Bring Up AI Accelerator Stack To Intel-Stratix10 FPGA

ECE699 Spring 2021 Master Of Engineering Project
Final Report

Tong Liu
t344liu@uwaterloo.ca

University of Waterloo
Department of Electrical and Computer Engineering
200 University Ave W, Waterloo, Ontario, N2L 3G1, Canada

August 18, 2021

1 Introduction

AI - Machine learning and Deep learning research and applications have become the hottest area in computer science and computer engineering, and widely deployed in varies user scenarios. There are strong increasing needs to push AI industry to move machine learning and deep learning application to more diversity hardware platforms depends on application requirement. Not like CPU, GPU, ASICs, and etc mainstream hardware platforms, FPGA based platform provides high flexibility, high performance and low power consumption, it becomes popular candidate of hardware platform for AI accelerator design.

1.1 Motivation and Goal of Project

1.1.1 Motivation

This project is a part of one FPGA based AI accelerator design which is supervised and mentored by Prof. Seyed M. Zahedi at ECE University of Waterloo. The goal of this AI accelerator design is using the advantage of flexibility of FPGA to design a flexible dynamic scheduling system to manage the dynamic setting for multi-processor configuration, for example systolic array processor. This system shall fully utilise FPGA resource to run multiple application at same time on the FPGA. Because sharing FPGA resource between multiple application is main challenge of operating system, in this design we shall address how to provide isolation to protect data between different application, for example taking care of memory isolation, taking care of memory bandwidth allocation. At end, ideally the system will be developed to have a dynamic setting scheduling framework, when system get applications on demand the system will make sure to meet their deadline. When system missed deadline at first time, it should be able to adjust system configuration to meet deadline.

1.1.2 Goal Of This Project

The FPGA based AI accelerator design is multi-stages project. The goal of the project is to develop a multiple AI cores accelerator system on a data centre level Intel Stratix10 FPGA based De10-PRO development board.
As first step of the development, we plan to port a open source accelerator stack - called VTA [1], to Intel Stratix 10 FPGA platform with support from an open source deep learn compiler framework - called TVM [2]. After porting VTA and TVM system to DE10-PRO development board, the next step of the project will be implement a multiple-core systolic array processor into FPGA system and making software available to manage and allocate memory usage and memory bandwidth for each core in a dynamic way while system is running.

1.2 The System Architecture

The figure1 shows the overview of system architecture for the current stage of the project. We take TVM framework run on a PC with any mainstream OS, it communicate with device - DE10Pro board via RPC protocol over TCP/IP communication. On DE10-PRO development board side a Linux system is running on top of ARM Cortex-A53 cores. Linux also controls IP cores which configured in FPGA fabric, FPGA IP core includes VTA core. Machine learning AI models such as Tensorflow or PyTorch application which compiled by TVM framework on PC can be send to DE10-PRO as client of RPC protocol, An RPC server runs on DE10-PRO, which receives AI model's binary code and data from TVM framework, then RPC server will call the VTA run-time library function to pass binary code to DDR memory buffer for VTA core to execute, and call CMA driver to allocate contiguous memory buffer to store incoming data for VTA DMA engine to load data into VTA core to compute. After VTA finished computing, VTA's DMA engine will store computed result back to contiguous memory buffer, the VTA control register will notify CPU to go to get result from the contiguous memory buffer, then send result back to TVM framework via RPC.

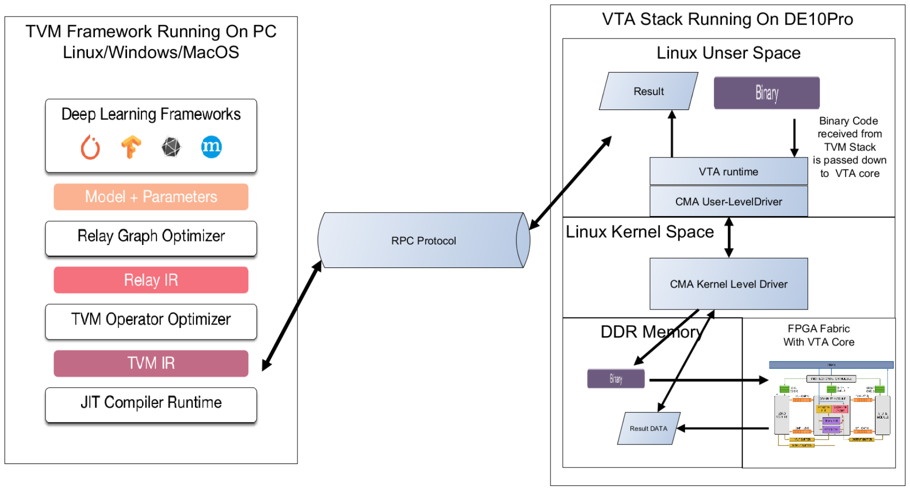


Figure 1: System Architecture Overview

1.3 TVM and VTA

As we are using TVM and VTA as fundamental framework for the first stage AI accelerator design, in this section we will have rapid overview of TVM and VTA.

1.3.1 TVM

TVM is an open source machine learning compiler framework for CPUs, GPUs, and machine learning accelerators. It enable to optimize and run computations efficiently on any hardware platform. Below figure 2 shows the overview of TVM.

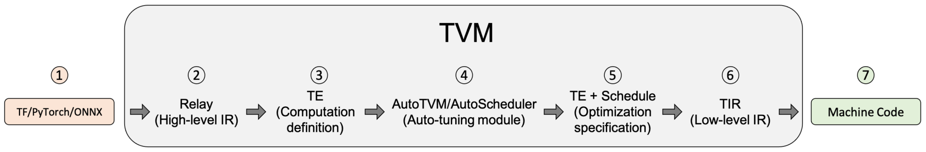

TVM
1
TF/PyTorch/ONNX
2
Relay (High-level IR)
3
TE (Computation definition)
4
AutoTVM/AutoScheduler (Auto-tuning module)
5
TE + Schedule (Optimization specification)
6
TIR (Low-level IR)
7
Machine Code

Figure 2: TVM Overview

Below description is giving a walking through of each phase of TVM.

- Phase 1, Import the model from a framework like Tensorflow, PyTorch, or Onnx.
- Phase 2, Translate to Relay - TVM's high-level model language. The model which has been imported into TVM is represented in Relay. Relay is a functional language and intermediate representation (IR) for neural

networks.

- Phase 3, Tensor Expression (TE) representation. Tensor Expression (TE) is a domain-specific language for describing tensor computations. To help in the process of converting Relay representation into TE representation, TVM includes a Tensor Operator Inventory (TOPI) that has pre-defined templates of common tensor operators.

- Phase 4, Search for the best schedule using the auto-tuning module AutoTVM or AutoScheduler. A schedule specifies the low-level loop optimizations for an operator or subgraph defined in TE. Auto-tuning modules search for the best schedule and compare them with cost models and on-device measurements. There are two auto-tuning modules in TVM.
  - **AutoTVM:** A template-based auto-tuning module. It runs search algorithms to find the best values for the tunable knobs in a user-defined template. For common operators, their templates are already provided in TOPI.
  - **AutoScheduler** (a.k.a. Ansor): A template-free auto-tuning module. It does not require pre-defined schedule templates. Instead, it generates the search space automatically by analyzing the computation definition. It then searches for the best schedule in the generated search space.

- Phase 5, Choose the optimal configurations for model compilation.

- Phase 6, Tensor Intermediate Representation (TIR) - TVM's low-level intermediate representation. After selecting the optimal configurations based on the tuning step, each TE subgraph is send to TIR and be optimized by low-level optimization. then, the optimized TIR is passed to the target compiler of the hardware platform. TVM supports several different compiler includs:
  - LLVM, which can target arbitrary microprocessor architecture including standard x86 and ARM processors, AMD GPU and NVPTX code generation, and any other platform supported by LLVM.
  - Specialized compilers, such as NVCC, NVIDIA's compiler.
  - Specialized targets, which are implemented through TVM's Bring Your Own Codegen framework.

- Phase 7, Compiling machine code. TVM can compile models down to a object module, which can then be run with a lightweight TVM runtime that provides C APIs to dynamically load the model, and entry points for other languages such as Python. TVM can also build a bundled deployment in which the runtime is combined with the model in a single package.

1.3.2 VTA

VTA stand for the **Versatile Tensor Accelerator**. It is an extension of the TVM framework designed to advance deep learning and hardware design. VTA is a programmable accelerator that exposes a RISC-like programming abstraction to describe compute and memory operations at the tensor level. VTA is more than a standalone accelerator design: it's an end-to-end solution that includes drivers, a JIT runtime, and an optimizing compiler stack based on TVM. The current version of VTA includes a hardware simulator, as well as the infrastructure to deploy VTA on FPGA hardware for fast prototyping. The figure 3 below shows overview of VTA stack.

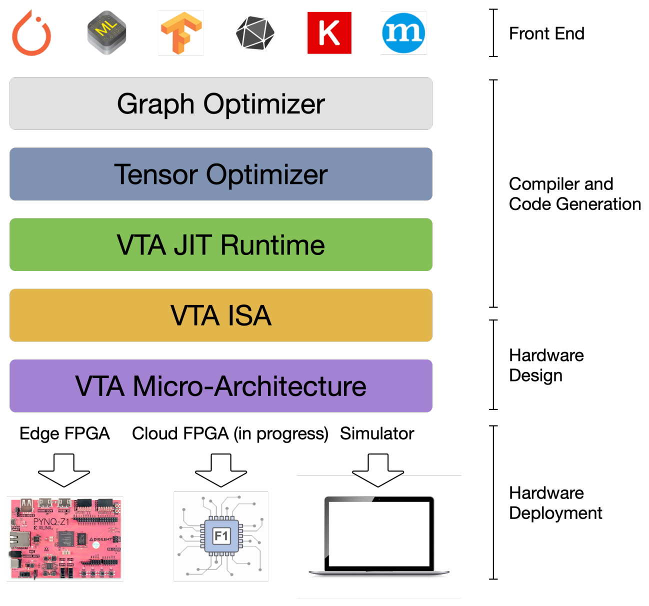


Figure 3: VTA Stack Diagram

VTA Hardware Design Overview
Figure 4 shows VTA hardware design overview. VTA is a generic deep learning accelerator built around a GEMM core, which performs dense matrix multiplication at a high computational throughput. VTA can serve as a template deep learning accelerator design, provides a clean tensor computation abstraction to the compiler stack. There are FETCH, LOAD, COMPUTE and STORE, total four modules communicate between each other through FIFO, and single

read/write SRAM blocks which support task pipeline parallelism. VTA hardware design is a template, it allows user to modify and customize it on different factors such as hardware data types, memory architecture, the GEMM core dimensions, hardware operators, and pipelining stages. By extending the TVM stack with deep learning hardware accelerator design, TVM/VTA become a transparent end-to-end deep learning stack from the high-level deep learning framework, all the way down to the hardware design and implementation. This creates software-to-hardware open source stack for deep learning systems.

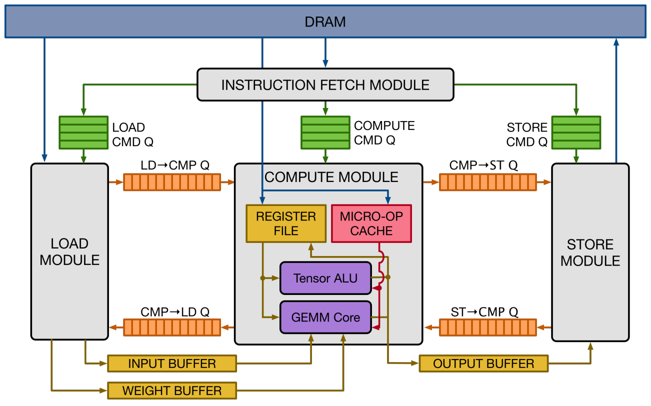


Figure 4: VTA Hardware Overview

1.4 Hardware Platform

1.4.1 Terasic DE10-Pro Development Kit

This project chose Terasic DE10-Pro Stratix® 10 SX FPGA Development Kit as development hardware platform. De10-Pro development kit provides the ideal hardware solution for designs that demand high capacity and bandwidth memory interfacing, ultra-low latency communication, and power efficiency. The DE10-Pro SX is designed for the most demanding high-end applications, empowered with the top-of-the-line Intel Stratix® 10 SX, delivering the best system-level integration and flexibility in the industry. Below diagram shows block digram of Terasic DE10PRO development board. Below block diagram figure5 shows overview of DE10-PRO development kit.

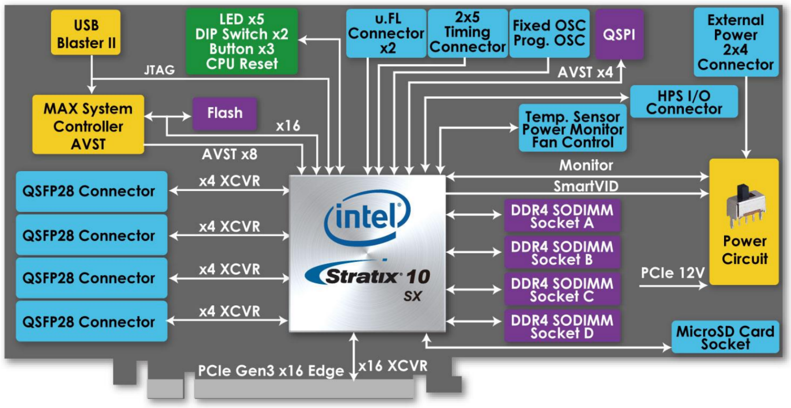


Figure 5: Terasic DE10PRO Diagram

1.4.2 Intel Stratix 10 FPGA

Terasic DE10-PRO with Intel Stratix10 provide data centre level performance. It features integrated transceivers that transfer at a maximum of 28.3 Gbps. It enables customer to deploy designs for high-speed connectivity applications. For designs that demand high capacity and high speed for memory and storage, the DE10-Pro SX delivers four SO-DIMM sockets that support DDR4 SDRAM. The DE10-Pro SX supports applications such as low-latency trading, cloud computing, high-performance computing, data acquisition, network processing, and signal processing. Figure6 shows details of internal block diagram of Intel Stratix10 SX device.

From the diagram we can see Intel Stratix 10 features four Cortex-A53 cores with L2 cache, Cache Coherency Unit, System MMU - called SMMU below, FPGA-to-HPS bridges, HPS-to-FPGA bridges, FPGA-to-SDRAM Access, and other SOC modules within Stratix 10 chip.

FPGA-to-HPS bridges, HPS-to-FPGA bridges and FPGA-to-SDRAM modules play important roles for communication between CPUs and FPGA modules, it includes both control and high speed data accessing. Specially SMMU will play a key role in managing and utilising memory in multi-core system, We will discuss SMMU in later section.

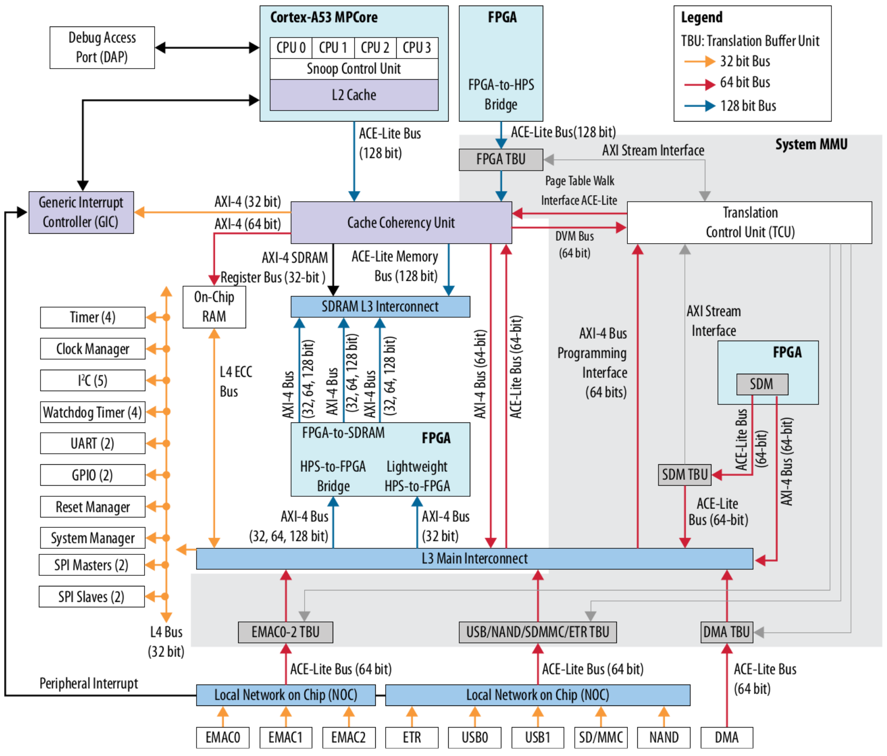


Figure 6: Stratix10 HPS Diagram

1.5 Challenges Of The Development

As we discussed in above sections, It seems very reasonable direction to archive the design goal of AI accelerator on DE10-PRO/Stratix10 FPGA, and it will for sure have great outcome in the end.
However, in reality because we are using TVM and VTA which are open source compiler framework and accelerator stack. In porting and implementation phase we faced numbers of technical barriers and unknown issues during developing. Some major issues are listed as below:

- As common drawback of open source project, it's lack of system design, and architecture document to support us to understand picture of all TVM system at beginning.
- In terms of VTA, the biggest challenge is there is no suitable driver for Intel Stratix 10 FPGA device, the exist sample design for Intel FPGA device is Cyclone V which is edge level FPGA device. It has big difference to Stratix 10 as Stratix 10 is Cloud server level FPGA device. The driver for Cyclone V device is 32bits CPU system, in contrast Stratix 10 system

is 64bits CPU system. This factor impacted in both FPGA design and Software driver developments.

- Also, the software bugs within the system become big road stopper which usually delays the progress.

1.6 Current Status and Achievement of Project

With continuously learning, experimenting and developing efforts, at the time being we are able to build TVM/VTA system on Host PC and DE10-PRO development board, Recreated FPGA system design for Stratix 10 device with VTA core build-in, Rewrite Linux VTA driver for 64bits system, and get entire data path works.
We are able to run tutorial application from TVM Host all the way down to DE10-PRO board, computed on VTA cores within Stratix 10 FPGA, and returning the result back to the TVM host.

2 Project Development Details

2.1 Setup For TVM/VTA On PC and DE10PRO Linux

2.1.1 Install Source

git clone –recursive https://github.com/apache/tvm tvm

2.1.2 Getting Right Packages

- sudo apt-get update
- sudo apt-get install -y python3 python3-dev python3-setuptools gcc libtinfo-dev zlib1g-dev build-essential cmake libedit-dev libxml2-dev

2.1.3 Development Environment Setup

- Install Cmake Tool
  sudo apt install cmake
- Install LLVM
  wget https://apt.llvm.org/llvm.sh
  chmod +x llvm.sh
  sudo ./llvm.sh 12 - For install llvm on PC
  sudo ./llvm.sh 10 - For install llvm on DE10-PRO

2.1.4 Tools For VTA and VTA Simulator and Chisel

Following items are needed for building VTA simulator, and VTA device IP core. Chisel is the tool will be used to generate VTA RTL code.
Install sbt

- Add 'sbt' to package manager Of Ubuntu 18.4 run following commands
  - echo "deb https://repo.scala-sbt.org/scalasbt/debian all main" | sudo tee /etc/apt/sources.list.d/sbt.list
  - echo "deb https://repo.scala-sbt.org/scalasbt/debian /" | sudo tee /etc/apt/sources.list.d/sbt_old.list
- curl -sL "https://keyserver.ubuntu.com/pks/lookup?op=getsearch=
  0x2EE0EA64E40A89B84B2DF73499E82A75642AC823" | sudo apt-key add
- sudo apt-get update
  apt-get install sbt

Install verilator
sudo apt install verilator

2.1.5 Build TVM on PC

Below are TVM building process:

- cd "tvm_root" folder
- mkdir build
- cp cmake/config.camke build/
- echo 'set(USE_VTA_TSIM ON)' » build/config.cmake
- echo 'set(USE_VTA_FSIM ON)' » build/config.cmake
- echo 'set(USE_LLVM llvm-config-12)' » build/config.cmake cd build
- cmake ..
- Make –jn (at here,n is CPU core/thread number)
- sudo make install

2.1.6 Build TVM VTA Simulator on PC

Build TVM Simulator TSIM

- Modify /Root of tvm/3rdparty/vta-hw/apps/tsim_example/CMakeLists.txt file.
- Change below part from -std=c++11 to -std=c++14 at line 34. set(CMAKE_CXX_FLAGS "-O2 -Wall –fPIC -fvisibility=hidden -std=c++14")
- Go to tvm_root/3rdparty/vta-hw/apps/tsim_example
  - Run 'export TVM_PATH="path_to_tvm" '
  - Run 'export VTA_HW_PATH=$TVM_PATH/3rdparty/vta-hw '
  - cp $VTA_HW_PATH/config/tsim_sample.json $VTA_HW_PATH/config/vta_config.josn
- Verilog backend
  Run 'make'
- Chisek3 Backend
  Run 'make run_chisel'

2.1.7 Build VTA Runtime Library on DE10-PRO Board

Below are building process for VTA on DE10-PRO board:

- cd "tvm_root" folder
- mkdir build
- cp cmake/config.camke build/
- echo 'set(USE_VTA_TSIM OFF)' » build/config.cmake
- echo 'set(USE_VTA_FSIM OFF)' » build/config.cmake
- echo 'set(USE_VTA_FPGA ON)' » build/config.cmake
- echo 'set(USE_LLVM llvm-config-10)' » build/config.cmake cd build
- cmake ..
- Make vta –jn (at here,n is CPU core/thread number, usually 2 or 4)
- sudo make install

2.2 Generate VTA IP Core Module Based On Scala and Chisel

VTA IP core's RTL module is generated by scala and chisel tools, run nelow commands will generate VTA IP core.

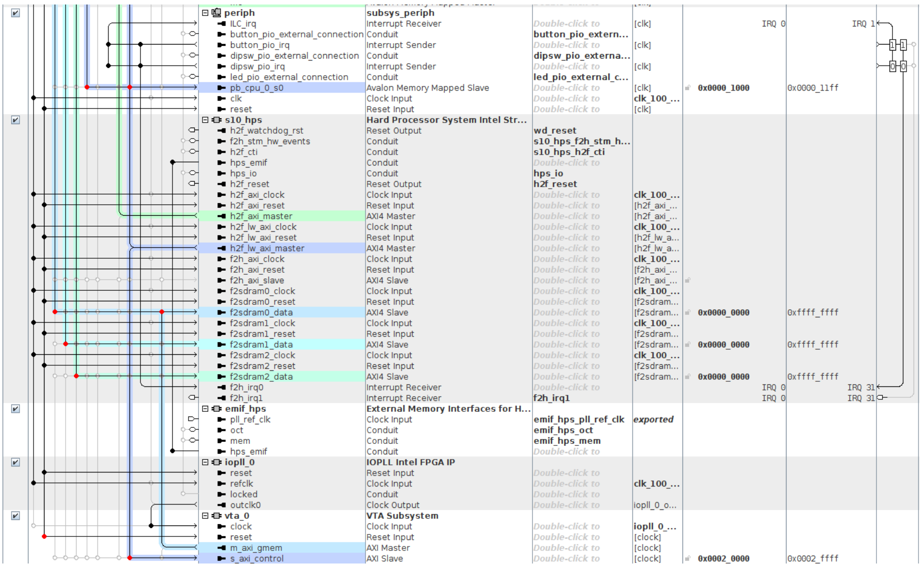
- run"cd tvm/3rdparty/vta-hw/hardware/intel"
- run "make"

2.3 Create FPGA Design With VTA Core For Intel Stratix 10 Device

As VTA reference design for Intel FPGA is Cyclone V on DE10NANO development board. It is not a good idea to directly modify VTA DE10NANO design for DE10-PRO FPGA board.
Therefore the current VTA design for Stratix10 on DE10-PRO is created based on DE10-PRO Golden Hardware Reference Design - Known as GHRD.

2.3.1 Create FPGA Design With VTA Core

This project uses Intel Quartes Pro 19.1, and platform designer tool for FPGA design changes. Here are steps within platform designer for creating new VTA ready FPGA design based on GHRD. Figure7 shows system connection diagram of VTA core within FPGA design.



periph	**subsys_periph**					
ILC_irq	Interrupt Receiver	*Double-click to*	[clk]	IRQ 0	IRQ 1	
button_pio_external_connection	Conduit	**button_pio_extern...**				
button_pio_irq	Interrupt Sender	*Double-click to*	[clk]			
dipsw_pio_external_connection	Conduit	**dipsw_pio_externa...**				
dipsw_pio_irq	Interrupt Sender	*Double-click to*	[clk]			
led_pio_external_connection	Conduit	**led_pio_external_c...**				
pb_cpu_0_s0	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x0000_1000**	0x0000_11ff	
clk	Clock Input	*Double-click to*	**clk_100_...**			
reset	Reset Input	*Double-click to*	[clk]			
s10_hps	**Hard Processor System Intel Str...**					
h2f_watchdog_rst	Reset Output	**wd_reset**				
f2h_stm_hw_events	Conduit	**s10_hps_f2h_stm_h...**				
h2f_cti	Conduit	**s10_hps_h2f_cti**				
hps_emif	Conduit	*Double-click to*				
hps_io	Conduit	**hps_io**				
h2f_reset	Reset Output	**h2f_reset**				
h2f_axi_clock	Clock Input	*Double-click to*	**clk_100_...**			
h2f_axi_reset	Reset Input	*Double-click to*	[h2f_axi_...			
h2f_axi_master	AXI4 Master	*Double-click to*	[h2f_axi_...			
h2f_lw_axi_clock	Clock Input	*Double-click to*	**clk_100_...**			
h2f_lw_axi_reset	Reset Input	*Double-click to*	[h2f_lw_a...			
h2f_lw_axi_master	AXI4 Master	*Double-click to*	[h2f_lw_a...			
f2h_axi_clock	Clock Input	*Double-click to*	**clk_100_...**			
f2h_axi_reset	Reset Input	*Double-click to*	[f2h_axi_...			
f2h_axi_slave	AXI4 Slave	*Double-click to*	[f2h_axi_...			
f2sdram0_clock	Clock Input	*Double-click to*	**clk_100_...**			
f2sdram0_reset	Reset Input	*Double-click to*	[f2sdram...			
f2sdram0_data	AXI4 Slave	*Double-click to*	[f2sdram...	**0x0000_0000**	0xffff_ffff	
f2sdram1_clock	Clock Input	*Double-click to*	**clk_100_...**			
f2sdram1_reset	Reset Input	*Double-click to*	[f2sdram...			
f2sdram1_data	AXI4 Slave	*Double-click to*	[f2sdram...	**0x0000_0000**	0xffff_ffff	
f2sdram2_clock	Clock Input	*Double-click to*	**clk_100_...**			
f2sdram2_reset	Reset Input	*Double-click to*	[f2sdram...			
f2sdram2_data	AXI4 Slave	*Double-click to*	[f2sdram...	**0x0000_0000**	0xffff_ffff	
f2h_irq0	Interrupt Receiver	*Double-click to*		IRQ 0	IRQ 31	
f2h_irq1	Interrupt Receiver	**f2h_irq1**		IRQ 0	IRQ 31	
emif_hps	**External Memory Interfaces for H...**					
pll_ref_clk	Clock Input	**emif_hps_pll_ref_clk**	***exported***			
oct	Conduit	**emif_hps_oct**				
mem	Conduit	**emif_hps_mem**				
hps_emif	Conduit	*Double-click to*				
iopll_0	**IOPLL Intel FPGA IP**					
reset	Reset Input	*Double-click to*				
refclk	Clock Input	*Double-click to*	**clk_100_...**			
locked	Conduit	*Double-click to*				
outclk0	Clock Output	*Double-click to*	iopll_0_o...			
vta_0	**VTA Subsystem**					
clock	Clock Input	*Double-click to*	**iopll_0_...**			
reset	Reset Input	*Double-click to*	[clock]			
m_axi_gmem	AXI Master	*Double-click to*	[clock]			
s_axi_control	AXI Slave	*Double-click to*	[clock]	**0x0002_0000**	0x0002_ffff	

Figure 7: Add VTA Core To FPGA Design

Here are main points of design modifications based off GHRD.

- Add PLL module, use system clock as input, export output
- Add VTA ip core, use PLL output as clock input
- s_axi_contrl connect to s10_hps's h2F_lw_axi_master
- m_axi_gmem connect to f2sdram0_data
- Assign VTA module with register offset start address 0x20000, and register address range with 0x10000

2.3.2 Generate Bitstream file For Programming Stratix 10 FPGA device

After design modification, run generating HDL, then back to Quartes run generate bitstream process. Then generated original sof file, run below highlighted script to generate new sof bitstream file with bootloader executable code build-in.
quartus_cpf –bootloader=software/u-boot/spl/u-boot-spl-dtb.hex output_files/DE10_Pro_GHRD.sof output_files/DE10_Pro_GHRD_hps.sof
In our case, it generates a sof file named **DE10_Pro_GHRD_hps.sof** in quartes project's output_files folder.
Once got final sof file generated, run below command to generate rbf files for programming FPGA device at runtime
quartus_cpf -c –hps -o bitstream_compression=on // output_files/DE10_Pro_GHRD_hps.sof output_files/DE10_Pro_GHRD_hps.rbf
In our case this command generates two rbf files in project's output_files folder.

- First rbf named **DE10_Pro_GHRD_hps_hps.rbf**, it should be programmed to DE10-PRO on board QSPI flash via JTAG by Quartes programmer.
- Second rbf named **DE10_Pro_GHRD_hps_core.rbf**, It should be stored on DE10-PRO system bootup SDCARD's boot partition, programmed to FPGA by U-boot command.

2.4 DE10-PRO U-boot Booting Sequence Changes

2.4.1 Program FPGA With Core Bitsream File

Right after system powered up, once u-boot booted up, before running any u-boot command, program FPGA device with core bitstream file which mentioned in previous section. Here is list of u-boot FPGA programming commands.

- fpga_load=fatload mmc 0:1 100000 DE10_Pro_GHRD_F2SDRAM_hps.core.rbf
- fpga_config=fpga load 0 100000 ${filesize}

First command load fpga bitstream data from **DE10_Pro_GHRD_F2SDRAM_hps.core.rbf** file to DRAM address 100000. The second command program bitstream data to FPGA device.

2.4.2 Run U-boot.scr Script

In order to make VTA communicate with Stratix 10 HPS system properly, right after FPGA device was programmed, system need to setup Stratix 10 Device internal HPS system and FPGA bridges with proper value. Below script is the u-boot script which setup Stratix 10's HPS and FPGA bridges. This process need to be done before bootloader load Linux Kernel into memory.

- Run bridge enable command
  bridge enable;
- Set the privilege filter bits for the H2F and LWH2F bridges and everything
  mw.l 0xFFD24800 0xFFFFFFFF;

- Clear the disable bit, F2H access
  mw.q 0xF70105A0 0x0000000000000000;
- Clear the non-secure and privileged valid bits, F2H access
  mw.q 0xF7010668 0x000000fffffc0000;
- Enable both secure and non-secure transactions for all masters (axi_ap, F2H, mpu) to system manager space
  mw.l 0xFFD2115C 0x01010001;
- Configure the SDRAM L3 interconnect F2SDRAM0 firewall region0 registers
  mw.l 0xF8020210 0x00000000;
  mw.l 0xF8020214 0x00000000;
  mw.l 0xF8020218 0x3FFFFFFF;
  mw.l 0xF802021C 0x00000000;

- Enable the SDRAM L3 interconnect F2SDRAM0 firewall region0
  mw.l 0xF8020204 0x00000001;
- ConfigureConfigure the SDRAM L3 interconnect F2SDRAM1 firewall region0 registers
  mw.l 0xF8020310 0x00000000;
  mw.l 0xF8020314 0x00000000;
  mw.l 0xF8020318 0x3FFFFFFF;
  mw.l 0xF802031C 0x00000000;

- Enable the SDRAM L3 interconnect F2SDRAM1 firewall region0
  mw.l 0xF8020304 0x00000001;
- Configure the SDRAM L3 interconnect F2SDRAM2 firewall region0 registers
  mw.l 0xF8020410 0x00000000;
  mw.l 0xF8020414 0x00000000;
  mw.l 0xF8020418 0x3FFFFFFF;
  mw.l 0xF802041C 0x00000000;
  Enable the SDRAM L3 interconnect F2SDRAM2 firewall region0
  mw.l 0xF8020404 0x00000001;

- Enable the F2SDRAM[012] in the DDR scheduler sideband manager
  mw.l 0xF8024050 0x00000092;
- Release the 3xF2S, F2H, H2F and LWH2F bridges from reset
  mw.l 0xFFD1102C 0x00000000;

2.5 Linux Kernel Driver Modifications To Support VTA Core

If we have deep look into VTA architecture, it has three DMA engines for input data LOAD, output result STORE and Instruction LOAD. See figure 8 below.

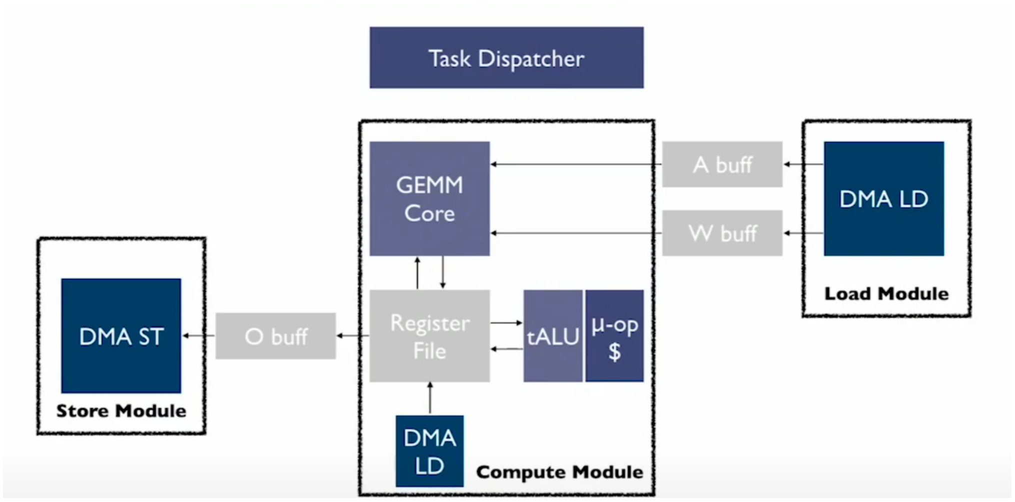


Figure 8: VTA Architecture Deep Dive

In order to support VTA transfer data in and out via DMA engine, operating system - Linux in our case, need to allocate contiguous memory buffer for VTA DMA engines.
This require Linux kernel configuration changes and an new CMA driver is needed to support contiguous memory management and data transfer as well.

2.5.1 Linux Kernel Customization and Modification

We use Terasic github Linux repository As a baseline code base. They can be cloned from https://github.com/terasic/linux-socfpga/ by git tool.
Here are works we have done for Linux kernel modifications:

- Kernel configuration change in .config file
  Changed CONFIG_CMA_SIZE_MBYTE to 64
  Changed CONFIG_FORCE_MAX_ZONEORDER to 14
- DMA Pool Size changing in Device Tree, Increased from 16MB to 44MB

- Modified CMA driver which should support 64bits CPU , and 32bits DMA channel.
  An open source CMA driver originally support DE10NANO board, which uses Intel Cyclone V device. In our work, CMA driver was modified to support 64bits CPU, 32 bits DMA channel, Instead of using direct Mmap function to map VTA DMA Buffer's physical address, we changed to use device read and write functions and **copy_to_user** and **copy_from_user** functions to transfer data between user space and VTA DMA buffer within kernel space.
- Run following command to build CMA driver
  source senenv.sh
  make
- After make command finished, copy cma.ko to DE10PRO root partition's /home/terasic folder. In our case,"terasic" is the user name, /home/terasic is the home folder of user "terasic".
- Run "sudo imsmod cma.ko" command on DE10Pro Linux once Linux finished bootup.

Source code files are avaliable on github repository -
https://github.com/torontotong/ECE699-Project.git

2.5.2 Build Linux Kernel

To build Linux kernel of stratix 10 of DE10PRO we use linaro gcc version 7.2.1, it could be downloaded from linaro website. Below shows general linux kernel build steps:

- check out linux kernel source code from github repository - https://github.com/torontotong/ECE699-Project.git.
- Go to root directory of the Linux kernel folder
- Run command "source set_env.sh"
- Run "make -jn" , n is avaliable core number on host PC
- Once make finished, find linux kernel file named - image, in folder /Root of Linux/arch/arm64/boot
- Copy image to DE10-PRO sdcard's boot partition.
- find linux devicetree file named - socfpga_stratix10_de10_pro.dtb, in folder /Root of Linux/arch/arm64/boot/dts/altera
- Copy socfpga_stratix10_de10_pro.dtb to DE10-PRO sdcard's boot partition.

2.6 VTA User Space VTA Driver Modifications

In order to match kernel level modification, we did numbers of changes in user space driver too. Here are the major changes.

- Convert CMA driver returned pointer to be 64bits
- Add New CMA IOCTL Commands

```
#define CMA_SET_WRITE_TARGET_ADDR
#define CMA_GET_READ_SRC_ADDR
```

- Add CMA read() and write() Corresponding Functions

  – int cma_memcpy_from_host(void* dst, const void* src, size_t size);
  – int cma_memcpy_to_host(void* dst, const void* src, size_t size);

Source code are avaliable on github repository -
https://github.com/torontotong/ECE699-Project.git

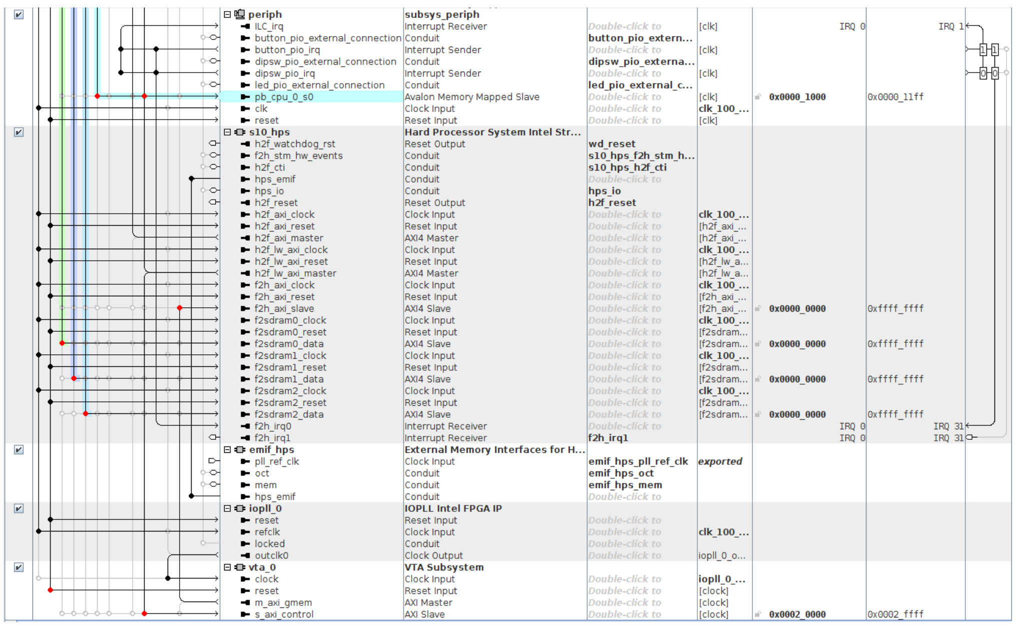
2.7 Remaining Issues For Future Work

2.7.1 SMMU Driver didn't Work

Currently, even we got TVM framework and VTA accelerator working, There is still a important part is not working as we expected. That is using SMMU to manage FPGA memory usage. As we mentioned in section of introduction of Intel Stratix 10, SMMU will play a key role in later stage of project to help dynamically allocating and managing memory usage between different accelerator cores. The Linux kernel has SMMU driver build-in, however, according to the comment from the DE10-PRO maker Terasic, the SMMU module was not guaranteed to work on DE10-PRO, Terasic hasn't done this part of verification yet. Being aware this comment, We had tried to enable SMMU driver within Linux kernel, and resulting with system crash while Linux was boot up. As spring 2021 term is approaching to end, SMMU issue become current stage's remaining issue. In next section, we will discuss more details of SMMU, such as, What is SMMU, what is the strategy to solve the SMMU driver issue, How to use SMMU in the system once driver issue was solved.

2.7.2 FPGA Design With F2H Bridge Didn't Work

Another issue we need to resolve in the future work. We actually tried to work on a FPGA design with using F2H bridge for VTA core to communicate with CCU module in HPS. In this design the VTA's m_axi_gemm port is connected to s10 HPS's f2h_axi_slave port. However the design didn't work, the VTA control register always lockup when try to check VTA status bit. The design configuration diagram shows in the figure 9.



periph	**subsys_periph**				
ILC_irq	Interrupt Receiver	*Double-click to*	[clk]	IRQ 0	IRQ 1
button_pio_external_connection	Conduit	**button_pio_extern...**			
button_pio_irq	Interrupt Sender	*Double-click to*	[clk]		
dipsw_pio_external_connection	Conduit	**dipsw_pio_externa...**			
dipsw_pio_irq	Interrupt Sender	*Double-click to*	[clk]		
led_pio_external_connection	Conduit	**led_pio_external_c...**			
pb_cpu_0_s0	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x0000_1000**	0x0000_11ff
clk	Clock Input	*Double-click to*	**clk_100_...**		
reset	Reset Input	*Double-click to*	[clk]		
s10_hps	**Hard Processor System Intel Str...**				
h2f_watchdog_rst	Reset Output	**wd_reset**			
f2h_stm_hw_events	Conduit	**s10_hps_f2h_stm_h...**			
h2f_cti	Conduit	**s10_hps_h2f_cti**			
hps_emif	Conduit	*Double-click to*			
hps_io	Conduit	**hps_io**			
h2f_reset	Reset Output	**h2f_reset**			
h2f_axi_clock	Clock Input	*Double-click to*	**clk_100_...**		
h2f_axi_reset	Reset Input	*Double-click to*	[h2f_axi_...		
h2f_axi_master	AXI4 Master	*Double-click to*	[h2f_axi_...		
h2f_lw_axi_clock	Clock Input	*Double-click to*	**clk_100_...**		
h2f_lw_axi_reset	Reset Input	*Double-click to*	[h2f_lw_a...		
h2f_lw_axi_master	AXI4 Master	*Double-click to*	[h2f_lw_a...		
f2h_axi_clock	Clock Input	*Double-click to*	**clk_100_...**		
f2h_axi_reset	Reset Input	*Double-click to*	[f2h_axi_...		
f2h_axi_slave	AXI4 Slave	*Double-click to*	[f2h_axi_...	**0x0000_0000**	0xffff_ffff
f2sdram0_clock	Clock Input	*Double-click to*	**clk_100_...**		
f2sdram0_reset	Reset Input	*Double-click to*	[f2sdram...		
f2sdram0_data	AXI4 Slave	*Double-click to*	[f2sdram...	**0x0000_0000**	0xffff_ffff
f2sdram1_clock	Clock Input	*Double-click to*	**clk_100_...**		
f2sdram1_reset	Reset Input	*Double-click to*	[f2sdram...		
f2sdram1_data	AXI4 Slave	*Double-click to*	[f2sdram...	**0x0000_0000**	0xffff_ffff
f2sdram2_clock	Clock Input	*Double-click to*	**clk_100_...**		
f2sdram2_reset	Reset Input	*Double-click to*	[f2sdram...		
f2sdram2_data	AXI4 Slave	*Double-click to*	[f2sdram...	**0x0000_0000**	0xffff_ffff
f2h_irq0	Interrupt Receiver	*Double-click to*		IRQ 0	IRQ 31
f2h_irq1	Interrupt Receiver	**f2h_irq1**		IRQ 0	IRQ 31
emif_hps	**External Memory Interfaces for H...**				
pll_ref_clk	Clock Input	**emif_hps_pll_ref_clk**	***exported***		
oct	Conduit	**emif_hps_oct**			
mem	Conduit	**emif_hps_mem**			
hps_emif	Conduit	*Double-click to*			
iopll_0	**IOPLL Intel FPGA IP**				
reset	Reset Input	*Double-click to*			
refclk	Clock Input	*Double-click to*	**clk_100_...**		
locked	Conduit	*Double-click to*			
outclk0	Clock Output	*Double-click to*	iopll_0_o...		
vta_0	**VTA Subsystem**				
clock	Clock Input	*Double-click to*	**iopll_0_...**		
reset	Reset Input	*Double-click to*	[clock]		
m_axi_gmem	AXI Master	*Double-click to*	[clock]		
s_axi_control	AXI Slave	*Double-click to*	[clock]	**0x0002_0000**	0x0002_ffff

Figure 9: FPGA Design With F2H Bridge

3 System Memory Management Unit - SMMU [5]

Although the SMMU driver didn't work for current stage, as SMMU plays a key role in next stage of project. At here We would like to take deep look into SMMU.

3.1 What is SMMU

In general SMMU is belong one big class of Computer component - IO Memory Management Unit , as known as IOMMU[3][4]. IOMMU translates device virtual address to physical addresses. In ARM's case the IOMMU is called System Memory Management Unit.
As we discussed in section 1.4.2. System Memory Management Unit - SMMU plays a key role for this project. SMMU plays similar role as MMU for CPU, SMMU translates addresses for DMA request from I/O devices before the requests are passed into the system interconnect. It also perform this translation of DMA address for isolation too. These are main functionalities the project will use to dynamically manage memory usage and bandwidth for multiple cores. figure 10 shows the data traffice flow of SMMU in DMA Traffic.

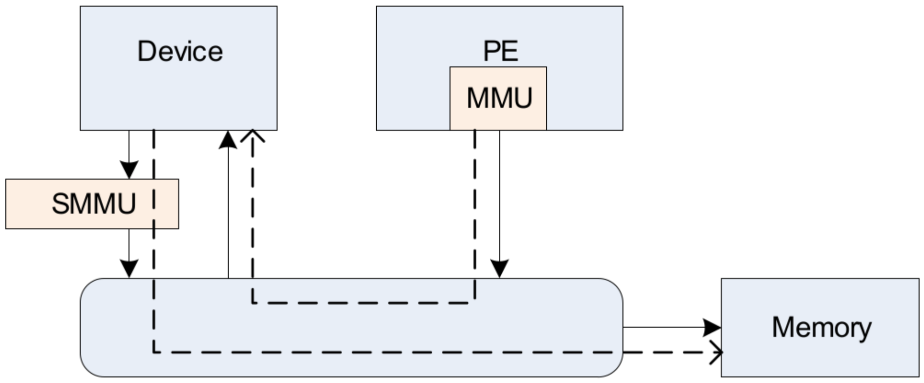


Figure 10: SMMU In DMA Traffic

The SMMU supports two stages of translation, Each stage of translation can be enabled independently. An address is logically translated from Virtual address to Intermediate Physical Address in stage 1, then the Intermediate Physical Address is input to stage 2 which translates the IPA to the output Physical Address. Stage 1 is intended to be used by a software context to provide isolation or translation to buffers within the context. Stage 2 is intended to be available in systems supporting the Virtualization Extensions and is intended to vitalize device DMA to guest Virtual Machine address spaces. When both stage 1 and stage 2 are enabled, the translation configuration is called nested. ARM provides three type of interfaces in SMMU for software to use.

- Memory-based data structures to map devices to translation tables which are used to translate client device addresses
- Memory-based circular buffer queues, a command queue, a event queue , and a PRI queue
- A set of Registers

3.2 Architecture And Data Structure Of SMMU

3.2.1 Stream Number

Since SMMU may have more than one client device, SMMU uses **StreamID** to differentiate sources. The StreamID is used as a key of one transaction configuration. Depends on the configuration system can determines the transaction is a bypass or translation, or transaction is a stage 1 or stage2 translation. SMMU registers keep the base addresses of initial root structure - Stream Table.

3.2.2 Translation Procedure

The SMMU uses a set of data structure in memory to manage translation data. A Stream table entry (STE) contains stage 2 translation table base pointers, and also locates stage 1 configuration structures, which contain translation table base pointers. A Context descriptor (CD) represents stage 1 translation, and a Stream table entry represents stage 2 translation.
SMMU uses two types structures:

- Configuration structure
  which map from the StreamID of a transaction to the translation table base pointers, configuration, and context under which the translation tables are accessed
- Translation table structures that are used to perform the VA to IPA and IPA to PA translation of addresses for stage 1 and stage 2, respectively.

The procedure for translation of a transaction is to locate configuration for that transaction by checking its StreamID and, optionally, SubstreamID, and then to use that configuration to locate translations for the address used.

3.2.3 Stream Table Lookup

SMMU uses transaction StreamID to determine a STE. It supports two types of Stream Table. The format is set by the Stream table base registers.

- **Linear Stream Table**
  A linear Stream table is a static array of STEs, start from 0 as StreamID. The size is configurable as a 2^n multiple of STE size up to the maximum number of StreamID bits supported in hardware by the SMMU.

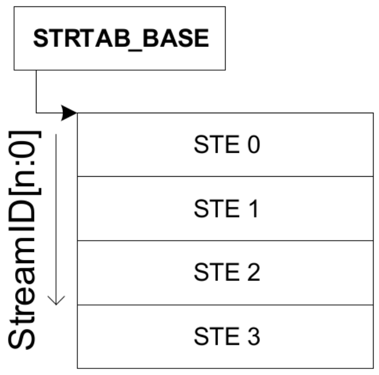


Figure 11: Linear Stream Table

- **2-Level Stream Table** A 2-level Stream table is a structure consisting of one top-level table that contains descriptors that point to multiple second-level tables that contain linear arrays of STEs. This structure saves memory and do not require large physically-contiguous memory for very large StreamID data.

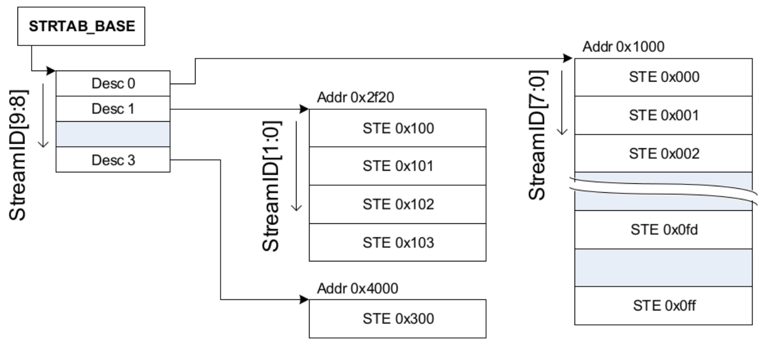


Figure 12: 2-Level Stream Table

3.2.4 StreamID and Context Descriptors

The STE contains the configuration for each stream indicating:

- traffic is enabled or not
- Is stage 1 translation
- Is stage 2 translation, and the relevant translation tables.
- translation tables for stage 1.

From figure 13, we can see that :
In case of stage 1 is used, the STE indicates the address of one or more CDs in memory using the STE's S1ContextPtr field.
In case of stage 2 is used, the STE contains the stage 2 translation table base pointer (to translate IPA to PA) and VMID.

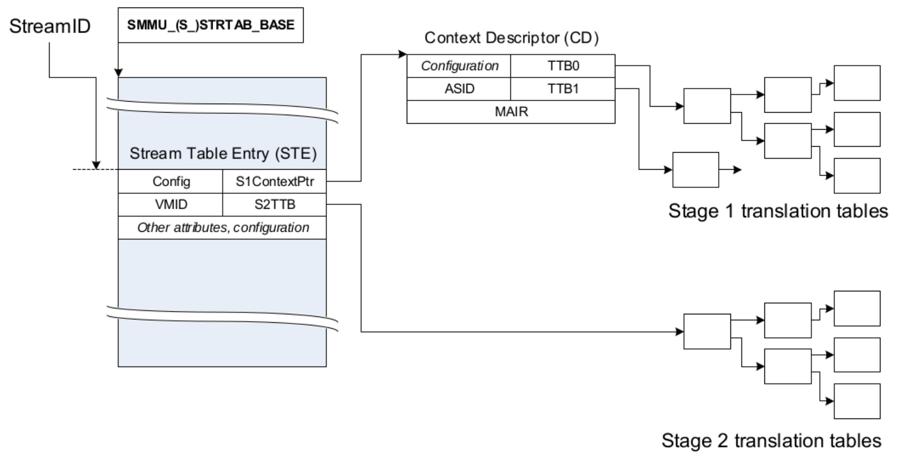


Figure 13: 2 Stage Translation Table

If there are multiple tasks running on the each device, and these tasks use different page tables at same time, SMMU use the CD table to manage each page. See figure 14.

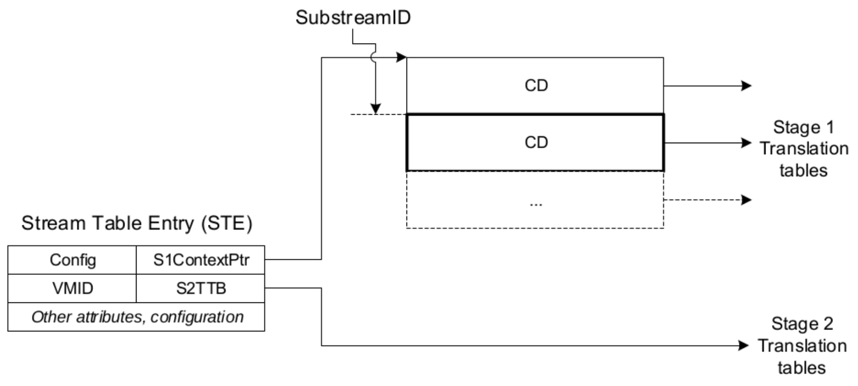

SubstreamID
CD
CD
...
Stage 1 Translation tables
Stream Table Entry (STE)
Config
S1ContextPtr
VMID
S2TTB
Other attributes, configuration
Stage 2 Translation tables

Figure 14: Multi CD For Streams

And figure 15 shows more complex layout in which a multi-level Stream table is used. Two of the STEs point to a single CD, or a flat array of CDs, whereas the third STE points to a multi-level CD table. With multiple levels, many streams and many substreams might be supported without large physically-contiguous tables.

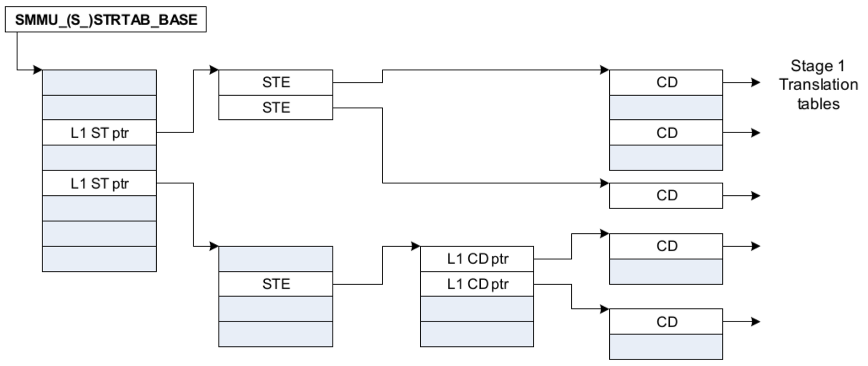


Figure 15: Multi-Level Stream and CD Tables

3.2.5 Translation Logic

A general view of SMMU translation stages shows in figure 16.

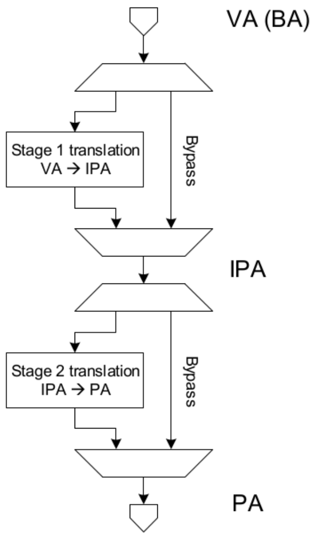


Figure 16: Translation Stages and Addresses

For SMMU to decide how to deal with a incoming transaction, here are numbers of steps to follow.

- If the SMMU is globally disabled the transaction passes through the SMMU without any address modification. Global attributes, such as memory type or Share-ability, might be applied from the SMMU_GBPA register of the SMMU. Or, the SMMU_GBPA register might be configured to abort all transactions.

- If the global bypass described in above does not apply, the configuration is determined:
  - An STE is located
  - If the STE enables stage 2 translation, the STE contains the stage 2 translation table base.
  - If the STE enables stage 1 translation, a CD is located. If stage 2 translation is also enabled by the STE, the CD is fetched from IPA space which uses the stage 2 translations. Otherwise, the CD is fetched from PA space
- Translations are performed, if the configuration is valid
  - If stage 1 is configured to translate, the CD contains a translation table base which is walked. This might require stage 2 translations, if stage 2 is enabled for the STE. Otherwise, stage 1 bypasses translation and the input address is provided directly to stage 2
  - If stage 2 is configured to translate, the STE contains a translation table base that performs a nested walk of a stage 1 translation table if enabled, or a normal walk of an incoming IPA. Otherwise, stage 2 bypasses translation and the stage 2 input address is provided as the output address
- A transaction with a valid configuration that does not experience a fault on translation has the output address applied and is forwarded.

3.3 IOMMU/SMMU in Linux Kernel [4]

This section describes an overview of the IOMMU/SMMU in Linux kernel and shows how I/O requests are propagated in the Linux kernel.

3.3.1 Overview of IOMMU/SMMU in Linux Kernel

As shown in figure 17, the iommu/smmu subsystem contains three layers:

- IOMMU DMA Layer: This layer receives the DMA requests from I/O devices and forwards the request to IOMMU generic layer. It is the glue layer between DMA-API and IOMMU-API.
- IOMMU Generic Layer (or IOMMU-API Layer): This layer provides generic IOMMU APIs for interaction with IOMMU DMA layer and hardware specific IOMMU layer

- Vendor hardware Specific layer: This is a hardware-specific driver in order to interact with the underlying vendor hardware. It also configures the proper I/O page table based on the requested DMA address so that IOMMU hardware can translate DMA address correctly

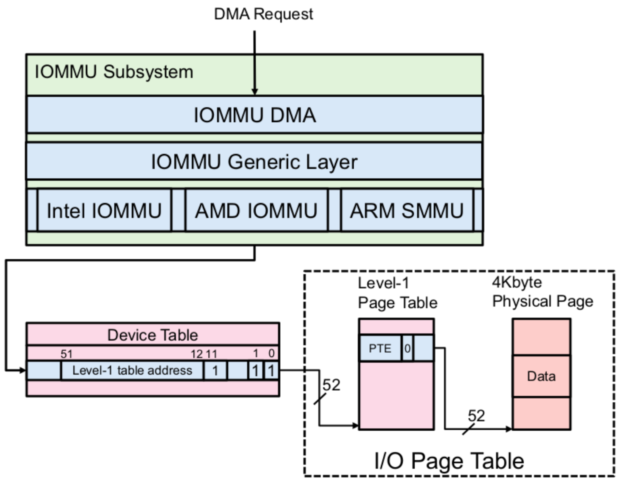


Figure 17: IOMMU/SMMU In Linux Kernel: High-level Overview

3.3.2 Interaction Between Device Driver,IOMMU/SMMU

When DMA requests are initiated from I/O devices, Linux kernel forwards and processes them in device driver, DMA subsystem and IOMMU/SMMU module as shown in Figure 18.

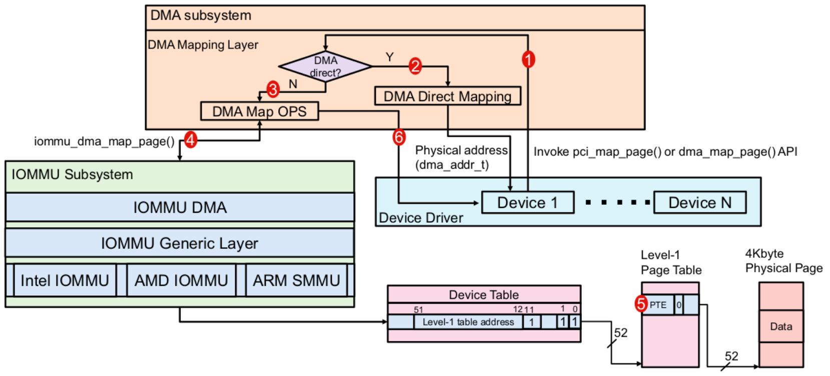


Figure 18: Interaction Device Drive, DMA and IOMMU/SMMU

The process steps through 1-5 in Figure 18 shown as follows:

- Device driver invokes dma_map_page() API to get the physical address. Note that the physical address means the guest physical address if the running OS is in virtualization environment
- If the DMA request is directly mapped, DMA subsystem returns the calculated physical address to device driver directly
- If the DMA request is not directly mapped, the DMA request is forwarded to IOMMU subsystem
- DMA subsystem invokes iommu_dma_map_page() to request IOMMU subsystem to map virtual address to physical address
- IOMMU subsystem maps virtual address to physical address and configures the corresponding I/O page table so that IOMMU hardware can proceed address translation correctly

3.4 Proposed Work For IOMMU/SMMU In Linux Kernel

As section 2.7 mentioned, due to the SMMU driver issue, as result current stage development couldn't use SMMU for DMA transaction. It as shown in figure 18, at step 2 within DMA mapping layer the IOMMU/SMMU was bypassed the tansactrion went to DMA direct Mapping.
Therefore, in future work once SMMU driver issue solved, the developemnt shall focus on the Step 3 -> 4 -> 5 -> 6 path which shown in figure 18 to involve SMMU into transaction.

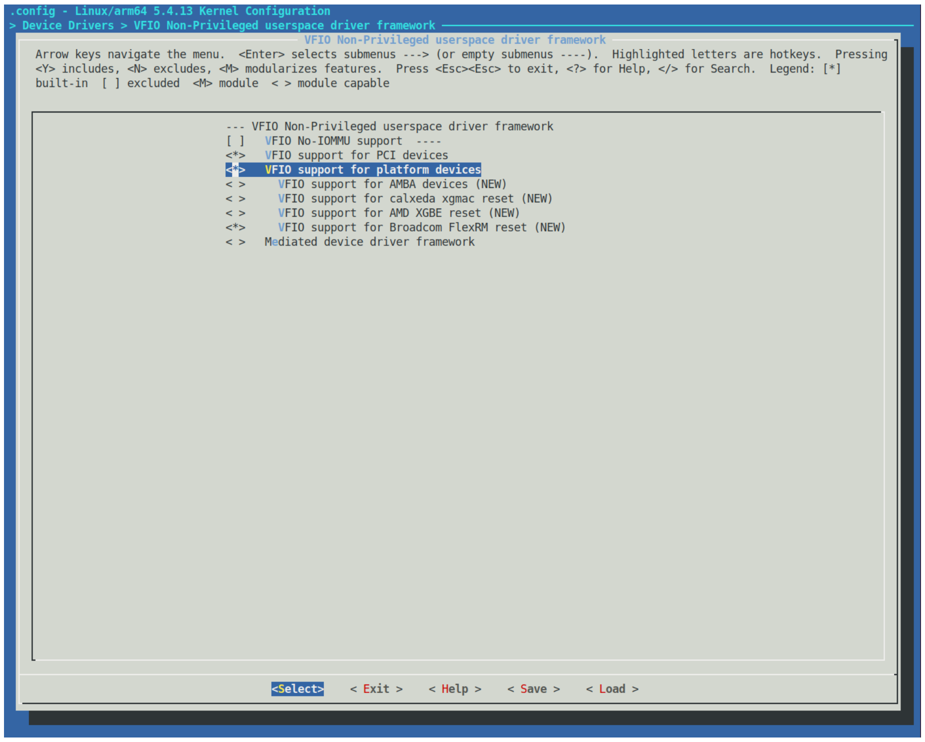
3.4.1 Recommended Development in Linux Kernel

In order to enable SMMU driver, the following Modificaton are needed:

- Modify devicetree source file, add below block into bottom of the file ../linux-socfpga/arch/arm64/boot/dts/altera/socfpga_stratix10_de10_pro.dts

```
&smmu {
    status = "okay";
};
```

- Run "make menuconfig" coomand to get into Linux Kernel Config menu as shown in firgutr 19, go to chose Device Driver -> VFIO Non-Provoleged userspace driver framework, enable "VFIO support for platform device"



```
.config - Linux/arm64 5.4.13 Kernel Configuration
> Device Drivers > VFIO Non-Privileged userspace driver framework
                VFIO Non-Privileged userspace driver framework
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).  Highlighted letters are hotkeys.  Pressing
  <Y> includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*]
  built-in  [ ] excluded  <M> module  < > module capable

                        --- VFIO Non-Privileged userspace driver framework
                        [ ]   VFIO No-IOMMU support  ----
                        <*>   VFIO support for PCI devices
                        <*>   VFIO support for platform devices
                        < >     VFIO support for AMBA devices (NEW)
                        < >     VFIO support for calxeda xgmac reset (NEW)
                        < >     VFIO support for AMD XGBE reset (NEW)
                        <*>     VFIO support for Broadcom FlexRM reset (NEW)
                        < >   Mediated device driver framework

                    <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 19: Linux Kernel Configure

- Using the VFIO Platform Driver will create new device file /dev/vfio/vfio in Linux. User space application will be able to use this device file to get mapped DMA buffer address and access DMA buffer via SMMU driver.

4 Summary

This work started as the ECE699 engineering project. However, it is trying to solve real world problem as start point of research project. The project got progress on making system works from a open source based project and platform. As the hardware platform difference caused numbers of engineering road blockers. At the end, with big effort of work, the system got running with all pieces of hardware, software and FPGA IP core, even there are still remaining issues. Author learned a lots of knowledge about AI accelerator, Machine learning models, FPGA design and computer system architecture like iommu/smmu. At here I would to thank Prof. Seyed M. Zahedi to provide author the opportunity to join the work.

References

[1] Thierry Moreau , Tianqi Chen , Luis Vega , Jared Roesch , Eddie Yan, Lianmin Zheng , Josh Fromm , Ziheng Jiang , Luis Ceze , Carlos Guestrin , Arvind Krishnamurthy
A Hardware-Software Blueprint for Flexible Deep Learning Specialization.
Paul G. Allen School of Computer Science Engineering, University of Washington,
Shanghai Jiao Tong University

[2] Tianqi Chen , Thierry Moreau , Ziheng Jiang , Lianmin Zheng , Eddie Yan Meghan Cowan , Haichen Shen , Leyuan Wang , Yuwei Hu , Luis Ceze , Carlos Guestrin , Arvind Krishnamurthy
TVM: An Automated End-to-End Optimizing Compiler for Deep Learning.
Paul G. Allen School of Computer Science Engineering, University of Washington, AWS, Shanghai Jiao Tong University, UC Davis, Cornell

[3] Muli Ben-Yehuda, Jon Mason, Orran Krieger,Jimi Xenidis, Leendert Van Doorn, Asit Mallick, Jun Nakajima,Elsie Wahlig
Utilizing IOMMUs for Virtualization in Linux and Xen
AMD

[4] Adrian Huang
An Introduction to IOMMU Infrastructure in the Linux Kernel
Lenovo Press

[5] *Arm System Memory Management Unit Architecture Specification*
Arm Limited

[6] Website For De10Pro Board: *https://rocketboards.org/*

[7] Tong Liu
ECE699 Spring 2021 Project Repository
https://github.com/torontotong/ECE699-Project.git