Split Learning on FPGAs

by

Hannah K. Whisnant

B.S., United States Military Academy at West Point (2018)

Submitted to the Institute for Data, Systems, and Society
in partial fulfillment of the requirements for the degree of

Master of Science in Technology and Policy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2020

Author ....

Institute for Data, Systems, and Society
September 3, 2020

Certified by..

Richard Younger
Technical Staff, MIT Lincoln Laboratory
Thesis Supervisor

Certified by..

Frank R. Field III
Senior Research Engineer, Sociotechnical Systems Research Center
Thesis Supervisor

Accepted by ....

Noelle E. Selin
Director, Technology and Policy Program
Associate Professor, Institute for Data, Systems, and Society and
Department of Earth, Atmospheric and Planetary Sciences

Split Learning on FPGAs

by

Hannah K. Whisnant

Submitted to the Institute for Data, Systems, and Society
on September 3, 2020, in partial fulfillment of the
requirements for the degree of
Master of Science in Technology and Policy

Abstract

MIT Lincoln Laboratory is developing a software-reconfigurable imaging architecture called ReImagine, the first field programmable imaging array (FPIA), which reflects a broader trend in the increased use of FPGAs in sensor systems in order to reduce size and power consumption without a corresponding loss in performance or flexibility. At the same time, the field of machine learning is diversifying to include distributed deep learning methods like split learning, which can help preserve privacy by avoiding the sharing of raw data and model details. In order to continue to expand the capabilities of architectures like ReImagine's and enable split learning and related techniques to be used in the growing body of FPGA-based sensor systems, we examine the relationship of emerging split learning applications to FPGA-based image processing platforms. We determine that the implementation of split learning methods on FPGAs is feasible, and outline use cases in the areas of health and short timescale physics that demonstrate the usefulness of these implementations to both organizations concerned with privacy-preserving machine learning methods and organizations concerned with the deployment of efficient, flexible, and low-latency sensor systems. We begin by conducting a survey of the modern FPGA landscape in terms of technical attributes, use in sensor systems, security and privacy features, and current machine learning implementations. We also provide an overview of split learning and other distributed deep learning methods. Next, we synthesize an example split learning model in HDL code to demonstrate the feasibility of implementing such a model on an FPGA. Finally, we develop use cases for split learning applications on FPGA-based sensor systems and offer conclusions about the future development of distributed deep learning on heterogeneous processing platforms.

Thesis Supervisor: Richard Younger
Title: Technical Staff, MIT Lincoln Laboratory

Thesis Supervisor: Frank R. Field III
Title: Senior Research Engineer, Sociotechnical Systems Research Center

Acknowledgments

I would like to thank the following people and organizations for their help in writing this thesis: Group 87 at MIT Lincoln Laboratory, particularly my advisor Rich Younger, who is always willing to entertain a new idea but models the importance of taking the time to think over whether a question is useful before it takes up too much bandwidth; the Camera Culture group at the MIT Media Lab, particularly Praneeth Vepakomma, who was resourceful, dedicated, endlessly patient, generous with his time, and instrumental in providing the direction required to get this thesis finished; and the entire staff of the Technology and Policy Program, including Barb DeLaBarre, whose office (our home away from home) is a place of kindness, understanding, honesty, and willingness to listen and share, Ed Ballo, who offers valuable reminders of the ways we can look outside the MIT bubble and appreciate the broader context in which we spend our days, Frank Field, who contributed greatly to the completion of this thesis and without whom I can't imagine how any one of us would have begun to conceptualize our role in the program or our program's role in the world, and Noelle Selin, whose straightforward way of listening and responding to the complex issues that surround running a program full of passionate, dedicated young people have made her a personal role model of mine.

I'd also like to thank the people who made the last two years a beautiful, exhilarating, and unforgettable experience, through good times and bad: my cohort in the Technology and Policy Program, by whom for two years I was constantly challenged, educated, inspired, impressed, supported, and accepted. I have never met a group of people so willing to have loud, pedantic arguments about public policy at parties, which is unironically my favorite thing about them all. In particular, I'd like to thank my roommates, Becca Browder, Tomas Green, and Nolan Hedglin, with whom I endured unprecedented times, and who never, ever failed to make each day better through their humor, empathy, and creativity.

Finally, I'd like to thank my family: Mom, Dad, Mason, Cooper, Scarlett, and Max, who are the most important part of my life, no matter where I go. I love you.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

1.1 Motivation

The world is blanketed in imaging sensors, from ubiquitous cell phone cameras to MRI machines to the powerful telescopes that document the night sky around the world. As has been observed time after time, there has been an unprecedented explosion in the collection of visual data, which in turn has led to the rapid and widespread development of machine learning techniques that allow individuals, governments, and industries transform that glut of raw data into understandable and actionable information.

Machine learning has the potential to improve the human conditions in countless ways—researchers imagine a world where self-driving cars using computer vision algorithms dramatically reduce the incidence of car accidents and health providers are able to use machine learning methods to offer more accurate diagnoses and improve health outcomes. However, as small, mobile sensors intrude more deeply into our everyday lives, transmitting personal data to central servers, concerns about individual privacy, particularly with respect to sensitive information like health data, have increased.

In order to enable organizations to train and implement useful machine learning algorithms without sacrificing privacy, the MIT Media Lab is developing split learning, a distributed deep learning method that allows multiple clients to train models without sharing raw data with each other or a central server. Split learning has been developed for use on CPUs and GPUs—the microprocessors on which most machine learning algorithms are implemented. However, not all valuable data is collected on sensor nodes based on microprocessors. A growing number of sensor nodes are based on FPGAs—reconfigurable hardware for which the development of applications offers a different set of challenges.

DARPA's ReImagine project, under development at MIT Lincoln Laboratory, is one example of an FPGA-based imaging sensor. ReImagine is a single, multi-functional imaging platform that can accommodate multiple modes of operation and interface with multiple types of imaging sensor. Ultimately, ReImagine is intended to be able to autonomously toggle between different modes using machine learning algorithms; however, FPGA developers face resource constraints and design difficulties that developers working with CPUs and GPUs do not.

This work seeks to explore the potential application of split learning and other distributed deep learning methods to FPGA-based sensor systems like ReImagine.

1.2 Problem Statement

This paper examines the possibility of conducting split learning on FPGAs from two perspectives. First is the perspective of the user of an FPGA-based sensor system like ReImagine, who is seeking to expand the set of tools that can be used on their devices and leverage the power of machine learning methods without prohibitively high communication, computation, and storage burdens and with some assurance of the privacy of the raw data that the sensor nodes in the system collect. The second is the perspective of a split learning application developer who has strict requirements with respect to factors like latency and SWaP, where FPGAs can frequently be optimized to outperform microprocessors.

In order to determine whether the implementation of split learning applications on FPGAs offers advantages to groups seeking to deploy privacy-preserving machine learning methods and groups seeking to add additional capabilities to FPGA-based sensor systems, we seek to answer the following questions:

- Is it technically feasible to implement split learning applications on FPGAs?
- How have developments in FPGA technology changed the landscape and potential of FPGAs as platforms for sensor systems and for new machine learning applications?
- What current and future use cases exist for the implementation of split learning on FPGAs?

1.3 Research Methodology

In this paper, we provide background for the ongoing ReImagine and split learning projects. We then conduct a literature review focused on the current landscape of FPGA technology, including the technical attributes of FPGAs, the current uses of FPGAs in sensor systems, the security and privacy features of modern FPGAs, and the tools and limitations that exist for the implementation of machine learning algorithms on FPGAs.

Next, we implement a simple client-side split learning model on a simulated FPGA in order to provide an initial demonstration of the technical feasibility of implementing split learning algorithms on FPGAs. We then conduct further analysis on the current uses of FPGAs in sensor systems and their features supporting security and privacy, comparing these with the requirements of split learning in order to develop potential use cases for split learning on FPGAs. Finally, we summarize our findings and offer conclusions and suggested directions for future work in the area of split learning on FPGAs.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 2

Background and Literature Review

2.1 Background

In this section, we provide overviews of the histories and current statuses of the ReImagine project at Lincoln Laboratory and the split learning effort at the Media Lab.

2.1.1 Overview: ReImagine Project

In 2016, the Defense Advanced Research Projects Agency (DARPA) announced a new program that aimed to create a single, multi-functional imaging sensor that could accommodate simultaneous, distinct modes of operation and interface with nearly any type of optical detector array [40]. The program, called ReImagine (short for Reconfigurable Imaging), would ultimately result in a camera able to autonomously respond to the images it receives in real time, using algorithms developed by industry partners to toggle between different imaging modes and adjust factors like frame rate and resolution. This system would be able to "adapt and change [its] personality and effectively morph into the type of imager that provides the most useful information for a given situation" [5].

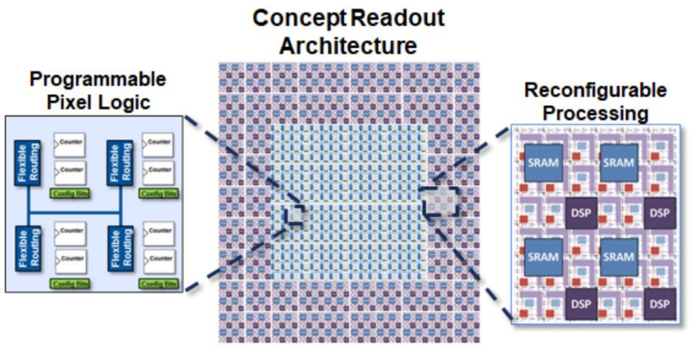
A system like ReImagine would be relevant to a wide variety of military and commercial applications. At the time ReImagine was proposed, existing imaging systems were becoming increasingly powerful and flexible. For example, the previous year, the Department of Energy had approved the construction of the Large Synoptic Survey Telescope, which would be used in astronomy and at 3.2 gigapixels would have the highest resolution of any camera created [31]. The year before that, MIT Lincoln Laboratory had developed a digital-pixel focal plane array (DFPA) that overcame some of the limitations of conventional focal plane arrays, allowing for higher dynamic ranges, lower size, weight, and power (SWaP), and some on-chip image processing capabilities [36]. For the most part, imaging systems are single-sensor systems that must be relegated to larger vehicles and platforms. They are also limited in application due to the way their readout integrated circuits, or ROICs, are traditionally implemented [40]. ROICs are integrated circuits that sample the signals that each pixel in a camera is receiving and transfer that data away for processing. Traditionally, ROICs are designed as application-specific integrated circuits (ASICs), hardware dedicated to a specific task that is very efficient, but with fixed logic that renders the ROIC, and thus the imaging system, inflexible.

ReImagine was conceived as a smaller, cheaper, and more power-efficient alternative to designing and manufacturing multiple large single-sensor imaging systems with traditional ROICs. Most importantly, ReImagine's architecture would be more similar to field-programmable gate array (FPGA) architecture than to ASIC architecture, creating a software-reconfigurable imaging system that supports real-time analysis of complex images in multiple modes, improving situational awareness and enabling more informed decision-making [40]. Further comparison between ASIC and FPGA architectures can be found in Section 2.2.2.

In place of an imaging system with a traditional ROIC, ReImagine would have a new architecture referred to as a field-programmable imaging array (FPIA). FPIAs are reconfigurable digital circuits with integrated FPGA processing. They combine ROIC circuitry that interfaces with sensors and FPGA circuitry, resulting in a reconfigurable ROIC that allows the imaging system to interface with many detectors and support many modes of operation [21].

Overall, the ReImagine architecture includes three tiers, layered one on top of the other. The lowest tier is an FPIA. This tier is being developed by MIT Lincoln Laboratory's Advanced Imager Technology group. The first generation of FPIA in development at MIT Lincoln Laboratory, the architecture of which is pictured in Figure 2-1 is called the Griffin FPIA.

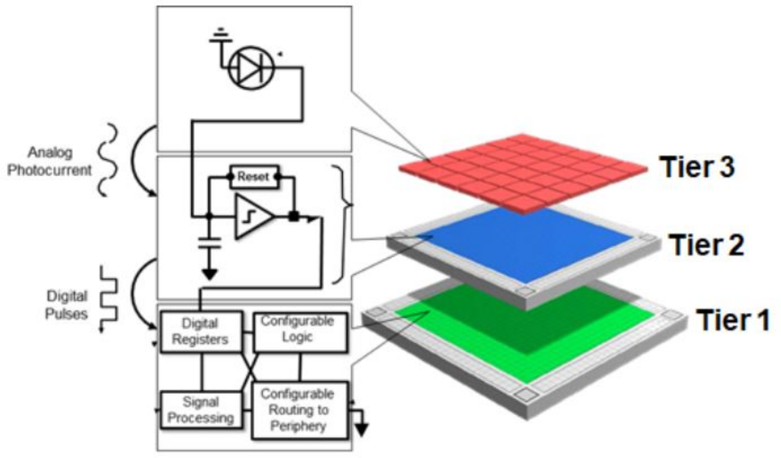
Figure 2-1: MIT Lincoln Laboratory FPIA [52].



Tiers 2 and 3 of the ReImagine architecture are not common to all applications, and comprise a variety of industry-produced analog detectors (Tier 3) and detector-specific interfaces that allow analog detectors to interface with the common digital circuit below (Tier 2) [52]. This three-tier architecture is shown in Figure 2-2. This thesis will focus primarily on the capabilities of Tier 1, the FPIA, and related architectures.

The first generation Griffin FPIA exists today, and meets many of the goals outlined by DARPA. As of 2018, the chip was the largest integrated circuit ever developed within the Department of Defense, with over 6.6 billion transistors and 9km of internal wiring on a 320 mm^2 area [52]. This was achieved using the 14 nm FinFET process, a method of fabrication that has allowed Lincoln Laboratory to reduce pixel size to a fraction of that of previous projects like the DFPA, which used a 65 nm CMOS process to create pixels with an area of 625 μm^2. The resulting decrease in pixel pitch allows the Griffin FPIA to consume less power than the DFPA, which means the chip can support an increased number of pixels in the array as well as on-chip processing [27]. While the 14 nm process is more expensive than the 65 nm process, the on-chip processing and reconfigurable design allows this cost to be amortized over multiple projects, as the same hardware can be used with many different detectors and for

Figure 2-2: DARPA ReImagine program 3D sensor architecture [52].

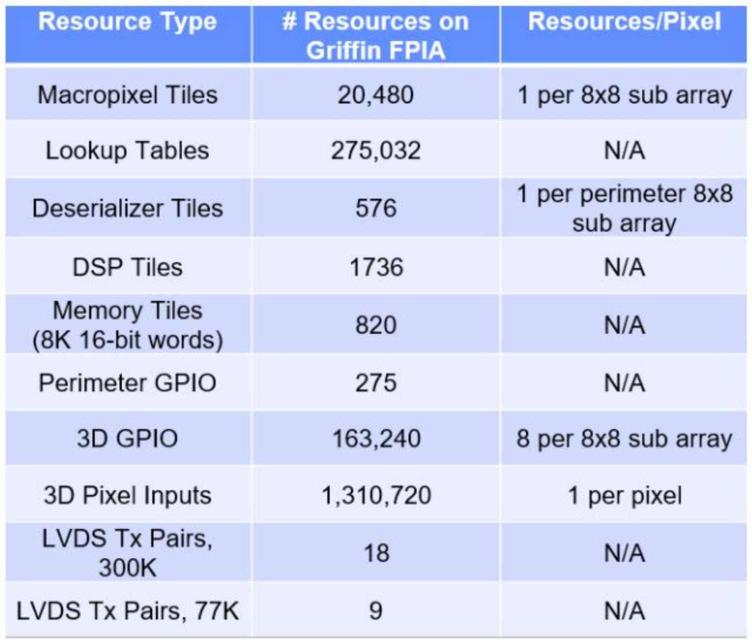


Tier 3	Waveband Specific Detector Array	Industry
Tier 2	Detector Specific Analog Interface	Industry
Tier 1	Reconfigurable Digital Circuit with Integrated FPGA Processing	MIT LL

many different applications [52].

With the first-generation Griffin FPIA fabricated, we are able to examine the algorithms that may be implemented using the available on-chip processing to accomplish DARPA's goal of creating an imaging system that can autonomously toggle between different imaging modes. One potentially fruitful area is machine learning: specifically, the use of convolutional neural networks (CNNs) similar to those that are already frequently used for image classification, like VGGNet and ResNet [25]. The Griffin FPIA's resources are constrained relative to the powerful GPUs or TPUs often used in machine vision, however, and so may not be appropriate for CNNs with computational costs measuring in the tens of billions of FLOPs. Therefore, it is worthwhile to examine machine vision techniques that are less computationally intensive for the edge devices involved. These include a new technique called split learning, which is being developed by MIT Media Lab's Camera Culture group in conjunction with the MIT Alliance for Distributed and Private Machine Learning.

Table 2.1: Technical summary of the Griffin FPIA [52].



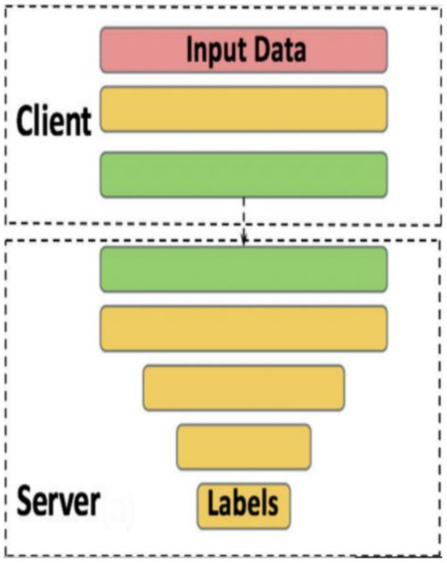
Resource Type	# Resources on Griffin FPIA	Resources/Pixel
Macropixel Tiles	20,480	1 per 8x8 sub array
Lookup Tables	275,032	N/A
Deserializer Tiles	576	1 per perimeter 8x8 sub array
DSP Tiles	1736	N/A
Memory Tiles (8K 16-bit words)	820	N/A
Perimeter GPIO	275	N/A
3D GPIO	163,240	8 per 8x8 sub array
3D Pixel Inputs	1,310,720	1 per pixel
LVDS Tx Pairs, 300K	18	N/A
LVDS Tx Pairs, 77K	9	N/A

2.1.2 Overview: Split Learning

Split learning is a new distributed deep learning method being developed by the MIT Media Lab's Camera Culture group. The goal of split learning is to preserve the privacy of clients' raw data by allowing servers to use deep learning models for training and/or inference without that data being shared directly [2]. This project is a part of the MIT Alliance for Distributed and Private Machine Learning, which has the goal of reducing the friction in data sharing that currently presents a challenge to large-scale machine learning. The split learning project is a part of their privacy-preserving machine learning research area. Additional research areas include automated machine learning and data markets [1].

At its simplest, split learning works by having the client, in possession of the input data, train the first layers of a deep network (such as a CNN, the type of neural network this thesis will primarily focus on). At a specific layer designated as the "cut layer," the client sends the outputs of that layer to a server, which then completes the rest of the training. The resulting gradients are then propagated back to the cut layer, and the gradients at the cut layer are sent back to the client. Finally, the client back-propagates gradients from the cut layer to the beginning of the network, and the process repeats until the split learning network is trained. Split learning, then, allows a full deep network to be trained (and subsequently used for inference) without the server ever having possession of the client's raw data [46].

Figure 2-3: Simplest split learning model architecture (single client) [46].



Split learning also allows for more complicated configurations, such as configurations where the data are "vertically partitioned" (i.e. spread between multiple clients that do not communicate with each other) or where the final layers are computed by the client, so that the resulting labels are not shared with the server. This flexibility makes split learning a tool that may be used in a wide variety of applications in sectors including finance and health [1]. Split learning can also be used in conjunction with techniques to reduce the invertibility of intermediate representations, thereby reducing the ability of the server to reconstruct the client's raw data from the outputs from the cut layer that the client sends. Members of the Camera Culture group have demonstrated a method that can reduce distance correlation (as measured by KL-divergence) from 0.95 to 0.19 in one example and 0.92 to 0.33 in another, without a corresponding loss of model accuracy [45]. This further reduces the server's ability to make inferences about client raw data.

Split learning was not the first distributed deep learning method to be developed, but in certain settings it has so far been shown to be the most accurate and resource-efficient. We will briefly describe two other prominent distributed deep learning methods, federated learning and large-batch synchronous stochastic gradient descent (SGD), and compare their performance with split learning.

In large-batch synchronous SGD, all clients are sent a copy of the current model by the server. Each client then uses its local dataset to train its copy of the model further, then sends the updated model weights to the server. The server then aggregates that information to improve the canonical model and repeats the process. Unlike previous asynchronous SGD approaches, synchronous SGD synchronizes updates between the server and client so that clients are not wasting resources and adding noise by updating old copies of the model. It also adds additional "workers" to compensate for the slowest clients, so that the server does not have to rely on the slowest client to provide its updates to update the canonical model [13]. Like split learning, large-batch synchronous SGD does not share raw data with the server.

Federated learning is very similar to large-batch synchronous SGD. In federated learning, however, only a fraction of the clients are sent a copy of the current model by the server. Each client then uses its local dataset to train its copy of the model further. When training its copy of the model, the client makes multiple passes, rather than one single update, then sends the updated model weights to the server. The server then aggregates that information to improve the canonical model and repeats the process. These changes make federated learning a more appropriate form of distributed deep learning than large-batch synchronous SGD in cases where there is a large number of clients, where client datasets are not very uniform in terms of size and distribution, and client communication may be limited [32]. Like split learning and large-batch synchronous SGD, federated learning does not share raw data with the server.

In experiments conducted using widely-used image datasets and image classification neural networks, split learning achieved much higher accuracy than large-batch SGD and federated learning methods on 100- and 500-client trials [47]. As shown in Figure 2-4, the validation accuracy of the neural network trained using the split learning method improved much more quickly (in terms of client-side computation required) than those trained using the federated learning or large-batch synchronous SGD methods.

Both the federated learning or large-batch synchronous SGD methods eventually achieved similar validation accuracy, but required significantly more computation and bandwidth than the split learning method. This difference is illustrated in Tables 2.2 and 2.3.

Figure 2-4: Validation accuracy vs. client-side computation (left: 100 clients with VGG neural network trained on CIFAR 10 dataset; right: 500 clients with Resnet-50 neural network trained on CIFAR 100 dataset) [47].

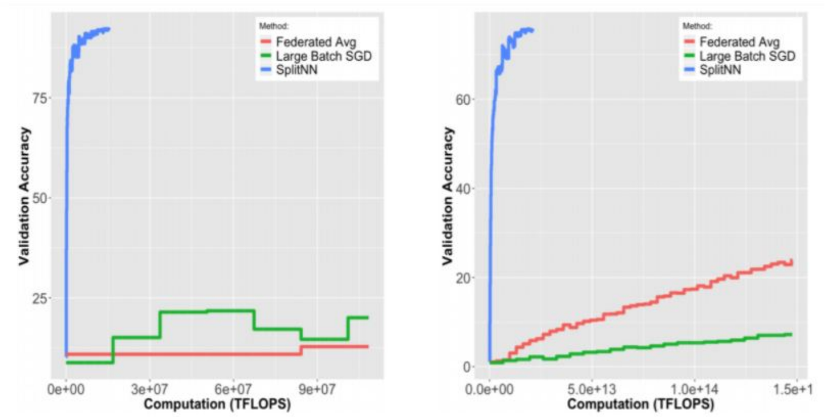
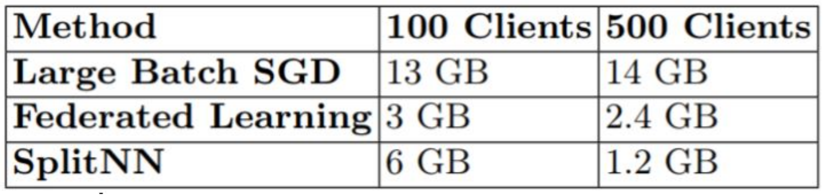
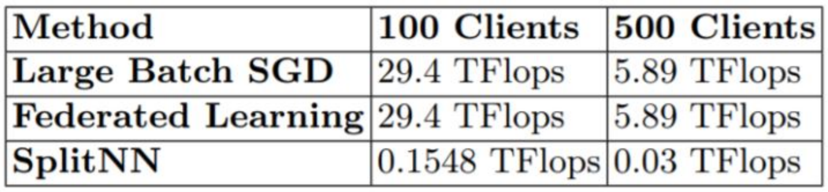


Table 2.2: Per-client bandwidth when training ResNet-50 neural network on CIFAR 100 dataset) [47].



Method	**100 Clients**	**500 Clients**
Large Batch SGD	13 GB	14 GB
Federated Learning	3 GB	2.4 GB
SplitNN	6 GB	1.2 GB

Table 2.3: Per-client computation when training VGG neural network on CIFAR 10 dataset) [47].



Method	**100 Clients**	**500 Clients**
Large Batch SGD	29.4 TFlops	5.89 TFlops
Federated Learning	29.4 TFlops	5.89 TFlops
SplitNN	0.1548 TFlops	0.03 TFlops

The communication and computation efficiency of split learning makes it a good candidate for implementation on chips like the Griffin FPIA, which is resource-constrained relative to the CPUs, GPUs, and other microprocessors commonly used in machine learning applications. However, it is important to note that in cases where the number of clients and/or the number of model parameters is relatively low, federated learning can be more communication efficient than split learning [39]. Therefore, in use cases with a notably low number of clients or model parameters where bandwidth limitations are the limiting factor for a successful split learning application, federated learning may be a more appropriate method of training neural networks. Potential use cases and their numbers of clients and parameters are explored in Chapter 4.

Split learning and other forms of distributed deep learning have a wide variety of potential uses. Their flexibility allows them to be used in a number of different settings, including within individual datacenters, between multiple datacenters holding different kinds of data, and across very large numbers of mobile devices. A number of applications in fields like medicine, finance, and manufacturing have already been proposed [24]. This combination of efficiency and flexibility makes split learning an appropriate choice to study in relation to ReImagine, a project that is intended to be adaptable to a wide range of use cases, including those that may even be determined after the hardware itself has been deployed. In Chapter 4, we will further examine the potential areas where split learning might be fruitfully executed on the Griffin FPIA based on number of clients, distribution of data between devices, and the importance of offering privacy guarantees in different use cases.

2.2 Literature Review

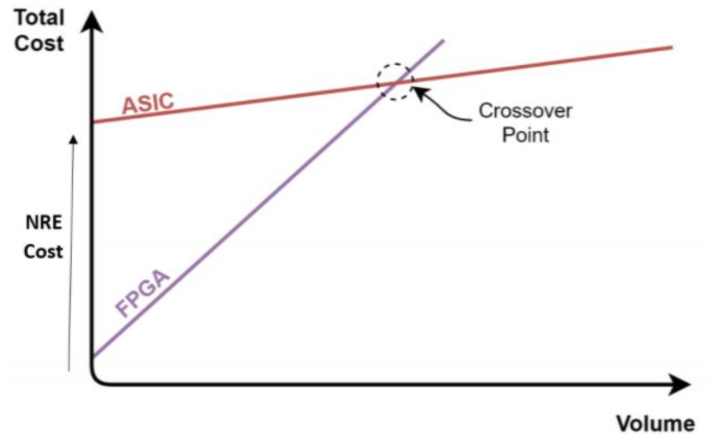
In this section, we offer an overview of FPGAs in general, and give background on how the technology works. Next, we offer an overview of their advantages and disadvantages compared with other commonly used integrated circuits, such as ASICs and microprocessors, in order to justify why a split learning application on an FPGA might on balance offer advantages in cost, performance, and flexibility. Finally, we will examine the relationship of FPGAs to the three major areas that would enable the productive use of split learning applications on imaging systems like ReImagine: the proliferation of the use of FPGAs in sensor systems, the general use of machine learning applications on FPGAs, and capabilities related to security and privacy that FPGAs offer.

2.2.1 What is an FPGA?

An FPGA is, at its most basic level, an integrated circuit that can be reconfigured by the user after it has been manufactured, allowing a single device to be used for a wide variety of different applications by a number of different end users. The world's first commercial FPGA, the Xilinx XC2064, debuted in the mid-1980s. It was designed to improve on existing devices that already supported programmable logic, like Programmable Array Logic (PALs), which were difficult to scale up without becoming slow and physically unwieldy [43]. Both PALs and FPGAs developed as an alternative to application-specific integrated circuits (ASICs), non-reprogrammable integrated circuits which were at the time the dominant form of custom IC.

ASICs were fast, easy to use, and generally cheap to manufacture per unit. However, ASICs required large non-recurring engineering (NRE) costs that could not be amortized across different customers or projects. Additionally, ASICs faced a large number of design failures and product changes during the design process that meant that meant many designs never went to market, and did not realize any profit for the manufacturer or the customer [43]. While ASICs are still in use today, particularly in applications where computations must be performed as fast as possible or where the number of units being produced is in the millions, in many cases FPGAs offer significant advantages. Despite their higher per-unit costs, FPGAs may be overall cheaper for customers due to the amortization of NRE costs across many different customers, and their reprogrammability, which may allow FPGAs to be used in many different applications. Additionally, FPGAs faced fewer design problems related to certain elements like I/O blocks, which no longer needed to be designed from scratch for each new application.

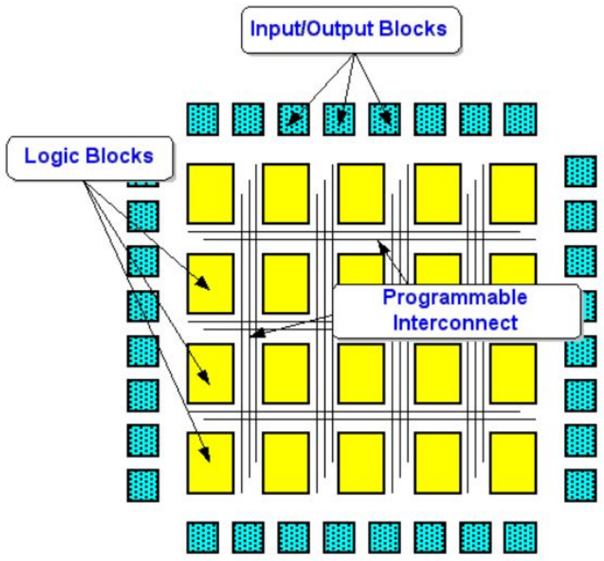
Figure 2-5: Visualization of how NRE cost savings affect individual clients. For a number of units to the left of the crossover point, the total cost of an FPGA is cheaper than that of an ASIC [35].



The XC2064 was tiny by modern standards—it only contained 64 logic blocks, for a total of fewer than 1000 gates [43]. FPGAs that small could be programmed entirely by hand, without the tools for automated synthesis, placement, and routing required in order to program large, modern FPGAs [41]. However, their basic architecture was the same: a grid-like layout of configurable logic blocks (CLBs) that are used to implement arbitrary boolean functions, connected to one another by programmable interconnects, and surrounded by I/O blocks to allow data to be read on and off the chip. Today's CLBs are implemented in look-up tables (LUTs).

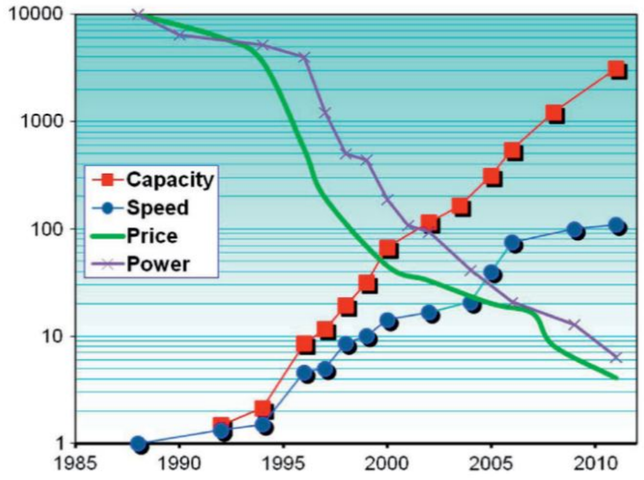
Over the next thirty years, the density of transistors in integrated circuits increased exponentially with Moore's Law. This, along with new manufacturing processes in foundries, allowed the capacity (in terms of CLB count) and speed (in terms of same-function performance) of FPGAs to skyrocket, even as their price and power usage per CLB decreased [43]. These changes made FPGAs desirable for larger and larger applications, eventually making automated tools for the synthesis, placement, and routing of FPGA programs not just desirable but required. New hardware-descriptive languages (HDLs) emerged, most famously Verilog and VHDL, allowing developers to automate the low-level programming of the device and create larger and more complex applications [41]. HDLs and other, higher-level synthesis tools will be

Figure 2-6: General FPGA architecture block diagram [4].



discussed further in Section 2.2.5.

Figure 2-7: Xilinx FPGA attributes relative to 1988 [43].



Modern FPGAs include not only configurable logic blocks, but dedicated logic blocks that allow them to comply with widely used communications standards and offer further cost and performance savings, as well as offer additional functionality like protection of the FPGA application design through encryption. These dedicated blocks include memory blocks, microprocessors, bitstream encryption, and ethernet connections [43]. Many modern FPGAs contain heterogenous computing elements on a single chip, and are referred to as System on a Chip (SoC) FPGAs.

2.2.2 FPGAs, ASICs, and Microprocessors

When examining the usefulness of FPGAs, it is important to compare them to the two other categories of computing hardware that can also be used for the implementation of any given application. The two other categories are ASICs, very fast and efficient custom integrated circuits that are designed for specific applications, and processors like CPUs and GPUs, which are the most widely used type of computing hardware in machine learning applications, and are extremely cheap, flexible, and easy to program.

In the previous section, we discussed what limitations on ASICs spurred the invention and development of FPGAs. FPGAs also offer a number of advantages compared to CPUs and GPUs, which are generally the same as their disadvantages when compared with ASICs. In fact, it is useful to think of FPGAs in general as a middle ground between ASICs and microprocessors—in most areas where ASICs perform well and CPUs/GPUs perform poorly, or vice versa, FPGAs fall somewhere in between. This means that, for example, for applications that require more flexibility than ASICs can provide but also require hardware that consumes less power than CPUs or GPUs do, FPGAs provide a desirable or vital alternative.

The major exception to the rule of FPGAs' position as a middle ground between ASICs and CPUs/GPUs is programmability. By programmability, we don't mean *reconfigurability* but the ease of development of new applications by developers. Where ASICs and CPUs/GPUs require either little programming at all or support a large number of very popular high-level languages that are relatively easy for human developers to understand, tools for high-level synthesis on FPGAs, while a rapidly growing area, are much less robust. High-level synthesis tools for FPGAs will be discussed further in Section 2.2.5.

Below, Figure 2.4 offers a visualization of broad trends in the attributes of FPGAs as compared to ASICs and processors like CPUs and GPUs. These attributes have influenced the use of FPGAs in sensor systems and to implement machine learning applications, efforts which will be discussed in the next two sections.

Table 2.4: A visualization comparing the attributes of FPGAs to ASICs and CPU-like processors.

	Processor	FPGA	ASIC
Design Cycle	🟩	🟨	🟥
Flexibility	🟩	🟩	🟥
Performance	🟥	🟨	🟩
Power	🟥	🟨	🟩
Programmability	🟩	🟥	🟩

2.2.3 FPGAs in Sensor Systems

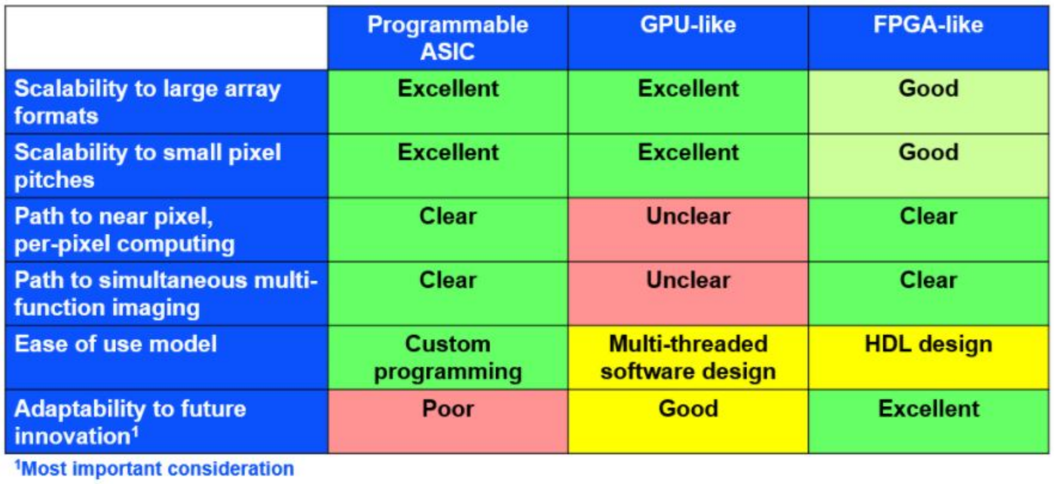
When we discuss sensor systems, we refer to a set of one or more sensor nodes, each containing a sensor, a processing element, a power source, and a transceiver that allows the node to communicate with other nodes or a central server. The sensor systems we are most concerned with in this paper are imaging systems like ReImagine.

Many sensor systems were developed for use in military applications, such as the identification, classification and tracking of objects in the battlefield [15]. However, due to the flexibility of sensors available, their underlying processing elements, and the possible configurations of networks of nodes, sensor systems are used in a wide variety of domains, from controls to agriculture to health to environmental monitoring. In general, the implementation of sensor systems is limited by the fabrication costs and power consumption of the individual nodes [15]. As discussed previously, ASICs that consume little power incur very large non-recurring engineering costs, which are often not amortized across enough devices to make their development cost effective. Processors like CPUs and GPUs, however, require significantly more power, and therefore much larger or more frequently replaced batteries. Often, nodes would have benefited from being smaller and lighter, or else are widely distributed enough to make manually changing or recharging batteries a time-consuming task.

FPGAs do not have to be used as standalone processing elements in sensor nodes, but can also be combined with other processing elements to take advantage of FPGAs' particular benefits as an accelerator for tasks they may be overall less suited for [15]. Each sensor system has its own particular set of requirements, which may or may not favor the use of FPGAs. Imaging systems with computer vision applications, for example, might simply do image capture and simple arithmetic operations, might do more complex segmentation, preprocessing, or convolution, or do highly complex image recognition and classification tasks [20]. Even among imaging systems, different requirements might favor dramatically different types of hardware.

The ReImagine project is an example of one such system. The requirements particular to ReImagine that go beyond general trends in power, programmability, and performance and that led to the implementation of its processing elements as FPGAs are summarized in Table 2.5.

Table 2.5: Hardware attribute considerations specific to ReImagine [52].



	Programmable ASIC	GPU-like	FPGA-like
Scalability to large array formats	Excellent	Excellent	Good
Scalability to small pixel pitches	Excellent	Excellent	Good
Path to near pixel, per-pixel computing	Clear	Unclear	Clear
Path to simultaneous multi-function imaging	Clear	Unclear	Clear
Ease of use model	Custom programming	Multi-threaded software design	HDL design
Adaptability to future innovation[1]	Poor	Good	Excellent

[1]Most important consideration

A more detailed analysis of the trends in the use of FPGAs in sensor systems will be given in Chapter 4. It is worth noting here, however, that the use of FPGAs in sensor systems is already occurring across a number of domains, including in synthetic-aperture radar systems to create detailed images of landscapes [29] and a number of different medical imaging applications, including nuclear medical imaging [54] and a number of different portable, high-frame rate ultrasound imaging systems [33][50][12]. Health in particular is a domain that is highly relevant to early split learning efforts, and could demonstrate fruitful uses of split learning on FPGAs. In the next two sections, we will review the attributes of FPGAs as they relate to two more areas vital to split learning efforts: machine learning and security and privacy.

2.2.4 Security and Privacy on FPGAs

Split learning applications are designed to help preserve privacy by preventing the sharing of clients' raw data with central servers. However, in addition to preventing the sharing of raw data or easily invertible intermediate representations through split learning and differential privacy techniques, split learning implementations must also account for two other privacy concerns. The first concern is the way the information flow of intermediate representations might leave the model vulnerable to malicious client or server actors. The second concern is verifiability, which allows clients or servers to prove they are faithfully executing the model without sharing raw data being used in that execution [24].

The latter two concerns go beyond questions of the design of split learning algorithms and depend in large part on the hardware on which these algorithms are executed. In particular, these concerns depend on the execution of algorithms in trusted execution environments (TEEs), secure, isolated areas of processors that provide guarantees of confidentiality (that the state of the code's execution remains secret), integrity (that the code's execution cannot be affected by an outside source) and attestation (that the code is executing faithfully based on its starting state) [24].

Security and privacy have also long been concerns for developers who design and use FPGAs. As discussed in Section 2.2.1, one of the earliest uses of dedicated logic blocks on FPGAs was for bitstream encryption, which protected the code being run on the FPGA from being read while it was being loaded onto the chip, or from being read off the chip itself. These blocks are now near-ubiquitous among commercial FPGAs.

FPGA developers leverage a number of additional techniques for device security throughout its life cycle. These techniques are leveraged during design manufacturing of the base array itself, the application design phase, the configuration of the FPGA, and the actual execution of applications on the device [42]. These techniques include, but are not limited to:

- Programs like the DoD Trusted Foundry Program that allow military and other government buyers to access trusted and assured microelectronics specific to their needs [8].
- Hardware trojan detection techniques like FANCI and VeriTrust that allow for the detection of malicious hardware trojans inserted during the application design phase [53].
- Bitstream authentication methods that detect tampering with the application design [42].
- On-chip cryptography blocks that allow data encryption, along with physically unclonable functions and other attributes unique to the physical device that may serve as a unique identifier or private decryption key [42].
- The use of existing secure enclaves like the ARM TrustZone on SoC FPGAs [19].

In Chapter 4, we will analyze which of the existing security and privacy features of FPGAs may make them suitable for achieving the security and privacy goals of split learning applications, and where FPGAs may still fall short as a platform.

2.2.5 Machine Learning on FPGAs

Possibly the clearest proof of the potential of machine learning applications on FPGAs is that a number of such applications are in use today. FPGAs are currently used to supplement CPUs/GPUs and accelerate CNNs, similar to the type of joint implementation sometimes used in FPGA-based sensor systems discussed briefly in Section 2.2.3. There are also a number of standalone FPGA-based CNN implementations. In all cases the use of FPGAs allows the developer to leverage the parallelism inherent to CNNs [6]. Despite these recent successes, there are a number of disadvantages to conducting machine learning on FPGAs that have so far limited these joint implementations, as well as a number of advantages that allow for future growth. The following sections will discuss the advantages and disadvantages of machine learning on FPGAs.

Disadvantages of Machine Learning on FPGAs

We begin our survey by examining some of the disadvantages that might deter or complicate the implementation of machine learning algorithms on FPGAs on a wider scale. These include domain-specific knowledge requirements, longer development times, and resource constraints unique to FPGAs.

Domain-Specific Knowledge Requirements Historically, application development for FPGAs has required fluency in hardware descriptive languages (HDLs), most commonly Verilog and VHDL. Programming in HDLs, unlike with traditional high-level programming languages, requires an understanding of hardware-level details of the implementation, particularly if the developer wishes to take full advantage of the optimizations that are possible with FPGAs, including reduction in latency due to the proximity of the detector to the on-chip processing elements in imaging systems like ReImagine, the ability take advantage of the parallelism inherent to the way FPGA applications can be routed through the device, and significantly reduced power consumption of FPGAs compared to CPUs and GPUs. Experience with software languages does not translate directly to proficiency with HDLs, which means the software developers that traditionally write applications do not typically favor FPGA implementations [26]. This disadvantage is increasingly mitigated by the emergence of high-level synthesis (HLS) tools, which automate the synthesis of designs at the register-transfer level (RTL) and below, allowing developers to create efficient applications for FPGAs by describing them at the algorithmic level. The existence of HLS tools also allows for the proliferation of libraries of machine learning resources for FPGAs. A survey of useful HLS tools is found in Section 3.1.

Longer Development Times Another disadvantage of developing machine learning algorithms for FPGAs is increased development time. This is mainly a function of longer compile times. For FPGAs, the process of placing and routing a design can take on the order of minutes to hours, as opposed to seconds or milliseconds for processors like CPUs/GPUs [48]. This orders-of-magnitude difference is a high cost to developers, who may need to compile many iterations of an application during the design phase. FPGA's dynamic reconfiguration abilities, while useful, are also susceptible to delays and increased runtime. This cost of reconfiguration can be mitigated to some extent by the use of pre-compiled compute kernels, or by the use of kernel simulation during many parts of the debugging process [26]. Still, the impact on developers is significant.

Resource Constraints The third major disadvantage of the use of FPGAs in machine learning is that of resource constraints. Implementations of very simple neural networks on FPGA actually emerged in 1992; however, in addition to the nascence of the field of machine learning, their most important limitation was the size constraints of FPGAs, as well as the number of operations per second they could handle [14]. As discussed in Section 2.2.1, FPGAs have evolved dramatically in the decades following this implementation, and now typically contain hardened multiply-accumulate (MAC) blocks, which allow them to do arithmetic much more quickly and efficiently. Additionally, the density of transistors on FPGAs has increased dramatically, increasing the complexity of applications that can be implemented on them as well as decreasing the time and power those applications consume. Although these trends still continue today, and are leveraged in modern designs of machine learning algorithms of FPGA, resource constraints remain a significant consideration [38].

The specific capabilities of FPGAs, in terms of storage, external memory bandwidth, and computational resources available, are highly dependent on which FPGA is being discussed. A summary of the available resources on a subset of Altera (acquired by Intel in 2015) and Xilinx devices, demonstrating the wide range of FPGA capacities, is shown in Table 2.6. However, popular and powerful image classification algorithms, such as AlexNet or VGGNet, which have tens of millions of parameters and require billions of operations per second, require more storage than is available on most commercial FPGAs. Therefore, model weights must be stored externally and transferred to the FPGA during evaluation of the model, significantly slowing computation. Repeatedly accessing external memory reduces the advantages of architectures like ReImagine, for which having computing resources physically very close to image inputs is an enormous advantage. This limitation makes careful optimization of FPGA designs and parallelization that reflects the requirements of a specific layer of a given convolutional neural network paramount.

Table 2.6: Select Altera and Xilinx devices with available resources [37].

Year	Feature Size	Xilinx FPGA family		Device	LUTs	DSP/Mult blocks	BRAM Kbits	LUTs /DSP	LUTs /BRAM	Altera FPGA family		Device	ALMs (LEs)	DSP /Mult blocks	BRAM Kbits	LEs/ DSP	LEs/ BRAM
2011	28 nm	Virtex 7	V	XC7V2000T	1,221,600	2,160	46,512	566	26								
			VX	XC7VX1140T	712,000	3,600	67,680	198	11								
			VH	XC7VH870T	547,600	2,520	50,760	217	11								
2010										Stratix V	GT	5SGTC7	622,000	512	50,000	1,215	12
											GX	5SGXBB	952,000	704	52,000	1,352	18
											GS	5SGSD8	695,000	3,926	50,000	177	14
											E	5SEEB	952,000	704	52,000	1,352	18
2009	40 nm	Virtex 6	LX	XC6VLX760	474,240	864	25,920	549	18								
			SX	XC6VSX475T	297,600	2,016	38,304	148	8								
			HX	XC6VHX565T	354,240	864	32,832	410	11								
2008										Stratix IV	GT	EP4S100G5	531,200	1,024	27,376	519	19
											GX	EP4SGX530	531,200	1,024	27,376	519	19
											E	EP4SE820	813,050	960	33,294	847	24
2006	65 nm	Virtex 5	LX	XC5VLX330	207,360	192	10,368	1,080	20								
			SX	XC5VSX240T	149,760	1,056	18,576	142	8								
			FX	XC5VFX200T	122,880	384	16,416	320	7								
										Stratix III	L	EP3SL340	337,500	576	16,272	586	21
											E	EP3SE260	255,000	768	14,688	332	17
2005	90 nm									Stratix II	GX	EP2SGX130/G	132,540	252	6,747	526	20
	130 nm											EP2S180	179,400	384	9,383	467	19
2004	90 nm	Virtex 4	LX	XC4VLX200	178,176	96	6,048	1,856	29								
			SX	XC4VSX55	49,152	512	5,760	96	9								
			FX	XC4VFX140	126,336	192	9,936	658	13								
2002	130 nm									Stratix	GX	EP1SGX40D	41,250	56	3,423	737	12
												EP1S80	79,040	88	7,428	898	11
2001	130 nm	Virtex II	Pro	XC2VP100	88,192	444	7,992	199	11								
			Pro X	XC2VPX70	66,176	308	5,544	215	12								
	0.15 um		V	XC2V8000	93,184	168	3,024	555	31	Mercury		EP1M350	14,400	0	115	-	125
2000	0.18 um									Excalibur		EPXA10	38,400	0	3,146	-	12
1999	0.18 um	Virtex E		XCV3200E	64,896	0	851	-	76								
1998	0.22 um									Flex 10KE		EPF10K200E	9,984	0	98	-	102
	0.25 um	Virtex		XCV1000	24,576	0	131	-	188								
1997	0.35 um	4000 E/XL		XC4085XL	12,544	0	0	-	-								
1996	0.3 um									Flex 10KA		EPF10K250A	12,160	0	41	-	297
1995	0.42 um									Flex 10K		EPF10K100	4,992	0	25	-	200
1992	0.6 um									Flex 8000		EPF81500A	1,296	0	0	-	-
1991	0.8um	4000 series		XC4025	2,048	0	0	-	-								
1985	2 um	2000 series		XC2018	400	0	0	-	-								

FPGA storage and computation are improving (as are interconnects that allow multi-FPGA configurations) just as machine learning algorithms are growing in size and complexity. Therefore, resource constraints will be a perennial challenge [38]. It should also be noted, however, that in cases where split learning is implemented, the existence of a cut layer relieves much of the storage and computation burden from the sensing device, as a number of the later layers are handed off to a central server for computation.

Advantages of Machine Learning on FPGAs

The disadvantages of FPGAs are offset by a number of high-level tools that can enable more rapid development of FPGA applications, as well as advantages inherent to FPGA hardware in terms of performance, flexibility, and potential for optimization.

High-Level Synthesis Tools The two major commercial vendors of FPGAs, Xilinx and Intel, have both long supported the use of HLS tools, which allow developers to avoid programming FPGAs at the RTL. There are many types of FPGAs, from HDL-based frameworks like SystemVerilog to C-based frameworks like Xilinx Vivado to CUDA/OpenCL-based frameworks with a strong focus on parallelism [7]. These HLS tools make FPGA programming accessible to many developers who would otherwise focus on CPUs and GPUs, and allow the development of complicated applications in a much shorter period of time, and in many fewer lines of code. HLS tools can also automatically perform optimizations that are difficult or time-consuming to achieve when programming at a lower level [26]. Section 3.1 presents a summary of the HLS tools that include interfaces for machine learning on FPGAs.

Optimization and Flexibility Another advantage of the use of FPGAs in machine learning is the ability to pursue optimizations that are not possible on microprocessors. On FPGAs, the bottleneck in communication between processor and memory can be alleviated by the laying of flexible data paths, and the programmer can take advantage of distributed on-chip memory. FPGAs are also highly parallelizable, and well-suited for the feed-forward nature of many machine learning models. Additionally, modern FPGAs generally support partial dynamic reconfiguration. This means that in a machine learning model implemented on an FPGA, individual layers can be reprogrammed while computation is still occurring in other layers, alleviating some of the memory constraints that are inherent to FPGAs. Together, these capabilities allow programmers to pursue optimizations that simply cannot occur on processors like CPUs and GPUs, although many types of hardware-level optimizations cannot be implemented using HLS tools, removing that advantage [26].

As discussed previously, many of the hardware optimization advantages that exist for FPGAs, but not for CPUs/GPUs, are also present in ASICs. However, ASIC platforms have extremely long design cycles compared to FPGAs, and are not reconfigurable once a design cycle has been completed. FPGAs, on the other hand, are highly flexible, allowing their designs to be optimized even after they've been deployed, or reconfigured for a new application entirely [48]. This flexibility can save both time and development costs in the design phase, especially given the rapid improvements on and changes to machine learning algorithms.

Performance Improvements FPGAs generally offer very high performance per watt, especially compared to processors like GPUs. As an example, one 2017 comparison of an LSTM algorithm implemented on an FPGA, a GPU, and a CPU found that the FGPA was 11.5 times more energy efficient than the GPU, and 40 times more energy efficient than the CPU [22]. The same algorithm was 3 times faster than the GPU and 43 times faster than the CPU. Similar results across a variety of studies [38] demonstrate that the power efficiency of FPGAs can be a significant advantage in two cases. On one hand, large-scale deployments of FPGAs in server-based applications can result in significant energy savings. On the other, in cases where FPGAs are embedded in systems with significant resource limitations can make possible applications that might otherwise be simply too energy intensive to run [26]. The speedups of the algorithms also constitute a significant advantage, as they increase throughput, meaning less time and/or fewer devices are required to implement the algorithm, resulting in cost and time savings.

Chapter 3

Implementation of Split Learning on FPGAs

When software developers program general-purpose processors like CPUs, they generally use high-level, human-readable languages like C or Python, which are then compiled or interpreted into binary or bytecode that the CPU can execute. Unlike CPUs, FPGAs must be configured with a bitstream that is loaded onto the device and determines how the device is programmed.

As is the case with binary executables, it is not feasible for humans to generate bitstreams of any significant complexity without the aid of higher-level languages. However, the toolchain required to translate high-level languages to an FPGA bitstream is significantly different from the toochain required to create executables for CPUs. In general, high-level languages for FPGAs require greater knowledge of the underlying hardware, and are often more difficult to use than high-level languages for CPUs. A high-level language must be translated into code in a hardware descriptive language (HDL) using a high-level synthesis tool. The HDL code must then be synthesized by a synthesis tool like Xilinx Vivado into a bitstream specific to the model of FPGA it will be loaded onto.

In order to demonstrate the feasibility of implementing split learning algorithms on FPGAs, we must understand the landscape of high-level synthesis tools available that allow developers to translate models from common machine learning frameworks like PyTorch and Tensorflow to hardware descriptive languages like Verilog and VHDL. We also must evaluate the resources available on modern commercial FPGAs and select a split learning algorithm to implement. This chapter will discuss those three areas, describe our synthesis of a split model on an FPGA, and provide analysis on what the demonstration implies for future split learning implementations on ReImagine and similar imaging platforms.

3.1 Interfaces for Machine Learning on FPGAs

A number of high-level synthesis tools exist that empower developers to more easily design for FPGAs, as discussed in Section 2.2.5. Many of these tools are based on CUDA or OpenCL, and are designed specifically for FPGA developers interested in implementing neural networks. While CUDA is used frequently in popular deep learning tools, it is also proprietary. Most of the tools we will look at use the parallel programming platform OpenCL, which is open-source and currently offers support for a wide variety of deep learning frameworks, notably Caffe [26].

These OpenCL-based implementations range from frameworks developed by academic researchers (for example, the "caffeinated FPGAs" framework developed in 2016 [16]), to more extensive development environments released by industry. For example, Intel released the Intel FPGA SDK for OpenCL in 2016, implementing the AlexNet image classification algorithm with performance similar to that of a high-end GPU in terms of images per second, and with much greater power efficiency [23]. At the end of 2019, Xilinx introduced Vitis AI, an extensive development environment that contains a library of existing models, and offers support for mainstream frameworks like Caffe and TensorFlow, as well as C++ and Python APIs [51].

Another important emerging HLS tool for machine learning development on FPGAs is hls4ml [18], a package for machine learning inference in FPGAs that is intended for use in low-latency particle physics application, and translates trained machine learning models from frameworks like PyTorch and Keras to Vivado HLS code, which can then be translated by Vivado to a HDL. hls4ml is the package which we have chosen to use to implement our split learning algorithm in Verilog code.

While support for machine learning on FPGAs is still not as extensive as it is for processors like CPUs, the field continues to see very rapid development as FPGA developers seek easier means of implementing neural networks, and FPGA manufacturers seek to expand the range of applications for their devices. With time, the available set of HLS tools will grow more powerful and accessible, increasing the feasibility of machine learning on FPGAs.

3.1.1 Sample Workflow

The possible workflows for the implementation of machine learning algorithms on FPGAs are varied, but they share a common overall structure, namely the translation of a model description and target platform specifications to a specialized model (often described in C or C++), then to Vivado HLS code, and then to Verilog. In this section, we will examine the workflow of one common tool, fpgaConvNet, in order to provide an example of how implementation might look for a developer.

fpgaConvNet was first released in May 2016 by the Intelligent Digital Systems Lab at Imperial College London. It supports input from the developer using both Caffe and Torch. fpgaConvNet also takes in the specifications of the FPGA to be targeted. fpgaConvNet automatically converts this input into Vivado HLS, which means it is limited to Xilinx FPGAs. The Vivado HLS code is then input to Vivado, which generates a bitstream that can be loaded directly onto an FPGA, as well as a report on the resource utilizations of the design [44].

The internal workflow of fpgaConvNet, which is very similar to that of hls4ml, is shown in Figure 3-1, and benchmarks for the performance of several common image recognition algorithms as generated by fpgaConvNet and implemented on FPGAs are shown in Table 3.1.

Figure 3-1: Flowchart of the fpgaConvNet framework [30].

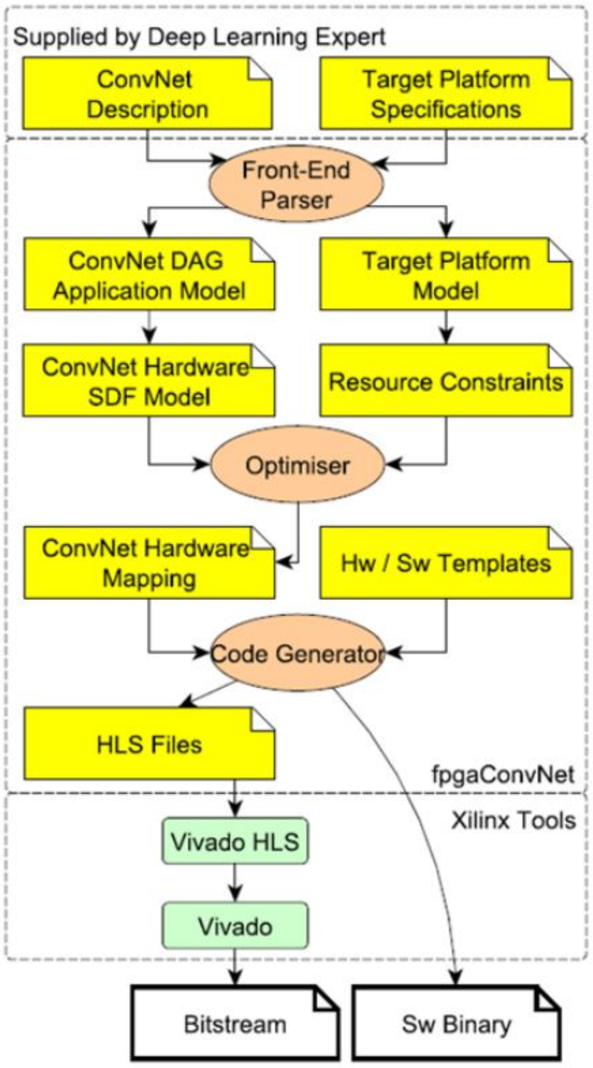
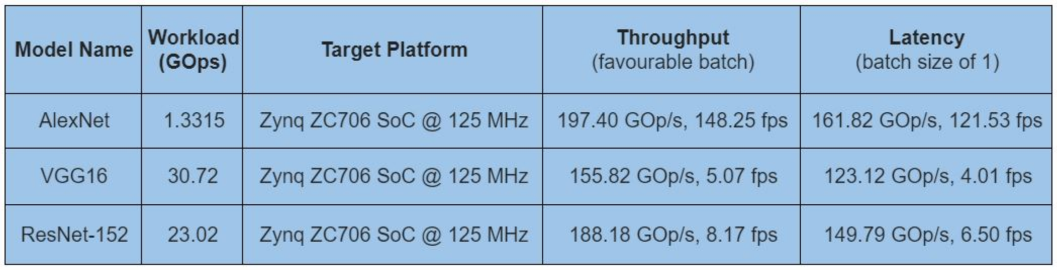


Table 3.1: Performance of common object recognition algorithms implemented using fpgaConvNet [30].



Model Name	**Workload (GOps)**	**Target Platform**	**Throughput** (favourable batch)	**Latency** (batch size of 1)
AlexNet	1.3315	Zynq ZC706 SoC @ 125 MHz	197.40 GOp/s, 148.25 fps	161.82 GOp/s, 121.53 fps
VGG16	30.72	Zynq ZC706 SoC @ 125 MHz	155.82 GOp/s, 5.07 fps	123.12 GOp/s, 4.01 fps
ResNet-152	23.02	Zynq ZC706 SoC @ 125 MHz	188.18 GOp/s, 8.17 fps	149.79 GOp/s, 6.50 fps

3.2 FPGA Resources

A summary of the resources on a subset of modern, commercially available FPGAs was provided in Section 2.2.5. Additionally, an analysis of the resources available on the first generation Griffin FPIA, developed through the ReImagine program, was provided in Section 2.1.

Comparing these tables, we see that ReImagine, though a unique platform, contains resources within the range of existing commercial FPGAs (it contains 275,032 LUTs and 1,736 DSP tiles, very similar to Xilinx's Virtex 6 SX's 290,600 LUTs and 2,016 DSP tiles), allowing us to compute resource consumption of designs on available commercial FPGA synthesis tools, specifically Xilinx's Vivado, that will be able to guide estimates of resource consumption on platforms like ReImagine, which are programmed using MIT Lincoln Laboratory's programming toolset for ReImagine, with an eye towards future applications.

3.3 Split Learning Algorithm

For our split learning algorithm, we modified a split learning neural network developed by the MIT Media Lab that uses four clients and a single server to train and evaluate a convolutional neural network that classifies handwritten digits from the MNIST database. The modifications of the network included the translation of the model from PyTorch to Keras due to hls4ml's current support for 2D convolutional layers in Keras, the use of a single client to simplify FPGA simulation, and the reduction in size of the first convolutional layer, which produced memory and runtime issues in Vivado HLS at its original size. The final model architecture is shown below, and had a validation accuracy of 98.9% over ten epochs:

```
client = Sequential()
client.add(Conv2D(16, (3, 3), input_shape=(input_shape)))
client.add(Activation('relu'))
client.add(MaxPooling2D(pool_size=(2, 2)))
client.add(Conv2D(32, (3, 3)))
client.add(Activation('relu'))
client.add(MaxPooling2D(pool_size=(2, 2)))

server = Sequential()
server.add(Flatten())
server.add(Dense(256))
server.add(Activation('relu'))
server.add(Dense(10))
server.add(Activation('softmax'))
```

3.4 Implementation

The code run on Google Colaboratory and used to generate the model architecture and weight files is included in Appendix A. The resulting JSON file containing the architecture of the sequential Keras model is included in Appendix B.

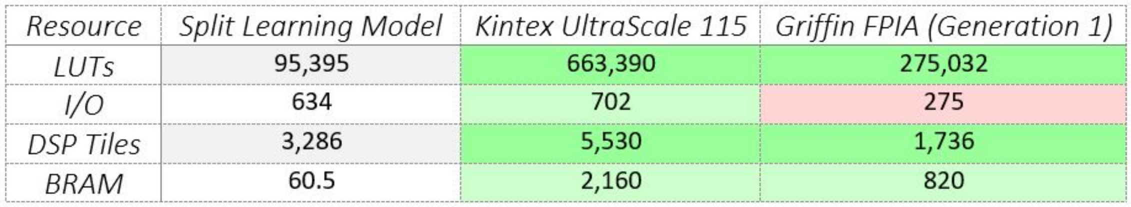
Using the JSON from Appendix B and the corresponding H5 file containing the trained weights of the model, we used the hls4ml package to synthesize the model in HDL code. The operating system used was RHEL 7.8, and the Vivado version used was 2020.1. The target device was Xilinx's Kintex UltraScale 115.

The report on the synthesis of the model, including resource utilization, can be found in Appendix C.

3.5 Results and Analysis

The results of the synthesis show that even before optimizations are conducted, the split learning algorithm can be implemented on a commercial FPGA. The Kintex UltraScale 115, however, has more available resources than a custom FPGA architecture like ReImagine. In Table 3.2, we have compared the utilization of four vital resources (LUTs, DSP tiles, I/O, and BRAM) in the split learning model, the Kintex UltraScale 115, and the first generation Griffin FPIA. It must be noted that resource implementations are individual to each FPGA manufacturer, meaning that Griffin LUTs, DSPs, or BRAMs may not be strictly equivalent to Xilinx Kintex LUTs, DSPs, or BRAMs. The estimates in the table below are therefore approximations.

Table 3.2: Resource utilization of the split learning model compared to device attributes.



Resource	Split Learning Model	Kintex UltraScale 115	Griffin FPIA (Generation 1)
LUTs	95,395	663,390	275,032
I/O	634	702	275
DSP Tiles	3,286	5,530	1,736
BRAM	60.5	2,160	820

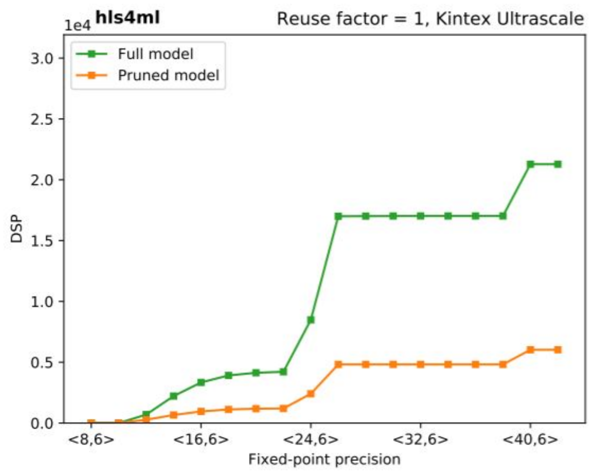
Even without optimizations, the Griffin FPIA contains the volume of resources required to implement the split learning model in each category except I/O. However, this table accounts only for perimeter I/O. In a real-world use-case, an FPGA-based sensor system like the Griffin FPIA would not receive input data from its perimeter I/O, but from its vast 3D inputs where the Tier 3 detector interfaces with Tier 1 of the platform (see Figure 2-2 for reference). This reduces the amount of perimeter I/O needed, as the device would only need to output the weights at its cut layer and (in training) the gradients at the cut layer during back propagation. Image data would not be received across the perimeter, reducing the need for perimeter I/O.

These results demonstrate the technical feasibility of the implementation of split learning models on FPGA-based sensor nodes in the case of not just large commercial FPGAs, but also more resourced-constrained platforms like the Griffin FPIA that are still under development.

A caveat to the demonstrated feasibility is that the MNIST dataset, while useful as a tool for proof of concept, is a relatively easy dataset to train an image classification model on, with ten clear categories and an individual image size of only 28x28. Useful split learning models of the kind discussed in Chapter 4 will likely be trained and evaluated on significantly larger images that are more difficult to create effective models for. These models will require more and larger layers, and utilize more resources. However, the team that created the hls4ml package has demonstrated success in optimizing machine learning models and reducing their resource utilization.

One important technique is the compression of models by pruning, meaning the removal of weights that are shown to have little or no effect on the model's accuracy. In one example provided, pruning reduced the number of model parameters by 69.5% and the number of DSP tiles used by 66.3% with only a 0.13% loss in accuracy [18]. The DSP utilization of this optimization is shown in Figure 3-2.

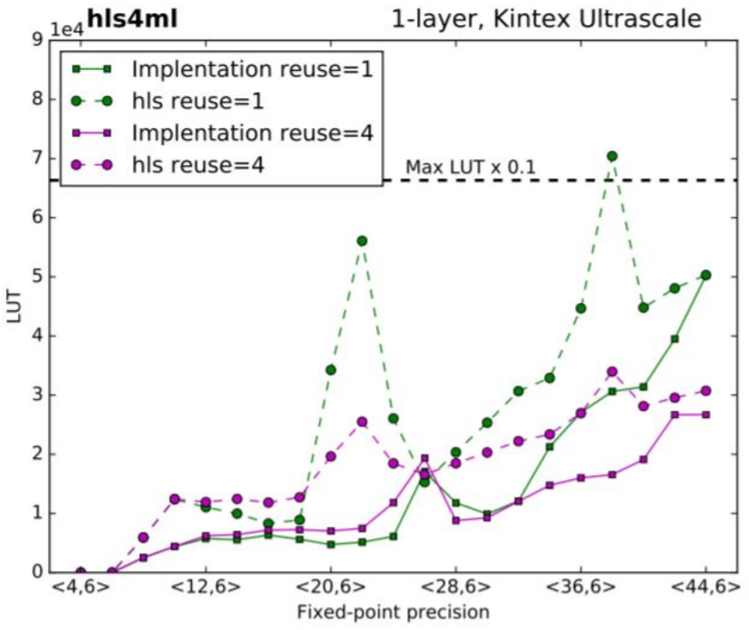
Figure 3-2: DSP utilization in pruned model [18].



Another useful optimization is reuse, where instead of being fully parallelized, the FPGA only processes a fraction of the model at a time, allowing individual multipliers to be reused multiple times. The resulting computation is slower, but requires far fewer LUTs, as Figure 3-3 illustrates. The figure also reflects that synthesis reports generally overestimate the resource utilizations of implementations of HDL designs on physical devices.

These kinds of optimizations will be vital in deploying larger, more complex split learning applications on resource-constrained platforms in the future, and can be implemented on a case-by-case basis dependent on the requirements of the specific pairing of device and model.

Figure 3-3: LUT utilization in reused and implemented ML model [18].



THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

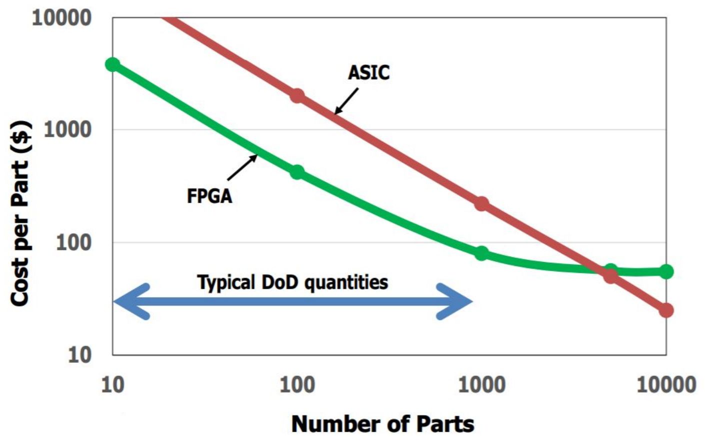
The Future of FPGAs in Sensor Systems

In this chapter, we provide analysis of the ways that the economic and technical viability of developing sensor systems on FPGAs has shaped the current landscape of such systems in order to provide an overview of what types of systems split learning applications might be deployed in. Next, we further analyze the security and privacy requirements of split learning in order to understand what advantages FPGAs might offer those trying to implement privacy-preserving machine learning methods, and where FPGAs still fall short. Finally, we outline a use case in which the implementation of split learning applications on FPGA-based sensor systems might offer significant advantages.

4.1 FPGA Sensor Systems: The Current Landscape

The landscape of FPGA sensor systems is as diverse as the devices themselves are. FPGAs are used in single-sensor systems, distributed systems featuring moderate numbers of nodes, and in large numbers in large-scale datacenters. Each type of system has its own economic and technical motivations for the use of FPGAs. Large-scale datacenters fall outside the scope of this work, which is focused on device counts at the scale shown in Figure 4-1. We will examine the economic and technical motivations for the use of FPGAs and the applicability of split learning methods in more detail in the following section by looking at individual examples of FPGAs in use in unique single-sensor systems and distributed systems featuring moderate numbers of nodes.

Figure 4-1: Visualization of the part quantities at which ASICs become cheaper than FPGAs [27].



4.1.1 Unique Single-Sensor FPGA-Based Systems

One example of an FPGA-based single-sensor system is found in NASA's Soil Moisture Active/Passive (SMAP) mission. SMAP monitors global soil moisture and freeze/thaw cycles in order to help improve weather and climate forecasting, which in turn improves agricultural productivity, improves human health outcomes through flood and famine prediction, and improves national security by enabling better understanding of ground mobility. SMAP contains the Radiometer Digital Electronics (RDE) subsystem, which conducts digital signal processing (DSP) and radio frequency interference (RFI) mitigation [11].

Two of the major technical concerns of the RDE were high temporal resolution, which allows a sufficiently high sampling rates to conduct sub-millisecond RFI detection and mitigation, and low power usage, which allows it to fly and operate on the resource-constrained SMAP satellite mission. Achieving low enough power usage required logic-level optimizations of the kind possible on FPGAs, but not general-purpose processors [10]. These types of concerns—the achievement of high sampling rates and low SWaP without the incurring expensive non-recurring engineering costs that ASIC designs require for single devices—make FPGAs the natural choice for the RDE and similar single-sensor systems, or systems that only use a handful of devices.

These types of systems are generally less subject to privacy concerns, given that they rarely handle personal data that necessitates privacy guarantees. In cases where single-sensor systems are implementing machine learning methods, split learning could be used between the FPGA-based sensor and a central server in order to reduce the resource utilization of the machine learning model on the FPGA by offloading the later layers to a separate server while speeding computation by allowing the FPGA to compute the layers up to the cut layer. However, single-sensor systems are unlikely to be the most fruitful area for the application of split learning techniques given that privacy considerations are generally not a central concern.

4.1.2 Distributed FPGA Sensor Systems (Moderate Device Count)

When describing distributed systems with a device count that could be classified as "moderate," we are describing systems with devices that are manufactured at a scale not large enough to make the use of ASICs a more cost-effective option (so are not used in ubiquitous devices like cell phone cameras). These types of systems may be spread across multiple physical locations and may be operated by one or more independent actors. As can be seen in Figure 4-1, this generally means device counts on the scale of tens, hundreds, or thousands, and includes most DoD quantities.

While not forming a single cohesive system, the set of radiology appliances like CT scanners and MRI units in health care settings throughout the world comprise a distributed set of imaging systems that contain vast amounts of useful data. There were roughly 10,000 CT scanners and 8,000 MRI units in the United States as of 2007 [3], placing them on the upper end of the estimated range of device counts for which FPGAs are advantageous from a cost perspective (some estimates place that range higher). It is a reasonable assumption that any data-sharing collaboration between health care entities would occur on the scale of tens or hundreds of devices. Therefore, we will apply radiology as our use case for moderately-sized distributed sensor systems.

Radiology appliances ideally have a number of attributes in common with the platform developed by the ReImagine program. It is often not possible to make a diagnosis with a single technique, so radiology appliances are multi-modal, combining techniques like CT, MRI, and ultrasound into a single appliance with multiple subsystems. Radiology also benefits from lower latency. In static imaging cases, lower latency allows scans to be taken more quickly and reduces the amount of radiation patients are exposed to. In interventional radiology, real-time imaging is required, and efficient dataflow is paramount [34].

These factors make FPGAs a good fit for many radiology applications, and a number of radiology devices already employ FPGAs, including the imaging systems referenced in Section 2.2.3. A number of FPGA architectures for the capture and reconstruction of radiology images are also described in [34].

Unlike the single-sensor systems that capture unique data described in Section 4.1.1, imaging of people's internal organs and the associated health information that can be derived from it is highly sensitive, and benefits from strong privacy guarantees. Deep learning methods for health, however, can be employed in a variety of useful applications for diagnosis if the barriers to collaboration between entities that may not individually hold adequate data to train models effectively can be lowered. These two factors have made health one of the primary initial research areas for the split learning effort, as described in [46]. The growing prominence of FPGAs in this field, combined with efforts to increase collaboration between entities, make health and other applications on a similar scale a promising early use case of split learning on FPGAs.

4.2 Meeting Security and Privacy Goals with FPGAs

As discussed in Section 2.2.4, in order for split learning implementations to realize the security and privacy guarantees of their algorithm, they must also include guarantees about 1) the information flow of intermediate results in the process and 2) verifiability, or the client or server's ability to prove that it has executed the desired behavior faithfully without revealing the raw data it holds. In this section we analyze the existing ability of FPGAs to provide those guarantees and the areas where improvement is still required.

Split learning methods on all types of hardware are also vulnerable to attacks like data poisoning, which can occur independently of the base hardware. Therefore, it is important to note that this section is not an exhaustive list of vulnerabilities, but focuses on areas where FPGAs differ most from general-purpose processors like CPUs.

4.2.1 Information Flow

The primary threat models related to the information flow throughout the split learning process are related to the administrator, the client, and the server. An administrator like an engineer or analyst may be enabled to act maliciously when granted access to various outputs from the system [24]. This is an important consideration, but not one tied directly to the hardware upon which the split learning model is implemented, so we will focus instead on threats from the client and server.

A malicious client or server is able to tamper with the training process, compromising the training and subsequent evaluation of the model for the rest of the parties in the system. These threats may be mitigated by tools like Secure Multi-Party Computation (MPC) and Trusted Execution Environments (TEEs) [24]. Secure MPC has been implemented previously on FPGAs [49]. This section will focus on TEEs on FPGAs, which are specific to the device being used.

On CPUs, TEEs like the ARM TrustZone can provide secure enclaves that enable the user to establish the confidentiality and integrity of the code running within them and provide attestation about that code's execution. TEEs cannot prevent physical/side channel attacks that both CPUs and FPGAs are vulnerable to, and they have limited memory, requiring the application designer to pick and choose key portions of the code to execute there [24].

TEEs like TrustZone are also present on certain FPGA SoCs, such as the Xilinx Zynq-7010 device. These enclaves offer the same protections on FPGA SoCs as they do on CPUs, and face the same limitations. However, FPGA SoCs are additionally vulnerable to the insertion of malicious IP, either by an engineer or from a third-party library, during the application design phase. Attacks like these, called hardware trojans, have been demonstrated several times with results including denial of service, privilege escalation, leakage of secure information, and the installation of malicious software [9].

Hardware trojan detection techniques like FANCI and VeriTrust, mentioned in Section 2.2.4, have shown partial success in countering these types of attacks. However, these kinds of attacks on the secure enclaves of FPGA SoCs have not been exhaustively studied, and the effectiveness of FPGA TEEs, though promising, remains an open question.

4.2.2 Verifiability

The TEEs discussed above are also important for the verifiability of split learning implementations. Additionally, many modern FPGAs offer not just bitstream encryption, which can help prevent attacks like model theft, but also bitstream authentication. Bitstream authentication can verify that an FPGA's configuration has not been changed from a trusted initial configuration [17].

Overall, the security and privacy capabilities of modern FPGAs are fairly robust. However, they are not as thoroughly explored as the equivalent capabilities on CPUs, and are sometimes vulnerable to additional types of attacks. Moreover, many security capabilities are present on high-end FPGA SoCs, but are unlikely to be available on particularly small or custom-designed FPGAs like the Griffin chip. In the future, this may change as projects like HETEE (Heterogeneous TEE), which are intended for heterogeneous computing environments and do not require specific hardware to be embedded into the SoC, emerge [55].

However, as these tools are still in their infancy, split learning developers wishing to implement their models on FPGAs, or FPGA developers looking to leverage the privacy protections of split learning, should integrate device security into their considerations from the outset. In implementing split learning on FPGAs, it will be important to evaluate the factors discussed in Section 4.3 and seek creative solutions to potential threats rather than relying on bitstream encryption, differential privacy techniques that distance intermediate representations from their raw data, and other inbuilt elements to fully guarantee the privacy and security of data and model.

4.3 Use Case for Split Learning on FPGAs

As described in Section 4.1, one highly promising use case for implementations of split learning models on FPGAs is in health, particularly radiology applications that can be built on existing FPGA-based imaging sensors and used to enable health entities to collaboratively develop models that will allow for more accurate diagnoses.

When considering the implementation of a split learning network on FPGAs in a healthcare environment, the developer may wish to take into account the following factors:

- The number of clients. The precise number of clients may be flexible, but an estimate of how many health care entities will participate and the number of devices per client can help the developer determine whether split learning is the most communication- and computation-efficient approach. It can also help determine whether the FPGA is the most appropriate hardware to use.

- The size of the input data and the number of parameters in the model. Like the number of clients, this can help determine the communication and computation efficiency of split learning as compared to other distributed deep learning methods.

- The overall size of the model. This can be quickly estimated through a single-device simulation using an HLS tool like hls4ml, and can help quickly determine whether optimization and implementation are worth pursuing on the target device.

- The importance of security guarantees. If a Trusted Execution Environment is deemed necessary to the security of the implementation, the developer will have to carefully select or create entirely new FPGA-based sensor systems that currently support TEEs like ARM TrustZone.

- The importance of privacy guarantees. Given sensitive health data, intermediate representations of raw data from health care entities should not be shared without a measure of differential privacy imposed on the data.

- The required latency. In situations like interventional radiology, FPGAs may be required instead of devices like CPUs to meet timing constraints.

- SWaP-C. An important consideration that depends on the resources of the health entities participating in the collaboration.

Additional areas where split learning's distribution of the burden of calculation (rather than preservation of privacy) might appeal to those looking to implement machine learning methods include particle physics and ultra-fast astronomy. Both fields currently leverage the extremely low latency of FPGAs to detect and record phenomena occurring on the nanosecond scale—in particle physics at the Large Hadron Collider [18], and in ultra-fast astronomy at the Assy-Turgen Astrophysical Observatory in Kazakhstan [28]. At that timescale, "triggers" are used that do simple, very fast analyses of incoming data to determine which frames should be saved for later analysis as they are exfiltrated off the device [18]. A split learning-type method might be used to train a model where the very early layers are shared by the simple trigger task and a more complex analysis task before the complex task forks over to a different device.

Finally, as the ReImagine program is sponsored by DARPA, there could be a number of defense applications for a reconfigurable imaging platform that could conduct complex image recognition tasks. Defense applications may have common ground with health applications, in terms of number of devices, and with particle physics or ultra-fast astronomy applications, in terms of the low latency required for an application such as identifying a missile launch and calculating its trajectory. Defense applications are also unlikely to require strong privacy guarantees for their raw data. However, they may have stricter SWaP requirements, as they may, for example, be deployed remotely on UAVs and be required to persist for a long time without recharging or a change of battery. At the same time, they are more likely to face malicious external actors, and so may require even stronger security guarantees.

It is this ability to delicately balance important factors against one another, across all domains, that makes FPGA-based sensor systems appealing for split learning implementations.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Summary and Conclusion

This work has explored the potential application of split learning and other distributed deep learning methods on FPGA-based sensor systems like ReImagine. In exploring this area, we sought to determine whether the implementation of split learning applications on FPGAs offers advantages to groups seeking to deploy privacy-preserving machine learning methods, as well as to groups seeking to add additional capabilities to FPGA-based sensor systems. In order to determine what advantages, if any, this approach might offer, we sought to answer three questions: Is it technically feasible to implement split learning applications on FPGAs? How have developments in FPGA technology changed the landscape and potential of FPGAs as platforms for sensor systems and for new machine learning applications? And, what current and future use cases exist for the implementation of split learning on FPGAs?

In determining the answers to these questions, we provided an overview of the technologies relevant to the implementation of split learning on FPGAs. We then synthesized a client-side split learning model for an FPGA. Next we conducted an analysis of the current use of FPGAs in sensor systems and the security and privacy capabilities thereof. Finally, we offered a number of potential use cases for the implementation of split learning on FPGAs.

Our survey found that there are a number of existing implementations of machine learning algorithms on FPGAs, and our successful synthesis of such a split learning model in HDL code showed that an implementation of a simple split learning algorithm is achievable on modern, commercial FPGAs, including relatively resource-constrained FPGA-based platforms like the Griffin chip being developed under the ReImagine program. This implementation was possible using tools that are accessible to developers operating without a granular knowledge of the hardware that these algorithms may be deployed on. Therefore, we affirm that the implementation of split learning algorithms on FPGAs is technically feasible. However, much progress is still required before these implementations achieve the accessibility of such tools for general-purpose processors.

Our analysis of developments in FPGA technology included the increased resource density that has led to performance improvements, the specialized logic blocks that have made certain operations more efficient and made a number of security and privacy capabilities native to FPGAs, and the high-level synthesis tools that have enabled developers to rapidly develop, test, and deploy increasingly complex applications. These developments have increasingly led to FPGAs taking the place of processors like CPUs and GPUs in settings where minimizing power usage and latency is of the utmost importance, and the place of ASICs in settings where long development cycles and high non-recurring engineering costs make the use of ASICs prohibitively expensive in terms of money and time. This has led to the use of FPGAs as processing elements in sensor nodes and motivated the development of projects like ReImagine. We have determined that in combination, these developments make FPGAs a promising future platform for sensor systems and machine learning applications, and that the future of distributed deep learning on sensor systems will include the deployment of such applications on FPGAs as well as more traditional platforms for neural networks.

Finally, we examined the landscape of FPGA-based sensor systems to identify use cases for the implementation of split learning on FPGAs. We found that the implementation of split learning on FPGAs is applicable to health-oriented use cases, particularly in radiology, due to the current use of FPGAs in radiological appliances and the importance of the preservation of privacy of raw data in healthcare settings, where a number of machine learning methods are already being developed collaboratively. We also identified future potential for use cases in particle physics, ultra-fast astronomy, and defense.

Overall, this work concludes that the implementation of split learning algorithms on FPGAs is a promising area for both groups seeking to deploy privacy-preserving machine learning methods and groups seeking to add additional capabilities to FPGA-based sensor systems. The expansion of split learning implementations to FPGAs allows developers of split learning algorithms to leverage data found on devices where traditional machine learning platforms like GPUs simply cannot meet latency, power, or other system requirements. Meanwhile, the use of split learning, or in certain cases other distributed deep learning methods, allows groups that have already deployed FPGA-based sensor systems to take advantage of the reconfigurable nature of the platform and add machine learning capabilities, even in cases where an individual device does not have enough data to effectively train a model, the data being collected is sensitive, or a complex machine learning model might be too large for the constraints of the device.

To make the deployment of split learning implementations on FPGAs a reality, rather than a theoretical possibility, there is still more work that must be done. On the technical side, future work in this area should include the loading and verification of split learning models onto physical FPGAs. This will also allow better benchmarking of resource utilization, allowing developers to understand the complexity of the models they will be able to implement. HLS tools for machine learning on FPGAs must continue to expand their capabilities and interfaces, making them more accessible to all developers, even those without extensive hardware experience. These HLS tools should also be a consideration in the development of programming toolsets specific to programs like ReImagine. Finally, we must demonstrate the training of split learning models on FPGAs, rather than simply demonstrating inference, at the same time analyzing the bandwidth limitations of such implementations, which might be significant depending on the dimensions of the cut layer.

Beyond these considerations, programs like the MIT Alliance for Distributed and Private Machine Learning should seek to collaborate with groups that deploy platforms not traditionally used for machine learning applications, but that nonetheless process valuable and sensitive data and can support split learning implementations. At the same time, groups that have deployed FPGA-based sensor systems should understand that distributed deep learning applications are among the tools available for use on their highly reconfigurable platforms, and evaluate their own privacy and resource needs to determine whether they can benefit from the advantages that split learning offers.

Overall, machine learning techniques on FPGAs are rapidly expanding, as is the use of FPGAs in sensor systems. The profusion of data on these platforms raises concerns about the privacy of the raw data that is shared across devices, but also creates opportunities for the development of new machine learning techniques that can result in better outcomes across a variety of domains. Throughout our survey of machine learning on FPGAs, it became clear that there is no existing paradigm that has dominated the deployment of machine learning methods on FPGAs. This presents a significant opportunity for the deployment of split learning methods on FPGAs: consideration of privacy and efficiency issues that are sometimes afterthoughts for researchers seeking to train accurate models, placed at the heart of the development of a new set of tools. Together, these challenges and opportunities make split learning on FPGAs a rich area worthy of future study.

Appendix A

Generating Split Learning Keras Model

```
# -*- coding: utf-8 -*-
"""split_learning_hls4ml.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1
   X0mSVxTAdJ1wAUL1Ctw0a_bBNsIfww1C

referenced https://github.com/transcranial/keras-js/blob/master/
   notebooks/demos/mnist_cnn.ipynb, Sept 01 2020
"""

KERAS_MODEL_FILEPATH = '../../demos/data/mnist_cnn/mnist_cnn.h5'

import numpy as np
np.random.seed(1337)  # for reproducibility

from keras.datasets import mnist
from keras.models import Sequential, Model
from keras.layers import Dense, Flatten, Activation, Input
from keras.layers import Conv2D, MaxPooling2D
from keras.utils import np_utils
from keras.callbacks import EarlyStopping, ModelCheckpoint

num_classes = 10

# input image dimensions
img_rows, img_cols = 28, 28

# the data, shuffled and split between train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# convert class vectors to binary class matrices
y_train = np_utils.to_categorical(y_train, num_classes)
y_test = np_utils.to_categorical(y_test, num_classes)


'''
referenced https://stackoverflow.com/a/48612403/10728413, Sept 01
    2020 for example of composing two sequential keras models
'''
alice_seq_split_small = Sequential()
alice_seq_split_small.add(Conv2D(16, (3, 3), input_shape=(
    input_shape)))
```

```
alice_seq_split_small.add(Activation('relu'))
alice_seq_split_small.add(MaxPooling2D(pool_size=(2, 2)))
alice_seq_split_small.add(Conv2D(32, (3, 3)))
alice_seq_split_small.add(Activation('relu'))
alice_seq_split_small.add(MaxPooling2D(pool_size=(2, 2)))

server_seq_split_small = Sequential()
server_seq_split_small.add(Flatten())
server_seq_split_small.add(Dense(256))
server_seq_split_small.add(Activation('relu'))
server_seq_split_small.add(Dense(10))
server_seq_split_small.add(Activation('softmax'))

seq_model_split_small = Sequential()
seq_model_split_small.add(alice_seq_split_small)
seq_model_split_small.add(server_seq_split_small)

seq_model_split_small.compile(loss='categorical_crossentropy',
    optimizer='adam', metrics=['accuracy'])

alice_seq_split_small.summary()
server_seq_split_small.summary()
seq_model_split_small.summary()

# Train
batch_size = 128
epochs = 10
seq_model_split_small.fit(x_train, y_train, batch_size=batch_size,
    epochs=epochs, verbose=2,
          validation_data=(x_test, y_test))
score = seq_model_split_small.evaluate(x_test, y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

from google.colab import files

```

```
json_config_a = alice_seq_split_small.to_json()
alice_seq_split_small.save_weights("alice_seq_split_small.h5")
with open("alice_seq_split_small.json", 'w') as f:
    f.write(json_config_a)
files.download('alice_seq_split_small.h5')
files.download('alice_seq_split_small.json')
```

Appendix B

Client-Side Split Learning Model Architecture

```
{
  "class_name": "Sequential",
  "config": {
    "name": "sequential",
    "layers": [

      {
        "class_name": "InputLayer",
        "config": {
          "batch_input_shape": [ null, 28, 28, 1 ],
          "dtype": "float32",
          "sparse": false,
          "ragged": false,
          "name": "conv2d_input"
        }
      },
      {
        "class_name": "Conv2D",
        "config": {
          "name": "conv2d",
          "trainable": true,
        "batch_input_shape": [ null, 28, 28, 1 ],
        "dtype": "float32",
        "filters": 16,
        "kernel_size": [ 3, 3 ],
        "strides": [ 1, 1 ],
        "padding": "valid",
        "data_format": "channels_last",
        "dilation_rate": [ 1, 1 ],
        "groups": 1,
        "activation": "linear",
        "use_bias": true,
        "kernel_initializer": {
          "class_name": "GlorotUniform",
          "config": { "seed": null }
        },
        "bias_initializer": {
          "class_name": "Zeros",
          "config": {}
        },
        "kernel_regularizer": null,
        "bias_regularizer": null,
        "activity_regularizer": null,
        "kernel_constraint": null,
        "bias_constraint": null
      }
    },
    {
      "class_name": "Activation",
      "config": {
        "name": "activation",
        "trainable": true,
        "dtype": "float32",
        "activation": "relu"
      }
    },
    {
```

```json
        "class_name": "MaxPooling2D",
        "config": {
          "name": "max_pooling2d",
          "trainable": true,
          "dtype": "float32",
          "pool_size": [ 2, 2 ],
          "padding": "valid",
          "strides": [ 2, 2 ],
          "data_format": "channels_last"
        }
      },
      {
        "class_name": "Conv2D",
        "config": {
          "name": "conv2d_1",
          "trainable": true,
          "dtype": "float32",
          "filters": 32,
          "kernel_size": [ 3, 3 ],
          "strides": [ 1, 1 ],
          "padding": "valid",
          "data_format": "channels_last",
          "dilation_rate": [ 1, 1 ],
          "groups": 1,
          "activation": "linear",
          "use_bias": true,
          "kernel_initializer": {
            "class_name": "GlorotUniform",
            "config": { "seed": null }
          },
          "bias_initializer": {
            "class_name": "Zeros",
            "config": {}
          },
          "kernel_regularizer": null,
          "bias_regularizer": null,
          "activity_regularizer": null,
          "kernel_constraint": null,
          "bias_constraint": null
        }
      },
      {
        "class_name": "Activation",
        "config": {
          "name": "activation_1",
          "trainable": true,
          "dtype": "float32",
          "activation": "relu"
        }
      },
      {
        "class_name": "MaxPooling2D",
        "config": {
          "name": "max_pooling2d_1",
          "trainable": true,
          "dtype": "float32",
          "pool_size": [ 2, 2 ],
          "padding": "valid",
          "strides": [ 2, 2 ],
          "data_format": "channels_last"
        }
      }
    ]
  },
  "keras_version": "2.4.0",
  "backend": "tensorflow"
}
```

Appendix C

Vivado HLS Synthesis Report

This report was generated by the version of the hls4ml tool found at the repository https://github.com/vloncar/hls4ml.git,
commit 0613c5e635d8e39c23177f18092a6006800171c7.

```
Copyright 1986-2020 Xilinx, Inc. All Rights Reserved.
--------------------------------------------------------------------------
| Tool Version : Vivado v.2020.1 (lin64) Build 2902540 Wed May 27
    19:54:35 MDT 2020
| Date         : Mon Aug 31 17:56:21 2020
| Host         : llcad-grid02.llan.ll.mit.edu running 64-bit Red Hat
    Enterprise Linux Workstation release 7.8 (Maipo)
| Command      : report_utilization -file vivado_synth.rpt
| Design       : myproject
| Device       : xcku115flvb2104-2
| Design State : Synthesized
--------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. CLB Logic
```

```
1.1 Summary of Registers by Type
2. BLOCKRAM
3. ARITHMETIC
4. I/O
5. CLOCK
6. ADVANCED
7. CONFIGURATION
8. Primitives
9. Black Boxes
10. Instantiated Netlists
11. SLR Connectivity
12. SLR Connectivity Matrix
13. SLR CLB Logic and Dedicated Block Utilization
14. SLR IO Utilization

1. CLB Logic
------------

+----------------------------+-------+-------+-----------+-------+
|          Site Type         |  Used | Fixed | Available | Util% |
+----------------------------+-------+-------+-----------+-------+
| CLB LUTs*                  | 95395 |     0 |    663360 | 14.38 |
|   LUT as Logic             | 87715 |     0 |    663360 | 13.22 |
|   LUT as Memory            |  7680 |     0 |    293760 |  2.61 |
|     LUT as Distributed RAM |     0 |     0 |           |       |
|     LUT as Shift Register  |  7680 |     0 |           |       |
| CLB Registers              | 40891 |     0 |   1326720 |  3.08 |
|   Register as Flip Flop    | 40891 |     0 |   1326720 |  3.08 |
|   Register as Latch        |     0 |     0 |   1326720 |  0.00 |
| CARRY8                     | 10564 |     0 |     82920 | 12.74 |
| F7 Muxes                   |  2304 |     0 |    331680 |  0.69 |
| F8 Muxes                   |     0 |     0 |    165840 |  0.00 |
| F9 Muxes                   |     0 |     0 |     82920 |  0.00 |
+----------------------------+-------+-------+-----------+-------+
* Warning! The Final LUT count, after physical optimizations and
   full implementation, is typically lower. Run opt_design after
   synthesis, if not already completed, for a more realistic count.


1.1 Summary of Registers by Type
--------------------------------

+-------+--------------+-------------+--------------+
| Total | Clock Enable | Synchronous | Asynchronous |
+-------+--------------+-------------+--------------+
| 0     |            _ |           - |            - |
| 0     |            _ |           - |          Set |
| 0     |            _ |           - |        Reset |
| 0     |            _ |         Set |            - |
| 0     |            _ |       Reset |            - |
| 0     |          Yes |           - |            - |
| 0     |          Yes |           - |          Set |
| 0     |          Yes |           - |        Reset |
| 2169  |          Yes |         Set |            - |
| 38722 |          Yes |       Reset |            - |
+-------+--------------+-------------+--------------+


2. BLOCKRAM
-----------

+-------------------+------+-------+-----------+-------+
|     Site Type     | Used | Fixed | Available | Util% |
+-------------------+------+-------+-----------+-------+
| Block RAM Tile    | 60.5 |     0 |      2160 |  2.80 |
|   RAMB36/FIFO*    |    0 |     0 |      2160 |  0.00 |
|   RAMB18          |  121 |     0 |      4320 |  2.80 |
|     RAMB18E2 only |  121 |       |           |       |
+-------------------+------+-------+-----------+-------+
* Note: Each Block RAM Tile only has one FIFO logic available and
   therefore can accommodate only one FIFO36E2 or one FIFO18E2.
   However, if a FIFO18E2 occupies a Block RAM Tile, that tile can
```

```
    still accommodate a RAMB18E2


3. ARITHMETIC
-------------

+----------------+------+-------+-----------+-------+
|    Site Type   | Used | Fixed | Available | Util% |
+----------------+------+-------+-----------+-------+
| DSPs           | 3286 |     0 |      5520 | 59.53 |
|   DSP48E2 only | 3286 |       |           |       |
+----------------+------+-------+-----------+-------+


4. I/O
------

+------------+------+-------+-----------+-------+
|  Site Type | Used | Fixed | Available | Util% |
+------------+------+-------+-----------+-------+
| Bonded IOB |  634 |     0 |       702 | 90.31 |
+------------+------+-------+-----------+-------+


5. CLOCK
--------

+----------------------+------+-------+-----------+-------+
|       Site Type      | Used | Fixed | Available | Util% |
+----------------------+------+-------+-----------+-------+
| GLOBAL CLOCK BUFFERs |    1 |     0 |      1248 |  0.08 |
|   BUFGCE             |    1 |     0 |       576 |  0.17 |
|   BUFGCE_DIV         |    0 |     0 |        96 |  0.00 |
|   BUFG_GT            |    0 |     0 |       384 |  0.00 |
|   BUFGCTRL*          |    0 |     0 |       192 |  0.00 |
| PLLE3_ADV            |    0 |     0 |        48 |  0.00 |
| MMCME3_ADV            |    0 |     0 |        24 |  0.00 |
+----------------------+------+-------+-----------+-------+
* Note: Each used BUFGCTRL counts as two GLOBAL CLOCK BUFFERs. This
  table does not include global clocking resources, only buffer
  cell usage. See the Clock Utilization Report (
  report_clock_utilization) for detailed accounting of global
  clocking resource availability.


6. ADVANCED
-----------

+-----------------+------+-------+-----------+-------+
|    Site Type    | Used | Fixed | Available | Util% |
+-----------------+------+-------+-----------+-------+
| GTHE3_CHANNEL   |    0 |     0 |        64 |  0.00 |
| GTHE3_COMMON    |    0 |     0 |        16 |  0.00 |
| IBUFDS_GTE3     |    0 |     0 |        32 |  0.00 |
| OBUFDS_GTE3     |    0 |     0 |        32 |  0.00 |
| OBUFDS_GTE3_ADV |    0 |     0 |        32 |  0.00 |
| PCIE_3_1        |    0 |     0 |         6 |  0.00 |
| SYSMONE1        |    0 |     0 |         2 |  0.00 |
+-----------------+------+-------+-----------+-------+


7. CONFIGURATION
----------------

+-------------+------+-------+-----------+-------+
|  Site Type  | Used | Fixed | Available | Util% |
+-------------+------+-------+-----------+-------+
| BSCANE2     |    0 |     0 |         8 |  0.00 |
| DNA_PORTE2  |    0 |     0 |         2 |  0.00 |
| EFUSE_USR   |    0 |     0 |         2 |  0.00 |
| FRAME_ECCE3 |    0 |     0 |         2 |  0.00 |
| ICAPE3      |    0 |     0 |         4 |  0.00 |
| MASTER_JTAG |    0 |     0 |         2 |  0.00 |
| STARTUPE3   |    0 |     0 |         2 |  0.00 |
+-------------+------+-------+-----------+-------+


8. Primitives
-------------

+----------+-------+---------------------+
| Ref Name |  Used | Functional Category |
+----------+-------+---------------------+
| LUT2     | 48748 |                 CLB |
| FDRE     | 38722 |            Register |
| LUT4     | 25300 |                 CLB |
| LUT3     | 25133 |                 CLB |
| CARRY8   | 10564 |                 CLB |
| LUT6     |  7228 |                 CLB |
| LUT5     |  5999 |                 CLB |
| SRLC32E  |  5632 |                 CLB |
| LUT1     |  4660 |                 CLB |
| DSP48E2  |  3286 |          Arithmetic |
| MUXF7    |  2304 |                 CLB |
| FDSE     |  2169 |            Register |
| SRL16E   |  2048 |                 CLB |
| OBUF     |   582 |                 I/O |
| RAMB18E2 |   121 |           Block Ram |
| INBUF    |    52 |                 I/O |
| IBUFCTRL |    52 |              Others |
| BUFGCE   |     1 |               Clock |
+----------+-------+---------------------+


9. Black Boxes
--------------

+----------+------+
| Ref Name | Used |
+----------+------+


10. Instantiated Netlists
-------------------------

+----------+------+
| Ref Name | Used |
+----------+------+


11. SLR Connectivity
--------------------

+--------------------------------+------+-------+-----------+-------+
|                                | Used | Fixed | Available | Util% |
+--------------------------------+------+-------+-----------+-------+
|SLR1 <-> SLR0                   |    0 |       |     17280 |  0.00 |
|  SLR0 -> SLR1                  |    0 |       |           |  0.00 |
|   Using TX_REG only            |    0 |     0 |           |       |
|   Using RX_REG only            |    0 |     0 |           |       |
|   Using Both TX_REG and RX_REG|    0 |     0 |           |       |
|  SLR1 -> SLR0                  |    0 |       |           |  0.00 |
|   Using TX_REG only            |    0 |     0 |           |       |
|   Using RX_REG only            |    0 |     0 |           |       |
|   Using Both TX_REG and RX_REG|    0 |     0 |           |       |
+--------------------------------+------+-------+-----------+-------+
|Total SLLs Used                 |    0 |       |           |       |
+--------------------------------+------+-------+-----------+-------+


12. SLR Connectivity Matrix
---------------------------

+-----------+------+------+
| FROM \ TO | SLR1 | SLR0 |
+-----------+------+------+
| SLR1      |    0 |    0 |
| SLR0      |    0 |    0 |
+-----------+------+------+


13. SLR CLB Logic and Dedicated Block Utilization
-------------------------------------------------

+----------------------------+------+------+--------+--------+
|          Site Type         | SLR0 | SLR1 | SLR0 % | SLR1 % |
+----------------------------+------+------+--------+--------+
| CLB                        |    0 |    0 |   0.00 |   0.00 |
|   CLBL                     |    0 |    0 |   0.00 |   0.00 |
|   CLBM                     |    0 |    0 |   0.00 |   0.00 |
| CLB LUTs                   |    0 |    0 |   0.00 |   0.00 |
|   LUT as Logic             |    0 |    0 |   0.00 |   0.00 |
|   LUT as Memory            |    0 |    0 |   0.00 |   0.00 |
|     LUT as Distributed RAM |    0 |    0 |   0.00 |   0.00 |
|     LUT as Shift Register  |    0 |    0 |   0.00 |   0.00 |
| CLB Registers              |    0 |    0 |   0.00 |   0.00 |
| CARRY8                     |    0 |    0 |   0.00 |   0.00 |
| F7 Muxes                   |    0 |    0 |   0.00 |   0.00 |
| F8 Muxes                   |    0 |    0 |   0.00 |   0.00 |
| F9 Muxes                   |    0 |    0 |   0.00 |   0.00 |
| Block RAM Tile             |    0 |    0 |   0.00 |   0.00 |
|   RAMB36/FIFO              |    0 |    0 |   0.00 |   0.00 |
|   RAMB18                   |    0 |    0 |   0.00 |   0.00 |
| URAM                       |    0 |    0 |   0.00 |   0.00 |
| DSPs                       |    0 |    0 |   0.00 |   0.00 |
| PLL                        |    0 |    0 |   0.00 |   0.00 |
| MMCM                       |    0 |    0 |   0.00 |   0.00 |
| Unique Control Sets        |    0 |    0 |   0.00 |   0.00 |
+----------------------------+------+------+--------+--------+
* Note: Available Control Sets based on CLB Registers / 8
```

```


14. SLR IO Utilization
----------------------

+-----------+-----------+---------+-----------+----------+
| SLR Index | Used IOBs | (%)IOBs | Used IPADs | (%)IPADs |
+-----------+-----------+---------+-----------+----------+
| SLR1      |         0 |    0.00 |          0 |     0.00 |
| SLR0      |         0 |    0.00 |          0 |     0.00 |
+-----------+-----------+---------+-----------+----------+
| Total     |         0 |         |          0 |          |
+-----------+-----------+---------+-----------+----------+

+-----------+------------+----------+-----+
| SLR Index | Used OPADs | (%)OPADs | GTs |
+-----------+------------+----------+-----+
| SLR1      |          0 |     0.00 |   0 |
| SLR0      |          0 |     0.00 |   0 |
+-----------+------------+----------+-----+
| Total     |          0 |     0.00 |   0 |
+-----------+------------+----------+-----+
```

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] Distributed deep learning and inference without sharing raw data. https://splitlearning.github.io/.

[2] Project Overview ‹ Split Learning: Distributed and collaborative learning. https://www.media.mit.edu/projects/distributed-learning-and-collaborative-learning-1/overview/.

[3] Table 120. Number of magnetic resonance imaging (MRI) units and computed tomography (CT) scanners: Selected countries, selected years 1990–2007. Technical report, Centers for Disease Control, National Center for Health Statistics., 2010. https://www.cdc.gov/nchs/data/hus/2010/120.pdf.

[4] Introduction to field programmable gate arrays (FPGA), August 2017. https://microcontrollerslab.com/fpga-introduction-block-diagram/.

[5] A Camera That Can See Unlike Any Imager Before It . *Defense Advanced Research Projects Agency News And Events*, September 2019. https://www.darpa.mil/news-events/2016-09-16.

[6] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Serot, and François Berry. Accelerating CNN inference on FPGAs: A Survey. May 2018. arXiv: 1806.01683.

[7] David F. Bacon, Rodric Rabbah, and Sunil Shukla. FPGA programming for the masses. *Communications of the ACM*, 56(4):56–63, April 2013.

[8] Kristen Baldwin. DoD Electronics Priorities. In *NDIA Electronics Division Kickoff Meeting*, January 2018. https://www.ndia.org/-/media/sites/ndia/divisions/electronics/past-proceedings/ndia-ed-baldwin-18jan2018-vf.ashx?la=en.

[9] E. M. Benhani, L. Bossuet, and A. Aubert. The Security of ARM TrustZone in a FPGA-Based SoC. *IEEE Transactions on Computers*, 68(8):1238–1248, August 2019.

[10] D. Bradley, C. Brambora, A. Feizi, R. Garcia, L. Miles, P. Mohammed, J. Peng, J. Piepmeier, K. Shakoorzadeh, and M. Wong. Preliminary results from the soil moisture active/passive (smap) radiometer digital electronics engineering test unit (etu). In *2012 IEEE International Geoscience and Remote Sensing Symposium*, pages 1077–1080, 2012.

[11] Molly E. Brown, Vanessa Escobar, Susan Moran, Dara Entekhabi, Peggy E. O'Neill, Eni G. Njoku, Brad Doorn, and Jared K. Entin. NASA's Soil Moisture Active Passive (SMAP) Mission and Opportunities for Applications Users. *Bulletin of the American Meteorological Society*, 94(8):1125–1128, August 2013.

[12] Damian Cacko, Mateusz Walczak, and Marcin Lewandowski. Low-Power Ultrasound Imaging on Mixed FPGA/GPU Systems. In *2018 Joint Conference - Acoustics*, pages 1–6, September 2018.

[13] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting Distributed Synchronous SGD. 2016. https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/45187.pdf.

[14] C. E. Cox and W. E. Blanz. Ganglion-a fast field-programmable gate array implementation of a connectionist classifier. *IEEE Journal of Solid-State Circuits*, 27(3):288–299, 1992.

[15] Antonio de la Piedra, An Braeken, and Abdellah Touhafi. Sensor Systems Based on FPGAs and Their Applications: A Survey. *Sensors*, 12(9):12235–12264, September 2012.

[16] Roberto DiCecco, Griffin Lacey, Jasmina Vasiljevic, Paul Chow, Graham Taylor, and Shawki Areibi. Caffeinated FPGAs: FPGA Framework For Convolutional Neural Networks. September 2016. arXiv: 1609.09671.

[17] Saar Drimer. Authentication of FPGA Bitstreams: Why and How. In Pedro C. Diniz, Eduardo Marques, Koen Bertels, Marcio Merino Fernandes, and João M. P. Cardoso, editors, *Reconfigurable Computing: Architectures, Tools and Applications*, volume 4419, pages 73–84. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[18] Javier Duarte, Song Han, Philip Harris, Sergo Jindariani, Edward Kreinar, Benjamin Kreis, Jennifer Ngadiuba, Maurizio Pierini, Ryan Rivera, Nhan Tran, and Zhenbin Wu. Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation*, 13(07), July 2018. arXiv: 1804.06913.

[19] M. E. S. Elrabaa, M. Al-Asli, and M. Abu-Amara. Secure Computing Enclaves Using FPGAs. *IEEE Transactions on Dependable and Secure Computing*, August 2019.

[20] Gabriel García, Carlos Jara, Jorge Pomares, Aiman Alabdo, Lucas Poggi, and Fernando Torres. A Survey on FPGA-Based Sensor Systems: Towards Intelligent and Reconfigurable Low-Power Sensors for Computer Vision, Control and Signal Processing. *Sensors*, 14(4):6247–6278, March 2014.

[21] Peter J. Grossman, Matthew Stamplis, Kate Thurmer, Bryan Tyrell, and John Frechette. Methods for enabling in-field selection of near-sensor digital imaging functions, April 2019. US Patent Application Publicaton No. US 2019/0104269 A1.

[22] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William J. Dally. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. February 2017. arXiv: 1612.00694 version: 2.

[23] Intel. Efficient Implementation of Neural Network Systems Built on FPGAs, and Programmed with OpenCL. 2016. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/solution-sheets/efficient_neural_networks.pdf.

[24] Peter Kairouz, H. Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Keith Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, Rafael G. L. D'Oliveira, Salim El Rouayheb, David Evans, Josh Gardner, Zachary Garrett, Adrià Gascón, Badih Ghazi, Phillip B. Gibbons, Marco Gruteser, Zaid Harchaoui, Chaoyang He, Lie He, Zhouyuan Huo, Ben Hutchinson, Justin Hsu, Martin Jaggi, Tara Javidi, Gauri Joshi, Mikhail Khodak, Jakub Konečný, Aleksandra Korolova, Farinaz Koushanfar, Sanmi Koyejo, Tancrède Lepoint, Yang Liu, Prateek Mittal, Mehryar Mohri, Richard Nock, Ayfer Özgür, Rasmus Pagh, Mariana Raykova, Hang Qi, Daniel Ramage, Ramesh Raskar, Dawn Song, Weikang Song, Sebastian U. Stich, Ziteng Sun, Ananda Theertha Suresh, Florian Tramèr, Praneeth Vepakomma, Jianyu Wang, Li Xiong, Zheng Xu, Qiang Yang, Felix X. Yu, Han Yu, and Sen Zhao. Advances and Open Problems in Federated Learning. December 2019. arXiv: 1912.04977.

[25] Navdeep Kumar, Nirmal Kaur, and Deepti Gupta. Major Convolutional Neural Networks in Image Classification: A Survey. In Maitreyee Dutta, C. Rama Krishna, Rakesh Kumar, and Mala Kalra, editors, *Proceedings of International Conference on IoT Inclusive Life (ICIIL 2019), NITTTR Chandigarh, India*, Lecture Notes in Networks and Systems, pages 243–258, Singapore, 2020. Springer.

[26] Griffin Lacey, Graham W. Taylor, and Shawki Areibi. Deep Learning on FPGAs: Past, Present, and Future. February 2016. arXiv: 1602.04283.

[27] Jay Lewis. Reconfigurable Imaging (ReImagine). ReImagine Proposers Day, September 2016. https://www.darpa.mil/attachments/Final_Compiled_ReImagineProposersDay.pdf.

[28] Siyang Li, George F. Smoot, Bruce Grossan, Albert Wai Kit Lau, Marzhan Bekbalanova, Mehdi Shafiee, and Thorsten Stezelberger. Program objectives and specifications for the Ultra-Fast Astronomy observatory. *AOPC 2019: Space Optics, Telescopes, and Instrumentation*, December 2019. arXiv: 1908.10549.

[29] Rui Liu, Daiyin Zhu, Die Wang, and Wanwan Du. FPGA Implementation of SAR Imaging Processing System. In *2019 6th Asia-Pacific Conference on Synthetic Aperture Radar (APSAR)*, pages 1–5, November 2019.

[30] Imperial College London. fpgaConvNet: A framework for mapping Convolutional Neural Networks on FPGAs. 2017. http://cas.ee.ic.ac.uk/people/sv1310/fpgaConvNet.html.

[31] LSST Project Summary. Project summary, Rubin Observatory, December 2019. https://docushare.lsst.org/docushare/dsweb/Get/Document-13936.

[32] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *JMLR: W&CP*, number 54 in Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS), February 2017.

[33] Mickael Njiki, Abdelhafid Elouardi, Samir Bouaziz, Olivier Casula, and Olivier Roy. A multi-FPGA architecture-based real-time TFM ultrasound imaging. *Journal of Real-Time Image Processing*, 16(2):505–521, April 2019.

[34] Daniele Passaretti, Jan Moritz Joseph, and Thilo Pionteck. Survey on FPGAs in Medical Radiology Applications: Challenges, Architectures and Programming Models. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 279–282, December 2019.

[35] Alexander Ruede. A Scientist's Guide to FPGAs. iCSC, 2019. https://indico.cern.ch/event/766995/contributions/3295773/attachments/1802757/2940958/Alexander_Ruede_-_A_Scientists_Guide_to_FPGAs_AScientistsGuideToFPGAs.pdf.

[36] Kenneth I Schultz, Michael W Kelly, Justin J Baker, Megan H Blackwell, Matthew G Brown, Curtis B Colonero, Christopher L David, Brian M Tyrrell, and James R Wey. Digital-Pixel Focal Plane Array Technology. *Lincoln Laboratory Journal*, 20(2):16, July 2014.

[37] L. Shannon, V. Cojocaru, C. N. Dao, and P. H. W. Leong. Technology scaling in fpgas: Trends in applications and architectures. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 1–8, 2015.

[38] Ahmad Shawahna, Sadiq M. Sait, and Aiman El-Maleh. FPGA-based Accelerators of Deep Learning Networks for Learning and Classification: A Review. *IEEE Access*, 7:7823–7859, 2019. arXiv: 1901.00121.

[39] Abhishek Singh, Praneeth Vepakomma, Otkrist Gupta, and Ramesh Raskar. Detailed comparison of communication efficiency of split learning and federated learning. September 2019. arXiv: 1909.09145.

[40] Special Notice DARPA-SN-16-68. Reconfigurable Imaging (ReImagine) Proposers Day, Defense Advanced Research Projects Agency, Defense Advanced Research Projects Agency Microsystems Technology Office, 675 North Randolph Street, Arlington, VA 22203-2114, September 2016.

[41] Russell Tessier, Kenneth Pocek, and André DeHon. Reconfigurable Computing Architectures. *Proceedings of the IEEE*, 103(3):332–354, March 2015.

[42] Stephen Trimberger and Jason Moore. FPGA Security: From Features to Capabilities to Trusted Systems. June 2014.

[43] Stephen M. Trimberger. Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology: This Paper Reflects on How Moore's Law Has Driven the Design of FPGAs Through Three Epochs: the Age of Invention, the Age of Expansion, and the Age of Accumulation. *IEEE Solid-State Circuits Magazine*, 10(2):16–29, 2018.

[44] Stylianos I. Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions. March 2018. arXiv: 1803.05900.

[45] Praneeth Vepakomma, Otkrist Gupta, Abhimanyu Dubey, and Ramesh Raskar. Reducing Leakage in Distributed Deep Learning for Sensitive Health Data. 2019. https://aiforsocialgood.github.io/iclr2019/accepted/track1/pdfs/29_aisg_iclr2019.pdf.

[46] Praneeth Vepakomma, Otkrist Gupta, Tristan Swedish, and Ramesh Raskar. Split learning for health: Distributed deep learning without sharing raw patient data. *32nd Conference on Neural Information Processing Systems (NIPS 2018)*, December 2018.

[47] Praneeth Vepakomma, Tristan Swedish, Ramesh Raskar, Otkrist Gupta, and Abhimanyu Dubey. No Peek: A Survey of private distributed deep learning. December 2018. arXiv: 1812.03288.

[48] Teng Wang, Chao Wang, Xuehai Zhou, and Huaping Chen. A Survey of FPGA Based Deep Learning Accelerators: Challenges and Opportunities. December 2018. arXiv: 1901.04988 version: 1.

[49] Pierre-Francois Wolfe, Rushi Patel, Robert Munafo, Mayank Varia, and Martin Herbordt. Secret Sharing MPC on FPGAs in the Datacenter. July 2020. arXiv: 2007.00826.

[50] Xun Wu, Jean L. Sanders, Xiao Zhang, Feysel Yalçın Yamaner, and Ömer Oralkan. An FPGA-Based Backend System for Intravascular Photoacoustic and Ultrasound Imaging. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 66(1):45–56, January 2019.

[51] Xilinx. Vitis AI User Guide. December 2019. https://www.xilinx.com/support/documentation/sw_manuals/vitis_ai/1_0/ug1414-vitis-ai.pdf.

[52] Richard Younger. Photons Reimagined: Large format digital sensors for fast photon counting and HDR imaging. CPAD Instrumentation Frontier Workshop 2018, December 2018.

[53] Jie Zhang, Feng Yuan, Linxiao Wei, Yannan Liu, and Qiang Xu. VeriTrust: Verification for Hardware Trust. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(7):1148–1161, July 2015.

[54] Zhixiang Zhao, Siwei Xie, Xi Zhang, Jingwu Yang, Qiu Huang, Jianfeng Xu, and Qiyu Peng. An Advanced 100-Channel Readout System for Nuclear Imaging. *IEEE Transactions on Instrumentation and Measurement*, 68(9):3200–3210, September 2019.

[55] Jianping Zhu, Rui Hou, XiaoFeng Wang, Wenhao Wang, Jiangfeng Cao, Lutan Zhao, Fengkai Yuan, Peinan Li, Zhongpu Wang, Boyan Zhao, Lixin Zhang, and Dan Meng. Enabling Privacy-Preserving, Compute- and Data-Intensive Computing using Heterogeneous Trusted Execution Environment. https://arxiv.org/ftp/arxiv/papers/1904/1904.04782.pdf.