ForgeBench: A Machine Learning Benchmark Suite and Auto-Generation Framework for Next-Generation HLS Tools

Andy Wanna[1*], Hanqiu Chen[1*], Cong (Callie) Hao[1]
[1] *School of Electrical and Computer Engineering* - Georgia Institute of Technology, [*]*Denotes Equal Contribution*
Email: {awanna3, haniqu.chen, callie.hao}@gatech.edu

***Abstract*—Although High-Level Synthesis (HLS) has attracted considerable interest in hardware design, it has not yet become mainstream due to two primary challenges. First, current HLS hardware design benchmarks are outdated as they do not cover modern machine learning (ML) applications, preventing the rigorous development of HLS tools on ML-focused hardware design. Second, existing HLS tools are outdated because they predominantly target individual accelerator designs and lack an architecture-oriented perspective to support common hardware module extraction and reuse, limiting their adaptability and broader applicability. Motivated by these two limitations, we propose ForgeBench, an ML-focused benchmark suite with a hardware design auto-generation framework for next-generation HLS tools. In addition to the auto-generation framework, we provide two ready-to-use benchmark suites. The first contains over 6,000 representative ML HLS designs. We envision future HLS tools being architecture-oriented, capable of automatically identifying common computational modules across designs, and supporting flexible dataflow and control. Accordingly, the second benchmark suite includes ML HLS designs with possible resource sharing manually implemented to highlight the necessity of architecture-oriented design, ensuring it is future-HLS ready. ForgeBench is open-sourced at https://github.com/hchen799/ForgeBench.**

***Index Terms*—High Level Synthesis, Benchmarks, Machine Learning**

I. Introduction

While High-Level Synthesis (HLS) has gained increased traction in hardware design, it has not become the mainstream yet because of two key limitations. ❶ **Outdated HLS benchmarks:** Existing hardware design benchmarks are not ML-ready and HLS-oriented, preventing the effective development of HLS tools for ML hardware. For instance, MachSuite [1] and Rosetta [2] comprise outdated algorithms and lack coverage of emerging ML models and applications. Other C benchmarks, such as Rodinia [3], designed for GPU benchmarking, and PolyBench [4], intended primarily for software compilers, have limited applicability to evaluating HLS tools. ❷ **Outdated HLS tools:** Although existing HLS tools [5–9] have significantly improved hardware design efficiency for accelerators, they struggle to design *architectures*. These tools synthesize each input design into a unique accelerator through static dataflow analysis tailored to specific programs. However, with the rapid evolution and diversity of ML applications, new individual hardware accelerators are developed for each new model, leading to significant hardware resource redundancy. This is because different ML models often share substantial computational components. For example, LLaMa [10] and GPT [11] can share the computation kernels of attention and General Matrix Multiplication (GEMM), differing only slightly in activation functions. Therefore, we envision that future HLS tools should be capable of automatically identifying and extracting common computational modules across different hardware accelerators for resource sharing, supporting flexible dataflow and control. Part a) of Fig. 1 depicts a high-level comparison of the existing HLS tools and how a modular (architecture-oriented) HLS tool may synthesize hardware. An example comparing standard HLS designs to a modularized implementation is highlighted with part b) of Fig. 1 with a simple nested loop.

Motivated by these two limitations, in this paper, we propose *ForgeBench*, an ML-focused benchmark suite designed specifically for next-generation HLS tools, together with a user-friendly and highly extensible design auto-generation framework, with details shown in Fig. 1. Our key contributions are summarized as follows:

- An HLS design generation framework enabling users to rapidly generate C/C++ HLS compatible hardware designs and obtain synthesis and implementation reports. The framework is highly extensible, facilitating the integration of new designs into the benchmarking suite.
- A ML-representative benchmark suite, containing over 6,000 different HLS designs, to test and evaluate the current C/HLS tools.
- The design generation framework is future-HLS-ready, including a benchmark suite with manually determined shared modules highlighting the necessity of HLS for architecture design.

II. ForgeBench Overview

ForgeBench is an automated, user-friendly framework that rapidly generates diverse modular HLS designs with minimal manual coding. It incorporates a highly extensible C/C++ template library featuring common ML kernels, enabling quick retrieval and assembly of functional modules. The generated designs serve as robust benchmarks for current and future ML-oriented HLS tools. ForgeBench also provides two ready-to-use benchmarks: ❶ over 6,000 ML test cases generated automatically via a provided Python script, ❷ modular HLS test cases, including HLS designs automatically generated by the tool and the unified designs with manually extracted shared modules. An overview of ForgeBench is depicted in Fig. 1.

A. Automatic and Customized Design Generation Workflow

As shown in part c) of Fig. 1, ForgeBench provides a Python-based HLS design code generator and supports parallel synthesis and implementation of commercialized HLS flows across multiple threads. It is also integrated with an extensible code library of HLS-oriented C/C++ kernels representing commonly used ML modules, offering customizable code templates for users.

In ForgeBench, users provide customized configurations for each HLS design through JSON files, which can be created manually or generated automatically using our provided Python scripts. Each JSON file corresponds to a distinct HLS design featuring different dataflows and ML module call sequences. In the JSON file, users specify the global BRAM and off-chip DRAM (including names and dimensions), as well as the top-level function interfaces for data communication. Users also define the sequence of ML module calls, detailing inputs, outputs, and specific configurations of each module call. As illustrated in the convolution example in Fig. 1, module call configurations include loop boundaries, input feature dimensions (height, width, channels), and convolution kernel size. Additionally, the JSON file specifies synthesis configurations, such as clock period, top function name, data types, and the chosen HLS workflow {CSIM, CO-SIM, Synthesis, Implementation}.

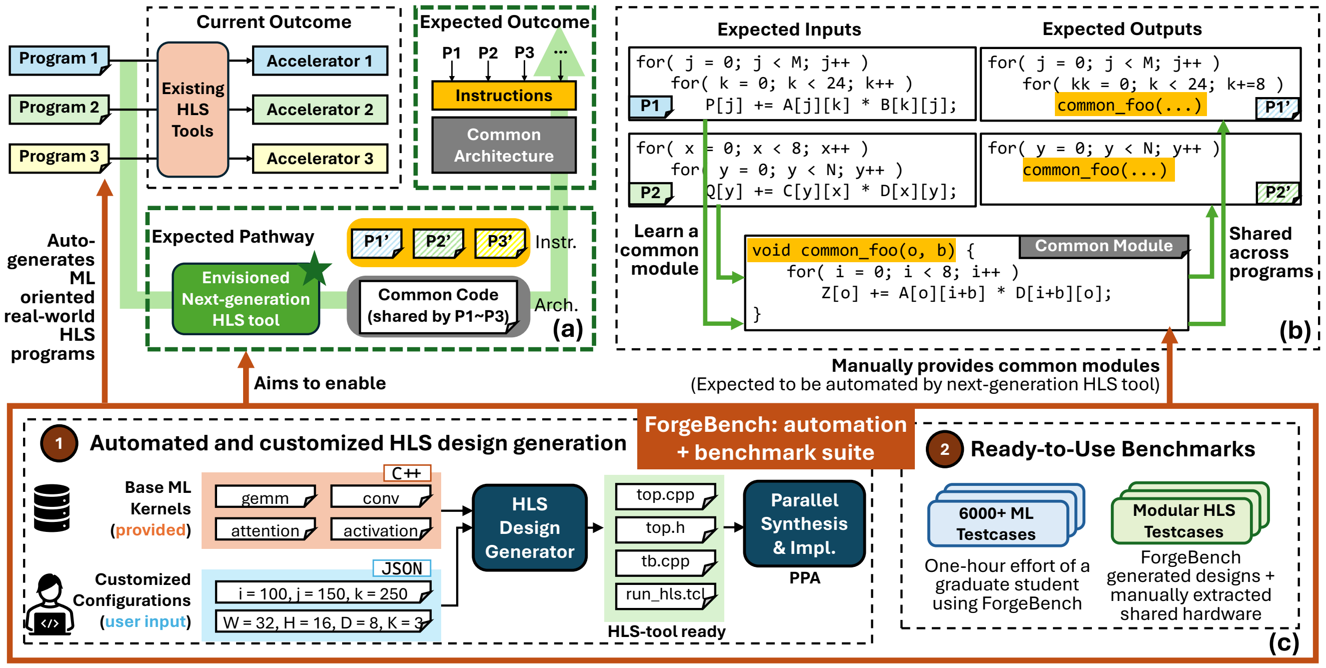
The HLS design generator takes user-defined configurations from JSON files as inputs, utilizes predefined ML module code templates, and automatically generates complete and HLS-ready C/C++ designs. ForgeBench then performs parallel synthesis and implementation across multiple threads, maximizing computational resource utilization and minimizing the execution time. Upon completion, ForgeBench automatically produces comprehensive performance, power and area (PPA) reports for user analysis. While currently, ForgeBench supports the Vitis HLS flow [9] for Xilinx FPGA designs, it can be easily extended to seamlessly support other HLS tool flows.

B. Supported ML Models and Operator Kernels

We support four different types of operators and three relevant ML models in ForgeBench.

1. GEMM & MLPs

ML Relevance – Linear Layers, comprising Generalized Matrix Multiplication (GEMM) operations, are fundamental to ML models ranging from MLPs to DNNs and LLMs. Thus, it is critical to benchmark their performance in HLS tools. The baseline GEMM operation generated by ForgBench is a simple nested 'i-j-k' loop. We also include operations



Current Outcome
Program 1
Program 2
Program 3
Existing HLS Tools
Accelerator 1
Accelerator 2
Accelerator 3
Expected Outcome
P1 P2 P3 ...
Instructions
Common Architecture
Expected Pathway
P1' P2' P3'
Instr.
Envisioned Next-generation HLS tool
Common Code (shared by P1~P3)
Arch.
(a)
Auto-generates ML oriented real-world HLS programs
Aims to enable
Expected Inputs
Expected Outputs
Learn a common module
Common Module
Shared across programs
(b)
Manually provides common modules
(Expected to be automated by next-generation HLS tool)
ForgeBench: automation + benchmark suite
1 Automated and customized HLS design generation
Base ML Kernels (provided)
C++
gemm
conv
attention
activation
Customized Configurations (user input)
JSON
i = 100, j = 150, k = 250
W = 32, H = 16, D = 8, K = 3
HLS Design Generator
top.cpp
top.h
tb.cpp
run_hls.tcl
HLS-tool ready
Parallel Synthesis & Impl.
PPA
2 Ready-to-Use Benchmarks
6000+ ML Testcases
One-hour effort of a graduate student using ForgeBench
Modular HLS Testcases
ForgeBench generated designs + manually extracted shared hardware
(c)

```
for( j = 0; j < M; j++ )
   for( k = 0; k < 24; k++ )
      P[j] += A[j][k] * B[k][j];
```
P1

```
for( x = 0; x < 8; x++ )
   for( y = 0; y < N; y++ )
      Q[y] += C[y][x] * D[x][y];
```
P2

```
for( j = 0; j < M; j++ )
   for( kk = 0; kk < 24; k+=8 )
      common_foo(...)
```
P1'

```
for( y = 0; y < N; y++ )
   common_foo(...)
```
P2'

```
void common_foo(o, b) {
   for( i = 0; i < 8; i++ )
      Z[o] += A[o][i+b] * D[i+b][o];
}
```

Fig. 1: Proposed benchmarking framework enabling future HLS tools for architecture design.

for vector-matrix multiplications and a dot product. All these operations can optionally be generated with a bias term. Due to the prominence of GEMM operations, they are rarely implemented as naive nested loops in ML accelerators. Standard optimizations seen in [12–15] include unrolling/tiling to compute the operation in parallel and modifying the loop order for better memory access patterns. Therefore, ForgeBench supports any valid loop re-ordering of 'i-j-k', with different unroll parameters for each. Optionally, any multiplication unit can be implemented inline instead of as a function. With these parameters, the benchmark can generate a myriad of vector/matrix multiplications with control over low-level hardware implementations, representative of ML accelerator designs.

Architecture Oriented HLS – There is a significant opportunity for hardware reuse across GEMM and vector operations. For example, GEMMs of different dimensions can be computed with a shared tile. Additionally, GEMMs can be composed with simpler vector/matrix multiplications or dot-product units. Reordering the computation loops does not mathematically affect the GEMM operator, enabling them to use the same shared module. This variety of hardware reuse categories forms an ideal test suite to benchmark future modular HLS tools.

2. *Convolution & DNNs*

ML Relevance – Convolution and batch normalization (BatchNorm) are two fundamental modules in DNNs [16–19]. ForgeBench provides base templates for both, enabling the creation of HLS designs for diverse DNN variants. ForgeBench supports configuring these modules with customizable numbers of channels and dimensions for input/output feature maps. The convolution operation is flexible with variable kernel sizes, padding, and stride, with an optional bias term. ForgeBench also includes a grouped convolution kernel as seen in DNNs such as MobileNet [18, 20, 21], intended to reduce computational complexity over the traditional.

Architecture Oriented HLS – DNN hardware designs contain significant opportunities for reusing functional modules. These include tiling strategies for convolution and BatchNorm computations, allowing small kernels to be effectively reused. Additionally, different models can share identical convolution blocks. For example, both ResNet-18 and ResNet-34 incorporate 3×3 convolutions with 64 channels and identical input feature dimensions, enabling the reuse of these functional blocks [16].

3. *Transformers & LLMs*

ML Relevance – LLMs are built with transformer architectures, relying on attention, normalization, and embedding encoding operations. These operators can all be configured with different sequence lengths and hidden dimensions of input language tokens in ForgeBench. Modern transformer architectures employ diverse attention mechanisms based on the traditional multi-head attention described in GPT [11]. As a result, multi-head attention in ForgeBench has parameters for the number of heads, number of head groups as seen in LLaMA [10], and an optional variable length sliding window motivated by Mistral [22]. Additionally, we support both layer normalization (LayerNorm) used in the GPT [11] models and the root mean square normalization (RMSNorm) employed by modern models like LLaMA [10]. Furthermore, modern models increasingly use rotary position embeddings (RoPE) [10, 22, 23]; hence, a corresponding template is included in the framework.

Architecture Oriented HLS – There are substantial hardware sharing opportunities across LLM operations. First, the tiling strategy in multi-head attention computation enables the reuse of smaller attention modules with fewer heads to construct larger ones. Second, each attention module can be decomposed into smaller computational modules, such as the GEMM and SoftMax modules, facilitating their reuse across different designs and architectures. Finally, similar to DNNs, various LLMs incorporate common computational blocks. For instance, despite differences in their attention mechanisms, GPT and LLaMA utilize the same feed-forward network, enabling large-scale module reuse.

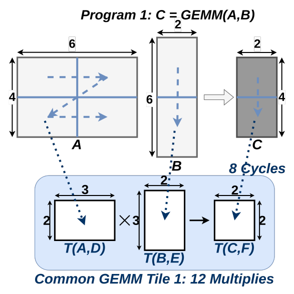
4. *General Helper Units*

In addition to the modules discussed above, ForgeBench contains a comprehensive collection of general-purpose modules that can be shared across various ML applications. These include load and store modules for data transfers between BRAM and DRAM, a range of activation function modules, a matrix addition module for residual connections, and element-wise matrix multiplication (Hadamard product) modules for gating networks. Other commonly used modules, such as dropout, max pooling, and average pooling, are also included. Together, these helper modules support building flexible end-to-end ML accelerator designs, enabling the representation of a plethora of models and evaluation of modular architecture-oriented HLS designs.

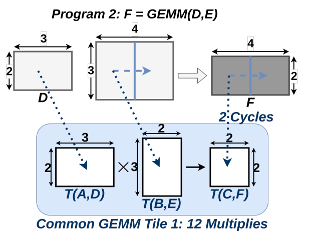
C. *Ready-to-Use Benchmark Suites*

We create two HLS benchmark suites with ForgeBench, demonstrating its convenience and effectiveness at generating relevant ML accelerator test cases, highlighted in Fig. 1 c).

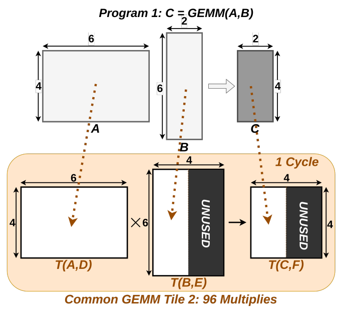
ML Testcases – The first is a large-scale general HLS benchmark suite, with over 6000 designs, described in section III-A. This suite is created by tuning the configuration parameters of base ML designs. Python scripts were used to explore ∼2000 unique configurations for each



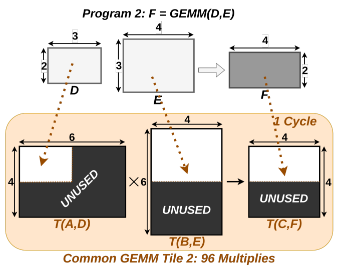
(a) Minimum Tiling GEMM(A,B)



(b) Minimum Tiling GEMM(D,E)



(c) Maximum Tiling GEMM(A,B)



(d) Maximum Tiling GEMM(D,E)

Fig. 2: Resource/Latency trade-off with common GEMM tiles.

test suite and generate the JSON config files for ForgeBench, each taking 1 PhD student around 1 hour to write. The JSON files and the generated HLS designs are publicly available. The scripts are also open-source and serve as a baseline for users to write and generate test cases specific to their tools.

Modular HLS Testcases – The second benchmark suite focuses on modularization and hardware reuse in ML HLS designs, described in detail in section III-B. We use ForgeBench first to generate 2-3 input designs per test case. The test cases are chosen with significant reuse opportunities across the input designs. We then provide the ideal modularized implementation, determined manually. These are added to the benchmark suite and serve as a reference output for future modular HLS tools.

When deciding on the shared computational modules for an architecture, there is a tradeoff between the resource usage of the common modules and the execution time of input designs. We highlight this tradeoff with two simple matrix multiplications as an example in Fig. 2. The depicted operations GEMM$(A,B) \rightarrow C$ and GEMM$(D,E) \rightarrow F$ have dimensions (4,6,2) and (2,3,4) respectively. Both operations can be computed using a tiled GEMM unit by carefully mapping the matrices to the hardware. The first option is the *minimum* tiling, highlighted in Fig. 2a and 2b, where the common tile is the greatest common denominator (GCD) of each dimension. This guarantees that both computations can be completed by tiling the input matrices to match the GCD dimensions and computing over multiple iterations. The GEMM operations in Fig. 2a and 2b take 8 and 2 iterations of the tiled computation, respectively. Alternatively, the *maximum* tiling can be used, where the common tile is determined with the highest common multiple (HCM) of each dimension, pictured in Fig. 2c and 2d. This tile size guarantees the matrices fit into common tiles, with every program computing within 1 iteration, at the expense of under-utilized hardware. This design choice is prominent in all function/tiling reuse test cases, emphasizing the necessity of a rigorous benchmark suite to evaluate future modular HLS tools and how to handle these decisions. Within the provided test suite, test cases - Tiled GEMM-Min/Max - and Vec/Mtx Mult - Dot/MMV/GEMM explicitly represent the choices of common module extraction.

D. Extensibility

Although ForgeBench currently spans three ML domains, it is not limited to these, as it is highly generalizable and extensible to emerging applications. First, ForgeBench incorporates numerous basic and widely used

Class	Model	Key Operations
DNNs	ResNet-N∗ [16]	Convolution, MaxPool, Matrix Add, ReLU, SoftMax, BatchNorm
	VGG∗ [17]	Convolution, MaxPool, ReLU, SoftMax, BatchNorm
	MobileNet-1/2/3 [18, 20, 21]	Grouped Convolution, ReLU, ReLU6, BatchNorm, HardSigmoid, HardSwish
	SqueezeNet [26]	Convolution, MaxPool, Matrix Add, ReLU, SoftMax, BatchNorm
	EfficientNet [19]	Grouped Convolution, AvgPool, Matrix Add, SiLU, Sigmoid, BatchNorm
LLMs	GPT-1/2∗ [11]	MultiHead Attention, GEMM, LayerNorm, Dropout
	LLaMA∗ [10]	Grouped MultiHead Attention, GEMM, RMS-Norm, SwiGLU, RoPE
	Mistral [22]	Sliding Window Attention, GEMM, RMS-Norm, SwiGLU, RoPE
	Gemma-1/2 [23, 27]	Grouped MultiHead Attention, GEMM, RMS-Norm, GeGLU, RoPE
	Griffin [28]	GEMM, RMS-Norm, GeLU, Element-wise Matrix Multiplication, Convolution

TABLE I: ML models supported by our framework. We provide JSON file design examples for ML models annotated with ∗.

Class	Base Design	Configurations	# Testcases
GEMM	$x \cdot A \cdot B \cdot y$	Vector/Matrix Dimensions, Loop Order, Unroll Factors, Computation Order	1920
DNN	Conv-BatchNorm-Activation	Feature Map & Kernel Dimensions, Grouped, Bias, Activation Function, Unroll Factors	2304
LLM	Attention-Dropout-Norm	Attention Dimensions, Heads, Grouped, With RoPE, With Dropout, Normalization Type, Dropout Probability	1944

TABLE II: Auto-generated ML benchmark suites that are ready to use.

operator modules for machine learning. Many complex operators can be decomposed into these fundamental components. For example, the SwiGLU operator used in LLaMA can be decomposed into the SiLU activation and element-wise multiplication, while the SiLU itself can be decomposed into the sigmoid function and element-wise multiplication. Second, even if a new operator cannot be decomposed into existing components, users can easily incorporate them into ForgeBench with minimal effort and generate synthesizable HLS designs with a JSON configuration file. Furthermore, test cases generated by ForgeBench can be seamlessly integrated with existing HLS tools such as AutoDSE [24] and HLSFactory [25]. These tools can take in our auto-generated benchmarks and increase the capacity and complexity of designs by including more complicated `HLS pragmas`, paving the way for more advanced HLS tool design in the future.

III. Results

A. Representing ML Models

Although different machine learning models can vary significantly in structure and data flows, most share common functional components. By defining the key operator modules in ForgeBench, described in section II-B, and specifying different sequences of module calls in the JSON file, we can generate hardware designs for various DNNs and LLMs. Table I summarizes the supported models and their key operators.

Furthermore, with ForgeBench, we generate an ML-focused test suite with 2304 DNN test cases, 1920 GEMM test cases, and 1944 LLM test cases. Each test case takes a baseline design and derives similar implementations with varying parameters, summarized in table II. The base DNN design is a convolution kernel, BatchNorm, and arbitrary activation function. The derivative test cases modify the convolution dimensions, parameters such as grouping or bias, unrolling the in/out channels, and the choice of activation function. The base GEMM design is a typical linear algebra operation, $x \cdot A \cdot B \cdot y$, where x,y are vectors and A,B are matrices. The test cases modify the dimensions, the unroll factor and order of the nested loops, and the order of operations. The LLM-focused testuite is the fundamental attention layer followed by an optional dropout layer and a normalization layer. We tune the input matrix and attention dimensions, the number of heads, the number of groups, the dropout probability, and the use of either LayerNorm or RMSnorm. While this test suite can directly be used to benchmark tools, it further highlights

Test Suite	Test Case	Resource Utilization (LUT%, DSP%) Before Modularization				Resource Utilization (LUT%, DSP%) After Modularization		Change (%, %)
		P1	P2	P3	Total	Shared	Total	
GEMM	Tiled GEMM - Min †	(12.07, 15.24)	(6.35, 5.08)	(18.18, 20.32)	(36.60, 40.64)	(4.03, 0.63)	(8.57, 0.63)	(-76.6, -98.45)
	Tiled GEMM - Max †					(42.95, 81.27)	(79.98, 81.27)	(118.5, 99.98)
	Vec/Mtx Mult - Dot † ∗	(45.34, 81.27)	(6.75, 10.16)	(1.39, 0.63)	(53.48, 92.06)	(0.06, 0.63)	(4.39, 1.27)	(-91.79, -98.62)
	Vec/Mtx Mult - MMV † ∗					(1.58, 5.08)	(11.9, 10.16)	(-77.75, -88.96)
	Vec/Mtx Mult - GEMM † ∗					(36.45, 81.27)	(58.03, 81.27)	(8.51, -11.72)
	i-j-k Orders ∗	(42.65, 81.27)	(44.35, 81.27)	(43.63, 81.27)	(130.6, 243.8)	(32.71, 81.27)	(61.42, 81.27)	(-52.98, -66.67)
	Vector Transpose ∗	(6.66, 10.16)	(4.92, 10.16)	–	(24.28, 20.32)	(1.99, 10.16)	(17.62, 10.16)	(-27.42, -50)
DNN	Activation Functions ‡ ∗	(2.36, 0.12)	(2.51, 0.12)	(2.07, 0.16)	(6.94, 0.4)	(0.24, 0.12)	(3.01, 0.24)	(-56.63, -40)
	Tiled Convolution † ∗	(10.97, 20.32)	(18.40, 20.32)	(18.44, 20.32)	(47.81, 60.96)	(21.95, 20.44)	(26.93, 20.63)	(-43.67, -66.15)
	DNN Blocks ‡	(24.62, 20.36)	(23.23, 20.44)	(67.26, 61.07)	(115.1, 101.9)	(40.50, 41.32)	(91.06, 81.98)	(-20.89, -19.52)
LLM	Tiled Attention † ∗	(7.16, 1.31)	(7.14, 1.31)	–	(14.3, 1.31)	(5.88, 1.31)	(7.99, 1.31)	(-44.13, -50)
	Functional Attention ‡	(21.87, 3.25)	(23.49, 2.62)	–	(45.36, 5.87)	(17.77, 1.99)	(25.69, 3.89)	(-43.36, -33.73)
	LLaMA/GPT Transformers ‡	(33.63, 9.44)	(15.30, 7.94)	–	(48.93, 17.38)	(8.77, 6.23)	(36.72, 9.6)	(-24.95, -44.76)

TABLE III: Resource usage of hardware test cases, annotated according to the available reuse †: tiling reuse, ‡: functional reuse, ∗: arithmetic reuse.

the ease of generating ML-oriented HLS test cases using ForgeBench, enabling user to create benchmarks focused on their unique use cases.

B. Benchmark Suite for Modularized HLS

In this section, we leverage ForgeBench to produce a set of benchmarks specific to module reuse. The benchmarks contain GEMM, DNN, and LLM-focused test suites. Each suite contains test cases with up to 3 input programs with varying types of hardware reuse. We explore three reuse scopes across the ML domains:

- **Tiling**: Computationally intensive operations are computed using many iterations of smaller similar operations.
- **Functional**: Operations call the same functions in different order/amount of times.
- **Arithmetic**: Operations have different hardware implementations but are mathematically equivalent.

Table III presents each test case, the targeted reuse, and the computational (DSP, LUT) resource utilization targeting at ZCU102 FPGA using Vitis HLS 2024.1. We report the resource utilization of the provided modularized implementations to highlight the necessity of a set of modular HLS tools. We do not present the delay or power of each design, as without a sufficient hardware architecture and control logic to schedule and execute the input designs, they have little meaning.

Details of each test case across the suites are described:

GEMM Testsuite – The first test case in this suite involves learning a standard GEMM tile across three programs each, computing a GEMM with bias of different dimensions, which can be identified. The programs P1–3 have dimensions (96,512,128), (128,256,64), (256,128,192) respectively. Correspondingly, the minimum and maximum tiles are (32,128,64) and (256,512,192), respectively. Each of these tilings is considered a separate test case. The second test case includes programs computing a GEMM, a vector-matrix product, and a dot product. Either of the three products can be used to compute the others, leading to a range of resource utilization, with a common dot product using the least, summarized in the table in the three Vec/Mtx Mult Testcases. Identifying this sharing requires both tiling the loops and some arithmetic equivalences. The 'i-j-k' Orders test case requires recognizing the GEMM equivalence over three permutations of 'i-j-k'. The common GEMM block may implement any arbitrary ordering of the three loops; the provided modularized code uses 'i-k-j'. The final case tests for reuse across a column/row vector-matrix product, which is equivalent under a transpose of the matrix.

DNN Testsuite – The first test case in the DNN testuite involves learning the shared arithmetic operators from different activation functions. The programs P1–3 implement activation functions: Sigmoid, Tanh, and Exponential Linear Unit (ELU), all sharing the exponent operator. In the second test case, P1–P3 correspond to three different 3×3 convolutions, each characterized by the configuration [input channels, output channels, height, width]. Specifically, P1 = [64, 64, 14, 14], P2 = [128, 128, 7, 7], and P3 = [128, 128, 14, 14]. The minimum tiling within these configurations is a [64, 64, 7, 7] convolution kernel. As a result, P1 and P2 can be computed in four iterations each, and P3 in sixteen iterations. In the third test case, we represent convolution blocks in different DNNs using [kernel size, channels]. P1 is a convolution block in VGG-19 [17], comprising four repeated [3×3, 256] convolutions with ReLU activation, followed by max pooling. P2 is from ResNet-18 [16] and includes two repeated [3×3, 256] convolutions with BatchNorm and ReLU. P3 is from ResNet-50 [16] and consists of [1×1, 64], [3×3, 64], and [1×1, 256] convolutions, along with BatchNorm and ReLU. Both P2 and P3 also include residual connections. The modular implementation shares the [3×3, 64] convolution, BatchNorm and ReLU operations across these three programs with correct tiling.

LLM Testsuite – In this test suite, the first case tests tiling two multi-head attention configurations: P1 with 16 attention heads and P2 with 4 attention heads. The modularized implementation uses the 4-head configuration as a shared computational module. Consequently, P1 requires four iterations to complete its computation. The second test case focuses on the functional decomposition of attention. P1 is the multi-head attention module from GPT, whereas P2 is the grouped multi-head attention module from LLaMA with RoPE encoding. These attentions can be decomposed into smaller computational components, revealing that both incorporate GEMM and SoftMax submodules, which can be shared and reused. In the third test case, P1 and P2 denote the GPT and LLaMA transformer blocks, respectively. Despite GPT using LayerNorm and GeLU as its activation function and LLaMA employing RMSNorm and SwiGLU, the multi-head attention and feed-forward network components can be shared across both architectures.

IV. Conclusion & Future Work

In this work, we introduce *ForgeBench*, a benchmark suite tailored explicitly for next-generation HLS tools focused on machine learning architecture design. ForgeBench is accompanied by a user-friendly and highly extensible framework for automated HLS design generation and evaluation. We also provide two ready-to-use benchmarks; one includes over 6,000 representative ML test cases, and the other includes modular HLS test cases with possible resource sharing manually implemented to highlight the necessity of designing architecture-oriented HLS tools.

Existing research and recent advancements strongly support our vision for future architecture-oriented HLS tool design. Notably, equivalence-graphs (e-graphs) have shown promise for extracting shared structures across programs [29], and have already proved effective at representing and optimizing hardware designs [30, 31]. Consequently, we envision future HLS tools leveraging e-graphs to automatically identify common functional modules across HLS designs. After extracting common modules, the tools will generate a flexible finite-state machine and embed scheduling information into instructions that govern state transitions, enabling flexible dataflows to reuse functional modules. Furthermore, we foresee that HLS tools will perform design space exploration during architecture generation to determine the optimal module reuse choice, balancing area, and latency in a holistic and automated manner. Given these developments, and considering that ForgeBench is fully open-sourced and easily extensible, we believe it will substantially benefit ongoing and future HLS tools design and implementation.

References

[1] B. Reagen *et al.*, "Machsuite: Benchmarks for accelerator design and customized architectures," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, 2014, pp. 110–119.

[2] Y. Zhou *et al.*, "Rosetta: A realistic high-level synthesis benchmark suite for software programmable fpgas," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 269–278. [Online]. Available: https://doi.org/10.1145/3174243.3174255

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[4] L.-N. Pouchet and U. Bondugula, "PolyBench." [Online]. Available: https://www.cs.colostate.edu/~pouchet/software/polybench/

[5] S. Liu *et al.*, "Overgen: Improving fpga usability through domain-specific overlay generation," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 35–56.

[6] L. Guo *et al.*, "Tapa: A scalable task-parallel dataflow programming framework for modern fpgas with co-optimization of hls and physical design," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 16, no. 4, Dec. 2023. [Online]. Available: https://doi.org/10.1145/3609335

[7] Y.-H. Lai *et al.*, "Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 242–251. [Online]. Available: https://doi.org/10.1145/3289602.3293910

[8] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "Dsagen: Synthesizing programmable spatial accelerators," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 268–281.

[9] Xilinx, "Vitis hls," https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html.

[10] H. Touvron *et al.*, "Llama: Open and efficient foundation language models," 2023. [Online]. Available: https://arxiv.org/abs/2302.13971

[11] A. Radford *et al.*, "Improving language understanding by generative pre-training," *OpenAI Blog*, 2018. [Online]. Available: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

[12] C. Zhang *et al.*, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 161–170. [Online]. Available: https://doi.org/10.1145/2684746.2689060

[13] T. Aarrestad *et al.*, "Fast convolutional neural networks on fpgas with hls4ml," *Machine Learning: Science and Technology*, vol. 2, no. 4, p. 045015, jul 2021. [Online]. Available: https://dx.doi.org/10.1088/2632-2153/ac0ea1

[14] C. Hao *et al.*, "Fpga/dnn co-design: An efficient design methodology for iot intelligence on the edge," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3316781.3317829

[15] H. Ye *et al.*, "Scalehls: a scalable high-level synthesis framework with multi-level transformations and optimizations: invited," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1355–1358. [Online]. Available: https://doi.org/10.1145/3489517.3530631

[16] K. He *et al.*, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[17] K. Simonyan *et al.*, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[18] A. G. Howard *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017. [Online]. Available: https://arxiv.org/abs/1704.04861

[19] M. Tan *et al.*, "Efficientnet: Rethinking model scaling for convolutional neural networks," *CoRR*, vol. abs/1905.11946, 2019. [Online]. Available: http://arxiv.org/abs/1905.11946

[20] M. Sandler *et al.*, "Mobilenetv2: Inverted residuals and linear bottlenecks," 2019. [Online]. Available: https://arxiv.org/abs/1801.04381

[21] A. Howard *et al.*, "Searching for mobilenetv3," 2019. [Online]. Available: https://arxiv.org/abs/1905.02244

[22] A. Q. Jiang *et al.*, "Mistral 7b," 2023. [Online]. Available: https://arxiv.org/abs/2310.06825

[23] G. Team *et al.*, "Gemma: Open models based on gemini research and technology," 2024. [Online]. Available: https://arxiv.org/abs/2403.08295

[24] A. Sohrabizadeh *et al.*, "Autodse: Enabling software programmers to design efficient fpga accelerators," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 27, no. 4, Feb. 2022. [Online]. Available: https://doi.org/10.1145/3494534

[25] S. Abi-Karam *et al.*, "Hlsfactory: A framework empowering high-level synthesis datasets for machine learning and beyond," in *2024 ACM/IEEE 6th Symposium on Machine Learning for CAD (MLCAD)*, 2024, pp. 1–9.

[26] F. N. Iandola *et al.*, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡0.5mb model size," 2016. [Online]. Available: https://arxiv.org/abs/1602.07360

[27] G. Team *et al.*, "Gemma 2: Improving open language models at a practical size," 2024. [Online]. Available: https://arxiv.org/abs/2408.00118

[28] S. De *et al.*, "Griffin: Mixing gated linear recurrences with local attention for efficient language models," 2024. [Online]. Available: https://arxiv.org/abs/2402.19427

[29] D. Cao *et al.*, "babble: Learning better abstractions with e-graphs and anti-unification," *Proc. ACM Program. Lang.*, vol. 7, no. POPL, Jan. 2023. [Online]. Available: https://doi.org/10.1145/3571207

[30] S. Coward *et al.*, "Rover: Rtl optimization via verified e-graph rewriting," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 43, no. 12, p. 4687–4700, Dec. 2024. [Online]. Available: https://doi.org/10.1109/TCAD.2024.3410154

[31] J. Cheng *et al.*, "Seer: Super-optimization explorer for high-level synthesis using e-graph rewriting," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1029–1044. [Online]. Available: https://doi.org/10.1145/3620665.3640392