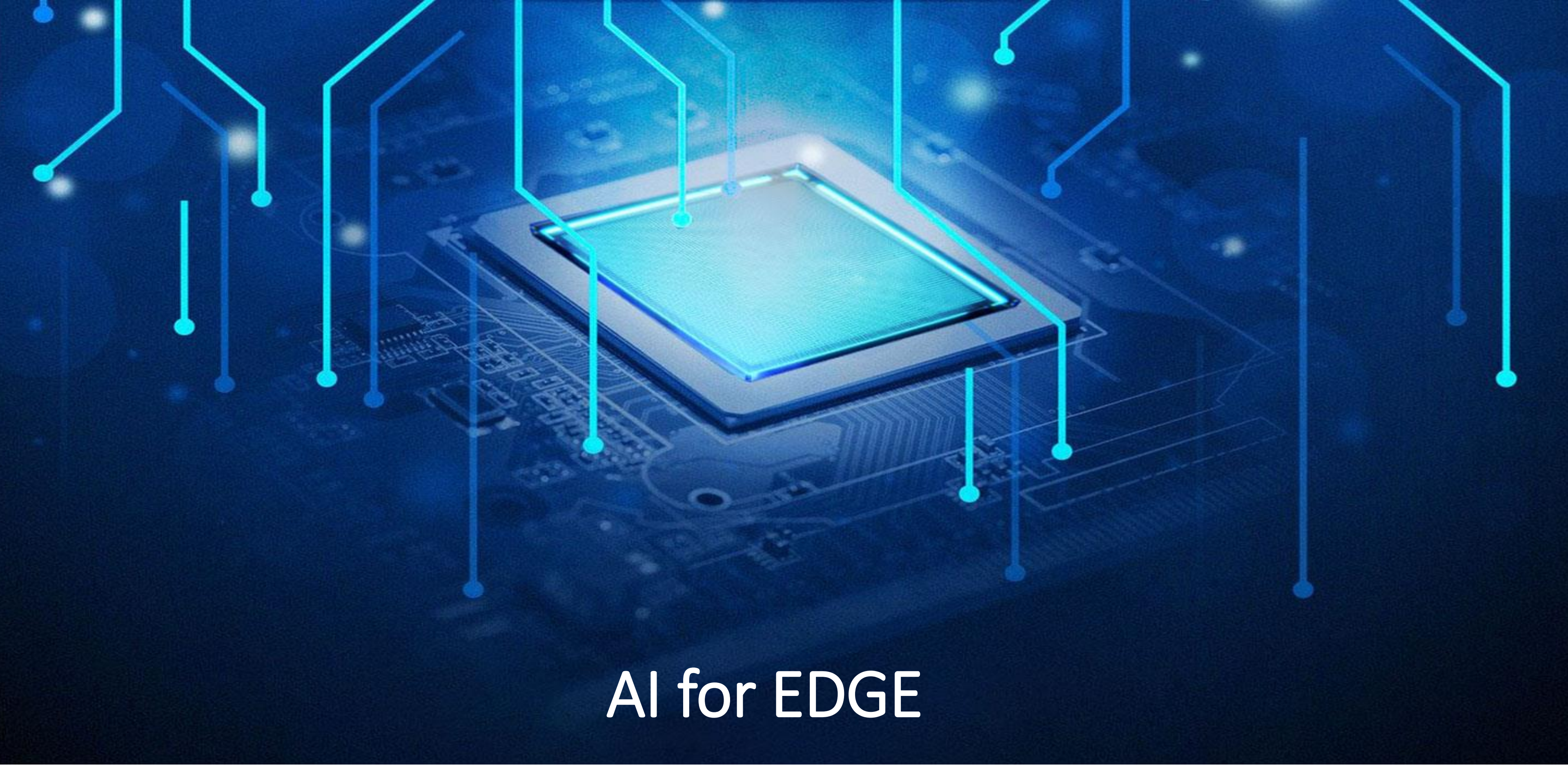


AI for EDGE

Compute architectures

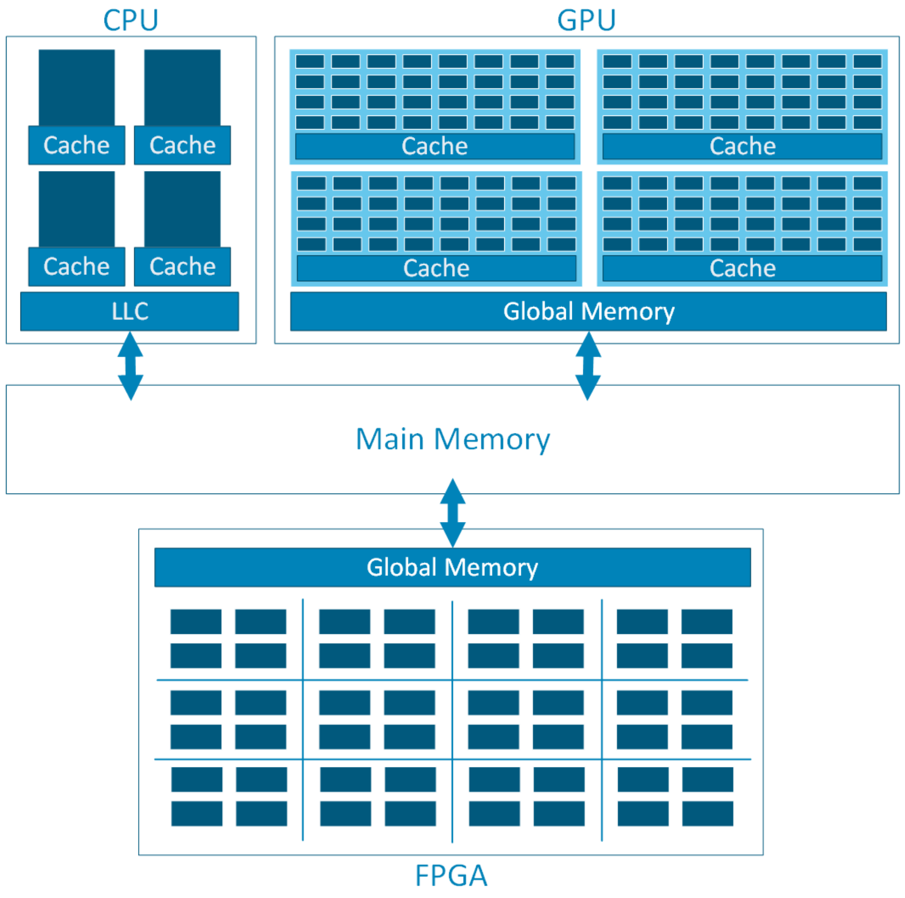
Classroom materials

Compute architectures - outline

- Heterogeneous compute environment
- Architecture comparison
  - CPU
  - GPU
  - FPGA
- How to match the workload to compute device
- Alternative for edge computing

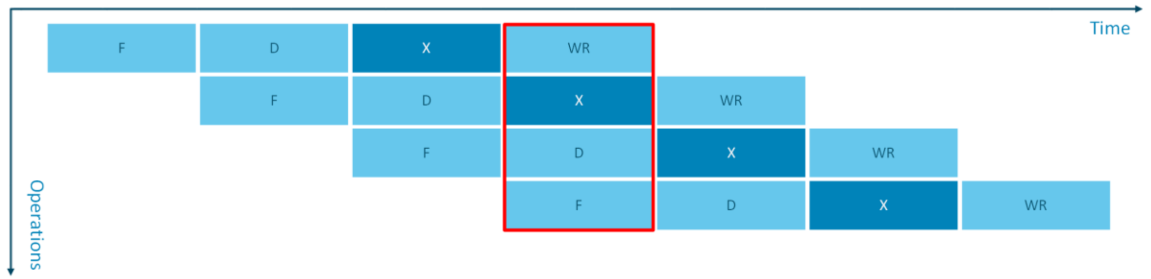
Shared-memory computing system

- One or more CPU (each core with own local cache)
- Shared LLC (last level cache)
- Set of accelerators:
  - GPUs
  - FPGAs
  - other specialized hardware



CPU architecture

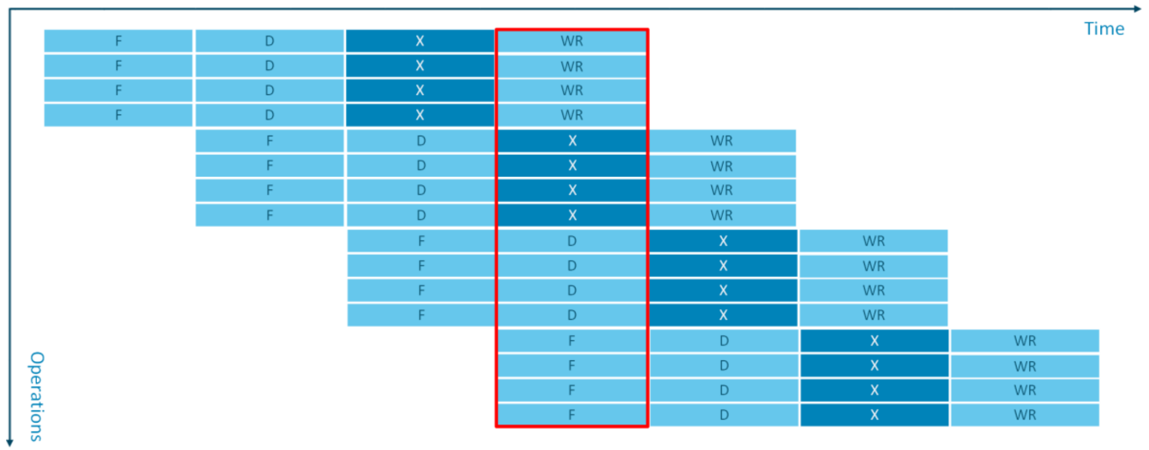
Scalar Pipelined Execution



F – fetch
D – decode
X – execute
WR – write to mem/reg

🟥 Operation-Level Parallelism

Superscalar Execution



F – fetch
D – decode
X – execute
WR – write to mem/reg

Classroom materials

CPU architecture

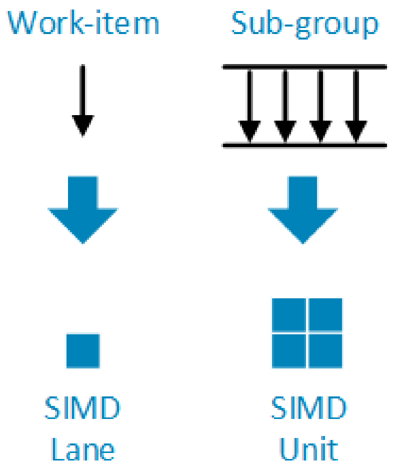
- Advantages:
    - automatic parallelism on sequential code
    - lower latency when compared to offload acceleration
    - accurate branch prediction
    - out-of-order superscalar execution
    - better energy efficiency than GPU configuration

CPU architecture

- Several types of parallelism to achieve performance:
  - SIMD (single instruction, multiple data) data parallelism
  - Thread-level parallelism (multiple threads, different logical cores)
  - Instruction-level parallelism

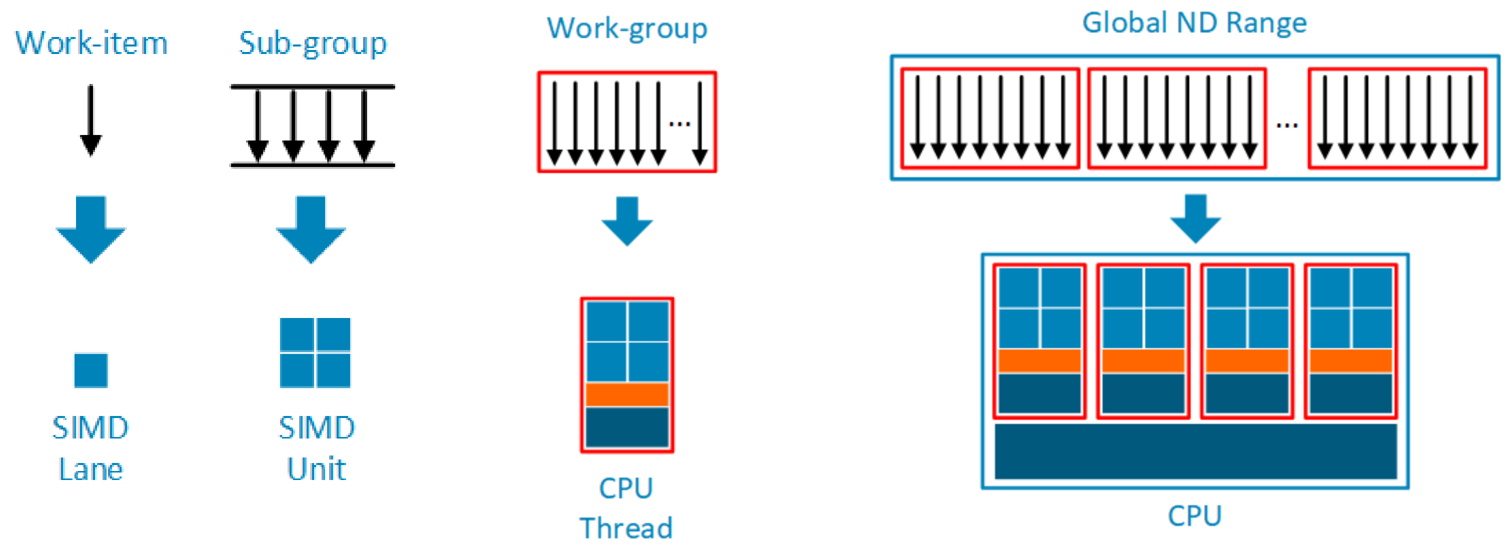
CPU architecture

- SIMD data parallelism
  - each work-item can map to CPU SIMD lane
  - vector data tapes are used to explicitly specify SIMD operations
  - compiler can perform loop vectorization to generate SIMD code
    - one loop iteration maps to a CPU SIMD lane
    - multiple loop iterations execute together in SIMD fashion



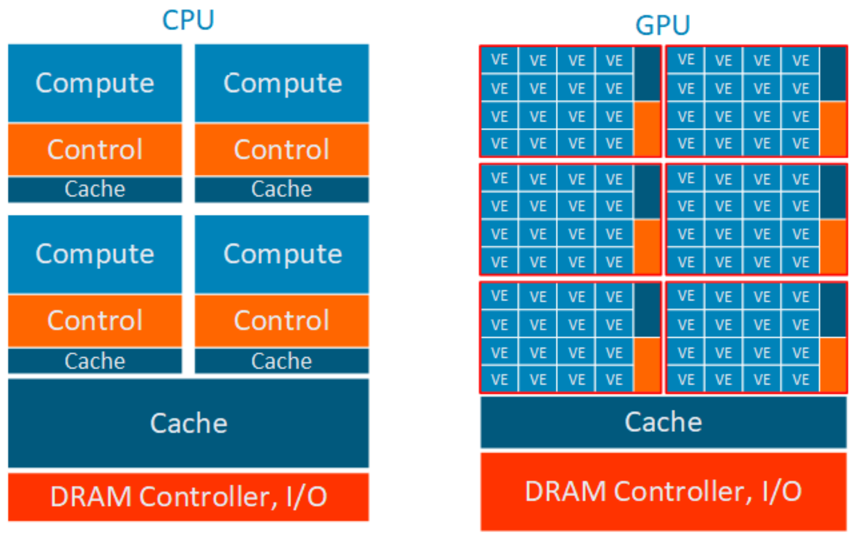
CPU architecture

- Thread-level parallelism
  - CPU core and hyper-thread parallelism (a machine with 2 cores and 4 hyper-threads can execute 4 work-groups in parallel)
  - different work-groups can execute on different logical cores in parallel



GPU architecture

- Massively-parallel, more specialized cores than CPUs
- Optimized for aggregate throughput across all cores
- Vector architecture (efficiently processes vector data)
- More silicon space to compute, less to cache and control



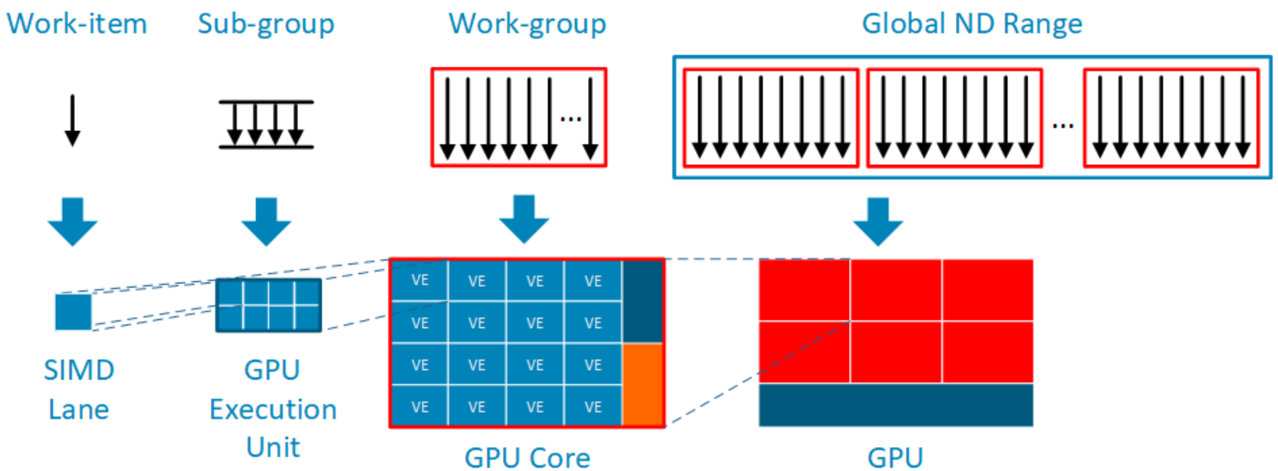
VE – vector engine (each VE can process multiple SIMD instruction streams)

SM – streaming multiprocessor (multiple VEs combine to form a compute unit with shared local memory and synchronization mechanisms)

GPU architecture

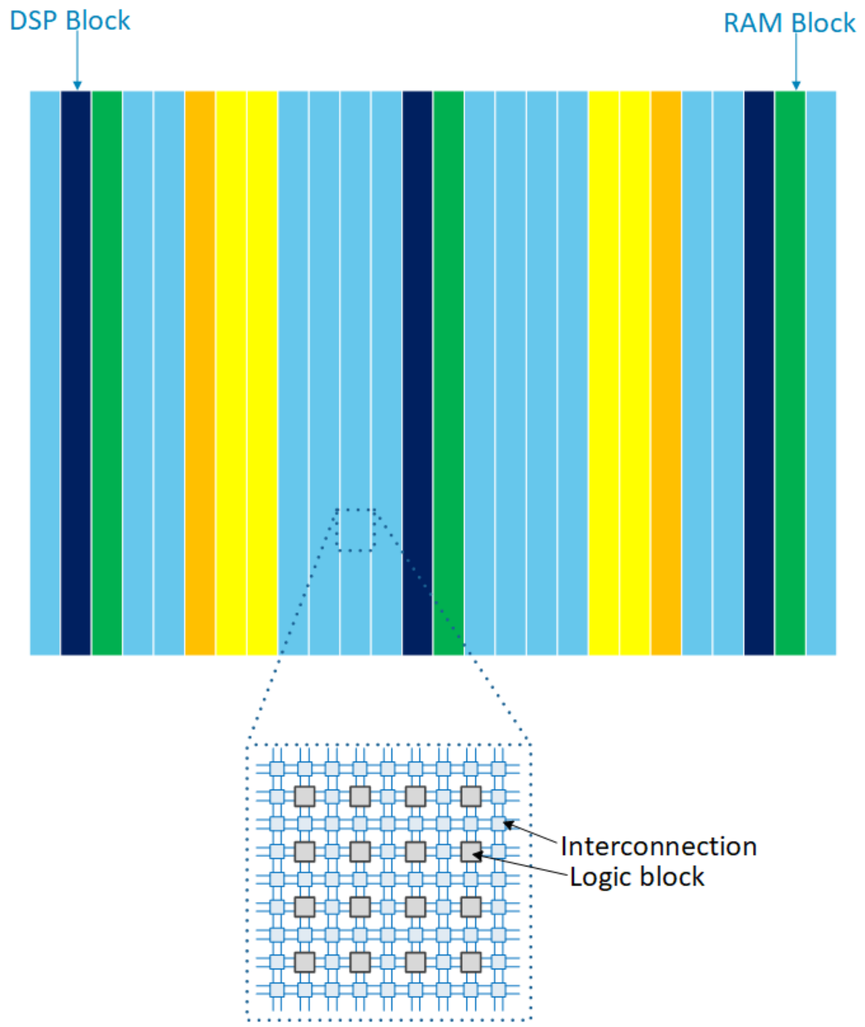
GPUs rely on large data-parallel workloads to achieve performance

- every work-item is mapped to a SIMD lane
- sub-groups are mapped to the vector engine
- work-groups, which include work-items that can synchronize and share local data, are assigned for execution on streaming multiprocessors
- single-task kernels are rarely utilized (NDRange kernels are needed to fully populate deep execution pipeline)



FPGA architecture

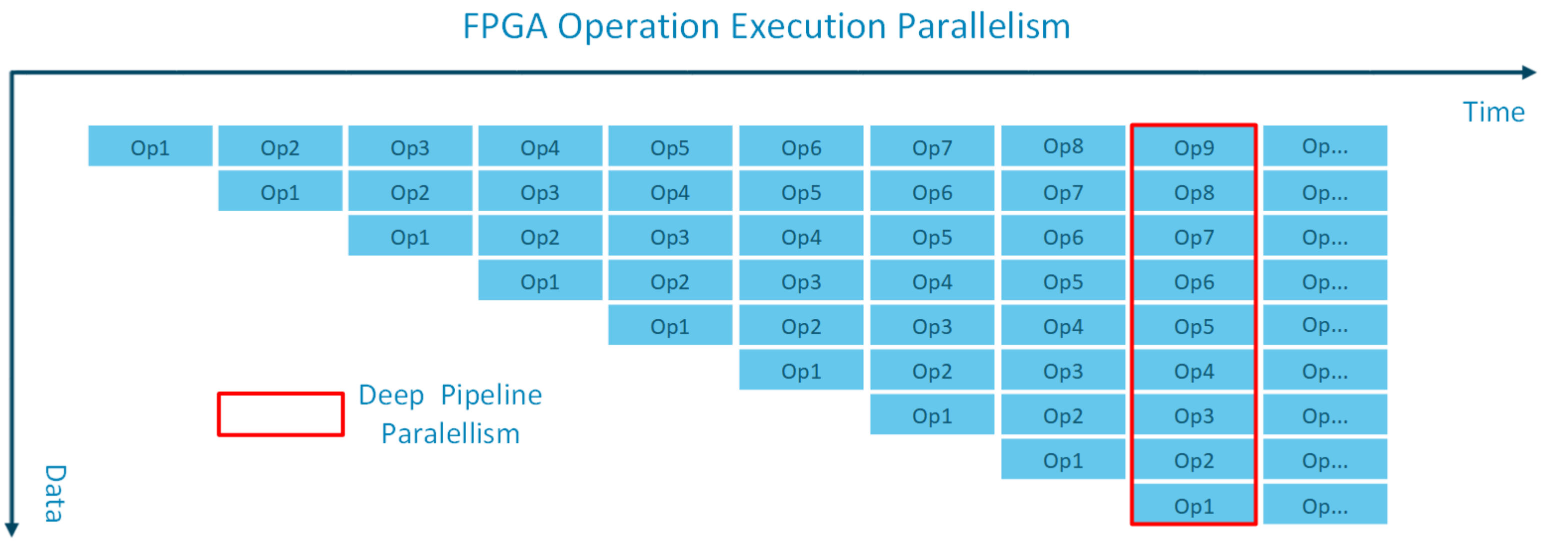
- Massive array of small processing units
- Resources are connected by a mesh of programmable wires
- Compute engines are defined by the user
- Data flows through customized deep pipelines



FPGA architecture

Pipeline parallelism can be combined with other types of parallelism:

- task parallelism (multiple pipelines)
- superscalar execution (multiple independent instructions executing in parallel)


FPGA Operation Execution Parallelism
Time
Data
Op1
Op2
Op3
Op4
Op5
Op6
Op7
Op8
Op9
Op...
Deep Pipeline Paralellism

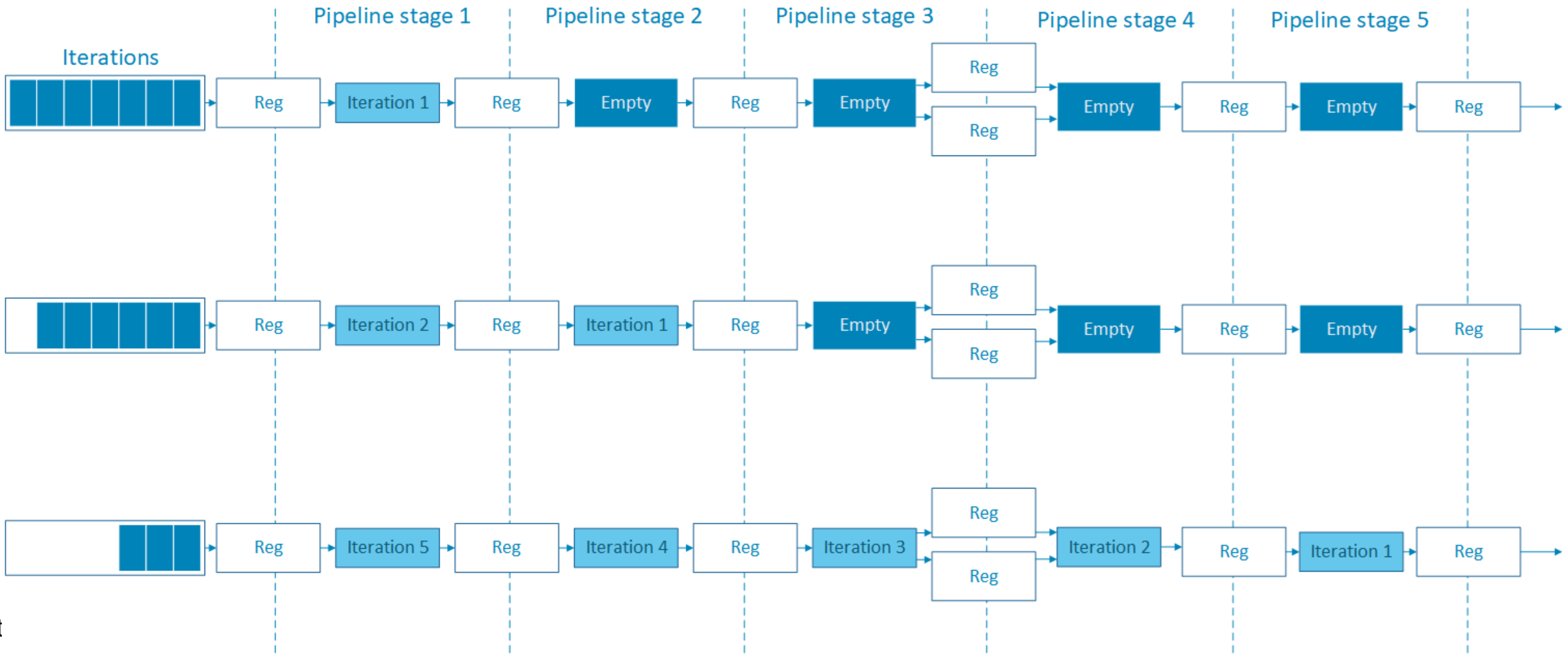
FPGA architecture

FPGA Advantages:

- Efficiency: no need for control units, instruction fetch and other execution overhead
- Flexibility: can be reconfigured to accommodate different functions and custom data type
- Custom Instructions: instructions not supported by CPUs can be easily implemented and executed on FPGAs
- Rich I/O: FPGA core can interact directly with various memory and custom interfaces

FPGA execution

- The operations in the kernel are laid out spatially
- The key to performance is to keep the deep pipeline fully occupied
- With single-task kernels, the FPGA attempts to pipeline loop execution
- Every clock cycle, successive iterations of the loop enter the first stage of the pipeline



Architecture summary

- CPUs:
  - the most widely used generic processors
    (not as compute-dense as GPUs, and not as compute-efficient as FPGAs)
  - modern CPUs support SIMD instructions
  - the most flexible with the broadest library support
- GPUs:
  - the most compute dense, massively data-parallel accelerators
  - employ a single instruction, multiple threads execution model (SIMT)
  - optimized for continuous reads and writes
- FPGAs:
  - compute engines defined by the user
  - reconfigurable, fine-grained datapath (1-bit resolution)
  - dependent kernels are deeply pipelined, independent are executed in parallel

Workload examples

- GPU: image processing, deep learning, data analysis
  - Little or no dependency across processed data
  - Simple control flow (minimal branching and loop divergence)
  - Matches data-types supported by the GPU
- FPGA: genomics sequencing, machine learning, image lossless compression
  - Suitable for algorithms that are easily expressed in serial code and may have dependencies across data elements
- CPU: task orchestration for heterogeneous computing system
  - Can still have superior performance in compute applications when vector, memory, and thread optimizations are applied

TinyML - motivation

How many transistors does it cost to implement a single multiplication in a network?

1. As a dedicated digital circuit:
   - m.in. 38 transistors/bit
2. As a reconfigurable digital circuit:
   - 32k transistors/weight
     (in 40 LUTs)

TinyML - motivation

- The average consumer CPU will draw between 65 to 85 watts of power, while the average GPU consumes anywhere between 200 to 500 watts.
- A typical microcontroller draws power in the order of milliwatts or microwatts, which is a thousand times less power consumption. This energy efficiency enables the TinyML devices to run on battery power while running ML applications on the edge.
- TinyML with its support for frameworks that include TensorFlow Lite, uTensor, and Arm’s CMSIS-NN, brings together AI and small connected devices.

TinyML

Definition: *This is a concept of implementing* ***mainly deep neural networks*** *directly on embedded devices with* ***highly limited resources*** *.*

This approach may include:

- adapting (reducing) the network architecture to reduced hardware resources
- reduction of power consumption during processing (e.g. for use in battery powered applications)

TinyML can be understood as a network algorithm compression method.

Benefits of TinyML

- **Energy efficiency**: Microcontrollers consume very little power, which delivers benefits in remote installations and mobile devices.
- **Low latency**: By processing data locally at the edge, data doesn't need to be transmitted to the cloud for inference. This greatly reduces device latency.
- **Privacy**: Data can be stored locally, not on cloud servers.
- **Reduced bandwidth**: With decreased dependency on the cloud for inference, bandwidth concerns are minimized.

- The future of TinyML using MCUs is promising for small edge devices and modest applications where an FPGA, GPU or CPU are not viable options.

Related content

- Data Parallel C++ (programming of heterogeneous systems using SYCL)
- Intel oneAPI programming guide
- Intel DevCloud
- TinyML is bringing deep learning models to microcontrollers
  https://thenextweb.com/news/tinyml-deep-learning-microcontrollers-syndication
- ESP32-CAM: TinyML Image Classification
  https://mjrobot.org/2022/02/10/esp32-cam-tinyml-image-classification-fruits-vs-veggies/