General purpose calculations on

HETEROGENEOUS COMPUTER SYSTEM (GPUs)

N. Shukla

High-Performance Computing Department CINECA

Casalecchio di Reno Bologna, Italy

Email: n.shukla@cineca.it

Acknowledgments

This lectures slides are inspired and adopted from various sources:

- Luca Ferraro and Sergio Orlandini (Rome, **CINECA**)
- Jeff Larkin (NVIDIA)
- Michael Klemm (AMD)
- Tom Deakin (University of Bristol, UK)
- Swaroop Pophale (ORNL, US)
- OpenMP GPU Offload Basics (Intel)
- and many others
- OpenMP 5.0.1 specification and examples https://www.openmp.org/resources/

Schedules of the day

Introduction to GPUs

OpenACC programming model

Nvidia Profiling

Offloadings with OpenMP

Objectives of this session

Motivation

What compel to use a Graphics Processing Units (GPUs)?

What are the key differences between GPUs and CPUS

Fundamentals of GPUS

Modern GPGPU architecture scheme

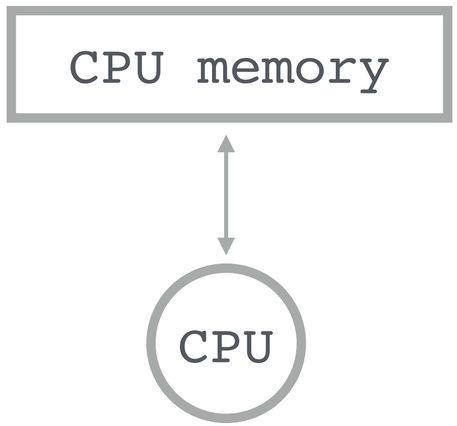
Programming models

OpenACC / OpenMP offloading

Closing thoughts

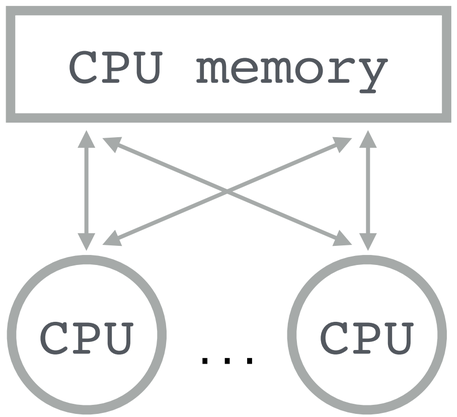
HPC System evolution

Changes in the architecture level also requires changes in the programming models



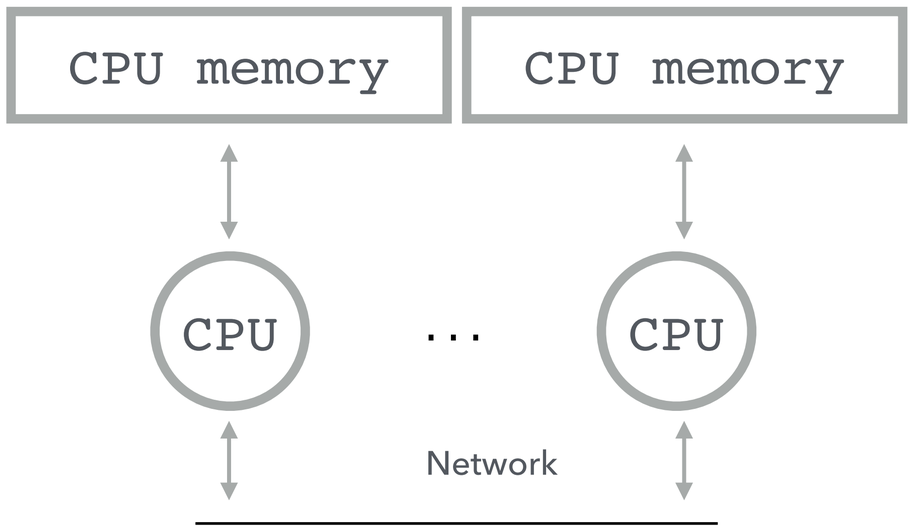
Single CPU

- Sequential computing
- Performance in Mflop/sec



Multiple CPU

- Shared memory
- Requires change in hardware and software architecture
- New standard: pThread or OpenMP
- Performance in Gflop/sec



Multi-node system

- Connected with high-speed and low latency network (200 GB/s via Infi band network)
- Requires change in hardware and software architecture
- New standard: MPI+OpenMP
- Performance in Tflop/sec

Golden era: 1977-2017

40 years of stunning progress in microprocessor design

- 1.4x annual performance improvement for 40+ years ~ 106x faster (throughput)

Three architectural innovations

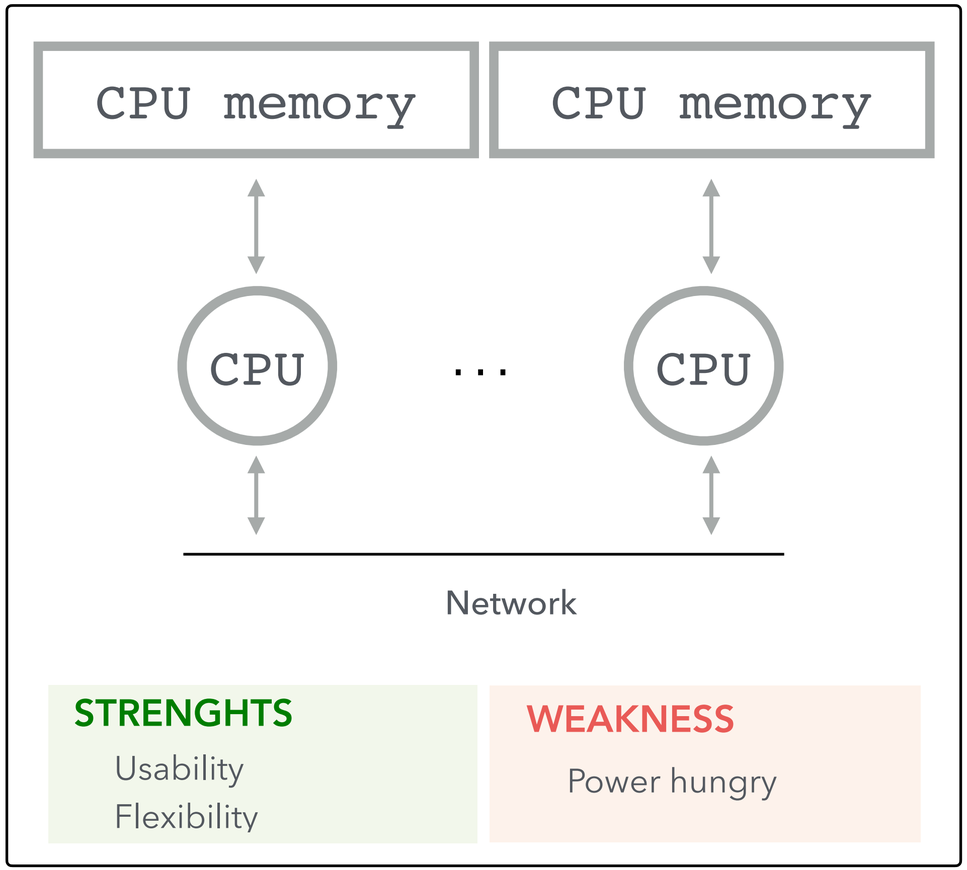
- Width 8 > 16 > 64 bit (~ 4x)
- Instruction level parallelisms: 4-10 cycles per instructions to 4+instructions per cycle (~10-20x)
- Multicore: one processor to 32 processor (~32x)

Clock rate: 3MHz to 4GHz (through technology & architecture)

Made possible by IC technology

- Moore's law: growth in transistor count
- Dennard scaling: power/transistor shrinks at same rate as transistors are added (constant per mm2 of silicon), held until 1997 and then began fade away
- 2007-2017: 45 to 16nm : 3.0x increase in energy/chip

Homogeneous HPC systems: from Giga to Exa and beyond …



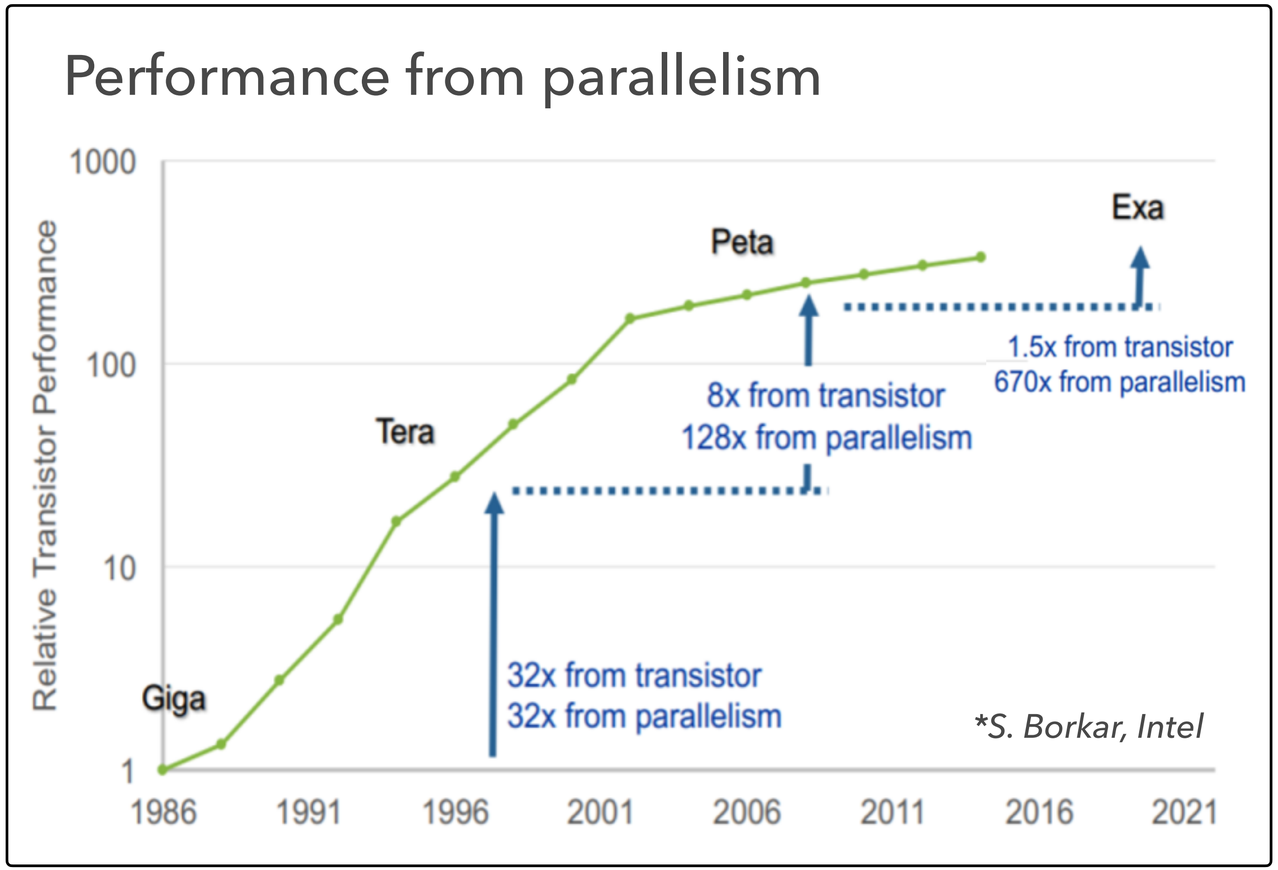
STRENGHTS

Usability

Flexibility

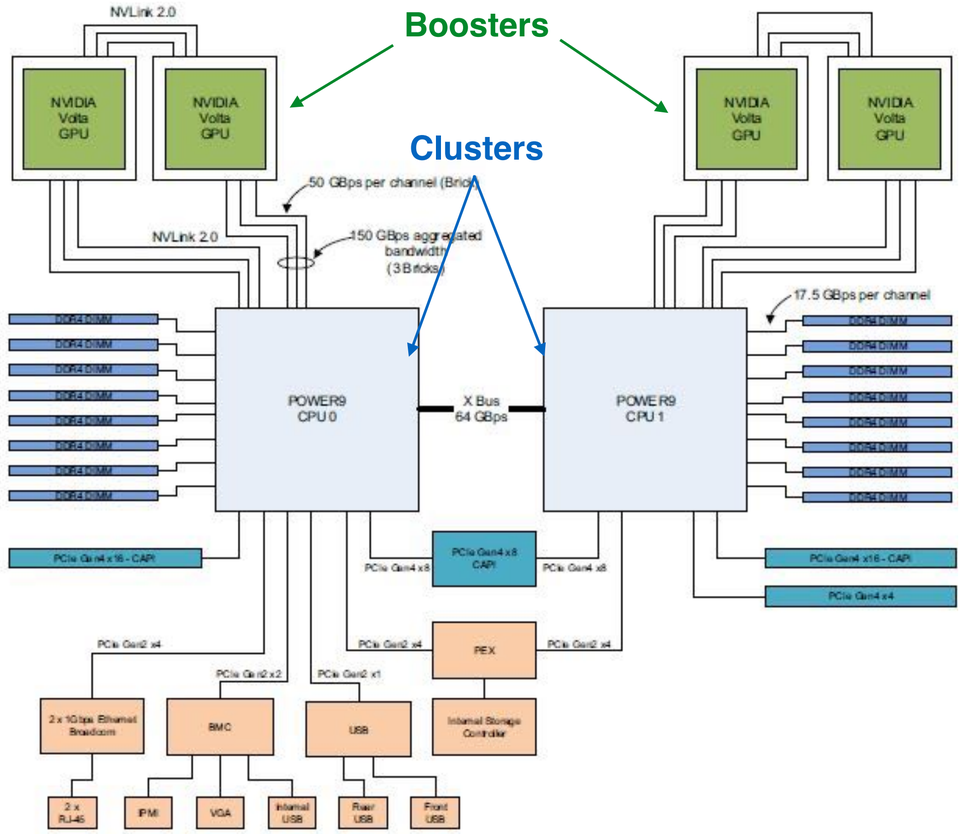
WEAKNESS

Power hungry



Do we still need to use CPUs then ?

Heterogenous HPC systems: abstract diagram of M100



Node Performance		
Theoretical Peak Performance	CPU (nominal/peak freq.)	691/791 GFlops
	GPU	31.2 TFlops
	Total	32 TFlops
Memory Bandwidth (nominal/peak freq.)		220/300 GB/s

STRENGHTS

- Energy efficient
- Scalability
- High Flexibility

WEAKNESSES

- Complexity

What is GPGPU?

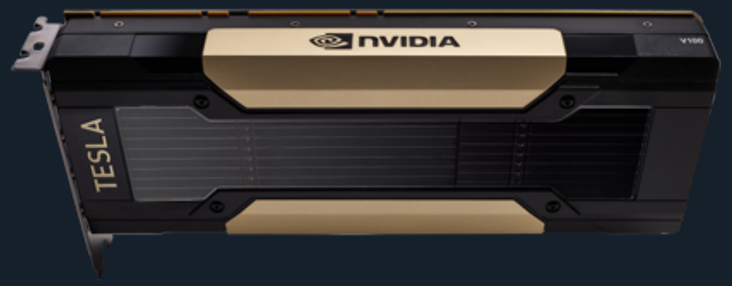
GPGPU stands for **G**eneral **P**urpose **G**raphics **P**rocessing **U**nits

Born in 90's used for video games

- rendered at 60 frame per second
- requires a set of transformation based on linear algebra and filters
- same set of operations are applied to each data point of the scene
- each operation is independent with respect to data
- executes in parallel using a huge number of threads independently

GPUs became programmable for HPC

- **NVIDIA** introduced a GeForce 3 GPU with hundreds of cores and thousands of threads in 2001
- first efforts in the directions to support general-purpose computing in addition to graphics
- Implementation of linear algebra using these early GPUs by mapping matrix data into into textures and applying shaders



CPUs and GPUs were designed with different goals in mind

CPUs are designed to minimize latency

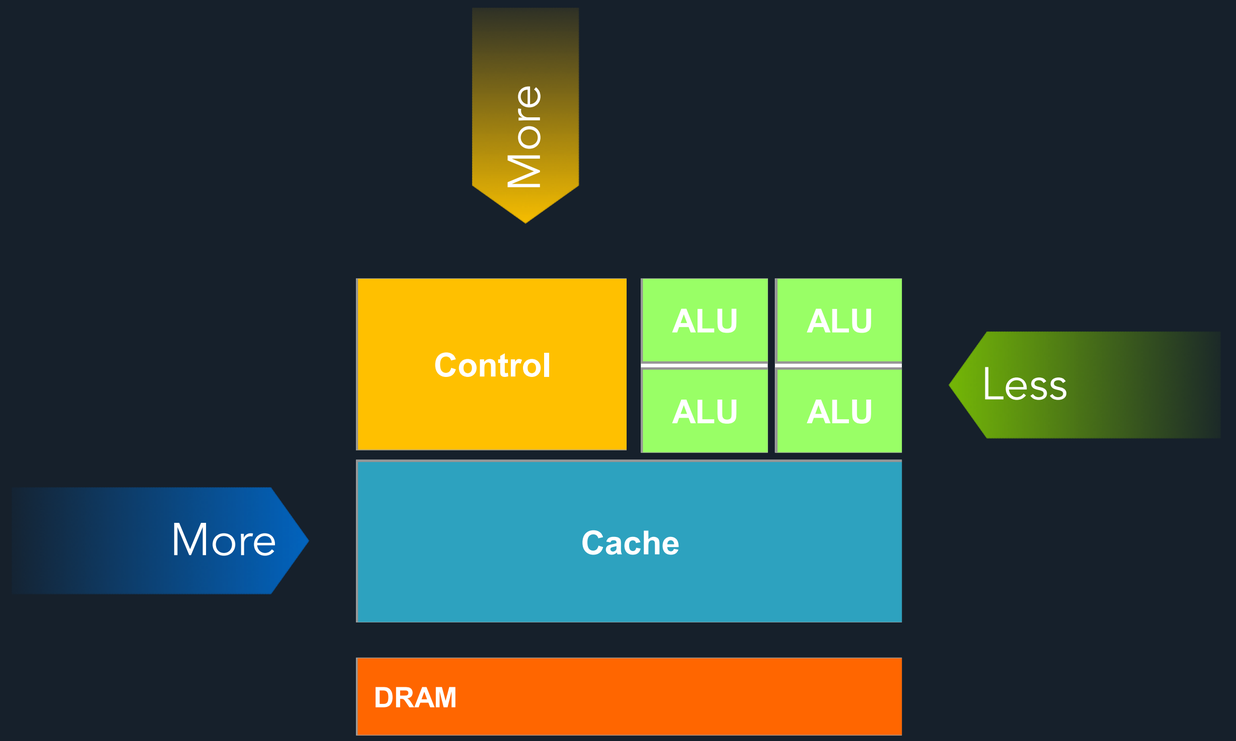
Majority of silicon is dedicated to

- Several Several Arithmetic Logic Units (ALU)
- Large cache

Memory hierarchy

- Very large Dynamic Random Memory Access (DRAM)
- Fast cache memory (Registers, L1-L3 cache)

Latency optimised via large caches



CPUs and GPUs were designed with different goals in mind

GPUs are designed to maximise throughput

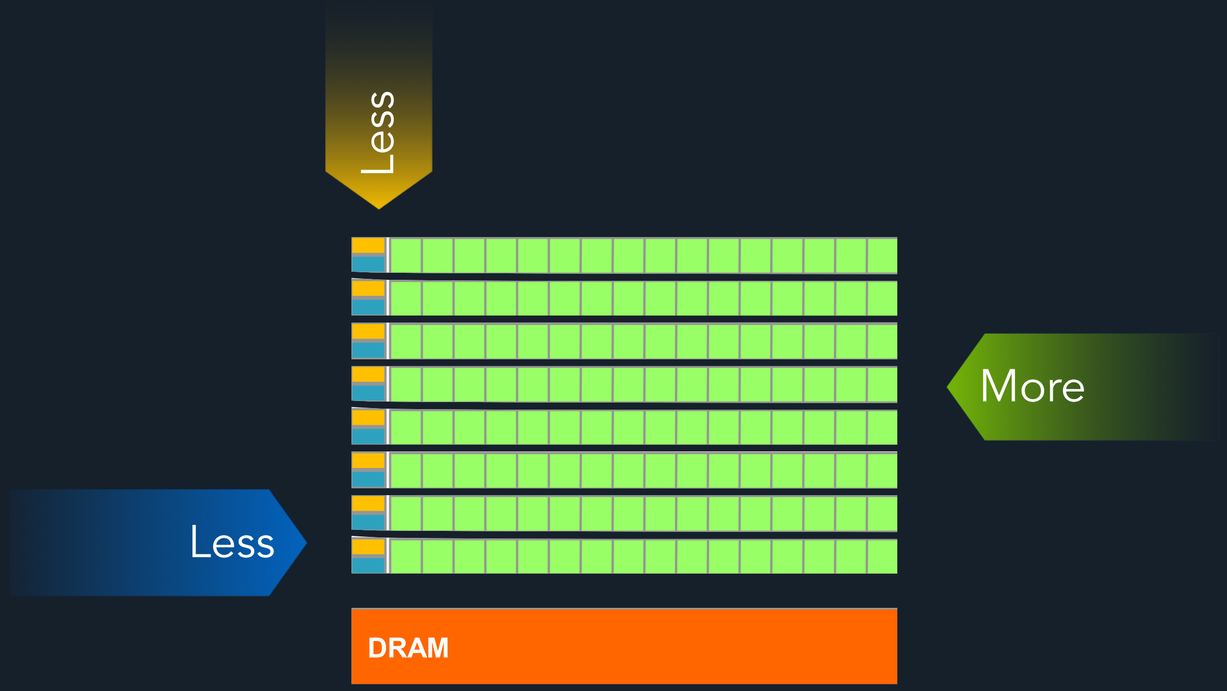
Majority of silicon is dedicated to

- Thousands of ALUs
- Each has its own control units and registers

Memory hierarchy

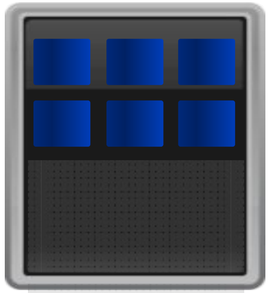
- High bandwidth DRAM
- Several cache memories

Memories and instruction latencies talent with computing operation



Summary: Low Latency or High Throughput?

CPU



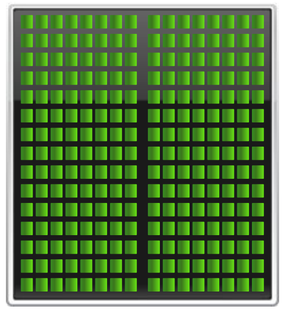
STRENGHTS

- Large main memory ~ 1TB+
- Faster clock speed ~ 4 GHz
- Latency optimised via large caches
- Small number of threads can run very quickly ~ Equal to number of cores

WEAKNESSES

- Low memory bandwidth ~200 GB/s
- Cache misses very costly
- Low performance/watt

GPU



STRENGHTS

- High bandwidth main memory ~ 1TB/s+
- Significant more compute resources ~ Few thousands cores
- High throughput
- High performance/watt

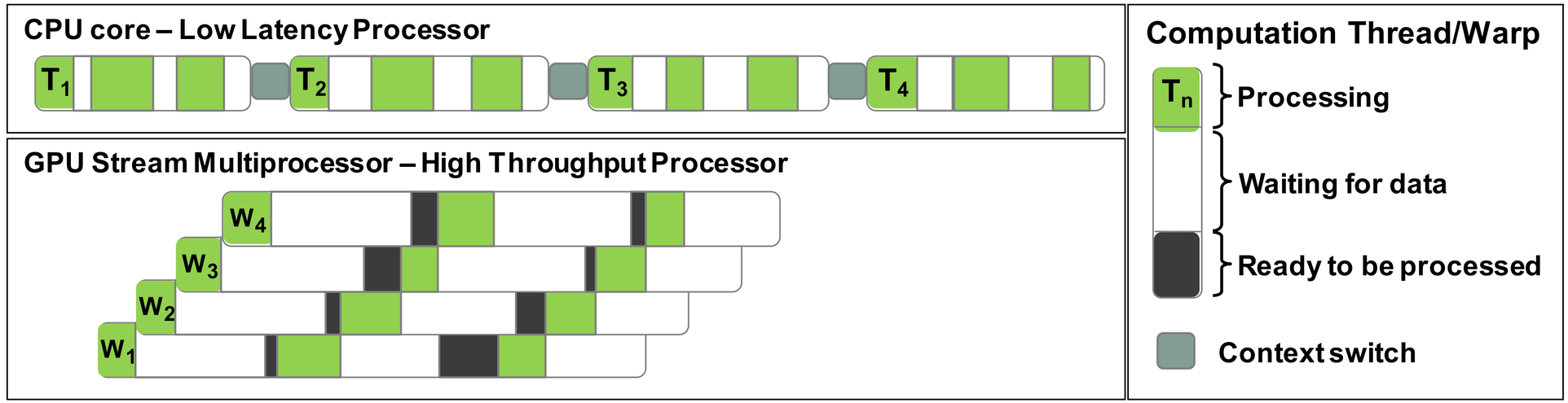
WEAKNESSES

- Relatively low memory capacity ~80 GB per GPU (Total 640 GB unified in 8 GPU system)
- Low per-thread performance: 4 time slower clock speed than CPU core

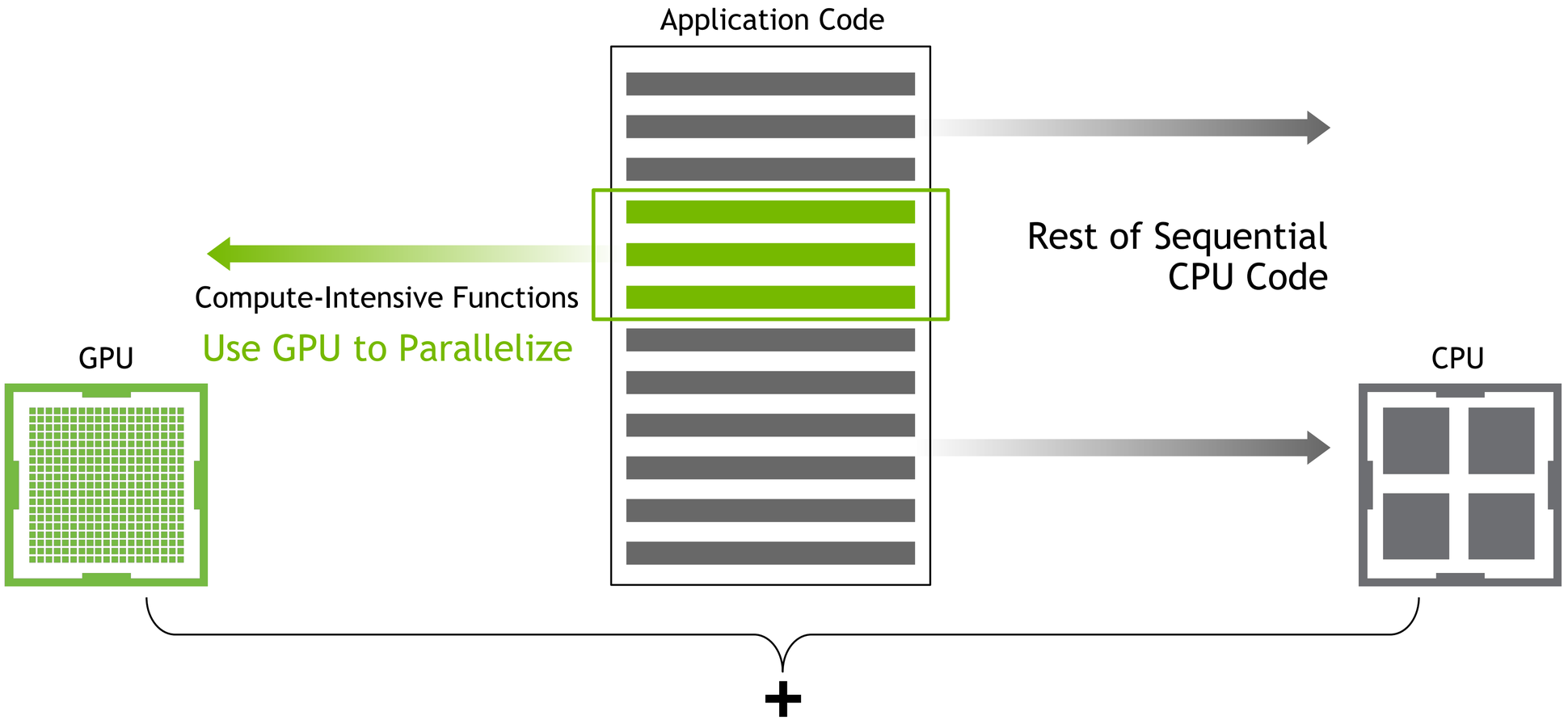
Low Latency or High Throughput?

GPU hides memory latency

- **CPU** architecture must **minimize latency** within each thread
- **GPU** architecture **hides latency** with computation from other thread warps



SMALL CHANGES, BIG SPEED-UP



GPU architecture scheme

Main Global memory

- Medium size (14-40 GB)
- Very high bandwidth (800-1200 GB/s)

Streaming Multiprocessors (SM)

- Group of independent cores and control units
- Interactions scheduler dispatchers
- Perform the actual computation
- Each SM has its own: Control units, registers, execution pipelines, caches

Speed v . Throughput

Speed

Throughput

Which is better depends on your needs…

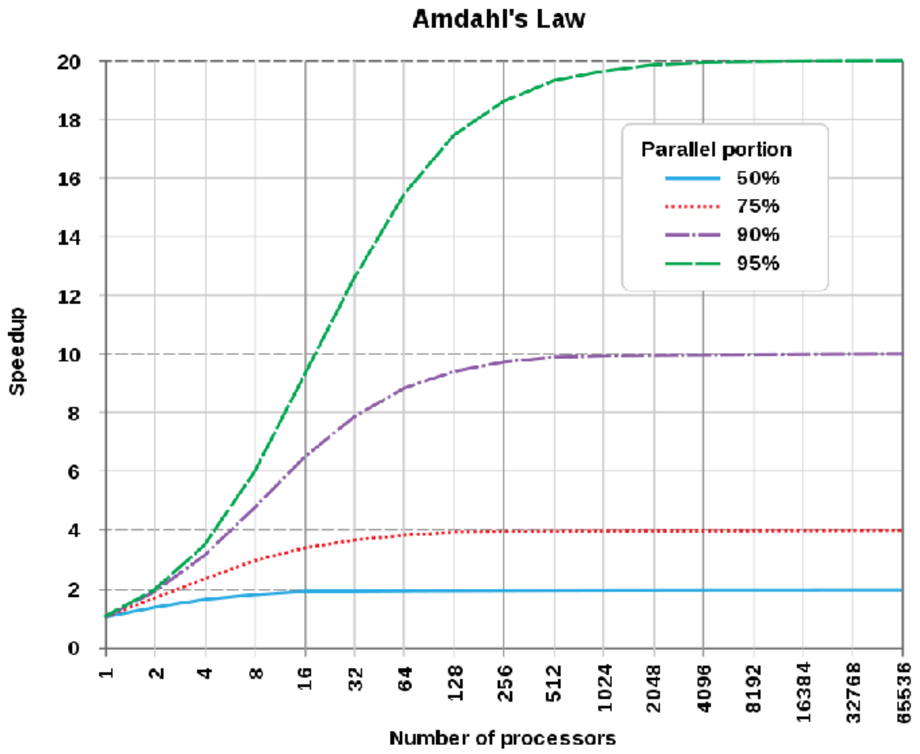
*Images from Wikimedia Commons via Creative Commons

Amdahl's law: serialisation limits performance

Amdahl's law states overall speedup s given the parallel fraction p of code and number of processes N

$$s = \frac{1}{1 - p + \frac{p}{N}} < \frac{1}{1 - p}$$

Limited by serial fraction, even for $N \to \infty$



Heterogeneous CPU–GPU System Architecture

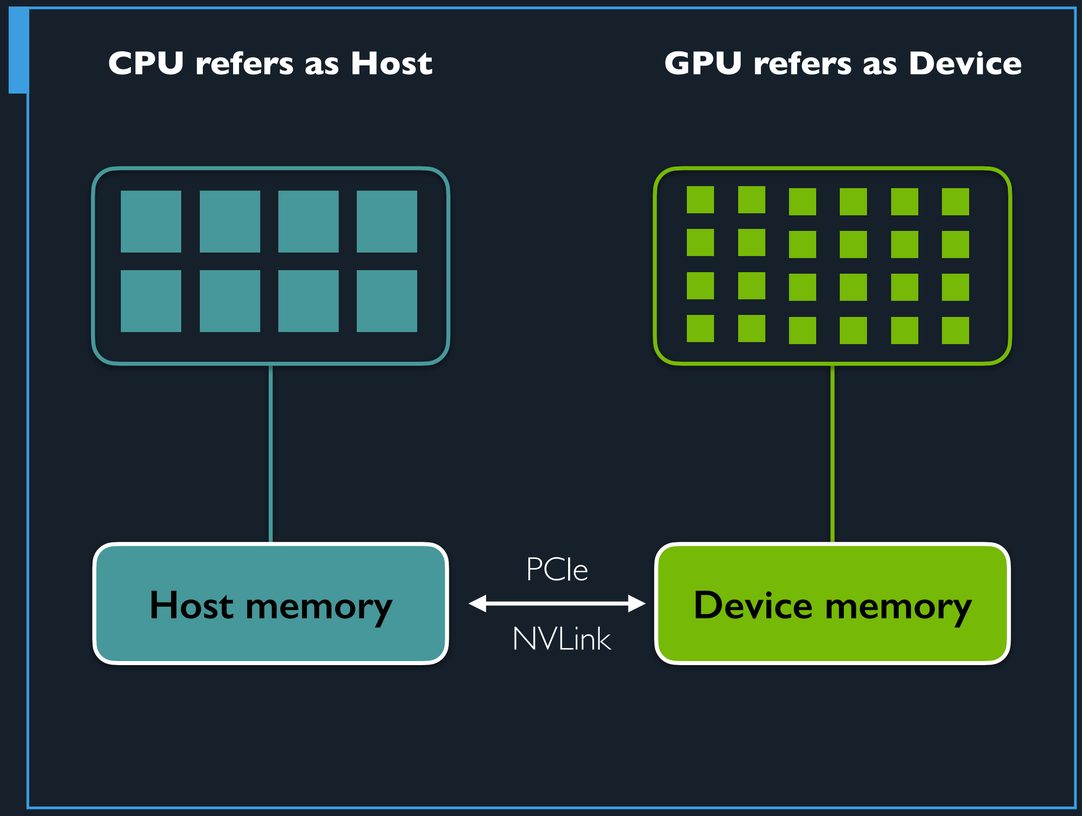
CPU and GPU work together for best benefit and performances

Device model

- Host-centric model with one host and multiple devices of the same type
- Devices are connected to host CPU via interconnect, such as PCIe or NVLink
- Host and device have separate memory spaces

GPGPU execution model

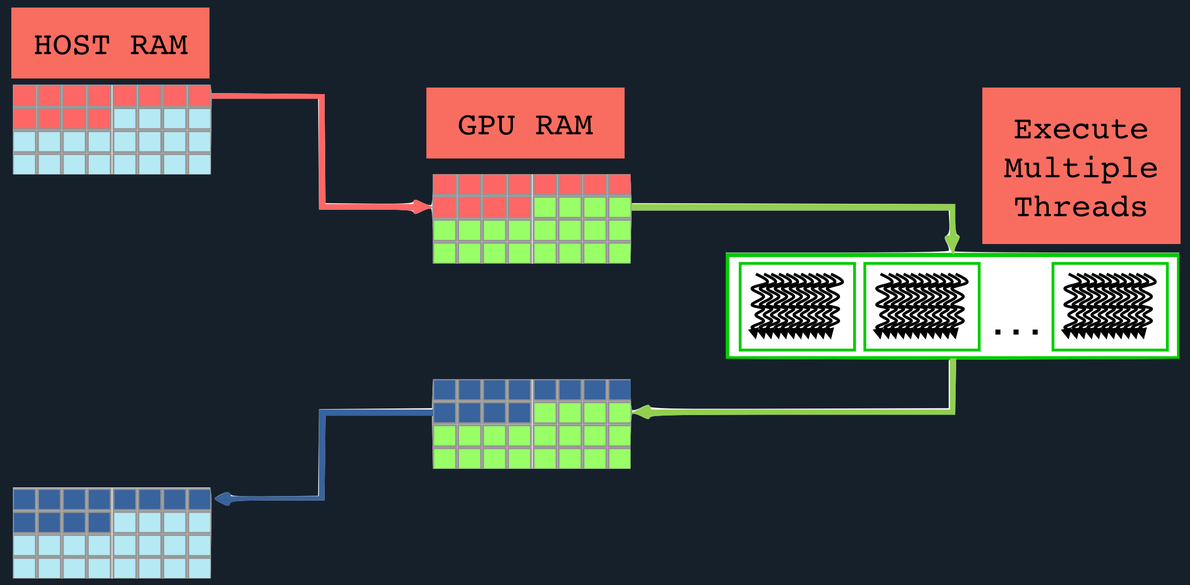
- A function which runs on a GPU is called a **Kernels**
- Kernels are executed as a set of threads that can run concurrently
- Each thread is mapped to a single CUDA core on the GPU
- CUDA threads executes in a Single Program Multiple Data (SPMD)



The data movement is a bottleneck

GPGPU programming model

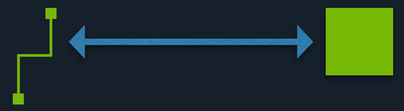
- Data must be moved from HOST to DEVICE memory in order to be processed by a CUDA kernel
- When data is processed, and no more needed on the GPU, it is transferred back to HOST
- Programmers choose the number of threads to run
- Each thread acts on a different data element independently
- The GPU parallelism is similar to the SPMD paradigm
- Threads belonging to the same block or team can cooperate together exchanging data through a shared memory cache area



Organisation of CUDA threads

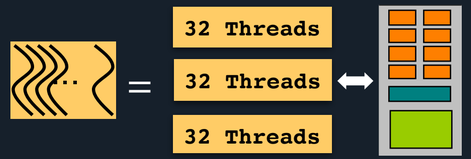
Thread

Kernels execute as a set of threads



Wrap

Collections of 32 threads
Smallest schedule

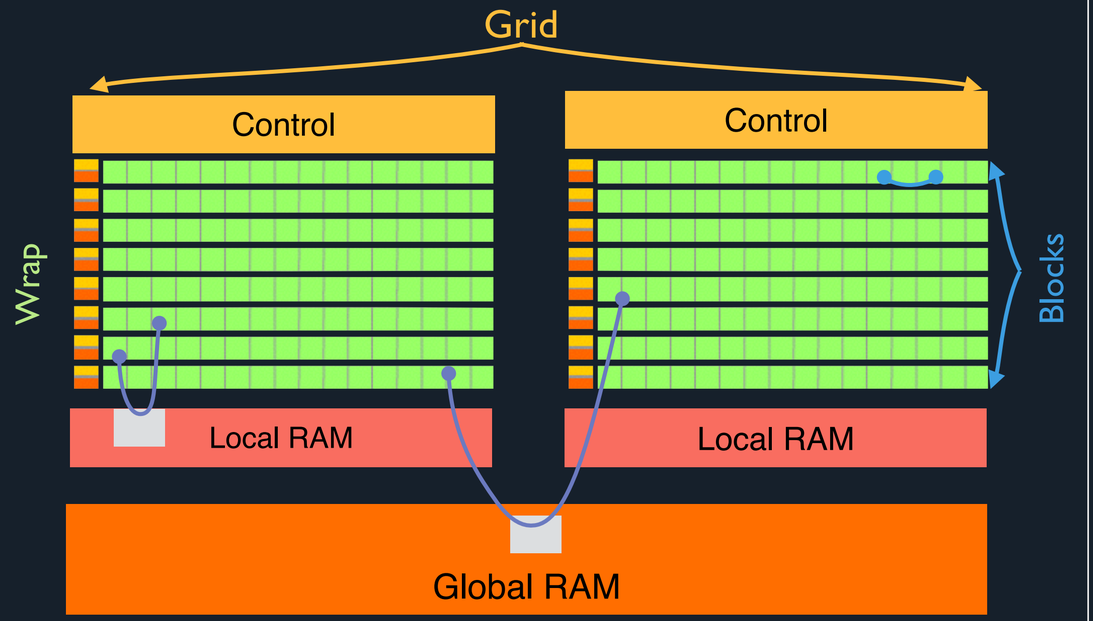


Thread blocks

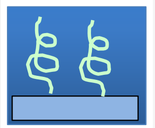
Collections of 32 wraps

Grid size

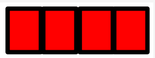
Collections of thread blocks



Hardware Diversity: Basic Building Blocks

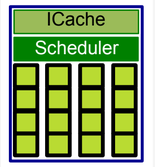


CPU Core: one or more hardware threads sharing an address space



SIMD: Single Instruction Multiple Data

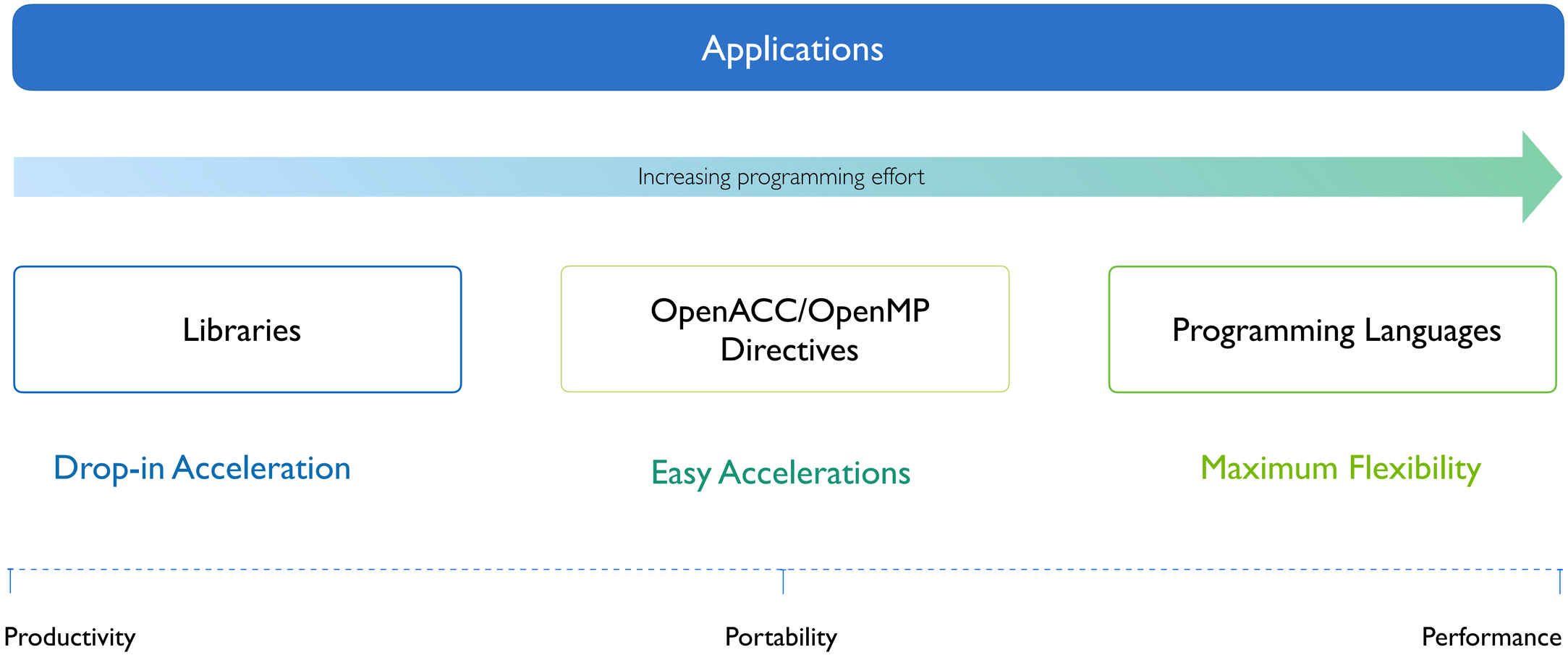
Vector registers/instructions with 128 to 512 bits so a single stream of instructions drives multiple data elements.



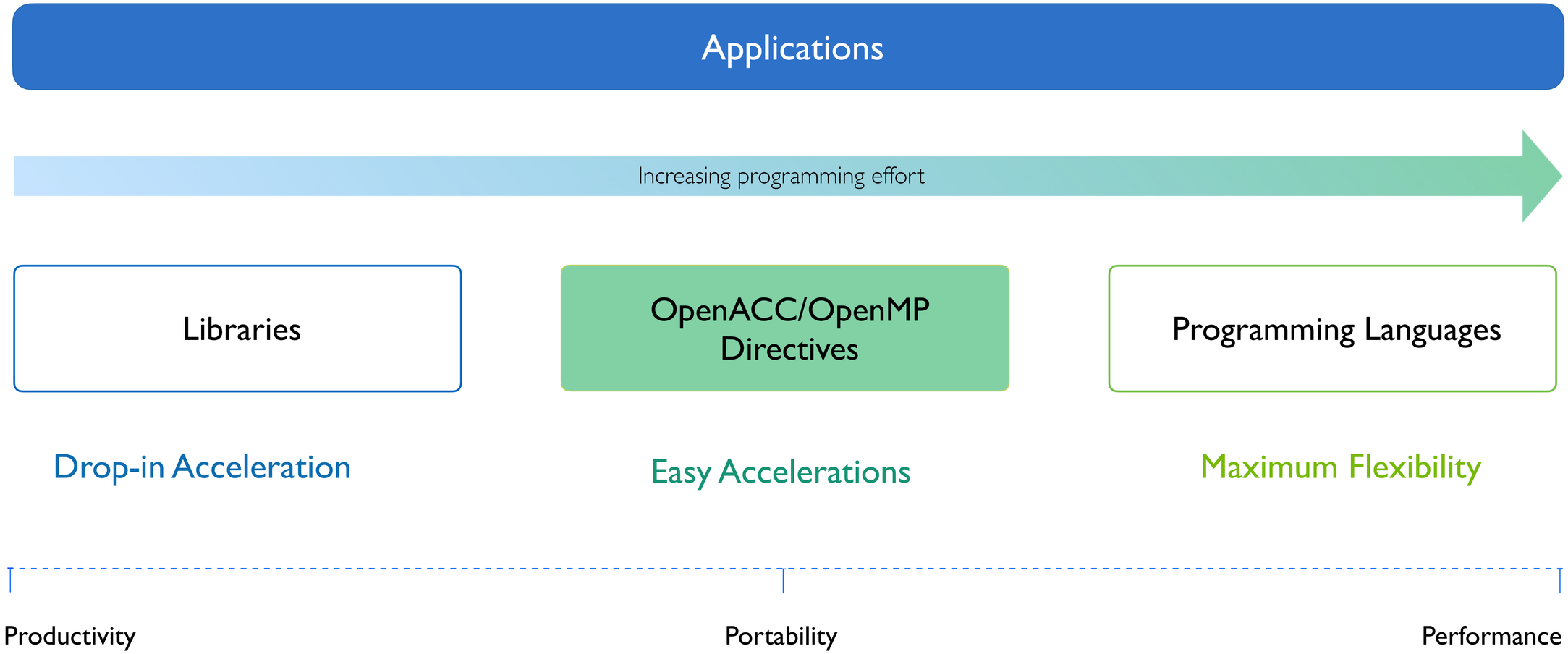
SIMT: Single Instruction Multiple Threads

A single stream of instructions drives many threads. More threads than functional units. Over subscription to hide latencies. Optimized for throughput.

3 Ways to Accelerate Applications



3 Ways to Accelerate Applications



What are OpenX (X = MP, ACC)?

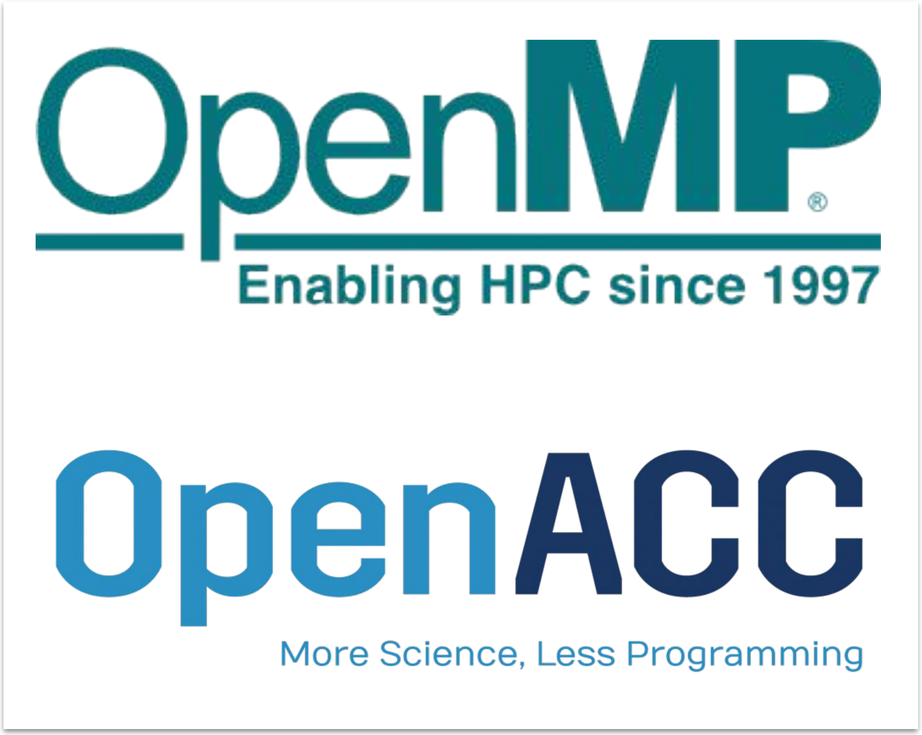
Directive-based programming model for parallel computing

Compiler directive approach

- NOT a programming language
- Extension for C/C++ and Fortan

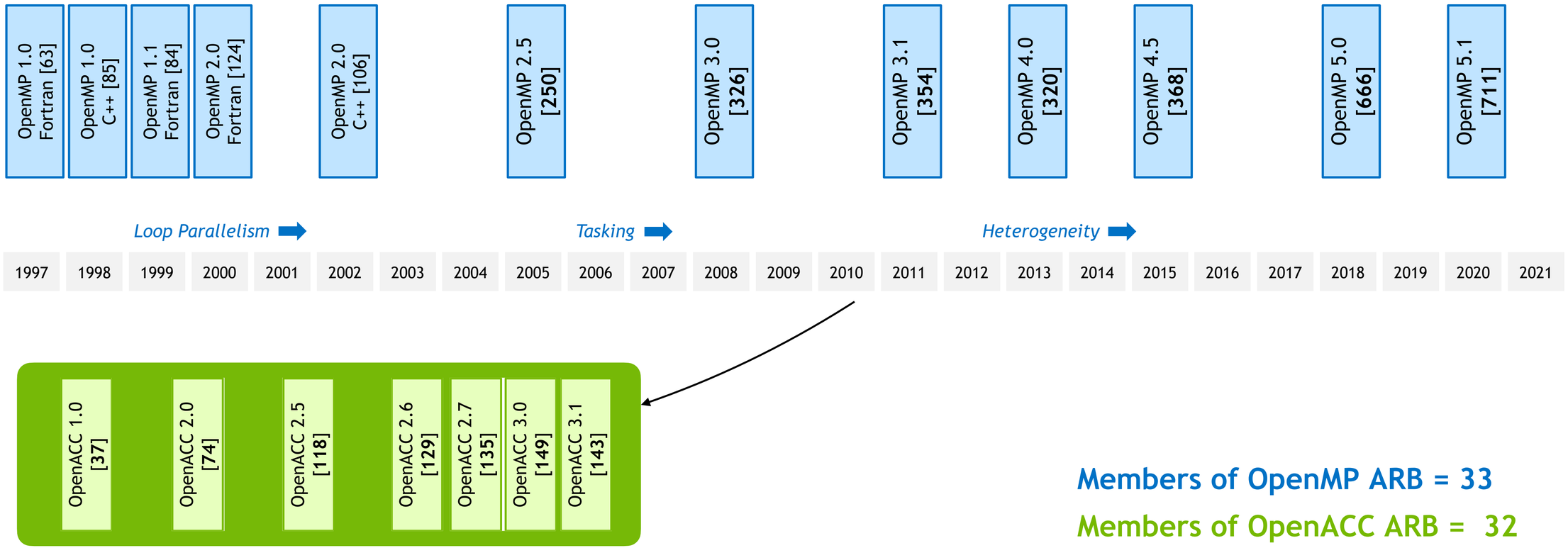
Host centric programming model

- CPU controls the accelerator
- Intensive computation is performed on GPU
- Easy portability



OpenX (X = MP, ACC)

Looking at TIME (pace of innovation) and SPACE (specification length)



Members of OpenMP ARB = 33

Members of OpenACC ARB = 32

OpenX (X = MP, ACC)

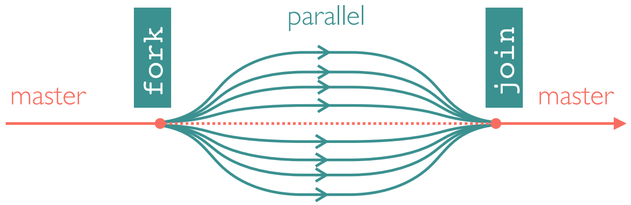
Much a like, directives approach to accelerator Fortran and C/C++ codes

OpenMP

Designed to replace low-level and multi-threaded programming solutions like POSIX, threads or Pthreads

Intend to target independent processor (shared memory)

Basic principle: fork-join model



OpenMP 4./4.5 onwards; offloading capabilities

OpenMP is more prescriptive

Compiler support

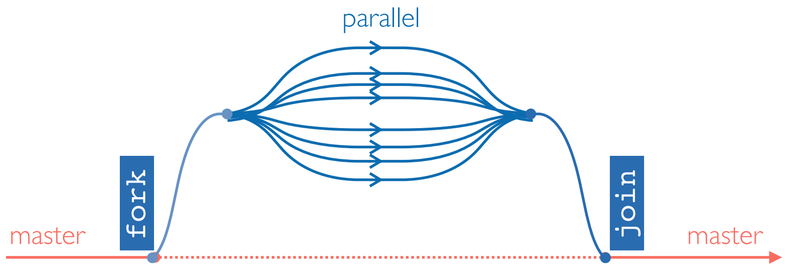
GCC, Intel, IBM XL, LLVM/Clang etc

OpenACC

Specifically targets GPU accelerators

It started after OpenMP

Basic principle: fork-join model



OpenMP is more descriptive

Compiler support

PGI, Cray, NVIDIA

OpenX (X = MP, ACC)

Same basic principle: Fork/join model

OpenMP: prescriptive nature

Programmer explicitly parallelizes the code

- Requires to perform requested parallelization and little/no analysis by the compiler

Reproducibility

- Parallelization will be performed the same way whatever the hardware the code runs on

Substantially different architectures require different directives

- To perform optimally on multiple different architecture, one has to write different sets of directives for different architectures
- Sometimes, it requires change in the code, for example it may be beneficial to switch the order of loops
- Fairly consistent behaviour between implementations

OpenACC: descriptive nature

Rely on compiler

- Compiler parallelises the code with guidance from the programmer

Compiler takes decision

- the code may parallelize or even may not be parallelized

Compiler executes information from the programmer and heuristics about the architecture to make decision

- The same code, compiled to run on GPU, or on Xeon Phi, or on CPU, may therefore yield different binary code
- Different compilers may yield difference performance
- Quality of implementation greatly affects the results

Why OpenMP/OpenACC for GPU programming

- **GPUs have a reputation for being difficult to use because of programming**
  - OpenMP/OpenACC aim to change that
- **Non-invasive**
  - Does not requires strong modifications the code
- **Portability**
  - Works on many-core GPUs and multi-core CPUs
  - Improve portability and readability of the code compare to other methods of using accelerators
- **Powerful**
  - GPU directives allow complete access to the massive parallel power of a GPU
  - Through some refactoring may help with performance
  - Support for C/C++ and Fortran

Objectives: Enable to accelerate your application with the GPU