Facilitating Academic Research with FPGA Support in a University Data Center

Jeevesh Choudhury
Arizona State University
jchoudh3@asu.edu

Thomas Jennewein
Arizona State University
tjennewe@asu.edu

Gil Speyer
Arizona State University
speyer@asu.edu

ABSTRACT

Field Programmable Gate Arrays (FPGAs) offer a practical solution that balances computational power with energy efficiency, which could address the growing demand for sustainable high-performance computing (HPC). Moreover, because they can be reconfigured and optimized for specific applications, FPGAs open up numerous possibilities for adaptive, high-performance workloads. However, the substantial expertise required to deploy FPGA designs has traditionally been daunting, requiring proficiency in Hardware Description Languages (HDL) such as SystemVerilog or VHDL. To address this accessibility barrier, the field has shifted toward high-level synthesis (HLS), which allows developers to program FPGAs using familiar languages like C++ and Python — mirroring the evolution seen in GPU programming.

In this paper, the resources available on the Sol HPC cluster at Arizona State University (ASU) [8] and the strategies employed to support and encourage researchers and instructors working with these nodes are examined. The practical challenges of using FPGAs, the integration of tools and libraries in the development workflow, and efforts to lower the expertise threshold required for effective use are explored. By sharing this experience, the aim is to contribute to the growing body of knowledge around accessible and sustainable FPGA development in HPC environments.

KEYWORDS

FPGA, HPC, Facilitation

1 INTRODUCTION

With the advent of the exascale era of high-performance computing, accelerators have become vastly more expansive and heterogeneous and often include various novel architectures besides GPUs, such as FPGAs, Vector Engines (VEs), Wafer-Scale Engines (WSEs), and Intelligence Processing Units (IPUs). Among such accelerators, FPGAs have often served as the platform to design, prototype, experiment, and deploy novel architectures that accelerate applications while leveraging a balance between computational speed and power efficiency. The versatility of FPGAs allow them to be used both as computational and network accelerators, as shown in Microsoft's Project Catapult [12], and they can be connected in varying configurations as shown in Lant, et al. [9] to exploit various compute advantages.

Due to the FPGA's low power and programmability features, implementation of edge computing designs has gained wide popularity. In the context of the data center, this use case would seem out of place. However, at a university, the development of projects on a reliable, well-supported datacenter platform can become an attractive alternative to local labs for both research teams and FPGA course instructors.

Nevertheless, deploying FPGAs in HPC clusters comes with its own set of challenges and impediments as illustrated in Chen et al. [4] and for scalability, FPGAs need to be connected to the data center network directly as shown in Weerasinghe et al. [13]. Another barrier to using FPGAs and similar heterogeneous accelerators is that their workflow and software stack is significantly dissimilar and non-standardized when compared to GPU or CPU workflows. Furthermore, depending on the vendor, these accelerators work with different software stacks as well, in spite of them implementing a similar workflow methodology.

Broadly speaking, development flow on an FPGA can be classified as:

- **Register-Transfer Level (RTL) Design Development**: This is the classical development flow on an FPGA where a researcher writes hardware description language (HDL) code using Verilog or VHDL which is synthesized onto the FPGA through an appropriate development suite such as Xilinx Vivado or Intel Quartus Prime. HDL Design requires greater expertise and is more relevant to VLSI engineers and microarchitecture researchers. FPGAs facilitate ASIC prototyping by serving as a platform to design, test, and optimize custom integrated circuits. This development workflow can possibly exploit cloud FPGAs with the important caveat that the final test on the FPGA of a compiled design bitstream may not be viable; an FPGA board often needs a complete power cycle to update the onboard design, and power cycle privileges may not be granted to every user.
- **C/C++ Application Development**: With advancing technologies, FPGA programming has been shifting to a paradigm similar to that of GPU programming. This development workflow splits the design into two code files: the kernel code that runs on the FPGA and the host code that runs on the host CPU and leverages the kernel code. High Level Synthesis (HLS) compilers convert C/C++ code into FPGA designs, thereby improving the approachability of FPGA programming. A user would only need a base understanding of C code to get started with this workflow, and the complete

https://doi.org/10.22369/issn.2153-4136/16/2/5

flow consists of a minimum of four steps to get a design running on the FPGA. These can be specified as:
(1) Compile the host code using any C/C++ compiler.
(2) Compile the kernel code into an HLS package.
(3) Link the HLS package into a binary file.
(4) Execute the compiled host code with the binary file.
This flow is easily deployable on HPC clusters, and applications made using this workflow do not require any heightened privileges for the user on the FPGA nodes.

- **Hybrid Development**: The aforementioned development flows can also be combined at various stages. Designs made in HDL code can be packaged into HLS modules and further linked to a design binary, thereby creating a combined workflow where traditional RTL projects can be deployed on HPC clusters with the C/C++ development flow without having to provide users increased administrator privileges on the node.

The tools required for these workflows vary, resulting in a suite of software tools, drivers, and resources available to the user for a comprehensive testbed environment. The development methodology is modular with multiple intermediate results in the design process, thereby implying that all stages need not be compiled, executed, and tested on the same resource, resulting in an inherent parallelism available to the workflow itself.

2 FPGA AS A SERVICE (FAAS)

With the increasing heterogeneity of HPC systems, FPGA resources have become popular as an extension to the pool of available resources offered by cloud services. Project Catapult by Microsoft and Amazon's AWS EC2 F2 instances are two examples of this growing trend. As mentioned earlier, the inherent heterogeneity of the FPGA node configuration requires a suite of software resources and modules to go from design to deployment. Creating and sustaining this development environment as a resource for a multitude of online users has its own challenges, but an added incentive to deploying such an environment on the cloud is that these resources are often self-contained suites that offer a wider breadth of features. Furthermore, they are designed in a modular fashion, enabling system administrators to upgrade or downgrade these integrated development environments (IDEs) as required. Therefore, the primary classification that inhibits cross-compatibility between IDEs generally boils down to the choice of software offerings. Intel and AMD are the prominent vendors in this field, as of this writing, and hence, the choice is between them as their FPGA software products are functionally similar.

In terms of performance benchmarks, the FPGA as a Service (FaaS) model has been evaluated to show better performance for compute-intensive workloads compared to CPU and GPU implementations while also leveraging relatively less power consumption as evaluated in Perepelitsyn et al. [11].

2.1 Compute and Software Resources

ASU's Sol has FPGA nodes available through the SLURM scheduler. However, while the OpenCL workflow from design to deployment is straightforwardy available to all users, full re-programmability of the cards and driver access is restricted by administrator permissions to prevent users from accidentally bricking the hardware. Providing elevated permissions to FPGA researchers is one possible solution for this. Bare metal servers are the ideal method for deploying FPGA resources on the cloud, as they allow researchers to fully exploit the re-programmability of FPGA hardware, and the possibility of such special access nodes on Sol is being explored.

Currently, Sol has two FPGA nodes with the following FPGAs:

(1) Alveo U280 with an AMD UltraScale+ XCU280 FPGA and 8GB of HBM2 memory [1].
(2) Bittware 520N-MX with an Intel Stratix 10 FPGA and 16GB of HBM2 memory [3].

Since the two FPGAs are from separate vendors, the requisite software environment for development on either can be easily provisioned for each, with AMD Xilinx Vivado being one of the available development environments and Intel Quartus Prime as its complement.

2.1.1 *AMD Xilinx Resources*.

(1) *AMD Xilinx Vivado*: Vivado is a comprehensive suite for the traditional FPGA workflow using HDL, and it supports Verilog, SystemVerilog, and VHDL. On Sol, Vivado 2022.1 Standard Edition is available as a module which can be loaded on any node and used accordingly. Vivado is a comprehensive IDE for RTL design, and most of its components can be used from any node on Sol to work from RTL design to bitstream generation. All boards supported in the Standard Edition are available for development with the edition on Sol. Consequently, within the cloud working environment, Sol offers the capability of offloading the design and testing of experimental VLSI architectures in a datacenter environment to researchers, thereby reducing the need for local machines required to run the same software.
While, with some effort, RTL design does allow users to use the full reconfiguration of an FPGA, significant domain expertise is necessary to competently exploit the resources available on an FPGA. ASU's Sol allows researchers to design, develop, and generate their custom architecture bitstreams with Vivado on any node, but deploying the bitstream on the Alveo U280 is restricted due to the partial reconfigurability feature of the board. As such, any design would require a debug core to be interfaced into them so as to not brick the Alveo U280 once its onboard design memory is updated. The card also needs a power cycle to update itself once a bitstream has been deployed, and, therefore, as a security measure, full reconfigurability is not granted to all users.
Since Vivado is compatible with various other FPGA design platforms, it is always plausible for researchers to simply use the software on any node on Sol to develop a design for any other FPGA of their choice. Any design files can always be easily transferred between the cloud and another local environment.
(2) *AMD Xilinx Vitis*: Vitis is an IDE designed for working with HLS using C/C++. The Vitis workflow involves splitting the code into the CPU/Host component and the Kernel/FPGA component, akin to specifying the CPU and GPU kernels

while working with HIP or CUDA code. The host code is compiled using a g++ compiler while the kernel code is compiled using Vitis v++ which translates the C/C++ implementation into an FPGA binary. The v++ compiler maps the code to a specific platform, and hence a .xpfm or .xsa platform file must be included when running the compiler command. This platform file can be the provided default for a specific board, or it can be custom made through Vivado. Vitis has various sub-components as well which can be used for timing analysis, code debugging, platform generation, and more. Figure 1. shows how the CPU and FPGA codes interact with each other.

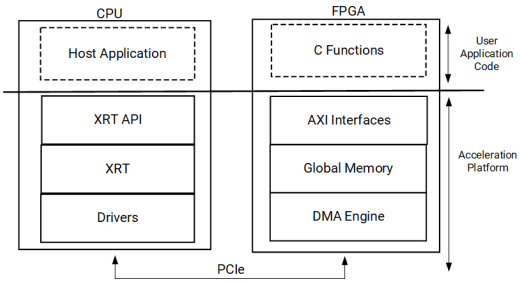

Figure 1: CPU-FPGA Code Interaction in Vitis [6]

(3) *Xilinx Runtime (XRT)*: XRT is a software stack composed of a mix of userspace and kernel driver components. It contains the collection of API keys that allows the host system to successfully communicate with PCIe based FPGA accelerators. The tools provided in this software stack not only help in synthesizing code to be deployed on the FPGA but also allow users to monitor and test the FPGA directly through a bash shell without having to go through Vivado or Vitis. Figure 2. shows the software stack structure of the XRT libraries.

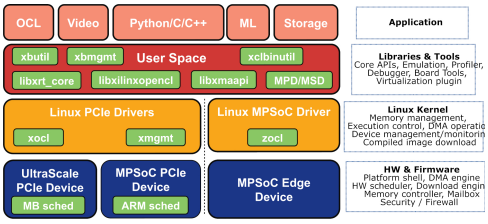

Figure 2: XRT Software Stack [14]

Figure 3. shows the software stack for XRT on Alveo based platforms such as the Alveo U280.

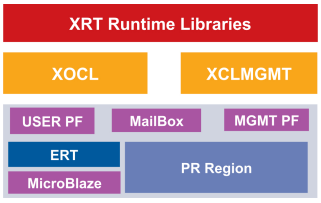

Figure 3: PCIe Stack for Alveo Platforms [14]

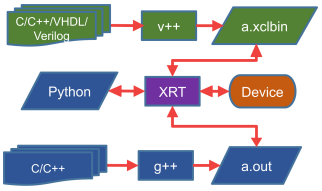

Figure 4: Compilation flow with XRT [14]

Figure 4. shows the compilation and execution workflow when working with XRT libraries to run kernels on the FPGA. It illustrates how the host code working on the host system CPU interacts with the compiled kernel binary deployed on the FPGA platform.

2.1.2 *Intel Altera Resources.*

(1) *Intel Quartus Prime*: Intel Quartus Prime is the comprehensive FPGA development suite available for Intel/Altera FPGAs. On Sol, there are multiple versions of Quartus Prime that are available, ranging between the Standard, Pro, and Lite edition. The recommended version is Intel Quartus Prime Pro 23.4 as that is the version that works ideally with the Bittware 520N-MX hardware that is available on Sol. Quartus Prime is similar to Vivado in that it boasts a comprehensive suite of tools, such as Quartus Programmer, Platform Designer, Timing Analyzer etc., which are designed to accommodate the many possible aspects of designing on Intel FPGAs. Unlike Vivado, which has separated off the Xilinx HLS components onto Vitis, Quartus Prime comprehensively includes the Altera HLS Compiler which acts as the working suite for compiling higher level code into its RTL equivalent for Intel FPGAs. The HLS Compiler is available in the Pro and Standard editions of Quartus Prime with the primary difference being the devices supported by those editions. Questa is also provided with Quartus Pro, which serves as an effective HDL simulator for behavioral verification of RTL modules.

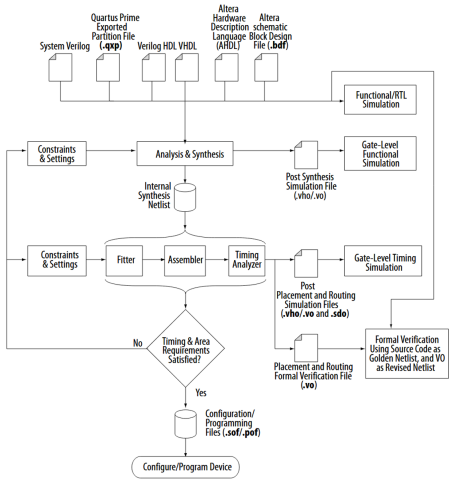

Figure 5: Intel FPGA Workflow [7]

Figure 5. shows a comprehensive workflow and the various file types involved in different stages of design and compilation for an Intel FPGA. While dissimilar in some aspects, the overall design philosophy and process is quite reminiscent of working with Xilinx FPGAs and thereby suggests that, while the tools may be different, RTL or C++ designs could possibly be ported from one vendor platform to another as long as the hardware resources supported the design.

(2) *Intel FPGA SDK for OpenCL*: OpenCL support for Intel FPGAs is provided through this package made available by default on the Bittware node on Sol. The Altera OpenCL (AOCL) utility can diagnose, program, and validate the FPGA, and it provides easy access without having to resort to working through Quartus. While AOCL has become a legacy tool as of writing this article, it is still readily compatible with many HLS design tools that allow extended programmability of the Intel FPGA. It is analogous to the XRT stack on Xilinx FPGAs and offers similar utilities for real-time hardware diagnostics.

(3) *Intel OneAPI*: Intel's OneAPI is a unified programming framework to write code for Intel FPGAs, CPUs, and GPUs. It is the current framework for programming Intel FPGAs through HLS built around C++ with SYCL. Multiple versions of OneAPI compilers and libraries are available on Sol with OneAPI 2023.2.1 being the most recent one. The OneAPI base toolkit is available as a module while its FPGA add-on is pre-installed on the Intel FPGA node, allowing for a seamless workflow with OneAPI tools.

Overall, both the AMD Xilinx and the Intel Altera resources provide tools for implementing the same design workflow and philosophy. Both vendors provide tools to facilitate RTL designs with Verilog and VHDL, and they also provide for the HLS design workflow which is much simpler and easier to learn with its similarities to GPU kernel programming.

2.2 Workflow

As mentioned above, modern FPGA programming is gradually shifting towards, and becoming unified with, GPU programming, often even using similar libraries, tools, and compilers. While the intricacies may be vendor specific, the overall design philosophy tends to remain the same, with the primary differences arising with the choice of coding language. In this section, we will attempt to codify the workflow based on HDL and C++ to create an abstract but comprehensive route from initial design to final deployment.

2.2.1 ***RTL Development Workflow.***

(1) *HDL Block Design*: The preliminary stage of any RTL design project begins with an abstract overview of the many functional modules to be implemented and how they should be interconnected and controlled. This stage involves a very high-level overview of the possible functional blocks, the state machines required for each block or groups of blocks, and the pipeline stages that will be incorporated into the design. These designs are then described in an appropriate HDL as per the discretion of the user. This design is then verified functionally and behaviorally with an associated testbench to ensure that the design is working as intended.

(2) *Synthesis*: Using an appropriate compiler, the HDL code prepared in the earlier stage is used to formulate a gate-level representation of the same code. This *gate level netlist* is a low level design using only primitive circuit components, representing the code with simple logic gates. It is then further optimized for timing, power, and performance according to the constraints and requirements as defined by the user. Functional and behavioral verification are usually executed using the earlier testbench once again to ensure that the translation from RTL code to the gate-level netlist did not create any behavioral anomalies.

(3) *Implementation*: Once the gate-level netlist is compiled and functional verification is done, the next step is to map these primitive components to the physical resources that are available on the FPGA. This step often uses a platform file or a board support package which contains the configuration and details of various blocks available on the FPGA for which the design is being implemented. *Place and Route* is the primary phase of this step where;
   - *Place:* Logic cells, required by the design, are placed at appropriate points on the FPGA board mapped by the compiler and,
   - *Route:* The connections for the placed logic cells are mapped to each other to form a complete circuit to implement the overall design.

   The placement and routing of the circuit is done in congruence with an appropriate platform and constraints files. Various verification methods pertaining to Design Rule Checks (DRC) are carried out in this phase as well to ensure that the designed circuit is in accordance with the physical resources available.

(4) *Timing Analysis*: Static Timing Analysis (STA) is the standard method for timing closure which analyzes the propagation delay through all relevant timing paths. In this context, slack is the margin by which the timing is met or violated. Setup slack indicates that the signal is not propagating too slowly, while hold slack signifies that the signal is not arriving too early. Positive slack implies that the circuit is meeting the timing constraint and that a faster clock can be achieved depending on the magnitude of the slack, while negative slack implies a timing violation. Achieving the highest operating frequency (lowest clock period) while maintaining timing closure is the ultimate goal of this step.
   Implementation and Timing Analysis are repeated iteratively until the timing violations in the circuit have been eliminated. Once all relevant timing paths have met timing and constraint requirements the design is ready to be exported to a bitstream.

(5) *Hardware Validation*: The implemented design is then exported to a device-specific binary configuration format which is used to program the physical hardware. Verification of the design is done by testing out the programmed FPGA in a real-time environment with an appropriate testbench. This step includes various design-specific tests to ensure that the hardware is performing well under various conditions and that there are no discrepancies between the simulated and implemented design and the final design on hardware.

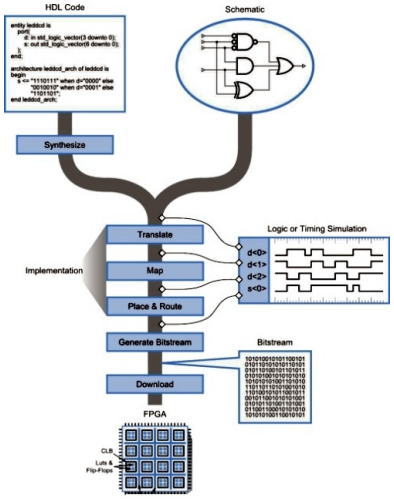

Figure 6: RTL Development Workflow [5]

Figure 6. shows an abstract and comprehensive overview of the aforementioned RTL workflow.

2.2.2 *C/C++ Application Workflow*.

(1) *C/C++ Algorithm Design*: First and foremost, the overall algorithm for implementing a design is formulated. This involves separating out the steps of the algorithm which can be computed on a host machine (usually with a multi-purpose CPU) and the steps that can be computed as an FPGA kernel. Depending upon the complexity and depth of the design to be implemented, the interaction between the host and FPGA kernel can range from simple function calls to more complex multi-stage data transfer and streaming operations. Once the overall algorithm is codified, functional verification is carried out with an appropriate testbench to ensure that the code is working as needed. This results in two primary components of the code:
- The host code running on the CPU and,
- The kernel code running on the FPGA

The kernel code is optimized further and then later recompiled to a full binary configuration which is deployed onto the FPGA in the proceeding steps.

(2) *HLS Optimization*: Once the algorithm is verified functionally, various HLS directives and pragmas are employed to optimize the kernel code for efficient hardware implementation. These optimizations include selecting appropriate data types (fixed-point vs. floating-point), loop pipelining and unrolling, array partitioning, dataflow optimizations, and interface specifications. The goal is to maximize throughput while minimizing resource utilization and meeting timing constraints.

(3) *RTL Generation and Co-Simulation*: After the functional verification and optimization of the kernel and host algorithm is done, the kernel code is compiled into an RTL module description. This is a synthesizable RTL representation of the algorithm which is useful in describing the datapath, control logic, and interfaces in a more hardware-centric manner. Co-simulation is then performed using the software testbenches to verify that the generated RTL produces identical functional behavior to the software reference, ensuring no behavioral discrepancies were introduced during the HLS compilation process.
Once it is verified that there are no functional anomalies, power, performance, and area analysis of the design can be conducted. HLS optimization and RTL generation and co-simulation can be carried out iteratively until the desired performance is achieved. Thus, these two steps act as a design exploration stage of the algorithm where performance bottlenecks, timing estimates, pipelining techniques, and memory partitioning and data flow can be tested and evaluated before the implementation of the full design.

(4) *Export and Testing*: Once the kernel module has been sufficiently tested and optimized, it can be compiled into a complete binary configuration for the FPGA. It is noteworthy that binaries from the HLS workflow are not interchangeable with those from the RTL workflow. HLS-generated binaries typically include additional metadata, such as kernel interfaces, scheduling, and pipelining information, providing more insight into the implemented design. In contrast, RTL workflows provide direct control over hardware implementation details. While HLS workflows offer higher-level abstraction and include runtime frameworks that facilitate simpler system integration, the resource efficiency of either approach depends primarily on the design quality and optimization effort rather than the source methodology itself.
Generation of the binary in this workflow takes a significant amount of time, typically above 3 hours. This is often because the HLS compilers carry out similar steps as in the RTL workflow, such as place and route, timing analysis, area reports, and resource optimizations, effectively going through the implementation and timing analysis steps repeatedly until an optimal design is achieved. The design summary and performance reports are also generated and exported during this step, which can be used for deeper analysis of the generated binary.
After the binary has been generated, it can be leveraged by the host code to be deployed onto the FPGA. Therefore, the host code can be a testbench to run simulations on the generated design to verify that functionality and performance have been preserved over the whole process, or it can be the primary algorithm interface to implement the overall design on the FPGA.

3 DISCUSSION

- **Cross-Compatibility of Workflows**: The workflows elaborated upon in Sections 2.2.1 and 2.2.2 are meant to be abstract overviews of the whole process. Each step is often self-contained and with various possible implementations. With the increasing diversity of FPGA tools and resources being made available to researchers, cross-compatibility between workflows is also becoming more straightforward. AMD Xilinx Vivado supports this cross-compatibility by

virtue of the IP Packaging Tool built into it, which can package an RTL design into a kernel module and which can be compiled into an application binary through Vitis HLS. While this approach can be employed to port older and simpler RTL designs to HLS binaries, the complexity of porting these designs mirrors the complexity of the RTL design itself.

- **Open-Source Resources**: Apart from the AMD and Intel tools mentioned in Sections 2.1.1 and 2.1.2, there are also many open source tools available to develop on FPGAs as well. Two open source tools deployed on Sol are OpenHLS [10] and ScaleHLS [16]. Both of these libraries are available as mamba environments on Sol. OpenHLS is a project that translates PyTorch neural network models to synthesizable RTL code and supports the Alveo U280 on the Sol cluster. The ScaleHLS project aims at compiling PyTorch code to its HLS C++ equivalent compatible with AMD Vitis.

- **Pre-Compiled Binaries**: To help researchers ease into the development workflow on Sol FPGAs, some pre-compiled projects have been made publicly available. These include:
  - Simple vector addition projects for the AMD and Intel nodes.
  - Custom and original implementations of the traveling salesman problem based on AMD's Vitis Tutorials [15].
  - An OpenCL implementation of a 2D FFT Accelerator for the Bittware 520N-MX based on Bittware's white paper [2].

  These projects contain the compiled kernel binaries, summary reports of the design, and host binaries to easily run the designs on the FPGA.
- **Run Time Benchmarking**: An important factor to consider when providing FPGA resources for the research community is program run time and card utilization. Table 1 presents some initial run time benchmarking and utilization for some basic programs.
  - The vector addition program simply adds two vectors of size 65536 to each other.
  - The prime number program calculates all prime numbers between 0 and 4095 and returns them in an output vector.
  - The triangular numbers program calculates the first triangular number to have over 500 divisors.
  - The second triangular numbers program was an attempt to solve the problem with a different method. It involved storing all divisors in an array and then iterating through the array when checking new numbers. This led to incredibly slow performance and the program did not finish in emulation or on hardware, even after eight hours.

	Kernel linking time (emu.)	Kernel Linking time (HW)	Comp. time (emu.)	Comp. time (HW)	Util.
Vector add. (65536 array)	13m 5s	1h 36m 49s	115.35s	9052μs	0%
Prime #s (4096 array)	13m 2s	1h 59m 0s	543s	5634μs	0%
Triang. #s	0h 41m 43s	4h 24m 11s	12849s	102524μs	~1.5%
Triang. #s (Array method)	0h 41m 48s	1h 22m 16s	DNF	DNF	N/A

Table 1: Some performance numbers on the Alveo U280 with the XRT Compilation Flow

4 RESULTS

To summarize, FPGAs have emerged as reconfigurable accelerators for specialized workloads in datacenters. They offer a unique balance between performance, flexibility, and energy efficiency tailored to specific applications. They perform well on compute-intensive tasks where parallelization and custom datapaths provide significant advantages. Despite these advantages, FPGAs have faced adoption challenges with long development cycles being a major hurdle. However, recent methods such as HLS and frameworks like XRT and OneAPI have been enabling wider deployment of FPGAs by allowing researchers to work with a wider arsenal of programming languages.

The codified workflows presented above offer a consistent platform to implement different projects as well as to reproduce results for similar projects. The workflows offer an accessible on-ramp for the onboarding of researchers onto FPGA projects on Sol and the versatility of the workflow imparts the capability for auto-tailorization of the flow with continued development.

The primary dichotomy between the workflows, as of this writing, arises not because of the overall development methodology with C/C++ and RTL, but rather due to the difference in vendors of the FPGAs. Open source tools are democratizing this barrier slowly but pragmatically, staying within one vendor's development environment does not seem to hamper the interchangeability of workflows significantly.

The datacenter FPGA landscape continues evolving with advances in memory integration (HBM2 and above), higher-speed interfaces (PCIe 5.0, CXL), multi-processor platforms (AMD Versal and Zynq MPSoCs), and improved development tools. As workloads become more specialized and energy efficiency becomes increasingly critical, FPGAs are positioned to play an expanding role in heterogeneous datacenter architectures, complementing CPUs and GPUs in optimized computing solutions.

ACKNOWLEDGEMENTS

The authors acknowledge Research Computing at Arizona State University for providing resources that have contributed to the results reported within this paper.

REFERENCES

[1] Advanced Micro Devices, Inc. 2024. Alveo U280 Data Center Accelerator Card User Guide. https://docs.amd.com/r/en-US/ug1314-alveo-u280-reconfig-accel.

[2] BittWare. 2021. *Accelerating 2D FFTs Using HBM2 and oneAPI on Stratix 10 MX*. Technical Report. BittWare. https://www.bittware.com/resources/hbm2-2d-fft-oneapi/ White Paper.

[3] BittWare, Inc. 2024. 520N-MX FPGA Accelerator Card. https://www.bittware.com/products/520n-mx/.

[4] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. 2014. Enabling FPGAs in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers (CF '14)*. Association for Computing Machinery, New York, NY, USA, Article 3, 10 pages. https://doi.org/10.1145/2597917.2597929

[5] Bruno Da Silva, An Braeken, and Abdellah Touhafi. 2018. FPGA-Based Architectures for Acoustic Beamforming with Microphone Arrays: Trends, Challenges and Research Opportunities. *Computers* 7, 3 (2018). https://doi.org/10.3390/computers7030041

[6] Héctor Gutiérrez Arance, Luca Fiorini, Alberto Valero Biot, Francisco Hervás Álvarez, Santiago Folgueras, Carlos Vico Villalba, Pelayo Leguina López, Arantza Oyanguren Campos, Valerii Kholoimov, Volodymyr Svintozelskyi, and Jiahui Zhuo. 2025. Porting MADGRAPH to FPGA Using High-Level Synthesis (HLS). *Particles* 8, 3 (2025). https://doi.org/10.3390/particles8030063

[7] Intel Corporation. 2024. *Intel® Quartus® Prime Pro Edition User Guide*. Intel Corporation. https://www.intel.com/programmable/technical-pdfs/qpp-ugs.pdf Document ID: 766292.

[8] Douglas Jennewein et al. 2023. The Sol Supercomputer at Arizona State University. In *Practice and Experience in Advanced Research Computing (PEARC '23)*. Association for Computing Machinery, New York, NY, USA, 6. (in press).

[9] Joshua Lant, Javier Navaridas, Mikel Luján, and John Goodacre. 2020. Toward FPGA-Based HPC: Advancing Interconnect Technologies. *IEEE Micro* 40, 1 (2020), 25–34. https://doi.org/10.1109/MM.2019.2950655

[10] Maksim Levental, Arham Khan, Ryan Chard, Kazutomo Yoshii, Kyle Chard, and Ian Foster. 2023. OpenHLS: High-Level Synthesis for Low-Latency Deep Neural Networks for Experimental Science. arXiv:cs.AR/2302.06751 https://arxiv.org/abs/2302.06751

[11] Artem Perepelitsyn and Vitaliy Kulanov. 2025. Methods of Deployment and Evaluation of FPGA as a Service Under Conditions of Changing Requirements and Environments. *Technologies* 13, 7 (2025). https://doi.org/10.3390/technologies13070266

[12] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 13–24. https://doi.org/10.1109/ISCA.2014.6853195

[13] Jagath Weerasinghe, Francois Abel, Christoph Hagleitner, and Andreas Herkersdorf. 2015. Enabling FPGAs in Hyperscale Data Centers. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*. 1078–1086. https://doi.org/10.1109/UIC-ATC-ScalCom-CBDCom-IoP.2015.199

[14] Xilinx. 2021. Xilinx Runtime (XRT) Documentation. https://xilinx.github.io/XRT/2021.1/html/index.html

[15] Xilinx. 2022. Vitis-Tutorials. https://github.com/Xilinx/Vitis-Tutorials

[16] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2022. Scalehls: A new scalable high-level synthesis framework on multi-level intermediate representation. In *2022 IEEE international symposium on high-performance computer architecture (HPCA)*. IEEE, 741–755.