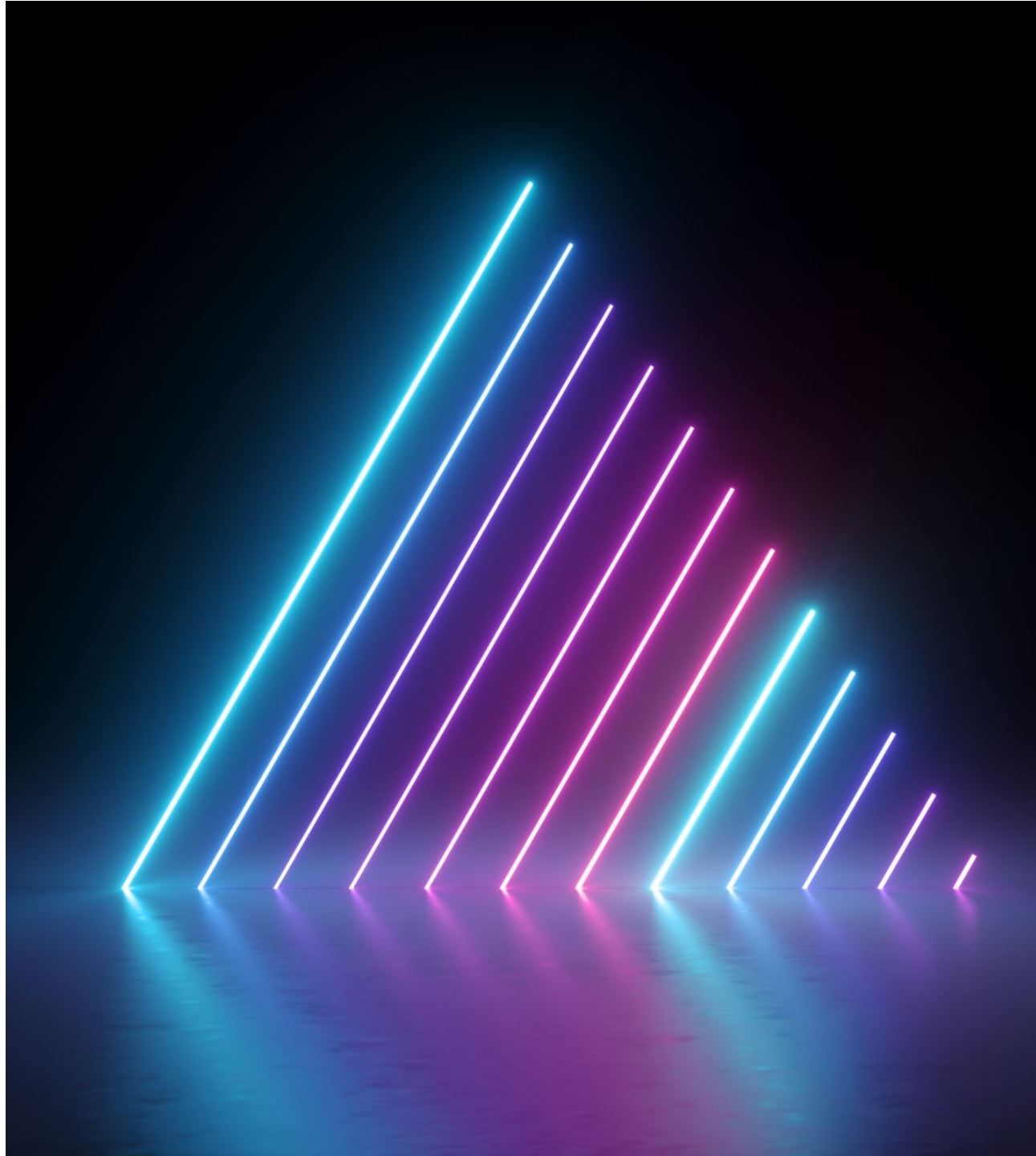
Advanced OpenMP Topics and GPU Programming with OpenMP

by

Kent Milfeld (TACC)

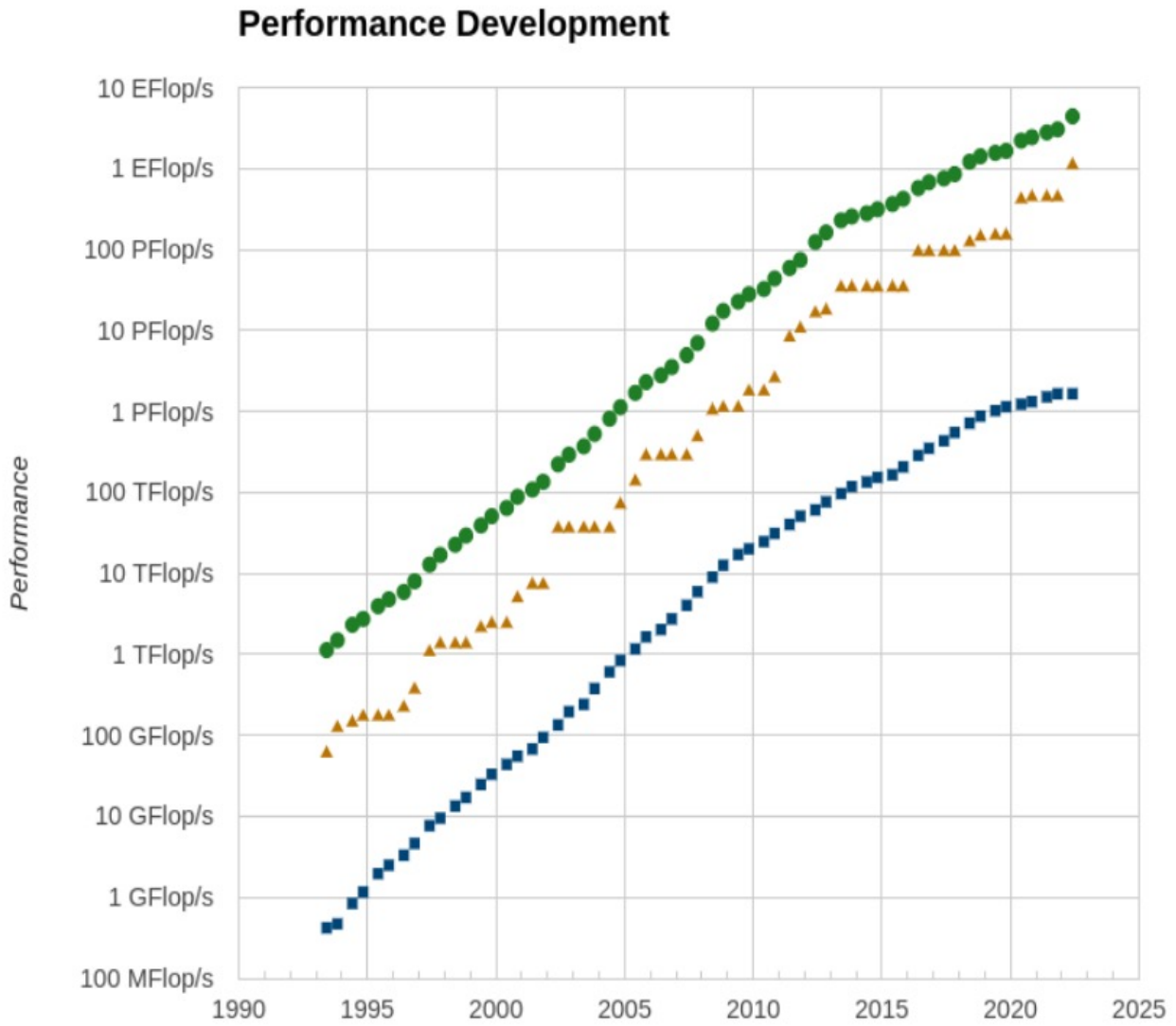
Emanuele Vitali (CSC)

https://ssl.eventilla.com/openmp



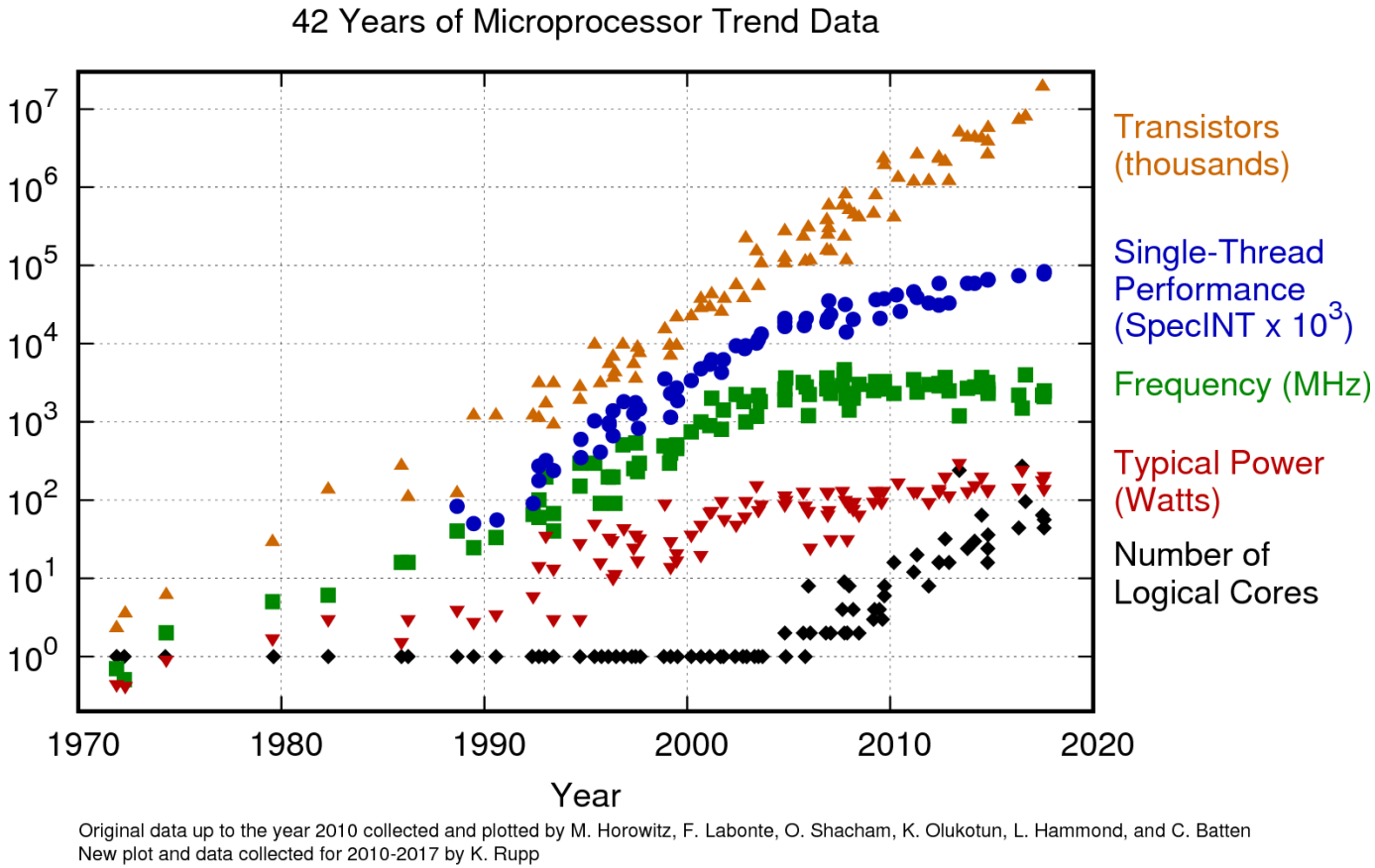
Introduction to GPUs in HPC

- High performance computing is fueled by ever increasing performance
- Increasing performance allows breakthroughs in many major challenges that humankind faces today
- Not only hardware performance, algorithmic improvements have also helped a lot



HPC through the ages

- Various strategies to achieve performances through the years:
  - Frequency, vectorization, multi-node, multi-core ...
  - Now performance is mostly limited by power consumption
- Accelerators provide compute resources with high parallelism to reach high performance at low relative power consumption



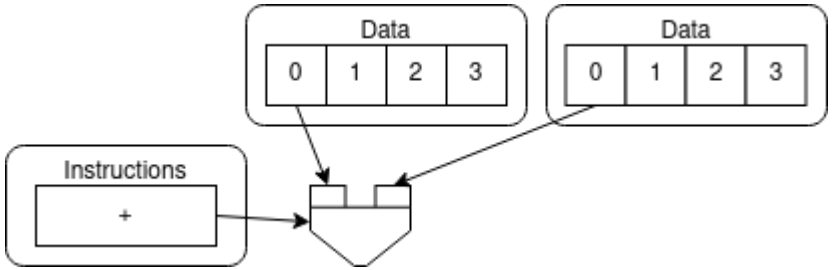
Accelerators

- Specialized parallel hardware for floating point operations
  - Co-processors for traditional CPUs
  - Based on highly parallel architectures
  - Graphics processing units (GPU) have been the most common accelerators during the last few years

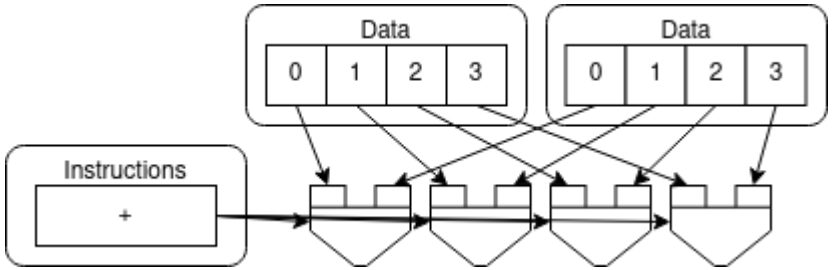
- Promise
  - Very high performance per node
  - Higher FLOPs per Watt

- Usually major rewrites of programs required

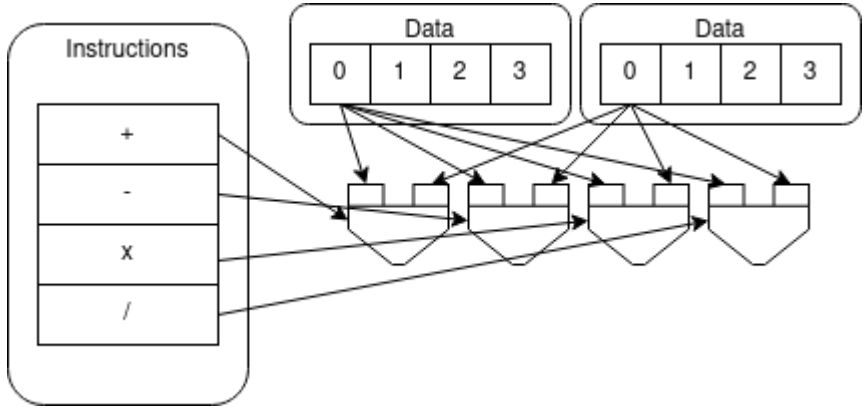
Flynn's taxonomy



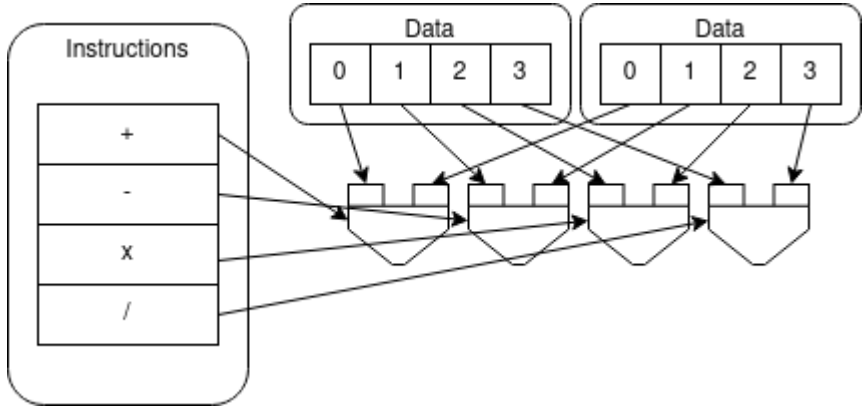
- Single Instruction Single Data (SISD)



- Single Instruction Multiple Data (SIMD)

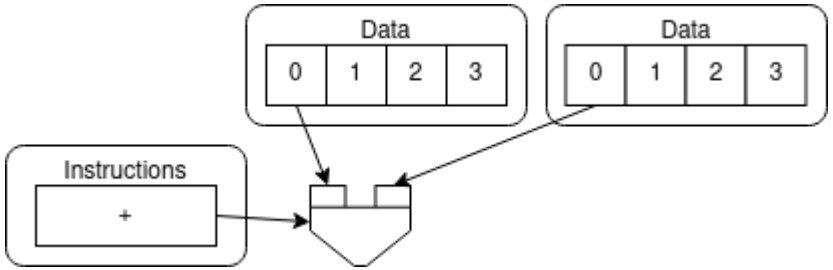


- Multiple Instruction Single Data (MISD)

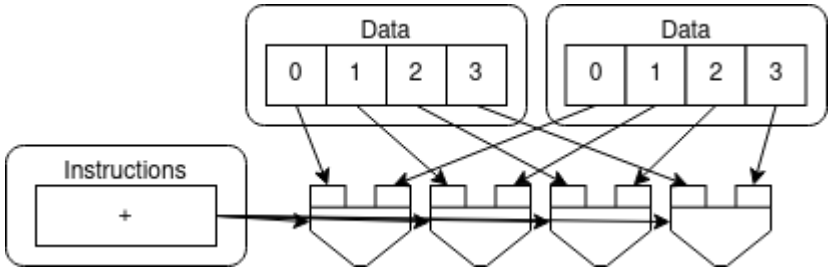


- Multiple Instruction Multiple Data (MIMD)

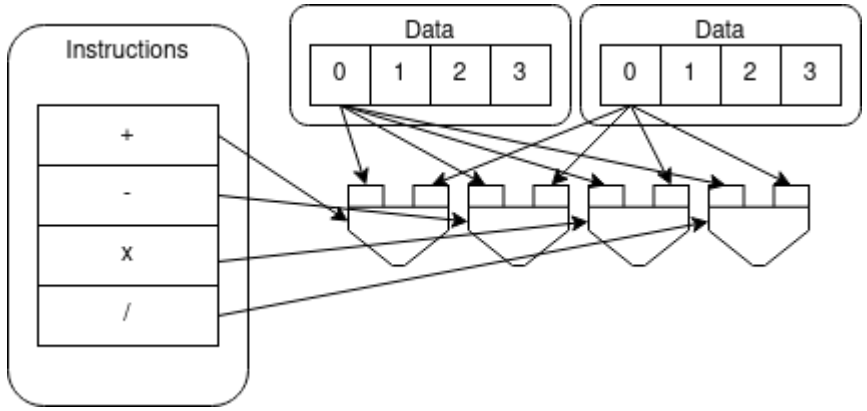
Flynn's taxonomy



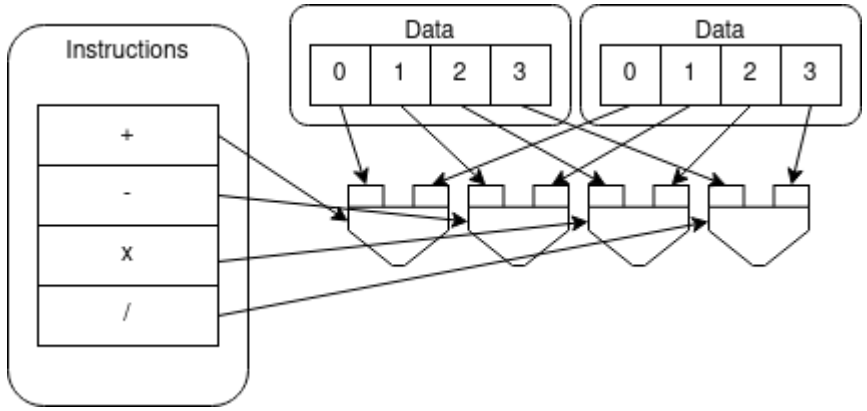
- 1 CPU Core



- GPU



- Not used



- CPU multicore

Different design philosophies: CPU

- General purpose
- Low latency per thread
- Large area dedicated to caches and control
  - Good for control-flow
  - Great for task parallelism (MIMD)
- Less silicon dedicated at Arithmetic-Logic Units (ALU)
  - Bad with parallel execution

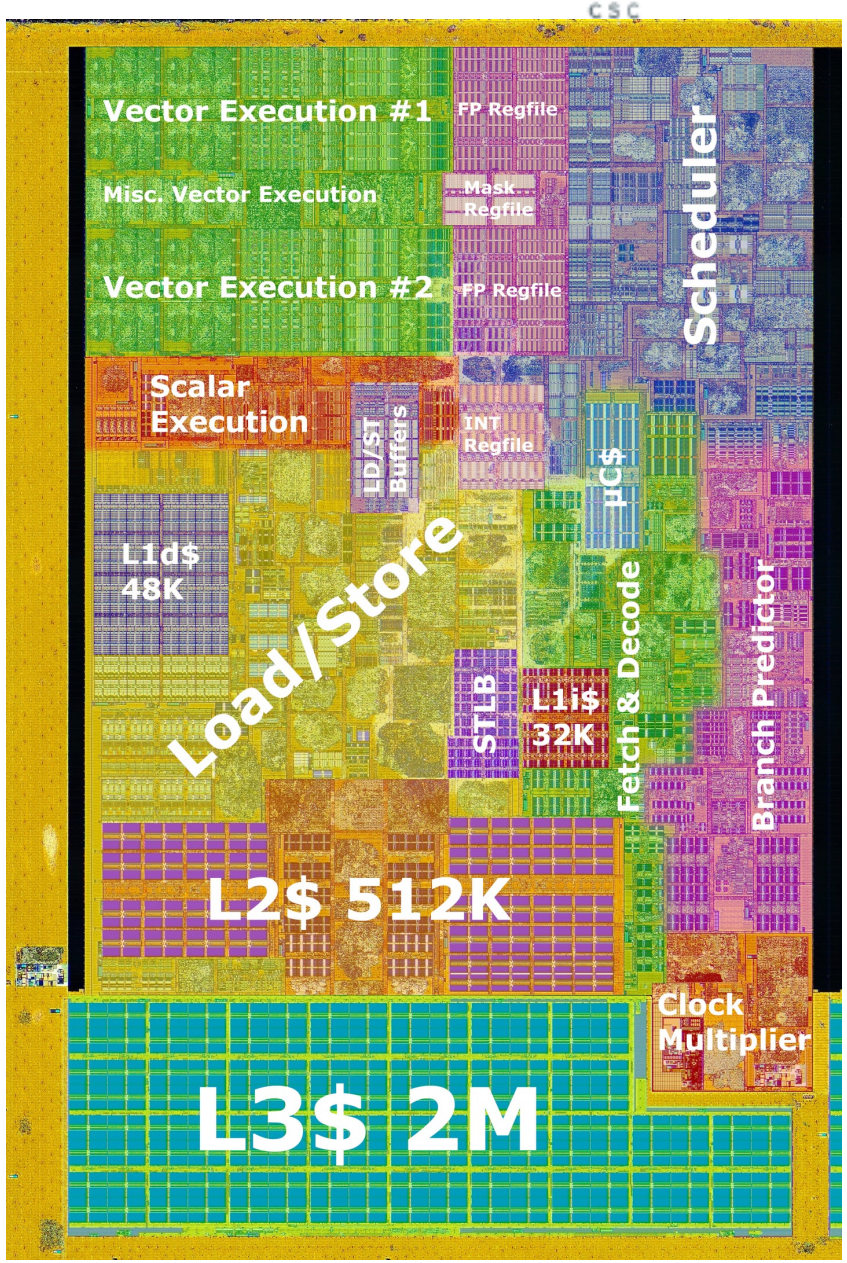


Image credits: https://nemez.net/die/RocketLake

Different design philosophies: GPU

- Most of the silicon dedicated to ALUs
  - Hundreds of floating-point execution units
  - Highly specialized for parallelism
- Great for data parallelism
- High-throughput
- Bad at control-flow processing

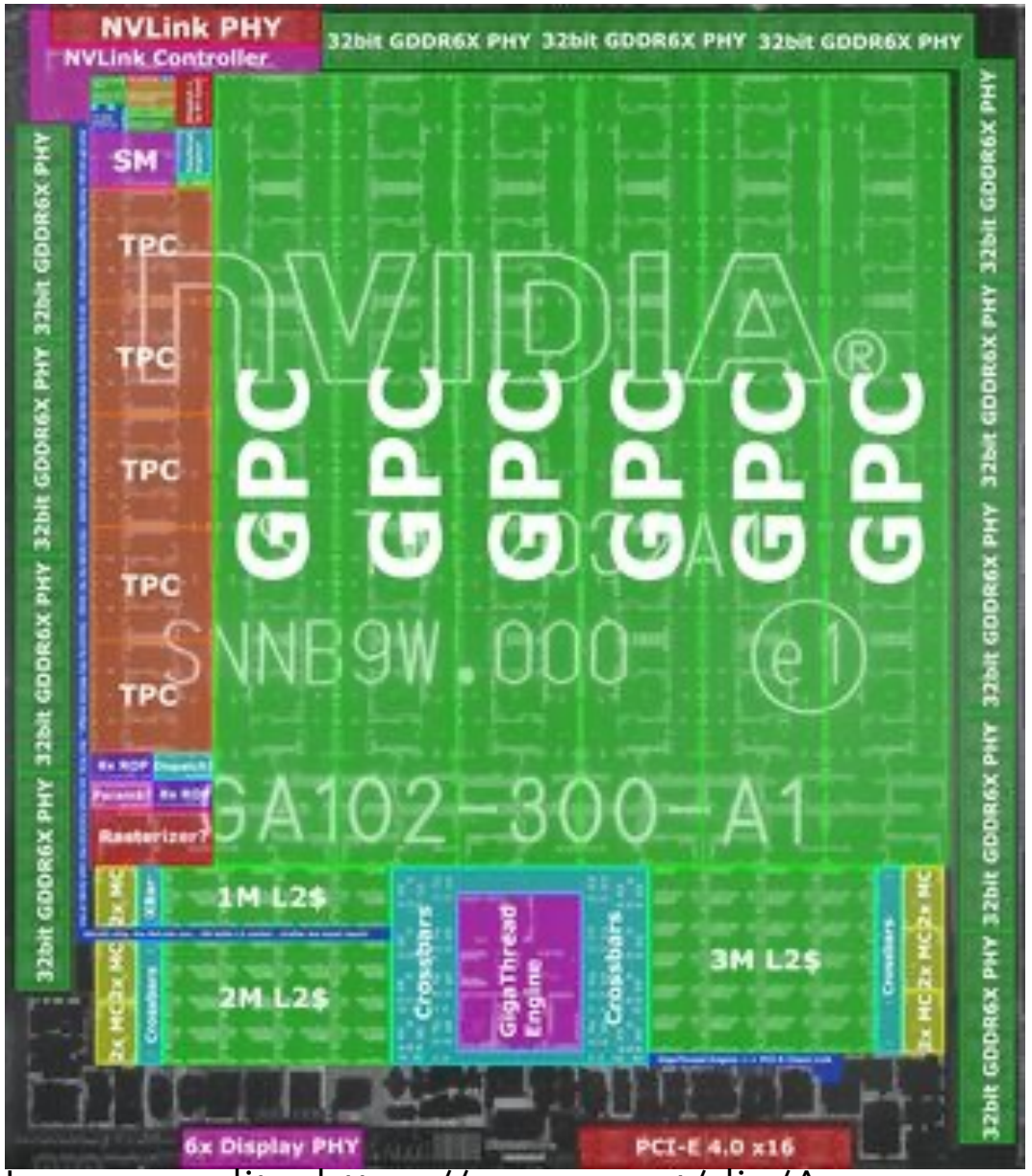
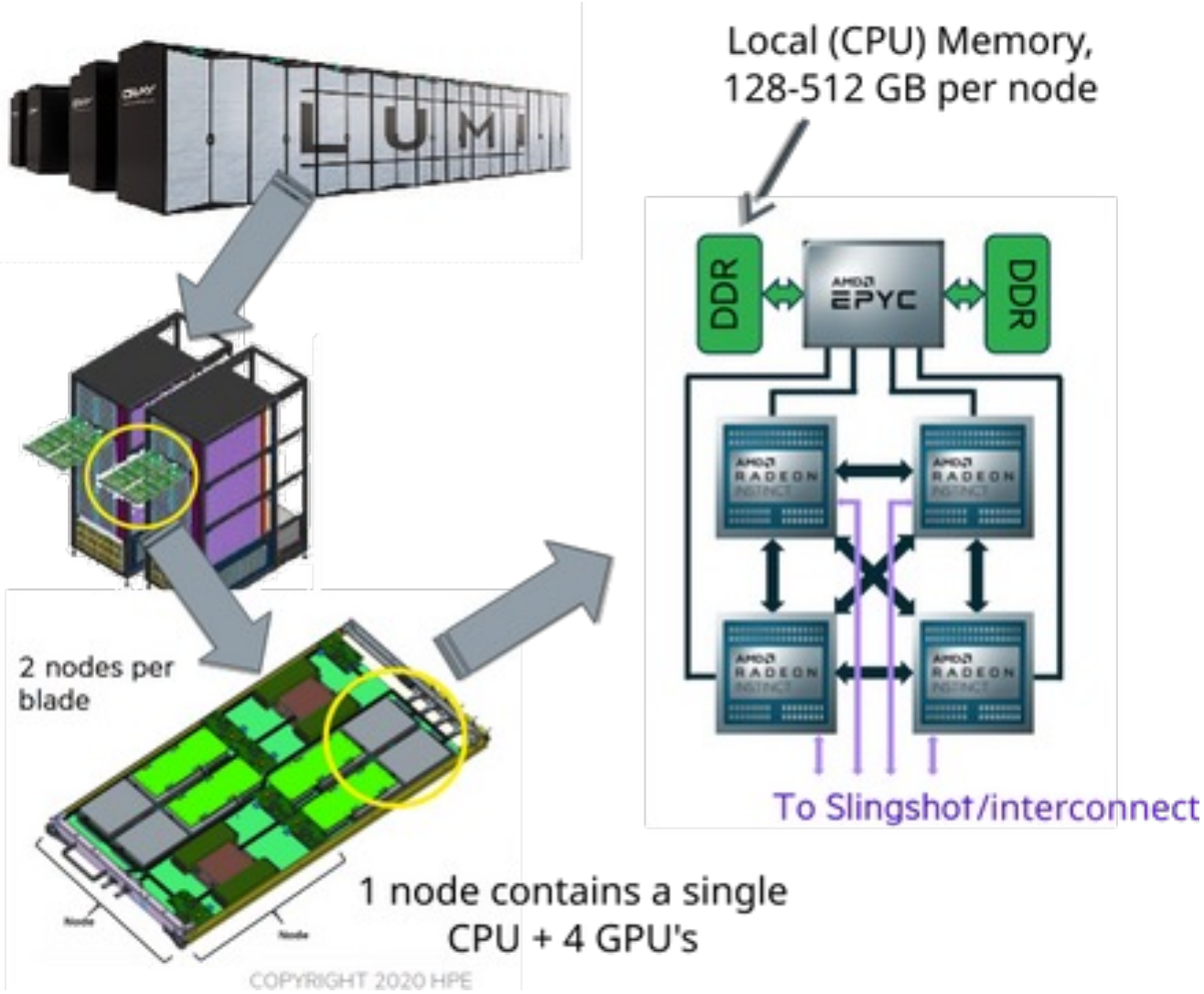


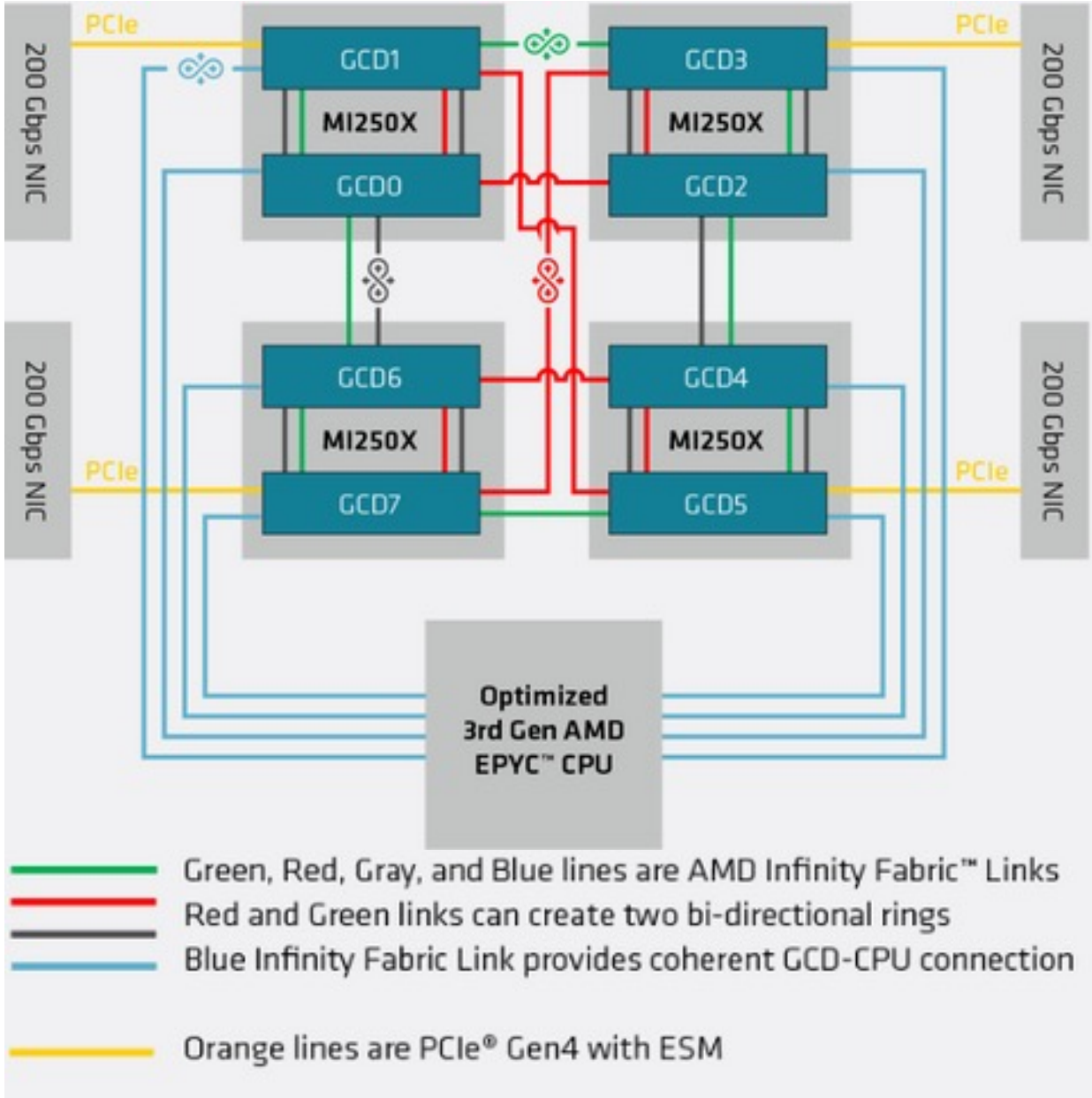
Image credits: https://nemez.net/die/Ampere

Lumi - Pre-exascale system in Finland



Accelerator model in LUMI

- GPU is connected to CPUs via Infinity Fabric
- Local memory in GPU
  - Smaller than main memory (128 GB)
  - Very high bandwidth (up to 3200 GB/s)
  - Latency high compared to compute performance
- Data must be copied from CPU to GPU over the Infinity Fabric

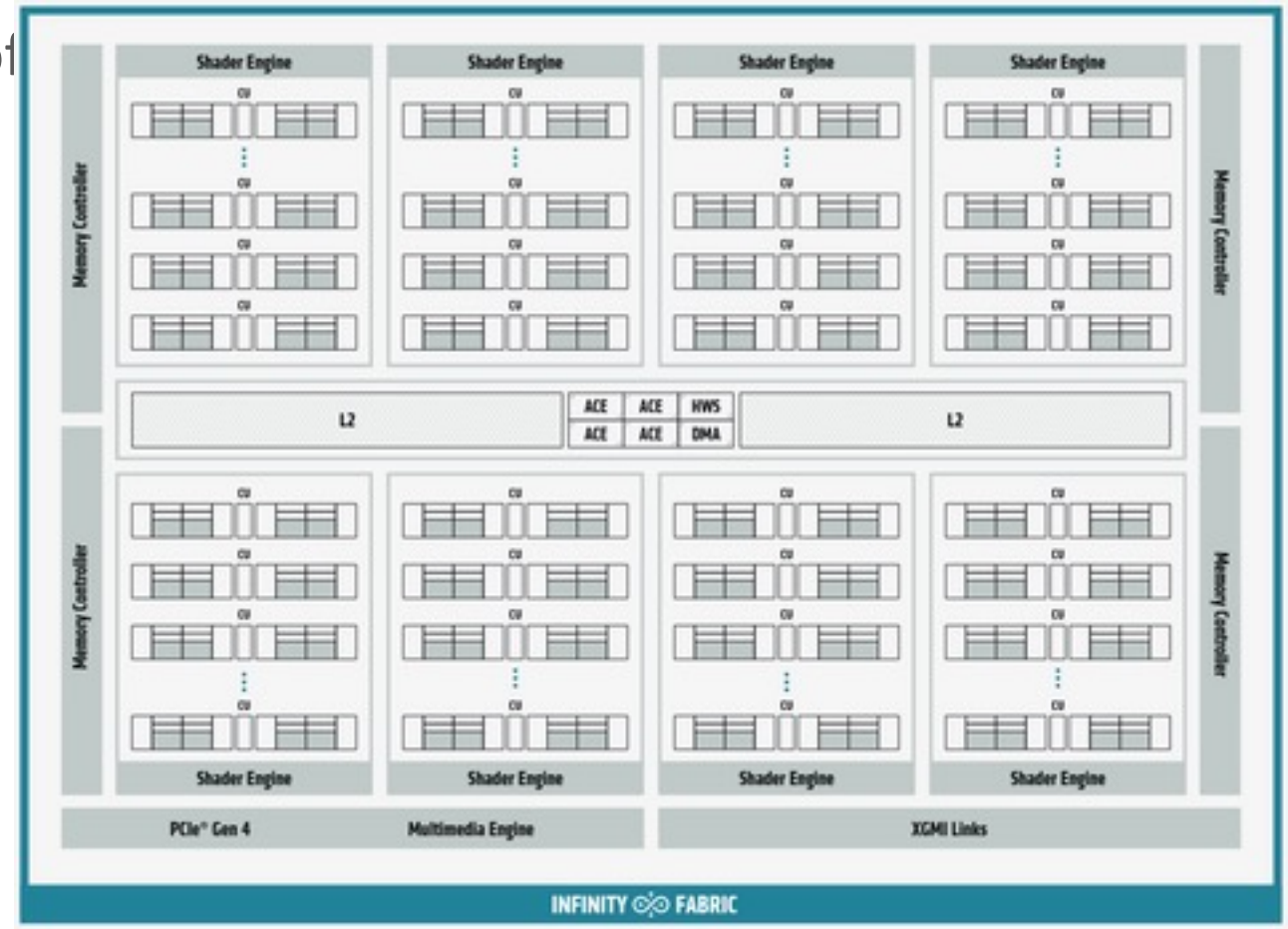


Heterogeneous Programming Model

- GPUs are co-processors to the CPU
- CPU controls the work flow:
  - **offloads** computations to GPU by launching **kernels**
  - allocates and deallocates the memory on GPUs
  - handles the data transfers between CPU and GPUs
- CPU and GPU can work concurrently
  - kernel launches are normally asynchronous

GPU architecture

- Designed for running tens of thousands of threads simultaneously on thousands of cores
- Very small penalty for switching threads
- Running large amounts of threads hides memory access penalties
- Very expensive to synchronize all threads



Advance features & Performance considerations

- Memory accesses:
  - data resides in the GPU memory; maximum performance is achieved when reading/writing is done in continuous blocks
  - very fast on-chip memory can be used as a user programmable cache

- **Unified Virtual Addressing** provides unified view for all memory

- Asynchronous calls can be used to overlap transfers and computations

Challenges in using Accelerators

- **Applicability**: Is your algorithm suitable for GPU?
- **Programmability**: Is the programming effort acceptable?
- **Portability**: Rapidly evolving ecosystem and incompatibilities between vendors.
- **Availability**: Can you access a (large scale) system with GPUs?
- **Scalability**: Can you scale the GPU software efficiently to several nodes?

Advance features & Performance considerations

1. Use existing GPU applications

2. Use accelerated libraries

3. Directive based methods
    1. **OpenMP**, OpenACC

4. Use native GPU language
    1. CUDA, HIP

Directive-based accelerator languages

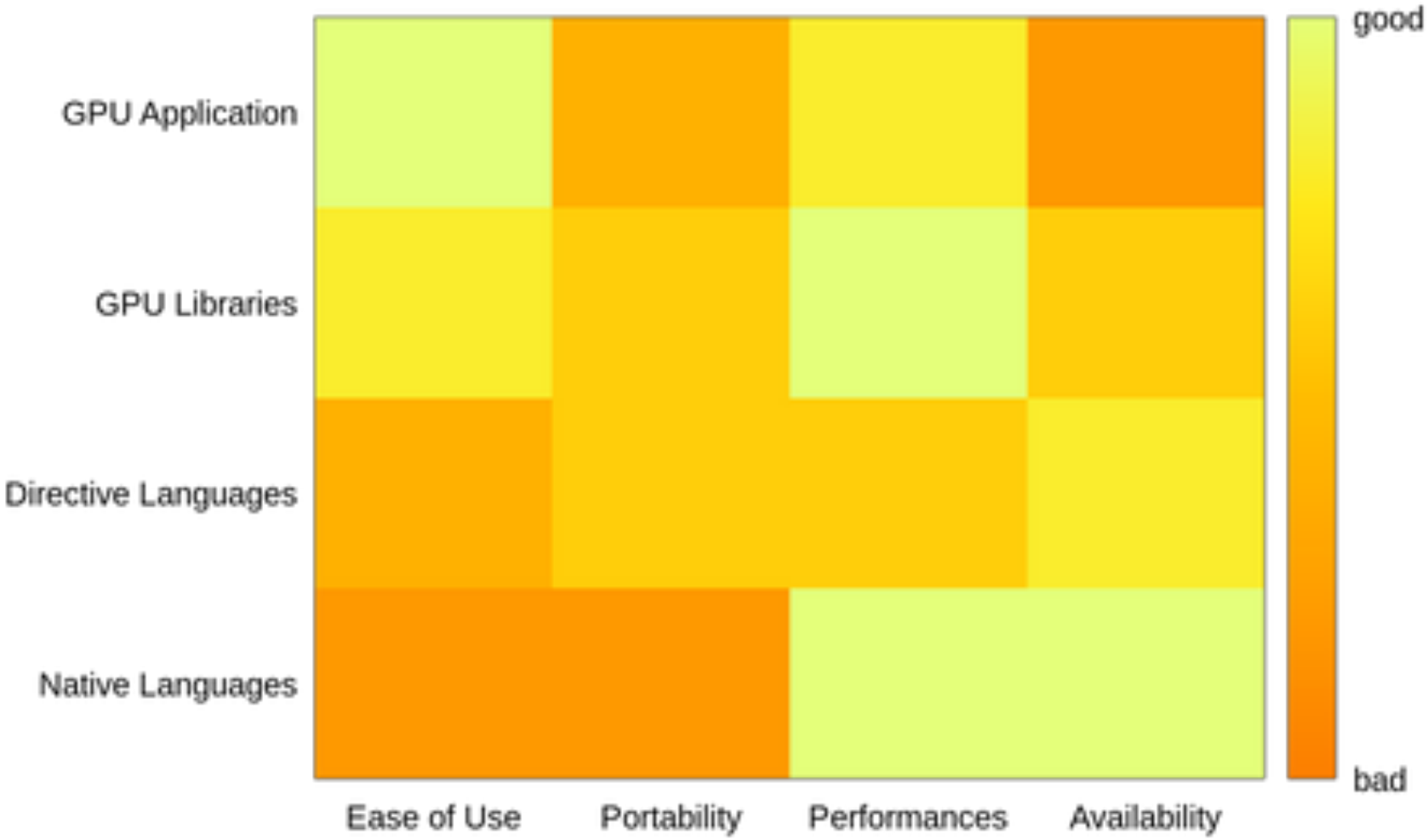
- - Annotating code to pinpoint accelerator-offloadable regions
- - OpenACC
  - created in 2011, latest version is 3.1 (November 2020)
  - mostly Nvidia
- - OpenMP
  - earlier only threading for CPUs
  - initial support for accelerators in 4.0 (2013), significant improvements & extensions in 4.5 (2015), 5.0 (2018), 5.1 (2020) and 5.2 (2021)

- Focus on optimizing productivity
- Reasonable performance with quite limited effort, but not guaranteed

Native GPU code: HIP / CUDA

- CUDA
  - has been the *de facto* standard for native GPU code for years
  - extensive set of optimised libraries available
  - custom syntax (extension of C++) supported only by CUDA compilers
  - support only for NVIDIA devices
- HIP
  - AMD effort to offer a common programming interface that works on both CUDA and ROCm devices
  - standard C++ syntax, uses nvcc/hcc compiler in the background
  - almost a one-on-one clone of CUDA from the user perspective
  - ecosystem is new and developing fast

Using GPUs


GPU Application
GPU Libraries
Directive Languages
Native Languages
Ease of Use
Portability
Performances
Availability
good
bad

Directive languages and performances

- "Write once, run everywhere"
  - It is true that you get portability
  - It is *not* true that you get *performance* portability

- It is possible to optimize code for performance on GPU!
  - It will however be probably slower on the CPU

Summary

- GPUs provide significant speed ups for certain applications
- GPUs are co-processors to CPUs
  - CPU offloads computations and manages memory
- High amount of parallelism required for efficient utilization of GPUs
- Programming GPUs
  - Directive based methods
  - CUDA, HIP