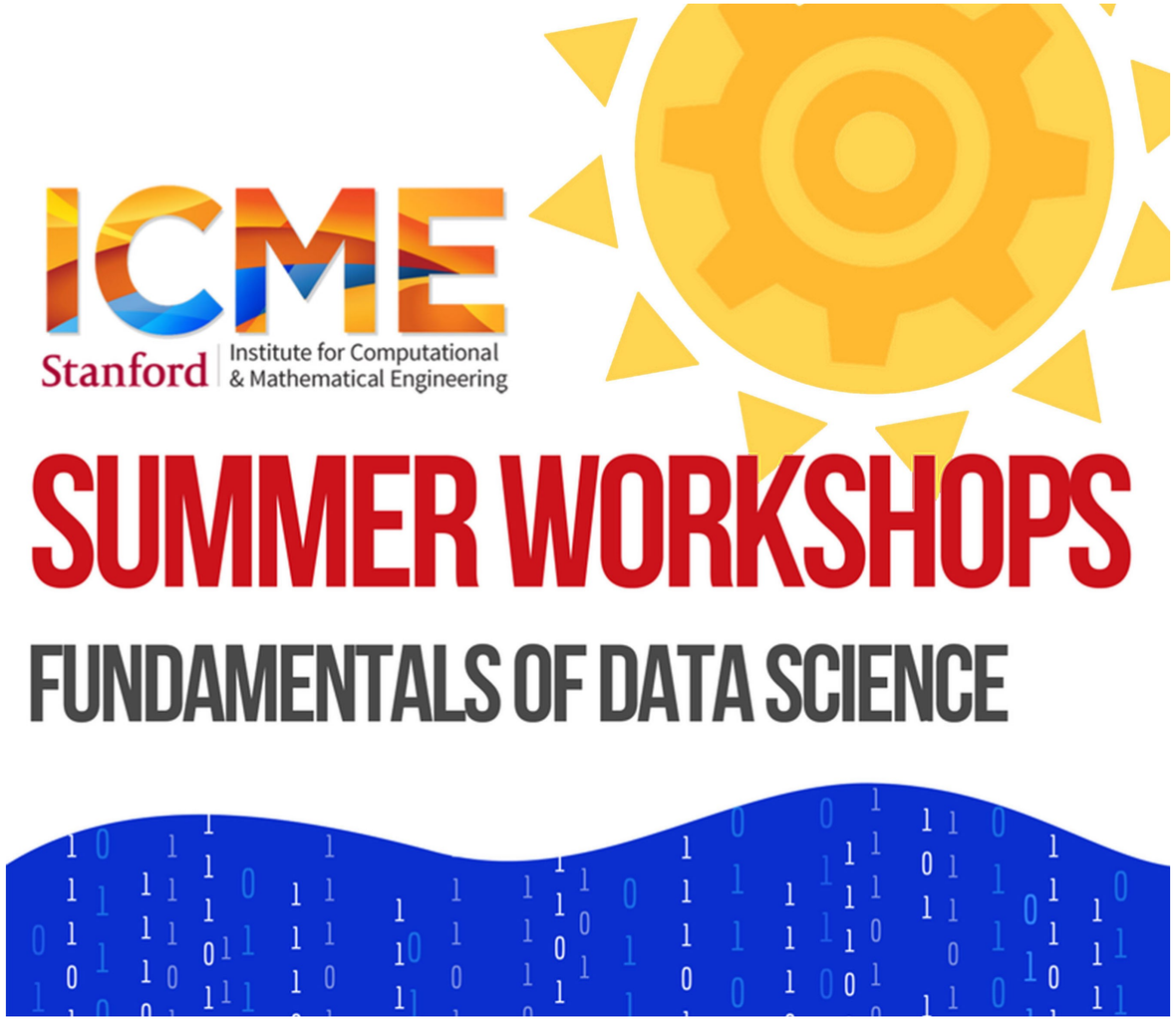


ICME Fundamentals of Data Science

Introduction to High-Performance Computing

Eric Darve

Logistics of workshop

- Schedule: Thursday and Friday, 1 to 4 PM
- GitHub page:

  https://ericdarve.github.io/icme-hpc-summer-2022/

- Shared Google notebooks:

  https://drive.google.com/drive/folders/1pwvw_HvZMtQqZl00nl95jDOUt-KPd_xy

Content of workshop

1. Introduction to High-Performance computing; history of HPC and current trends
2. How to program a multicore processor using OpenMP? This is the basis of HPC programming and an easy entry point.
3. GPU processors provide the backbone of HPC platforms for number crunching. We will cover OpenACC to write HPC programs for GPU processors.

Part 1

Introduction to HPC

History of HPC computers

What is high-performance computing?

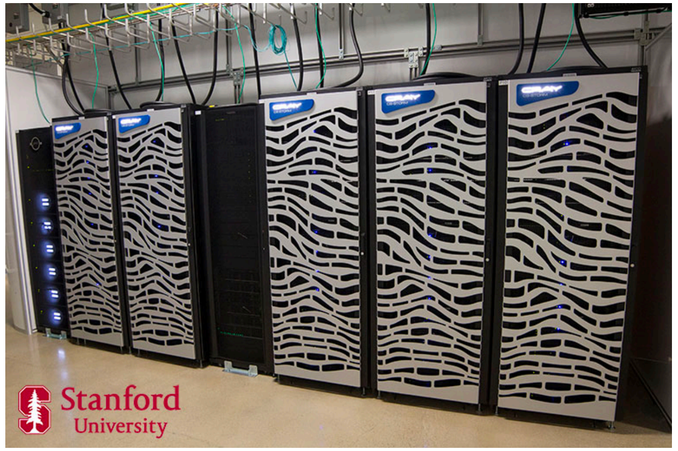
- Standard computers perform tasks sequentially, that is, transaction-by-transaction.
- This means that the next transaction, or job, happens only when the computer completes the previous one.
- In contrast, HPC uses many resources such as processors to complete many jobs **simultaneously.**

Supercomputers

- For the most part, HPC occurs on **supercomputers.**
- These powerful systems help companies solve problems that could otherwise be insurmountable.
- These problems, or tasks, require **processors** that can carry out instructions faster than standard computers.
- This is achieved by running **many processors in parallel** to obtain answers within a practical duration.

HPC hardware

- HPC jobs require fast **disks** and high-speed **memory.**
- HPC systems include computing and data-intensive servers with powerful **CPUs** that can be **vertically stacked.**
- HPC systems often have powerful **graphics processing units** (GPUs) that can run general-purpose computations.



Stanford XStream compute cluster

NVIDIA HPC system

HPC system designs

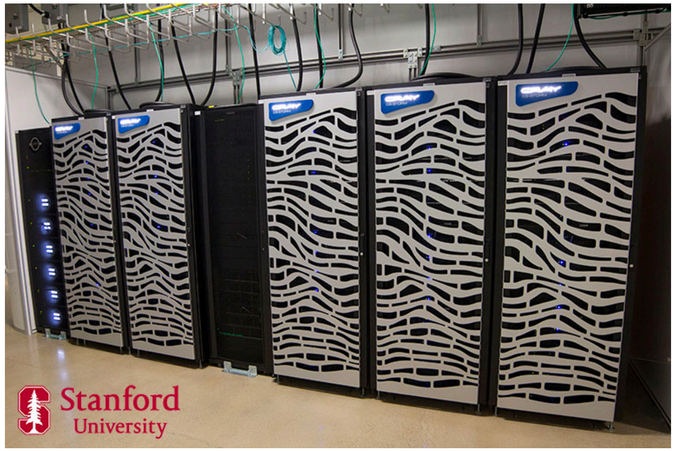
What is parallel computing?

Parallel computing HPC systems involve **hundreds of processors,** each running calculation payloads **simultaneously.**

HPC system designs

What is cluster computing?

- Cluster computing is a type of **parallel HPC system** consisting of a collection of computers working together as an integrated resource.
- Clusters can accommodate multiple applications and resources. They are managed by policy-based **scheduling** and can handle a dynamic workload consisting of **large numbers of jobs.**



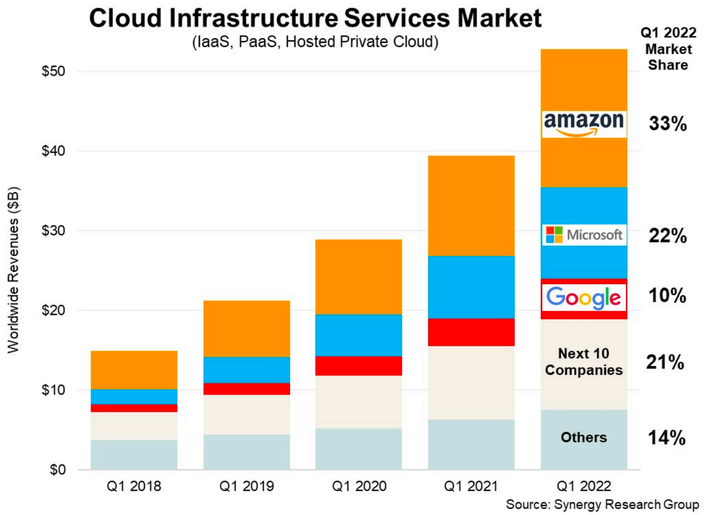
Stanford XStream compute cluster

HPC system designs

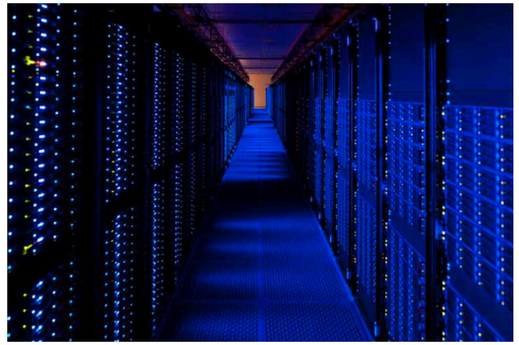
What are grid and distributed computing?

Grid and distributed computing HPC systems connect the processing power of multiple computers within a network. The network can be a grid at a single location or distributed **across a wide area in different places,** linking network, compute, data, and instrument resources.

Main vendors: **Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform**



https://www.srgresearch.com/



Amazon data center

Applications of HPC

Some of the key applications include:

- **Big data:** massive multi-dimensional datasets
- **Data analytics**
- Extreme performance **database**
- **Machine learning**

Applications!

Automotive and aerospace

- CFD-aerodynamic modeling
- FEA-impact and structural strength analysis
- CAD and CAM

Banking, financial services markets and insurance

- Monte Carlo simulations
- Risk analysis
- Fraud detection

Electronics design automation (EDA)

- Chip design and optimization
- Circuit simulation and verification
- Manufacturing optimization

Film, media and gaming

- Rendering
- Computer-aided graphics
- Computer-generated images (CGI)
- Transcoding and encoding
- Real-time image analysis and processing

Government and defense

- Intelligence agency
- Fraud analysis
- Climate modeling
- Weather forecasting
- Energy
- Nuclear stewardship
- Exploration

Life sciences

- Genomic processing and sequencing
- Pharmaceutical design
- Molecular modeling and biology simulation
- Protein docking

Oil and gas

- Seismic data processing
- Reservoir simulation and modeling
- Geospatial analytics
- Terrain and topology mapping
- CFD-aerodynamic modeling
- Wind simulation

Retail

- Inventory analysis
- Logistics and supply chain optimization
- Sentiment analysis
- Marketing offers

A short history of HPC

The beginnings:

- HPC market: **scientific** discoveries; **Fortran** (Formula Translation)
- **Cray** Research: supercomputers; focus: floating-point operations
- 1960's: specialized and expensive supercomputers; cold war, strategic necessity
- 1980's: number of processors goes up. Multiple processors (sometimes hundreds) are connected through a network.

A short history of HPC

- Vendors turn to **commodity markets** where processors are sold in large quantities.
  - Monolithic supercomputer systems splinter as many commodity components can be purchased from competing vendors.
  - The **economic barrier of entry** is lowered by at least a factor of ten.

Beowulf clusters

- Operating system: UNIX.
- 1981, Linus Torvalds released a freely available version of **Linux.**
- Message Passing Interface (MPI) library.

Borg, a 52-node Beowulf cluster used by the McGill University pulsar group to search for pulsations from binary pulsars

Beowulf clusters

- The performance of “Beowulf Clusters” (named for the NASA project that developed these systems) comes close to that of supercomputers of the day: commodity-grade computers + free and open source software + MPI.
- **High-performance interconnects** are developed; the market settles on InfiniBand (Mellanox)
- Name “supercomputer” replaced by **“HPC systems.”**

The multi and many core explosion

Three issues limit an increase in clock speed:

1. **Memory Speed:** the gap between processor and memory speed continued to grow
2. **Instruction Level Parallelism:** the increasing difficulty of finding enough parallelism in a single instruction stream
3. **Power Wall:** increased processor frequency causes an increase in operating temperature

The multi and many core explosion

- The era of **multi-core:** dual-core processors; more cores added to each new generation of processors
- Commodity **Graphics Processing Units** (GPUs) that contain large numbers (hundreds to thousands) of small, efficient cores.

Exascale computing and co-design

"On the Role of Co-design in High Performance Computing," R. F. Barrett et al.

The co-design strategy is based on developing partnerships with **computer vendors and application scientists** and engaging them in a highly **collaborative and iterative design process** well before a given system is available for commercial use. The process is built around identifying leading edge, high-impact scientific applications and providing concrete optimization targets rather than focusing on speeds and feeds (FLOPs and bandwidth) and percent of peak. **Rather than asking "what kind of scientific applications can run on an Exascale system" after it arrives, this application-driven design process instead asks "what kind of system should be built to meet the needs of the most important science problems."** This leverages eep understanding of specific application requirements and a broad-based computational science portfolio.

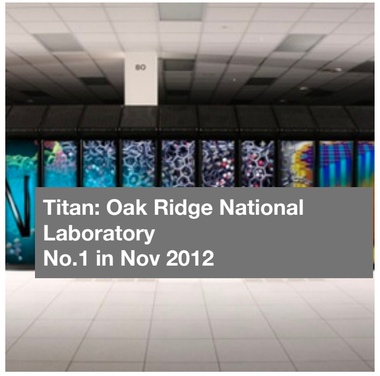
HPC history in pictures

CM-5: Los Alamos National Lab
No.1 in Jun 1993

ASCI Red: Sandia National Laboratory
No.1 from Jun 1997 until Jun 2000

BlueGene/L: Lawrence Livermore National Laboratory
No.1 from Nov 2004 until Nov 2007

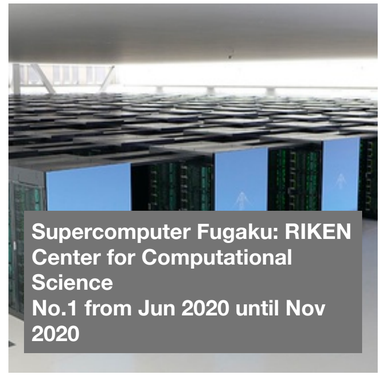
Tianhe-1A: National Supercomputing Center in Tianjin
No.1 in Nov 2010



Titan: Oak Ridge National Laboratory
No.1 in Nov 2012

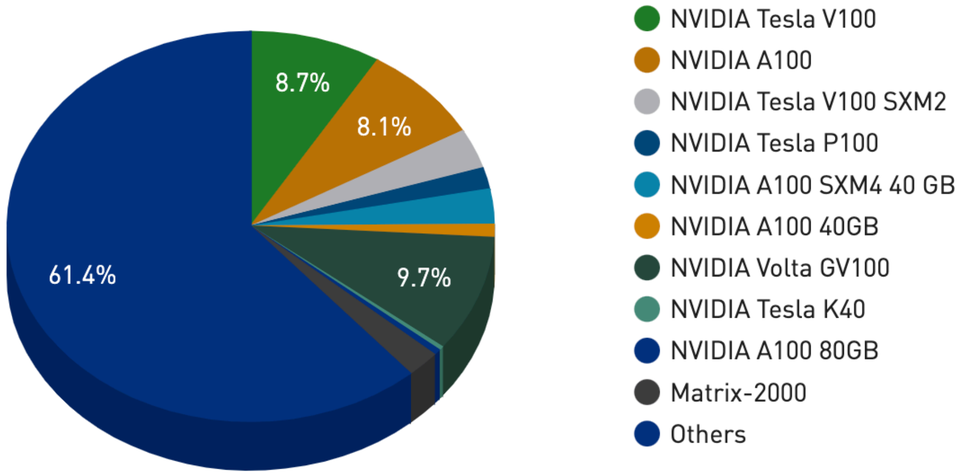
Sunway TaihuLight: National Supercomputing Center in Wuxi
No.1 from Jun 2016 until Nov 2017

Summit: DOE/SC/Oak Ridge National Laboratory
No.1 from Jun 2018 until Nov 2019



Supercomputer Fugaku: RIKEN Center for Computational Science
No.1 from Jun 2020 until Nov 2020

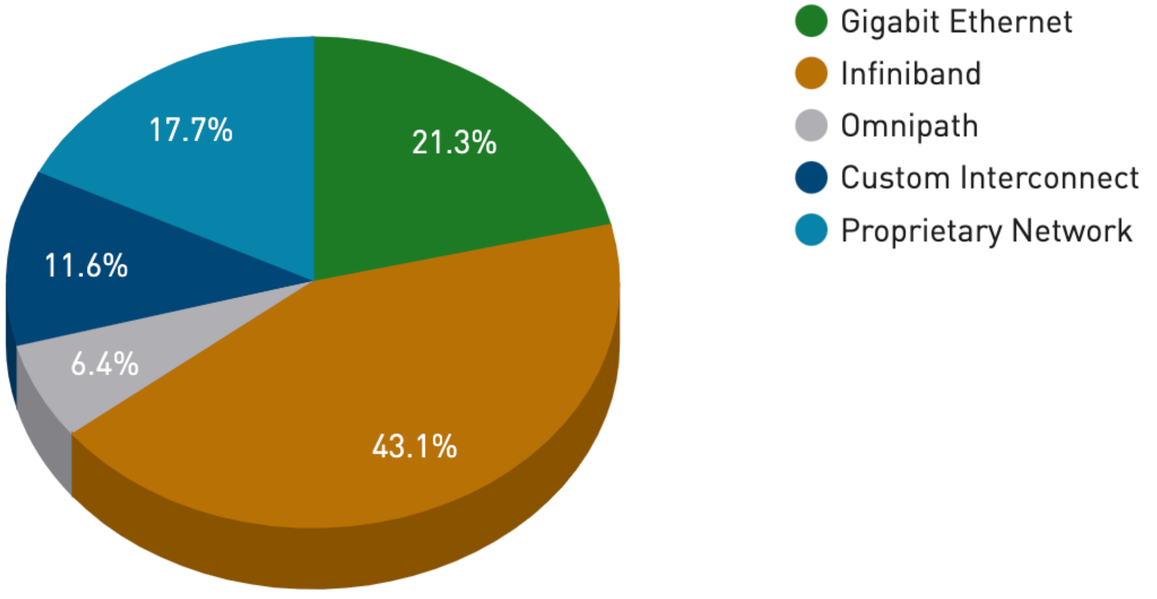
Accelerator/Co-Processor Performance Share



LINPACK achieved Theoretical peak

	Accelerator/Co-Processor	Count	System Share (%)	Rmax (GFlops)	Rpeak (GFlops)	Cores
1	NVIDIA Tesla V100	80	16	243,448,930	475,572,809	5,059,976
2	NVIDIA A100	15	3	226,001,000	324,135,290	2,125,952
3	NVIDIA Tesla V100 SXM2	12	2.4	91,975,490	182,486,069	2,059,208
4	NVIDIA Tesla P100	8	1.6	49,751,640	73,680,456	1,005,472
5	NVIDIA A100 SXM4 40 GB	5	1	81,312,000	115,202,938	869,192
6	NVIDIA A100 40GB	4	0.8	30,133,600	47,814,630	315,812
7	NVIDIA Volta GV100	4	0.8	269,439,000	362,564,722	4,408,096
8	NVIDIA Tesla K40	3	0.6	8,824,090	14,612,320	201,328
9	NVIDIA A100 80GB	2	0.4	13,806,000	18,688,410	124,160
10	Matrix-2000	1	0.2	61,444,500	100,678,664	4,981,760
11	NVIDIA 2050	1	0.2	2,566,000	4,701,000	186,368
12	NVIDIA Tesla K40m	1	0.2	2,478,000	4,946,790	64,384
13	NVIDIA Tesla K40/Intel Xeon Phi 7120P	1	0.2	3,126,240	5,610,481	152,692
14	NVIDIA Tesla P100 NVLink	1	0.2	8,125,000	12,127,069	135,828

Interconnect Family Performance Share

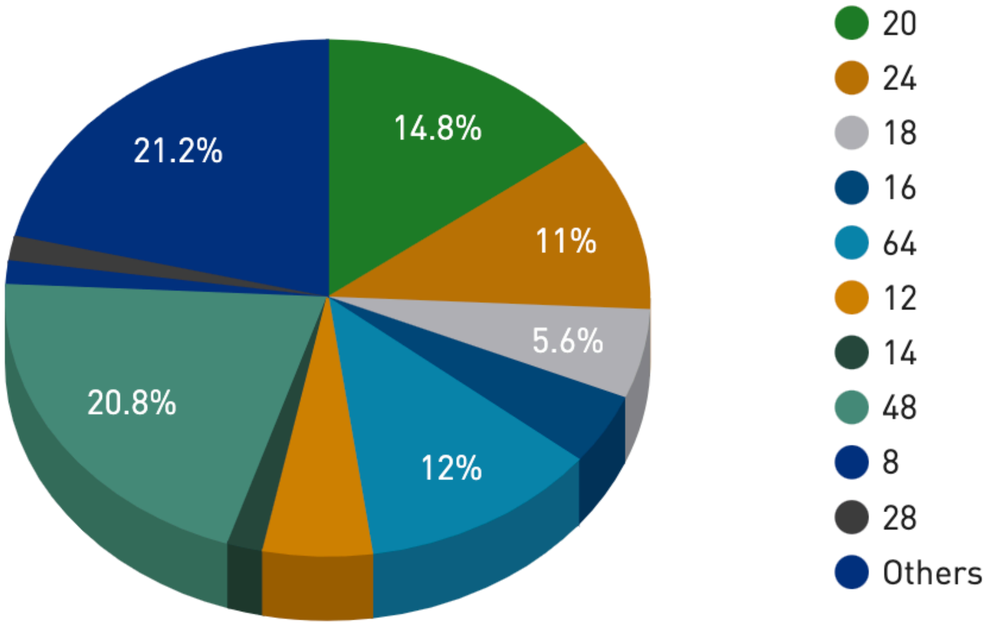


Mellanox

Intel

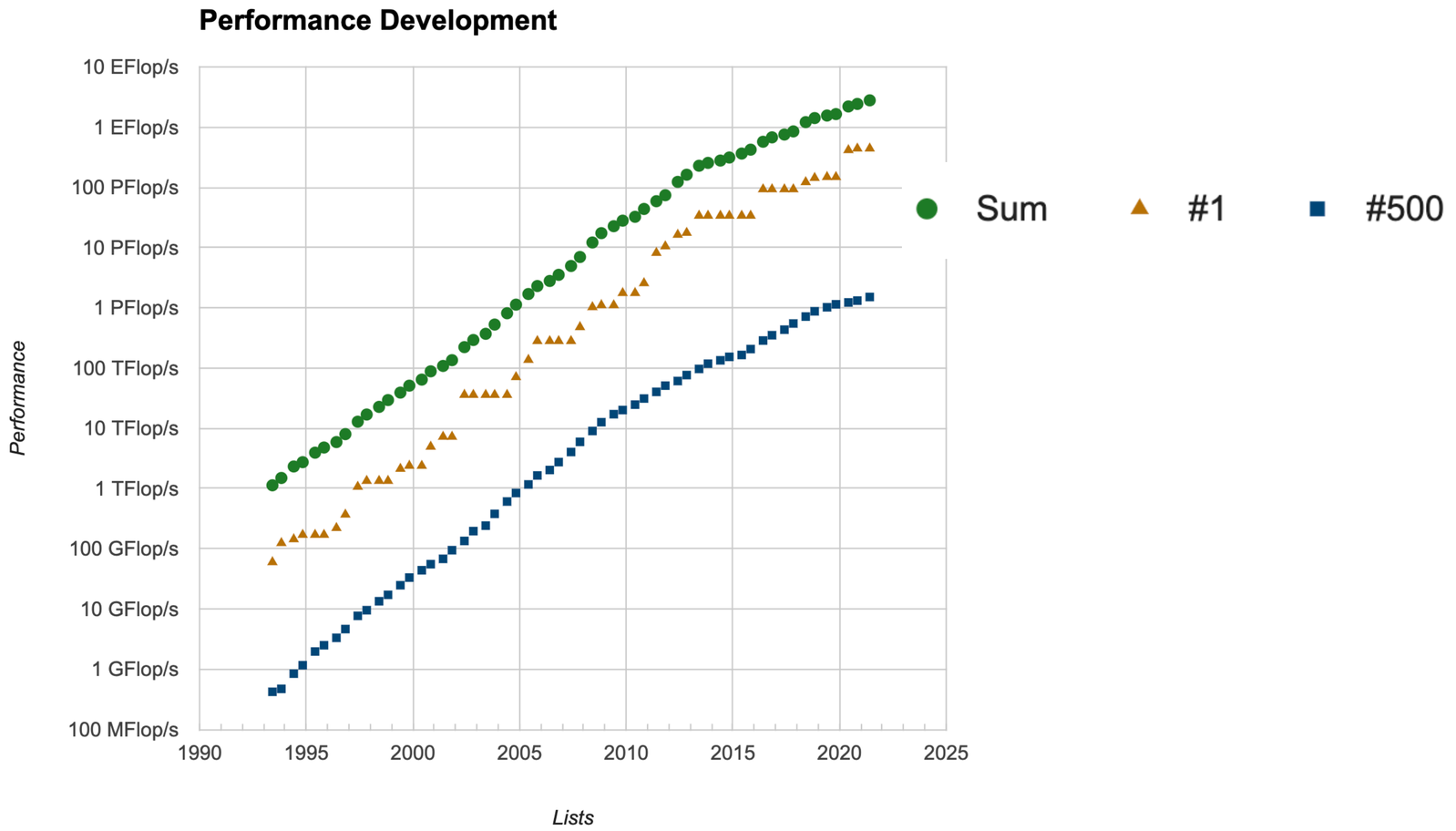
	Interconnect Family	Count	System Share (%)	Rmax (GFlops)	Rpeak (GFlops)	Cores
1	Gigabit Ethernet	247	49.4	592,043,220	1,210,167,243	19,827,104
2	Infiniband	168	33.6	1,200,636,818	1,832,049,310	22,511,764
3	Omnipath	42	8.4	178,516,898	281,314,234	4,361,016
4	Custom Interconnect	37	7.4	322,955,564	483,020,993	21,616,076
5	Proprietary Network	6	1.2	491,906,300	597,474,433	8,609,792

Cores per Socket Performance Share



- 20
- 24
- 18
- 16
- 64
- 12
- 14
- 48
- 8
- 28
- Others

Performance development



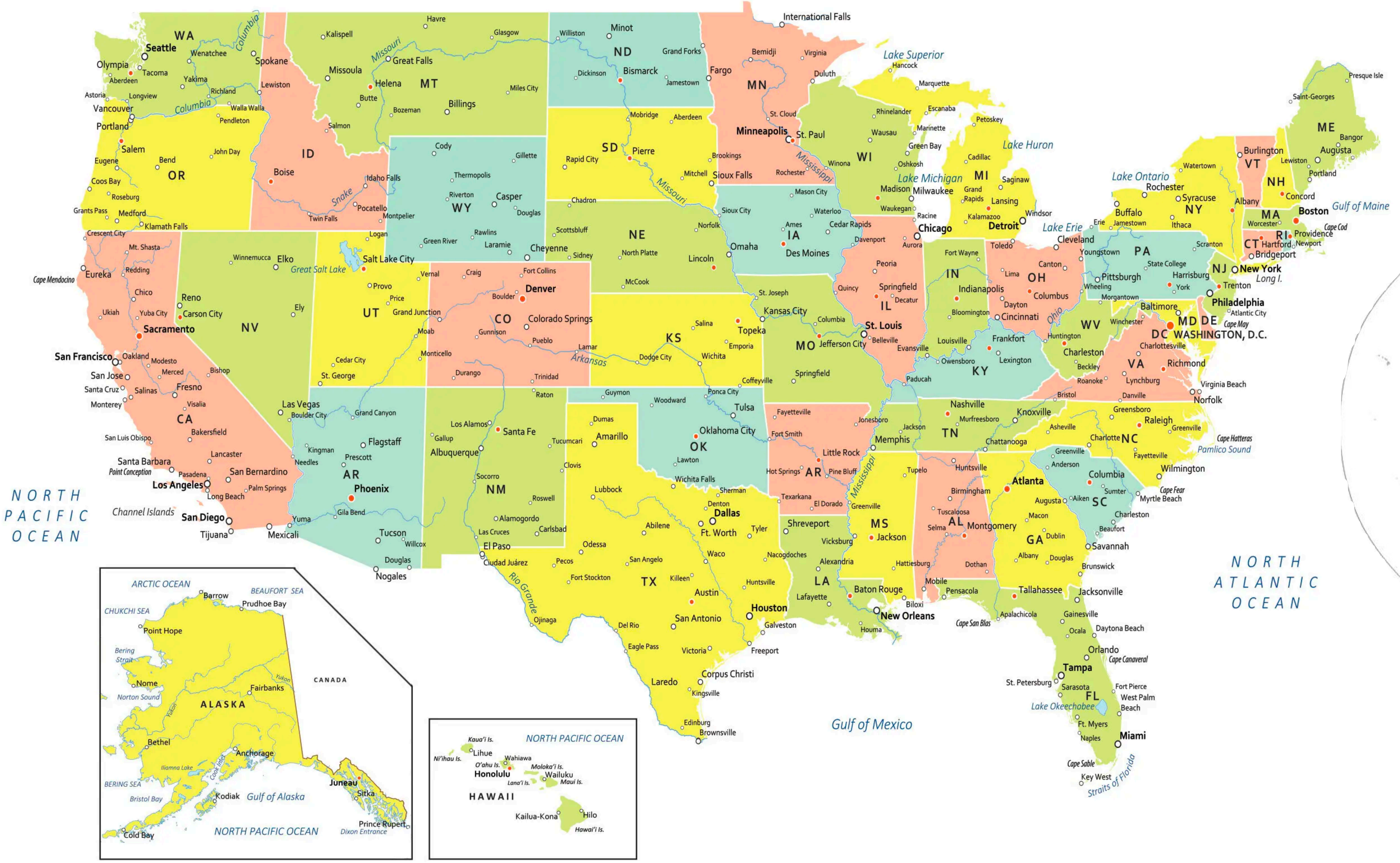
But before diving in further...
Let's get to know each other

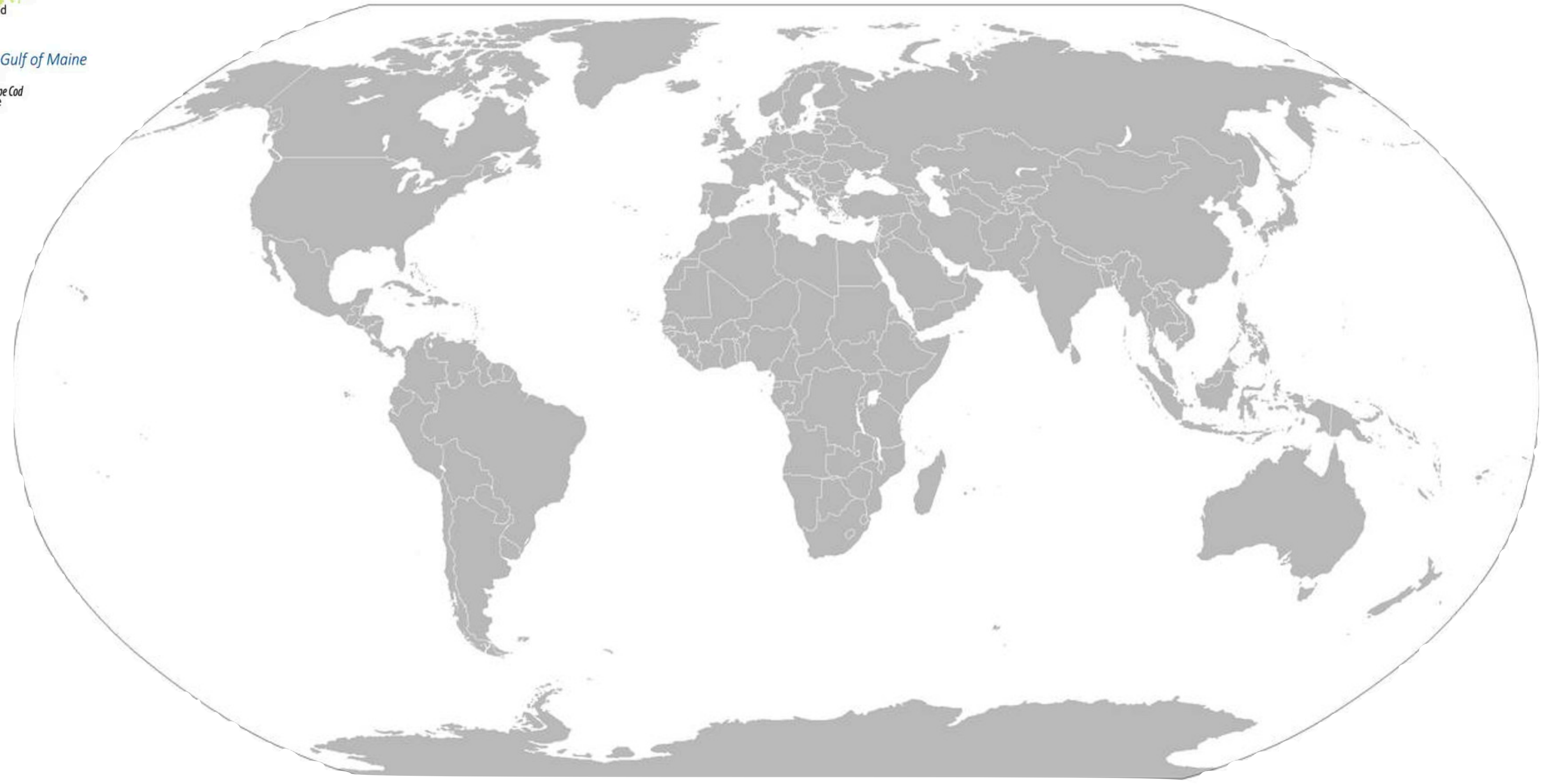
Instructor

- Eric Darve, ME, ICME, darve@stanford.edu
- Numerical linear algebra, machine learning for mechanics and engineering, parallel computing

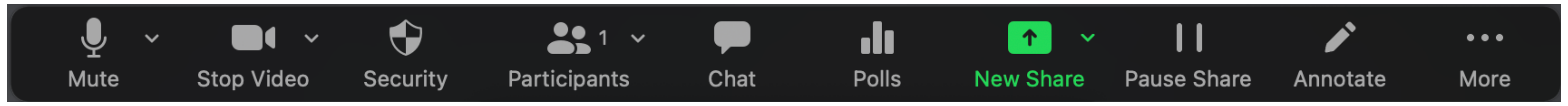
Teaching Assistant

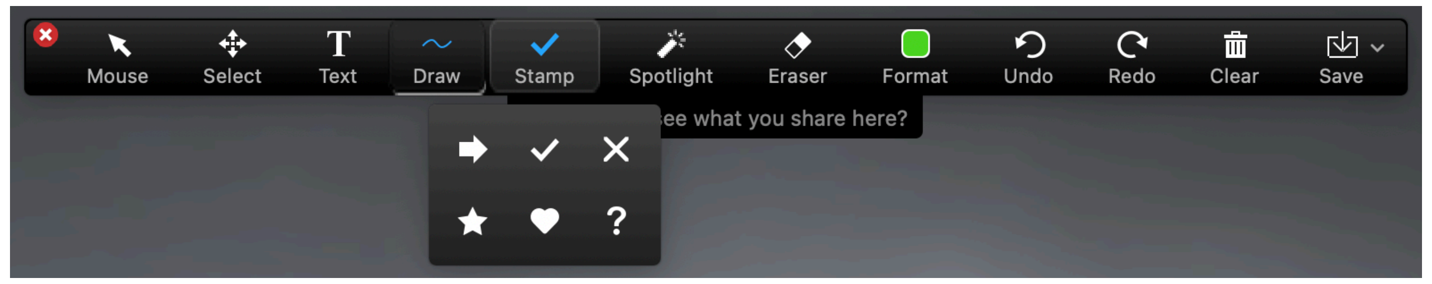
- Chenzhuo Zhu
- I am a graduating Ph.D. student at Stanford EE, advised by Prof. Bill Dally. I am interested in computer architecture and memory system design for data center applications.
- I grew up in Beijing, China. I received my B.S degree from Tsinghua University before coming to Stanford.
- I have worked as a teaching assistant for CME courses on scientific and parallel programming.
- I am also a snowboarder and a private pilot.





Mark your current location using zoom “Annotate → Stamp”





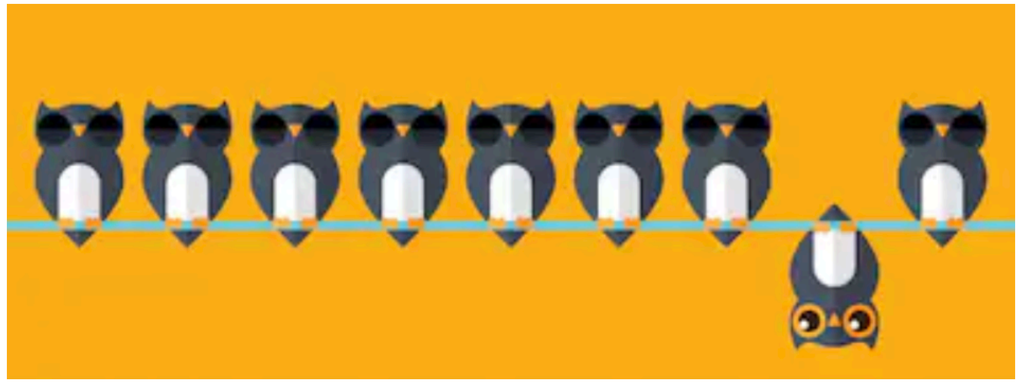
What this class is about

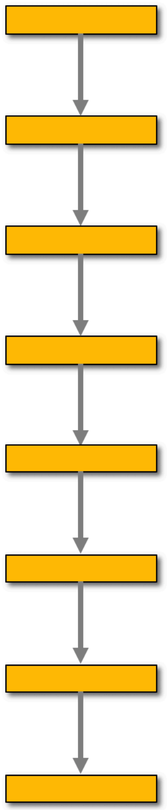
We will get an overview of parallel programming using two very accessible but powerful techniques:

1. **Shared memory** multicore processors using OpenMP.
2. **GPU** computing (NVIDIA + AMD) using OpenACC.

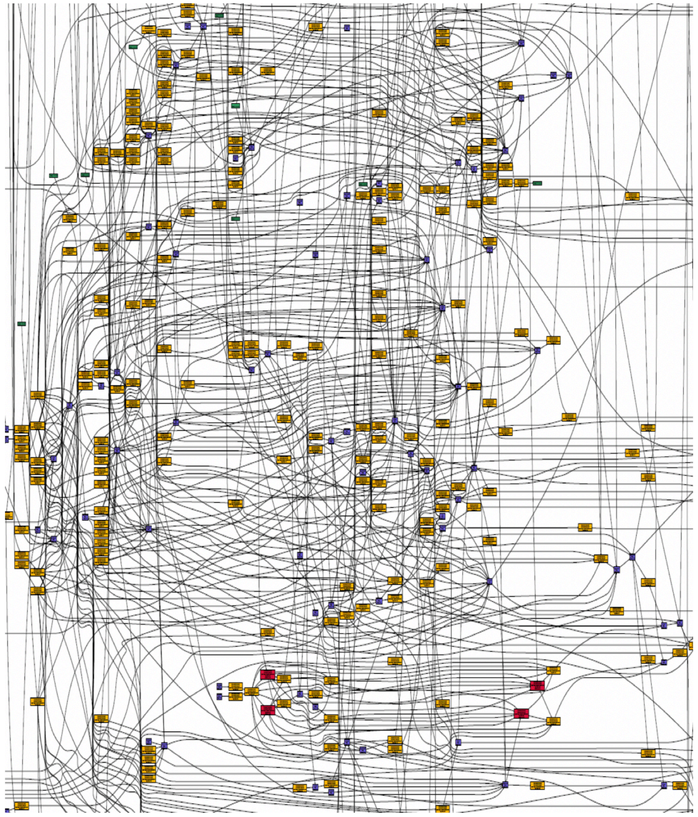
Thinking parallel

Parallel programs often look very different from sequential programs.



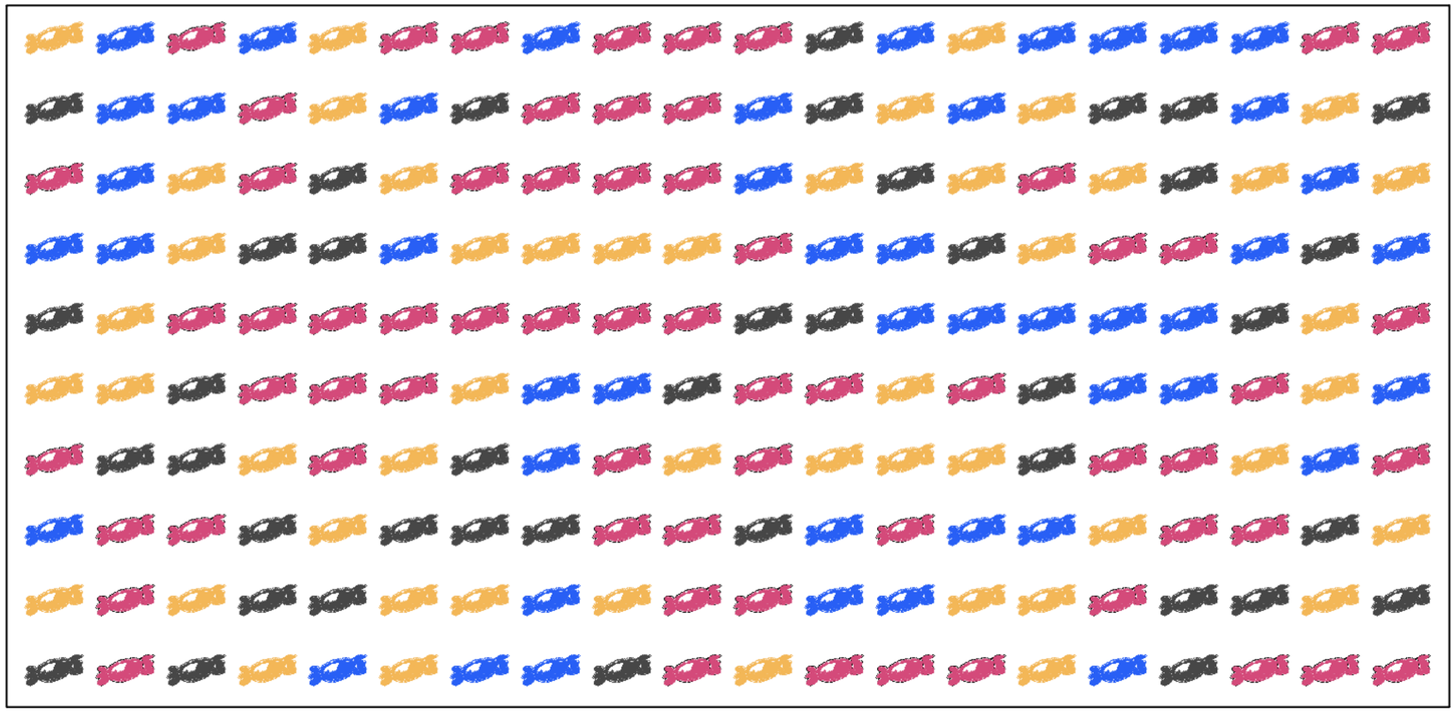


Sequential thinking

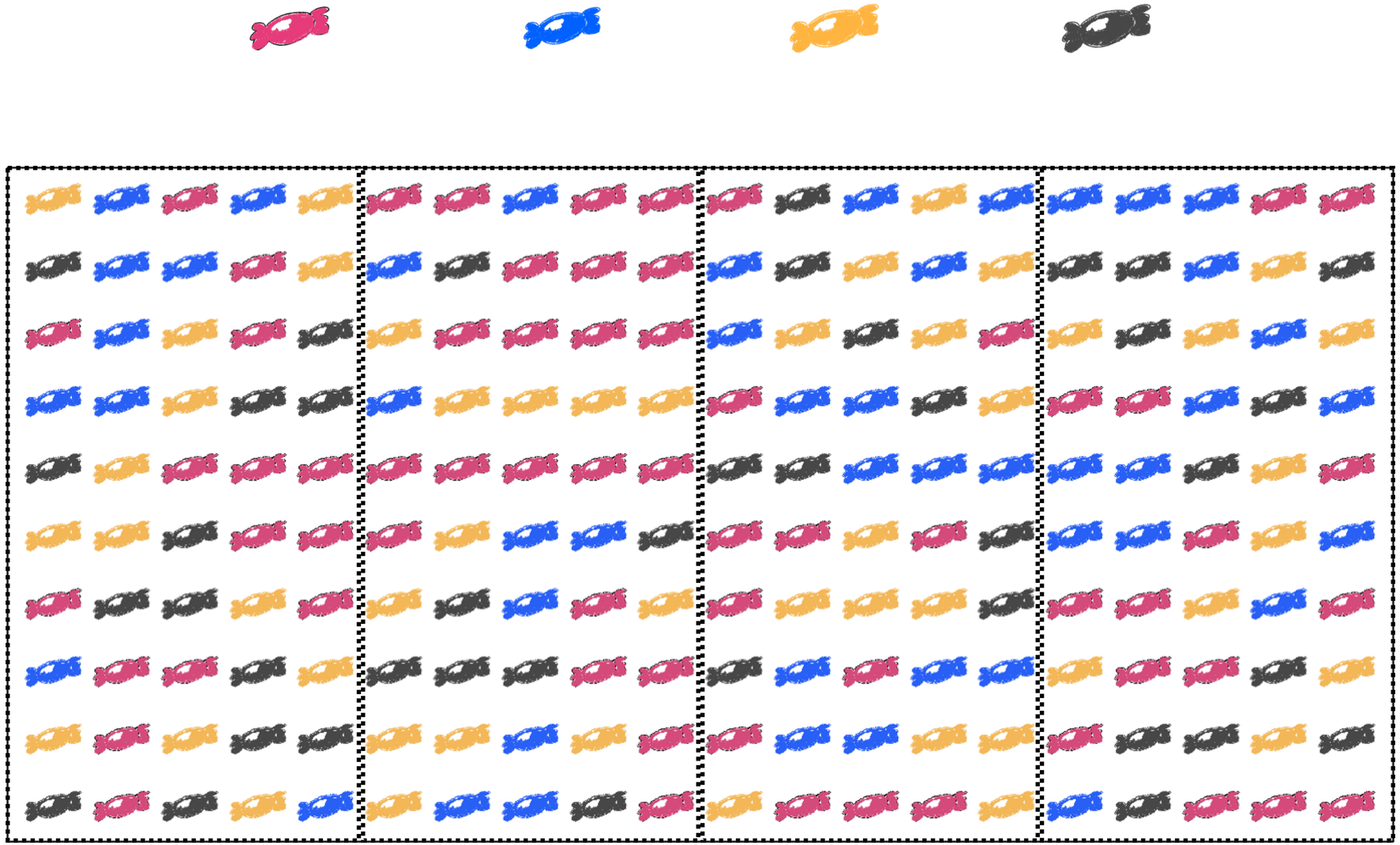


Parallel thinking

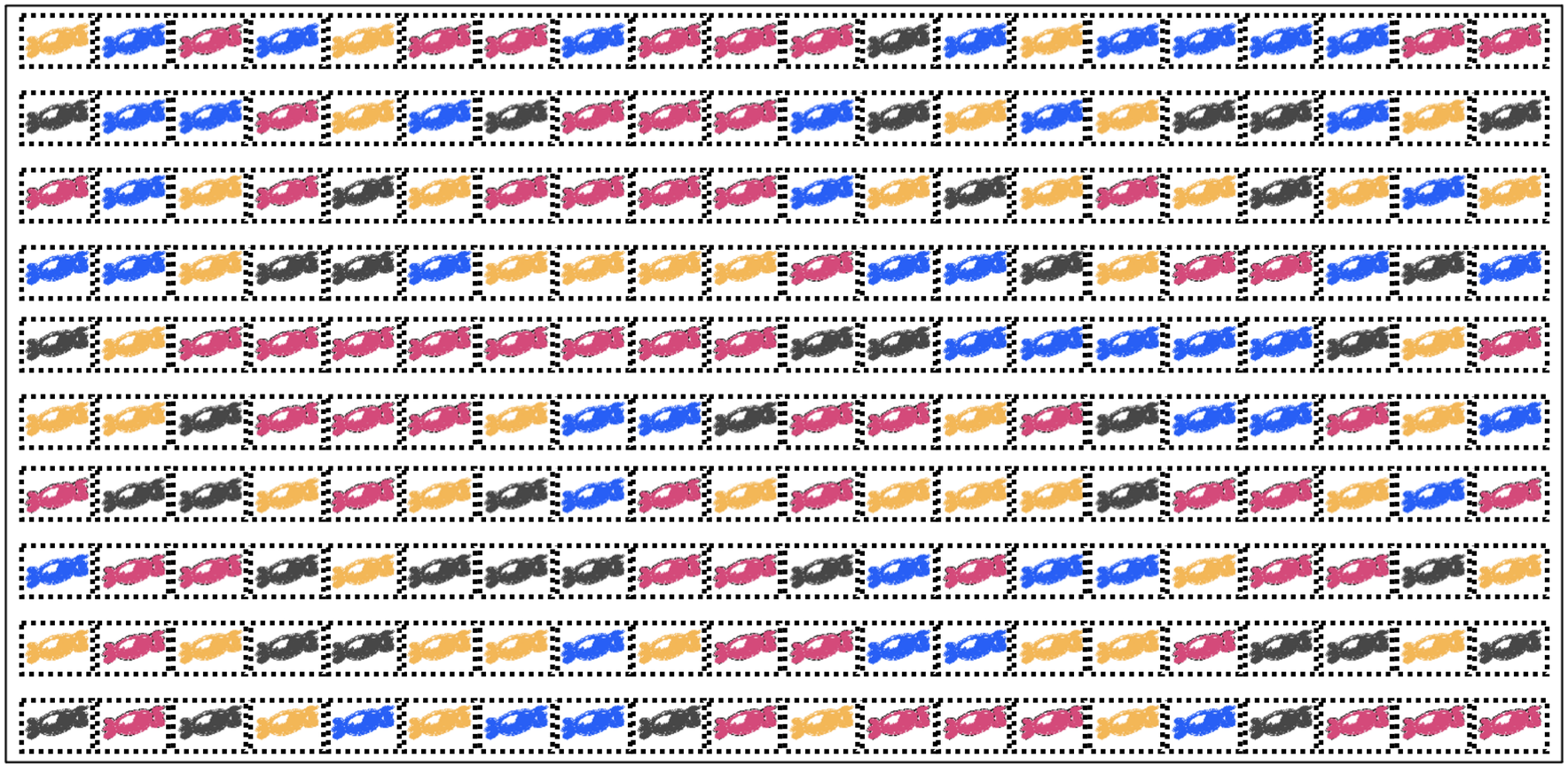
How would you count the number of candies of each color?



Using 2 cores



Using 4 cores



Using 200 cores

The initial step of counting candies goes fast.

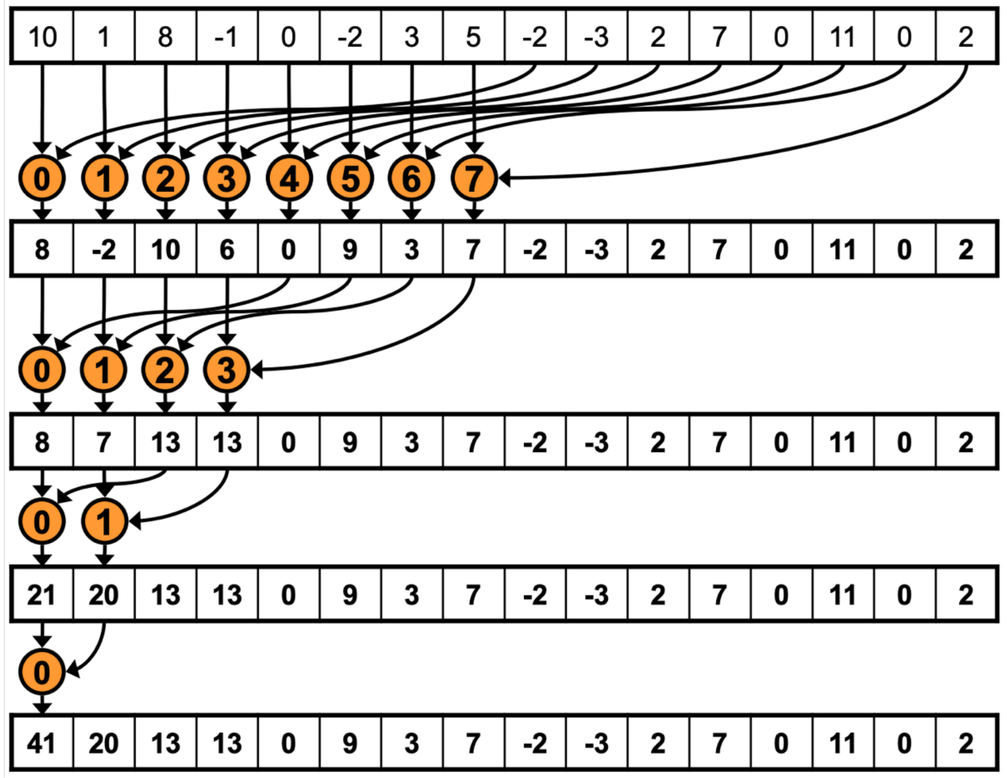
But after that, the following steps of reduction become problematic.

Can we use a single core to calculate the final sum for each candy?

- This is inefficient

How can we compute the final reduction efficiently and in parallel?

Efficient algorithm requires a tree reduction



Counting candies in parallel is more complicated than we initially thought.

Part 2

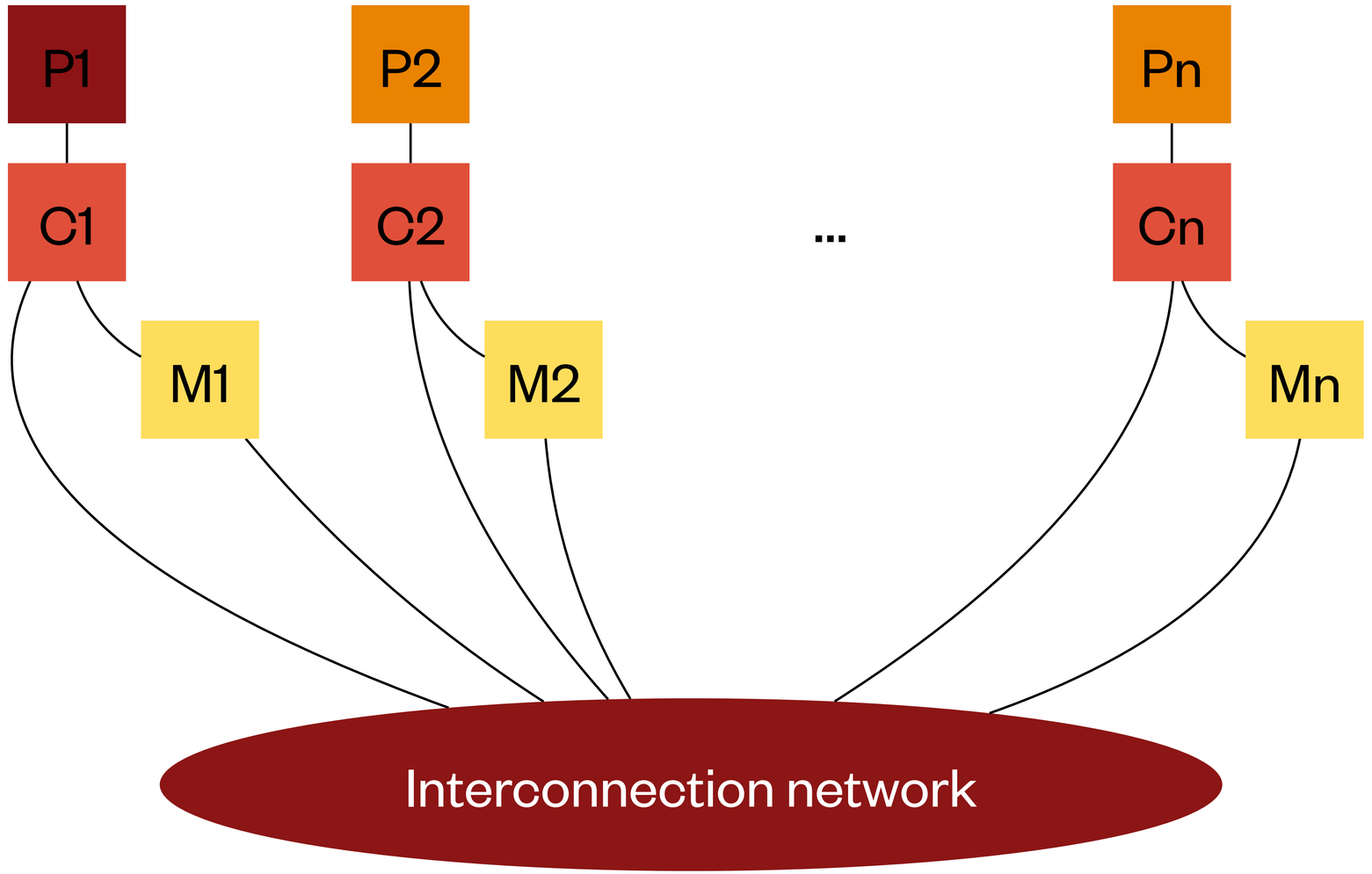
Programming multicore processors using OpenMP

Let's talk about shared memory computing.

This is the simplest method.

It applies to your laptop, desktop computer, or even your phone.

Shared memory processor architecture



Processing elements

Cache

Global memory

Network

Shared memory processor architecture

Schematic

- Several processors or **cores**
- A shared physical **memory** (global memory)
- An interconnection **network** to connect the processors with the memory

Multicore performance

- **Memory** is key to developing high-performance multicore applications
- More cores do not necessarily mean faster execution.
- **Memory traffic and time to access memory are often more critical than flops.**
- Memory is hierarchical and complex.

Our teaching platform

- Google Colab
- Allows running **CPU and GPU sample codes in the cloud** with no setup required!
- Demo: `colab_demo.ipynb`

OpenMP and OpenACC

In this workshop, we will focus on two solutions:

- **OpenMP** to program multicore processors
- **OpenACC** to program GPUs

OpenMP makes scientific multithreaded programming very easy!

- OpenMP simplifies multicore programming significantly.
- In many cases, adding one line of code is sufficient to make it run in parallel.
- OpenMP is the standard approach in scientific computing for multicore processors.

Goals of OpenMP

Standardization:

- Provide a **standard** among a variety of shared memory architectures/ platforms.
- Jointly defined and endorsed by a group of major computer hardware and software vendors.

Simple but powerful:

- Establish a simple and limited set of **directives** for programming shared memory machines.
- Significant parallelism can be implemented by using just 3 or 4 directives.
- *This goal is becoming less true with each new release, unfortunately.*

Goals of OpenMP

Ease of use:

- Provide the capability to parallelize a serial program **incrementally**.
- Provide the ability to implement both coarse-grained and fine-grained parallelism.

Portability:

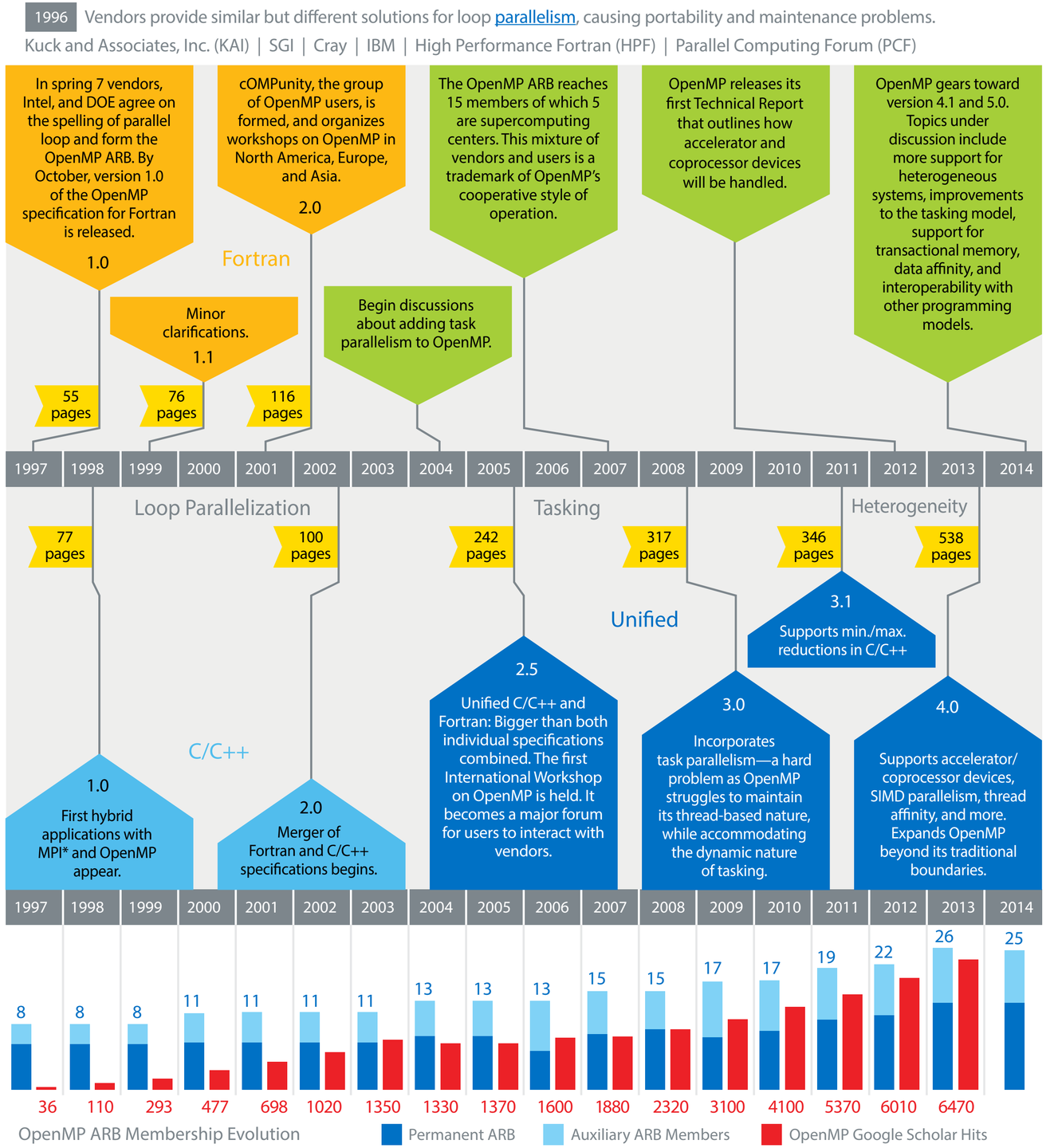
- The API is specified for **C/C++ and Fortran.**
- Public forum for API and membership
- **Most major platforms** have been implemented, including Unix/Linux and Windows.

Reference material

- OpenMP website `https://openmp.org`
- Wikipedia `https://en.wikipedia.org/wiki/OpenMP`
- LLNL tutorial `https://hpc-tutorials.llnl.gov/openmp`
- Intel `https://www.intel.com/content/dam/www/public/apac/xa/en/pdfs/ssg/Programming_with_OpenMP-Linux.pdf`

History of OpenMP

https://www.openmp.org/uncategorized/openmp-timeline/



Hello World example

Let's take a simple piece of code to get started:

```
for (int i = 0; i < n; ++i) {
  x[i] = i;
  y[i] = std::sqrt(float(i));
}
```

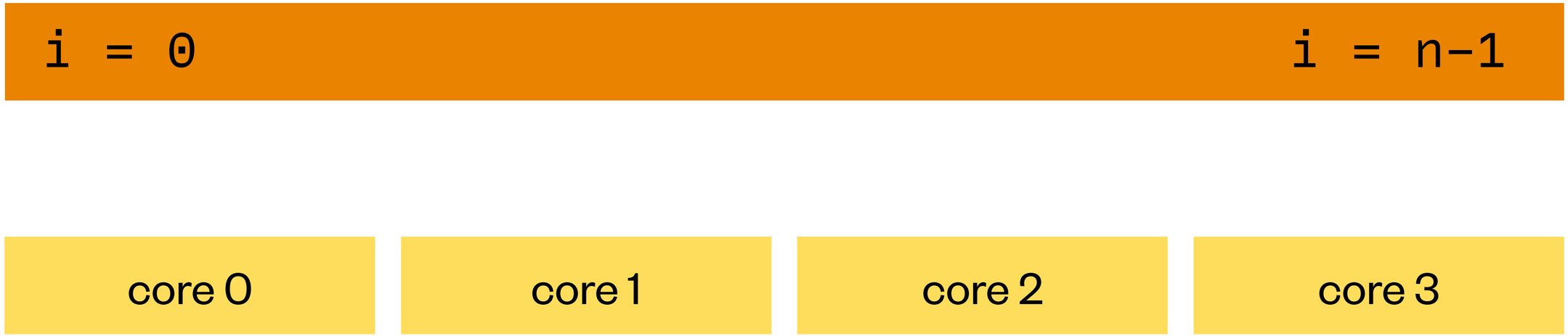
Demo: `omp_hello_world.ipynb`

How can we parallelize this code?

- Assume you have **multiple cores** that can do computation **in parallel.**
- The for loop can be split across the cores, and each core can compute a small chunk of the iterations.

```
for (int i = 0; i < n; ++i) {
  x[i] = i;
  y[i] = std::sqrt(float(i));
}
```

How can we parallelize this code?




```
for (int i = 0; i < n; ++i) {
  x[i] = i;
  y[i] = std::sqrt(float(i));
}
```

We can distribute the computation across the different cores using OpenMP.

OpenMP code

Let's parallelize the first loop that calculates x[i] and y[i].

```
  const int num_threads = 4;
  omp_set_num_threads(num_threads);
#pragma omp parallel for
  for (int i = 0; i < n; ++i) {
    x[i] = i;
    y[i] = std::sqrt(float(i));
    core[i] = omp_get_thread_num();
  }
  for (int i = 0; i < n; ++i)
    printf("Iteration %d was computed by thread %d\n", i, core[i]);
```

Output

```
Iteration 0 was computed by thread 0
Iteration 1 was computed by thread 0
Iteration 2 was computed by thread 1
Iteration 3 was computed by thread 1
Iteration 4 was computed by thread 2
Iteration 5 was computed by thread 2
Iteration 6 was computed by thread 3
Iteration 7 was computed by thread 3
```

Unit testing

Writing HPC code for scientific applications is not easy.

Thinking in parallel is much more complicated than thinking sequentially and is error-prone.

An important strategy to find errors is to use **unit testing:**

- Write a small piece of code in a function
- Immediately **test that the function works as expected**

GoogleTest

An excellent library for **unit testing is GoogleTest.**

It provides a simple infrastructure to write and manage tests.

Let's learn how it works and how to use it.

Example of test

Function to initialize a vector:

```
void init_i() {
  for (int i = 0; i < n; ++i) x[i] = i;
}
```

Demo: `googletest.ipynb`

How to write a unit test

- Use the macro `TEST`
- Use testing macros like `ASSERT_EQ`.
- Many additional macros and functions available in the library.

```
TEST(demoTest, init) {
  init_i();
  for (int i = 0; i < n; ++i) ASSERT_EQ(x[i], float(i));
}
```

Beware of roundoff errors

- Calculations on a computer **are not exact.**
- **Each operation generates a small roundoff error.**
- The approximate number of accurate digits depends on the precision of the floating point format.

Single precision	Double precision
~ 7.2	~ 15.9

“Exact” test

This test does not suffer from roundoff errors.

```
void init_1() {
  for (int i = 0; i < n; ++i) x[i] = 1;
}
void sum_x() {
  sum = 0;
  for (int i = 0; i < n; ++i) sum += x[i];
}
TEST(demoTest, sum) {
  init_1();
  sum_x();
  ASSERT_EQ(sum, float(n));
}
```

With larger numbers, roundoff errors start appearing

```
void init_i() {
  for (int i = 0; i < n; ++i) x[i] = i;
}
void sum_x() {
  sum = 0;
  for (int i = 0; i < n; ++i) sum += x[i];
}
TEST(demoTest, sum_i) {
  init_i();
  sum_x();
  const float expd = float(n * (n - 1) / 2.);
  ASSERT_NEAR(sum, expd, n * expd * mach_eps);
  printf("Roundoff errors are equal to: %9.6f; tolerance threshold: %9.6f.\n",
         abs((sum - expd) / expd), n * mach_eps);
}
```

```
Roundoff errors are equal to:  0.000042; tolerance threshold:
0.001192.
```

Is my calculation accurate?

Determining whether a calculation is “correct” can be difficult.

Is the difference due to a coding error, or is it a roundoff error?

Unit testing and debugging

- Unit testing is one of the many methods you need to learn to be able to write correct code for complex applications.
- Additional documentation at:

  `https://google.github.io/googletest/`

```
TEST(ompTest, omp_loop) {
#pragma omp parallel for
  for (int i = 0; i < n; ++i) {
    x[i] = i;
    y[i] = i * i;
  }
  for (int i = 0; i < n; ++i) {
    ASSERT_EQ(x[i], i);
    ASSERT_EQ(y[i], i * i);
  }
#pragma omp parallel for
  for (int i = 0; i < n; ++i) {
    z[i] = x[i] + y[i];
  }
  for (int i = 0; i < n; ++i) {
    ASSERT_EQ(z[i], (float)(i + i * i));
  }
}
```

Let's test our OpenMP Hello World!

```
Running main() from googletest-main/googletest/src/gtest_main.cc
[==========] Running 4 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 4 tests from ompTest
[ RUN      ] ompTest.omp_loop
[       OK ] ompTest.omp_loop (0 ms)
[ RUN      ] ompTest.omp_reduction
[       OK ] ompTest.omp_reduction (0 ms)
[ RUN      ] ompTest.omp_schedule
[       OK ] ompTest.omp_schedule (0 ms)
[ RUN      ] ompTest.omp_collapse
[       OK ] ompTest.omp_collapse (0 ms)
[----------] 4 tests from ompTest (1 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 1 test suite ran. (1 ms total)
[  PASSED  ] 4 tests.
```

Demo: `omp_lab.ipynb`

More on OpenMP

- OpenMP is a vast topic with a lot of additional functionalities.
- We will only review some of the main features.
- An important one is the **reduction** operator.

Reduction

Consider the following code:

```
float sum = 0;
for (int i = 0; i < n; ++i) {
  sum += x[i];
}
```

Race condition

```
    sum += x[i];
```

If multiple cores attempt to update the variable sum at the same time, the result becomes undetermined.

This will lead to an erroneous result. **This is a bug!**

OpenMP reduction

We need to tell the compiler that `sum` should be computed differently.

Adding numbers is called a reduction operation.

We have to use the OpenMP **reduction** clause to get the correct code.

Reduction clause

```
  float sum = 0;
#pragma omp parallel for reduction(+ : sum)
  for (int i = 0; i < n; ++i) {
    sum += x[i];
  }
```

- The final result will now be correct.
- Other reduction operators: `-`, `*`, `max`, `min`
- `+` logical and boolean operators

How should we schedule loops?

- **Loop scheduling is critical for performance.**
- OpenMP has extensive functionalities to improve performance of for loop executions.
- This can be achieved by specifying different loop scheduling policies.

Example of static policy

```
#pragma omp parallel for schedule(static, 32)
  for (int i = 0; i < n; ++i) {
    z[i] = x[i] + y[i];
  }
```

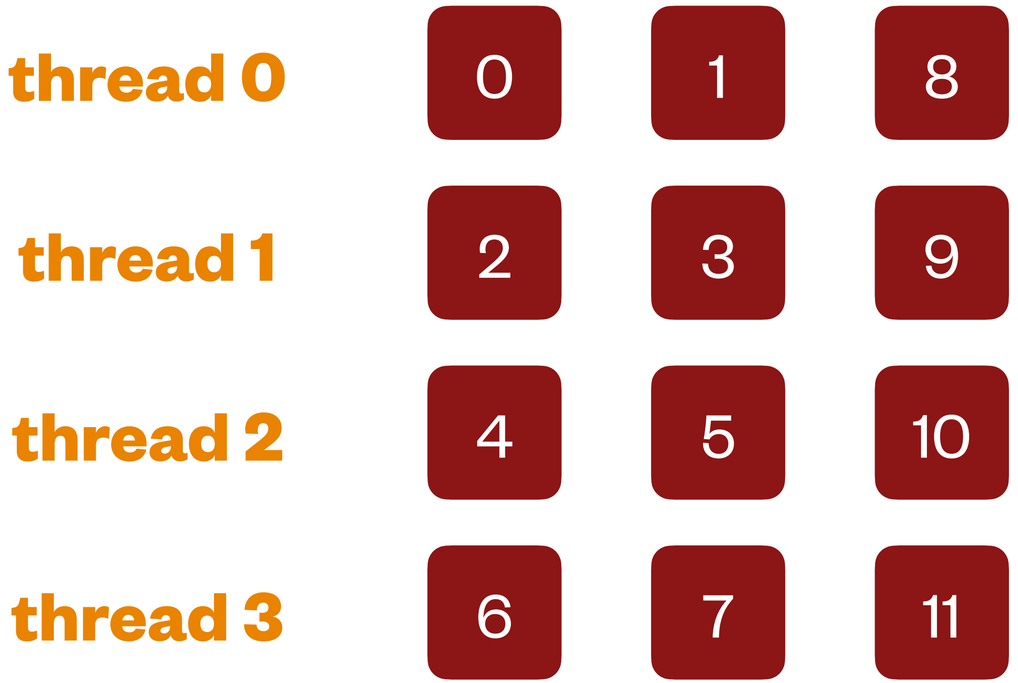
What does this do?

schedule(static, 1)

```
#pragma omp parallel for schedule(static, 1)
```

schedule(static, 2)

```
#pragma omp parallel for schedule(static, 2)
```



Example

```
#pragma omp parallel for
  for (int i = 0; i < n; ++i) {
    x[i] = i;
    y[i] = i * i;
  }
#pragma omp parallel for schedule(static, 32)
  for (int i = 0; i < n; ++i) z[i] = x[i] + y[i];

  for (int i = 0; i < n; ++i) ASSERT_EQ(z[i], (float)(i+i*i));
```

schedule(dynamic, 1)

```
#pragma omp parallel for schedule(dynamic, 1)
```

Example

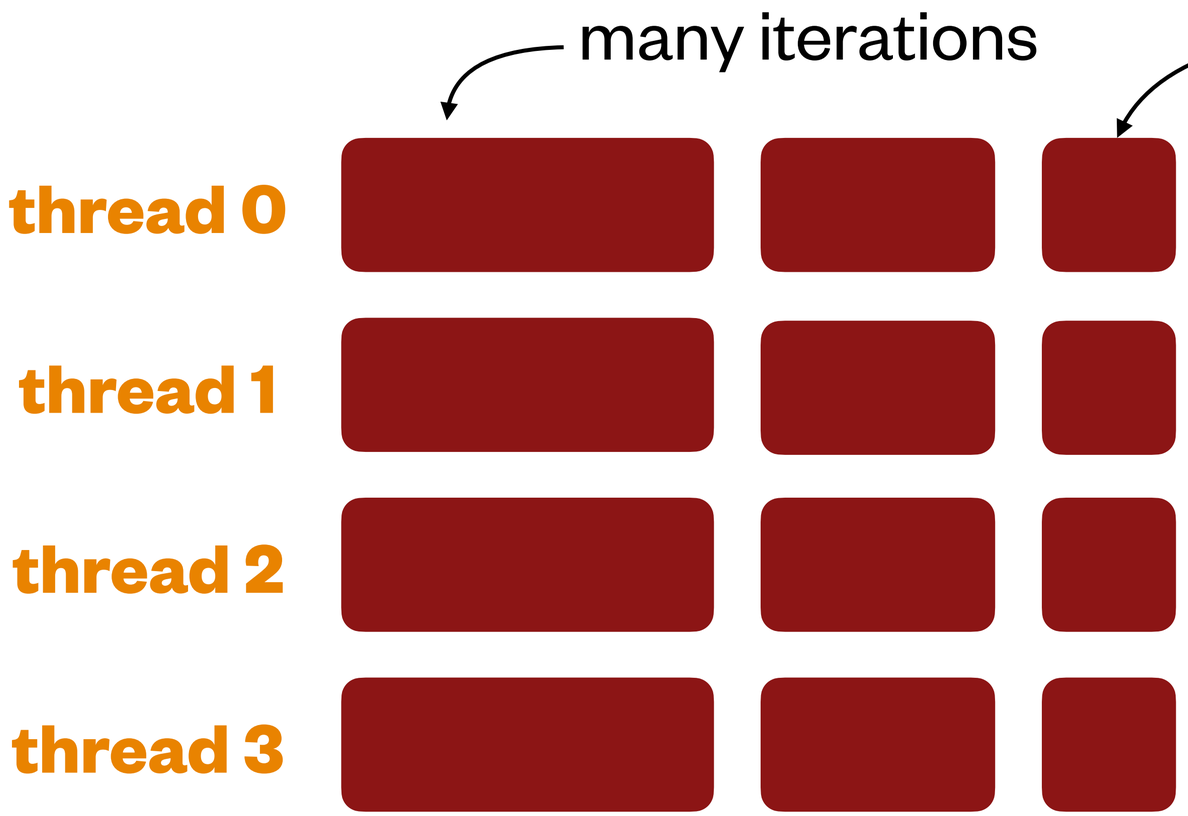
chunk size

```
#pragma omp parallel for schedule(dynamic, 32)
  for (int i = 0; i < n; ++i) z[i] = x[i] + y[i];

  for (int i = 0; i < n; ++i) ASSERT_EQ(z[i], (float)(i+i*i));
```

schedule(guided)

```
#pragma omp parallel for schedule(guided)
```



- Large iteration chunks are assigned initially.
- As the calculation progresses, smaller chunks are assigned.
- In most cases, this allows all cores to finish their computation simultaneously, which is optimal.

Nested loops

- What happens if the **number of iterations is small?**
- Few iterations are assigned to each core.
- This may lead to a significant **load imbalance,** i.e., one of the cores finishes much later than the others.

Balancing the workload

- In general, it is better to parallelize loops **with many iterations.**
- This makes it easier for the scheduler to assign work to the cores so that they **all finish around the same time.**
- If your loop does not have enough iterations, you have the option of "merging" it with the following nested loop.
- This is called **loop collapse** in OpenMP.

Example of loop collapse

```
#pragma omp parallel for collapse(2)
  for (int i = 0; i < n0; ++i) {
    for (int j = 0; j < n0; ++j) {
      x[i * n0 + j] = i * n0 + j;
      y[i * n0 + j] = i - j;
    }
  }
  for (int i = 0; i < n; ++i) ASSERT_EQ(x[i], float(i));
  for (int i = 0; i < n0; ++i)
    for (int j = 0; j < n0; ++j) ASSERT_EQ(y[i * n0 + j], float(i - j));
```

Example of loop collapse

```
#pragma omp parallel for collapse(2)
  for (int i = 0; i < n0; ++i)
    for (int j = 0; j < n0; ++j) z[i * n0 + j] = x[i * n0 + j] + y[i * n0 + j];

  for (int i = 0; i < n0; ++i)
    for (int j = 0; j < n0; ++j) ASSERT_EQ(z[i * n0 + j], float(i * (n0 + 1)));
```

Many other OpenMP concepts

- `atomic`
- `critical`
- `single`
- `task`
- `barrier, taskwait`
- …
- Reference guides

Part 3

Programming GPU processors using OpenACC

GPUs!

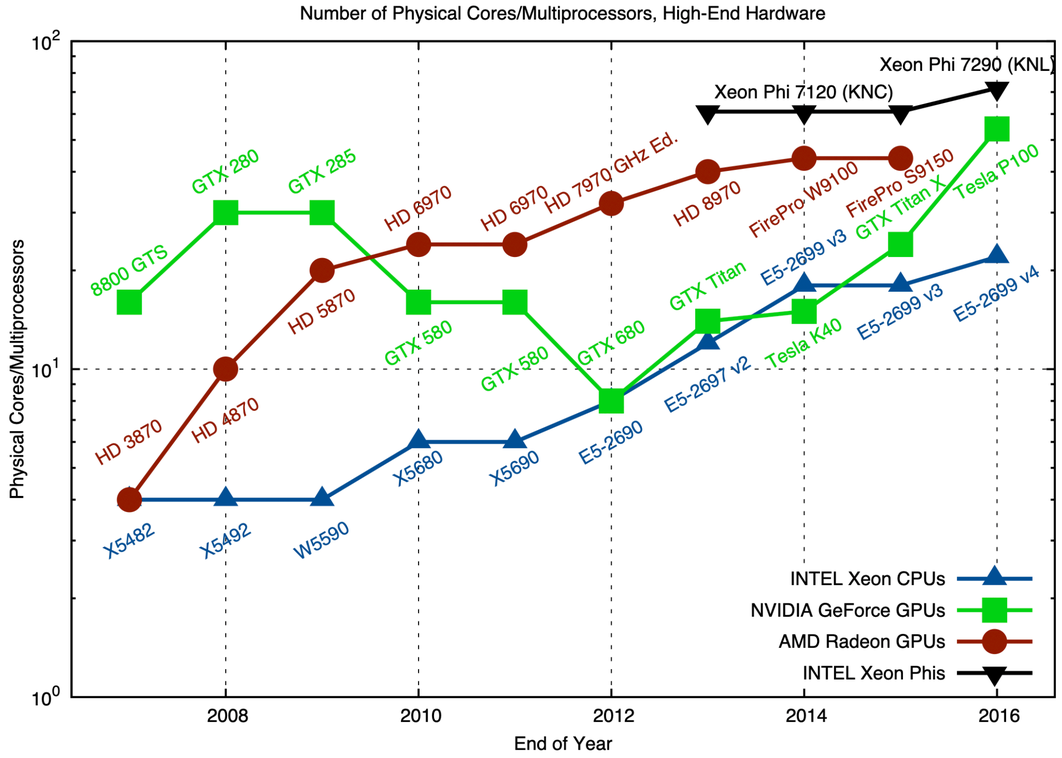
GPU for scientific computing

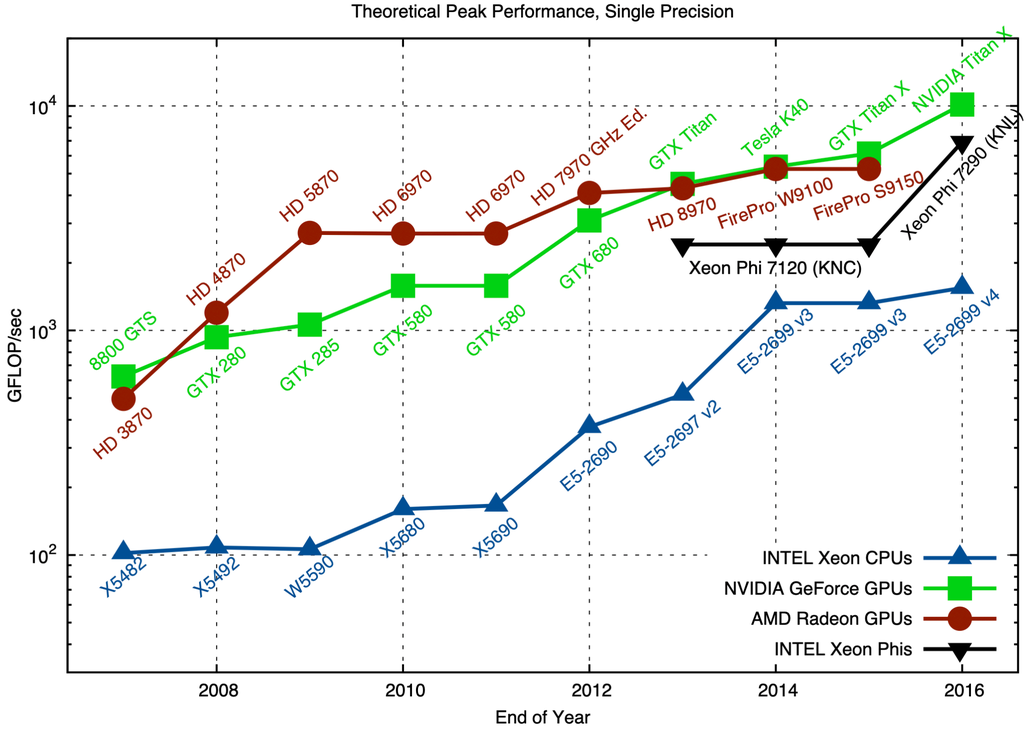
GPUs from NVIDIA and AMD boast significant single and double precision **performance due to a huge number of cores.**

These are specialized processors that can deliver high-performance but only on certain types of calculations:

- **Massive** amount of parallel operations
- Must be mainly **data parallel**
- Requires offloading large chunks of computations to the GPU to amortize the cost of **transferring** data to/from the GPU.

Performance trends





Karl Rupp

https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

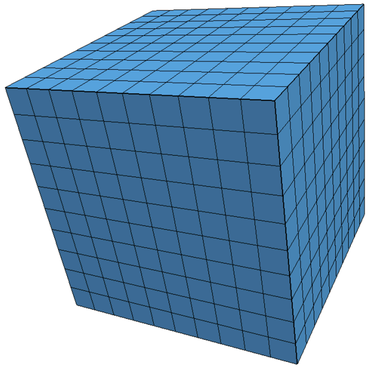
History

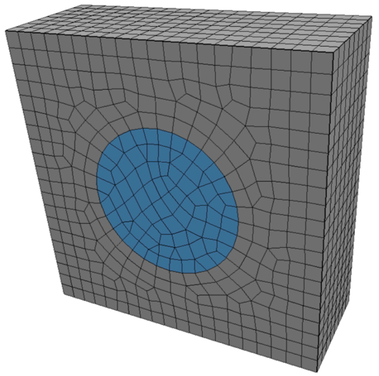
- GPUs initially focused on 3D graphics = computing the color of each pixel on the screen based on a 3D scene model.
- Featured example: ray tracing global illumination (RTXGI)

GPGPU

General purpose GPU computing:

- Extension to general scientific computations.
- Solving equations on a grid is similar to rendering: perform the same regular calculations on a large dataset.



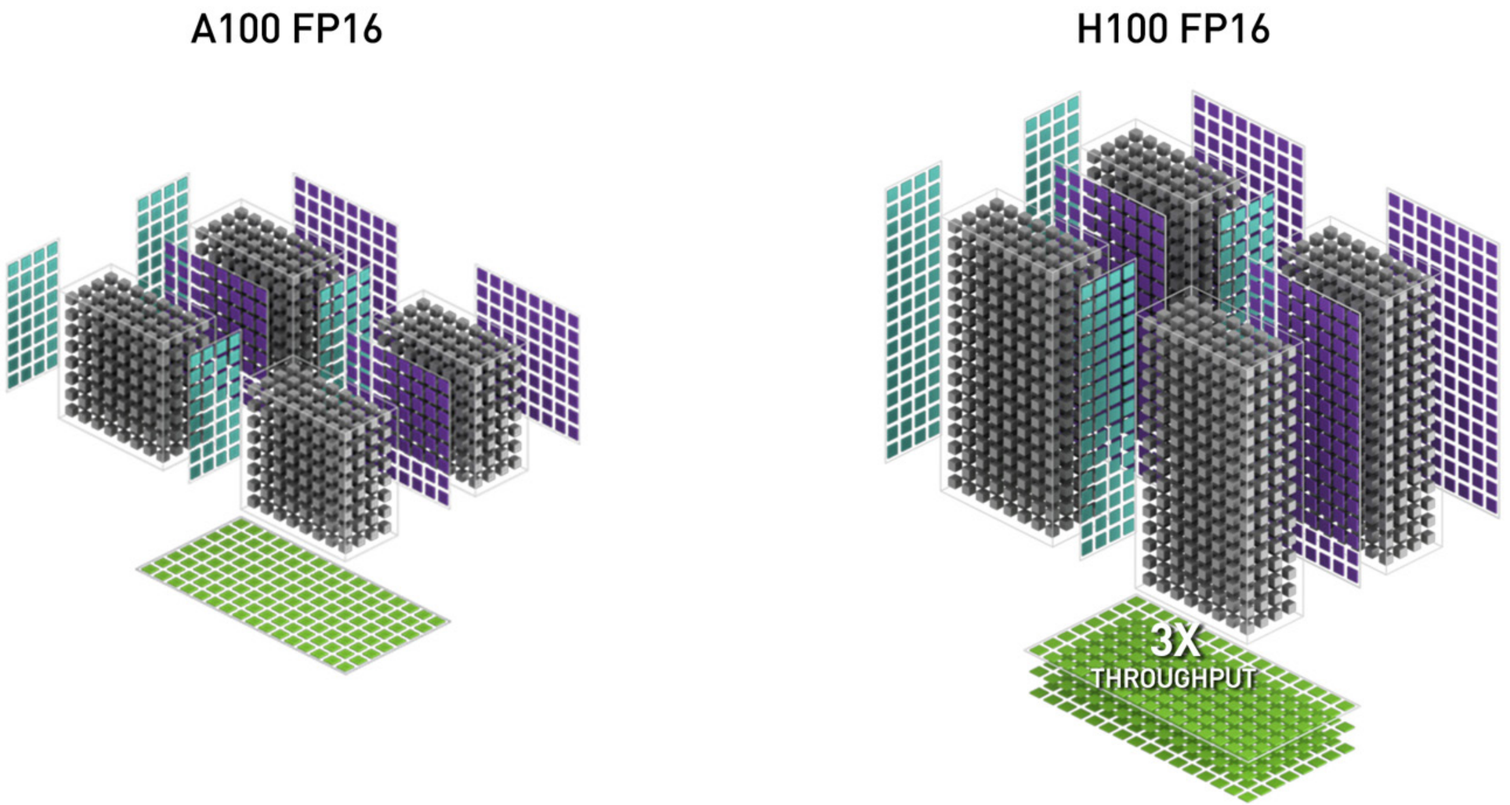


Deep learning

Recent advances in GPU computing target deep learning.

1. **Linear algebra:** matrix-matrix multiplications.
2. **Mixed precision arithmetic:** represents floating point numbers using different numbers of binary bits.

Tensor Cores



H100 FP16 Tensor Core has 3x throughput compared to A100 FP16 Tensor Core

GPUs are great for

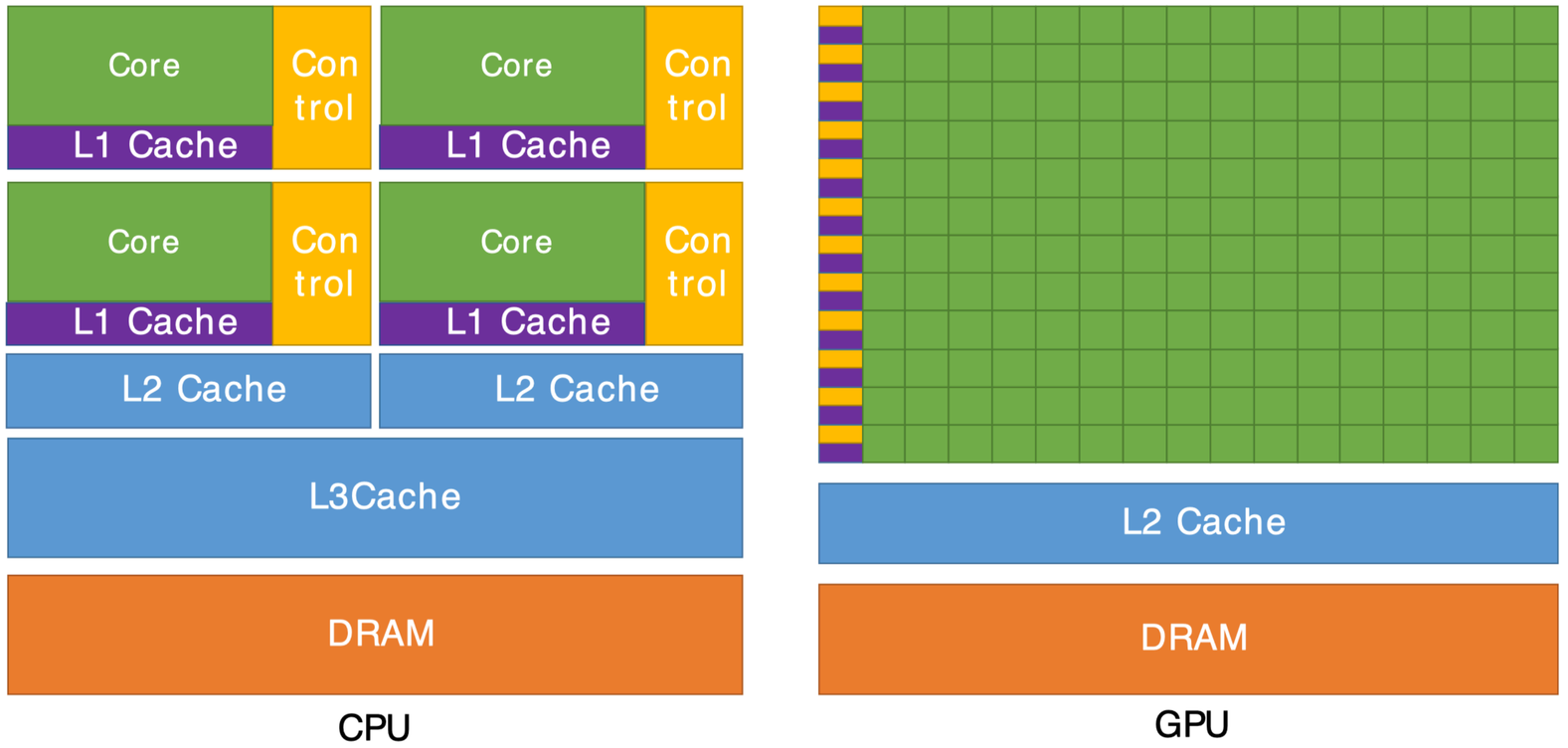
- **Dense linear algebra** with a massive amount of flops
- **Partial differential equation solvers:** finite-difference and regular grid calculations
- **Deep neural networks**

Less suitable for:

- **Irregular** calculations with branching and uneven workloads
- Long series of **sequential** operations

What does a GPU processor look like?

Schematic organization



The GPU devotes more transistors to data processing.

GH100 (Hopper) with 144 Streaming Multiprocessors (SM)

NVLink allows GPU processors to communicate without using the CPU.

GH100 Streaming Multiprocessor (SM)

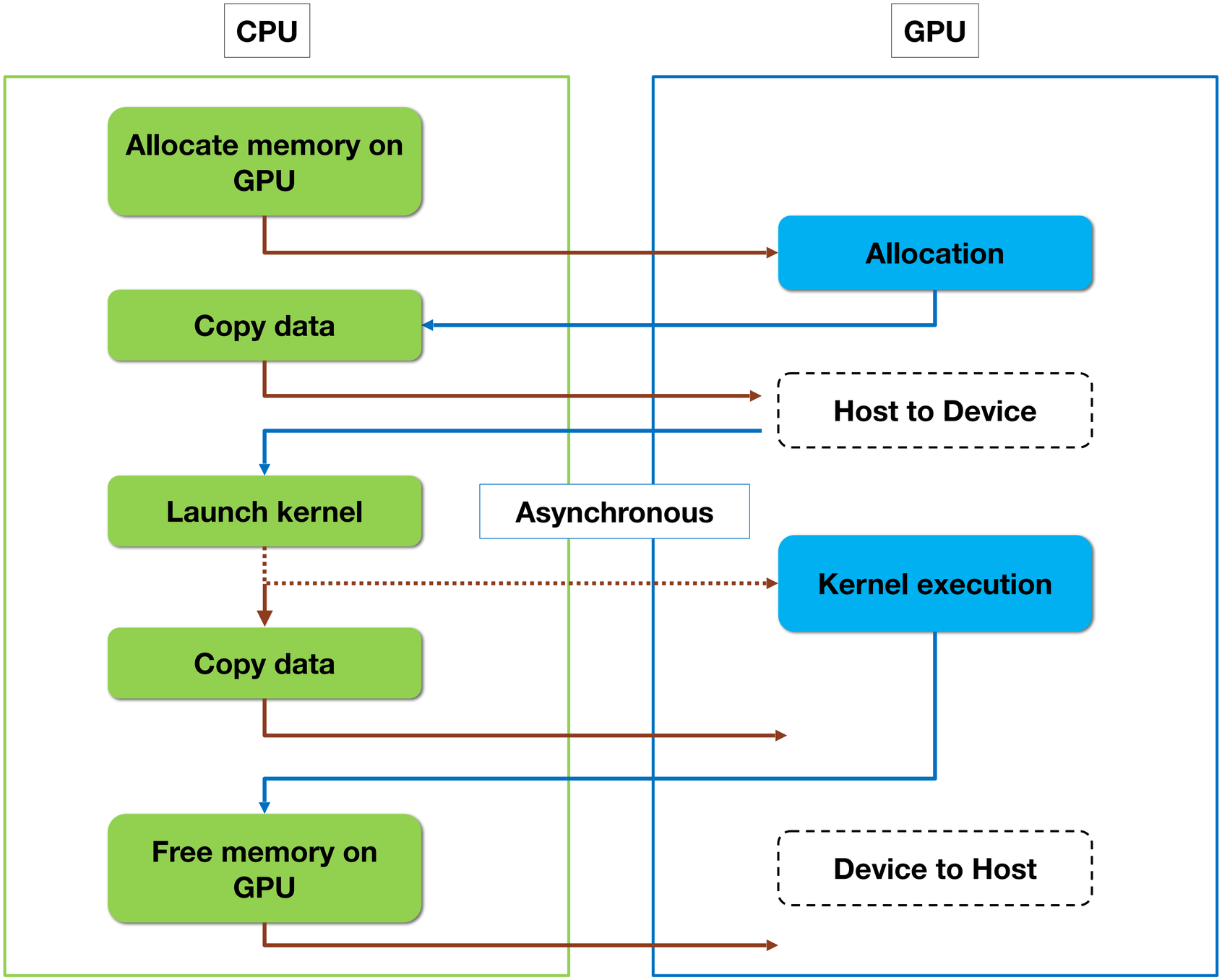
- *Special Functions Units* (SFUs): execute transcendental instructions such as sin, cosine, reciprocal, and square root.
- *Dispatch Unit:* instruction dispatch

How to program GPUs
Introduction

GPU processors are co-processors

- GPUs are different from conventional processors.
- They only work as co-processors.
- This means you need a host processor (e.g., Intel Core/Xeon).

- Your program runs on the host and uses an application programming interface (API) to move data back and forth to the GPU and run programs on the GPU.
- You cannot log on to the GPU directly or run an OS on the GPU.


CPU
GPU
Allocate memory on GPU
Allocation
Copy data
Host to Device
Launch kernel
Asynchronous
Kernel execution
Copy data
Free memory on GPU
Device to Host

OpenACC

- OpenMP can be used to program GPUs, but this is a recent, less robust addition to the language.
- Support is currently somewhat limited.
- We will cover instead **OpenACC,** which was designed from the beginning to program GPUs.
- OpenACC = OpenMP for **accelerator** processor.

Other programming solutions

Vendor agnostic:

- OpenCL
- Numba

Vendor specific:

- CUDA
- HIP

OpenCL

- OpenCL: Open Computing Language
- Framework for writing programs that execute across heterogeneous platforms, e.g., GPUs, digital signal processors (DSPs)
- OpenCL provides a standard interface for parallel computing using task- and data-based parallelism.
- OpenCL is an open standard maintained by the non-profit technology consortium Khronos Group.

Python

Python has extensions that allow generating GPU code.

Example: **Numba**

Numba

Many uses.

Just-in-time compilation of Python code for performance:

```
from numba import jit
import random

@jit(nopython=True)
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

Numba multi-threaded programming

Example of **parallel** for loop in Numba:

```
@jit(nopython=True, parallel=True)
def simulator(out):
    # iterate loop in parallel
    for i in prange(out.shape[0]):
        out[i] = run_sim()
```

Numba for NVIDIA GPUs (CUDA)

```
from numba import cuda, float32

@cuda.jit
def matmul(A, B, C):
    i, j = cuda.grid(2)
    if i < C.shape[0] and j < C.shape[1]:
        tmp = 0.
        for k in range(A.shape[1]):
            tmp += A[i, k] * B[k, j]
        C[i, j] = tmp
```

Numba for AMD ROC GPUs

```
from numba import roc, float32

@roc.jit
def matmul(A, B, C):
    i = roc.get_global_id(0)
    j = roc.get_global_id(1)
    if i < C.shape[0] and j < C.shape[1]:
        tmp = 0.
        for k in range(A.shape[1]):
            tmp += A[i, k] * B[k, j]
        C[i, j] = tmp
```

Vendor specific solutions

NVIDIA CUDA

- Currently, the standard for writing GPU code.
- It only targets **NVIDIA GPUs.**
- **Very mature and robust.**
- But **complex** to use and requires significant code changes
- Initial release: June 23, 2007 (14 years ago)

AMD GPUs

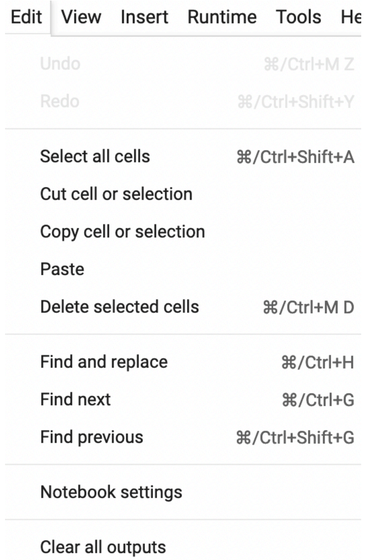
- AMD has a proprietary language for programming its GPUs called **HIP.**
- HIP can generate code for **AMD and NVIDIA GPUs.**
- HIP is close to CUDA.
- HIP is designed to allow developers to convert CUDA code easily.
- Part of the open-source **ROCm** stack.

Let’s get started with OpenACC

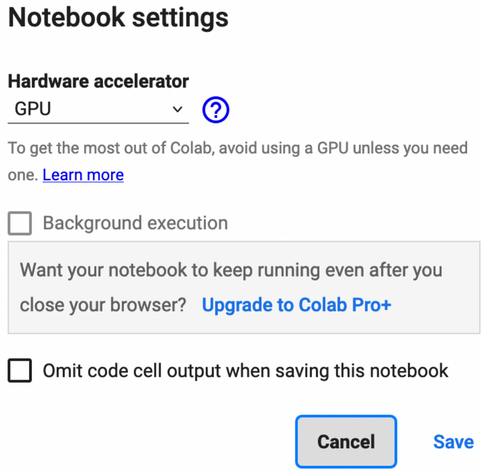
GPU access

First, you will need a GPU!

Go to: Notebook settings



Then select GPU under Hardware accelerator



Demo: `openacc_lab.ipynb`

Hardware

You should be able to test which GPU you have access to:

```
[2]   1 !nvidia-smi --query-gpu=gpu_name,gpu_bus_id,vbios_version --format=csv

    name, pci.bus_id, vbios_version
    Tesla T4, 00000000:00:04.0, 90.04.A7.00.01
```

Demo: `openacc_lab.ipynb`

Installation

You will need to install the NVIDIA HPC SDK, which contains the OpenACC compiler and can generate code for NVIDIA GPUs.

Run the cells at the beginning of the notebook to install:

1. `install_hpc.sh` installs the NVIDIA HPC SDK
2. `install_gtest.sh` installs Google Test

The installation takes a few minutes.

Our first GPU parallel loop

Very similar to OpenMP (at least in appearance)

```
  const int n = 32000000;
  float* x = new float[n];

#pragma acc parallel loop
  for (int i = 0; i < n; ++i) x[i] = i;

  for (int i = 0; i < n; ++i) ASSERT_EQ(x[i], float(i));
```

Offloads to GPU

Demo: `openacc_lab.ipynb`

Let's see what the compiler says

```
acc_lab.cpp:
accTest_loop_Test::TestBody():
      7, Generating NVIDIA GPU code
         10, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
      7, Generating implicit copyout(x[:32000000]) [if not already present]
```

What are these messages saying?

Let's focus on gang/vector first.

Gang, worker, vector

This is less important for us.

These variables are used because they reflect the way the hardware is organized.

Thread: smallest execution unit in the program.

Gang, worker, vector

Vector: a group of threads that can coordinate and execute "together."

Worker: a group of vectors that can coordinate and execute "together;" this is a less important concept.

Gang: a group of workers/vectors.

Gang, worker, vector

For optimization purposes, the sizes of a vector, worker, or gang can be specified.

This is a more advanced optimization.

```
#pragma acc parallel loop num_gangs(40) num_workers(32) vector_length(32)
  for (int i = 0; i < n; ++i) x[i] = i;
```

```
accTest_vector_loop_Test::TestBody():
     37, Generating NVIDIA GPU code
         40, #pragma acc loop gang(40), worker(32), vector(32) /* blockIdx.x threadIdx.y threadIdx.x */
     37, Generating implicit copyout(x[:32000000]) [if not already present]
```

CPU and GPU memories

- The memory of the CPU and the GPU are physically **separate.**
- So data need to be transferred between the two memories before a calculation can be run on the GPU.
- In this case, the compiler detected that x was initialized on the GPU.
- It automatically generated instructions to copy the result from the GPU to the CPU memory.

Explicit data transfer clauses

```
#pragma acc parallel loop copyout(x[:n])
  for (int i = 0; i < n; ++i) x[i] = i;
```

Clause	Description
`copy`	create space, initialize by copying to the device, copy back to host at the end, release device memory
`copyin`	same but without copy back to host
`copyout`	same but without initial copy to the device
`create`	create space at the beginning, release at the end
`present`	no action taken

Array shaping

```
#pragma acc parallel loop copyout(x[:n])
```

`x[:n]`

- `copy(array[starting_index:length])`
- The first number is the **start index** of the array.
- The second number is **how much data** is to be transferred.

Demo: `openacc_lab.ipynb`

Data locality

- (x,y) initialization
- Vector z computation

```
for (int i = 0; i < n; ++i) {
  x[i] = i;
  y[i] = i * i;
}
for (int i = 0; i < n; ++i) {
  ASSERT_EQ(x[i], float(i));
  ASSERT_EQ(y[i], float(i * i));
}
for (int i = 0; i < n; ++i) z[i] = x[i] + y[i];
for (int i = 0; i < n; ++i) ASSERT_EQ(z[i], float(i * (i + 1)));
```

Demo: `openacc_lab.ipynb`

Data transfer

By default, the code would:

- Copy (x,y) to the device.
- Copy them back to the host for testing.
- Copy (x,y) again to the device.
- Copy z to the host.

The copies of (x,y) can be optimized.

acc enter data

```
#pragma acc enter data create(x[:n], y[:n])
#pragma acc parallel loop
  for (int i = 0; i < n; ++i) {
    x[i] = i;
    y[i] = i * i;
  }
```

`enter data create`

Creates the data on the GPU and leaves it there until instructed to delete the data.

acc update self

```
#pragma acc update self(x[:n], y[:n])
  for (int i = 0; i < n; ++i) {
    ASSERT_EQ(x[i], float(i));
    ASSERT_EQ(y[i], float(i * i));
  }
```

- Because we used `enter data create`, the data is not automatically copied back to the host.
- We need to add `update self` to copy the data from device to host.
- Also: `#pragma acc update device()`

acc exit data

```
#pragma acc exit data delete(x[:n], y[:n])
```

This deletes the data on the GPU and frees resources.

Each `enter` much be matched with an `exit`.

Full code

```
#pragma acc enter data create(x[:n], y[:n])
#pragma acc parallel loop
  for (int i = 0; i < n; ++i) {
    x[i] = i;
    y[i] = i * i;
  }

#pragma acc update self(x[:n], y[:n])
  for (int i = 0; i < n; ++i) {
    ASSERT_EQ(x[i], float(i));
    ASSERT_EQ(y[i], float(i * i));
  }

#pragma acc parallel loop copyout(z[:n])
  for (int i = 0; i < n; ++i) z[i] = x[i] + y[i];

#pragma acc exit data delete (x[:n], y[:n])

  for (int i = 0; i < n; ++i) ASSERT_EQ(z[i], float(i * (i + 1)));
```

Reduction

As in OpenMP, we need to use a special construct when we have a reduction.

```
sum += x[i];
```

Otherwise, the different cores will attempt to access and modify `sum` at the same time, **which is a bug.**

Reduction example

```
  float sum = 0;
#pragma acc data create(x[:n])
  {
#pragma acc parallel loop
    for (int i = 0; i < n; ++i) x[i] = 1;
#pragma acc parallel loop reduction(+ : sum)
    for (int i = 0; i < n; ++i) sum += x[i];
  }
  for (int i = 0; i < n; ++i) ASSERT_EQ(sum, float(n));
```

Note the `data create` to avoid unnecessary copies.

collapse

- GPU are massively parallel processors.
- They can contain thousands of cores.
- Example: GeForce RTX 3090 Ti: **10,752 cores.**
- So we need to generate as much concurrency (parallelism) in our code.
- **Loop fusion** is critical for performance when the number of iterations is not large enough.

for loop with collapse

```
#pragma acc parallel loop collapse(2)
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      z[i * n + j] = x[i * n + j] + y[i * n + j];
    }
  }
```

- i and j loops will be merged and executed in parallel.
- Without collapse, only the i loop is parallelized while the j loop is executed sequentially.
- **`collapse` allows generating n*n parallel threads instead of just n.**
- This can improve **performance** significantly.

Complete example with collapse

```
#pragma acc enter data create(x[:n * n], y[:n * n])
#pragma acc parallel loop collapse(2)
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j) {
      x[i * n + j] = i * n + j;
      y[i * n + j] = i - j;
    }

#pragma acc update self(x[:n * n], y[:n * n])
  for (int i = 0; i < n * n; ++i) ASSERT_EQ(x[i], float(i));
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j) ASSERT_EQ(y[i * n + j], float(i - j));

#pragma acc parallel loop collapse(2) copyout(z[:n * n])
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j) z[i * n + j] = x[i * n + j] + y[i * n + j];

#pragma acc exit data delete(x[:n * n], y[:n * n])
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j) ASSERT_EQ(z[i * n + j], float(i * (n + 1)));
```

n body problem

Let's look at a more complex real-life application.

We want to model the gravitational interactions between n bodies with mass.

This is similar to modeling the motions of the planets around the sun in the solar system.

Gravitational force

We start from $F_i = m_i a_i$.

The acceleration is given by the gravitational force:

$$F_i = m_i \sum_{j \neq i} m_j \frac{r_j - r_i}{\|r_j - r_i\|_2^3}$$

Equations of motion

$$F_i = m_i \sum_{j \neq i} m_j \frac{r_j - r_i}{\|r_j - r_i\|_2^3}$$

Equations of motion:

$$\frac{d^2 r_i}{dt^2} = \sum_{j \neq i} m_j \frac{r_j - r_i}{\|r_j - r_i\|_2^3}$$

Time integrator

- We numerically solve these equations using the **velocity Verlet** time integrator.
- It's not very accurate, but it remains stable over many time steps.
- This is a two-step method.

Velocity Verlet

- **Step 1:** advance the velocity

$$v_i^{n+1} = v_i^n + \Delta t \sum_{j \neq i} m_j \frac{r_j - r_i}{\|r_j - r_i\|_2^3} \Bigg|_n$$

- **Step 2:** advance the position

$$r_i^{n+1} = r_i^n + \Delta t \, v_i^{n+1}$$

- **Repeat**

Force computation

```
for (int i = 0; i < n; i++) {
  real fx, fy, fz; fx = fy = fz = 0;
  for (int j = 0; j < n; j++) {
    real3 ff = forceComputation(pos[i].x, pos[i].y, pos[i].z,
                                pos[j].x, pos[j].y, pos[j].z, pos[j].w);
    fx += ff.x;
    fy += ff.y;
    fz += ff.z;
  }
  force[i].x = fx;
  force[i].y = fy;
  force[i].z = fz;
}
```

Demo: `nbody.ipynb`

Time step

```
  for (int i = 0; i < n; i++) {
    // acceleration = force / mass;
    // new velocity = old velocity + acceleration * deltaTime
    vel[i].x += force[i].x * dt;
    vel[i].y += force[i].y * dt;
    vel[i].z += force[i].z * dt;

    // new position = old position + velocity * deltaTime
    pos[i].x += vel[i].x * dt;
    pos[i].y += vel[i].y * dt;
    pos[i].z += vel[i].z * dt;
  }
```

Demo: `nbody.ipynb`

Time integration

```
for (int i = 0; i < iterations; i++) {
  seqIntegrate(pos, vel, force, dt, n);
}
```

Demo: `nbody.ipynb`

Parallel time stepping loop

```
#pragma acc data copy(pos[:n], vel[:n]) copyout(force[:n])
  for (int i = 0; i < iterations; i++) {
    integrate(pos, vel, force, dt, n);
  }
```

- Optimize the movement of data by **reducing memory copies between host and device.**
- Only done before the iterations start and after they are complete.

Nested parallel loops with reduction

```
#pragma acc parallel loop
  for (int i = 0; i < n; i++) {
    real fx, fy, fz;
    fx = fy = fz = 0;
#pragma acc loop reduction(+ : fx, fy, fz)
    for (int j = 0; j < n; j++) {
      real3 ff = forceComputation(pos[i].x, pos[i].y, pos[i].z, pos[j].x,
                                  pos[j].y, pos[j].z, pos[j].w);
      fx += ff.x;
      fy += ff.y;
      fz += ff.z;
    }
    force[i].x = fx;
    force[i].y = fy;
    force[i].z = fz;
  }
```

Demo: `nbody.ipynb`

Parallel time step

```
#pragma acc parallel loop
  for (int i = 0; i < n; i++) {
    vel[i].x += force[i].x * dt;
    vel[i].y += force[i].y * dt;
    vel[i].z += force[i].z * dt;
    pos[i].x += vel[i].x * dt;
    pos[i].y += vel[i].y * dt;
    pos[i].z += vel[i].z * dt;
  }
```

Demo: `nbody.ipynb`

Performance results

```
1 !name=nbody; nvc++ -I. -acc=host -O -o $name $name.cpp && ./$name 4096 20
2 !name=nbody; nvc++ -I. -acc=multicore -O -o $name $name.cpp && ./$name 4096 20
3 !name=nbody; nvc++ -I. -acc=gpu -O -o $name $name.cpp && ./$name 4096 20
4 !name=nbody; nvc++ -I. -mp=multicore -O -o $name $name.cpp && ./$name 4096 20
5 !name=nbody; g++ -std=c++17 -I. -O -o $name $name.cpp && ./$name 4096 20
```

```
n = 4096 bodies for 20 iterations
OpenACC:        2185.000000 ms: 3.071344 GFLOP/s
Sequential:     2206.000000 ms: 3.042106 GFLOP/s
n = 4096 bodies for 20 iterations
OpenACC:        1959.000000 ms: 3.425669 GFLOP/s
Sequential:     2197.000000 ms: 3.054568 GFLOP/s
n = 4096 bodies for 20 iterations
OpenACC:        380.000000 ms: 17.660228 GFLOP/s
Sequential:     2155.000000 ms: 3.114100 GFLOP/s
n = 4096 bodies for 20 iterations
OpenMP:         2197.000000 ms: 3.054568 GFLOP/s
Sequential:     2169.000000 ms: 3.094000 GFLOP/s
n = 4096 bodies for 20 iterations
c++:            4257.000000 ms: 1.576436 GFLOP/s
Sequential:     4257.000000 ms: 1.576436 GFLOP/s
```

We only use a **single CPU thread** on Google compute.

```
 1 !name=test_nbody; nvc++ -I. -acc=gpu -o $name $name.cpp gtest_main.a && ./$name
```

```
test_nbody.cpp:
Running main() from googletest-main/googletest/src/gtest_main.cc
[==========] Running 5 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 5 tests from nbodyTest
[ RUN      ] nbodyTest.iterations_small_0
8 tests PASSED. Maximum error = 0.
6 tests PASSED. Maximum error = 0.
6 tests PASSED. Maximum error = 0.
[       OK ] nbodyTest.iterations_small_0 (259 ms)
[ RUN      ] nbodyTest.iterations_small_1
128 tests PASSED. Maximum error = 1.13687e-13.
96 tests PASSED. Maximum error = 9.31323e-10.
96 tests PASSED. Maximum error = 7.45058e-09.
[       OK ] nbodyTest.iterations_small_1 (0 ms)
[ RUN      ] nbodyTest.iterations_medium_0
4096 tests PASSED. Maximum error = 2.27374e-13.
3072 tests PASSED. Maximum error = 1.49012e-08.
3072 tests PASSED. Maximum error = 1.49012e-07.
[       OK ] nbodyTest.iterations_medium_0 (14 ms)
[ RUN      ] nbodyTest.iterations_medium_1
4096 tests PASSED. Maximum error = 4.76837e-07.
3072 tests PASSED. Maximum error = 3.8147e-06.
3072 tests PASSED. Maximum error = 1.22935e-07.
[       OK ] nbodyTest.iterations_medium_1 (32 ms)
[ RUN      ] nbodyTest.iterations_large
16384 tests PASSED. Maximum error = 5.96046e-07.
12288 tests PASSED. Maximum error = 3.8147e-06.
12288 tests PASSED. Maximum error = 4.02331e-07.
[       OK ] nbodyTest.iterations_large (457 ms)
[----------] 5 tests from nbodyTest (764 ms total)

[----------] Global test environment tear-down
[==========] 5 tests from 1 test suite ran. (764 ms total)
[  PASSED  ] 5 tests.
```

Accuracy

Roundoff errors between CPU and GPU in single precision

We hope you enjoyed this workshop!