PCIeHLS

PCIeHLS: an OpenCL HLS framework

Malte Vesper[a], Dirk Koch[a], and Khoa Pham[a]
[a]University of Manchester, United Kingdom

Abstract

One of the goals of high level synthesis (HLS) is to make designing hardware accelerators running on FPGAs accessible to developers with a software background (usually implying developers with little foundations in hardware design). While high level synthesis generates accelerator kernels, it generally does not assist with integrating the generated kernels into a system. In this paper we introduce PCIeHLS, a framework which helps in providing the required infrastructure consisting of memory, PCIe interface, ICAP for partial reconfiguration, and clock managers. PCIeHLS provides several partial regions, allowing to load multiple modules at the same time. Consequently, multiple kernels can be used simultaneously for multi threading or to run several independent applications. Moreover, regions can be combined to host larger accelerators and accelerators can be relocated at the bitstream level.

1 Introduction

High level synthesis (HLS) makes hardware acceleration accessible to programmers without knowledge of hardware design. HLS allows to program in popular and prevalent languages, like C, C++, or OpenCL [6], or even to reuse existing code (with minor adaptations) to generate hardware accelerators. And with OpenCL, there exists a widely accepted industry standard that allows programming for different accelerator targets including CPUs, GPUs and FPGAs. OpenCL for FPGAs is supported by the major FPGA vendors, and in addition, by third party EDA tools. However, while there exists a sophisticated tool ecosystem for building accelerators from OpenCL, these accelerators still have to be integrated into a system. This comprises two major tasks: the hardware integration and the software integration. Unfortunately, both steps come with a significant complexity and it needs profound hardware design skills as well as skills in writing drivers for interfacing an FPGA to a CPU when using protocols such as PCIe (which is the de facto standard for interfacing FPGA accelerators). In this paper, we present the PCIeHLS framework that fills this gap by providing a hardware interface and corresponding software libraries for generating, integrating and running applications that call OpenCL accelerator kernels.

The overall idea of the framework is that OpenCL can be used to build accelerators with a fixed interface definition (in our example, we started with 32-bit datapaths). And having accelerators using identical interfaces is then used to build an I/O infrastructure for those accelerators that can be reused by literally any OpenCL hardware module (as long as it fits the resources allocated on the target FPGA). Similarly, standardized interfaces allow in turn to provide generic drivers such that it is very easy to build Applications that use those OpenCL accelerators. Finally in order to provide more flexibility, the framework is supporting partial reconfiguration for loading accelerator modules at run-time.

While most of the mentioned features are provided by the frameworks from the major FPGA vendors Altera and Xilinx, PCIeHLS comes with some distinct features including:

- *Module Encapsulation*. The OpenCL accelerators are implemented stand-alone without the need to have a static system. In PCIeHLS, we use hardmacros for interfacing reconfigurable modules with a static system (which is providing the I/O infrastructure). This allows 1) building the static system with the macro, 2) building reconfigurable modules with the macro, or 3) implementing everything in one run.
- *Multi Module Support*, meaning that the PCIeHLS framework can host and run multiple OpenCL modules in parallel.
- *Module Relocation and Multi OpenCL Kernel Instantiation*. PCIeHLS supports running the same module accelerator bitstream at different reconfigurable regions on the FPGA and, if resources permit, the same accelerator can be configured more than once (e.g., for running multiple instances of the same application).

Module Encapsulation enormously simplifies the design of OpenCL accelerators as reconfigurable modules are built without the rather complex static system. This removes many possible pitfalls and one feature enabled by this flow is that even different FPGA CAD tool versions can be used to compile and implement the modules. This is possible as we integrate modules at the bitstream level. This also means that accelerators can easily be reused or shared among different PCIeHLS setups. This is very native for software developers that are used to simply exchange/install software binaries instead of compiling each application from ground up with all possible obstacles.

The paper continues with an overview on the PCIeHLS framework with all its components. In Section 3 we compare the PCIeHLS framework with existing work. The system is presented in Section 4 followed by revealing implementation details in Section 5

2 The PCIeHLS Framework

PCIeHLS is designed for Xilinx FPGAs. Xilinx's High Level Synthesis for OpenCL framework SDAccel [4] generates kernels with two AXI interfaces [7], 1) an AXI Lite slave interface to control the kernel, and 2) an AXI Memory Mapped Master interface which is used by the kernel for fetching operands and for writing back results when running a kernel.

SDAccel comes with very restricted license policies and there are only very few hardware platforms supported by SDAccel. The last holds in particular for many very popular FPGA-boards used in research projects. This is where PCIeHLS provides an academic alternative that is not based on SDAccel, but that uses Vivado HLS (which is also used by SDAccel). While Vivado HLS provides us with a compiler to implement accelerator kernels from OpenCL specifications, it is in the default flow from the vendor Xilinx up to the user to provide the remaining infrastructure. The bare minimum required is a memory with an AXI interface together with a controller that configures and launches the kernel. On top of that, there should be a way to access the memory from the outside to load data into the accelerators workspace and to retrieve results. PCIeHLS provides a system with the memory and an external connection through PCI express at 8 lanes gen3 performance. In addition, it allows to swap the kernels through partial reconfiguration. As OpenCL kernels tend to run significantly slower than the rest of the system (e.g., PCIe and memory) and at different speeds, each kernel runs in a separate clock domain. This allows to run the kernels at their respective maximum frequencies to optimize performance.

A high-level view on the PCIeHLS framework is shown in Figure 1. The user provides OpenCL specifications and the application code. PCIeHLS then uses the standard Vivado HLS compilation flow (with some constraints needed to ensure that the AXI master interface is generated according to the standard defined by PCIeHLS (currently 32bit datapaths). After the OpenCL compilation, the resulting RTL netlist is physicaly implemented using the Vivado tool flow. This is done through compile scripts that are provided with the PCIeHLS framework. The place and route process is constrained such that the accelerator modules will be placed into bounding boxes. As described in more detail in Section 4 and Section 5, our prototype system provides four reconfigurable regions with identical layout of the FPGA resource primitives (i.e. the relative position of logic slices, memory blocks, and multiplier blocks). Any differences between the fabric in the reconfigurable regions are marked as PROHIBIT, and thus effectively voided. By default, the module bounding box dimension is set to host an accelerator in one of those regions. However, if resources are not sufficient, it is possible to increase the bounding box of the module in order to use two adjacent reconfigurable regions for hosting one larger accelerator. In the latter case, one interface will be left unused.

After placement and routing, a full bitstream is generated and the tool BitMan [10] is used to cut out the region that was used for the implementation of the accelerator. All the steps described so far are carried out at system design-time and the result is a set of applications and configuration bitstream binaries that form a library that is later used by the run-time system.

At run-time, an application firstly has to ensure that all needed accelerators are configured to the FPGA. For this step, the BitMan tool is used by the run-time system to relocate accelerator bitstreams to the target reconfigurable region. Placement positions are mainly determined by a simple first possible region fit policy (because all regions are identical). The only exception to this is the case were only a single slot module is present. In this case a new single slot module is placed next to the loaded module, to allow loading a module spanning two slots. The functionality of BitMan can be seen as a run-time linker for configuration bitstreams.

After configuration is completed, compute data is uploaded by DMA to the DDR3 memory of the FPGA board. Then the control registers inside the OpenCL kernels are set with all needed parameters (typically the begin pointers of all arrays) and the execution is started through writing a control register inside the accelerator module. The system then polls a status register to detect if computation of a workgroup completed. This process may repeat over and over again in a system.

In summary, PCIeHLS eases the use of OpenCL HLS results with Xilinx FPGAs by providing the required infrastructure. PCIeHLS is the first academic OpenCL run-time environment to our knowledge that takes care of:

- Building run-time reconfigurable relocatable accelerator modules
- Interfacing the kernels to user software
- Providing local memory for the kernels
- Enabling partial reconfiguration of the kernels
- Automatic optimization and adaptation of the clock frequency for the running kernels

3 Related Work

The large FPGA vendors Intel and Xilinx both offer tool flows to generate accelerators from OpenCL specifications as well as the required infrastructure to run these accelerators. The tool flow often uses PCIe to communicate with the drivers and allows for partial reconfiguration of the different kernels. However, the kernels need to be implemented once for each region they are used in. Xilinx's SDAccel [4] while providing drivers and infrastructure, is limited to a handful of boards. In particular the VC707, VC709 and the Sume board, which are popular in academia, are not supported. Intel's FPGA SDK for OpenCL [3] shares similar limitations. LegUp [5] is another alternative that provides high level synthesis combined with the necessary infrastructure. It either gener-

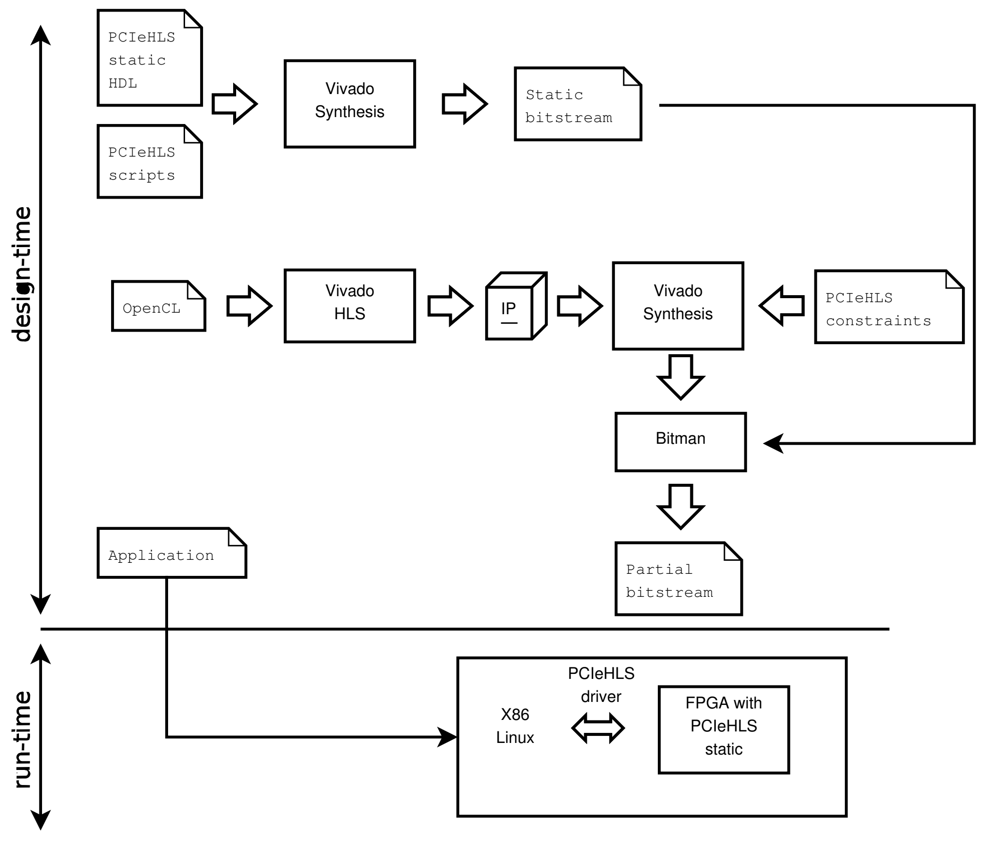

Figure 1 Logical system overview.

ates a pure hardware accelerator or a hardware-/software-codesign. In case of the codesign the program runs on a MIPS softcore or the ARM built-in on the Cyclone V-SoC. LegUp focuses on running everything on the FPGA. This means for the use case we target, accelerators for programs running on a host PC, the user has to create the infrastructure to establish the link.

PCIeHLS implements accelerators as run-time reconfigurable relocatable modules which has been demonstrated several times before. For instance, early work included the PARBIT approach from Horta [11]. Later, Kalte demonstrated this in his REPLICA work on an Xilinx XCV2000E device [12]. More modern approaches were presented by Koch with the ReCoBus-Builder [13] and by Beckhoff with the GoAhead tool [14]. However, all these approaches do not work together with the Vivado tool suite that introduced a new TCL interface for applying constraints. Moreover, newer devices got substantially more complex and in particular constraining the clock network is in a system of the complexity that we aim for quite a challenging task (see also Section 4.1 and 4.2). Nevertheless, for carrying out the communication between the static system and the partial modules, we adopted the *bus macro* concept in this work. This was originally introduced in [15] (by using tristate drivers) and later in [16] using LUTs as connection points.

4 System

We implemented the system as a block design, since we use Xilinx IPs for a majority of the required components. The design is shown in Figure 2. It contains a PCIe block, which we use to interface to the host system. From there, we use the AXI bus for the on-chip communication. The AXI bus interface feeds into the ICAP module allowing for partial reconfiguration of the FPGA and thus the HLS kernels. Furthermore, there is a direct connection to a Mixed Mode Clock Manager (MMCM), through which the clocks for the different kernels can be adjusted. We derive the MMCM parameters from a table that is looked up with the maximum operating frequency reported after the final place and route. The decoupler is also controlled directly through AXI. It decouples slots of unused modules from the bus to avoid invalid bus transactions from toggles during reconfiguration. Furthermore, it applies a reset to the module once it has been (re)configured, in order to ensure proper startup of the module. The AXI Lite control port of each kernel is connected to the PCIe allowing the host PC to control the kernels.

4.1 Memory

Since we use two memory DIMMs, the memory interface generator (MIG) [1] generates a memory controller with two AXI interfaces. To hide this from the user, we have added an AXI crossbar directly in front of the controller. Therefore, the rest of the system sees only one large memory. The memory is a slave to each HLS kernel and the

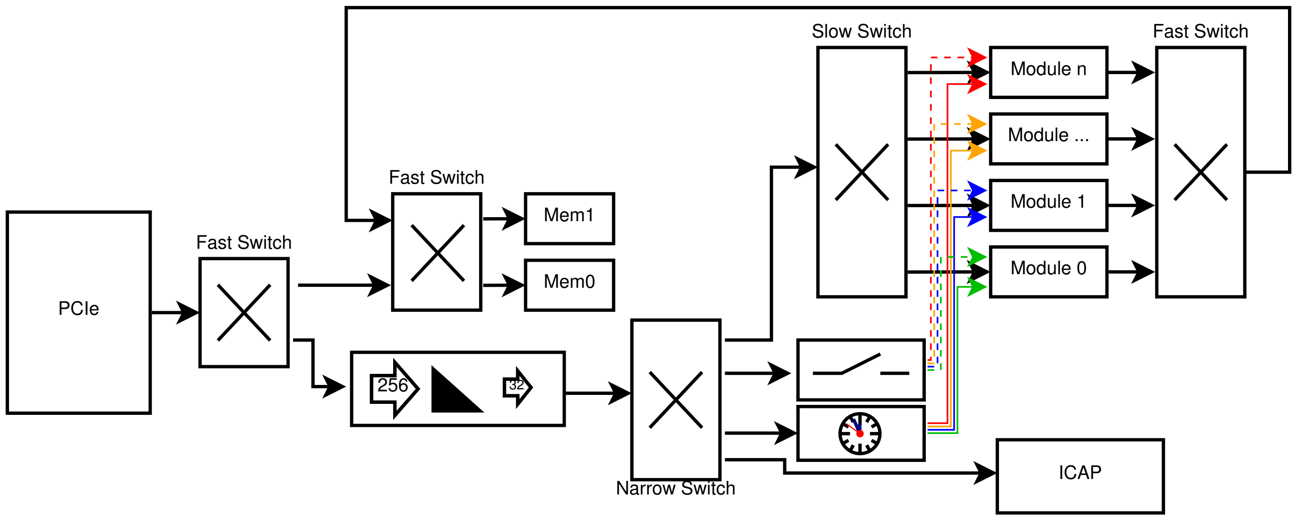

Figure 2 Logical system overview.

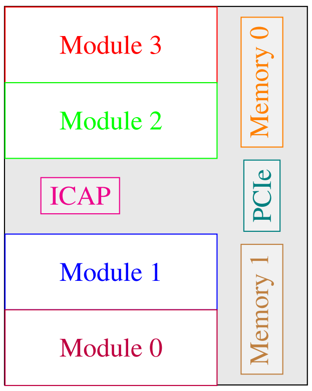

Figure 3 Coarse floorplan, showing the physical placement of the functional blocks and the reconfigurable regions allocated for the modules.

AXI-PCIe bridge. This allows the host PC to access the memory on the FPGA-board directly.

4.2 Clock domains

The system uses a multitude of clock domains. Both memory controllers require their own 200 MHz domains; PCIe 3 with 8 lanes requires 250 Mhz for maximum throughput, and ICAP is limited to 100 Mhz in Virtex 7 devices. On top of that, there are extra clock domains for the different HLS modules. The individual domains for the HLS modules allow us to tune the frequency for the loaded kernel, and thus run each kernel at its respective maximum frequency. The crossings between the different clock domains are handled by Xilinx's AXI clock converter [2] for the AXI buses, and by our own IP for simple wires (i.e. reset). The Xilinx AXI IP is based on Fifo's, while we use two resynchronization stages for wires. Furthermore, our IP has the ability to ensure that a pulse is registered in the other clock domain, which we need to reliably reset the kernels.
Figure 3, shows the coarse floorplan of PCIeHLS. Please note that each block in this picture runs in its individual clock domain.

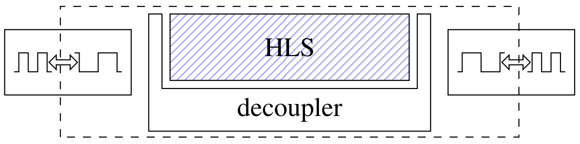

Figure 4 User module surroundings.

4.3 User modules

Figure 4, shows the immediate surrounding of each user module. Each module is placed between two axi clock domain converters, and a decoupler. To generate the user modules, it is sufficient to provide an IP from Vivado HLS. Since our system is geared towards OpenCL, we provide the standard interface used for OpenCL by Vivado HLS: an AXI Lite slave port for control of the kernel and an AXI Memory Mapped master port for memory accesses by the module.

4.4 Partial reconfiguration

For implementing the system reconfigurable, such that OpenCL accelerators can be loaded dynamically at run-time, we created dedicated macros that act as interfaces between the static system (with the PCIe and memory infrastructure) and the actual accelerator modules (see also Figure 5). We then used constraints to allocate some FPGA resources for the static system while leaving larger areas empty that will later host the accelerator modules in the run-time system. The only communication between the accelerator modules and the static system is through the macro that we created.
So far, we reserved only a few hundred wires as a proof of concept using 32-bit AXI connections. However, the principle should scale up to wider buses delivering higher throughput.
A distinct feature of the PCIeHLS framework is that modules can be relocated at the bitstream level. This is done at run-time by the tool BitMan [10]. The tool internally keeps an image of the present FPGA configuration (which is the initial bitstream at system start). Then bitstreams

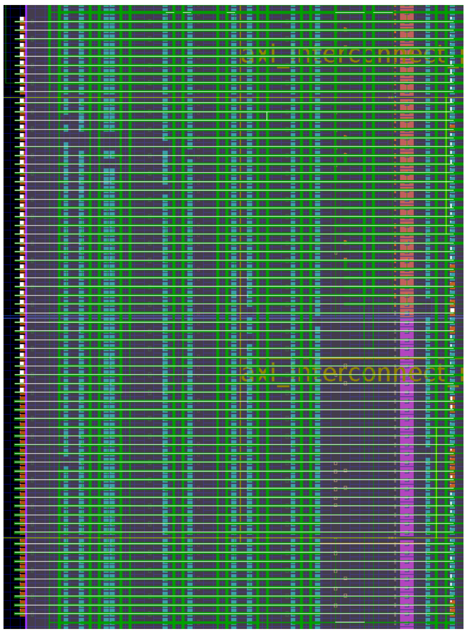

Figure 5 Bridge between a slot and the static system.

are loaded into that image by BitMan and the tool then generates a bitstream that provides the configuration for an accelerator at exactly the desired target reconfigurable region.

This bitstream is then sent through PCIe to an AXI4 slave port that ultimately connects to internal configuration access port ICAP. However, before sending the actual configuration data, the AXI bus connecting the region that is due to be reconfigured is decoupled. After the configuration process has been completed, the module is reset and the bus is enabled. With this, the accelerator module is ready to be used by the application.

5 Implementation

In this section we describe the implementation flow used to generate the static system and the partial bitstreams in such a way that we can relocate them.

5.1 Static system

The static system is placed in a pblock, with the CONTAIN_ROUTING constraint set to true. This forces the placer to position all cells in to the area we designated for the static system. However, some cells are placed outside this pblock due to other constrains. This is the PCIe reset IOPAD and clocking resources.

The clocking resources are placed outside the pblock, since the required clocking resources (i.e. BUFHs) are located in the clock spline, which is in the center of the FPGA [8]. Since we can not partially reconfigure the clockspine [9], we have to configure it correctly for our partial modules in the static system, despite not implementing the partial modules in this stage. To ensure this, we preroute the clock for the partial modules onto the horizontal wires as Xilinx does. However, in contrast to Xilinx's PR flow, we route the clock for each partial module onto the same horizontal

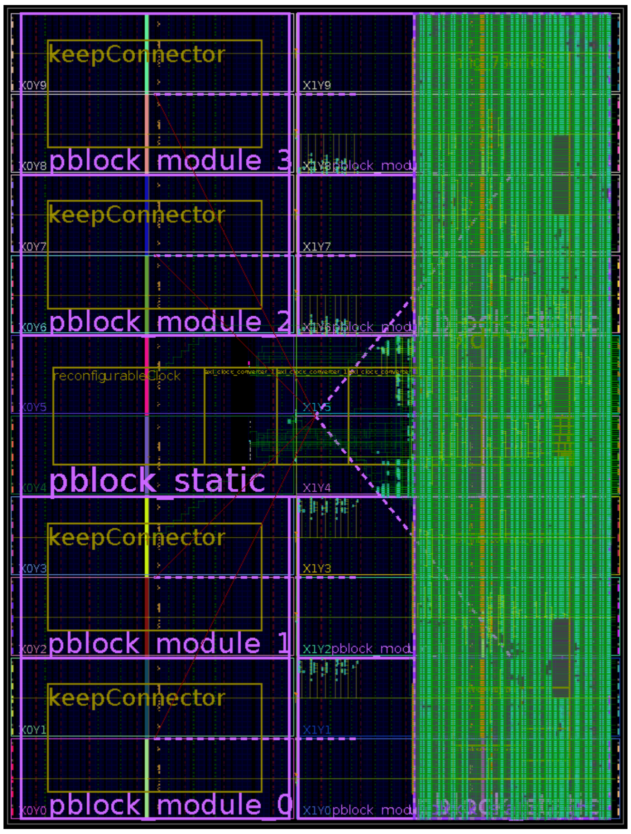

Figure 6 Floorplan of the static system.

wire. This enables modules to connect their clock from the same wire in each partial region. In other words, the static system provides clock routing to all reconfigurable regions using the same relative clock routing resources. The modules are then build accordingly which enables module relocation without the need to manipulate the clock routing.

We can not move the I/O-Pad, as we can not change the physical connection of the FPGA to the PC. To minimize the impact of the PAD we manually route the connection of the PAD. This allows us to pick the shortest route through the area of the partial module. Because this I/O column is not used by the module, there will be no interference of the used I/O pin with the accelerator modules. The routing used to connect the I/O pad is occupied in all partial regions, preventing use in a partial module. This way, we can still relocate modules in vertical direction.

5.2 Partial modules

To save implementation time for the partial modules, we do not generate them in the context of the static system. While generating them separately from the static system reduces complexity, care has to be taken to ensure that the module fits into the regions left by the static system. To ensure this, we implement the module in a pblock, containing the placement and routing.

While this ensures proper placement and routing, we have to take care of the clocks. We block all BUFHs except for the BUFH belonging to the horizontal wire on which the static system drives the clock. Afterwards we route the clock for the module explicitly onto the horizontal wire

Resource	Count	Percentage
LUTs	77 K	17.6 %
BRAM	41	2.8 %
FlipFlops	85 K	9.8 %

Table 1 Resource utilization.

chosen in the static system.
In addition to taking care of the clocks, collisions with cells leaked into the area of the partial module have to be avoided (i.e. PCIe Reset I/O-Pad). For this purpose we generate blockers by checking the region of each partial module for cells and nets. Every cell and net is copied to the relative positions in our module implementation box.
This intersection of obstacles ensures that a module does not conflict with the static system in any slot.

5.3 Resource utilization

Table 1 gives an overview of the resource utilization. As can be seen in Figure 6, the static system uses a lot of the space assigned to it. However, the main reason forcing us to size the static pblock as we did is not the resource utilization (maximum of 50 % for the flip flops), but rather the routing. Due to the physical locations of the memory controllers and the PCIe block visible in Figure 3, a lot of wires have to converge next to the PCIe block. Wires from the AXI buses of the partial modules for memory access, the buses to the memory controllers, and the PCIe bus converge here.

6 Conclusion

In this paper we presented PCIeHLS, a framework to run Vivado HLS kernels as partial modules in several slots with a single bitstream. The fixed static system alleviates the user from issues such as timing closure, yet allows the user to port designs to other boards. Due to the separate generation of the bitstreams for the partial modules, the user is free to use a newer version of Vivado without the need to regenerate the static system. Generation of the partial modules is not impacted by timing closure, since the modules run in their own clockdomains as fast (or slow) as the implementation allows.

6.1 Future work

In an attempt for more open source tools in the flow, we aim to support LegUp as an alternative input tool. Evicting kernels to enable preemptive multitasking is another area for research, however, it would require to read back the states of the modules. Finally, we tested PCIeHLS only with relatively simple vector operations so far and more advanced test cases are needed. While we assume that this will not pose a problem, it still has to be proven.

Acknowledgement

This work is kindly supported by the project Reconfigurable Tera Stream Computing funded by the Defence Science and Technology Laboratory, UK under grant DSTLX-10000092266. This work is partially supported by the European Commission under the H2020 Programme and the ECOSCALE project (grant agreement 671632). In addition we thank Xilinx for supporting our research through their University Program.

7 Literature

[1] Xilinx: UG586, Zynq-7000 AP SoC and 7 Series Devices Memory Interface Solutions v4.2, April 2017
[2] Xilinx: PG059 AXI Interconnect v2.1, April 2017
[3] Intel: Intel FPGA SDK for OpenCL, 2017.05.08, May 2017
[4] Xilinx: UG1164, SDAccel Enviornment, Platform Development Guide, v2016.4, March 2017
[5] Canis A., et al.: From software to accelerators with legup high-level synthesis, CASES 2013
[6] Khronos OpenCL Working Group: The OpenCL Specification, version 1.0, 8 December 2008
[7] ARM: AMBA AXI and ACE Protocol Specification; AXI3, AXI4, and AXI4-Lite, ACE and ACE-Lite, 2013
[8] Xilinx: UG472, 7 Series FPGAs Clocking Resources v1.13, March 2017
[9] Xilinx: UG909, Vivado Design Suite User Guide, Partial Reocnifguration v2017.1, April 2017
[10] Pham, K., Horta, E. and Koch, D.: BITMAN: A tool and API for FPGA bitstream manipulations, DATE 2017
[11] Horta, E., Lockwood, J. and Kofuji, S.: Using PARBIT to Implement Partial Run-Time Reconfigurable Systems, FPL, 2002
[12] Kalte, H., Lee, G., Porrmann, M. and Rückert, U.: REPLICA: A Bitstream Manipulation Filter for Module Relocation in Partial Reconfigurable Systems, IPDPS, 2005
[13] Koch, D., Beckhoff, C. and Teich, J., ReCoBus-Builder – a Novel Tool and Technique to Build Statically and Dynamically Reconfigurable Systems for FPGAs, FPL, 2008
[14] Beckhoff, C. and Koch, D. and Torresen, J.: GoAhead: A Partial Reconfiguration Framework, FCCM, 2012
[15] Xilinx: XAPP290, Two Flows for Partial Reconfiguration: Module Based or Difference Based, 2002
[16] Lysaght, P., Blodget, B., Mason, J., Young, J. and Bridgford, B.: Invited Paper: Enhanced Architecture, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs, FPL, 2006