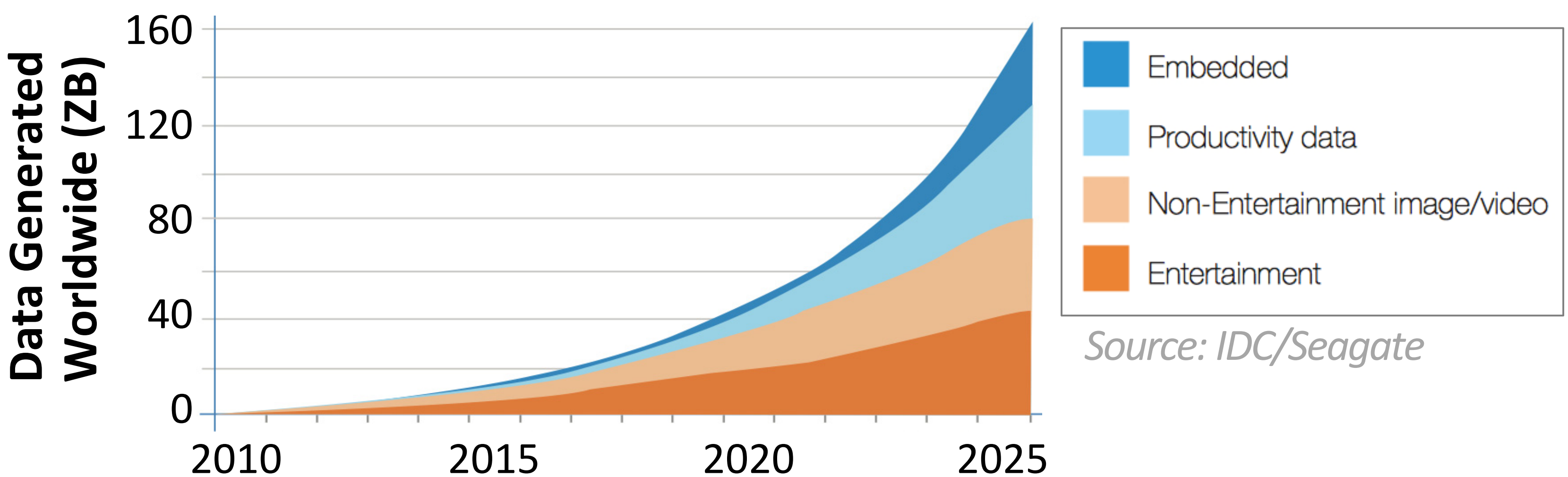
Future of Computing I: Diverging Computer System Design

15-213/18-213/15-513/18-613: Introduction to Computer Systems
28th Lecture, April 28, 2020

The Proliferation of Computing

- **Before 2000**
  - Personal computers: desktops (more predominant) and laptops
  - Servers: delivered mostly static web pages (limited PHP/Perl/ASP)
- **Today**
  - Smartphones/tablets greatly outnumber desktops and laptops
  - Servers and the cloud provide and store changing content
  - Big data revolution: amount generated is growing exponentially



One-Size-Fits-All Is Going Away

- **Computers have very different needs and target metrics**
  - Used to be just performance
  - Power/energy matter greatly now
- **Specialization can help cut down energy**
  - Mobile devices use systems-on-chip (SoCs)
  - Servers use highly-multithreaded CPUs
- **x86 is no longer the dominant ISA**
  - Recall: RISC vs. CISC from the first machine programming lecture
  - ARM ISAs (which are RISC) are now used in the vast majority of mobile devices
  - x86 (which is CISC) still reigns supreme in servers, desktops, laptops

Computer System Design is Diverging

- **Several types of systems are becoming popular**
  - Graphics processing units (GPUs)
  - Mobile systems-on-chip (SoCs)
  - Data centers and cloud computing
  - Internet of things (IoT)/edge computing
- **A few promising designs may emerge in the future**
  - Processing-in-memory (PIM)
  - Neuromorphic computing

Computer System Design is Diverging

- **Several types of systems are becoming popular**
  - Graphics processing units (GPUs)
  - Mobile systems-on-chip (SoCs)
  - Data centers and cloud computing
  - Internet of things (IoT)/edge computing
- **A few promising designs may emerge in the future**
  - Processing-in-memory (PIM)
  - Neuromorphic computing

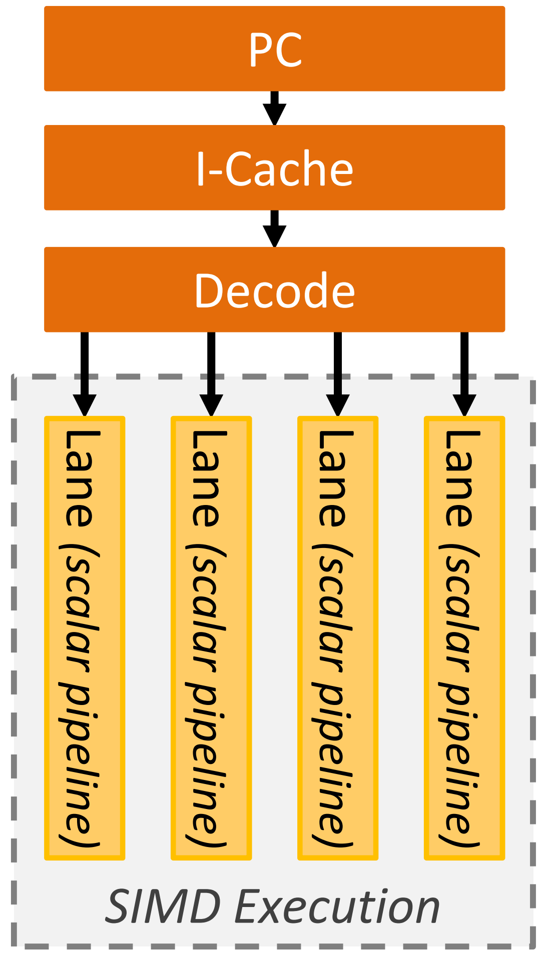
How Do We Run Thousands of Threads?

- **CPUs become increasingly inefficient**
  - Small number of cores: need thousands of context switches
  - Large number of cores
    - Lots of hardware and power need to be used
    - How do we handle consistency and snooping?

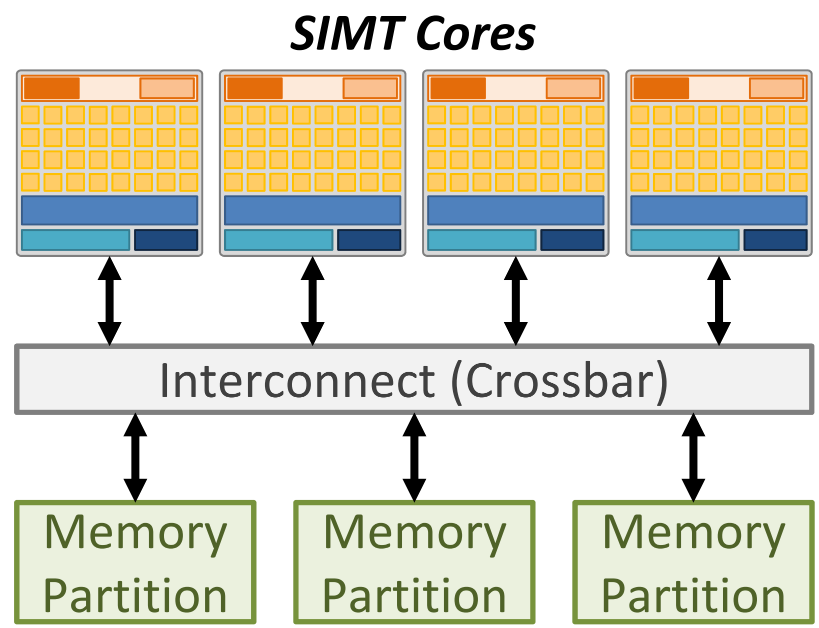
- **One option: SIMT (single instruction, multiple thread)**
  - Run multiple copies of a single thread in **lockstep** (they all execute the *same* instruction at the *same* time) on different pieces of data
  - Programs use the **multithreaded** programming model (a.k.a. **single program, multiple data** or **SPMD**), but there are **key differences** from the multithreading you are used to

How to Implement SIMT

- **Write a program using threads**
  - Each thread executes the same code but operates on a different piece of data
  - Each thread has its own context (i.e., can be treated/restarted/executed independently)
- **Group threads together dynamically (i.e., in hardware)**
  - A group is known as a warp or a wavefront
  - Essentially a vector formed by hardware
- **SIMT processors can share common control flow logic for a warp across a number of scalar execution lanes (one lane per thread)**



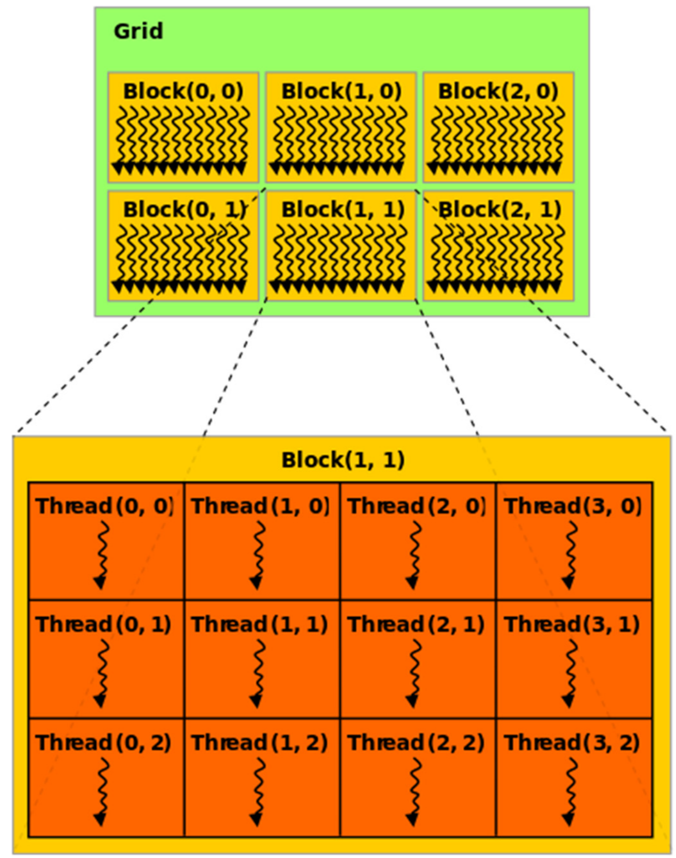
Graphics Processing Units (GPUs)



- **A cluster of SIMT cores (known as SMs or shaders) that share a memory hierarchy**
- **Each SIMT core operates on one or more warps**
- **Original purpose was for graphics workloads**
  - Designed to operate in parallel on thousands of pixels/vertices/fragments
  - Used to have special cores for each step of the graphics pipeline
  - SIMT cores now general purpose enough to execute all graphics pipeline stages (generically called a shader core)
- **Now also used for general-purpose GPU (GPGPU) programming**

Using a GPU: Program

- **All terms here based on NVIDIA CUDA**
- **Basic unit of programming: kernel**
  - A piece of code that can be run in parallel
  - A program consists of multiple kernels
  - Each kernel is assigned to a **grid** of threads
- **Basic unit of execution: thread block**
  - A group of threads that can be executed in parallel
  - Thread block is limited to 1024 threads
  - Multiple blocks (of the same thread count) can be combined to form a grid
- **Kernels and thread blocks are managed by a software runtime**



Using a GPU: Execution Flow

1. Host (i.e., CPU) sends a request to the GPU runtime to start a program
2. Runtime copies memory from host address space to the GPU address space (separate memory in discrete GPUs)
3. Runtime allocates per-thread resources (e.g., registers, scratchpad)
4. GPU executes each kernel in the program
5. Runtime copies results from GPU address space to host address space

Common Issues in GPUs

- **Sharing memory with the CPU is challenging**
  - GPU has its own physical memory and address space
    - Not managed by the OS!
    - Requires program to copy data between the CPU and GPU
  - **Unified Virtual Memory**
    - Shared address space between the CPU and the GPU
    - No more need to copy data back and forth
    - Big issue: coordinating virtual-to-physical page mappings
- **Thread divergence makes lockstep execution inefficient**
  - Each thread can have control flow instructions (e.g., branches)
  - **Branch divergence** occurs when threads inside a warp branch to different execution paths
  - **Memory divergence** occurs when some threads hit in a cache and others must go to main memory

Resource Allocation and Program Portability

- **GPUs have a several resources that must be allocated**
  - Programmer dictates the resources needed per thread
  - Runtime simply provides what each thread/warp needs until it cannot fit any more threads on the SM
- **How does a programmer know how to allocate resources?**
  - **Performance tuning:** for each GPU architecture, **test out different resource allocations and assign the best one**
  - GPU architectures tend to keep the ratio of resources per warp context/per SM constant within a GPU generation
- **Requires retuning every time a program is ported to a different architecture**
- **Auto-tuning tries to automate resource allocation (with mixed results)**

Heterogeneous Computing

- **While GPUs can help with massive multithreading, they clearly have their own challenges**
- **Reality: different types of compute require different types of hardware and systems**
- **Today: heterogeneous computing reigns supreme**
  - CPUs handle more traditional workloads
  - GPUs handle highly parallel programs and graphics
  - Other hardware accelerators are designed for very common tasks

- **We could just have separate chips for each…**
- **… But today we put them all into a single system-on-chip (SoC)**

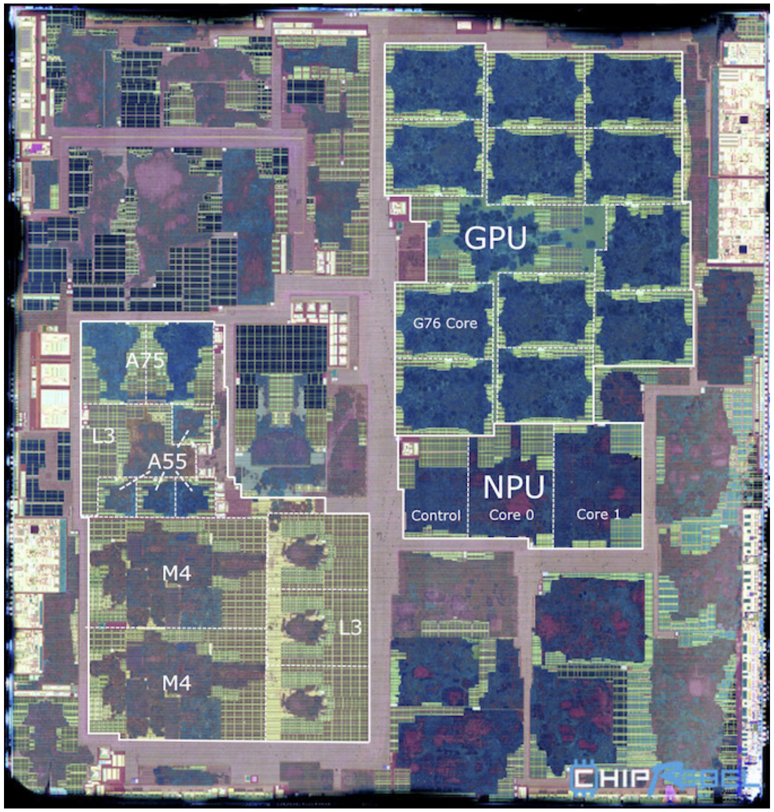
Why a System-on-Chip?

- **There used to be separate chips for almost everything**
  - Floating-point units (e.g., the Intel x87)
  - Caches
  - Memory and I/O controllers
  - Discrete modems and accelerators (if present)
- **A few fundamental changes have made it more desirable to combine these on a single chip**
  - Smaller communication distances: faster latencies, higher bandwidth, and lower energy
  - Better use of the available transistors and chip area
- **CPUs integrated some, but not all, units (e.g., FPUs, caches, memory controllers) over the last few decades**
- **1970: the first SoC (used by Pulsar for the first digital watch)**
- **Mid-2000s: SoC development led to the smartphone revolution**

Common System-on-Chip Components

- Processor cores
- Graphics processing units (GPUs)
- Caches (L1/L2/L3 today)
- Digital signal processors (DSPs)
  - Accelerators that perform signal processing operations for sensors, multimedia processing
  - Often made up of vector extensions
- Networking modems (e.g., WiFi, 4G LTE)
- AI/ML accelerators (i.e., neural processing units)
- On-chip interconnect

Samsung Exynos 9820 (2019)



Source: AnandTech/Chip Rebel

How Do We Use an SoC?

- **Each SoC can have a different set of components**
- **Before: one fixed set of resources, then write software for them**
- **Now: software informs the hardware design!**
  - Start with basic structures (e.g., CPU, cache, GPU)
  - Analyze software to find most common operations/tasks
  - Define an SoC architecture (using basic structures, premade blocks known as IP cores, and custom-designed logic)
  - Optimize your software for your SoC
- **System design can be challenging**
  - How do we manage and coordinate all of these components?
  - Burden often left on the systems programmer
  - Runtimes or APIs are commonly used by application developers

SoCs Have Helped Move Us to the Cloud

- **Traditional model**
  - Compute everything locally
  - Worked great for small-data workloads of the past
  - Difficult to shrink the size of a computer (e.g., an SoC)
- **Today: data centers and cloud computing**
  - Your computer sends a request across the network
  - Giant "farms" of computers perform a significant portion of the computation
  - Result is sent back to your computer
  - A key enabler of smartphones
  - These farms typically service billions of requests each second (think Google or Facebook)
  - Requires highly-available, reasonably fast network

Cloud Computing vs. Data Centers

- **Data center**
  - The company providing a service owns and maintains its own servers for the service (or pays someone to do so)
  - Machines are dedicated for that company
  - Can (but don't always) run code natively

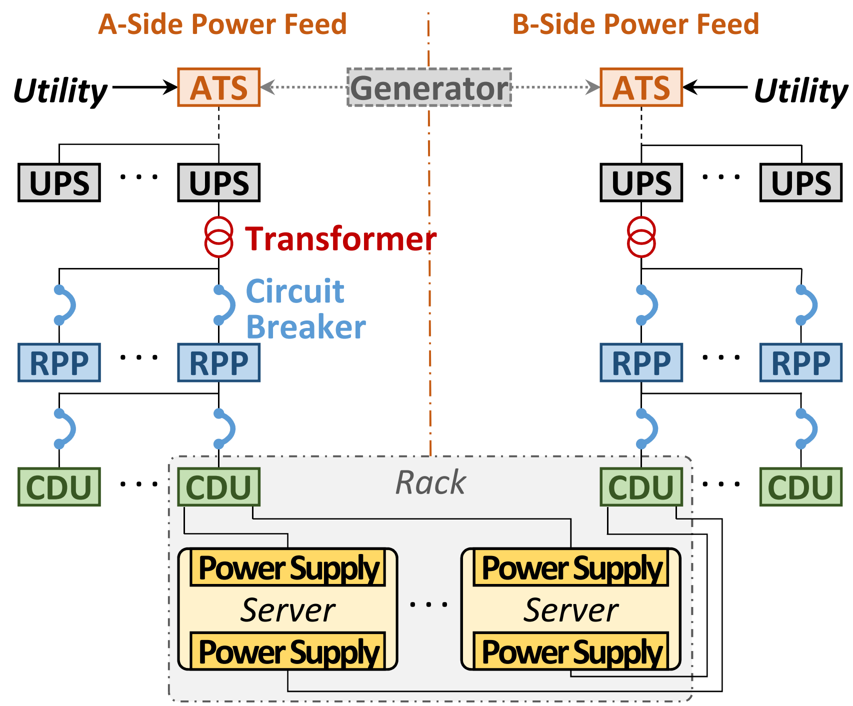
- **Cloud computing**
  - The company providing a service runs the service on someone else's servers
  - Machines are shared across many companies and services
  - Typically use **virtual machines** (VMs) or **containers** to allow multiple services to run on a single server without having access to each other's data, and to allow for job migration
  - Examples: Amazon AWS, Microsoft Azure, Google GCP

Running Programs in the Cloud

- **You wrote a program for OS G, but the cloud runs OS H ☹**
- **Virtual machines let you run your program inside OS H!**
  - System virtual machines (i.e., full virtualization)
    - Hypervisor runs inside OS H (the host OS), provides an interface to emulate all of the hardware
    - OS G (the guest OS) runs inside the hypervisor, and *thinks* it is running directly on a machine (the one faked by the hypervisor)
    - Lots of overhead (e.g., 4-level page tables can require as many as 24 memory accesses!)
  - Process virtual machines (i.e., managed runtime environments)
    - Create a platform-independent environment for programs
    - Examples: Java VM, .NET framework
- **Containers: one OS can run multiple isolated kernels**

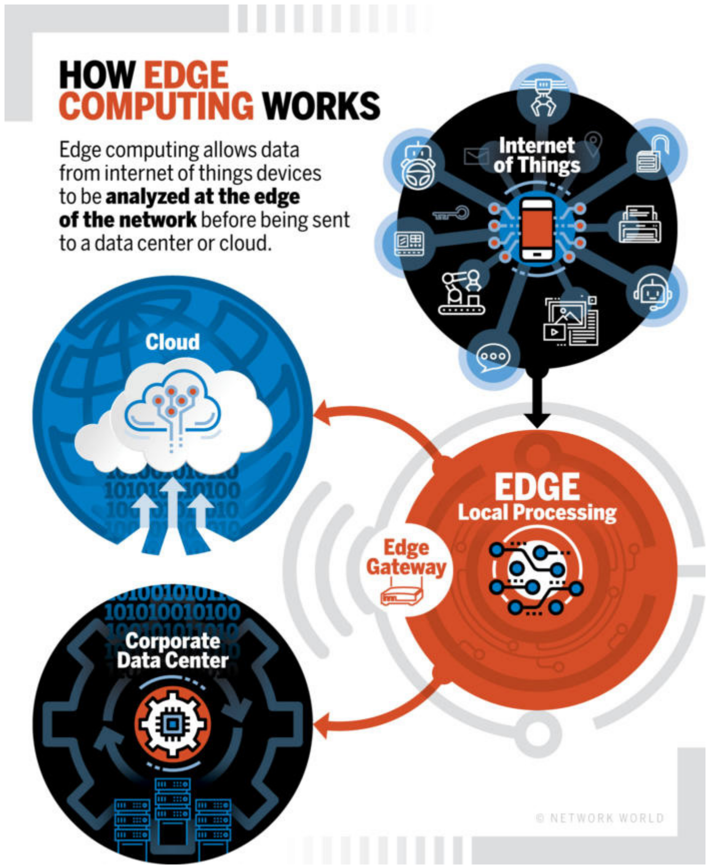
Data Centers Require Significant Power

- **Globally, data centers consume 3% of the world's total power in 2017**
  - 2% of global emissions
  - Projected to be as much as 20% by 2025
- **Need to be efficient but reliable**
  - Redundant power feeds and infrastructure
  - Load varies from day to day, and minute to minute in a day: data centers need to be overprovisioned, and must adjust based on the current load



Bringing Back Local Compute

- Large amounts of data sent to the cloud
- What if our devices could be smart and process (some of the) data for us?
- Internet of Things (IoT)
  - A very wide, distributed network of devices that can all talk with each other
  - Many IoT devices are simpler than smartphones (e.g., smart sensors) – designed to be deployed everywhere
- Edge computing
  - Cloud computing + IoT model pushed almost all compute from a smartphone to data centers
  - Now we're pushing back, because the Internet can't scale as rapidly as data: bandwidth limited, energy hungry



Source: Network World

Rethinking the Computer

- **Today's computers are built off of assumptions made going back to the 1940s**
  - Spatial/temporal locality
  - Instruction-based computation
  - Today's levels of abstractions
- **Applications and use cases have changed significantly**
  - Machine learning and data analytics
  - IoT and edge computing
  - Drones and autonomous vehicles
  - Precision medicine and bioinformatics
  - Mobile apps
- **Shouldn't our computers change as well?**

Computer System Design is Diverging

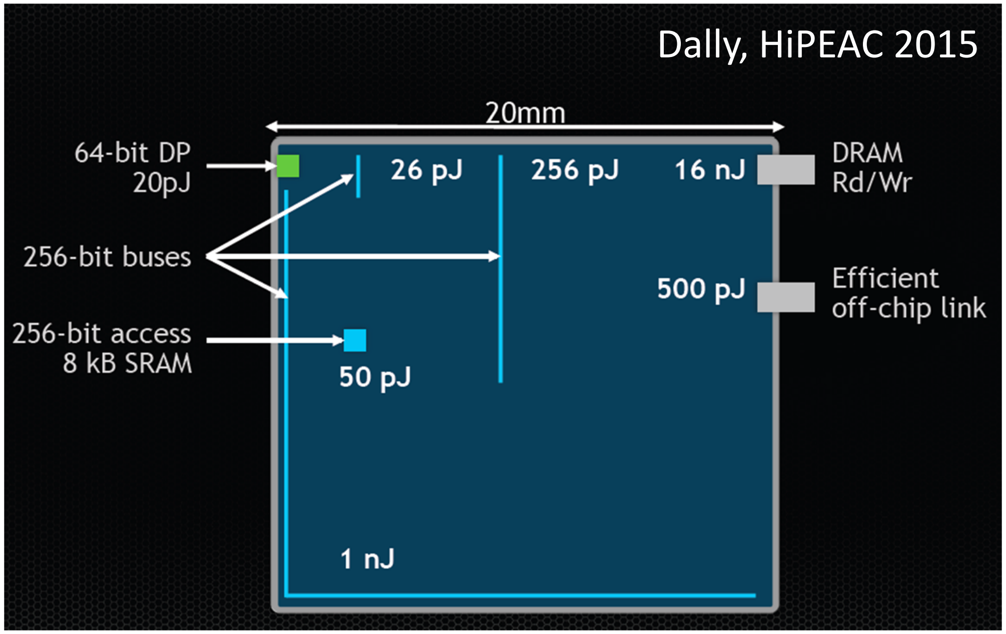
- **Several types of systems are becoming popular**
  - Graphics processing units (GPUs)
  - Mobile systems-on-chip (SoCs)
  - Data centers and cloud computing
  - Internet of things (IoT)/edge computing
- **A few promising designs may emerge in the future**
  - Processing-in-memory (PIM)
  - Neuromorphic computing

Hardware Hasn't Kept Up with the Times

- **Beefy processing engines (CPUs, GPUs, accelerators)**
  - Large numbers of cores, high degrees of multithreading
  - Out-of-order execution in CPUs
  - Many low-power optimizations
- **Designed for infrequent memory accesses**
  - Caches highly dependent on locality
  - Long, narrow off-chip memory channel to connect CPU with DRAM
- **While programs are becoming more data-centric, computer architectures remain compute-centric**

The Cost of Data Movement in Modern CPUs

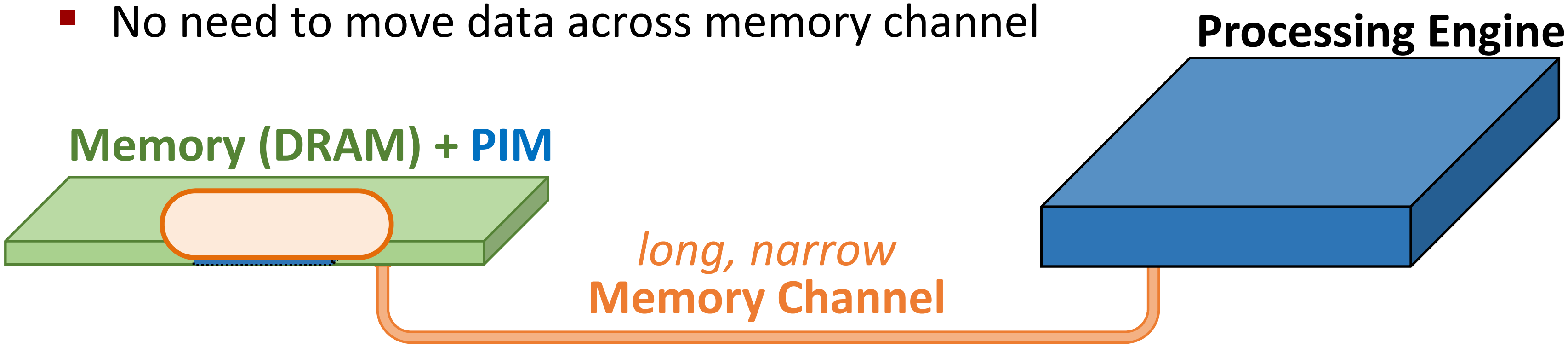
- **In terms of energy costs, data movement dominates compute**
- **DRAM responsible for 25–50% of a computer's total energy**
- **Off-chip memory channel: ~30% of DRAM energy**



- **Data movement is a major bottleneck in modern systems**
  - High energy spent on off-chip communication
  - Pin-limited bandwidth
  - High latency
  - Identified as the **von Neumann bottleneck** by Jim Backus in 1977

Can We Avoid Moving Data Around?

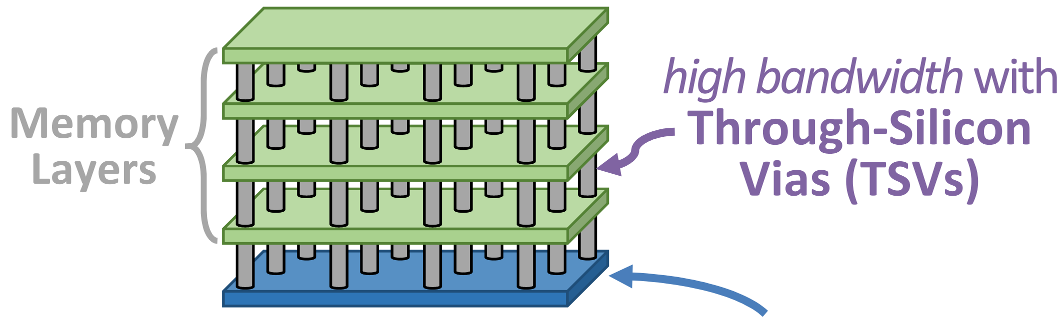
- **Processing-in-memory (PIM)**
  - Add some compute capability to memory
  - No need to move data across memory channel



- **PIM has been proposed as early as 1970**
- **New innovations in memory design have finally brought PIM close to a reality**
- **Kind of like an SoC: add new components/functionality, but this time near memory**

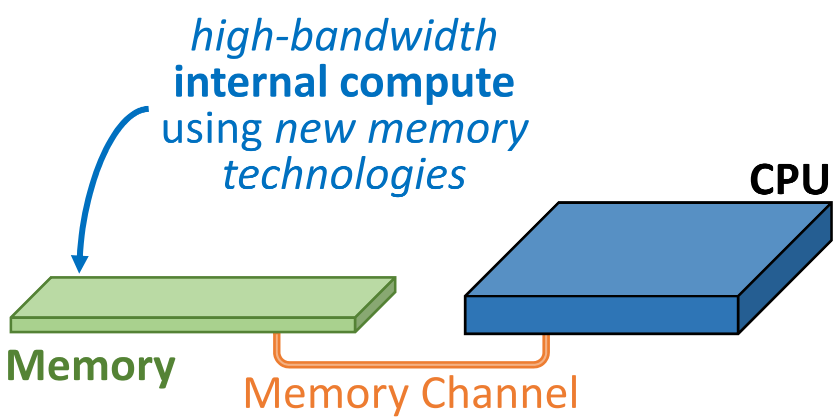
Two Variants of PIM

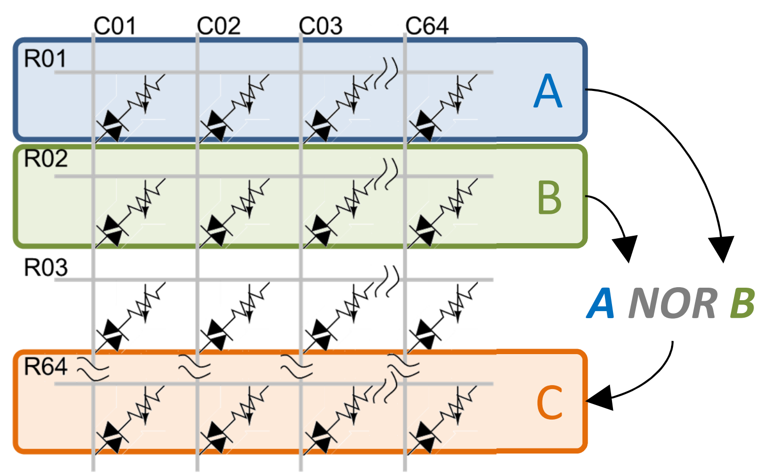
- **Variant 1: Processing-Near-Memory**



we can add *small processing engines to the* **Logic Layer** or on nearby chips

- **Variant 2: Processing-Using-Memory**





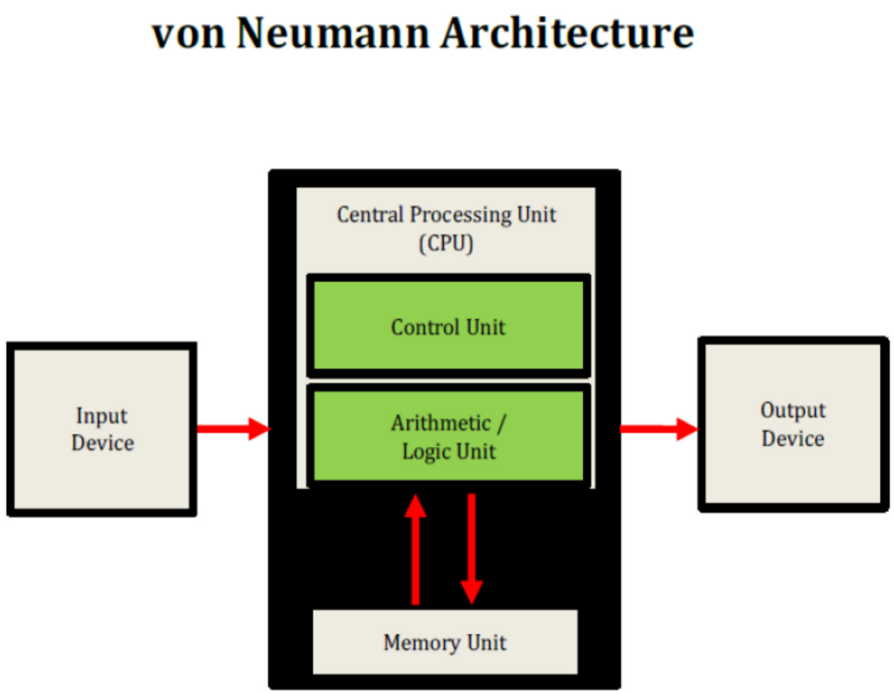
Great… How Does This Affect Systems?

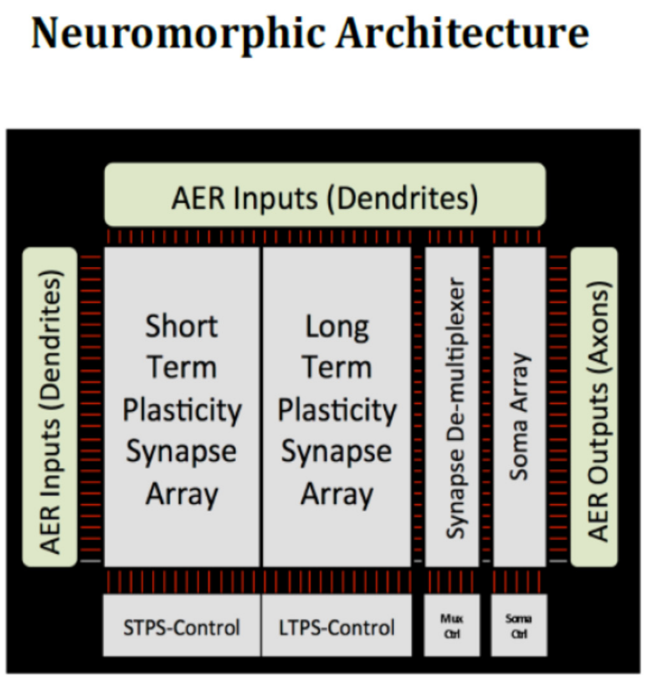
- **Once PIM hardware exists, programmers must be able to use it**
  - Tough sell: force them to **learn a new programming model**
  - Path to broad adoption: **adapt PIM to existing models**
- **Unfortunately, PIM logic can't easily make use of a lot of systems essentials**
  - Support for multithreading: OS needs to be exposed to PIM
  - Virtual memory: expensive for PIM to access TLBs in the CPU
  - Coherence/consistency: these can introduce a lot of traffic between the CPU and PIM
- **How do compilers generate code for PIM logic?**
- **What about handling branches?**
- **Active research area: solving these challenges in the coming years**

Motivating Neuromorphic Computing

- **Artificial neural networks are the hot stud of computing now**
  - Forms implicit relationships between inputs and outputs
  - Can learn and represent very powerful models
  - However, ANNs are *not* accurate representations of our brain
- **What can our brain do?**
  - We can track things moving in real time as we see them
  - We can learn with uncertainty (ANNs need to experience everything)
  - And yet our brain runs at only a few Hz (vs. GHz for ANN accelerators)
- **Many applications can benefit from designing computers that look more like our brain**

Neuromorphic Architectures





Source: US DOE Report

- **Several chips exist: IBM TrueNorth, Intel Loihi**
- **How do you use this?**
  - Replace CPUs in existing systems? Add as accelerators?
  - IBM made its own object-oriented language (Corelet)

Summary

- **Computing is looking more and more heterogeneous**
  - Many different types of hardware
  - Many different types of use cases
- **There may be more radical hardware changes ahead**
  - Keeping up with significant shifts in applications
  - We need to think of what systems support will look like after these changes!
- **Does it mean that what you've learned in 213 is useless?**
  - No! Most of the core ideas will still stick around for decades
  - New systems are still built on the same underlying principles
- **It's an exciting time to be working in systems!**