Massively Parallel Algorithms
Introduction

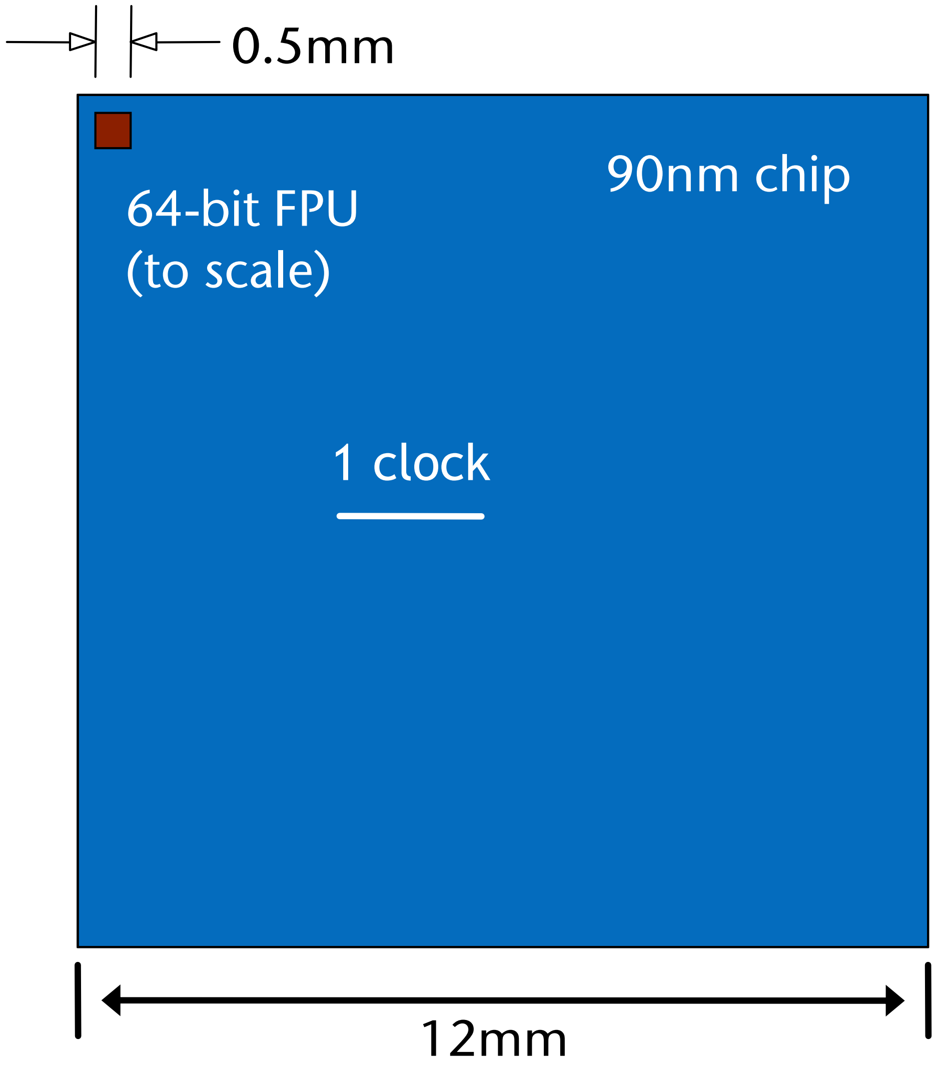
G. Zachmann

University of Bremen, Germany

cgvr.cs.uni-bremen.de

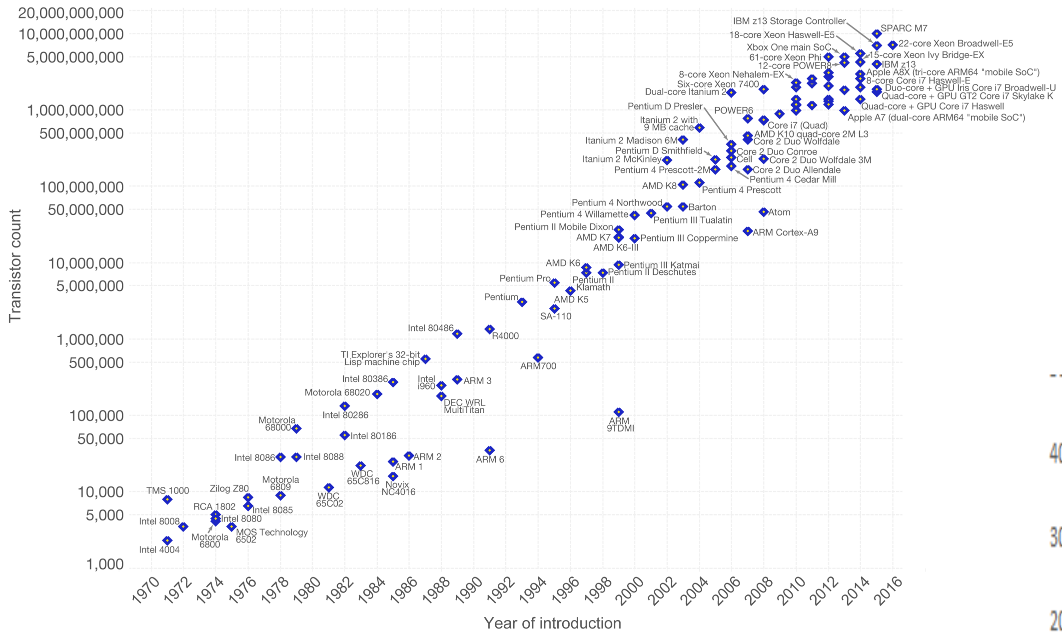
Why Massively Parallel Computing?

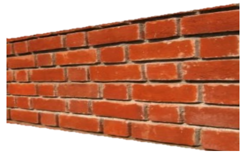
- "Compute is cheap" …
- … "Bandwidth is expensive"
  - Main memory is ~500 clock cycles "far away" from the processor (GPU or CPU)



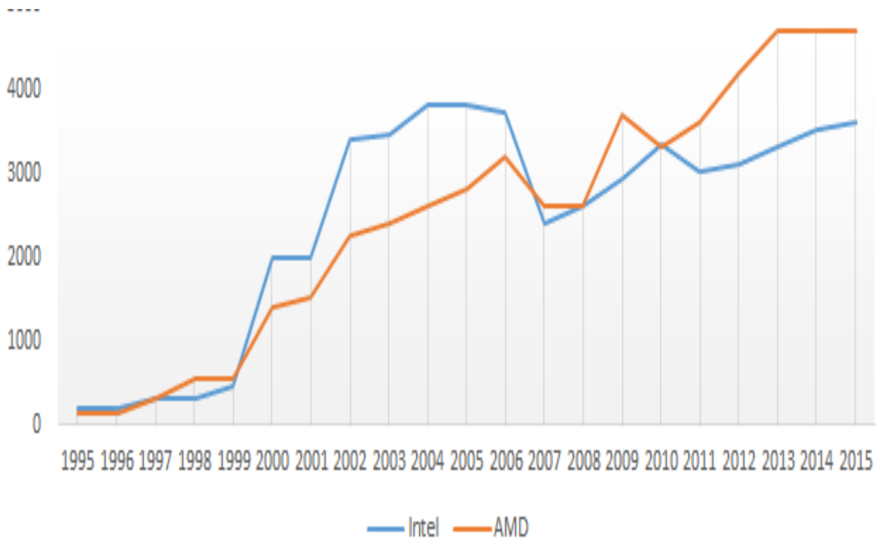
Moore's Law & The Brick Wall

Moore's Law (it's really only an observation)

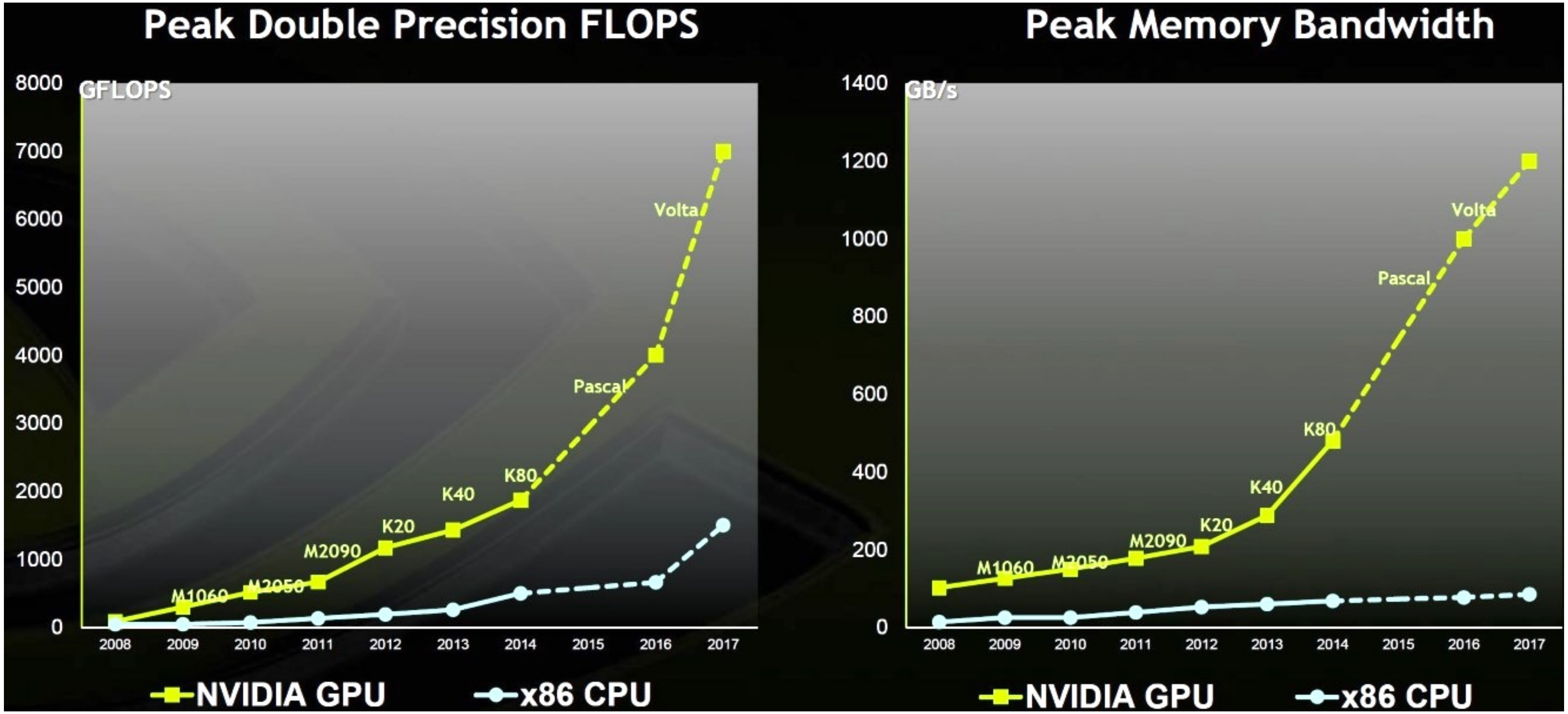




CPU speed



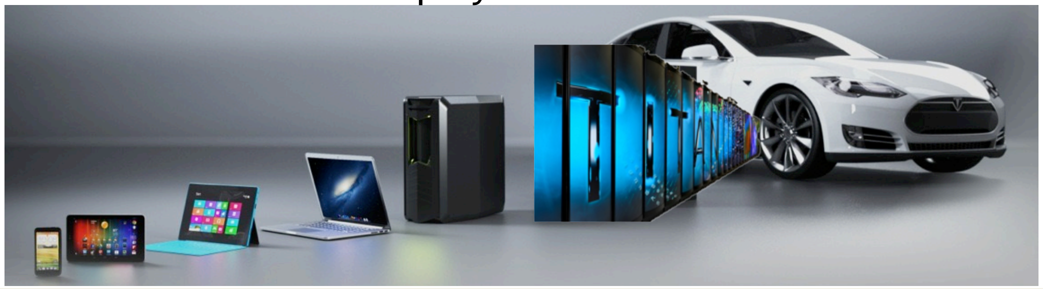
"More Moore" with GPUs

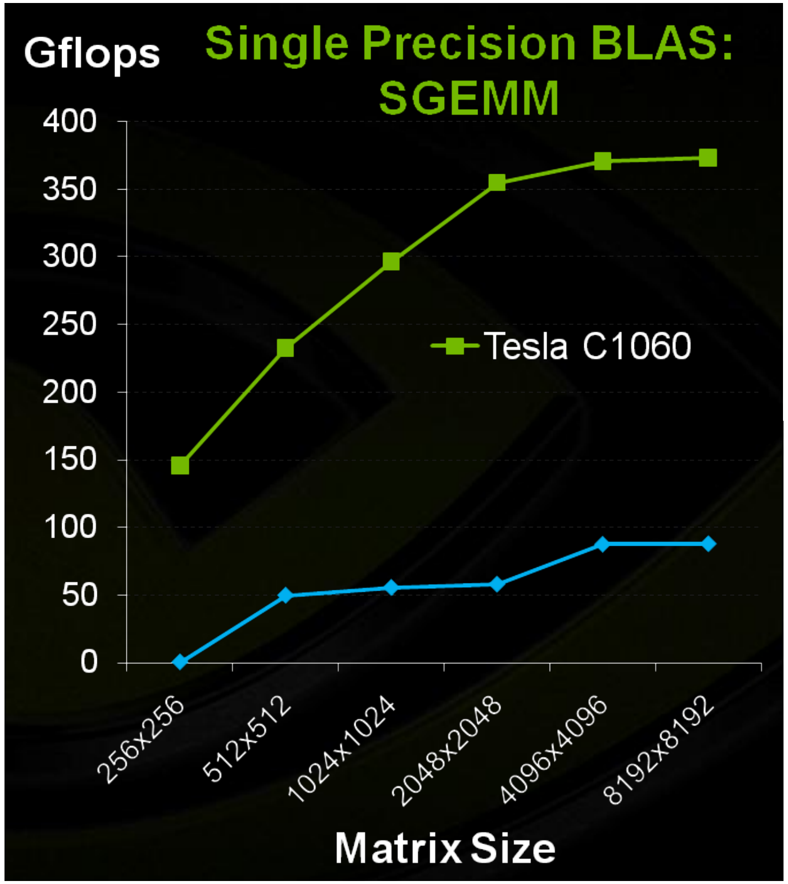


Theoretical Peak Performance

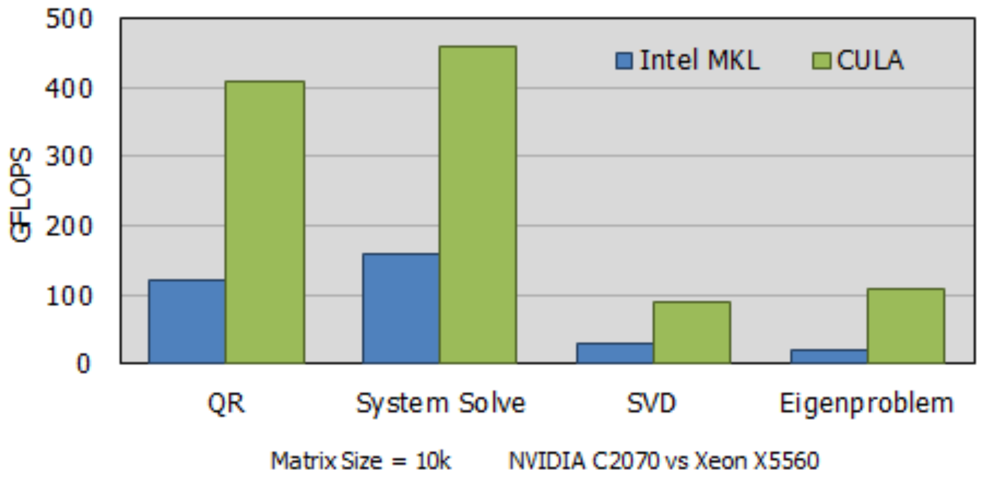
Memory Bandwidth

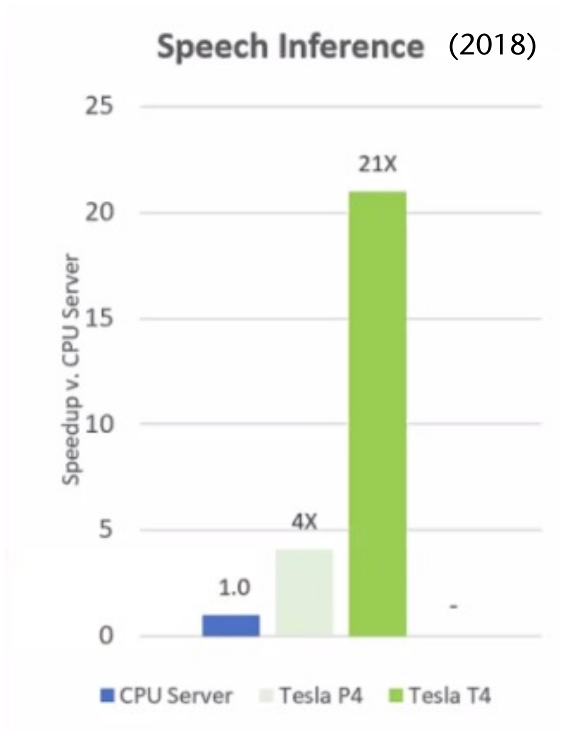
Deployment cases





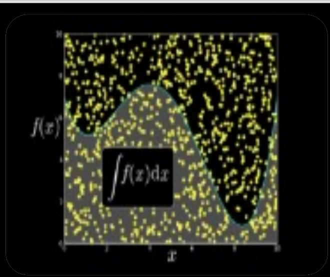
CUBLAS: CUDA 2.3, Tesla C1060
MKL 10.0.3: Intel Core2 Extreme, 3.00GHz

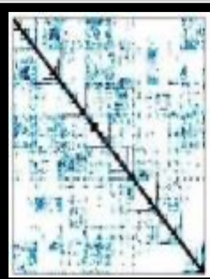




GPU Accelerated Libraries ("Drop-In Acceleration)


NVIDIA cuBLAS


NVIDIA cuRAND


NVIDIA cuSPARSE


NVIDIA NPP

Vector Signal Image Processing

GPU Accelerated Linear Algebra

Matrix Algebra on GPU and Multicore


NVIDIA cuFFT

IMSL Library

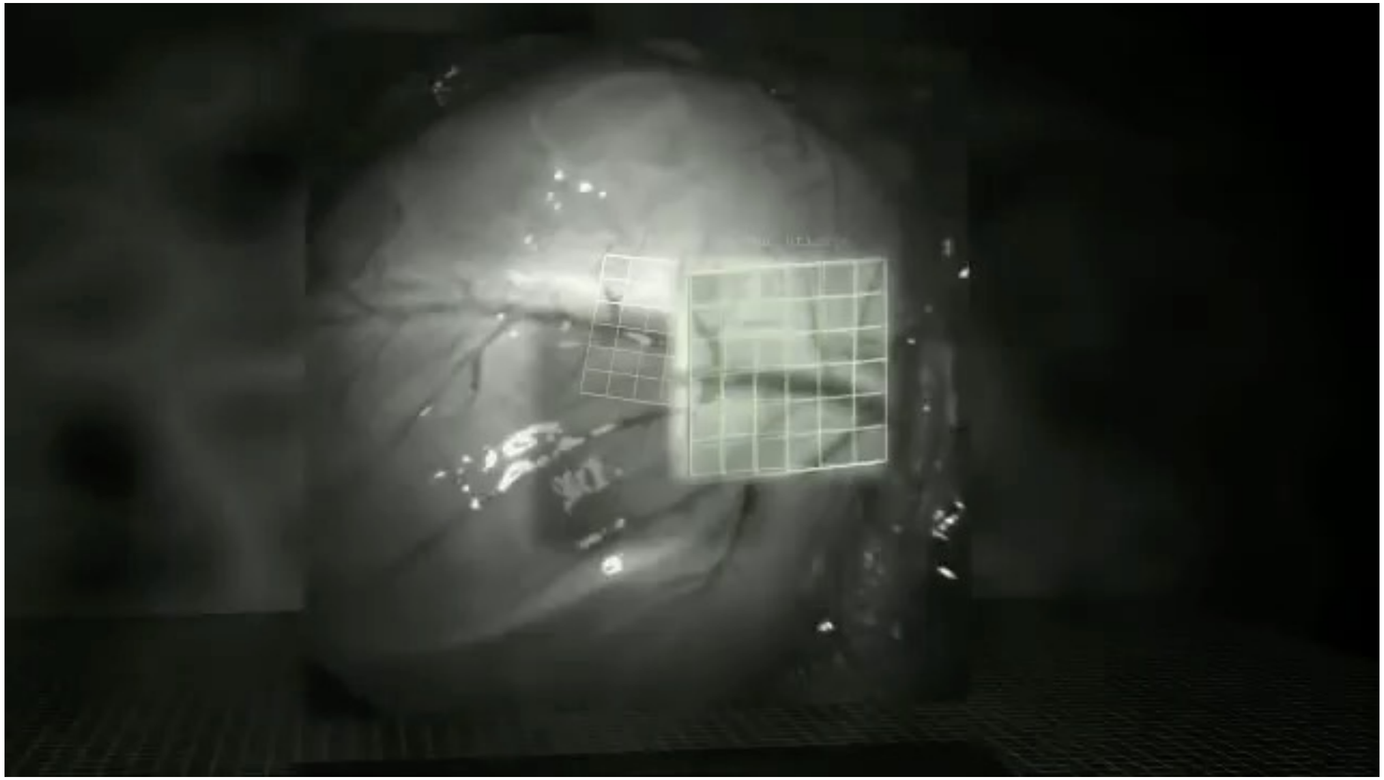
ArrayFire Matrix Computations

Sparse Linear Algebra

C++ STL Features for CUDA

Operating on a Beating Heart

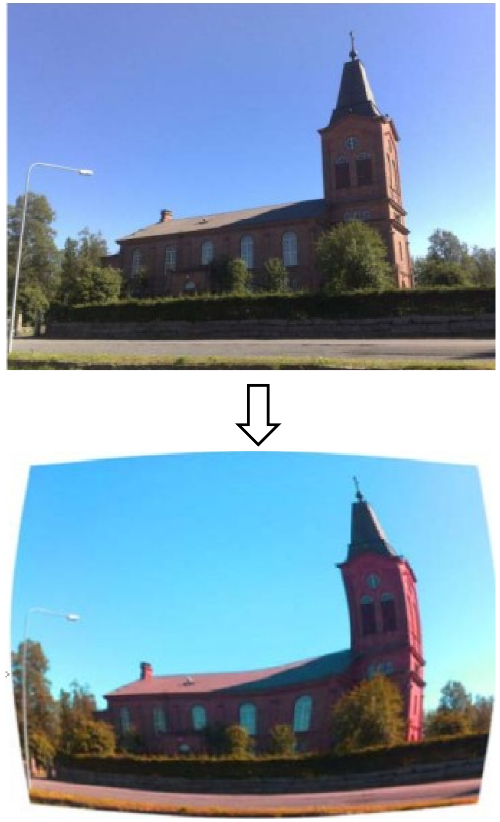
- Only 2% of surgeons will operate on a beating heart
- Patient stands to lose 1 point of IQ every10 min with heart stopped
- GPU enables real-time motion compensation to virtually stop beating heart for surgeons



Rogerio Richa

When Power Consumption Matters

- Energy consumption is a serious issue on mobile devices
- Example: image processing on a mobile device (geometric distortion + blurring + color transformation)
- Power consumption:
  - CPU (ARM Cortex A8): 3.93 J/frame
  - GPU (PowerVR SGX 530): 0.56 J/frame (~14%)
    - 0.26 J/frame when data is already on the GPU



- *High parallelism at low clock frequencies* (110 MHz) is better than (i.e., "gives you more bang for the buck") *low parallelism at high clock frequencies* (550 Mhz)
  - Power dissipation increases super-linearly with frequency

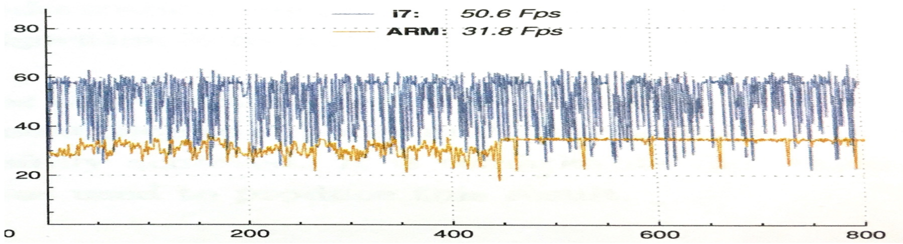
Another Experiment Relating Computational/Electrical Efficiency

- Task: FEM simulation on CPU vs GPU
- Architectures:

	CPU		GPU	
	Intel i7 4930k	**Tegra ARMv7 Cortex-A15**	**Kepler GK110A**	**Kepler GK20A**
Clock speed	3.4 GHz	1.9 GHz	1.25 GHz	0.85 GHz
Max Power Consumption	130W	~2W	250W	2W

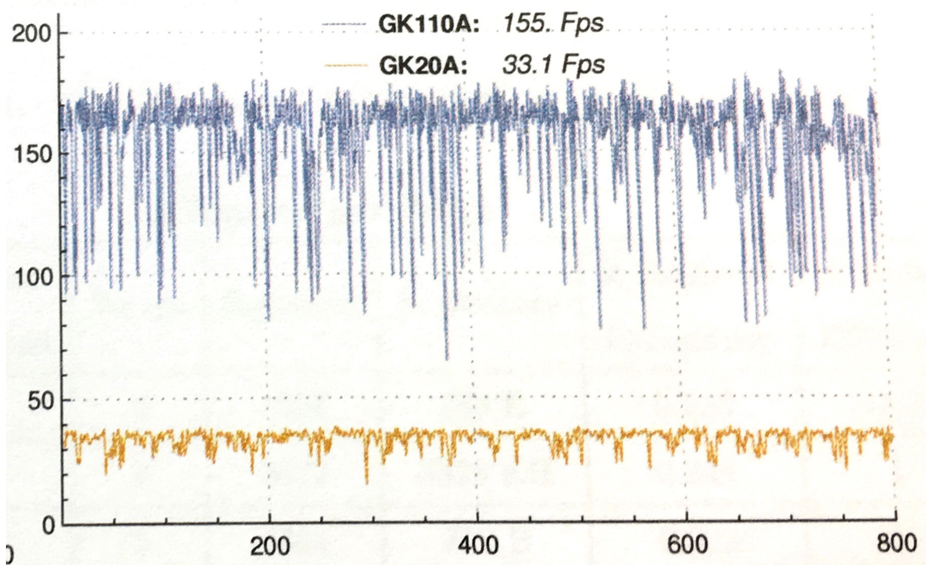
- Comparison with respect to FPS:

FPS



Frame

FPS

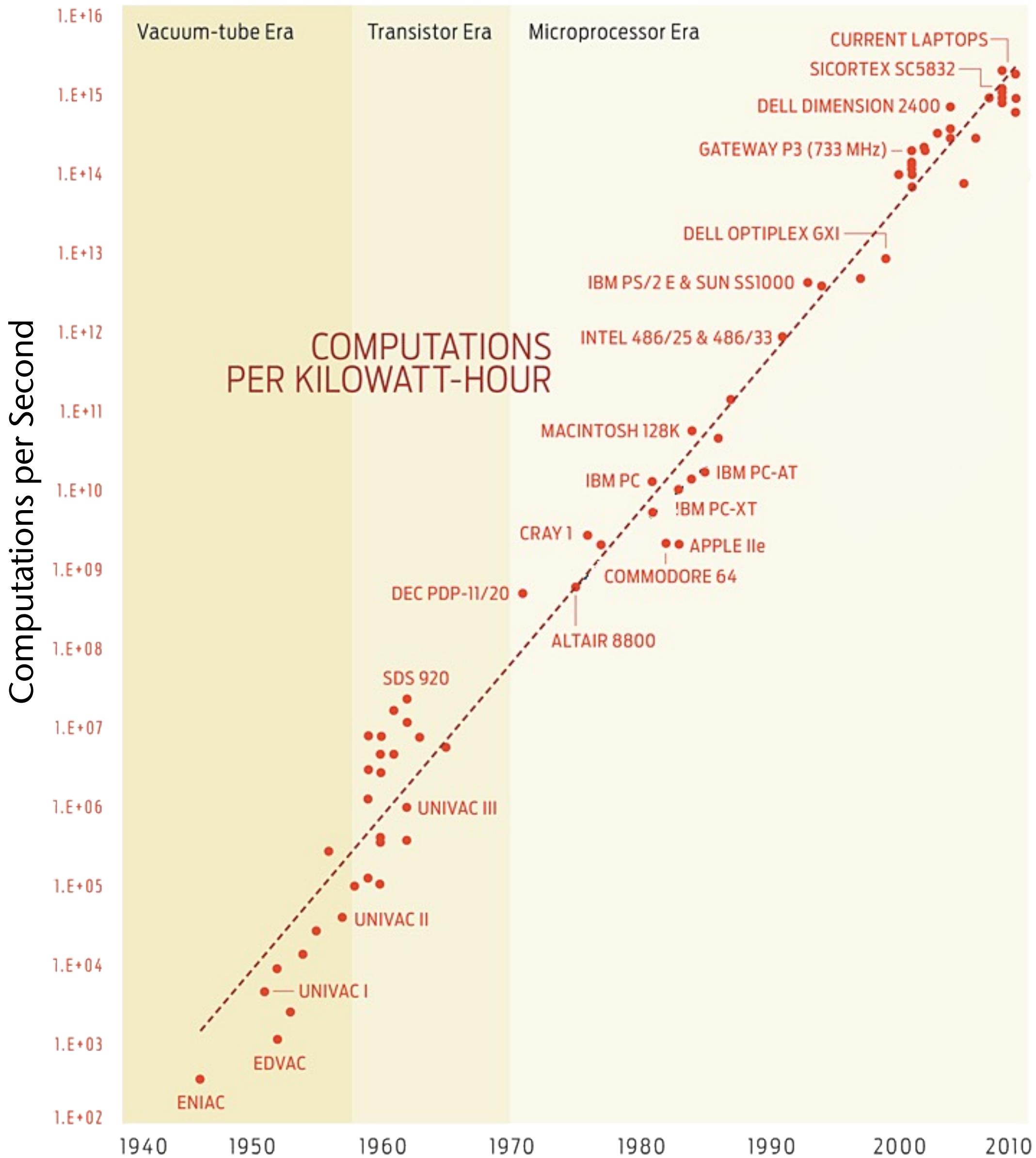


Frame

- Average energy efficiency:

	Intel i7 4930k	Tegra ARMv7 Cortex-A15	Kepler GK110A	Kepler GK20A
Efficiency in J/frame	2.6	0.06	1.6	0.06

The Trend of Electrical Efficiency of Computation



If a MacBook Air were as inefficient as a 1991 computer, the battery would last 2.5 seconds.

"Assessing Trends in the Electrical Efficiency of Computation Over Time" Koomey et al., 2009

Areas Benefitting from Massively Parallel Algos

- Computer science (e.g., visual computing, database search)
- Computational material science (e.g., molecular dynamics sim.)
- Bio-informatics (e.g., alignment, sequencing, …)
- Economics (e.g., simulation of financial models)
- Mathematics (e.g., solving large PDEs)
- Mechanical engineering (e.g., CFD and FEM)
- Physics (e.g., *ab initio* simulations)
- Logistics (e.g. simulation of traffic, assembly lines, or supply chains)

Some Statistics of the TOP500

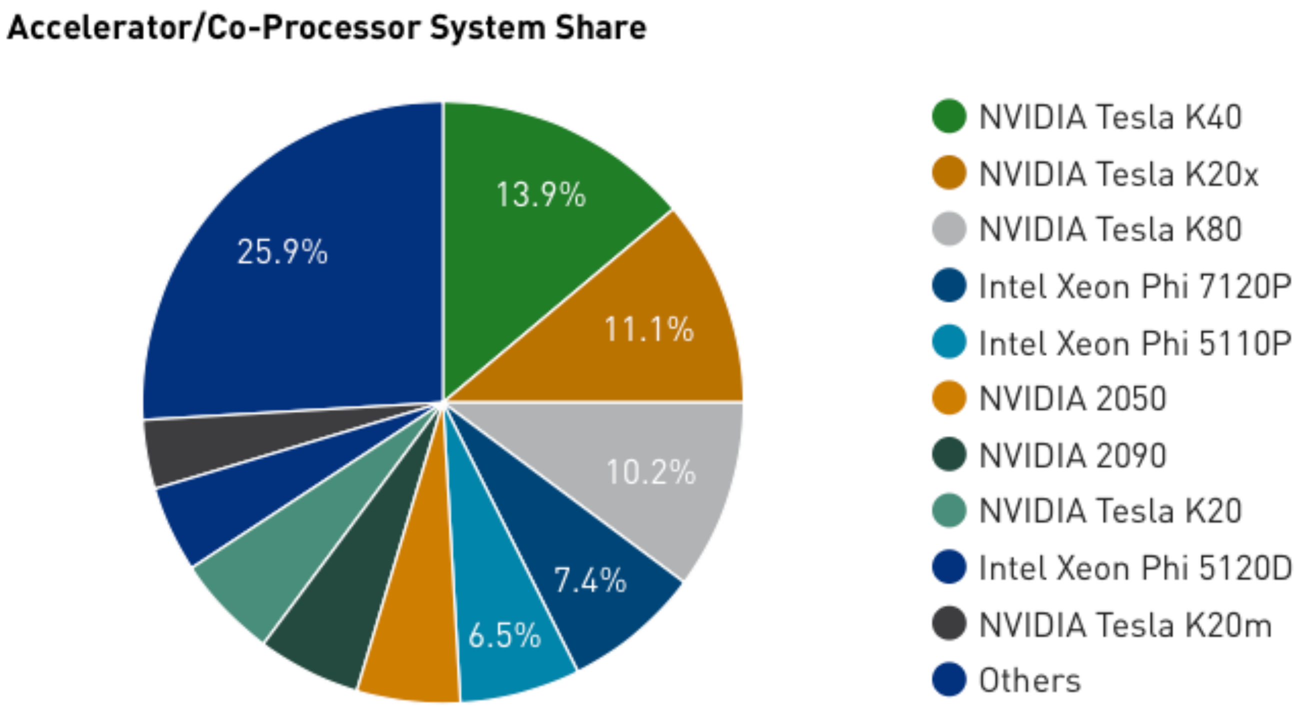
- Our target platform (GPU) is being used among the TOP500 [Nov 2015]:

TITAN - CRAY XK7 , OPTERON 6274 16C 2.200GHZ, CRAY GEMINI INTERCONNECT, NVIDIA K20X

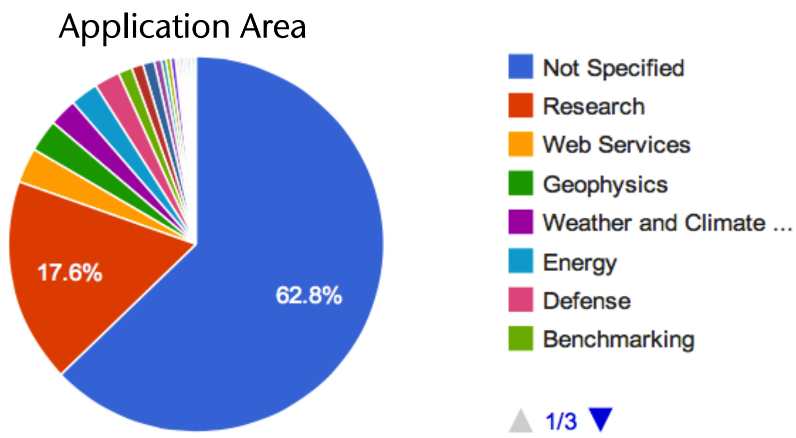
Site:	DOE/SC/Oak Ridge National Laboratory
System URL:	http://www.olcf.ornl.gov/titan/
Manufacturer:	Cray Inc.
Cores:	560,640
Linpack Performance (Rmax)	17,590 TFlop/s
Theoretical Peak (Rpeak)	27,112.5 TFlop/s
Power:	8,209.00 kW
Memory:	710,144 GB
Processor:	Opteron 6274 16C 2.2GHz
Interconnect:	Cray Gemini interconnect
Operating System:	Cray Linux Environment

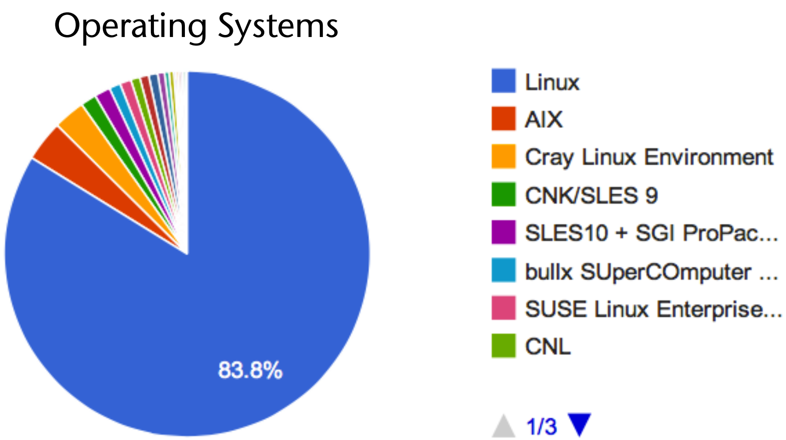
RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWER (KW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209

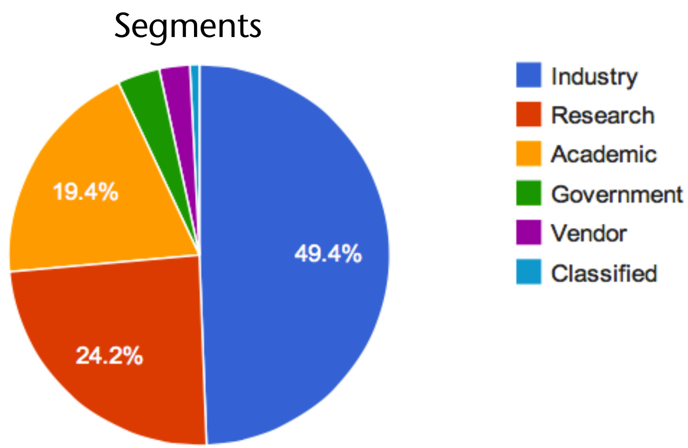
Source: www.top500.org


Accelerator/Co-Processor System Share
13.9%
11.1%
10.2%
7.4%
6.5%
25.9%
NVIDIA Tesla K40
NVIDIA Tesla K20x
NVIDIA Tesla K80
Intel Xeon Phi 7120P
Intel Xeon Phi 5110P
NVIDIA 2050
NVIDIA 2090
NVIDIA Tesla K20
Intel Xeon Phi 5120D
NVIDIA Tesla K20m
Others

- Who does parallel computing:
  - Note that respondents had to choose just one area
  - "Not specified" probably means "many areas"

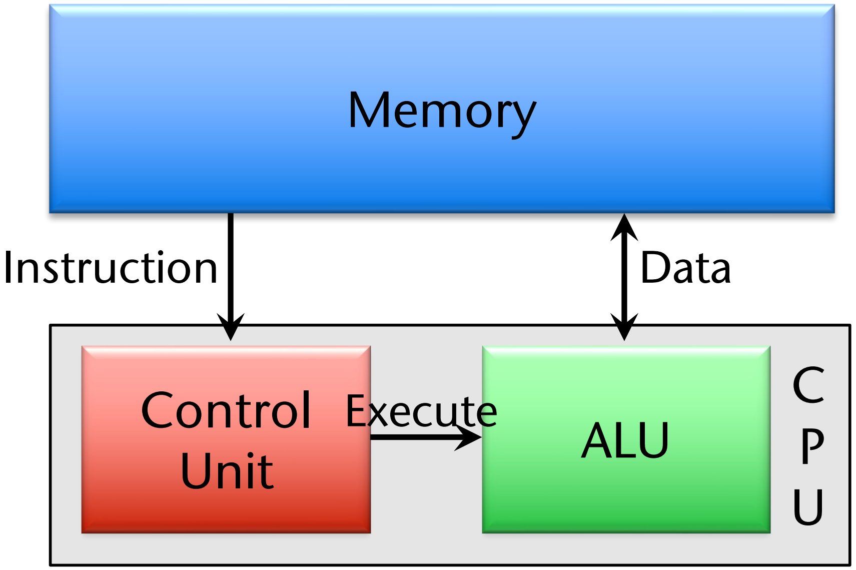


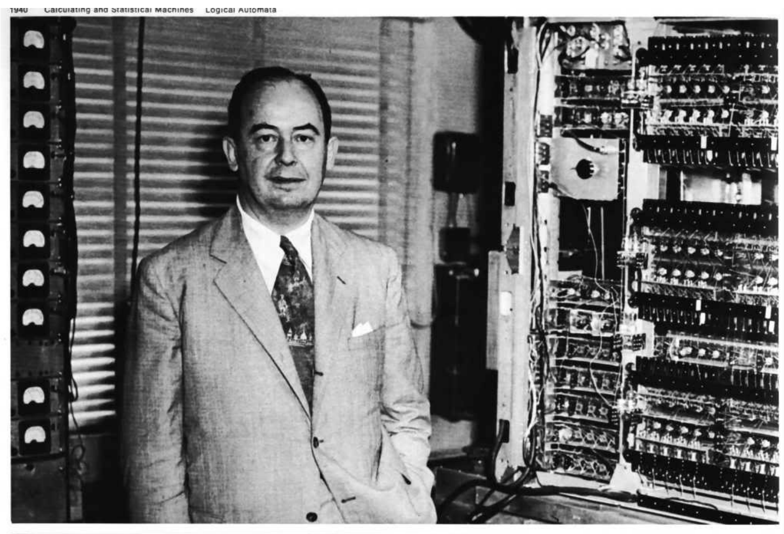




The Von-Neumann Architecture

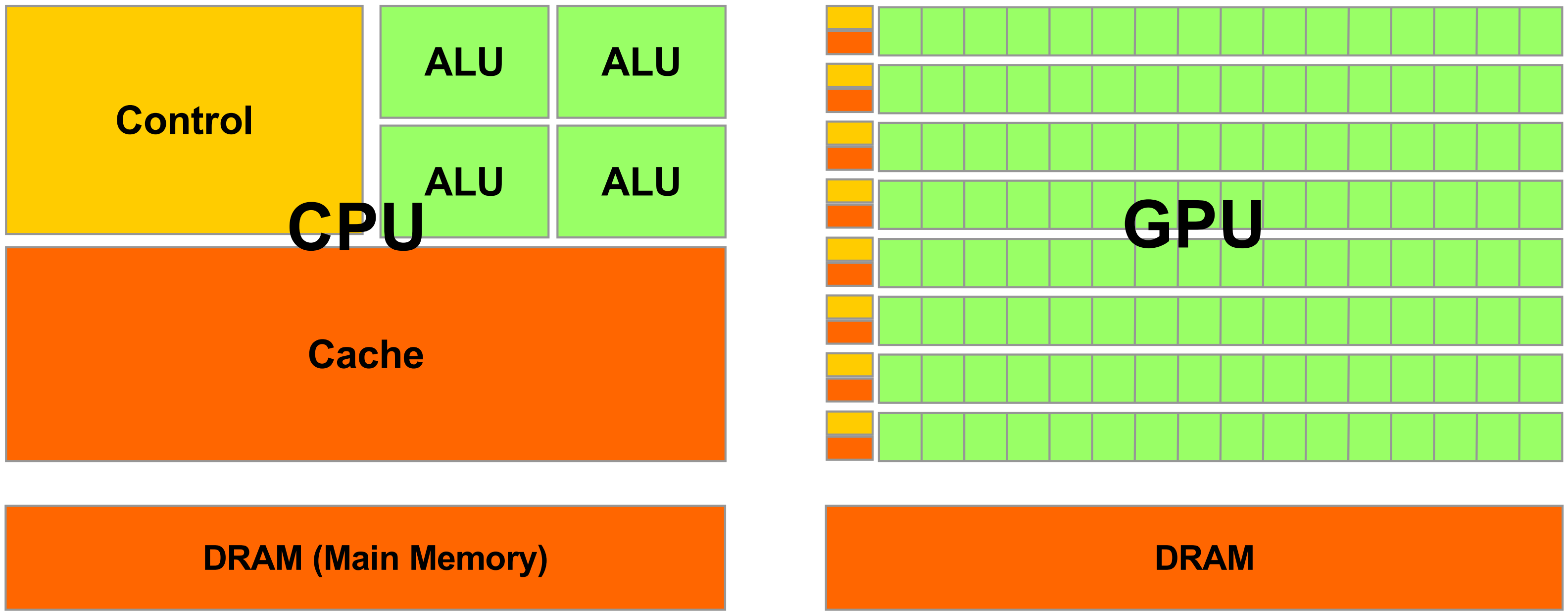
- Uses the stored-program concept (revolutionary at the time of its conception)
- Memory is used for both program instructions and data

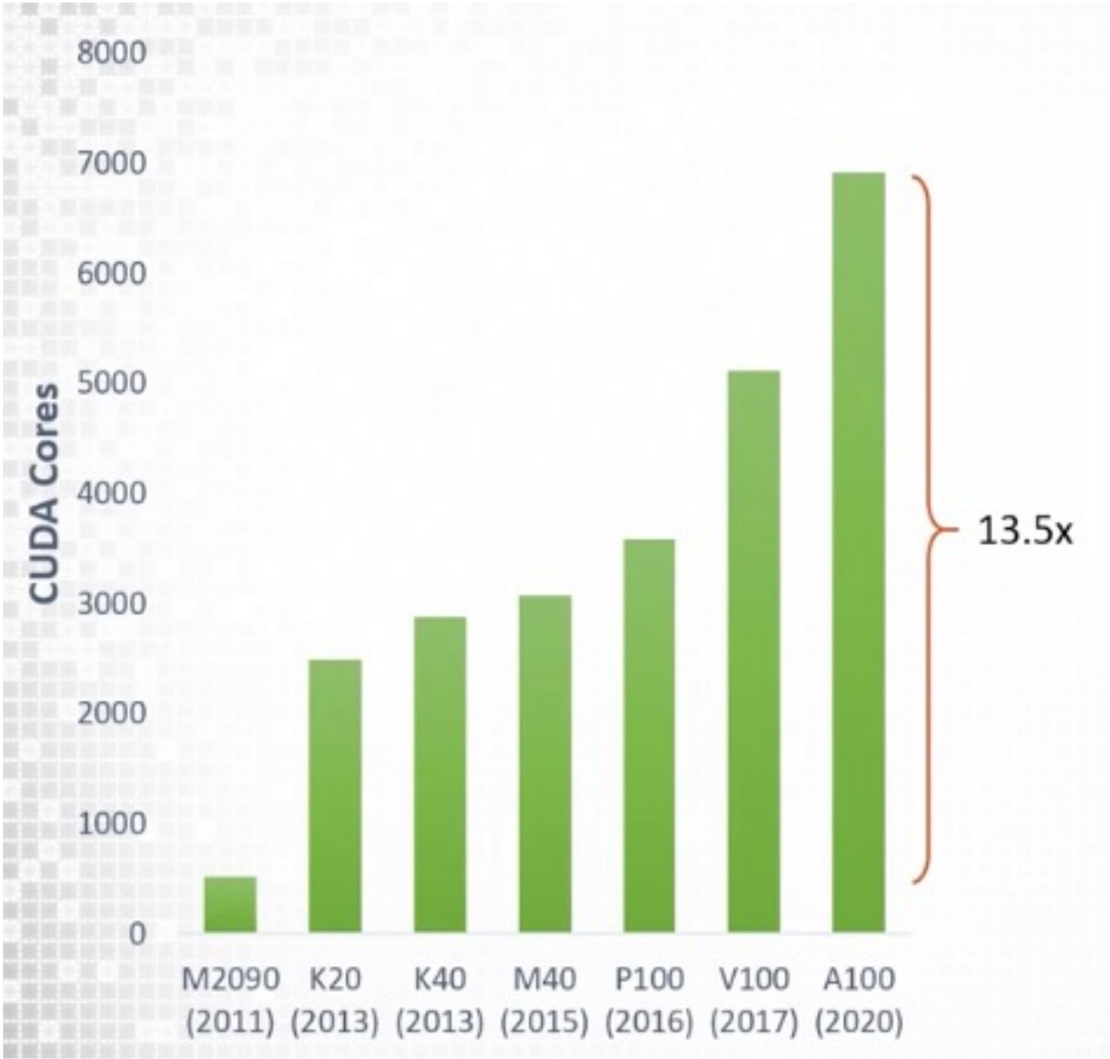




The GPU = the New Architecture

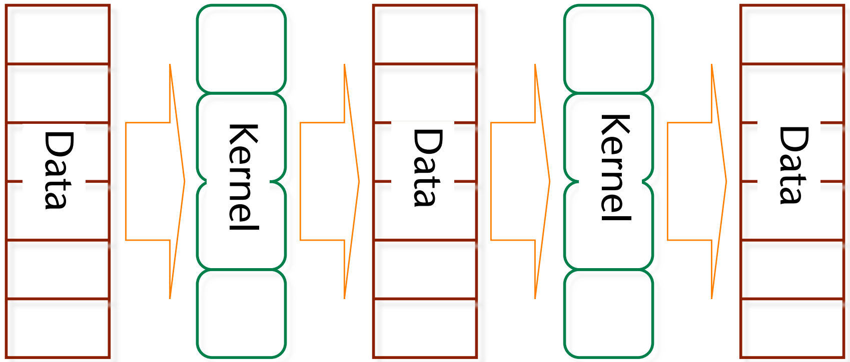
- CPU = lots of cache, little SIMD, a few cores
- GPU = little cache, massive SIMD, lots of cores (packaged into "streaming multi-processors")




CUDA Cores
8000
7000
6000
5000
4000
3000
2000
1000
0
M2090 (2011)
K20 (2013)
K40 (2013)
M40 (2015)
P100 (2016)
V100 (2017)
A100 (2020)
13.5x

The Stream Programming Model

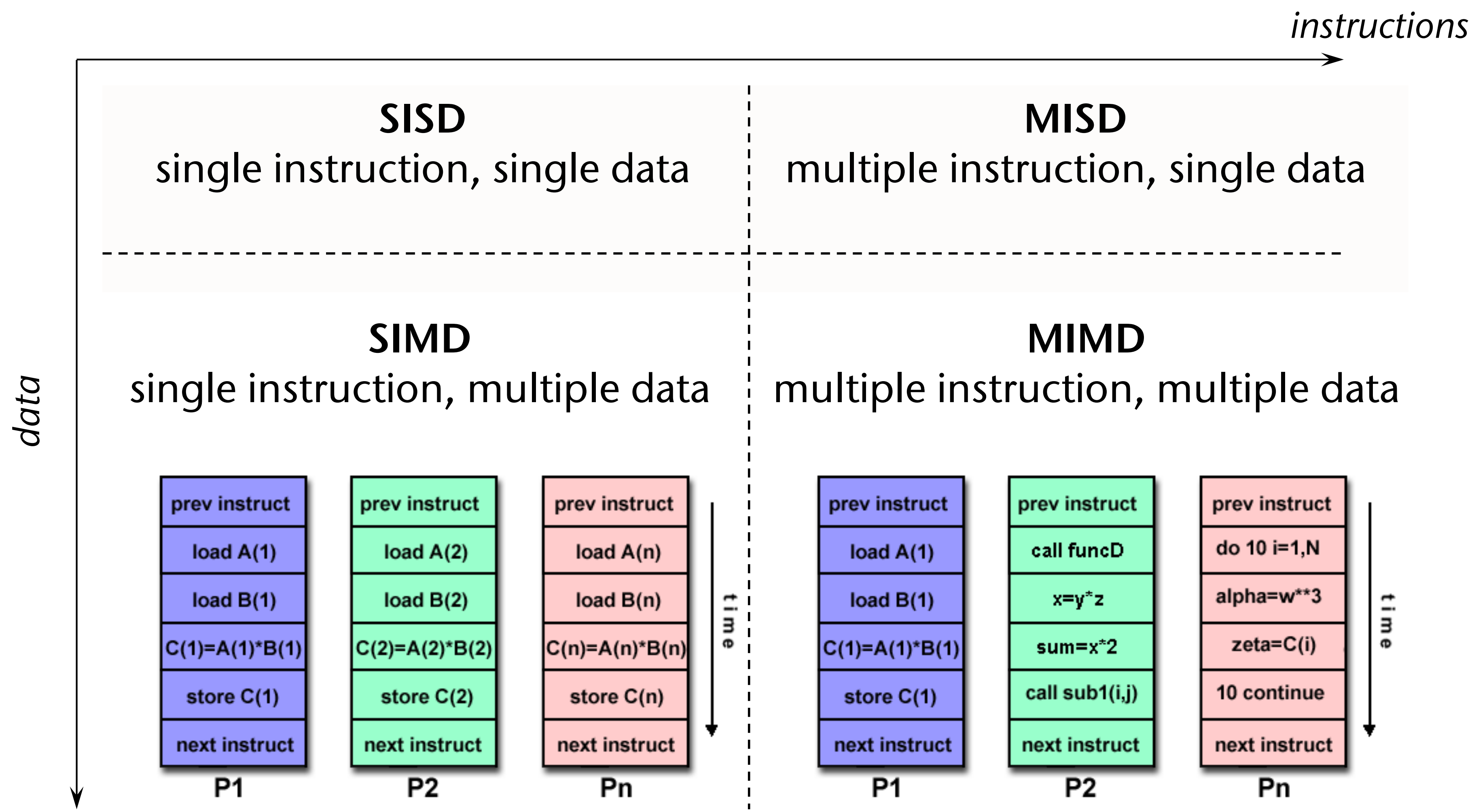
- Novel programming paradigm that tries to organise data & functions such that (as much as possible) only streaming memory access will be done, and as little random access as possible:
  - Stream Programming Model = *"Streams of data passing through computation kernels."*
  - *Stream* := ordered, homogenous set of data of arbitrary type (array)
  - *Kernel* := program to be performed on *each* element of the input stream; produces (usually) one new output stream




```
stream A, B, C;
kernelfunc1( input: A,
             output: B );
kernelfunc2( input: B,
             output: C);
```

Flynn's Taxonomy

- Two dimensions: instructions and data
- Two values: single and multiple

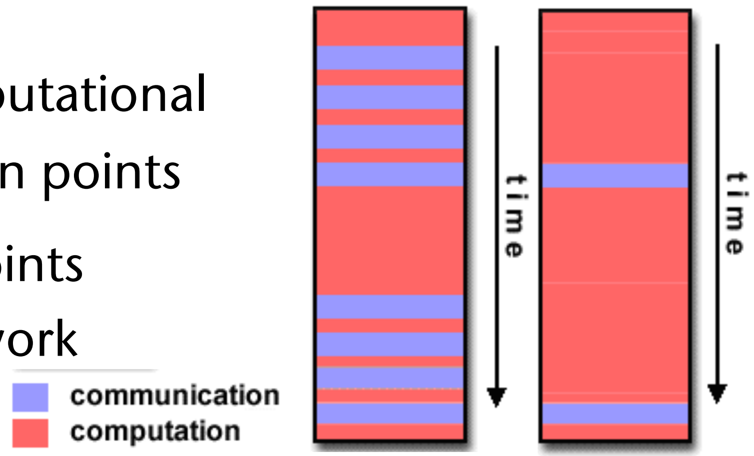


Some Terminology

- **Task** := logically discrete section of computational work; typically a program or procedure
- **Parallel Task** := task that can be executed in parallel by multiple processors, such that this yields the correct results
- **Shared memory** :=
  - Hardware point of view: all processors have direct access to common physical memory
  - Software point of view: all parallel tasks have the same "picture" of memory and can directly address and access the same logical memory locations, regardless of where the physical memory actually exists
- **Communication** := exchange of data among parallel tasks, e.g., through shared memory

- Synchronous communication := requires some kind of "handshaking" (i.e., synchronization mechanism)
- Asynchronous communication := no sync required
  - Example: task 1 sends a message to task 2, but doesn't wait for a response
  - A.k.a. non-blocking communication
- Collective communication := more than 2 tasks are involved

- **Synchronization** := coordination of parallel tasks, very often associated with communications; often implemented by establishing a **synchronization point** across tasks
  - Example: a task may not proceed further until another task (or *all* other tasks) reaches the same or logically equivalent point
  - Synchronization usually involves waiting by at least one task, and can therefore cause a parallel application's execution time to increase
- **Granularity** := qualitative measure of the ratio of computation to synchronization
  - Coarse granularity: large amounts of computational work can be done between synchronization points
  - Fine granularity: lots of synchronization points sprinkled throughout the computational work



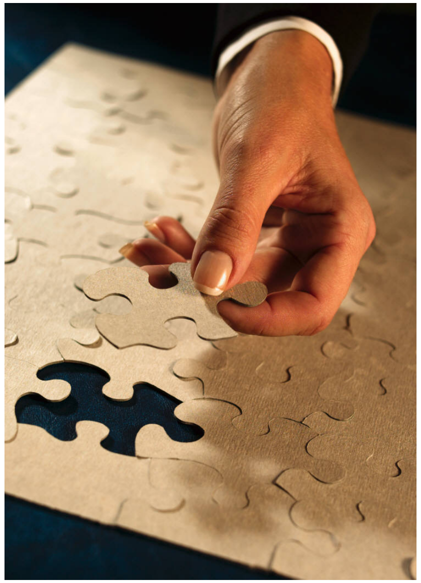
- Observed Speedup := measure for performance of parallel code

$$\text{speedup} = \frac{\text{wall-clock execution time of best known sequential code}}{\text{wall-clock execution time of your parallel code}}$$

  - One of the simplest and most widely used indicators for a parallel program's performance

Amdahl's Law

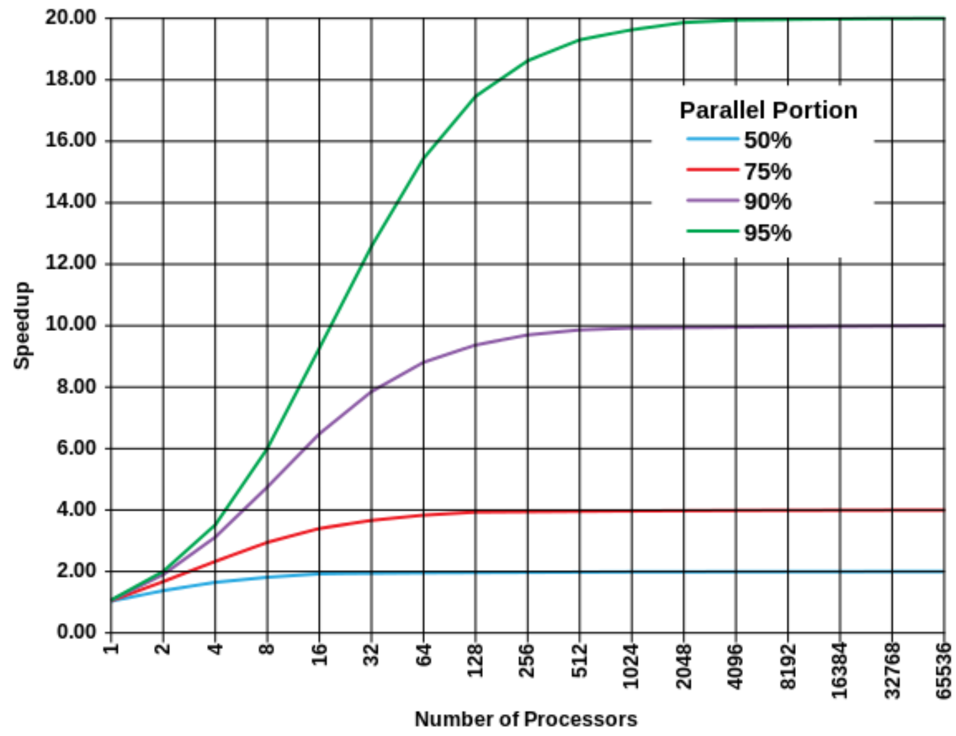
- Quiz:
  - Suppose we want to do a 5000 piece jigsaw puzzle
  - Time for one person to complete puzzle: *n* hours
  - How much time do we need, if we add 1 more person at the table?
  - How much time, if we add 100 persons?



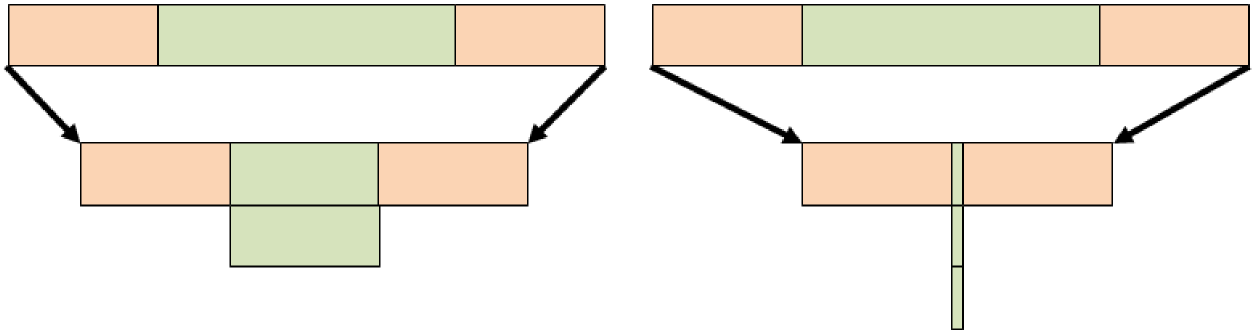
Amdahl's Law (the "Pessimist")

- Assume a program execution consists of two parts: *P* and *S*, where
  P = time for parallelizable part ,
  S = time for inherently sequential part
- W.l.o.g. set $P + S = 1$
- Assume further that the time taken by *N* processors working on *P* is $\frac{P}{N}$
- Then, the maximum speedup achievable is

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

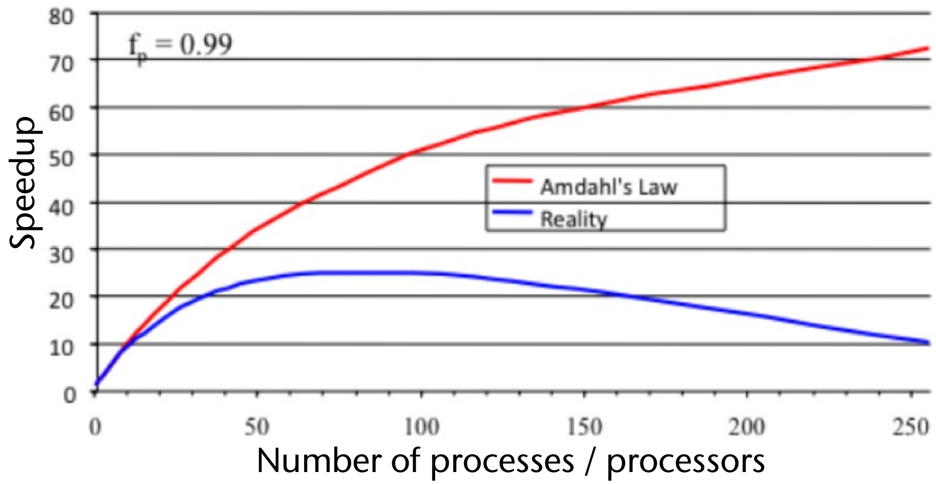


- Graphical representation of Amdahl:



(You can squeeze the parallel part as much as you like, by throwing more processors at it, but you cannot squeeze the sequential part)

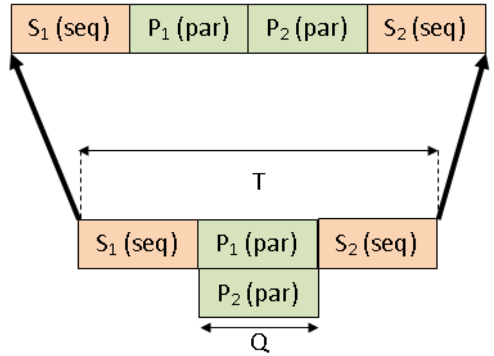
- With conventional parallelization, the speedup can be even worse than Amdahl's prediction!
  - Work is distributed among a number of processes *communicating with each other*, e.g., via message passing
  - Due to *parallel overhead*



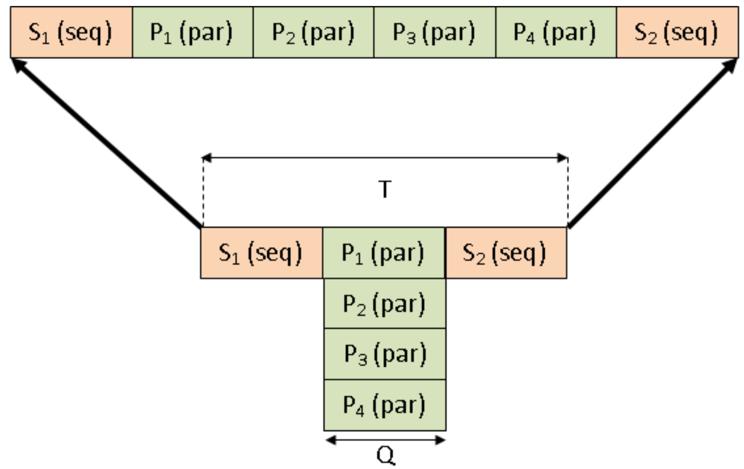
- Parallel Overhead := amount of time required to coordinate parallel tasks, as opposed to doing useful work; can include factors such as: task start-up time, synchronizations, data communications, scheduling, I/O, etc.
- Scalable problem := problem where parallelizable part P increases with problem size

Gustafson's Law (the "Optimist")

- Assume a family of programs, that all run in a fixed time frame T, with
  - a sequential part S,
  - and a time portion Q for parallel execution,
  - $T = S + Q$



- Assume, we can deploy N processors, working on larger and larger problem sizes in parallel

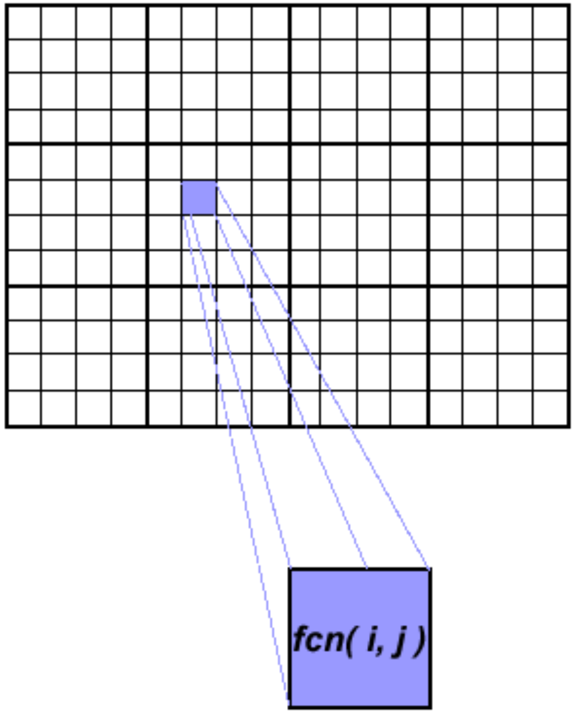


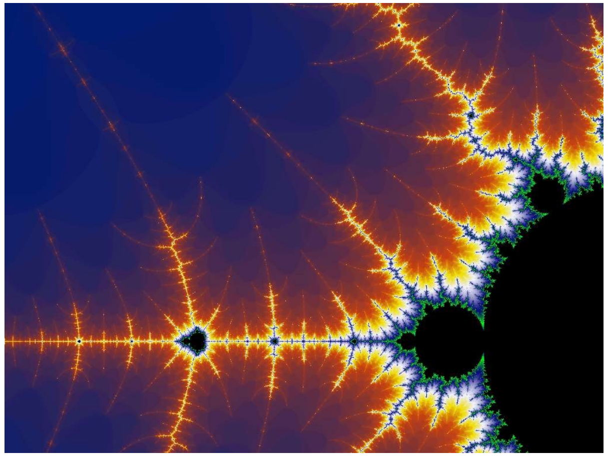
- So, Gustafson's speedup is

$$\text{speedup}_G(N) = \frac{S + QN}{S + Q} \rightarrow \infty \;, \quad \text{with } N \rightarrow \infty$$

Examples of Easily Parallelizable Problems

- Compute an image, where each pixel is just a function of its coordinates
  - E.g. Mandelbrot set
  - Question: is rendering a polygonal scene one of this case?
- Such parallel problems are called "*embarrassingly parallel*"
  - There is nothing embarrassing about them ☺
- Other examples:
  - Brute-force searches in cryptography
  - Large scale face recognition
  - Genetic algorithms
  - SETI@home , and other such distributed comp.





Example of Inherently Sequential *Algorithm*

- Calculation of the Fibonacci series (1,1,2,3,5,8,13,21,...) by use of the formula:

$$F(k+2) = F(k+1) + F(k)$$

- The problem here is data dependence
- This is one of the common inhibitors to parallelization
- Common solution: different algorithm
- Other algorithm for Fibonacci?

$$F_n = \frac{\varphi^n - \psi^n}{\varphi - \psi} = \frac{\varphi^n - \psi^n}{\sqrt{5}}$$

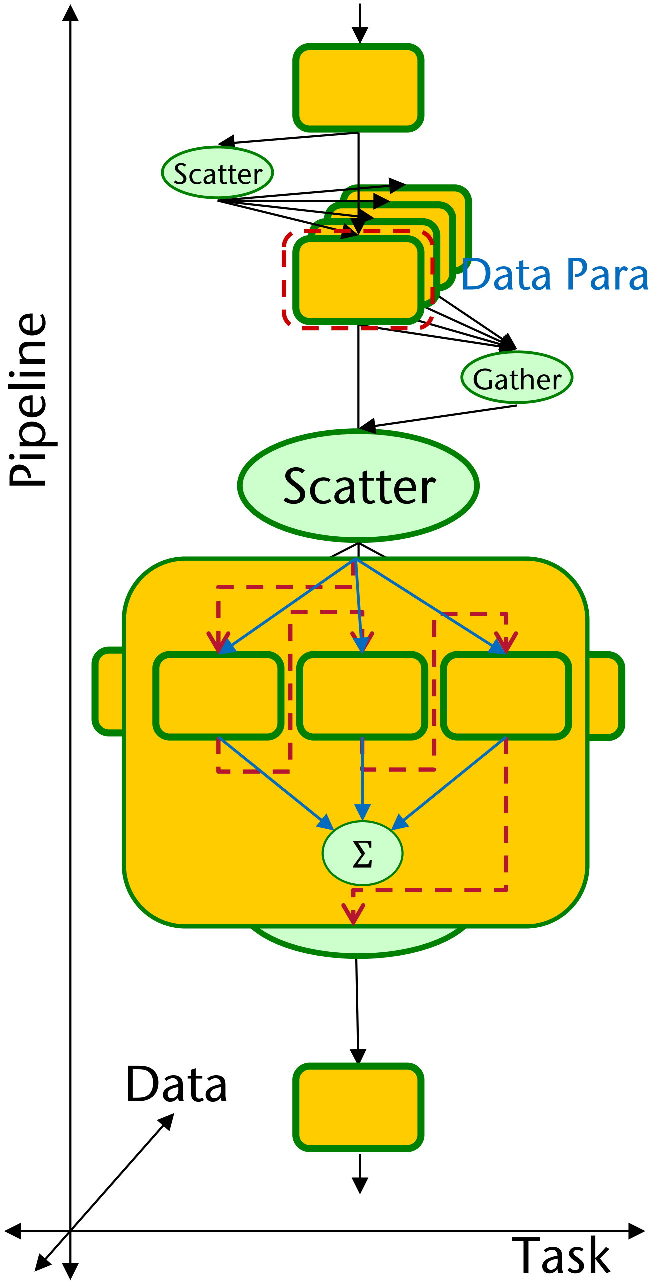
$$\psi = \frac{1-\sqrt{5}}{2} = 1 - \varphi = -\frac{1}{\varphi} \approx -0.6180339887\cdots$$

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1.6180339887\cdots$$

Example of Inherently Sequential Problem (?)

- RSA encryption
  - One RSA operation with a 1k-bit key requires roughly 768 modular multiplications of large integers, and each multiplication is dependent on the result of the previous multiplication
  - Trivial parallelizations are:
    - Parallelize the individual multiplication operation (via, e.g., FFT)
    - Encrypting each packet of the message in parallel
  - If you find a non-trivial parallel algorithm for RSA, please talk to me ☺

Another Taxonomy for Parallelism



- Pipeline parallelism := between producers and consumers
- Task parallelism := explicit in algorithm; each task works on a different branch/section of the control flow graph, where none of the tasks' output reaches the other task as input (similar to MIMD)
  - Sometimes also called thread level parallelism
- Data parallelism := all data packets have to be treated same/similarly (e.g. SIMD)

- An example of data (level) parallelism:

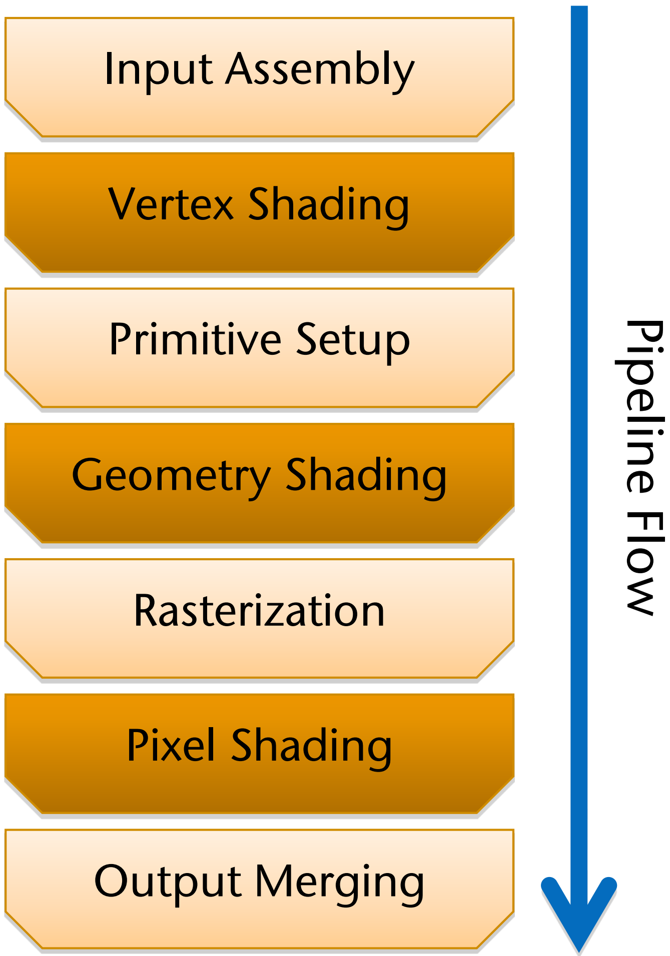
```
do_foo_parallel( array d ):
  if myCPU = "1":
    lower_limit := 0
    upper_limit := d.length / 2
  else if myCPU = "2":
    lower_limit := d.length/2 + 1
    upper_limit := d.length

  for i from lower_limit to upper_limit:
    foo( d[i] )

do_foo_parallel<<on both CPUs>>( global_array )
```

- This is what we are going to do mostly in this course!

- Examples of pipeline parallelism:
  - The graphics (hardware) pipeline (OpenGL / DirectX)
  - The app-cull-draw (software) pipeline



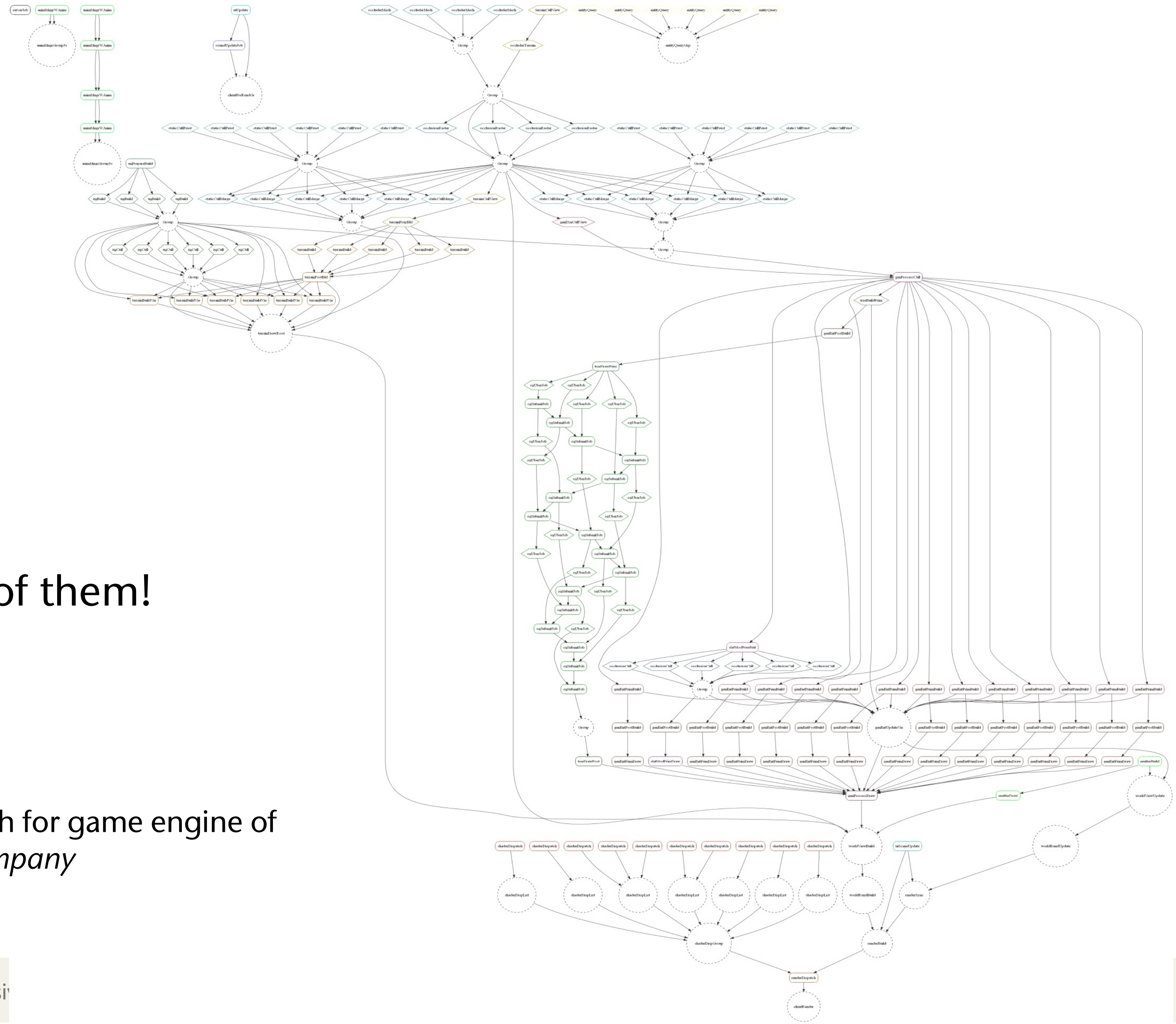
A word about instruction level parallelism (ILP)

- Mostly done inside CPUs / cores
  - I.e., this is parallelism on the hardware level
  - Done by computer architects at the time the hardware is designed
- Example:

```
1: e = a + b
2: f = c + d
3: g = e * f
```

  - Lines 1 & 2 (ADD/MOV instr. for the CPU) can be executed in parallel
- Techniques employed in CPUs to achieve ILP:
  - Instruction pipelining
  - Out-of-order execution
  - Branch prediction

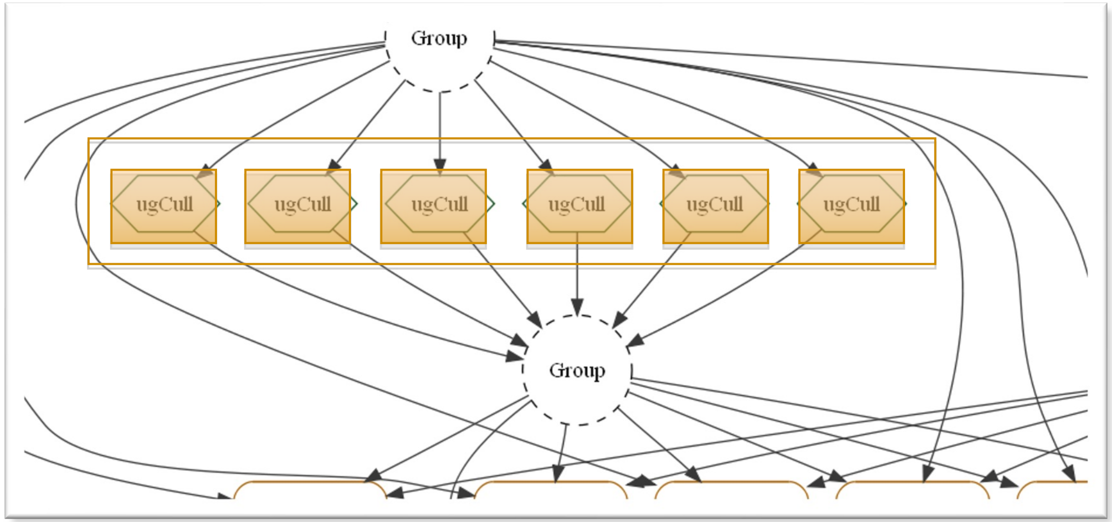
Which Parallelism Paradigm in Daily Life?



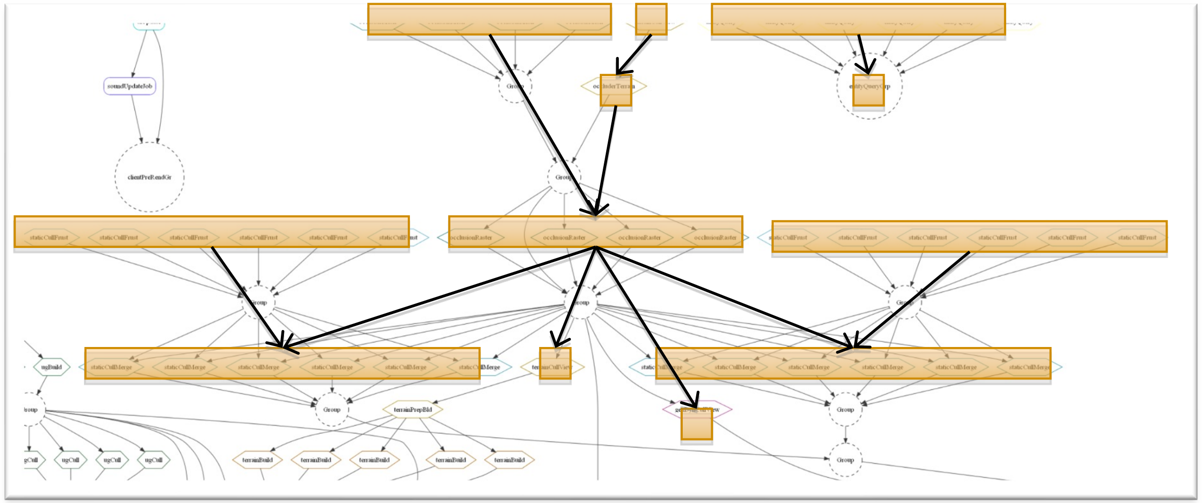
Answer: all of them!

Part of the
computation graph for game engine of
Battlefied: Bad Company
provided by DICE

- Data parallelism:

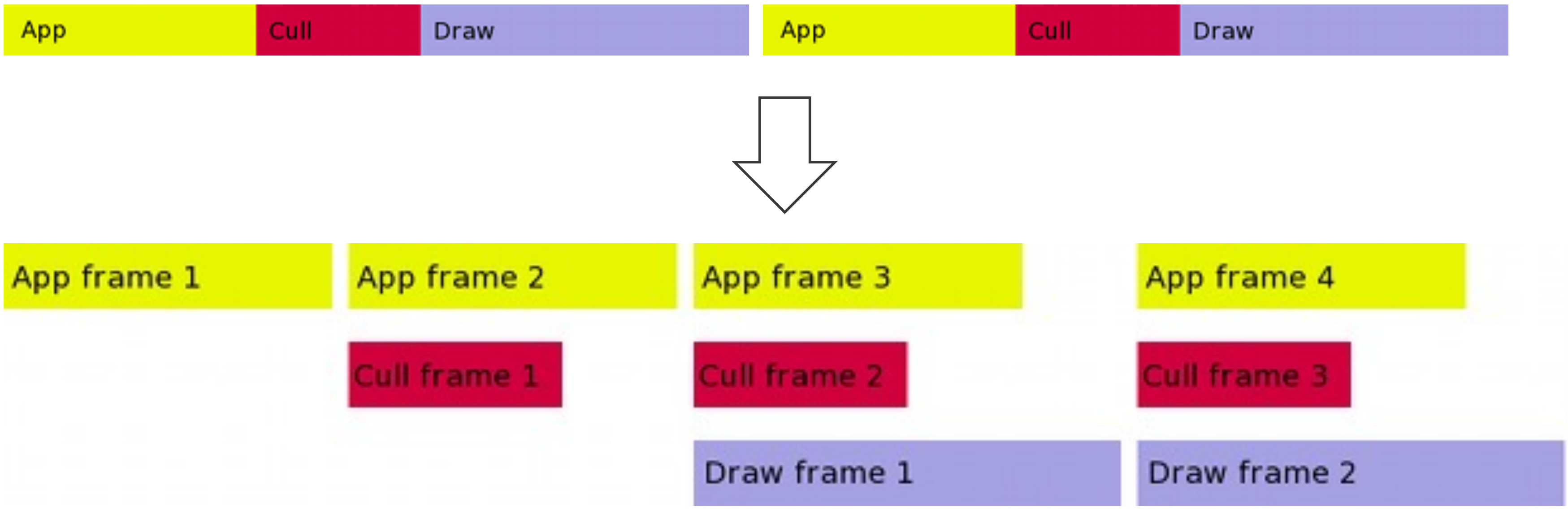


- Task parallelism:



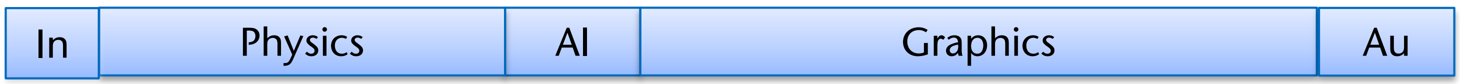
From Tim Foley's "Introduction to Parallel Programming Models"

- Pipeline parallelism:



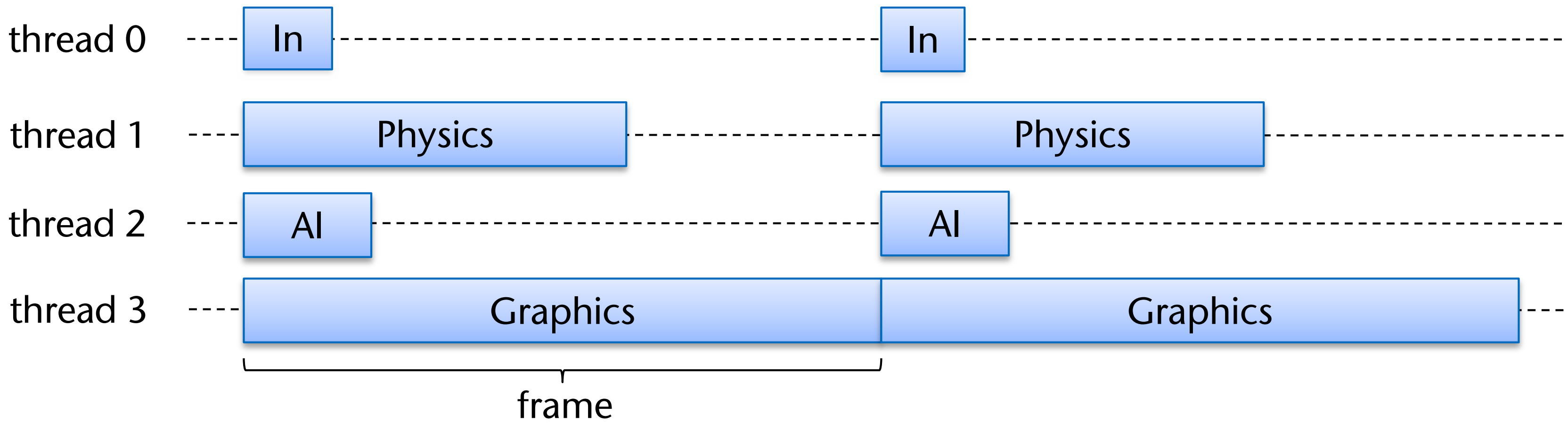
Reconciling Task Parallelism

- Typical game workload (subsystems in % of overall time "budget"):
  - Input, Miscellaneous: 5%
  - Physics: 30%
  - AI, Game Logic: 10%
  - Graphics: 50%
  - Audio: 5%



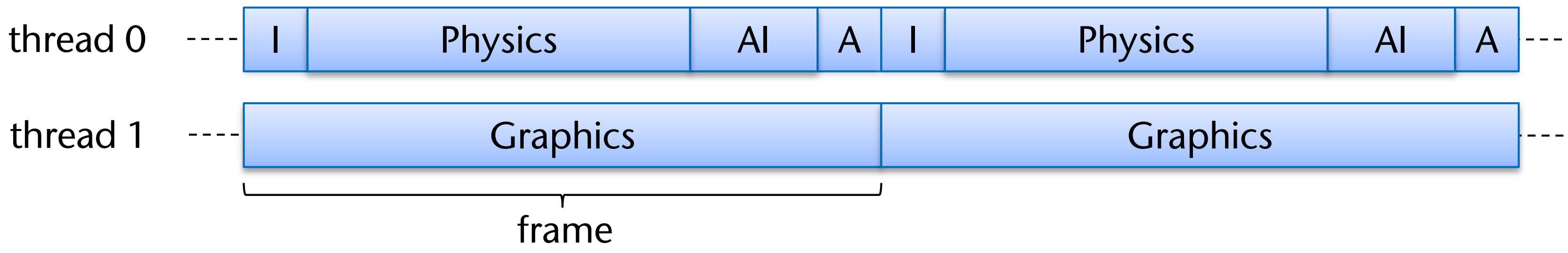
Parallelism Anti-Pattern

- Naïve solution: assign each subsystem to a thread



- Problems
  - Communication/synchronization
  - Load imbalance
  - Preemption could lead to *thrashing*
- Don't do this

- Better: group subsystems into threads with equal load



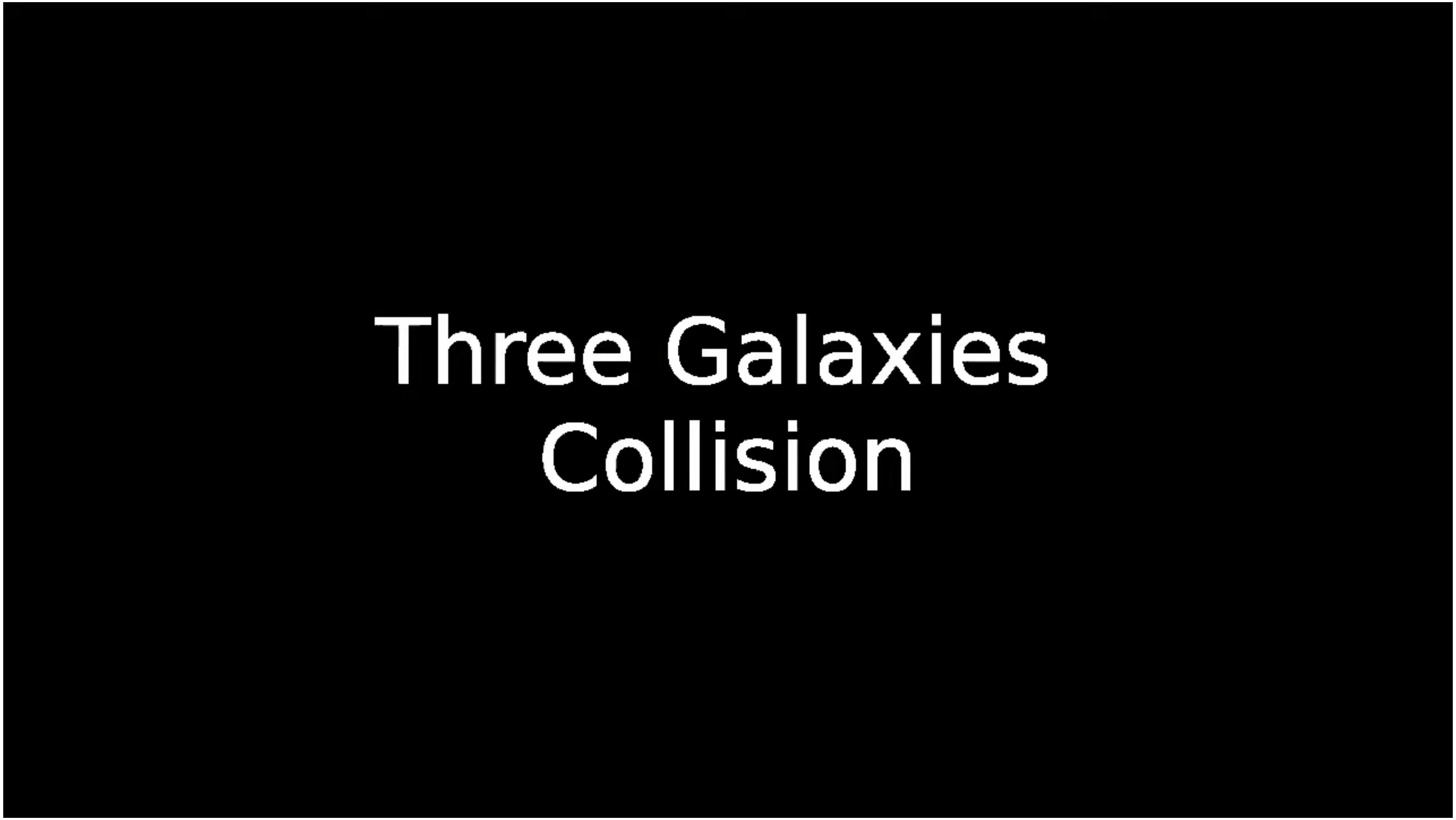
- Problems
  - Communication/synchronization
  - Poor scalability (4, 8, … threads)

Enough Classifications … Demo Time!

Comparison between single core, multi-core, GPU

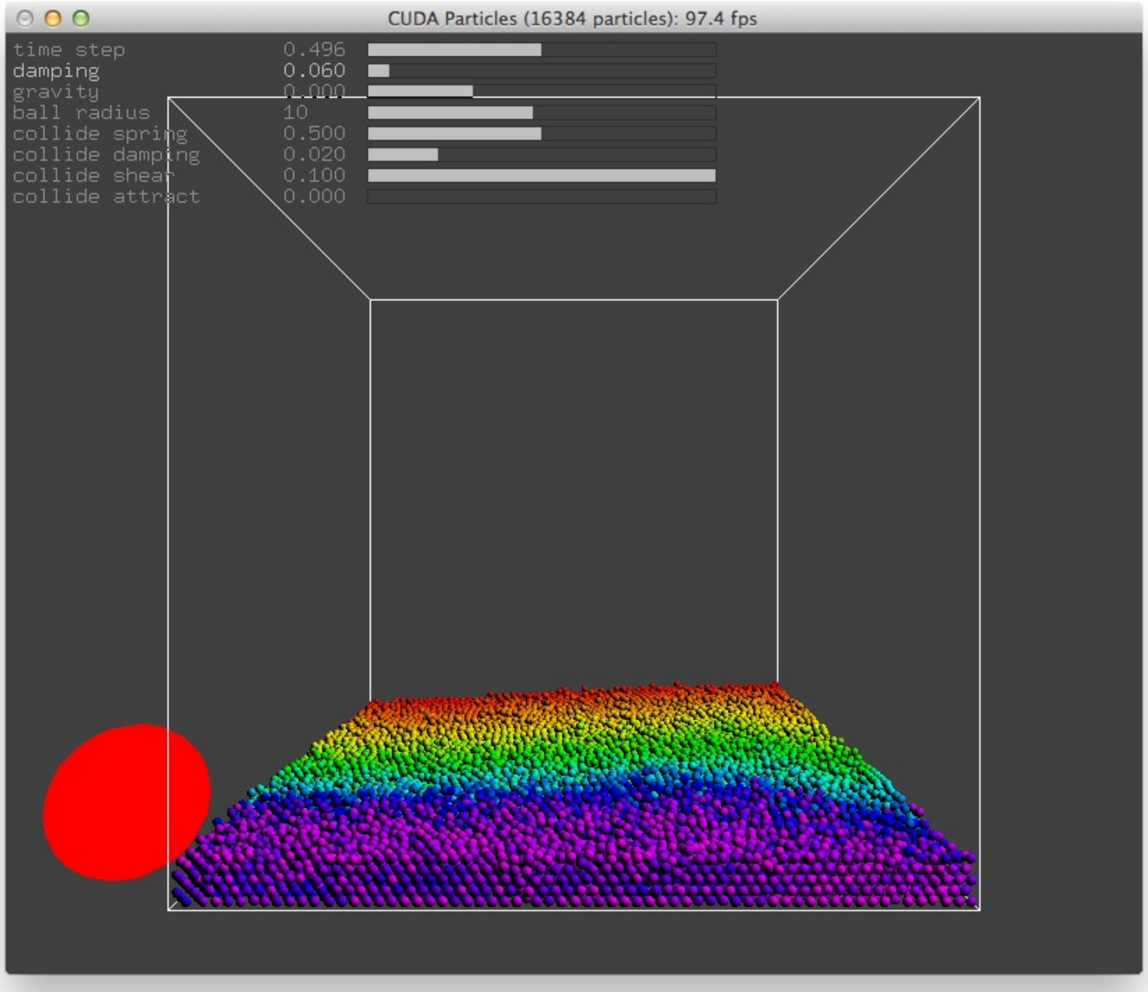
~/Code/MassPar_examples_CUDA_and_OpenCL/OpenCL/NBody_Simulation/

More Examples: Particle / N-Body Sim. (e.g., Galaxy Simulations)



2.4m particles

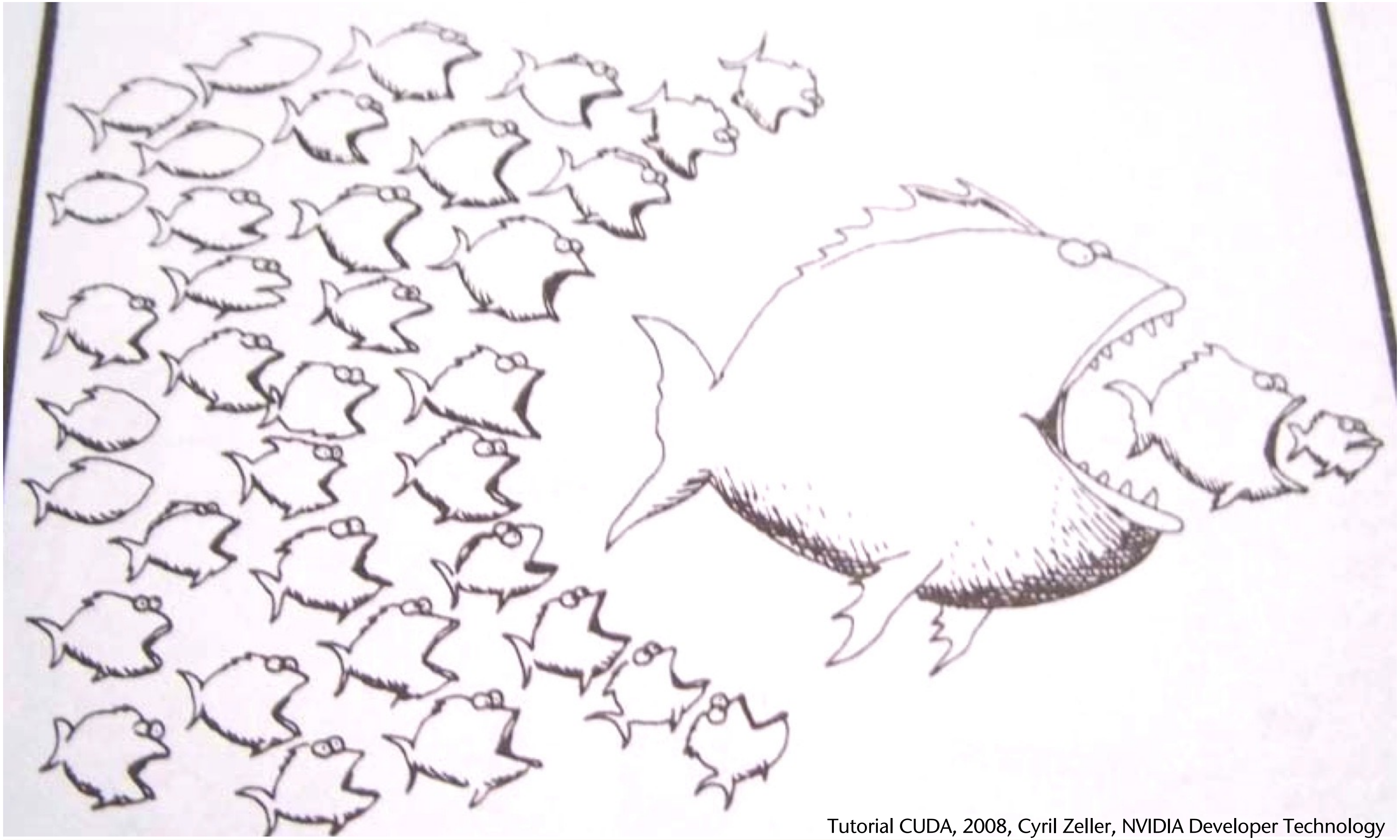
[Wojciech Mo]



~/Code/MassPar_examples_CUDA_and_OpenCL/CUDA/particles


Simulation mode: HW
F1 for help
~30fps

Illustrated History of Parallel Computing



Tutorial CUDA, 2008, Cyril Zeller, NVIDIA Developer Technology