Practical course: Advanced System Programming

Hardware Accelerators

https://dse.in.tum.de/

Antoine Kaufmann

About this lab

- What you will learn: **whole-system perspective on hardware acceleration**
  - What components are involved? (driver, interconnect, accelerator)
  - How do they communicate efficiently?
- What you will *not* learn
  - Hardware implementation at the HW level
  - Specific HW acceleration framework
  - Acceleration techniques for specific application domains

- You will need a working docker setup (Windows, Linux confirmed working)
  - Strong recommendation: use devcontainer config, e.g. with vscode

Is Progress in Computing Doomed?

- General purpose processors not getting faster
  - Multi-core gains have slowed too
- Demand for compute is still growing rapidly
  - New application domains, more data, deeper analysis
- **Need to use transistors more efficiently**

Specialization as The Way Forward

General-purpose computer → system built for a specific task

Hardware **Software**

Shor …cific …on

Long …odels (quantum, proteomics, …)

Post-Moore Systems: Hardware-Software co-Design

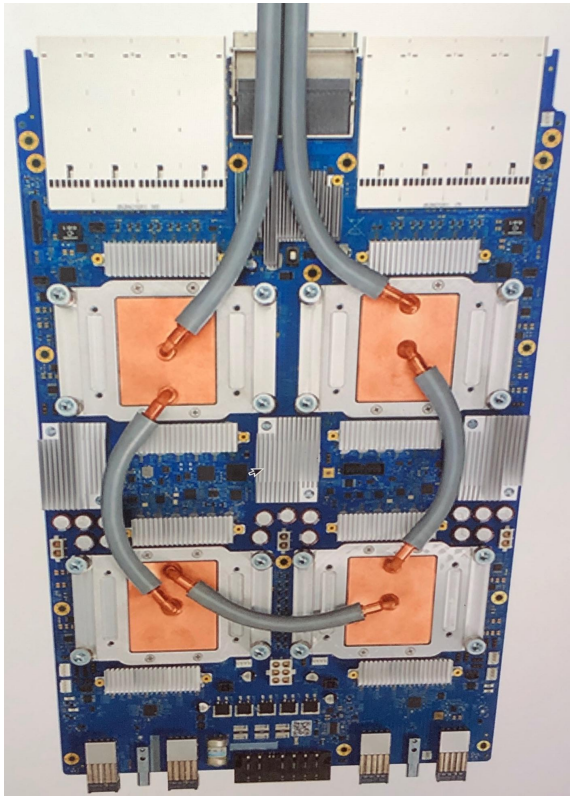
- Carefully split functionality between HW & SW
- Jointly design optimized software and hardware

Example Post-Moore System: Google's Deep Learning Stack

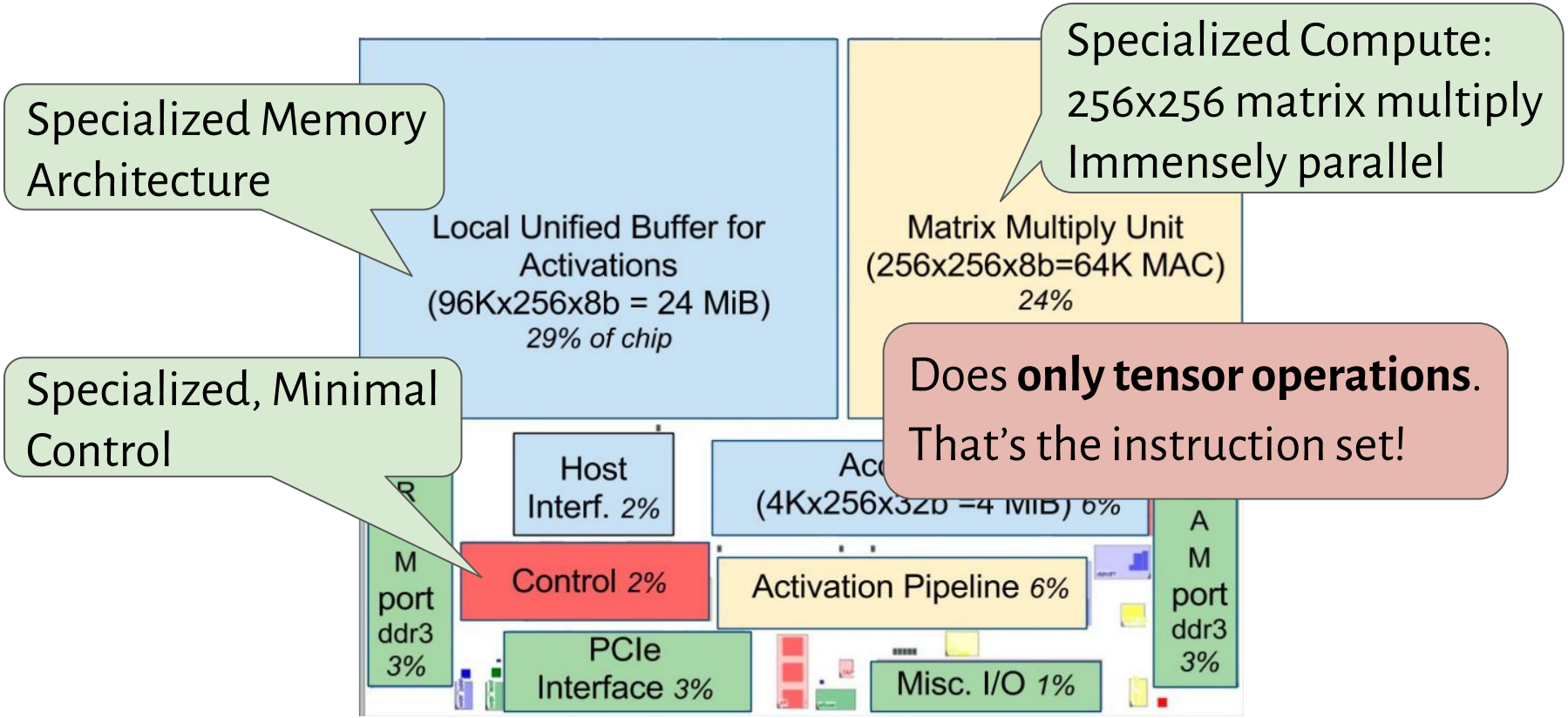
Goal: Accelerate neural networks

TPU (accelerator) + TensorFlow =

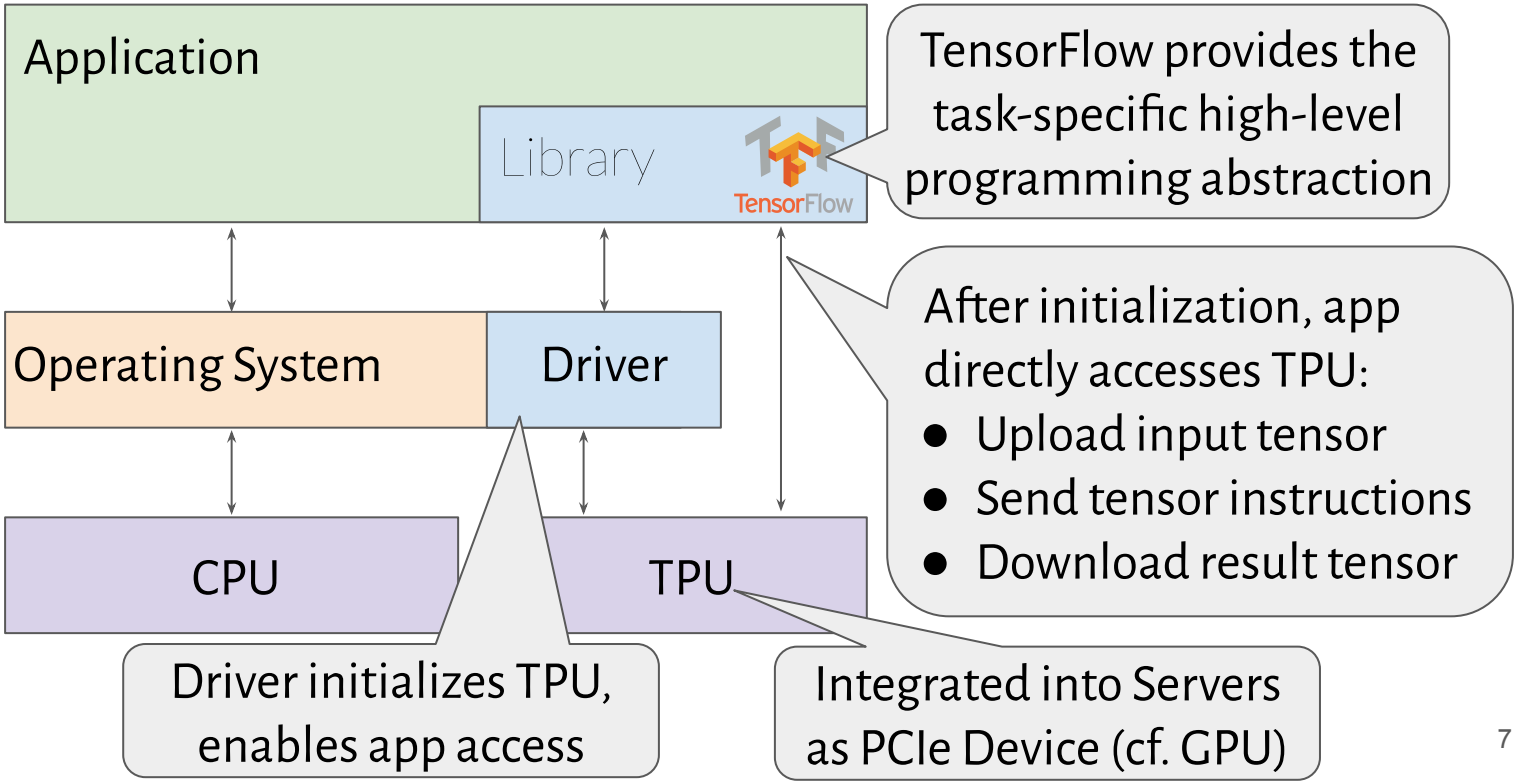
- 15-30x faster than CPU and GPU
- 30-80x less energy per operation
- <0.5x chip area



Why is the TPU faster? Specialization!



The Full System: TPU + TensorFlow



This Lab

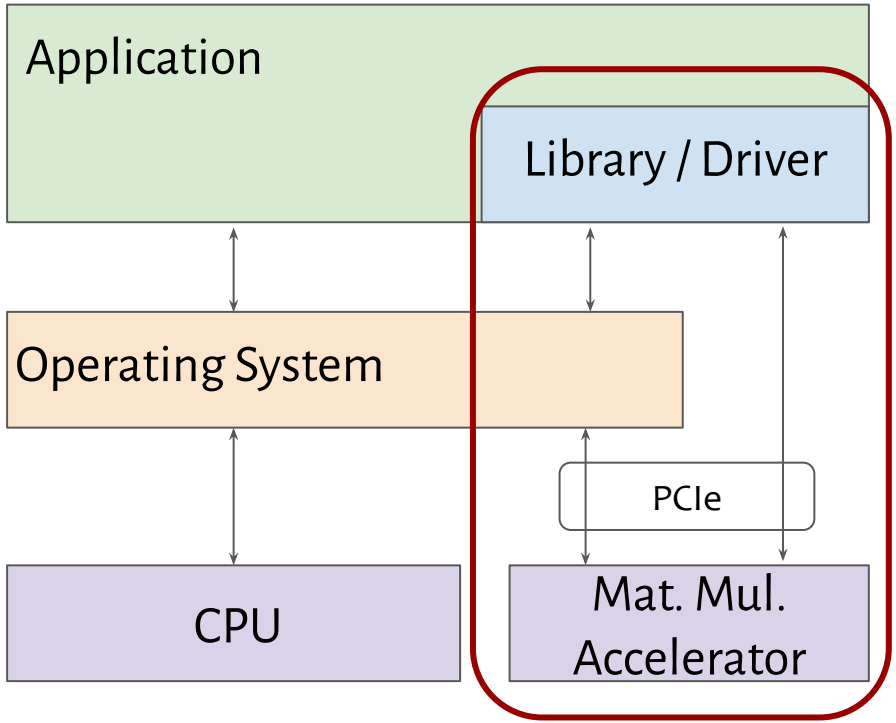
Design Full HW & SW stack for a simple matrix multiplication accelerator

- Focus on HW-SW Communication

You will Implement (part of):

- User-level PCIe Driver
- HW Accelerator (Simulation Model)

Split into 3 sub-tasks



Background: PCI(e)

We will build an accelerator that integrates through PCIe

PCIe is an I/O interconnect, connecting devices with rest of the system.

Device interface comprises:

- Enumeration
- PIO/MMIO Device Registers
- Direct Memory Access (DMA)
- Interrupts (will skip for this lab)

PCI(e) Background: Enumeration

Goal: allow SW to locate and identify devices

- What devices are available?
  - Vendor and Device IDs
- What features does a device support?
  - Capabilities and other registers
- Where are a devices PIO registers located?
  - Base Address Registers (BARs)

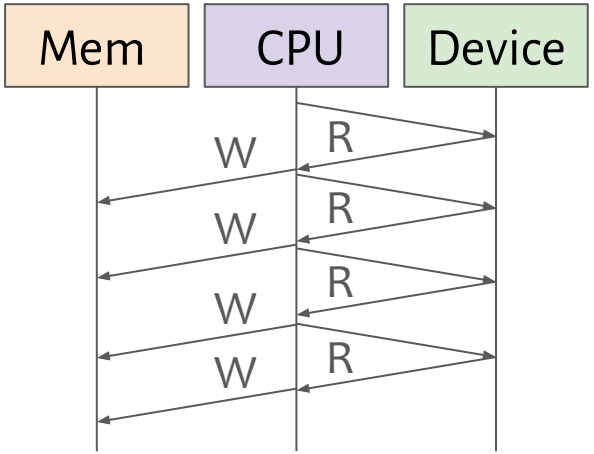
```
$ sudo lspci -vnn
00:02.0 VGA compatible controller [0300]: [8086:5917] (prog-if 00 [VGA])
        Subsystem: Dell UHD Graphics 620 [1028:081b]
        Flags: bus master, fast devsel, latency 0, IRQ 141, IOMMU group 0
        Memory at eb000000 (64-bit, non-prefetchable) [size=16M]
        Memory at a0000000 (64-bit, prefetchable) [size=256M]
        I/O ports at f000 [size=64]
        Expansion ROM at 000c0000 [virtual] [disabled] [size=128K]
        Cap: [40] Vendor Specific Information: Len=0c <?>
        Cap: [70] Express Root Complex Integrated Endpoint, MSI 00
        Cap: [ac] MSI: Enable+ Count=1/1 Maskable- 64bit-
        Cap: [d0] Power Management version 2
        Cap: [100] Process Address Space ID (PASID)
        Cap: [200] Address Translation Service (ATS)
        Cap: [300] Page Request Interface (PRI)

00:14.0 USB controller [0c03]: [8086:9d2f] (rev 21) (prog-if 30 [XHCI])
        Subsystem: Dell Sunrise Point-LP USB 3.0 xHCI Controller [1028:081b]
        Flags: bus master, medium devsel, latency 0, IRQ 128, IOMMU group 3
        Memory at ec330000 (64-bit, non-prefetchable) [size=64K]
        Cap: [70] Power Management version 2
        Cap: [80] MSI: Enable+ Count=1/8 Maskable- 64bit+
...
```

PCI(e) Background: Programmed I/O (PIO) Registers

PIO/MMIO: CPU initiates Read or Write to Location (Register) on Device

- Old: I/O space, CPU uses special instructions
- New: memory mapped; CPU just accesses memory



- Devices exposes many registers at different addresses
  - PCI: up to 6 (3) BARs with up to 2^{32} (2^{64}) bytes each
- Device can actively react to register accesses
  - Mental Model (RPC):
    CPU sends read/write request message, device responds

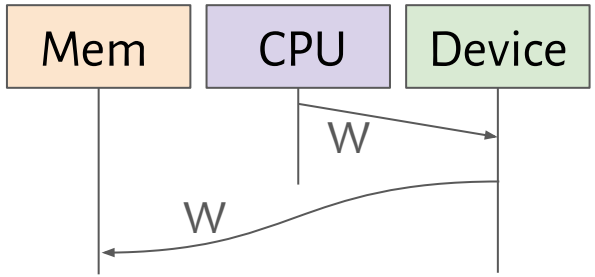
- "Unidirectional": always issued by CPU
- Synchronous: CPU core waits for response before retiring instruction

PCI(e) Background: Direct Memory Access (DMA)

DMA: Device initiates Read or Write access to Host Memory

- Problem: transferring large quantities of data via PIO wastes many CPU cycles
- Solution: allow device to read/write memory directly without CPU involvement

- Each PCI(e) device can directly access host memory by *physical address*
  - (In newer systems IOMMU can mediate, but for the lab we leave it disabled)
- How device uses this is depends on the specific device
  - Host sets up some parameters through MMIO first (e.g. physical address of buffers in memory)



- Unidirectional: always issued by device
- Asynchronous: device can request multiple transfers in parallel & do other things

Lab Step 1: Simple, Synchronous PIO

1. Design a simple PIO interface for the MatMul Accelerator
2. Implement HW Accelerator (Simulation Model) with that interface
3. Implement Driver for HW accelerator interface

Accelerator operations can take a long time → synchronously poll/wait for completion

- Driver writes I/O command registers on device to issue request
- Accelerator stores data in its internal buffers
- Driver repeatedly reads status register to detect completion
- Finally, driver reads data word-by-word from data registers

PIO Example: Serial Port

```
void serial_putc (uint8_t byte) {
    while ((READ_REG (LSR_REG) & LSR_THRE) == 0);
    WRITE_REG (THR_REG, byte);
}

uint8_t serial_getc () {
    while ((READ_REG (LSR_REG) & LSR_DR) == 0);
    return READ_REG (RBR_REG);
}
```

Lab Step 2: DMA for CPU-efficient Data Transfers

1. First, modify driver & accelerator avoid redundant data transfers
2. Then accelerate remaining transfers with DMA

Motivation: Copying data between CPU and accelerator consumes many CPU cycles

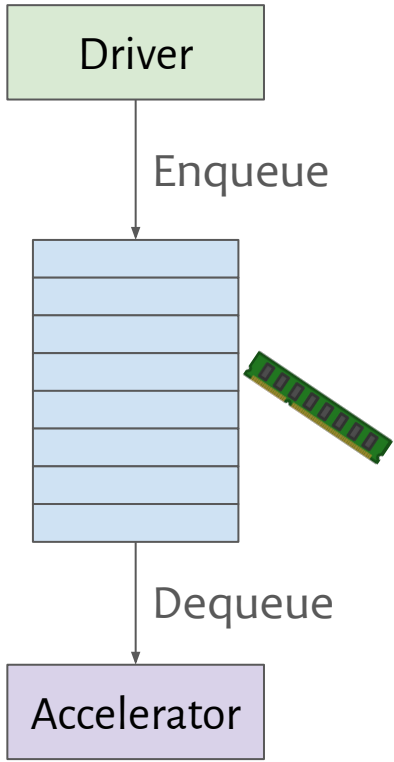
- Use DMA to allow accelerator to do the data transfer
- Driver writes ***physical*** memory address, length of matrix to registers
- Driver then writes control register to trigger operation
- Accelerator now reads/writes the matrix

Lab Step 3: Asynchronous Queue HW-SW Interface

1. Design asynchronous queue interface for mat. mul. Accelerator
2. Implement accelerator functionality
3. Implement software driver

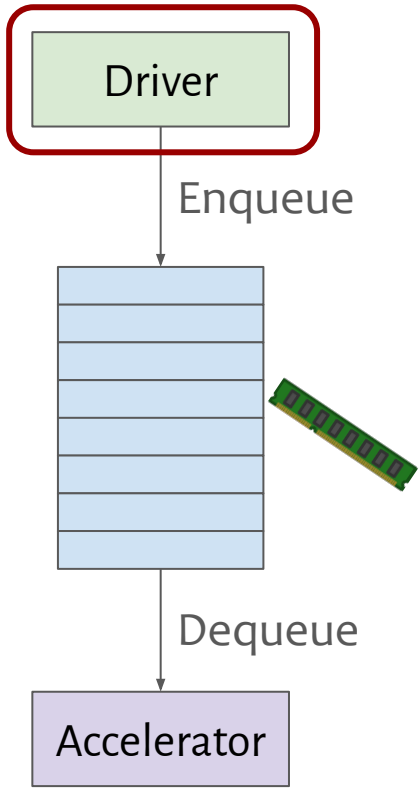
Motivation: accelerator idle between operations waiting for CPU

- Use in-memory queue for *asynchronous* operation of SW & HW
- Driver adds sequence operations into the queue
- Accelerator processes operations from queue at its own pace
  - Directly accesses queue through DMA
- Completion of operations signalled through separate queue



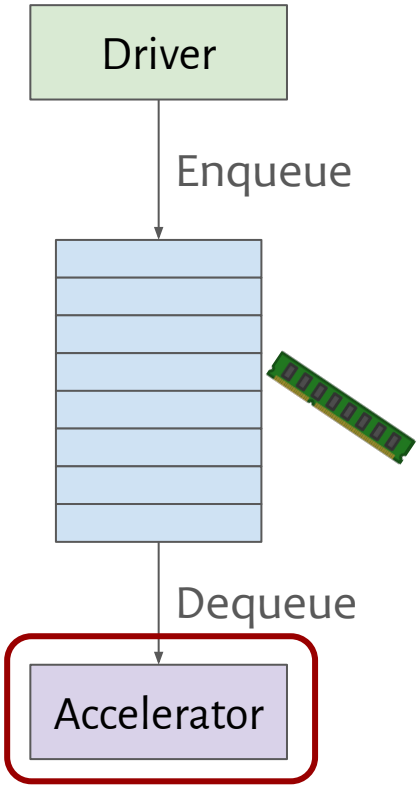
Lab Step 3: Asynchronous Queue — Driver

- Queues are kept in main memory
- Implemented as circular queues / ring-buffers
- Driver allocates & initializes queue,
  writes address & length to device registers
- CPU signals enqueue through tail pointer register
- Driver then asynchronously polls notification queues,
  waiting for device to write there



Lab Step 3: Asynchronous Queue — Accelerator

- Accelerator waits for driver to initialize queues
- Accelerator accesses queues via DMA operations
  - Only read request queue, only write response queue
- Driver explicitly signals enqueue through tail index register write
  - Polling is too expensive via DMA over PCIe
- Accelerator then fetches entries from request queue, processes them, writes to completion queue
- Optimization: coalesce DMAs for multiple queue entries



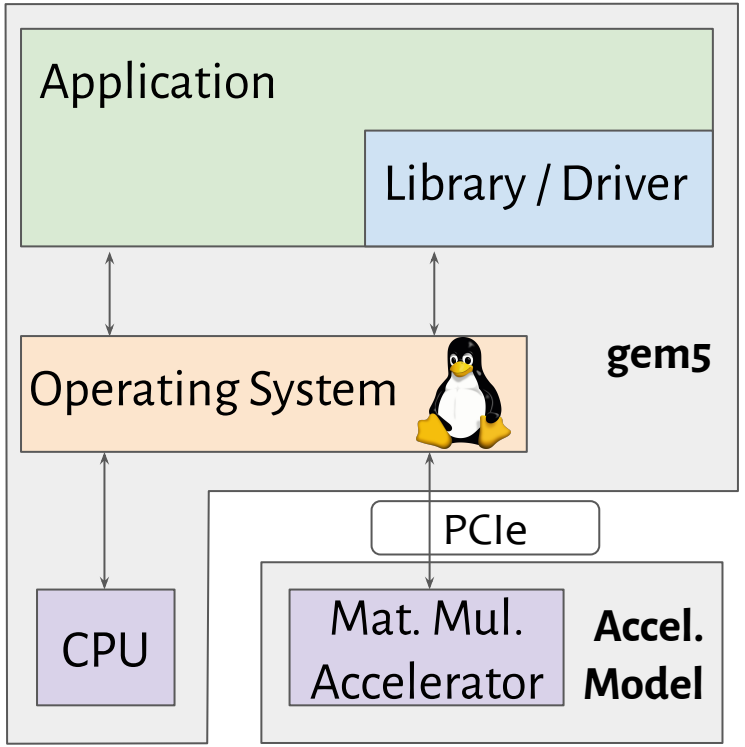
Lab Evaluation in Simulation

- **Goal:** evaluate full system performance
  - Including app, OS, HW, and interactions thereof
- **Problem:** will stop short of prototyping HW
- **Solution:** evaluate in full-system simulation →

SimBricks

https://simbricks.github.io

(If interested, you can use this for many more complex & interesting system setups as well.)



References

- HW Acceleration
  - TPU Paper (TPUv4): Google's AI/ML accelerator (https://arxiv.org/pdf/1704.04760.pdf, https://dl.acm.org/doi/pdf/10.1145/3579371.3589350)
  - VTA Paper: Open-Source tensor accelerator (https://arxiv.org/pdf/1807.04188.pdf)
- Device I/O, PCIe:
  - https://personal.utdallas.edu/~sridhar/ios/ref/virt_book.pdf#section.6.2
- Drivers
  - VFIO documentation: https://www.kernel.org/doc/Documentation/vfio.txt
- Simulation:
  - SimBricks: https://simbricks.github.io
  - Gem5: https://www.gem5.org