Open source FPGA NVMe accelerator platform for BPF driven ML processing with Linux/Zephyr

Linux Plumbers 2022, Dublin, 2022-09-14

Karol Gugala, kgugala@antmicro.com

antmicro

ANTMICRO

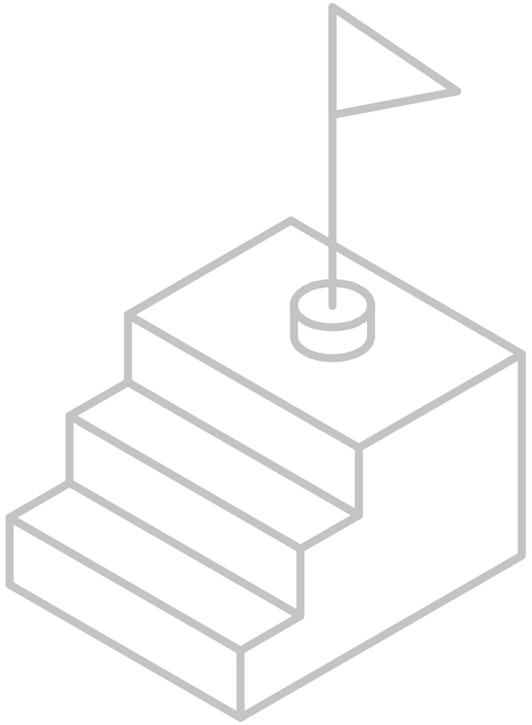
- Founded in 2009, Antmicro provides commercial open source engineering services, platforms and tools (SW, HW, FPGA, ASIC)
- Introducing new design methodologies and workflows based on open source
- Applying those methodologies and using Zephyr to build real products and development platforms - like the one we will describe today, an open source NVMe accelerator platform

WHAT IT IS ALL ABOUT

- The goal of the project is to provide a platform for research on computational storage
- Build an open source platform for NVMe accelerators development on a flexible FPGA SoC platform - Xilinx US+ MPSoC
- Create an open source NVMe FPGA core
- Prepare firmware that handles essential NVMe operations
- Expand initial NVMe implementation with custom accelerator-related extensions

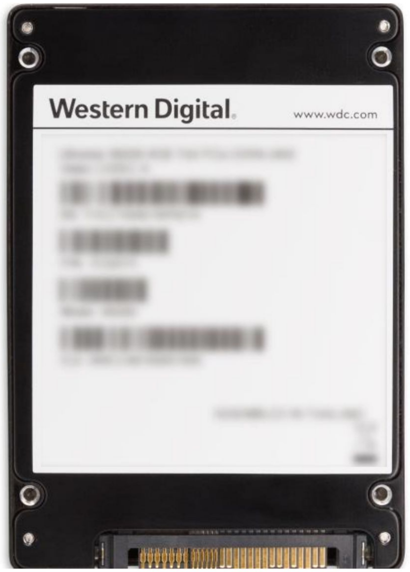
WHY DO WE NEED ACCELERATORS IN NVMe DRIVES?

- Machine Learning usually operates on large amounts of data
- Transferring data back and forth generates bottlenecks and costs
- NVMe accelerators reside close to stored data
- They allow us to process the data on the fly, or perform computation on already stored data, detect interesting patterns
- Data can be processed directly without consuming compute resources / spinning up machines

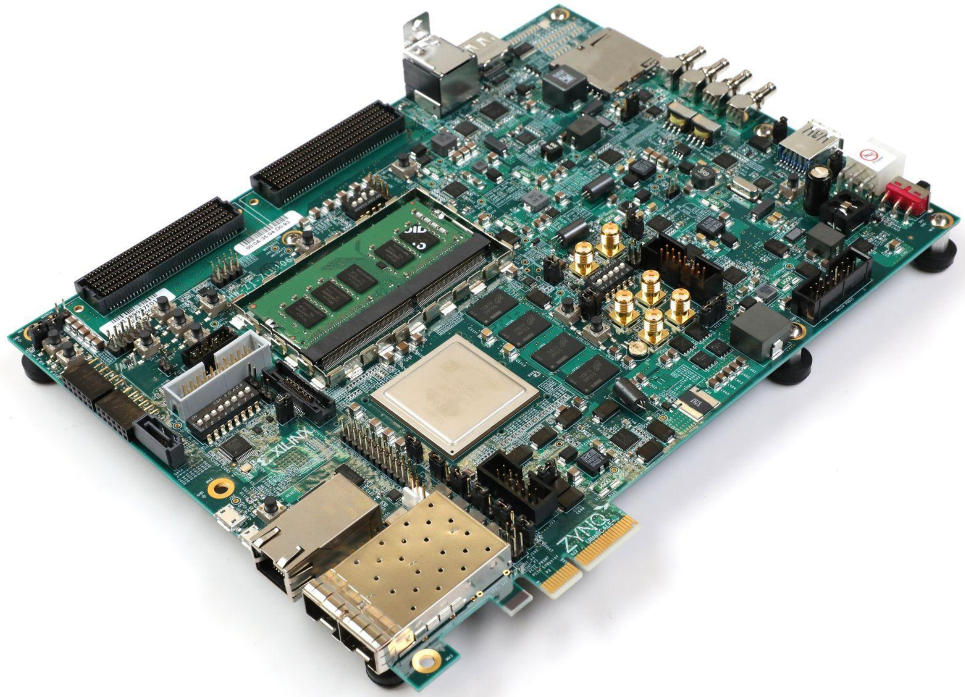


TARGET HW PLATFORM

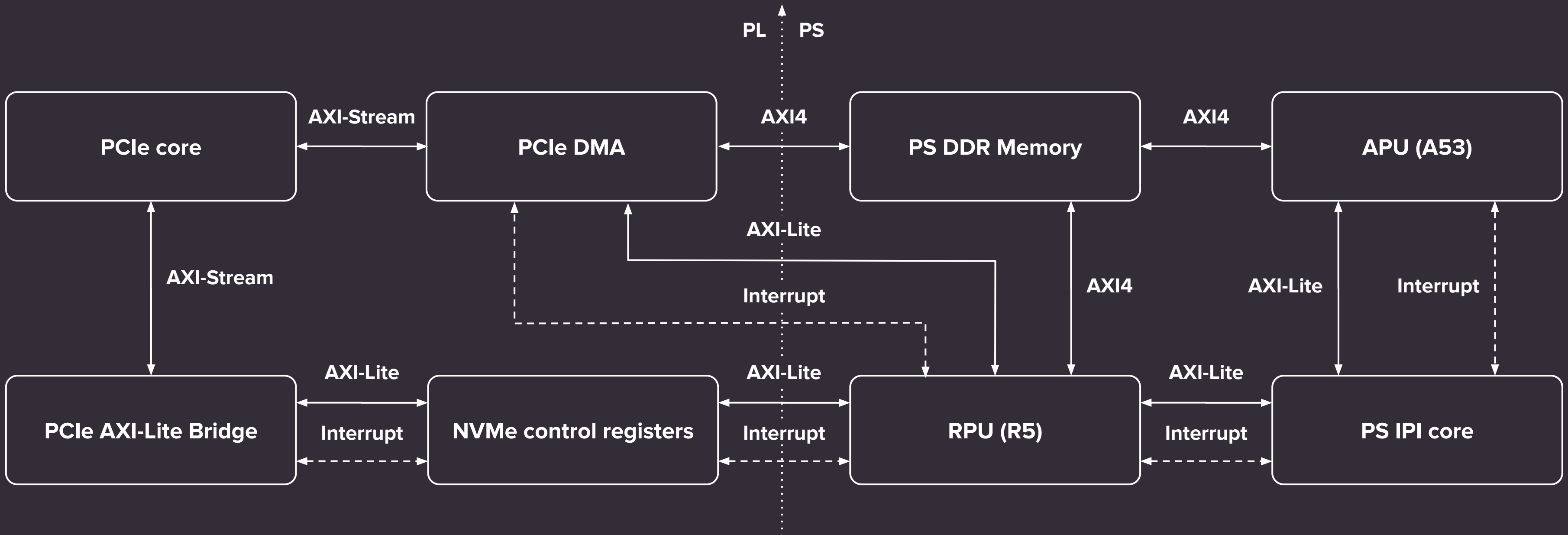
- FPGA Based PCIe ML/AI Accelerator Device in U.2 Formfactor
- Xilinx Ultrascale+ MPSoC XCZU7EV
- 4GB DDR
- Gen3 x4 PCIe 2.5” SFF
- 25W Max Power



DEVELOPMENT PLATFORM - ZCU106



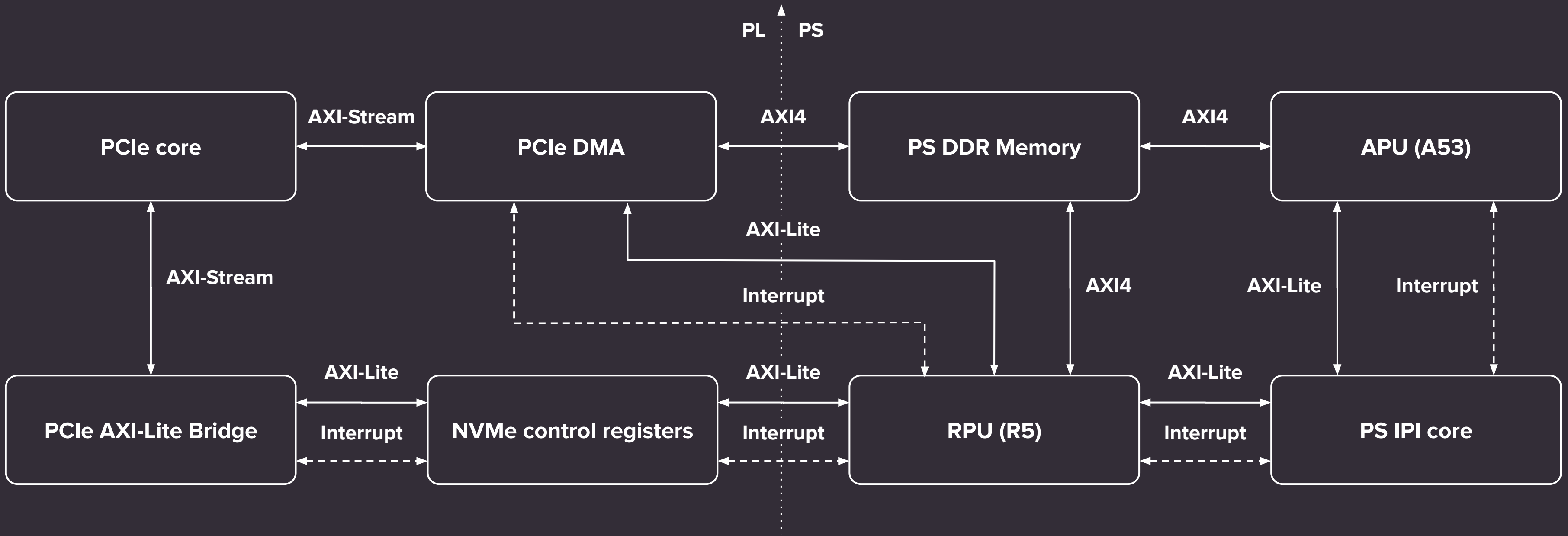
SYSTEM OVERVIEW



NVMe COMMANDS

- Admin commands:
  - Identify
  - GetLog
  - Queue management
- IO commands:
  - Read
  - Write
  - Flush

SYSTEM OVERVIEW

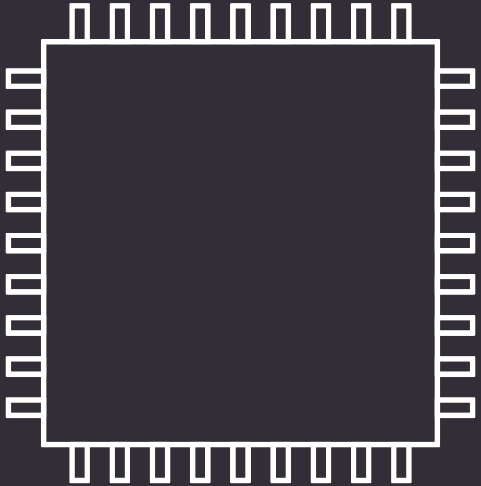


BASE NVMe COMMAND SET HANDLING

- FPGA NVMe subsystem is handled by software running on the Cortex R5 CPU complex
- The R5 software is a Zephyr app and it handles base NVMe commands
  - NVMe registers accesses from host generates interrupts handled by the software running on the R5 cores
  - The software handles commands queues, data transfers, control messages etc.
- All the “unknown” commands are passed to a Linux service for processing

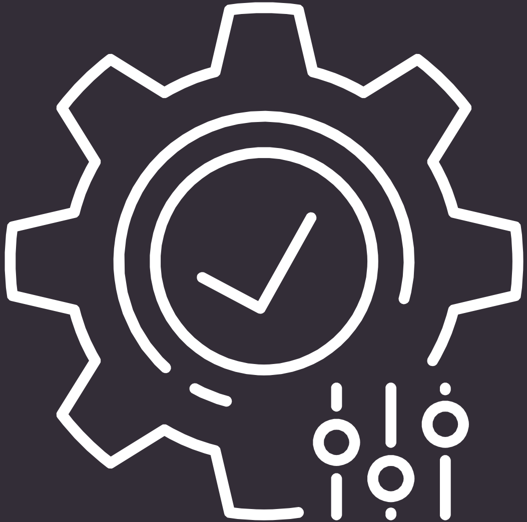
HOW TO HANDLE DYNAMIC FPGA LOGIC

- The dynamic nature of the FPGA logic makes it harder to maintain both parts (FPGA logic and software) and keep them in line
- Each change may impact both both sides of the system:
  - Adding e.g. new NVMe commands may reorganize the memory layout of the device
  - Commands queues length may change



GENERATE THE CODE ON THE FLY

- A solution to that is to generate both sides of the code on-the-fly
- The build system we introduced parses the NVMe 1.4 specification (a pdf file) and generates the following:
  - NVMe registers logic (Chisel code)
  - Zephyr app register definitions (header files)
  - Zephyr app register access logic (C code)
- The generator itself is available on GitHub



EXTRACTED REGISTER MAP IN CHISEL

```
//CSRRegMap.scala

// Generated on 13/09/2022 08:29:20 with NVM-Express-1_4-2019.06.10-Ratified.pdf,
git_sha 03aa526

package NVMeCore

import chisel3._

object CSRRegMap {
        val regMap = Map [Int, BaseRegister] (
                0x0 -> Module(new ReadOnlyRegister(new CAP_0, 32)),
                0x4 -> Module(new ReadOnlyRegister(new CAP_1, 32)),
                0x8 -> Module(new ReadOnlyRegister(new VS, 32)),
                 …
                0xe10 -> Module(new StorageRegister(new PMRSWTP, 32)),
                0xe14 -> Module(new StorageRegister(new PMRMSC_0, 32)),
                0xe18 -> Module(new StorageRegister(new PMRMSC_1, 32)),
        )
}
```

EXTRACTED REGISTER DEFINITION IN CHISEL

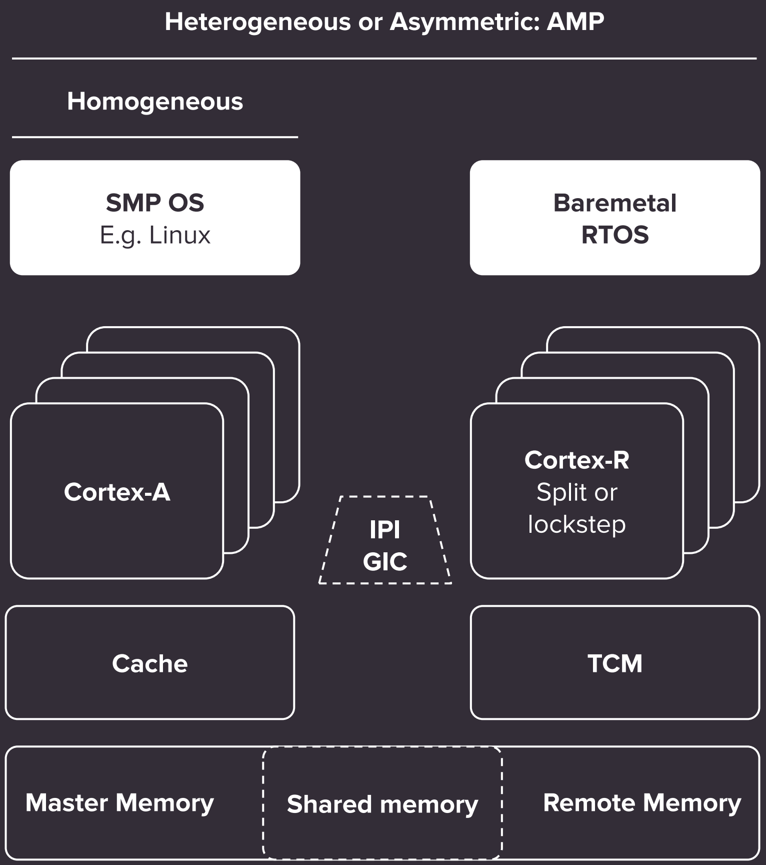
```
//RegisterDefs.scala

// Generated on 13/09/2022 08:29:19 with registers.json, git_sha 03aa526

package NVMeCore

import chisel3._

class CAP_0 extends RegisterDef {
        val TO = UInt(8.W)
        val Reserved_2 = UInt(5.W)
        val AMS = UInt(2.W)
        val CQR = Bool()
        val MQES = UInt(16.W)
}

…
```

NON-STANDARD NVMe COMMANDS

- As mentioned earlier, all the commands not known to Zephyr app are passed upwards for further processing to Linux running on the second, Cortex-A53 CPU complex
- This makes the platform easily extendable and perfect for experimenting with NVMe specification extensions

OPENAMP

- Framework for systems with asymmetric multiprocessing
- Provides easy method of communication between CPUs in AMP system
- RPU side runs Zephyr and is controlled from Linux application (using openAMP)
- Linux application implements openAMP communication and interfaces NVMe blocks with eBPF virtual machine



ACCESSING THE ACCELERATOR

- APU runs a service fetching the all the unhandled NVMe commands and checks if they are accelerator-specific - it communicates via rpmsg
- The custom command are used to control various aspects of the system:
  - loading the firmware for the accelerator, as well as ML model and inputs
  - Controlling the accelerator flow (resetting, starting/stopping)

ACCELERATOR FIRMWARE

- https://github.com/iovisor/ubpf
- Accelerator firmware is in fact an eBPF bytecode
- It is executed inside BPF virtual machine running in Linux user space
- The firmware can be generated from C source using LLVM

MACHINE LEARNING RUNTIME ON APU

- The uBPF virtual machine was extended with functions for running the machine learning models with given inputs
- The runtime used for the ML models is TensorFlow Lite
- TensorFlow Lite has a native implementation for most of the available ML operations and can run models directly on APU
- TensorFlow Lite also provides a delegation mechanism, allowing the developers to move computations of certain operations to the dedicated accelerator hardware

TensorFlow Lite

eBPF

SPECIFICATION OF NEW OPERATIONS IN BPF (VM CODE)

```c
void vm_tflite_apu (char *ibuf, char *obuf, int isize, int osize, int model_size)
{
    char *model_buf = ibuf;
    char *input_buf = ibuf+model_size;

    tflite_handler(model_buf, input_buf, obuf, model_size, isize, osize,   false);
}

void vm_tflite_vta (char *ibuf, char *obuf, int isize, int osize, int model_size)
{
    char *model_buf = ibuf;
    char *input_buf = ibuf+model_size;

    tflite_handler(model_buf, input_buf, obuf, model_size, isize, osize,   true);
}

...

void register_functions (struct ubpf_vm *vm)
{
        ubpf_register(vm,  1, "print", (void*)vm_print);
        ubpf_register(vm,  2, "tflite_apu" , (void*)vm_tflite_apu);
        ubpf_register(vm,  3, "tflite_vta" , (void*)vm_tflite_vta);
}
```

EXAMPLE ACCELERATOR FIRMWARE

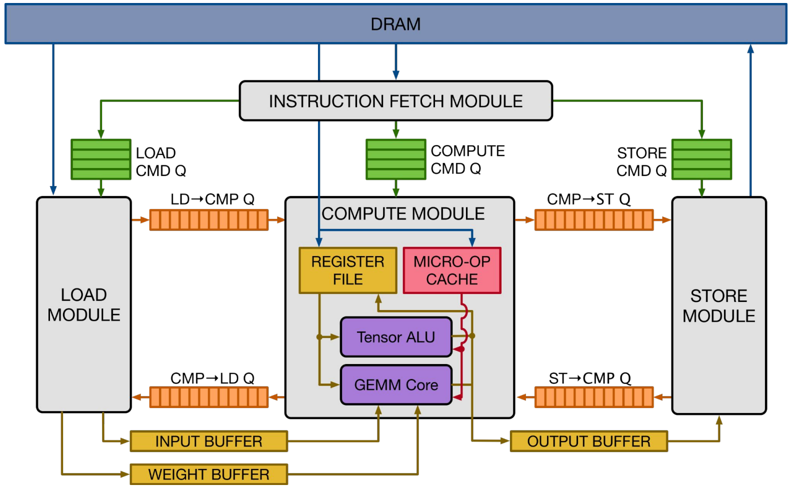
```c
static void (*print)(char*) = (void *)1;
static void (*tflite_vta)(char*, char*, int, int, int) = (void *)3;

int bpf_prog(char *imem, char *omem)
{
        const int model_size = 1024;
        const int input_size = 8;
        const int output_size = 4;
        char expected_output[] = {0x7, 0xC, 0x4, 0x5};

        tflite_vta(imem, omem, input_size, output_size, model_size);

        for (int i = 0; i < output_size; i++) {
                if (omem[i] != expected_output[i]) {
                        print("ADD test failed\n");
                        return i;
                }
        }

        print("ADD test passed\n");
        return -1;
}
```

SUPPORT FOR EXTERN FUNCTIONS IN FIRMWARE (COMING SOON)

```
extern void (*print)(char*);
extern void (*tflite_vta)(char*, char*, int, int, int);

int bpf_prog(char *imem, char *omem)
{
        const int model_size = 1024;
        const int input_size = 8;
        const int output_size = 4;
        char expected_output[] = {0x7, 0xC, 0x4, 0x5};

        tflite_vta(imem, omem, input_size, output_size, model_size);

        for (int i = 0; i < output_size; i++) {
                if (omem[i] != expected_output[i]) {
                        print("ADD test failed\n");
                        return i;
                }
        }

        print("ADD test passed\n");
        return -1;
}
```

ACCELERATOR DESIGN - VTA

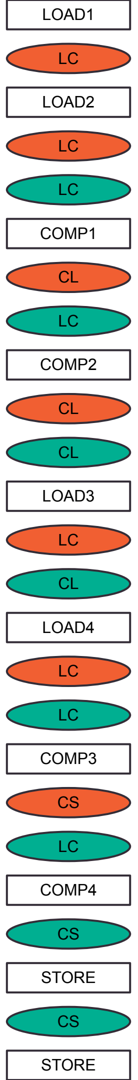
- github.com/apache/tvm-vta
- VTA (Versatile Tensor Accelerator)
  - Programmable and customizable accelerator IP core
  - Written in Chisel
  - Part of the Apache TVM framework
- Consists of three main modules - LOAD module, COMPUTE module and STORE module
- Each module has its specific instruction queue
- Order of execution is determined by dependency queues
- TensorFlow Lite executed within eBPF virtual machine can delegate certain operations to the VTA, utilizing its high parallelism

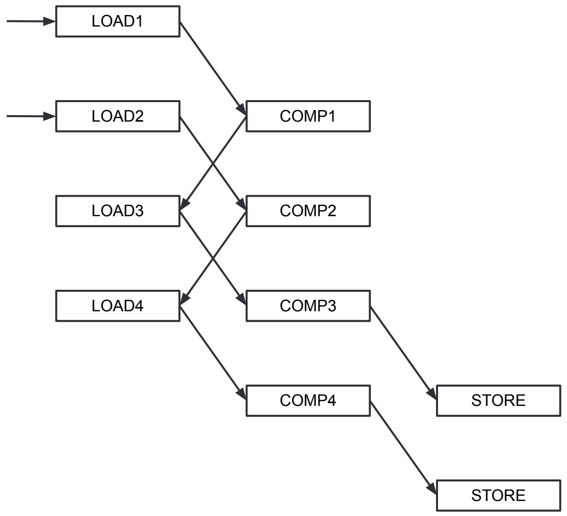


https://tvm.apache.org/docs/topic/vta/dev/hardware.html

VTA DELEGATE IN TENSORFLOW LITE

- TensorFlow Lite delegate prepares the instruction stream that is passed asynchronously to the VTA
- Instruction stream consists of:
  - VTA LOAD instructions (VTALoadBuffer2D)
  - VTA STORE instructions (VTAStoreBuffer2D)
  - VTA micro-op kernel (VTAUop), consisting of:
    - Outer and inner loop for GEMM/ALU operations
    - GEMM/ALU operations (one or many), deployed on various VTA SRAM addresses
  - Dependency pushes and pops for 4 synchronization queues (LOAD->COMPUTE, COMPUTE->LOAD, COMPUTE->STORE, STORE->COMPUTE)
- Dependency queues allow modules to work independently, allowing to hide I/O latency
- Once VTA finishes the instructions, APU collects results





HOW TO GET ON WITH THE PROJECT

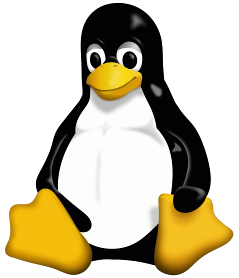
- All the code is available on GitHub
  https://github.com/antmicro/alkali-csd-projects
- It will be donated to CHIPS Alliance soon

PROJECT STRUCTURE

- Automated build system for building the project
  - https://github.com/antmicro/alkali-csd-projects
  - Contains examples for the supported boards
  - Uses Alkali firmware and hardware submodules
- Alkali hardware (FPGA) repository
  - https://github.com/antmicro/alkali-csd-hw
  - Generates hardware description file and bitstream
- Alkali firmware repository
  - https://github.com/antmicro/alkali-csd-fw
  - Generates APU and RPU applications, U-Boot, Linux and rootfs

SUMMING IT UP

- Open source NVMe development platform is a perfect framework for research and development of computational storage concepts
- Beside the complete platform, there is a number of useful blocks developed within the project that can be reused to build an NVMe device
- Contributions are welcome!



antmicro

THANK YOU
FOR YOUR ATTENTION!