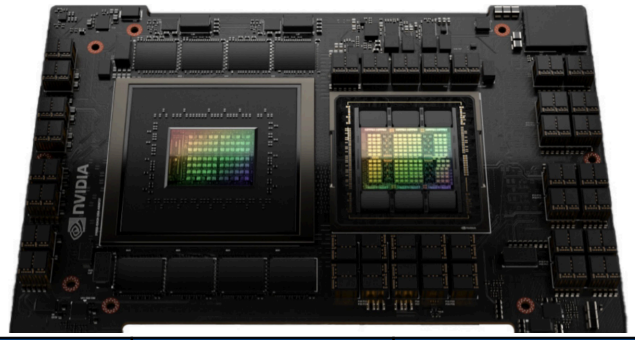
Lecture 1: Motivation, Parallel Computing and Challenges

CS599: Programming Massively Parallel Multiprocessors and Heterogenous Systems (Understanding and programming the devices powering AI)

Jonathan Appavoo

Why GPUs

- A lot of processing power
- At relatively low cost
- Eg.,



"This computational power is available and inexpensive;...A typical latest-generation card costs $400–500 at release and drops rapidly as new hardware emerges." [Owens et. al 2005] https://onlinelibrary.wiley.com/doi/10.1111/j.1467-8659.2007.01012.x

	Nvidia P100 (2016)	Nvidia V100 (2017)	Nvidia A100 (2020)	Nvidia H100 (2022)	Nvidia B100 (2024)
Cores	3584	5120	6912	14592	16384
Peak INT32 TIPS	10.6	15.7	19.5	40.0	41.0
Peak FP32 TFLOPS	10.6	15.7	19.5	39.0	39.9
Peak FP64 TFLOPS	5.3	7.8	9.7	19.5	19.7
FP8 Tensor PFLOPS	-	-	2	5.8	20
Memory Bandwidth (GB/s)	732	900	1600	3350	3900
Approx. Cost (MSRP, USD)	~9,000	~10,000–12,000	~11,000–15,000	~35,000–40,000	~30,000–40,000*
Peak Power (W)	300	300	400	700	700

Japanese supercomputer, Earth Simulator, cost about $350 million"

ES was the fastest supercomputer in the world from 2002 to 2004.

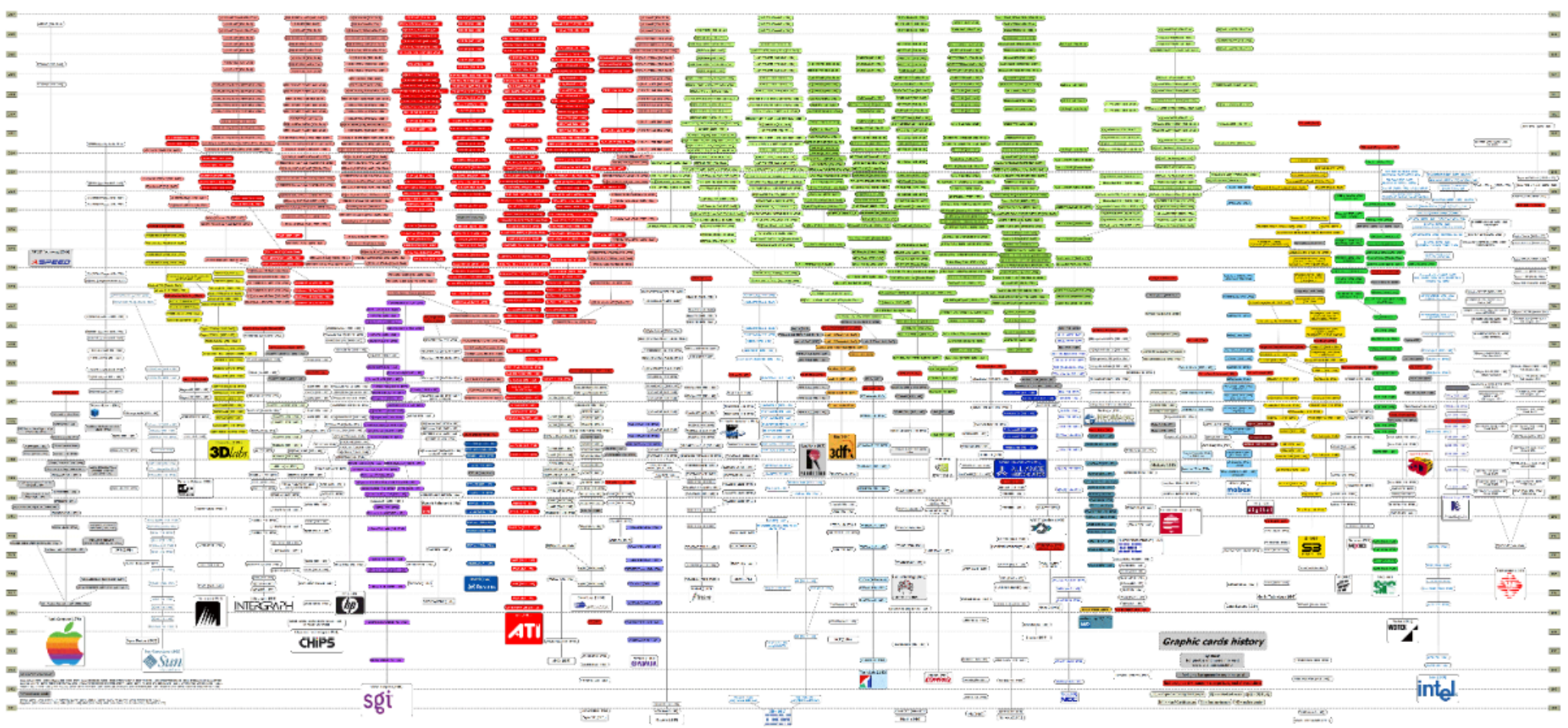
Earth Simulator

Active	2002–2009
Operators	NASDA, JAERI, JAMSTEC
Location	JAMSTEC Yokohama Institute for Earth Sciences
Architecture	640 processor nodes (each consists of 8 vector arithmetic processors) interconnected by single-stage crossbar switches
Operating system	SUPER-UX
Space	65 m × 50 m (213 ft × 164 ft)
Memory	10 TB total
Speed	40 TFLOPS (peak)
Ranking	TOP500: 1, June 2002

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

GPU History : How did we get here?

2017 →

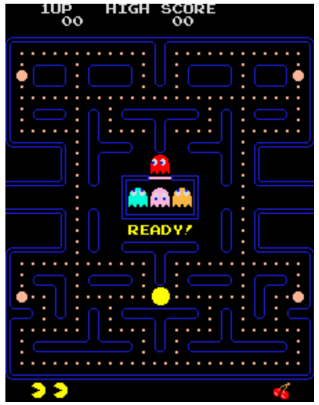


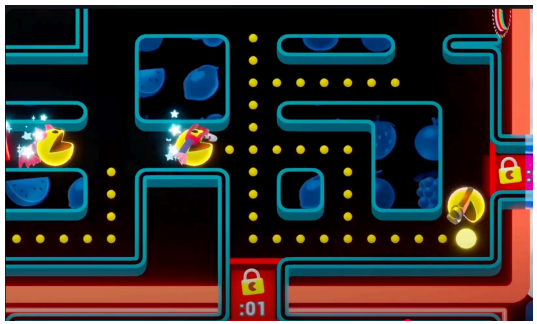
1981 →

https://www.vgamuseum.info/index.php/history-tree

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

GPU Development driven by game playing





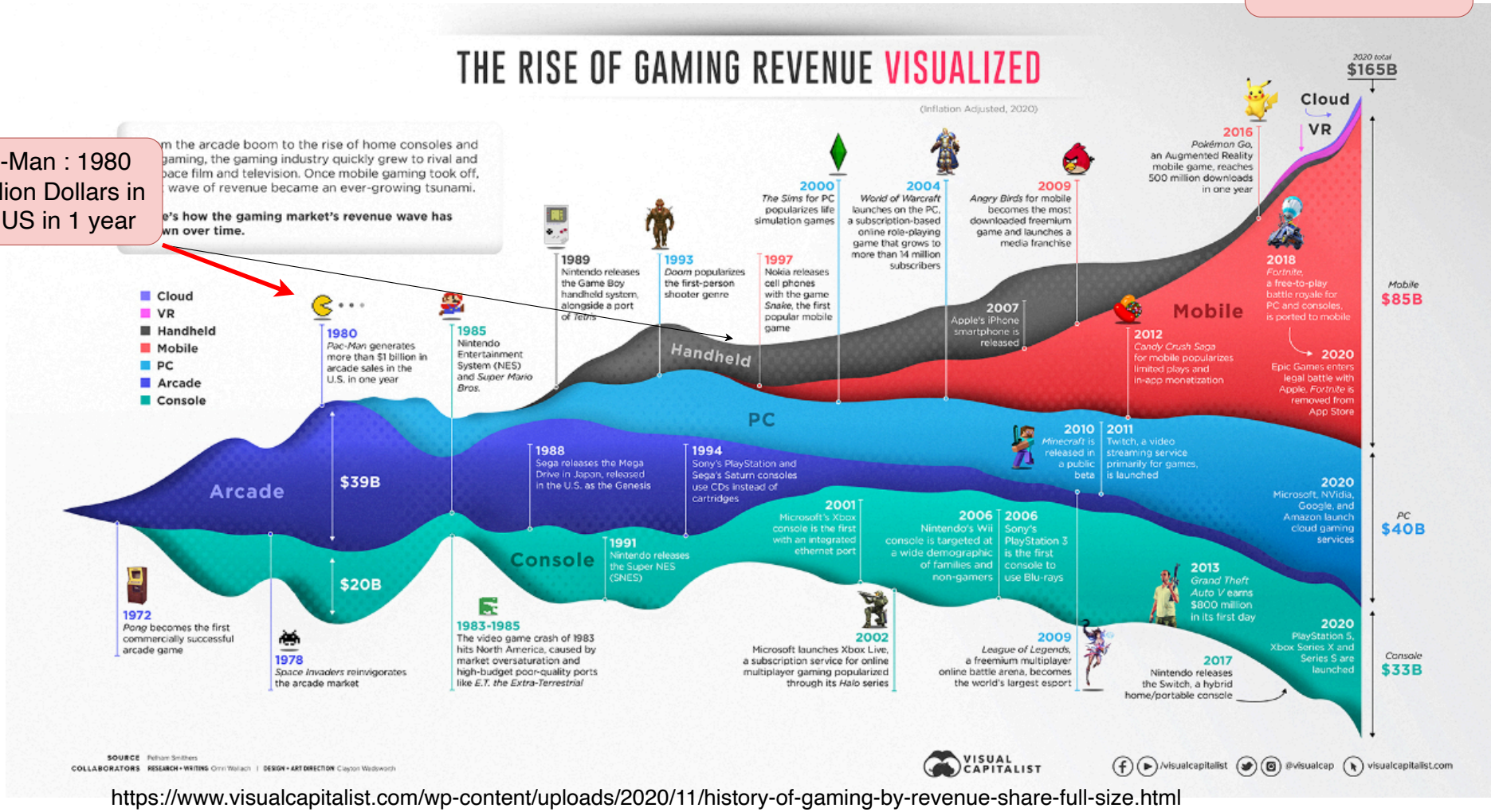
"Game players (and developersGPU parallel performance on throughput problems has doubled every 12 to 18 months, pulled by the insatiable demands of the 3D game market." https://ieeexplore.ieee.org/document/4523358?arnumber=4523358

An enormous industry that drove mass market GPU production:
" GPU market of half a billion units per year" [Owens et. al 2008] https://ieeexplore.ieee.org/document/4490127

It's all about Games (aka Money)

2020: 165 Billion

Pac-Man : 1980
1 Billion Dollars in the US in 1 year



https://www.visualcapitalist.com/wp-content/uploads/2020/11/history-of-gaming-by-revenue-share-full-size.html

Graphics: FP-intensive & "embarrasingly parallelizable"

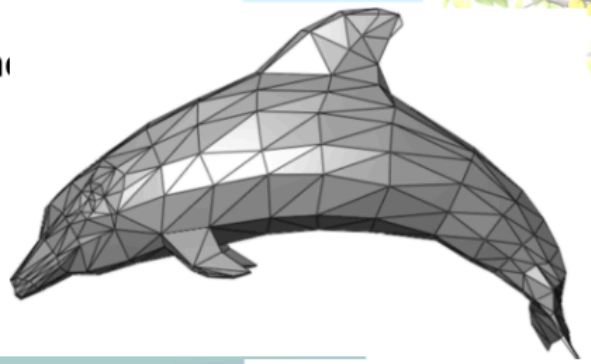
https://en.wikipedia.org/wiki/Ray_tracing_(graphics)

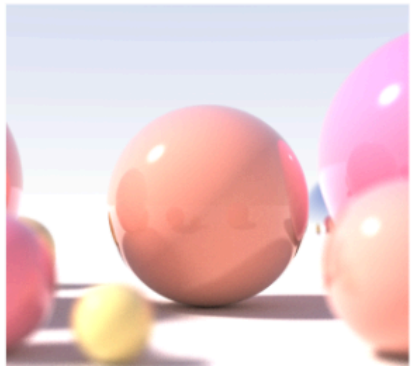
2D:

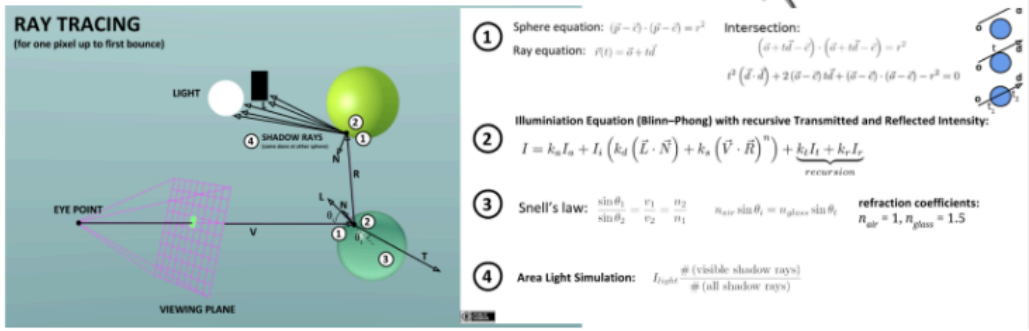
- Shifting & Scaling
- Fading & Filtering
- Rotating: $\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$

3D:

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

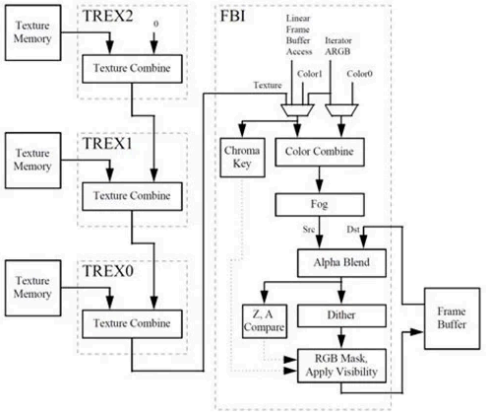






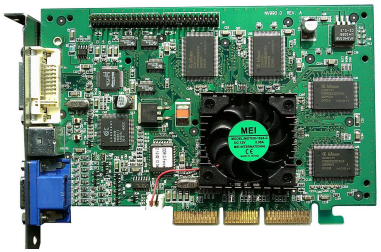
Early GPU's with fixed function pipeline, evolve to support programmable shaders

3Dfx's Voodoo1 1996 (Wikipedia)



https://www.computer.org/publications/tech-news/chasing-pixels/famous-graphics-chips3d

GeForce 256 1999 (Wikipedia)



Application tasks (move objects according to application, move/aim camera)	CPU	CPU	CPU	CPU	3D Application and API
Scene level calculations (object level culling, select detail level, create object mesh)	CPU	CPU	CPU	CPU	
Transform	CPU	CPU	CPU	GPU	3D Graphics Pipeline
Lighting	CPU	CPU	CPU	GPU	
Triangle Setup and Clipping	CPU	Graphics Processor	Graphics Processor	GPU	
Rendering	Graphics Processor	Graphics Processor	Graphics Processor	GPU	
	1996	1997	1998	1999	

https://developer.download.nvidia.com/assets/gamedev/docs/TransformAndLighting.pdf

GeForce FX Series 2002 (Wikipedia)

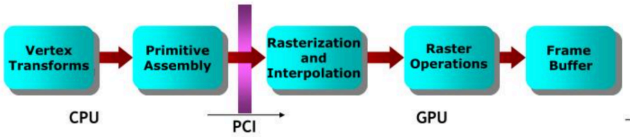
NVIDIA GeForce FX GPUs

Cinematic Computing for Every User

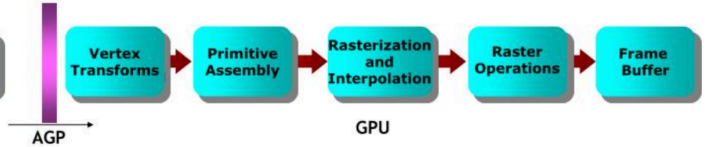
- ***Pixel shader 2.0+.*** DirectX 9.0 exposes true programmability of the pixel-shading engine. This makes procedural shading on a GPU possible for the first time.
- ***Vertex shader 2.0+.*** DirectX 9.0 dramatically enhances the power of the previous DirectX vertex shader by increasing the length and flexibility of vertex programs.
- ***High-precision, floating-point color.*** DirectX 9.0 breaks the mathematical precision barrier that has limited PC graphics in the past. Precision, and therefore visual quality, is increased with 128-bit floating-point color per pixel.
- ***Advanced shader operations in OpenGL through extensions:*** OpenGL uses extensions to enable the use of vertex and pixel shaders in its API. Those extensions are seamlessly integrated into NVIDIA ForceWare graphics drivers.

With Cg and the GeForce FX GPUs, developers have the ability to take full advantage of the API to develop stunning visual effects.

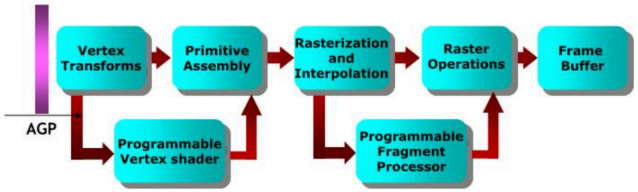
https://www.pny.com/file%20library/company/support/product%20brochures/geforce%20graphics/user%20guides%20and%20tutorials/fxgpu_v1.pdf



https://www.slideserve.com/liam/slide-1-computer-information-science



https://www.slideserve.com/liam/slide-1-computer-information-science



https://www.slideserve.com/liam/slide-1-computer-information-science

The Hack

https://gfxcourses.stanford.edu/cs149/fall24/lecture/gpuarch

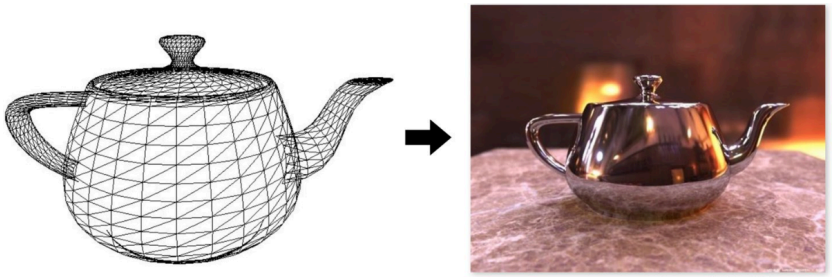
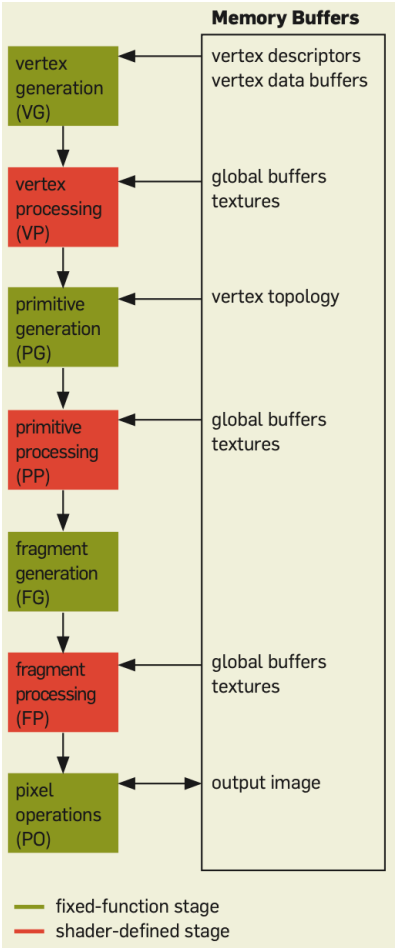


Image credit: Henrik Wann Jensen

1. Given a triangle, determine where it lies on screen, given the position of a virtual camera
2. For all output image pixels covered by the triangle, compute the color of the surface at that triangle



https://dl.acm.org/doi/10.1145/1400181.1400197

Run once per fragment (per pixel covered by a triangle)

OpenGL shading language (GLSL) shader program:
defines behavior of fragment processing stage

```
uniform sampler2D myTexture;
uniform float3 lightDir;
varying vec3 norm;
varying vec2 uv;

void myShader()
{
  vec3 kd = texture2D(myTexture, uv);
  kd *= clamp(dot(lightDir, norm), 0.0, 1.0);
  return vec4(kd, 1.0);
}
```

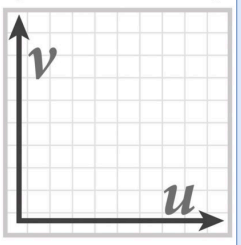
read-only global variables

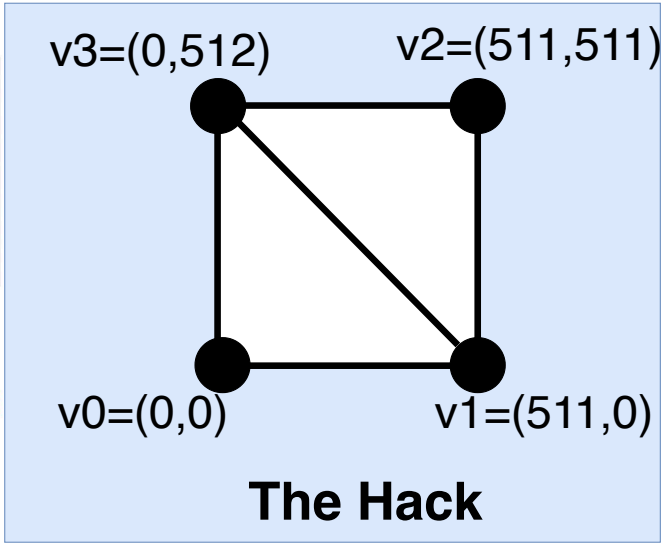
Inputs whose value changes per pixel: think of these as shader function parameters

"Shader" function (a.k.a function invoked to compute the color of the pixel)

per-pixel output: RGBA surface color at pixel

myTexture is a texture map

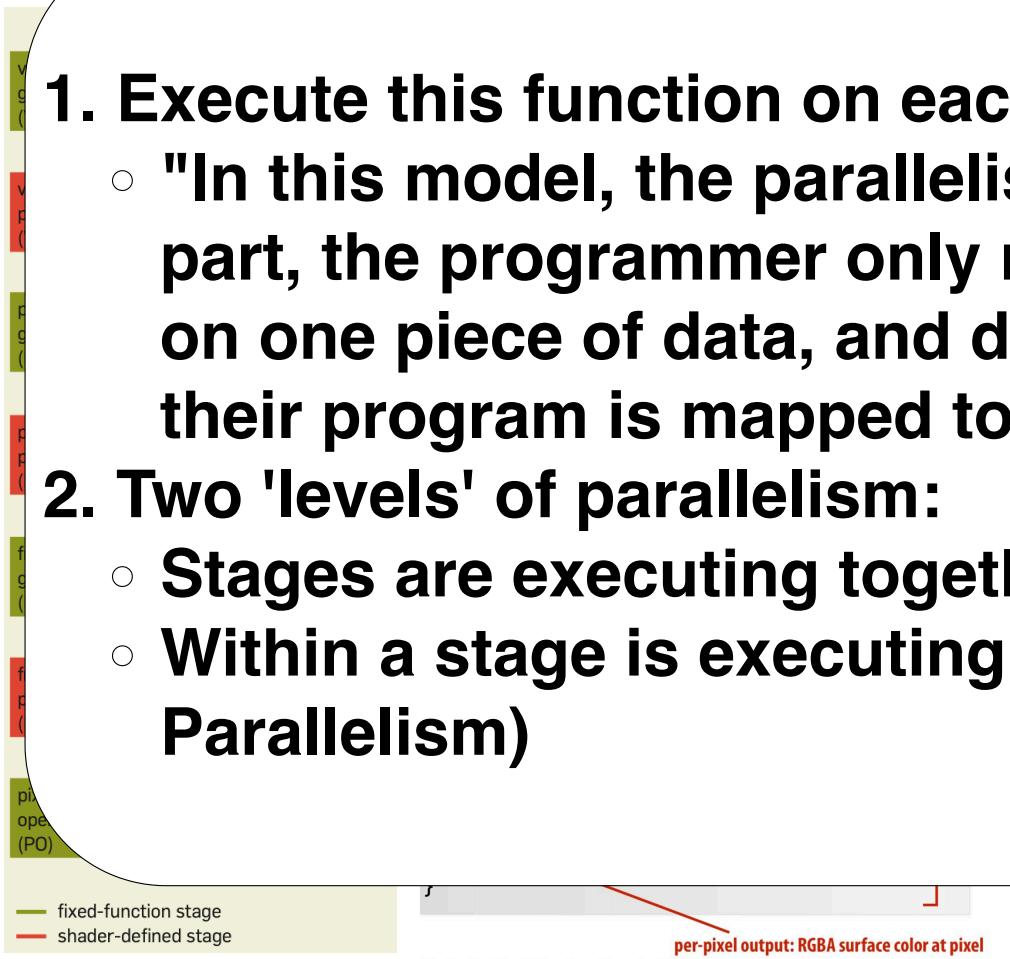




https://gfxcourses.stanford.edu/cs149/fall24/lecture/gpuarch

The Hack

1. Execute this function on each input 'pixel'.
   - "In this model, the parallelism is implicit. For the most part, the programmer only needs to think about operating on one piece of data, and doesn't need to worry about how their program is mapped to hardware." https://pharr.org/matt/blog/2018/04/18/ispc-origins
2. Two 'levels' of parallelism:
   - Stages are executing together (Task Parallelism)
   - Within a stage is executing across pixels (Data Parallelism)



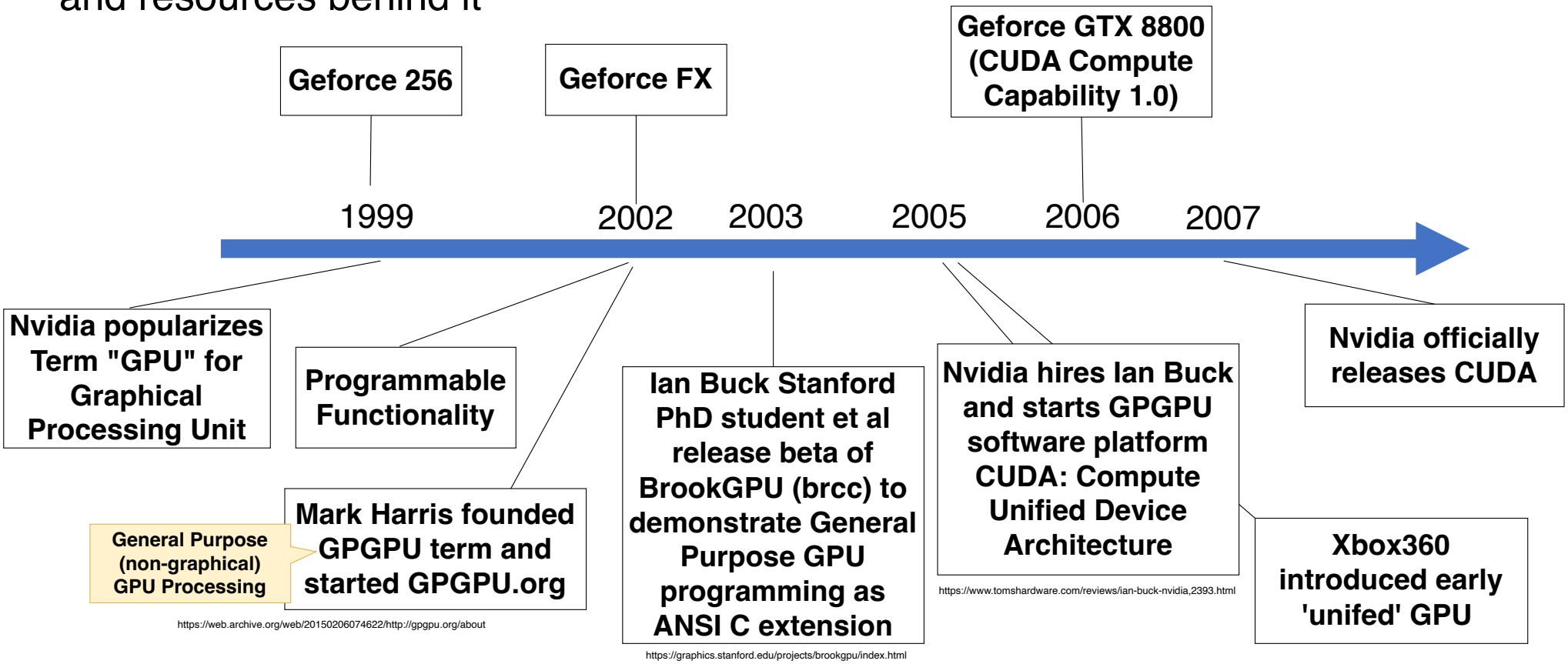
https://dl.acm.org/doi/10.1145/1400181.1400197

The Hack

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

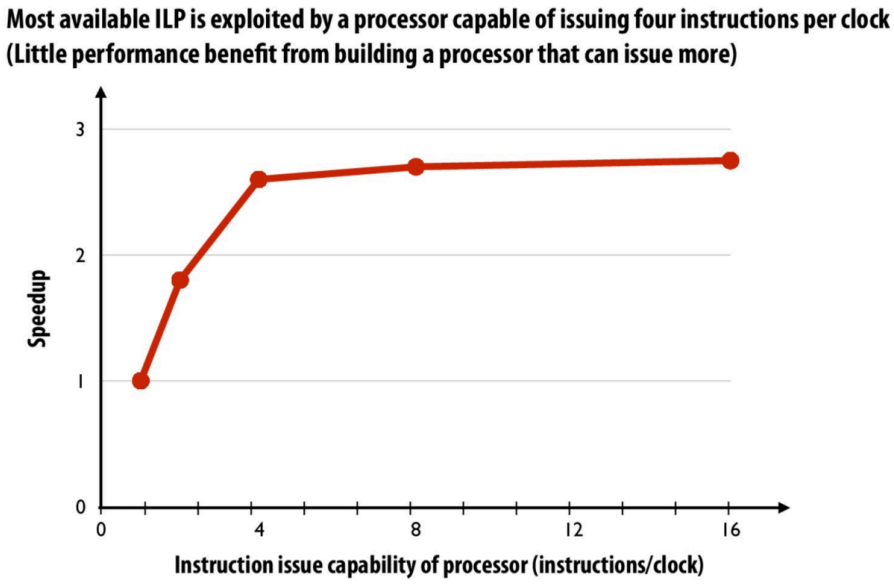
Programmability: Compute Unified Device Architecture

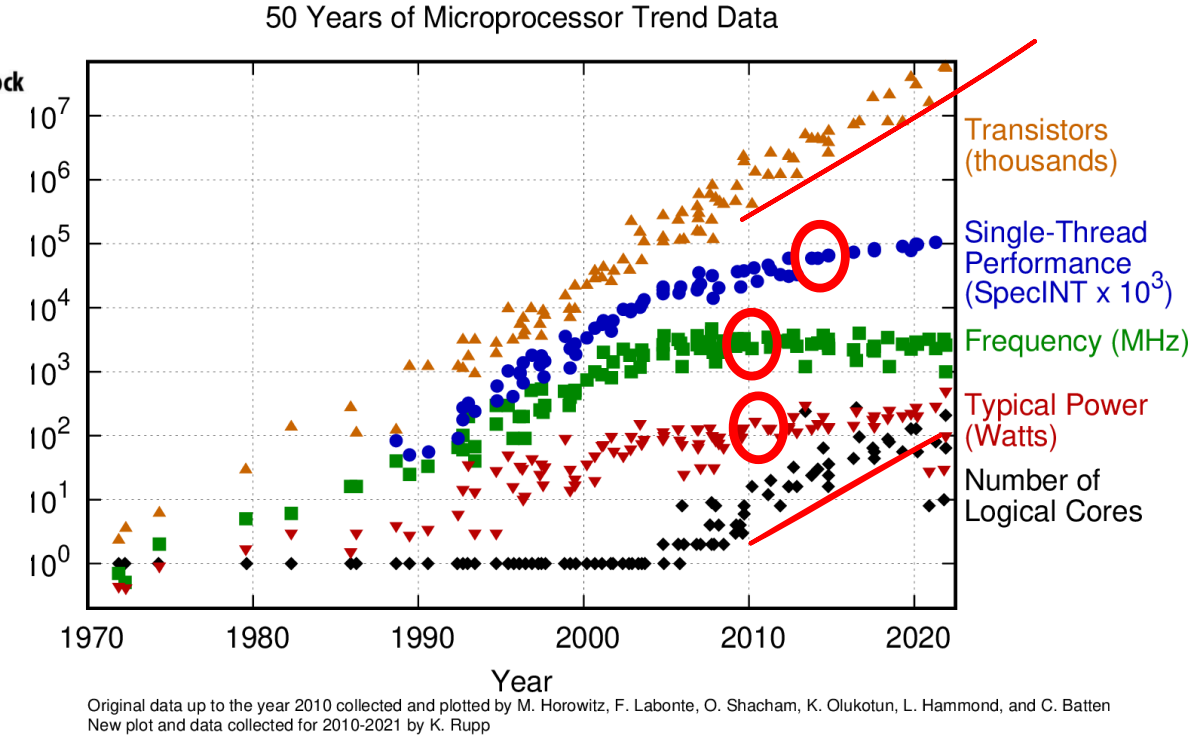
GPU's start being used for more than graphics -- Nvidia notices puts money and resources behind it



Why GPUs

- End of the road for single threaded CPU performance scaling
  - frequency stalled due to "power-wall" can't cool it feasibly
  - ILP: deep pipelining, out-of-order execution -- stalled
- Made parallel computing mainstream TLP (more cores)





GPUs Parallelism to the Extreme

Given origin GPU vendors have been focused on massive parallel computing both the hardware AND software

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

GPUs Parallelism to the Extreme

Given origin GPU vendors have been focused on massive parallel

1. **Computational requirements are large.** Real time rendering requires billions of pixels per second, and each pixel requires hundreds or more operations.
2. **Parallelism is substantial.** (and exibits locality)
3. **Throughput is more important than latency**.
   GPU implementations of the graphics pipeline prioritize throughput over latency. (billions of operations per clock but clock can be slower and individual operations take many cycles)

But bad when calculations involve significant branching

Some Parallel Computing Background

Some key terms and concepts.
For a more complete but brief traditional HPC oriented coverage see Lawrence Livermore National Lab "Introduction to Parallel Computing Tutorial":

https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial

Amdahl's Law

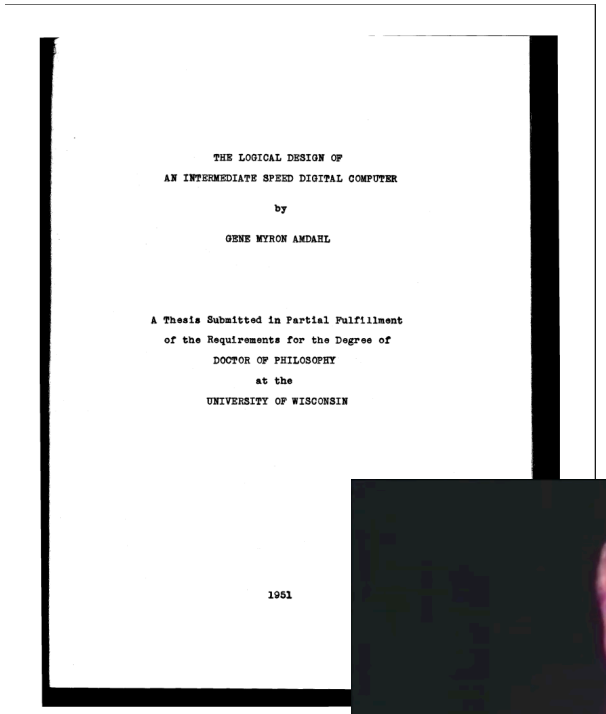
What's the value of adding more procesors?

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

Gene Amdahl (1922-2015)

Tribute to the Peers

https://pages.cs.wisc.edu/~bezenek/Stuff/amdahl_thesis.pdf



THE LOGICAL DESIGN OF
AN INTERMEDIATE SPEED DIGITAL COMPUTER

by

GENE MYRON AMDAHL

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
DOCTOR OF PHILOSOPHY
at the
UNIVERSITY OF WISCONSIN

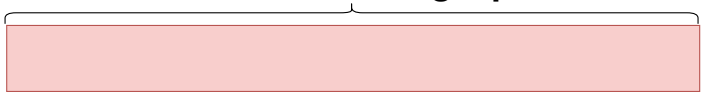
1951

https://youtu.be/Qt05LJW_dgs
Very fun 1983 lecture -- You can learn a lot about ideas & optimizations that we take for granted, and how they were developed

Amdahl's Law

Simple but important Back-of-the-envelope!

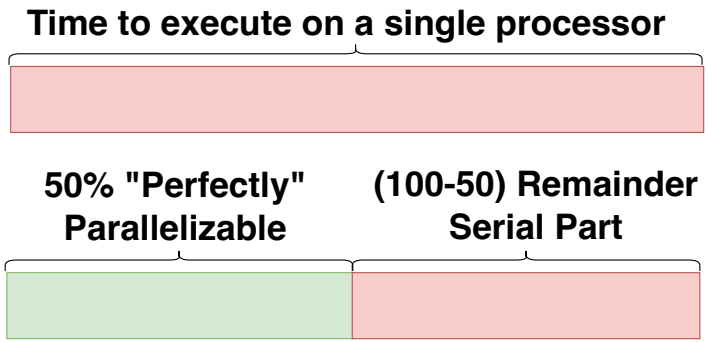
Time to execute on a single processor



Time to execute on a single processor (uni-processor) is α

Amdahl's Law

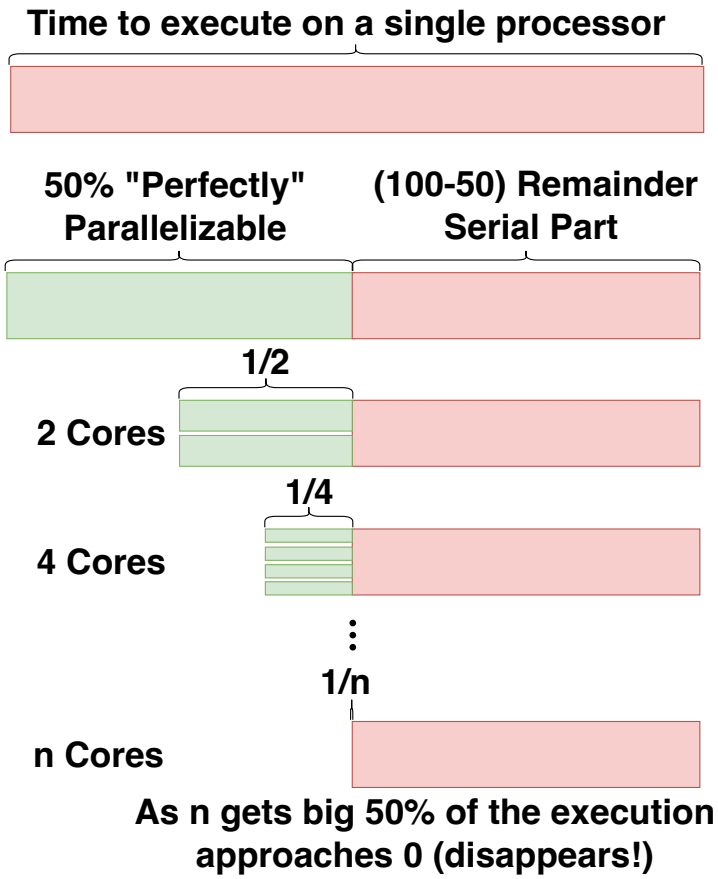
Simple but important Back-of-the-envelope!



Time to execute on a single processor (uni-processor) is α
Assume that P is the faction the execution that is "Perfectly Parallelizable"

Amdahl's Law

Simple but important Back-of-the-envelope!



Time to execute on a single processor (uni-processor) is α

Assume that P is the faction the execution that is "Perfectly Parallelizable" -- time to execute on N processors is reduced by $\frac{1}{N}$

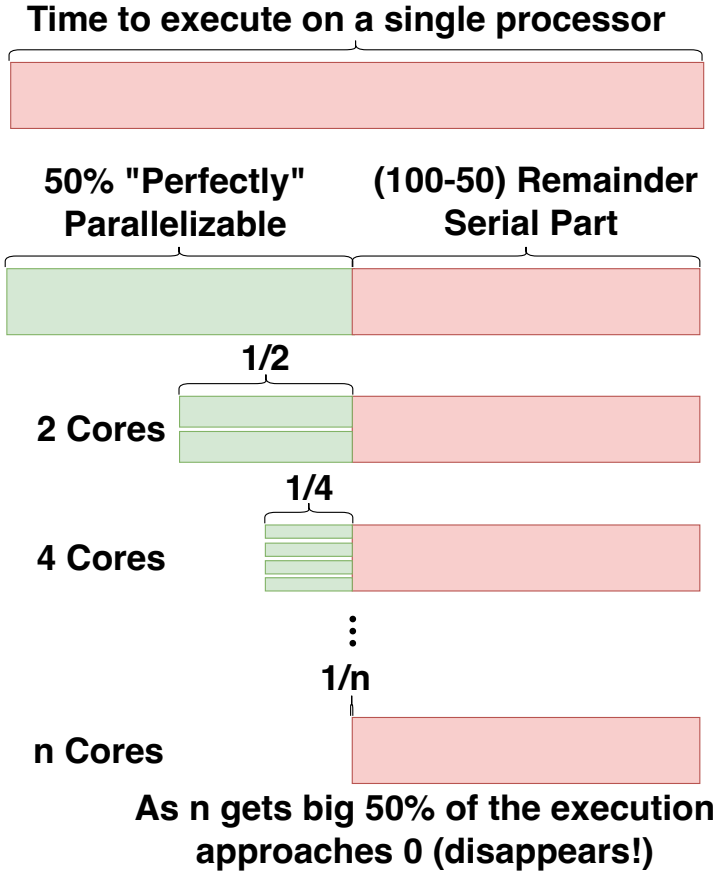
Execution time on N processors is then

$$(\alpha \times \frac{P}{N}) + \alpha \times (1 - P)$$

$$\alpha \times (\frac{P}{N} + (1 - P))$$

Amdahl's Law

Simple but important Back-of-the-envelope!



Time to execute on a single processor (uni-processor) is α

Assume that P is the faction the execution that is "Perfectly Parallelizable" -- time to execute on N processors is reduced by $\frac{1}{N}$

Execution time on N processors is then

$$(\alpha \times \frac{P}{N}) + \alpha \times (1 - P)$$

$$\alpha \times (\frac{P}{N} + (1 - P))$$

Speedup is defined to be the uniprocessor time divided by the time to execute on N processors

$$\text{speedup} = \frac{\alpha}{\alpha \times (\frac{P}{N} + (1 - P))}$$

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

$$\lim_{N \to \infty} \text{speedup} = \frac{1}{1 - P}$$

Amdahl's Law

It's worth thinking about

Case 1: A computation that takes 1 day

P	N=1	N=2	N=8	N=128	N=512	N=16384
0.50	24.00	18.00	13.50	12.09	12.02	12.00
0.75	24.00	15.00	8.25	6.14	6.04	6.00
0.90	24.00	13.20	5.10	2.57	2.44	2.40
0.95	24.00	12.60	4.05	1.38	1.24	1.20
0.99	24.00	12.12	3.21	0.43	0.29	0.24

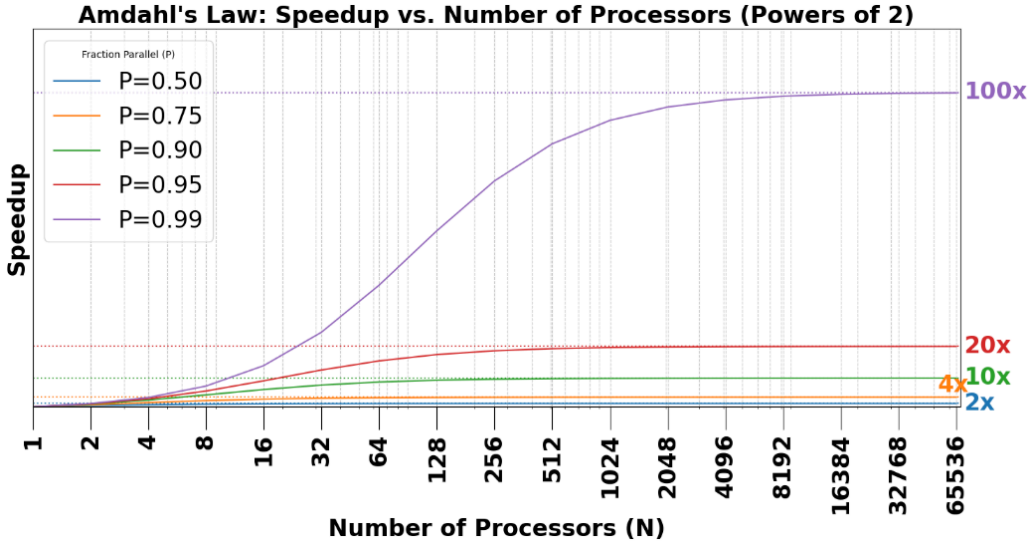
less than half an hour

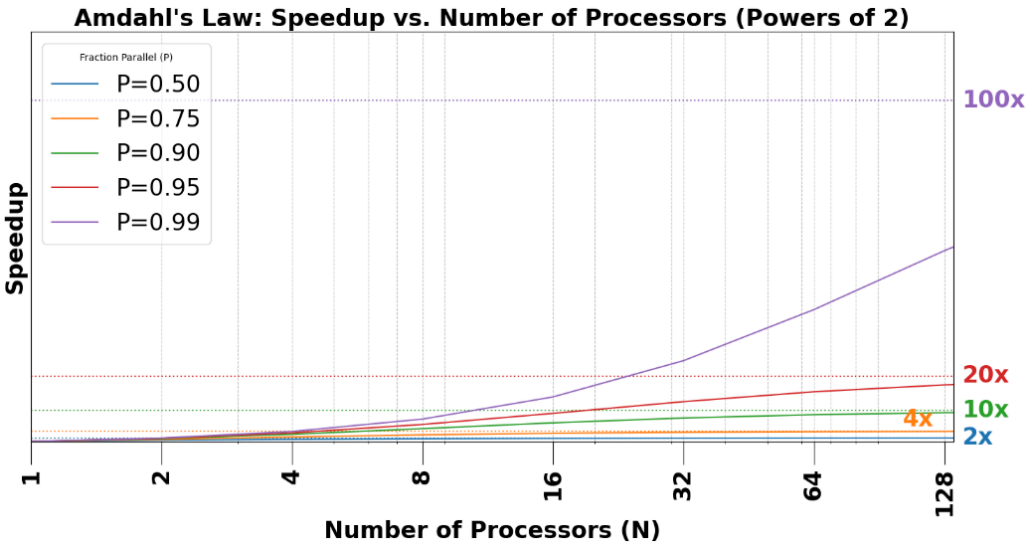
Case 1: A computation that takes 10s

P	N=1	N=2	N=8	N=128	N=512	N=16384
0.50	10.000s	7.500s	5.625s	5.039s	5.010s	5.000s
0.75	10.000s	6.250s	3.438s	2.559s	2.515s	2.500s
0.90	10.000s	5.500s	2.125s	1.070s	1.018s	1.001s
0.95	10.000s	5.250s	1.688s	0.574s	0.519s	0.501s
0.99	10.000s	5.050s	1.337s	0.177s	0.119s	0.101s

under a second

a blink of an eye





Amdahl's Law

"Key Implications and Takeaways"

1. **The Bottleneck is the Sequential Part:** Even a small percentage of serial/seqential code can severely limit the overall speedup.
2. **Diminishing Returns:** As the number of processors N increases, the benefit of adding more processors decreases.
3. **Focus on Increasing** P: The most effective way to achieve a significant speedup is often to optimize the serial/sequential portion of the code, thereby increasing P rather than just adding more processors.
4. **Not a Guarantee**: Amdahl's Law provides a theoretical maximum speedup. Real-world parallelization introduces overheads (e.g., communication between processors, load imbalance, I/O, synchronization) : **Speedup will always be less than predicted by Amdahl's law**.

Text of this slide was developed with the help of Gemini 2.5 Flash -- GenAI

Strong vs Weak Scaling

Gustafson-Barsis's Law

"Problem Size" & "Embarrassingly Parallel"

https://dl.acm.org/doi/pdf/10.1145/42411.42415

REEVALUATING AMDAHL'S LAW

JOHN L. GUSTAFSON

At Sandia National Laboratories, we are currently engaged in research involving massively parallel processing. There is considerable skepticism regarding the viability of massive parallelism; the skepticism centers around *Amdahl's law*, an argument put forth by Gene Amdahl in 1967 [1] that even when the fraction of serial work in a given problem is small, say, s, the maximum speedup obtainable from even an infinite number of parallel processors is only $1/s$. We now have timing results for a 1024-processor system that demonstrate that the assumptions underlying Amdahl's 1967 argument are inappropriate for the current approach to massive ensemble parallelism.

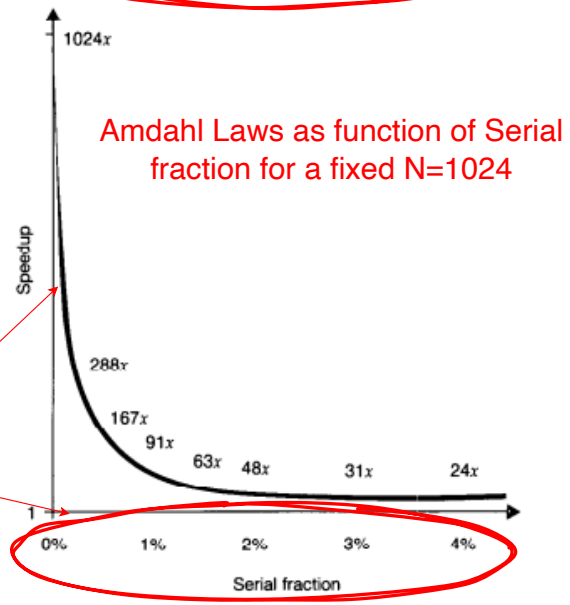
If N is the number of processors, s is the amount of time spent (by a serial processor) on serial parts of a program, and p is the amount of time spent (by a serial processor) on parts of the program that can be done in parallel, then Amdahl's law says that speedup is given by

$$Speedup = (s + p)/(s + p/N)$$
$$= 1/(s + p/N),$$

where we have set total time $s + p = 1$ for algebraic simplicity. For $N = 1024$ this is an unforgivingly steep function of s near $s = 0$ (see Figure 1).

The steepness of the graph near $s = 0$ (approximately $-N^2$) implies that very few problems will experience even a 100-fold speedup. Yet, for three very practical applications ($s = 0.4$–0.8 percent) used at Sandia, we have achieved speedup factors on a 1024-processor hypercube that we believe are unprecedented [2]: *1021* for beam stress analysis using conjugate gradients, *1020* for baffled surface wave simulation using explicit finite differences, and *1016* for unstable fluid flow using flux-corrected transport. How can this be, when Amdahl's argument would predict otherwise?

The expression and graph both contain the implicit assumption that p is independent of N, which *is virtually never the case*. One does not take a fixed-sized problem and run it on various numbers of processors



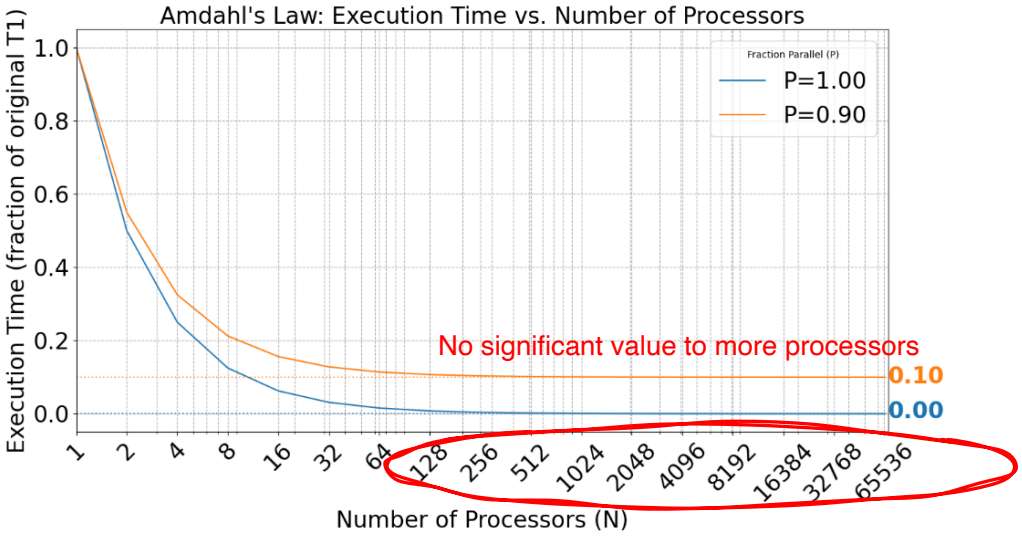
Amdahl Laws as function of Serial fraction for a fixed N=1024

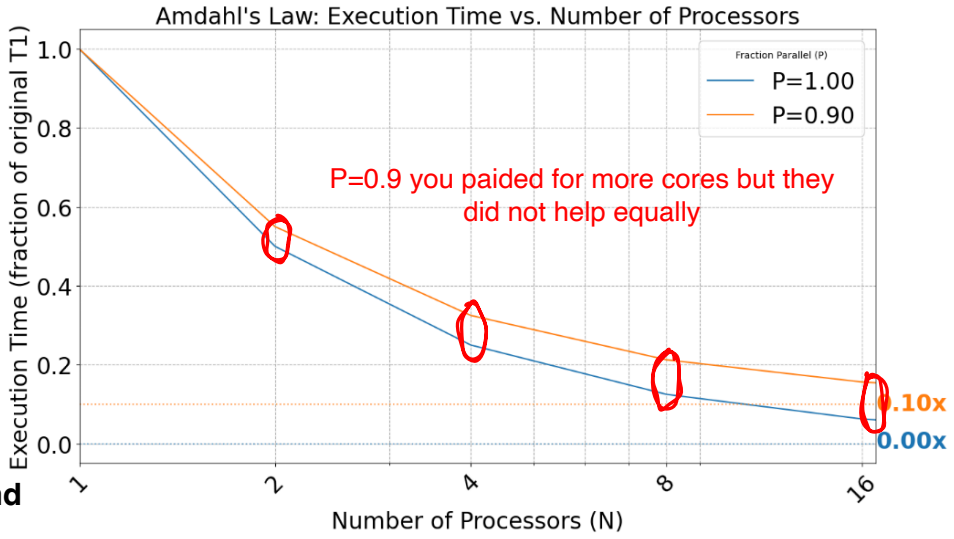
FIGURE 1. Speedup under Amdahl's Law

Strong Scaling

Amdhal's Law & Execution Time

- Given a "Fixed Problem Size", how fast can you execute it as a function of the number of processors
- What is the value of adding/buying more processors?
- Metric is speedup
- Ideal is linear speedup : $P = 1.0$
- But even small Serial portion $P = 0.90, S = 1 - P = 0.10$ never disappears and dominates no matter how large N gets





While dismissed by Gustafson et al it important in general purpose and commercial computing ... time is money -- more hardware == faster

Gustafson-Barsis's Law

Scaled Speedup

- Strong Scaling, fixed problem size, sequential portion asymptotically dominates. The value of more processors is limited
- Gustafson's observation based on Scientific computing at Sandia National Labs (HPC Research) -- work on "massive" parallel systems (1024 processors)
  - Refute Amdahl's Law implication that the value of more processors is faster execution
  - Want a measure that reflects the practice of using more processors to do more work in the same amount of time
  - More processors can work on "bigger problems"
- Scaled Speedup -- scale base case (numerator) with problem size

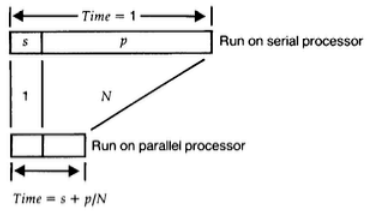


FIGURE 2a. Fixed-Sized Model for $Speedup = 1/(s + p/N)$

$$\begin{aligned} Scaled\ speedup &= (s + p \times N)/(s + p) \\ &= s + p \times N \\ &= N + (1 - N) \times s. \end{aligned}$$

In contrast with Figure 1, this function is simply a *line*, and one with a much more moderate slope: $1 - N$. It is thus much easier to achieve efficient parallel performance than is implied by Amdahl's paradigm. The two approaches, fixed sized and scaled sized, are contrasted and summarized in Figure 2a and b.

Our work to date shows that it is *not* an insurmountable task to extract very high efficiency from a massively parallel ensemble, for the reasons presented here. We feel that it is important for the computing

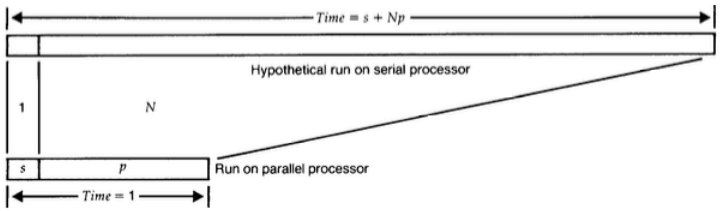


FIGURE 2b. Scaled-Sized Model for $Speedup = s + Np$

except when doing academic research; in practice, *the problem size scales with the number of processors*. When given a more powerful processor, the problem generally expands to make use of the increased facilities. Users have control over such things as grid resolution, number of time steps, difference operator complexity, and other parameters that are usually adjusted to allow the program to be run in some desired amount of time. Hence, it may be most realistic to assume *run time*, not *problem size*, is constant.

As a first approximation, we have found that it is the *parallel* or *vector* part of a program that scales with the problem size. Times for vector start-up, program loading, serial bottlenecks, and I/O that make up the s component of the run do *not* grow with problem size. When we double the number of degrees of freedom in a physical simulation, we double the number of processors. But this means that, as a first approximation, the amount of work that can be done in parallel *varies linearly with the number of processors*. For the three applications mentioned above, we found that the parallel portion scaled by factors of 1023.9969, 1023.9965, and 1023.9965. If we use s and p to represent serial and parallel time spent on the *parallel* system, then a serial processor would require time $s + p \times N$ to perform the task. This reasoning gives an alternative to Amdahl's law suggested by E. Barsis at Sandia:

research community to overcome the "mental block" against massive parallelism imposed by a misuse of Amdahl's speedup formula; speedup should be measured by scaling the problem to the number of processors, not by fixing problem size. We expect to extend our success to a broader range of applications and even larger values for N.

REFERENCES

1. Amdahl, G.M. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, vol. 30 (Atlantic City, N.J., Apr. 18–20). AFIPS Press, Reston, Va., 1967, pp. 483–485.
2. Benner, R.E., Gustafson, J.L., and Montry, R.E. Development and analysis of scientific application programs on a 1024-processor hypercube. SAND 88-0317, Sandia National Laboratories, Albuquerque, N.M., Feb. 1988.

CR Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*parallel processors*

General Terms: Theory

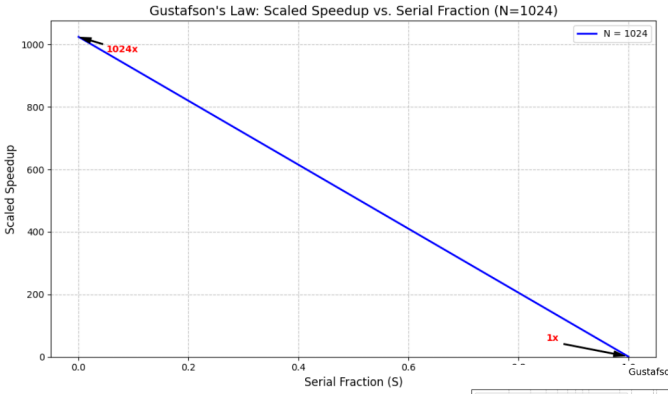
Additional Key Words and Phrases: Amdahl's law, massively parallel processing, speedup

Author's Present Address: John L. Gustafson, Sandia National Laboratories, Albuquerque, NM 87185.

Weak Scaling

Gustafson-Barsis's Law

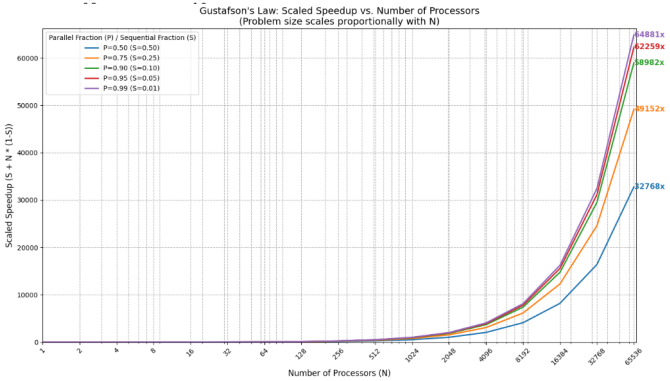
- Add an equal amount of work for each new processor
- This new work is done in parallel
- So execution time remains constant
- Work done scales with the number of processors
- As long as there is some parallel portion, adding processors is useful!

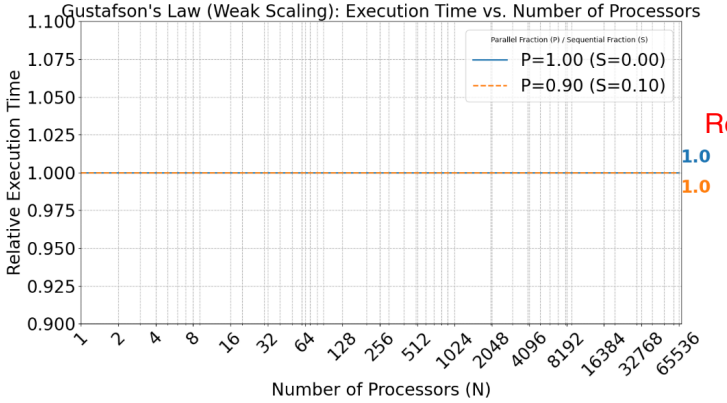


Scaled Speedup as a function of Serial Fraction for fixed N=1024. The numerator is scaled

$$s + (p \times N)$$

Scaled Speedup as a function of N for $p = 0.5, 0.75, 0.9, 0.95, 0.99$
As N grows, Scaled Speedup improves





Relative Execution Time as a function of N is Constant!

Gotcha's

Buyer be ware

- Both Amdahl's Law and Gustafson-Barsis's Law have assumptions that are theoretical in nature
  - You would rarely use the same implementation or algorithm for small N and larger N -- fraction serial vs parallel is not always easy to identify
  - There are lots of potential sources for overheads that grow with N, **even if you are scaling the work**
    - Inter-processor communication
    - I/O bandwidth
    - Memory capacity
    - Memory bandwidth
    - Synchronization overheads
    - Load imbalances

"Times for vector start-up, program load-
ing, serial bottlenecks, and I/O that make up the s
component of the run do not grow with problem size."

Bottom Line

Summary: Strong Scaling vs. Weak Scaling

Feature	Strong Scaling	Weak Scaling
Problem Size	Fixed (constant total work)	Scaled (fixed work per processor; total work grows with N)
Goal	Solve the same problem faster	Solve a larger problem in roughly the same time
Related Law	Amdahl's Law	Gustafson-Barsis's Law
Ideal Plot	Execution Time vs. Processors (downward curve) or Speedup vs. Processors (upward, flattening curve)	Execution Time vs. Processors (flat line)
Key Limitation	Sequential portion of the code	Communication overhead, load imbalance

Table was developed with the help of Gemini 2.5 Flash -- GenAI

Embarrassingly Parallel

"Some computational problems are 'embarrassingly parallel': they can easily be divided into components that can be executed concurrently. Such problems sometimes arise in scientific computing or in graphics, but rarely in systems."

Herlihy & Shavit, "The Art of Multiprocessor Programming", 2008 Elsevier, Chapter 1, p 14

Flynn's Taxonomy

tively on each of two alternate task paths [14]. This is a direct "branch" to "composition" transformation.

System Organizations and Their Effectiveness in Resource Use

SISD and Stream Inertia

Serious inefficiencies may arise in confluent SISD organizations due to turbulence when data interacts with the instruction stream. Thus an instruction may require an argument that is not yet available from a preceding instruction (direct composition), or (more seriously) an address calculation is incomplete (indirect composition). Alternately, when a data conditional branch is issued, testable data must be available before the branch can be fully executed (although both paths can be prepared). In conventional programs delays due to branching usually dominate the other considerations; then in the following we make the simplified assumption that inertia delay is due exclusively to branching. In addition to providing a simple model of performance degradation, we can relate it to certain test data derived from a recent study. The reader should beware, however, that elimination of branching by the introduction of composition delay will not alter the turbulence situation. For a machine issuing an instruction per Δt, if the data interaction with the stream is determined by the ith instruction and the usage of such data is required by the $(i+1)$th instruction, a condition of maximum turbulence is encountered, and this generates the maximum serial latency time $(L-1)\Delta t$ which must be inserted into the stream until the overlapped issuing conditions can be reestablished.

If we treat the expected serial latency time of an instruction $L\cdot\Delta t$ as being equivalent to the total execution time for the average instruction (from initiation to completion), we must also consider an anticipation factor N as the average number of instructions between (inclusively) the instruction stating a condition and the instruction which tests or uses this result. Clearly, for $N \geq J/L$ instructions no turbulence (or delay) will result.

Thus under ideal conditions one instruction each $L\cdot\Delta t/J$ (time units) would be executed. Turbulence adds a delay:

$$\text{delay} = \left(L - \frac{NL}{J}\right)\Delta t, \qquad \text{for } N < \frac{J}{L}$$
$$= 0, \qquad \text{for } N \geq \frac{J}{L}.$$

Given a block of M instructions with a probability of encountering a turbulence causing instruction p the total time to execute these instructions would be

$$T = \left[\frac{L}{J}[M(1-p)] + 1 + pM\left(L - \frac{NL}{J}\right)\right]\Delta t.$$

The additional "1" in the expression is due to the startup of the instruction overlapping. If we define performance as the number of instructions executed per unit time, then

$$\text{perf.} = \frac{M}{T} = \frac{M}{\left[\frac{L}{J}[M(1-p)] + pM\left(L - \frac{NL}{J}\right) + 1\right]\Delta t}$$
$$= \frac{1}{\frac{L\cdot\Delta t}{J}\left[(1-p) + p(J-N) + \frac{J}{L\cdot M}\right]}.$$

The last term in the denominator drops out as M becomes large. Then

$$\text{perf.} = \frac{J}{L\Delta t}\cdot\frac{1}{[1 + p(J - N - 1)]}.$$

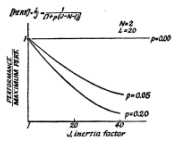
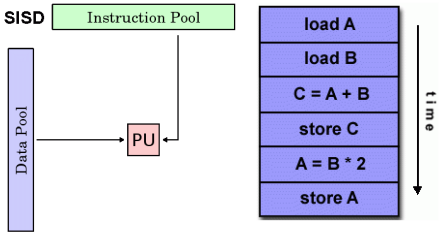


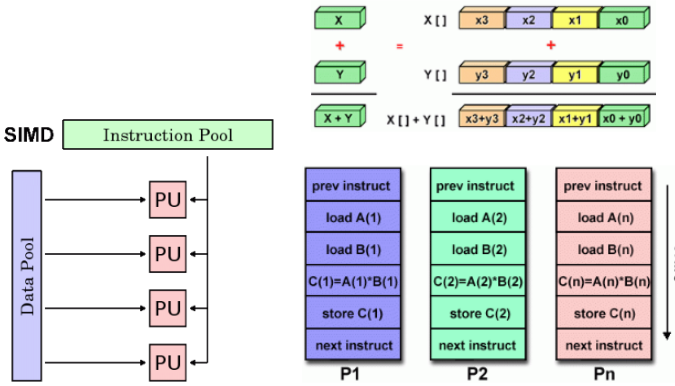
Fig. 4. Stream inertia and effects of branch.

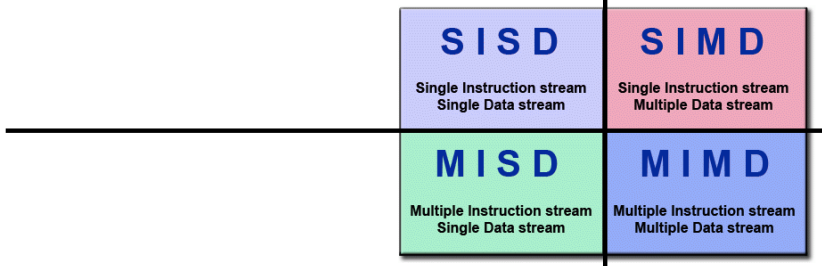
Fig. 4 shows the effect of turbulence probability p for various J and L. In particular, if $J=20$ and L were 20 time units, $N=2$ instructions and a turbulence probability of 10 percent, the performance of a system would degrade from a maximum possible of 1 (instructions/time unit) to 1/2.7 or about 30 percent of its potential.

A major cause of turbulence in conventional programs is the conditional branch; typically the output of a computer would include 10–20-percent conditional branches. A study by O'Regan [12] on the branch problem was made using the foregoing type of analysis. For a typically scientific problem mix (five problems: root finding; ordinary differential equation; partial differential equations; matrix inversion; and Polish string manipulation), O'Regan attempted to eliminate as many data conditional branches as possible using a variety of processor architectures (single and multiple accumulators, etc.). O'Regan did not resort to extreme tactics, however, such as multiplying loop sizes or transformations to a Boolean test (mentioned earlier). For four of the problems selected the best (i.e., minimum) conditional branch probability attainable varied from $p=0.02$ to $p=0.10$. The partial differential equation results depend on grid size, but $p=0.001$ seems attainable. The best (largest) attainable N (the set-test offset) average was less than 3. No attempt was made in the study to evaluate degradation due to other than branch dependent turbulence.

https://users.cs.utah.edu/~hari/teaching/paralg/Flynn72.pdf

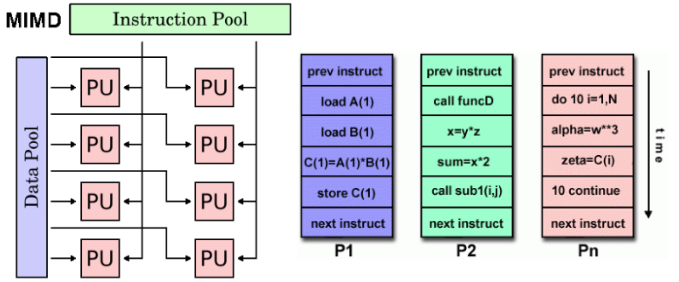






Nvidia GPU's are a variant called SIMT vs SIMD Vector processors

Standard General Purpose Multiprocessors
Your Laptop and Most Servers



1. Error checking through redundant computation
2. **Maybe Systolic Array**?

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

https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial

Data Parallel (vs Task Parallel)

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

"...parallelism comes for the simultaneous operations across large sets of data, rather from multiple threads of control.", Data Parallel Algorithms, Hillis & Steele, 1986

"...independent evaluation of different pieces of data is the basis of data parallelism. Writing data parallel code entails (re)organizing the computation around the data...", Hwu, et al. PMPP

- Vector Addition,
- Matrix multiplication,
- Image/Video procesing (graphics),
- Monte Carlo simulation,
- Map-Reduce,
- Finite difference methods,
- N-body simulations,
- Fuild dynamics,
- Moleculer dynamics,
- SQL table operations,
- Radix sort,
- **Tensor Compuations (Neural Networks/Deep Learning/ML/AI)**

Some Observations

Data Parallel = Weak Scaling + Embarrassingly Parallelism

- General-purpose computing is not like this ... thought to be the domain of HPC -- graphics and scientific computing
- But....
  - Gaming is a massive industry
    - Motivating mass production of Data Parallel Hardware (GPU's)
  - Access to Data Parallel Hardware allows Neural Network Research and AI/ML to rapidly make advances... as it too is Data Parallel
  - All of a sudden, it is no longer a niche

"GPUs were created for 3D rendering, an embarrassingly parallel application. Massive parallelism is a part of the GPU's core programming model." Ian Buck
https://www.tomshardware.com/reviews/ian-buck-nvidia,2393-2.html

Some Observations

Da **ism**

- C IPC
- E
  - ○
  - ○
  - ○

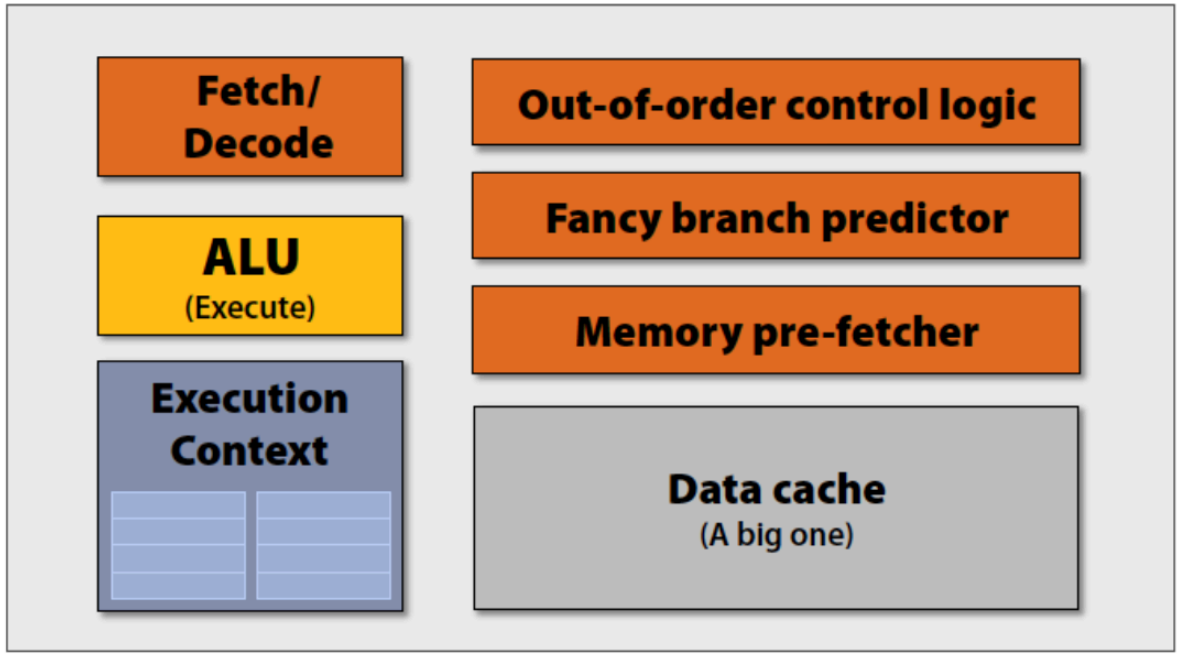
"GPUs were created
part of the GPU's co
https://www.tomshard

Abstraction and Software matters and is part of the reason for GPU success:

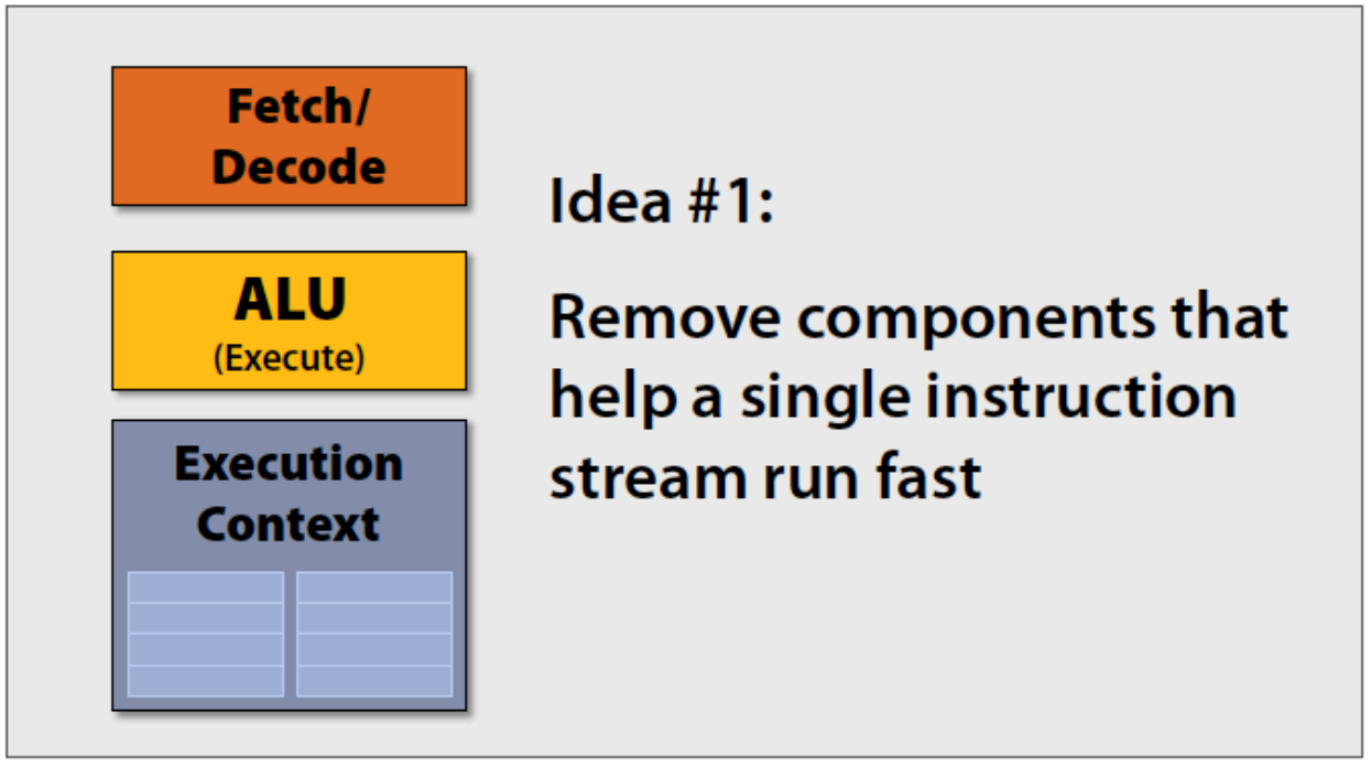
- Writting high performance parallel code is still hard
- but the abstractions of a language like CUDA guide programmers in writting code that "hopefully" scale with new future hardware
- **The Holy Grail**: buy new hardware and get proportionally better performance -- at least that's the dream/marketing.
  - Ride the the transistor scaling curve.

At a high-level, the road from CPU to GPU

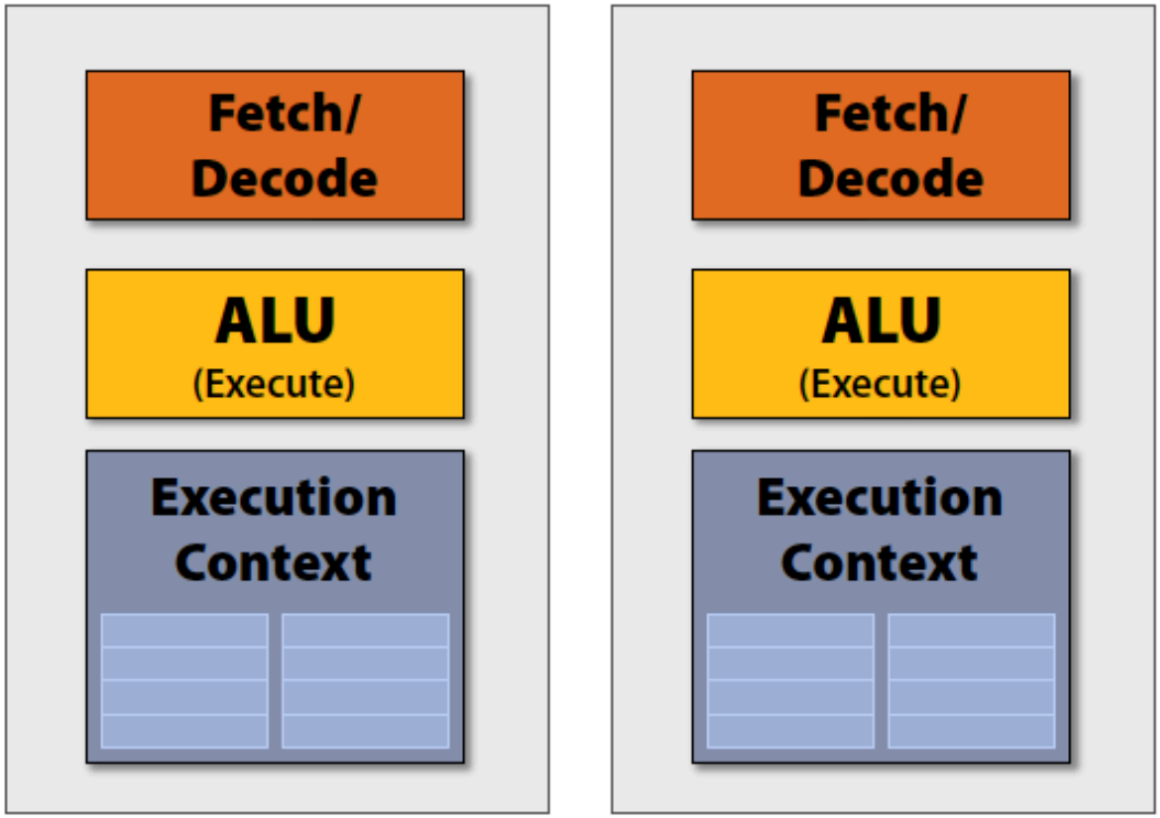
CPU-style cores



GPUs slim it down

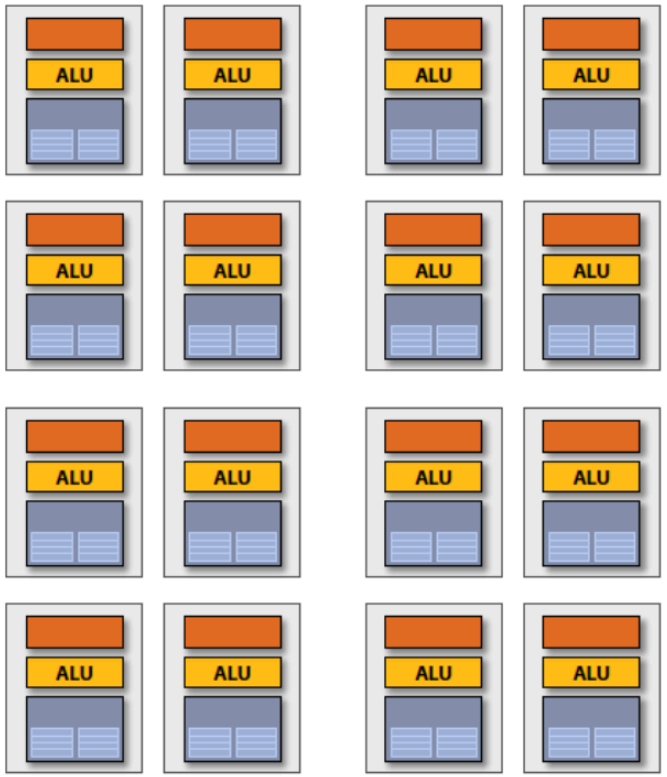


Double # cores with freed-up space



In fact: 16 cores fit in the ~ same space

- 16 independent instruction streams
- But in reality, the streams we will execute will often not be that different/independent

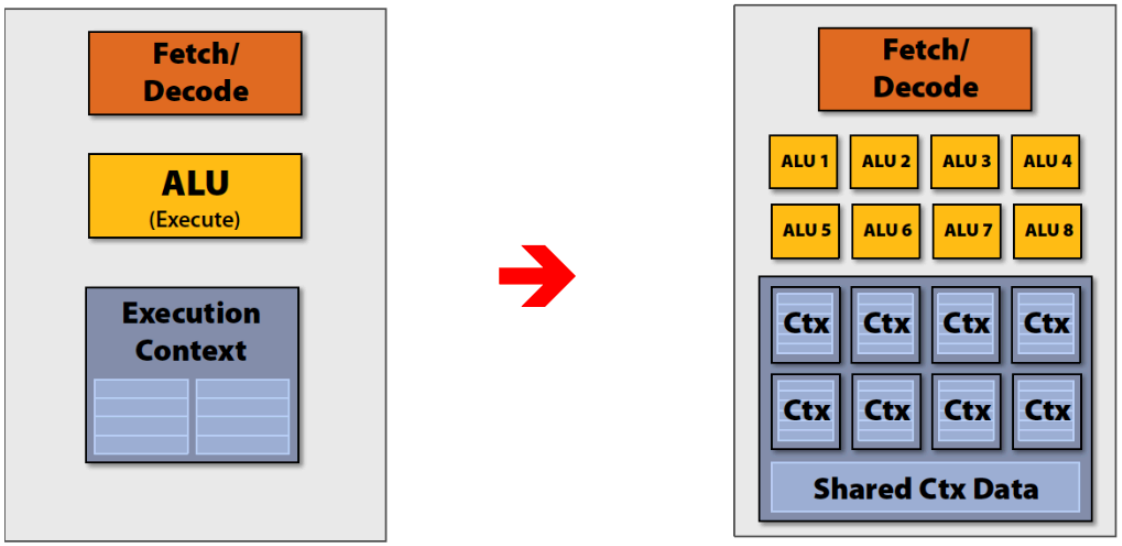


Do not take the specific numbers too seriously; the main point is that a manufacturer chooses to forego complex cores with a larger number of simpler cores.

Saving more space

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

➔ SIMD processing



In Flynn's original taxonomy, a modern GPU would be a type of SIMD called an Array Processor -- today called a SIMT Single Instruction Multiple Threads

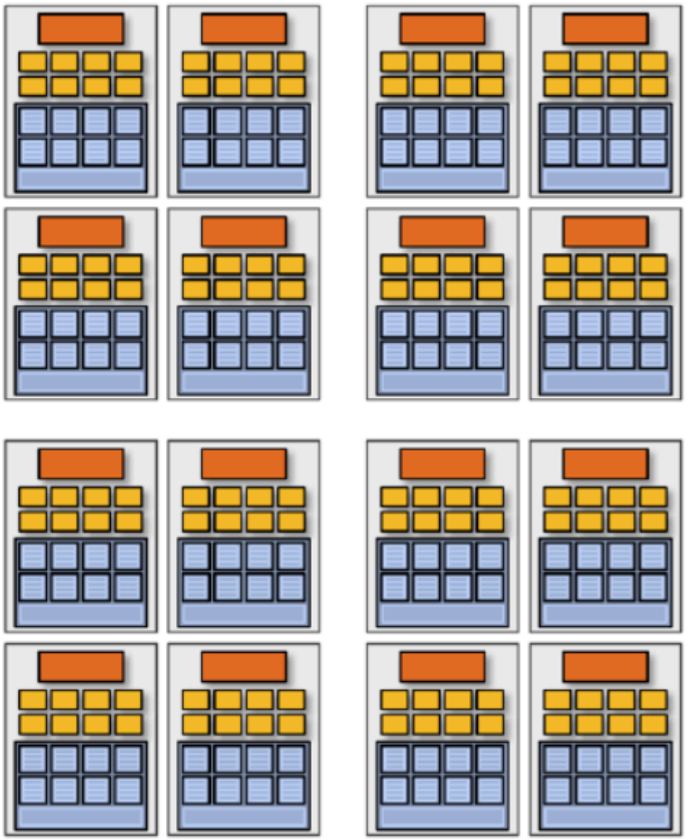
Now much more compute "power" in same space

But with constraints ... don't get fooled

Example:

128 instruction streams in parallel

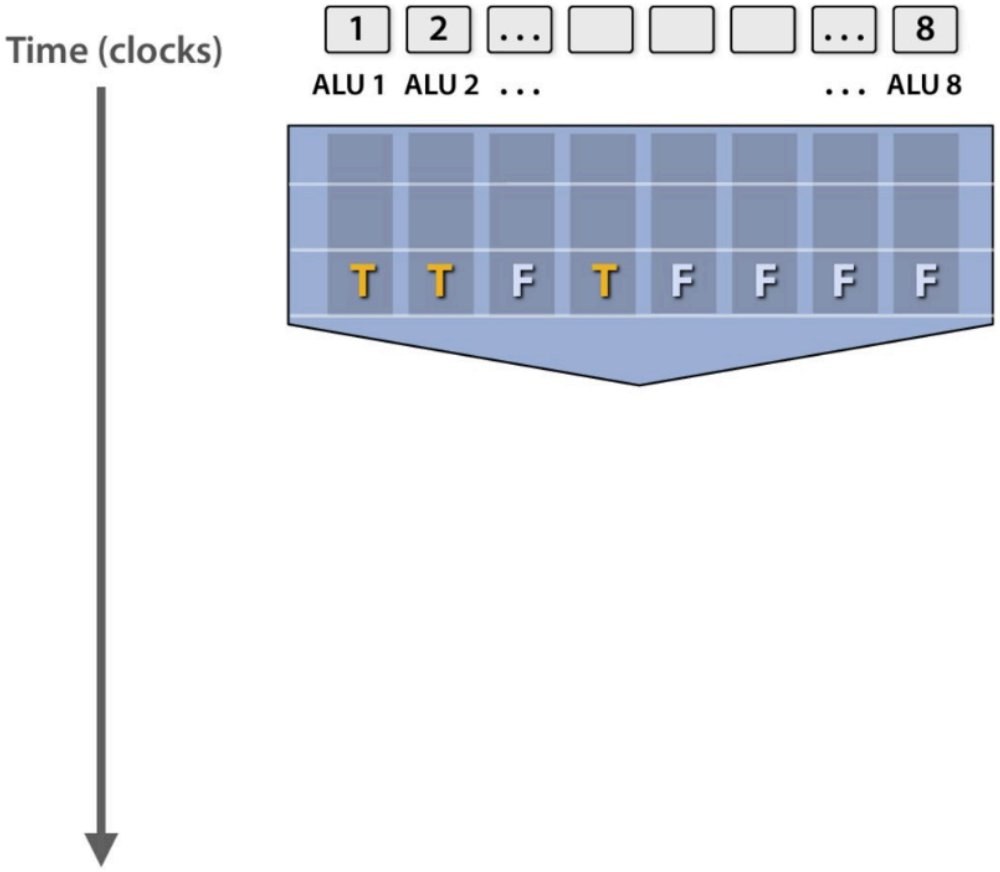
16 independent groups of
8 synchronized instruction streams



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

These specific numbers are just an example

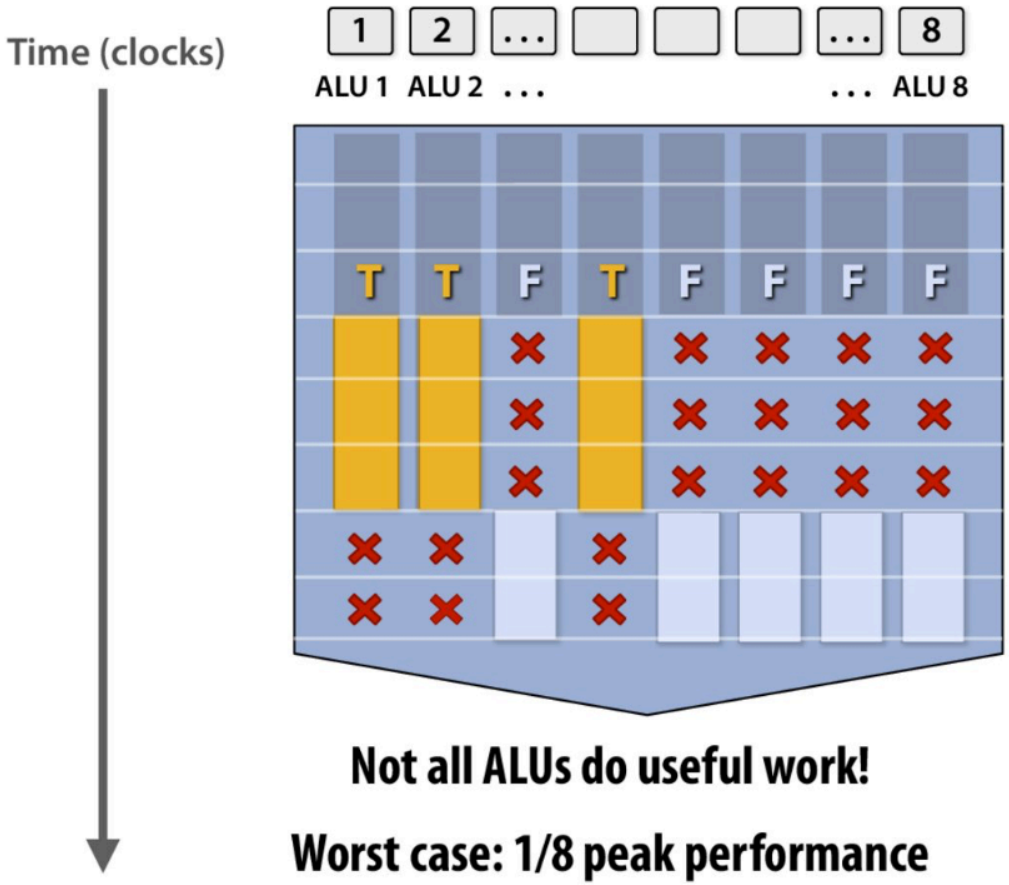
What about conditional execution?



(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 operation 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 lots of caches ... sort of

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

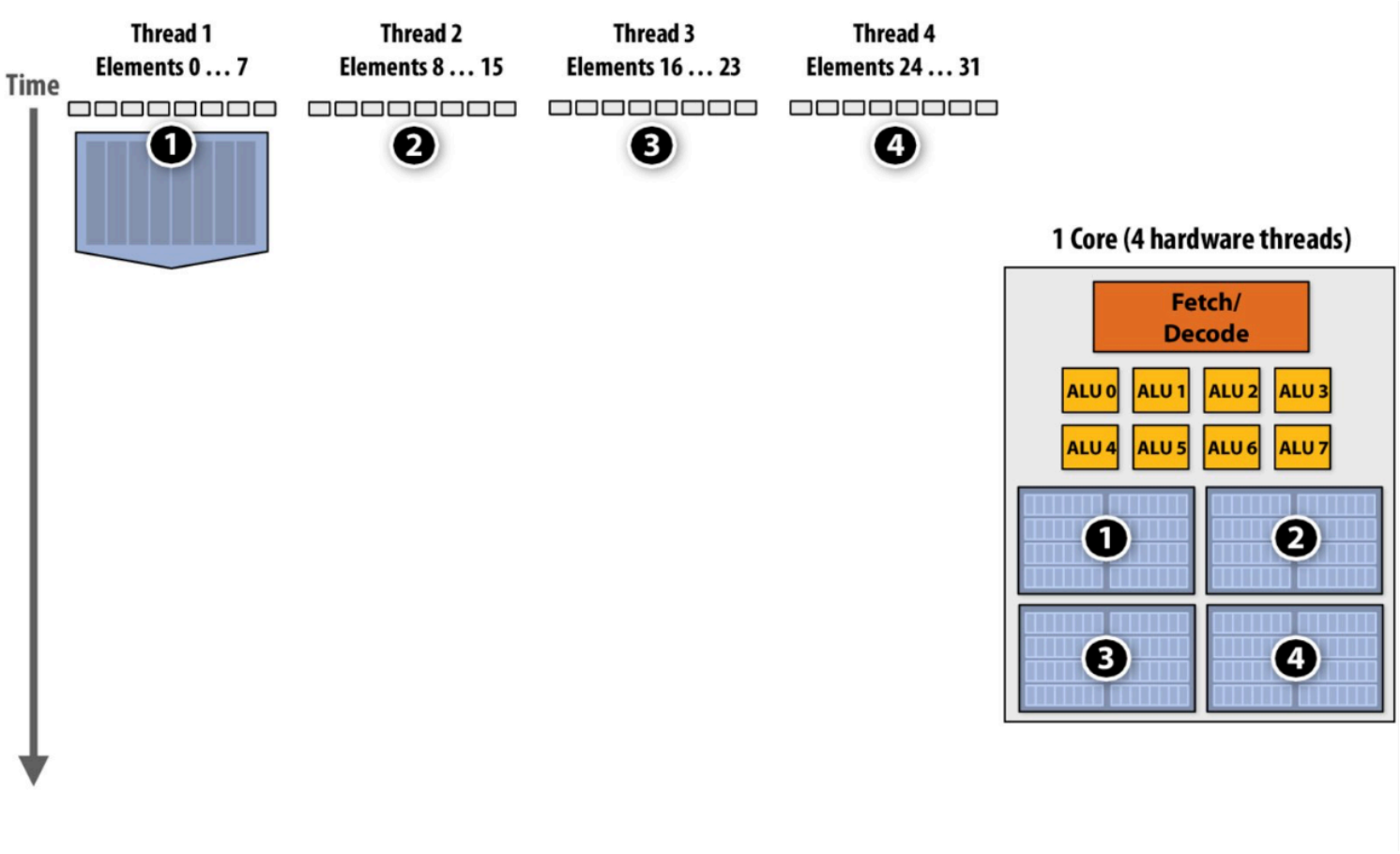
- memory access stalls will hurt performance

How best to addess? Some ideas...

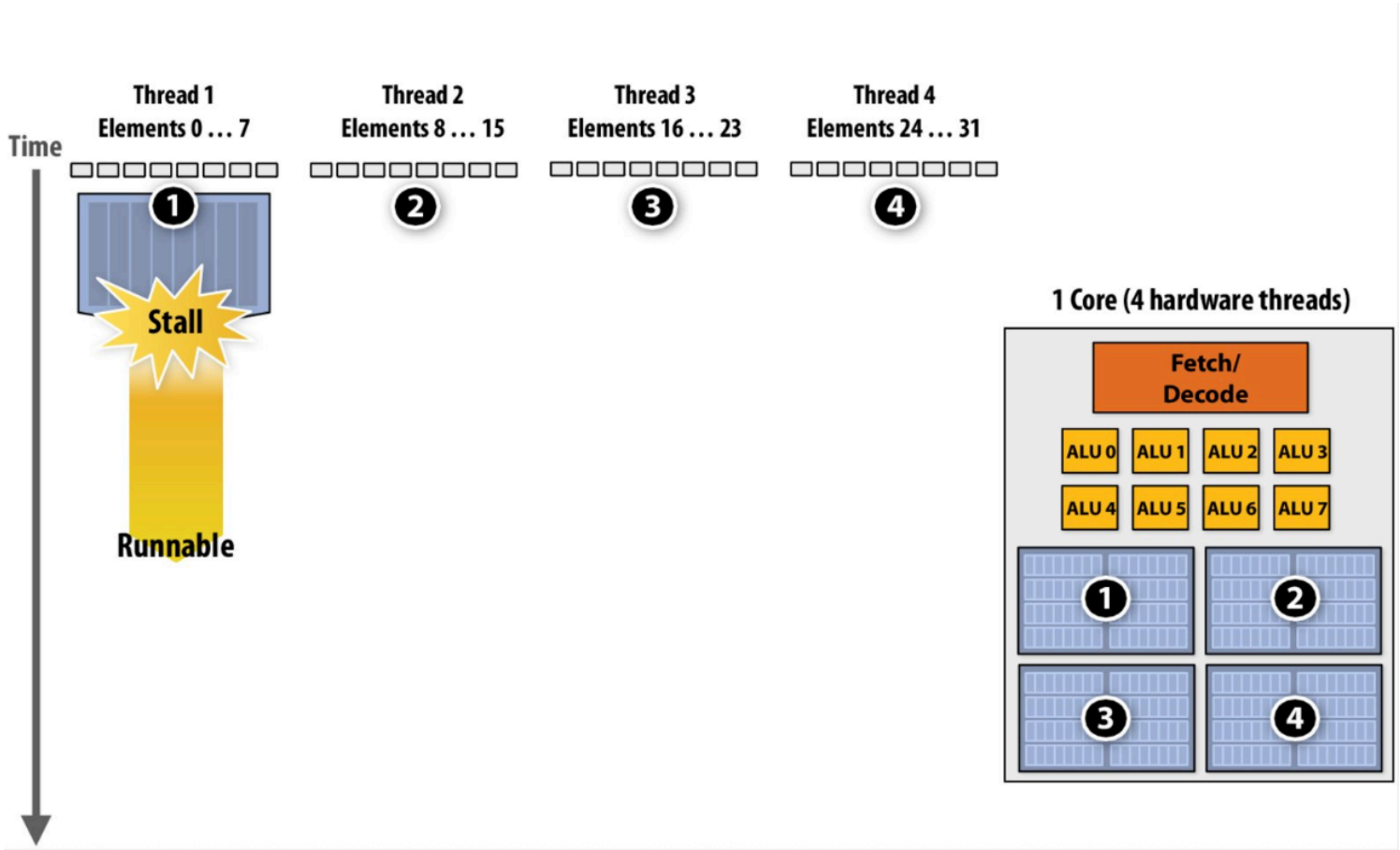
- prefetching...
- there are caches but less transparent -- programmer-controlled
  - registers (100's)
  - "shared memory"
- multithreading & context switching

Highest performance requires effort

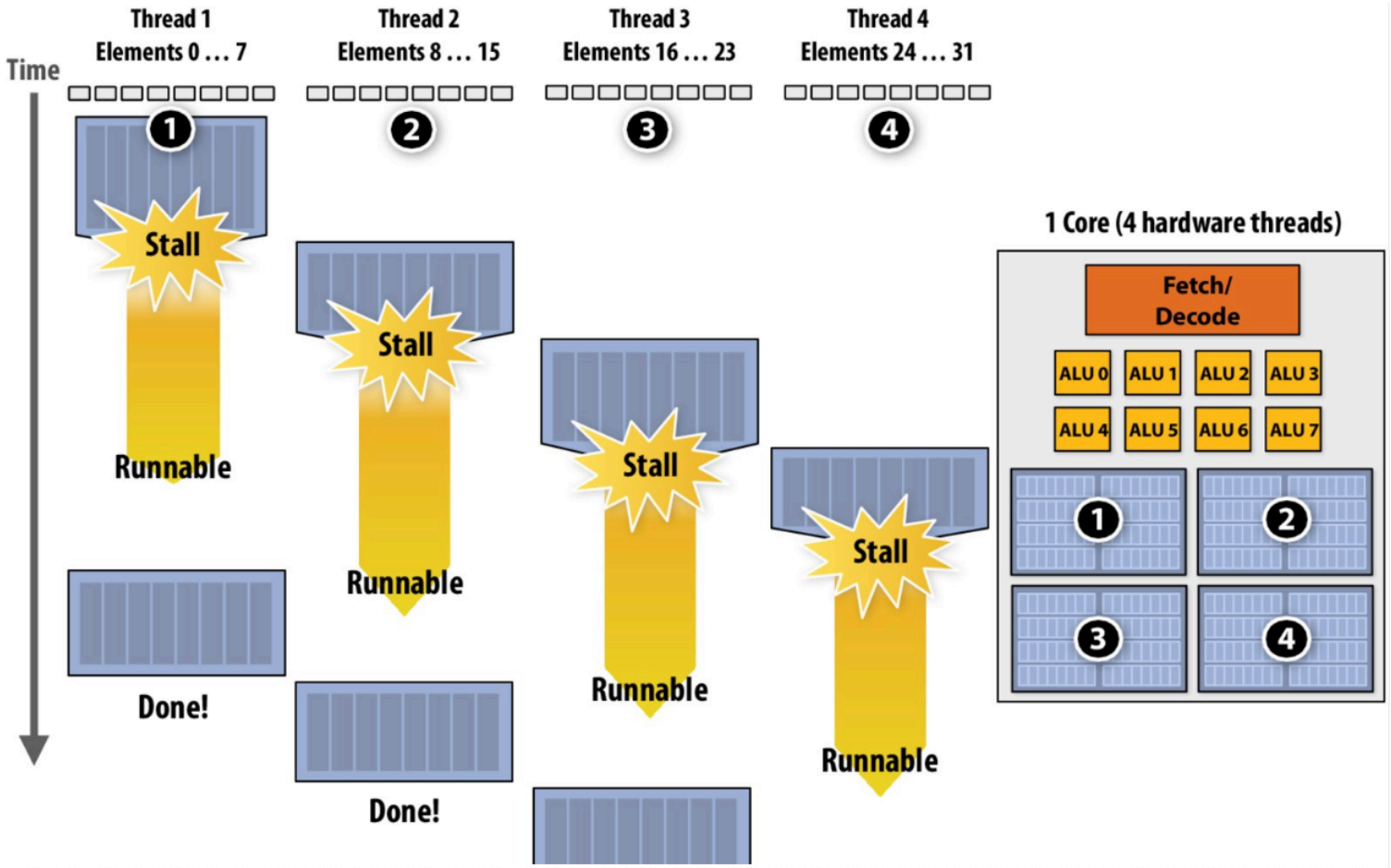
Hiding Stalls with Multithreading 1



Hiding Stalls with Multithreading 2



Hiding Stalls with Multithreading 3



GPU context switching

Instantaneous from one cycle to the next

your program needs many (> ~10) thread per core

The exact number is device dependent: rule of thumb go big or go home ;-)

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-architecture details change with every GPU generation
  - Programs are not automatically performance portable
- Worse, many micro-architectural features are not documented, or their interactions are complex
  - much experimentation is required...

Nvidia alone has released over 400 types of chips

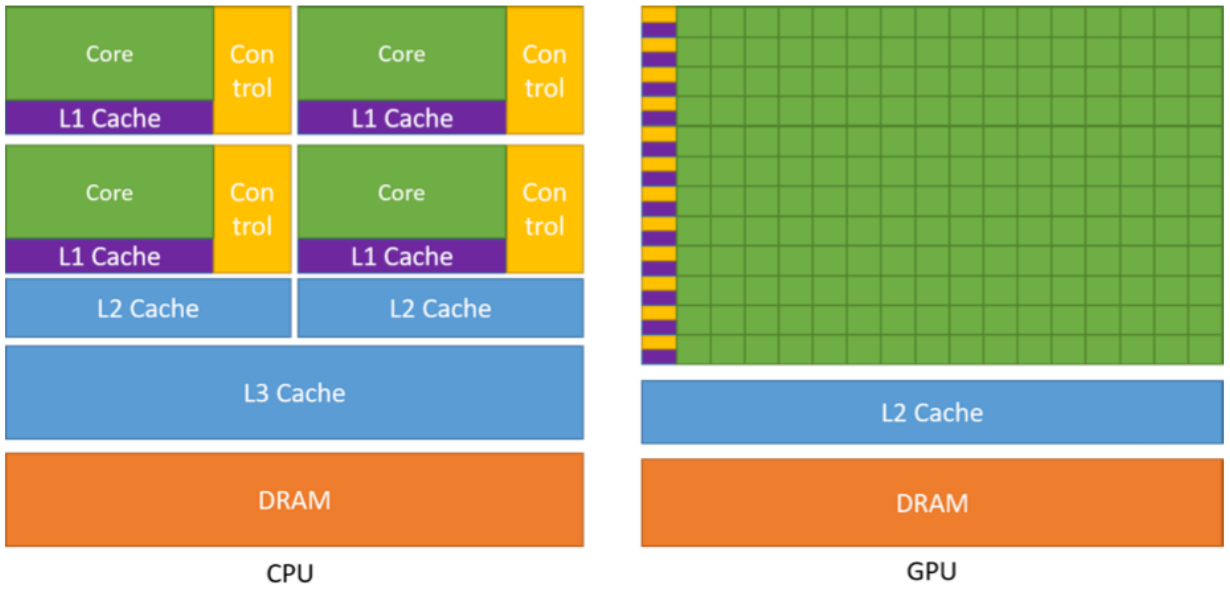
Architecture	Year	GeForce / RTX Products	Quadro / Pro Products	Tesla / Data Center	Features
Tesla	2008	GTX 8800, GTX 9800, GTX 280		Tesla 1060	Baseline architecture
Fermi	2010	GTX 460, GTX 480, GTX 580		Tesla 2070	Caches
Kepler	2012	GTX Titan Black, GTX Titan Z		Tesla K80	FP performance
Maxwell	2014	GTX 960, GTX 980, GTX Titan X			Power efficiency
Pascal	2016	GTX 1060, GTX 1070, GTX 1080, GTX Titan X		Tesla P100	16-bit FP ops
Volta	2017			Tesla V100, Titan V	Tensor cores for deep learning, NVLink
Turing	2018	RTX 2080	Quadro RTX 6000	T4	Ray tracing (RT) cores, NGX for graphics
Ampere	2020	RTX 3080, RTX 3090, RTX 3070, RTX 3060	RTX A6000	A100, A40, A30	3rd Gen Tensor & RT cores, FP16, AI/HPC boost
Ada Lovelace	2022	RTX 4090, RTX 4080, RTX 4070 Ti, RTX 4060, RTX 4050	RTX 6000 Ada Gen, L40		DLSS 3, improved ray tracing, efficiency
Hopper	2022			H100, H200	Transformer engine, exascale AI/HPC
Blackwell	2024	TBA (likely RTX 50 series in future)		B100, B200	Chiplet design, petaFLOPS scale, NVLink, 192GB HBM3E, generative AI focus

Product Line	Description
GeForce / RTX	Desktop & gaming graphics cards
Quadro / RTX (Pro)	Professional workstation graphics
Tesla / Data Center (incl. P/V/T4/A/H/B series)	HPC, datacenter, AI/ML accelerators
Jetson	Embedded systems
Tegra	Mobile/automotive/console chips

GPU Design Summary

This difference in capabilities between the GPU and the CPU exists because they are designed with different goals in mind. While the CPU is designed to excel at executing a sequence of operations, called a *thread*, as fast as possible and can execute a few tens of these threads in parallel, the GPU is designed to excel at executing thousands of them in parallel (amortizing the slower single-thread performance to achieve greater throughput).

The GPU is specialized for highly parallel computations and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. The schematic Figure 1 shows an example distribution of chip resources for a CPU versus a GPU.



CUDA C++ Programming Guide, Release 13.0 Figure 1 Page5

Extra References

- "GPU Computing" https://ieeexplore.ieee.org/document/4490127
- "A Closer look at GPUs" https://dl.acm.org/doi/10.1145/1400181.1400197
- "A survey of general-purpose computation on graphics hardware" https://research.nvidia.com/sites/default/files/pubs/2005-08_A-Survey-of/ASurveyofGeneralPurposeComputationonGraphicsHardware.pdf
- "NVIDIA Tesla: A Unified Graphics and Computing Architecture" https://ieeexplore.ieee.org/document/4523358?arnumber=4523358
- "The GeForce 6800", https://www.cs.cmu.edu/afs/cs/academic/class/15869-f11/www/readings/montrym05_geforce6800.pdf
- "Data Parallel Algorithms", https://doi.org/10.1145/7902.7903
- https://web.archive.org/web/20150207011210/http://gpgpu.org/developer
- https://www.tomshardware.com/reviews/ian-buck-nvidia,2393.html
- https://graphics.stanford.edu/~ianbuck/thesis.pdf
- https://cseweb.ucsd.edu/~ravir/274/15/papers/p777-buck.pdf
- https://graphics.stanford.edu/projects/brookgpu/
- https://github.com/hibengler/BrookGPU?tab=readme-ov-file
- The Connection Machine, https://en.wikipedia.org/wiki/Connection_Machine
- "The Connection Machine", https://dspace.mit.edu/bitstream/handle/1721.1/14719/18524280-MIT.pdf
- Tera Computer, https://en.wikipedia.org/wiki/Tera_Computer_Company
- "The Tera Computer System", https://dl.acm.org/doi/pdf/10.1145/255129.255132
- Transputer, https://en.wikipedia.org/wiki/Transputer
- "Supercomputing with Transputers - past, present and future", https://dl.acm.org/doi/10.1145/77726.255192
- Bill Dally, Burton Smith, Gene Amdhal, Alan Gara
- "Systolic Arrays for VLSI", https://www.eecs.harvard.edu/htk/static/files/1978-cmu-cs-report-kung-leiserson.pdf
- Intel SPC and Larrabee https://pharr.org/matt/blog/2018/04/30/ispc-all

Extra Info

	Nvidia P100 (2016)	Intel E7-8894 v4 (2016)	Nvidia V100 (2017)	Intel Platinum 8180 (2017)	Nvidia A100 (2020)	Intel Platinum 9282 (2020)	Nvidia H100 (2022)	Intel Platinum 8490H (2022)	Nvidia B100 (2024)	Intel Max 6700/8500 (2024)
Cores	3,584	24	5,120	28	6,912	56	14,592	60	16,384	128
Threads	-	48	-	56	-	112	-	120	-	128
Peak INT32 TIPS	10.6	0.34	15.7	2.2	19.5	8.9	40	13	41	24
Peak FP32 TFLOPS	10.6	0.03	15.7	0.04	19.5	0.075	39	0.11	39.9	0.23
Peak FP64 TFLOPS	5.3	0.03	7.8	0.04	9.7	0.075	19.5	0.11	19.7	0.23
Memory Bandwidth (GB/s)	732	115	900	134	1,600	400	3,350	460	3,900	480
Approx. Cost (MSRP, USD)	~9,000	~7,500	~10,000–12,000	~10,000	~11,000–15,000	~25,000	~35,000–40,000	~11,000	~30,000–40,000*	~10,000–14,000
Peak Power (W)	300	165	300	205	400	400	700	350	700	350

Lecture 1: Motivation and Challenges

CS599: Programming Massively Parallel Multiprocessors and Heterogenous Systems (Understanding and programming the devices powering AI)

Jonathan Appavoo

Slide Title

Slide Title

- Body Level One
  - Body Level Two
    - Body Level There
      - Body Level Four
        - Body Level Five

Slide Title

Slide Title

- Body Level One
  - Body Level Two
    - Body Level There
      - Body Level Four
        - Body Level Five

Footer

Slide Title

Slide Title

Section Title

Slide Title