Alkali CSD NVMe accelerators test platform

Antmicro

2023-02-02

CONTENTS

CHAPTER

ONE

INTRODUCTION

This document describes the Western Digital NVMe accelerators test platform. The main goal of this project is to develop a proof of concept of an open source NVMe accelerator platform. Initial work will be done using Xilinx ZCU106 platform and then it will be continued on the Basalt platform provided by Western Digital.

The document is divided into the following chapters:

- *Repository reference* lists the repositories used for this project.
- *System architecture* contains information about the project's architecture, such as *FPGA design*, *Host Software*, *APU Software* and *RPU Software*
- *NVMe commands and extensions* describes in detail the extended NVMe command set developed as part of this project.
- *TensorFlow Lite model preparation* describes the preparation and usage of TensorFlow Lite models.
- *VTA accelerator* describes the VTA accelerator usage details.
- *Operations accelerated on VTA accelerator* describes the implemented operations with the VTA delegate.
- *Flashing and connecting the Basalt board* describes how to flash firmware to QSPI using NVMe commands.

CHAPTER TWO

REPOSITORY REFERENCE

Multiple repositories are used to store this project. This short overview aims to clarify purpose of each repository:

- alkali-csd-projects is a core repository for building hardware and software for Western Digital NVMe accelerator test platform.
- alkali-csd-hw contains FPGA design sources for the Alkali project, used for generating bitstream for Western Digital NVMe accelerator test platform.
- alkali-csd-fw contains drivers, Linux (based on Buildroot) for APU and Zephyr application for RPU, as well as delegate software and test scripts and software for the accelerator.

CHAPTER

THREE

SYSTEM ARCHITECTURE

This chapter provides information about the architecture of the project.

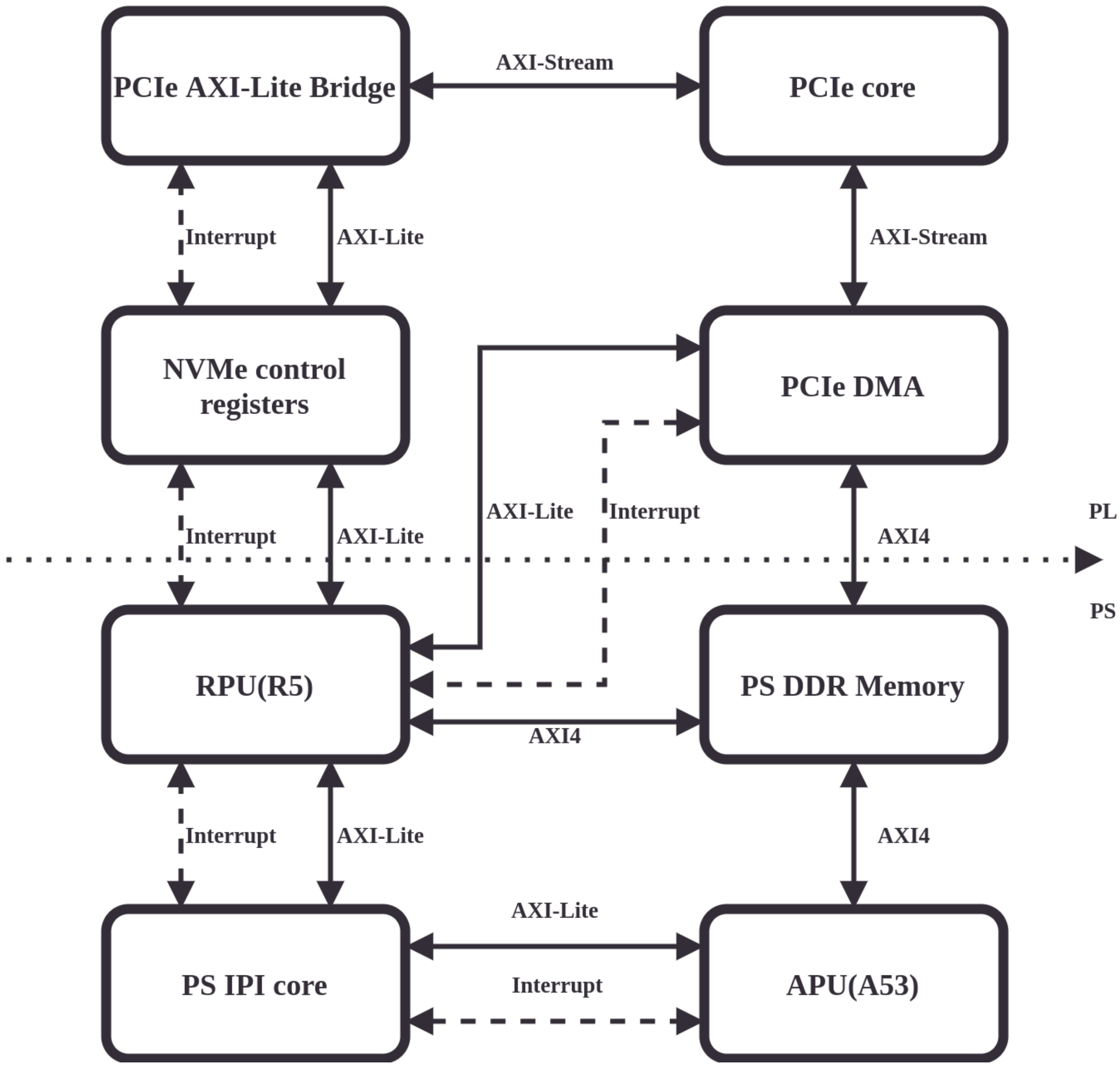
3.1 FPGA design

This chapter describes the FPGA design that will be used on the target platform. It will utilize Zynq US+ MPSoC's PCIe hard block and additional supporting logic to handle NVMe physical layer and expose the NVMe configuration registers and DMAs to the RPU.

A diagram that presents the connections between various elements of the design is shown below:

There are four major parts that can be distinguished:

- PCIe core, which will handle the basic PCIe protocol and will provide:
  - Host access to NVMe control registers using BAR space
  - Access to Host memory for PCIe DMA.
  - Support for generating Host interrupts.
- The RPU core, NVMe control reigsters and PCIe DMA that will be responsible for handling the base NVMe protocol:
  - RPU will implement NVMe handling logic.
  - NVMe control registers will provide register space required by the NVMe specification and will generate required interrupts, e.g. on writes to `Doorbell` registers).
  - PCIe DMA will be responsible for transferring data to and from the Host memory.
- The APU core and PS IPI core that will be responsible for handling custom NVMe commands:
  - The IPI core will be used to communicate between RPU and APU.
  - APU will implement logic that handles custom NVMe commands.
- Shared PS DDR memory will be used by both APU and RPU cores and will be used as buffer before transferring NVMe data to/from the Host.


PCIe AXI-Lite Bridge
AXI-Stream
PCIe core
Interrupt
AXI-Lite
AXI-Stream
NVMe control registers
PCIe DMA
AXI-Lite
Interrupt
Interrupt
AXI-Lite
AXI4
PL
PS
RPU(R5)
AXI4
PS DDR Memory
Interrupt
AXI-Lite
AXI4
AXI-Lite
Interrupt
PS IPI core
APU(A53)

3.1.1 Building the design

Instructions for building the design (and the whole project) are present in the alkali-csd-projects README and alkali-csd-hw README.

Note: Building the design will require Vivado 2019.2 to be installed on the development PC.

3.1.2 NVMe register module

The NVMe register module is an IP core written in Chisel that implements Controller Registers described in the NVMe Specification.

The register map follows the register layout from the specification and by default contains 5 pairs of doorbell registers.

On top of implementing basic Controller Registers, this module contains additional logic that is used by the firmware running on the RPU for:

- Generating RPU interrupts on Host writes

  On each Host write, register address is saved to Host Write Address FIFO which can later be accessed by the RPU to see which registers have been written and require action.

- Generating Host interrupts

  RPU can trigger one of the 32 MSI interrupts by writing to a control register with correct bits set, e.g. writing a value with bit 31 set sends interrupt 31.

To be able to use those additional features, the core contains three additional registers:

- Interrupt Status Register - by default located at `0x1028` which returns `1` when there is at least one entry in Host Write Address FIFO.
- Interrupt Data Register - located at `0x102c` returns oldest entry from Host Write Address FIFO when read.
- Host Interrupt Register - located at `0x1030` triggers Host interrupt when written.

Generating NVMe IP

NVMe IP is written in Chisel which means that synthesizable Verilog must be first generated from sources. The sources for generating NVMe IP is present in alkali-csd-hw/tree/main/chisel.

It is built within alkali-csd-hw build flow, described in its README.

To just generate the NVMe IP, the following dependencies are required:

- Scala (2.13.0)
- SBT
- Java

Note: A Dockerfile for building HW artifacts is a recommended environment for generating the NVMe IP.

Once the dependencies are present, run:

```
git clone https://github.com/antmicro/alkali-csd-hw.git
cd chisel
make verilog
```

Updating register definitions

You can update the register definitions using nvme-registers-generator.

Those generator scripts take the NVMe Specification `.pdf` file as an input and based on that generate register definitions that can be used in Chisel.

For details on generating register definitions check the nvme-registers-generator README file in the repository.

Adding new registers

NVMe IP can be expanded with additional registers, either unimplemented optional ones or vendor specific.

To add a new register, you can use the following steps:

1. Check if your register layout is defined in `RegisterDefs.scala` (this only applies if you want do add an unimplemented register). Take a look into `CSRRegMap.scala` if your register was already defined - it might be already present as a simple `StorageRegister`.

2. Add new register layout definition to `CSR.scala` if needed.

   `IRQHOST` can be used as an example:

   ```
   class IRQHOST extends RegisterDef {
     val REQ = UInt(32.W)
   }
   ```

3. Instantiate new register as a module in `CSRFile.scala`, make all needed connections and add it to `regMap`.

   `IRQHOST` can be used as an example:

   ```
   val irqHost = Module(new AutoClearingRegister(new IRQHOST, dataWidth))

   println(f"adding Host interrupt register at 0x${irqHostBase}%x")
   regMap(irqHostBase) = irqHost

   io.irqHost := irqHost.fields.REQ
   ```

3.1.3 VTA module

The VTA is an IP core written in HLS that is used by TFLite to accelerate certain operations. You can find more information about using it in *VTA accelerator*.

To build the VTA and whole HW design, follow the README in alkali-csd-hw.

3.1.4 Built artifacts

The ready-to-use binaries for the hardware are available under alkali-csd-hw releases.

3.2 Memory map

Multiple memory areas are used in the system, this includes both register areas for IP cores and shared memory:

Base	Size	Name
0x6000_0000	128 MiB	RPU FW area
0x6800_0000	384 MiB	NVMe ramdisk
0xA000_0000	64 KiB	PCIe DMA IP
0xA001_0000	64 KiB	NVMe IP
0xB000_0000	4 KiB	VTA Fetch IP
0xB000_1000	4 KiB	VTA Load IP
0xB000_2000	4 KiB	VTA Compute IP
0xB000_3000	4 KiB	VTA Store IP

3.2.1 PCIe and NVMe Cores

Location of PCIe and NVMe cores in memory map is set in Vivado design located in alkali-csd-hw/tree/main/vivado. After making changes to their addresses you need to adjust nvme.overlay for RPU firmware to contain correct base addresses.

3.2.2 VTA Cores

Location of VTA cores in memory map is set in Vivado design located in alkali-csd-hw/tree/main/vivado. After making changes to their addresses you need to adjust vta_params.hpp file used by the `apu-app` to contain correct base addresses.

3.2.3 RPU-APU shared memory

The main memory is shared between Linux running on the APU and Zephyr RTOS app running on the RPU. To ensure that the two don't interfere with each other a memory range dedicated to the RPU needs to be defined. It can then be used in Linux to reserve that part of the RAM and in Zephyr to limit size of the application and its buffers. In both cases this is defined via the devicetree.

For RPU it is defined in nvme.overlay as `sram0` which represents area for the firmware. In APU case it is declared in:

- an300 - `zynqmp-an300-nvme.dts` added in alkali-csd-fw/blob/main/br2-external/common/patches/linux/0012-dts-add-an300-support.patch
- zcu106 - `zynqmp-zcu106-nvme.dts` added in lkali-csd-fw/blob/main/br2-external/common/patches/linux/0003-dts-add-separate-devicetree-for-NVMe-ZCU106.patch

as `reserved-memory`.

3.2.4 Ramdisk area

Ramdisk is also located in the main memory and is shared between Linux on the APU and Zephyr on the RPU. For RPU it is defined in nvme.overlay as `sram1`. In APU case it is declared in `zynqmp-basalt-nvme.dts` as part of `reserved-memory` and you need to adjust alkali-csd-fw/blob/main/apu-app/src/lba.h in `apu-app` when changing ramdisk location in Zephyr. To access a particular page you need to first calculate it's offset with `(lba * RAMDISK_PAGE) + RAMDISK_BASE` and then either use that address directly (in case of RPU) or MMAP it (on APU).

Note: Accessing ramdisk area from the BPF code is achieved with help of `Use local storage as accelerator input` and `Use local storage as accelerator output` commands. Those commands take LBA value to calculate correct offset, MMAP it and then pass it as a pointer to your main BPF function.

3.3 Host Software

This chapter describes the software that will be used on the host PC. The host software will communicate with the target platform using the NVMe interface.

It is located in alkali-csd-projects/tree/main/host-app.

3.3.1 Building the app

The host application is a simple C program that uses reads, writes and ioctls to communicate with the accelerator. To build it, run:

```
git clone https://github.com/antmicro/alkali-csd-projects.git
cd alkali-csd-projects
make host-app
```

To build the host application along with additional files needed to run an example, run:

```
EXAMPLE=add make example/build
```

For more details check Running examples section of alkali-csd-projects README.

3.3.2 Using the app

The easiest way to use the application is to utilize the wrapper script which is located in the alkali-csd-projects/tree/main/host-app directory:

```
cd host-app
./run.sh <path to NVMe device> <path to BPF source file> <input file> <output␣
↪file>
```

The example of ADD operation can be executed with `make` in the root `alkali-csd-projects`:

```
EXAMPLE=add NVME_DEVICE=/dev/<nvmedevice> make example/load
```

Where:

- `/dev/<nvmedevice>` is the path to the NVMe accelerator,
- `EXAMPLE` is the example available under alkali-csd-projects/tree/main/examples/tflite_vta/add
- `example/load` is a target building the example and running the `run.sh` script to send data to process to the accelerator.

The results will be stored in `build/examples/add/output.bin`.

The program that is running in the accelerator is alkali-csd-projects/blob/main/examples/tflite_vta/add/bpf.c. It is a simple C file that contains the BPF program that will be built using *clang*.

It runs a sample alkali-csd-projects/blob/main/examples/tflite_vta/add/model.tflite TFLite model on VTA accelerator on input specified in the alkali-csd-projects/blob/main/examples/tflite_vta/add/input-vector.bin.

3.4 APU Software

This chapter describes the software that will be running on the APU (A53 cores) part of the Zynq US+ MPSoC. The APU software is responsible for processing the custom NVMe commands.

The firmware is available under alkali-csd-fw.

3.4.1 Building the APU software

To build the system and necessary software, follow alkali-csd-fw README.

3.4.2 APU base system

Linux Kernel linux-xlnx is used as the operating system for the APU. Buildroot is used to build all dependencies and utilities for the APU and creates rootfs.

The rootfs contains basic set of system utilities, APU and RPU applications.

3.4.3 uBPF Virtual Machine

uBPF Virtual Machine a user space software allowing execution of a BPF programs. the uBPF library is integrated with the APU application and is used to execute BPF payloads sent from the host. The capabilities of BPF programs can be easily extended by adding external functions that can be called from the BPF binary. Such functions are implemented in alkali-csd-fw/tree/main/apu-app/src/vm.

Currently, the BPF programs allow to delegate inference of TFLite models to the VTA delegate.

The example of such program is ADD runner in alkali-csd-projects project.

3.4.4 Userspace custom NVMe command handler

Userspace application is used to handle custom Accelerator-related NVMe commands. Communication with firmware running on the RPU is achieved by using rpmsg. All vendor specific commands detected by the firmware are passed through to this application.

The application is available in alkali-csd-fw/tree/main/apu-app.

Adding support for new NVMe commands

Adding support for additional commands is fairly simple. Commands are dispatched by `handle_adm_cmd` and `handle_io_cmd` functions located in alkali-csd-fw/blob/main/apu-app/src/cmd.cpp. To handle another command, simply expand the `switch` responsible for calling handlers by calling your new handler and then `send_ack` with proper ACK type (`PAYLOAD_ACK_DATA` when command returns data, `PAYLOAD_ACK` otherwise). Buffer with the command will be provided using `recv` and `mmap_buf` represents buffer for data transferred to or from host.

3.5 RPU Software

This chapter describes the software that will be running on the RPU (R5 cores) part of the Zynq US+ MPSoC. The RPU software will be responsible for handling the functionality required by the NVMe standard including regular read/write transactions.

3.5.1 Operating system

The RPU software uses Zephyr RTOS as the operating system.

3.5.2 NVMe firmware overview

The NVMe app runs from a reserved part of the APU DDR memory. The exact location of this area is specified in *RPU-APU shared memory*. This space is used for the app itself as well as for various buffers needed to process the NVMe commands. The app also uses a small chunk (60B) of memory at `0x0` to store reset and exception vectors, but that memory range is mapped into the TCM memory.

The debug output from RPU is provided on serial port 0.

The app contains custom drivers for the two peripherals implemented in the PL:

- Verilog-PCIe DMA core
- Custom NVMe register module, described in *NVMe register module*

Both of these peripherals generate interrupts when RPU attention is needed. The NVMe register module generates an interrupt for each Host register write, and the DMA generates interrupts after finishing a transfer.

At the moment the app supports a minimal set of Admin commands sent by the NVMe Linux driver:

- Obtaining Identify Controller/Namespace structure with `Identify`
- Obtaining SMART data structure using `Get Log Page`
- Manipulating I/O Submission/Completion Queues with `Create/Delete I/O Completion/Submission Queue`
- Configuring the amount of queues using `Set Features`

I/O commands are also supported but for the moment support is minimal - all commands are marked as successful in theirs Completions.

This level of supported commands allows the drive to successfully register in the system and nvme-cli can be used to perform basic operations, e.g. identifying the drive, dumping SMART data. More details on that can be found in *Basic NVMe operations*.

3.5.3 Building and running NVMe firmware

For building instructions follow alkali-csd-fw README.

Note: It is recommended to build the whole project by following alkali-csd-projects README.

3.5.4 Adding support for new NVMe commands

Adding support for additional commands is fairly simple. Commands are dispatched by `handle_adm` and `handle_io` functions located in alkali-csd-fw/blob/main/rpu-app/src/cmd.c. To handle a new one you simply need to add another entry in the `switch` block which calls handler for your command. For example, this is how a handler for `Write` command is called:

```
case NVME_IO_CMD_WRITE:
    nvme_cmd_io_write(priv);
    break;
```

Note: Commands can be also passed to APU for processing which is the case for FW update commands and vendor commands. You can get more information about handling them on the APU side in *Adding support for new NVMe commands*.

Once in handler, you will have access to a buffer with your command provided via `priv` variable. You can use that to retrieve all needed command fields just like in this example:

```
typedef struct cmd_cdw10 {
        uint32_t fid : 8;
        uint32_t rsvd : 23;
        uint32_t sv : 1;
} cmd_cdw10_t;

typedef struct cmd_cdw14 {
        uint32_t uuid_idx : 7;
        uint32_t rsvd : 25;
} cmd_cdw14_t;

typedef struct cmd_sq {
        nvme_sq_entry_base_t base;
        cmd_cdw10_t cdw10;
        uint32_t cdw[3];
        cmd_cdw14_t cdw14;
} cmd_sq_t;

void nvme_cmd_adm_set_features(nvme_cmd_priv_t *priv)
{
        cmd_sq_t *cmd = (cmd_sq_t*)priv->sq_buf;

        switch(cmd->cdw10.fid) {
```

(continues on next page)

(continued from previous page)

```
                case FID_NUMBER_OF_QUEUES:
                        number_of_queues(priv);
                        break;
                default:
                        printk("Invalid Set Features FID value! (%d)\n", cmd->
↪cdw10.fid);
        }

        nvme_cmd_return(priv);
}
```

This snippet also shows another important point - all handlers must include `nvme_cmd_return` or `nvme_cmd_return_data`. Without that no NVMe response will be sent and the command will timeout.

NVME COMMANDS AND EXTENSIONS

This chapter discusses a proposal of the NVMe protocol with vendor-specific commands. Custom commands are extending both Admin and I/O commands sets. All the additional commands are encoded in the vendor-specific commands address space.

4.1 Basic NVMe operations

To perform basic NVMe operations once the device gets probed successfully, you can use nvme-cli. This tool allows you to send various commands to NVMe devices.

Currently supported commands are:

- *list*
- *list-subsys*
- *id-ctrl*
- *id-ns*
- *list-ns*
- *smart-log*
- *set-feature*
- *read*
- *write*
- *fw-download*
- *fw-commit*

4.2 NVMe Vendor extensions

In order to use the accelerator you need to be able to control it over NVMe interface. The control functionality can be implemented with additional, vendor-specific commands. This way the accelerator device will be compatible with generic NVMe software. Custom software will be required to use the additional accelerator functionalities. Some of the commands defined in the NVMe standard support vendor extensions which can be used to implement basic features, e.g. retrieving accelerator logs.

4.2.1 Get Log Page (Log Page Identifier 0xC0)

The get log page command returns a data buffer containing the log page for the specified accelerator. The command returns a variable length buffer containing a list of status descriptors.

Accelerators are identified using numeric ID values. Accelerator ID is provided in the `Log Specific Identifier` field. Bits 15:0 of the `Log Specific Identifier` field contain Accelerator ID.

Accelerator logs are packed into Log Entry descriptors. The tables below contain the descriptors' structure:

Table 4.1: Get Accelerator Log Page data structure

Bytes	Description
0:3	Log page length (in bytes)
4:15	reserved for future use
16:N	Accelerator log entry descriptor list

Table 4.2: Accelerator Log Entry descriptor

Bytes	Description
0:3	Descriptor length (in bytes)
4:11	Timestamp
12:15	Entry type ID
16:N	Accelerator-specific information (optional)

The data inside the optional information block can be used to provide more information for the log entry, e.g. error message string.

Entry types are identified with unique IDs. The exact ID list is to be defined. Below is an example list:

Table 4.3: Entry type IDs

ID	Descrption
0	Invalid firmware ID was selected
1	Invalid input buffer configuration
2	Invalid output buffer configuration
3	Accelerator specific error

4.3 Custom NVMe commands

Controlling accelerator-related features will require a set of custom commands on top of what NVMe provides. The NVMe standard supports defining vendor-specific commands that use a separate range of opcodes. The following sections lists custom (vendor-specific) commands extending the Admin and I/O sets.

4.3.1 Admin command set extension

Custom admin commands will be used to obtain information about the device, status of the accelerators and will enable basic accelerator control.

The `DPTR` field of an NVMe command frame will be used to specify buffer location for commands that transfer data to or from the device.

Accelerator Identify (0xC2)

The identify command returns a data buffer that describes information about the custom accelerators available in the device. The command may also be used to determine if the connected device is an NVMe accelerator device. The data structure has a variable length. The length is determined by reading the first 8 bytes (confirming that the first 4 bytes hold the magic value). Once the length is known, the whole buffer can be retrieved.

Accelerators are described using descriptors. The tables below depict data structures used by the Accelerator Identify command.

Table 4.4: Accelerators Idenfity descriptor structure

Bytes	Description
0:3	Magic value (“WDC0”)
4:7	Descriptor length (in bytes)
8:15	Reserved for future use
16:N	Accelerator descriptor list

Table 4.5: Accelerators descriptor list entry

Bytes	Descritpion
0:3	Accelerator descriptor length (in bytes)
4:5	Accelerator ID (unique within the device)
6:7	Reserved for future use
8:N	Accelerator capabilities list

Table 4.6: Accelerator capabilities list entry

Bytes	Description
0:3	Capability ID
4:15	Capability specific data

Capabilities are identified with unique IDs. The exact ID list is to be defined. Below is an example list:

Table 4.7: Capabilites IDs

ID	Descrption
0	Accelerator supports firmware exchange
1	Accelerator supports input buffer of size defined in bytes 8:15 of the capability descriptor
2	Accelerator supports output buffer of size defined in bytes 8:15 of the capability descriptor

Get Accelerator Status (0xC6)

The Get Accelerator Status command is used to retrieve information about the current status of the selected Accelerator available in the system. The command returns a variable length buffer containing a list of status descriptors.

Accelerators are identified using numeric ID values. Accelerator ID will be provided in the `CDW12` field. Bits 15:0 of the `CDW12` field contain Accelerator ID. Bit 31 of the `CDW12` is the Retain Asynchronous Event (`RAE`) flag - when set to true, status information will not be modified until accessed with bit set as false. This mechanism allows the host to read the header of the status data buffer, determine the length of the whole transaction and finally read the whole buffer.

Accelerators statuses are packed into Status descriptors. Statuses are tied to accelerators capabilities, e.g. buffers can report how much data was processed. The tables below summarize the descriptors' structure:

Table 4.8: Get Accelerator Status data structure

Bytes	Description
0:3	Descriptor length
4:7	Status ID
8:31	reserved for future use
32:N	Accelerators status descriptors list

Table 4.9: Accelerator status descriptor

Bytes	Description
0:3	Capability ID
4:31	Status specific data

Global Accelerator Control (0xC0)

This Global Accelerator Control command is used to enable/disable accelerator subsystem.

`CDW12` field contains operation ID.

Operation identifier will take one of the specified values:

- `0x00` - Enable accelerator subsystem
- `0x01` - Disable accelerator subsystem

4.3.2 I/O commands

I/O commands are used to transfer data to and from the accelerators.

Bits 15:0 of the `CDW12` field contain Accelerator ID.

Send data to accelerator (0x81)

This command is used to fill accelerator input buffer with data from host memory.

`CDW10` field contains the number of dwords to transfer.

Read data from accelerator (0x82)

This command is used to copy the accelerator output buffer to host memory.

`CDW10` field contains the number of dwords to transfer.

Send Firmware to accelerator (0x85)

This command is used to upload firmware from the host memory to the selected accelerator firmware buffer.

`CDW10` field contains the number of dwords to transfer. `CDW13` field contains Firmware ID.

Read Firmware from accelerator (0x86)

This command is used to download firmware with selected ID from accelerator firmware buffer to host.

`CDW10` field contains number of dwords to transfer. `CDW13` field contains Firmware ID.

Use local storage as accelerator input (0x88)

This command is used to fill accelerator input buffer with data from local storage.

This command reuse `CDW10` to `CDW13` field layout from standard `Read` command relocated to `CDW12` to `CDW15`. `CDW14` and `CDW15` from original `Read` command will not be used.

Use local storage as accelerator output (0x8c)

This command is used to copy accelerator output buffer to local storage.

This command reuse `CDW10` to `CDW13` field layout from standard `Write` command relocated to `CDW12` to `CDW15`. `CDW14` and `CDW15` from original `Write` command will not be used.

Basic Accelerator Control (0x91)

This Basic Accelerator Control command is used to control the selected accelerator.

The `CDW13` field will hold the operation identifier.

Operation identifier will take one of the specified values:

- `0x00` - Reset accelerator - revert the accelerator to a blank stopped state.
- `0x01` - Start accelerator - verify the accelerator configuration (i.e. data buffers, eBPF app) and start processing.

- `0x02` - Stop accelerator - stop processing without modifying the configuration.
- `0x03` - Set active firmware - select the firmware which will be

More operations are to be defined.

The result of a certain operation can be retrieved with the `Get Accelerator Status` command.

`Start accelerator` operation uses additional fields.

`DPTR` field contains location of Argument List in host memory. `CDW10` field contains Argument List length in dwords. `CDW14` field contains Firmware ID.

Argument List will contain 0 or more concatenated entries. The table below depicts Argument List entry.

Table 4.10: Argument List entry

Bytes	Description
0:3	Argument entry length in bytes
4:N	Argument entry value

4.4 Example command flow

An example command flow that utilizes NVMe command set extensions is shown below.

1. Get basic information about the system:
    1. Send the `Accelerator Identify` command with the length set to 8 bytes.
        1. Verify that the first 4 bytes of the response match the magic value.
        2. Use the remaining 4 bytes as the Identify structure length.
    2. Send the `Accelerator Identify` command again using the length obtained earlier.
        1. Process the list of accelerators.
        2. Process the list of capabilities for each accelerator.
2. Enable accelerator subsystem
    1. Send the `Global Accelerator Control` command with the operation ID set to `Enable accelerator subsystem`.
3. Load firmware to the accelerator (applicable only to accelerators supporting firmware reloading)
    1. Check accelerator capabilities to verify that firmware loading is supported. The host software should cache the capabilities, so that it does not have to read it each time.
    2. Send the `Send Firmware to accelerator` command.
        1. Use the `accelerator ID` field to select which accelerator will receive the firmware.
        2. Set the `firmware ID` to select firmware slot.
4. Send data to accelerator input buffer

   1. Check accelerator capabilities to get input buffer size and use that as the upper limit on input data size. The host software should cache the capabilities so that it does not have to read it each time.
   2. Use the `accelerator ID` field to select which accelerator will receive the data.
   3. Send the data using either
      - `Send data to accelerator`, or
      - `Use local storage as accelerator input`

5. Start processing
   1. Send the `Basic Accelerator Control` command.
      1. Use the `accelerator id` to select which accelerator should start.
      2. Set operation ID to `Start accelerator`.
      3. Set the `Firmware ID` field to select firmware slot that should be used by the accelerator.
      4. Pack all the firmware arguments into the `Argument List` field
      5. Set correct list length and use `DPTR` to point to list location in memory.
6. Monitor accelerator status
   1. Send the `Get Accelerator Status` command with the length set to 4 bytes and `RAE=1`.
      1. Use the `Accelerator ID` field to select the target accelerator.
      2. Use the returned value as the status structure length.
   2. Send the `Get Accelerator Status` command again using the retrieved length and `RAE=0`.
      1. Check the `Status ID` field to see if the accelerator is still processing, is stopped or if an error has occurred.
      2. Use status descriptors to check capability-specific status information, e.g. the amount of output data produced for output data buffer capability.
7. Retrieve accelerator logs
   1. Send the `Get Log Page` command with `Log Page Identifier = 0xC0`, `RAE=1` and the length set to 4 bytes.
      1. Use the `Log Specific Identifier` field to provide target accelerator ID.
      2. Use the returned value as the log page length.
   2. Send the `Get Log Page` command again using the retrieved length and `RAE=0`.
      1. Process the log entry list.
8. Retrieve data from the accelerator output buffer
   1. Use the output data length obtained from `Get Accelerator Status` to see how much output data was produced.
   2. Transfer the data using either:
      - `Read the data from accelerator`, or

- `Use the local storage as accelerator output`

CHAPTER

FIVE

TENSORFLOW LITE MODEL PREPARATION

This chapter describes the preparation and conversion process of models to the TensorFlow Lite FlatBuffers format.

5.1 TensorFlow Lite models and runtime

The framework used for running inference in the *APU software* is TensorFlow Lite. TensorFlow Lite provides:

- a compiler for optimizing and converting the TensorFlow model to the *.tflite* model,
- an interpreter and runtime for running the model.

The TFLite models are represented and stored in a FlatBuffers format. The interpreter loads the model from file, and upon invoking it runs the inference.

In TensorFlow Lite, it is possible to run all or some of the model operations (matrix multiplication, convolution, vector operations and more) on an accelerator using TFLite Delegates. Delegates are libraries that tell if the current operation during runtime can be executed on the accelerator instead of the CPU (in this case APU), and if so they also implement the communication of the host (APU) with the target in order to delegate the operation and receive results.

In this project, the APU has a delegate for the Versatile Tensor Accelerator (VTA). This accelerator computes the most popular operations present in the deep learning models, such as GEMM, MIN, MAX, ADD, MUL, operations on matrices and vectors. The acceleration is performed on quantized models (with INT8 precision).

5.2 Test models used in development

For the test purposes during development, models are generated in ONNX format using the alkali-csd-fw/blob/main/apu-app/scripts/simple-models.py.

5.3 Compiling the TensorFlow Lite models with examples

The detailed description on how to compile a TensorFlow model and get the `.tflite` file is present in TensorFlow Lite Converter Overview.

The models from *Test models used in development* are compiled using random data as calibration dataset in the alkali-csd-fw/blob/main/apu-app/scripts/convert-to-tflite.py.

CHAPTER

SIX

VTA ACCELERATOR

This chapter covers the VTA accelerator - its model, structure, instructions and accessing.

Note: The full specification of the VTA accelerator, along with examples can be found in VTA Design and Developer Guide in the Apache TVM documentation.

6.1 Basic information

VTA (Versatile Tensor Accelerator) is a generic deep learning accelerator designed for efficient linear algebra calculations. It is a simple RISC-like processor consisting of four modules:

- Fetch module - loads instruction streams from DRAM, decodes them and routes them to one of the following modules based on instruction type,
- Load module - loads data from shared DRAM with the host to VTA's SRAM for processing,
- Store module - stores data from VTA's SRAM to shared DRAM,
- Compute module - takes data and instructions from SRAM and computes micro-Op kernels containing ALU (add, sub, max, . . .) and GEMM operations.

Both GEMM and ALU operations are performed on whole tensors of values.

The separate modules work asynchronously, which allows to hide memory access latency (loading new data and storing previous results while compute module processes current data). The order of operations between all three modules is ensured with dependency FIFO queues.

6.2 Key parameters of the VTA accelerator

VTA is a configurable accelerator, where the computational and memory capabilities are parameterized.

As mentioned in *Basic information*, GEMM and ALU are operating on tensors. The dimensionalities of those tensors are specified with the following parameters:

- `VTA_BATCH` - 1
- `VTA_BLOCK_IN` - 16
- `VTA_BLOCK_OUT` - 16

GEMM core computes the following tensors:

```
out[VTA_BATCH * VTA_BLOCK_OUT] = inp[VTA_BATCH * VTA_BLOCK_IN] * wgt[VTA_BLOCK_IN␣
↪* VTA_BLOCK_OUT]
```

It means that with the default settings the GEMM multiples 1x16-element input vector by 16x16 weight matrix and produces 1x16-element output vector.

ALU core computes the following tensors:

```
out[VTA_BATCH * VTA_BLOCK_OUT] = func(out[VTA_BATCH * VTA_BLOCK_OUT], inp[VTA_
↪BATCH * VTA_BLOCK_OUT])
```

It means that with the default settings the ALU core computes requested operation on 1x16 vectors.

Next, there are parameters controlling the number of bits in tensors:

- `VTA_INP_WIDTH` - number of bits for input tensor elements, 8
- `VTA_OUT_WIDTH` - number of bits for output tensor elements, 8
- `VTA_WGT_WIDTH` - number of bits for weights tensor elements, 8
- `VTA_ACC_WIDTH` - number of bits for accumulator (used in GEMM and ALU for storing intermediate results), 32
- `VTA_UOP_WIDTH` - number of bits representing micro-op data width, 32
- `VTA_INS_WIDTH` - length of a single instruction in VTA, 128

Note: The last parameter should not be modified

Another set of parameters configures buffer sizes (in bytes) for:

- `VTA_INP_BUFF_SIZE` - input buffer size, 32768 B
- `VTA_OUT_BUFF_SIZE` - output buffer size, 32768 B
- `VTA_WGT_BUFF_SIZE` - weights buffer size, 262144 B
- `VTA_ACC_BUFF_SIZE` - accumulator buffer size, 131072 B
- `VTA_UOP_BUFF_SIZE` - micro-op buffer size, 32768 B

The above parameters affect directly such aspects as:

- Data addressing in SRAM,
- Computational capabilities,
- Scheduling of operations.

6.3 VTA instructions

There are four instructions in VTA:

- `LOAD` - loads a 2D tensor from DRAM into the input buffer, weight buffer or register file, and micro-kernel into the micro-op cache.
- `STORE` - stores a 2D tensor from the output buffer to DRAM.
- `GEMM` - performs a micro-op sequence of matrix multiplications,
- `ALU` - performs a micro-op sequence of ALU operations.

The instructions have 128-bit length, storing both operation type and their parameters.

6.4 The structure of the VTA accelerator

Note: More thorough documentation can be found in VTA Design and Developer Guide.

As described in *Basic information*, there are four modules - FETCH, LOAD, COMPUTE and STORE.

FETCH module receives instructions from DRAM, and forwards them to one of the other three modules.

Each of the modules work asynchronously, fetching the instructions from the fetch module and performing actions.

The API for communicating with the VTA via its driver implementation is provided in the alkali-csd-fw/blob/main/apu-app/src/vta/vta_runtime.h.

The following subsections will provide both high-level look at operations, as well as low-level functions used to implement them.

6.4.1 Shared DRAM between VTA and host

To perform `LOAD` and `STORE` operation between the shared DRAM and VTA's SRAM modules, the shared (memory mapped) space needs to be allocated.

Managing shared buffers is done via `VTABufferAlloc(size_t size)` (allocating the memory mapped region) and `VTABufferFree(void *bufferaddr)` (releasing the memory mapped region).

6.4.2 LOAD/STORE modules

`LOAD` and `STORE` modules are responsible for passing data between shared DRAM and SRAM buffers in VTA.

They perform 2D transfers, allowing to apply padding and stride of the data on-the-fly.

> **Warning:** Some parameters in the below functions are going to have **in unit elements** disclaimer. It is the smallest tensor the SRAM can accept, and it depends on the SRAM type. The meaning of unit elements is specified in the *VTA memory/addressing scheme*.

To load the data from DRAM to VTA's SRAM, the `VTALoadBuffer2D` method is used

```
VTALoadBuffer2D(
    VTACommandHandle cmd,
    void* src_dram_addr,
    uint32_t src_elem_offset,
    uint32_t x_size,
    uint32_t y_size,
    uint32_t x_stride,
    uint32_t x_pad_before,
    uint32_t y_pad_before,
    uint32_t x_pad_after,
    uint32_t y_pad_after,
    uint32_t dst_sram_index,
    uint32_t dst_memory_type);
```

Where:

- `cmd` - VTA command handle, created using `VTATLSCommandHandle()`
- `src_dram_addr` - source DRAM address, allocated in shared space
- `src_elem_offset` - the source DRAM offset **in unit elements**
- `x_size` - the lowest dimension (x axis) size in **unit elements**
- `y_size` - the number of rows (y axis)
- `x_stride` - the x axis stride
- `x_pad_before` - start padding on x axis
- `y_pad_before` - start padding on y axis
- `x_pad_after` - end padding on x axis
- `y_pad_after` - end padding on y axis
- `dst_sram_index` - destination SRAM index
- `dst_memory_type` - destination memory type (memory types are specified in *VTA memory/addressing scheme*)

To load the data from VTA's SRAM to DRAM, the `VTAStoreBuffer2D` method is used:

```
VTAStoreBuffer2D(
    VTACommandHandle cmd,
    uint32_t src_sram_index,
    uint32_t src_memory_type,
    void* dst_dram_addr,
    uint32_t dst_elem_offset,
    uint32_t x_size,
    uint32_t y_size,
    uint32_t x_stride);
```

Where:

- `cmd` - VTA command handle
- `src_sram_index` - the beginning location of the data in given SRAM, **in unit elements**
- `src_memory_type` - source memory type (memory types are specified in *VTA memory/addressing scheme*)
- `dst_dram_addr` - pointer to DRAM memory
- `dst_elem_offset` - offset from the `dst_dram_addr`
- `x_size` - size of the tensor on x axis **in unit elements**
- `y_size` - size of the tensor on y axis
- `x_stride` - stride along x axis

Warning: Only `VTA_MEM_ID_OUT` SRAM is supported as `src_memory_type` in `VTAStoreBuffer2D`.

The above functions create 128-bit instructions that are passed to instruction fetch module, and later passed to `LOAD`/`STORE` modules.

6.4.3 COMPUTE module

`COMPUTE` module loads data from SRAM buffers - input, weight or accumulator buffers (more information in *VTA memory/addressing scheme*), and performs either `GEMM` or `ALU` operations.

The instructions for COMPUTE module are wrapped in so-called micro-op kernels - a set of instructions applied on whole ranges of SRAM buffers.

The micro-op definition starts with specifying optional outer and inner loops, created using:

```
VTAUopLoopBegin(
    uint32_t extent,
    uint32_t dst_factor,
    uint32_t src_factor,
    uint32_t wgt_factor);
```

Where:

- `extent` - the extent of the loop, in other words the number of iterations for a given loop (outer or inner)

- dst_factor - the accum factor, is a factor by which the iterator is multiplied when computing address for ACC SRAM
- src_factor - the input factor, is a factor by which the iterator is multiplied when computing address for INP SRAM
- wgt_factor - the weight factor, is a factor by which the iterator is multiplied when computing address for WGT SRAM

The end of such loop is marked with VTAUopLoopEnd(). From the driver perspective, it changes the parameters of all VTAUopPush functions within the loop's scope. All of those VTAUopPush are treated as list of micro-op instructions (uop_instructions), and those instructions along with loops are micro-op kernel.

The COMPUTE module instructions are created using:

```
VTAUopPush(
    uint32_t mode,
    uint32_t reset_out,
    uint32_t dst_index,
    uint32_t src_index,
    uint32_t wgt_index,
    uint32_t opcode,
    uint32_t use_imm,
    int32_t imm_val);
```

- mode - 0 (VTA_UOP_GEMM) for GEMM, 1 (VTA_UOP_ALU) for ALU
- reset_out - 1 if ACC SRAM in given address should be zeroed, 0 otherwise
- dst_index - the ACC SRAM base index
- src_index - the INP SRAM base index for GEMM, the ACC SRAM base index for second value for ALU
- wgt_index - the WGT SRAM base index
- opcode - ALU opcode, tells what operation is computed
- use_imm - tells if the immediate value imm_val should be used instead of tensor provided in src_index
- imm_val - immediate value in ALU mode, applied as a second value in ALU operation

The imm_val immediate value is a 16-bit signed integer.

The GEMM operation pseudo-code looks as follows

```
for (e0 = 0; e0 < extent0: e0++)
{
    for (e1 = 0; e1 < extent1; e1++)
    {
        for (instruction : uop_instructions)
        {
            src_index, wgt_index, dst_index = get_src_wgt_dst_
↪indices(instruction);
            acc_idx = dst_index + e0 * dst_factor0 + e1 * dst_factor1;
```

(continues on next page)

(continued from previous page)

```
            inp_idx = src_index + e0 * src_factor0 + e1 * src_factor1;
            wgt_idx = wgt_index + e0 * wgt_factor0 + e1 * wgt_factor1;
            ACC_SRAM[acc_idx] += GEMM(INP_SRAM[inp_idx], WGT_SRAM[wgt_idx]);
        }
    }
}
```

And the ALU operation pseudo-code looks as follows

```
for (e0 = 0; e0 < extent0: e0++)
{
    for (e1 = 0; e1 < extent1; e1++)
    {
        for (instruction : uop_instructions)
        {
            src_index, dst_index = get_src_dst_indices(instruction);
            acc_idx_1 = dst_index + e0 * dst_factor0 + e1 * dst_factor1;
            acc_idx_2 = src_index + e0 * src_factor0 + e1 * src_factor1;
            if (use_imm)
            {
                ACC_SRAM[acc_idx1] = ALU_OP(ACC_SRAM[acc_idx1], imm_val);
            }
            else
            {
                ACC_SRAM[acc_idx1] = ALU_OP(ACC_SRAM[acc_idx1], ACC_SRAM[acc_
↪idx2]);
            }
        }
    }
}
```

6.5 VTA module synchronization mechanism

The VTA LOAD, STORE, COMPUTE work asynchronously. It allows to perform data loading, storing and computations in parallel, which makes latency hiding possible.

However, it requires proper synchronization mechanism so all instructions are executed in a correct order. For this purpose, dependency queues are created.

There are four dependency queues:

- LOAD->``COMPUTE`` dependency queue - tells COMPUTE module that data has finished loading and processing can start.
- COMPUTE->``LOAD`` dependency queue - tells LOAD module that COMPUTE module has finished processing and new data can be loaded.
- STORE->``COMPUTE`` dependency queue - tells COMPUTE module that computed data from ACC SRAM is stored in shared DRAM and can be overriden with new computations.

- COMPUTE->``STORE`` dependency queue - tells STORE module that COMPUTE module has finished processing and data is ready to be stored in shared DRAM.

There are two methods for managing those dependency queues:

- VTADepPush(from, to) - for pushing a token of "readiness",
- VTADepPop(from, to) - for popping a "readiness" token from the given queue. If the token is not present, the module waits until VTADepPush pushes a new token.

This allows to control latency hiding and all of the algorithm's flow.

6.6 VTA memory/addressing scheme

VTA accelerator consists of several SRAM modules. Each of them is characterized by three parameters:

- kBits - number of bits per element,
- kLane - number of lanes in a single element,
- kMaxNumElements - maximum number of elements.

There are following SRAM modules:

- UOP SRAM (VTA_MEM_ID_UOP) - memory for storing micro-op kernels' instructions,
- WGT SRAM (VTA_MEM_ID_WGT) - memory for storing weights,
- INP SRAM (VTA_MEM_ID_INP) - memory for storing inputs,
- ACC SRAM (VTA_MEM_ID_ACC) - accumulator memory, holding the intermediate results and ALU input tensors,
- OUT SRAM (VTA_MEM_ID_OUT) - provides the casted 8-bit values from the ACC SRAM.

Table 6.1: VTA memory types

Memory type	kBits	kLane	kMaxNumElements
VTA_MEM_ID_WGT	VTA_WGT_WIDTH (8)	VTA_BLOCK_IN * VTA_BLOCK_OUT (16 * 16)	VTA_WGT_BUFF_DEPTH (1024)
VTA_MEM_ID_INP	VTA_INP_WIDTH (8)	VTA_BATCH * VTA_BLOCK_IN (1 * 16)	VTA_INP_BUFF_DEPTH (2048)
VTA_MEM_ID_ACC	VTA_ACC_WIDTH (32)	VTA_BATCH * VTA_BLOCK_OUT (1 * 16)	VTA_ACC_BUFF_DEPTH (2048)
VTA_MEM_ID_OUT	VTA_OUT_WIDTH (8)	VTA_BATCH * VTA_BLOCK_OUT (1 * 16)	VTA_OUT_BUFF_DEPTH (2048)
VTA_MEM_ID_UOP	VTA_UOP_WIDTH (32)	1	VTA_UOP_BUFF_DEPTH (8192)

VTALoadBuffer2D can write to INP, WGT and ACC SRAMs. VTAStoreBuffer2D can read from OUT SRAM (not ACC SRAM).

Warning: It means that values need to be properly requantized and clamped to prevent overflows.

CHAPTER

SEVEN

OPERATIONS ACCELERATED ON VTA ACCELERATOR

This chapter describes the currently supported TFLite operations on VTA accelerator.

7.1 TensorFlow Lite delegation scheme

TensorFlow Lite allows to delegate certain operations to an accelerator using the Delegate API. It consists of:

- `SimpleDelegateInterface` - is executed during initialization of model runtime, it decides what operations should be delegated to the accelerator based on its capabilities.
- `SimpleDelegateKernelInterface` - is executed during inference, it implements the communication with the accelerator to compute and obtain results.

7.1.1 SimpleDelegateInterface

In the VTA delegate implementation, the `VTADelegate` class derives from `SimpleDelegateInterface`.

This class requires implementing following methods:

- `IsNodeSupportedByDelegate` - this function decides whether the node (operation in the neural network model) can be delegated or not to the accelerator.
- `Initialize` - performs initialization actions for the delegation checker, not the accelerator itself.
- `Name` - returns the name of the delegate.
- `CreateDelegateKernelInterface` - creates an object inheriting from `SimpleDelegateKernelInterface`.

`IsNodeSupportedByDelegate` receives:

- `registration` data, such as `builtin_code` or `custom_name`, telling the type of the operator (e.g. `kTfLiteBuiltinAdd`, `kTfLiteBuiltinConv2d`.
- `node` is a pointer to the current node that is being considered for delegation. It contains such data as indices for inputs, outputs, intermediate and temporary tensors (indices to tensors in `context->tensors` array)
- `context` - a TFLite context containing list and count of tensors in the model, execution plan and methods for getting and manipulating tensors.

It returns `true` when the node can be delegated, `false` otherwise.

In `VTADelegate`, the currently supported operators are:

- `kTfLiteBuiltinAdd` - tensor addition of elements in 8-bit format.
- `kTfLiteBuiltinConv2d` - 2D convolution, with 8-bit inputs, kernels, outputs and 32-bit bias.

Warning: The support for `kTfLiteBuiltinConv2d` is not complete.

7.2 Adding a new operator to the delegate with 8-bit precision

The neural network models usually operate using 32-bit floats, as required during the training process to train them efficiently. However, VTA accelerator can only operate on quantized models, where weights and activations are 8-bit.

TensorFlow Lite provides methods for quantizing neural networks, as well as necessary parameters to infer quantized models during inference.

During the quantization process, the algorithm passes the calibration dataset through the neural network, and computes the following parameters for every tensor in the network (input tensors, output tensors, activation tensors, weights' tensors):

- `scale` - 32-bit float
- `zero_point` - 8-bit signed integer

During inference, the floating-point operations are simulated with integers using dequantization and requantization. Dequantization is the process of representing quantized number in a higher-precision form (e.g. 32-bit integers representing fixed-point arithmetics, as in TensorFlow Lite). Requantization is a process of bringing higher precision values (e.g. 32-bit accumulators) to 8-bit representation.

The requantization (or rather quantization) and dequantization process includes inputs and outputs.

The formula for quantization with given `scale` and `zero_point` is following:

```
Q(r) = int(s / scale) + zero_point
```

The dequantization, on the other hand, is computed as follows:

```
D(q) = cast<type>(q - zero_point) * scale
```

Where `type` corresponds to the type after dequantization (in terms of network's outputs it is 32-bit float).

When it comes to operations within the neural network, each input tensor (activations, weights, `...`) of the node (operation) needs to be dequantized, and the final output of a given node (operation) needs to be requantized.

What is more, to prevent overflows, the clamping of the outputs is performed on every outputs.

To sum up, the flow of every node should look as follow:

- Dequantize all input tensors
- Compute the operation on dequantized input tensors
- Requantize the output tensors (in higher precision)
- Clamp values in output tensors to range (the range may differ depending on the operator, usually it is the range of values in quantized values)
- Cast the output tensors to a target type
- Return the outputs.

The `scale` and `zero_point` parameters can be computed per-tensor, or per-channel. The details which variant is used in a given operation is described in TensorFlow Lite Quantization specification.

Note: It is possible to simplify requantization/dequantization process for certain operations by simplifying formulas.

In TensorFlow Lite, since the operations are supposed to be quantized and scales are 32-bit floating points, they are firstly decomposed into a normalized fraction and an integral power of two (shift):

```
scale = multiplier * 2 ^ (shift)
```

E.g. for value 96 the multiplier is 0.75 and shift is 7. The shift in TensorFlow Lite a 32-bit signed integer, and the multiplier is again 32-bit floating point value in range [0.5-1.0].

Secondly, the multiplier is multiplied by `2 ^ 31` and stored as a 32-bit Integer.

The above approach present in TensorFlow lite requires up to 64-bit registers during requantization when the 32-bit integer value (subtracted by zero point) is multiplied by 32-bit signed integer representing multiplier.

VTA accelerator cannot follow this scheme since the ACC SRAM has only 32-bit width.

To address this, the VTA delegate follows a customized approach where multiplier and shift are 16-bit signed integers instead of 32-bit signed integers. This, in the peak processing requires 32-bit registers, which still fits in VTA capabilities.

To sum up, the computation of multiplier and shift looks as follows:

```
q = frexp(scale, &shift32);
q_fixed = round(q * (1 << 15));
if (q_fixed == (1 << 15))
{
    q_fixed /= 2;
    ++shift32;
}
multiplier = static_cast<int16_t>(q_fixed);
shift = static_cast<int16_t>(shift32);
```

Dequantization of values is computed as follows:

```
valoffset = offset + val; // max 7 bits required
valshift = valoffset * (1 << left_shift); // ~15 bits required
valscaledraw32 = valshift * multiplier; // ~32 bits required
valscaled = (valscaledraw32 + nudge) >> 15; // ~16 bits required
finval = valscaled >> -shift;
```

Left shift is a constant value equal to 7. Nudge is a value used for rounding to nearest.

Note: The left shift is being embedded in the scaling factor.

Requantization of values is computed as follows:

```
valscaled = val * qdata.multiplier;
valshifted = valscaled >> (15 - qdata.shift);
valoffset = valshifted + qdata.offset;
valclamped = max(MIN, min(valoffset, MAX));
```

7.3 ADD operator

The current implementation supports adding signed 8-bit integer tensors and returning signed 8-bit integer. The operation can be represented as follows:

```
(Y_q - z_y) * s_y = (A_q - z_a) * s_a + (B_q - z_b) * s_b
Y_q = 1/s_y * [(A_q - z_a) * s_a + (B_q - z_b) * s_b] + z_y
Y_q = s_yinv * [(A_q - z_a) * s_a + (B_q - z_b) * s_b] + z_y
```

Where:

- z_y, s_y - zero point and scale for output tensor,
- z_a, s_a - zero point and scale for 1st input tensor,
- z_b, s_b - zero point and scale for 2nd input tensor,
- Y_q - quantized output,
- A_q - quantized 1st input,
- B_q - quantized 2nd input,
- s_yinv - inverted s_y.

The aim is to compute Y_q.

The scales are going through additional processing before converting to multipliers and shifts:

```
doubled_max_scale = 2 * max(s_a, s_b);
s_a' = s_a / doubled_max_scale;
s_b' = s_b / doubled_max_scale;
s_yinv' = doubled_max_scale / ((1 << left_shift) * s_y)
```

Usage of doubled_max_scale is to prevent having too small scales for 16-bit multipliers and shifts to store.

Firstly, the inputs are dequantized, so the above formula takes the following form:

```
Y_q = s_yinv * [A' + B'] + z_y
```

Warning: Current implementation performs dequantization on CPU. Those operations may need to be performed on VTA in the future to perform operations entirely on VTA.

The `A'` and `B'` are 16-bit signed integers that are passed to VTA's ACC SRAM buffer. They are aligned to have size divisible by `VTA_BATCH * VTA_BLOCK_OUT` - those are the smallest units on which VTA performs ALU operations.

After this, the delegate sends the vectors of the following length:

```
maxelements = VTA_ACC_BUFF_DEPTH / NUM_THREADS / 2 *  VTA_BLOCK_OUT
```

Where 2 stands for two input vectors to be stored in the ACC SRAM, and `NUM_THREADS` is a number of "threads" of processing in the VTA, can be either 1 or 2.

The idea of threading in VTA comes from asynchronous nature of `LOAD`, `STORE` and `COMPUTE` modules - the `COMPUTE` module can process data while `LOAD` module handles data loading from DRAM and `STORE` module stores results in DRAM. This approach is called latency hiding. The "threading" is achieved by proper management of dependency queues between the modules.

To sum up, the `LOAD` module fills half of the SRAM based on which "thread" it works on, while the `COMPUTE` module processes the data on the other half of the SRAM.

The processing of `COMPUTE` module consists of following operations:

```
A = A + B
A = A * multiplier
A = A >> (15 - shift)
A = A + offset
A = MIN(A, 128)
A = MAX(A, -127)
```

After the above operations, the ACC SRAM contains the results that can safely be casted to 8-bit integers - it can be loaded using `VTAStoreBuffer2D`.

The operation is repeated until all the elements in the input tensors are processed.

The implementation of the operation is present in `alkali-csd-fw/apu-app/src/vta-delegate-ops.cpp`.

7.4 CONV2D operator

Two dimensional convolution in TensorFlow Lite for VTA takes 8-bit input, 8-bit weights, 32-bit bias and returns 8-bit outputs. Weights are quantized symmetrically, which means that zero point for them equals 0. Assuming `x` is a convolution operator, the operations look like this:

```
(Y_q - z_y) * s_y = (s_w * W) x [s_i * (I - z_i)] + (s_b * B)
(Y_q - z_y) * s_y = s_w * s_i * [W x (I - z_i)] + (s_b * B)
```

The quantization algorithm assures that `s_b = s_w * s_i` (approximately). This leads to:

```
(Y_q - z_y) * s_y = s_w * s_i * [W x (I - z_i) + B]
Y_q = [(s_w * s_i) / s_y] * [W x (I - z_i) + B] + z_y
```

It means that convolution `W x (I - z_i)` can be performed without dequantization (values are 8-bit). The result of convolution is 32-bit, to which the 32-bit bias is added.

The only floating-point parameter here is `[(s_w * s_i) / s_y]` - it can be applied at the very end of processing (only before adding `z_y`). For this parameter the multiplier and shift are computed.

When loading data from TensorFlow Lite, the first step is to convert the data to proper, VTA-compliant layout.

Layouts for convolution data are following:

- input: `N I Hi Wi`
- weights: `O I Hk Wk`
- output: `N O Ho Wo`

Where:

- `N` - batch size,
- `I` - number of input channels,
- `Hi` - input height,
- `Wi` - input width,
- `O` - output channels,
- `Hk` - kernel height,
- `Wk` - kernel width,
- `Ho` - output height,
- `Wo` - output width.

The expected layouts by VTA are:

- input: `N' I' Hi Wi n i`
- weights: `O' I' Hk Wk o i`
- output: `N' O' Ho Wo n o`

Where:

- `n` - subgroup of batch dimension of size `VTA_BATCH` (1),
- `i` - subgroup of input channels' dimension of size `VTA_BLOCK_IN` (16),
- `o` - subgroup of output channels' dimension of size `VTA_BLOCK_OUT` (16),
- `N'` - number of batch subgroups `n`,
- `I'` - number of input channels subgroups `i`,
- `O'` - number of output channels subgroups `o`.

To convert data to this layout the original dimensions need to be:

- zero-padded so they are divisible by block computable by VTA
- rearranged so the data can be passed just for processing directly to VTA.

During convolution, for particular sample `n`, input pixel `(h,w)` and particular kernel pixel `(hk, wk)` partial convolution result is computed for 16 input channels and 16 output channels (using 16x16 weights).

Current implementation assumes that:

- at least a single input row should fit into INPUT SRAM,
- at least a single kernel (for 16 output channels) should fit into WGT SRAM,
- at least for 16 output channels, full output row, needed biases, multipliers and shifts should fit into ACC SRAM.

The pseudocode for the current implementations is as follows:

```
for each batch subgroup
    for each output channel subgroup
        LOAD weights for current output channels to WGT SRAM
        LOAD biases for current output channels to ACC SRAM
        LOAD multipliers for current output channels to ACC SRAM
        LOAD shifts for current output channels to ACC SRAM
        for each output row
            COMPUTE micro-op
                VTAFOR output channels to compute
                    VTAFOR rows to compute
                        RESET outputs in ACC SRAM
            for each input channel subgroup
                Load input row for given input channels to INP SRAM
                COMPUTE micro-op
                    VTAFOR output channels to compute
                        VTAFOR rows to compute
                            for kernel rows
                                for kernel cols
                                    RUN GEMM on data
            VTA ALU ADD bias to convolution output
            VTA ALU MUL outputs by scale multiplier
            VTA ALU SHR outputs by scale shift
            VTA ALU ADD zero_point to outputs
            Store partial outputs from OUT SRAM in DRAM
```

The implementation of the operation is present in `alkali-csd-fw/apu-app/src/vta-delegate-ops.cpp`.

7.5 Further work

- Finish testing CONV2D operator.
- Test and debug (if necessary) sequence of VTA operations.
- Load data with or without preprocessing depending on context (next VTA op vs loading data from TFLite context).
- Add loading padding and stride data from model's structure.
- Run and benchmark VTA accelerator on large network.

7.6 Resources

- TVM VTA Getting started guide
- Example demonstrating sample IR code
- TFLite tutorial on delegate implementation
- In alkali-csd-fw repository, the sources regarding delegate provide lots of useful information regarding VTA, delegating system and quantization scheme, they are also documented:
  - `apu-app/src/vta-delegate.hpp`
  - `apu-app/src/vta-delegate.cpp`
  - `apu-app/src/vta-delegate-ops.cpp`
  - `apu-app/src/vta/sim_driver.cc`

CHAPTER

EIGHT

FLASHING AND CONNECTING THE BASALT BOARD

8.1 Flashing through NVMe interface

Using the *fw-download* and *fw-commit* commands you can flash new firmware to the onboard QSPI flash.

First you need to send the firmware file to the device with:

```
sudo nvme fw-download <nvme device> --fw=<boot.bin file> --xfer=131072
```

And then trigger the flashing process with:

```
sudo nvme fw-commit <nvme device>
```

After running the *fw-commit* command you need to monitor *apu-app* logs to determine when flashing completes. Once completed then app prints out how much data was written:

```
Writing 50856272 bytes of firmware to /dev/mtd0
50856272 bytes written
```

8.2 Flashing via JTAG using Vivado

In case of anything going wrong and you cannot access Basalt via NVMe you can flash it via JTAG and Vivado

1. First you need to ensure that Basalt won't boot from QSPI because once it does the flashing will fail (Vivado hangs). There are two options available:

   a). Select JTAG boot mode

      To do this use the tiny dipsitches on the board, they should be accessible through a window in the cover. Set MODE to `3'b000` by turning on all of them (on = low).

   b). Erase the QSPI flash

      This is possible provided that you have access to the UART and at least U-Boot works. Interrupt the boot process by hitting enter in the UART console right after powering on the board (PCIe need not to be connected). Once you get an U-Boot prompt (it should say "ZynqMP>") issue the following commands to erase the flash completely:

```
sf probe 0 0 0
sf erase 0 0x8000000
```

Once the flash is erased the board needs to be power cycled.

2. Vivado needs the `BOOT.bin` file as well as `fsbl.elf` separately. **Do not extract and use the FSBL from BOOT.bin as it won't work** But you can use the same FSBL that was used to created `BOOT.bin`.
3. Connect a JTAG adapter compatible with Vivado to the J2 connector on the Basalt board.
4. Launch Vivado (tested with 2019.2 and 2021.2). It is best to do that from a terminal as during flashing it will output some progress information there which is not visible in GUI
5. Open a new HW target, You should see two "devices": `xczu7_0` and `arm_dap_1`.
6. Right click the first one and select "Add configuration memory device". Select `mt25ql01g-qspi-x4-single` for the flash type (whether it is x1, x2 or x4 should not matter).
7. Point Vivado to the `BOOT.bin` file for Basalt as well as to the extracted `fsbl.elf` file.
8. Uncheck the "Erase" option (the flash has been already erased) and "Verify" (for speedup, unless you want to be absolutely sure that the flashing succeeds). Click "OK"
9. Vivado should start flashing which may take ~30min. In the terminal (used to run Vivado) you should see the following (or simial) output:

```
Using default mini u-boot image file - ../Vivado/2021.2/data/xicom/
↪cfgmem/uboot/zynqmp_qspi_x4_single.bin
===== mrd->addr=0xFF5E0204, data=0x00000222 =====
BOOT_MODE REG = 0x0222
Downloading FSBL...
Running FSBL...
===== mrd->addr=0xFFD80044, data=0x00000000 =====
===== mrd->addr=0xFFD80044, data=0x00000003 =====
Finished running FSBL.

U-Boot 2021.01-00102-g43adebe (Oct 11 2021 - 01:44:06 -0600)

Model: ZynqMP MINI QSPI SINGLE
Board: Xilinx ZynqMP
DRAM:  WARNING: Initializing TCM overwrites TCM content
256 KiB
EL Level:       EL3
Multiboot:      16384
In:    dcc
Out:   dcc
Err:   dcc
ZynqMP> sf probe 0 0 0
SF: Detected n25q00a with page size 256 Bytes, erase size 64 KiB, total␣
↪128 MiB
ZynqMP> Sector size = 65536.
sf write FFFC0000 0 20000
```

(continues on next page)

(continued from previous page)

```
device 0 offset 0x0, size 0x20000
SF: 131072 bytes @ 0x0 Written: OK
ZynqMP> sf write FFFC0000 20000 20000
device 0 offset 0x20000, size 0x20000
SF: 131072 bytes @ 0x20000 Written: OK
```

The last three lines should continuously repead with increasing addresses.

10. Once the flashing is complete power cycle the board.

8.3 Connecting the board to the PC

For instructions on connecting the board to the PC follow alkali-csd-projects README.