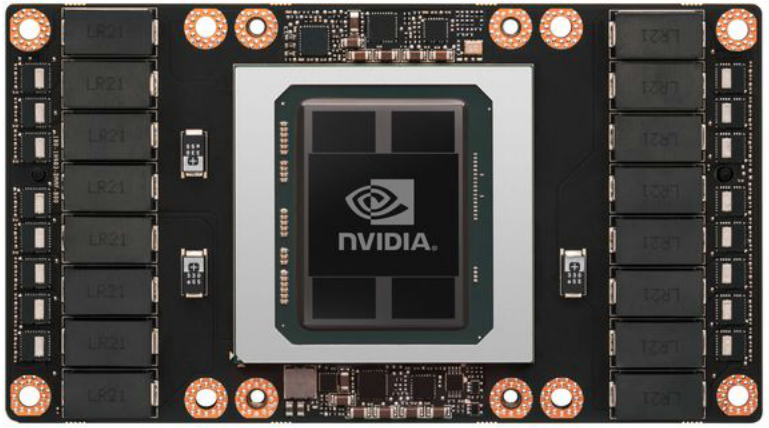
ECE 1782

Motivation and Challenges

Why GPUs

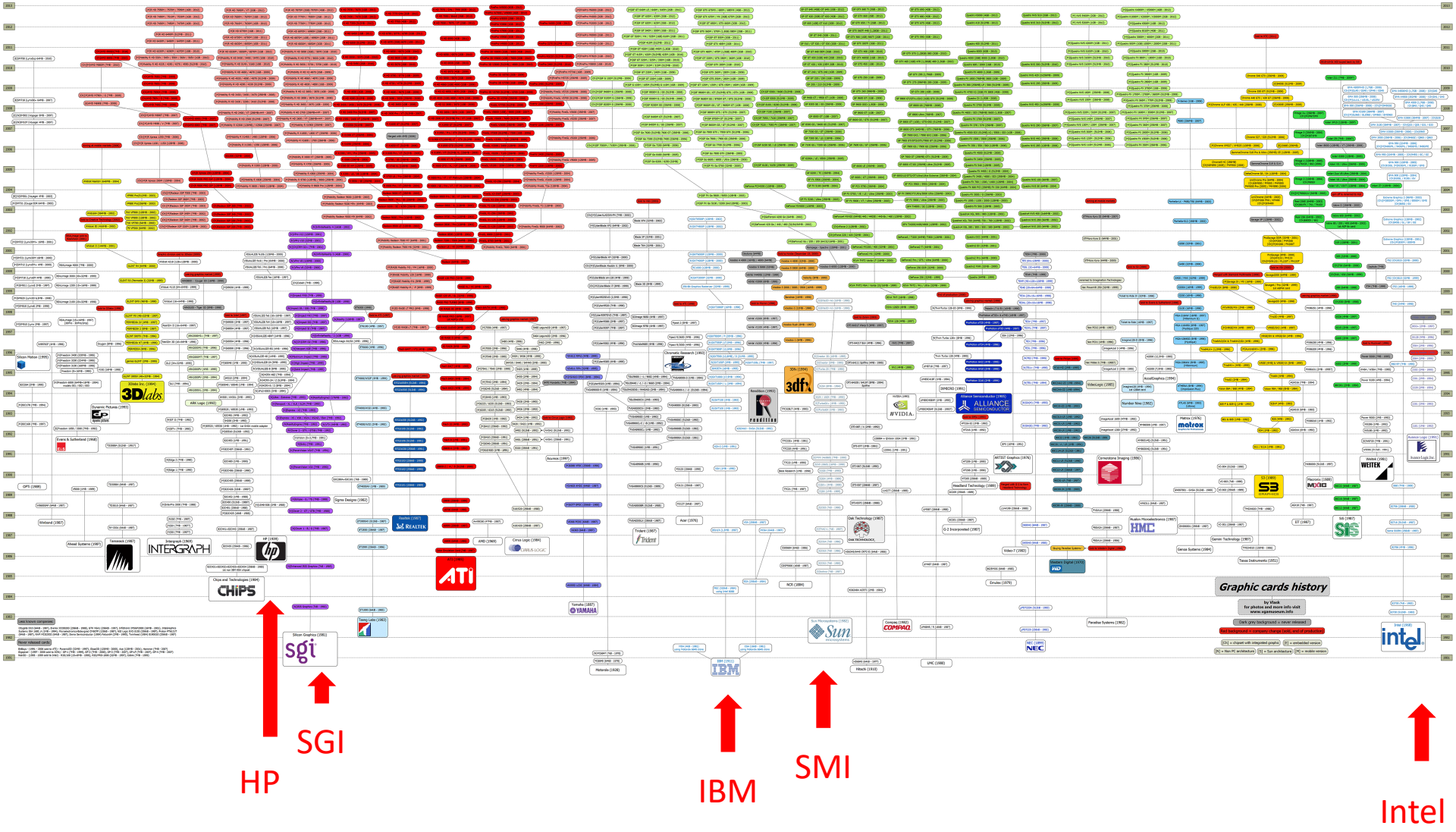
- A lot of processing power
- At relatively low cost



E.g.,

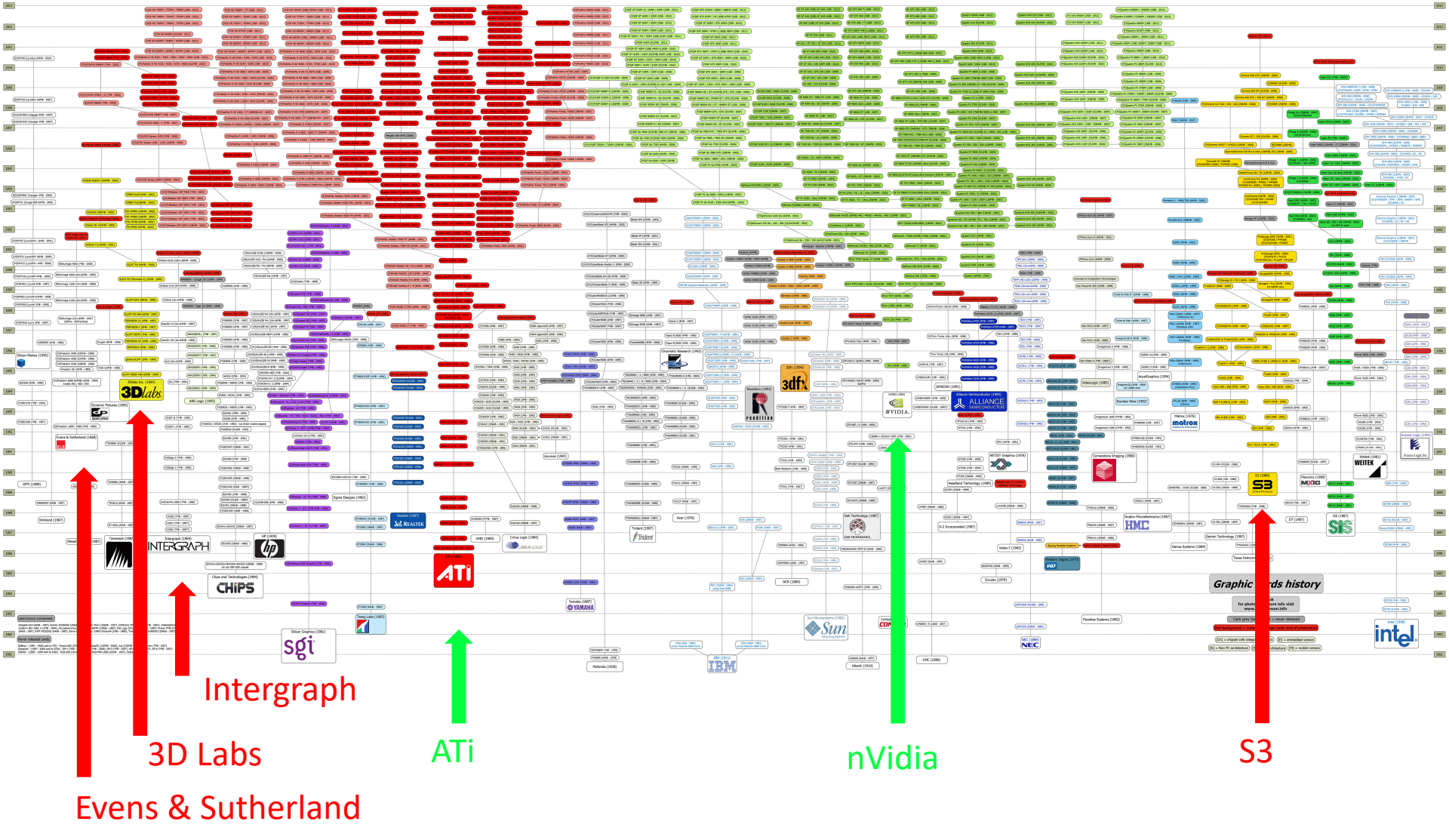
	2016 Nvidia Tesla P100:	2020 Nvidia Ampere A100
Cores	3,584	6,912
SP floating point	10.6 Teraflops	19.5 Teraflops
DP floating point	5.3 Teraflops	9.7 Teraflops
Memory bandwidth	732 GB/s	1.6 TB/s

History of graphics engines

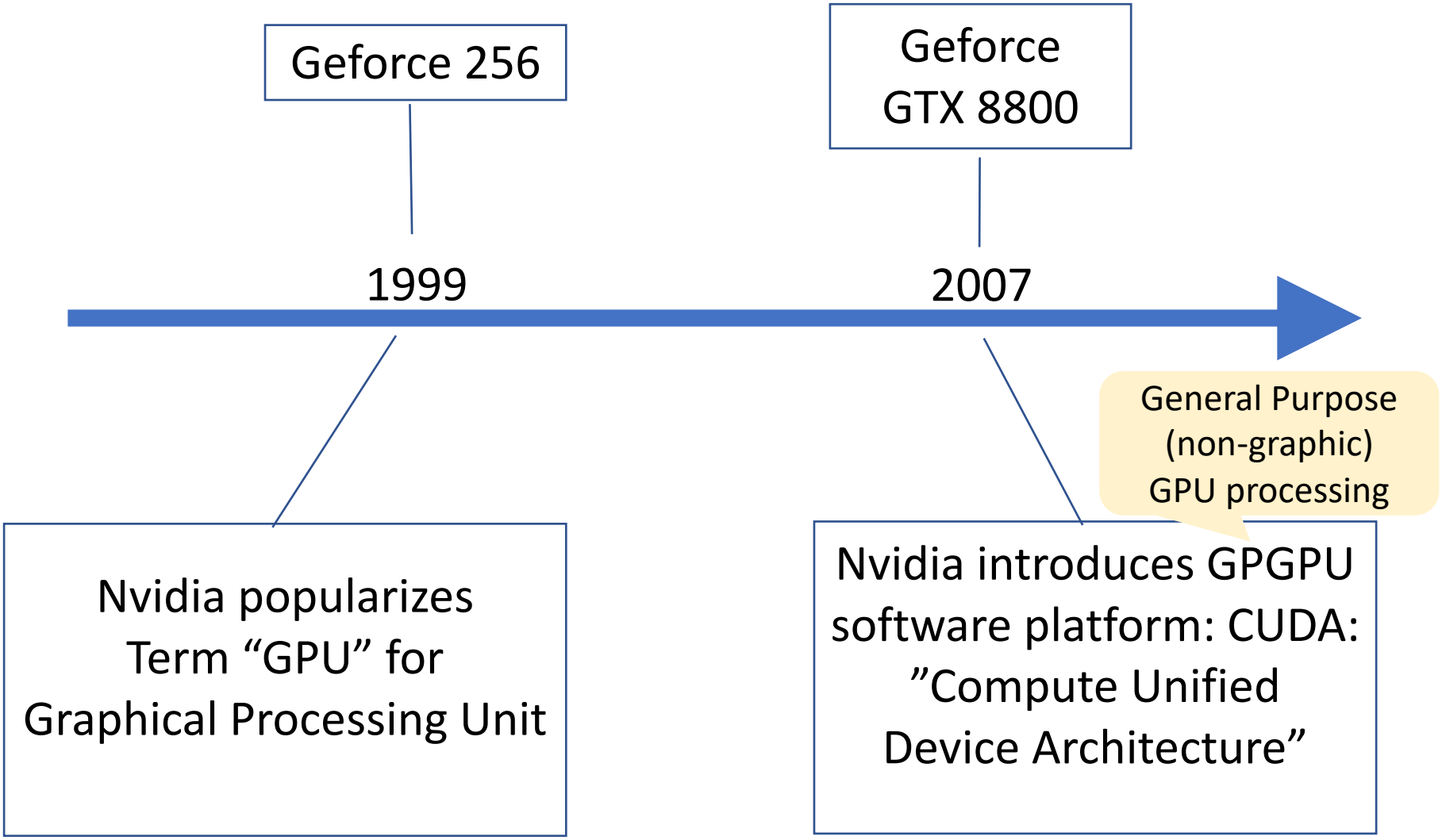


See also: https://en.wikipedia.org/wiki/Graphics_processing_unit

History of graphics cards



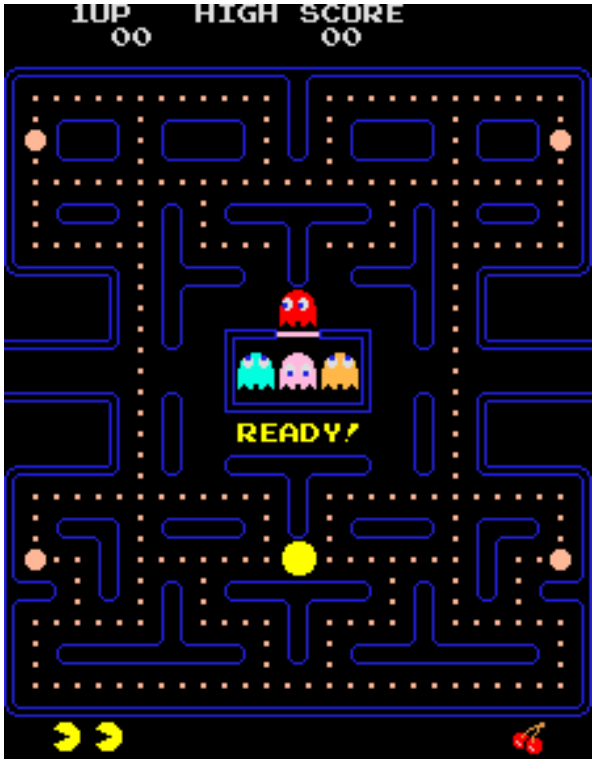
Some GPU History



GPUs driven by game playing

1989: Intel 486: 1st chip w. 1M transistors
486DX: integrated FPU
1993: Pentium: faster FPU, but not fast enough
1997: Intel introduces MMX: SIMD vector proc.

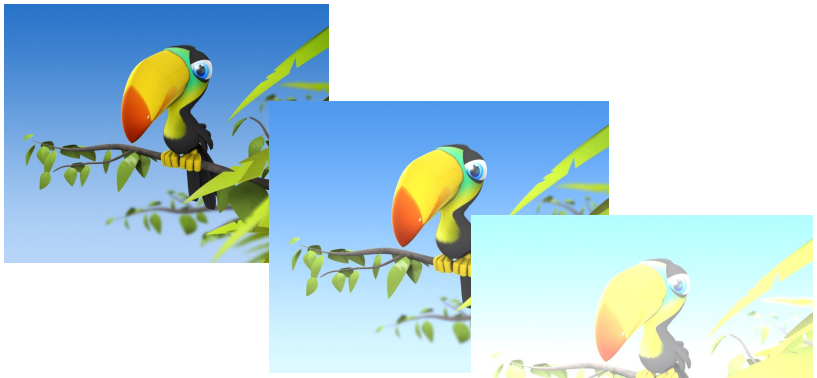
But: game players were monsters & insatiable
you give them 2X better, they want 10X
you give them 10X, they want 100X more



Graphics operations: FP-intensive embarrassingly parallelizable

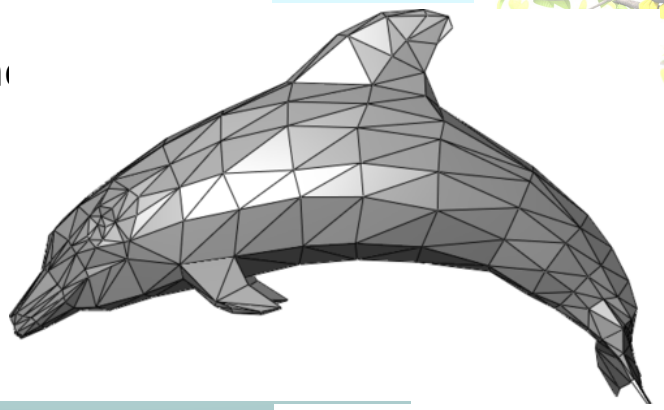
2D:

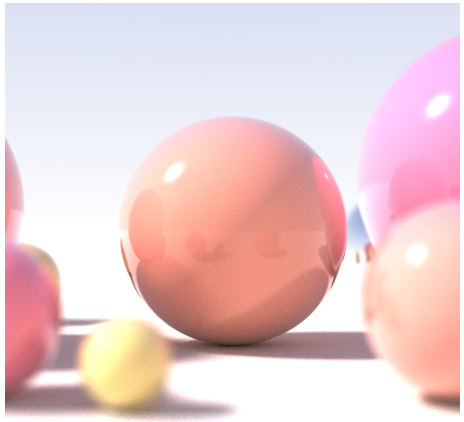
- Shifting & Scaling
- Fading & Filtering
- Rotating: $\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$

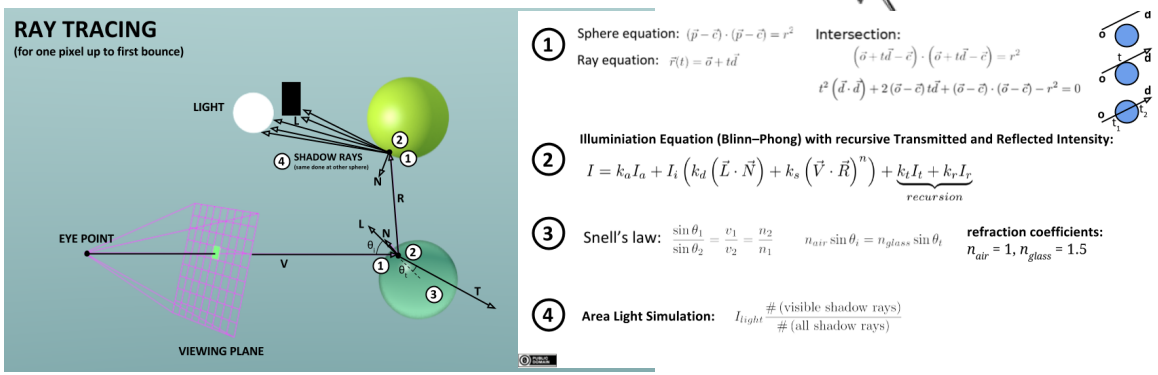


3D:

- Objects represented as 3D triangular mesh
- Transform (move, rotate, scale)
- Paint / Texture mapping
- Shading & Ray Tracing
- Rasterize → convert into pixel







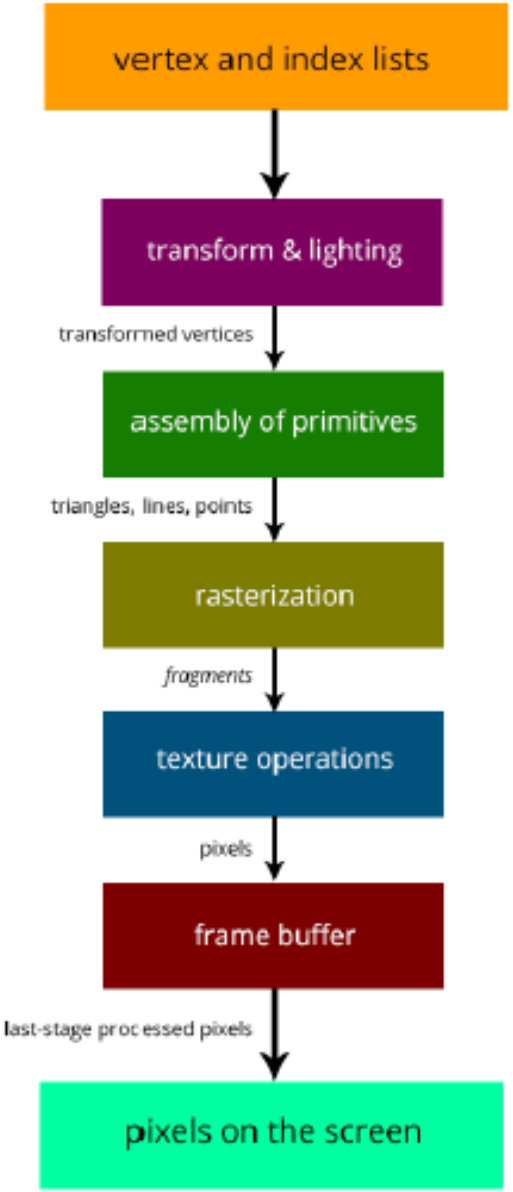
https://en.wikipedia.org/wiki/Ray_tracing_(graphics)

GPUs driven by game playing

1989: Intel 486: 1st chip w. 1M transistors
486DX: integrated FPU
1993: Pentium: faster FPU, but not fast enough
1997: Intel introduces MMX: SIMD vector proc.

But: game players were monsters & insatiable
you give them 2X better, they want 10X
you give them 10X, they want 100X more

Early GPUs: Fixed function pipelines:
preset fcts, limited options



GPUs driven by game playing

1989: Intel 486: 1st chip w. 1M transistors
486DX: integrated FPU
1993: Pentium: faster FPU, but not fast enough
1997: Intel introduces MMX: SIMD vector proc.

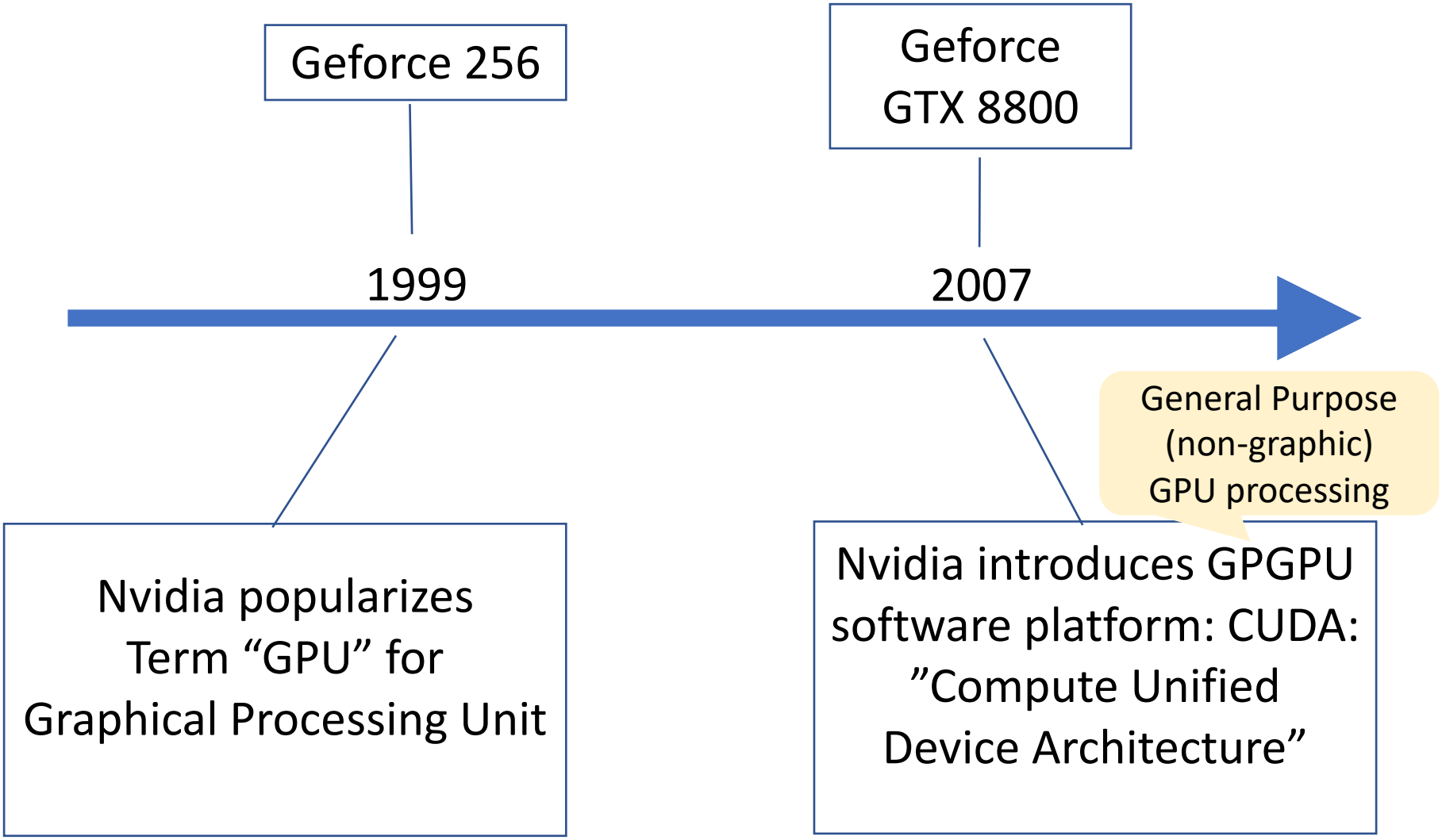
But: game players were monsters & insatiable
you give them 2X better, they want 10X
you give them 10X, they want 100X more

Early GPUs: Fixed function pipelines:
preset fcts, limited options
Then shaders:

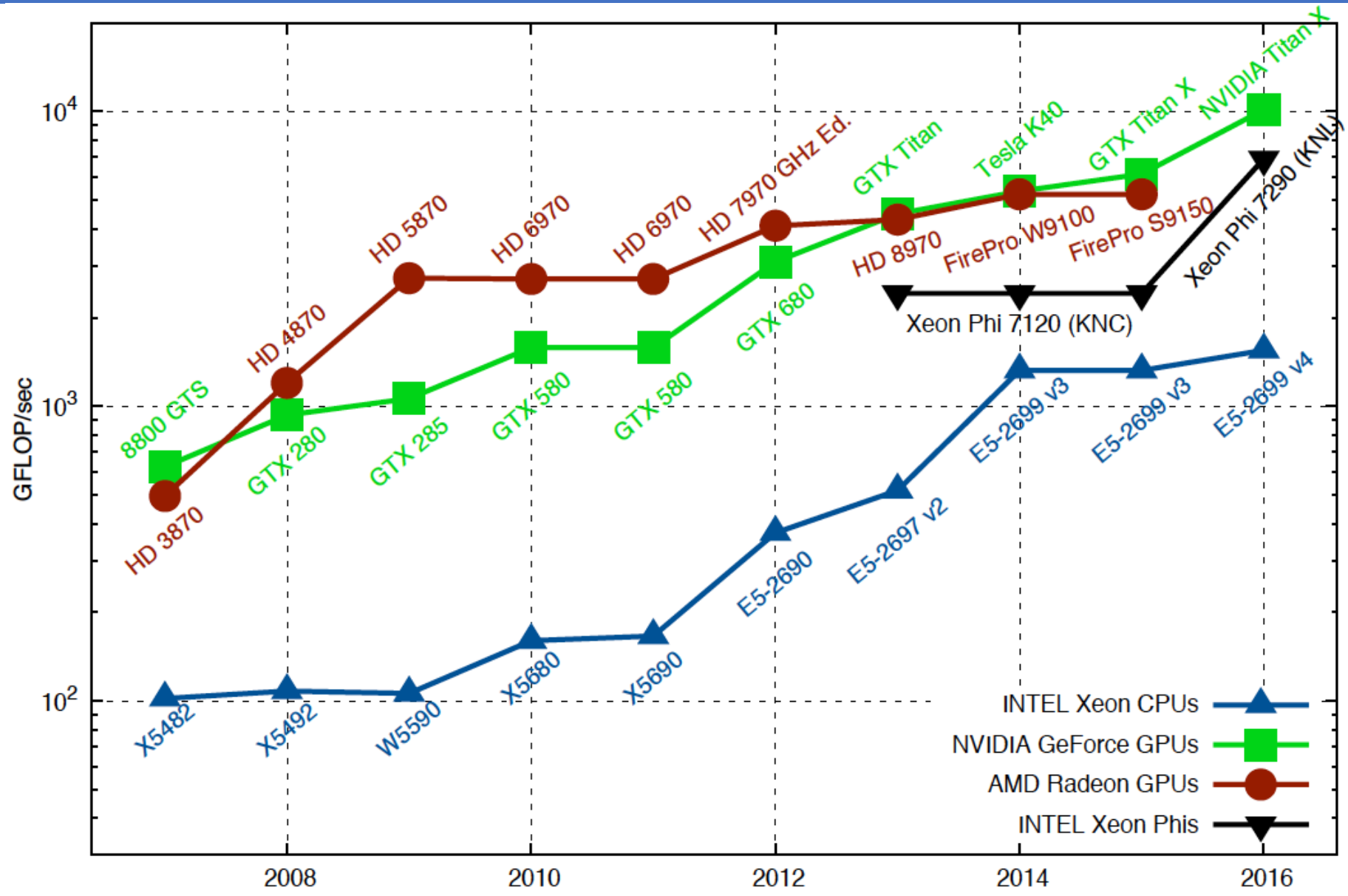
- could implement own fcts
- CLSL: C-like language
- could sneak in general purpose programming

Soon thereafter: Nvidia notices there is $$$ in GPGPUs…

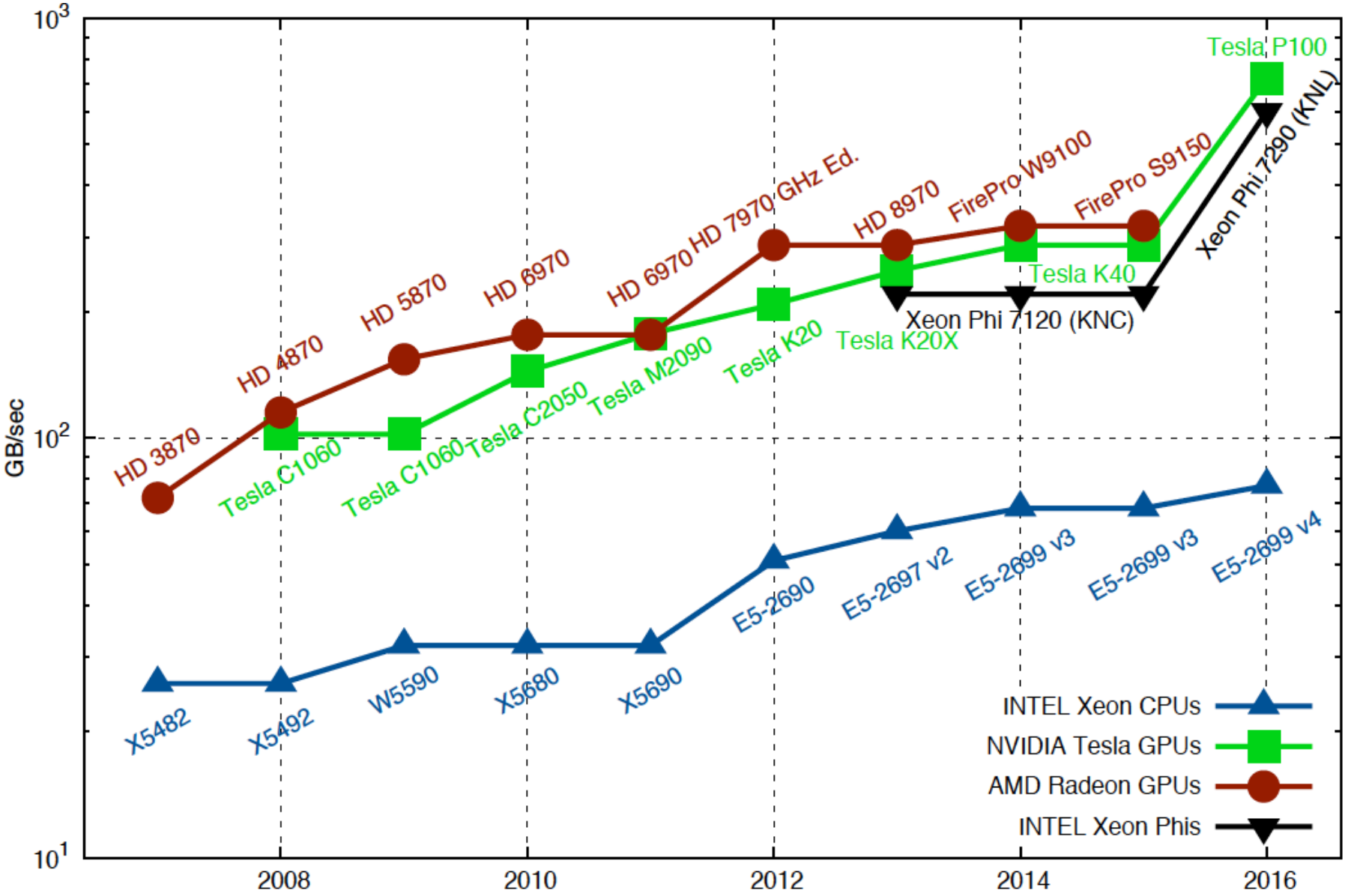
Some GPU History



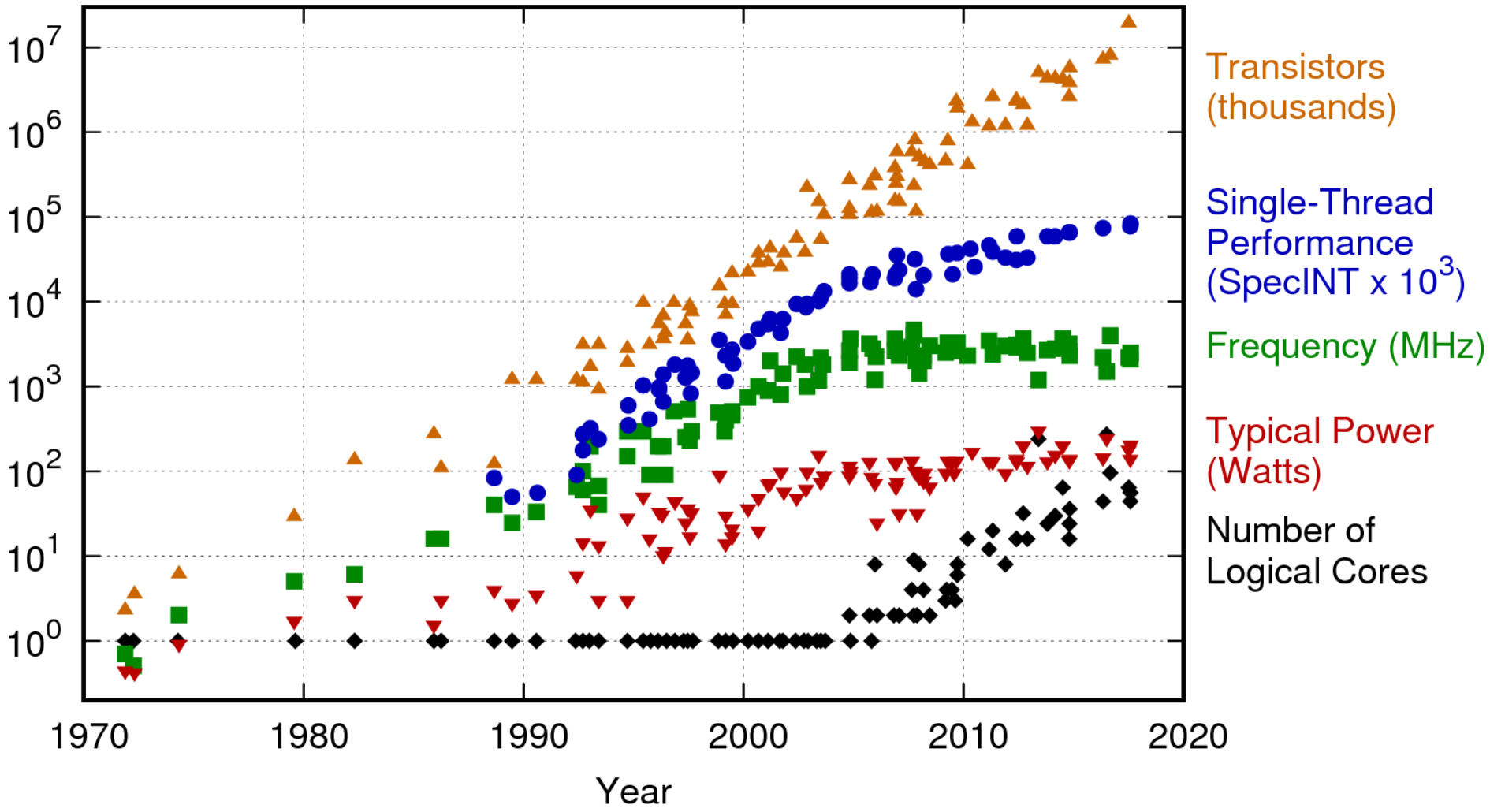
Single Precision Floating Point Ops



Theoretical Peak Memory Bandwidth



40 years of Microprocessor Trend Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

Multicore Processors

Guiding principle: Make each thread execute as quickly as possible with increasingly complex microarchitectural features:

- Pipelining of execution
- Out of order execution
- Branch prediction
- Pre-fetching
- Large amount of multilevel cache
- Hyperthreading

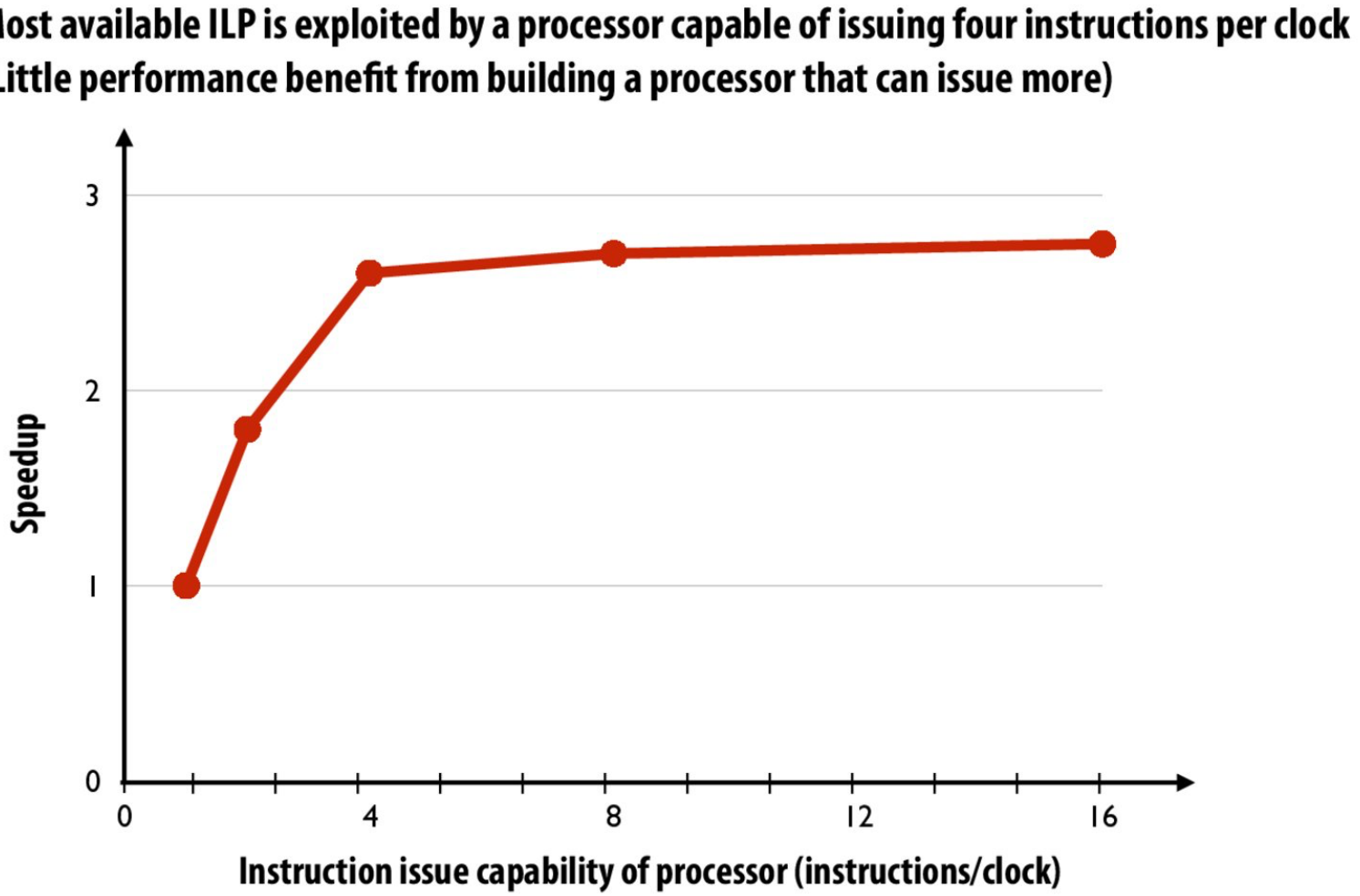
Very hard to squeeze out more!
Even with lots of available transistors.

Until 15 Years Ago: Primary Sources for Performance Improvements

1. Exploiting instruction-level parallelism
   - deep pipelining
   - out-of-order execution

2. Increasing CPU clock frequency

Diminishing returns on superscalar execution



Power draw superlinear with Frequency

Dynamic power $\propto$ capacitive load $\times$ voltage^2 $\times$ frequency

Static power: transistors burn power even when inactive due to leakage

Maximum allowed frequency determined by processor's core voltage

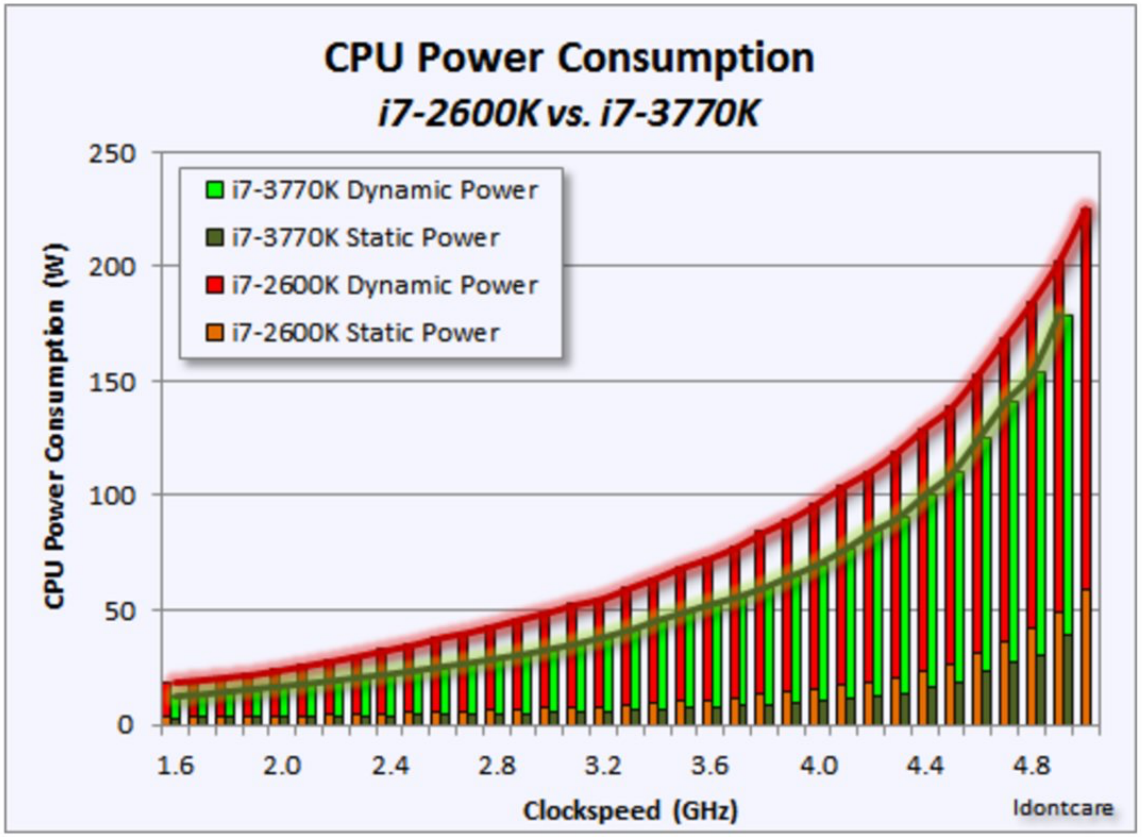
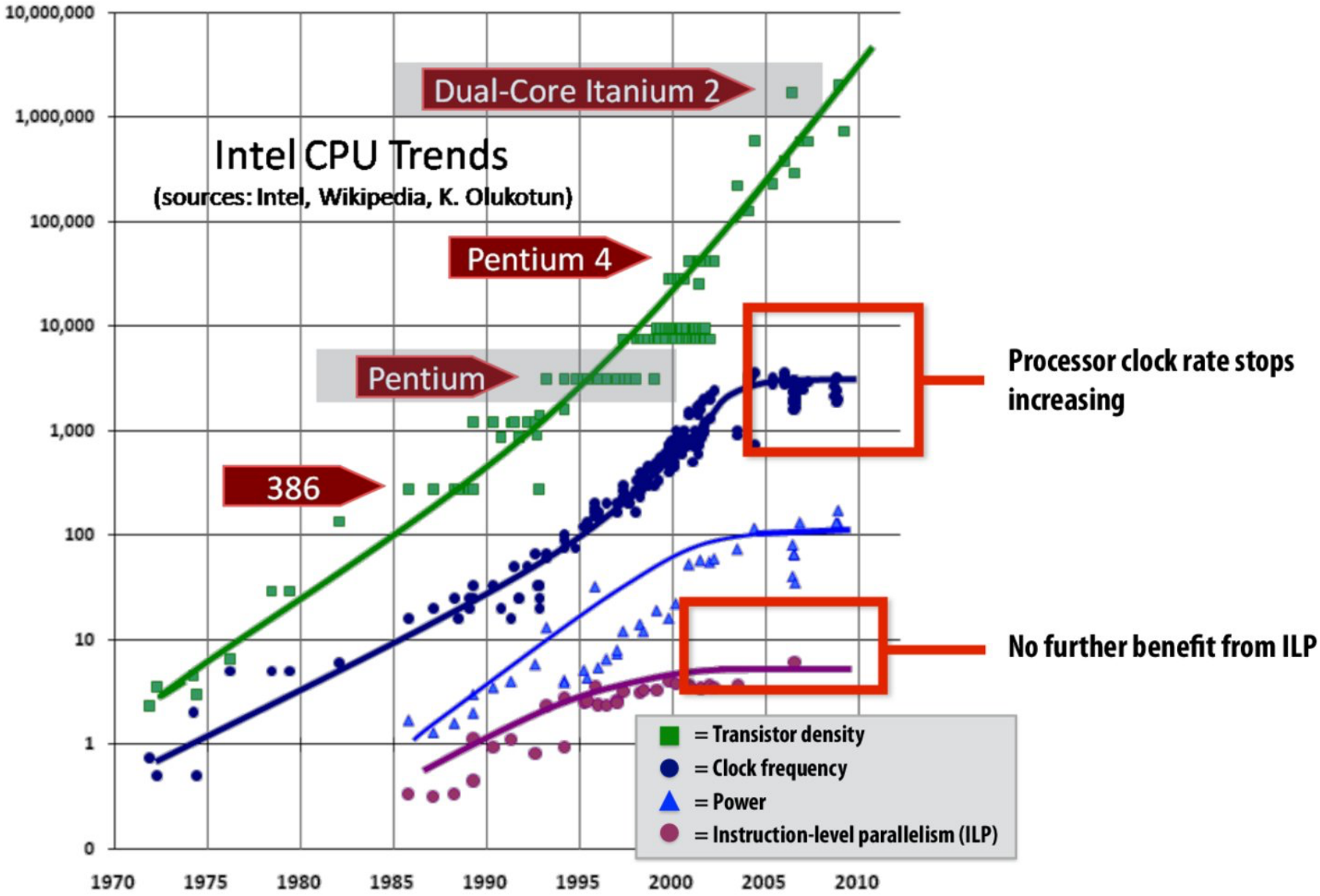


Image credit: "Idontcare": posted at: http://forums.anandtech.com/showthread.php?t=2281195

End of the Road for CPU Guiding Principle



Bottom line:

1. Hard to further scale single-core performance

2. Current strategy: increase number of execution units for parallelism
   (or build processing units for specialized purposes)

➔ Software must be written to be parallel to see performance gains.
Can no longer just wait a year for new processor technology as a strategy for improved performance.

GPUs to the extreme

Many computing units operating in parallel

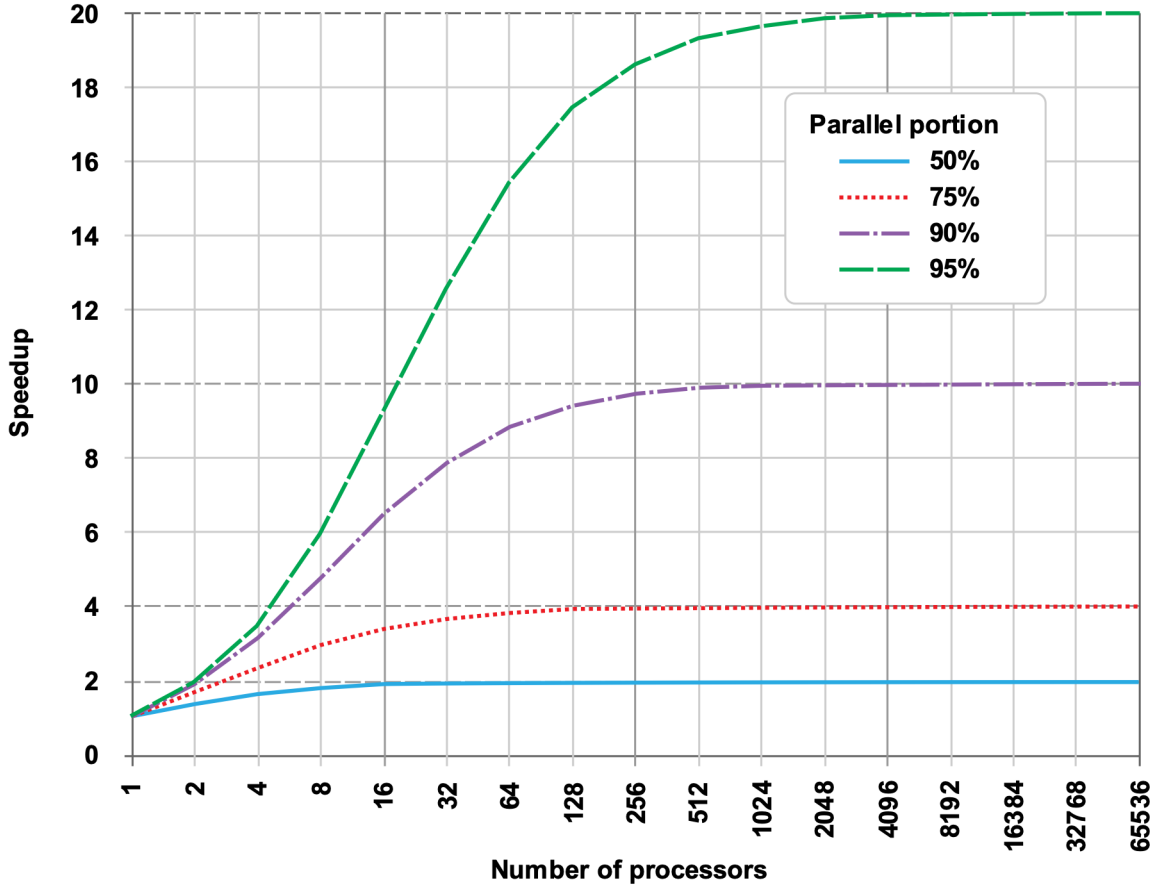
Ideal for simple, but repetitive tasks, like:

- large data set processing
- machine learning
- dense linear algebra
- finite-difference
- finite-element
- image recognition
- etc.

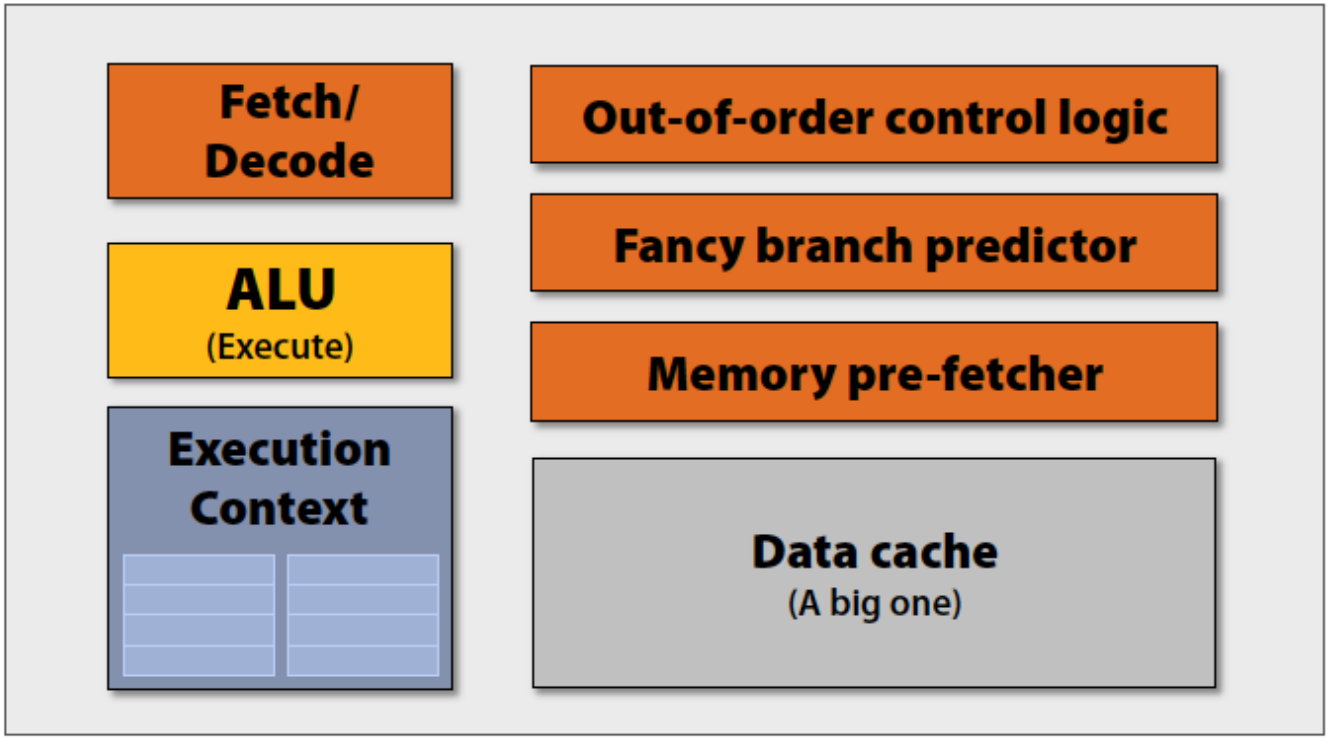
But bad when calculations involve significant branching

Amdahl's Law

$$Speedup < \frac{1}{1-p}$$



CPU-style cores

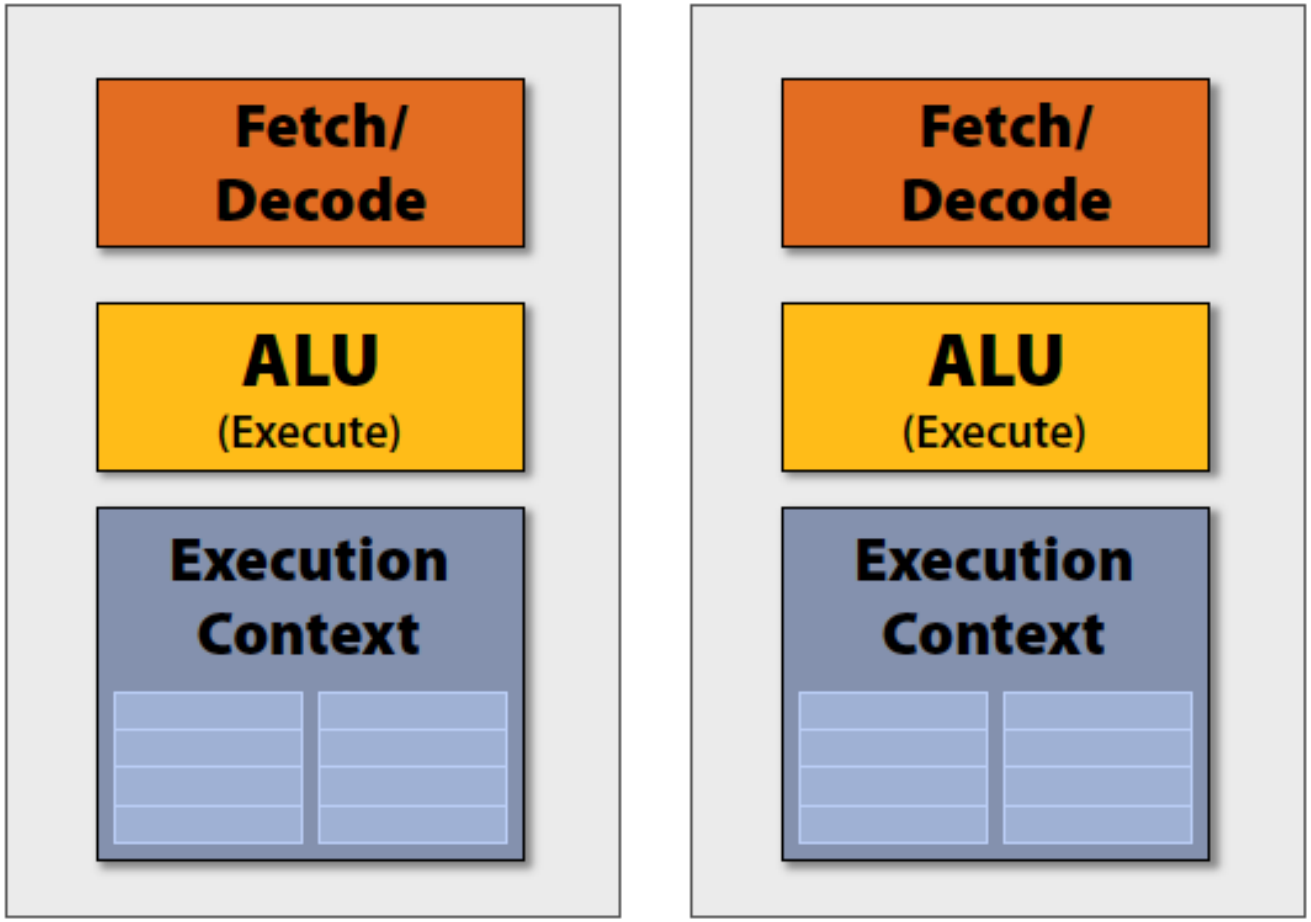


GPUs slim it down

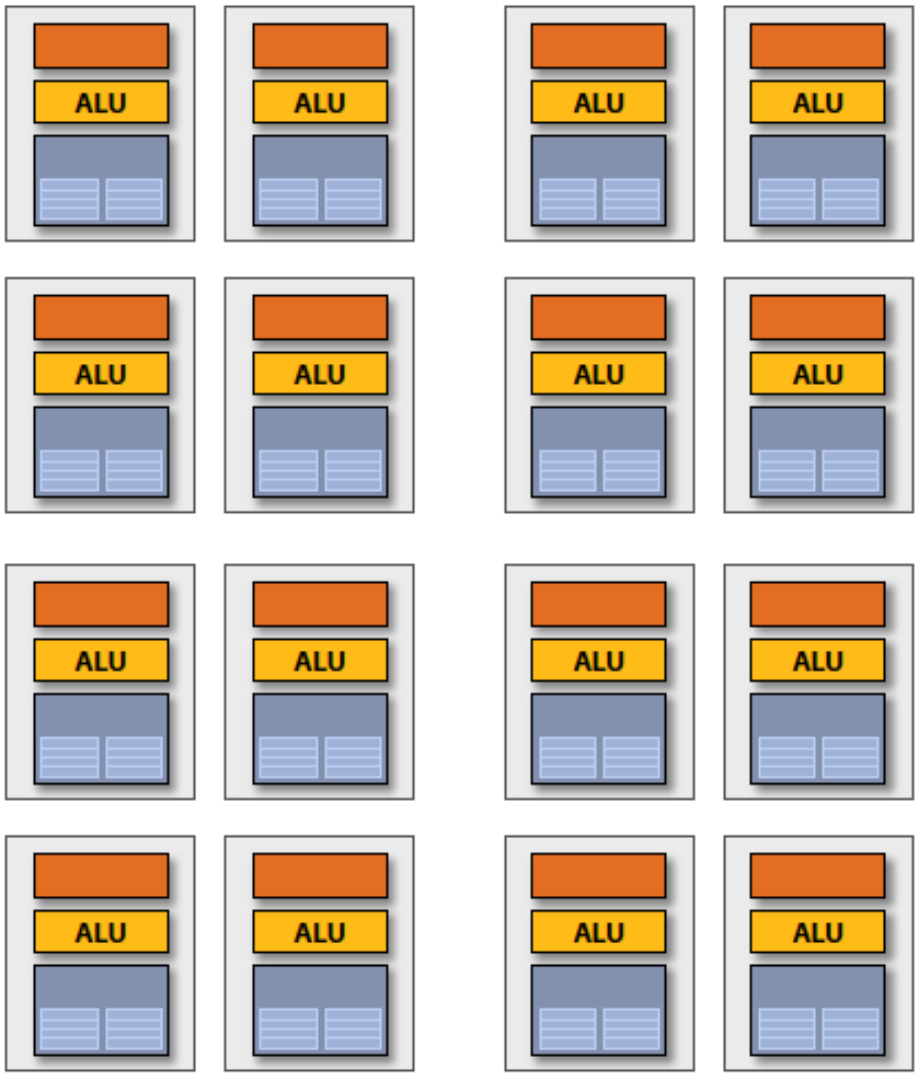
Idea #1:

Remove components that help a single instruction stream run fast

Double # cores with freed-up space



In fact: 16 cores fit in ~ same space



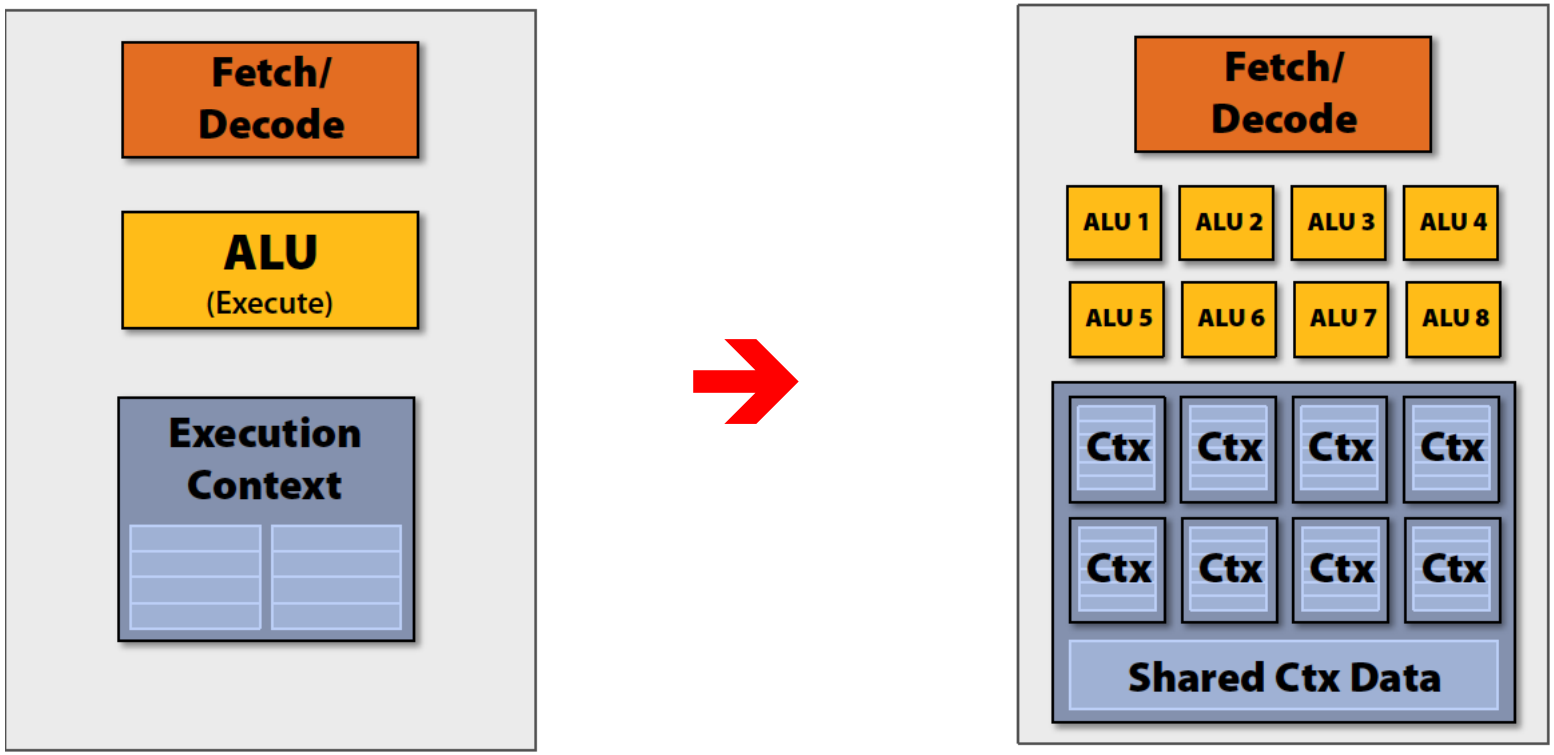
16 independent instruction streams

But in reality, often not that different/independent

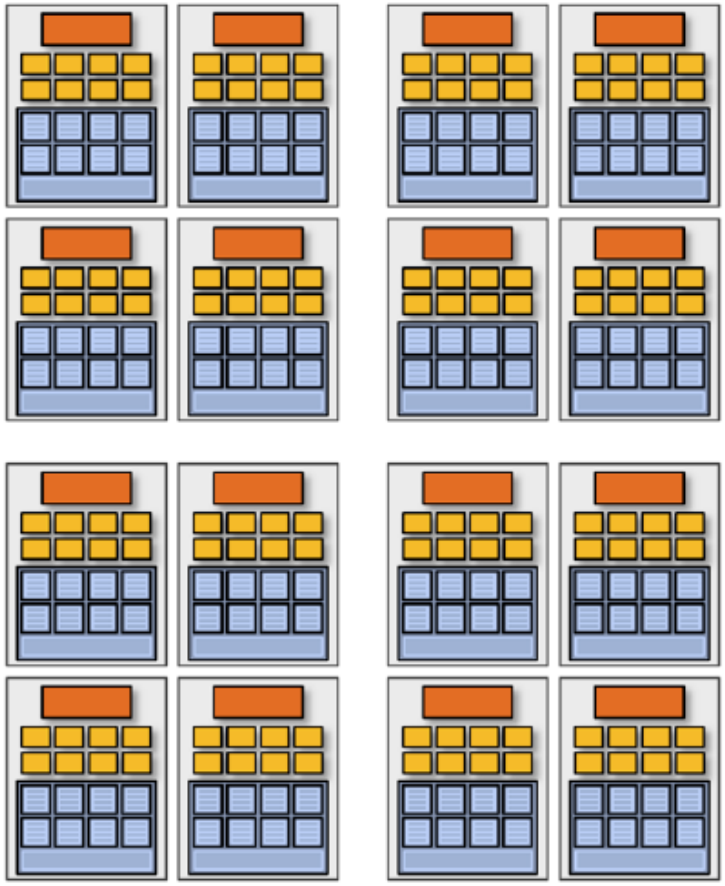
Saving more space

Idea #2: Amortize cost & complexity of managing an instruction stream across many ALU cores

➔ SIMD processing



Now much more power in same space



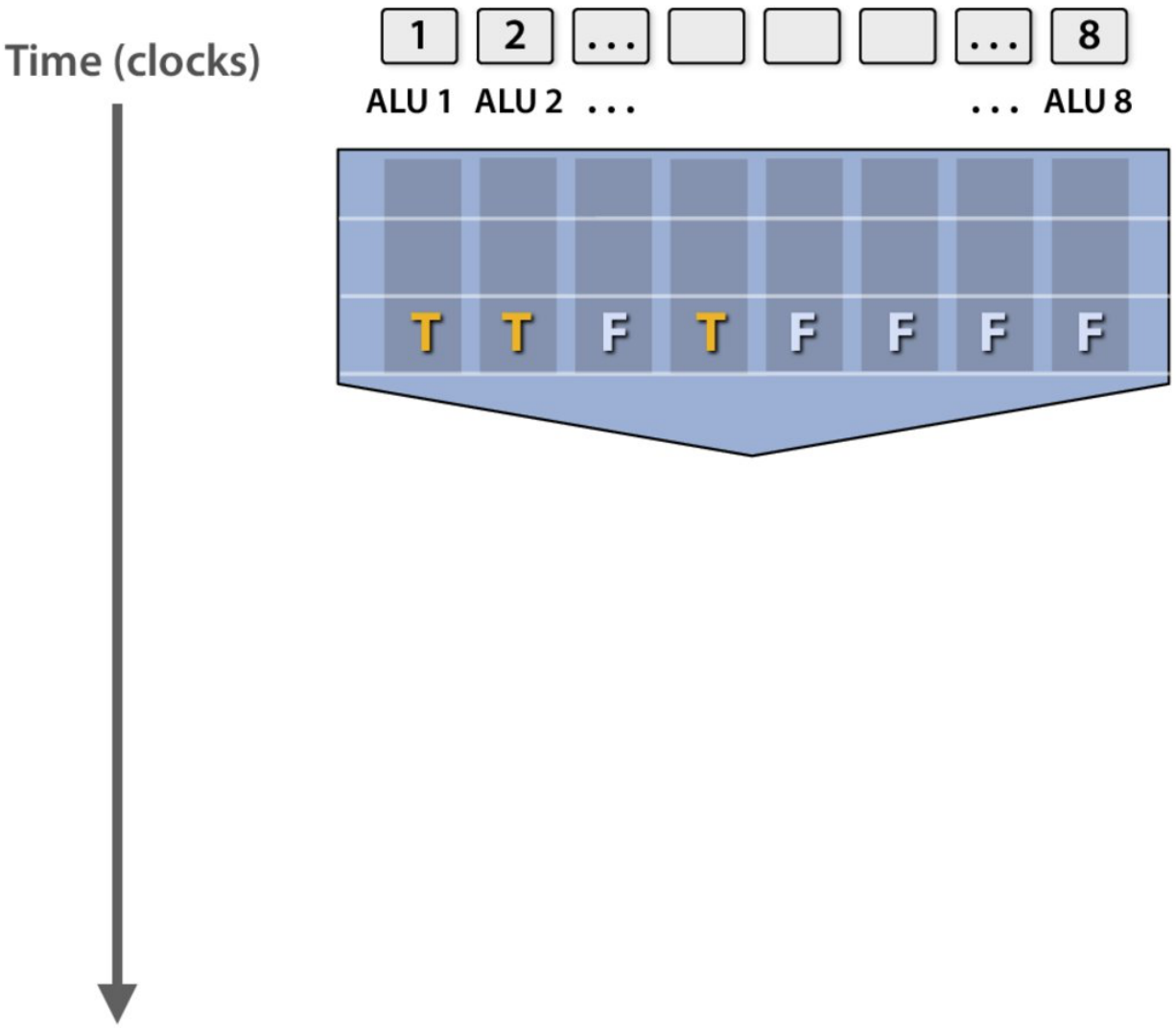
Example:

128 instruction streams in parallel

16 independent groups of
8 synchronized streams

But what if not all 8 streams want to execute same instructions?

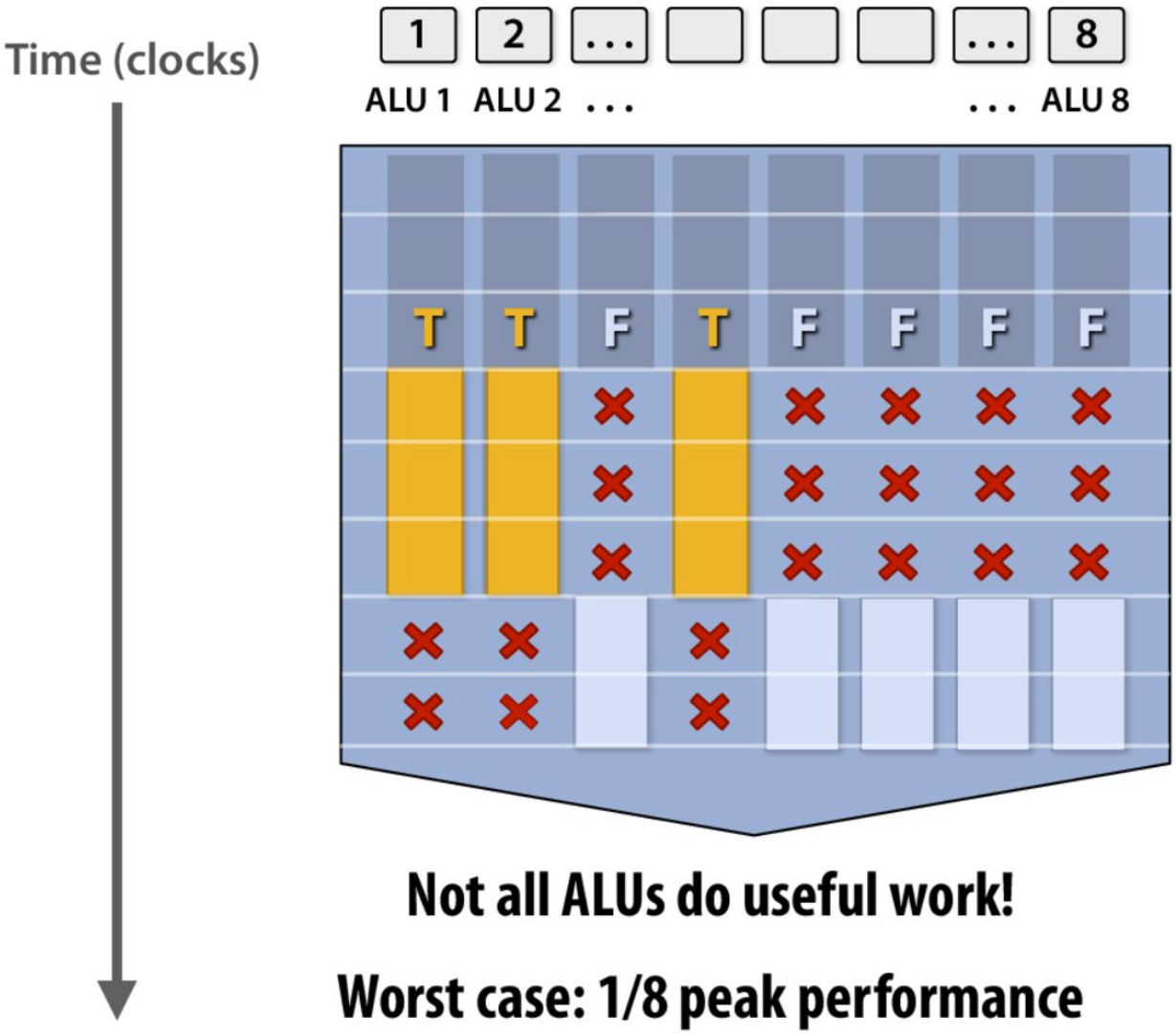
What about conditional execution?



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<unconditional code>

float x = A[i];

if (x > 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp;
}

<resume unconditional code>


result[i] = x;
```

Mask (discard) output of ALU



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<unconditional code>

float x = A[i];

if (x > 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp;
}

<resume unconditional code>

result[i] = x;
```

What about memory access times? Remember, we got rid of caches!

Getting rid of caches makes sense!

- caches ineffective on big-data programs

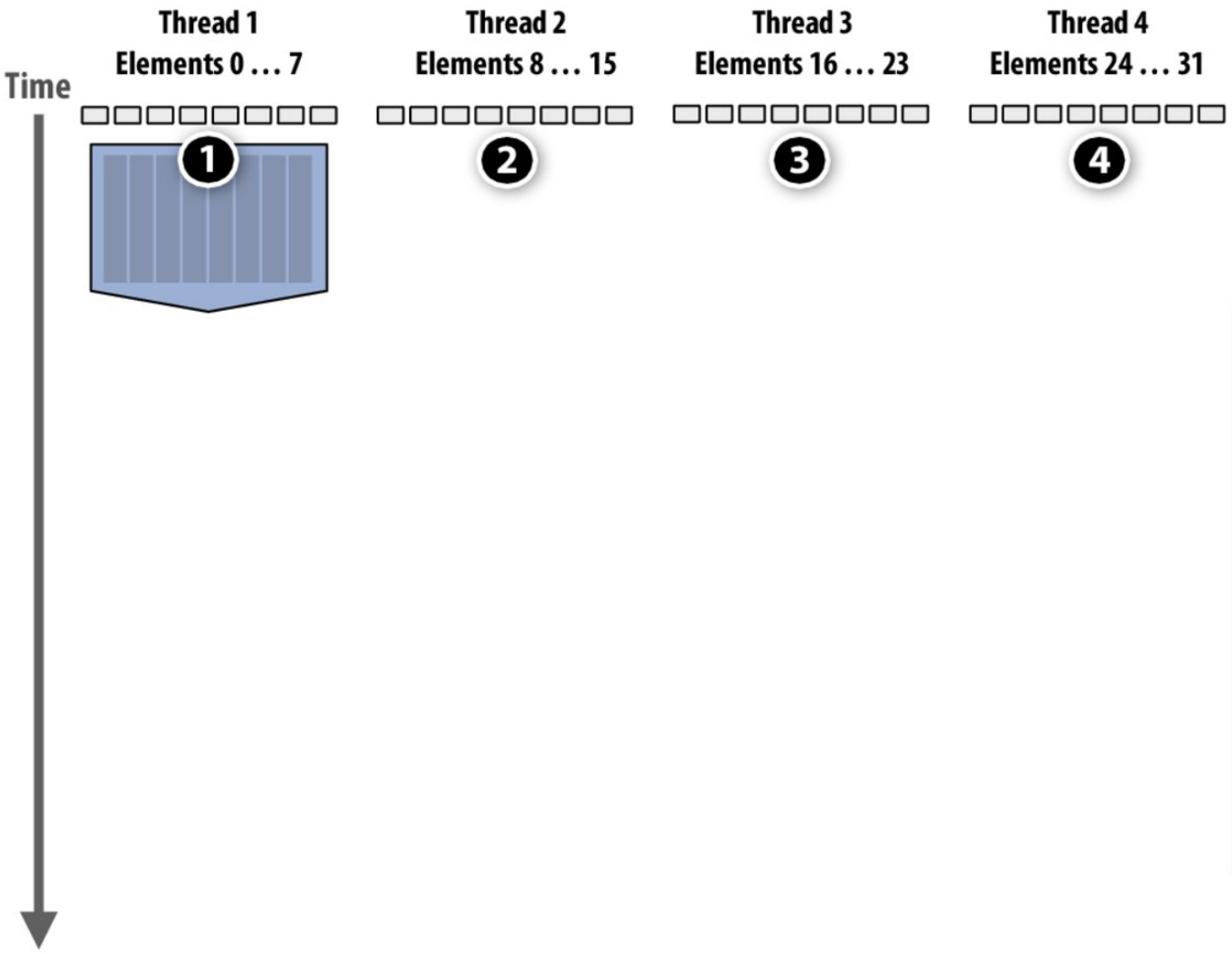
Still: latency to access memory: 100's cycles!

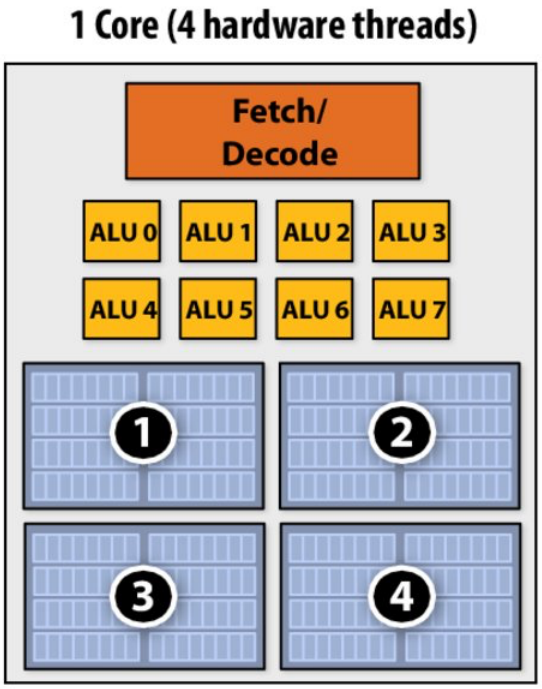
- memory access stalls will hurt performance

How best to address? Some ideas…

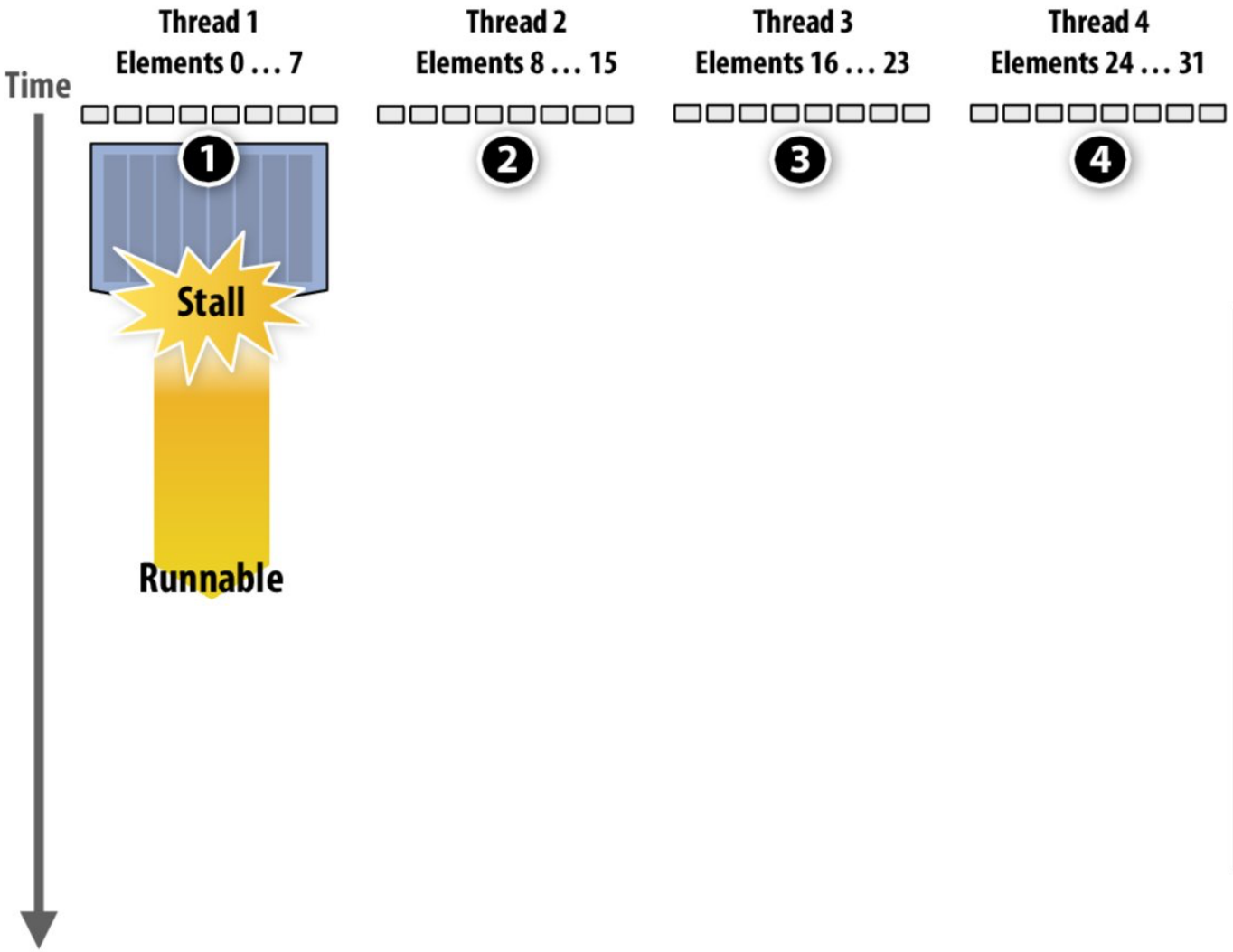
- prefetching…
- programmer controlled caches:
  - registers (100's)
  - "shared memory"
- multithreading & context switching

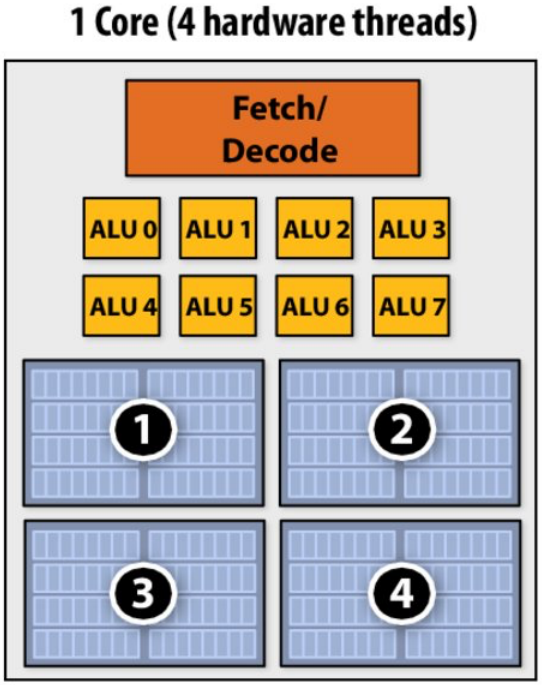
Hiding Stalls with Multithreading 1



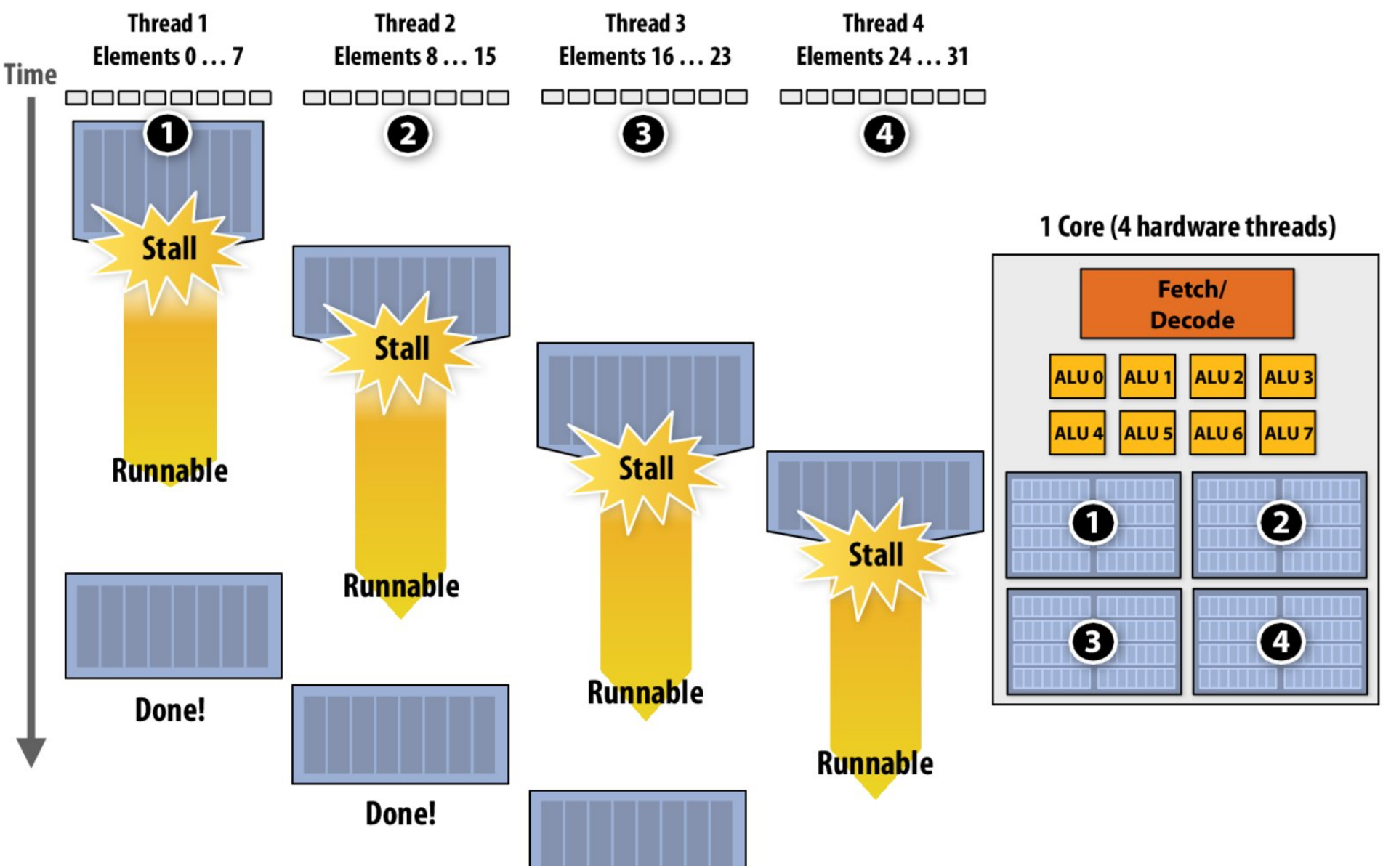


Hiding Stalls with Multithreading 2





Hiding Stalls with Multithreading 3



GPU context switching

Instantaneous from one cycle to the next

➔ your program needs ~10 threads per core

Course objective: learn how to prog. GPUs

Co-processor accelerators

with 1000's of processing cores

organized through a special architecture

Parallel programming is challenging

- Recall Amdahl's Law
  - you can only exploit 1,000+ processing cores
    if you can find the parallelism and
    largely eliminate code that is not parallelizable

GPU Programming is Challenging

- You cannot program a GPU
  unless you understand the architecture
- You cannot achieve good performance
  unless you understand the micro-architecture
- Micro-architectural details change with every GPU generation ➔ progs not really portable
- Worse, many micro-architectural features are not documented. ➔ much experimentation required…

Nvidia alone released over 400 types of chips

https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units

Architectures					
Tesla	Fermi	Kepler	Maxwell	Pascal	Turing
2008	2010	2012	2014	2016	2018
Products					
GTX 8800 GTX 9800 GTX 280	GTX 460 GTX 480 GTX 580	GTX Titan Blk GTX Titan Z	GTX 960 GTX 980 GTX Titan X	GTX 1060 GTX 1070 GTX 1080 GTX Titan X	RTX 2080 Quadro RTX 6000
Tesla 1060	Tesla 2070	Tesla K80	-	Tesla P100	NYA
↑	↑	↑	↑	↑	↑
Baseline architecture	Caches	FP performance	Power efficiency	16-bit FP ops	NGX for graphics

Product Lines		
	GeForce, GTX: desktop Quadro: High performance prof. desktop Tesla: HPC	Jetson: embedded systems Tegra: mobile