Adrastea: An Efficient FPGA Design Environment for Heterogeneous Scientific Computing and Machine Learning

Aaron R. Young[1], Narasinga Rao Miniskar[1], Frank Liu[1], Willem Blokland[1], and Jeffrey S. Vetter[1]

Oak Ridge National Laboratory, Oak Ridge TN 37830, USA,
youngar@ornl.gov

Abstract. We present Adrastea, an efficient FPGA design environment for developing scientific machine learning applications. FPGA development is challenging, from deployment, proper toolchain setup, programming methods, interfacing FPGA kernels, and more importantly, the need to explore design space choices to get the best performance and area usage from the FPGA kernel design. Adrastea provides an automated and scalable design flow to parameterize, implement, and optimize complex FPGA kernels and associated interfaces. We show how virtualization of the development environment via virtual machines is leveraged to simplify the setup of the FPGA toolchain while deploying the FPGA boards and while scaling up the automated design space exploration to leverage multiple machines concurrently. Adrastea provides an automated build and test environment of FPGA kernels. By exposing design space hyper-parameters, Adrastea can automatically search the design space in parallel to optimize the FPGA design for a given metric, usually performance or area. Adrastea simplifies the task of interfacing with the FPGA kernels with a simplified interface API. To demonstrate the capabilities of Adrastea, we implement a complex random forest machine learning kernel with 10,000 input features while achieving extremely low computing latency without loss of prediction accuracy, which is required by a scientific edge application at SNS. We also demonstrate Adrastea using an FFT kernel and show that for both applications Adrastea is able to systematically and efficiently evaluate different design options, which reduced the time and effort required to develop the kernel from months of manual work to days of automatic builds.

Keywords: Design Space Exploration, FPGA Development Environment, Heterogeneous Computing, Scientific Computing, Machine Learning

1 Introduction

Field-Programmable Gate Arrays (FPGAs) enable the development and deployment of custom hardware designs via programmable logic blocks, which can be reconfigured to perform custom functions and allow for flexible, reconfigurable

computing. FPGAs excel at applications that run close to the edge (near the data collection) and for workloads that require low-latency solutions. FPGAs also excel at applications that can be expressed as data-flows between tasks and where the tasks can be implemented as a processing pipeline. Then during execution, the data flows between kernels in a stream, and the processing is done via pipelines to maximize performance.

Some examples of low-latency applications that work well on FPGAs include traditional real-time industrial control applications where fixed latencies are mandatory[18, 4] and other typical low-latency FPGA applications, including financial technology[14, 8, 16], communication and networking[17, 20, 6, 2], and real-time cryptosystems [10, 9].

Although historically, FPGAs have been primarily used for embedded systems and custom low-level designs, FPGAs are becoming increasingly accessible as general compute accelerator cards. FPGAs are now being packaged in server-grade PCIe cards, with software support to enable quick reloading of designs and handling common tasks such as execution control and data transfer from the host to the FPGA. High-Level Synthesis (HLS) allows FPGA kernels to be written in higher-level languages like OpenCL, then compiled to the FPGA design through the HLS design flow.

Although FPGAs can now be used effectively in a shared server environment and HLS eases the complexity of FPGA kernel design, FPGA development is still challenging. Challenges with FPGA development include kernel deployment, proper toolchain setup, programming methods, interfacing with FPGA kernels, time consuming HLS compilation/synthesis time, and, more importantly, the need to explore design space choices to get the best performance and area usage from the FPGA kernel design. We present Adrastea, an efficient FPGA design environment for developing optimized kernels for high-performance applications to address these challenges. Adrastea provides an automated and scalable design flow to parameterize, implement, and optimize complex FPGA kernels and associated interfaces. In this paper, we show how virtualization of the development environment via virtual machines is leveraged to simplify the setup of the FPGA toolchain while deploying the FPGA boards and while scaling up the automated design space exploration to leverage multiple machines concurrently. Adrastea provides an automated build and test environment for FPGA kernels. By exposing design space hyperparameters, Adrastea can automatically search the design space in parallel to optimize the FPGA design for a given metric, usually performance or area. Adrastea also simplifies the task of interfacing with the FPGA kernels by providing a simplified interface API.

To demonstrate the capabilities of Adrastea, we implement a complex Random Forest machine learning kernel with 10,000 input features while achieving extremely low computing latency without loss of prediction accuracy, which is required by a scientific edge application at SNS. While developing this kernel, Adrastea was utilized to systematically and efficiently evaluate different design options, which reduced the time and effort required to develop the kernel. The Random Forest kernel implemented with Adrastea is 5x area-efficient and can

perform inference in 60 nanoseconds. We have also demonstrated the ease of using Adrastea by leveraging Adrastea to perform a design space search on a Fast Fourier Transform (FFT) kernel. We demonstrated that the Adrastea framework results in an accelerated FPGA development and optimization cycle able to perform months of effort in days. The main contribution of this paper is summarized as follows:

- We design and implement an efficient FPGA design environment Adrastea, with the capabilities of designing both the kernel implementation of FPGA and the interface, as well as the capabilities to perform FPGA optimizations;
- We demonstrate the efficiency and effectiveness of Adrastea for FPGA design on Random Forest and Fast Fourier Transform applications.

The remaining parts of this paper are organized as follows: In Section 2, we provide a background on FPGA design, the Xilinx Vitis Toolchain, and a brief review of related work on FPGA build frameworks. In Section 3, we discuss the design and use of Adrastea. In Section 4, we provide examples of building two applications using the Adrastea framework and discuss results from the design space explorations, followed by a conclusion and future work in Section 5.

2 Background

In this section, we provide more details on the principles of FPGA development and a general description of the Xilinx Toolchain.

2.1 FPGA Design

FPGAs require a different programming style from CPUs and GPUs, and code written for a CPU or GPU will likely need to be rewritten to meet the desired performance goals. When writing code for the FPGA, three paradigms are helpful to keep in mind for designing FPGA kernels [23]. These paradigms are producer-consumer, streaming data, and pipelining. Together these paradigms are a useful way to think about designing for FPGAs. Computation can be broken into tasks that operate on a data stream. These tasks then read the data from a stream, perform computation on the data, then send the output data to the next task. The tasks and communication can then be pipelined both within a task, at the instructions level, or between tasks, at the task level. The performance of the pipeline is expressed by the initiation interval (II) and the iteration latency. From this, the total latency can be computed as shown in Equation 1.

$$\text{Total Latency} = \text{Iteration Latency} + \text{II} \cdot (\text{Number of Items} - 1) \tag{1}$$

The total application latency is the total latency of the critical dataflow path, and the application throughput is determined by the width of the data streams and the largest II. There is a trade-off that occurs between the latency, II, area, and clock frequency of the design, and Adrastea can be used to aid in the design space search to optimize the kernel to find the best kernel design for a given application and target FPGA.

2.2 Xilinx Vitis Toolchain

The Xilinx Vitis unified software platform [11] is a comprehensive development environment to build and deploy performance-critical kernels accelerated on Xilinx platforms, including Alveo cards, cloud FPGA instances, and embedded FPGA platforms. Vitis supports FPGA accelerated application programming using C, C++, OpenCL, and RTL by providing a variety of pragmas for pipelining, dataflow, and optimizations. High-Level Synthesis (HLS) is used to convert kernels written in high-level programming languages like C, C++, and OpenCL C into RTL kernels which can then be implemented with the FPGA hardware. In addition, the Xilinx Runtime Library (XRT) provides APIs to facilitate communication between the host application and accelerators. XRT eases accelerator life cycle management and execution management. It also handles memory allocation and data communication between the host and accelerators.

2.3 Related Work

One of the recent related works is HLS4ML[5], an open-source hardware-software codesign workflow to translate machine learning kernels into hardware designs, including FPGAs. The major difference between HLS4ML and this work is the capability to explore various design choices for general FPGA designs, while HLS4ML is more focused on optimization for ML kernels.

DARClab developed an automatic synthesis option tuner dedicated to optimizing HLS synthesis options [22]. This HLS design space explorer leverages previous results to explore the HLS parameters effectively. Our work is more general on the design space knobs allowing the application developer to expose any knobs they want to explore.

Coyote is a new shell for FPGAs that aims to provide a full suite of operating system abstractions. In our work, we use the Alveo Shell from Xilinx, but Adrastea could be extended to use other FPGA tool flows and FPGA shells.

Cock et al. designed a new hybrid CPU and FPGA server system to enable hybrid systems research, and the system uses a cache-coherent socket to socket protocol to connect to the FPGA [3]. In this work, we leverage the off-the-shelf servers and a PCIe-based Alveo FPGA. Cabrera et al. explored CXL and the implications a CXL interface would have on the programming models for FPGA kernel design [1]. However, a CXL-enabled FPGA card and server are not available off the shelf at the time of writing this paper.

3 Adrastea Design Environment

In this section, we present the design of an efficient FPGA development environment, with the code name Adrastea, that we created to support the automated process of building and optimizing general FPGA hardware designs capable of deployment at the edge. Adrastea leverages and combines various related efforts from ORNL to build a powerful and efficient FPGA development

environment that solves key challenges with FPGA design, including 1) setting up a scalable build environment 2) providing convenient interfaces to the FPGA kernel 3) performing design space exploration

The FPGA build environment leverages the Vitis tool flow discussed in Subsection 2.2 to synthesize and implement the FPGA designs. Subsection 3.1 covers the virtual machine-based deployment of the development environment, which enables sharing of computing resources, maintaining multiple versions of the toolchain, and scaling of build servers for design space experiments. Finally, CMakefiles provided by IRIS simplify the build process by generating makefiles that automate the kernel builds.

Adrastea supports easier interfacing to the FPGA kernel through the use of the IRIS runtime as specified in Subsection 3.2. This interface allows common handling of the host communication logic and easy-to-use APIs which is common across multiple accelerator types. With this API, it is easy to have a kernel generate and run targeting multiple execution targets, including CPU, GPU, DSP, and FPGA.

By leveraging DEFFE, Adrastea can easily launch large design space search and optimization runs across multiple build servers. This capability is further discussed in Subsection 3.3.

With all the components combined, Adrastea implements the design loop shown in Figure 1. Step 1) leverages DEFFE to build the FPGA kernels using

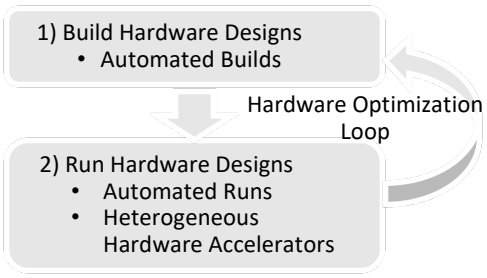

Fig. 1: Hardware optimization loop.

automated build flows. These builds can be parameterized with architecture-level parameters in addition to the synthesis and implementation flags. Different build configurations can also be run in parallel on the build servers to allow these long-running processes to execute concurrently. Once the kernel builds are finished, the designs can then move to step 2) where they are loaded and run using FPGA hardware or in emulation. As indicated by the curved arrow, the build logs and execution results collected from the FPGA implementation and can then be used to provide feedback to the automatic generation steps to improve on the resulting hardware design. This infrastructure creates a basis for performing automated hardware optimization loops and continuous integration where both building and running designs can be completely automated and scheduled.

3.1 FPGA Building Environment

There are two main challenges with this update loop: 1) the hardware builds take multiple hours to run; 2) hardware testing must be done on a system with the hardware available. Both of these issues are addressed by the toolchain infrastructure deployed on a computer cluster for experimental computing. To address the first issue, a scalable compute cluster for building is utilized to allow many hardware builds to be executed in parallel. Although there is not an easy way to accelerate a single build, multiple builds run in parallel can be used to evaluate different build configurations concurrently. To address the second issue, a system with FPGA accelerators is available for hardware execution. The Slurm job scheduler is used to allocate both build resources and FPGA hardware resources for submitted jobs. The Vitis Unified Software Development Platform is a large software installation, ∼110 GB, with strict operating system requirements, and only one version of the Xilinx Runtime library (XRT) can be installed at a time. To make the setup of multiple machines and the changing of versions easier, the development platform is installed inside a Kernel-based Virtual Machine (KVM). PCIe passthrough is used to pass the hypervisor's FPGA hardware to the VM. The QCOW2 disk image format is used by the Virtual Machine (VM) so that common system files, packages, and Vitis can be installed on the backing disk image file. Then each VM instance based on that backing file uses Copy-on-Write (COW) to create overlay images that only store changes from the base image. The backing file then resides in Network File System (NFS) with the overlay images stored in a local drive of the hypervisor. This allows the unchanging, large, base files to be stored in a single shared file, with the files that change more often being stored in the faster local storage.

Slurm and GitLab-CI runners are installed in the VMs using Ansible to enable easy launching of jobs with a continuous integration (CI) pipeline or via Slurm. The FPGA is configured as a Generic Resource (GRES) in Slurm to allow Slurm to allocate the FPGA to jobs. The GitLab-CI pipelines and other automatic build launching scripts, including DEFFE, make use of Slurm to allocate build resources and coordinate the use of the FPGAs. Developers using the system also leverage Slurm to allocate resources for building and testing FPGA designs. The VMs are divided into multiple partitions in Slurm; the FPGA build partitions have nodes with the Vitis toolchain but without FPGA hardware, and the FPGA run partitions have both the toolchain and FPGA hardware. The different versions of Vitis also have their own partitions. There are two partitions, a build and a run partition, for each of the available Vitis versions. Since Ansible and VMs with overlay disk images are used, additional nodes can easily be added or removed and the Vitis version can be swapped out. Startup and teardown scripts automate the process of adding or removing new VM instances, making it trivial to change the version of Vitis that is running on the build servers. Changing the VM where the FPGA is assigned requires more steps since the process includes shutting down the VMs, changing the PCIe passthrough configuration, starting the VM with the FPGA, loading the Vitis shell which corresponds to that version, rebooting the system to load the new FPGA image, starting all the VMs on the system,

loading the user shell for the FPGA, and finally using the `xbutil` to validate the FPGA setup.

The infrastructure enabling the design flow shown in Figure 1 is implemented in the compute cluster as shown in Figure 2. Figure 2 shows the same hardware optimization loop updated with the build and run nodes that are used to build the hardware in parallel and run the designs on hardware in the respective VMs as scheduled by Slurm. Since the VMs mount the same network drives and home

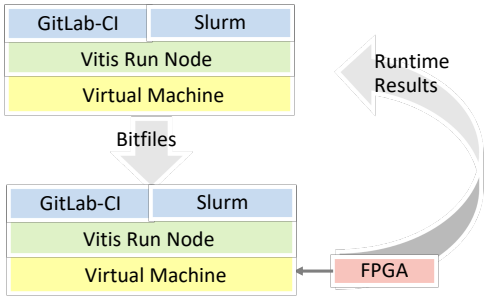

Fig. 2: Infrastructure supporting the hardware optimization loop.

directories, the bitfiles, runtime results, and other artifacts can easily be shared between nodes in the cluster.

3.2 IRIS

Adrastea uses the IRIS task-based programming model for interfacing with the FPGA kernels [12]. IRIS provides a unified task-based programming interface, which can be targeted for heterogeneous compute units such as multi-core CPUs, GPUs (NVidia/AMD), FPGA (Intel and Xilinx), Hexagon DSPs, etc. It also provides flexibility to the programmer to write task core kernels either using OpenMP, OpenACC, CUDA, HIP, OpenCL, etc. Memory transfers between the host and the devices are managed through IRIS's memory coherence runtime management. Through IRIS, Adrastea provides the programmer with a simplified interface which can be called using either a C++ or python application. A simple python function to call the random forest kernel to run on FPGA is shown in Figure 3.

3.3 DEFFE

DEFFE (Data Efficient Framework for Exploration) [13] is intended for design space exploration. It is configurable with parameters (often referred to as knobs) with possible ranges of values and with cost metrics that are used for evaluation. It has the flexibility to provide a custom evaluate (Python/Bash) script to evaluate each set of sample parameter-value combinations. It also provides flexibility

```
import iris
def run_random_forest(args, test_data):
   features= test_data.data
   Y = test_data.target
   Y_predict = np.zeros(Y.size, dtype=np.int8)
   SIZE = Y.size
   iris.init()   # Initialize IRIS Run-time
   mem_features = iris.mem(features.nbytes)
   mem_Y_predict = iris.mem(Y_predict.nbytes)
   task = iris.task()   # Create IRIS task
   task.h2d(mem_features, 0, features.nbytes, features)
   task.kernel("rf_classifier", 1, [0], [1], [1],
         [mem_features, SIZE, mem_Y_predict], #Parameters
         [iris.iris_r, 4, iris.iris_w] )   # Parameters information
   task.d2h(mem_Y_predict, 0, Y_predict.nbytes, Y_predict)
   cu = iris.iris_cpu
   if args.cu == 'fpga':
      cu = iris.iris_fpga
   task.submit(cu)
   iris.finalize()
```

Fig. 3: Adrastea Python interface through IRIS

to extract the results from the evaluation using a customized extract script (Python/Bash). These scripts use arguments or environment variables to receive the parameters from DEFFE. Additionally, DEFFE has a configurable machine learning model which can be used for workload characterization. DEFFE is supported by different variants of sampling techniques such as DOEPY (Design of Experiments) [19] sampling techniques and machine learning-based sampling techniques. The configurability of DEFFE enables fast design space exploration with the parallel execution either using multi-core threading or by using a massively parallel, multiple system Slurm environment. This paper uses the build environment with Slurm as discussed in Section 3.1 and explores the *one-dim* sampling technique for exploring the FPGA design choices.

3.4 Experiment Setup via Git

One of the challenges with building FPGA kernels and conducting design space search experiments is tracking the history of how the experiment was performed and allowing the experiment to be easily reproducible. Source code, including source code for FPGA designs, is commonly stored in source code repositories like Git in order to track changes in the code. Git can also be used to keep track of the automatic build scripts and DEFFE experiment configurations. To ensure history tracking and reproducibility of a hardware build, we first create an experiment repository to hold the DEFFE configuration along with the scripts to automate the FPGA build. Iris and DEFFE repositories are then added as sub-modules to the repository. The generators and source code for the design can either be included in its own repository and added as a sub-module or directly included in the experiment repository. When the hardware is built, or DEFFE is run, the hash of the current commit is included in the logs of the build. That way, to reproduce the build or to rerun a DEFFE experiment, you only need to check out the same commit. The commit information is all that is needed to

reproduce the experiment since the repository includes the same version of the sub-modules and all the scripts that were used for the build.

Sometimes we also want to store the history of how and when a bitfile was built. To do this we take the hash of the date, folder name, and commit id. This hash is then stored in a file that is committed to the repository and included as a read-only output in the hardware kernel. Then to reproduce the build or learn how a deployed design was built, you only need to read the hash from the kernel and search the repository for the hash. Then you know exactly the commit and, therefore, the state and folder of the repository that was used to build that hardware kernel.

The method of using Git to store the entire build process and experiment flow and keeping track of the hash with the output and result enables reproducibility and provides historical information of the artifacts.

3.5 Complete Adrastea Build Flow

By leveraging the different components that make up Adrastea, a complete FPGA design flow is built as shown in Figure 4. As input to DEFFE, the design space knobs, cost metrics, experiment setup, and other configuration is passed to DEFFE via a config.json file. Additionally, the customized evaluate script and extract script are passed, which contain the instructions to perform the build, FPGA execution and results (cost metrics) extraction. DEFFE has more components, but the main ones used in the Adrastea build flow are shown. The design space is sampled, and the experiment folders are created. Each experiment folder contains the parameterized scripts used for the run for that set of parameters. Next, DEFFE builds the FPGA design in parallel using the build partition compute units of Slurm via the Xilinx Vitis Toolchain. The build flow used by the evaluate script for ML workloads is also shown in the figure. The ML model, along with the build knobs, is passed to an optimized ML code generator script which generates the HLS code. Then the HLS code is compiled using the Vitis and Vivado toolchains into an FPGA bitstream. After which, DEFFE uses the run partition compute units of Slurm to evaluate the FPGA kernel using the IRIS runtime. Finally, DEFFE writes the results from the design space search into a spreadsheet. This spreadsheet can then be read into a graphing program and the results from the experiment can be plotted. We use a Jupyter Notebook with Seaborn and Pandas to plot the figures shown in this paper.

4 Example Applications Leveraging Adrastea

We have evaluated the effectiveness of Adrastea for an SNS application with a Random Forest (RF) classification model deployed on Xilinx FPGAs and also deployed other state-of-the-art random forest classification models. We also used Adrastea with an FFT kernel to demonstrate how quickly and easily new applications can be built and explored using the Adrastea development environment.

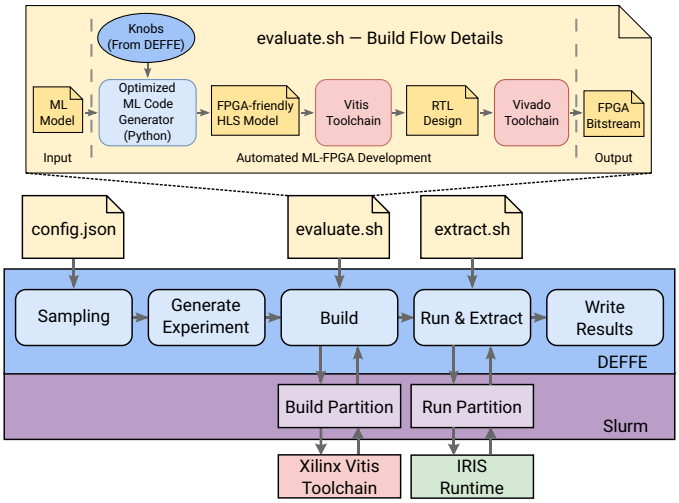

Fig. 4: Adrastea Build Flow

Experimental Setup We deployed the Adrastea development environment on an experimental cluster using five servers as hypervisors. One Atipa server with dual-socket Intel Xeon Gold 6130 CPUs, 192 GiB of RAM, and a Xilinx Alveo U250 was used as the hypervisor for the *run VM*. The *run VM* was configured with 24 vCPUs, 92 GiB of RAM, and with PCIe passthrough to access the U250. Four HPE DL385 Servers with dual-socket AMD EPYC 7742 CPUs and 1 TB of RAM were used as the hypervisor for the *build VMs*. The *build VMs* are configured with 240 vCPUs and 960 GB of RAM. The Xilinx Alveo U250 board is used to evaluate the FPGA designs.

4.1 SNS and other state-of-the-art RF models design space exploration

The SNS facility at Oak Ridge National Laboratory (ORNL) has the world's highest pulsed power proton accelerator delivering 1.4 MW of a proton beam repeated at 60 Hz onto a stainless steel vessel filled with liquid mercury to generate neutrons for material research [7]. In order to detect beam loss and abort the experiment, we designed a custom random forest implementation using Adrastea [15]. The application requires a highly complex random forest model which uses 10000 features to make a meaningful prediction and the model needs to be updated frequently. We leveraged Adrastea not only to optimize the implementation for RF models to detect the errant beam within 100ns but also for faster development, design space exploration, and deployment of the random forest model in the SNS environment FPGA hardware.

Design parameters and exploration Adrastea has provided a python script to generate the Xilinx OpenCL code for the random forest model with flattened nodes while analyzing the dependency of nodes in random forest trees. It has several knobs to finetune the random forest generation and build process to explore optimal hyper parameters, such as 1) bitwise optimization flag to enable bitwise arithmetic operators instead of logical operators, 2) different type random forest tree code generation such as if-else conditional trees, flattened version of tree code generation, 3) configurable datatype of features such as floating-point (FP32), 8-bit fixed-point (FX8), and varying fixed-point bitwidth controlled by accuracy threshold percentages ranging from 10 to 100 (FT[10-100]), 4) type of fixed-point approach such as quantization or power of 2 values representation, 5) voting algorithm logic approach either with direct BRAM based array accesses, flattened approach either with or without static single assignment optimization. The design space exploration also explores the optimal frequency setting for the generated random forest kernel. The design space exploration parameters for the Adrastea-based random forest model are shown in Table 1. We have configured all these parameters in the DEFFE configuration file. DEFFE sampled the essential set of these parameter values combination and evaluated them using massively parallel Slurm jobs. Each Slurm job is configured to use eight threads, resulting in the ability to run 100 simultaneous runs across the FPGA build machines. Completed FPGA implementations were then tested serially on the run partition of Slurm where the Alveo U250 FPGA is connected. The build time of each sample evaluation is $\sim$ 4 hrs, but the execution time on FPGA hardly takes milliseconds. Hence, it is reasonable to have multiple build systems and one FPGA run system.

Table 1: Parameters used for experiments

Parameter	Type	# of Values	Values
Frequency (MHz)	Build	5	50, 100, 200, 300*, 500
Decision tree	Generator	2	Conditional (conditional_trees), Flattened*
Bitwise optimization	Generator	2	Yes*, No (no_bitwise)
Feature datatype	Generator	12	Float (FP32), Fixed 8-bit(FX8), FT100*, FT90, FT80, FT70, FT60, FT50, FT40, FT30, FT20, FT10
Fixed point type	Generator	2	Quantization*, Power2 (no_quantization)
Voting	Generator	3	Array (array_votes), SSA*, No-SSA (no_votes_ssa)

*: Base Parameters (The best)

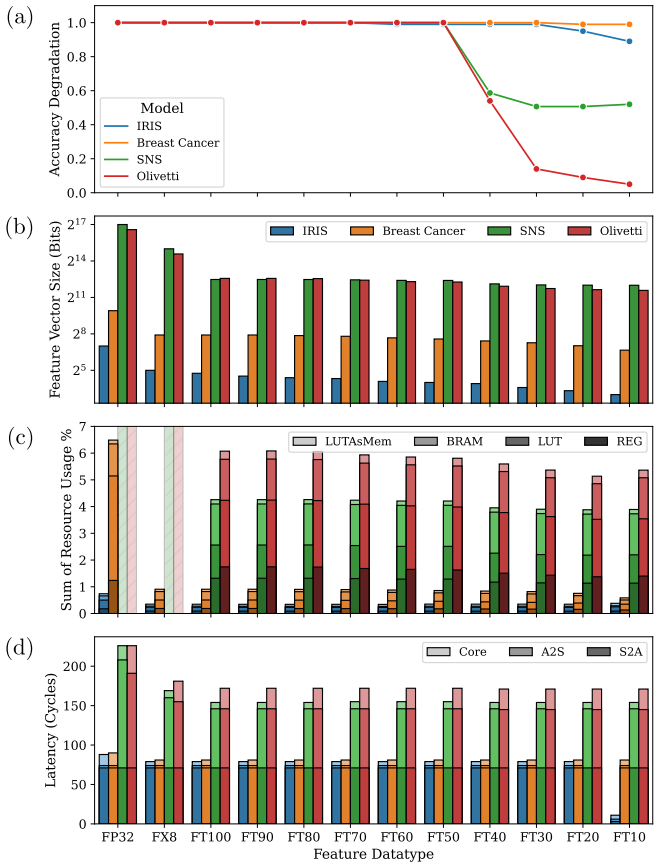

Fig. 5: Optimization metrics of interest for different input feature datatypes.

Results We have experimented with the random forest model not only for the SNS dataset but also for other standard random forest models such as IRIS (4 features), Breast cancer (30 features), and Olivetti (4096 features) datasets which have 100 trees with maximum depth set to 20. The experiment results for varying datatype of features and its impact on the latency, resource usage, feature vector size, and prediction accuracy are shown in Figure 5. Accuracy degradation is used instead of raw classification accuracy of the testing data since the focus is the accuracy loss from reducing the bits used to represent the input features.

It can be seen that the representation of features with FX8 will lead straight away to a 4x reduction in feature vector size when compared to 32-bit floating-point features. Further exploration of variable fixed-point datatypes with accuracy threshold will lead to higher gains. For example with 70% (FT70) threshold the multi-precision fixed-point feature vector size reduction gains are 23x for SNS, 17x for Olivetti, 4.3x for Breast Cancer, and 6.4x for IRIS, as shown in

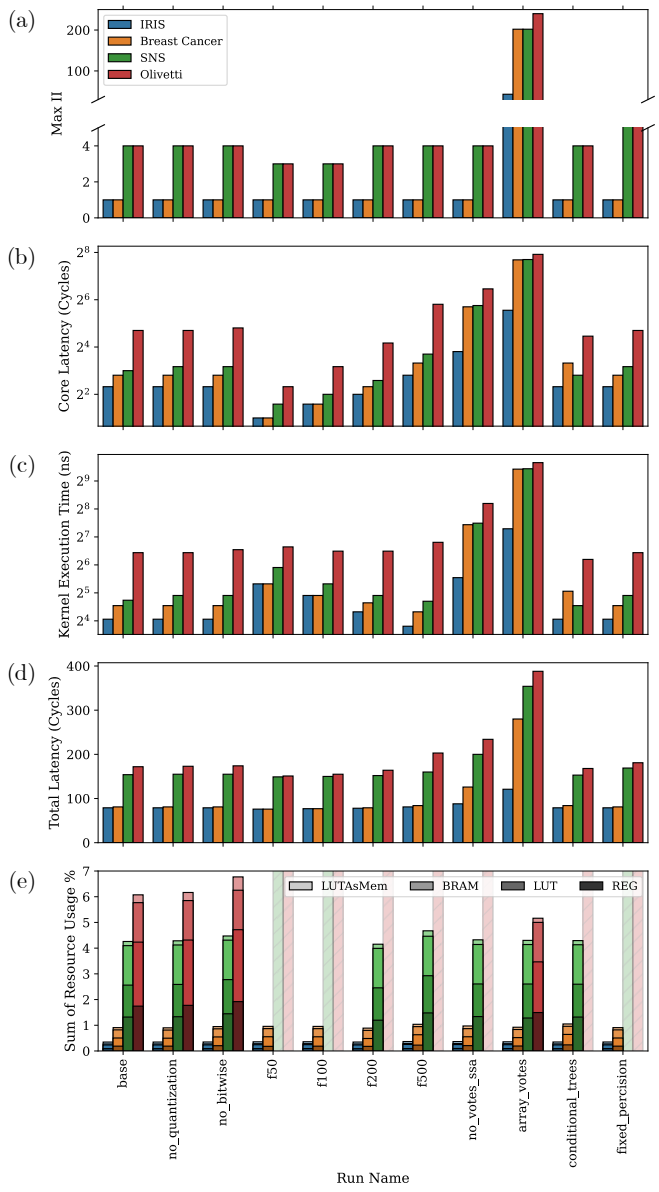

Fig. 6: Comparison with different build options.

Figure 5(b). Moreover FT70 has resulted in no accuracy degradation compared to the floating-point datatype for all benchmarks, as shown in Figure 5(a).

The overall FPGA resource usage of our fixed point design of random forest models is ∼2%. Figure 5(c) shows the breakdown of resource usages of registers

(REG), look-up tables (LUT), block RAM (BRAM), and LUTs as memory (LUTAsMem) as a percentage of the available resources of that type.

The total latency of the models is shown in Figure 5(d) for kernels running at 300MHz frequency. Total latency is calculated as the sum of the latencies (cycles) for the AXI4 to stream (*A2S*) conversion for reading input features, the core functionality, which contains the random forest classifier (*Core*), and the stream to AXI4 (*S2A*) conversion for the classification output. We have also shown the effect of other design space parameters in the Figure 6.

4.2 FFT

FFT is a data-intensive and also compute-intensive algorithm used to convert time domain data to frequency domain for many applications. FFT is an important component of an Atomic Force Microscope (AFM) and requires a very wide FFT length and size [21]. We have applied design space exploration on 1-D (single-dimensional) FFT, optimized and available in the Xilinx Vitis library. We have ported the Xilinx Vitis 1-D FFT implementation in the Adrastea environment using IRIS run-time scheduler APIs. We have applied DEFFE design space exploration on the generated FFT implementation. The FFT implementation has two design parameters, FFT length (Window) and FFT Size (number of elements), along with the frequency of kernel to explore, which will have an impact on resource utilization and runtime latency. We have measured the latency of FFT algorithm in execution time instead of cycles as it has many iterative loops and kernel functions. The results of the design space exploration are shown in Figure 7. FFT length is explored for the range of 1024 to very large window size of 64K (65536). The 64K FFT build was observed to be successful for 50MHz frequency. The user can now choose the right design point for the required FFT application.

4.3 Effectiveness of Adrastea

Though there are highly efficient high-level synthesis tools for FPGA development, it is still a challenge for programmers to adapt to FPGA development. These tools simplified the FPGA core kernel programming as easy as writing C++/OpenCL code. However, it takes weeks for programmers to write interface code as they have to understand how the FPGA interface works. Using Adrastea, one can integrate their kernel in the Adrastea environment and call the kernel from the host program with a few simple function calls. This API can be called from either a C++ or a python application. Adrastea makes design space exploration as simple as providing a JSON configuration file and starting DEFFE. DEFFE then leverages Slurm to build in parallel to get the design space exploration results in a day, whereas, in the traditional FPGA design environment, manual exploration takes months to realize and perform the exploration. In summary, Adrastea enables the FPGA development from programming to deployment within a day while also conducting a design space exploration, which not only saves programmers time but also eases the FPGA development.

Table 2: Adrastea to Speedup Design

	Traditional FPGA design	Adrastea based SNS Random Forest	Adrastea based FFT
Adrastea Integration Time	-	1 hr (Easy setup)	1 hr
Interface Programming Time	Weeks (Needs expertise)	1 hr (Very easy similar to function call)	1 hr
Build script Writing	2 Days (Needs expertise)	1 hr (Very easy as configuring a CMake variable)	1 hr
Average Build Time	-	$\sim$ 4 hrs	$\sim$ 9 hrs
Number of Builds	-	90	54
Design Space Exploration Time	Months $\sim$ 13 days (SNS) $\sim$ 21 days (FFT) (Sequential build time)	$\sim$ 15 hrs	$\sim$ 26 hrs
Graphing Time	6 hrs	3 hrs (Data is already tabulated)	3 hrs

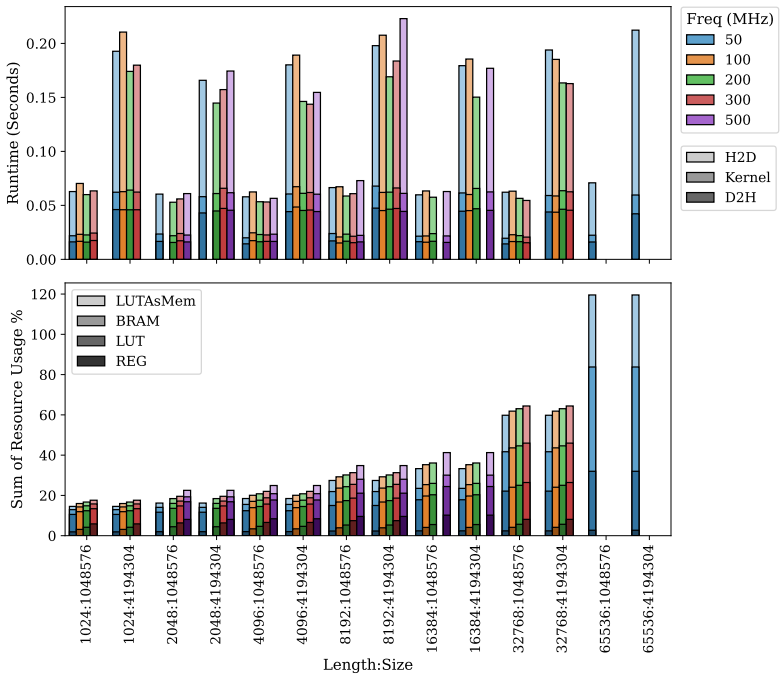

Fig. 7: FFT experiment results. Missing data points did not build successfully due to exceeded resource utilization.

5 Conclusion and Future Work

This paper presents Adrastea, an automated and efficient design environment to design, implement, and optimize complex FPGA kernels and their interfaces. By leveraging the power of Adrastea, we can implement and optimize complex random forest machine learning models (SNS, Olivetti, etc.,) and complex FFT kernels on a Xilinx Alveo U250 FPGA within a few days. In contrast, it would take months to design the kernel and perform the same design space exploration with a traditional FPGA design environment. This speedup is achieved from the simplified interfacing code, and parallel building of the FPGA designs in a compute cluster.

For future work, we plan to enhance Adrastea's capability further and utilize Adrastea to explore other mission-critical FPGA computing kernels. We also plan to leverage Adrastea to optimize applications that leverage multiple heterogeneous accelerator types.

Acknowledgments This research used resources of the Experimental Computing Laboratory (ExCL) at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725

Bibliography

[1] Cabrera, A.M., Young, A.R., Vetter, J.S.: Design and analysis of cxl performance models for tightly-coupled heterogeneous computing. In: Proceedings of the 1st International Workshop on Extreme Heterogeneity Solutions, ExHET '22. Association for Computing Machinery, New York, NY, USA (2022). DOI 10.1145/3529336.3530817. URL https://doi.org/10.1145/3529336.3530817

[2] Chacko, J., Sahin, C., Nguyen, D., Pfeil, D., Kandasamy, N., Dandekar, K.: Fpga-based latency-insensitive ofdm pipeline for wireless research. In: 2014 IEEE high performance extreme computing conference (HPEC), pp. 1–6. IEEE (2014)

[3] Cock, D., Ramdas, A., Schwyn, D., Giardino, M., Turowski, A., He, Z., Hossle, N., Korolija, D., Licciardello, M., Martsenko, K., Achermann, R., Alonso, G., Roscoe, T.: Enzian: An open, general, cpu/fpga platform for systems software research. In: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2022, p. 434–451. Association for Computing Machinery, New York, NY, USA (2022). DOI 10.1145/3503222.3507742. URL https://doi.org/10.1145/3503222.3507742

[4] Dufour, C., Cense, S., Ould-Bachir, T., Grégoire, L.A., Bélanger, J.: General-purpose reconfigurable low-latency electric circuit and motor drive solver on fpga. In: IECON 2012-38th Annual Conference on IEEE Industrial Electronics Society, pp. 3073–3081. IEEE (2012)

[5] Farabet, C., Poulet, C., Han, J.Y., LeCun, Y.: CNP: An fpga-based processor for convolutional networks. In: 2009 International Conference on Field Programmable Logic and Applications, pp. 32–37. IEEE (2009)

[6] Giordano, R., Aloisio, A.: Protocol-independent, fixed-latency links with fpga-embedded serdeses. Journal of Instrumentation **7**(05), P05,004 (2012)

[7] Henderson, S., Abraham, W., Aleksandrov, A., Allen, C., Alonso, J., Anderson, D., Arenius, D., Arthur, T., Assadi, S., Ayers, J., et al.: The spallation neutron source accelerator system design. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **763**, 610–673 (2014)

[8] Huang, B., Huan, Y., Xu, L.D., Zheng, L., Zou, Z.: Automated trading systems statistical and machine learning methods and hardware implementation: a survey. Enterprise Information Systems **13**(1), 132–144 (2019)

[9] Islam, M.M., Hossain, M.S., Hasan, M.K., Shahjalal, M., Jang, Y.M.: Fpga implementation of high-speed area-efficient processor for elliptic curve point multiplication over prime field. IEEE Access **7**, 178,811–178,826 (2019)

[10] Javeed, K., Wang, X.: Low latency flexible fpga implementation of point multiplication on elliptic curves over gf (p). International Journal of Circuit Theory and Applications **45**(2), 214–228 (2017)

[11] Kathail, V.: Xilinx vitis unified software platform. In: Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '20, p. 173–174. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3373087.3375887. URL https://doi.org/10.1145/3373087.3375887

[12] Kim, J., Lee, S., Johnston, B., Vetter, J.S.: IRIS: A Portable Runtime System Exploiting Multiple Heterogeneous Programming Systems. In: Proceedings of the 25th IEEE High Performance Extreme Computing Conference, HPEC '21, pp. 1–8 (2021). DOI 10.1109/HPEC49654.2021.9622873

[13] Liu, F., Miniskar, N.R., Chakraborty, D., Vetter, J.S.: Deffe: a data-efficient framework for performance characterization in domain-specific computing. In: Proceedings of the 17th ACM International Conference on Computing Frontiers, pp. 182–191 (2020)

[14] Lockwood, J.W., Gupte, A., Mehta, N., Blott, M., English, T., Vissers, K.: A low-latency library in fpga hardware for high-frequency trading (hft). In: 2012 IEEE 20th annual symposium on high-performance interconnects, pp. 9–16. IEEE (2012)

[15] Miniskar, N., Young, A., Liu, F., Blokland, W., Cabrera, A., Vetter, J.: Ultra low latency machine learning for scientific edge applications. In: Proceedings of 32nd International Conference on Field Programmable Logic and Applications (FPL' 22). IEEE (2022)

[16] Morris, G.W., Thomas, D.B., Luk, W.: Fpga accelerated low-latency market data feed processing. In: 2009 17th IEEE Symposium on High Performance Interconnects, pp. 83–89. IEEE (2009)

[17] Puš, V., Kekely, L., Kořenek, J.: Low-latency modular packet header parser for fpga. In: 2012 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), pp. 77–78. IEEE (2012)

[18] Rodríguez-Andina, J.J., Valdes-Pena, M.D., Moure, M.J.: Advanced features and industrial applications of fpgas—a review. IEEE Transactions on Industrial Informatics **11**(4), 853–864 (2015)

[19] Sarkar, T.: DOEPY design of experiments. https://doepy.readthedocs.io/en/latest/. Accessed: 2020-09-30

[20] Sidler, D., Alonso, G., Blott, M., Karras, K., Vissers, K., Carley, R.: Scalable 10gbps tcp/ip stack architecture for reconfigurable hardware. In: 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, pp. 36–43. IEEE (2015)

[21] Somnath, S., Belianinov, A., Kalinin, S.V., Jesse, S.: Rapid mapping of polarization switching through complete information acquisition. Nature Communications **7**(1) (2016). DOI 10.1038/ncomms13290

[22] Wang, Z., Schafer, B.C.: Learning from the past: Efficient high-level synthesis design space exploration for fpgas. ACM Trans. Des. Autom. Electron. Syst. **27**(4) (2022). DOI 10.1145/3495531. URL https://doi.org/10.1145/3495531

[23] Xilinx: Vitis high-level synthesis user guide (ug1399) (2022). URL https://docs.xilinx.com/r/en-US/ug1399-vitis-hls