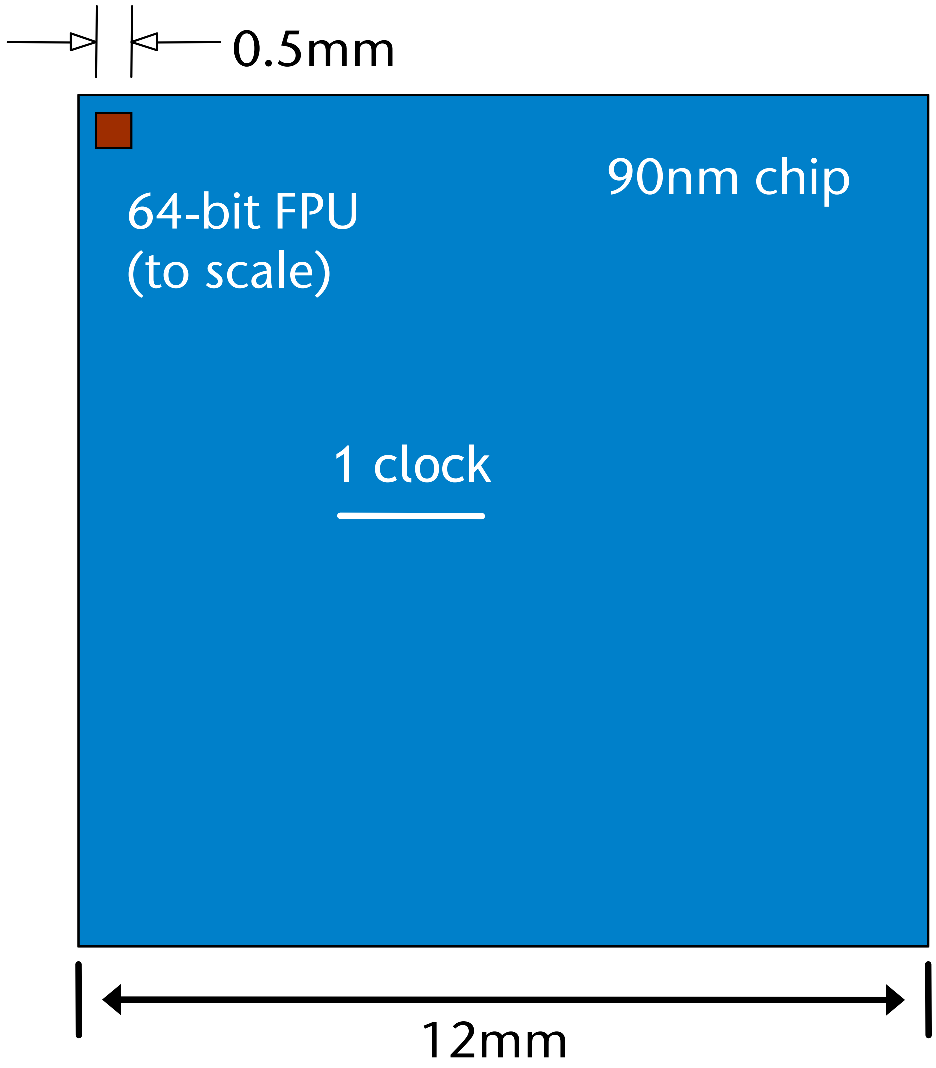
Massively Parallel Algorithms
Introduction

G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de

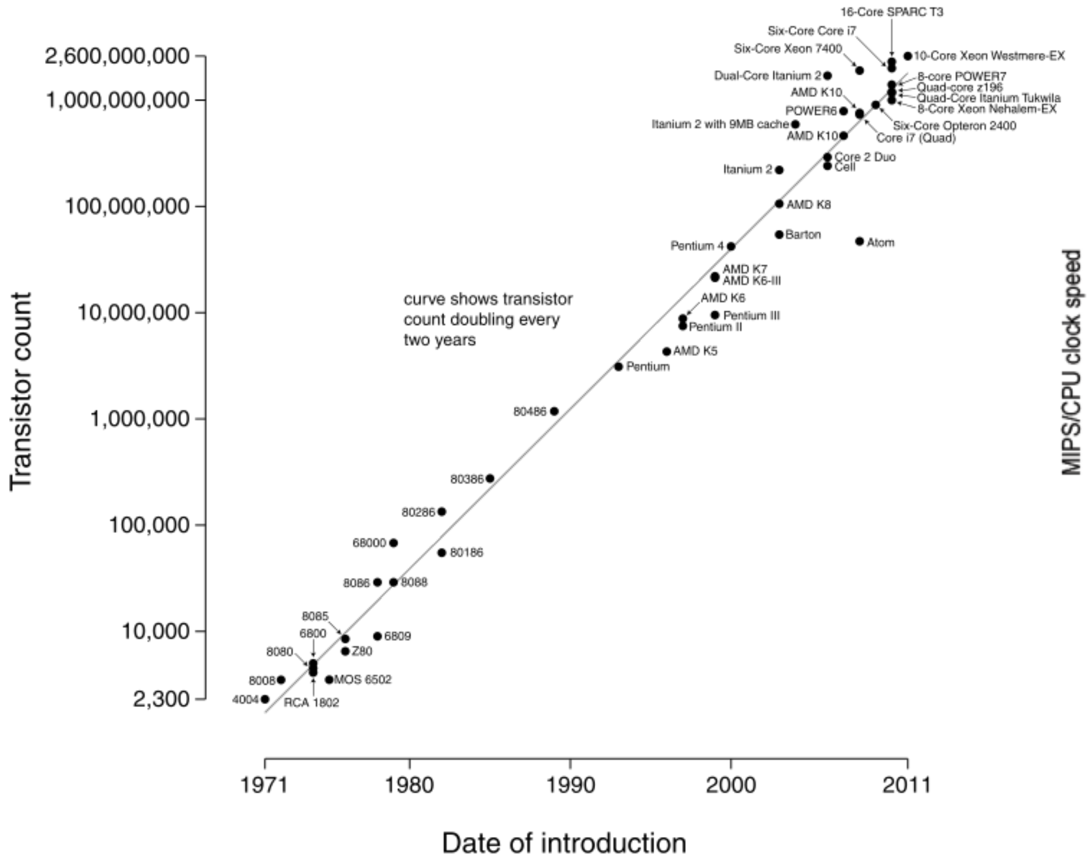
Why Massively Parallel Computing?

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

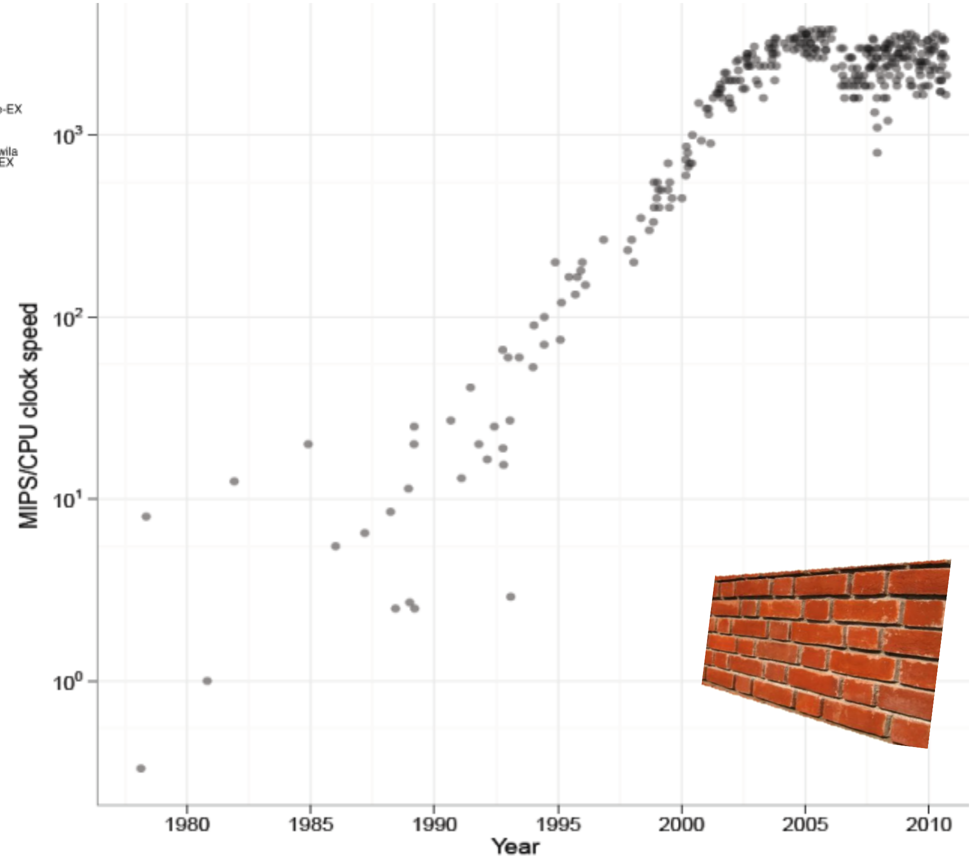


Moore's Law & The Brick Wall

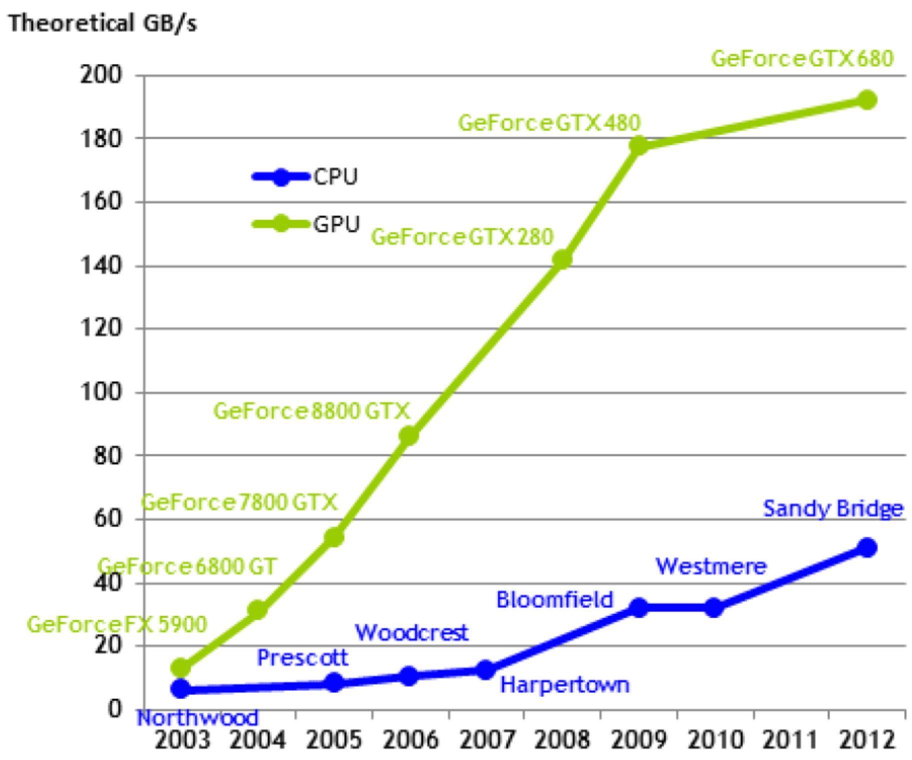
Moore's Law (it's really only an observation)



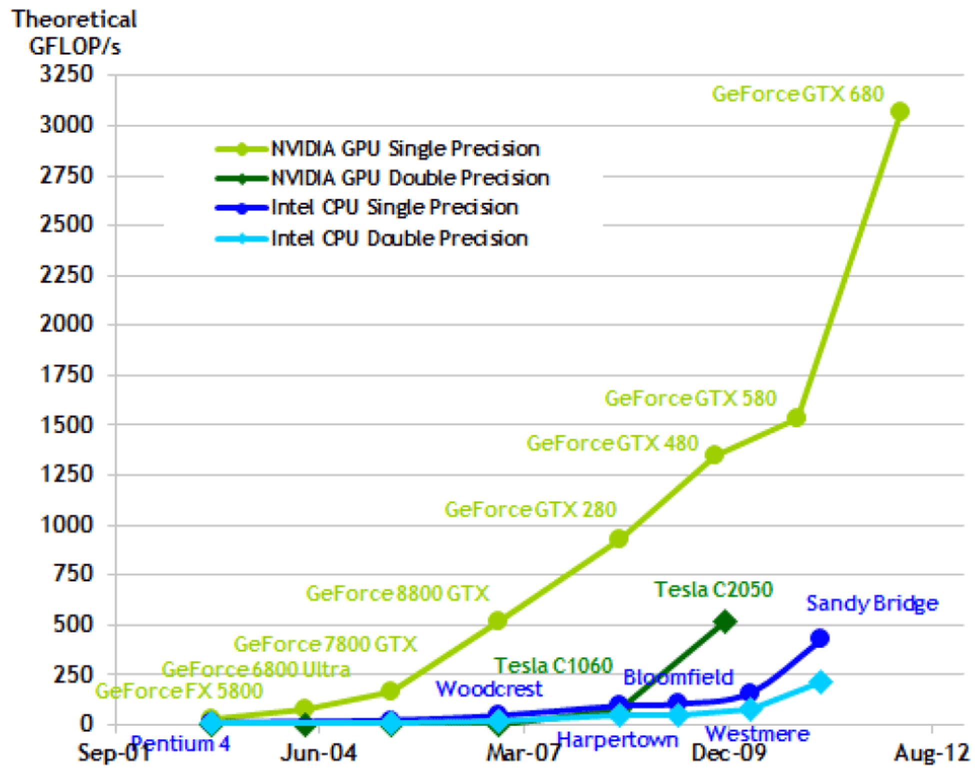
CPU speed



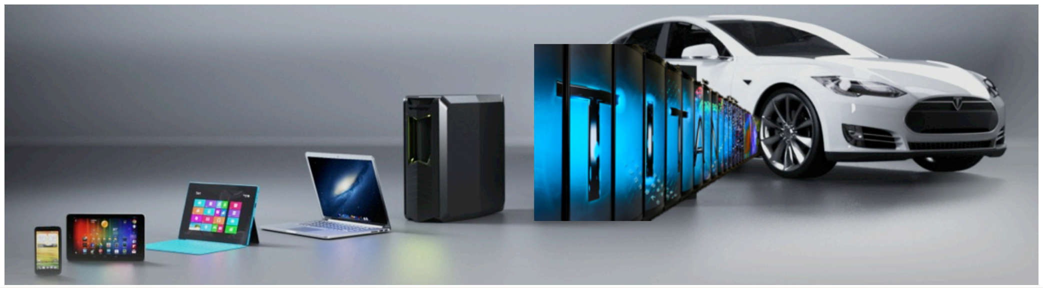
"More Moore" with GPUs

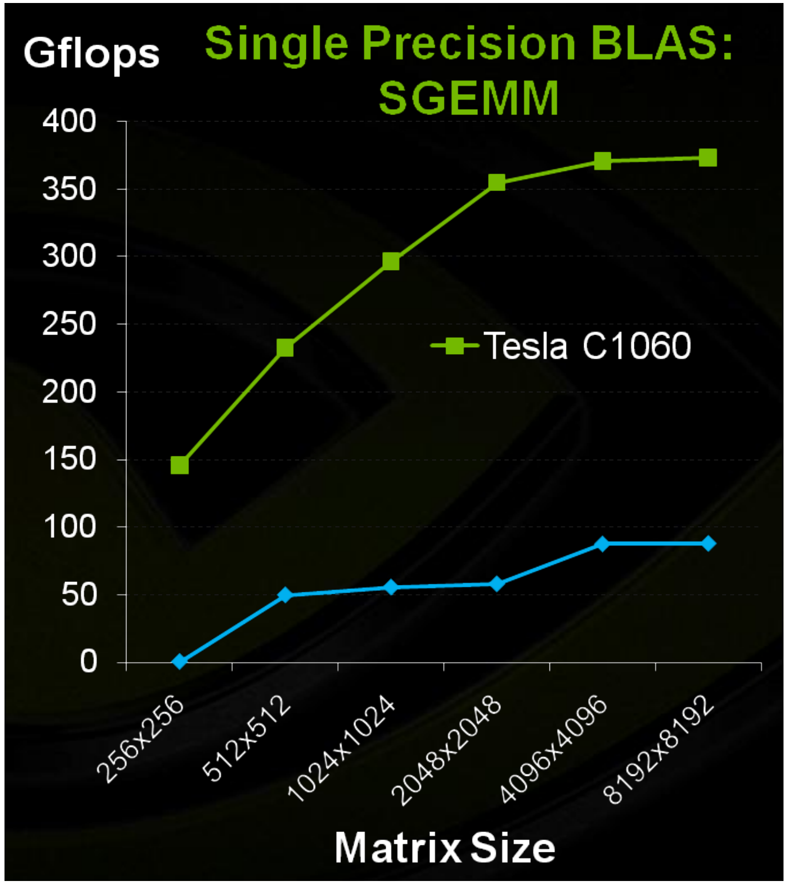


Memory Bandwidth

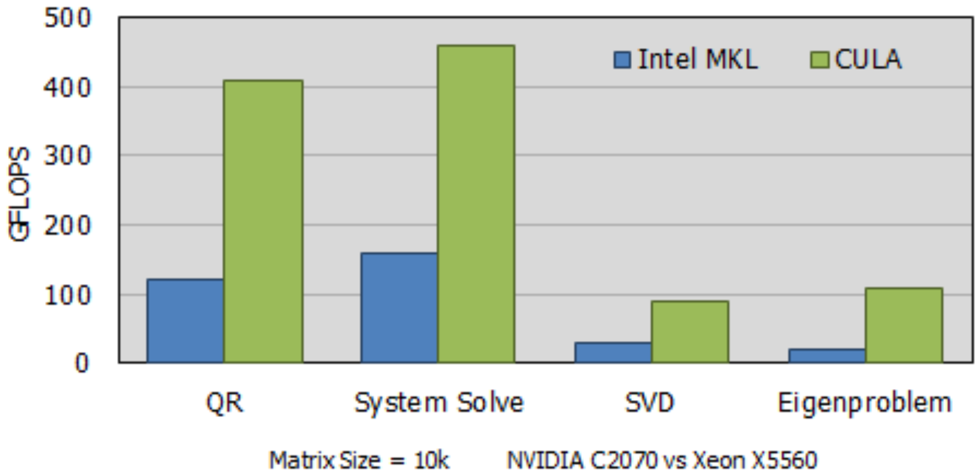


Theoretical Peak Performance





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



GPU Accelerated Libraries ("Drop-In Acceleration)

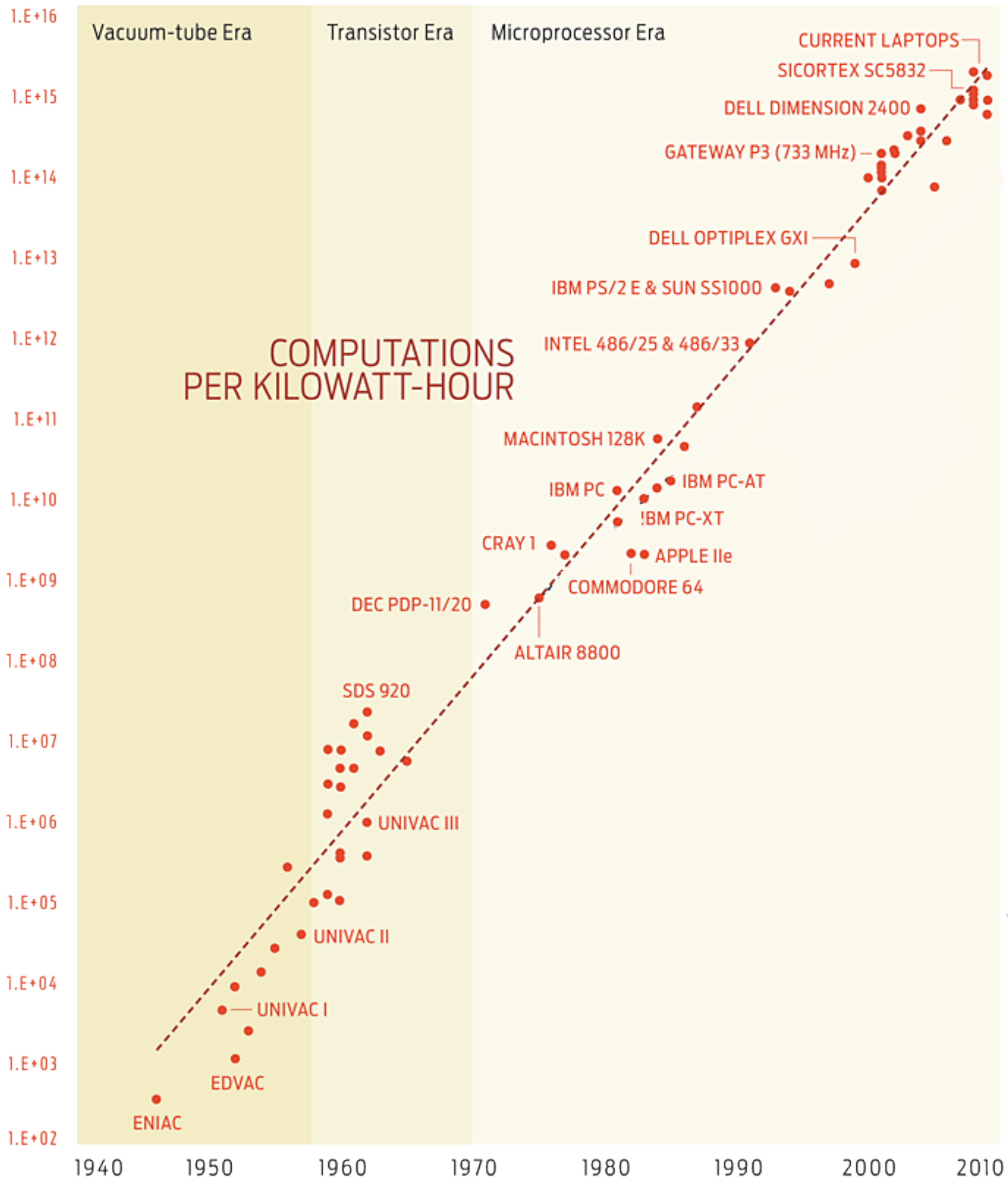
When Power Consumption Matters

- 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
  low parallelism at high clock frequencies (550 Mhz)
  - Power dissipation increases super-linearly with frequency

⇩

The Trend of Electrical Efficiency of Computation

- Like Moore's law, there is a trend towards more compute power per kWh



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

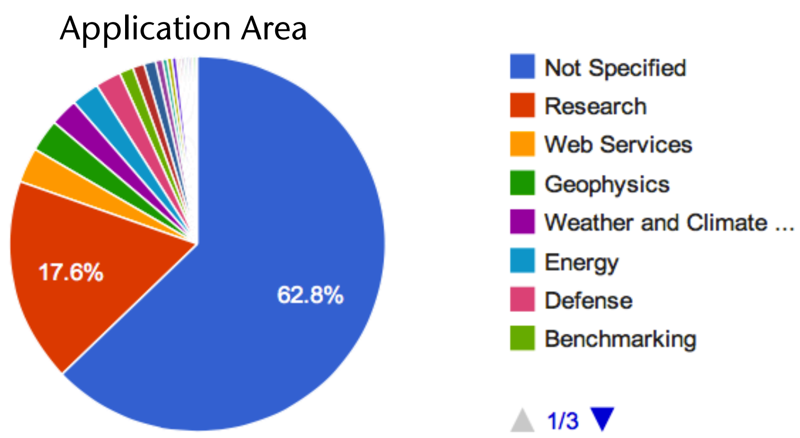
Assessing Trends in the Electrical Efficiency of Computation Over Time" Koomey et al., 2009

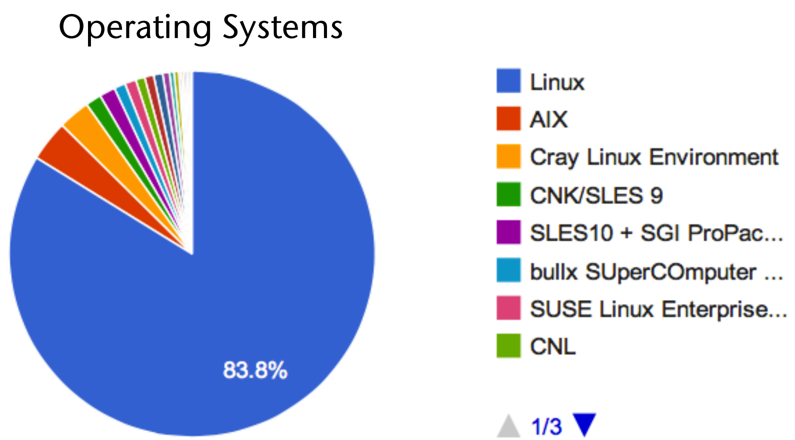
Areas Benefitting from Massively Parallel Algos

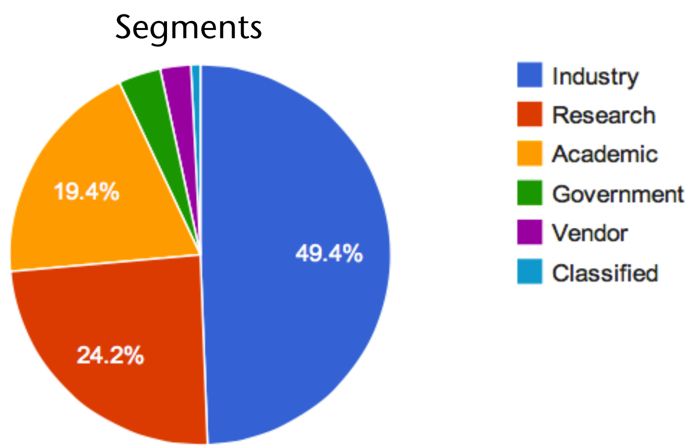
- Computer science (e.g., visual computing, database search)
- Computational material science (e.g., molecular dynamics sim.)
- Bio-informatics (e.g., alignment, sequencing, …)
- Economics (e.g., simulation of financial models)
- Mathematics (e.g., solving large PDEs)
- Mechanical engineering (e.g., 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

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







- Our target platform (GPU) is being used among the TOP500 [Nov 2012]:

Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x

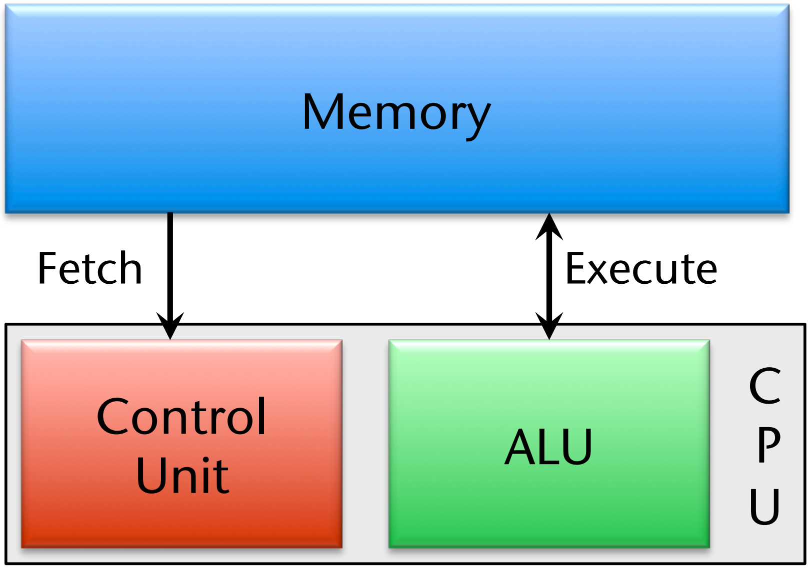
Site:	DOE/SC/Oak Ridge National Laboratory
System URL:	http://www.olcf.ornl.gov/titan/
Manufacturer:	Cray Inc.
Cores:	560640
Linpack Performance (Rmax)	17590.0 TFlop/s
Theoretical Peak (Rpeak)	27112.5 TFlop/s
Power:	8209.00 kW
Memory:	710144 GB
Interconnect:	Cray Gemini interconnect
Operating System:	Cray Linux Environment

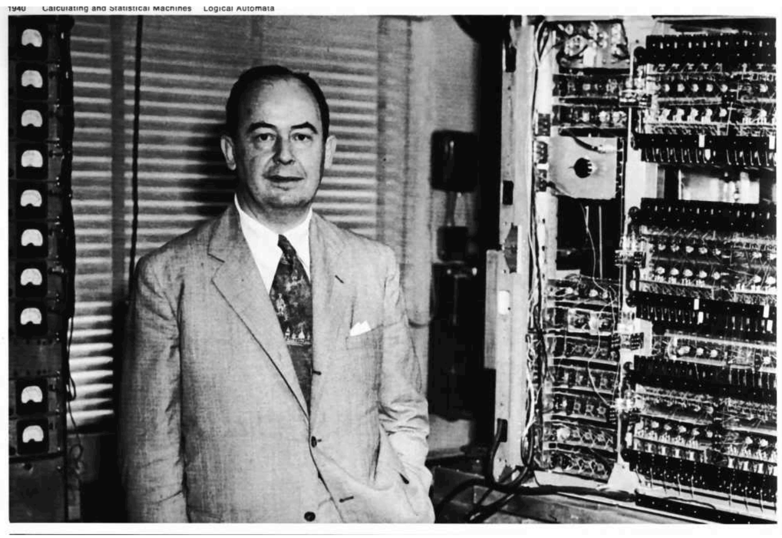
List	Rank	System	Vendor	Total Cores	Rmax (TFlops)	Rpeak (TFlops)	Power (kW)
11/2012	1	Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x	Cray Inc.	560640	17590.0	27112.5	8209.00

Source: www.top500.org

The Von-Neumann Architecture

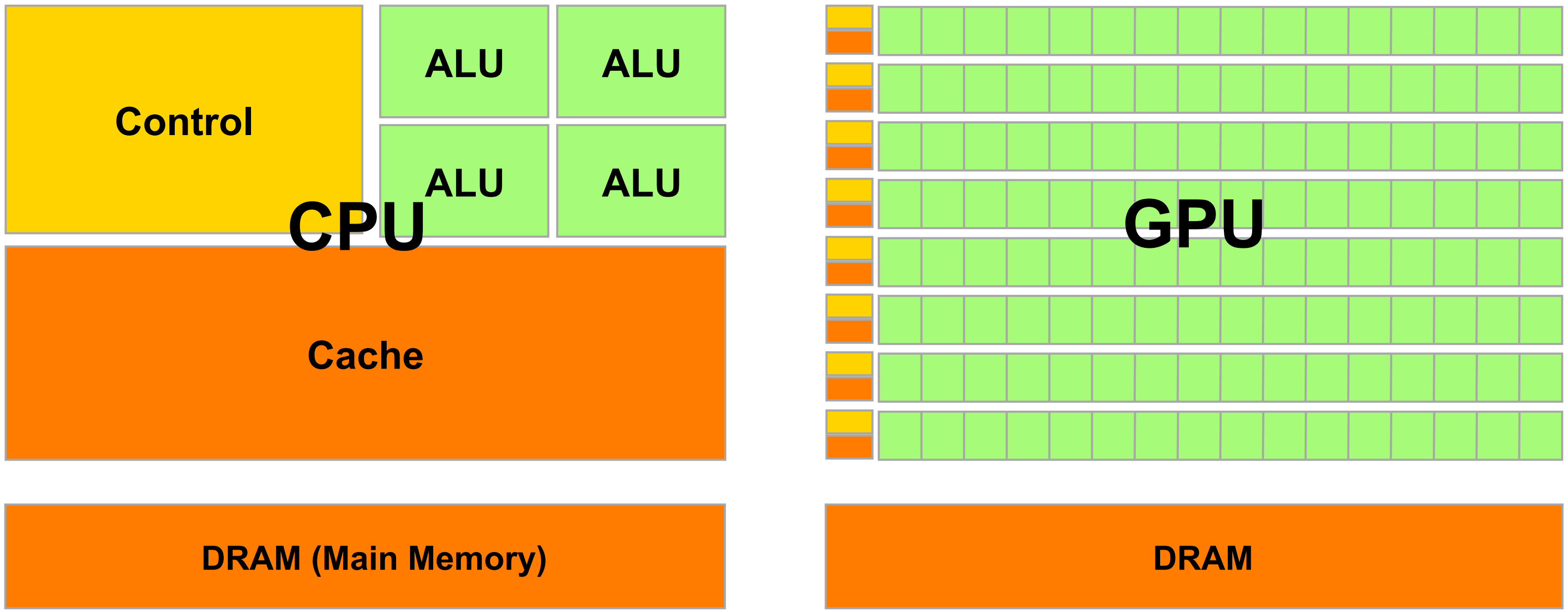
- Uses the stored-program concept (revolutionary at the time of its conception)
- Memory is used for both program instructions and data





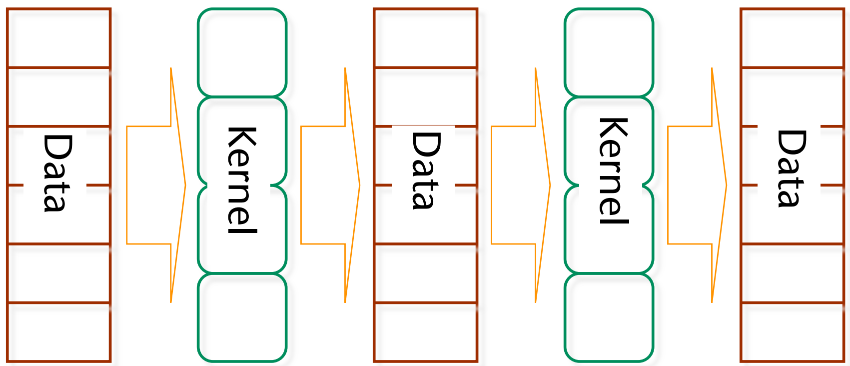
The GPU = the New Architecture

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



The *Stream Programming Model*

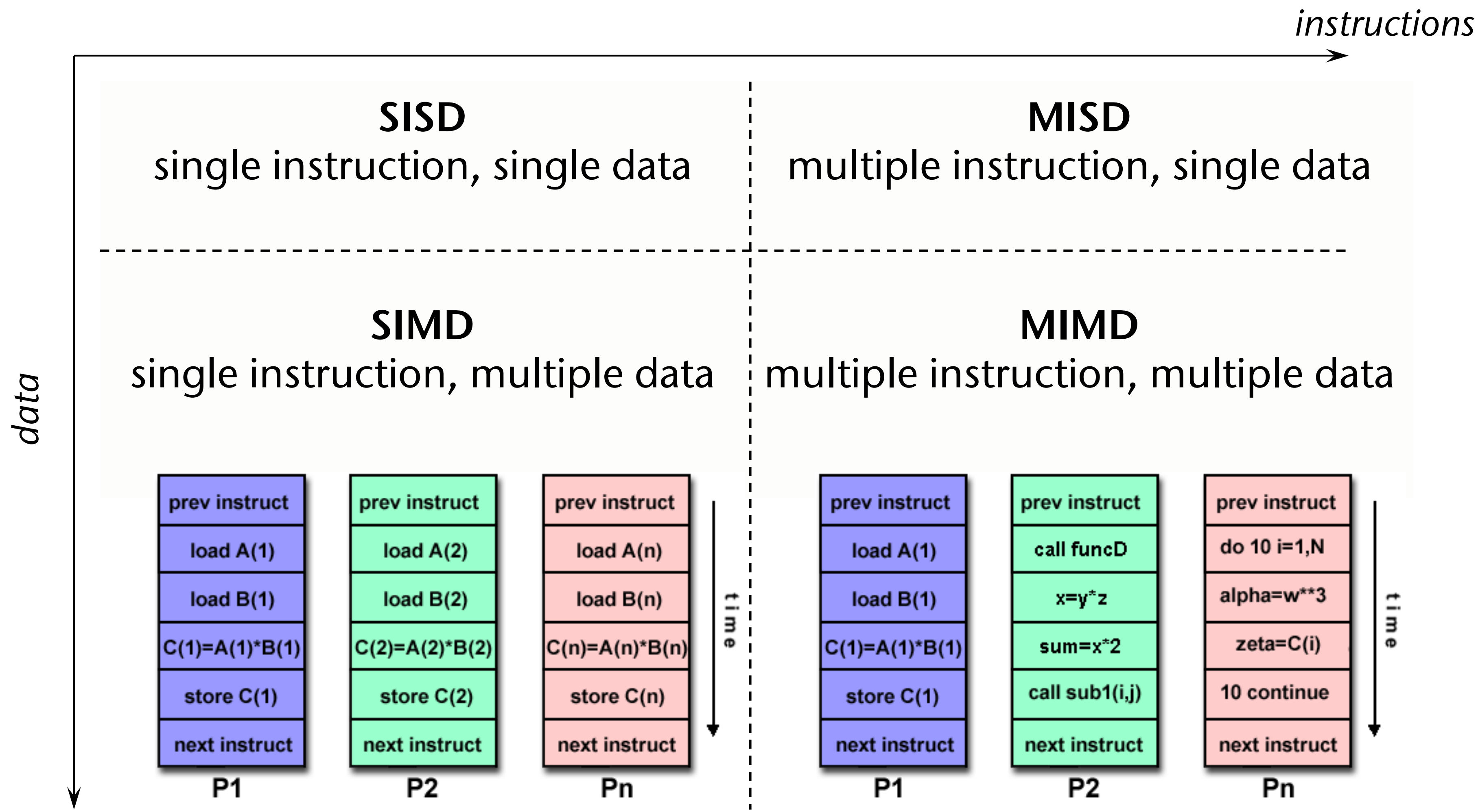
- Novel programming paradigm that tries to organise data & functions such that (as much as possible) only *streaming memory access* will be done, and as little *random access* as possible:
  - Stream Programming Model = *"Streams of data passing through computation kernels."*
  - *Stream* := ordered, 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

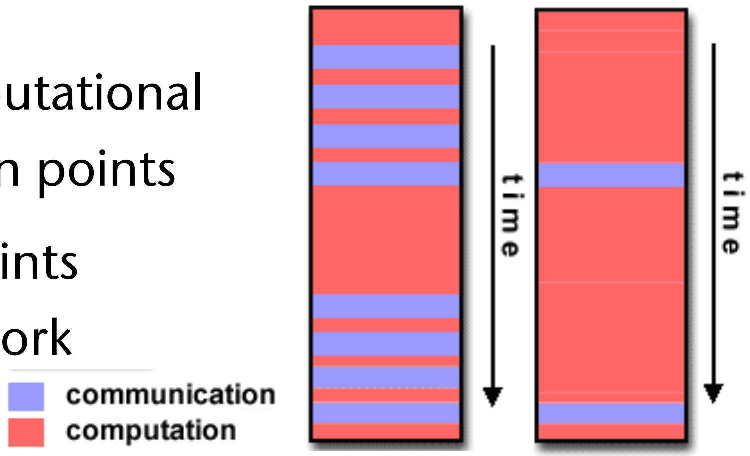
- Two dimensions: instructions and data
- Two values: single and multiple



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

- **Synchronization** := coordination of parallel tasks, very often associated with communications; often implemented by establishing a **synchronization point** within an application where 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



- 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

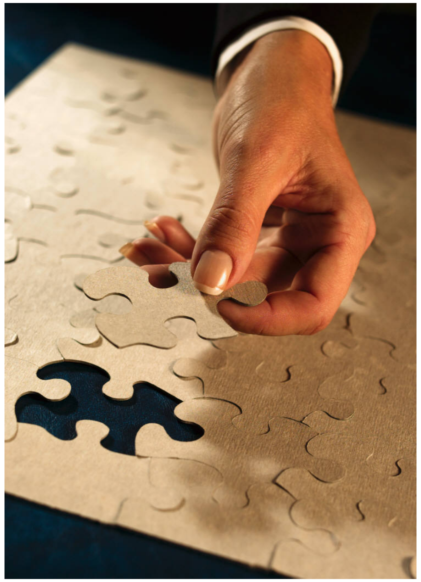
- Observed Speedup := measure for performance of parallel code

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

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

Amdahl's Law

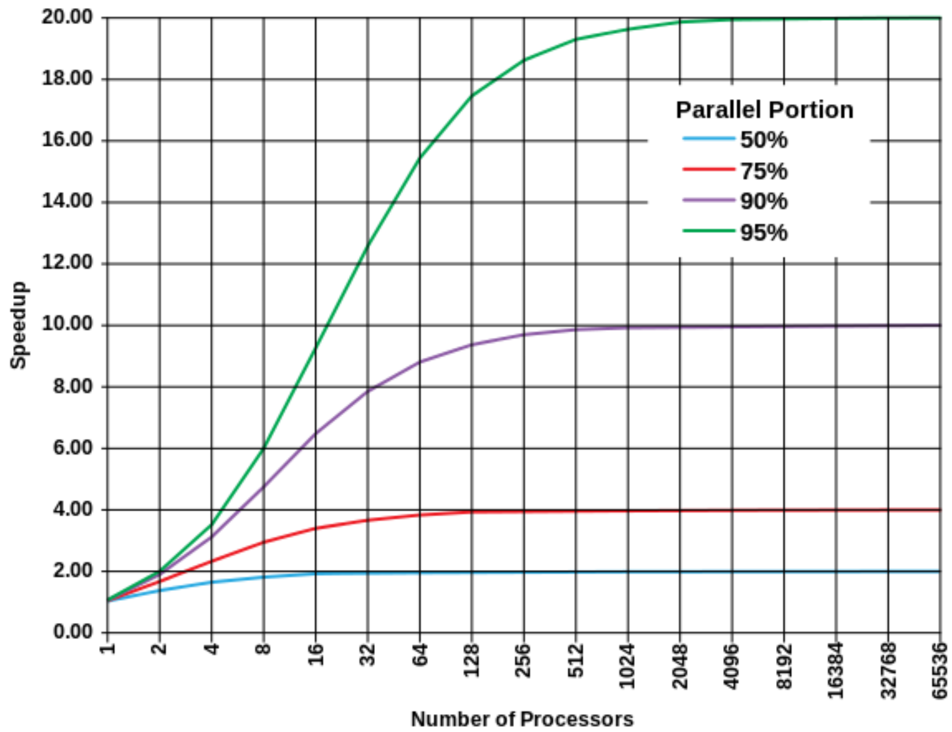
- Quick discussion:
  - Suppose we want to do a 5000 piece jigsaw puzzle
  - Time for one person to complete puzzle: *n* hours
  - How much time do we need, if we add 1 more person at the table?
  - How much time, if we add 100 persons?



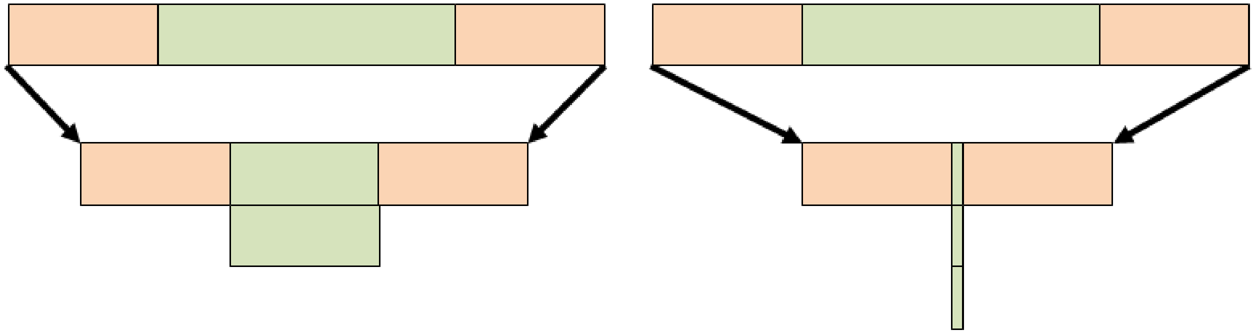
Amdahl's Law (the "Pessimist")

- Assume a program execution consists of two parts: *P* and *S*
- *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:

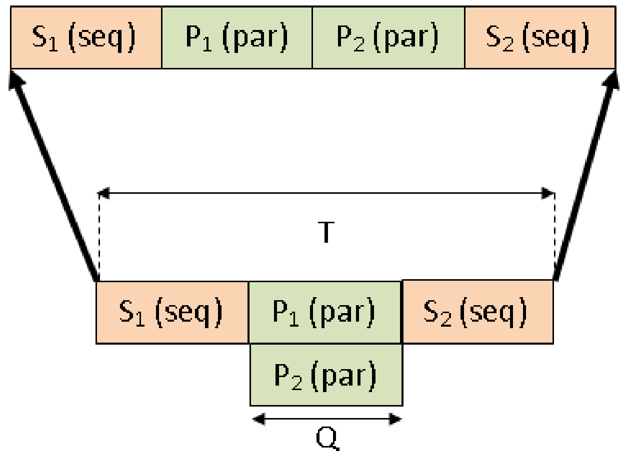


(You can squeeze the parallel part as much as you like, by throwing more processors at it, but you cannot squeeze the sequential part)

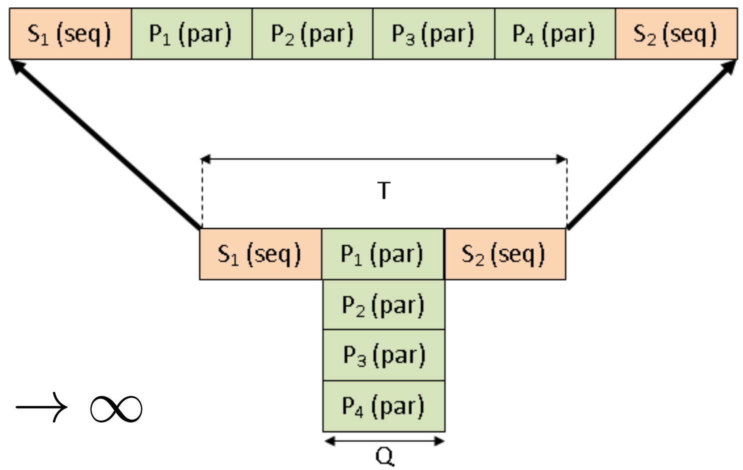
- Parallel Overhead := amount of time required to coordinate parallel tasks, as opposed to doing useful work; can include factors such as: task start-up time, synchronizations, data communications, etc.
- Scalable problem := problem where parallelizable part P increases with problem size

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

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