Introduction to Parallel & Distributed Computing

Lecture 1. Introduction

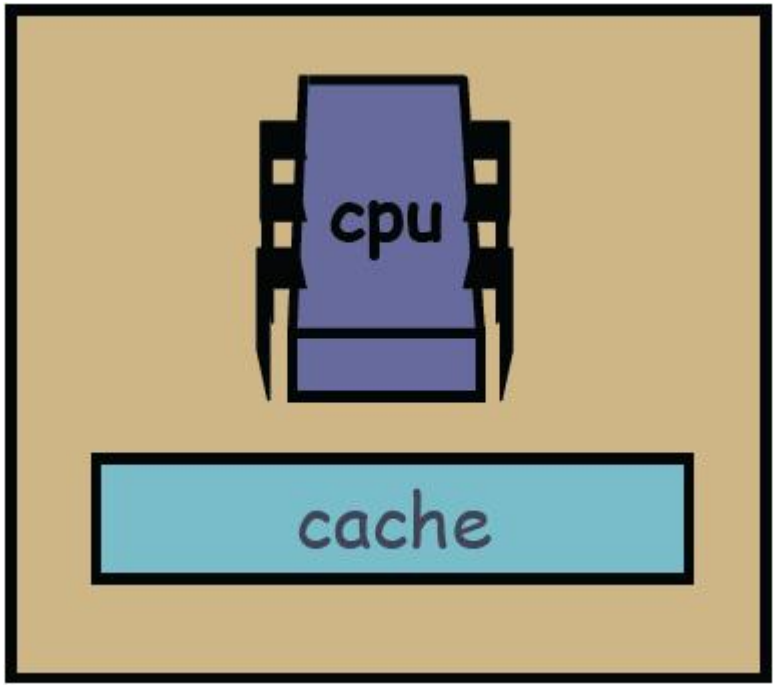
Spring 2024

Instructor: Guojie Luo (罗国杰)

gluo@pku.edu.cn

Vanishing from your Desktops: the Uniprocessor

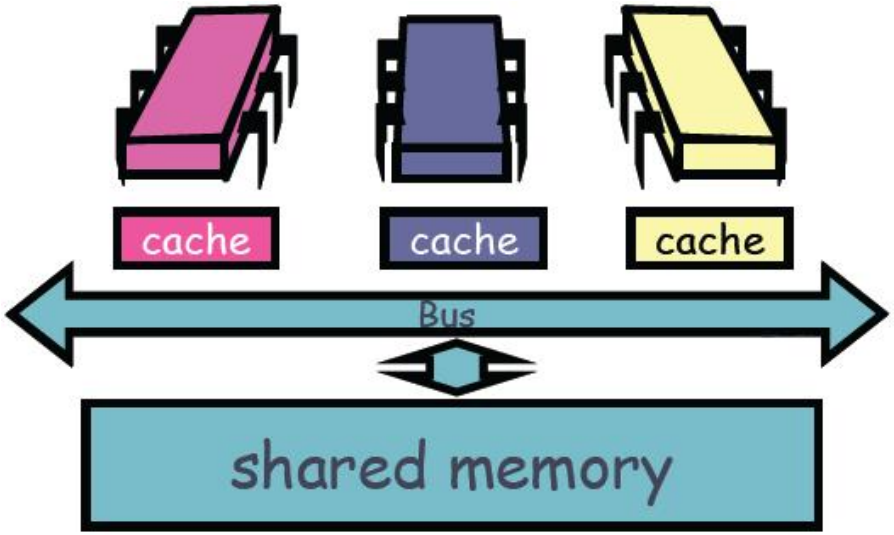
- **Uniprocessor**
  - Single processor plus associated caches(s) on a chip
  - Traditional computer until 2003
  - Supports traditional *sequential* programming model
- **Where can you still find them?**



Source: Herlihy & Shavit, Art of Multiprocessor Programming
Source: CS133 Spring 2010 at UCLA (Kaplan)

Traditional Server: Shared Memory Multiprocessor (SMP)

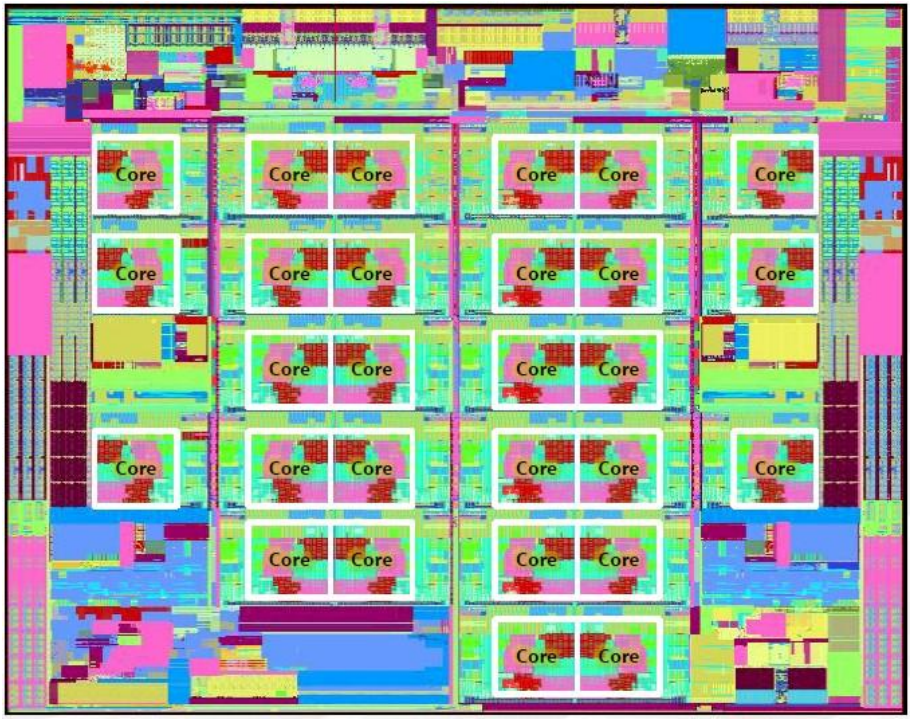
- **Multi-chip computer systems**
  - **High-performance computing**
    - **Servers**
    - **Supercomputers**
- **Each processor chip had a CPU and cache**
  - **Multiple-chips connected by a bus to a shared main memory**

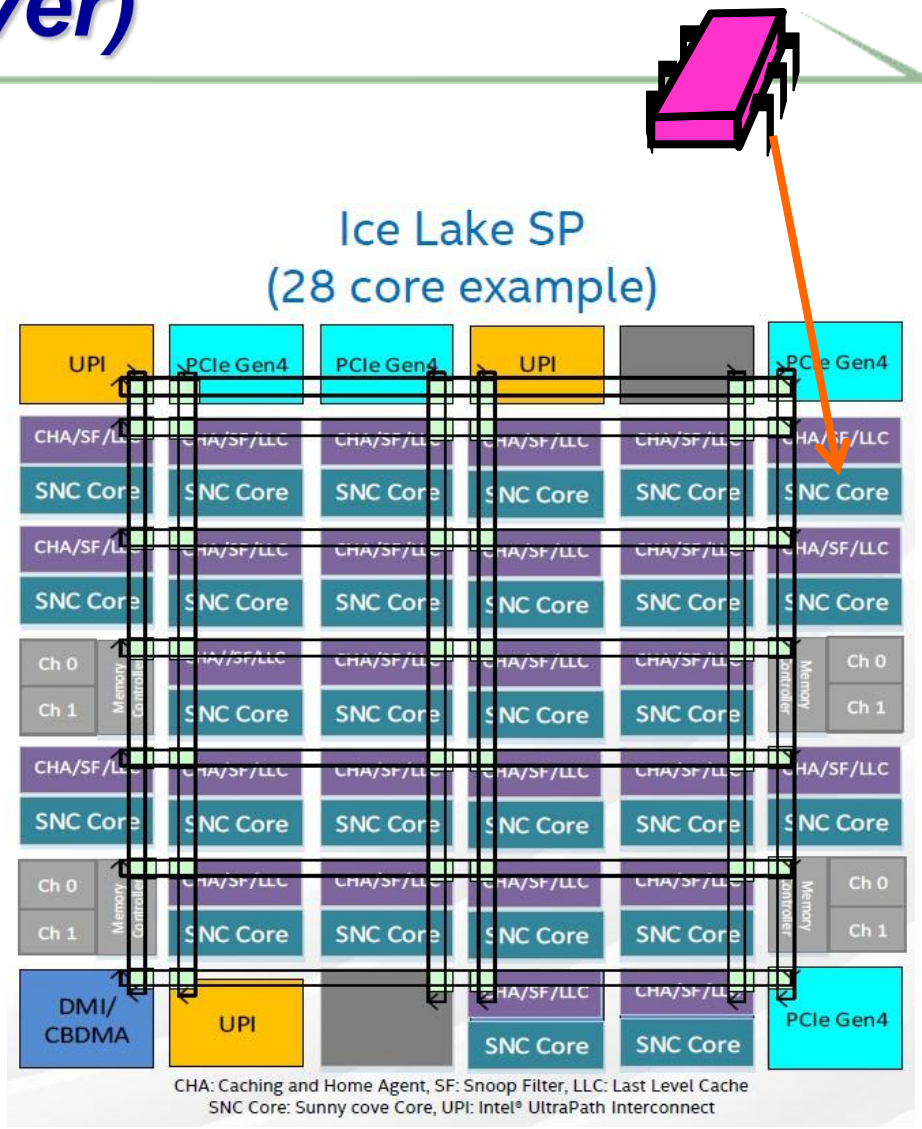


Source: Herlihy & Shavit, Art of Multiprocessor Programming
Source: CS133 Spring 2010 at UCLA (Kaplan)

Processor Chip (Server)

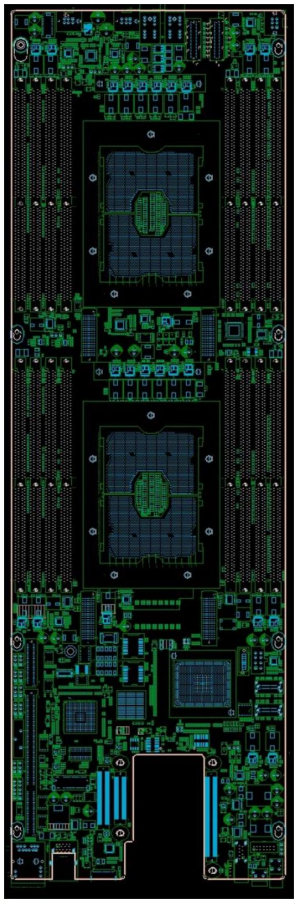
- All on the same chip

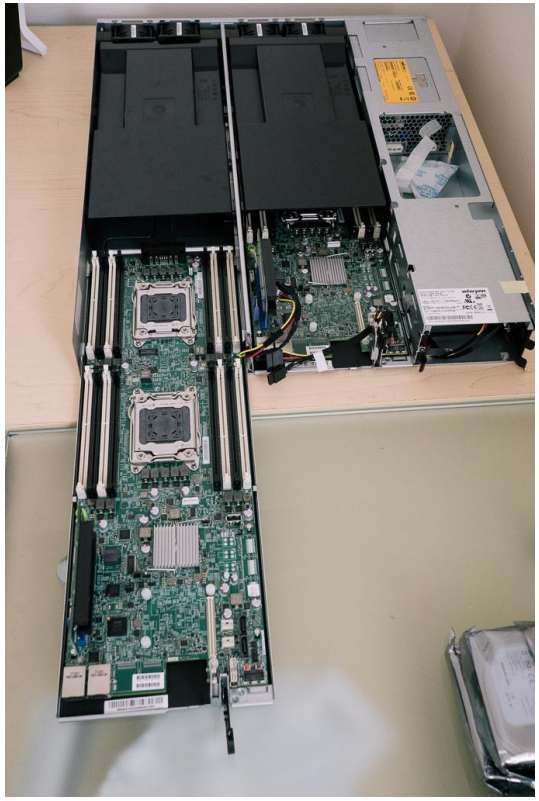


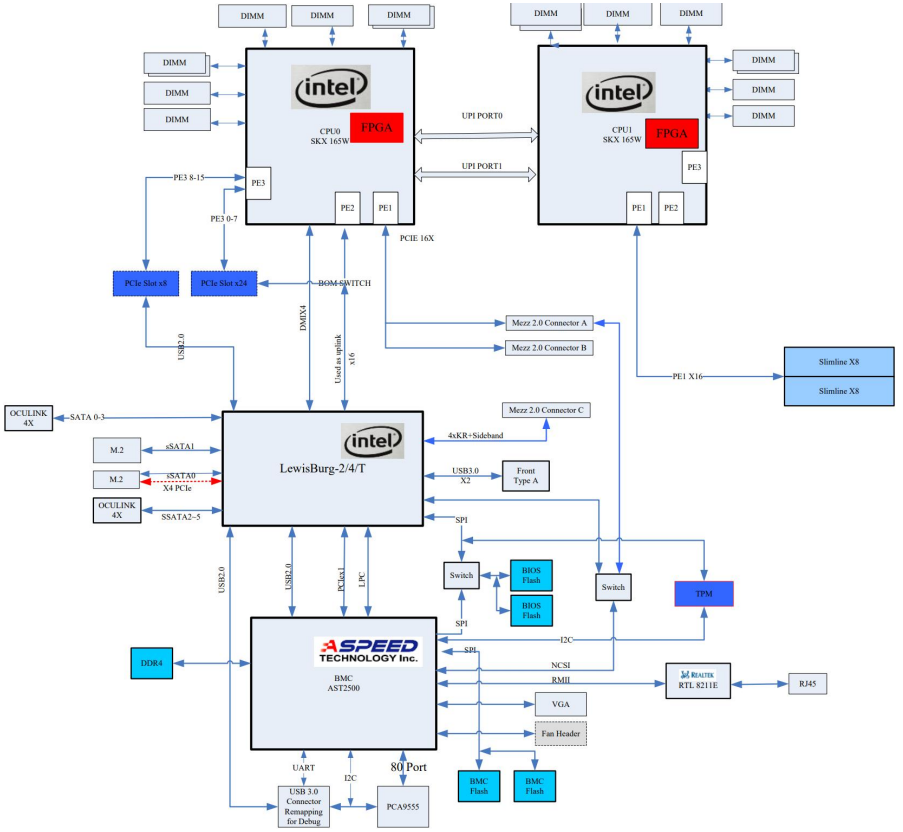


https://www.nextplatform.com/2021/04/19/deep-dive-into-intels-ice-lake-xeon-sp-architecture/

Compute Node Platform (e.g., 2 Sockets with 16-channel DDR4)



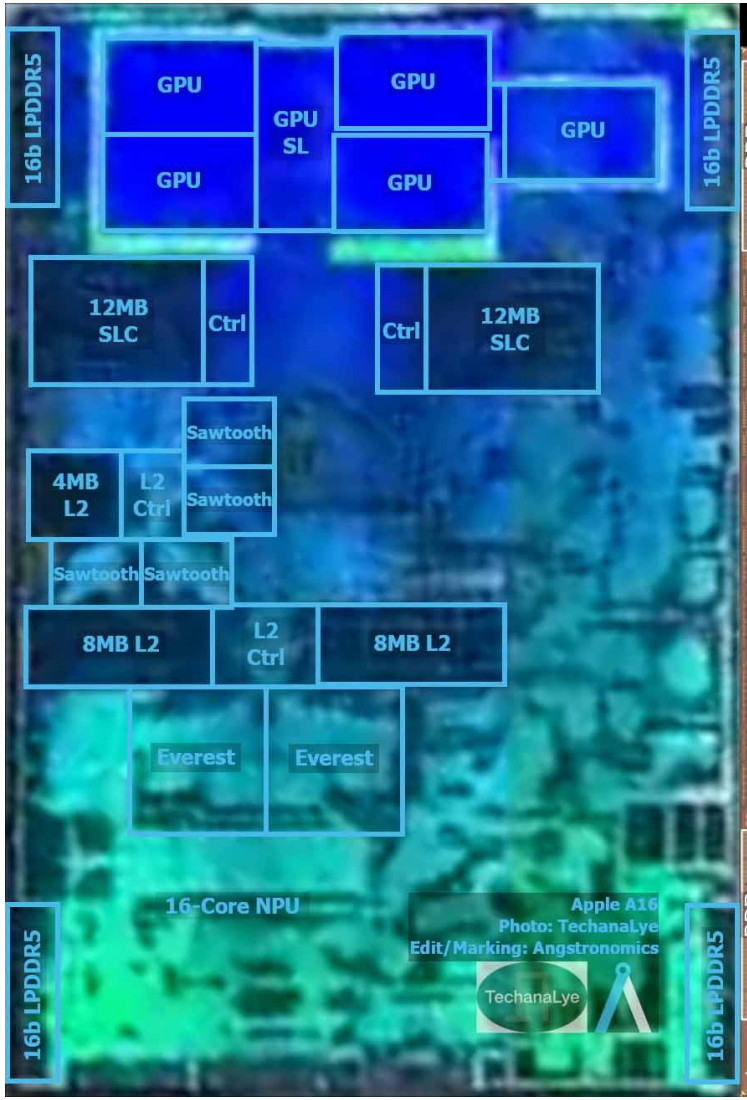




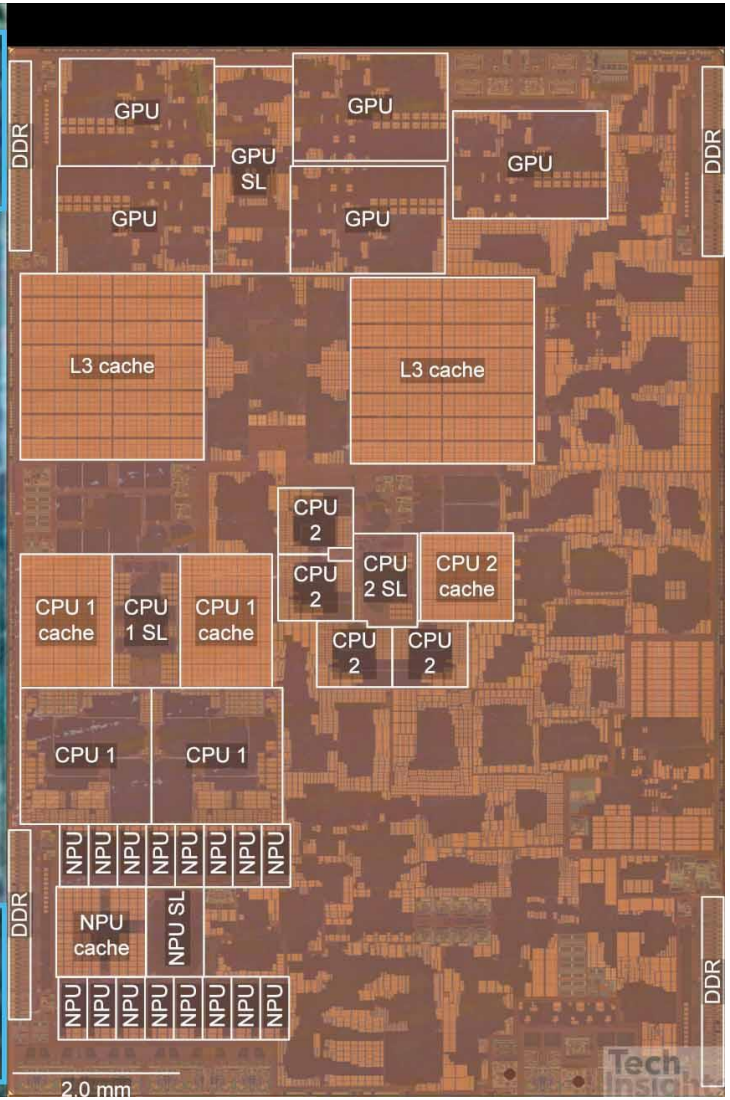
source: opencompute.org, inspursystems.com, ployfractal.com

Even Cell Phone Has Multiple Cores!

Apple A16

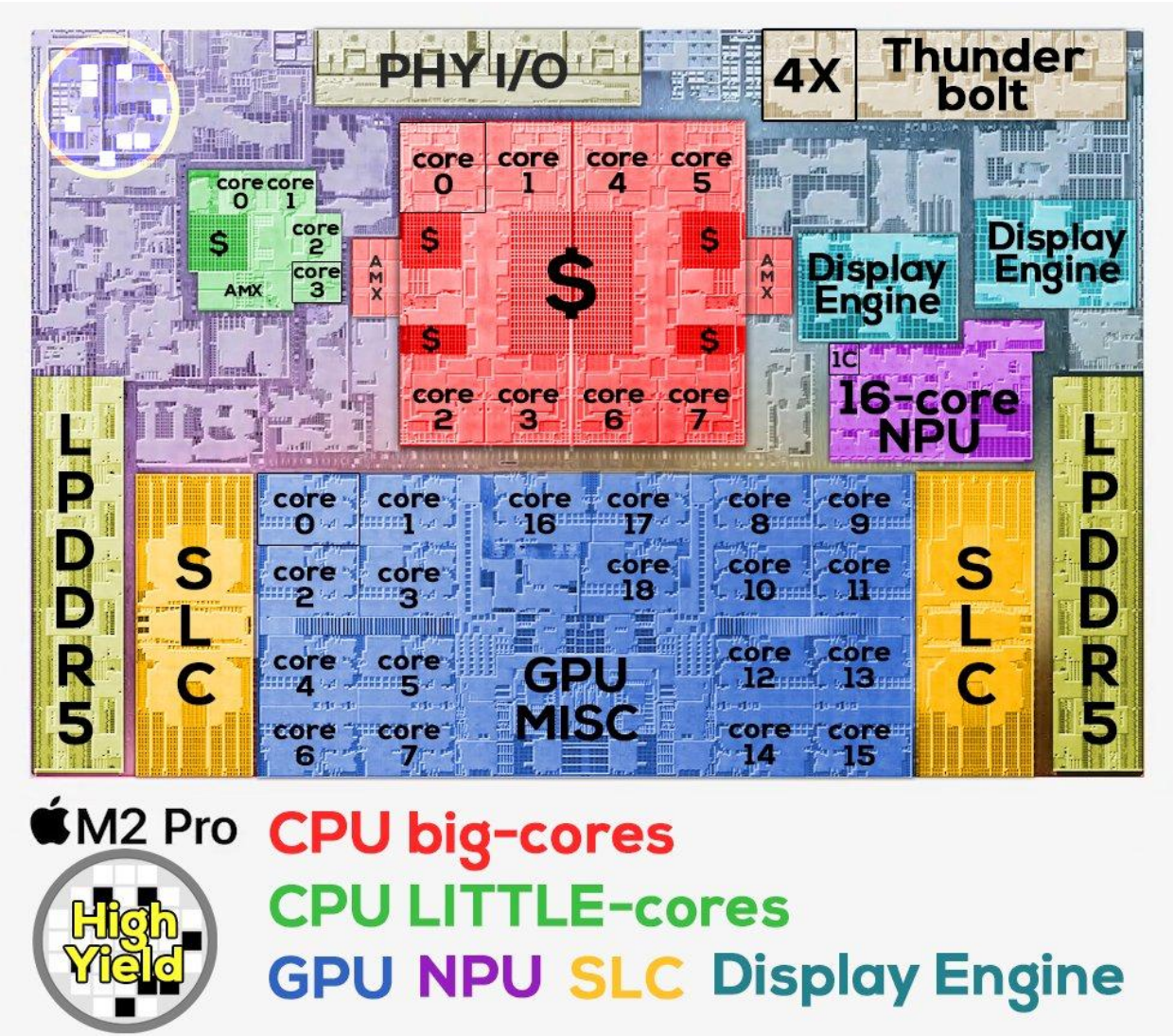


Apple A15



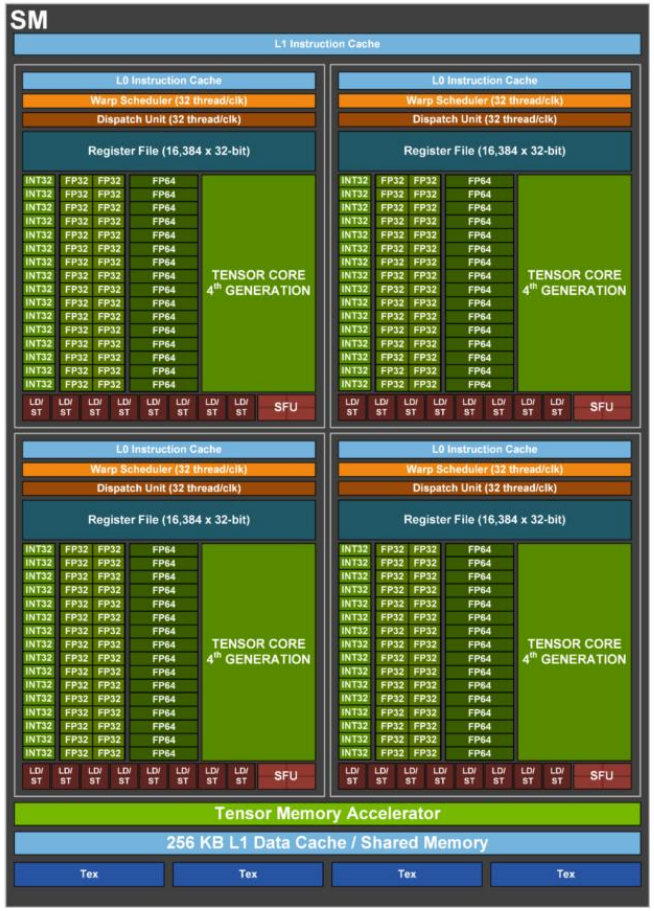
Source: https://wccftech.com/a16-bionic-die-shot-details/

Processor Chip on your Laptop

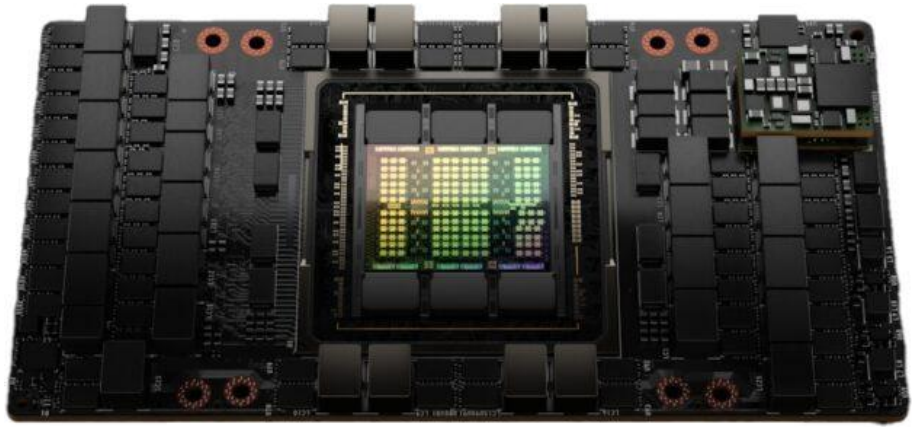


Source: https://twitter.com/highyieldYT/status/1615487519625297924

Chip Multiprocessor on Graphics Card



Streaming Multi-Processor (SM)
128 FP32 CUDA Cores per SM
4 fTensor Cores per SM

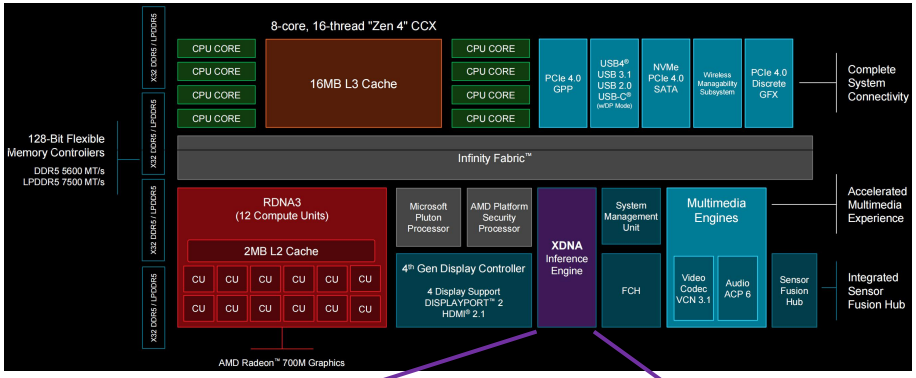


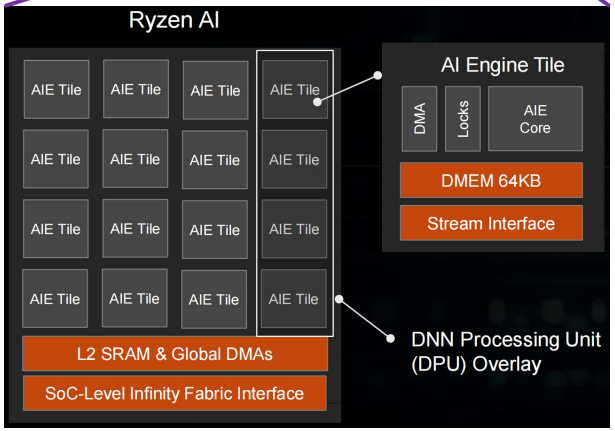
GH100 Full GPU with 144 SMs

Source: https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

AI PC or AI Server

AMD Ryzen 7040





Intel Core Ultra Processor (Meteor Lake)

AlphaGo

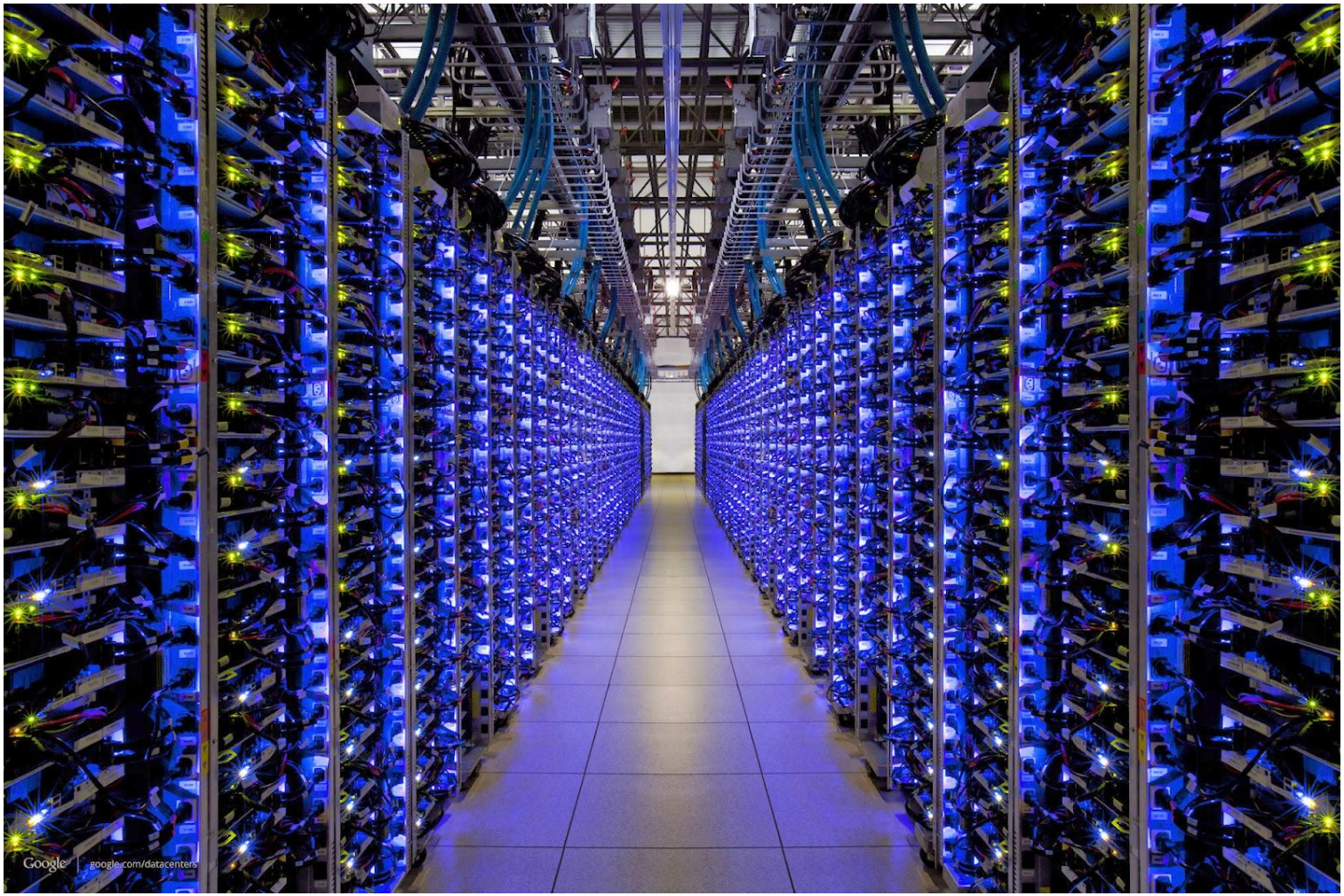
Configuration and performance (before TPU was used)

Configuration	Search threads	No. of CPU	No. of GPU	Elo rating
Single[	40	48	1	2,151
Single	40	48	2	2,738
Single	40	48	4	2,850
Single	40	48	8	2,890
Distributed	12	428	64	2,937
Distributed	24	764	112	3,079
Distributed	40	1,202	176	3,140
Distributed	64	1,920	280	3,168

- **Elo ranking: The difference in the ratings between two players serves as a predictor of the outcome of a match. Two players with equal ratings who play against each other are expected to score an equal number of wins. A player whose rating is 100 points greater than their opponent's is expected to score 64%; if the difference is 200 points, then the expected score for the stronger player is 76%.**
- **Source: wikipedia: https://en.wikipedia.org/wiki/AlphaGo and Nature Jan. 2016**

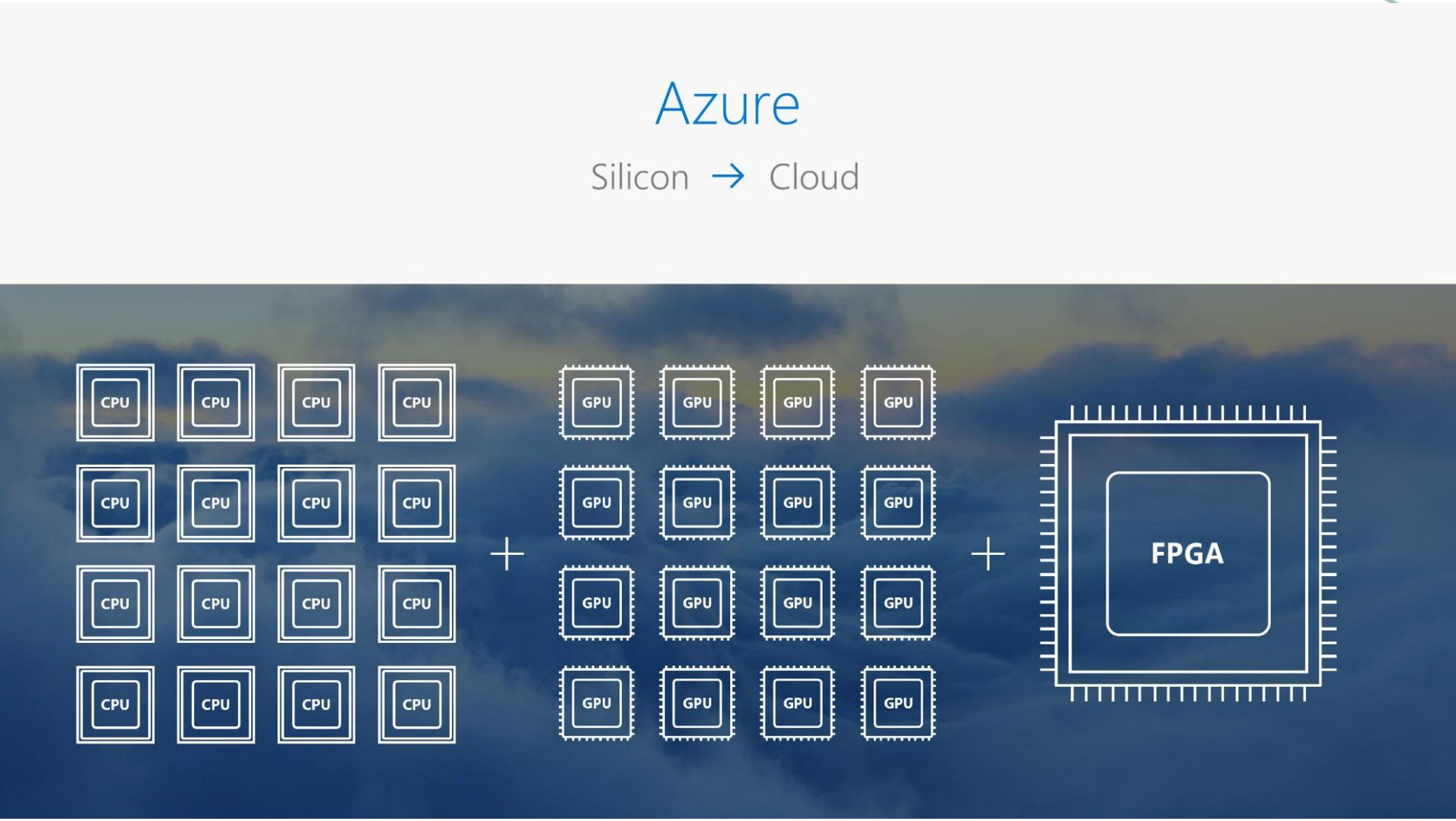
Cloud Computing is Used Everywhere

Large-scale parallel and distributed computing



Source: http://datacentervoice.com/wp-content/uploads/2015/10/data-center.jpg

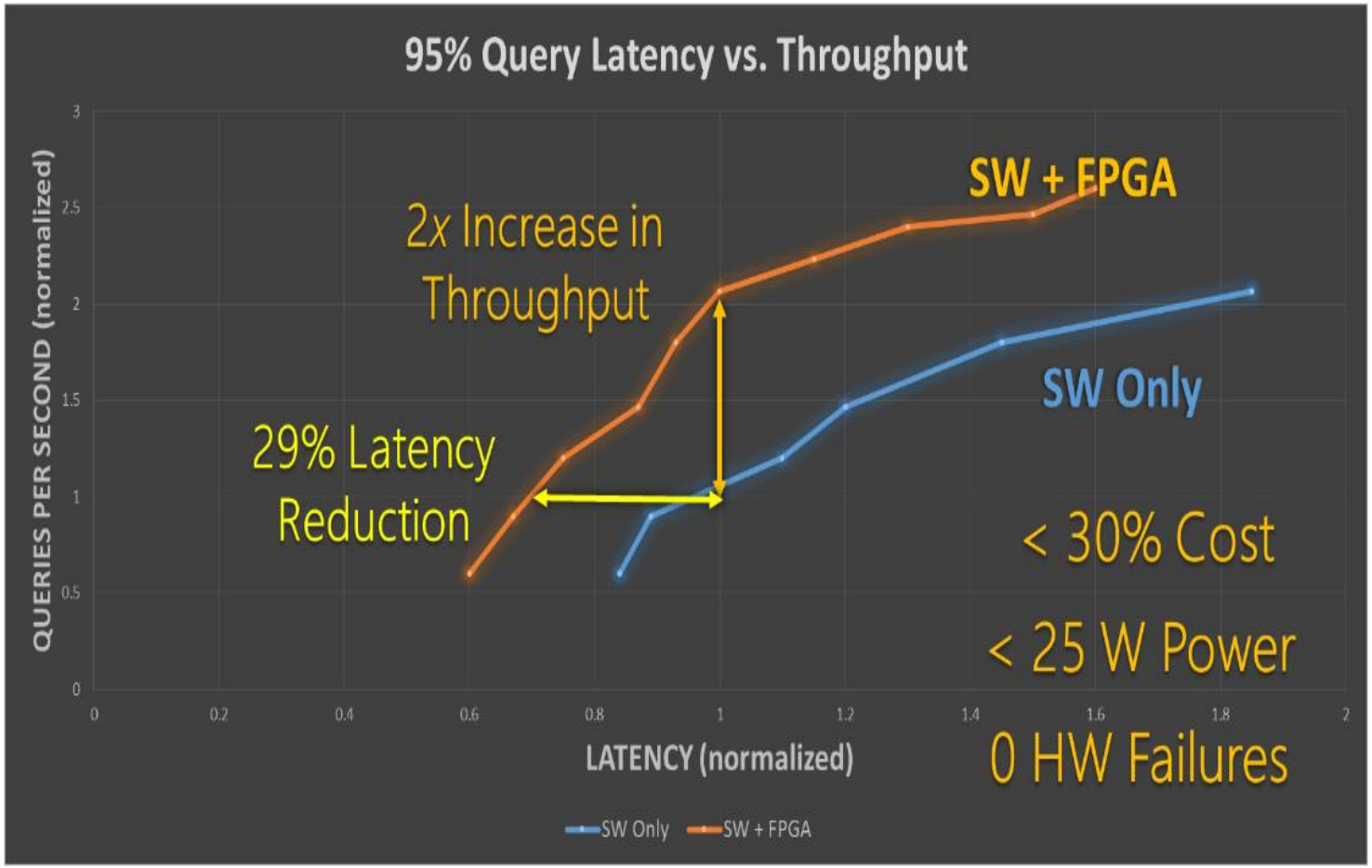
FPGAs in Microsoft Datacenters & Azure



A. Putnam, "*A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services*", ISCA'2014

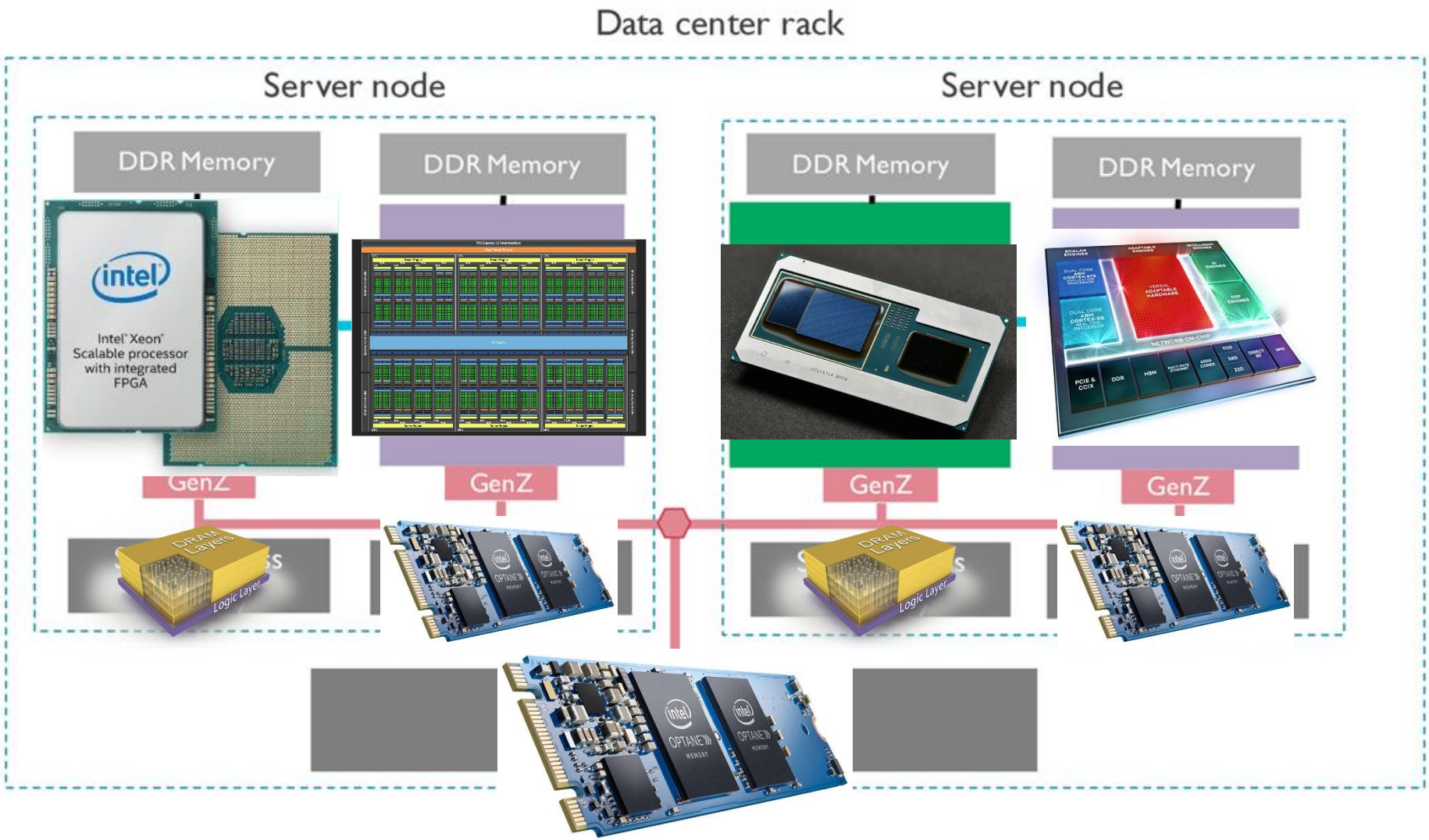
Accelerating Large-Scale Services: Bing Search

1,632 Servers with FPGAs Running Bing Page Ranking Service
(~30,000 lines of C++)



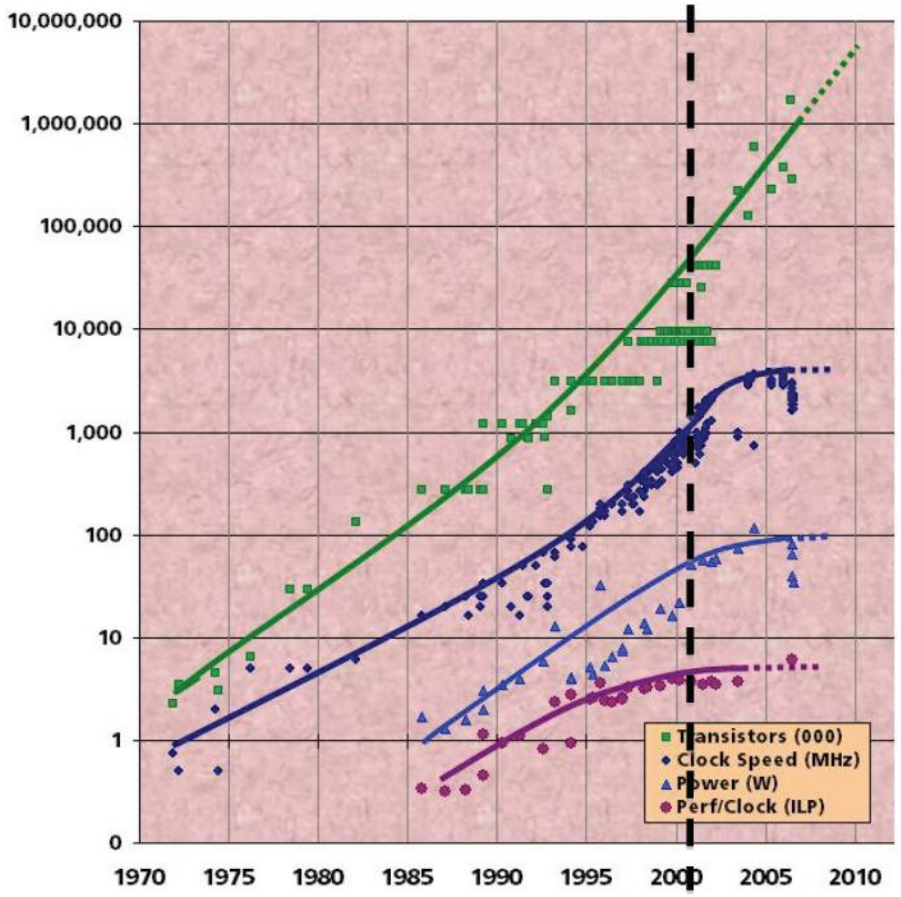
A. Putnam, "*A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services*", ISCA'2014

Computing System Becomes Complex



How did this happen?

- **Moore's Law**
  - Every 18 months, #transistors on chip doubles
- **Until early 2000s**
  - Single processor performance got better all the time
  - The same sequential code would automatically get faster on new hardware
  - Computer marketing
    - … all about the MHz/GHz



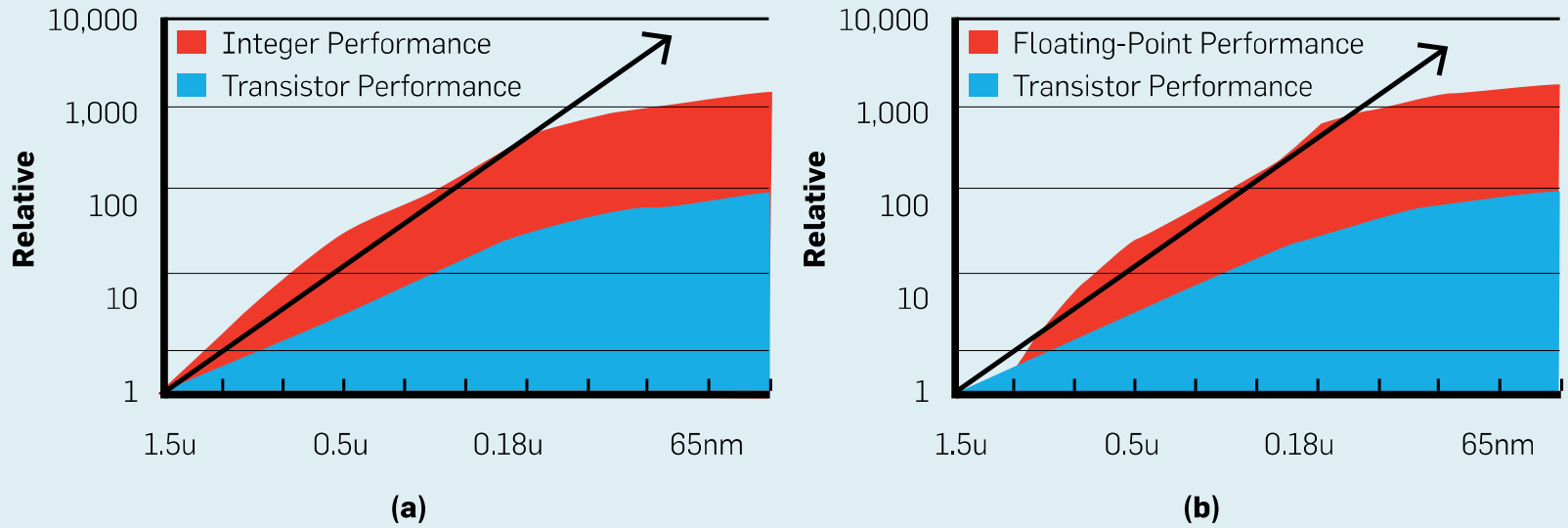
Source: Intel, Microsoft (Sutter) and Stanford (Olukotun, Hammond)
Source: CS133 Spring 2010 at UCLA (Kaplan)

Dennard Scaling of CMOS Transistors

- **Ever generation (~ 2 years): reduce transistor dimensions (& voltage) by 30%**
  - Robert Dennard (IBM), 1971
  - [Dennard, R. et al., IEEE JSSC 1974]
- **Benefits of Dennard scaling (s = 0.7)**
  - Transistor area (s2): Doubles the number of transistors
  - Transistor delay (s): improve by 30%
  - Transistor power ($P = CV^2f =$ s2): improves by 50%
  - Transistor energy (E = PD = s3): improves by 66%
- **Extra transistors can be used for**
  - More efficient on-chip memory organizations (L1, L2, or even L3 cache)
  - Deeper pipelines
  - Better instruction scheduling support to exploit (Instruction-Level Parallelism, or ILP), e.g.
    - Hardware finds independent instructions in a sequential program that can execute simultaneously
    - Hardware predicts which branches will be taken
      - Executes instructions likely to execute before branch is actually resolved
  - Nice feature: improvements are transparent to programmers

Performance Improvement of Intel Processors in 1980s and 1990s

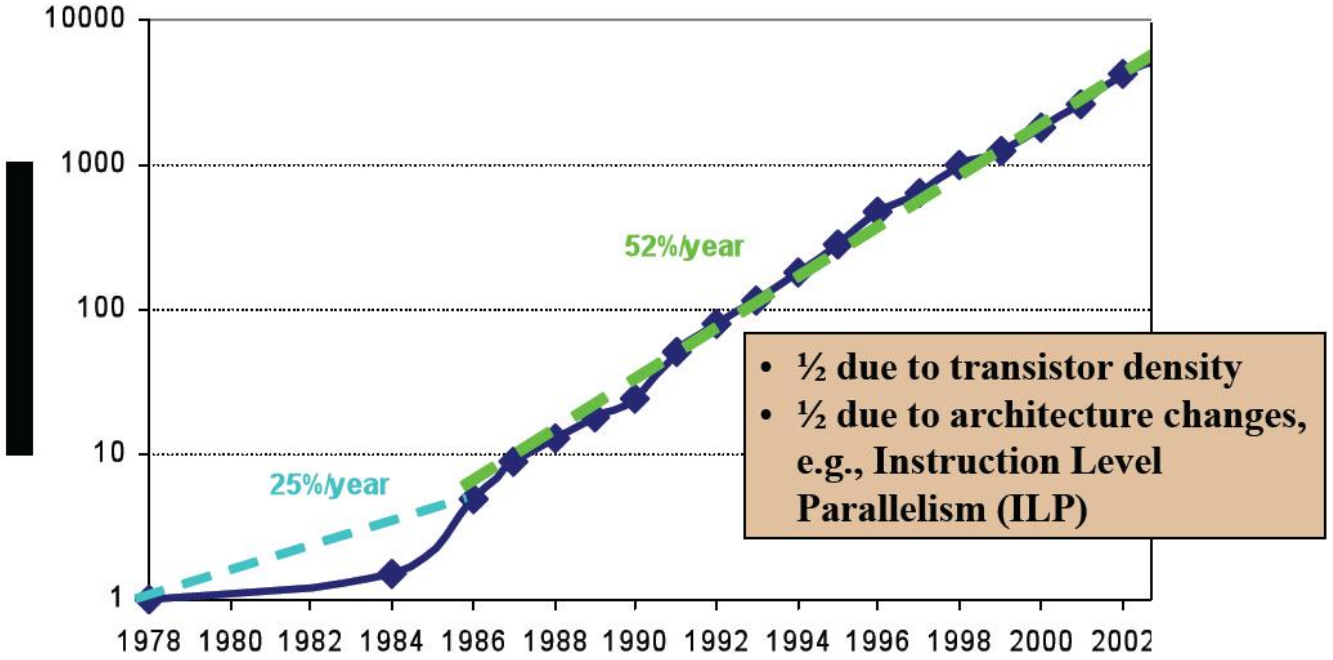
Figure 6. Performance increase separated into transistor speed and microarchitecture performance.



Source: Borkar & Chien: "The Future of Microprocessors", CACM, May 2011

Performance Scaling

Application performance was increasing by 52% per year as measured by the widely used SpecInt benchmark suite



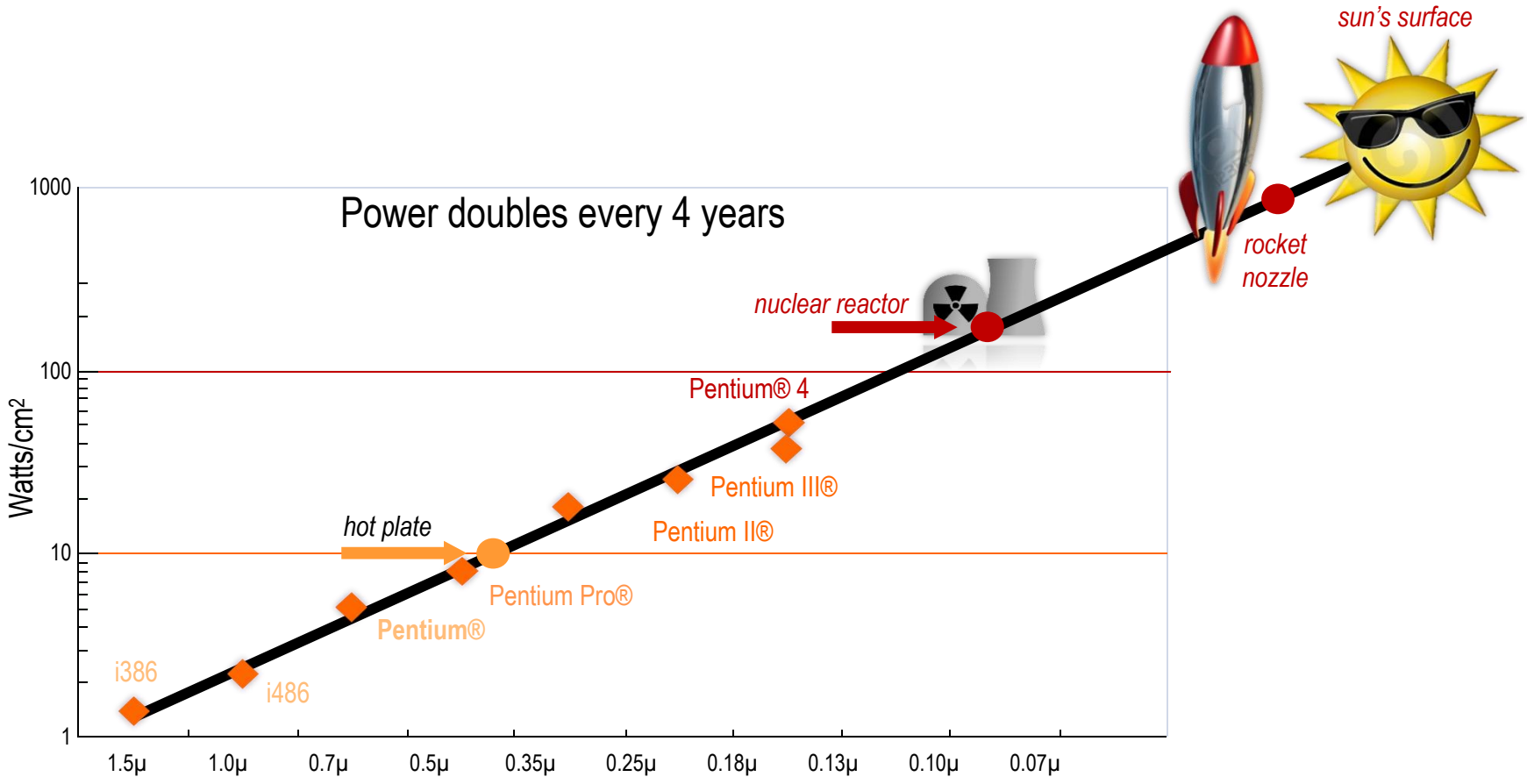
- VAX : 25%/year 1978 to 1986
- RISC + x86: 52%/year 1986 to 2002

Source: Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 4th edition, 2006
Source: CS133 Spring 2010 at UCLA (Kaplan)

The End of Dennard Scaling

- **All good things must come to an end**
- **Reason?**
- **Leakage power!**
  - In Dennard scaling, one has to scale both the supply voltage (Vdd) and threshold voltage (Vt)
  - When Vt is too small, transistor leakages
  - At some point, we have to stop Vt scaling, and in turn stop (or reduce) Vdd scaling
- **Impact – if we keep frequency scaling, power goes up exponentially**

Challenge with Processor Design – Power Barrier



Based on Fred Pollack (Intel) and Michael Taylor (UCSD)

1990s: How to make a faster processor

- **Increasing the clock speed (frequency scaling)**
  - Deeper pipelines… more/shorter stages
  - BUT… eventually chips get too hot
- **Speculative Superscalar (SS)**
  - Multiple instructions can execute at a time

    (*I*nstruction-*L*evel *P*arallelism, or ILP)
    - Hardware finds independent instructions in a sequential program that can execute simultaneously
    - Hardware predicts which branches will be taken
      - Executes instructions likely to execute before branch is actually resolved

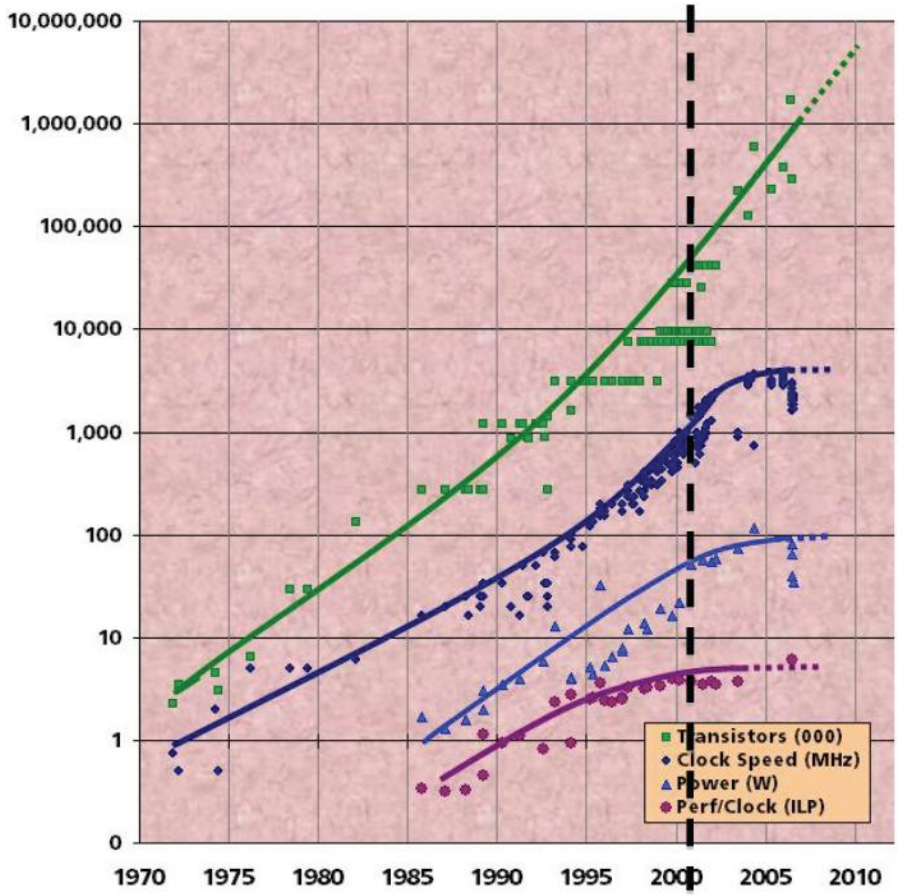
    BUT… eventually diminishing returns

Nice feature: programmers did not need to know about this

Source: CS133 Spring 2010 at UCLA (Kaplan)

2000s: How to make a faster processor

- Chip density grows by 2X every 18 months
  - Clock speed does not
- Diminishing returns seen by speculative superscalar
  - Only so much ILP to exploit
- Use additional transistors to create more/simpler processors on chip

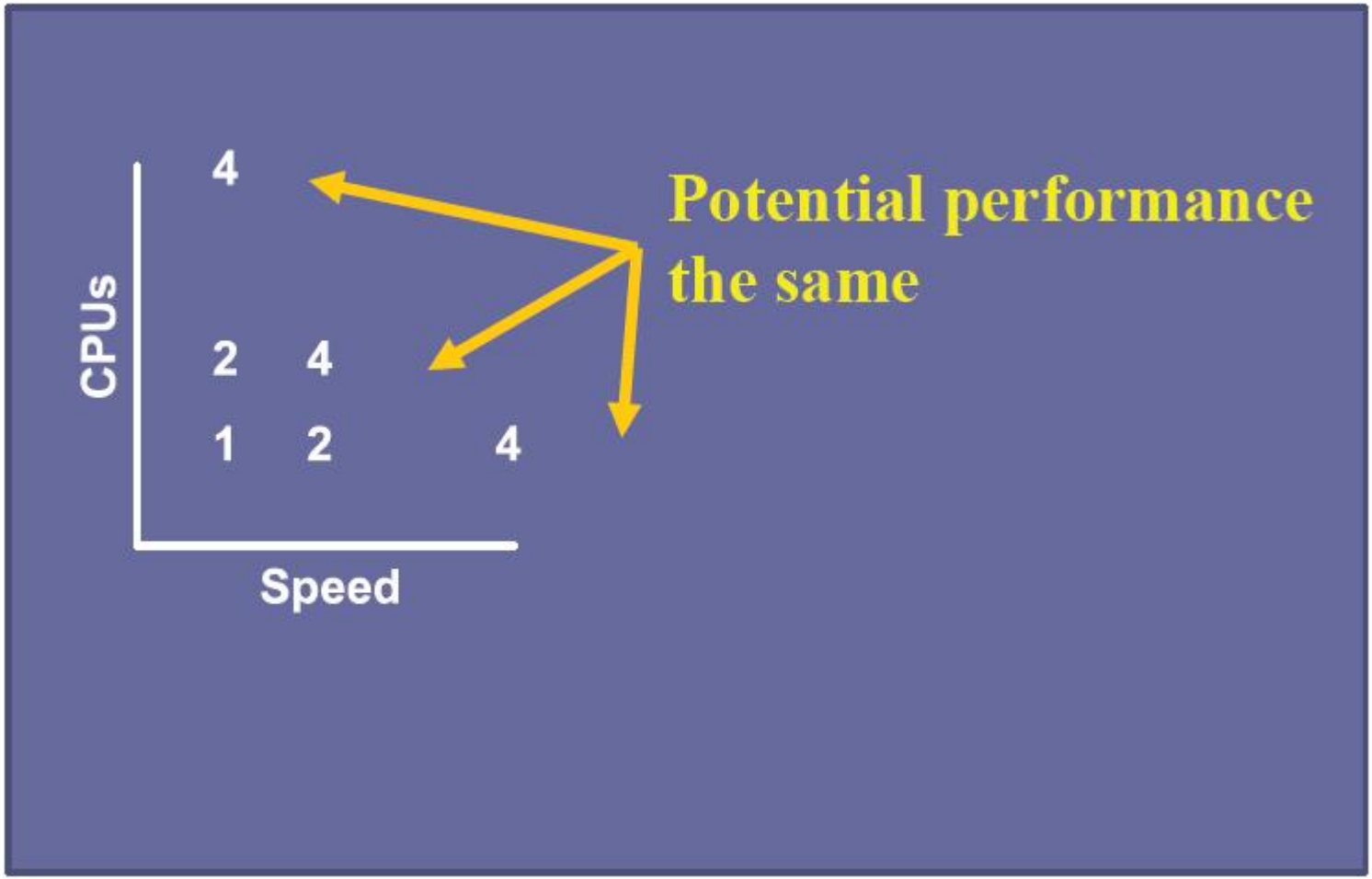


Source: Intel, Microsoft (Sutter) and Stanford (Olukotun, Hammond)
Source: CS133 Spring 2010 at UCLA (Kaplan)

Why Parallelism Help?

- **A very rough explanation:**
- **Dynamic power $P = CV^2f$**
  - If we keep V and f constant, C scales down by $s \sim s^2$
  - Power of each transistor or each core (if we don't add complexity) scales down by $s \sim s^2$
- **To keep a constant power budget, we can increase the number of cores by $1/(s \sim s^2)$**
- **This is a much simplified analysis as there are other factors (e.g. leakage power)**

How can Simpler Processors Help?



Source: Intel
Source: CS133 Spring 2010 at UCLA (Kaplan)

How does a "Human-Formation Computer" Compute?

- Is 9,918,302,881 a prime number?

How does a "Human-Formation Computer" Compute?

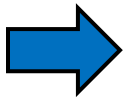
- Is 9,918,302,881 a prime number?
- Square root of 9,918,302,881 is roughly 99,590
- Divide 99,589 (< 100,000) soldiers into
  - 10 divisions
  - Each division into 10 brigades
  - Each brigade into 10 regiments
  - Each regiment into 10 battalions
  - Each battalions into 10 companies
- Each soldier has a unique ID, from 2 to 99,590
  - each checks whether 9,918,302,881 is divisible by their ID
  - and reports the result to their direct commander

Moore's Law Reinterpreted

- Number of cores per chip can double every two years
- Clock speed will not increase (possibly decrease)
- Need to deal with systems with many of concurrent threads
- Need to deal with inter-chip parallelism as well as intra-chip parallelism
- Need to deal with heterogeneity and specialization (not all the cores are the same)
  - More later

Why Writing (Fast) Parallel Programs is Hard…

- **Finding enough parallelism (Amdahl's Law)**
- **Parallelization require great care …**
  - Granularity
  - Locality
  - Load balance
  - Coordination and synchronization
- **Performance modeling**



All of these things makes parallel programming harder than sequential programming.

Source: CS133 Spring 2010 at UCLA (Kaplan)

Finding Enough Parallelism

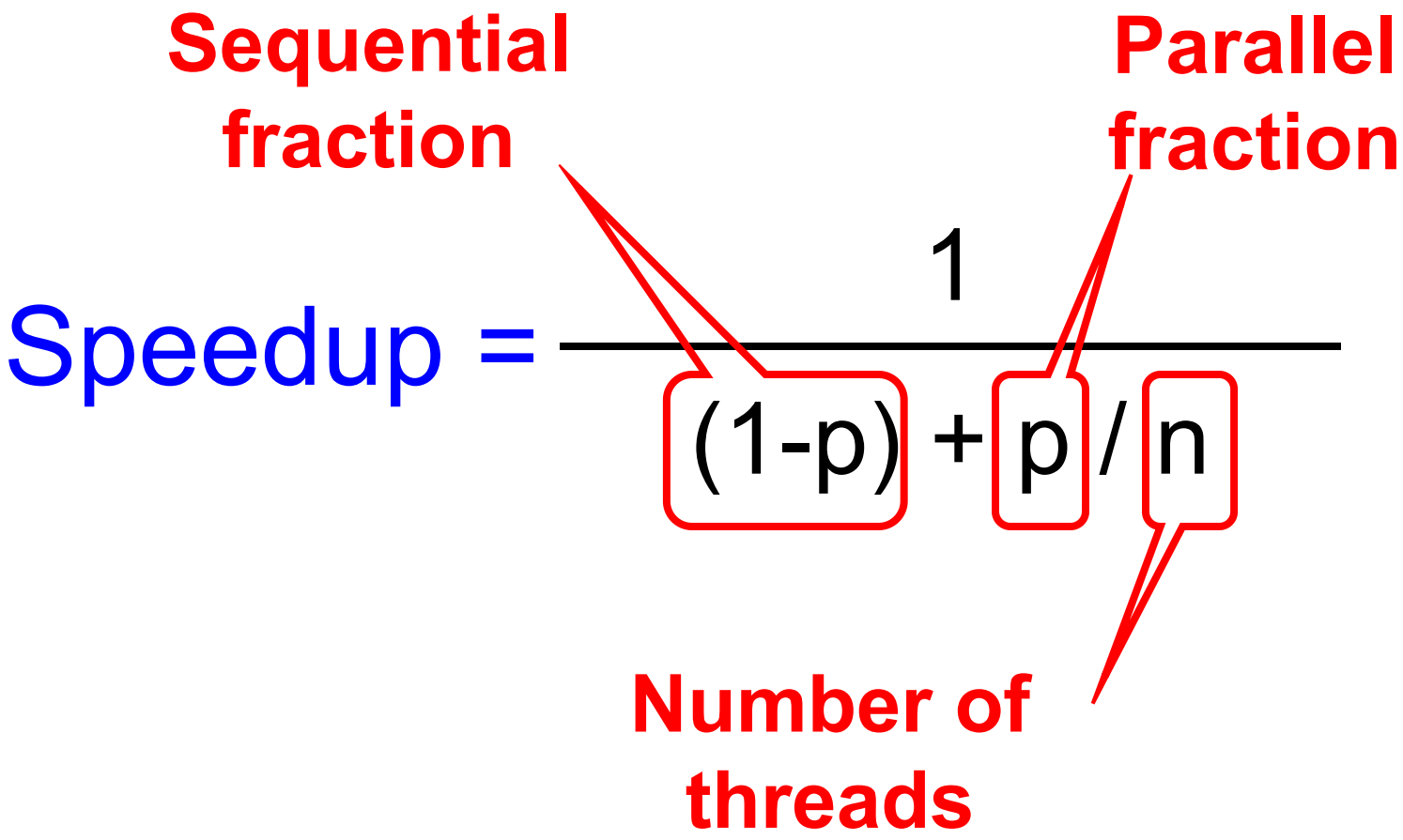
- **We want as much of the code as possible to execute concurrently (in parallel)**
- **Amdahl's law**
  - Suppose only part of an application can be parallelized
  - A larger sequential part implies reduced performance
  - This relation is not linear …
  - Even if the parallel part speeds up perfectly… performance is limited by the sequential part

Amdahl's Law

$$\text{Speedup} = \frac{1\text{-thread execution time}}{n\text{-thread execution time}}$$

Source: Herlihy & Shavit, Art of Multiprocessor Programming

Amdahl's Law



Source: Herlihy & Shavit, Art of Multiprocessor Programming

Amdahl's Law: Example

- Ten processors
- 60% concurrent, 40% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = 2.17 = \frac{1}{1 - 0.6 + \frac{0.6}{10}}$$

Source: Herlihy & Shavit, Art of Multiprocessor Programming

Amdahl's Law: Example

- Ten processors
- 80% concurrent, 20% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = 3.57 = \frac{1}{1 - 0.8 + \frac{0.8}{10}}$$

Source: Herlihy & Shavit, Art of Multiprocessor Programming

Amdahl's Law: Example

- Ten processors
- 90% concurrent, 10% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = 5.26 = \frac{1}{1 - 0.9 + \frac{0.9}{10}}$$

Source: Herlihy & Shavit, Art of Multiprocessor Programming

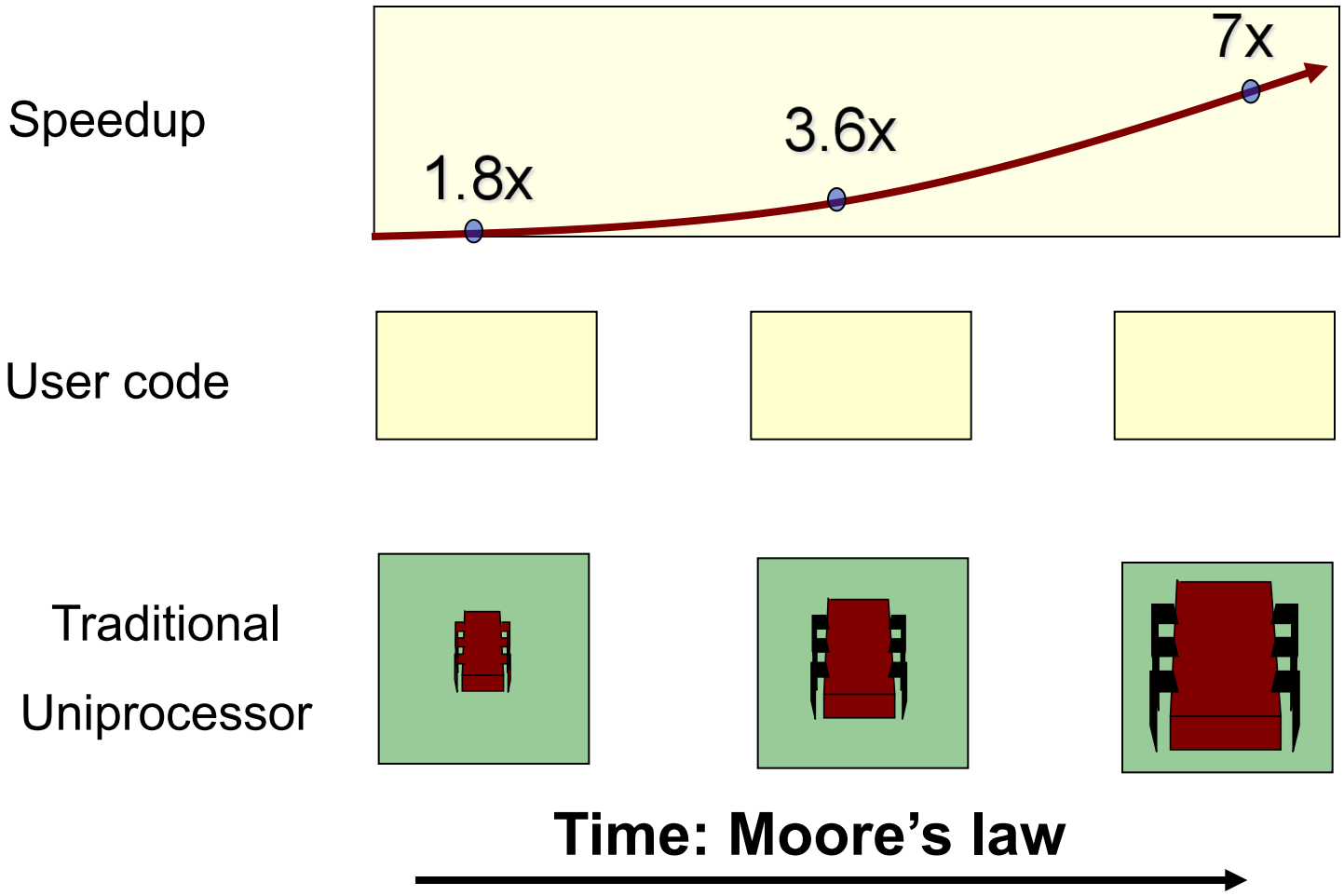
Amdahl's Law: Example

- Ten processors
- 99% concurrent, 01% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = 9.17 = \frac{1}{1 - 0.99 + \frac{0.99}{10}}$$

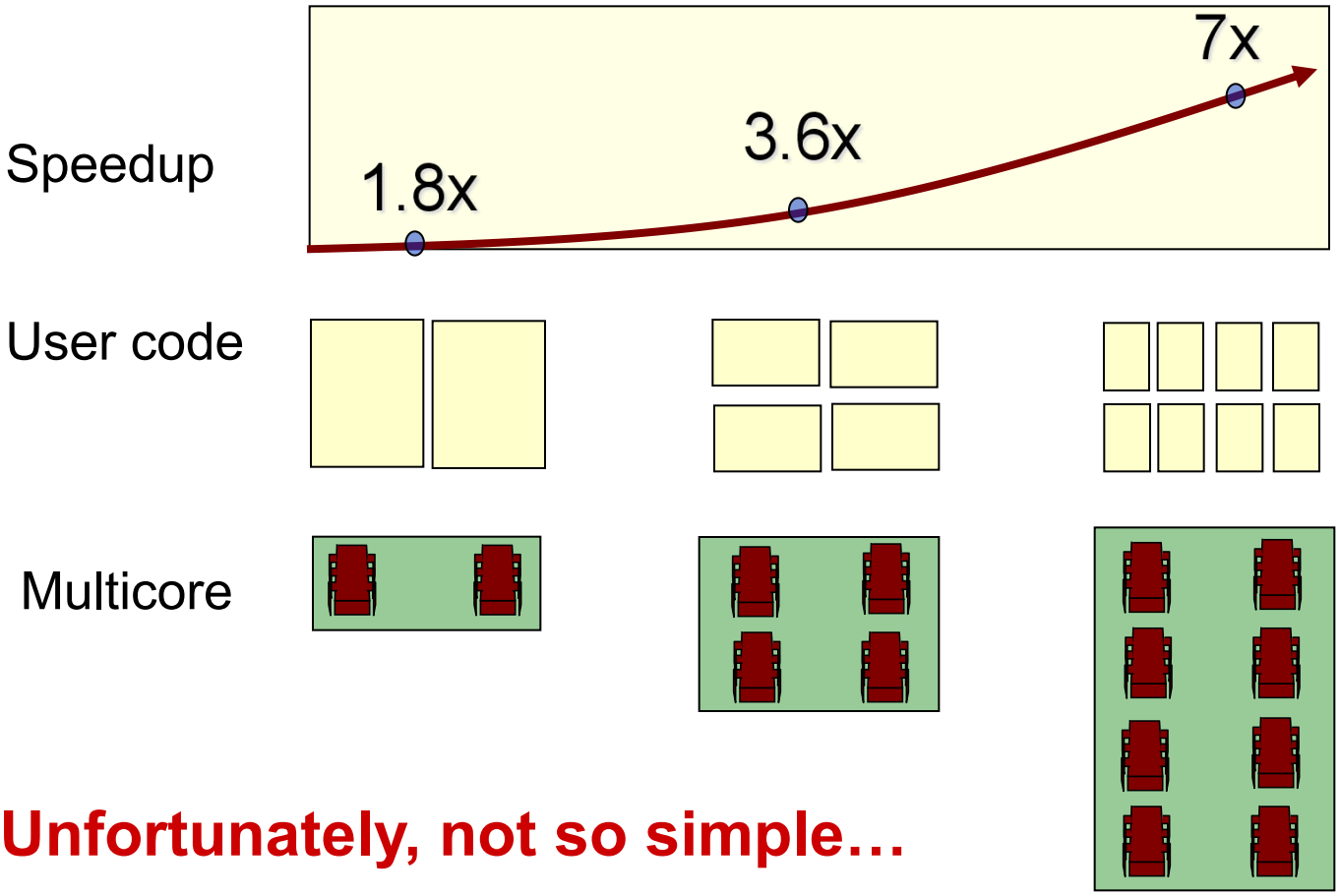
Source: Herlihy & Shavit, Art of Multiprocessor Programming

Traditional Scaling Process



Source: Herlihy & Shavit, Art of Multiprocessor Programming

Ideal Scaling Process



Unfortunately, not so simple…

Source: Herlihy & Shavit, Art of Multiprocessor Programming

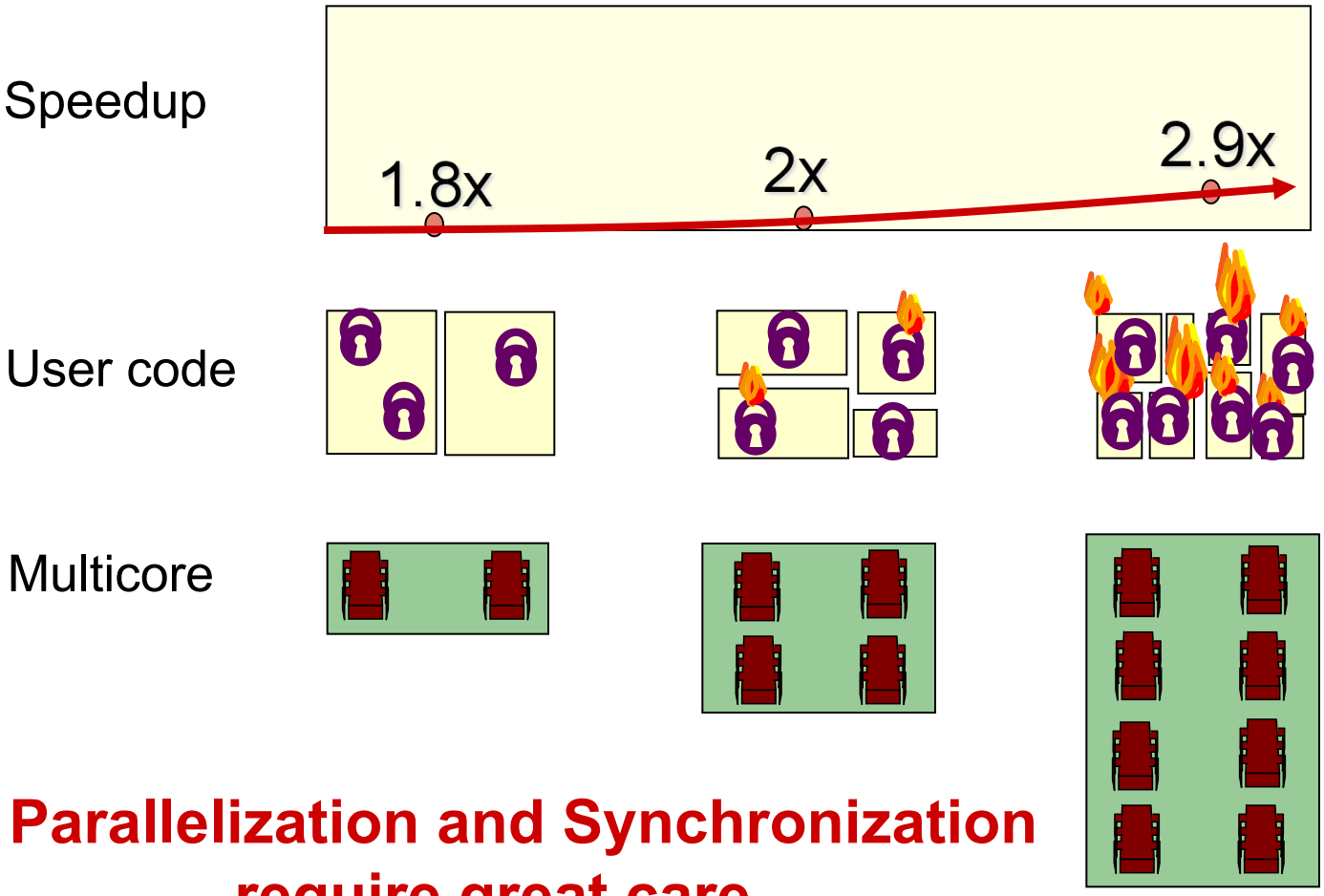
Amdahl's Law (in practice)

Amdahl's Law (in practice)

data dependencies
+
I/O bottlenecks

Source: Herlihy & Shavit, Art of Multiprocessor Programming

Actual Scaling Process

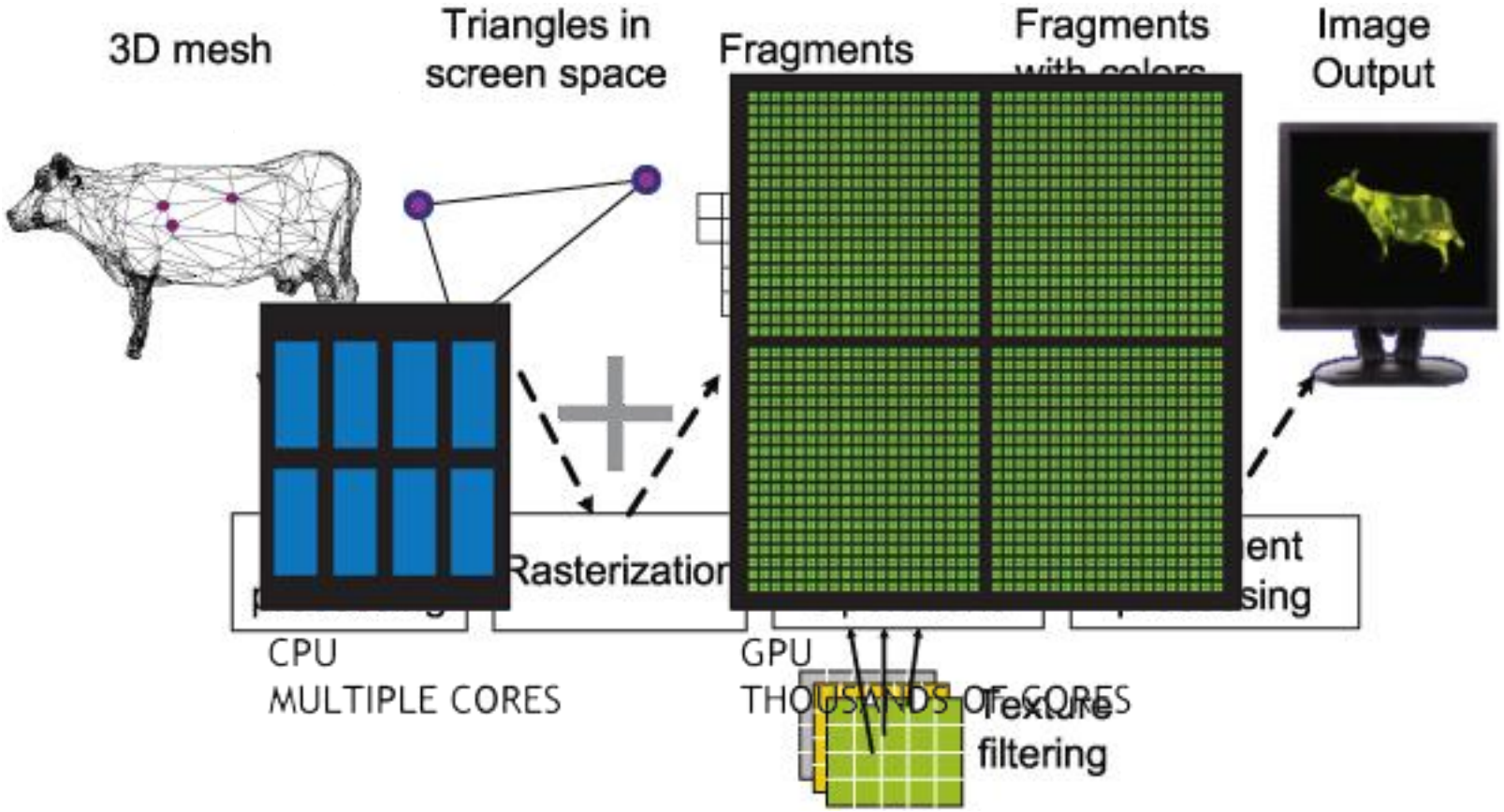


Parallelization and Synchronization require great care…

Source: Herlihy & Shavit, Art of Multiprocessor Programming

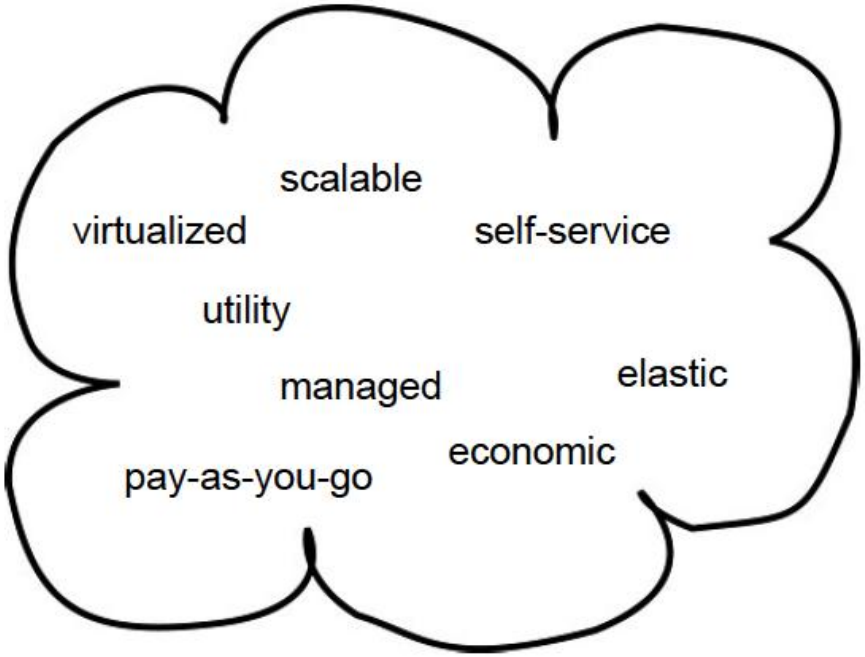
More Parallelism by Scaling Up

- **From multi-core to many cores**
  - And we have GPU programming



More Parallelism by Scaling Out

- **From parallel to distributed**
  - "Big data" too big to fit on one computer
- **And there is cloud computing**



Source: Andy Konwinski, "Cloud Computing using MapReduce, Hadoop, Spark," ParLab Bootcamp 2012

More Parallelism by Scaling Out

- **Internet can be seen as a large parallel/distributed computing environment**
  - The "cloud"
    - A set of computers on the internet available on demand, like a public utility
  - Google's MapReduce
    - A software framework enabling the computing of large data sets on clusters of computers
      - Can **map** a parallel algorithm to worker nodes in the cloud
      - **Reduce** results from worker nodes to a single output/answer

What is Cloud Computing?

- **Cloud**
  - Large internet services running on 10,000s of machines (Amazon, Microsoft, 阿里, 华为, etc.)
- **Cloud computing**
  - Services that let external customers rent cycles and storage
  - IaaS
    - Amazon EC2: VMs at $0.0042~$109.20/hour, billed hourly
    - Amazon S3: storage at $0.023/GB/month
  - SaaS
    - OpenAI: GPT-4-Turbo at $0.01 / 1K (input) and $0.03 / 1K (output) tokens

Core Cloud Concepts

- **Virtualization**
- **Self-service (personal credit card) & pay-as-you-go**
- **Economic incentives**
  - Provider: sell unused resources
  - Customer: no upfront capital costs building data centers, buying servers, etc.

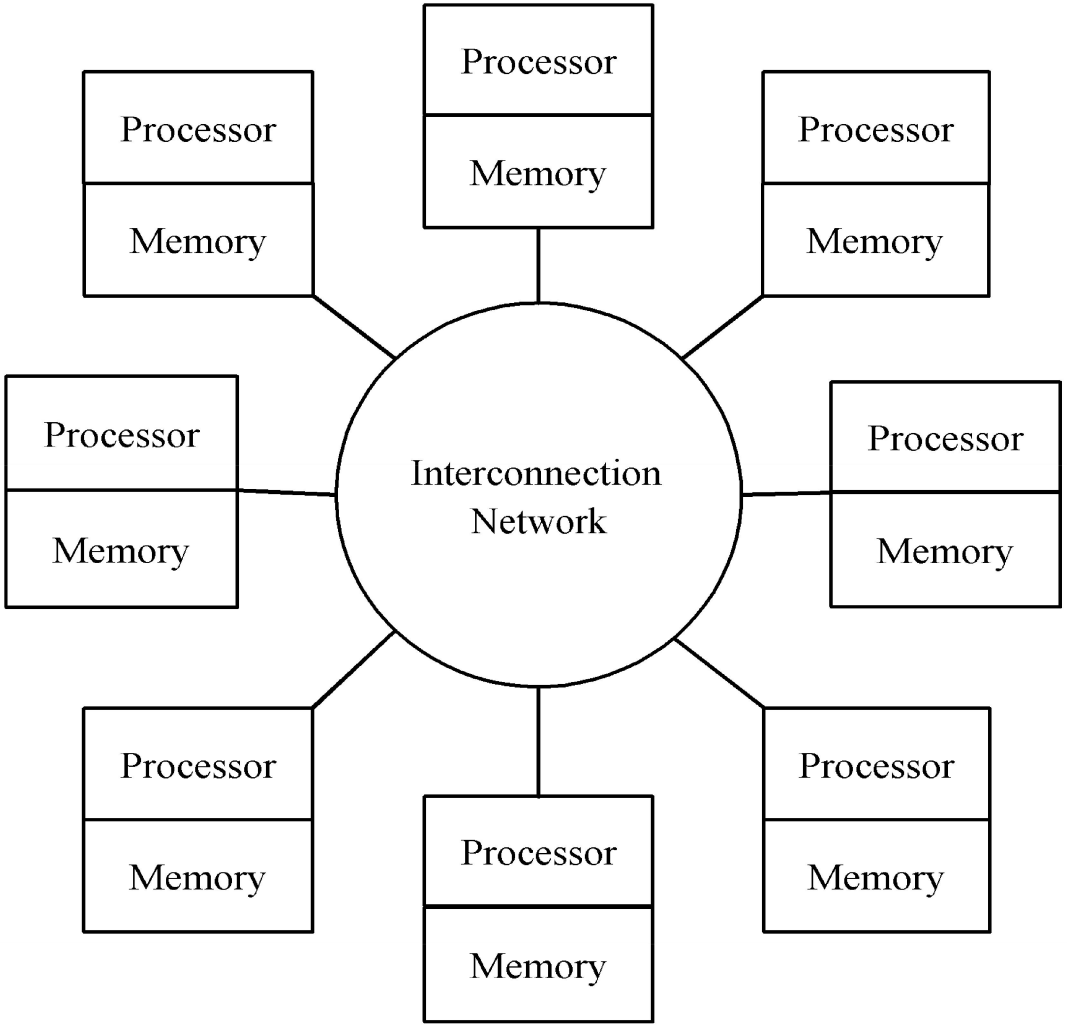
Source: Andy Konwinski, "Cloud Computing using MapReduce, Hadoop, Spark," ParLab Bootcamp 2012

Moving Target

- **Infrastructure as a Service (virtual machines)**
- **Why?**
  - Managing lots of machines is still hard
  - Programming with failures is still hard
- **Solution: higher-level frameworks, abstractions**

Source: Andy Konwinski, "Cloud Computing using MapReduce, Hadoop, Spark," ParLab Bootcamp 2012

Message Passing in the Cloud?



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Cloud Environment Challenges

- **Cheap nodes fail, especially when you have many**
  - **Example**
    - **Mean time between failures (MTBF) for 1 node = 3 years**
    - **MTBF for a raw 1000-node system ≈ 1 day**
  - **Solution: restrict programming model so you can efficiently "built-in" fault-tolerance (art)**
- **Commodity network = low bandwidth**
  - **Solution: push computation to the data**

Source: Andy Konwinski, "Cloud Computing using MapReduce, Hadoop, Spark," ParLab Bootcamp 2012

HPC/MPI in the Cloud

- **EC2 provides virtual machines, so you can run MPI**
- **Fault-tolerance**
  - Not standard in most MPI distributions
  - Recent restart/checkpointing techniques, but need the checkpoints to be replicated as well
    - https://ftg.lbl.gov/projects/CheckpointRestart
- **Communication?**

Source: Andy Konwinski, "Cloud Computing using MapReduce, Hadoop, Spark," ParLab Bootcamp 2012

What is MapReduce?

- **Data-parallel programming model for clusters of commodity machines**

- **Pioneered by Google**
  - Processes 20 PB of data per day
- **Popularized by Apache Hadoop project**
  - Used by Yahoo!, Facebook, Amazon, …

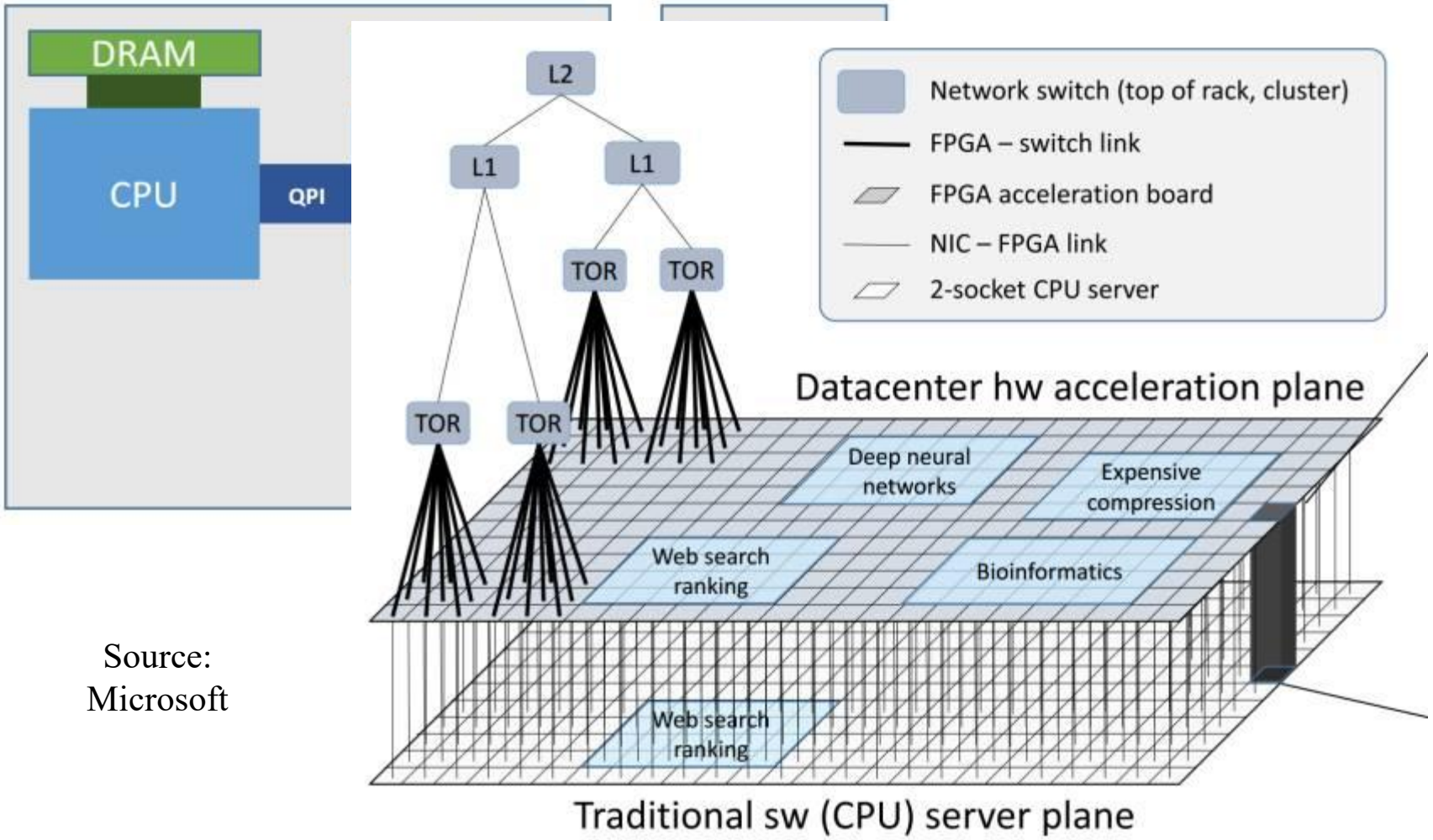
Source: Andy Konwinski, "Cloud Computing using MapReduce, Hadoop, Spark," ParLab Bootcamp 2012

MapReduce Goals

- **Cloud Environment**
  - Commodity nodes (cheap, but unreliable)
  - Commodity network (low bandwidth)
  - Automatic fault-tolerance (fewer admins)
- **Scalability to large data volumes**
  - Scan 100 TB on 1 node @ 50 MB/s = 24 days
  - Scan on 1000-node cluster = 35 minutes

Source: Andy Konwinski, "Cloud Computing using MapReduce, Hadoop, Spark," ParLab Bootcamp 2012

General-Purpose Parallel Computing ⇒ Customized Parallel Computing



Source: Microsoft

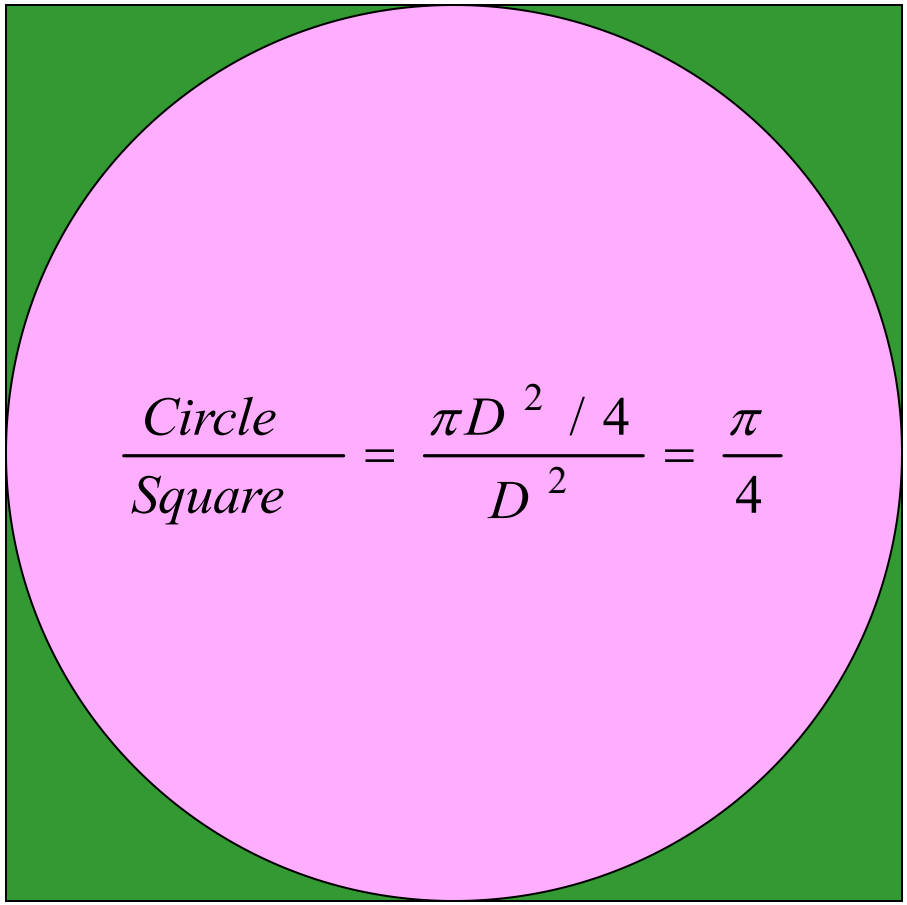
What Makes Parallel and Distributed Programming Easier?

- **Standardized parallel & distributed programming models**
  - OpenMP
    - for multithreaded programming
  - OpenCL/CUDA
    - for massively multithreaded programming
    - OpenCL is also for heterogeneous computing
  - Message Passing Interface (MPI)
    - for distributed programming
  - MapReduce
    - for new generation of distributed programming
- **Why do they help?**
  - Longer life-cycle for programs
  - Code works across platforms
  - Automatic scaling?

How does the Code Look Like?

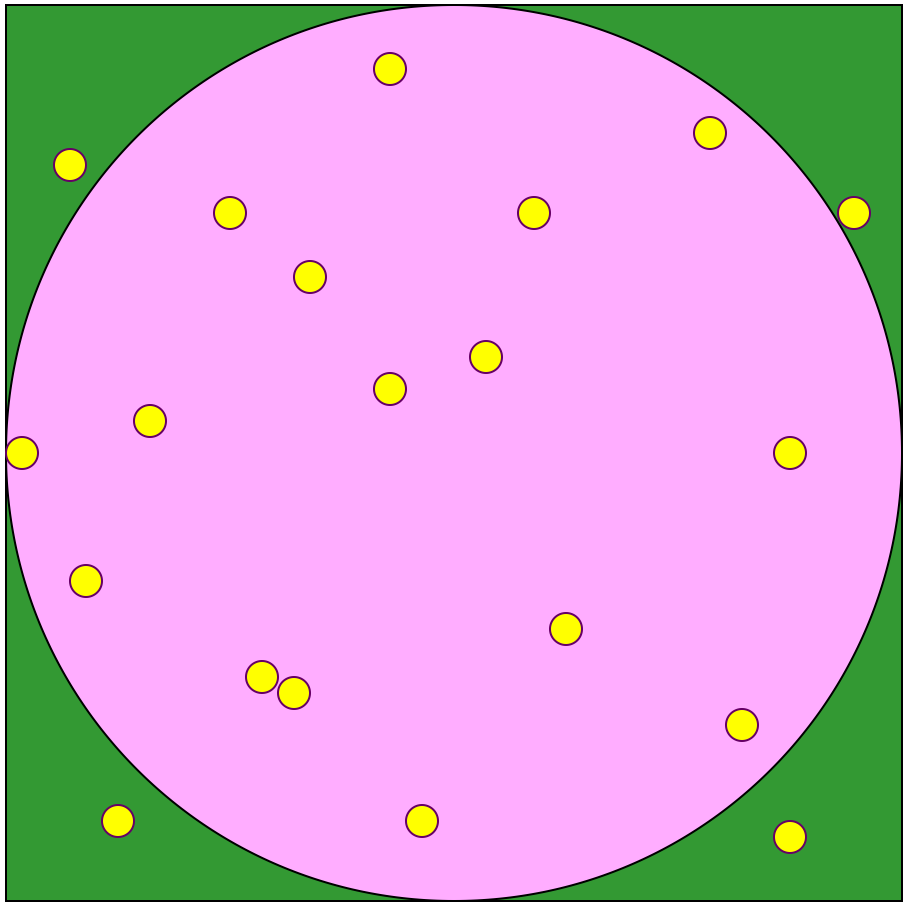
- **Shared memory**
  - pthreads
  - OpenMP
- **Distributed memory**
  - MPI
  - Spark
- **Massively multi-threaded**
  - (shared memory on device; separated host/device memories)
  - OpenCL

Example of Monte Carlo Method



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Example of Monte Carlo Method



$$\frac{16}{20} \approx \frac{\pi}{4} \Rightarrow \pi \approx 3.2$$

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Compute π: POSIX Threads (1/2)

```
// global variables
int thread_count;
int num_samples;
double* local_count;
// multithreaded version
double compute_pi () {
  thread_handles = (pthread_t*) malloc (thread_count*sizeof(pthread_t));
  local_count = (double*) malloc (thread_count*sizeof(double));
  // parallel part
  for (thread = 0; thread < thread_count; thread++)
    pthread_create(&thread_handles[thread], NULL, thread_sum, (void*)thread);
  for (thread = 0; thread < thread_count; thread++)
    pthread_join(thread_handles[thread], NULL);
  // sequential part
  int count = 0;
  for (thread = 0; thread < thread_count; thread++)
    count += local_count [thread];
  double pi = 4.0 * count / num_samples;
  return pi;
}
```

Compute π: POSIX Threads (2/2)

```
void* thread_sum(void* rank) {
  int my_rank = (int) rank;
  int my_count = 0;
  // domain decomposition
  for (int i = my_rank; i < num_sampes; i += thread_count) {
    double x = (double)rand() / RAND_MAX;
    double y = (double)rand() / RAND_MAX;
    my_count += (x*x + y*y < 1);
  }
  local_count [my_rank] = my_count;
  return NULL;
}
```

Compute π: OpenMP Version

```
/*  compile as $> gcc -fopenmp -lm
 *  run as        $> OMP_NUM_THREADS=2 ./a.out
 */
double compute_pi(int num_samples) {
   int i;
   int count = 0;
#pragma omp parallel for reduction(+: count) schedule(static)
   for (i = 0; i < num_samples; ++i) {
      double x = (double)rand() / RAND_MAX;
      double y = (double)rand() / RAND_MAX;
      count += (x*x + y*y < 1);
   }
   double pi = 4.0 * count / num_samples;
   return pi;
}
```

Compute π: OpenCL Version

```
/* to be executed on deivce */
__kernel void pi(const int  niters,  const float  step_size,
                 __local  float*  local_count,  __global float*  partial_count) {
   int num_wrk_items  = get_local_size(0);
   int local_id       = get_local_id(0);
   int group_id       = get_group_id(0);
   float x, y = 0.0f;
   int count = 0;
   int i,istart,iend;
   istart = (group_id * num_wrk_items + local_id) * niters;
   iend   = istart+niters;
   for(i= istart; i<iend; i++) {
      x = …; // a random number in [0,1];
      y = …; // a random number in [0,1];
      count += (x*x + y*y < 1.0f);
   }
   local_count[local_id] = count;
   barrier(CLK_LOCAL_MEM_FENCE);
   reduce(local_count, partial_count);
}

/* to be executed on host */
pi_res = 0.0f;
for (unsigned int i = 0; i < nwork_groups; i++) { pi_res += h_pcnt[i]; }
pi_res *= step_size;
```

Compute π: MPI Version (1/2)

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
int main( int argc, char *argv[] )  {
  int n, my_rank, numprocs, i;
  double pi, h, x, y;
  int my_count, count;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
  while (1) {
    if (my_rank == 0) {
      printf("Enter the number of samples: (0 quits) ");
      scanf("%d",&n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Compute π: MPI Version (2/2)

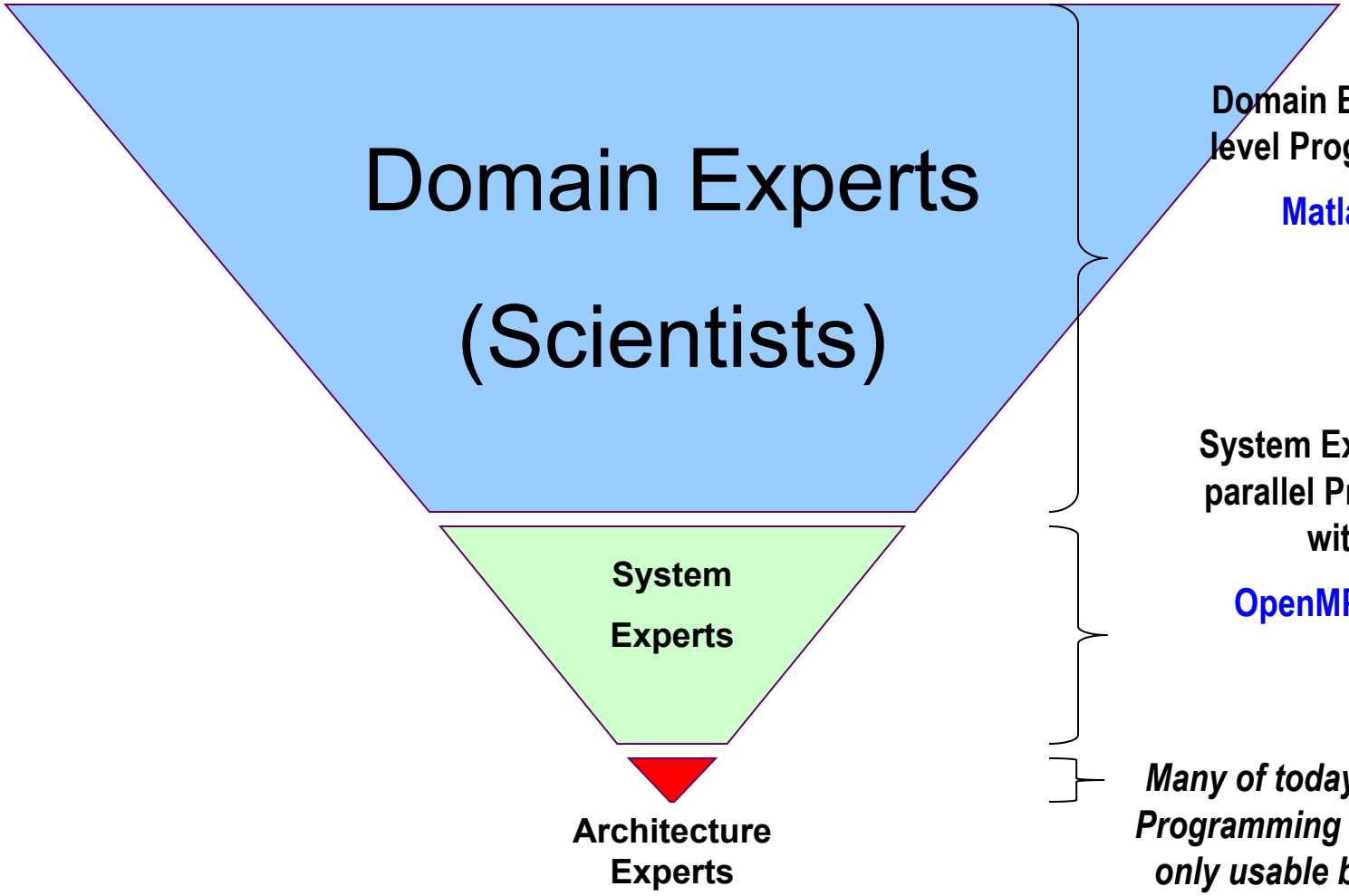
```
  if (n == 0)  break;
  else {
    my_count = 0;
    for (i = my_rank; i <; i += numprocs) {
      x = …; // a random number in [0,1]
      y = …; // a random number in [0,1]
      my_count += (x*x + y*y < 1);
    }
    MPI_Reduce(&my_count, &count, 1, MPI_INT, MPI_SUM, 0,
                        MPI_COMM_WORLD);
    if (my_rank == 0) printf("pi is approximately %.16f\n", 4.0*count/n));
   }
  }
  MPI_Finalize();
  return 0;
}
```

Compute π: Spark Version

```
List<Integer> l = new ArrayList<Integer>(NUM_SAMPLES);
for (int i = 0; i < NUM_SAMPLES; i++) {
  l.add(i);
}

long count = sc.parallelize(l).filter(new Function<Integer, Boolean>() {
  public Boolean call(Integer i) {
    double x = Math.random();
    double y = Math.random();
    return x*x + y*y < 1;
  }
}).count();
System.out.println("Pi is roughly " + 4.0 * count / NUM_SAMPLES);
```

Inverted Pyramid of Skills



Domain Experts Use the Infrastructure and Innovate Their Field

System Experts Design and Renovate the Architecture and Infrastructure

Summary

- **Modern computing systems "scale" in both parallelism (amount of computing units) and heterogeneity (types of computing units)**
  - Thanks to the end of Dennard scaling
- **How to program this monster?**
  - Hopefully there will be fully-automatic compilers and runtimes
  - But we are not there yet! (So we need domain experts who can hack system.)