SystemNaim FYP Final Report

Naim Govani

May 31, 2021

Contents

Abstract

Do abstract here

Chapter 1

Project Introduction

Chapter 2

Background

2.1 Introduction

The field of multi-FPGAs has been heavily researched however there are few papers interested in utilising HLS to streamline the process of implementing a multi-FPGA. [13] and [5] are examples of papers which do concern themselves with these topics but both were written over twenty years ago.

Therefore, it is of interest to this project to do a review of the most up to date research in fields surrounding and including its scope. This chapter will go through four different topics:

- Related works.
- HLS tools and their optimisations.
- FPGA-to-FPGA communication protocols.
- Multi-FPGA Logic Partitioning.

Thus, hopefully, giving the reader a better overview of what is possible if this project were to reach its full potential.

2.2 Related Work

As previously mentioned, [13] and [5] are papers which have already attempted to solve the issues raised by this project. The solution in the former uses the GNU C++ compiler as well as a method of describing the algorithm as a set of difference equations. One point the authors explicitly state is that they utilise C++'s operator overloading feature in order to map software operations to specific hardware modules. Nonetheless, once the circuit has been generated, it is then automatically partitioned using a method based on Fiducia-Mattheyses bi-partitioning heuristic [7] with the goal of this stage to maximise logic utilisation in each FPGA as well as minimising IO consumption.

The method by which the FPGAs communicated with each other is not discussed in the paper, all that is mentioned is the method used for partitioning. There is, however, mention of a host CPU, and it may be reasonable to assume that this CPU acts as a master for all FPGAs meaning there is no direct communication between FPGAs.

The latter paper, [5], opts to place a psuedo-processor on each FPGA within the topology and then have a top-level controller supply each FPGA with the instructions it is meant to execute. The HLS aspect then merely needs to convert the supplied source code into a list of instruction which is the intended use of standard compilers. The motivation for this approach was to make multi-FPGA systems more accessible to software developers, which is its main target demographic. Alongside this, the paper also presents how it uses a 4 dimensional space, with one dimension being each FPGA system, to optimise the datapath of the system. This then supposedly creates a control path where the FPGAs are used in parallel wherever possible.

2.3 HLS Tools & Optimisations

2.3.1 Introduction

Different HLS tools have vastly different implementations [18] and many of them, especially commercial tools, do not reveal their methodologies. LegUp[3] for example, compiles the input source code using the LLVM compiler and then profiles it on a modified MIPS CPU emulator in order identify which parts of the code would benefit from hardware acceleration. It then synthesis those specific parts to dedicated hardware, while the rest of the code is run using a MIPS CPU programmed onto the FPGA.

Vivado HLS, on the other hand, makes no attempt to suggest to the user where acceleration is possible. Rather, it merely gives you tools known as PRAGMAS, found in [25], to help the compiler better understand how to convert the C code to hardware. The output is then a set Verilog files the user can implement as they see fit, though it is recommended to use the Vivado Design Suite [24] as part of the development cycle. As of now, Vivado has not revealed how they go from C to Verilog.

A third contender is Bambu [19], which leverages the GCC IR in order to optimise the resulting hardware and focuses mainly on memory intensive programs. In short, each implementation has its own intended purpose and thus excels in a specific area, but the fact that there are many implementations indicate that this field has much room from improvement and is actively under development.

2.3.2 Input Language

For the most part, HLS tools tend be based around compiling a subset of C or C++ to Verilog [18], however, there is an argument to be made that functional programming languages have a certain affinity towards hardware development. [2] presents an early form of the idea, Lava, which takes Haskell as input and produces circuit specifications. The tool was built in the early days of HDLs, and thus was intended to be a competitor rather than a part of the tool chain. More modern approaches such as [14] are libraries for existing languages and instead produce Verilog or SystemVerilog which can be later synthesised by the appropriate tools. But why bother with straying so far from the norm. In [27], it is suggested that the firm mathematical founding in which functional languages are based can assist in verifying hardware designs as well as offering benefits such as recursive functions and type inference. While the author in [6], suggests that the functional paradigm can allow for a greater exploitation of parallelism, one of the driving reason to using FPGAs.

Evidently, the field of research in using functional languages is of interest to quite a few individuals. [1] lists a few papers in its introduction whilst also contributing to the field itself, but it has yet to be seen whether Functional HLS will overtake standard C-based HLS anytime soon.

2.3.3 Loop Pipelining

One of the major benefits of running a system on an FPGA is its ability to perform computation in parallel. Loop pipelining is a perfect example of this, where a HLS tool will take a for loop or while loop written in the source code and run segments of it concurrently in order to reach a low initiation interval(II). Page 120 of [25] and page 11 of [12] show how the two leading HLS tools, Vivado HLS and Intel High Level Synthesis Compiler, implement pipelining respectively. The II is a measure of how many a new iteration of the loop can start and a lower value, ideally one, means that the section of code containing the loop will be computed faster.

There are issues, however, with attempting to reach an $II = 1$. [17] explores how memory dependencies, two parts of a loop accessing the same memory resources, can cause increases in II due to the HLS compiler not being able to find a way to resolve these dependencies. Instead, it proposes the idea of splitting loops in order to reach an II of 1 on each of these sub-loops and then to pipe the data from one sub-loop to the next. Another approach shown in [16] opts to implement parametric hardware to resolve similar dependencies.

In summary, for simple programs with static loop behaviour the loop pipelining strategies found in commercial tools is very effective and likely the most important optimisation. However, when the behaviour becomes uncertain more effort is required on the user's side to help the compiler reach lower II's.

2.3.4 Data Streaming

In order to process data in parallel on an FPGA the data must first be available to the device or module doing the processing, an obvious point but one that must be considered for any HLS system. In general, there are two approaches to solve the problem of ensuring data availability, the first is atomic transactions while the second is data streaming/dataflow. Say that we have two modules, A and B which are connected such that the output of A, which is an array, is the input of B. Let us also say that both modules iterate over an array doing some processing, i.e. multiplying the elements by 2 or adding 5, on said array. In an atomic system we would first let A iterate over the entire array and then pass the array to B and allow it to compute its function.

In this case, even though the modules themselves may be pipelined there isn't much parallelism being exploited on the module level. B is sitting idle for the entire time A is operating and vice-versa. The solution to this is dataflow, which is where every time A completes an iteration it then passes that data to B allowing to start its computation. This results in both modules being utilised at the same time and reduces the overall latency of the system. Section 3.3 of [8] explores further, why a user might opt for this method and the considerations they must take if they choose to do so.

In relation to HLS tools, this optimisation is pivotal to achieving high throughput in any system that operators on large amount of data. Pages 20-24, 95-97 and 126-128 of [25] show how Vivado HLS allows its users to access such functionality.

2.4 FPGA-to-FPGA Communication

2.4.1 Communication Medium

One of the major considerations that must be made when deciding the structure of a multi-FPGA system is the communication medium between the FPGAs. Many options are possible, [15] utilises SFP+ connections whilst [21] connects the board using a PCI-E interface. Both examples require ports to be connected to FPGA I/O pins and are usually approaches that are, usually, only used when working development boards. If, one only has access to the pure FPGA itself, unlikely nowadays, then pin implementations are a much more apt choice. [10] goes into very fine detail about the necessity of a good pin assignment algorithm to reduce unnecessary resource consumption.

It seems that more modern implementations tend to use a standardised communication medium such as Ethernet, PCI-E or Serial, which makes sense given the large overhead usually required when developing a new communication protocol. [23] even explores making the communication medium agnostic to the user application, which allows the user logic to access a single interface which then handles to packaging of data for any medium. For example, one FPGA may only have access to an Ethernet port while another only has an RS232 port, but this system would allow these two FPGA's to transfer data between them.

2.4.2 Purpose of the System

While there are many options for communication mediums available to a designer, one must first question the purpose of the system in order to evaluate the efficacy of a communication method.

A **PCI-E** communication medium tends to be chosen when there is a single host communicating with multiple client's such as in [21] or [9]. PCI-E does not allow for clients to interact directly with each other and instead only allows host-client communication.

In contrast, **Ethernet** allows the user to identify each device in their topology with a number

(MAC Address) and then lets each device send data to every other device in the network (Section 8 in [11]) making it more suited to Peer-2-Peer systems.

Both have their place in the ecosystem with latest PCI-E generations reaching 128 GB/s [20] compared to Ethernet reaching at most 1.25 GB/s [22], whilst is Ethernet better suited for more complicated topologies with numerous hosts.

2.5 Logic Partitioning

2.5.1 Manual vs. Automatic

Logic partitioning concerns itself with how a designer split's the computation across the multi-FPGA system. Generally speaking, there are two approaches to this a manual split and an automatic split. A manual split is where the designer is given control of the computation processed on every FPGA, however usual implementations following this methodology tend to be a single design replicated on multiple FPGAs. [4, 15, 21] all have one FPGA overlay that supports communication either with a host or other peers and is able to slot into a pre-existing network in a modular fashion. Thus, it would seem that any manual splitting is done with scalability in mind and a repeatable design is used to reduce decrease complexity and development time.

The second approach is an automatic split, and it seems that papers regarding this method tend to concern themselves with more general input than a specific use case. Both papers mentioned in section 2.2 and [26] are examples of this approach. Interestingly, [5] uses a similar practice to what was described of the manual approach by having a single design, that has a more general use-case, overlayed onto many FPGAs. Whereas [13, 26] instead opt to use a partitioning algorithm, which splits a design to fit the requested number of devices.

2.5.2 Control Schema

Another aspect to logic partitioning, automatic or manual, is the method by which the workload is distributed. Once again, there seem to be two main approaches. One is for a host device to control each FPGA within system, supplying it with data and also telling it when to start execution. This seems to be the case for [5] and [21].

On the other hand, [4, 15, 26] choose to allow each FPGA to perform its own computation and communicate, whenever it needs to, with other devices on the network. This method while having more critical points, is much more scalable than the first approach which results in a constraint being put upon the system by the hosts computational ability.

Chapter 3

Project Design

3.1 HLS

3.2 Inter-connect

3.3 Communication Channel

Chapter 4

Implementation

4.1 HLS

4.2 Inter-connect

4.3 Communication Channel

Chapter 5

Evaluation

5.1 Introduction

5.2 System Performance

5.3 Channel Performance

5.4 Usability

Chapter 6

Future Work

6.1 General HLS Optimisations

6.2 Inter-FPGA Data Streaming

6.3 Port Expansion

Bibliography

[1] A. B. Ablak and I. Damaj. "HTCC: Haskell to Handel-C Hardware Compiler". In: *2016 Euromicro Conference on Digital System Design (DSD)*. 2016, pp. 192–199. DOI: 10.1109/DSD.2016.24.

[2] Per Bjesse et al. "Lava: hardware design in Haskell". In: *ACM SIGPLAN Notices* 34.1 (1998), pp. 174–184.

[3] Andrew Canis et al. "LegUp: High-level synthesis for FPGA-based processor/accelerator systems". In: Jan. 2011, pp. 33–36. DOI: 10.1145/1950413.1950423.

[4] Guohao Dai et al. "ForeGraph: Exploring Large-Scale Graph Processing on Multi-FPGA Architecture". In: FPGA '17. Monterey, California, USA: Association for Computing Machinery, 2017, pp. 217–226. ISBN: 9781450343541. DOI: 10.1145/3020078.3021739. URL: https://doi.org/10.1145/3020078.3021739.

[5] A. A. Duncan, D. C. Hendry, and P. Gray. "An overview of the COBRA-ABS high level synthesis system for multi-FPGA systems". In: *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No.98TB100251)*. 1998, pp. 106–115. DOI: 10.1109/FPGA.1998.707888.

[6] S. A. Edwards et al. "FHW Project : High-Level Hardware Synthesis from Haskell Programs". In: 2019.

[7] C. M. Fiduccia and R. M. Mattheyses. "A Linear-Time Heuristic for Improving Network Partitions". In: *19th Design Automation Conference*. 1982, pp. 175–181. DOI: 10.1109/DAC.1982.1585498.

[8] J. de Fine Licht et al. "Transformations of High-Level Synthesis Codes for High-Performance Computing". In: *IEEE Transactions on Parallel & Distributed Systems* 32.05 (2021), pp. 1014–1029. ISSN: 1558-2183. DOI: 10.1109/TPDS.2020.3039409.

[9] J. Gong et al. "An efficient and flexible host-FPGA PCIe communication library". In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014, pp. 1–6. DOI: 10.1109/FPL.2014.6927459.

[10] S. Hauck and G. Borriello. "Pin assignment for multi-FPGA systems". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16.9 (1997), pp. 956–964. DOI: 10.1109/43.658564.

[11] "IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture". In: *IEEE Std 802-2014 (Revision to IEEE Std 802-2001)* (2014), pp. 1–74. DOI: 10.1109/IEEESTD.2014.6847097.

[12] Intel. *Intel® High Level Synthesis Compiler Pro Edition: User Guide*. 2020. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/hls/ug-hls.pdf.

[13] Isshiki and Dai. "Bit-serial pipeline synthesis for multi-FPGA systems with C++ design capture". In: *1996 Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*. 1996, pp. 38–47. DOI: 10.1109/FPGA.1996.564741.

[14] JaneStreet. *hardcaml*. https://github.com/janestreet/hardcaml. 2020.

[15] Weiwen Jiang et al. "Achieving Super-Linear Speedup across Multi-FPGA for Real-Time DNN Inference". In: *ACM Trans. Embed. Comput. Syst.* 18.5s (Oct. 2019). ISSN: 1539-9087. DOI: 10.1145/3358192. URL: https://doi.org/10.1145/3358192.

[16] J. Liu, S. Bayliss, and G. A. Constantinides. "Offline Synthesis of Online Dependence Testing: Parametric Loop Pipelining for HLS". In: *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 2015, pp. 159–162. DOI: 10.1109/FCCM.2015.31.

[17] J. Liu, J. Wickerson, and G. A. Constantinides. "Loop Splitting for Efficient Pipelining in High-Level Synthesis". In: *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2016, pp. 72–79. DOI: 10.1109/FCCM.2016.27.

[18] R. Nane et al. "A Survey and Evaluation of FPGA High-Level Synthesis Tools". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (2016), pp. 1591–1604. DOI: 10.1109/TCAD.2015.2513673.

[19] C. Pilato and F. Ferrandi. "Bambu: A modular framework for the high level synthesis of memory-intensive applications". In: *2013 23rd International Conference on Field programmable Logic and Applications*. 2013, pp. 1–4. DOI: 10.1109/FPL.2013.6645550.

[20] Trenton Systems. *PCIe Gen 4 vs. Gen 3 Slots, Speeds*. Sept. 2020. URL: https://www.trentonsystems.com/blog/pcie-gen4-vs-gen3-slots-speeds.

[21] Deguang Wang et al. "An Efficient Design Flow for Accelerating Complicated-Connected CNNs on a Multi-FPGA Platform". In: *Proceedings of the 48th International Conference on Parallel Processing*. ICPP 2019. Kyoto, Japan: Association for Computing Machinery, 2019. ISBN: 9781450362955. DOI: 10.1145/3337821.3337846. URL: https://doi.org/10.1145/3337821.3337846.

[22] David Weedmark. *The Three Most Common Ethernet Speeds*. May 2019. URL: https://smallbusiness.chron.com/three-common-ethernet-speeds-69375.html.

[23] A. Wu et al. "A flexible FPGA-to-FPGA communication system". In: *2017 19th International Conference on Advanced Communication Technology (ICACT)*. 2017, pp. 836–843. DOI: 10.23919/ICACT.2017.7890234.

[24] Xilinx. *Vivado Design Suite User Guide*. 2018. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug910-vivado-getting-started.pdf.

[25] Xilinx. *Vivado HLS Optimization Methodology Guide*. 2017. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1270-vivado-hls-opt-methodology-guide.pdf.

[26] Tack Boon Yee, Mark Zwolinski, and Andrew D Brown. "Multi-FPGA Synthesis with Asynchronous Communication Subsystems". In: *IFIP International Conference on Very Large Scale Integration (VLSI-SOC 2005) (01/01/05)*. 2005. URL: https://eprints.soton.ac.uk/261305/.

[27] K. Zhai et al. "Hardware synthesis from a recursive functional language". In: *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 2015, pp. 83–93. DOI: 10.1109/CODESISSS.2015.7331371.