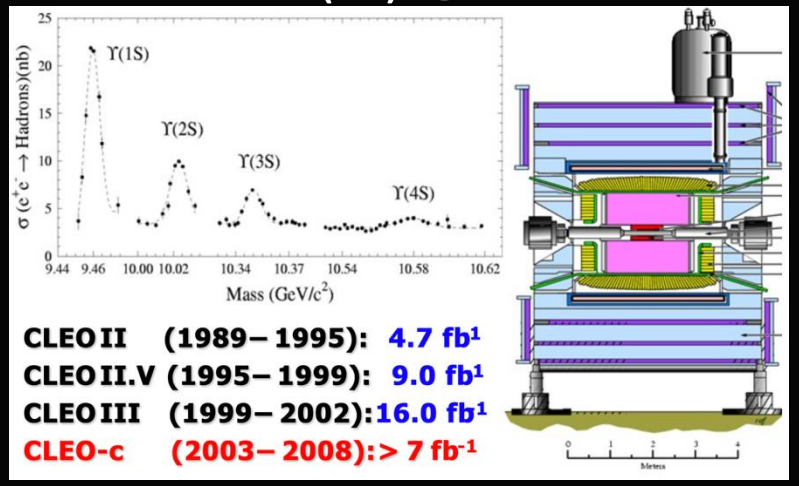
Lightning Introduction to Parallel Programming

David Lange

Princeton University

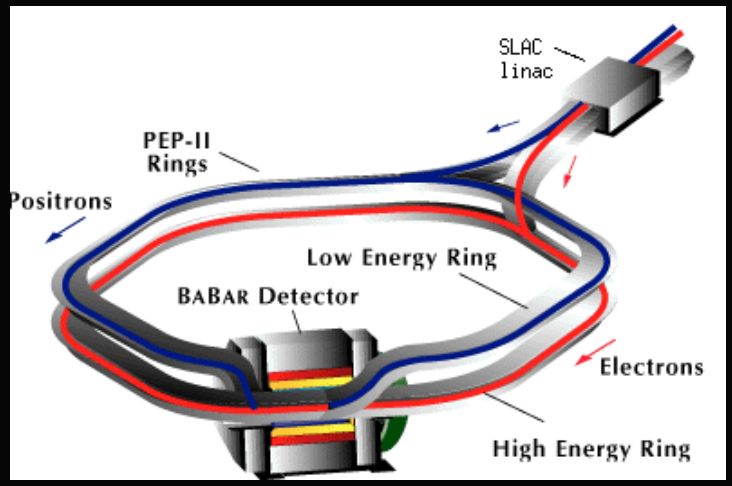
My career in high-energy physics

CLEO-II(.5) @ Cornell



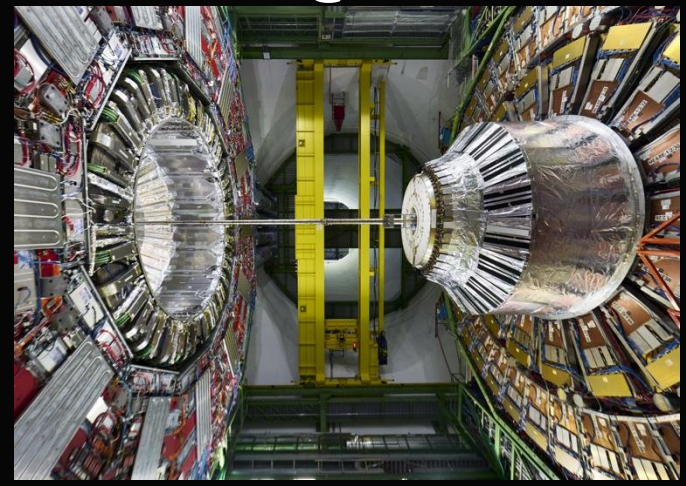
- Silicon detector calibration + operations
- Form-factor analysis in semileptonic decays
- Event generators

BABAR @ SLAC



- CP violation analysis
- RPC detector operations + software
- Event generators (“EvtGen”)
- Event reconstruction software

CMS @ CERN

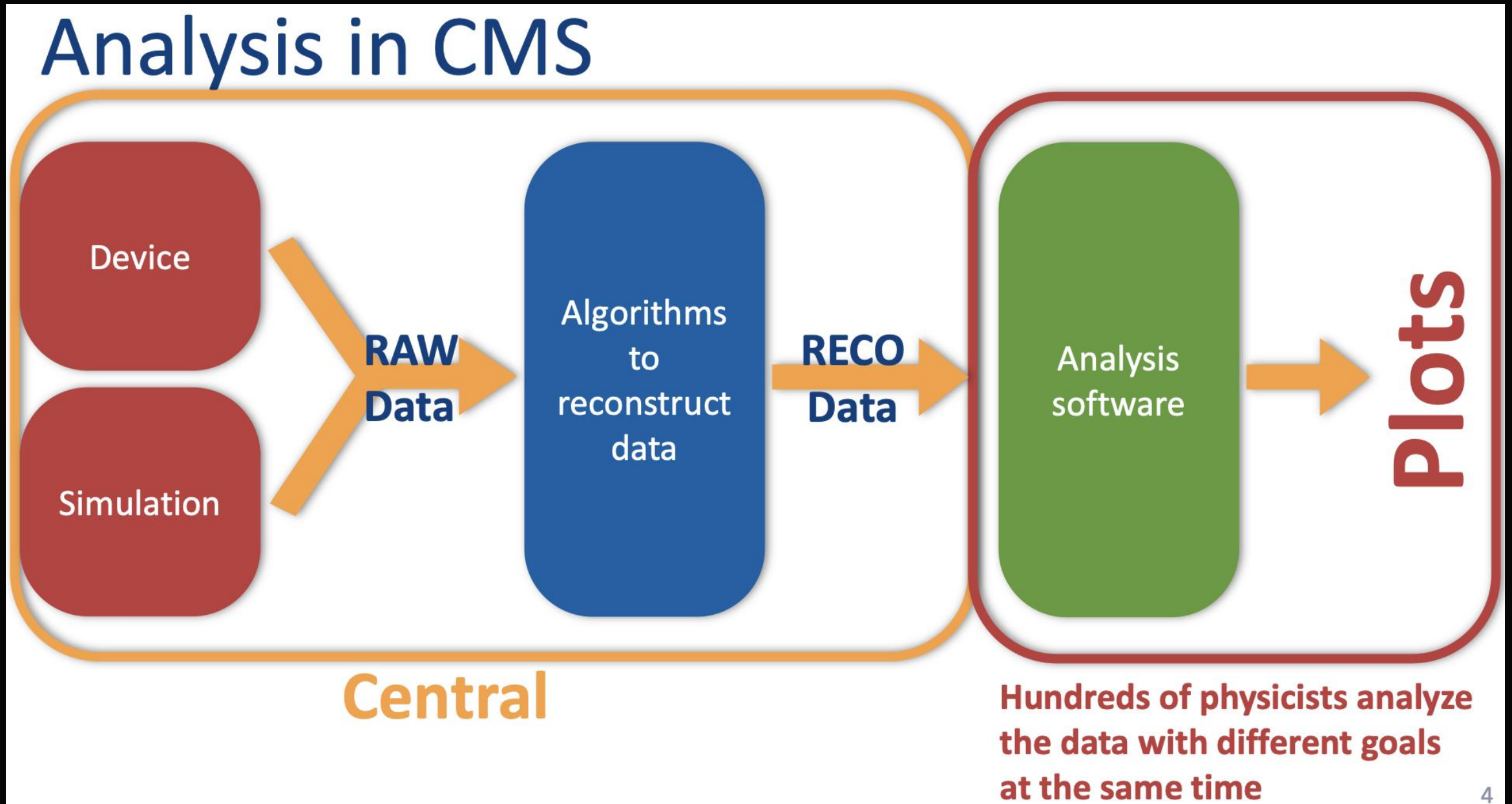


- Event reconstruction software
- Simulation techniques
- Computing resource projections

My Goal for Today

- Why worry about parallel programming for science?
- Considerations for high energy physics codes
- Methods , approaches and hardware
- Languages, tools and resources (likely to be skipped today)
- Using GPUs with Python

Bulk data processing vs time to insight



How can we get our science done faster or cheaper?

- Reduce the complexity of algorithms in the application
  - Write better code
  - Consider approximations
  - Consider needed/appropriate numerical precision
- Increase the speed and capacity of the compute
  - Increase clock frequency
- Find tasks within the application that can be performed in parallel

How can we get our science done faster?

- Reduce the complexity of algorithms in the application
  - Write better code
  - Consider approximations
  - Consider needed/appropriate numerical precision
- Increase the speed and capacity of the compute
  - Increase clock frequency
- Find tasks within the application that can be performed in parallel

How can we get our science done faster?

- Reduce the complexity of algorithms in the application
  - Write better code
  - Consider approximations
  - Consider needed/appropriate numerical precision
- Increase the
  - Increase c
- Find tasks w llel

We won't talk about this today

Efficient coding and optimization techniques can easily fill an entire course (or an entire career…)

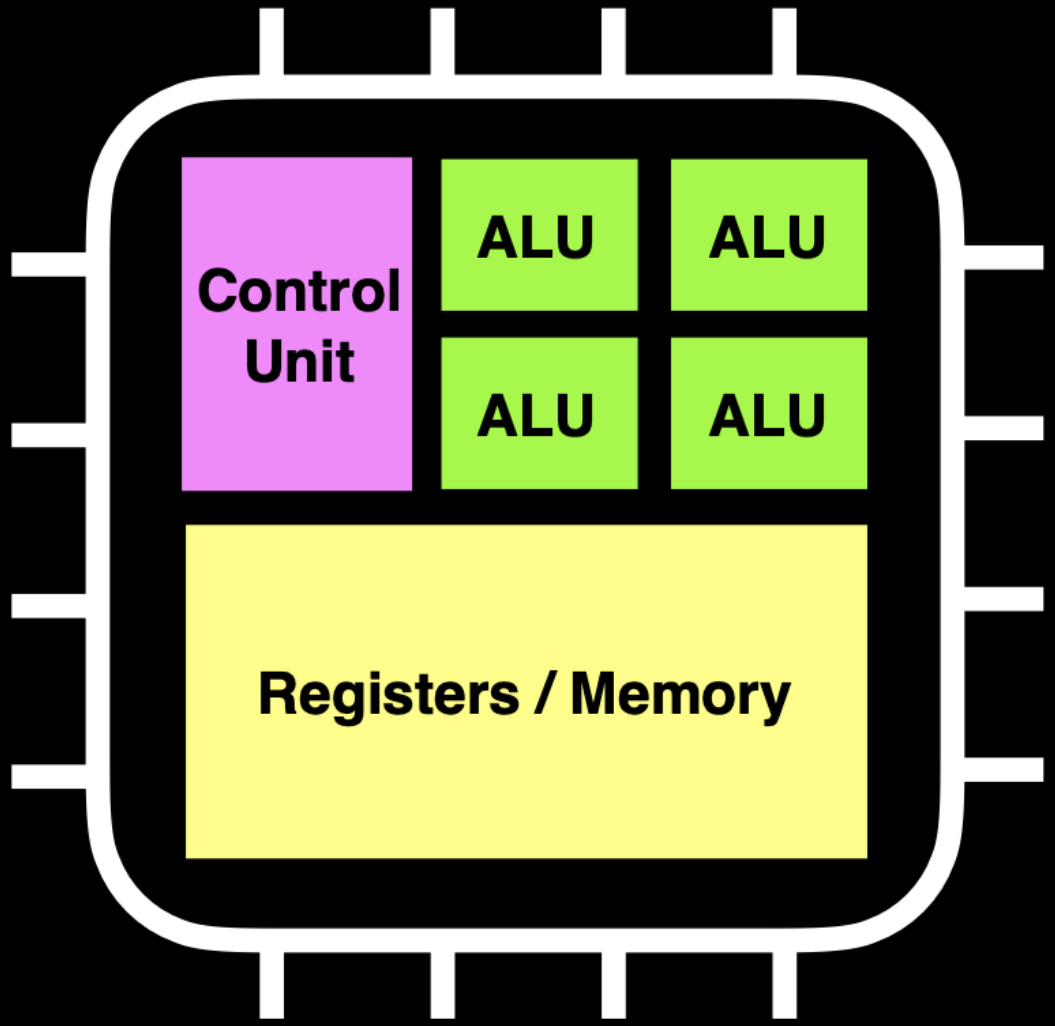
How can we get our science done faster?

- Reduce the complexity of algorithms in the application
  - Write better code
  - Consider approximations
  - Consider needed/appropriate numerical precision

- Increase the speed and capacity of the compute
  - Increase clock frequency

- Find tasks within the application that can be performed in parallel

Central Processing Units (CPUs)



Arithmetic Logic Unit (ALU):
To perform arithmetic and logic operations

Registers / Memory:
Fast memory for Input and output of ALUs

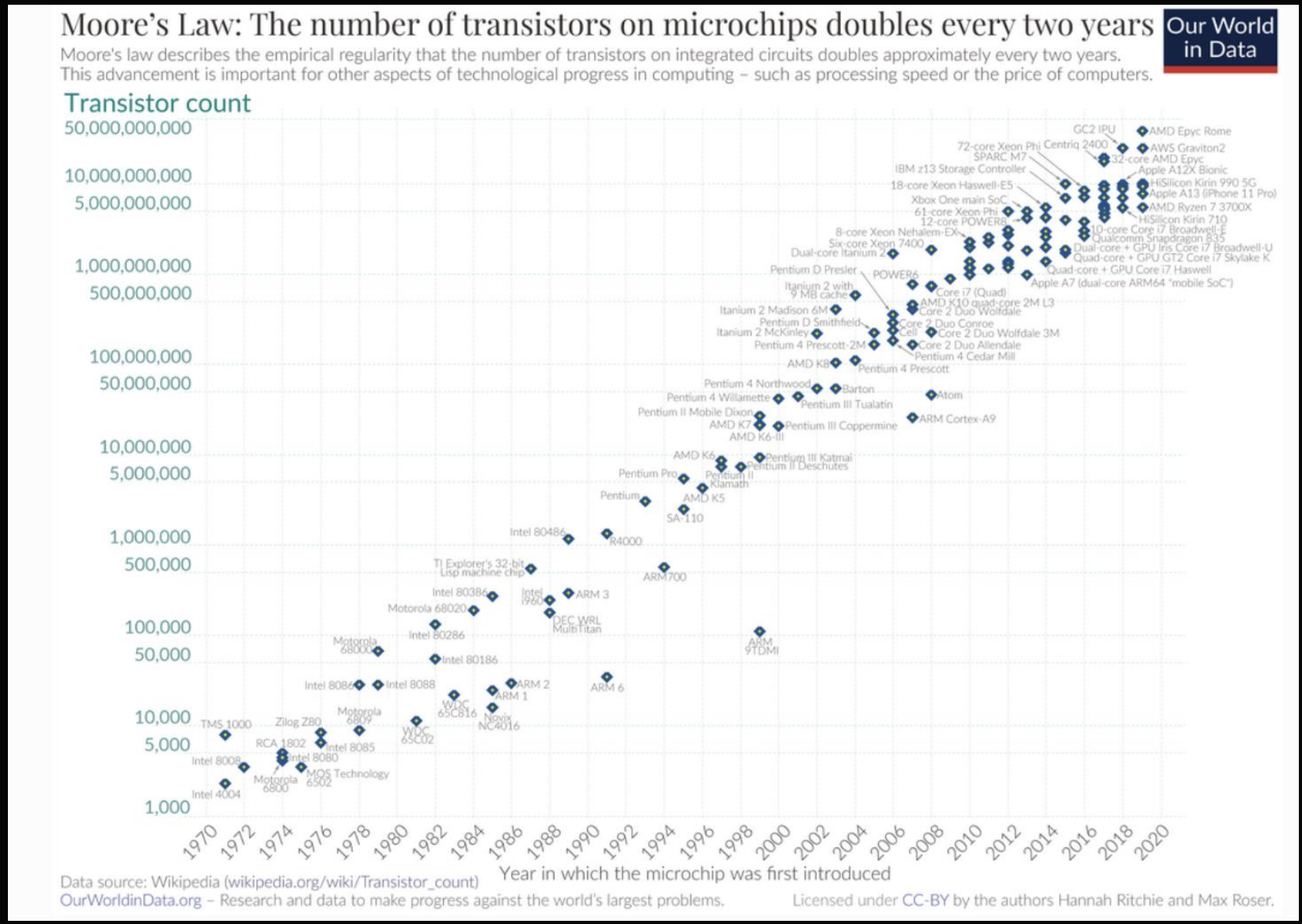
Control Unit:
Directs operation of the processor; moving memory, executing ALUs, etc.

Fundamentally, "Get Numbers" and "Calculate"
Memory *Logic*

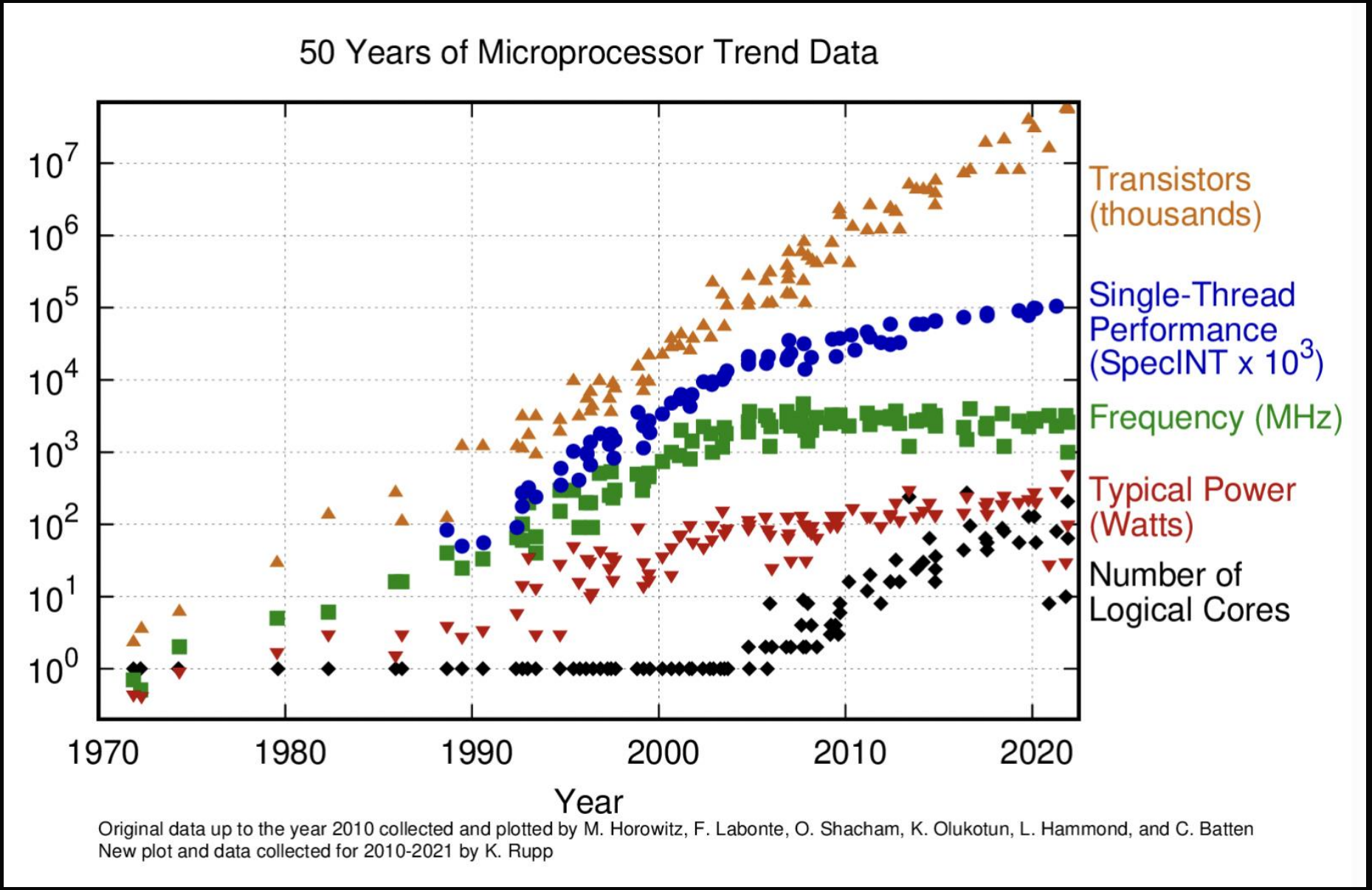
How to go faster?

- *More logic units in same space = more transistors*
- *Faster clocking = higher frequency*
- *I/O performance needs to keep up…*

More transistors? Moore's Law



Trends in transistors to compute architectures

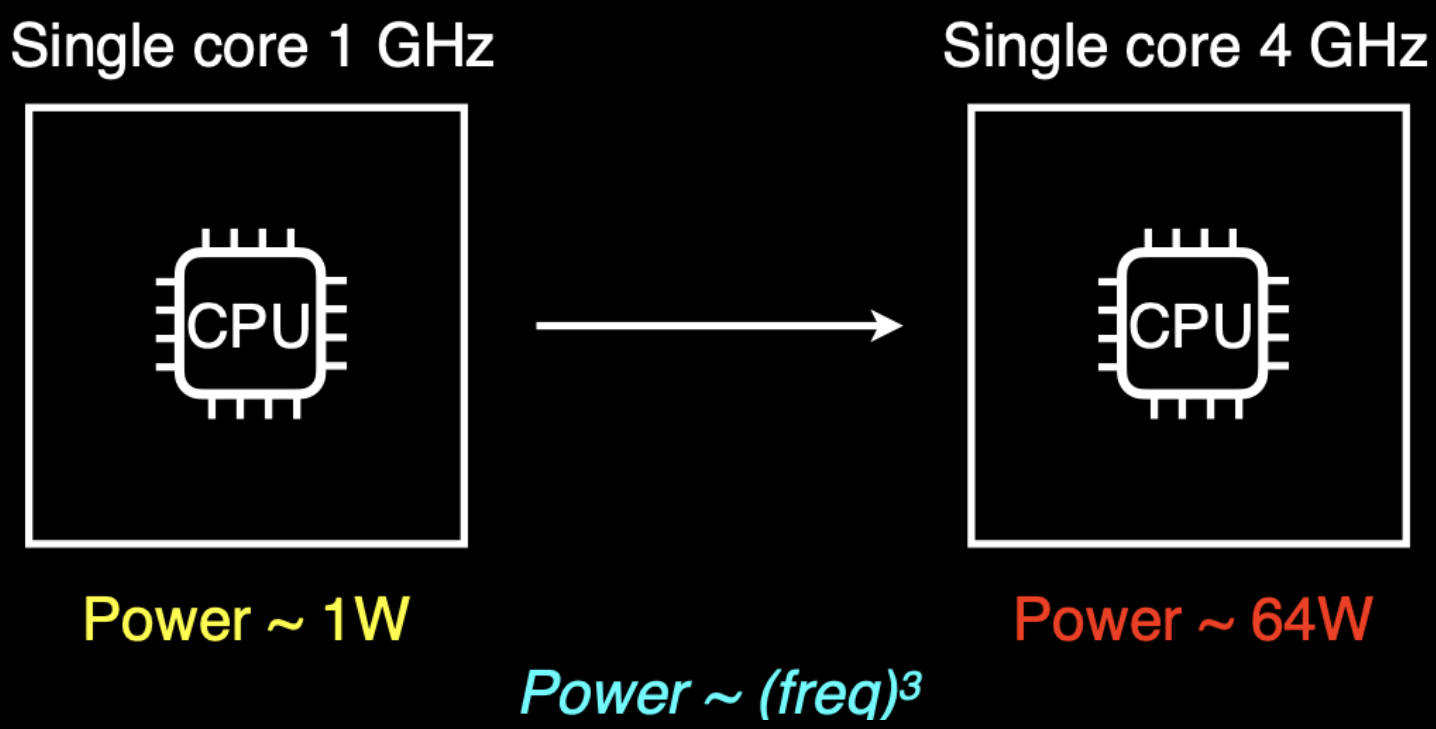


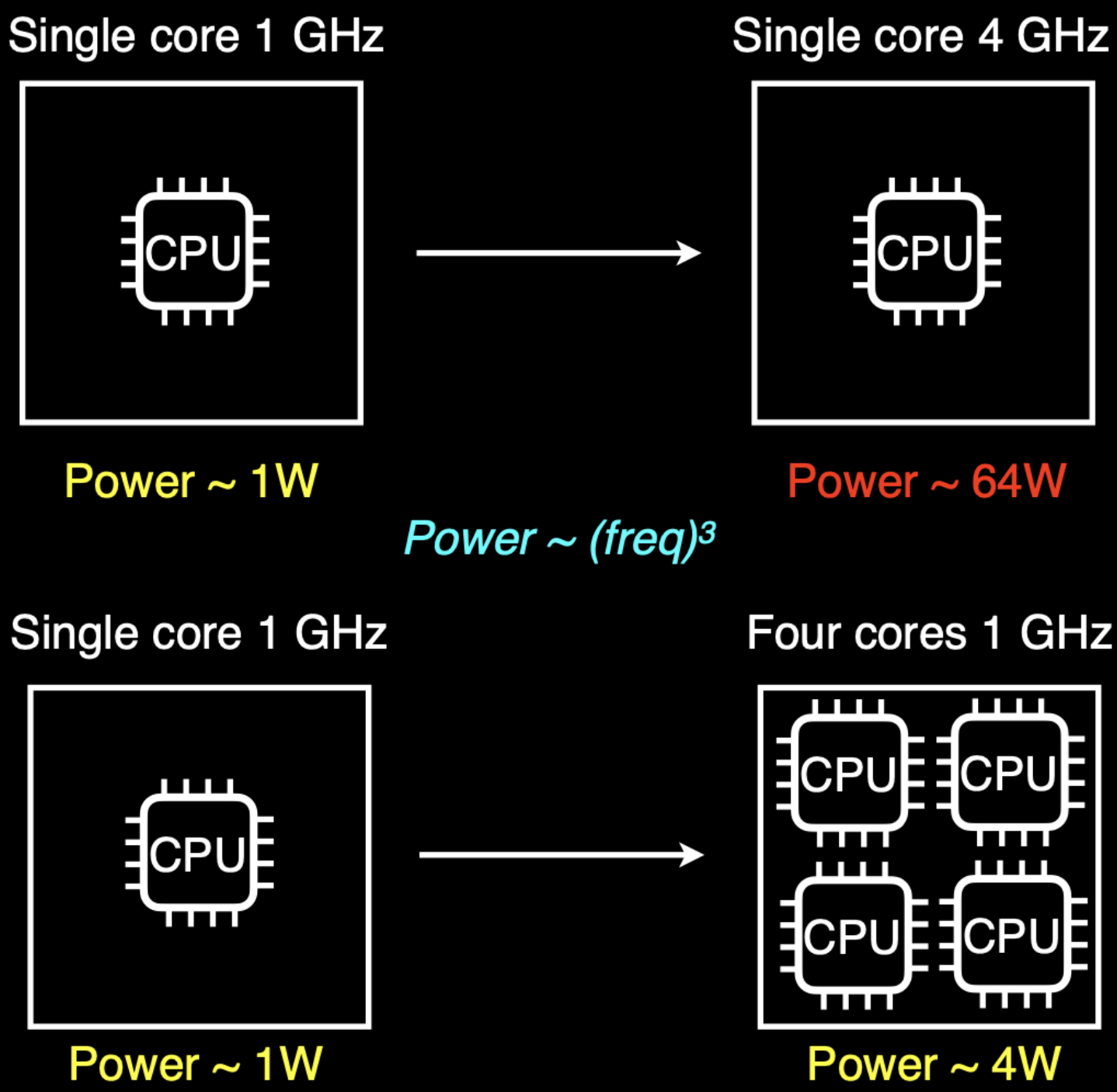
What happened in the early 2000s?

- **Dennard scaling** suggested that each individual transistor in a new generation would be cooler and draw less power
  - Broke down in the early 2000s. This effectively stopped the continuous trend of clock speed increase in CPUs

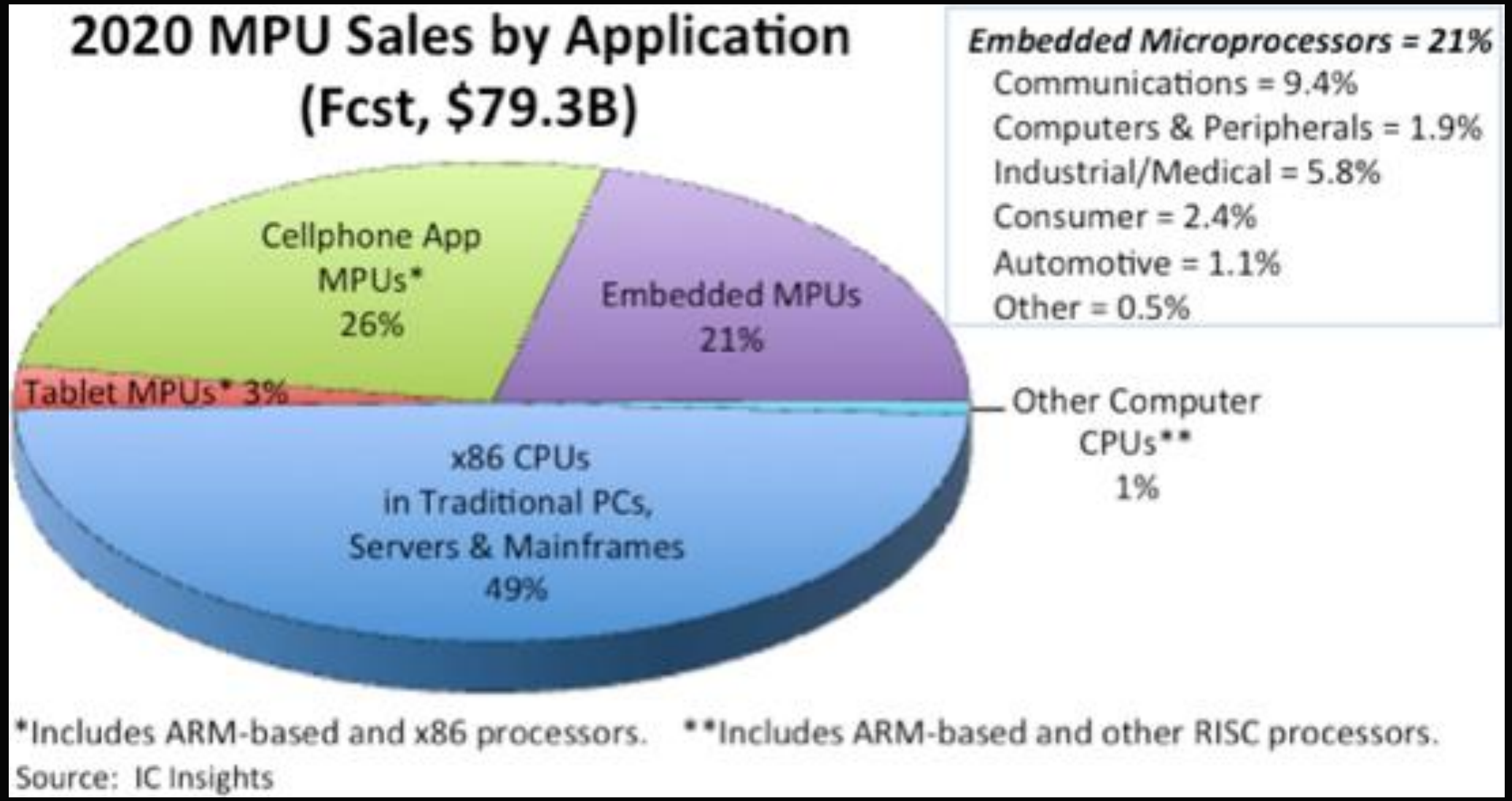
Why?

- More power per core brings two problems: more power (more money) to run the CPU and higher heat removal requirements (yet more power/money)

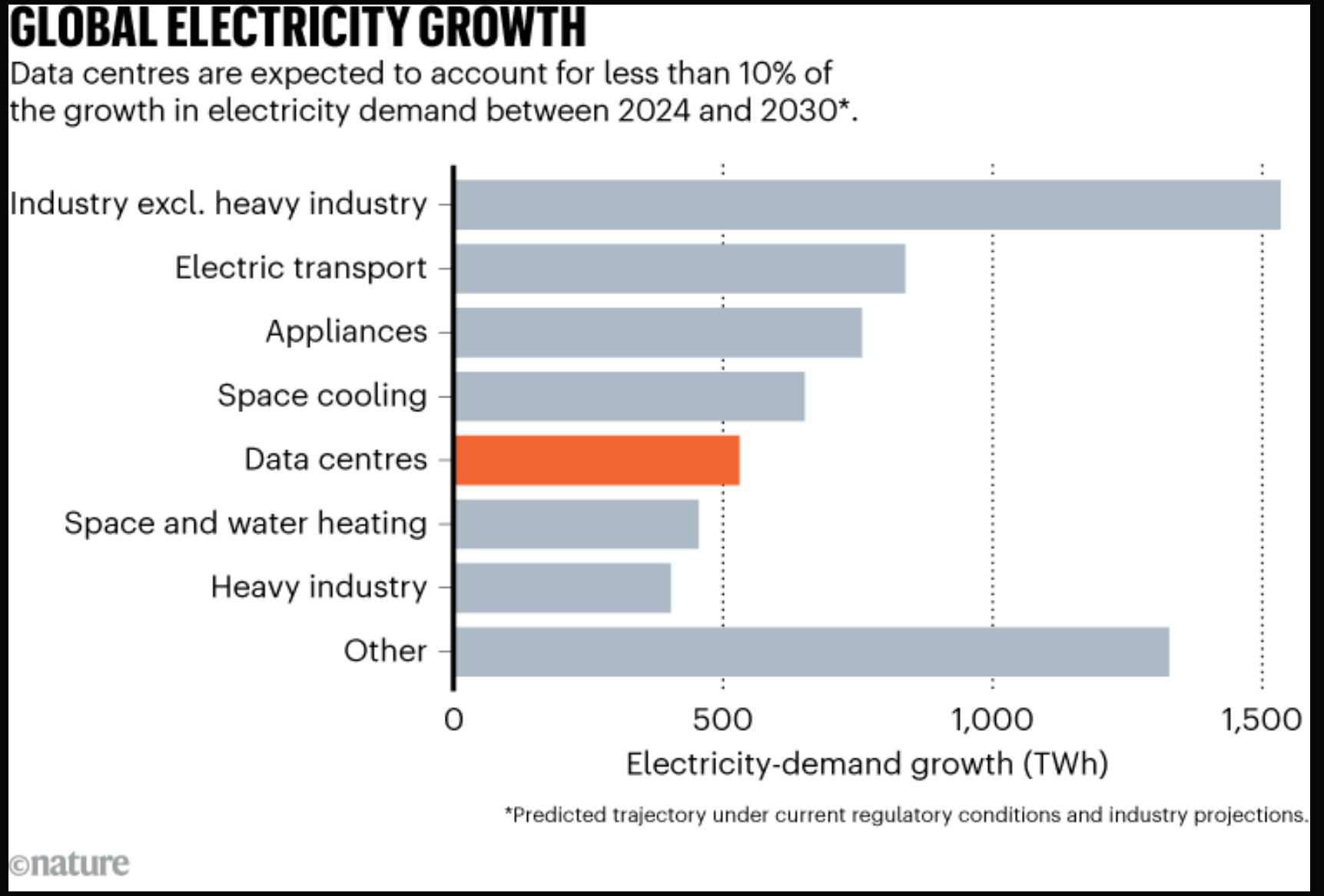

Single core 1 GHz
CPU
Single core 4 GHz
CPU
Power ~ 1W
Power ~ 64W
Power ~ (freq)³


Single core 1 GHz
CPU
Single core 4 GHz
CPU
Power ~ 1W
Power ~ 64W
Power ~ (freq)³
Single core 1 GHz
CPU
Four cores 1 GHz
CPU
CPU
CPU
CPU
Power ~ 1W
Power ~ 4W

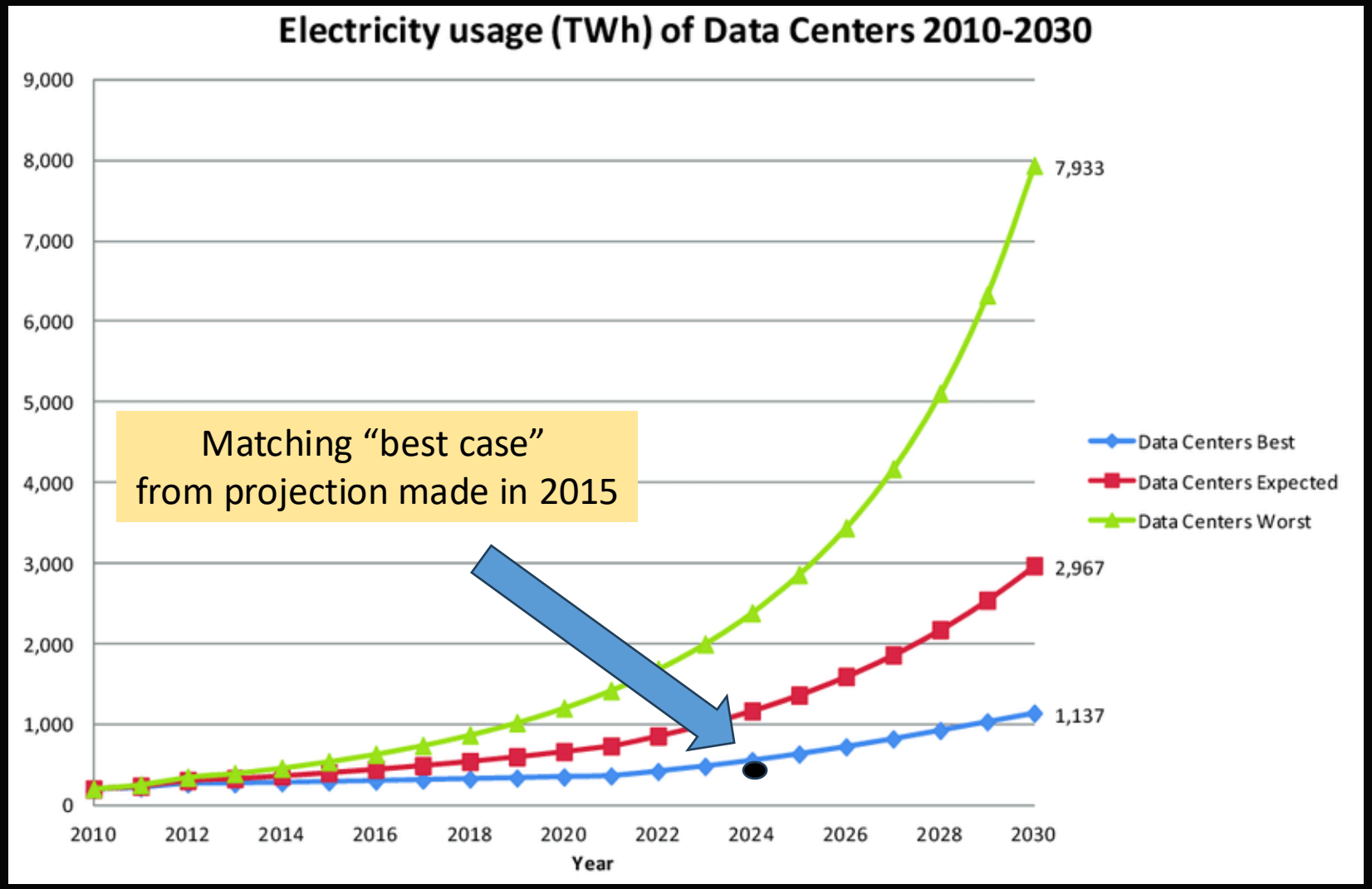
Data centers and low-power applications drive the market



Power is a leading consideration in data centers (too)



Just how important is energy in data centers?



http://dx.doi.org/10.3390/challe6010117

How can we get our science done faster?

- Reduce the complexity of algorithms in the application
  - Write better code
  - Consider approximations
  - Consider needed/appropriate numerical precision
- Increase the speed and capacity of the compute
  - Increase clock frequency
- Find tasks within the application that can be performed in parallel

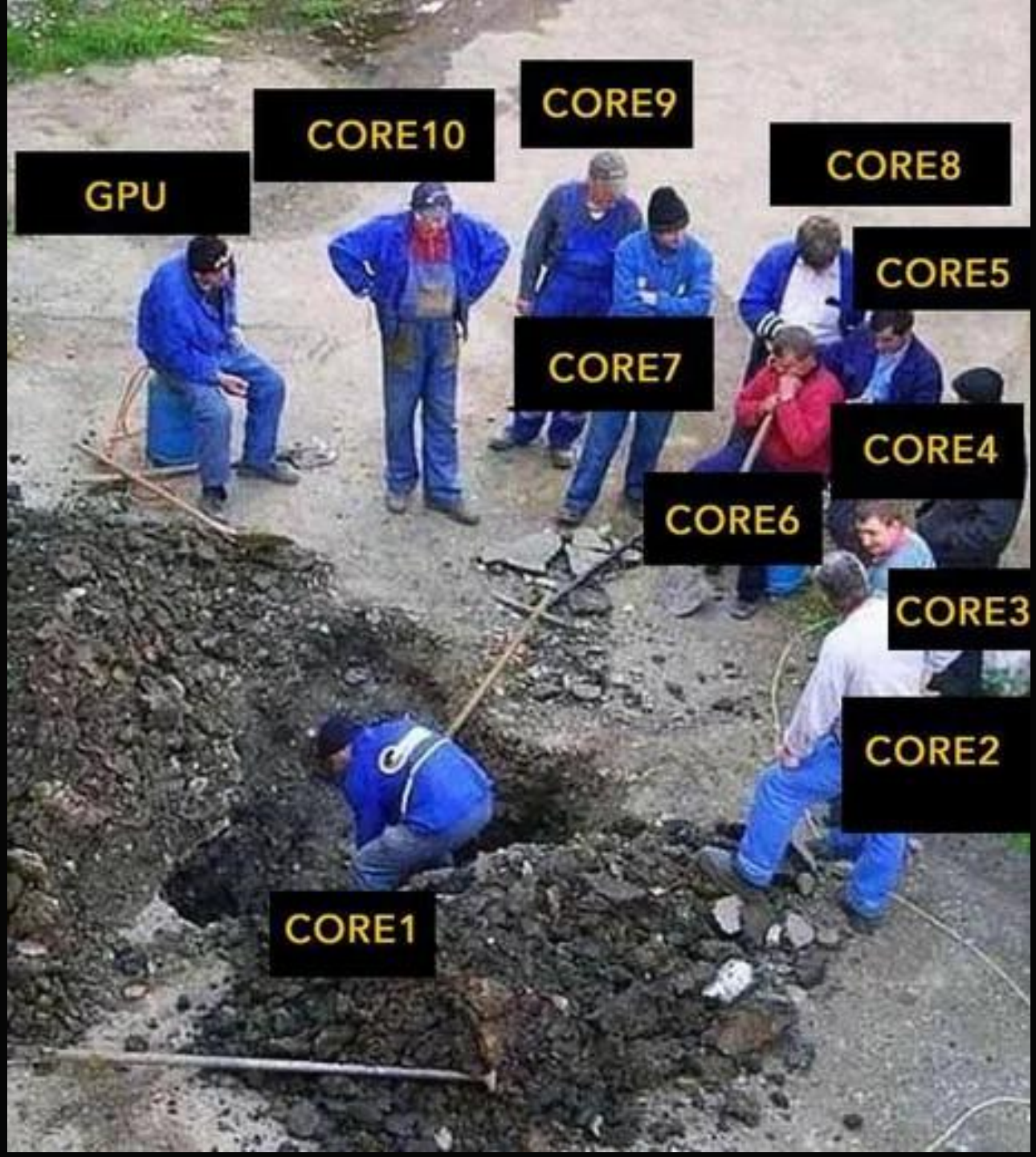
Parallelization of certain tasks of our program, so that they are executed simultaneously (threads)

For this it is necessary to:

- Have parallel platforms where to run.
  - We just saw that this is not a problem. All modern compute is parallel to some extent

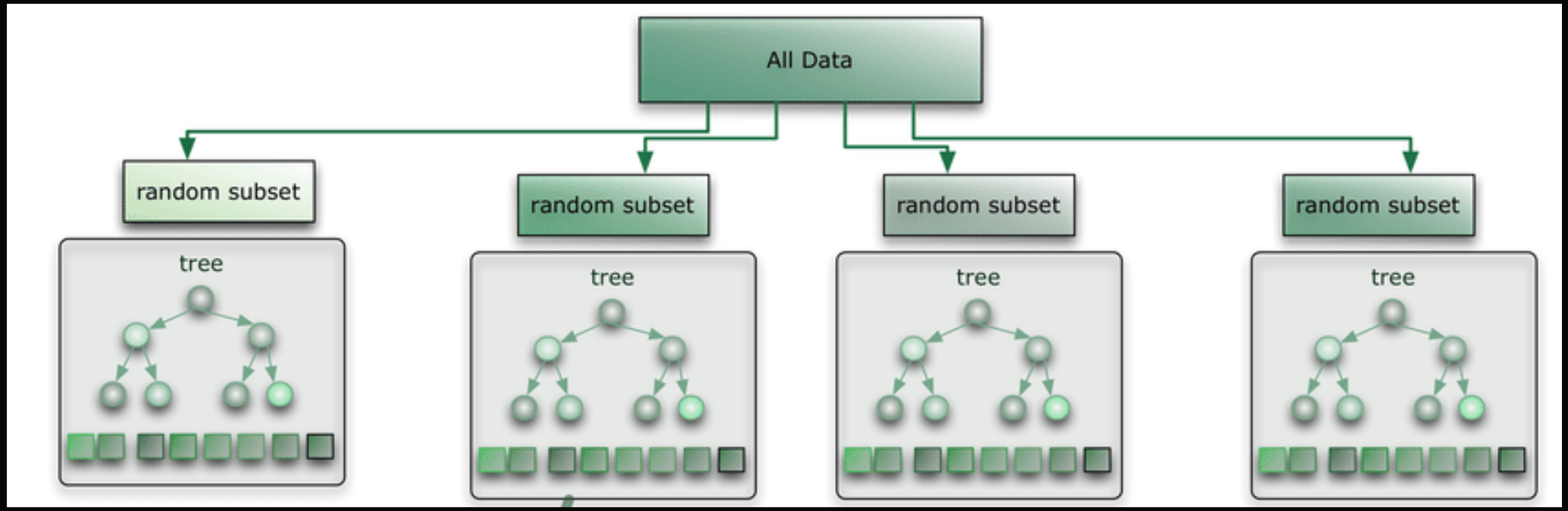
- Adapt your application to be **efficiently** executed in parallel.
  - How easy is this?

Parallel hole digging

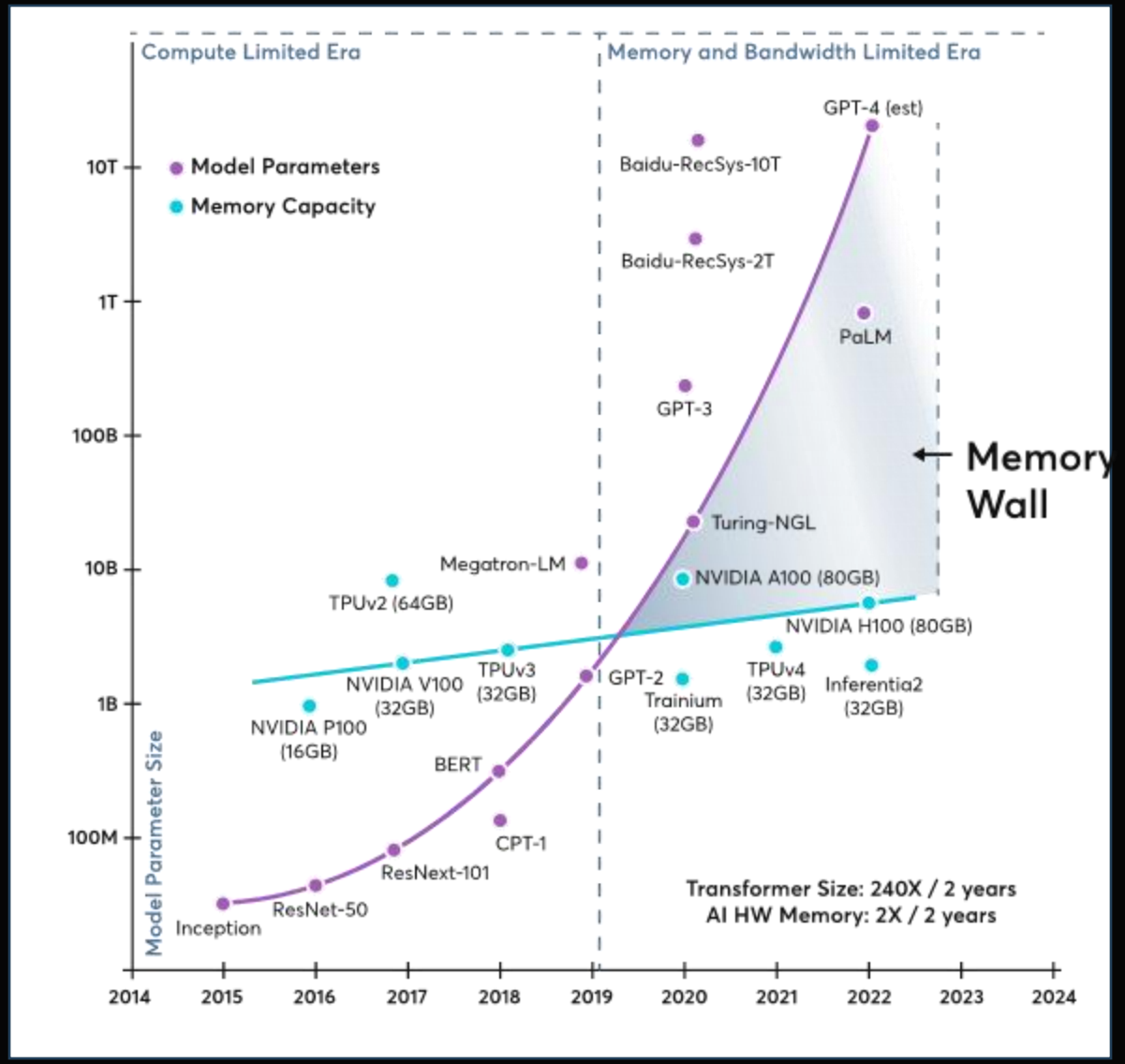


Why not just “embarrassingly” parallel?

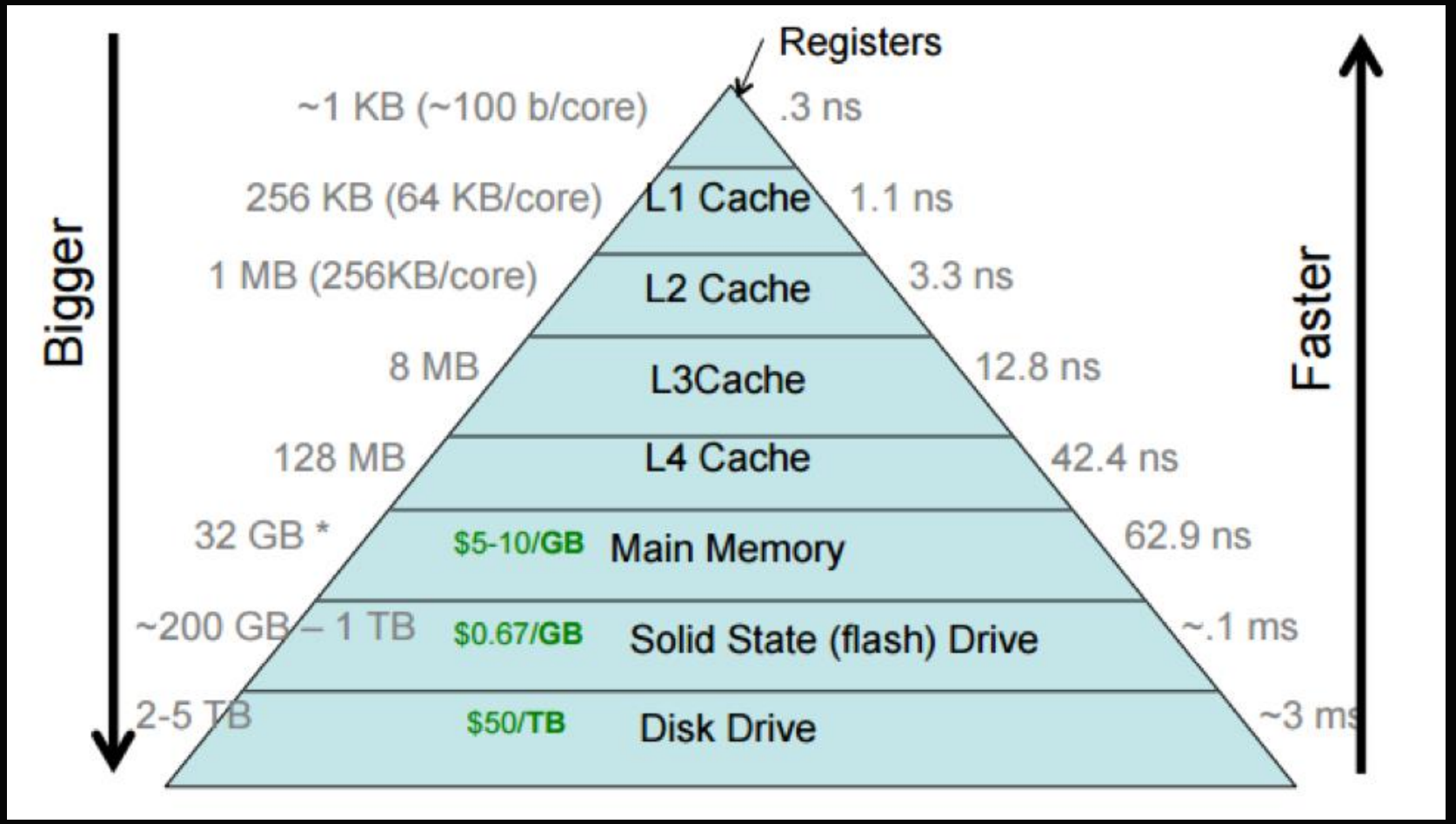
- HEP experimentalists have lots of events(100s of billions per year), and **events can be analyzed independently**.
  - Can’t we just write applications that do the equivalent of digging lots of holes?



Memory demands out pace supply



Another way to look at the importance of memory footprint



Memory efficient algorithms will run faster!

What does all this mean?

Science does not drive the market for modern computing, so we must adapt to market trends

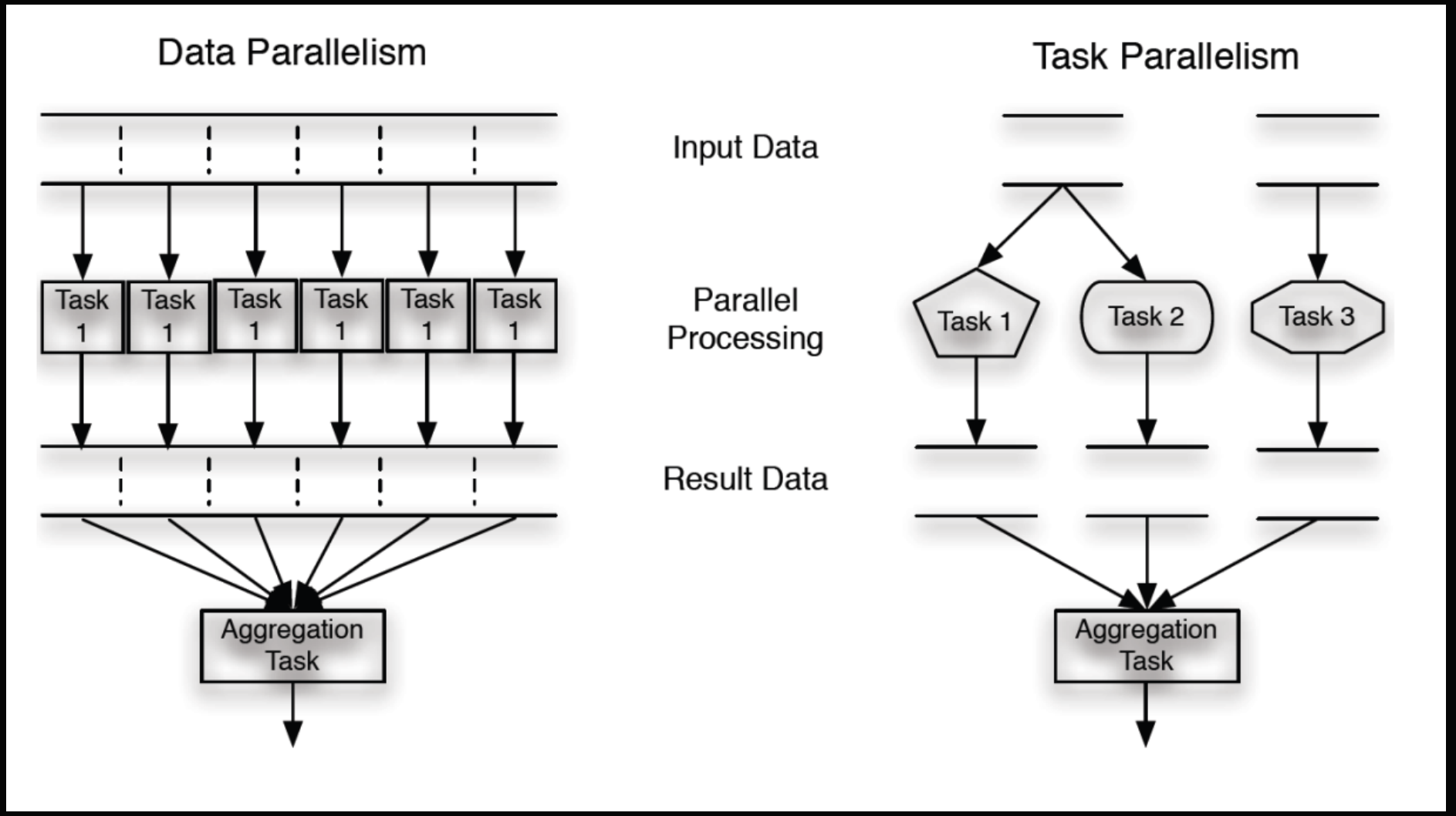
To take full advantage of modern computing architectures, we need to think about building applications from algorithms that are capable of exploiting parallelism opportunities that speed up the application and allow it to use less memory

What does all this mean?

We have to ”think parallel” when writing code

We should adopt parallel capable libraries to help us do that

Task Parallelism and Data parallelism examples



https://medium.com/@geminae.stellae/introduction-to-parallel-computing-with-opencl-2ee91c30b8b6

Programming considerations

- Tasks need to be able to run in parallel
- Aggregation tasks are potential bottlenecks
- Our approach may be different depending on if our application aims to minimize “time-to-insight” or to minimize “cost per event”

Amdahl's "Law": (P)arallel code fraction determines how much faster your application can go using N cores/threads

Amdahl's law gives the theoretical speed up of a code when running on N processors, given the fraction of the code that is parallel, P:

$$S = \frac{1}{(1 - P) + (P/N)}$$

To use a 64 core CPU at high efficiency, how much of your application must be parallel?

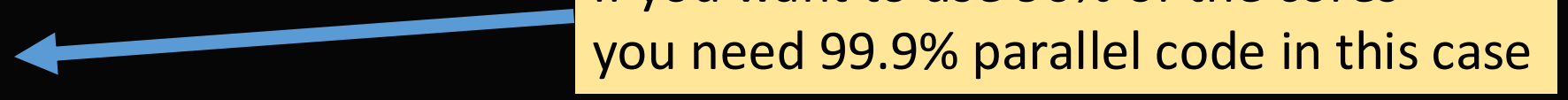
a) 50%

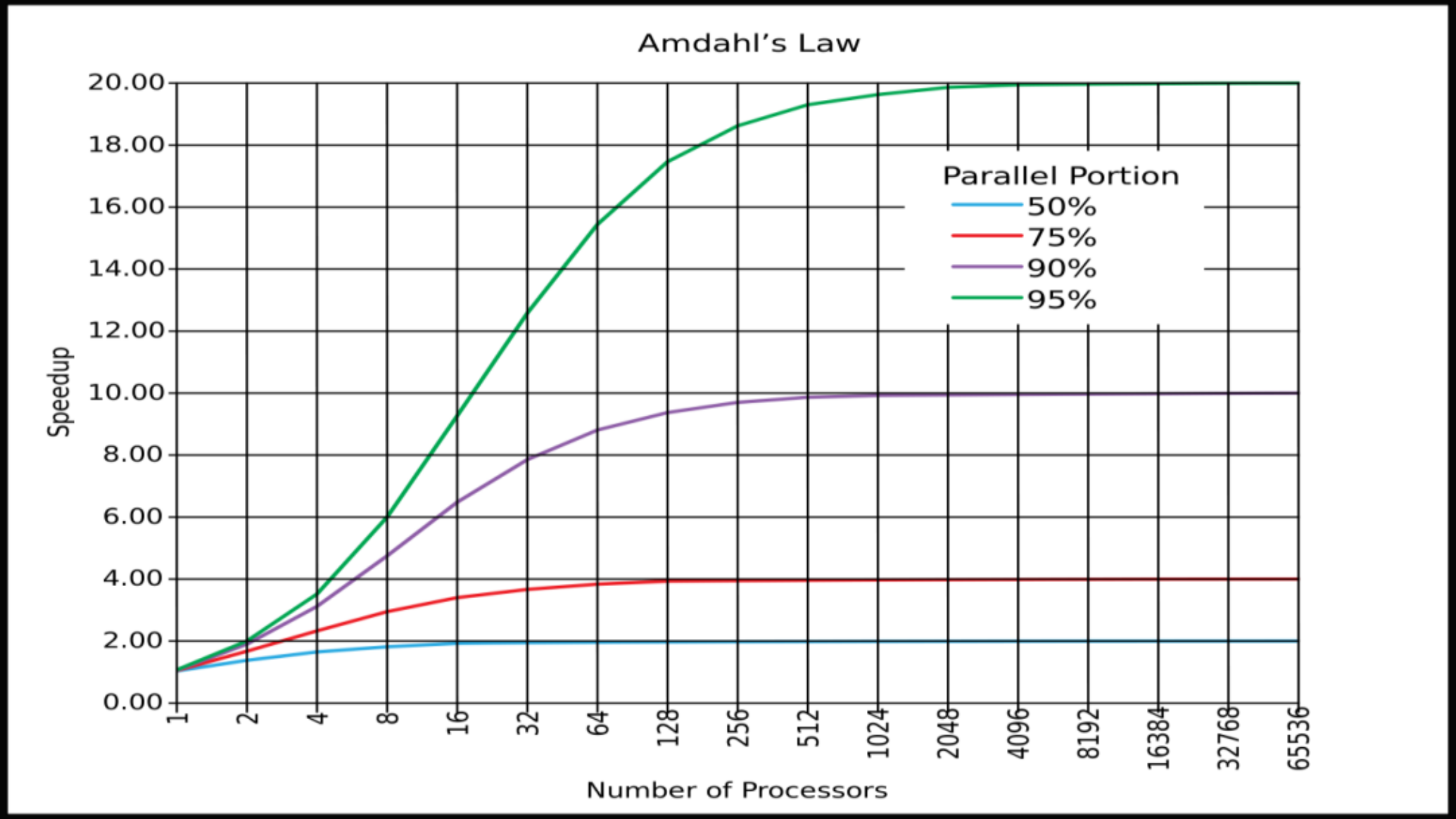
b) 90%

c) 99%

d) 99.9%

e) 99.99%




Amdahl's Law
Speedup
20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
Parallel Portion
50%
75%
90%
95%
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of Processors

Heterogeneous computing

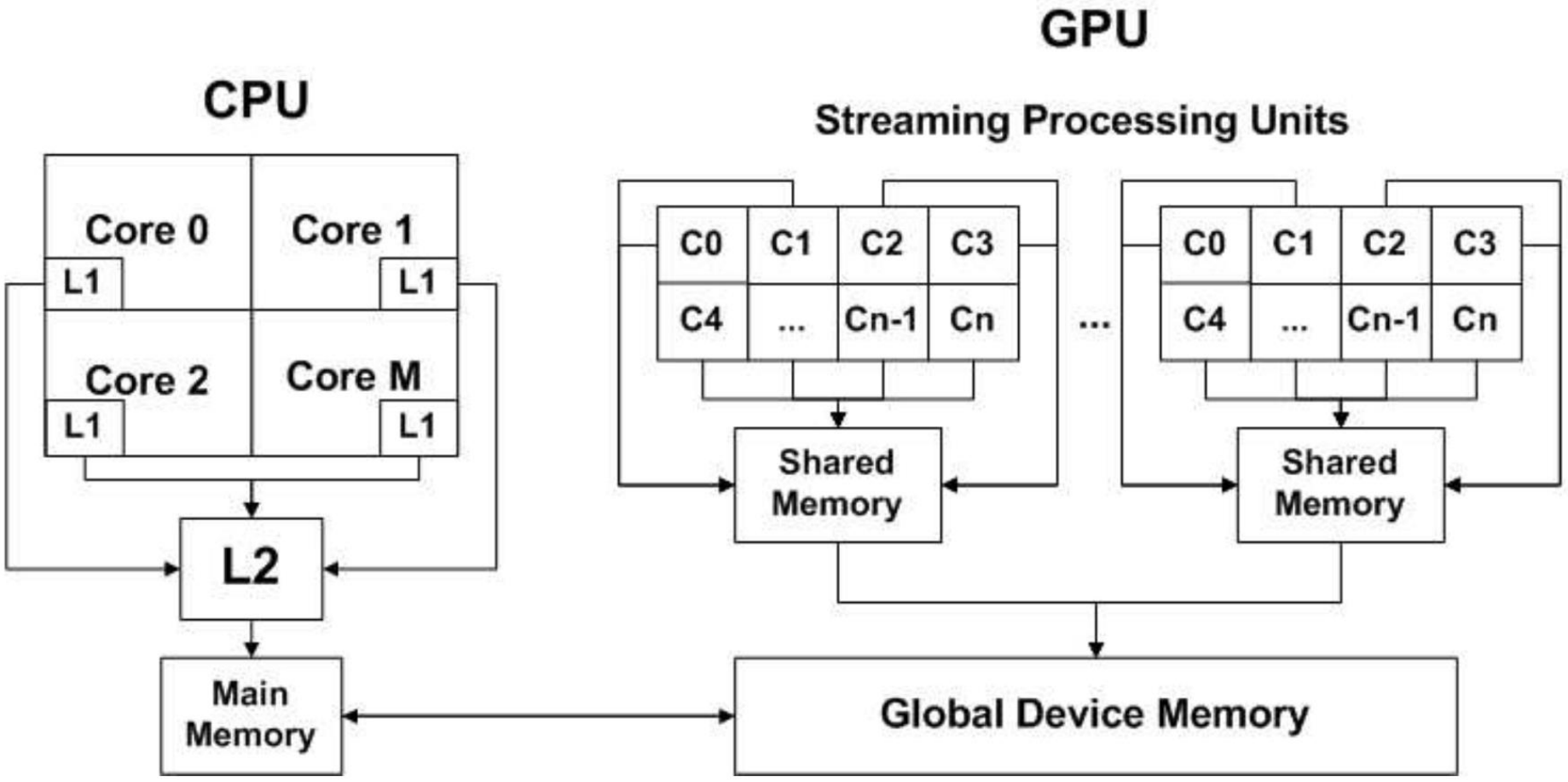


Figure 1: A Popular Heterogeneous Computing Platform

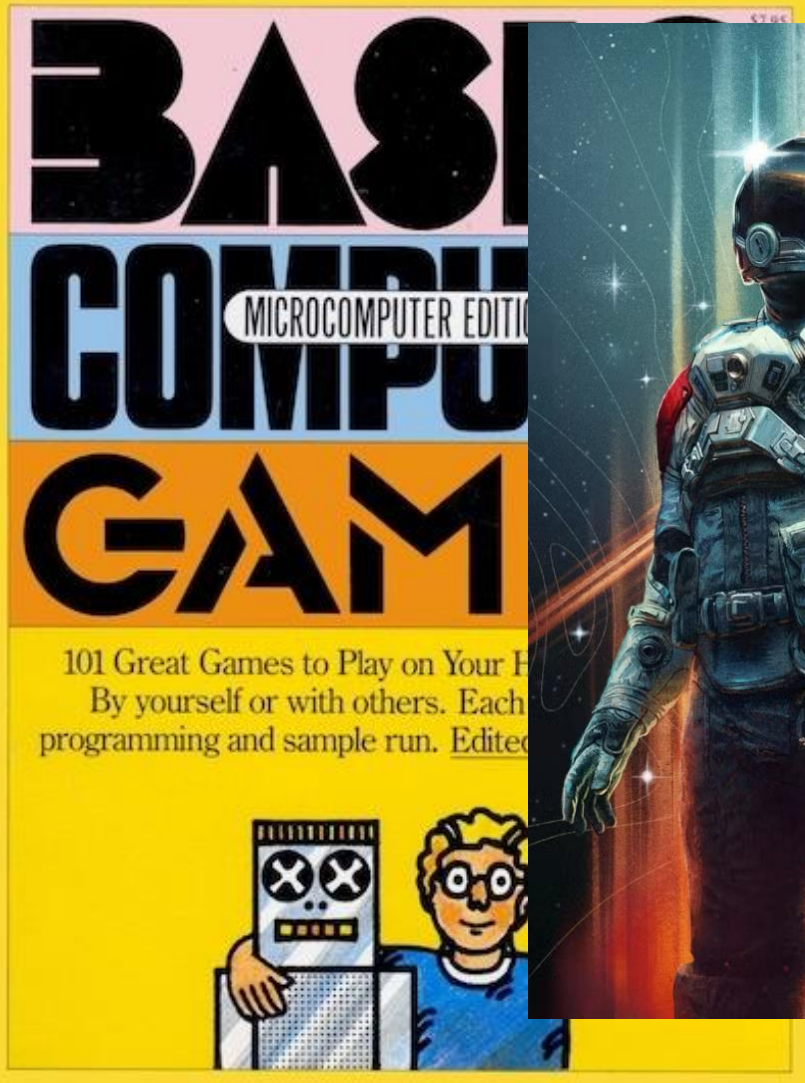
What is a GPU? Graphics Processing Units

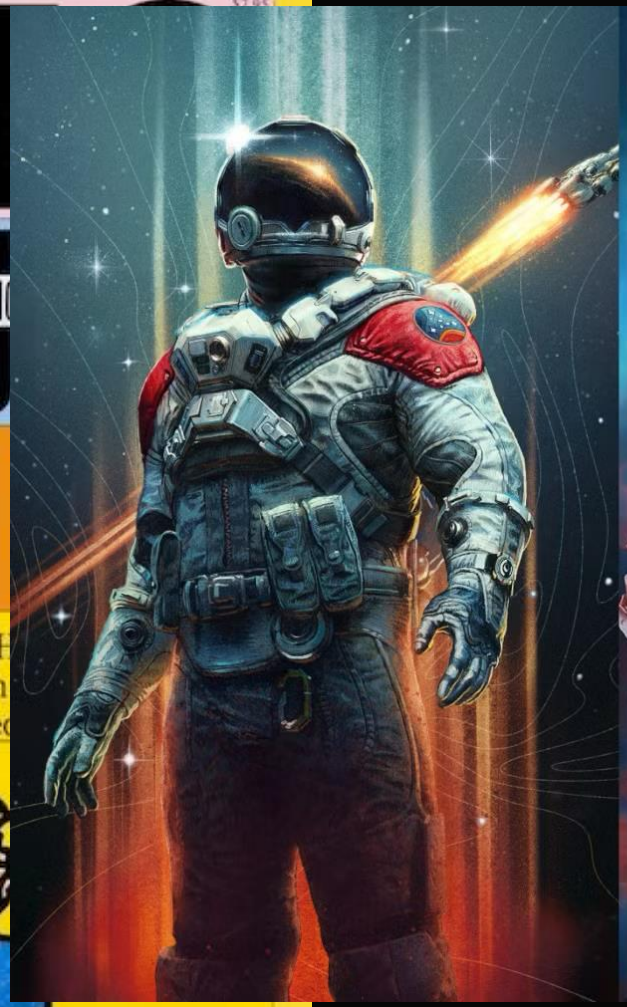
- The processor inside the graphics cards was purpose-specific for graphics only.

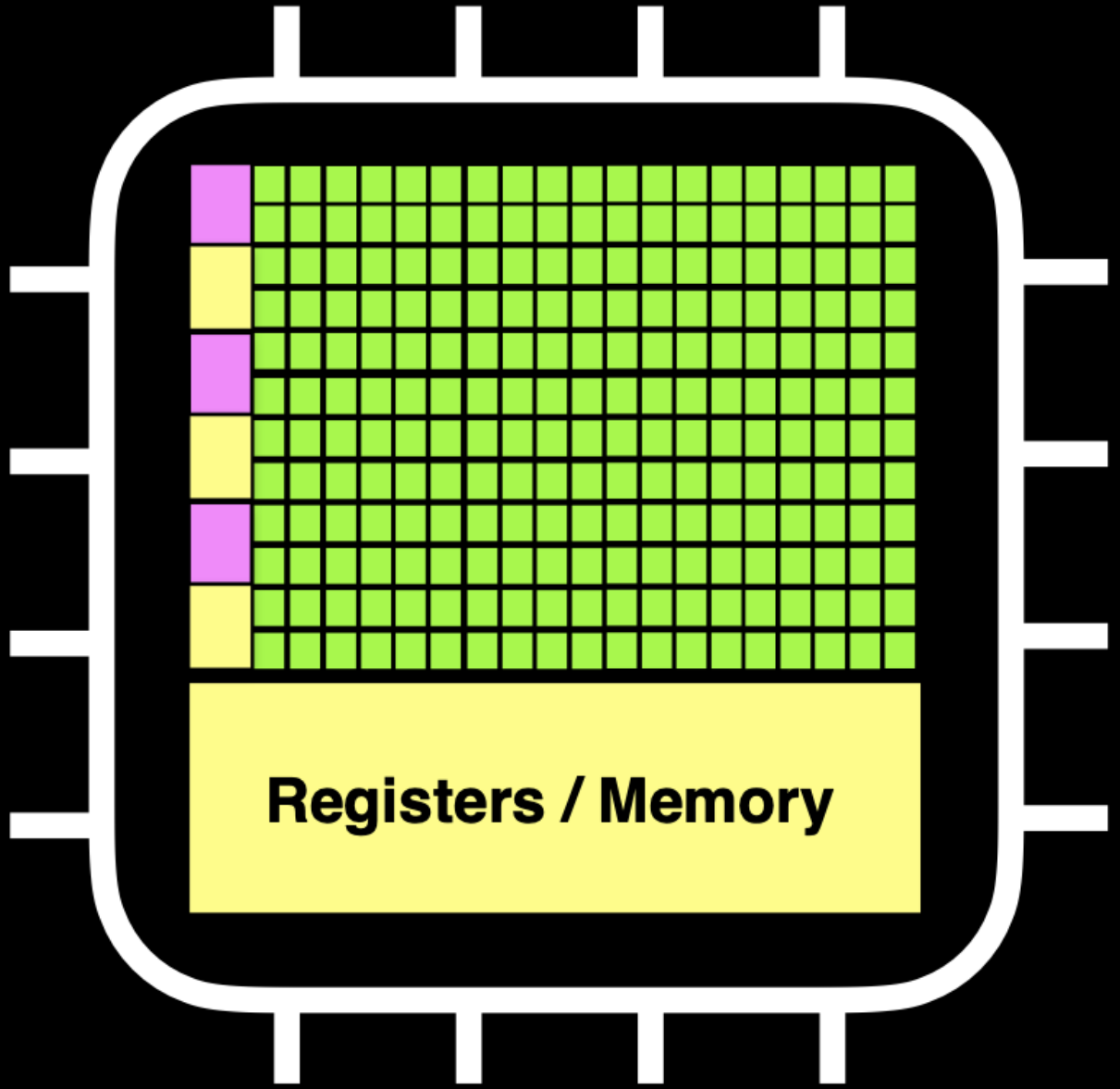
- GPGPU: Programming techniques to enable the GPU as a parallel co-processor.

- Currently: Intuitive programming and high number of processors (e.g. over 16000 cuda cores in RTX4090).

Wait aren’t GPUs for gaming?







Arithmetic Logic Unit (ALU):

To perform arithmetic and logic operations

Registers / Memory:

Fast memory for Input and output of ALUs

Control Unit:

Directs operation of the processor; moving memory, executing ALUs, etc.

GPU devotes more silicons to computing

Flynns classification of computer architecture

- Based on the number of instruction and data streams that can be processed simultaneously, computing systems are classified into four major categories

i.e. single cpu-core machine

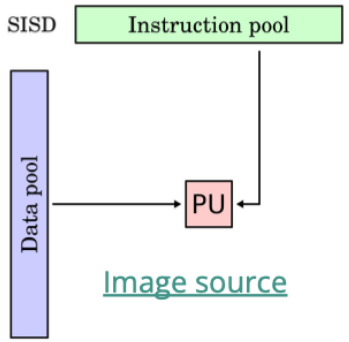


Image source

i.e pipeline architectures - not commonly used

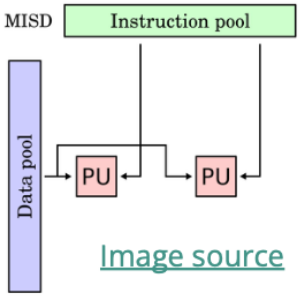
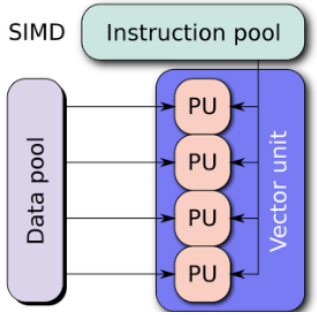


Image source

Data streams	Instruction streams: Single	Instruction streams: Many
Single	**SISD** (**S**ingle **I**nstruction **S**ingle **D**ata)	**MISD** (**M**any **I**nstructions **S**ingle **D**ata)
Many	**SIMD** (**S**ingle **I**nstruction **M**any **D**ata)	**MIMD** (**M**any **I**nstructions **M**any **D**ata)



i.e. vector processor cpus

i.e multi cpu-core machines / grid computing etc.

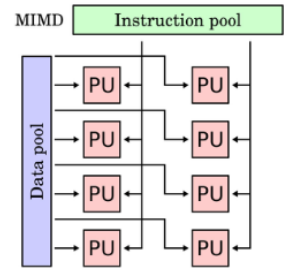
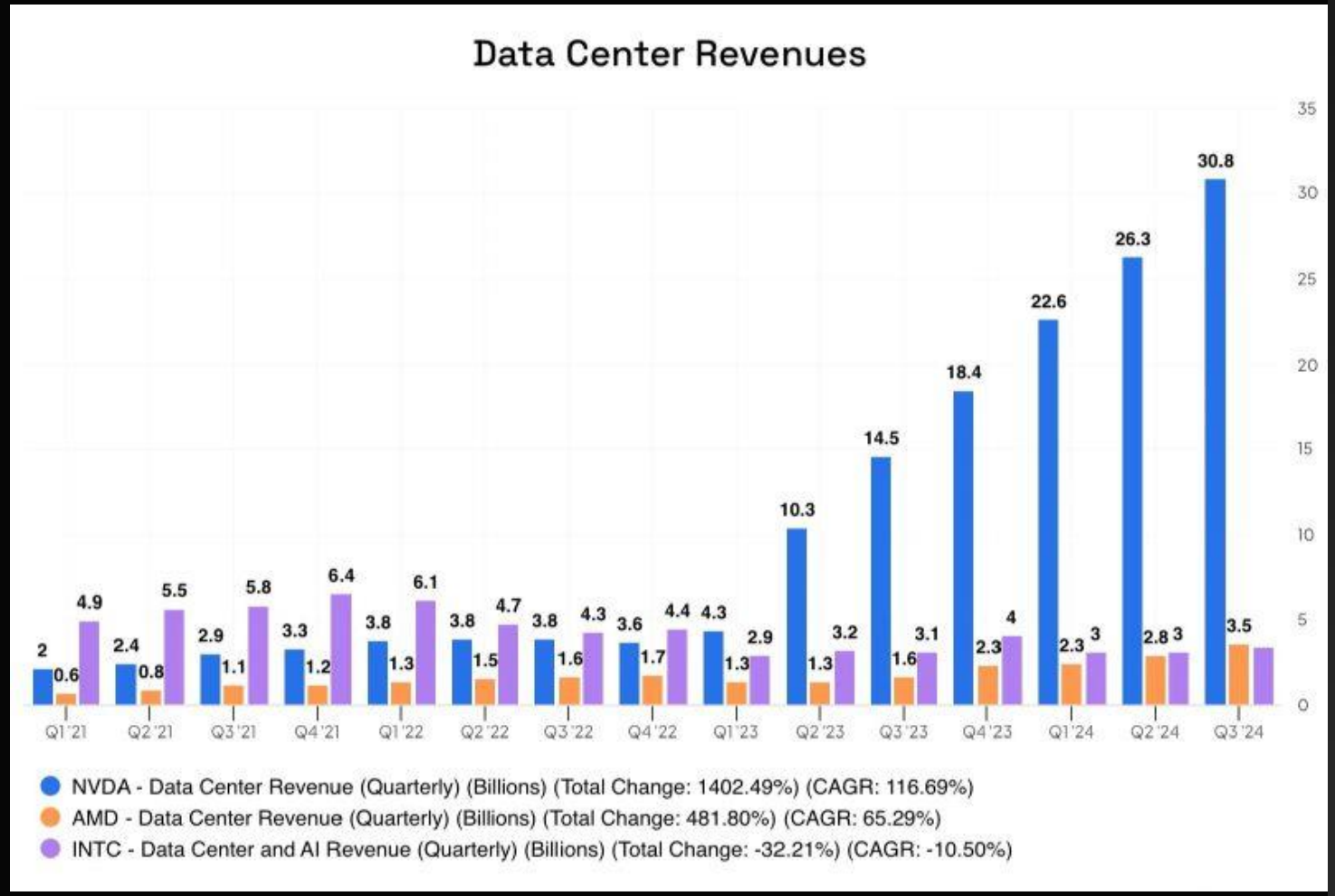


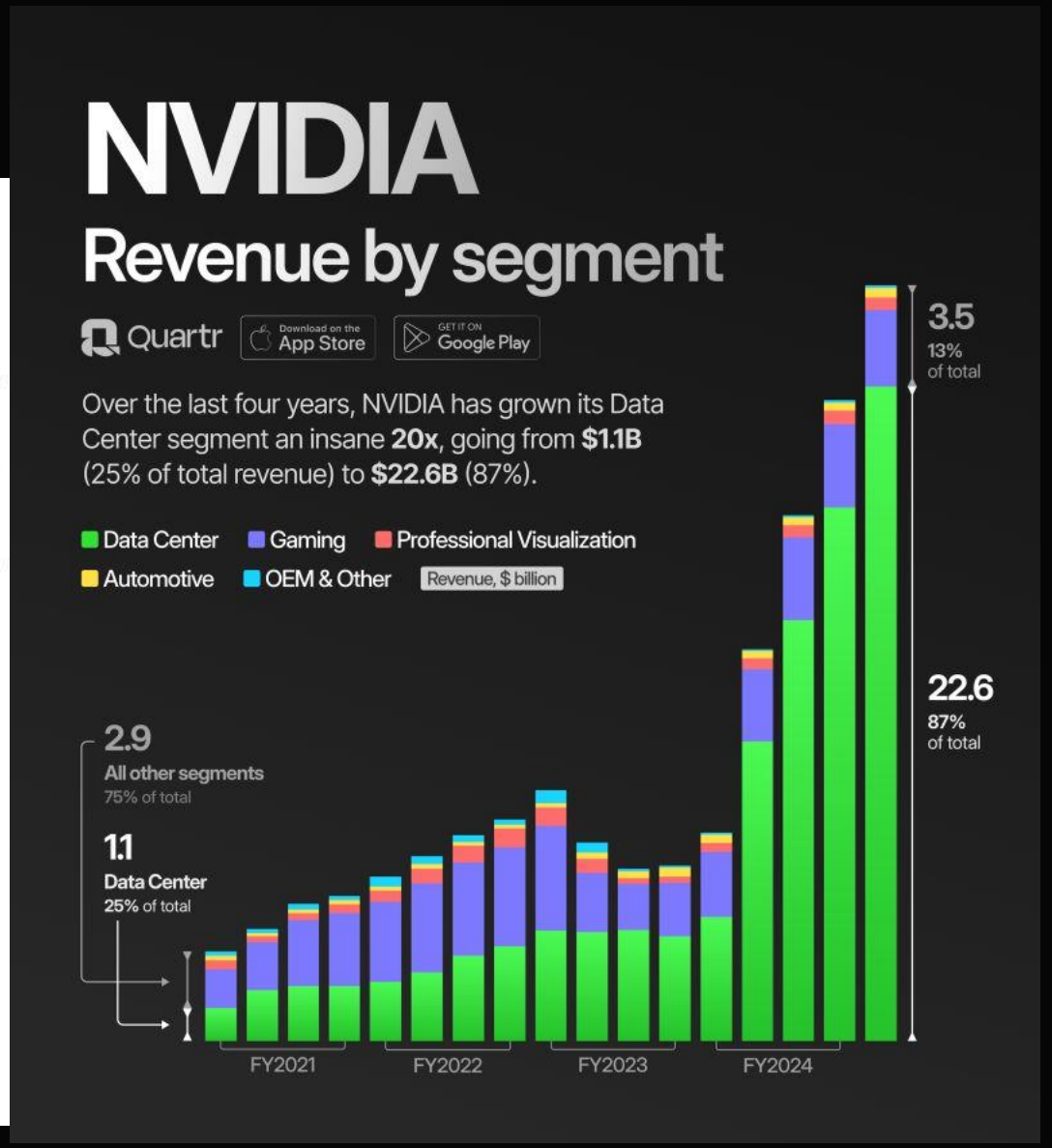
Image source

Other types of hardware accelerators

- FPGA (Field Programmable Gate Array)
  - FPGAs can be reprogrammed to suit the needs of an application or required functionality
  - More about FPGAs later this week
- ASIC: Integrated circuit customized for a specific purpose/application
- TPU: Optimized to perform matrix-multiplication operations

GPUs have risen together with AI



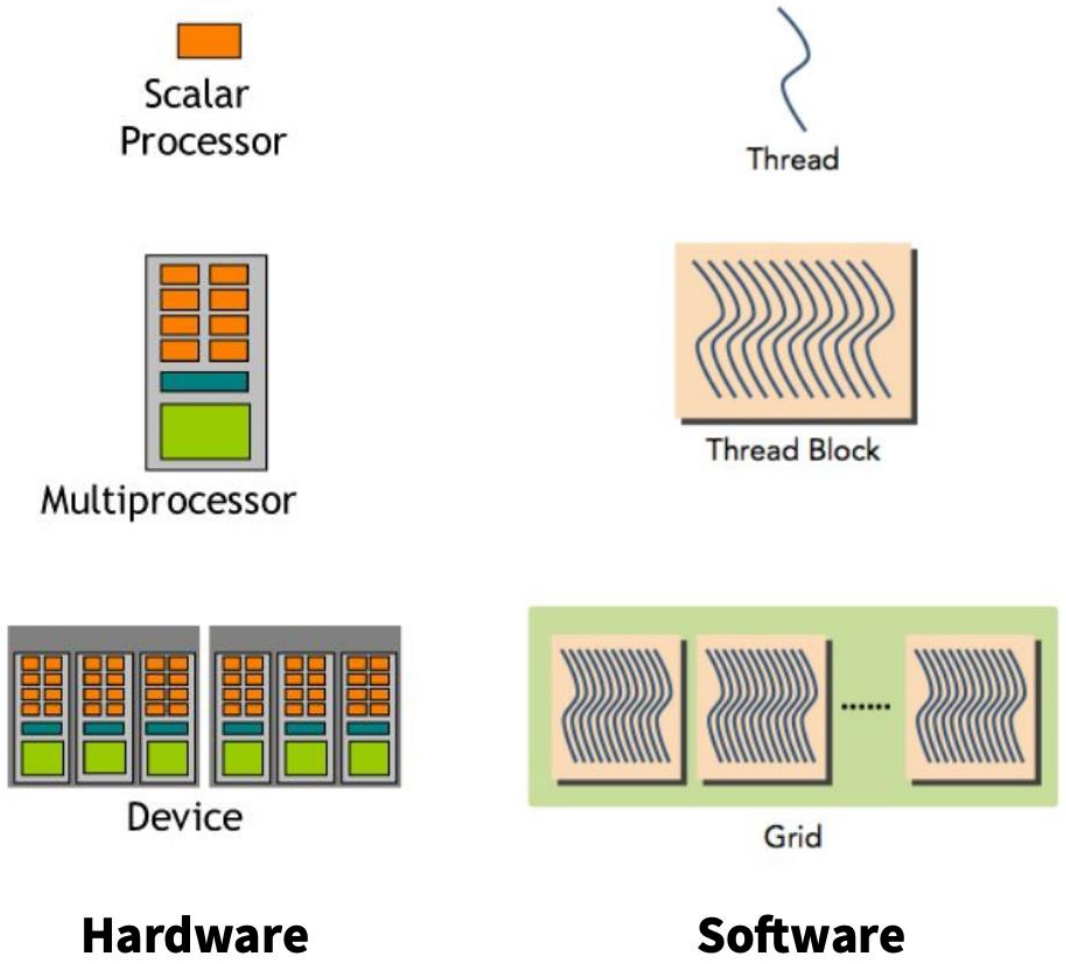


How does a GPU differ from a many-core CPU?

- CPU: Follows the Single instruction, multiple threads (SIMT) execution model. Asynchronous programming model where threads are executed independently

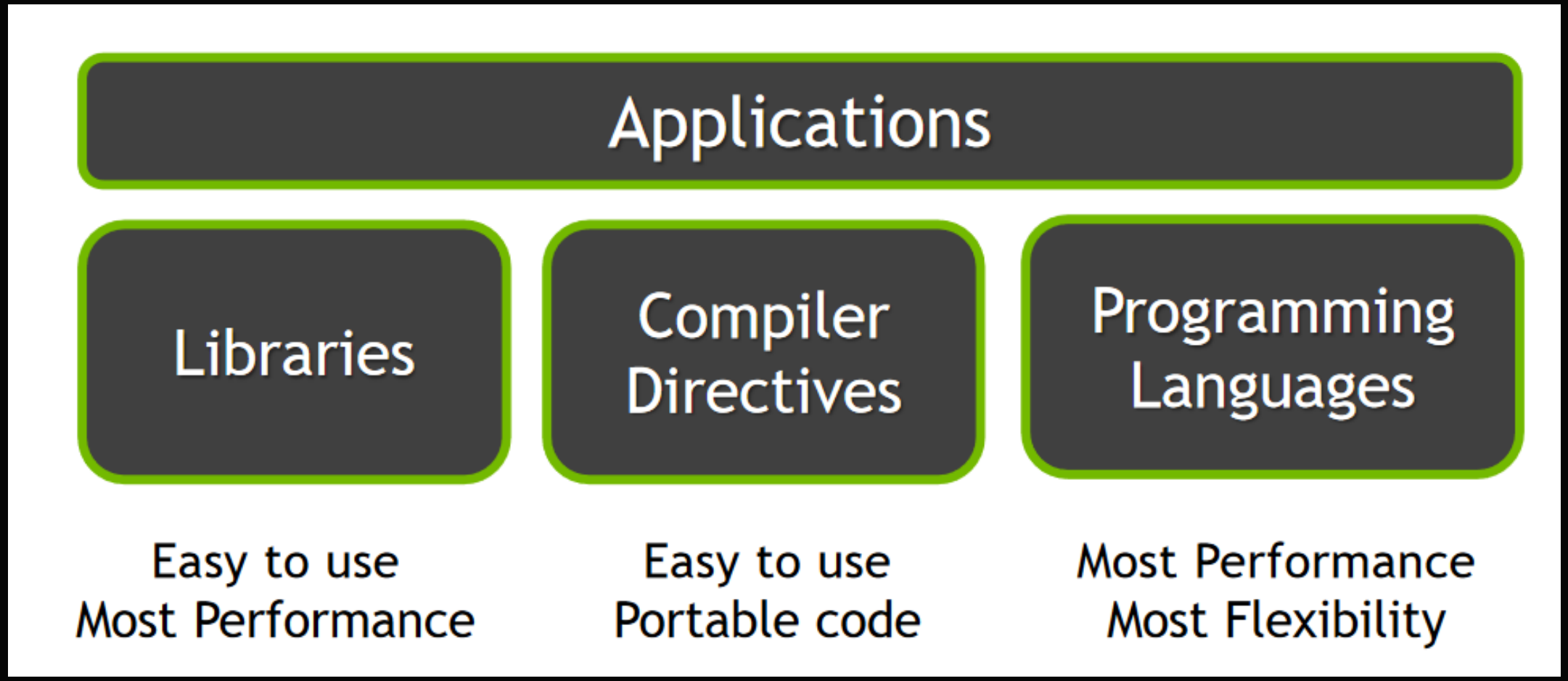
- GPU: High data throughput and massive parallel computing: a GPU consist of hundreds of cores performing the same operation on multiple data items in parallel.
  - GPU architectures are built around a scalable array of Streaming Multiprocessors (SM). Each SM in a GPU is designed to support concurrent execution of hundreds of threads
  - Lower clock speeds, higher latencies, higher throughput, lower power per performance

Hardware to software



- **A scalar processor or CUDA core is equivalent to a software thread**
- Scalar processors are grouped into a SM
- Each execution of a GPU function is done concurrently on a number of threads referred to as a **thread block**
- Each thread block is executed by one SM and cannot be migrated to other SMs in GPU
- The **set of thread blocks** executing the GPU function is called a **grid**.
- In CUDA terminology the GPU is referred to as the device

Software for accelerator hardware



GPU Accelerated Libraries

Linear Algebra
FFT, BLAS,
SPARSE, Matrix

cuLA|tools

MAGMA

CUSP

Numerical & Math
RAND, Statistics

ArrayFire

Data Struct. & AI
Sort, Scan, Zero Sum

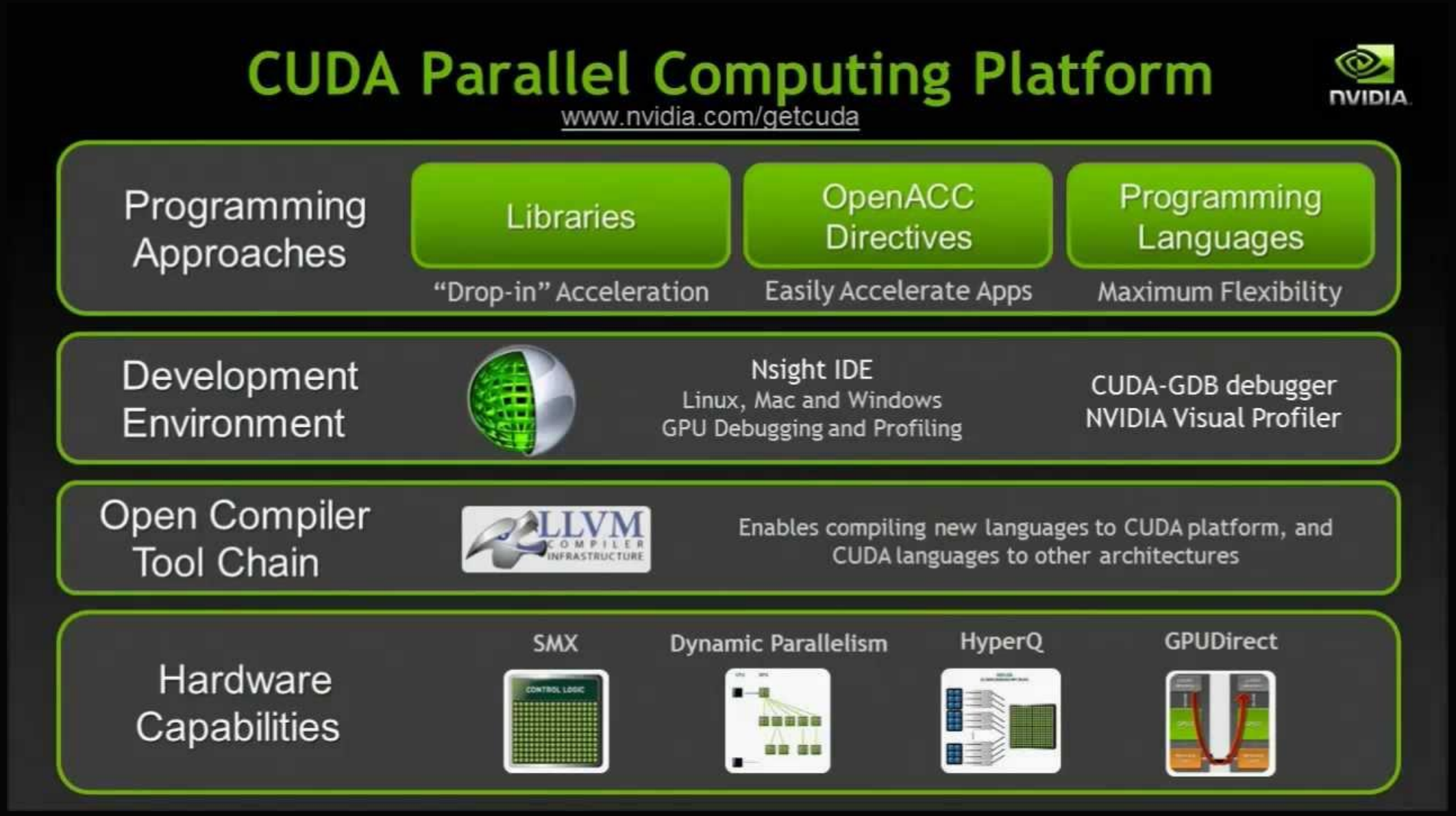
cuDNN

Visual Processing
Image & Video

NVIDIA
Video
Encode

Sundog Software

Nvidia GPU programming: CUDA



To effectively use such a high thread count, “kernels” must be identified and sent to the accelerator to compute

- The kernel expresses the portion of the application that is parallelizable
  - On a NVIDIA GPU: it will be executed in parallel by different CUDA threads

- Linear algebra is the classic example
  - Large arrays allocated in memory shared across multiple (many) cores/threads simultaneously
  - Each thread operates on a smaller portion of these arrays
  - Avoid conflicts and synchronization points between threads
    - Some intermediate variables may be computed on multiple threads
    - Extra calculations can keep threads synchronized

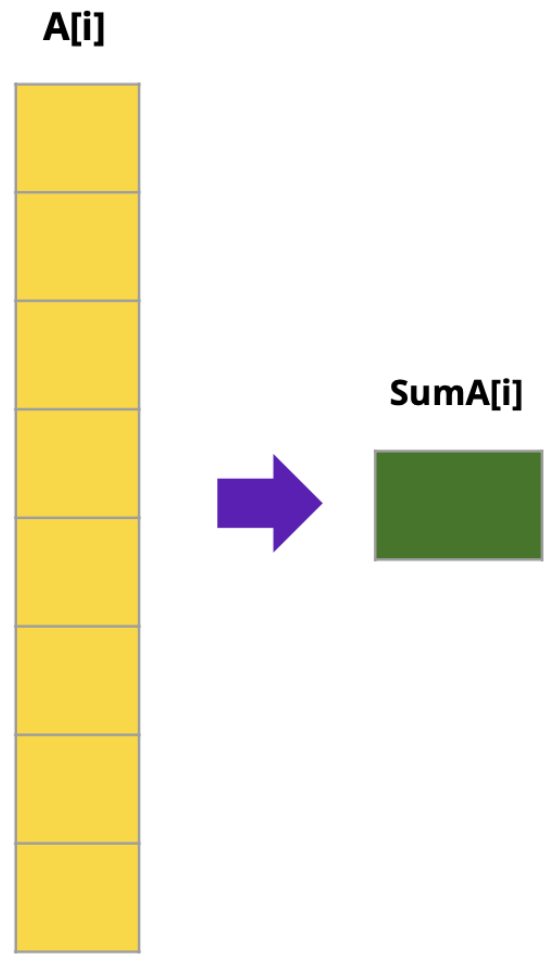
A simple CUDA example?

Adding elements in a vector

Let's start by writing a CUDA kernel that calculated the sum of the elements of a vector :

```
__global__ void add_array(float* A, float* sum) {
   int idx = threadIdx.x + blockIdx.x * blockDim.x;
   if (idx < N) {
        *sum +=A[idx];
   }
}
```

- There are 3 instructions that will be executed :
  - **Load** the value of A for each thread
  - **Read** the value of c
  - **Modify** the value of c



What goes wrong here?

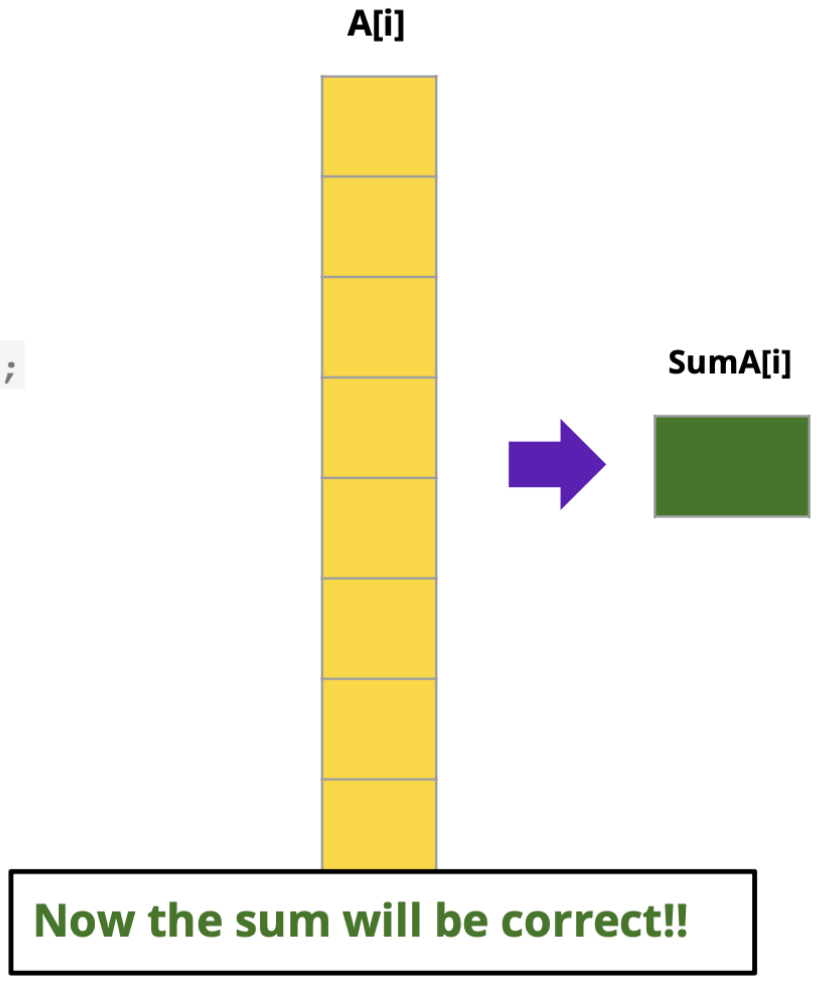
Extra care is needed any time a thread writes to shared memory

Adding elements in a vector

Using **atomicAdd** to sum the vector elements :

```
__global__ void add_array(float* A, float* sum) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < N) {
        atomicAdd(sum,A[idx]);
    }
}
```

Each read-modify-write access cannot be interrupted



Programming model

- There is no magic compiler flag that makes your code run in parallel. So your approach really depends on what you are doing and where you intend to run.

- You must design from the start for parallelism

- I can’t be exhaustive today about how to do this… Let me talk about a few examples

Memory bandwidth and data movement

- To do a calculation on a GPU, all needed data must be copied to the GPU (eg, host to device transfer)
- GPU memory bandwidth is 100s or 1000s of GB/second. However this can be a limitation:
  - If your compute kernel is too small
  - If your application needs to frequently move data back and forth between the device and host.
- Common tricks:
  - Dimensionality reduction
  - Efficiently representations (eg, sparse matrices)
  - Building a series of kernels that can be performed without intermediate data transfer (even if one might be less efficient on a GPU by itself)

Which code results in more a performant parallel application? (independently of if a GPU, vector CPU, etc)

```
row = blockIdx.x*blockDim.x + threadIdx.x;
for (col=0; col<N; col++)
    output[row][col]=2*input[row][col];
```

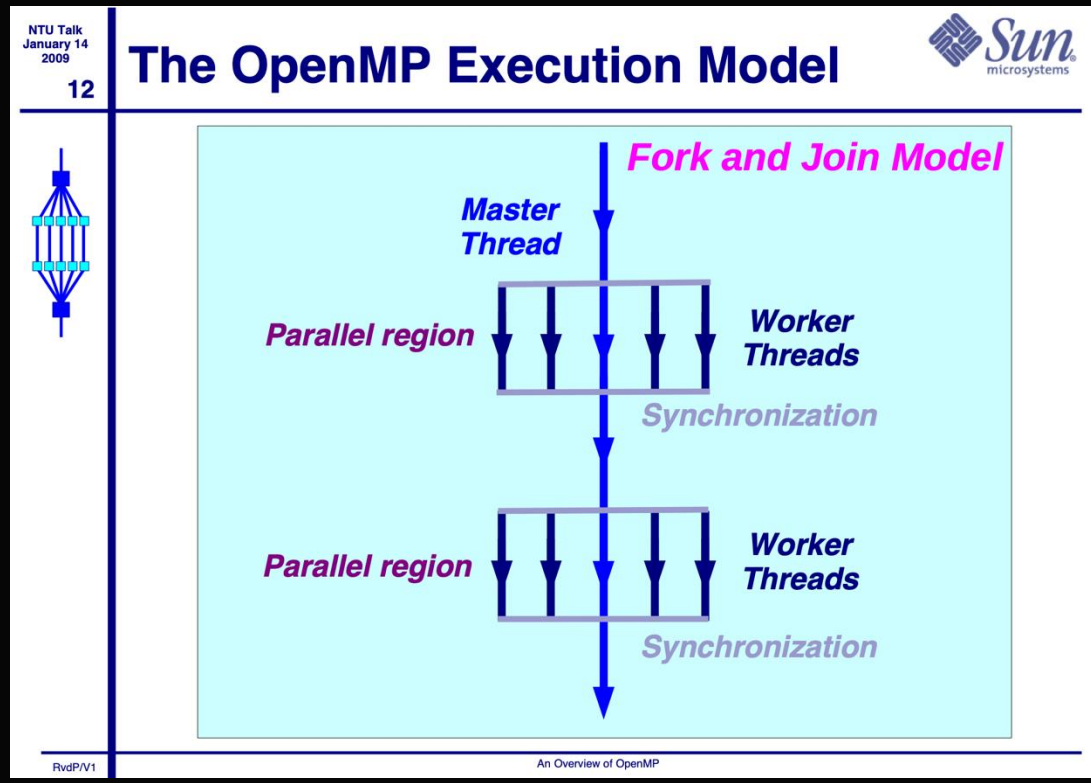
```
col = blockIdx.x*blockDim.x + threadIdx.x;
for (row=0; row<N; row++)
    output[row][col]=2*input[row][col];
```

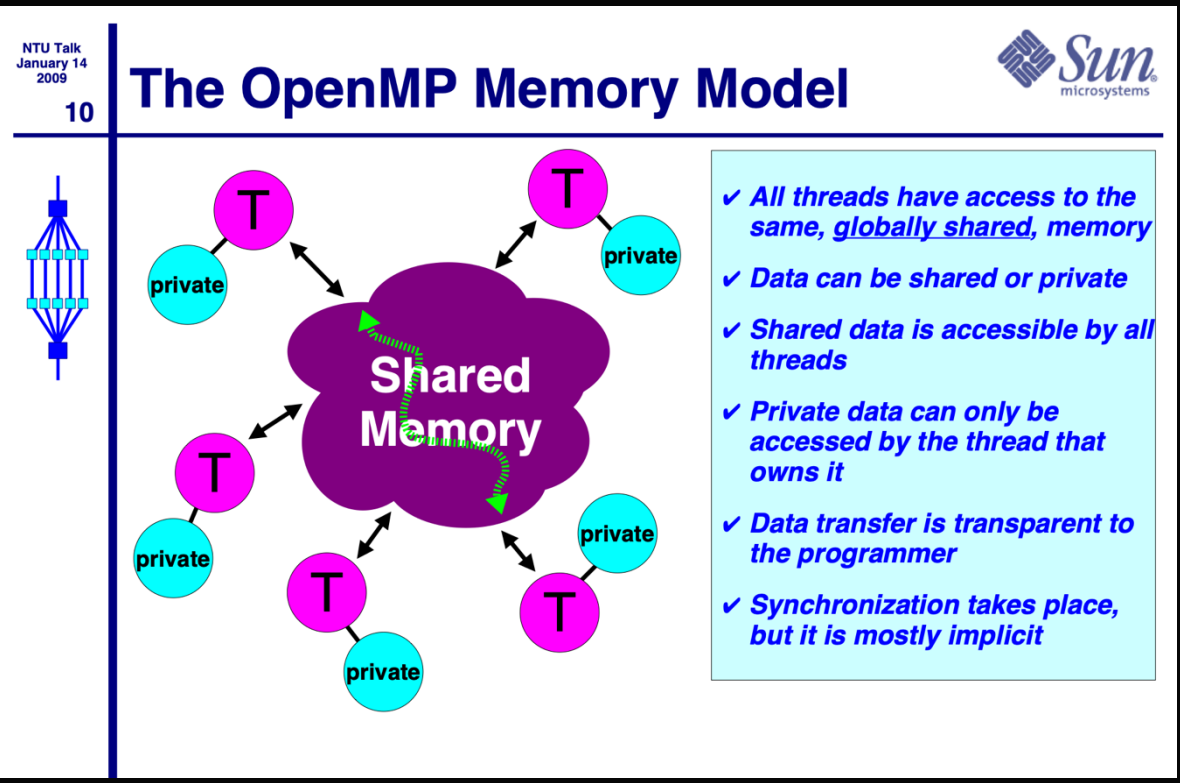
For GPUs:

- Maximum bandwidth is when data for multiple threads can be loaded in a single transaction
- This happens if data access patterns are ”close enough”. Eg, if 16 consecutive threads all access data from within the same memory segment.
- This condition is met when consecutive threads read consecutive memory addresses. If not, memory accesses are serialized, significantly degrading performance.

Tools and Libraries: OpenMP

- The OpenMP API supports multi-platform shared-memory parallel programming in C/C++ and Fortran. The OpenMP API defines a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer. (https://www.openmp.org)





Tools and Libraries: std::thread

```
// thread example
#include <iostream> // std::cout
#include <thread> // std::thread

void foo() { // do stuff... }
void bar(int x) { // do stuff... }

int main() {
        std::thread first (foo); // spawn new thread that calls foo()
        std::thread second (bar,0); // spawn new thread that calls bar(0)
        std::cout << "main, foo and bar now execute concurrently...\n";

        // synchronize threads:
        first.join(); // pauses until first finishes
        second.join(); // pauses until second finishes

        std::cout << "foo and bar completed.\n";
        return 0;
 }
```

Similarly in python….

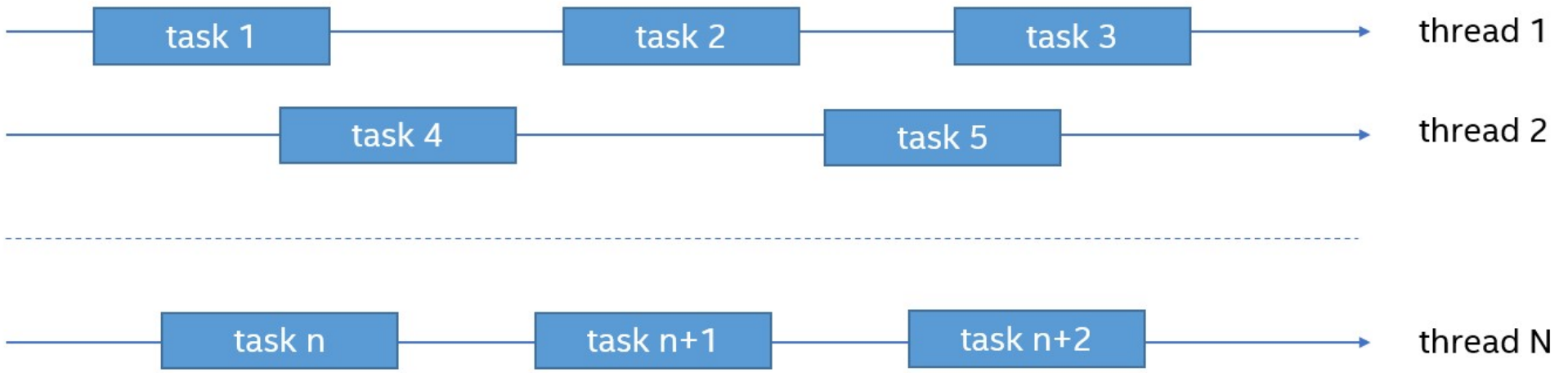
```
import threading
def print_cube(num):
    print("Cube: {}" .format(num * num * num))
def print_square(num):
    print("Square: {}" .format(num * num))

if __name__ =="__main__":
    t1 = threading.Thread(target=print_square, args=(10,))
    t2 = threading.Thread(target=print_cube, args=(10,))
    t1.start()
    t2.start()
    t1.join()
    t2.join()
    print("Done!")
```

Tools and Libraries: TBB

TBB is well suited for scalable applications that:

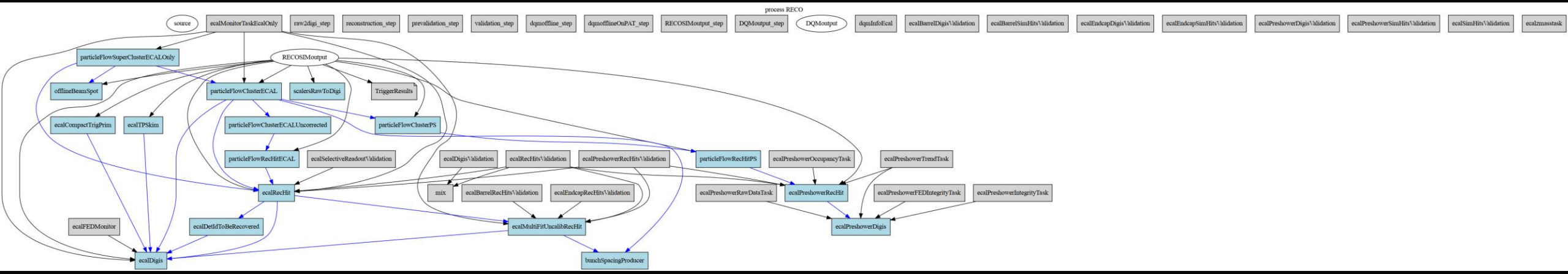
- **Specify logical parallel structure instead of threads**.
- Emphasize data-parallel programming.
- Take advantage of concurrent collections and parallel algorithms.

https://github.com/uxlfoundation/oneTBB



The CMS multi-threaded framework uses TBB to schedule 1000s of tasks per event and to allow multiple events to be processed concurrently

- Dependency (task) graph from a small set of CMSSW algorithms



Does heterogeneity mean I need to write code for each different architecture?

- Maybe…but doesn’t each GPU vendor has their own API..

- Fortunately, there are a number of approaches that aim for performance portability. Eg, to enable portability across accelerators through the abstraction of the underlying levels of parallelism without giving much (too much) performance on any given backend

Example: Alpaka

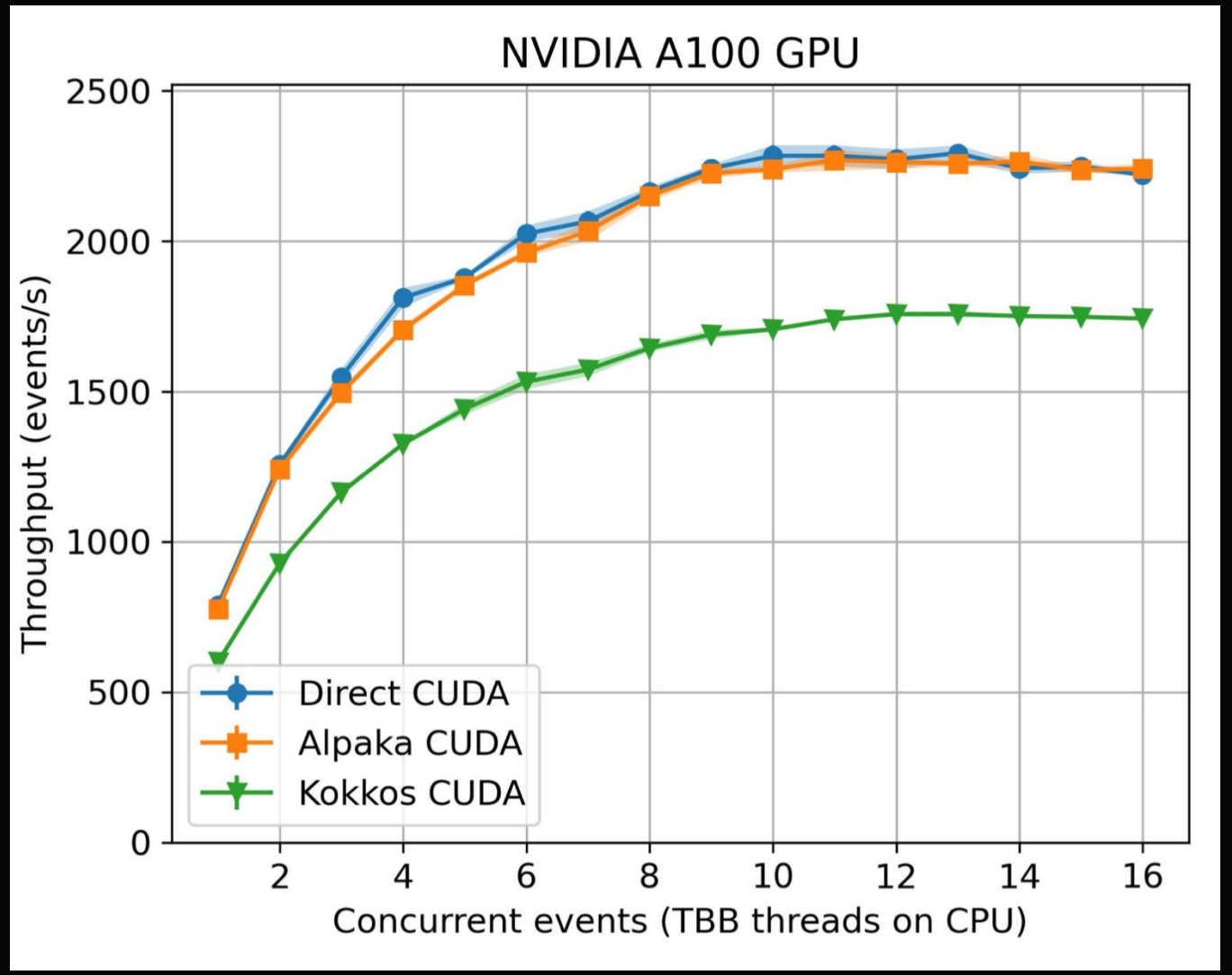
- The **alpaka** library is a header-only C++20 abstraction library for accelerator development. https://alpaka-group.github.io/alpaka/

Accelerator Back-ends

Accelerator Back-end	Lib/API	Devices	Execution strategy grid-blocks	Execution strategy block-threads
Serial	n/a	Host CPU (single core)	sequential	sequential (only 1 thread per block)
OpenMP 2.0+ blocks	OpenMP 2.0+	Host CPU (multi core)	parallel (preemptive multitasking)	sequential (only 1 thread per block)
OpenMP 2.0+ threads	OpenMP 2.0+	Host CPU (multi core)	sequential	parallel (preemptive multitasking)
std::thread	std::thread	Host CPU (multi core)	sequential	parallel (preemptive multitasking)
TBB	TBB 2.2+	Host CPU (multi core)	parallel (preemptive multitasking)	sequential (only 1 thread per block)
CUDA	CUDA 12.0+	NVIDIA GPUs	parallel (undefined)	parallel (lock-step within warps)
HIP(clang)	HIP 6.0+	AMD GPUs	parallel (undefined)	parallel (lock-step within warps)
SYCL(oneAPI)	oneAPI 2024.2+	CPUs, Intel GPUs and FPGAs	parallel (undefined)	parallel (lock-step within warps)

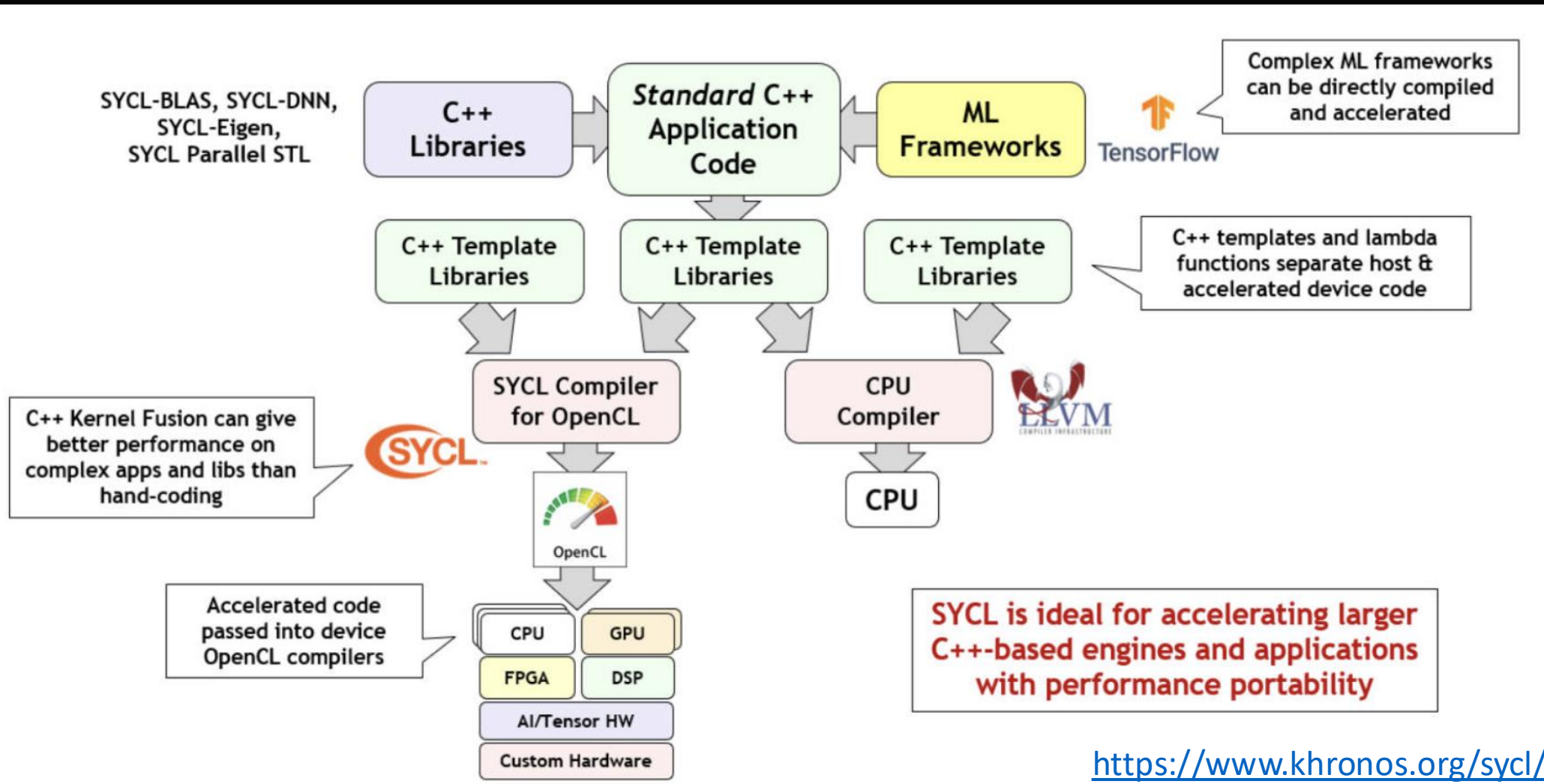
Alpaka performance examples

- CMS algorithm evaluation on a CPU comparing a CUDA implementation with an Alpaka implementation running on NVIDIA A100
- Essentially no loss of throughput seen (while enabling other backends with the same code)



CHEP presetnation

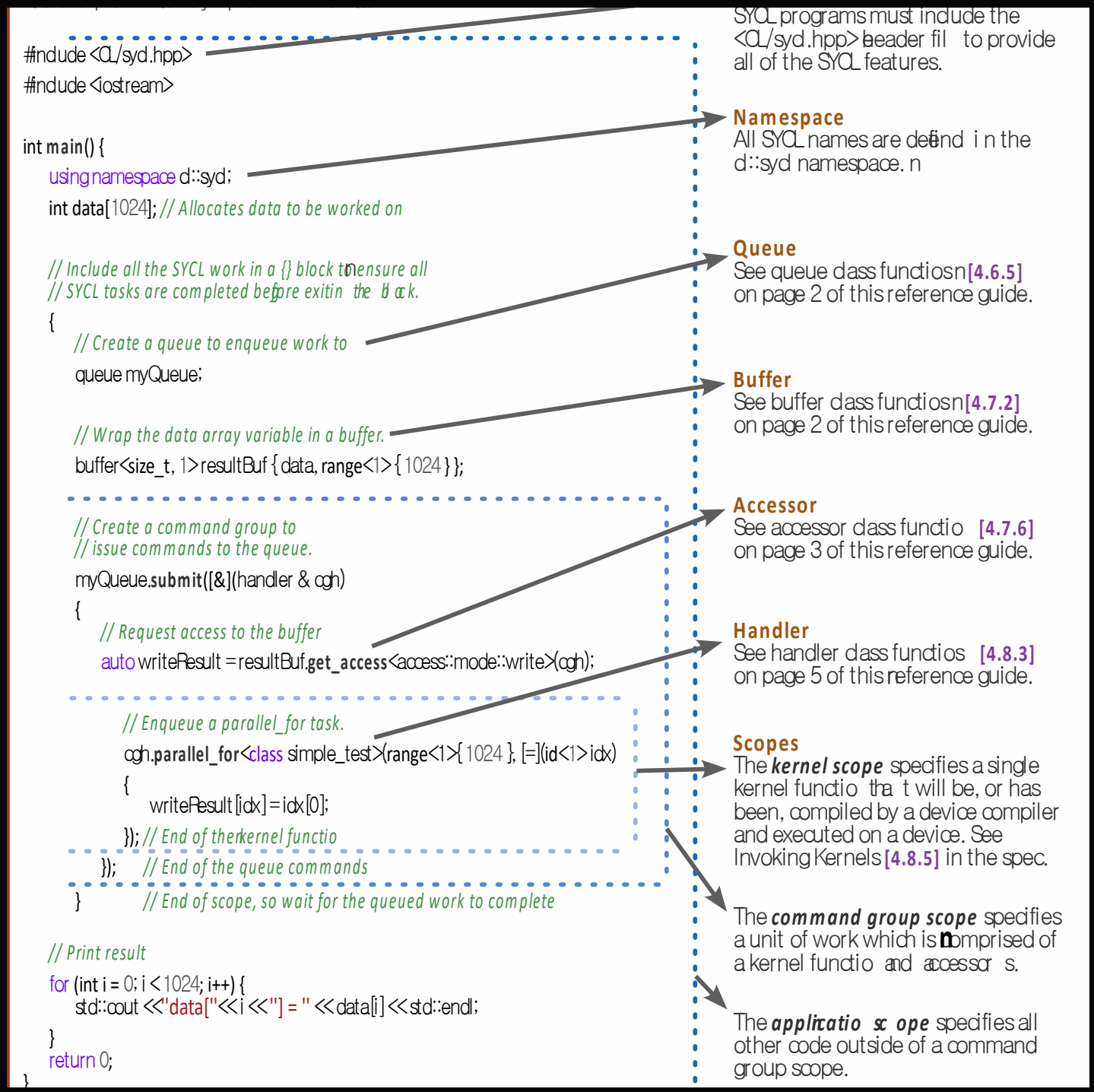
Example: SYCL



https://www.khronos.org/sycl/

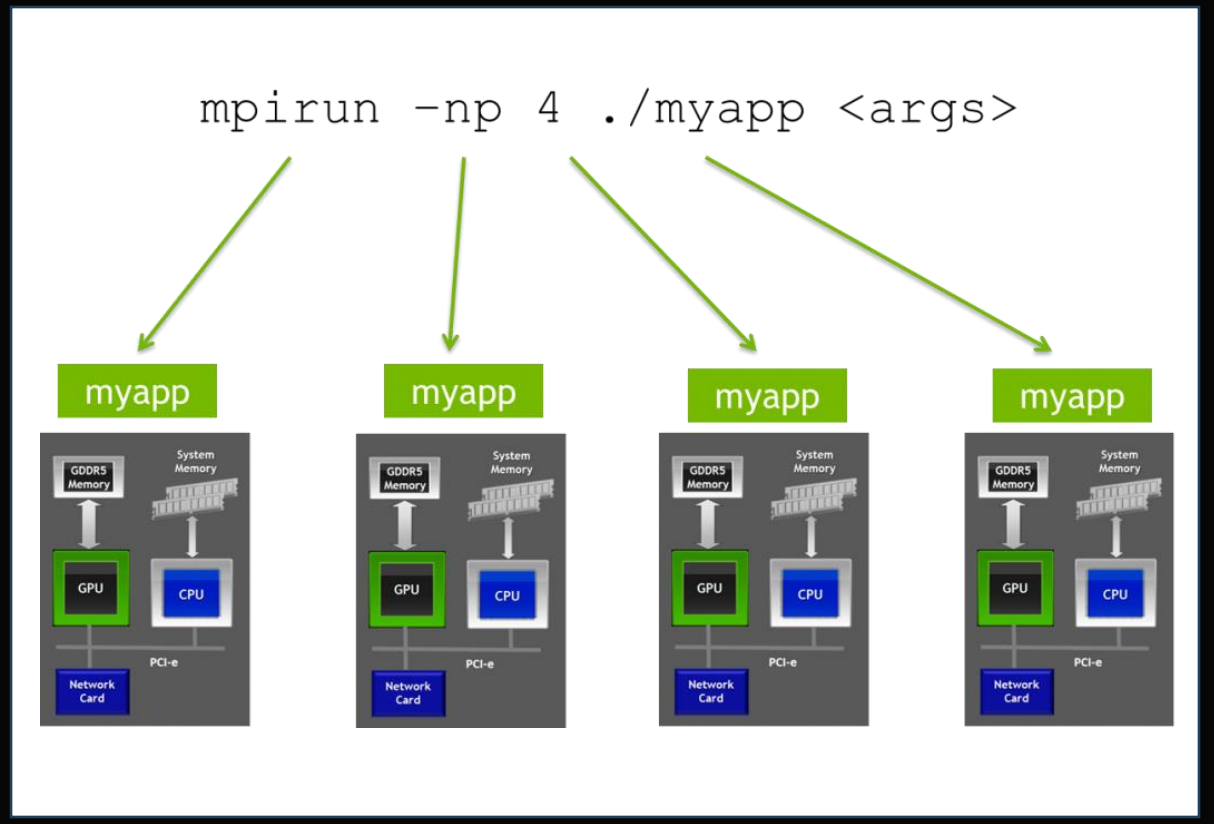
Typical Use of SYCL for Single Source C++ Parallel Programming

SYCL code example..

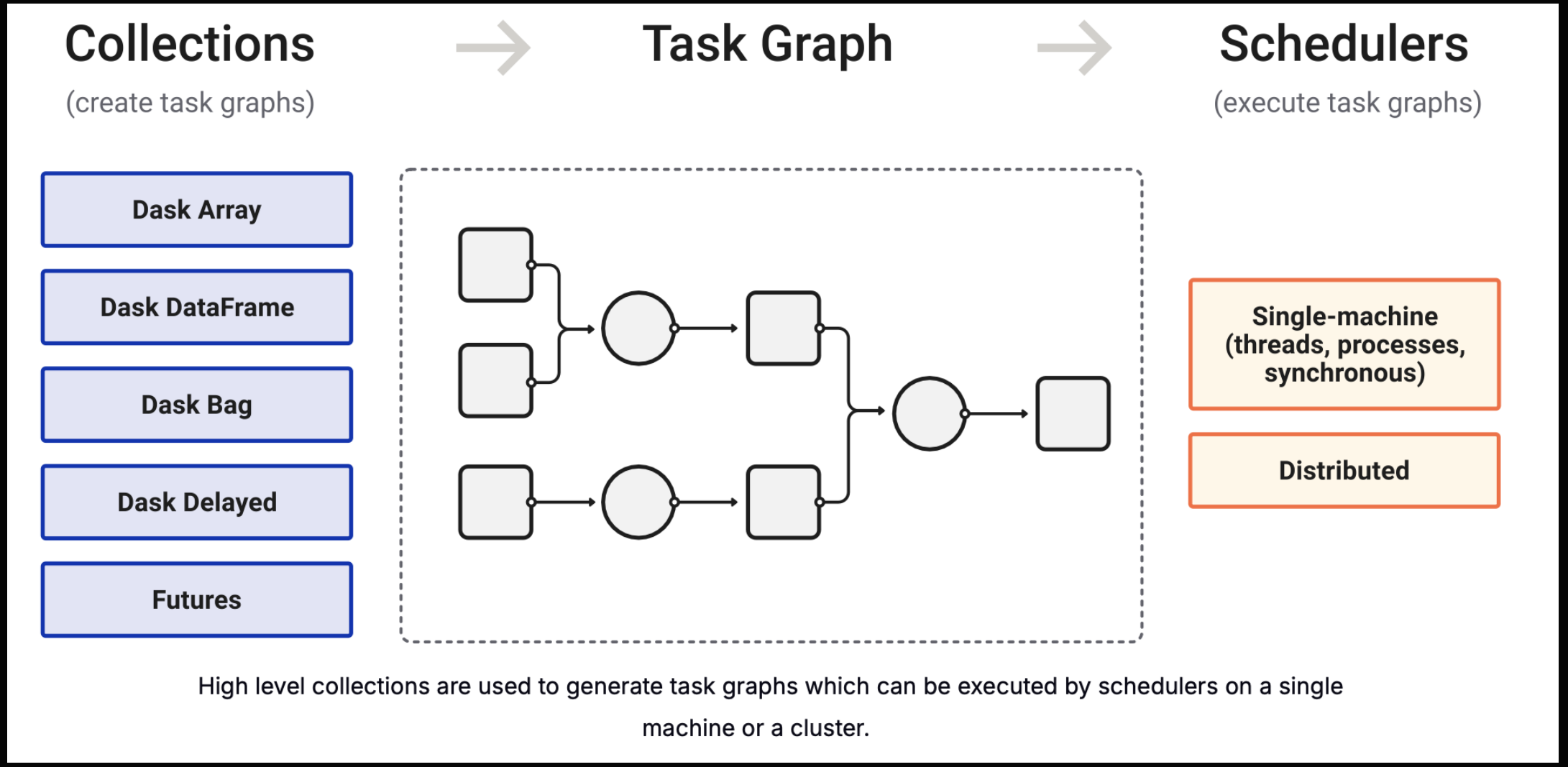


Libraries for scaling codes to large problems: MPI

- Message Passing Interface (MPI) is interface defining a model of parallel computing where each parallel process has its own local memory, and data must be explicitly shared by passing messages between processes.
- Using MPI allows programs to scale beyond the processors and shared memory of a single compute server, to the *distributed memory* and processors of multiple compute servers combined together.



Another library for horizontal scaling but for Python: Dask



High level collections are used to generate task graphs which can be executed by schedulers on a single machine or a cluster.

Scientific Python on GPUs

Python and GPU

Our introduction to array-oriented programming started by addressing Python's slowness (or the slowness of interactive environments in general).

But GPUs are also best addressed in an array-at-a-time fashion: the memory layout explicitly favors arrays over objects addressed by pointers or references (even in C++).

It should not be a surprise, then, that Python's GPU libraries are array libraries.

Python and ML

- Most users of GPUs in Python are using them for Machine Learning (ML).

- Since ML libraries are focused on ML, rather than programming in general, they provide an interface that is identical for CPUs and GPUs, usually by a device flag in the array object.
  - Switching from device="cpu" to device="cuda"simply speeds up the calculation by an order of magnitude with no other changes.

- In this lesson, we'll focus on GPU programming in which the differences from CPU programming are visible.

Most libraries specifically address CUDA, Nvidia's brand of GPU programming

- This reflects the very tight monopoly Nvidia has on GPUs for general-purpose programming.
  - CuPy has *experimental* support for AMD but Numba deprecated support for AMD due to lack of community interest.
  - There are Python projects that support OpenCL, Vulkan, Kokkos, and SYCL.

- Unfortunately, this excludes many laptops, including all MacOS: CUDA is something you'll more likely use on a data center than a laptop.

Installing a working version of CUDA can be hard, as it depends strongly on the operating system and GPU hardware

- The GPU driver must be installed in the operating system (Windows or Linux) and not with any Python package installers (pip, conda, uv, pixi). You might get it directly from Nvidia or through the operating system's package installer (such as apt for Ubuntu).
- The CUDA compiler and its libraries are distinct from the device driver, and they can be installed with conda. The story is more complicated for CUDA 11, but for CUDA 12, just include cuda-version=12 in the list of packages to install.
- Nvidia publishes a table of version compatibility between GPU hardware, drivers, and CUDA packages.

Lets have a look at our setup

Two useful shell commands:

- nvidia-smi is Nvidia’s tool to find out what hardware is installed and what it is doing
- Numba comes with a numba -s command line tool to print the versions of everything: incompatibilities between driver and CUDA package can be found here.

Programming model

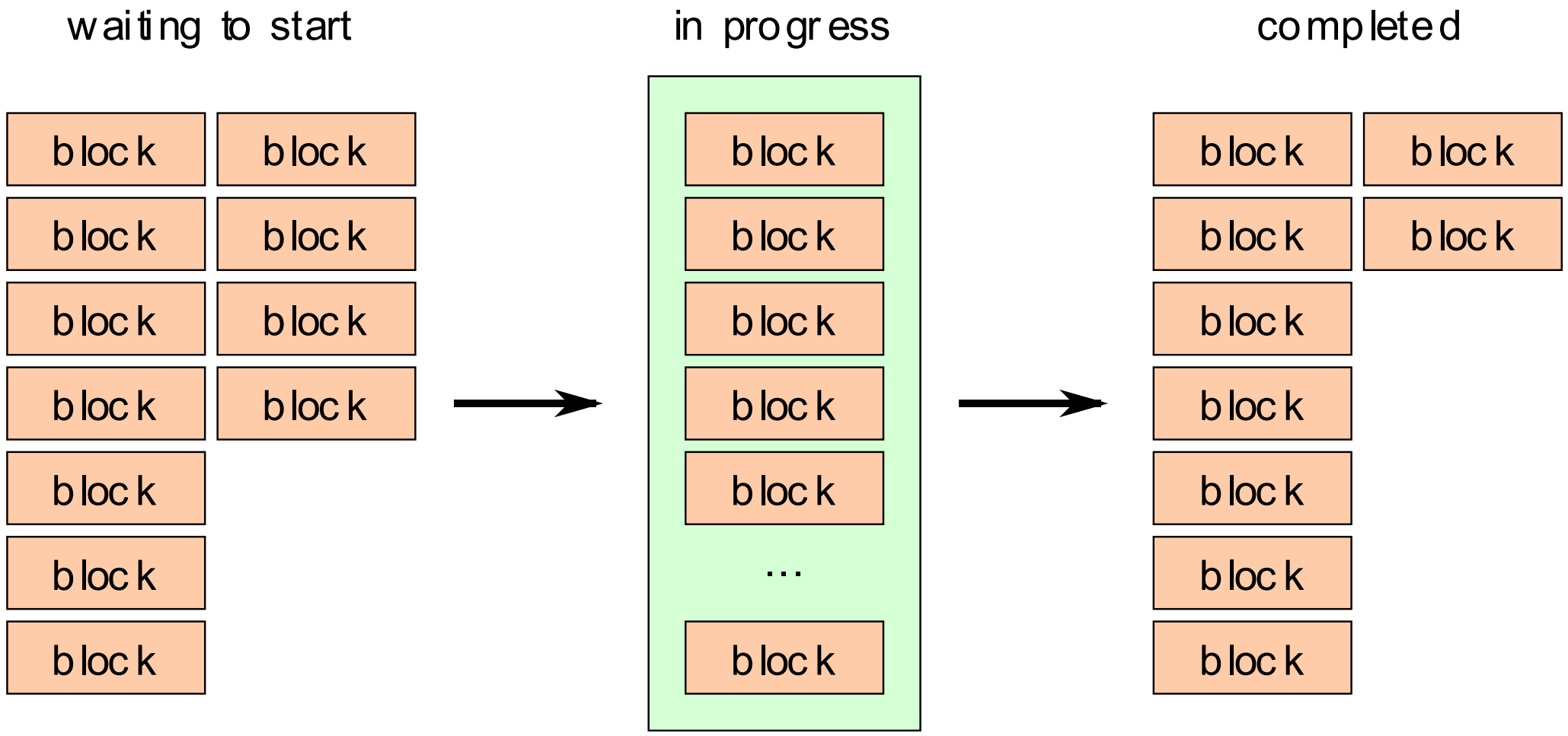
An algorithm intended for a CPU, like this:

```
void some_function(float* array, int index) {
    array[index] = ...;
}

for (int i = 0;  i < 1000000;  i++) {
    some_function(array, i);
}
```

would get translated into something like this for a GPU:

```
__global__ some_function(float* array) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    array[index] = ...;
}

some_function<<<blocks_per_grid, threads_per_block>>>(array);
```

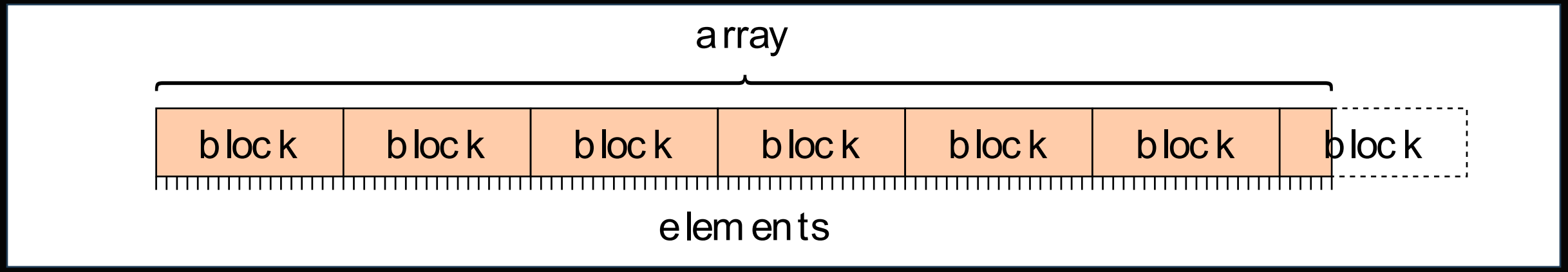
- The explicit iteration over elements of an array on the CPU is replaced by a GPU "kernel" function that applies to just one array element
- Each kernel-execution "thread" might run at any time.
- Execution is organized into "blocks": threads within a block run at the same time on shared resources, and the GPU schedules block execution in a way that keeps its streaming multiprocessors busy.
- The array must already be in the GPU's global memory.


waiting to start
in progress
completed
block
block
block
block
block
block
block
block
block
block
block
block
block
block
block
block
block
block
block
block
block
block
block
block
...
block
block
block
block
block
block
block
block
block
block

- The **blocks_per_grid** and **threads_per_block** are units of work—they control how computations are scheduled on the GPU—not units of array allocation. However, for 1-dimensional data, it's often best to define them in terms of the array size:

```
threads_per_block = 1024;   // maximum possible on most GPUs
blocks_per_grid = int(ceil(1.0 * size_of_array / threads_per_block));
```

- This minimizes the number of blocks needed to make sure that each array element is updated by exactly one thread.



If the `size_of_array` doesn't fit neatly into an integer number of blocks, a few threads will be wasted. To make sure that they don't update uninitialized data, you usually need to check for an excess in the kernel function:

```
__global__ some_function(float* array) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < size_of_array) {
        array[index] = ...;
    }
}
```

- Now for more examples to help get you started with Python libraries on GPUs

Wrap up

Parallel programming libraries exist for any language and some abstract away the underlying heterogeneous hardware

BUT: you can not avoid formulating your algorithms to be efficient running in parallel – be it as independent tasks or data parallel

- Use libraries where you can. They will result in more efficient than code you (or I) can write

Types of parallelism