Software and Hardware for AI

Overview

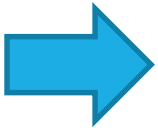
What is AI?

Intelligence: ability to **extract knowledge** from observations

This knowledge is used to **solve tasks in different contexts and environments** (automation)

Old way: Memorize

- Human experts code the machines
- Goods: we know what we are doing.
- Bads: requires **explicit** solutions (not available for some problems).



Modern way: Generalize

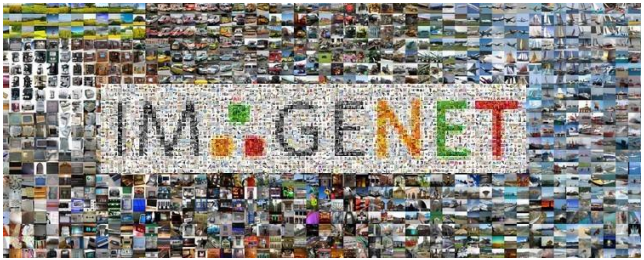
- Let machines teach themselves how to solve a problem (**implicit**).
- Goods: universally applicable
- Bads: lack of understandability/robustness.
- Requires **training**.

Where did the revolution in AI come from?

- The use of GPUs for computation.
- The share of huge datasets on Internet.
- Github/Arxiv new ways of sharing research.
- The return of representation learning.

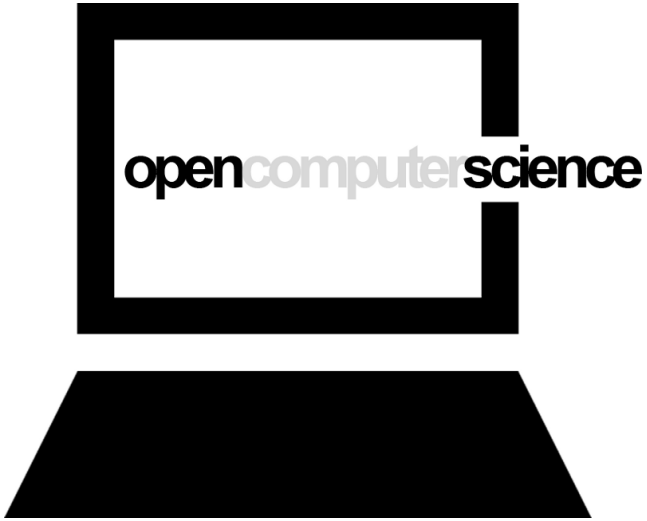
Where did the revolution in AI come from?

- The use of GPUs for computation.
- The share of huge datasets on Internet.
- Github/Arxiv new ways of sharing research.
- The return of representation learning.



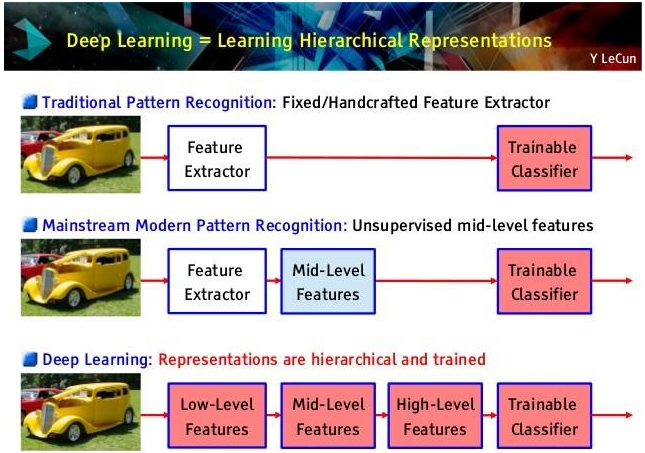
Where did the revolution in AI come from?

- The use of GPUs for computation.
- The share of huge datasets on Internet.
- Github/Arxiv new ways of sharing research.
- The return of representation learning.



Where did the revolution in AI come from?

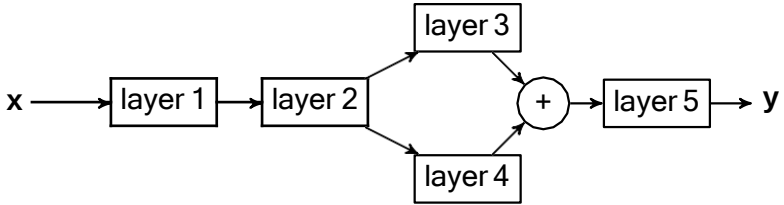
- The use of GPUs for computation.
- The share of huge datasets on Internet.
- Github/Arxiv new ways of sharing research.
- The return of representation learning.



Where did the revolution in AI come from?

Main idea

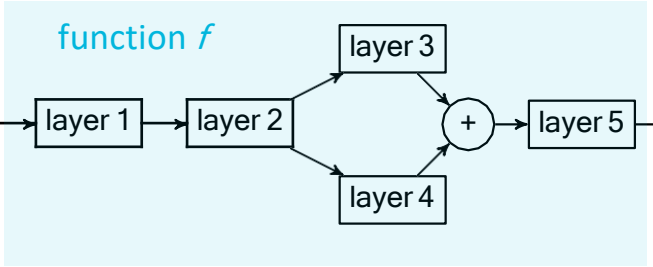
- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)



Where did the revolution in AI come from?

Main idea

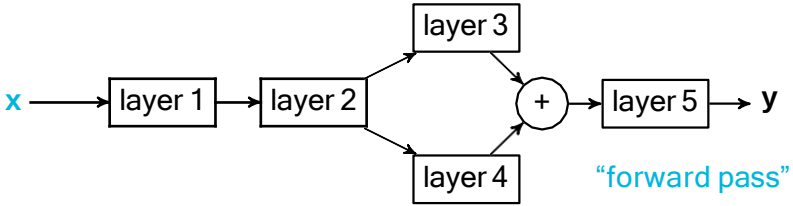
- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)



Where did the revolution in AI come from?

Main idea

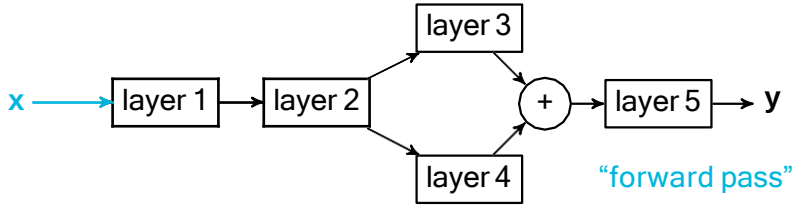
- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)



Where did the revolution in AI come from?

Main idea

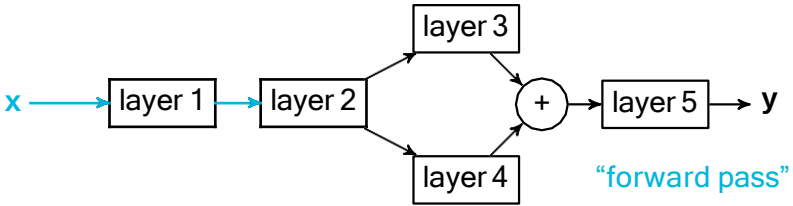
- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)



Where did the revolution in AI come from?

Main idea

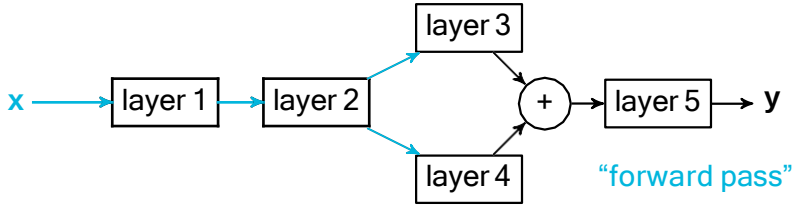
- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)



Where did the revolution in AI come from?

Main idea

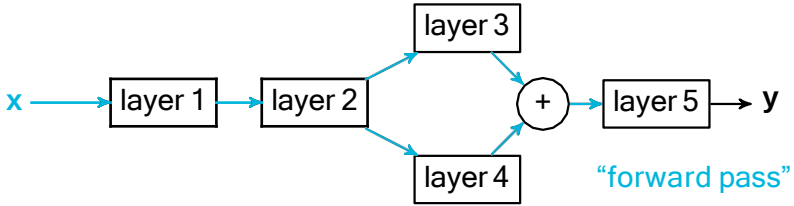
- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)



Where did the revolution in AI come from?

Main idea

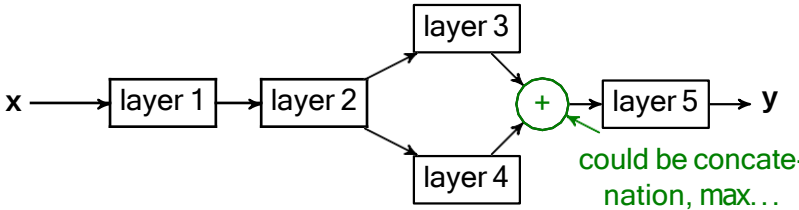
- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)



Where did the revolution in AI come from?

Main idea

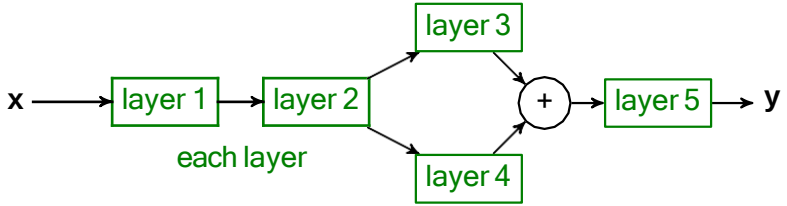
- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)



Where did the revolution in AI come from?

Main idea

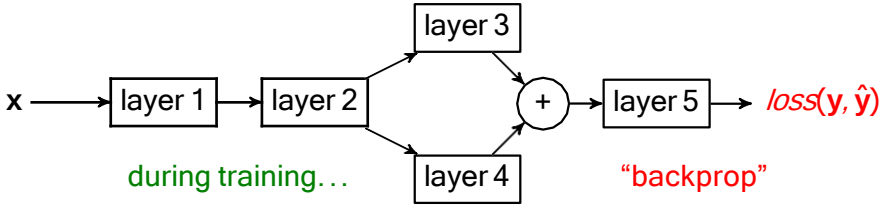
- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)



Where did the revolution in AI come from?

Main idea

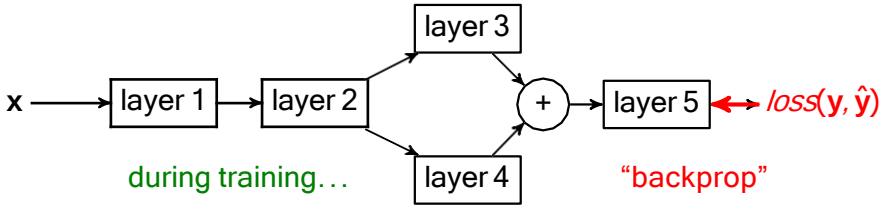
- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)



Where did the revolution in AI come from?

Main idea

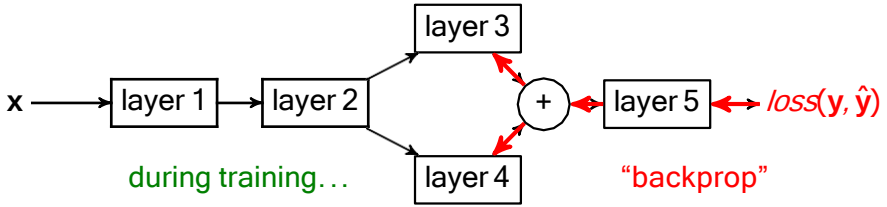
- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)



Where did the revolution in AI come from?

Main idea

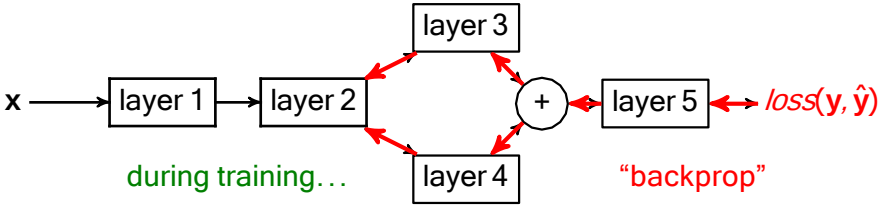
- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)



Where did the revolution in AI come from?

Main idea

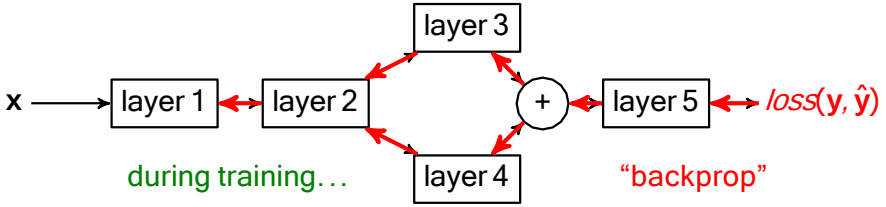
- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)



Where did the revolution in AI come from?

Main idea

- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)



Where did the revolution in AI come from?

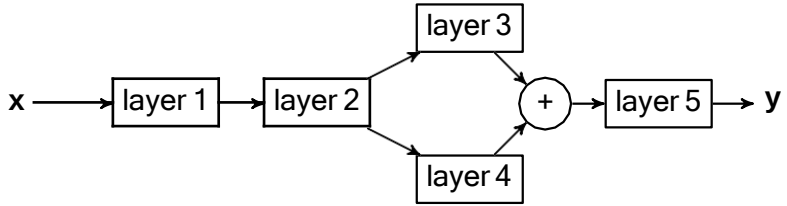
Main idea

- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)

Where did the revolution in AI come from?

Main idea

- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)

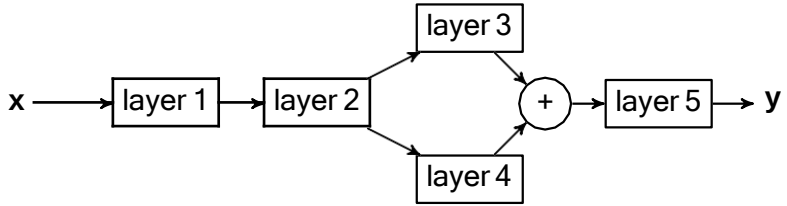


Number of layers, choice of the architecture are **hyperparameters**

Where did the revolution in AI come from?

Main idea

- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)



Number of layers, choice of the architecture are **hyperparameters**

Where did the revolution in AI come from?

Layers

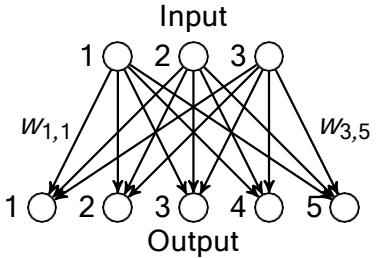
- $\mathbf{x} \mapsto h(\mathbf{W}\mathbf{x} + \mathbf{b})$.
  - h is a nonlinear parameterwise function (often without parameters),
  - $\mathbf{W}$ is a tensor:
    - Can be agnostic of the structure: **fully-connected layers**,
    - Can be structure-dependent: **convolutional layers**.

Where did the revolution in AI come from?

Layers

- $\mathbf{x} \mapsto h(\mathbf{W}\mathbf{x} + \mathbf{b})$.
  - h is a nonlinear parameterwise function (often without parameters),
  - $\mathbf{W}$ is a tensor:
    - Can be agnostic of the structure: **fully-connected layers**,
    - Can be structure-dependent: **convolutional layers**.

Fully connected layer

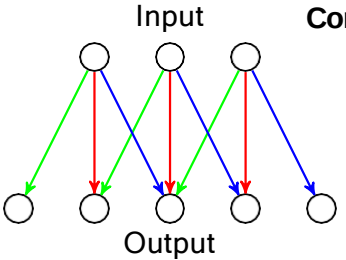


$$\begin{matrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} & w_{1,5} \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} & w_{2,5} \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} & w_{3,5} \end{matrix}$$

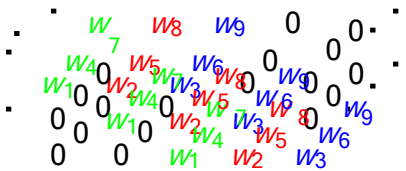
Where did the revolution in AI come from?

Layers

- $\mathbf{x} \mapsto h(\mathbf{W}\mathbf{x} + \mathbf{b})$.
  - h is a nonlinear parameterwise function (often without parameters),
  - $\mathbf{W}$ is a tensor:
    - Can be agnostic of the structure: **fully-connected layers**,
    - Can be structure-dependent: **convolutional layers**.



Convolutional layer



What are the potential targets ?

- CPU
- GPU
- ASICs
  - IPU (Graphcore)
  - TPU (Google)
  - Edge TPU (Google)
  - Eyeriss (MIT)
  - ...
- FPGA

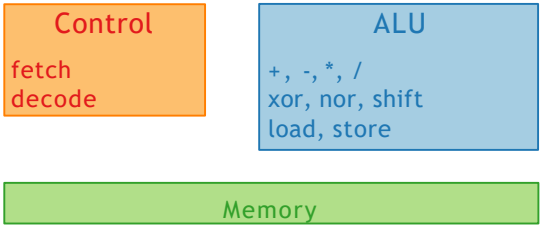
What are the differences between them ?

Which use case for each target ?

What are the potential targets ?

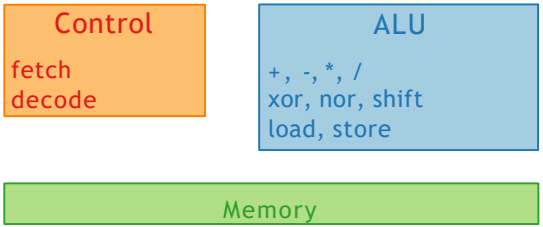
- CPU
- GPU
- ASICs
  - IPU (Graphcore)
  - TPU (Google)
  - Edge TPU (Google)
  - Eyeriss (MIT)
  - ...
- FPGA

What are the elements of a CPU ?



- Control: Fetches and decodes instructions, controls the ALU,
- ALU: Arithmetical and Logical Unit, performs all computations, exchanges data between memory and register file,
- Memory: Stores data.

What are the elements of a CPU ?

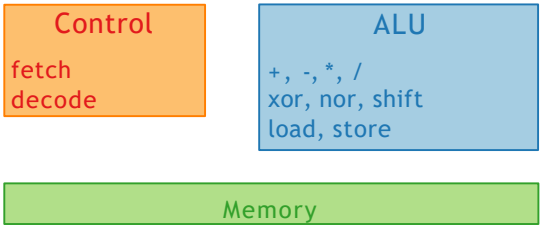


- There are many ways to increase the overall performance of a CPU architecture.
- Two key features will be described:

  1- Increasing the computational parallelism

  2- Reducing data accesses time with close and fast memories.

What are the elements of a CPU ?



- There are many ways to increase the overall performance of a CPU architecture.
- Two key features will be described:

  1- Increasing the computational parallelism

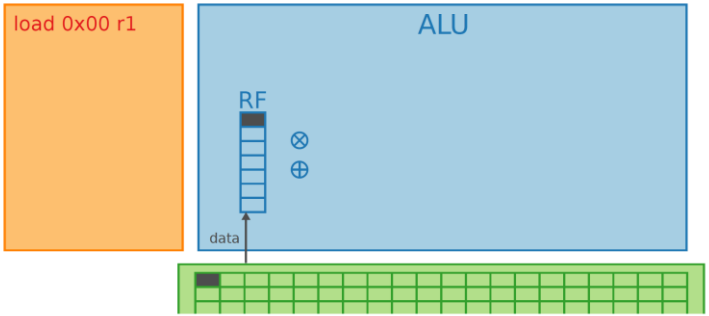
  2- Reducing data accesses time with close and fast memories.

Increasing Parallelism : SIMD

- **SIMD**: Single Instruction Multiple Data Hardware feature in ALU
- Available in Intel CPUs (SSE, AVX)
- Available in ARM CPUs (Neon)

Increasing Parallelism : SIMD

"Normal" Single Instruction Single Data (SISD) example



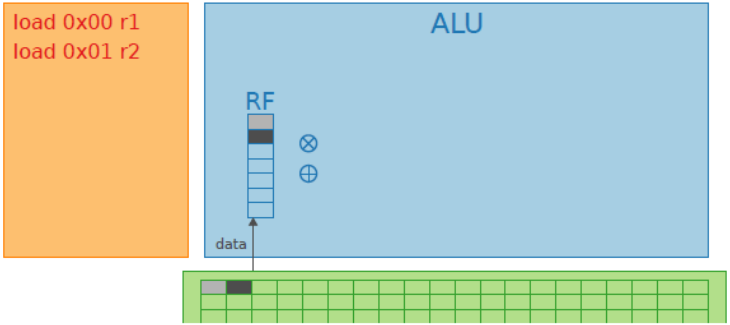
1- Load data from memory to register file

2- Execute addition

3-Execute addition

Increasing Parallelism : SIMD

"Normal" Single Instruction Single Data (SISD) example



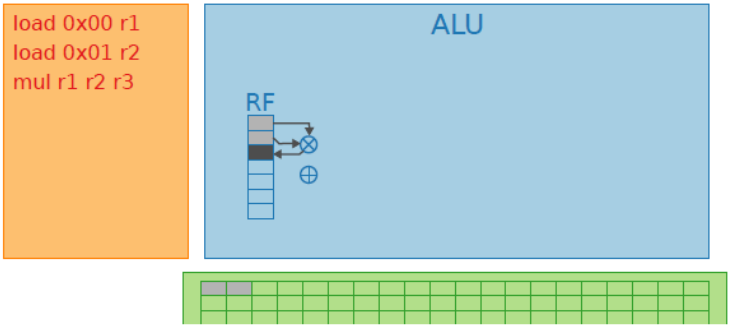
1- Load data from memory to register file

2- Execute addition

3-Execute addition

Increasing Parallelism : SIMD

"Normal" Single Instruction Single Data (SISD) example



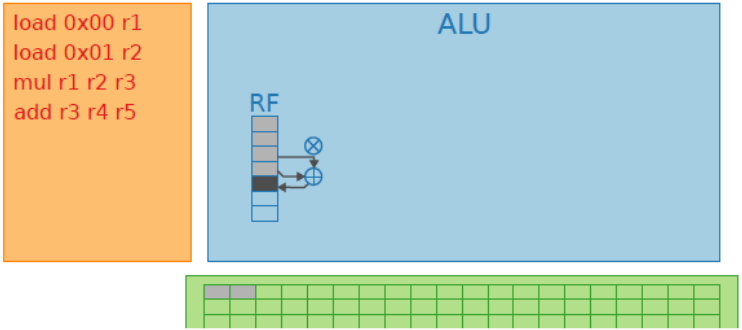
1- Load data from memory to register file

2- Execute addition

3-Execute addition

Increasing Parallelism : SIMD

"Normal" Single Instruction Single Data (SISD) example

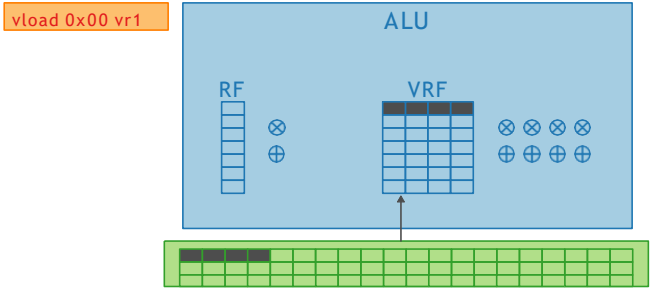


1- Load data from memory to register file

2- Execute addition

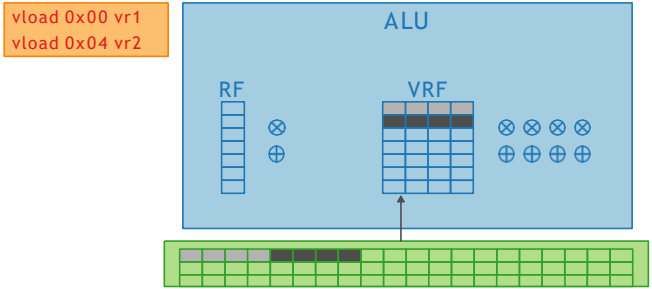
3-Execute addition

Increasing Parallelism : SIMD



- Single Instruction Multiple Data Additional hardware
- Parallel load
- Parallel arithmetic
- Increase number of computations per instruction

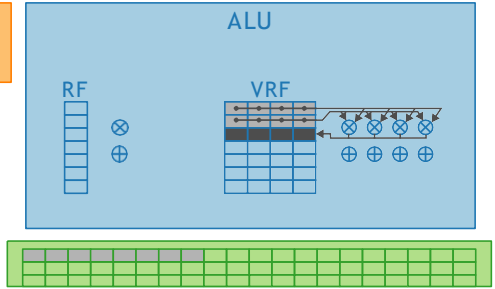
Increasing Parallelism : SIMD



- Single Instruction Multiple Data
- Additional hardware
- Parallel load
- Parallel arithmetic
- Increase number of computations per instruction

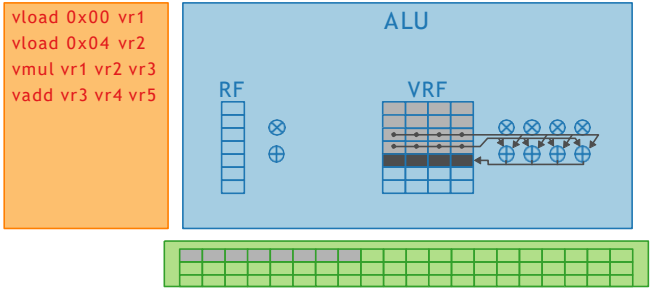
Increasing Parallelism : SIMD

```
vload 0x00 r1
vload 0x04 r2
vmul vr1 vr2 vr3
```



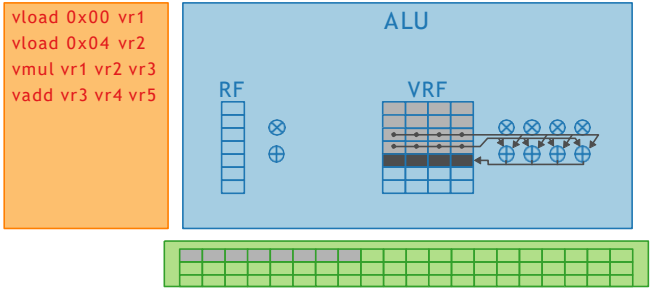
- Single Instruction Multiple Data
- Additional hardware
- Parallel load
- Parallel arithmetic
- Increase number of computations per instruction

Increasing Parallelism : SIMD



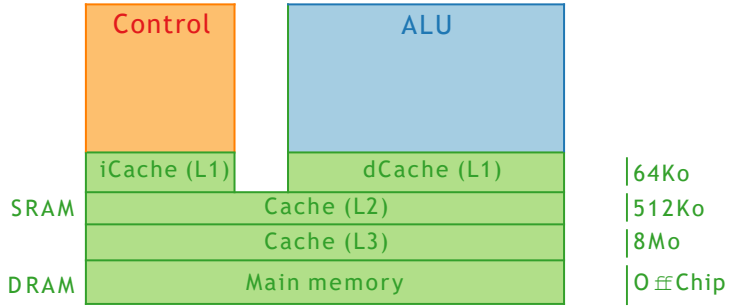
- Single Instruction Multiple Data
- Additional hardware
- Parallel load
- Parallel arithmetic
- Increase number of computations per instruction

Increasing Parallelism : SIMD



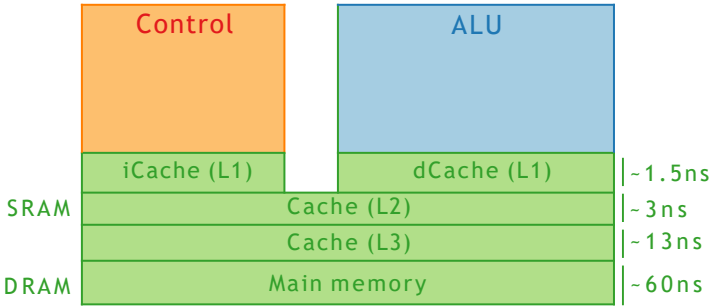
- Increased parallelism
- Multiple quantization formats handled (8-, 16-, 32-, 64-bit)
- The more quantized, the more parallel
- Need aligned data in memory

Cache hierarchy



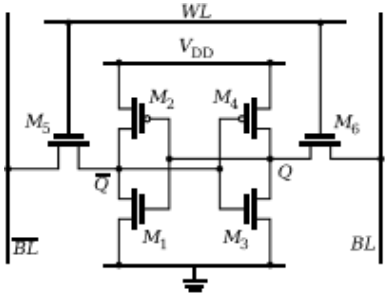
- Cache Hierarchy
- SRAM vs DRAM
- Primary access
- Cache Hit
- Cache Miss

Cache hierarchy

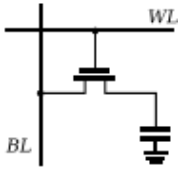


- Cache Hierarchy
- SRAM vs DRAM
- Primary access
- Cache Hit
- Cache Miss

Cache hierarchy



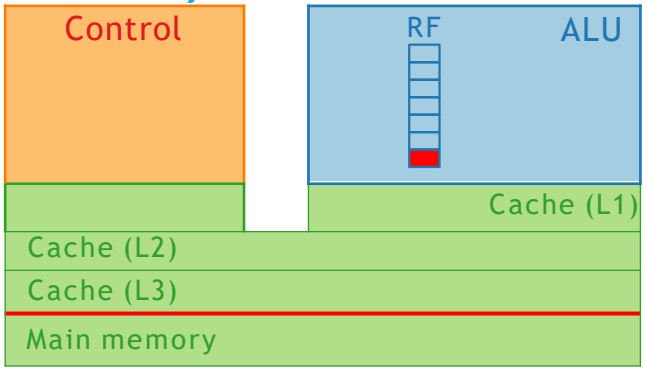
SRAM



DRAM

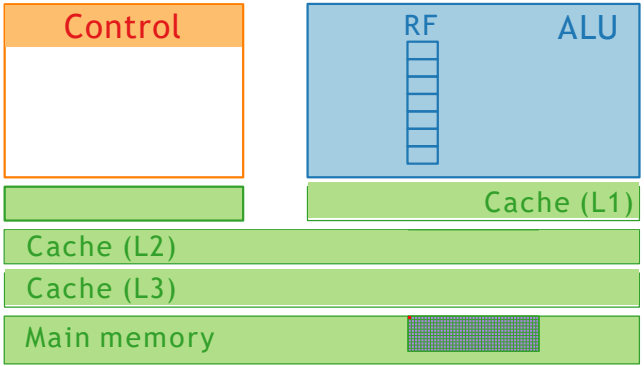
- SRAM 6T (typically) vs DRAM 1T
- SRAM is more expensive
- DRAM is denser
- DRAM`needsrefreshment
- SRAM is faster

Cache hierarchy



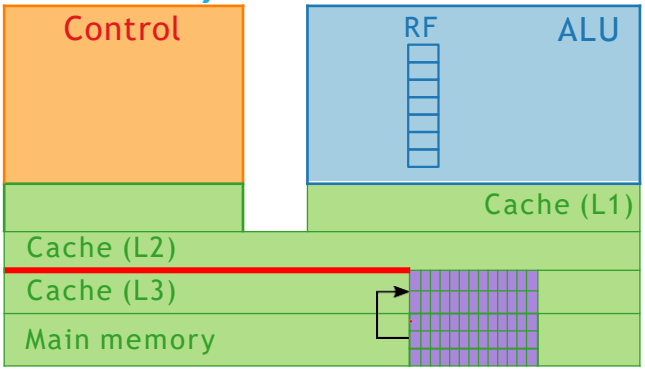
- Cache Hierarchy
- SRAM vs DRAM
- Primary access
- Cache Hit
- Cache Miss

Cache hierarchy



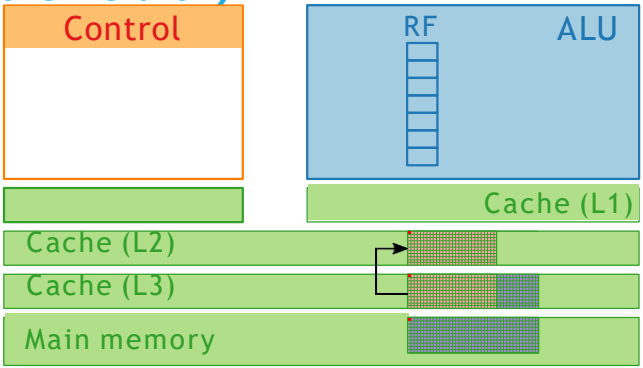
- Cache Hierarchy
- SRAM vs DRAM
- Primary access
- Cache Hit
- Cache Miss

Cache hierarchy



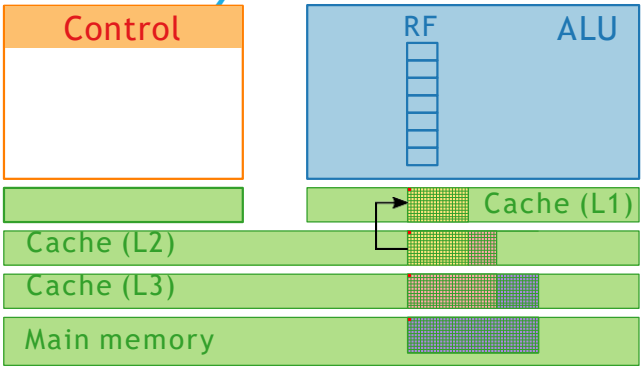
- Cache Hierarchy
- SRAM vs DRAM
- Primary access
- Cache Hit
- Cache Miss

Cache hierarchy



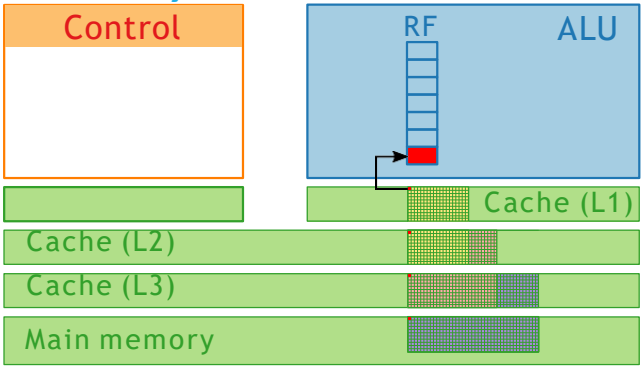
- Cache Hierarchy
- SRAM vs DRAM
- Primary access
  - Cache Hit
  - Cache Miss

Cache hierarchy



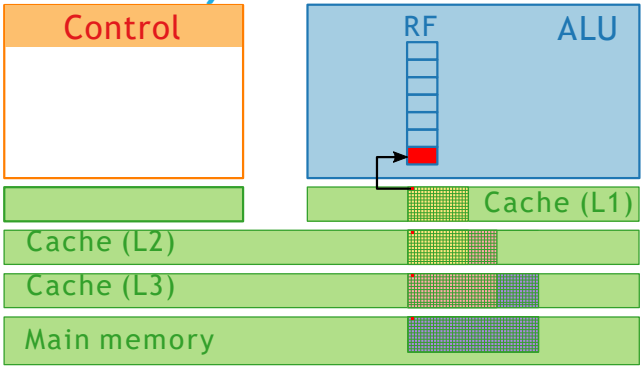
- Cache Hierarchy
- SRAM vs DRAM
- Primary access
- Cache Hit
- Cache Miss

Cache hierarchy



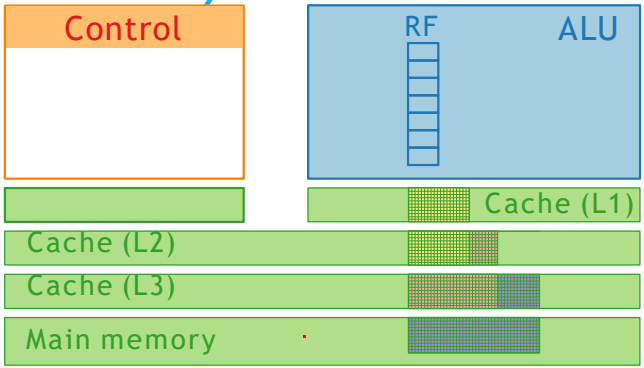
- Cache Hierarchy
- SRAM vs DRAM
- Primary access
- Cache Hit
- Cache Miss

Cache hierarchy



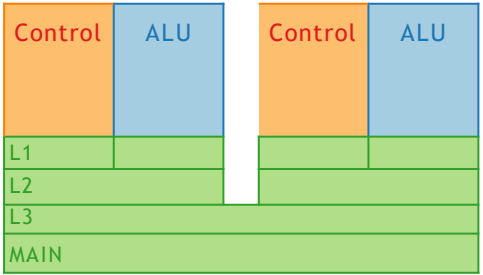
- Cache Hierarchy
- SRAM vs DRAM
- Primary access
- Cache Hit
- Cache Miss

Cache hierarchy



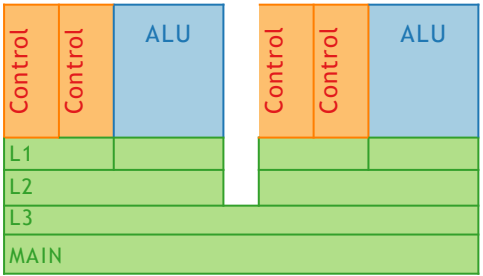
- Cache Hierarchy
- SRAM vs DRAM
- Primary access
- Cache Hit
- Cache Miss

Multicore



- Add CPU cores on the same chip
- Last Level Cache (LLC) is shared between cores
- Linear increasing of computing capacity

Simultaneous Multi Threading (SMT)



- Known as "Hyperthreading" which is Intel's own SMT implementation
- Multiple instruction threads (here 2) are processed on each core
- Sublinear increasing of computing capacity, resources are shared

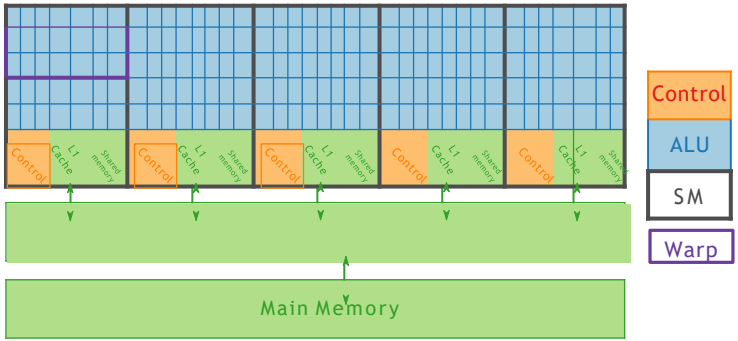
What are the potential targets ?

- CPU
- GPU
- ASICs
  - IPU (Graphcore)
  - TPU (Google)
  - Edge TPU (Google)
  - Eyeriss (MIT)
  - ...
- FPGA

What are the differences between them ?

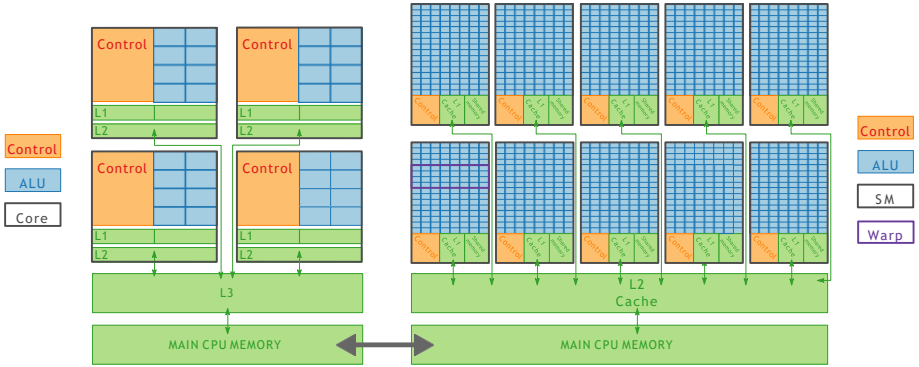
Which use case for each target ?

GPU



- GPUs have a huge computation power
- Simpler control
- Each core execute warps of 32 threads (Nvidia)
- Same instructions in each thread, but different execution contexts
- Yields higher throughput, but also higher latency

CPU vs GPU



- Sequential vs Parallel

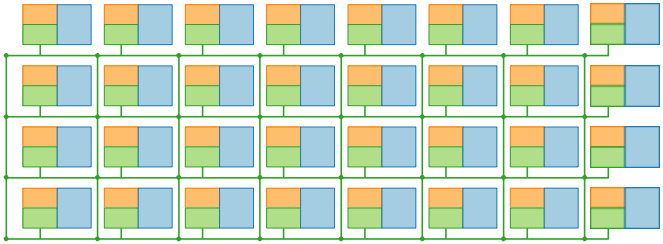
What are the potential targets ?

- CPU
- GPU
- ASICs
  - IPU (Graphcore)
  - TPU (Google)
  - Edge TPU (Google)
  - Eyeriss (MIT)
  - ...
- FPGA

What are the differences between them ?

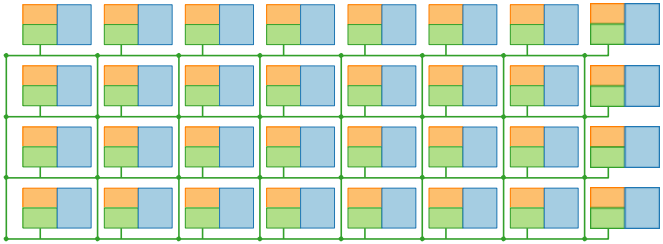
Which use case for each target ?

ASICs : Example of Graphcore's IPU



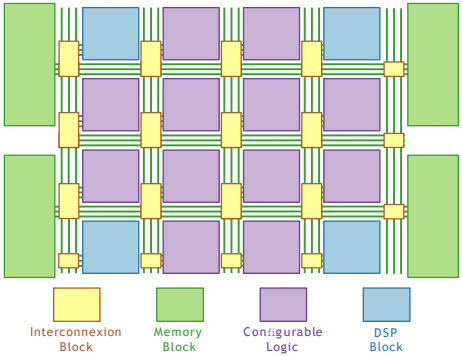
- Manycore approach :
- Each core handles 6 independent threads
- Fully distributed cache memory
- 256Ko / core

ASICs : Example of Graphcore's IPU



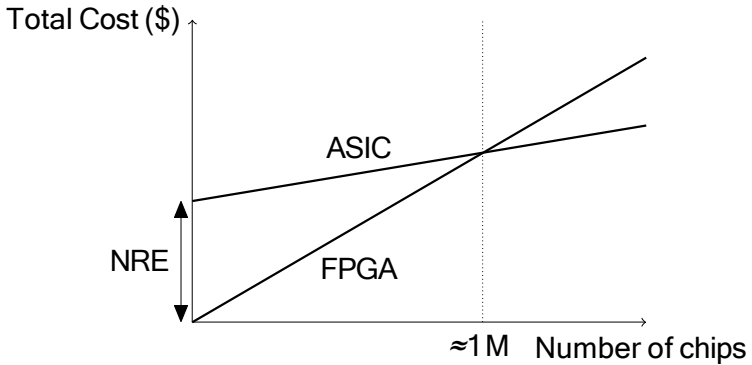
- Claims better efficiency ($/Gops, kWh/Gops)
- Claims faster inference
- Cautious: lack of independent benchmarks

FPGAs : (Re)Configurable Integrated Circuits



- Designing a custom architecture
- No "Non Recurring Engineering" compared to custom ASIC
- Prototyping
- Small markets

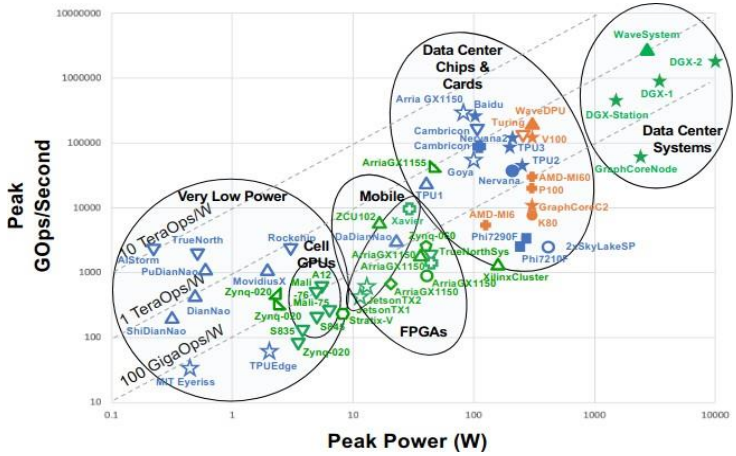
FPGAs : (Re)Configurable Integrated Circuits



Remote vs. Local use cases

Use case	Key features	Targets
Remote	Throughput Cost ($/Gops) Scaling	GPU TPU IPU
Local	Availability Power consumption Cost ($/unit) Latency Data privacy	CPU Edge TPU Embedded GPU (Tegra) FPGA

Power!!

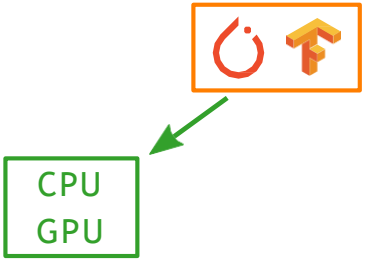


A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi and J. Kepner, "Survey and Benchmarking of Machine Learning Accelerators," 2019 IEEE High Performance Extreme Computing Conference (HPEC), 2019, pp. 1-9, doi: 10.1109/HPEC.2019.8916327.

And what about software ?

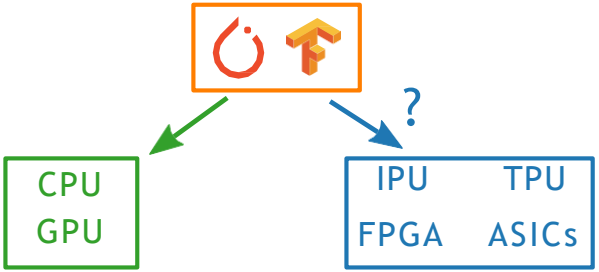
- High level frameworks
- Broadly used
- Programmed and optimized to be used on CPU and GPU
- Not systematically ported on each target
- Supporting these frameworks becomes critical for chips makers

And what about software ?



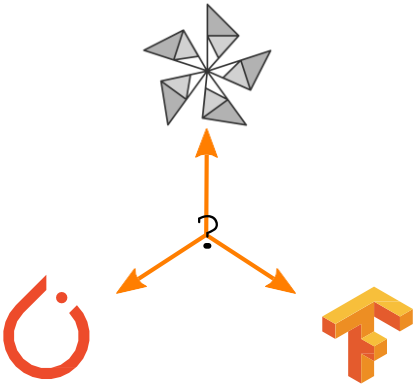
- High level frameworks
- Broadly used
- Programmed and optimized to be used on CPU and GPU
- Not systematically ported on each target
- Supporting these frameworks becomes critical for chips makers

And what about software ?

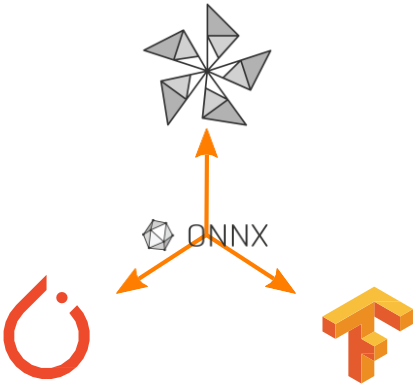


- High level frameworks
- Broadly used
- Programmed and optimized to be used on CPU and GPU
- Not systematically ported on each target
- Supporting these frameworks becomes critical for chips makers

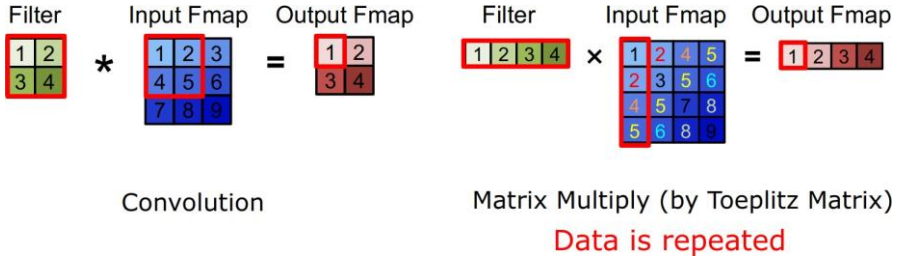
Interoperability ?



Interoperability ?



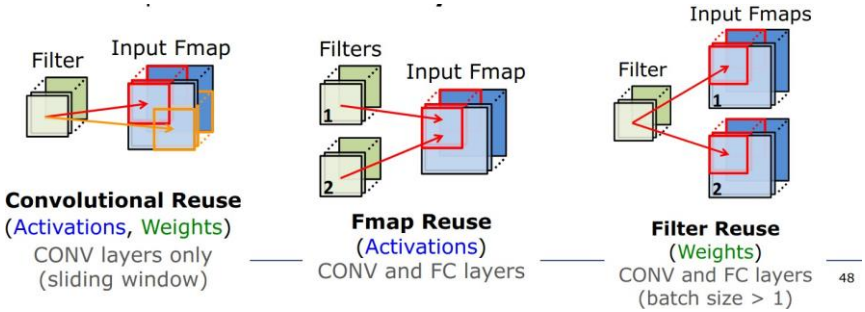
Software for CPU & GPU: matrix multiplication



- Use existing optimized libraries
- Repeating Data

From: http://eyeriss.mit.edu/2019_neurips_tutorial.pdf

Software for CPU & GPU: matrix multiplication



- Keep data in caches
- Activations and / or weights

From: http://eyeriss.mit.edu/2019_neurips_tutorial.pdf