OpenCL support for Enzian

Master Thesis

Author(s):
Wüthrich, Fabian

Publication date:
2021-09

Permanent link:
https://doi.org/10.3929/ethz-b-000511824

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 352

Systems Group, Department of Computer Science, ETH Zurich

OpenCL support for Enzian

by

Fabian Wüthrich

Supervised by

Anastasiia Ruzhanskaia
Prof. Dr. Timothy Roscoe

March 2021–September 2021

DINFK

Abstract

Recently, there has been growing interest in custom and reconfigurable hardware. However, commercial hardware platforms have limitations for research, because they are optimized for specific use-cases or have proprietary parts. To address this problem, the Systems Group at ETH built Enzian, a new computing platform tailored for research and for exploring novel hardware/software co-designs. However, there is a lack of existing use cases that could promote Enzian as a viable research platform. In addition, Enzian, like most CPU-FPGA hybrid systems, is hard to use for computer scientists without a hardware development background. In this thesis, we add OpenCL support to Enzian, which not only represents a real-world use case but also increases the usability of the platform. Using the flexibility of the Enzian computing platform, we explore different designs and compare their performance with existing systems. Our preliminary experimental results show that Enzian can support key features of existing OpenCL implementations. Because of OpenCL, we were able to port existing applications to Enzian with little effort. Hence, OpenCL improves the usability of Enzian and serves as a good use case to prove the feasibility of the platform. To summarize, Enzian proved to be a viable platform for computer systems research and will hopefully support researchers in developing innovative hardware solutions.

Acknowledgements

I would like to express my great appreciation to my supervisor Anastasiia Ruzhanskaia for her valuable and constructive feedback throughout the development of this thesis. I also want to thank Prof. Dr. Timothy Roscoe for giving me the opportunity to work with Enzian and for making this thesis possible. My grateful thanks are also extended to the other members of the Systems Group, in particular to Abishek Ramadas, who helped with the ECI layer and to Dr. David Cock, who put me into contact with the staff from Xilinx. On that note, I would like to thank Xilinx for their assistance with Vitis. Finally, I wish to thank Manuela Heuberger and my family for their support and encouragement throughout my study.

Contents

Chapter 1

Introduction

Recently, there has been growing interest in custom and reconfigurable hardware. For instance, large internet companies use custom ASICs for machine learning [24, 39]. Search engines deploy FPGAs in their data centers to accelerate a variety of online services [7, 12]. Products such as Amazon F1 [2] bring FPGAs to the cloud, so that even companies without large data centers can use them. The reasons for this broad adoption of custom hardware are twofold. First, there has been a significant improvement in design tools and manufacturing processes. Second, FPGAs are power efficient and can be reconfigured for different workloads. However, the cost for developing custom hardware is still immense and only viable for large companies.

As a consequence, academic computer scientists use the same custom computing platforms for their research as the industry. This approach has worked for the last few decades because hardware has become faster but the underlying paradigm has not changed much. However, today's custom hardware is so radically different that research using this equipment is increasingly unrealistic.

For many universities, it is a real challenge to gain access to this kind of new hardware. Although R&D departments continue to develop novel platforms such as Microsoft Catapult [31] and Intel Harp [19], these systems are only used internally or companies provide limited access, for example, as a cloud service. Additionally, these platforms often lack documentation and are hard to program. Therefore, academic researchers spend their time understanding a platform that will be obsolete in a couple of years instead of designing innovative hardware solutions.

The Systems research group at ETH addresses these problems by building Enzian, a computer which is not designed for a specific use case such as machine learning or computer vision but rather for flexibility and for exploring new hardware/software co-designs. Enzian is a heterogeneous system that

consists of a 48-core ARM CPU and a large Xilinx FPGA. The CPU and FPGA are not as usually connected over PCIe but over the native cache coherence protocol of the CPU.

To promote Enzian as a viable research platform, it is necessary to implement and benchmark existing use cases. Enzian, like most CPU-FPGA hybrid systems, is difficult to use for computer scientists without a hardware background. By adding support for a well-known industry standard, like OpenCL, we can address both of the previous points. On the one hand, we demonstrate that Enzian supports the same functionality as existing FPGA accelerators. On the other hand, we improve the usability of Enzian and make the platform available for the large community of OpenCL developers.

This thesis adds OpenCL support for Enzian. Instead of implementing the complete OpenCL API, we adapted the Vitis software platform from Xilinx. Vitis is a framework to create accelerated applications for Xilinx FPGAs. In particular, Vitis provides a complete OpenCL implementation. However, Xilinx does not officially support Enzian so we modified the Vitis tools to make them compatible with the Enzian board. This includes the following contributions:

- A shell that provides the necessary infrastructure for OpenCL kernels on the FPGA.
- Build scripts to adjust the Vitis build tools for Enzian.
- Patches for a Linux driver to execute the Xilinx Runtime on Enzian.

Our results show that Enzian can execute OpenCL applications and that their performance is predictable. Finally, we point out which key features of OpenCL and Vitis are supported by Enzian.

Chapter 2

Background

2.1 OpenCL

This section gives a brief introduction into the Open Computing Language (OpenCL) and covers the important concept for this work in detail. Xilinx officially supports OpenCL 1.2 so the following sections are based on this version [61]. For a in-depth study of OpenCL see [13] or [26].

OpenCL is a heterogeneous programming framework maintained by the non-profit technology consortium Khronos Group. OpenCL supports a wide range of heterogeneous platforms consisting of CPUs, GPUs, FPGAs and other hardware accelerators. OpenCL provides a well-defined abstraction of the underlying hardware so porting applications to other platforms requires little effort. Despite this abstraction, OpenCL applications are still adaptable enough to obtain high performance from a given hardware platform. OpenCL also enables parallel computing using task- and data-based parallelism.

The OpenCL specification is defined by four models:

1. Platform model: Defines the abstract hardware model that makes the kernels portable. The model specifies that there is one coordinator (host) that manages one or more accelerators (devices). A device executes functions which are called kernels.
2. Execution model: Defines the execution environment and the interaction between the host and devices. It further specifies the concurrency model for the kernel execution.
3. Memory model: Defines the abstract memory hierarchy for the kernels independent from the underlying hardware.
4. Programming model: Defines how the concurrency models is mapped to the physical hardware. Typically, a device is divided into compute units (CU) and each CU has several processing elements (PE).

As an example to illustrate these models, let us consider a cloud service that offers FPGA acceleration. For this service, an accelerator card like the Xilinx u250 is connected to the x86 CPU over PCIe. The platform model specifies the accelerator card as a device and the CPU as the host. The host instantiates the execution environment and defines the degree of parallelism e.g. how many kernels should be instantiated in the programmable logic (PL). This is the execution model. The memory model is responsible for allocating memory and transfer the data over PCIe. Finally, the programming models maps the kernels to different CUs on the accelerator card and initiates the computation.

From an application developer's perspective, OpenCL consists of two parts. The kernel function is usually a computationally intensive task and is written in a C-like language called OpenCL C. The kernel is executed on the device. The host program runs on a conventional CPU and uses the OpenCL API to interact with the kernel. In the following sections we discuss the OpenCL API and the kernel function in detail.

2.1.1 OpenCL API

The host program uses the OpenCL API to setup the environment and to manage the execution of kernels on the devices. The OpenCL API is defined in a C header file (`opencl.h`). There is also a C++ API available (`cl.hpp`) that wraps the C API and is easier to use. The header file is usually provided by the vendor-specific implementation of the OpenCL API. Each implementation supports only devices that a vendor knows how to interact with. For example, the Intel FPGA SDK for OpenCL can interact with Intel FPGAs but not with FPGAs from Xilinx. The OpenCL Installable Client Driver (ICD) allows implementations from multiple vendors to coexist on a system [41].

The OpenCL API is divided into two layers. The *platform layer* is used to discover platform capabilities and devices. It is also used to setup the execution environment. The *runtime layer* maps the kernels onto devices and manages the memory.

Platform Layer

A developer can use the platform layer to discover platform capabilities and devices. The platform and devices are managed in a context. The context coordinates the host-device interaction. Additionally, the context manages memory objects and keeps track of which kernel is executed in each device. The code for creating a context is normally written once and can be reused for all projects that run on the same hardware. Before creating a context, the developer must obtain information about the platform and the devices on the system.

Listing 2.1 shows sample code to create a context. This example is written in C++ because we used C++ for our benchmark and test programs.

Listing 2.1: Example code for using the platform layer

```
#define CL_HPP_CL_1_2_DEFAULT_BUILD
#define CL_HPP_ENABLE_EXCEPTIONS

#include <vector>
#include <iostream>
#include <CL/opencl.hpp>

int main()
{
    std::vector<cl::Platform> platforms;
    cl::Platform::get(&platforms);
    cl::Platform platform = platforms.front();

    std::vector<cl::Device> devices;
    platform.getDevices(CL_DEVICE_TYPE_ACCELERATOR,
                                    &devices);
    cl::Device device = devices.front();

    cl::Context context(device);

    ...
```

Initially, the program defines two constants. The first constant sets the OpenCL version (OpenCL 1.2) and the other constant enables exceptions, which simplifies error handling. Then, the code includes the OpenCL header file.

In the `main` function, the program queries the available platforms on the system. Usually, there is only one platform installed but the program could also filter the platforms by vendor name or other properties. In this simple example, we just pick the first platform.

Next, the platform tries to get a list of devices and picks again the first available device for simplicity. We can query different types of devices. Here we ask for accelerators, but we could also filter for other device types like CPUs or GPUs.

Once the program has a reference to a device, it can create a context that includes the platform and the device we have created previously. The context can now be passed to the runtime layer to execute a kernel.

Runtime Layer

The host application uses the runtime layer to interact with the kernel that runs on a device. OpenCL objects such as memory, program or kernel objects are created within a context. The host submits commands to a command queue to request actions by the device.

Listing 2.2 continues after the previous code and shows all the necessary steps to create a kernel.

Listing 2.2: Example code for creating a kernel

```
...

cl::CommandQueue q(context, device);

char *file = ...
int fileSize = ...
cl::Program::Binaries bins{{file, fileSize}};
cl::Program program(context, device, bins);

cl::Kernel kernel(program, "vadd");

...
```

The program creates a command queue with the context and the device from the previous listing. There are also two flags that can be passed to the command queue constructor:

`CL_QUEUE_PROFILING_ENABLE` Enables profiling for this command queue. Profiling collects additional information, that can be used to analyze the execution of a program.

`CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` Enables the out-of-order execution mode. The host application can run several copies of the same kernel or multiple different kernels on a device. There are two approaches to execute multiple kernels with optimal performance. With a single out-of-order queue, multiple kernel executions can be requested through the same command queue. The kernels are executed concurrently and in any order. With multiple in-order queues, each kernel execution is requested by a different command queue. The kernels are dispatched from different queues and concurrently executed.

Next, we create a program from which we can obtain a reference to a kernel. A program is a collection of kernels that are identified by the `__kernel` qualifier in OpenCL C code. A program can be generated *online* or *offline*. The first method compiles the program at runtime. This allows the compiler to apply device specific optimizations that could increase performance. However,

runtime compilation is not suitable for FPGAs because the compilation takes a long time. Thus, many vendors use a separate tool chain to create the FPGA binary. The binary is then loaded into memory and passed to the program.

In our example, we used the second approach and loaded a binary into memory. This binary is then, together with the context and a device, passed to the `cl::Program` constructor to create a program. We do not show an example for the online program generation because Vitis does not support this method.

The kernel is created by providing a reference to a program and the name of the kernel (`vadd` in our example). The name must match with the label of the `__kernel` identifier. At that point, we have a reference to a kernel and can start with the kernel execution.

Kernels often access large arrays or images to run a computation on that data. In many systems, the host and a device do not share a common address space, so the data needs to be transferred to the device before the computation can start. The OpenCL runtime manages the data transfers and dependencies transparently for the programmer. OpenCL uses the concept of *memory objects* as a reference to the data that is required by the kernel. A memory object is only valid in a single context so it cannot be shared between different contexts. There are two types of memory objects: *buffers* and *images*. Buffers are similar to C arrays and the values are laid out contiguously in memory. Images allow for certain performance optimizations, but we do not use images in this thesis.

Listing 2.3 allocates memory and executes a kernel. This example has an input and an output buffer that are used to transfer data from and to the kernel.

Listing 2.3: Example code that allocates memory and executes a kernel

```
...

int size = 32;
std::vector<int> in(size);
std::vector<int> out(size);
std::fill(in.begin(), in.end(), 42);
std::fill(out.begin(), out.end(), 0);

int size_in_bytes = sizeof(int) * size;
cl::Buffer in_buf(context, CL_MEM_READ_ONLY,
                                size_in_bytes);
cl::Buffer out_buf(context, CL_MEM_WRITE_ONLY,
                                size_in_bytes);
```

```
kernel.setArg(0, in_buf);
kernel.setArg(1, out_buf);
kernel.setArg(2, size);

q.enqueueWriteBuffer(in_buf, CL_TRUE, 0,
                     size_in_bytes, in.data());

q.enqueueTask(kernel);
q.finish();

q.enqueueReadBuffer(out_buf, CL_TRUE, 0,
                     size_in_bytes, out.data());

verify_result(&in, &out);

...
```

Initially, the program creates two vectors and fills them with arbitrary values. Then, we create two buffer object that act as a reference to the underlying memory. We can pass a number of flags to the constructor of `cl::Buffer` to control the allocation of the buffer. For example, the `CL_MEM_READ_ONLY` flag can be used to create a read-only buffer.

Next, we associate the buffers with the arguments of the kernel by calling `setArg` on the kernel object. The first argument specifies the argument index. The other argument can either be a scalar value or a reference to a memory object. Scalar values are used for small data transfers, such as constants or configuration data. They are passed as input arguments to the kernel and the host application can only write to such an argument. Memory objects are used for large data transfers and the host application can both read and write to these argument.

OpenCL provides different functions to transfer data between the host and a device. Typically, the functions `enqueueReadBuffer` and `enqueueWriteBuffer` are used for simple applications with no specific access patterns or performance requirements. We also used these function in our code. For example, the call to `enqueueReadBuffer` transfers the values from the `in` vector to the input buffer. The second argument, where we passed `CL_TRUE`, specifies if the function is blocking or non-blocking.

There are other functions to transfer data in OpenCL. Xilinx, for example, recommends to use `enqueueMigrateMemObjects` for performance reasons. This function allows to preemptively allocate memory, so the runtime can overlap memory operations with other unrelated operations. This can potentially hide memory latencies and increase performance.

Another important function is `enqueueMapBuffer`. This function maps a buffer into the host address space and returns a reference to that mapped region. The host application can use this reference like a normal pointer to access data in the buffer. Then, the host applications calls `enqueueMigrate MemObjects` to transfer the buffer to the device.

Once the data has been successfully moved to the kernel, we can start the execution with `enqueueTask`. Many host applications, especially in GPU acceleration, use the function `enqueueNDRangeKernel` to start the kernel. This function spawns multiple executions of a kernel to parallelize the work. On FPGAs, however, the parallelism happens inside the hardware. Thus, a single monolithic kernel has often a better performance than many small kernels.

The `enqueueTask` function is non-blocking, so we have to call `clFinish` before we verify the data. `clFinish` blocks until all previous commands in the command queue have completed. Once the kernel has finished its computation, we can read the result from the output buffer and verify it.

This concludes our discussion of the OpenCL API. The complete source code of our example application can be found in Appendix B.1. The next section discusses OpenCL C, a programming language to create OpenCL kernels.

2.1.2 OpenCL C

OpenCL C is a programming language to write OpenCL kernels. The language is based on C, but has some restrictions in order to run efficiently on accelerator hardware. Instead of the `main` function, the entry point is marked with `__kernel`. The host application can refer to this attribute to extract a kernel from a program. Memory pointers are annotated with different region qualifiers. For example, the `__global` attribute indicates that the data is stored in global memory. Xilinx added additional attributes to OpenCL C, that can be used to give optimization hints for the high-level synthesis.

Listing 2.4 shows a simple kernel written in OpenCL C. The kernel adds a constant to an input vector and writes the result to an output vector. This example has only one kernel, which is marked with the `__kernel` attribute. The memory attributes are also present. Both vectors are in global memory whereas the vector size resides in the constant section of the global memory. There is also a custom attribute from Xilinx, which gives a hint for pipelining the loop.

Listing 2.4: OpenCL C kernel

```
__kernel void vadd(
    __global int *in,
    __global int *out,
    __constant int size
```

```
) {
    __attribute__((xcl_pipeline_loop(1)))
    for (int i = 0; i < size; i++) {
        out[i] = in[i] + 42;
    }
}
```

2.2 Enzian

Enzian is a cache-coherent asymmetric NUMA system with two nodes. One node is a 48-core ThunderX CPU from Marvell, which has 128 GiB DDR4 memory attached. The other node is a large Xilinx FPGA with access to 512 GiB DDR4 memory. The two nodes are connected over the Enzian Coherent Interconnect (ECI). ECI is based on the cache coherence protocol of the Marvell CPU, which allows fast communication between multiple CPU nodes. The FPGA emulates the coherence protocol and acts as an additional CPU node. Thus, ECI enables a fine-grained interaction between the FPGA and the CPU.

Figure 2.1 shows the block diagram of Enzian. The box on the left side represents the CPU node with the ThunderX processor and four DDR4 slots. This node has access other interfaces such as PCIe, NVMe or QSFP for network transceivers. The node on the left hand side contains the FPGA

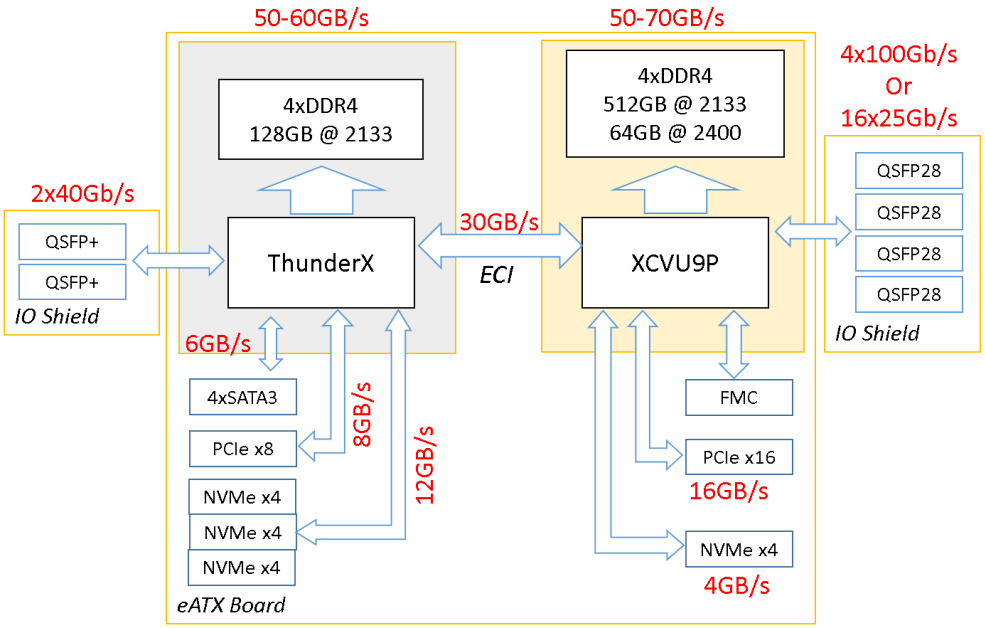

Figure 2.1: Enzian Block Diagram [1]

(XCVU9P) and again four DDR4 slots. The interfaces of this node are similar to the ones attached to the CPU node. ECI connects the two nodes and is shown in the middle of the diagram.

The cache-coherence protocol splits a shared address space between multiple nodes. If one node accesses a memory location from another node, ECI propagates the changes over the interconnect. Besides cache-coherent memory access, the protocol supports I/O registers and inter-processor interrupts.

There are two modules for ECI on the FPGA side, Enzian DMA or Enzian Directory Controller (DirC) [42]. Enzian DMA is a module that provides direct access to the CPU DDR memory. The module takes an address and a size as input and returns the data over an AXI-Stream interface. Writes are handled similarly. Enzian DirC, on the other hand, implements a directory controller with three interfaces. The first interface is attached to the CPU, the second interface is connected to the FPGA and the third interface can be attached to any byte-addressable memory, such as the FPGA DDR. The directory controller ensures that all memory changes are propagated to the correct interfaces. The layers below these modules are not relevant for this thesis, so we do not cover them. Enzian did not support inter-processor interrupts at the time of writing.

On the CPU side, Enzian runs Ubuntu 20.04 LTS with a custom Linux kernel. We can access the FPGA from the CPU by mapping the FPGA address range into our program, for example, a kernel module. As mentioned earlier, the address space is split between the CPU and the FPGA. ECI uses two bits (bit 40 and 41) of the address to specify the index of a node. The CPU has index zero, and the FPGA has index one. Listing 2.5 shows the address map of Enzian.

Listing 2.5: Enzian address map

```
// CPU memory
0x0000000000000000 - 0x000000fffffffff
// FPGA memory
0x0000010000000000 - 0x000001fffffffff
```

The first address range is reserved for the CPU DDR memory. The second address range is for the DDR on the FPGA. ECI propagates reads or writes to these addresses to the other node and ensures cache-coherency.

2.3 Xilinx Vitis

Vitis is a framework that integrates all Vitis software development into one unified environment. Before Vitis was introduced, two separate software development workflows existed. The first workflow was for embedded

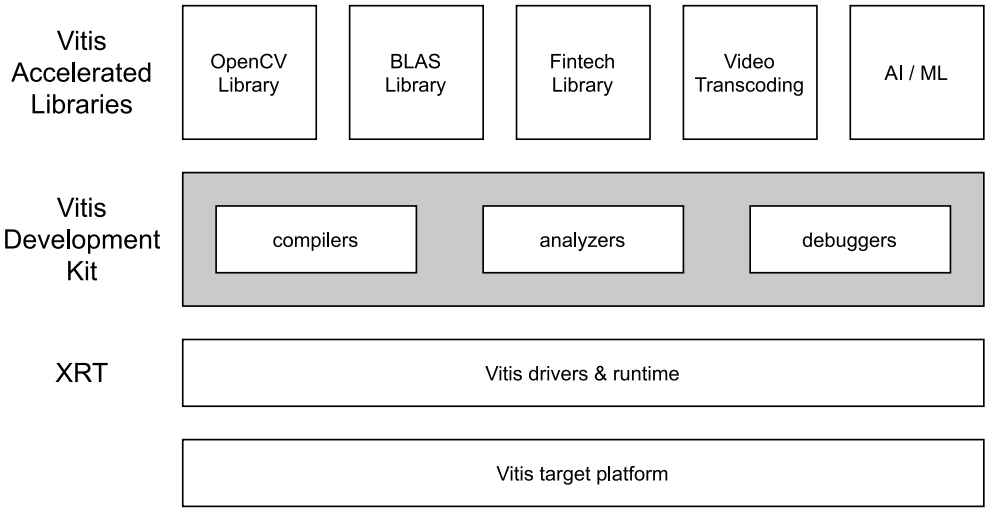

Figure 2.2: Vitis Architecture

software development and targeted embedded devices such as the Zynq-7000 SoC or the Zynq UltraScale+ MPSoC. Embedded developers used the Xilinx Software Development Kit (SDK) to create programs for embedded devices. The other workflow was for developing accelerated applications that run on Alveo accelerator cards. A tool called SDAccel was used to create these applications. These two workflows still exist in Vitis but redundant parts were removed.

In this thesis, the focus is on the application acceleration development flow and the Xilinx Runtime (XRT), as this is the place where the OpenCL API is implemented. The Vitis application acceleration development flow is a framework to develop and deploy FPGA accelerated applications using well-known programming languages for both the hardware and software components. The software component (host program) is developed using C or C++ and runs on a x86 or ARM processor. The hardware component (kernel) is developed in C/C++, OpenCL C or RTL and runs on the FPGA. OpenCL API calls are used by the host program to interact with the kernel over XRT.

Figure 2.2 shows the overall architecture of Vitis. The lowest layer is the Vitis target platform which is shown at the bottom. A platform defines various aspects about the system on which the application is running. The FPGA shell is also part of the platform and provides a interface for the Vitis runtime to communicate with the kernel on the FPGA fabric. A layer above the platform is the Xilinx Runtime (XRT). XRT provides an API and drivers for the host application to interact with the platform. XRT also handles transactions between the host and the accelerated kernels. Next, the Vitis development kit provides the software development stack. Besides compilers and cross-

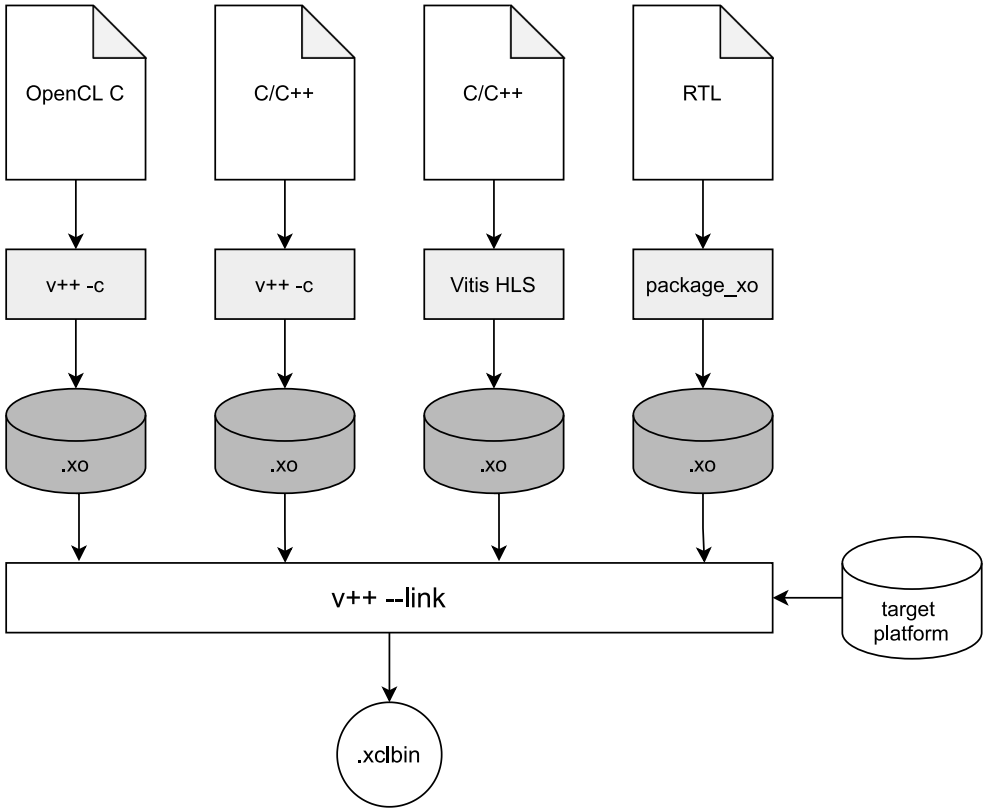

Figure 2.3: Vitis device build process

compilers, the development kit contains analyzers to profile an application and measure its performance. Debugging and emulation facilities help to locate and fix issues in the application. At the highest level, Vitis provides various libraries for existing algorithms that are performance-optimized for FPGA acceleration. An application can reuse these libraries with minimal code changes.

The Vitis build process has two separate workflows. The device binary is built by the Vitis compiler and the GNU compiler collection (GCC) is used to create the host binary. The following two sections discuss the build process in detail.

2.3.1 Device Binary

The device binary is created with the Vitis compiler which uses similar concepts as GCC. Similar to GCC, the Vitis compiler creates object files from the source code and links the object files to an executable. Figure 2.3 shows the complete build process. Kernels are developed in either C/C++, OpenCL C or RTL. Then each kernel, is individually compiled into a Xilinx object file

(*.xo). Various tools are available to create an object file. Kernels developed in C/C++ or OpenCL C are compiled directly with the Vitis compiler (v++). Developers who prefer a graphical interface can work with Vitis HLS IDE and export an *.xo file from there. Hardware developers can develop a kernel in Vivado in RTL and use the package_xo Tcl command to export an object file. The object files are all linked with the target platform using the v++ --link command. The Vitis linker creates a FPGA binary file (*.xclbin) which must be passed to XRT to run the accelerated application.

Listing 2.6 shows an example command to create a Xilinx object file using the Vitis compiler.

Listing 2.6: Example command to create a Xilinx object file

```
v++ -c -t sw_emu --platform xilinx_u200_xdma_201830_2
    -k vadd -o vadd.xo ./src/vadd.cpp
```

The example passes a number of arguments to the compiler. The first argument (-c) tells the compiler to compile an object file. There is also a flag (-l) for linking, which we discuss in the next section. The second argument (-t) defines the build target. The build target can be sw_emu, hw_emu or hw. The first two targets are for simulation and the third target creates an actual device binary. The --platform flag defines the device for which the object file is built. In this example, we compile for a Alveo u200 accelerator card. The last three arguments define the kernel name, the output file and the source file. Executing this command creates a vadd.xo file that can be passed to the linker.

A Xilinx object file is a ordinary ZIP file, that contains an IP and some metadata. The Vitis compiler takes the C/C++ or OpenCL C source files and passes them to Vitis HLS. Vitis HLS uses high-level synthesis to convert the C/C++ functions into RTL and creates an IP from the synthesised code. In addition, Vitis HLS creates metadata which describes for example the ports of the IP. The IP and the metadata is used by the Vitis linker to create the final design of the system.

Listing 2.7 shows an example command to link the object files with the target platform.

Listing 2.7: Example command to link object files and a platform

```
v++ -l -t sw_emu --platform xilinx_u200_xdma_201830_2
    -o vadd.xclbin vadd.xo
```

The arguments for linking are almost the same as for compiling. We can pass multiple object files to the linker and they get all linked with the platform to create a final design for the FPGA. The linker determines important architectural details during the linking phase. In particular, this is where the

number of Compute Units (CUs) to instantiate in hardware is specified, the CUs are assigned to Super Logic Regions (SLRs) and where the kernel ports are connected to memory interfaces. The Vitis compiler provides a variety of options to customize the linking phase which we do not cover in this section. We refer the interested reader to the Vitis compiler command reference [59] for a complete list of all options.

2.3.2 Host Binary

The host program is written in C/C++ and uses OpenCL API calls to interact with the kernels on the device. The host binary is built either with the GNU C++ compiler for x86 based systems or with the GNU C++ ARM cross-compiler for embedded devices. Each source file is compiled to an object file (`*.o`) and is then linked with shared libraries from XRT to create an executable. The XRT shared libraries provide the implementation for the OpenCL API. The following listing shows an example command to compile and link a host application.

```
g++ -Wall -g -std=c++11 host.cpp -o host \
             -I${XILINX_XRT}/include/     \
             -L${XILINX_XRT}/lib/         \
             -lOpenCL -lpthread -lrt -lstdc++
```

As this example shows, compiling and linking follows the normal g++ workflow. The environment variable `XILINX_XRT` points to the installation directory of XRT and is used to direct g++ to the header files and shared libraries that XRT provides.

There is also the possibility to cross-compile the host application for embedded systems with an ARM CPU. In this thesis, we compile the host program directly on Enzian so we skip cross-compiling.

2.3.3 Xilinx Runtime (XRT)

The Xilinx Runtime (XRT) is the runtime environment for Vitis applications. XRT provides user space libraries and Linux kernel drivers to interact with FPGAs on accelerator cards or in embedded systems. XRT is open source and the code is available on Github [65].

Figure 2.4 shows a block diagram of the XRT architecture. At the top layer, we have the host application, that was linked with the XRT shared libraries. The host application uses several user space libraries from the red box, to manage the kernels on the FPGA device. In addition, the user space layer includes command line tools for end-users and system administrators. The Linux kernel drivers in the orange box export an API for the user space libraries, that controls access to the devices. The blue layer below the kernel

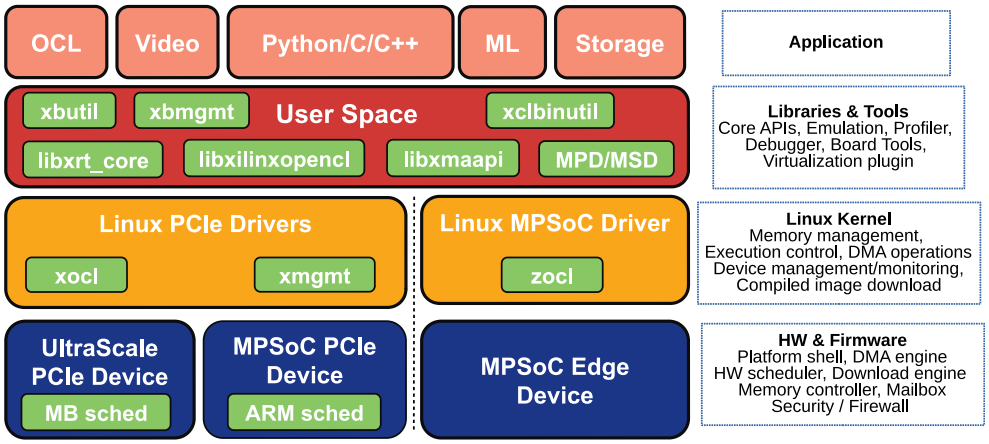

Figure 2.4: XRT architecture [64]

provides software that runs directly on the devices. For example on the Alveo cards, the scheduler for managing the compute units is executed on the internal Microblaze CPU. In the following sections, we discuss each layer in detail.

User Space Libraries

XRT provides four user space libraries for the host application. The first library is `libxrt_core` which exports a C/C++ API. The API is defined in the header file `xrt.h`. This library gives access to every functionality of XRT and is tailored for developers that require full control over the runtime. However, application that use the `libxrt_core` library run only on accelerators supported by XRT and are not portable. Applications that should run on multiple accelerator platforms should use the OpenCL API instead. XRT implements the complete OpenCL 1.2 API in the `libxilinxopencl` library. XRT provides an Installable Client Driver (ICD) so the OpenCL runtime is able to locate the vendor implementation and redirect API calls appropriately.

Command Line Tools

XRT has three command line tools that can be called from user space. The first two tools, `xbutil` and `xbmgmt`, are used for device management. They support both the Alveo cards and embedded systems. The FPGA shell on the device is divided into user functions and management functions to provide different security levels. The user functions allow end users to load and run their applications, while management functions are used by administrators to manage the device. The `xbutil` command provides the user functions. The `xbmgmt` utility, which must run as root, is used for management functions.

These two security levels are particularly useful in cloud environments. The third tools is `xclbinutil` and it us used to query or manipulate the `xclbin` file.

Linux Kernel Drivers

Xilinx provides two different Linux drivers for their platforms. The first driver is for PCIe based platforms and is divided into two modules called `xocl` and `xmgmt`. The second driver is called `zocl` and supports edge devices. In a edge device, the CPU and the FPGA are connected over an AXI bus. We used only the `zocl` driver in this thesis, so we do not cover `xocl`.

The `zocl` driver supports devices like the ZYNQ-7000 and ZYNQ Ultrascale+ MPSoC. In addition, users can create their own hardware platforms, which are also supported by the driver. The edge driver is not as sophisticated as the PCIe driver because it does not include the security features provided by `xocl` and `xclmgmt`. The CPU and FPGA are connected over an AXI bus. Data is either transferred over a shared memory such as the CPU DRAM to which both the CPU and the FPGA have access or by using a memory management unit (MMU) that provides shared virtual memory (SVM).

The Linux edge driver has just one module called `zocl`. This module does memory allocation, programs the MMU and has an internal scheduler. Additionally, `zocl` provides facilities to upload a FPGA binary to the device.

In the following sections, we discuss common concepts for accelerator drivers and show how each driver type implements these concepts.

Image Download The user application comes as a `*.xclbin` container file and needs to be downloaded to the FPGA. This container has a FPGA bitstream and metadata about the platform. The metadata is divided into several sections which define the memory topologies, the IPs and other specifications of the system. The format of the xclbin file is defined in the `xclbin.h` header file. For PCIe based platforms, `xclmgmt` has an ioctl system call to load a xclbin file. The same system call is supported by `zocl` for edge devices.

Both drivers extract the bitstream and program the FPGA using the Linux FPGA manager API. After programming the device, the driver processes each section of the xclbin file. Based on the sections in the xclbin file, the driver initializes the memory manager and instantiate the compute units. After this procedure, the driver is ready to receive commands from the host application.

Memory Management XRT exposes a high-level API to user space for memory management. The central concept of this API is a Buffer Object (BO). A

BO hides all the complexity that is involved in managing memory in heterogeneous systems with multiple address spaces. A developer can simply allocate a buffer and transfer data between the host and a FPGA device. The user space library forwards all memory requests via ioctls to the kernel driver.

The actual memory management happens inside the kernel. XRT must support a variety of memory topologies. For instance, on PCIe based systems a developer can initialise multiple memory controllers as hard IPs on the FPGA, each with its own address space. Usually, device memory is not exposed to the CPU, so XRT uses host memory pages to back the device memory to support, for example, mmap calls. The host and device memory are kept in sync with a PCIe DMA engine. Another example are edge devices, where the CPU and the FPGA usually share the same memory. In addition, edge devices can instantiate soft memory controller directly in the FPGA fabric or configure the ARM SMMU for SVM.

To support all these memory topologies, XRT uses the Linux DRM framework [8]. Initially developed for GPU drivers, many concepts of DRM can be reused for accelerators, especially the memory management. XRT uses two different memory allocators from DRM. The `drm_mm` allocator is used to reserve memory on the FPGA. To allocate memory in the CPU DRAM, XRT uses the Continuous Memory Allocator (CMA). The Linux DRM framework hides a lot of the complexity of DMA transfers and coherency.

Scheduler XRT has a dedicated component, called the Kernel Domain Scheduler (KDS), that is responsible for scheduling. KDS is part of the driver on edge devices and runs in kernel space. On PCIe based systems, KDS is a separate program that runs on the Microblaze CPU in the FPGA. The KDS was probably moved to the Microblaze to increase performance.

XRT uses the concept of a Compute Unit for scheduling. A CU is an instance of a kernel. The abstraction of a CU is necessary because an application can have more than one instance of a kernel. In OpenCL a device is divided into several compute units, and compute units have several processing elements. In that regard, the OpenCL and XRT definition of a CU agree. However, XRT does not use processing elements.

The xclbin file defines the number of CUs in an program. The driver parses the xclbin file and registers a new sub device for each CU in the Linux kernel. The user space library submits execution requests over ioctl system calls. The requests are defined in the file `ert.h`. KDS receives these requests and runs the requested CUs if possible.

KDS controls a CU over a set of registers. These registers can be accessed over an AXI-Lite port, that is connected to the kernel. The first couple of

registers are the same for all kernels. The first register is usually reserved for control signals. The scheduler uses these signals to start the kernel or check the status of an execution. Then follow various registers to control the interrupts. After this static section, each kernel has registers to control the arguments of the kernel function. A kernel argument can either be a scaler value or a pointer to an array. Scalar values are directly written into the register of the respective argument. For array arguments, XRT writes the start address of the array to the register and the kernel fetches the data over AXI using this address.

As described in the previous section, the first register has various control signals. These signals may change with the kernel execution model. A kernel can have three execution models.

`AP_CTRL_HS` This model and has two control bits, `ap_start` and `ap_done`. KDS writes a one to `ap_start` to start the kernel. Then, KDS waits until `ap_done` is set by the kernel. After processing the result of the kernel computation, KDS can start another kernel run. As you can see, this model supports only sequential execution so there is, for example, no way to hide a data movement latency. For that reason, this model is now deprecated.

`AP_CTRL_CHAIN` This is the default model for Vitis kernels. The model separates input and output synchronization. With this separation, a new kernel execution can be started, even if the previous execution is still running. This allows a pipelined execution model that can achieve better performance. The kernel inputs are controlled with the register bits `ap_start` and `ap_ready`. KDS writes the memory addresses to the correct registers and sets `ap_start` to one. If `ap_ready` is one, the kernel is ready to receive data for the next kernel execution, even if the previous run has not been finished. The scheduler can now write the new memory addresses and start the kernel again. The outputs are synchronized with the `ap_done` and `ap_continue` bits. If the kernel has a result ready, the bit `ap_done` is set to one. XRT can now read the result from the output of the kernel. If XRT is ready for the next result, it sets `ap_continue` to one and the kernel provides the result of the next execution. To benefit from this pipelined execution, the host code must use the correct API calls, for example, the `clEnqueueMigrateMemObjects` OpenCL function to move data ahead of kernel execution.

`AP_CTRL_NONE` This model does not have any control signals and the kernel is running as soon as input data is available (*free-runnning*). This mode is particularly useful in combination with streaming interfaces.

To monitor the kernel execution, KDS can either actively poll the `ap_done` signal or use an interrupt to receive a notification from the kernel. KDS noti-

fies the user space library asynchronously over the POSIX poll mechanisms when the kernel finished its execution.

Hardware Layer

The lowest layer of the XRT stack, is the hardware layer. This layer contains hardware definitions, FPGA shells and device firmware. Although the XRT code is available on Github, most of the hardware layer consists of proprietary IPs and binary files, so it is hard to understand for the end-user. Anyway, the hardware definition and FPGA shell for a device are normally packed into a *platform*. Users have the possibility to create their own platforms, that are supported by XRT. See chapter 3.1 for a detailed discussion about custom platforms.

2.4 AXI

The Advanced eXtensible Interface (AXI) is a protocol for on-chip communication. Initially developed by ARM, the protocol is now widely used by various chip manufacturers, including Xilinx. Many Xilinx IP cores provide an AXI interface. Using AXI, different IPs can easily communicate with each other. This section covers the basics of the AXI4 protocol to provide the background for the platform creation described in Section 3.1. For details, we refer to the AXI4 specification [5].

The AXI protocol describes a communication standard between a master and a slave interface. The standard defines the following channels:

- read address
- read data
- write address
- write data
- write response

Figure 2.5 shows the channel architecture of a read transaction. Initially, the master interface puts an address and control information about the transfer on the read address channel. Then the slave interface sends the requested data back over the read data channel. Figure 2.6 shows the channel architecture of a write transaction. The master interface writes data by describing the transfer on the write address channel. Then the master puts the data to write on the write data channel. Finally, the slave interface informs the master interface about the transaction status over the write response channel.

AXI defines a simple handshake protocol that uses the signals `xVALID` and `xREADY`. The sender drives the `xVALID` signal to notify the receiver that the

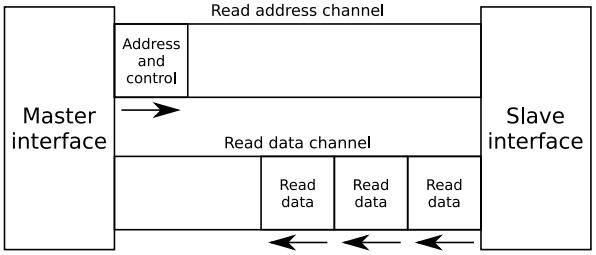

Figure 2.5: AXI read channels [49]

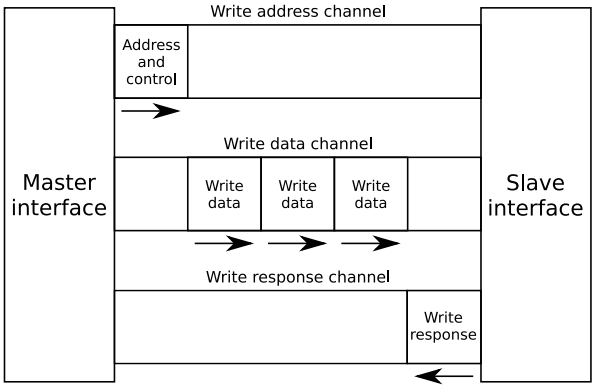

Figure 2.6: AXI write channels [49]

payload on the channel is valid and can be read on the next clock cycle. The receiver sets the `xREADY` signal when he is ready to receive the data. When both the `xVALID` and `xREADY` are high in the same clock cycle, the payload is transferred. The sender can now keep the `xVALID` signal high to send more data or terminate the session by driving `xVALID` low. Using this simple mechanism, both the sender and the receiver can control the flow of the data.

An individual data transfer is called a *beat*. The size of a beat is defined by the *data width* of the interface, that is, the number of wires for the `xDATA` signal. For example, an interface with a data width of 1024, transfers 1024 bits or 128 bytes per beat and has 1024 wires for the `xDATA` signal. Another important parameter is the *address width* of an interface. The address width defines the address range that an interface can access. For example, an interface with an address width of two can access the addresses `0x0` to `0x3`. Two connected AXI interfaces must have the same data and address width to work correctly. In some cases, we can use converter IPs to connect interfaces with different parameters.

There are two other variants of AXI: AXI-Lite and AXI Stream. AXI-Lite is a subset of the AXI protocol with reduced features and complexity. In AXI-Lite, all bursts have only one beat, and all transfers use the full data width, which can be either 32 or 64 bits. These simplifications allow AXI-Lite to remove

a part of the signal and provide a register-based interface for reading and writing to a slave device. AXI Stream is another AXI flavor that is not relevant in this thesis.

Chapter 3

Implementation

The OpenCL API is specified in a set of headers and requires a vendor-specific implementation of the complete API. OpenCL applications are not directly linked against a specific implementation. Instead, a developer links an OpenCL application against an Installable Client Driver (ICD) loader. This mechanism allows the installation of multiple OpenCL implementations on the same system. The ICD loader exports the OpenCL API entry points and redirects the calls to the correct implementation. The ICD is the actual OpenCL implementation that implements each function of the API.

With this in mind, we need the following components to add OpenCL support to Enzian:

- OpenCL API header files
- ICD loader
- Installable Client Driver (ICD)

The OpenCL header files are available on Github [14] or as a Linux package (`opencl-headers`). Multiple ICD loaders [15] [11] are available as a Linux package, so we can easily install them on Enzian. For the Installable Client Driver we have two options. We could either implement our own ICD or adapt an existing one. The first option implies that we implement every OpenCL API call. This is by no means trivial, if not impossible, in the time frame of this thesis. Therefore, we decided to adapt an existing ICD.

There are many ICDs available, that can be divided into two categories. The first category contains ICDs developed by device manufacturers. For example, Xilinx has an OpenCL implementation for their FPGAs as part of the Vitis framework. These ICDs support only devices from a specific vendor. The other category includes community-developed ICDs, which support devices from various manufacturers. PoCL [25], for example, is an open-source implementation of the OpenCL standard that supports GPUs

and CPUs from different vendors. As these ICDs support many devices, they often lack support for vendor-specific features.

In this thesis, we evaluated two OpenCL implementations, one for each of the above categories. From the vendor category, we chose the OpenCL implementation from Xilinx, because Enzian has a Xilinx FPGA. From all the community-driven projects, only PoCL seemed to be actively maintained. Thus, we selected PoCL as the candidate for the other category.

Xilinx provides an OpenCL implementation as a part of their Vitis framework. Vitis has components covering both the host part (CPU) and the device part (FPGA) of OpenCL. The Xilinx Runtime (XRT) is responsible for the host part and implements the complete OpenCL API. The Vitis compiler (`v++`) covers the device part. This compiler takes a kernel, written in OpenCL C, as input and creates a bitstream for the FPGA by using high-level synthesis (HLS). Besides these basic functionalities, Vitis has various tools to debug and profile accelerated applications. All these features can also be a disadvantage, because they make Vitis a complex piece of software. Thus, it could be hard to understand and adapt Vitis to the Enzian platform.

PoCL, on the other hand, is not as sophisticated as Vitis, because it only provides the host part of OpenCL. That is, it handles the OpenCL API calls and uses a hardware abstraction layer to interact with the FPGA. Although PoCL provides an implementation of the OpenCL API, we must still implement the hardware abstraction layer. For the device part, we need a separate tool to convert an OpenCL kernel into a bitstream for a Xilinx FPGA. This process is usually divided into two steps. First, an HLS tool compiles a kernel written in a high-level language like C/C++ into Verilog or VHDL. Then, vendor-specific tools are used to generate a bitstream from these HDL files. In our case, we would use Vivado to generate a bitstream for the FPGA on Enzian.

A quick survey of HLS tools revealed that only the HLS compilers from Intel and Xilinx support OpenCL C. We could use the Intel compiler in combination with Vivado to create a bitstream, but we consider this impractical. Thus, the only viable option for the device part is the Xilinx HLS compiler. To summarize this approach, we would use PoCL for the host part of OpenCL and the Xilinx HLS compiler for the device part.

After comparing both OpenCL implementations, we decided to adapt the Xilinx Vitis framework for Enzian. The main reason for this decision was that PoCL does not provide the device part of OpenCL. To cover this part, we still needed the Xilinx HLS compiler. We did not find any related work which combines PoCL with the Xilinx HLS compiler. Therefore, we preferred the Vitis framework that combines both the host and the device part in one tool chain.

In the following sections, we cover the adaption the Vitis framework to the Enzian platform. We start with creating an FPGA shell, or platform, that is compatible with Vitis. Then, we adapt the Vitis build tools so they produce a host and a device binary for Enzian. Finally, we change the Xilinx Runtime (XRT) so it can communicate with the Enzian FPGA.

3.1 FPGA Shell / Platform

Vitis creates executables for many different devices using the same application code base. These systems often have not much in common except that a CPU is connected to an FPGA. For example, in Alveo systems, the host application runs on an x86 CPU, and the CPU accesses the accelerator card through PCIe. Zynq devices, on the other hand, use an ARM processor, and the FPGA is connected over the on-chip interconnect.

Consequently, Vitis requires detailed information about the system on which an accelerated application is running. A *platform* is a container that includes all this information. In addition, the platform contains the necessary infrastructure to run a kernel on the FPGA. This aspect of the platform is comparable to an FPGA shell.

A platform consists of a hardware and a software part. The hardware part is further divided into two sections. First, it contains the metadata of the FPGA shell, that is, a description of the resources that are available for the kernel. For example, the metadata defines what clocks are present or which AXI interfaces are available for data movement. Second, the hardware platform contains the IPs and other infrastructure necessary to run a kernel on the FPGA. This part is the implementation of the FPGA shell, and the kernel cannot modify these IPs. The hardware platform is created in Vivado and exported as a Xilinx Support Archive (XSA) file.

The software platform defines the runtime environment for the host application on the CPU. A software platform consists of one or several *domains*. A domain is either a Board Support Package (BSP) for an embedded system or an operating system with various software drivers. Usually, the software platform is used to generate SD card images for embedded devices. As we did not need these features for Enzian, we kept the software platform to a bare minimum.

The following two sections describe how we created the hardware and the software part of the Enzian platform. The Vitis compiler can then use this platform to create host and device binaries, which we discuss later.

3.1.1 Hardware Platform

The hardware platform is created with the Vivado Design Suite. The foundation of a hardware platform is a Vivado project which include a FPGA design for the device we want to support. Besides a working Vivado design, Vitis requires a set of base components that are available to the kernel. An example for such a component is an AXI interconnect that provides access to a shared memory. These components are then added to a metadata section of the hardware platform. The Vitis compiler parses the metadata and makes the correct connections between the kernel and the components.

The components and the metadata of the platform are configured in the Vivado IP integrator. With the IP integrator, a designer can create sophisticated FPGA designs by instantiating and connecting IPs on a design canvas. The output of the IP integrator is a *block design*. An HDL module in Vivado can include this block design like a regular IP. Vivado automatically creates an HDL wrapper, which can be instantiated in the HDL module. The block design uses external ports to define the interface of the HDL wrapper. For example, an AXI interface can be defined as an external port, and Vivado adds this interface to the HDL wrapper. The IP integrator also has settings to configure platform metadata, for example, to specify the default clock for the kernel. Vivado uses

The Vivado project for Enzian already has a block design to instantiate the Microblaze processor, which controls the ECI link. Initially, we created a different block design to keep the platform-related changes separate from the original design. However, Vitis supports only one block design per platform, so we had to add the platform IPs to the existing block design. This approach is suboptimal because it is unclear which IPs belong to ECI and which IPs belong to the platform.

Vitis requires the following components for a hardware platform [57]:

C1 A Processing System IP block representing the host part of the system

C2 An interface to control the kernel from the host

C3 An interface for data transfer

C4 One interrupt to signal the completion of a computation

C5 One clock source that drives the kernel

These components are the minimum for a working platform. Normally, a platform has, for example, multiple DDR channels and thus multiple interfaces for data transfer. We now explain how these components look in a hardware platform and how Vitis connects the platform and the kernel.

Figure 3.1 shows a platform with each of the above components. We can see the Processing System IP (**C1**), in the center of Figure 3.1. Then, the control

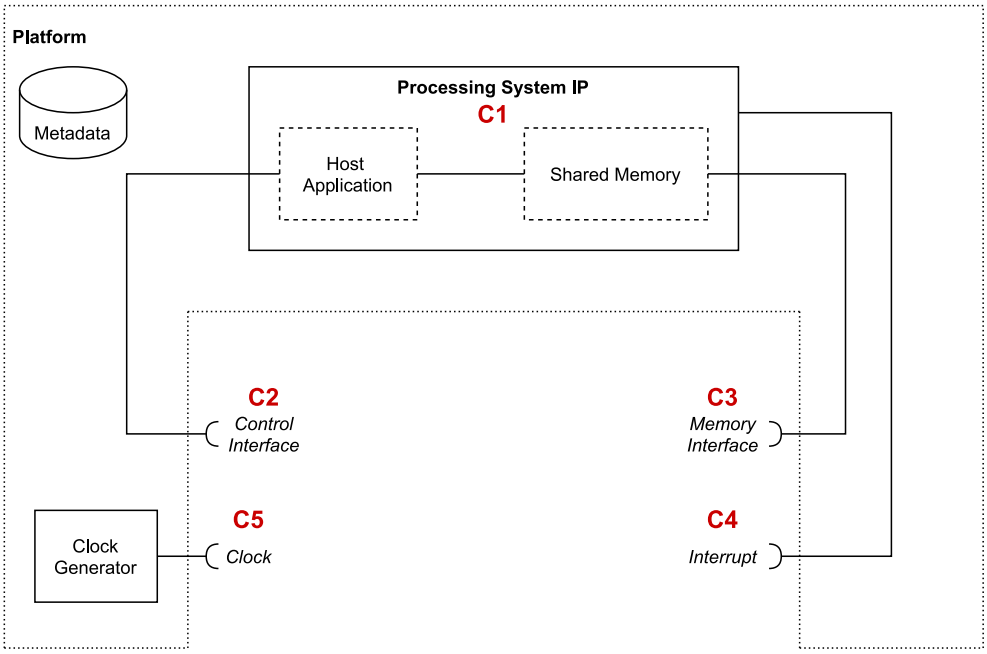

Figure 3.1: Simplified model of a platform

interface (**C2**) is connected with the host application. The host application can also access shared memory. Next, we see the connection between the memory interface (**C3**) and shared memory. The interrupts (**C4**) are also wired to the Processing System IP. The clock generator provides a clock source (**C5**) for the kernel. Finally, the platform contains metadata to describe all of the previous parts. The metadata is shown in the top left corner of Figure 3.1.

Now we examine the abstract model of a kernel, given in Figure 3.2. We notice that the interfaces of the kernel correspond to the interfaces of the platform in Figure 3.1. The kernel has also custom RTL logic, which does some computation. This logic is controlled by a set of registers, that the control interface can access.

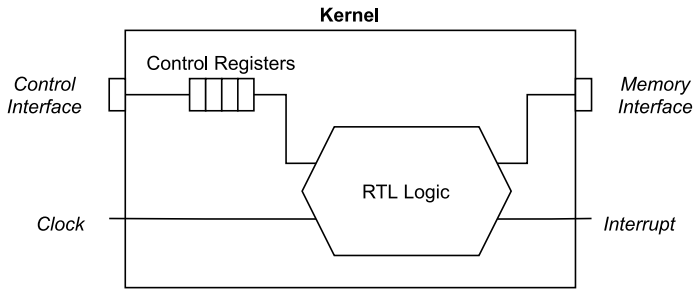

Figure 3.2: Abstract view of a kernel

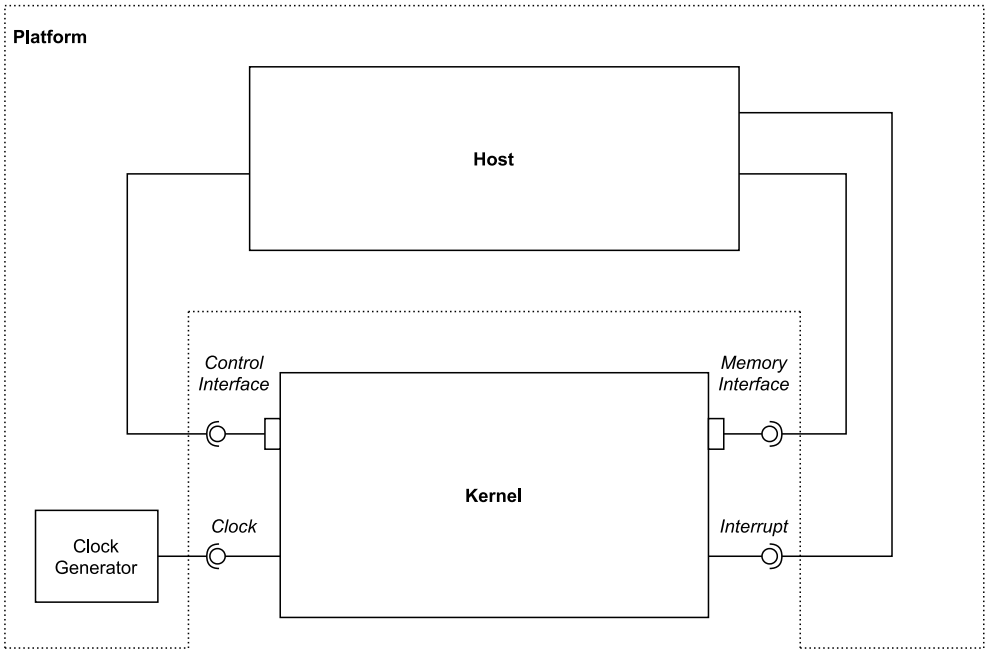

Figure 3.3: FPGA design created by the Vitis compiler

The Vitis linker takes a platform and a kernel as input and creates a FPGA design, as depicted in Figure 3.3. We can see that the Vitis compiler connected the interfaces of the platform with the interfaces of the kernel. Vitis can now create a bitstream from this design to program the FPGA. In the remaining part of this section, we cover each component (**C1**-**C5**) in detail and explain how we implemented them in Enzian.

C1 Processing System IP

A Processing System IP is used in designs that target the Zynq architecture. Zynq devices are SoCs that combine an ARM CPU, called the Processing System (PS), and an FPGA, or Programmable Logic (PL), on a single die. The Processing System IP is a wrapper to integrate the PS into a Vivado design. The IP provides various settings e.g. to configure the DDR controller or to adjust the PL clocks. The IP also has AXI interfaces that can be used to communicate with other IPs.

Figure 3.4 illustrates a Processing System IP with one AXI master interface (1), four AXI slave interfaces (2), and an interrupt line (3). The interfaces of the Processing System IP can be added to the platform metadata and can, for example, be used for data transfer. Therefore, it is straightforward to create a hardware platform for a device, if a Processing System IP already exists. However, Enzian is not supported by Xilinx, so there is no Processing System IP available.

Figure 3.4: Processing System IP block

To solve this problem, we use the fact that Vivado can add interfaces of an arbitrary IP to the platform metadata. We have two possibilities to get an IP with the necessary interfaces. First, we could create a custom IP for Enzian with an interface similar to a Processing System IP. This approach is better aligned to the platform creation instructions given by Xilinx, but the development cycle is slower as we work with two Vivado projects, one for the platform and one for the custom IP. Second, we could use existing IPs that already provide the necessary interfaces. For each IP, we add the necessary interfaces to the platform metadata, and we get similar a functionality as if we would use a Processing System IP. By using existing IPs, we can experiment with different designs in one Vivado project and have a quick feedback loop. We started with the later approach to create an initial prototype.

C2 Control Interface

A platform must expose at least one AXI-Lite master interface for controlling the kernel. Typically, a Processing System IP or an AXI interconnect provides this interface. The host can control the kernel over a set of registers, as we can see in Figure 3.2. For example, setting the first bit of register `0x0` to one starts the kernel execution. The kernel makes these registers accessible over an AXI-Lite port. Likewise, the AXI-Lite port from the platform can be used to set registers from an application on the CPU. The Vitis linker connects the interface of the platform with the interface of the kernel during the linking phase (see Figure 3.3. Once this connection is established, an application on the host can read or write to the kernel registers over the AXI-Lite bus.

In Vivado, we declared an external port in the block design and connected this port to the AXI-Lite interface that is coming from the host. In the block design, we added an AXI SmartConnect IP and connected the external port to a slave port of the SmartConnect. A SmartConnect is the successor of the AXI Interconnect IP and has better support for the IP integrator.

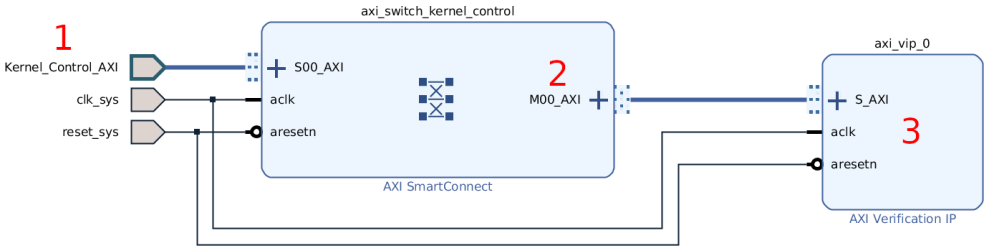

Figure 3.5: Kernel control part of the block design

Next, we added the master port of the SmartConnect to the platform metadata, so the Vitis compiler finds this interface during the linking phase. Before we continued with the memory interface, we wanted to check if we created a correct block design. However, the validation did not pass and the IP integrator produced an error.

The IP integrator complained because the master port of the SmartConnect was unconnected. The port is later connected to the kernel during the linking phase. However, the IP integrator expects that every interface has a valid connection. As a workaround, we connected the master interface of the SmartConnect to an AXI Verification IP. This IP is only used during simulation and does not allocate any resources on the FPGA. Vivado automatically assigns an address to the AXI Verification IP, so it is accessible over the AXI-Lite interface that is coming from the CPU. As we do not use this IP and to prevent conflicts with the Vitis linker, we disabled the path from the AXI-Lite port to the Verification IP in the address editor of the IP Integrator.

The AXI Verification IP fixed the validation error, but we still need to add an interface to the platform metadata. To accomplish this, we marked the next AXI master port of the SmartConnect as the control interface of the platform.

Figure 3.5 shows the relevant parts of the block design. (1) is the AXI-Lite interface that is coming from the CPU. The host application can later access the control registers of the kernel over this interface. (2) is the problematic interface that was initially unconnected. As we can see, it is now attached to an AXI Verification IP (3). The control interface for the kernel is not visible in this diagram. We added this interface only to the platform metadata, as shown in Figure 3.9. We discuss this figure and the complete block design at the end of this chapter.

C3 Memory Interface

The platform must have at least one interface that is connected to some kind of memory. The host application on the CPU uses this memory to exchange data with the kernel on the FPGA (*shared memory*). In most cases, the CPU's main memory is used as shared memory. Other architectures are

also possible. For example, the Alveo cards use the DDR on the FPGA as shared memory. To create a valid platform, we need to add the memory interface to the platform, similar to the control interface from the previous section.

If we want to add a memory interface to the Enzian platform, we have to decide which memory should be shared between the CPU and the FPGA. The CPU, as well as the FPGA, have access to their own DRAM subsystems. We have also block memory in the FPGA, but it is not suitable for this purpose because of the limited storage capabilities. The ideal platform would provide access to both the CPU and the FPGA DRAM. The Vitis linker could then pick the best memory based on the accelerated application and connect the kernel accordingly. However, for an initial prototype, we focus on a single shared memory.

There are two different systems to access memory on Enzian. The first system is Enzian DMA, which allows the FPGA to access the CPU DRAM coherently. The CPU can obviously access its DRAM, so we have a memory that both the CPU and the FPGA can access. The second system is the Enzian Directory Controller. The directory controller is implemented in the FPGA fabric and has three interfaces. The first interface takes memory requests from the CPU, the second interface handles requests from the FPGA, and the third interface is attached to a byte-addressable memory, for example, the DDR memory controller on the FPGA. With this system, it would be possible to use the FPGA DDR as shared memory. Unfortunately, the FPGA interface of the directory controller was missing at the time of writing. As only the CPU can access the FPGA memory with the directory controller, we cannot exchange data between the host application and the kernel. For that reason, we use the Enzian DMA system for memory transfers and the CPU DRAM as shared memory.

The Enzian DMA system is implemented in the module `eci_dma` in the Enzian Vivado project. Figure 3.6 shows the architecture of the module. A memory transfer is initiated at the right side of the diagram. The block *Debugging* has VIOs to create a read or write description of a memory transfer. In addition, the debugging module contains ILAs to visualize the data or the status of a transfer. To start a memory transaction the debugging module sends a read or write descriptor to the `axi_dma` module. The `axi_dma` module converts the descriptor into signals for the AXI bus that leaves the `axi_dma` module on the left side. The AXI signals are then split between two modules. Read requests go to the module `axi_eci_rd_slv`, and write requests are handled by `axi_eci_wr_slv`. Both modules convert an AXI request into an ECI package and send the package over the Enzian Coherent Interconnect to the memory system of the CPU.

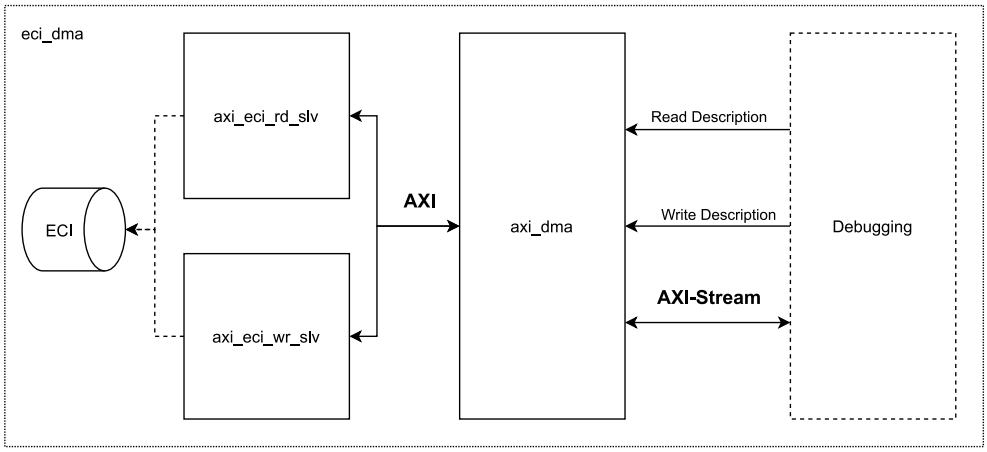

Figure 3.6: Enzian DMA architecture

The previous section described the transmission of read/write descriptors, but the debugging module sends or receives the actual data over an AXI-Stream interface, shown in the lower right part of Figure 3.6. The `axi_dma` module converts the AXI-Stream interface into a standard AXI interface. Then, the data is forwarded by either `axi_eci_rd_slv` or `axi_eci_wr_slv` to ECI and, finally, to the memory system of the CPU.

To add a memory interface to our platform, we could remove the debugging module and use the AXI-Stream port on `axi_dma` to access the CPU memory. This approach requires that Vitis supports the AXI-Stream protocol for platform memory interfaces. Vitis supports AXI-Stream as a platform memory interface, but with some limitations.

Looking at Figure 3.6 again, we can see that the left side of `axi_dma` is connected to a standard AXI interface. This interface is indicated as **AXI** in the middle of Figure 3.6. Standard AXI interfaces are fully supported by Vitis, so this interface is exactly what we need. Consequently, we can remove the `axi_dma` module because we do not need the AXI-Stream to AXI conversion.

Now that we have a suitable AXI port for the memory interface, we need to bring the AXI port into the block design. Like for the control interface, we can add an external port to the block design and attach the AXI port to it. Then, we added an additional SmartConnect to the block design and connected it to the external port.

Figure 3.7 shows the complete path of the memory interface from the CPU to the kernel. The diagram is divided into two parts. On the left you can see the `eci_dma` module and on the right we have the block design. If the kernel wants to access a memory location in the CPU DRAM, it sends a memory request to the SmartConnect. Then, then SmartConnect forwards the request, over the external port, to the `eci_dma` module. Inside the `eci_dma`, the

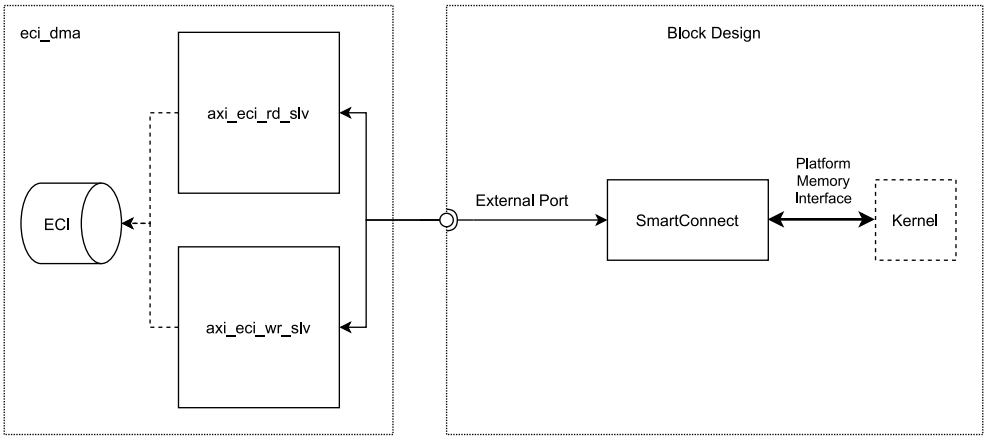

Figure 3.7: Path of the memory interface from the CPU to the kernel

request goes either to `axi_eci_rd_slv` or `axi_eci_wr_slv` and is forwarded to the CPU over ECI. The arrow between the SmartConnect and the kernel, is the memory interface (see **C3** in Figure 3.1).

Similar to the control interface, we also added an AXI Verification IP to prevent the issue with the unconnected port. A kernel can now send read and write requests to the CPU DRAM to access the shared memory.

During our experiments, we discovered a bug in the Enzian DMA subsystem, which changed the order of the cache lines in a read response. As long as the FPGA writes to a memory location in DRAM and reads the value back, everything works as expected. However, if the CPU writes a value to a memory location and the FPGA reads that memory location, the value is incorrect. As a simple example, let us assume that the CPU writes two cache lines into its DRAM. The FPGA reads the first cache line and gets the correct result. Now, if the FPGA tries to access the second cache line, Enzian DMA returns the value of the first cache line. This problem occurs most likely because the responses from the CPU over ECI are out of order. We did not investigate this problem further because there is an improved version of Enzian DMA in the Coyote project [29]. This version is more sophisticated than the initial version of Enzian DMA because the implementation is fully pipelined. Furthermore, the Coyote version is well-tested and has already been used in other projects. To fix the cache line bug, we integrated the Enzian DMA version of Coyote into our implementation, which was straightforward as the module interface did not change. The Enzian DMA module from Coyote fixed the bug and increased the throughput of Enzian DMA due to the pipelining.

Figure 3.8 shows the memory interface section of the block design. The design is similar to the control interface in the previous section and has

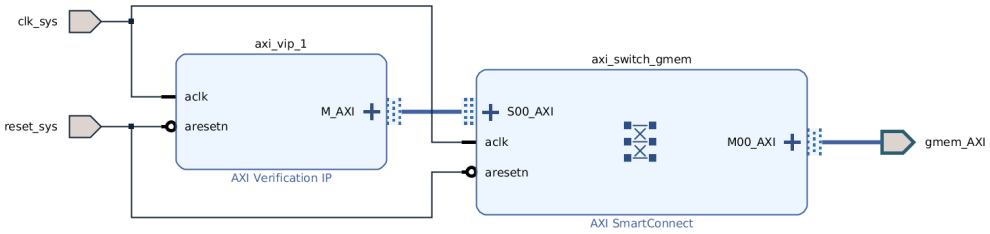

Figure 3.8: Memory interface part of the block design

again a SmartConnect and an AXI Verification IP. The SmartConnect has the memory interface that we added to the platform metadata. The AXI Verification IP fixes the problem with the unconnected ports. The directions of the AXI ports are now reversed. The external port `gmem_AXI` is an AXI slave port instead of an AXI master port like in the control interface part. The ports use the full AXI protocol and are not just AXI-Lite ports, but this is not visible in this figure.

C4 Interrupt

The Vitis documentation states that at least one interrupt signal is required for a valid platform. The kernel uses an interrupt to notify the host application about the completion of a computation.

Enzian has an interrupt line that goes from the FPGA to the CPU, but this part has not been tested very well, and we found little documentation about interrupts on Enzian. During our experiments, we discovered that interrupts are not a mandatory requirement for a platform. The scheduler of the host application falls back to polling if no interrupts are available on the platform. We did not include interrupts for our initial prototype, but they can be easily added in a future release of the platform.

C5 Clock

A platform requires at least one clock that is enabled in the metadata. It is also possible to create a platform with more than one clock. The Vitis linker can then pick a suitable clock to drive the kernel. If multiple clocks are enabled, the platform must define one default clock.

Enzian has a 300 MHz clock that comes directly from the board and drives most of the system. There is also a 50 MHz clock for the Microblaze processor. We used the 300 MHz clock to achieve optimal performance. We added the clock over an external port to the block design and enabled it in the platform metadata. Vitis 2020.1 has a known issue, that occurs when only one clock is enabled in the platform. With only one clock in the platform, the Vitis linker cannot write the correct clock information into the device binary. We added a

second dummy clock to the platform to fix this problem. The dummy clock is connected to the same wire as the default clock and is not used by the kernel. With this issue resolved, we successfully added a clocking infrastructure to the platform.

Block Design

This section describes the metadata section and the complete block design of the Enzian platform.

The metadata is configured in the Platform Interfaces tab of the IP integrator. Figure 3.9 shows the enabled interfaces of the platform. We enabled two clocks, clk_sys and clk_sys_dummy, in the External Interfaces section. Further, we can see the control interface, which is labeled M02_AXI. Below the control interface, is the memory interface (S02_AXI). Vivado includes this metadata when it creates the platform file.

Figure 3.10 shows the complete block design of the Enzian platform. The part of the block design that is responsible for controlling the ECI link is not shown. We can see that most of the previous components are present, except for **C1** and **C4**. The control interface part is indicated by **C2**, whereas the memory interface is shown in **C3**. Then, the two clocks are labeled with **C5**. The Vitis linker can now use this design to connect the kernel to the correct interfaces. We added a kernel (K) to Figure 3.10 to illustrate this. There are still unconnected ports on the kernel, for example, the interrupt port, but they are not required for a working platform.

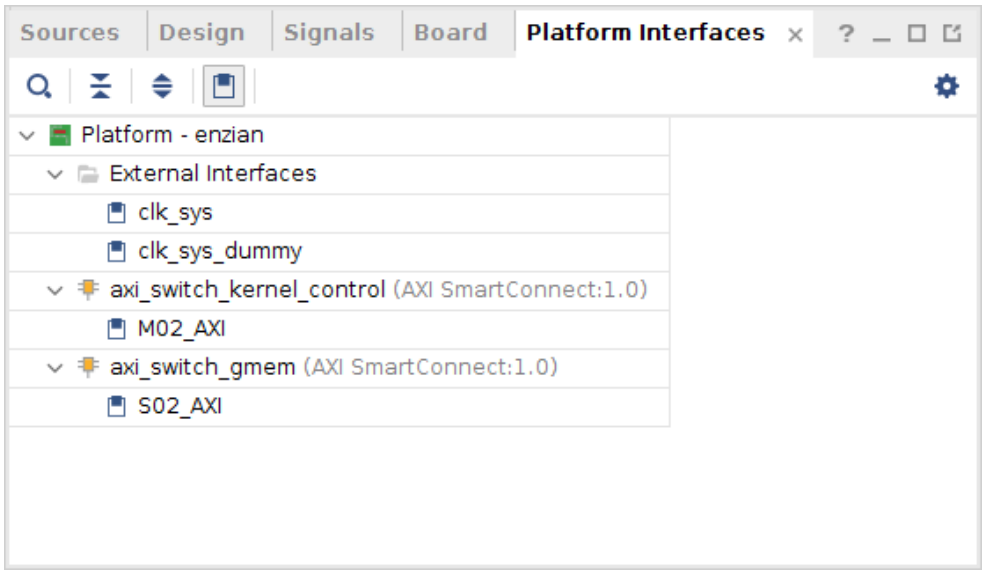

Figure 3.9: Platform Interfaces tab of the IP integrator

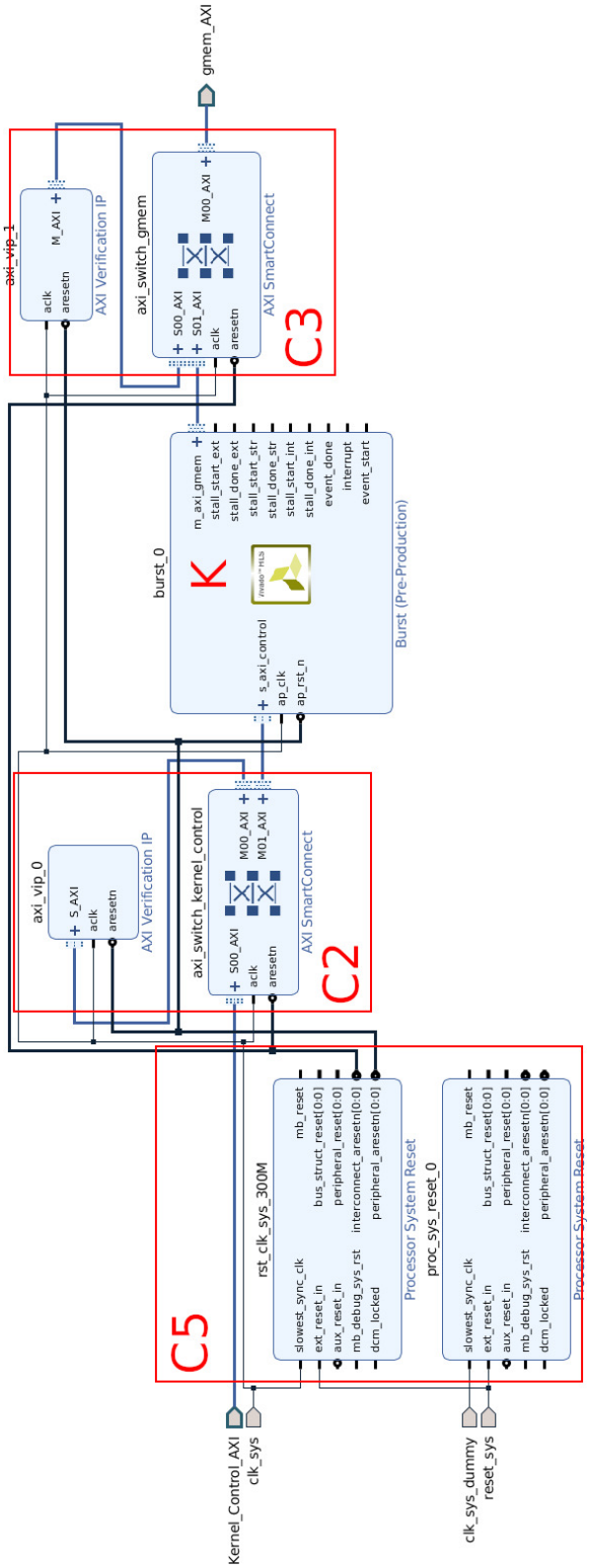

Figure 3.10: Final block design with an example kernel

Export the hardware platform

The last step to create a hardware platform is to export the design from Vivado and save it in an XSA file. The platform can be exported with the following TCL command:

```
write_hw_platform -unified <output.xsa>
```

This command generates the XSA file that contains all the necessary parts to create a platform for Vitis. The remaining steps to create a platform are done in the Vitis IDE and are discussed in the next section.

3.1.2 Software Platform

A software platform consists of one or several domains. A domain defines the environment in which the host application is running. The runtime environment includes various drivers and libraries required by the host application. These software packages are either included in a BSP for embedded systems or are provided as part of an operating system, for example, in a server environment with Alveo accelerator cards.

Every aspect of the software platform can be configured in the Vitis IDE or with the Xilinx Software Command-Line Tool (XSCT). The Systems group has already created a project for the Vitis IDE. The project includes a domain for the Microblaze CPU. This domain provides the necessary libraries for the program that brings up the ECI link.

In addition to the Microblaze domain, we need a domain for the runtime environment of the host application. The host application runs in Ubuntu on the ThunderX CPU. Vitis has no official support for custom platforms with such an architecture, although this is a setup similar to the Alveo cards. Therefore, we did not add a domain for the ThunderX and installed the required libraries manually on Ubuntu (see Section 3.3.1).

To create the software platform for Enzian, we imported the XSA file that contains the hardware platform into the Vitis IDE and followed the instructions to add the Microblaze domain. After building the project, we had a file (`*.xpfm`), that contains both the hardware and the software platform.

In the next section, we discuss how Vitis uses the platform file to create a host and a device binary.

3.2 Vitis Build Tools

This section covers how we adapted the Vitis build process to support the Enzian platform. In the best case, the build tools work without any changes by just using the Enzian platform file from the previous section. Unfortunately,

the tools did not produce the expected output, and we had to make changes at specific points of the build process. We tried to keep these modifications to a minimum so that the tools work as if Enzian were supported by Vitis.

The build process follows two separate workflows. One workflow builds the device binary with the Vitis compiler. The other workflow creates the host application with the GNU compiler collection. We discuss how we adapted each workflow in the following two sections.

3.2.1 Device Binary

Vitis includes a compiler, called `v++`, to build the FPGA binary. This section covers how we adapted `v++` to support the Enzian platform. Section 2.3.1 has a detailed description of the build process.

The Vitis compiler requires a source file of a kernel and a platform file to create a device binary. We already had a platform file from the previous section, but we still needed a kernel to run the build. We opted for a simple kernel, called `vadd`, that takes two vectors as inputs, performs a vector addition, and writes the result to a third vector. Listing 3.1 shows the source code of the initial version of the `vadd` kernel.

Listing 3.1: Initial version of the `vadd` kernel

```
void vadd(
    int *in1, // vector 1
    int *in2, // vector 2
    int *out, // result
    int size  // vector size
)
{
    for(int i = 0; i < size; ++i)
    {
        out[i] = in1[i] + in2[i];
    }
}
```

Even for such a simple kernel, we had to make several changes to the code and the build process, to run this example on Enzian. The first set of changes were necessary during the compilation of the kernel. The second set of changes were applied to the linking phase. We go over each set of changes separately.

Compile Phase

During the compile phase, v++ creates a Xilinx object file from a kernel source file. We used the following command to compile the `vadd` kernel:

```
v++ -c -t hw --platform enzian.xpfm -k vadd
    -o vadd.xo ./vadd.cpp
```

This command did not complete successfully and produced an error. The Vitis compiler was not able to assign a memory interface to the argument `in2`.

The Enzian platform provides only one AXI memory interface for the kernel. The AXI protocol allows only one read and one write simultaneously. However, inside the loop body in Listing 3.1, the kernel reads `in1[i]` and `in2[i]` at the same time. This results in two concurrent read requests. Usually, the Vitis compiler assigns one of the conflicting arguments to another memory interface. Unfortunately, this does not work for the Enzian platform because it has only one memory interface. Thus, the compiler produces an error and aborts the build.

There are two solutions to this problem. Firstly, we could extend the Enzian platform with additional memory interfaces, which allows the Vitis compiler to assign the arguments correctly. Secondly, we could change the source code of the `vadd` kernel to avoid these two concurrent write requests.

Suppose we want to add an additional memory interface. In that case, we need to divide the CPU address space and assign a separate address range to each interface. In addition, the host application must allocate the memory for an argument in the correct address range. In summary, adding an additional interface is too much effort for an initial prototype.

Instead, we removed the concurrent read by adding an integer constant instead of a value from a vector. Now we have only one read in the loop body. The revised version of the `vadd` kernel is given in Listing 3.2.

Listing 3.2: Revised version of `vadd` without concurrent reads

```
void vadd(
    int *in,  // input vector
    int *out, // result vector
    int size  // vector size
)
{
    for(int i = 0; i < size; ++i)
    {
        out[i] = in1[i] + 42;
    }
}
```

With this new version, the Vitis compiler builds the `vadd` kernel successfully and produces a Xilinx object file.

Before we passed the object file to the linker, we checked if the Vitis compiler created the correct AXI interfaces. As mentioned in Section 2.3.1, a Xilinx object file is a regular zip archive with a different file extension (`*.xo`). The zip archive contains an IP and some metadata. This IP represents the kernel logic in HDL and can be included in a Vivado project like any other third-party IP.

To examine the `vadd` kernel, we created a new block design in a Vivado project and added the IP. Figure 3.11 shows the IP of the `vadd` kernel with its AXI interfaces. The control interface is on the left side and is called `s_axi_control`. The memory interface is on the right side and labeled `m_axi_gmem`. The memory interface is a full AXI port whereas the control interface is a AXI-Lite port. Note that the memory interface has more wires as the control interfaces because AXI-Lite uses only a subset of the AXI wires. We omitted the irrelevant wires of the memory interface for clarity.

If we want to connect the AXI ports of the kernel IP in Figure 3.11 with AXI ports of the Enzian platform, certain requirements must match. The two essential requirements are the address width and the data width. The address width defines the number of wires of the `AxADDR` signal and with that the address range of the port. For example, the control interface in Figure 3.11 has six wires for the `s_axi_control_AWADDR` signal. This corresponds to an address width of six so the port can access the addresses from `0x0` to `0x63`. The data width, on the other hand, defines the word size of the port. If we look at the `s_axi_control_RDATA` signal in Figure 3.11, we can see it has 32 wires or a data width of 32. This means that each data transfer (beat) transmits 32 bits of data. We refer to Section 2.4 for details about the AXI protocol.

In the following sections, we compare the interfaces of the kernel IP with the interfaces of the Enzian platform and address all inconsistencies. All relevant signals are highlighted in Figure 3.11.

The Enzian platform imposes two requirements to the control interface and the memory interface. First, the address width for both interfaces must be at least 40-bit. The ECI protocol dictates this address width. Second, there is a restriction on the data width. The data width of the control interface must be 64 bits because the ThunderX CPU has the same word size. The data width of the memory interface must be 1024 bits. This restriction is given by the Enzian DMA module. Enzian DMA can only read or write at cache line granularity and a cache line has 1024 bits [42].

Irrespective of the Enzian requirements, Vitis has its own default values for the kernel interfaces. As Figure 3.11 shows, the default address width of the kernel control interface is six (see `s_axi_control_AxADDR`). Six wires

Figure 3.11: Initial IP of the `vadd` kernel

are enough to to address all control registers. Vitis only supports 32-bit platforms, i.e., a word size of 32. Therefore, the data width of the control and the memory interface is 32 by default. This matches with the kernel IP in Figure 3.11. The signals `s_axi_control_WDATA` and `s_axi_control_RDATA` of the control interface have both 32 wires. Similarly, for the memory interface where only the signal `m_axi_gmem_WDATA` is shown. Actually, Vitis does not always set the default data width to 32. The data width of the memory interface depends on compiler optimizations and can vary from 32 up to 512 bits.

Comparing the requirements of Enzian with the Vitis default settings, reveals three problems. The first issue is the data width of the control interface. While Enzian requires a data width of 64, Vitis has a default data width of

32. A mismatch in the data width means wrong values in the kernel control registers and unexpected behaviour of the kernel.

The second problem is the address width of the memory interface. Vitis created a 32-bit wide address port, but 40 bits are required to access all addresses on the ThunderX. In the worst case, the MSB bits of an address are truncated by the 32-bit address width of the memory interface and the kernel accesses a wrong memory location. The address width mismatch of the control interface is not an issue, because all control register addresses can be represented by the 40-bit address width of the Enzian platform.

The last issue is the data width of the memory interface. Enzian can only write cache lines and requires a data width of 1024 bits. Vitis, on the other hand, varies the data width of the kernel based on compiler optimizations. Again, a data width mismatch means, that values end up at the wrong memory location or dead locks occur. All of these problems need to be resolved to successfully run the `vadd` kernel on Enzian.

The first two issues, the data width mismatch of the control interface and the wrong address size of the memory interface, have a straightforward solution. For the last problem, the data width mismatch of the memory interface, exists multiple solutions each with their own pros and cons. We discuss these solutions in Section 3.2.1. In the remaining part of this section we present our solution for the first two problems.

As discussed previously, the Vitis compiler calls Vitis HLS under to hood to convert the kernel source, for example written in C/C++, into an IP. If we want to change the kernel interfaces, we need to influence the high-level synthesis of the kernel. Vitis HLS has various configuration commands to control the results of the synthesis [63]. The command `config_interface` specifies the options for the IP interfaces. Among these options, `s_axilite_data64` and `m_axi_addr64` are particularly interesting. Both options take `true` or `false` as inputs, which enables or disables the option respectively. The first option (`s_axilite_data64`) sets the data width of the kernel control port to 64 bit. The control port of the kernel has now the same data width as the port of the Enzian platform. The second option (`m_axi_addr64`) globally enables 64-bit addresses for all memory ports of the kernel. The documentation states that this option is enabled by default in the Vitis flow, but we did not observe that. Anyway, enabling this option creates a kernel with the correct address width for the Enzian platform.

Figure 3.12 shows the kernel IP with the correct interfaces. As we can see, the signals `s_axi_control_WDATA` and `s_axi_control_RDATA` have now 64 wires, so this matches with the data width of the Enzian platform, which is also 64. Next, we see that the address width of the memory interface has also changed to 64 bits (see `m_axi_gmem_AWADDR`). This corresponds with the address width of the Enzian platform. Finally, the data width of the memory

Figure 3.12: `vadd` kernel IP with correct interfaces

interface has increased to 1024 (see `m_axi_gmem_WDATA`). The data width of the memory interface is now equal with the cache line size of the ThunderX. This ensures that the kernel writes only complete cache lines which was required by Enzian. Section 3.2.1 discusses how we changed the data width of the memory interface.

The previous options solve the first two problems, but we still need to pass them to Vitis HLS during the build. The Vitis compiler has the flag `--advanced.prop solution.hls_pre_tcl`, that takes a path to a Tcl script. The script runs before the high-level synthesis. Thus, we can use this script to enable the two options, we discussed in the previous section. With this script the Vitis compiler runs without any manual intervention and creates a Xilinx object file.

Linking Phase

During the linking phase, the Vitis compiler takes Xilinx object files and a platform as input and links them together to produce the device binary (`*.xclbin`). We used the following command to link the `vadd` kernel from the previous section and the Enzian platform:

```
v++ -l -t hw --platform enzian.xpfm -o vadd.xclbin \
                                       ./vadd.xo
```

The linker goes through several steps to produce the device binary. It starts with extracting the platform file and it generates a Vivado project from these files. Next, the linker adds the kernel to the block design. Then, the kernel interfaces are linked to the platform interfaces. In addition, the linker assigns addresses to certain slave interfaces. At this stage, the block design is complete and the linker goes through synthesis and implementation to generate a bitstream. When the bitstream is ready, the linker generates metadata for the runtime environment and packages the bitstream and the metadata in a device binary, or xclbin file.

The linking process can be interrupted at any of the above steps. This is especially useful for debugging or for making manual changes to the FPGA design. The Vitis compiler has the option `--to_step` to run the linking up to a certain step. Once the linker is interrupted, we can open the Vivado project which the linker has created. The Vivado project file is in the folder `_x/link/vivado/vpl/prj/` relative to the build directory. Then, we can edit the FPGA design like in a normal Vivado project. After we completed our changes, we can continue the linking process with `--from_step`.

In our first test build, the linker run without any errors and produced a xclbin file. However, the kernel did not work in hardware because it was unable to read any input data. To debug this problem, we interrupted the linker after the block design was updated (`--to_step vpl.update_bd`). We examined the memory interface of the Enzian platform, in particular the address range. The linker connected the kernel interface correctly to the platform interface and all AXI parameters for both interfaces were correct. However, we noticed that the linker assigned a wrong address range to the platform memory interface. The AXI interconnect routes the AXI signals based on this address range to the correct interface. The interconnect drops all requests that have an address outside of an address range. Consequently, if the address range of the memory interface is incorrect, the AXI interconnect rejects memory requests from the kernel and the requests are never sent to the ECI.

The linker assigned a random start address to the platform memory interface and the size of the address range was only 64 kilobytes. We decided to give the kernel full access to the CPU address space. Accordingly, we defined an address range that starts at address `0x0` and ends at the last representable

address. Giving full access is not desirable from a security perspective, but we wanted to keep the design simple for this initial prototype.

The documentation does not describe how the linker assigns these address ranges. Thus, we tried to reverse engineer the address assignment process but we were not successful. Instead, we manually assigned the correct address range to the platform memory interface. Later, we automated this assignment with a custom Tcl script. The script made it possible to run the build process without any manual interventions. We passed this script with the option

```
param=compiler.userPostDebugProfileOverlayTcl=
        fix_address.tcl
```

to the Vitis compiler, which is given in Listing 3.3.

Listing 3.3: Tcl script to assign the correct address range

```
set addr_seg [get_bd_addr_segs \
    {vadd_1/Data_m_axi_gmem/SEG_gmem_AXI_Reg}]
set_property offset 0x0000000000000000 addr_seg
set_property range 16E addr_seg
```

The first two lines save the address range, or address segment, in the variable `addr_seg`. The third line sets the start address to `0x0`. The last line extends the address range to the maximum value. The script includes the name of the kernel, as you can see on the second line (`vadd_1`) of Listing 3.3. The kernel name is usually different from application to application, so we have to adapt this script for each new application that we develop for Enzian. It would, therefore, be interesting future work to fully reverse engineer the address assignment of Vitis and to adapt the Enzian platform file so that the addresses are assigned correctly.

Up to this point, we solved two problems that prevented the Vitis compiler to create a correct device binary. The first problem from the previous section was a mismatch between the address and data width of the kernel and the platform interface. The second problem occurred during the linking phase, where the linker assigned wrong addresses to the memory interface of the platform. We solved both problems with custom scripts that make the build process compatible with Enzian. There is still the problem with the data width of the memory interface and we discuss several solutions for this problem in the next section.

Data Width Conversion

As mentioned before, Enzian DMA can only read or write at cache line granularity (1024 bits). This requirement simplifies the implementation of Enzian DMA significantly but implies that the data width of the AXI interface

is 1024. The AXI interface of the kernel has also a specific data width. For example, the `vadd` kernel has a data width of 32 because the kernel arguments are 32-bit integer arrays. To connect the AXI interface of Enzian DMA with the AXI interface of the kernel, we used a SmartConnect, as described in the previous sections.

Note that both interface have a different data width. However, the AXI standard requires that the data width of two connected interfaces must match. Hence, the SmartConnect has to insert some logic to convert the data size. In the case of our `vadd` kernel, the SmartConnect needs to convert the 32-bit data width of the kernel to the data width of Enzian DMA which is 1024. The logic to convert the data sizes is called *AXI Downsizer* or *AXI Upsizer* respectively. The SmartConnect documentation [55] describe the data width conversion in detail but in our case it is sufficient to know that some transformation is ongoing.

Figure 3.13 summarizes the whole setup with the `vadd` kernel as an example. On the left, we can see the `vadd` kernel that is connected to the SmartConnect. Similarly, the Enzian DMA module is connected to the SmartConnect as well as to the Enzian Coherent Interconnect. The data width of both connections is shown in the figure. The AXI Upsizer inside the SmartConnect handles the data width conversion.

During our initial hardware tests, we observed incorrect results from the `vadd` kernel. We found that the incorrect results were caused by the data width conversion of the SmartConnect. The SmartConnect padded the remaining bits with zeros and sent a 1024-bit request for each 32-bit request. This behaviour produced many requests to the Enzian DMA and the design did not work correctly. Therefore, we need to increase the data width of the kernel to reduce the number of requests.

There are three approaches to increase the data width of the kernel. Firstly, we can use a special type for the kernel arguments. This type statically defines the data width of the kernel AXI interface. Secondly, we can change the data width with a custom script that is similar to the script in Listing 3.3. Lastly, we could use compiler optimizations to increase the data width. In the remaining part of this section, we cover each solution separately.

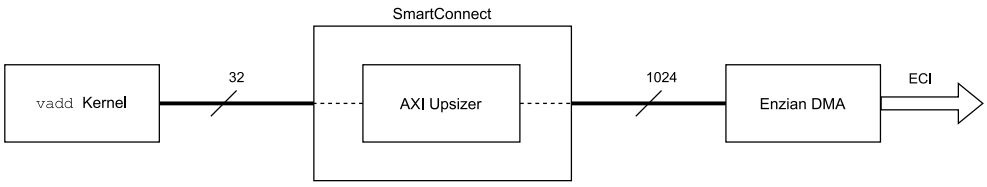

Figure 3.13: Block diagram that illustrates the data width issue

Kernel Argument Type The types of the kernel arguments define the AXI memory interface of a kernel. In particular, the number of bits of a native C/C++ type determines the data width of the AXI interface. For example, the `vadd` kernel has two arrays of type `int` as kernel arguments. An `int` value has 32 bits in C/C++, so Vitis HLS creates an AXI interface with a data width of 32.

The native C/C++ types are on 8-bit boundaries (8, 16, 32 and 64 bits) but an FPGA supports arbitrary bit lengths for arithmetic operations. Vitis HLS provides arbitrary precision types for C++ to specify types with arbitrary bit widths [52]. For example, if we want to multiply two 20-bit integers, we do not need a 32-bit multiplier. Instead, we can use an arbitrary precision type to specify that only 20 bits are used in the calculation.

With this in mind, we can use arbitrary precision types to specify the data width of the AXI interface. Listing 3.4 shows a version of the `vadd` kernel that uses arbitrary precision types.

Listing 3.4: `vadd` kernel with arbitrary precision types

```
#include <ap_int.h>

typedef ap_int<1024> enzian_int;

void vadd(
    enzian_int *in,
    enzian_int *out,
    int num_blocks
)
{
    for(int i = 0; i < num_blocks; ++i)
    {
        out[i] = in1[i] + 42;
    }
}
```

To use arbitrary precision types, we must include the `ap_int.h` header file that is provided by XRT. Then, we define a new type called `enzian_int`. This type has a bit-width of 1024 which is equal to the cache line size. We use this type for the kernel arguments that define the input and output arrays. Instead of passing the vector size to the kernel, we use the last kernel argument to specify the number of blocks that the kernel needs to process. One block is equal to a cache line and contains 32 `int` values. Therefore, the kernel performs 32 additions per loop iteration. The number of blocks is calculated by the host application and passed to the kernel.

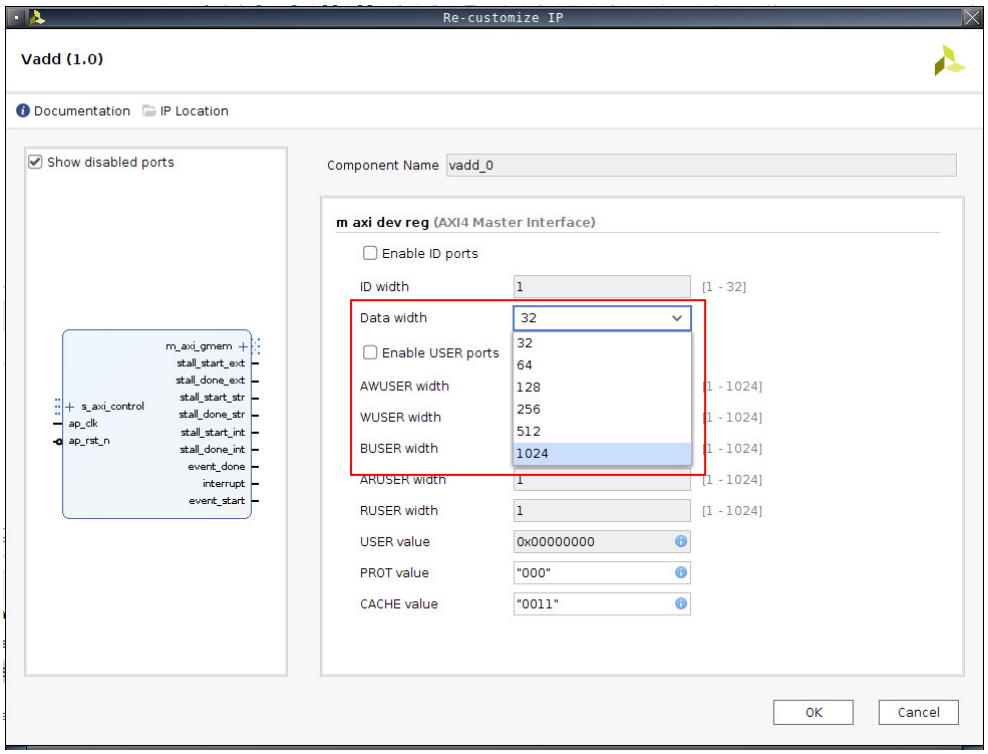

Figure 3.14: IP options of the `vadd` kernel

The `enzian_int` type has 1024 bits so Vitis HLS creates an AXI interface with a data width of 1024. Now the kernel and Enzian DMA have the same data width so the SmartConnect does not include any upsizer logic. This solves the problem with the wrong data width conversion.

Arbitrary precision types are only available for C++ kernels. For OpenCL C kernels, we can use OpenCL Vector Data Types [17] that provide a similar functionality. The type `long16` represents a vector of 16×64-bit integer values. Conveniently, this data type has 1024 bits which is the required data width of the AXI interface.

Custom Script While we were debugging the address assignment issue in Section 3.2.1, we discovered another approach to change the data width of the kernel AXI interface.

As we described previously, the Vitis linker creates a block design that includes the kernel as a regular IP. This IP provides an option to change the data width of the AXI interface, as you can see in Figure 3.14. We can use the drop-down menu to set the data width to 1024 bits and Vivado creates a corresponding AXI interface. Now the kernel IP has the same data width as Enzian DMA and the SmartConnect can again omit the AXI upsizer.

This approach is not documented by Xilinx and can result in unexpected behaviour. Nevertheless, we successfully used this method for the matrix-matrix multiplication benchmark in Section 4.2.

Compiler Optimizations The last option uses compiler optimizations to increase the data width of the AXI interface. Vitis has two modes to transfer arrays to the kernel [58]:

- Individual data transfer
- Burst data transfer

With individual data transfers, the kernel reads or writes a single element per address. For example, Listing 3.5 shows a kernel that performs a single read and a single write operation.

Listing 3.5: Individual data transfer

```
void single(int *d) {
    static int acc = 0;

    acc += *d;
    *d  = acc;
}
```

The synthesised logic generates an address on the AXI interface to read a single value and an address to write a single value. The AXI interface has a data width of 32 because the kernel argument has type `int`. Hence, the interface transfers one 32-bit value per address.

With burst data transfers, the kernel reads or writes multiple consecutive values starting from a base address. Burst data transfers allow the compiler to increase the data width of the kernel because multiple read or write requests can be bundled together. Burst mode is possible if the kernel uses the `memcpy` function or a pipelined `for` loop. We prefer `memcpy` because pipelining a loop is not always possible.

Listing 3.6 shows a version of the `vadd` kernel which uses `memcpy`.

Listing 3.6: `vadd` kernel with `memcpy`

```
#include <string.h>
#define BUF_SIZE 32

void vadd(int *in, int *out, int n_elements){
    int buf[BUF_SIZE];
    for (int i = 0; i < n_elements; i += BUF_SIZE) {
        memcpy(buf, in + i, BUF_SIZE * sizeof(int));
```

```
            for (int j = 0; j < BUF_SIZE; j++)
                buf[j] += 42;

            memcpy(out + i, buf, BUF_SIZE * sizeof(int));
        }
    }
```

In each outer loop iteration, we copy 32 integers from the input vector to an internal buffer, called `buf`. Then, we add a constant to each element of the buffer. Finally, we copy the buffer back to the output array. The calls to `memcpy` are translated into burst transfers. Therefore, the compiler can increase the data width of the `vadd` kernel. In this example, the data width was increased to 512 bits. This still requires an upsizing of the data width to 1024. In contrast to the upsizing from 32 to 1024, we could not observe invalid behaviour of the upsizer logic when upsizing from 512 to 1024.

3.2.2 Host Binary

After we successfully created a device binary, we also need a host application to run the `vadd` kernel. The host application uses the OpenCL API to discover the available devices, to allocate buffers for data exchange and to finally execute the kernel.

Listing 3.7 shows an excerpt of the host application which runs the `vadd` kernel.

Listing 3.7: Host application for the `vadd` kernel

```
// Step 1: Initialize the OpenCL environment
cl::Device device = ...;
cl::Context context(device);
cl::CommandQueue q(context, device);
cl::Program program(context, devices, xclbin);
cl::Kernel krnl_vector_add(program, "vadd");

// Step 2: Create buffers and initialize test values
cl::Buffer in_buf(context,  CL_MEM_READ_ONLY, size);
cl::Buffer out_buf(context, CL_MEM_WRITE_ONLY, size);

int *in = (int *)q.enqueueMapBuffer( ... );
int *out = (int *)q.enqueueMapBuffer( ... );

for(int i = 0 ; i < size; i++){
    in[i] = i;
    out[i] = 0;
}
```

```
// Step 3: Run the kernel
krnl_vector_add.setArg(0, in_buf);
krnl_vector_add.setArg(1, out_buf);
krnl_vector_add.setArg(2, size);

q.enqueueMigrateMemObjects({in_buf}, ... );
q.enqueueTask(krnl_vector_add);
q.enqueueMigrateMemObjects({out_buf}, ... );

q.finish();

...
```

The host application goes through three steps to run the `vadd` kernel. First, the application initializes the OpenCL environment. Several API calls are necessary to obtain a reference to a device, the FPGA accelerator in this case. We omitted this part because platform and device discovery is already discussed in Section 2.1.1. Using the device reference, we create an OpenCL context and a command queue. Then we load device binary (`*.xclbin`) and create a program object. From the program, we get a reference to the `vadd` kernel. These were all API calls to create an OpenCL environment. Second, the host application allocates a buffer by creating a `cl::Buffer` object. The host application cannot access a buffer directly, so we have to map the buffer into the host address space with a call to `enqueueMapBuffer`. This function returns a pointer to the buffer and we can initialize the test data. Finally, the third step executes the kernel. The host application assigns the buffer and the size of the vectors to the kernel arguments. Then a call to `enqueueMigrateMemObjects` transfers the input vector and the data size to the kernel. Then, the host application calls `enqueueTask` to run the kernel. After the execution, the result is transferred back to the host again with `enqueueMigrateMemObjects`. Because the OpenCL API is asynchronous, we have to call `finish` to wait until every command is finished.

Unlike the device binary build, there were no changes necessary to compile the host application. We compiled the host application like a regular C++ program, as discussed in Section 2.3.2. We run the compilation directly on Enzian because the GNU compiler collection was already installed and we wanted to avoid cross-compilation. We used the following command to compile the host application:

```
g++ -Wall -g -std=c++14 -lOpenCL -lpthread -lrt
        -lstdc++ -o host.exe host.src
```

The library OpenCL is provided by XRT which we cover in the next section. The other libraries are shipped with Ubuntu 20.04.

This section concludes our discussion about the build process of the `vadd` kernel. We used the Vitis compiler (`v++`) to create a device binary (`vadd.xclbin`). Then, we compiled the host application with GCC (`g++`) and got the host binary `host.exe`. In the next section, we discuss our changes to the Xilinx Runtime (XRT) and how we run the `vadd` kernel with XRT.

3.3 Xilinx Runtime (XRT)

The Xilinx Runtime (XRT) is the execution environment for the host application. At the beginning of this section, we describe how we built XRT, and installed it on Enzian. Next, we summarize the steps of our initial hardware test. During this test, we manually interacted with the kernel without XRT. Finally, we run a kernel using the whole XRT stack. This experiment demonstrates the first execution of a simple OpenCL application on Enzian.

3.3.1 Cross Compilation

The XRT source code is available on GitHub [65] and the documentation [64] has instructions for building the software stack. XRT uses the CMake build system. The user space libraries, the kernel drivers and the command line tools are all built from the same repository. The output artifact of the build is an RPM or a Debian package.

With these packages, we can install the user space libraries and the command line tools to the correct folders on Linux. During the compilation of a host application, GCC automatically locates the XRT libraries and links them with the host application.

The drivers are installed with Dynamic Kernel Module Support (DKMS). DKMS is a framework that helps installing drivers that live outside the kernel source tree. The driver source code is included in the RPM or Debian package. During the package installation, the driver is compiled with the installed Linux header files. This mechanism prevents the redistribution of a package when a new kernel version had been released.

The ThunderX CPU on Enzian has an ARM architecture, so we need to compile XRT for the ARM instruction set. We have two options to build XRT for ARM. We could either build XRT directly on Enzian or we use a cross-compiler on a x86 system.

We decided to cross-compile XRT on the internal build server because the ThunderX has a poor single-core performance. XRT already supports ARM cross-compilation because edge devices also have an ARM processor. There is

the script `build/cross_compile.sh` in the XRT repository to start the ARM build process. We executed this script and the build produced the desired artifacts.

Enzian runs Ubuntu 20.04, so we can install XRT with the Debian package that was created during the build. The Debian package is in the folder `build/Enzian_aarch64/`. We deployed the package with `rsync` to Enzian and installed it with the following command:

```
sudo apt install --reinstall ./xrt_*.deb
```

After installing XRT on Enzian, we had an environment to compile host applications and to run Vitis accelerated applications.

3.3.2 Initial Hardware Test

For our first experiment, we wanted to run the `vadd` kernel from Section 3.2.1. The `vadd` kernel adds a constant number to an input vector and writes the result to an output vector.

XRT is a complex software stack with user space code interacting with a kernel driver. Therefore, we did not run the `vadd` kernel directly with XRT for an initial test. Instead, we inferred the necessary steps to run a kernel from the XRT source code. These steps are listed below:

- Program the FPGA with the bitstream from the device binary (xclbin)
- Allocate a buffer for the input and output vector
- Set the buffer addresses and the vector size as kernel arguments
- Start the execution
- Read and verify the result of the computation

We executed each of the above step manually to fully understand the execution of the kernel. After, we verified that the interaction with the kernel works as expected, we did a second experiment where we run the kernel using the complete XRT software stack. This experiment is discussed in the next section. In this section, we describe each of the previous steps.

Program the FPGA

Before we can start interacting with the kernel, we need to program the FPGA with a bitstream. Usually, XRT extracts the bitstream from the device binary and uses an ioctl system call to pass the bitstream to the Linux driver. Then, the driver programs the FPGA using the Linux FPGA subsystem[9]. Programming the FPGA like this is preferable, but we cannot use the FPGA subsystem because it does not support the Enzian FPGA.

Instead, we used the Hardware Manager in Vivado to program the FPGA. This approach is not optimal because we have to open Vivado each time we run an accelerated application. However, programming the FPGA with the Hardware Manager is sufficient for a first prototype and a driver for the FPGA subsystem can be implemented in the future.

The Hardware Manager needs a bitstream file (`*.bit`) to program the FPGA. Instead, Vitis creates a device binary (`*.xclbin`), which includes the bitstream. There are two possibilities to extract the bitstream from the device binary. First, we could use the command line tool `xclbinutil` and extract the bitstream with the following command:

```
xclbinutil -i vadd.xclbin \
    --dump-section BITSTREAM:RAW:bitstream.bit
```

The bitstream is saved to `bitstream.bit` and can now be used by the Hardware Manager. The other possibility is to open the Vivado project in the build directory, as described in Section 3.2.1. If we open the Hardware Manager in this project, the correct bitstream is automatically selected and we can program the FPGA. We prefer the later approach because it saves the extra step over the command line tool.

After the bitstream is loaded, the Hardware Manager resets the FPGA to its initial state. This also reboots the CPU. During the reboot, the CPU and FPGA perform the training of the ECI link. If the training was successful, the ECI link is ready and the CPU boots into Linux.

Buffer Allocation

Once Linux has started successfully, we can allocate memory for the input and the output vector of the `vadd` kernel. XRT uses the Linux DRM Framework to handle buffer allocation, but for our case we created a small C program, called `buf`. The source code is shown in Listing 3.8.

Listing 3.8: C program for buffer allocation

```
uintptr_t in_paddr, out_paddr;
volatile uint32_t *in_buf, *out_buf;

// buffer allocation
in_buf = aligned_alloc(1024, 4096);
out_buf = aligned_alloc(1024, 4096);

// init data
for (int i = 0; i < 32; i++)
    in_buf[i] = i;
memset((void*)out_buf, 0, 4096);
```

```
// look up physical addr
virt_to_phys(&in_paddr, (uintptr_t)in_buf);
virt_to_phys(&out_paddr, (uintptr_t)out_buf);

// print result
printf("paddr: 0x%jx 0x%jx\n", in_paddr, out_paddr);
print_buffer(out_buf);
```

Initially, the program allocates memory for the input and the output buffer. Note that the memory must be aligned to the cache line size (1024 bit) because Enzian DMA can only transfer full cache lines. Next, the input and output buffer are initialized with test data. Then, the program calls `virt_to_phys` to look up the physical start address of the buffers. The function `virt_to_phys` parses `/proc/<pid>/pagemap`, which exposes the address mapping of a process to the user space [43]. Notice that root permissions are required to access the address map. Finally, the program prints the addresses and outputs the result buffer in an endless loop (`print_buffer`). We omitted error handling to make this example more concise.

Once the program is running, we can send the physical addresses of the buffers to the kernel. The next section covers how we transfer these addresses to the kernel. After receiving the addresses, the kernel uses Enzian DMA to read the input buffer and to write the result in the output buffer. The program outputs the output buffer continuously and we can verify the result.

Kernel Arguments

Next, we need to set the arguments of the kernel function. As stated in Section 3.1.1, the arguments are set by various registers. Vitis HLS creates a register map during the kernel compilation. The register map of the `vadd` kernel is shown in Listing 3.9.

Listing 3.9: Register map of the `vadd` kernel

```
0x00 : Control signals
       bit 0  - ap_start
       bit 1  - ap_done
       bit 2  - ap_idle
       bit 3  - ap_ready
       bit 4  - ap_continue
       bit 7  - auto_restart
       others - reserved
0x08 : Global Interrupt Enable Register
       bit 0  - Global Interrupt Enable
       others - reserved
```

```
0x10 : IP Interrupt Enable Register
       bit 0  - enable ap_done interrupt
       bit 1  - enable ap_ready interrupt
       others - reserved
0x18 : IP Interrupt Status Register
       bit 0  - ap_done
       bit 1  - ap_ready
       others - reserved
0x20 : Data signal of in
       bit 63~0 - in[63:0]
0x28 : reserved
0x30 : Data signal of out
       bit 63~0 - out[63:0]
0x38 : reserved
0x40 : Data signal of n_elements
       bit 63~0 - n_elements[63:0]
0x48 : reserved
```

The first register (`0x00`) has several signals to control the kernel execution. We discuss this register in the next section. The next three registers (`0x08-0x18`) are for interrupt control. Interrupts are not in the scope of this work, so these registers can be safely ignored.

After the interrupt registers, we can see the registers for the kernel arguments. The `vadd` kernel has three arguments. Both `in` and `out` are pointers to an array. The `in` array holds the input data whereas the `out` array stores the result. The third argument `n_elements` is a scalar value for the number of elements in the array. We can see how the registers `0x20` to `0x40` correspond to the arguments:

- Register `0x20` maps to argument `in`
- Register `0x30` maps to argument `out`
- Register `0x40` maps to argument `n_elements`

The first two arguments are arrays, so we need the start address of the corresponding buffer. The C program from the previous section writes the first physical address of each buffer to the console, which also happens to be the start address of the buffer. The last element is a scalar value, so we can directly write the number of vector elements into register `0x40`.

The registers can be accessed from the CPU over the Enzian Coherent Interconnect (ECI). The Linux kernel maps the registers to the following address range:

```
0x0000900000000000 - 0x000097dfffffffff
```

ECI defines this address range and passes it over the device tree to the Linux kernel. If a program reads a value from this range, the read request goes over ECI to the kernel and the kernel returns the value of the register. Similarly, if a program writes to a certain address in that range, the new value is transferred over ECI to the kernel.

To manually access the registers, we created another C program `reg` that is shown in Listing 3.10. Error handling is omitted.

Listing 3.10: C program to access the kernel registers

```
int main(int argc, char *argv[])
{
    int fd = 0, size = 4096;
    uint64_t paddr, reg, val;
    volatile uint64_t *ptr;

    reg = strtol(argv[1], NULL, 16);
    if (argc > 2)
        val = strtol(argv[2], NULL, 16);

    fd = open("/dev/mem", O_RDWR);
    paddr = 0x0000900000000000;
    ptr = mmap(NULL, size, PROT_READ | PROT_WRITE,
                           MAP_SHARED, fd, paddr);
    if (argc > 3) {
        ptr[reg / 8] = val;
    } else {
        val = ptr[reg / 8];
        printf("Value: %lx\n", val);
    }

    munmap(addr, size);
    close(fd);

    return EXIT_SUCCESS;
}
```

The program can be used to both read or write to a register. For reading a register, we have to specify the register offset as an argument and the program prints the content of that register. If we want to write to a register, we provide the offset and a value to should go to the register. The program works similarly for read and writes. After, parsing the input arguments, we open `/dev/mem` and get a file descriptor back. `/dev/mem` is a special file that provides access to the physical address space. Only users with root permission can access this file. Next, we pass the file descriptor and the

start address of the registers to `mmap`. `mmap` returns a pointer to that memory location and we can access the registers like an array. Note that we have to divide the register offset by eight because each register has a 64-bit word size.

Now we have all the required tools to configure and execute the `vadd` kernel.

Kernel Execution

In this section, we briefly summarize how we prepared the kernel for its first execution and run it. Initially, we started `buf` and the start address of the `in` and `out` buffer was printed on the console. We will denote the start address of `in` as `addr_in` and the start address of `out` as `addr_out`. The `buf` program runs now in the background and prints the output buffer periodically on the console.

Next, we configure the kernel arguments. We used the following commands to set the arguments:

```
./mem 0x20 <addr_in>
./mem 0x30 <addr_out>
./mem 0x40 <vector_size>
```

The first argument corresponds to the register offset as specified in Listing 3.9. The second argument is the value that will be written to the register. The kernel is now configured and ready for execution.

As previously mentioned, we can control the kernel execution with the content of register `0x0`. The `vadd` kernel uses the `AP_CTRL_CHAIN` execution model, which is the default model for Vitis kernels. Section 2.3.3 describes this execution model and the purpose of each bit in register `0x0`. The following command starts the kernel execution:

```
./mem 0x0 0x1
```

This command sets the first bit (`ap_start`) of register `0x0` and the `vadd` kernel starts reading the input data.

The `vadd` kernel reads the input vector over the AXI memory interface. The kernel writes the address in register `0x20`, the start address of the `in` buffer, and the size of the vector in register `0x40` to the AXI read address channel. This read request is converted into ECI packages and send to the memory controller of the CPU. The CPU responds with the requested data and the kernel saves the input vector in a BRAM buffer.

Once the input vector is available, the kernel starts executing the custom logic. Our custom logic is adding a constant number to each value of the input vector. The result of this computation is again stored in BRAM.

When the result is ready, the kernel writes the start address of the `out` buffer (register `0x30`) and the vector size to the write address channel of the AXI interface. Once the handshake with the CPU has completed, the kernel writes the result back to the CPU buffer of the AXI write data channel. The output buffer on the CPU is continuously printed by `buf`, so we can verify that the result of the computation is correct.

In summary, this was our first successful run of a kernel across all layers. Now that we have a detailed understanding of the kernel execution and know for certain that the hardware works, we can run the kernel using the XRT software stack.

3.3.3 Experiments with the complete XRT Stack

We almost have all the necessary parts to run our first OpenCL application on Enzian. However, the Linux drivers that are shipped with XRT are not compatible with Enzian. Therefore, we need to change these drivers so they recognize the Enzian hardware. We discuss these changes in the first part of this section. In the second part, we show how to run the `vadd` kernel with the XRT stack as an OpenCL application.

Linux Driver

XRT includes two Linux drivers that support different devices. The `xocl` driver is for PCIe based accelerator cards and the `zocl` driver supports edge devices. Figure 2.4 shows how these two drivers are embedded in the XRT software stack.

In general, `xocl` is more sophisticated as `zocl` because the former was developed for a cloud environment where multiple untrusted user have access to an accelerator. Hence, `xocl` includes several security features that are not relevant for a research computer like Enzian. Another issue with `xocl` is the scheduler that controls the kernels on the FPGA. `xocl` implements the scheduler as a separate program, that runs on the Microblaze processor inside the FPGA. On Enzian, the Microblaze processor runs already a program to control the ECI link, so the scheduler cannot use the Microblaze. The `zocl` driver, on the other hand, has a simpler design and the scheduler runs on the host CPU. Therefore, we decided to add Enzian support to the `zocl` driver.

We had to make two changes to the `zocl` driver. The first change was necessary because `zocl` did not recognize the Enzian hardware. Thus, the `probe` function of the driver was never called and `zocl` did not create the device nodes that XRT expected. The second change was related to the word size of the kernel control registers. As mentioned previously, XRT supports only 32-bit systems but Enzian has a 64-bit architecture. Hence, we had to change `zocl` to support registers with 64-bit words.

Device Probing The XRT installer added the `zocl` driver as a module to the Linux kernel on Enzian. As the next step, we wanted to load the driver and check if the `probe` function gets called. When a driver is added to the kernel, it is idle until a supported device is detected. Once a device shows up, the Linux kernel calls the `probe` function of the driver which supports the device. The `probe` function initializes the device and the driver is ready to interact with it.

We added a print statement to the `probe` function of the `zocl` driver to check if it probed successfully. Then we executed the following command to load the driver:

```
sudo modprobe zocl
```

The print statement did not show up in the kernel log so the `probe` function has not been called. This was expected because `zocl` has no support for Enzian.

The `probe` function is called if a certain device shows up in the *device tree*. Specifically, if the device tree contains a node called `zocl`. The ThunderX CPU exposes a device tree to the Linux kernel but the required node was missing. Xilinx devices, on the other hand, have this node in their device tree so the `zocl` driver probes correctly.

We present two solutions for this problem. We could either extend the device tree with the missing node or add a device manually to the kernel. On Enzian, the UEFI firmware generates the device tree. If we want to add the missing node to UEFI, we need to recompile and flash a new firmware image. However, flashing a new firmware image can be tricky.

Instead, we created a kernel module that registers a device in the kernel. The source code of the module is shown in Listing 3.11.

Listing 3.11: Kernel module that registers a platform device

```
static struct platform_device enzian = {
    .name = "zocl-drm"
};

static int __init enz_init(void)
{
    int ret;

    ret = platform_device_register(&enzian);
    if (ret)
        return ret;

    return 0;
```

```
}

static void __exit enz_exit (void)
{
    platform_device_unregister(&enzian);
}

module_init(enz_init);
module_exit(enz_exit);
```

When the module is loaded, the `enz_init()` function is executed. This function calls `platform_device_register()` which takes a `platform_device` struct as input. The struct has the `name` field set to `zocl-drm`.

If the kernel could not find a driver using the conventional methods, like device tree matching, it looks for a driver with the same name as the device. Our kernel module registers a device called `zocl-drm` which happens to be the name of the `zocl` driver. Therefore, the kernel matches our dummy device with the `zocl` driver and calls the `probe` function.

By using this kernel module, we were able to successfully initialize the `zocl` driver. In future releases, it might be beneficial to extend the device tree so the `zocl` driver is loaded at boot time.

Control Registers As we mentioned in Section 3.2.1, the control registers of the kernel have 64-bit words. However, the `zocl` driver supports only registers with 32-bit words. In particular, the driver uses the kernel functions `ioread32()` and `iowrite32()`. To support 64-bit words, we replaced these two functions with their 64-bit counterparts (`ioread64()` and `iowrite64()`). This breaks compatibility with existing Xilinx boards but is sufficient for our prototype. In the future, we could support Enzian and Xilinx boards and enable the correct functions with a build argument. The word size could also be converted in the FPGA shell (platform), but we did not explore this approach.

Another problem is the base address of the control registers. In Section 3.3.2 we discussed that the registers are mapped to a certain address in the CPU address space. The driver needs to know this base address to access the registers and to control the kernel. Normally, this base address is defined in the metadata of the device binary (xclbin). The driver extracts the base address from the metadata and initializes the registers. However, this address was not present in our device binaries. We suspect that the Enzian platform file is not correct and the Vitis compiler could not find the address during the build phase. The platform file format is not documented and we could not find the correct place in the platform file to specify the address. Instead,

we hard-coded the base address in the `zocl` driver so the API for accessing the registers is correctly initialized.

After we applied these two changes, the XRT stack was ready to run the `vadd` kernel.

Execute a kernel with XRT

In this section, we describe how we finally run the `vadd` kernel using XRT and OpenCL API calls.

We started this experiment with extracting the bitstream and programming the FPGA (see Section 3.3.2). After the bitstream is loaded, Enzian boots into Linux. Once Linux finished the booting, we loaded the `zocl` driver and the dummy device with the following two commands:

```
sudo modprobe zocl
sudo insmod enzian-dev.ko
```

For this experiment, we denote the device binary as `vadd.xclbin` and the host binary as `host.exe`. To start the `vadd` kernel, we executed the following command:

```
XRT_XILINX=/usr ./host.exe vadd.xclbin
```

This command executes the OpenCL API calls, shown in Listing 3.7. XRT translates the OpenCL calls into ioctl system calls. These ioctls are handled by the `zocl` driver which interacts with the kernel on the FPGA.

Listing 3.12 shows the output of our experiment.

Listing 3.12: Output of the `vadd` kernel executed with XRT

```
Found Platform
Platform Name: Xilinx
INFO: Reading vadd.xclbin
Loading: vadd.link.xclbin
Trying to program device[0]: edge
Device[0]: program successful!
TEST PASSED
```

The host application verifies the result of the `vadd` kernel by doing the same vector addition internally. If both results match, the string `TEST PASSED` is printed to the terminal.

In summary, we successfully run an OpenCL application with the XRT software stack which concludes this chapter. In the next chapter, we test several OpenCL API calls and discuss the limitations of the Enzian platform.

Chapter 4

Evaluation

The goal of our evaluation is to show that Enzian can execute OpenCL applications and that their performance is comparable to existing systems. We also want to point out the limitations of our platform, as a reference for future work.

We start with verifying that our implementation does not add a significant performance overhead to ECI. Then, we run a more sophisticated OpenCL application on Enzian and compare its performance to existing FPGA accelerators. Finally, we show which key features of OpenCL and Vitis work on Enzian.

4.1 ECI Performance

In this section, we want to show that our implementation does not add any significant overhead to ECI. To do so, we measured the throughput of ECI with two OpenCL kernels. One kernel reads data from the CPU memory to the FPGA block memory. Similarly, the other kernel writes data from the FPGA to the CPU memory. We compare these findings with measurements from a previous work [48] to show the overhead of OpenCL. These previous results were measured with an optimized FPGA design, that used the same underlying modules for accessing ECI as our implementation. So far these measurements achieved the best throughput on ECI.

We run both kernels for various transfer sizes and averaged the measurements over 10000 runs. Enzian has two links, each with a theoretical bandwidth of 15 GiB/s. The previous measurement restricted all traffic to a single link, so we also use just one link. The FPGA runs with a frequency of 300 MHz similar to the other measurement.

Figure 4.1 shows that our OpenCL implementation has a lower throughput for all transfer sizes. Our implementation achieved a maximum throughput

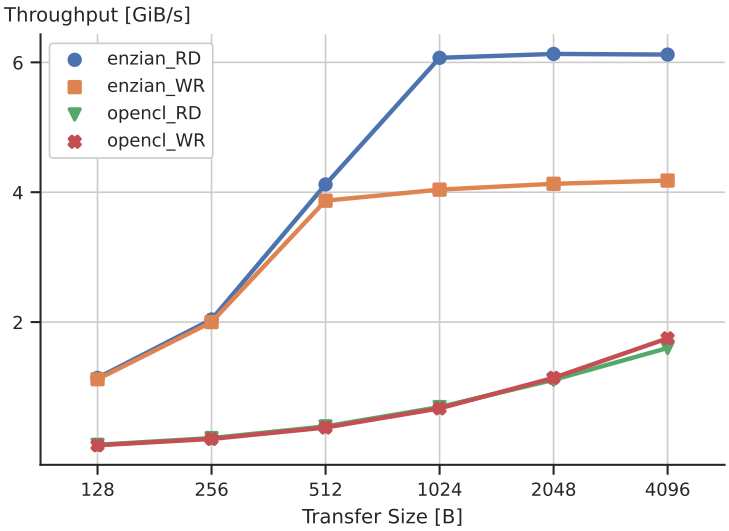

Figure 4.1: ECI Performance Plot

of 1.74 GiB/s for writes, whereas the optimized design went up to 6.12 GiB/s for reads.

Although these numbers suggest that our implementation adds a significant overhead, we identified three other reasons for the low throughput. First, we can only run one kernel on the FPGA. This is a limitation of our prototype, which we discovered during the evaluation in Section 4.4.1. One kernel can probably not saturate ECI due to the limited frequency of the FPGA. Second, the other FPGA design was manually optimized to achieve the best possible throughput on ECI. Our OpenCL kernels were compiled with Vitis using high-level synthesis. Although the Vitis compiler applied some optimizations like pipelining, it can never reach the same level of optimization as the other FPGA design. Finally, we could not run larger transfers than 4 KB because the kernels got stuck with larger values. We were not able to fix this issue in the limited time frame of this thesis. However, as we can see in Figure 4.1, the OpenCL throughput is not saturated and could increase with larger transfer sizes.

In sum, the explanation above suggests that OpenCL might not add a significant overhead to ECI. However, the findings above must be quantified by further measurements.

4.2 Matrix-Matrix Multiplication

So far we have only tested small OpenCL applications. With this experiment, we want to show that Enzian can also run a real-world application. To that end, we execute an existing OpenCL application [56] that performs a matrix-matrix multiplication on Enzian. We run the same application on a commercial FPGA accelerator as a reference for comparison.

We were able to compile and execute the application without any changes to the existing source code. In fact, we only had to change the data width of the kernel, so it accesses the matrices at cache line granularity, as described in Section 3.2.1. Other than that, the porting of the application to Enzian worked without any issues. This example shows that a user of our implementation can run existing OpenCL applications with minimal effort.

We compare Enzian with an Xilinx Alveo u250 accelerator card. The two systems have the same FPGA but a different link to the CPU. On Enzian, the CPU and FPGA are connected over ECI, which has a maximum bandwidth of 15 GiB/s (one link). The Alveo card uses a 16-lane PCIe Gen3 bus to communicate with the CPU. This bus has a maximum theoretical bandwidth of 16 GiB/s per direction [51].

Figure 4.2 shows that the throughput on the Alveo card is on average two-times higher than it is on Enzian. The Alveo card achieved a maximum throughput of 280.91 MiB/s whereas Enzian got up to 157.31 MiB/s, both with a 16×16 matrix. Note that we used small matrix sizes because of timing violations with larger matrices on Enzian.

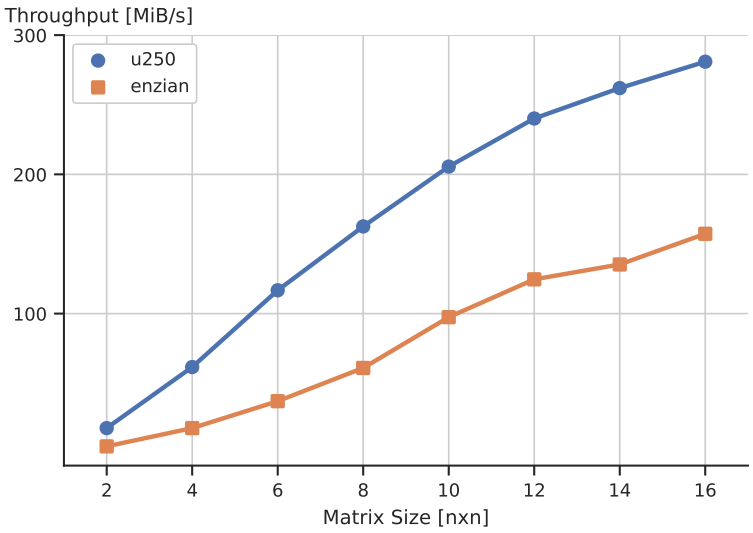

Figure 4.2: Matrix-Matrix Multiplication Performance Plot

Usually, the performance of matrix-matrix multiplication algorithms is reported in operations per second. However, we do not want to evaluate an algorithm but the overall performance of our system. Therefore, we measured the throughput, averaged over 10000 runs.

We expected a higher throughput for both Enzian and the Alveo card. Therefore, we calculated the approximated throughput of the kernel manually. We give an example for Enzian with a 16×16 4-byte integer matrix. We assume a memory latency of 500 ns per cache line. Each matrix is stored in eight contiguous 128-byte cache lines, so we have a latency of $8 \times 500\,ns = 4\,\mu s$ per matrix. The kernel reads two matrices and writes to one matrix, so the overall latency is $12\,\mu s$. With a transfer size of $3 \times 8 \times 128\,B = 3\,KiB$, we get a throughput of 244 MiB/s, which is in the range of the measured value (157.31 MiB/s).

Therefore, we concluded that kernel is not well optimized because the computation is dominated by memory operations. To achieve a higher throughput, we could either increase the matrix size or run multiple kernels in parallel. However, the former is not possible due to timing violations and the latter does not work because our prototype supports only one kernel.

If we look again at Figure 4.2, we can see that the Alveo card is on average 2x faster than Enzian. As the kernel is memory bound, we suspected that the Alveo card has a higher interconnect throughput. To confirm this, we also executed the benchmark from the previous section on the Alveo card.

Figure 4.3 shows that the Alveo card has a higher throughput than Enzian,

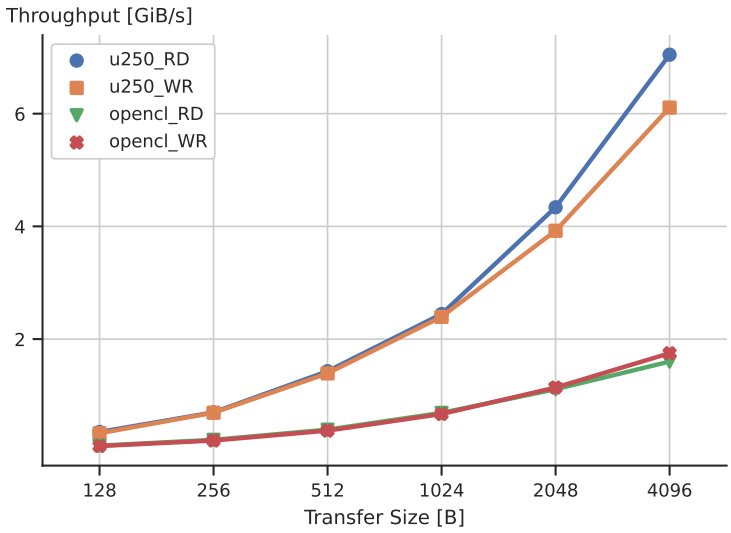

Figure 4.3: Interconnect Performance Plot

which supports our previous assumption. To bring the throughput on Enzian to the same level as the Alveo card, we must address the issues mentioned in Section 4.1.

In summary, we showed that Enzian can run a sophisticated OpenCL application and that the performance of our implementation is predictable.

4.3 OpenCL API

This section checks the functionality of the OpenCL API after we applied our changes. We first tried to run the OpenCL conformance test suite [27] to verify that our implementation is still valid. However, this test suite is quite complex and it would require a significant effort to build and run all tests. Therefore, we opted for a simpler approach. Instead of using a pre-defined test suite, we created several small programs to test the most important parts of the API. We identified these parts based on our experience, that we gathered from our previous experiments. We describe the tests for the platform layer and the runtime layer in a separate section.

4.3.1 Platform Layer

The API calls in the platform layer can be used to discover the available devices and to create a context which is the foundation of the runtime environment. We had already tested the platform layer during our experiments with the `vadd` kernel in Chapter 3 and were able to successfully create a context. Thus, the goal of this section is to collect various properties of the platform layer and verify if they match with our expectations.

We wrote a small test program (see Appendix B) to print various properties of the platform layer with the help of OpenCL calls. Listing 4.1 shows the output of the program.

Listing 4.1: Output of test program for the Platform Layer

```
1 platform(s) detected
CL_PLATFORM_NAME: Xilinx
CL_PLATFORM_VENDOR: Xilinx
CL_PLATFORM_VERSION: OpenCL 1.0
CL_PLATFORM_PROFILE: EMBEDDED_PROFILE
CL_PLATFORM_EXTENSIONS: cl_khr_icd

---

1 device(s) detected
CL_DEVICE_NAME: edge
CL_DEVICE_VERSION: OpenCL 1.0
```

```
CL_DEVICE_VENDOR: Xilinx
CL_DRIVER_VERSION: 1.0
CL_DEVICE_TYPE: CL_DEVICE_TYPE_ACCELERATOR
CL_DEVICE_EXTENSIONS:
CL_DEVICE_IMAGE_SUPPORT: 1
CL_DEVICE_COMPILER_AVAILABLE: 0
CL_DEVICE_ADDRESS_BITS: 64
CL_DEVICE_GLOBAL_MEM_SIZE: 4 GiB
CL_DEVICE_LOCAL_MEM_SIZE: 16 KiB
CL_DEVICE_MEM_BASE_ADDR_ALIGN: 1024
CL_DEVICE_MIN_DATA_TYPE_ALIGN_SIZE: 128
CL_DEVICE_MAX_MEM_ALLOC_SIZE: 128 MiB
CL_DEVICE_MAX_CLOCK_FREQUENCY: 100
CL_DEVICE_MAX_COMPUTE_UNITS: 0
CL_DEVICE_MAX_WORK_GROUP_SIZE: 4294967295
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS: 3

---

Context successfully created
```

The output is divided into three sections. The first section describes the platform. The next section outputs properties about devices and the final section indicates that the context was created successfully.

The program detected one platform on Enzian that supports the *embedded profile* of OpenCL 1.0. The embedded profile is a subset of the OpenCL specification with relaxed floating-point requirements for embedded devices. The platform has the `cl_khr_icd` extension enabled which is the Installable Client Driver (ICD) (Section 3). In conclusion, there seems to be no error in the platform properties and the platform should work as expected.

For the device, on the other hand, we expected different values. For instance, the device name (`CL_DEVICE_NAME`) is not correct correct so it is not possible to explicitly filter for Enzian devices. We expected that the size of the global memory (`CL_DEVICE_GLOBAL_MEM_SIZE`) is equal to the size of the CMA region that is allocated by the Linux kernel during boot time. However, it seems that Xilinx hard-coded the global memory size to 4 GiB. Note that we converted the numbers from bytes to a readable format. The size of the local memory (`CL_DEVICE_LOCAL_MEM_SIZE`) is 16 KiB but we do not know if this value is correct. We could not find any Xilinx documentation that defines the location of the local memory. Other OpenCL implementations define the caches close to the CPU/GPU as local memory but this definition is not applicable to FPGAs. The maximum clock frequency and the maximum number of compute units are also not correct. The FPGA on Enzian runs

with a clock frequency of 300 MHz instead of 100 MHz. Currently, the Enzian platform supports only one compute unit so the maximum number of compute units should be one and not zero. Based on our understanding, the other properties seemed to be correct.

The last section just confirms that the program created a context and that no error occurred during the test.

To conclude, the platform layer of our implementation is not fully compatible with the OpenCL standard. In particular, the device properties do not agree with the Enzian specification. These incorrect properties could prevent certain OpenCL applications from running and should be addressed in future work.

4.3.2 Runtime Layer

We have already tested the OpenCL runtime layer during our experiments with the `vadd` kernel and during the measurements at the beginning of this chapter. Therefore, we did not write a separate program to test the runtime layer. Instead, we summarize the limitations of the runtime layer, that we encountered during our previous experiments. We go over each concept of the runtime layer separately, and point out the shortcomings of the Enzian platform.

Program

OpenCL has two workflows to get a reference to a program. It can either be compiled from the source code at runtime or we can provide a binary representation of the program so no runtime compilation is necessary. Runtime compilation is usually the preferred method to generate a program, because the compiler can apply runtime optimizations. However, Vitis supports only the binary workflow due to the long build time of FPGA bitstreams. This was also indicated in the output of the test program from the previous section (see Listing 4.1). The device property `CL_DEVICE_COMPILER_AVAILABLE` was zero, which means that no runtime compiler is available. This is not a limitation of the Enzian platform, but just a fact that Enzian users need to be aware of.

Nevertheless, there is a feature missing in our implementation, that normally exists in other OpenCL implementations. As we mentioned in Section 3.3.2, we had to program the FPGA with Vivado. This interrupted our workflow because we had to switch to Vivado each time we run a different application. The programming of a device is usually transparent to the end user in other OpenCL implementations. For example, Vitis programs the FPGA of supported devices automatically when the user starts an OpenCL program. We argue that this feature should be implemented in future revisions of the Enzian platform, because it improves the usability of Enzian significantly.

Apart from the previous issue, we could not find any other limitation regarding the program concept.

Kernel

During our previous experiments, we created a number of kernel objects and did not encounter any error or unexpected behaviour. However, the current version of Enzian supports only one kernel, because this simplifies the implementation of the FPGA shell. OpenCL defines that a program can have multiple kernels, so our implementation is not compatible with the OpenCL specification. We cover this limitation in more detail during the evaluation of Vitis in Section 4.4.1.

Command Queue

Again we used the command queue during our previous experiments so we can submit commands to the queue and communicate with the FPGA. There are two features of the command queue that we had not tested before. The first feature is out-of-order execution and the other feature is profiling. Section 2.1.1 covers these features in detail.

As we mentioned in the previous section, our implementation does not support multiple kernels. Although we could create a command queue with out-of-order execution enabled, it would not increase the performance of an application. The command queue has only one kernel available, so it can send the commands only to that kernel, which results in a serial execution. To confirm this assumption, we ran some of our benchmarks in out-of-order mode and could not see any performance improvements.

The profiling feature collects comprehensive information about a kernel execution. This data is particularly useful for finding performance bottlenecks. A user has two possibilities to access the profiling data in our application. The data can either be collected with the OpenCL API or with Vitis.

The OpenCL API provides access to the profiling information via events. Each function that enqueues a command takes an event object as input. This event object can then be used to measure the execution time of a command. For example, if we want to measure the execution time of a kernel, we pass an event object to `enqueueTask`. After the execution has finished, we can use `getProfilingInfo` to query the start and end time of the execution in nanoseconds. To test this feature on Enzian, we created another version of the bandwidth benchmark with these profiling events. The execution times measured with the profiling events were in the range of our initial measurements. Thus, we conclude that OpenCL profiling is supported by Enzian.

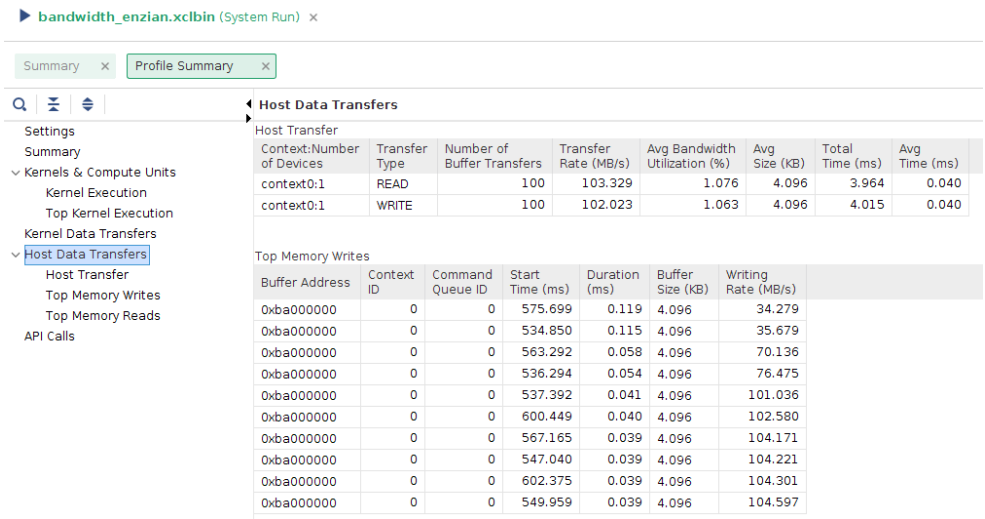

Figure 4.4: Excerpt from a profiling report in the Vitis Analyzer

Vitis has various profiling tools to inspect the build process and the execution of a kernel. We focus on execution profiling in this section. Vitis generates two different reports with profiling data. The first report can be generated from any application and contains general information about the execution. The second report requires some special IPs, that must be added to the device binary during the build process. Vitis uses these IPs to collect additional data for an in-depth report of the kernel execution.

We tried to generate both reports for the bandwidth benchmark. For the first report, we passed a flag to the Xilinx Runtime (XRT) and XRT collected the runtime data. Then, we used the Vitis Analyzer to visualize the report. Figure 4.4 shows an excerpt of the report. The report lists all transfers from the host to global memory. These numbers provide little insight because Enzian uses the CPU DDR as global memory. The interesting part is the *Kernel Data Transfers* section. This section is above the currently open *Host Data Transfers* section on the left hand side. However, this section is only available in the other report and contains no data in the report we just generated.

As we mentioned before, the second report requires additional IPs in the FPGA design to collect data. We tried to add these IPs but the Vitis compiler aborted the build with the following error:

```
ERROR: [VPL 41-237] Bus Interface property DATA_WIDTH
    does not match between
    /System_DPA/dpa_mon1/MON_S_AXI(32) and
    /axi_switch_kernel_control/M01_AXI(64)
```

The error indicates that the data width of the additional IPs and the Enzian platform does not match. We already solved this problem for the kernel control interface (see Section 3.2.1). However, we did not fix this problem for the profiling, so Enzian does not support extended profiling reports.

4.4 Vitis Features

This section evaluates other important features of the Vitis framework which are not part of the OpenCL standard. We selected these features based on our experience from working with Vitis throughout this thesis. This selection is by no means complete, but it should give the reader a first impression on how well Enzian is supported by Vitis.

4.4.1 Multiple Kernels

In our previous experiments, a program had only one kernel and Vitis created one instance of this kernel in the FPGA fabric. However, certain applications may benefit from having multiple instances of the same kernel (spatial parallelism) or from distributing a task across different kernels (temporal parallelism). Vitis supports both types of parallelism. First, the Vitis linker can create multiple instance of the same kernel in the FPGA fabric. The host application can send commands to each kernel instance individually. Second, it is possible to add different kernel functions to a program. Each kernel function has to be marked with a distinct `__kernel` attribute. Vitis can even connect these kernels together over a streaming interface. A developer could use this technique to build complex pipelines that can significantly improve performance.

We conducted two experiments to check if Enzian supports multiple kernels. In the first experiment, we reused the `vadd` kernel and passed additional flags to the Vitis linker. These flags instructed the linker to create multiple instances of the `vadd` kernel. However, the build did not complete successfully because there were not enough memory interfaces available. As mentioned in Section 3.1, the Enzian platform has only one memory interface to access the shared memory. Further, multiple kernels cannot share a memory interface because of the underlying protocol. The linker could not find a memory interface for each kernel and, thus, aborted the build.

In the second experiment, we observed a similar behaviour. We created a test program that has two function with different `__kernel` attributes. Vitis successfully compiled the program but during the platform linking we encountered the same error message as in the first experiment.

To summarize, Vitis on Enzian does currently not support multiple kernel instances. In Section 6.1 we briefly cover how to add multiple memory

interfaces, which enables multiple kernel support.

4.4.2 Emulation

Vitis provides an emulation environment to quickly build a kernel and verify that it works. The Vitis compiler has two different build targets to create an emulation binary from a kernel. The first build target is for software emulation. The compiler converts the kernel into a C binary, which can be run on a normal x86 system. Software emulation is useful to verify the program logic but it does not provide a cycle-accurate emulation. The other build target is for hardware emulation. The kernel code is compiled into an RTL model and is run in the Vivado simulator. This emulation takes more time but provides a cycle-accurate view of the kernel logic. Overall, emulation is an essential feature of an FPGA framework so it would be beneficial if Enzian supports it.

The Vitis documentation states that hardware emulation requires some extra IPs in a custom platform [54]. We did not add these IP to the Enzian platform. Thus, Enzian does not support hardware emulation and we did not run any experiments related to hardware emulation.

To verify that software emulation is supported by Enzian, we compiled the `vadd` kernel with the build target `sw_emu`. This build target creates a dummy device binary (`vadd.xclbin`), which contains a C model of the kernel. Then, we used the command line tool `emconfigutil` to create a configuration file, that is necessary for the emulation. After these additional steps, we reused the host binary from Section 3.2.2, to start the emulation on our internal build server. The emulation of the `vadd` kernel executed as if it would run on Enzian and produced the same result as on real hardware.

In conclusion, this experiment showed that software emulation with the Enzian platform works as expected. However, hardware emulation is not supported by Enzian and requires extra steps to implement.

4.4.3 Kernel Programming Languages

Vitis supports kernels written in C/C++, OpenCL C or RTL. During previous experiments, we created both C/C++ and OpenCL C kernels. Therefore, we can confirm that Enzian supports kernels written in those two languages.

We did not verify that Enzian can run an RTL kernel, but we are confident that RTL kernels are also supported. RTL kernels are created in Vivado and exported to a Xilinx object file (`*.xo`) with the `package_xo` command. Afterwards, the object file is passed to the linker and connected to the platform. This is the same process for every kernel independent of the implementation language. We extensively tested the linking phase and we

did not change anything related to Vivado. Therefore, Enzian should support RTL kernels to the same extend as C/C++ or OpenCL C kernels.

4.4.4 Streaming Data Transfer

Vitis has a programming model that supports direct streaming of data between to endpoints. This programming model is usually more efficient, because it is not necessary to migrate the complete data set to the FPGA before the computation can start. With data streaming, a kernel can start executing as soon as the first data arrives. Hence, it would be useful if Enzian supports streaming data transfers, in particular, for high-performance applications.

There are two types of streaming data transfers. The data can either be streamed between the kernel and the host (H2K) or between two kernels (K2K). The first type enables a fast communication between the host and a kernel with minimal overhead. With the other type, kernels can directly exchange data with each other without transferring data back to global memory.

The Vitis documentation does not describe a method to add support for H2K streaming data transfer to a custom platform. Due to the lack of any other documentation, we did not implement H2K streaming data for Enzian. K2K streaming could potentially work on Enzian. However, the current implementation does not support multiple kernels so we were unable to test K2K streaming.

4.4.5 Command Line Tools

The Vitis software suite offers the following command line tools:

`v++` The Vitis compiler to create Xilinx object files from kernels and link them together for the device binary.

`emconfigutil` This tool creates a configuration file for the hardware or software emulation.

`platforminfo` This utility tool reports platform metadata including information about interfaces, clocks and the memory topology.

`kernelinfo` This tool displays information about Xilinx object files.

`xclbinutil` This tool displays information about device binaries.

`xbmgmt` / `xbutil` These tools have many options for managing the FPGA device.

In this section, we briefly cover which tool is supported by Enzian. We did not test every feature of the Vitis compiler (`v++`), but the tool worked

reliably during our experiments. Therefore, we conclude that `v++` supports the Enzian platform. We also tested the `emconfigutil` during our experiment with software emulation and it worked as expected with the Enzian platform.

We run the `platforminfo` tool with the platform file which we created in Section 3.1. The tool produced the correct output (see Listing C.1) and printed all relevant information about the platform. The `kernelinfo` tool worked also as expected. Listing C.2 shows the output of the `vadd` kernel.

During our experiments, we extensively used `xclbinutil` to inspect and manipulate various device binaries. We have never encountered any issues so the tool fully supports Enzian device binaries.

According to the Vitis documentation, the `xbmgmt` tool does not work with custom platforms like Enzian. This tool was also not included in the Debian package so we could not install it on Enzian. The `xbutil` tool, on the other hand, was part of the Debian package so we installed it on Enzian. However, as we tried to query information about the Enzian platform, it exited with an error. We suspect that the Linux driver caused this error. Most likely, the driver was unable to read some information about the hardware. The support of this tools is not essential to run a Vitis application on Enzian.

Chapter 5

Related Work

Programming heterogeneous accelerators, especially CPU-FPGA based systems, remains a challenge and is an active field of research. This chapter surveys various FPGA programming frameworks with focus on OpenCL. We begin with an overview of different OpenCL implementations. Apart from commercial implementations, we also cover projects from academia and the open-source community. Finally, we discuss why OpenCL is not always a good fit for FPGAs and present alternative programming models.

Almost all significant CPU or GPU manufacturers provide an OpenCL implementation for their devices [21, 3, 40]. These implementations are actively maintained and are usually compliant with the latest OpenCL version. Each implementation supports only the devices of a specific vendor and these devices are limited to CPUs or GPUs.

We discovered two commercial OpenCL implementations specifically for FPGAs. First, Xilinx supports OpenCL as part of their Vitis software suite [62]. The Xilinx Runtime (XRT) implements the OpenCL API on the host, and a compiler tool chain converts kernels, written in OpenCL C, into a device binary. Second, Intel has the FPGA SDK for OpenCL [20] that provides similar functionality as Vitis. We could not find OpenCL implementations from other FPGA manufacturers. The two previous implementations are not conformant with the latest OpenCL version. Thus, they do not support the latest features of the API.

Apart from the commercial implementations, there are also community-driven OpenCL implementations. FreeOCL [6] is an OpenCL implementation for CPUs and the source code is licensed under GPLv3. However, the project seems not to be actively maintained anymore. PoCL [25], on the other hand, is an active project that has OpenCL support for various CPUs and GPUs. PoCL implements most of the OpenCL API calls and provides a hardware abstraction layer, so adding support for new devices is straightforward.

Similar to the commercial FPGA implementation, the community solutions do usually not support the latest OpenCL features.

There are also several academic OpenCL implementations. An early work of Czajkowski et al. [10] describes a prototype of an OpenCL compiler for FPGAs. They used the Altera OpenCL library for the host runtime environment and created a custom compiler based on LLVM [30] to build the FPGA binary. This work is essential as the device binary part of commercial OpenCL implementations is often not disclosed.

Mirian et al. [36] created UT-OCL, an OpenCL framework for embedded systems using FPGAs. UT-OCL provides the complete OpenCL software stack and is similar to commercial frameworks, such as Xilinx Vitis. Although UT-OCL is targeting embedded systems, it provides valuable insights into various details of an OpenCL implementation. The author published other interesting work [34, 35, 33, 32] that explores different aspects of CPU/FPGA based systems.

Another line of research tries to program FPGAs with high-level languages, such as Java or Python. These tools can be divided into two categories. The first category converts a kernel, written in a high-level language, into OpenCL code and uses an OpenCL implementation to run the kernel on an FPGA. For example, Aparapi [4] is a framework that converts Java to OpenCL code that can then be executed on an FPGA [45, 44]. PyOpenCL [28, 47] is another example for Python. The second category of tools creates HDL code from a high-level language [22, 46]. The HDL code is then synthesized with a vendor tool chain.

While OpenCL makes FPGAs more accessible, it does not always leverage the full capabilities of an FPGA [23, 38]. In particular, there is a mismatch between the OpenCL programming model and the FPGA architecture. This mismatch can be explained by the fact that OpenCL was initially designed for GPUs [37].

Despite the popularity of OpenCL for FPGA programming, some companies moved away from the OpenCL standard. For example, Apple, the creator of OpenCL, deprecated OpenCL in favor of their Metal API [50]. Another example is Xilinx that uses a C++ API instead of OpenCL in the most recent version of their Vitis software suite [66].

During our survey, we could not find many alternative programming models for FPGAs. A recent work of Huan et al. [18] explores different execution strategies for heterogeneous CPU/FPGA systems and provides valuable insights into the future of FPGA programming models.

Chapter 6

Conclusion

In this thesis, we added support for OpenCL applications to Enzian. This shows that Enzian is a flexible research platform that can support most functionality of existing Xilinx boards. In addition, we improved the usability of Enzian by opening the platform for the large community of OpenCL developers.

Instead of creating a new OpenCL implementation, we reused an existing OpenCL implementation from Xilinx, which is part of the Vitis framework. We adapted Vitis so it recognizes Enzian as a supported device. Future Enzian users can now benefit from the full functionality of the Vitis software suite. Apart from OpenCL applications, Vitis supports accelerated applications written in C/C++ or Python and provides a set of performance-optimized libraries for various applications.

Nevertheless, Vitis on Enzian does not provide the same experience as if it runs on an officially supported platform. We tried to make the Vitis workflow as streamlined as possible but there are still some limitations. In the following sections, we briefly address these limitations and discuss compelling ideas for future work.

6.1 Memory Topology

In Vitis, it is possible to define the memory topology of a platform like Enzian. The compiler can then select the memory subsystems with the best performance for a given application. In addition, we can use different ports of a memory controller to increase performance or to run multiple kernels concurrently. For example, the Alveo accelerator cards have multiple DDR banks so a kernel can read two large vectors from different banks simultaneously, which improves performance. Another example is a Computer Vision pipeline that runs on the FPGA and uses a different kernel for each pipeline

stage. A developer can also decide if, for example, the data should be stored in the CPU DDR or FPGA DDR. Apparently, a correct memory topology model has many advantages.

However, when we started with the implementation, we could only use the CPU DDR as shared memory. The module to access FPGA DDR was still in development. This limited our design space and affected the performance benchmarks because we could not place the data close to the FPGA. In addition, we added only one memory interface to the Enzian platform. This simplified the initial development but also added two restrictions. First, there is a performance penalty because the same memory port handles all memory requests. Second, it is not possible to run multiple kernels on Enzian. Vitis needs a separate memory port for each kernel, or otherwise, the build will fail.

Therefore, we suggest for future revisions to model the memory topology of Enzian accurately. This includes adding support for the FPGA DDR and splitting the shared memory into multiple ports.

6.2 Sub-Cache Line Access

Our implementation supports only kernels that access complete cache lines. Sub-cache line reads or writes are not possible. This is a restriction of Enzian DMA. We had to rewrite most kernels during our experiments so they access the data in cache line chunks. This makes it hard to port existing OpenCL applications and introduces unnecessary bugs. Therefore, we suggest providing a module that allows sub-cache line read or writes. For example, a kernel can send a request to read five 32-bit integers from a start address, and the module handles all details with ECI and cache lines.

6.3 Program the FPGA with Vitis

Vitis programs the FPGA if an accelerated application is executed for the first time. The Linux driver `zocl`, that is part of Vitis, programs the FPGA with the Linux FPGA Manager framework. Our implementation currently does not support the FPGA Manager. Instead, we used the Hardware Manager in Vivado to program the FPGA. This did interrupt the Vitis workflow because we had to open Vivado each time we run a different application. Future work could make Enzian compatible with the Linux FPGA Manager. Other projects might also benefit because they can load a bitstream from Linux, instead of using special tools like Vivado.

After programming the FPGA with Vitis is possible, we could also implement Dynamic Function eXchange (DFX) [53]. DFX allows to reconfigure parts of the FPGA within an active design and has additional security features. With

DFX in place, Enzian could provide similar functionality as the Alveo cards in a cloud environment.

6.4 Device Tree Probing

The Vitis Linux driver is usually loaded via Device Tree bindings on officially supported platforms. In our implementation, we created a kernel module that adds a dummy device so that the Linux driver probes successfully. It would be little effort to extend the Enzian UEFI image and add the necessary nodes to the device tree. After the correct nodes are present, the Linux driver would probe automatically and the Vitis workflow would not be interrupted by loading an additional kernel module.

6.5 Shared Virtual Memory (SVM)

The architecture of Enzian, with its coherent interconnect, is well suited to implement Shared Virtual Memory (SVM). The basic idea of SVM is that the host and the device have a common virtual address space. This makes it possible to share complex data structures such as lists or trees between the host and a kernel. SVM does not only simplify OpenCL programming but could also increase performance. The OpenCL standard introduced SVM in Version 2.0 [16], and Vitis supports SVM on some Edge devices. Enabling SVM on Enzian would allow for a fine-grained interaction between the host and the kernel. This fine-grained interaction could give rise to novel programming patterns for CPU/FPGA-based systems.

Appendix A

Acronyms

XSA Xilinx Support Archive . . . 25
BSP Board Support Package . . . 25
PS Processing System . . . 28
PL Programmable Logic . . . 28
XSCT Xilinx Software Command-Line Tool . . . 38
CU Compute Unit . . . 15
SLR Super Logic Region . . . 15
XRT Xilinx Runtime . . . 15
ICD Installable Client Driver . . . 16
DFX Dynamic Function eXchange . . . 82
BO Buffer Object . . . 18
CMA Continuous Memory Allocator . . . 18
KDS Kernel Domain Scheduler . . . 18
ECI Enzian Coherent Interconnect . . . 10
DKMS Dynamic Kernel Module Support . . . 53
SVM Shared Virtual Memory . . . 83

Appendix B

Example Code

Listing B.1: Example of a host application [60]

```
#define cl_hpp_cl_1_2_default_build
#define cl_hpp_enable_exceptions

#include <vector>
#include <iostream>
#include <cl/opencl.hpp>

char *read_binary_file(const std::string
                &xclbin_file_name, unsigned &nb);

int main()
{
    std::string binaryFile = argv[1];

    std::vector<cl::Platform> platforms;
    cl::Platform::get(&platforms);
    cl::Platform platform = platforms.front();

    std::vector<cl::Device> devices;
    platform.getDevices(CL_DEVICE_TYPE_ACCELERATOR,
                                  &devices);
    cl::Device device = devices.front();

    cl::Context context(device);

    cl::CommandQueue q(context, device);

    int fileSize;
```

```
    char *file = read_binary_file(binaryFile,
                                          fileSize);
    cl::Program::Binaries bins{{file, fileSize}};
    cl::Program program(context, device, bins);

    cl::Kernel kernel(program, "vadd");
    int size = 32;
    std::vector<int> in(size);
    std::vector<int> out(size);
    std::fill(in.begin(), in.end(), 42);
    std::fill(out.begin(), out.end(), 0);

    int size_in_bytes = sizeof(int) * size;
    cl::Buffer in_buf(context, CL_MEM_READ_ONLY,
                                    size_in_bytes);
    cl::Buffer out_buf(context, CL_MEM_WRITE_ONLY,
                                    size_in_bytes);

    kernel.setArg(0, in_buf);
    kernel.setArg(1, out_buf);
    kernel.setArg(2, size);

    q.enqueueWriteBuffer(in_buf, CL_TRUE, 0,
                         size_in_bytes, in.data());

    q.enqueueTask(kernel);
    q.finish();

    q.enqueueReadBuffer(out_buf, CL_TRUE, 0,
                         size_in_bytes, out.data());

    verify_result(&in, &out);
}

char *read_binary_file(const std::string
                    &xclbin_file_name, unsigned &nb)
{
    std::ifstream bin_file(xclbin_file_name.c_str(),
                           std::ifstream::binary);
    bin_file.seekg(0, bin_file.end);
    nb = bin_file.tellg();
    bin_file.seekg(0, bin_file.beg);
    char *buf = new char[nb];
    bin_file.read(buf, nb);
```

```
    return buf;
}
```

Listing B.2: Program to test the OpenCL platform layer

```
// g++ -Wall cl_platform.cpp -o ./cl_platform -
   lOpenCL

#define CL_HPP_CL_1_2_DEFAULT_BUILD
#define CL_HPP_TARGET_OPENCL_VERSION 120
#define CL_HPP_MINIMUM_OPENCL_VERSION 120
#define CL_USE_DEPRECATED_OPENCL_1_2_APIS
#define CL_HPP_ENABLE_EXCEPTIONS

#include <vector>
#include <iostream>
#include <CL/cl2.hpp>

int main()
{
    std::vector<cl::Platform> platforms;
    cl::Platform::get(&platforms);
    std::cout << platforms.size() << " platform(s)
       detected" << std::endl;
    for (auto p : platforms) {
        std::cout << "CL_PLATFORM_NAME: " << p.
           getInfo<CL_PLATFORM_NAME>() << std::endl;
        std::cout << "CL_PLATFORM_VENDOR: " << p.
           getInfo<CL_PLATFORM_VENDOR>() << std::endl
           ;
        std::cout << "CL_PLATFORM_VERSION: " << p.
           getInfo<CL_PLATFORM_VERSION>() << std::
           endl;
        std::cout << "CL_PLATFORM_PROFILE: " << p.
           getInfo<CL_PLATFORM_PROFILE>() << std::
           endl;
        std::cout << "CL_PLATFORM_EXTENSIONS: " << p.
           getInfo<CL_PLATFORM_EXTENSIONS>() << std::
           endl;
    }
    cl::Platform platform = platforms.front();

    std::cout << std::endl << "---" << std::endl<<
       std::endl;
```

```
std::vector<cl::Device> devices;
platform.getDevices(CL_DEVICE_TYPE_ALL, &devices)
    ;
std::cout << devices.size() << " device(s)
    detected" << std::endl;
for (auto d : devices) {
    std::cout << "CL_DEVICE_NAME: " << d.getInfo<
        CL_DEVICE_NAME>() << std::endl;
    std::cout << "CL_DEVICE_AVAILABLE: " << d.
        getInfo<CL_DEVICE_AVAILABLE>() << std::
        endl;
    std::cout << "CL_DEVICE_VERSION: " << d.
        getInfo<CL_DEVICE_VERSION>() << std::endl;
    std::cout << "CL_DEVICE_VENDOR: " << d.
        getInfo<CL_DEVICE_VENDOR>() << std::endl;
    std::cout << "CL_DRIVER_VERSION: " << d.
        getInfo<CL_DRIVER_VERSION>() << std::endl;
    std::cout << "CL_DEVICE_TYPE: " << "
        CL_DEVICE_TYPE_ACCELERATOR" << std::endl;
    std::cout << "CL_DEVICE_EXTENSIONS: " << d.
        getInfo<CL_DEVICE_EXTENSIONS>() << std::
        endl;
    std::cout << "CL_DEVICE_IMAGE_SUPPORT: " << d
        .getInfo<CL_DEVICE_IMAGE_SUPPORT>() << std
        ::endl;
    std::cout << "CL_DEVICE_COMPILER_AVAILABLE: "
         << d.getInfo<CL_DEVICE_COMPILER_AVAILABLE
        >() << std::endl;
    std::cout << "CL_DEVICE_ADDRESS_BITS: " << d.
        getInfo<CL_DEVICE_ADDRESS_BITS>() << std::
        endl;
    std::cout << "CL_DEVICE_GLOBAL_MEM_SIZE: " <<
         d.getInfo<CL_DEVICE_GLOBAL_MEM_SIZE>() <<
         std::endl;
    std::cout << "CL_DEVICE_LOCAL_MEM_SIZE: " <<
        d.getInfo<CL_DEVICE_LOCAL_MEM_SIZE>() <<
        std::endl;
    std::cout << "CL_DEVICE_MAX_CLOCK_FREQUENCY:
        " << d.getInfo<
        CL_DEVICE_MAX_CLOCK_FREQUENCY>() << std::
        endl;
    std::cout << "CL_DEVICE_MAX_COMPUTE_UNITS: "
        << d.getInfo<CL_DEVICE_MAX_COMPUTE_UNITS
        >() << std::endl;
```

```
        std::cout << "CL_DEVICE_MAX_MEM_ALLOC_SIZE: "
            << d.getInfo<CL_DEVICE_MAX_MEM_ALLOC_SIZE
           >() << std::endl;
        std::cout << "CL_DEVICE_MAX_WORK_GROUP_SIZE:
           " << d.getInfo<
           CL_DEVICE_MAX_WORK_GROUP_SIZE>() << std::
           endl;
        std::cout << "
           CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS: " << d
           .getInfo<
           CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS>() <<
           std::endl;
        std::cout << "CL_DEVICE_MEM_BASE_ADDR_ALIGN:
           " << d.getInfo<
           CL_DEVICE_MEM_BASE_ADDR_ALIGN>() << std::
           endl;
        std::cout << "
           CL_DEVICE_MIN_DATA_TYPE_ALIGN_SIZE: " << d
           .getInfo<
           CL_DEVICE_MIN_DATA_TYPE_ALIGN_SIZE>() <<
           std::endl;
    }
    cl::Device device = devices.front();

    std::cout << std::endl << "---" << std::endl<<
       std::endl;

    cl::Context context(device);
    std::cout << "Context successfully created" <<
       std::endl;
}
```

Appendix C

CLI Tools Output

Listing C.1: Output of `platforminfo enzian.xpfm`

```
==========================
Basic Platform Information
==========================
Platform:           enzian_v2_top
File:               enzian.xpfm
Description:
enzian_v2_top


====================================
Hardware Platform (Shell) Information
====================================
Vendor:                 ethz
Board:                  enzian
Name:                   enzian
Version:                0.0
Generated Version:      2020.1
Software Emulation:     1
Hardware Emulation:     0
FPGA Family:            virtexuplus
FPGA Device:            xcvu9p
Board Vendor:           xilinx.com
Board Name:             xilinx.com:vcu118:2.0
Board Part:             xcvu9p-flga2104-2L-e
Maximum Number of CU:   60

=================
Clock Information
```

```
=================
  Default Clock Index: 0
  Clock Index:         0
    Frequency:         300.000000
  Clock Index:         1
    Frequency:         300.000000

==================
Memory Information
==================
  Bus SP Tag: gmem
======================
Feature ROM Information
======================

============================
Software Platform Information
============================
Number of Runtimes:            1
Default System Configuration:  enzian_v2_top
System Configurations:
  System Config Name:                        enzian
  System Config Description:                 enzian
  System Config Default Processor Group:     domain_cpu
  System Config Default Boot Image:          standard
  System Config Is QEMU Supported:           0
  System Config Processor Groups:
    Processor Group Name:      domain_cpu
    Processor Group CPU Type:  cpu
    Processor Group OS Name:   standalone
  System Config Boot Images:
    Boot Image Name:           standard
    Boot Image Type:
    Boot Image Data:
    Boot Image Boot Mode:
    Boot Image RootFileSystem:
    Boot Image Mount Path:
    Boot Image QEMU Args:
    Boot Image QEMU Boot:
    Boot Image QEMU Dev Tree:
Supported Runtimes:
  Runtime: C/C++
```

Listing C.2: Output of `kernelinfo vadd.xo`

```
=== Kernel Definition ===
name: vadd
language: c
vlnv: xilinx.com:hls:vadd:1.0
preferredWorkGroupSizeMultiple: 0
workGroupSize: 1
debug: true
containsDebugDir: 1
sourceFile: vadd/cpu_sources/vadd.cpp
=== Arg ===
name: in
addressQualifier: 1
id: 0
port: M_AXI_GMEM
size: 0x8
offset: 0x20
hostOffset: 0x0
hostSize: 0x8
type: void*
=== Arg ===
name: out
addressQualifier: 1
id: 1
port: M_AXI_GMEM
size: 0x8
offset: 0x30
hostOffset: 0x0
hostSize: 0x8
type: void*
=== Arg ===
name: size
addressQualifier: 0
id: 2
port: S_AXI_CONTROL
size: 0x8
offset: 0x40
hostOffset: 0x0
hostSize: 0x4
type: unsigned long long
=== Port ===
name: M_AXI_GMEM
mode: master
```

```
range: 0xFFFFFFFF
dataWidth: 32
portType: addressable
base: 0x0
=== Port ===
name: S_AXI_CONTROL
mode: slave
range: 0x1000
dataWidth: 64
portType: addressable
base: 0x0
```

Bibliography

[1] Gustavo Alonso, Timothy Roscoe, David Cock, Mohsen Ewaida, Kaan Kara, Dario Korolija, David Sidler, and Zeke Wang. Tackling hardware/-software co-design from a database perspective. 2020.

[2] Amazon. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/. Accessed: 2021-06-23.

[3] AMD. AMD ROCm. https://rocmdocs.amd.com/en/latest/. Accessed: 2021-08-19.

[4] Aparapi. Aparapi. https://aparapi.com/. Accessed: 2021-08-19.

[5] ARM. AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite. https://developer.arm.com/documentation/ihi0022/e/?lang=en. Accessed: 2021-07-13.

[6] Roland Brochard. FreeOCL. http://www.zuzuf.net/FreeOCL/. Accessed: 2021-08-19.

[7] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49. IEEE Press, 2016.

[8] Linux Community. Linux Direct Rendering Manager (DRM). https://www.kernel.org/doc/html/v5.4/gpu/drm-mm.html. Accessed: 2021-08-02.

[9] Linux Community. Linux FPGA subsystem. https://www.kernel.org/doc/html/latest/driver-api/fpga/intro.html. Accessed: 2021-08-12.

[10] Tomasz S Czajkowski, David Neto, Michael Kinsner, Utku Aydonat, Jason Wong, Dmitry Denisenko, Peter Yiannacouras, John Freeman, Deshanand P Singh, and Stephen D Brown. Opencl for fpgas: Prototyping a compiler. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2012.

[11] Vincent Danjean. OpenCL ICD Loader. https://github.com/OCL-dev/ocl-icd. Accessed: 2021-08-17.

[12] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 1–14. IEEE Press, 2018.

[13] Benedict R. Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL*. 2nd edition, 2013.

[14] Khronos Group. OpenCL API Headers. https://github.com/KhronosGroup/OpenCL-Headers. Accessed: 2021-08-17.

[15] Khronos Group. OpenCL ICD Loader. https://github.com/KhronosGroup/OpenCL-ICD-Loader. Accessed: 2021-08-17.

[16] Krohnos Group. OpenCL Specification 2.0 Shared Virtual Memory. https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf#page=167. Accessed: 2021-08-23.

[17] Krohnos Group. Vector Data Types. https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/vectorDataTypes.html. Accessed: 2021-08-20.

[18] Sitao Huang, Li-Wen Chang, Izzat El Hajj, Simon Garcia de Gonzalo, Juan Gómez-Luna, Sai Rahul Chalamalasetti, Mohamed El-Hadedy, Dejan Milojicic, Onur Mutlu, Deming Chen, and Wen-mei Hwu. Analysis and modeling of collaborative execution strategies for heterogeneous

cpu-fpga architectures. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, page 79–90, New York, NY, USA, 2019. Association for Computing Machinery.

[19] Intel. Hardware Accelerator Research Program (HARP). https://wiki.intel-research.net/index.html. Accessed: 2021-08-19.

[20] Intel. Intel FPGA SDK for OpenCL. https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html. Accessed: 2021-08-19.

[21] Intel. Intel OpenCL SDK. https://software.intel.com/content/www/us/en/develop/tools/opencl-sdk.html. Accessed: 2021-08-19.

[22] Keerthan Jaic and Melissa C. Smith. Enhancing hardware design flows with myhdl. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, page 28–31, New York, NY, USA, 2015. Association for Computing Machinery.

[23] Jiantong Jiang, Zeke Wang, Xue Liu, Juan Gómez-Luna, Nan Guan, Qingxu Deng, Wei Zhang, and Onur Mutlu. Boyi: A systematic framework for automatically deciding the right execution model of opencl applications on fpgas. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, page 299–309, New York, NY, USA, 2020. Association for Computing Machinery.

[24] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium*

on Computer Architecture, ISCA '17, page 1–12, New York, NY, USA, 2017. Association for Computing Machinery.

[25] Pekka Jääskeläinen, Carlos Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. pocl: A Performance-Portable OpenCL Implementation. *International Journal of Parallel Programming*, August 2014.

[26] David R. Kaeli, Perhaad Mistry, Dana Schaa, and Dong Ping Zhang. *Heterogeneous Computing with OpenCL 2.0*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2015.

[27] Khronos. OpenCL Conformance Tests. https://github.com/KhronosGroup/OpenCL-CTS. Accessed: 2021-08-29.

[28] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation. *Parallel Computing*, 38(3):157–174, 2012.

[29] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. Do OS abstractions make sense on FPGAs? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 991–1010. USENIX Association, November 2020.

[30] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[31] Microsoft. Project Catapult. https://www.microsoft.com/en-us/research/project/project-catapult/. Accessed: 2021-08-19.

[32] Vincent Mirian and Paul Chow. An implementation of a directory protocol for a cache coherent system on fpgas. In *2012 International Conference on Reconfigurable Computing and FPGAs*, pages 1–6, 2012.

[33] Vincent Mirian and Paul Chow. Using an opencl framework to evaluate interconnect implementations on fpgas. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2014.

[34] Vincent Mirian and Paul Chow. Evaluating shared virtual memory in an opencl framework for embedded systems on fpgas. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, 2015.

[35] Vincent Mirian and Paul Chow. Exploring pipe implementations using an opencl framework for fpgas. In *2015 International Conference on Field Programmable Technology (FPT)*, pages 112–119, 2015.

[36] Vincent Mirian and Paul Chow. Ut-ocl: an opencl framework for embedded systems using xilinx fpgas. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, 2015.

[37] Vincent Mirian and Paul Chow. Enabling fpgas as a true device in the opencl standard: Bridging the gap for fpgas. In *Proceedings of the 5th International Workshop on OpenCL*, IWOCL 2017, New York, NY, USA, 2017. Association for Computing Machinery.

[38] Marziyeh Nourian, Mostafa Eghbali Zarch, and Michela Becchi. Optimizing complex opencl code for fpga: A case study on finite automata traversal. In *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 518–527, 2020.

[39] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, and Guy Boudoukh. Can fpgas beat gpus in accelerating next-generation deep neural networks? In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 5–14, New York, NY, USA, 2017. Association for Computing Machinery.

[40] Nvidia. Nvidia OpenCL SDK. https://developer.nvidia.com/opencl. Accessed: 2021-08-19.

[41] Khronos OpenCL. OpenCL ICD Installation Guidelines. https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_ICD_Installation.pdf. Accessed: 2021-06-30.

[42] Abishek Ramdas, David Cock, Timothy Roscoe, and Gustavo Alonso. The Enzian Coherent Interconnect (ECI): opening a coherence protocol to research and applications. In *LATTE 21 Workshop on Languages, Tools, and Techniques for Accelerator Design*, 2021.

[43] Ciro Santilli. Pagemap Script. https://github.com/cirosantilli/linux-kernel-module-cheat/blob/master/lkmc/pagemap.h. Accessed: 2021-08-12.

[44] Oren Segal, Philip Colangelo, Nasibeh Nasiri, Zhuo Qian, and Martin Margala. Sparkcl: A unified programming framework for accelerators on heterogeneous clusters, 2015.

[45] Oren Segal, Martin Margala, Sai Rahul Chalamalasetti, and Mitch Wright. High level programming framework for fpgas in the data center. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2014.

[46] Takefumi Miyoshi. Synthesijer. https://synthesijer.github.io/web/. Accessed: 2021-08-19.

[47] Yohann Uguen and Eric Petit. Pyga: A python to fpga compiler prototype. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Artificial Intelligence and Empirical Methods for Software Engineering and Parallel Computing Systems*, AI-SEPS 2018, page 11–15, New York, NY, USA, 2018. Association for Computing Machinery.

[48] Various. An open, general, cpu/fpga platform for os research. submitted, 2021.

[49] Wikipedia. Advanced eXtensible Interface. https://en.wikipedia.org/wiki/Advanced_eXtensible_Interface. Accessed: 2021-07-19.

[50] Mike Wuerthele. OpenGL, OpenCL deprecated in favor of Metal 2 in macOS 10.14 Mojave. https://appleinsider.com/articles/18/06/04/opengl-opencl-deprecated-in-favor-of-metal-2-in-macos-1014-mojave. Accessed: 2021-08-19.

[51] Xilinx. Alveo U200 and U250 Data Center Accelerator Cards Data Sheet. https://www.xilinx.com/content/dam/xilinx/support/documentation/data_sheets/ds962-u200-u250.pdf. Accessed: 2021-08-23.

[52] Xilinx. C++ Arbitrary Precision Integer Types. https://www.xilinx.com/html_docs/xilinx2021_1/vitis_doc/integer_types.html. Accessed: 2021-08-22.

[53] Xilinx. Dynamic Function eXchange UG909. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug909-vivado-partial-reconfiguration.pdf. Accessed: 2021-08-23.

[54] Xilinx. Enabling Hardware Emulation for Extensible XSA. https://www.xilinx.com/html_docs/xilinx2021_1/vitis_doc/create_embedded_platforms.html#zuc1604356222330. Accessed: 2021-08-23.

[55] Xilinx. SmartConnect LogiCORE IP Product Guide PG247. https://www.xilinx.com/support/documentation/ip_documentation/smartconnect/v1_0/pg247-smartconnect.pdf. Accessed: 2021-08-20.

[56] Xilinx. Systolic Array Implementation (OpenCL Kernel). https://github.com/Xilinx/Vitis_Accel_Examples/tree/2020.1/ocl_kernels/cl_systolic_array. Accessed: 2021-08-23.

[57] Xilinx. Using Vitis Embedded Processor Platforms. https://www.xilinx.com/html_docs/xilinx2020_1/vitis_doc/hcb1561793640896.html. Accessed: 2021-07-22.

[58] Xilinx. Vitis Array Data Transfer Modes. https://www.xilinx.com/html_docs/xilinx2020_1/hls-guidance/qoa1585574520885.html. Accessed: 2021-08-20.

[59] Xilinx. Vitis Compiler Command Reference. https://www.xilinx.com/html_docs/xilinx2020_1/vitis_doc/vitiscommandcompiler.html#wrj1504034328013. Accessed: 2021-08-02.

[60] Xilinx. Vitis Getting Started. https://github.com/Xilinx/Vitis-Tutorials/blob/master/Getting_Started/Vitis/example/src/host.cpp. Accessed: 2021-08-23.

[61] Xilinx. Vitis Host Application. https://www.xilinx.com/html_docs/xilinx2020_1/vitis_doc/devhostapp.html. Accessed: 2021-06-30.

[62] Xilinx. Vitis Unified Software Platform. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html. Accessed: 2021-08-20.

[63] Xilinx. Vits HLS Configuration Commands. https://www.xilinx.com/html_docs/xilinx2020_1/vitis_doc/vyw1583260160301.html. Accessed: 2021-08-06.

[64] Xilinx. Xilinx Runtime. https://xilinx.github.io/XRT/2020.1/html/index.html. Accessed: 2021-08-02.

[65] Xilinx. XRT Github. https://github.com/Xilinx/XRT. Accessed: 2021-08-02.

[66] Xilinx. XRT Native APIs. https://xilinx.github.io/XRT/2021.1/html/xrt_native_apis.html. Accessed: 2021-08-19.