
Example: Oracle Sparc T5

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



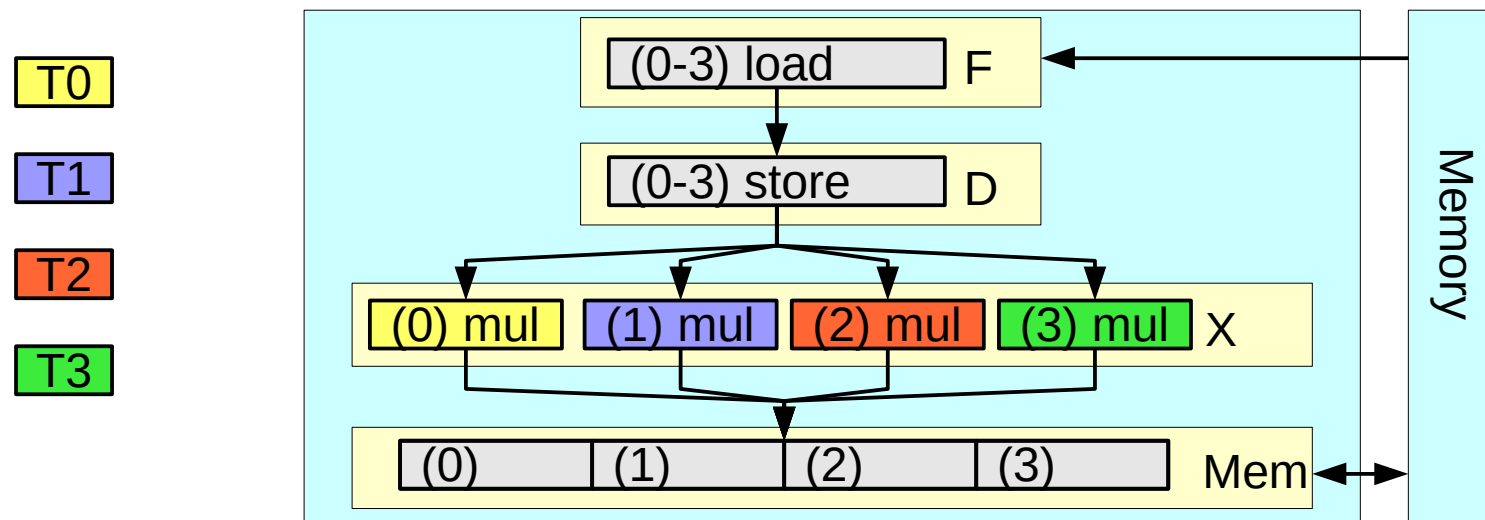
Clustered multi-core

- For each **individual unit**, select between
 - ◆ Horizontal replication
 - ◆ Vertical time-multiplexing
- Examples
 - ◆ Sun UltraSparc T2, T3
 - ◆ AMD Bulldozer
 - ◆ IBM Power 7, 8, 9
- 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

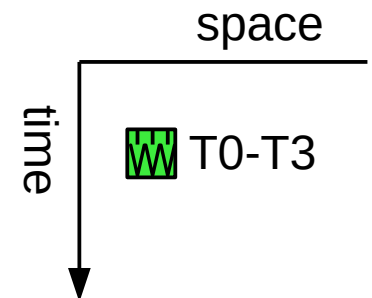
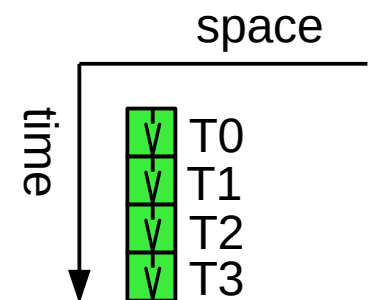
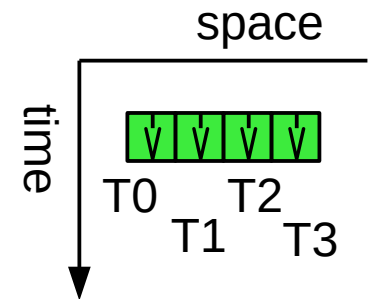


- In NVIDIA-speak
 - ◆ SIMT: Single Instruction, Multiple Threads
 - ◆ Convoy of synchronized threads: *warp*
- Extracts DLP from multi-thread applications

How to exploit common operations?

Multi-threading implementation options:

- Horizontal: replication
 - ♦ **Different** resources, **same** time
 - ♦ Chip Multi-Processing (CMP)
- Vertical: time-multiplexing
 - ♦ **Same** resource, **different** times
 - ♦ Multi-Threading (MT)
- Factorization
 - ♦ **If** we have common operations between threads
 - ♦ **Same** resource, **same** time
 - ♦ Single-Instruction Multi-Threading (SIMT)

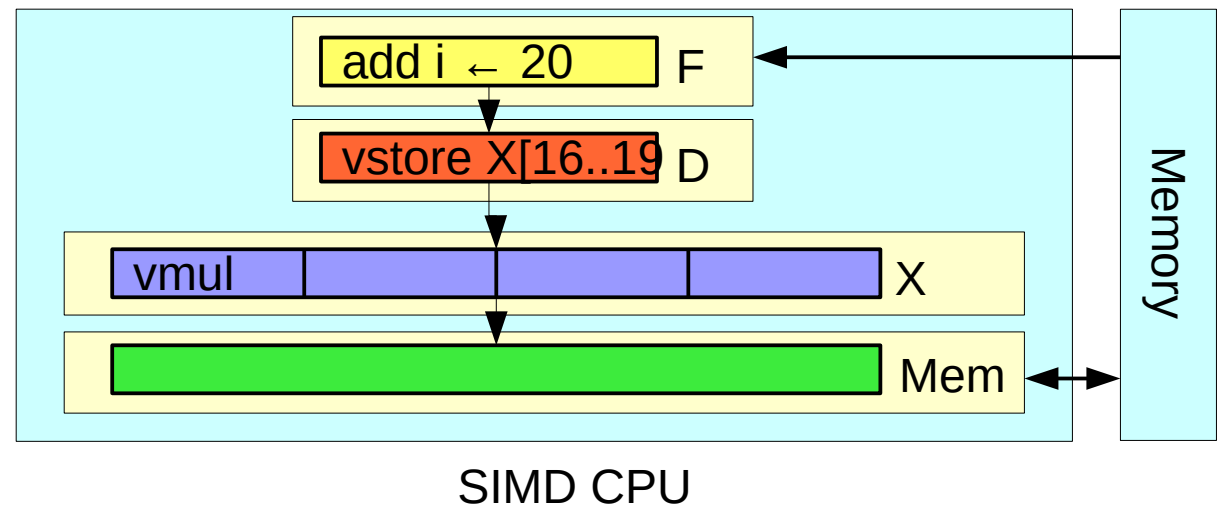


Explicit SIMD

- 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



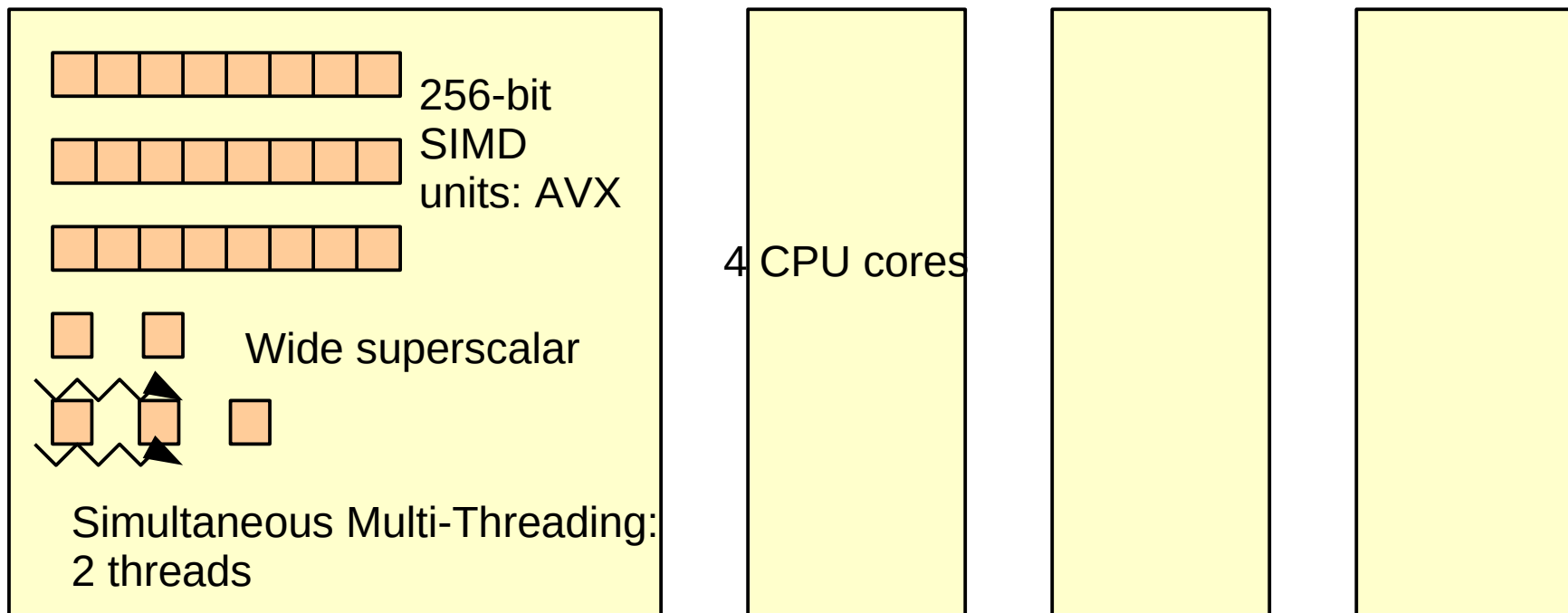
- Examples
 - ◆ Intel MIC (16-wide)
 - ◆ AMD GCN GPU (16-wide×4-deep)
 - ◆ Most general purpose CPUs (4-wide to 16-wide)

Outline

- 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

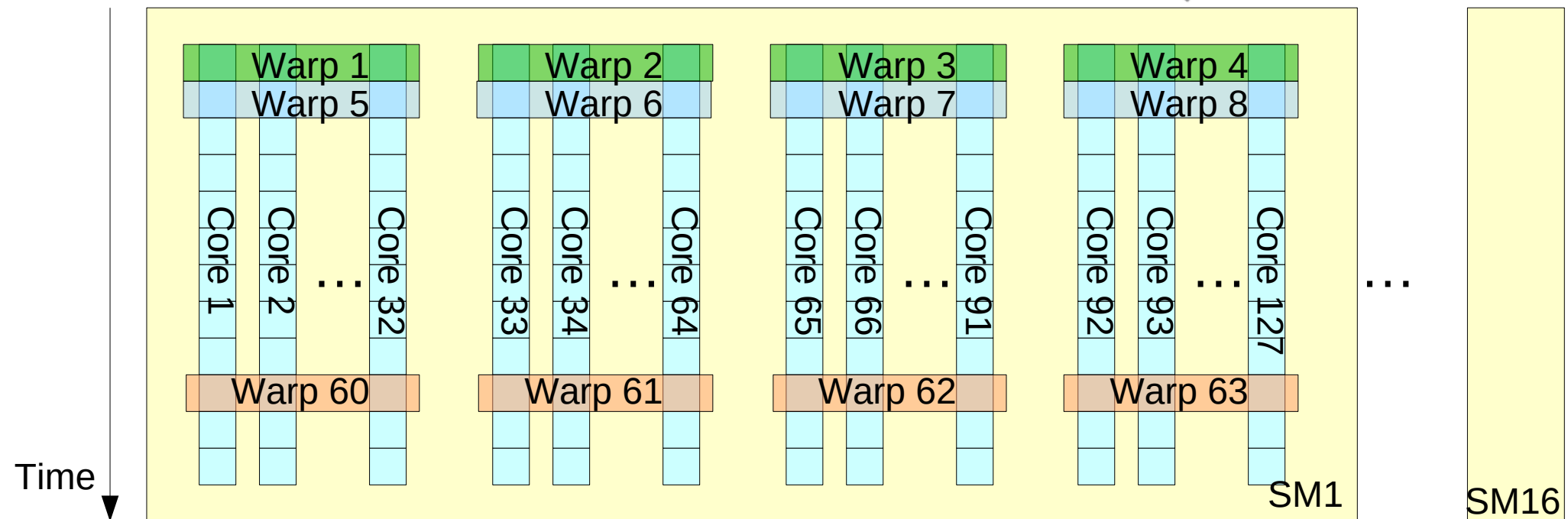
Example CPU: Intel Core i7

- Is a wide superscalar, but has also
 - ◆ Multicore
 - ◆ Multi-thread / core
 - ◆ SIMD units
- ➔ Up to 116 operations/cycle from 8 threads



Example GPU: NVIDIA GeForce GTX 980

- SIMT: warps of 32 threads
- 16 SMs / chip
- 4×32 cores / SM, 64 warps / SM



- ➔ 4612 Gflop/s
- ➔ Up to 32768 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

	Horizontal	Vertical
ILP	8	
TLP	4	2
DLP	8	

SIMD (AVX) Cores Hyperthreading

8	
16	2
2	

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

IBM Power 8

Oracle Sparc T5

Sparc T:
focus on TLP

10	
12	8
4	

2	
16	8

Cores

Threads

How to read the table

- Given an application with known ILP, TLP, DLP
how much throughput / latency hiding can I expect?
 - For each cell, take minimum of existing parallelism and hardware capability
 - The column-wise product gives throughput / latency hiding

	Sequential code no TLP, no DLP	Horizontal	Vertical
ILP	10 →	$\min(8, 10) = 8$	
TLP	1 →	$\min(4, 1) = 1$	2
DLP	1 →	$\min(8, 1) = 1$	

↓
Max throughput = $8 \times 1 \times 1$
for this application
Peak throughput = $8 \times 4 \times 8$
that can be achieved

→ Can only hope for ~3% of peak performance!

Classification: GPU and many small-core

Intel MIC

	Horizontal	Vertical
ILP	2	
TLP	60	4
DLP	16	

SIMD Cores

Nvidia Kepler

	Horizontal	Vertical
ILP	2	
TLP	16×4	32
DLP	32	

Cores × units SIMT Multi-threading

AMD GCN

	Horizontal	Vertical
ILP		
TLP	20×4	40
DLP	16	4

Tilera Tile-GX

	Horizontal	Vertical
ILP	3	
TLP	72	
DLP		

Kalray MPPA-256

	Horizontal	Vertical
ILP	5	
TLP	17×16	
DLP		

GPU: focus on DLP, TLP horizontal and vertical

Many small-core: focus on horizontal TLP

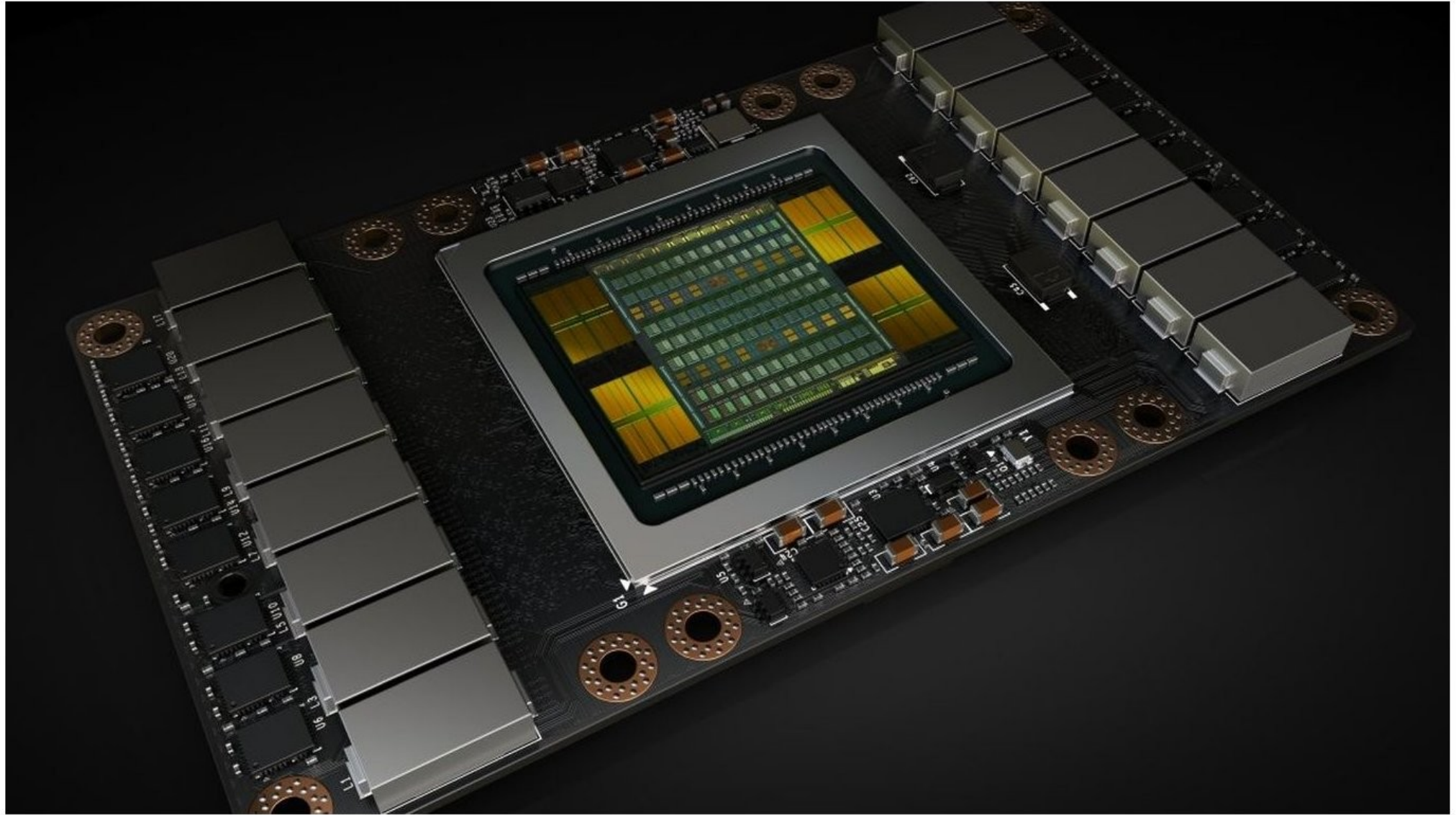
Takeaway

- Parallelism for throughput and latency hiding
- Types of parallelism: ILP, TLP, DLP
- All modern processors exploit the 3 kinds of parallelism
- GPUs focus on Thread-level and Data-level parallelism

Outline

- 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

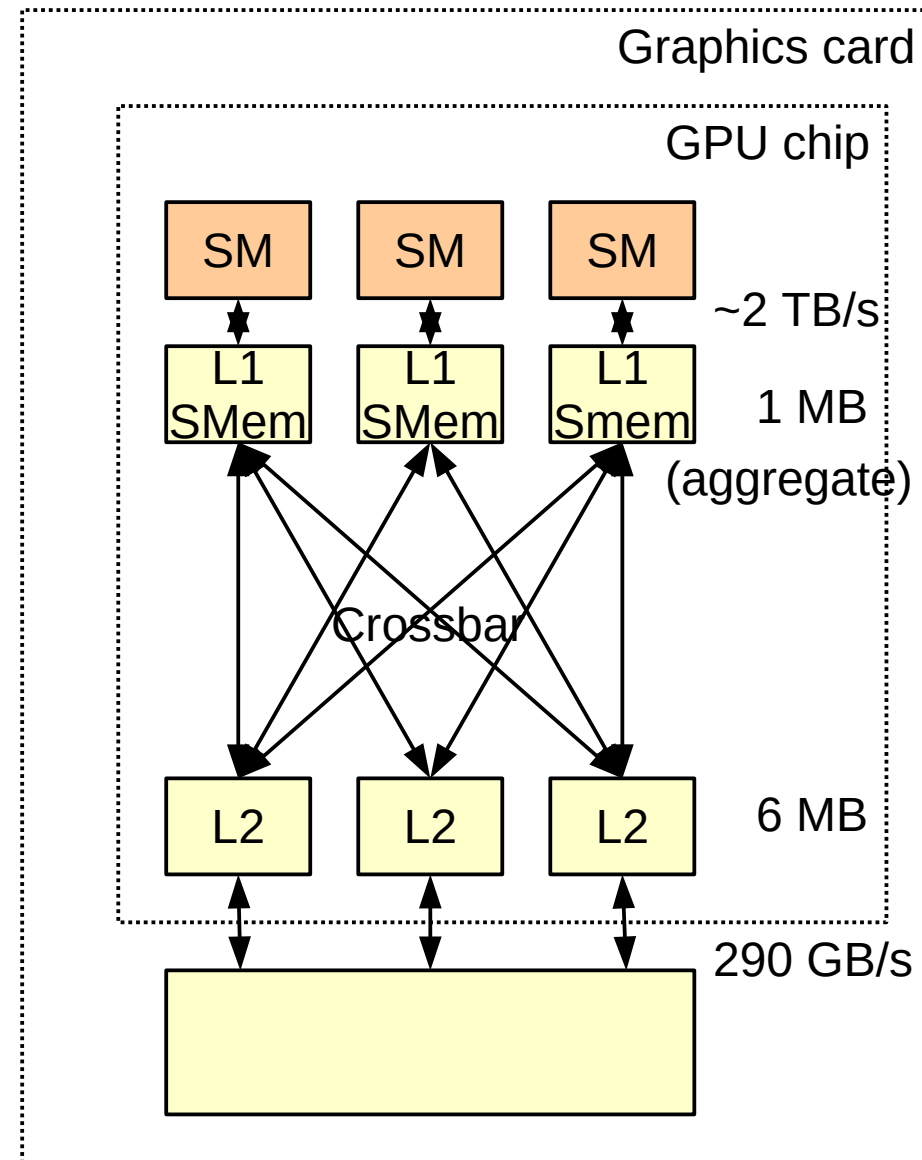
What is inside a graphics card?



NVIDIA Volta V100 GPU. Artist rendering!

GPU high-level organization

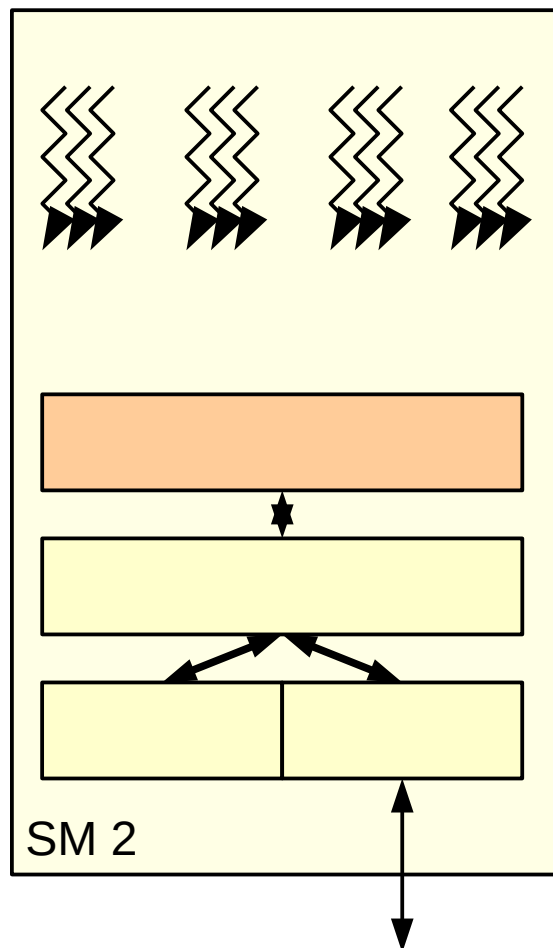
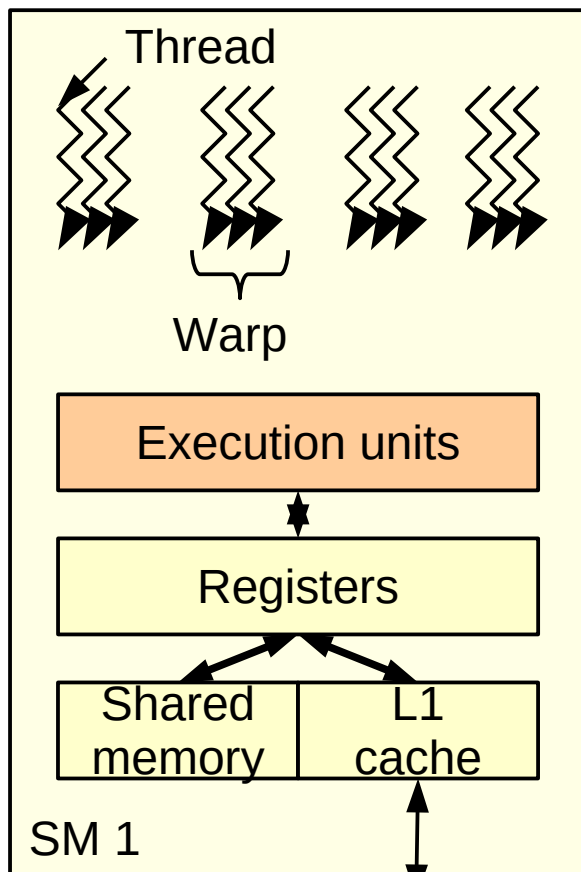
- Processing units
 - ◆ Streaming Multiprocessors (SM) in Nvidia jargon
 - ◆ Compute Unit (CU) in AMD's
 - ◆ Closest equivalent to a CPU core
 - ◆ Today: from 1 to 20 SMs in a GPU
- Memory system: caches
 - ◆ Keep frequently-accessed data
 - ◆ Reduce throughput demand on main memory
 - ◆ Managed by hardware (L1, L2) or software (Shared Memory)



GPU processing unit organization

Each SM is a highly-multithreaded processor

- Today: 24 to 48 warps of 32 threads each
→ ~1K threads on each SM, ~10K threads on a GPU



...

Outline

- 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
- High-level performance modeling

First-order performance model

Questions you should ask yourself,
before starting to code or optimize

- Will my code run faster on the GPU?
- Is my existing code running as fast as it should?
- Is performance limited by computations or memory bandwidth?

Pen-and-pencil calculations can (often) answer such questions

Performance: metrics and definitions

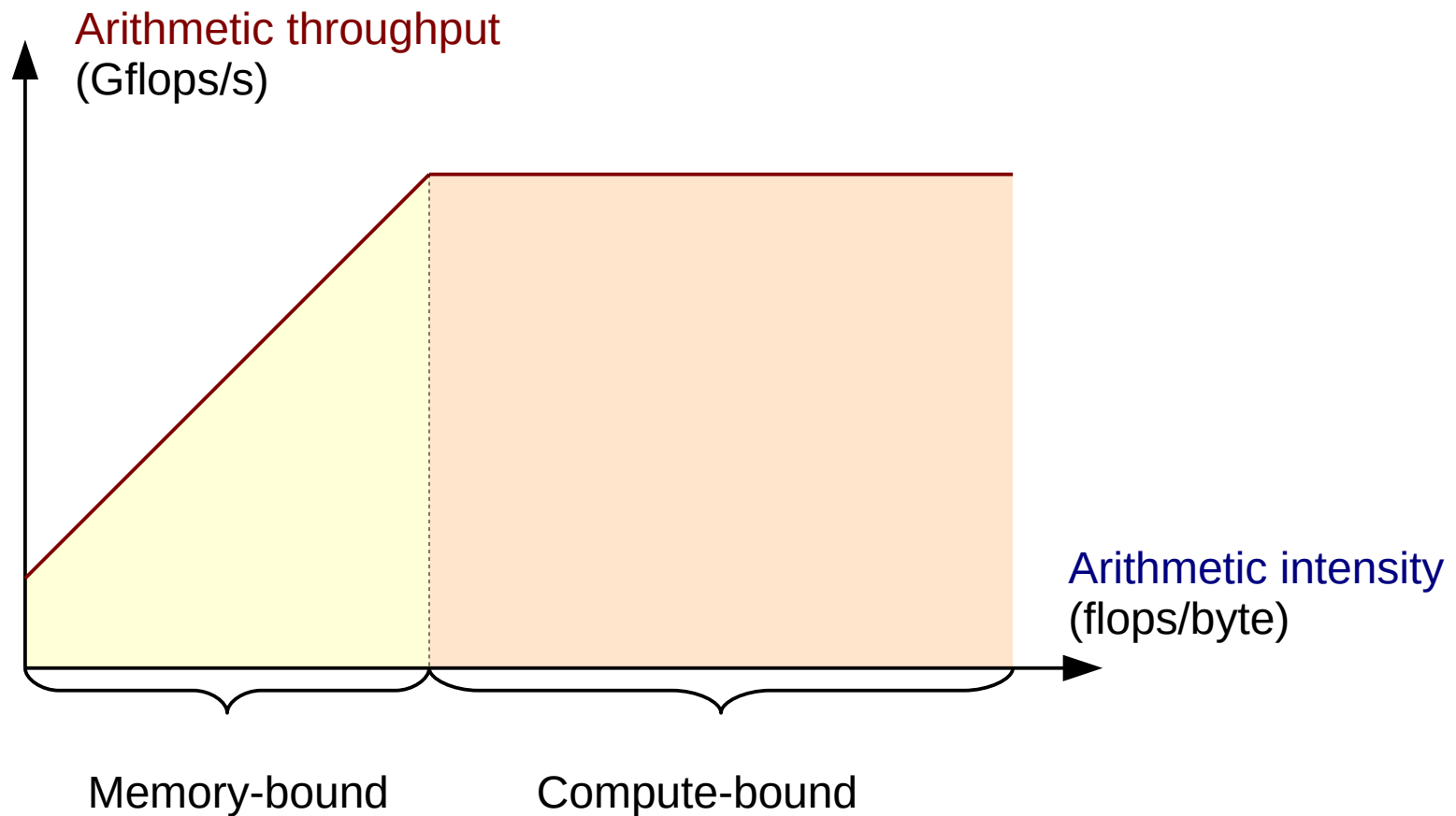
Optimistic evaluation: upper bound on performance

Assume perfect overlap of computations and memory accesses

- Memory accesses: *bytes*
 - ◆ Only external memory, not caches or registers
- Computations: *flops*
 - ◆ Only “useful” computations (usually floating-point) not address calculations, loop iterators..
- **Arithmetic intensity**: *flops / bytes*
= computations / memory accesses
 - ◆ Property of the code
- **Arithmetic throughput**: *flops / s*
 - ◆ Property of code + architecture

The roofline model

- How much performance can I get for a given arithmetic intensity?
 - ◆ Upper bound on arithmetic throughput, as a function of arithmetic intensity
 - ◆ Property of the architecture

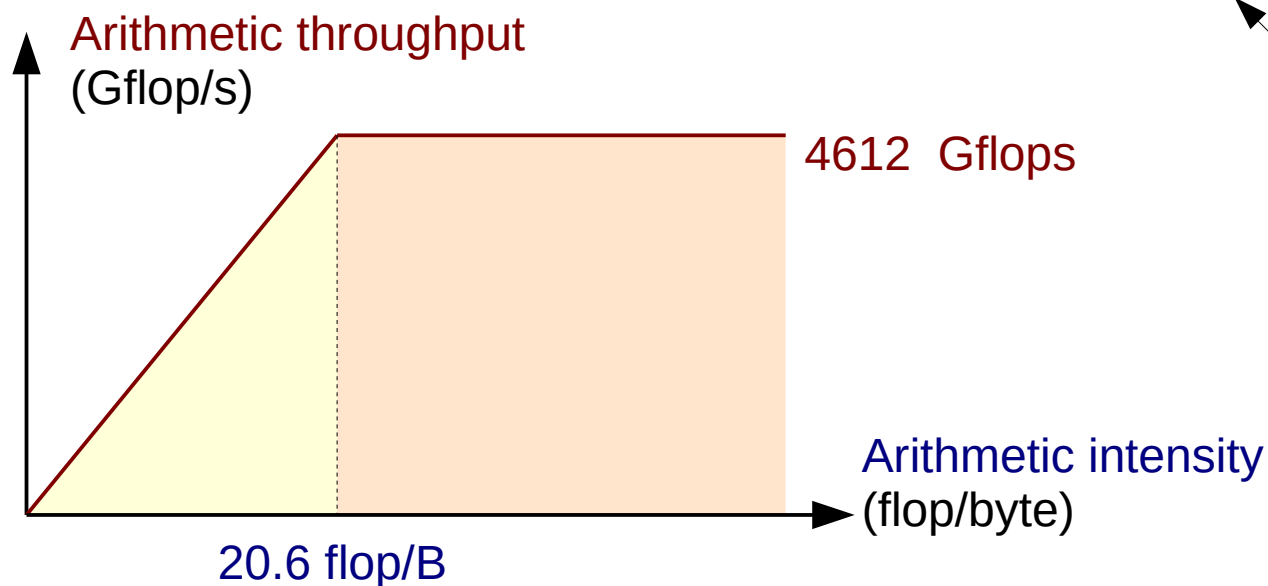


Building the machine model

- Compute or measure:

- ♦ Peak memory throughput *GTX 980: 224 GB/s*
- ♦ Ideal arithmetic intensity = peak compute throughput / mem throughput

$$\begin{aligned} \text{GTX 980: } 4612 \text{ (Gflop/s)} / 224 \text{ (GB/s)} &= 20.6 \text{ flop/B} \\ &\times 4 \text{ (B/flop)} = 82 \text{ (dimensionless)} \end{aligned}$$

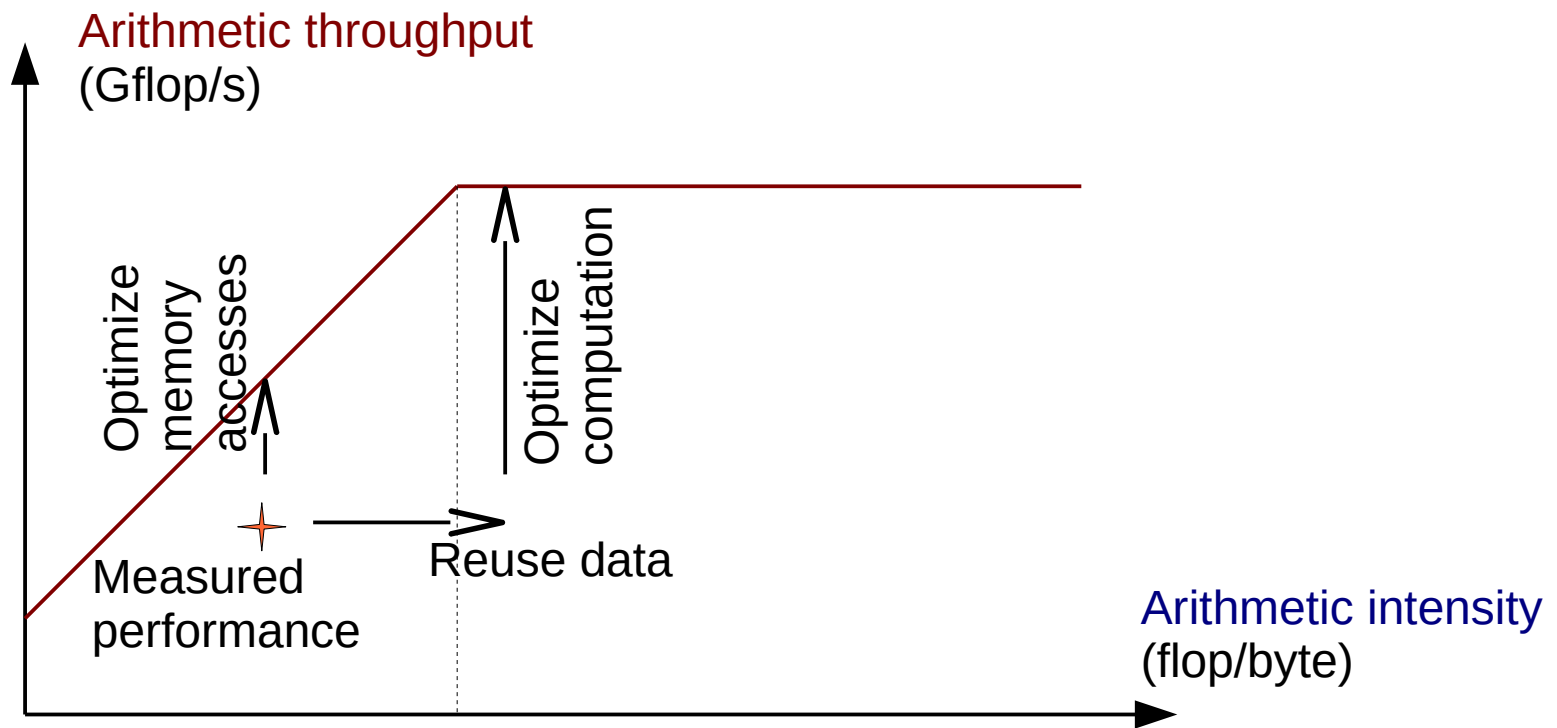


Beware of units:
float=4B, double=8B !

- Achievable peaks may be lower than theoretical peaks
 - ♦ Lower curves when adding realistic constraints

Using the model

- Compute arithmetic intensity, measure performance of program
- Identify bottleneck: memory or computation
- Take optimization decision



Example: dot product

```
for i = 1 to n
  r += a[i] * b[i]
```

- How many computations?
- How many memory accesses?
- Arithmetic intensity?
- Compute-bound or memory-bound?
- How many Gflop/s on a GTX 980 GPU?
 - ◆ With data in GPU memory?
 - ◆ With data in CPU memory?
- How many Gflop/s on an i7 4790 CPU?

GTX 980: 4612 Gflop/s, 224 GB/s

i7 4790: 460 Gflop/s, 25.6 GB/s

PCIe link: 16 GB/s

Example: dot product

```
for i = 1 to n
  r += a[i] * b[i]
```

- How many computations? → 2 n flops
- How many memory accesses? → 2 n words
- Arithmetic intensity? → 1 flop/word = 0.25 flop/B
- Compute-bound or memory-bound? → Highly memory-bound
- How many Gflop/s on a GTX 980 GPU?
 - ♦ With data in GPU memory? $224 \text{ GB/s} \times 0.25 \text{ flop/B} \rightarrow 56 \text{ Gflop/s}$
 - ♦ With data in CPU memory? $16 \text{ GB/s} \times 0.25 \text{ flop/B} \rightarrow 4 \text{ Gflop/s}$
- How many Gflop/s on an i7 4790 CPU?
 $25.6 \text{ GB/s} \times 0.25 \text{ flop/B} \rightarrow 6.4 \text{ Gflop/s}$
Conclusion: don't bother porting to GPU!

GTX 980: 4612 Gflop/s, 224 GB/s

i7 4790: 460 Gflop/s, 25.6 GB/s

PCIe link: 16 GB/s

Takeaway

- 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
- Performance models
 - ◆ Quick estimates and back-of-the-envelope calculations can save time
- Next time: GPU programming in CUDA